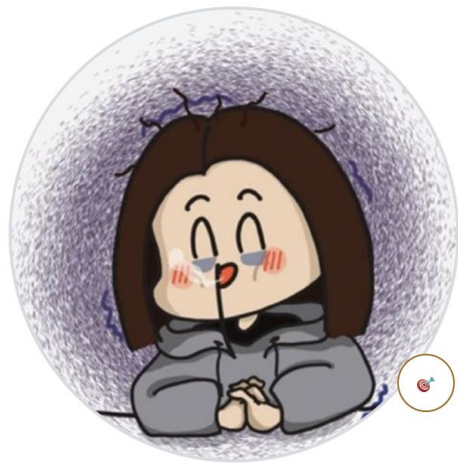
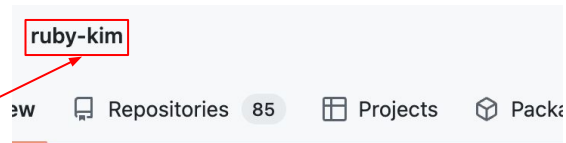


실습 파일 & 교안 받아주세요!!

- <https://github.com/mentorships/Invite-guide> 접속
- 가이드에 따라 repo 초대 API 실행하기
- Response가 200이 나왔을 시, 이메일을 확인 후 Accept 클릭하기
- 초대 완료!

항목	설명
GitHub Username	본인의 GitHub 프로필에 있는 사용자명 (오른쪽 이미지 참고)
Repository 이름	OZ-Coding-BE16-Flask
X-API-KEY	oz-be16-251219



Ruby Kim

ruby-kim

자기소개

- 이름: 김루비
- 회사: Sibel Health
 - 외국계 스타트업 재직 중: 2021.09 ~ 현재
 - 클라우드 엔지니어
- 기타
 - KDT 구름, 엘리스, 오즈코딩스쿨 등에서 강연 및 멘토링 진행: 2022.01 ~ 현재
 - K-SW Purdue Program 수료: 정부 4개월 미국 생활/프로젝트비 전액 지원
 - 창업 프로그램(예비창업패키지, 에코스타트업, 학생 창업유망팀 300 등) 최종선정
 - SW책 출간 및 해커톤 수상(금상, 은상) 경험

커리큘럼

- 1일차: Flask 기초 & 웹 기본 이해
- 2일차: Flask REST API & 데이터 다루기
- 3일차: Flask DB 연동 & 데이터 저장
- 4일차: Flask 웹소켓 & 실시간 기능
- 5일차: Flask 미니 프로젝트



1일차: Flask 기초 & 웹 기본 이해

목 차

- Flask 소개 및 환경 세팅
- 웹 기본 원리 & Flask 라우팅
- 템플릿 엔진 Jinja2
- 간단한 입력-출력 웹앱 완성하기
- Static 파일 & 모듈화

Flask 소개 및 환경 세팅

Flask란?

- 파이썬으로 만든 마이크로 웹 프레임워크
 - 라우팅, HTTP 요청/응답 처리, 템플릿 렌더링 등 기본 웹 서버 기능 제공
 - (참고) 마이크로: 핵심 기능은 작고 단순 & 필요 기능은 플러그인을 통해 확장 가능
- 2010년 4월 1일 만우절 장난으로 인해 탄생했으며, 전 세계적으로 많이 사용되는 중
- Netflix, Reddit, Lyft, LinkedIn 등 다양한 기업에서 활용 중

Flask 핵심 철학

- Simplicity (단순성): 설치 후 몇 줄 만으로 웹 서버 실행 가능
- Flexibility (유연성): 기본은 가볍게, 필요한 기능만 확장 라이브러리로 선택적 추가
- Explicitness (명시성): 코드가 명확하고 파이썬스러움 -> 어떻게 동작하는지 직관적 확인 가능
- Developer Experience (DX): 잘 정돈된 개발문서, 버전 간 호환성 👍

Flask 특징

- URL 라우팅: 주소(URL)를 함수와 쉽게 연결
- 템플릿 엔진 (Jinja2): HTML에 데이터 삽입/제어문 활용 가능
- HTTP 요청 파싱: GET, POST, PUT, DELETE 등 자유롭게 처리
- 세션 & 쿠키 관리: 로그인 상태 유지 같은 기능 지원
- 대화형 디버거: 에러 발생 시 브라우저에서 확인 가능
- 확장성 좋은 생태계: DB, 인증, 보안 등 수백 가지 확장 라이브러리 존재
- 작고 가볍지만 실무에서도 충분히 활용 가능

Flask와 경쟁 프레임워크 비교

프레임워크	특징
Flask	<ul style="list-style-type: none">• 마이크로 프레임워크: 기본은 심플, 확장으로 원하는 기능 추가
Bottle	<ul style="list-style-type: none">• Flask와 비슷한 극소형 프레임워크, 모든 기능이 단일 .py에 있음• 확장에 있어서 불편함
FastAPI	<ul style="list-style-type: none">• 현대적인 프레임워크: 자동 문서화, 타입 힌트 활용, ASGI 기반
Django	<ul style="list-style-type: none">• 풀스택 프레임워크: ORM, Admin, 인증 전부 내장됨. 프로젝트 구조가 강제됨

Flask 설치 및 환경 구성

- Flask 설치

가상환경 만들기 (윈도우) : `python -m venv venv`

`venv\Scripts\activate.bat`

가상환경 만들기 (맥/리눅스): `python3 -m venv venv`

`source venv/bin/activate`

`pip3 install flask`

- 개발 환경 설정

- IDE: VSCode 또는 PyCharm 추천

- (API 테스트 도구: Postman, Swagger UI 또는 cURL)

Flask 기본 코드 구조

[실습 1] Hello, Flask! API 만들기

- 실행: python main.py

```
from flask import Flask

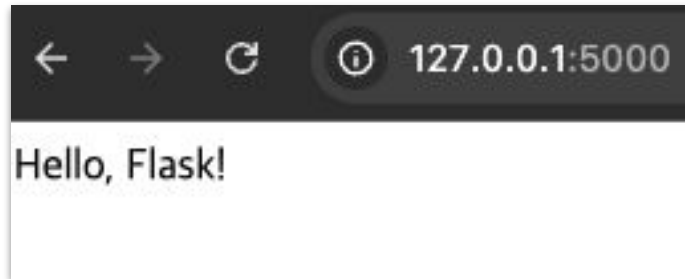
app = Flask(__name__)

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

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

Flask 기본 코드 구조

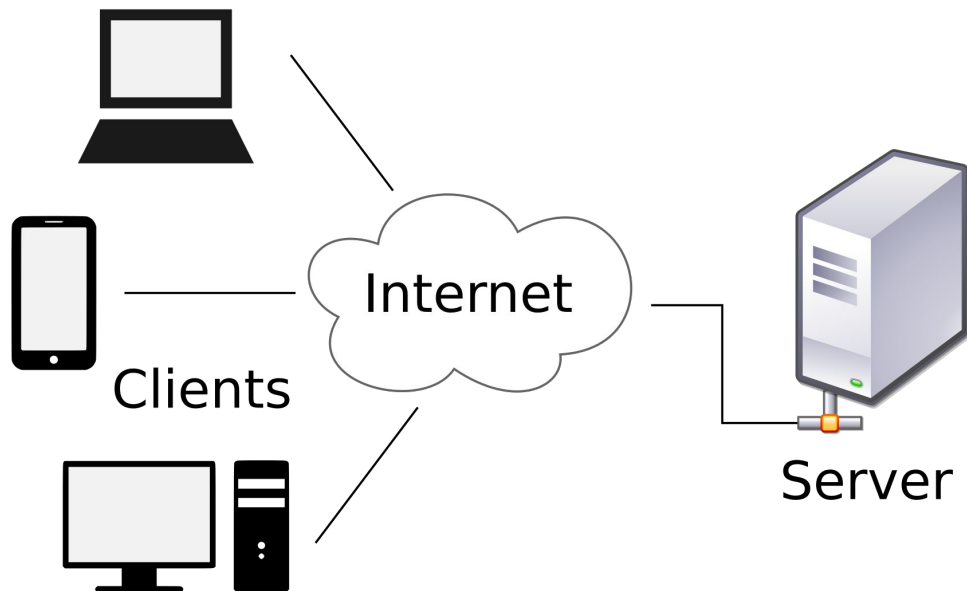
[실습 1] **Hello, Flask!** API 만들기



```
* Serving Flask app 'main'
* Debug mode: on
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
* Running on http://127.0.0.1:5000
Press CTRL+C to quit
* Restarting with stat
* Debugger is active!
* Debugger PIN: 103-214-391
127.0.0.1 - - [15/Sep/2025 01:11:04] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [15/Sep/2025 01:11:04] "GET /favicon.ico HTTP/1.1" 404 -
```

웹 기본 원리

클라이언트 - 서버 구조



HTTP 요청 / 응답 & 상태코드

- 요청 구조

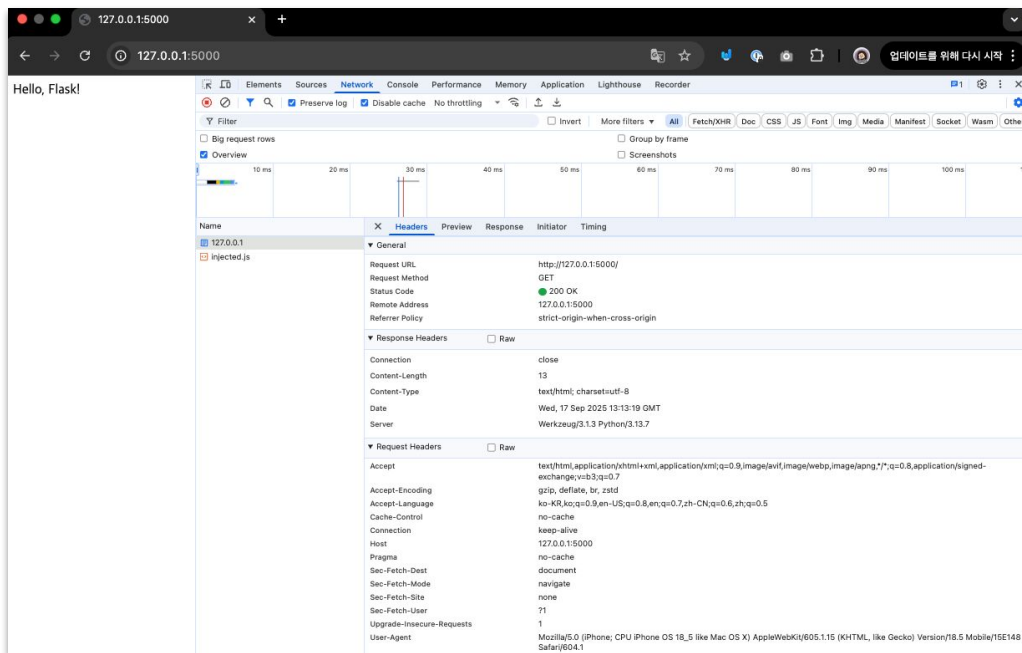
- URL, 메서드 (GET, POST, PUT, DELETE)
- 헤더(Header), 바디(Body)

- 응답 구조

- 상태코드: 200 OK, 404 Not Found, 500 Internal Server Error 등
- 개발자도구 -> Network 탭 -> 요청/응답 확인
 - 맥북: option + command + I
 - 윈도우: F12

HTTP 요청 / 응답 & 상태코드

- [실습 2] HTTP 요청 직접 보내보기: 브라우저에 직접 URL 입력



HTTP 요청 / 응답 & 상태코드

- [실습 2] HTTP 요청 직접 보내보기: 터미널에 요청하기 (client URL)

```
└─$ curl http://127.0.0.1:5000  
Hello, Flask!%
```

라우팅 / 라우터란?



Client

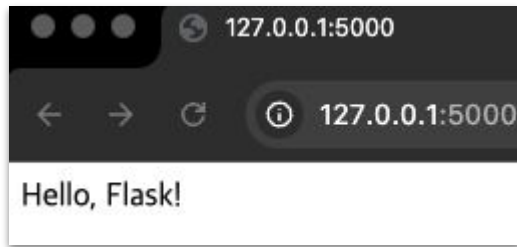
주소창에

<http://127.0.0.1:5000> 입력



Flask 서버

```
@app.route("/")
def hello_world():
    return "Hello, Flask!"
```



응답

- 라우팅(routing): URL 요청과 함수를 연결하는 매니저/규칙
- 라우터(router): 이 규칙을 실제로 관리하고 실행하는 친구 (Flask 내장)

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!"
```

```
?main ~/Desktop/personal/OZ-Coding-BE16-Flask/251219 (Day 1)/0\M-^K00\M-\n03> curl http://127.0.0.1:5000/user/oz-be16
oz-be16님 만나서 반갑습니다%
?main ~/Desktop/personal/OZ-Coding-BE16-Flask/251219 (Day 1)/0\M-^K00\M-\n03> curl http://127.0.0.1:5000/user/flask
flask님 만나서 반갑습니다%
```

템플릿 엔진 (Jinja2)

템플릿 엔진 Jinja2

- 템플릿 엔진: Python에서 만든 데이터를 HTML에 끼워 넣어주는 도구
- Flask는 기본적으로 Jinja2를 사용
 - {{변수명}} <- 변수 바인딩
 - {% ... %} <- 제어문
- Flask에서 **render_template()**을 통해 .html파일과 데이터를 합쳐 최종 웹페이지를 만들 수 있음

Flask에서의 폴더 구조

- 무조건 templates 폴더 안에 html 템플릿을 넣을 것!!

```
project/  
| app.py  
└ templates/  
    └ hello.html
```

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

변수 바인딩 예시

[실습 4] 조건문 바인딩 해보기

```
@app.route("/user/<username>")
def user(username):
    return render_template("user.html", username=username)
```

main.py

```
<html>
<head>
  <title>User</title>
</head>
<body>
  {% if username == "admin" %}
    <h1>관리자님 환영합니다!</h1>
  {% else %}
    <h1>안녕하세요, {{ username }} 님!</h1>
  {% endif %}
</body>
```

templates/user.html

변수 바인딩 예시

[실습 4] 반복문 바인딩 해보기

```
@app.route("/fruits")
def fruits():
    fruits = ["사과", "바나나", "딸기", "포도"]
    return render_template("fruits.html", fruits=fruits)
```

main.py

```
<!DOCTYPE html>
<html>
<head>
    <title>Fruits</title>
</head>
<body>
    <h1>과일 목록</h1>
    <ul>
        {% for fruit in fruits %}
            <li>{{ fruit }}</li>
        {% endfor %}
    </ul>
</body>
</html>
```

templates/fruits.html

변수 바인딩 예시

[실습 4] 반복문 + 조건문 바인딩 해보기

```
<!DOCTYPE html>
<html>
<head>
  <title>Fruits</title>
</head>
<body>
  <h1>과일 목록</h1>
  <ul>
    {% for fruit in fruits %}
      {% if fruit == "딸기" %}
        <li>{{ fruit }} 🍓 (인기!)</li>
      {% else %}
        <li>{{ fruit }}</li>
      {% endif %}
    {% endfor %}
  </ul>
</body>
</html>
```

templates/fruits.html

Flask 변수 바인딩 실행 흐름



Client

브라우저가 /hello 요청



Flask 서버

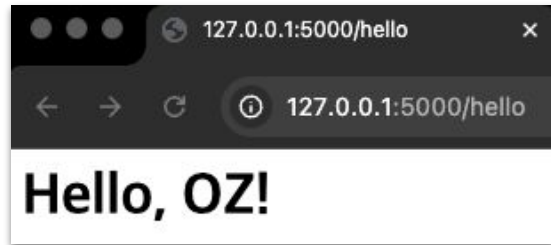
Flask가 hello() 함수 실행

-> Flask가 hello.html을 templates

폴더에서 찾음

-> {{name}}에 “OZ” 삽입

최종 HTML 생성



응답

간단한 입력-출력 웹앱 만들기

[실습 5] 간단한 입력-출력 웹앱 만들기

목표

- 이름 입력 -> 화면에 출력되는 웹앱 만들기
- HTML 입력 폼(form) + Flask 라우팅 + 변수 바인딩
- 정적 파일 (static) 활용해서 CSS 살짝 맛보기

[실습 5] 간단한 입력-출력 웹앱 만들기

- request.args 소개
 - 브라우저에서 GET 요청 시 URL 뒤에 붙는 ?키=값을 가져오는 방법
 - Flask에서는 request.args.get(“키”) 사용

[실습 5] 간단한 입력-출력 웹앱 만들기

- request.args 소개

- 예시

- 브라우저 요청: </greet?name=OZ>

- Flask: request.args.get("name")

- 결과: "OZ"

- 참고 문서:

- <https://flask.palletsprojects.com/en/stable/quickstart/>

[실습 5] 간단한 입력-출력 웹앱 만들기

- Skeleton 코드의 주석을 참고하여 완성해보세요.
 - app.py
 - templates/greet.html



Static 파일 & 모듈화

정적 (Static) 파일

- Flask에서 서버 코드와 상관없이 그대로 전달되는 파일
 - CSS: 스타일 정의
 - JS: 브라우저 동작 보조
 - 이미지, 폰트, 아이콘 등 리소스
- Flask 규칙: 정적 파일은 **static/** 폴더에 넣어야 함
 - URL로 접근할 때는 /static/... 사용

정적 (Static) 파일

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

정적 (Static) 파일

```
body {  
    font-family: Arial, sans-serif;  
    background-color: #f8f9fa;  
    text-align: center;  
    margin-top: 50px;  
}  
  
h1 {  
    color: #2c3e50;  
}  
  
input[type="text"], button {  
    padding: 8px 12px;  
    font-size: 14px;  
    margin-top: 10px;  
}
```

static/style.css

정적 (Static) 파일

```
<!DOCTYPE html>
<html>
<head>
  <title>OZ 인사하기</title>
  <!-- static 경로는 이렇게 불러옵니다 -->
  <link rel="stylesheet" href="/static/style.css">
</head>
<body>
  
  <h1>이름을 입력하세요</h1>
  <form action="/greet" method="get">
    <input type="text" name="name" placeholder="이름 입력">
    <button type="submit">인사하기</button>
  </form>
</body>
</html>
```

templates/index.html

모듈화 (프로젝트 구조화)

- 초반에는 하나의 python 파일에 모든 코드를 넣을 수 있음
- 하지만 기능이 많아진다면? 유지보수가 어려워짐 -> **역할 분리 필요**

모듈화 (프로젝트 구조화)

- 기본 분리 원칙
 - 코드 (app.py)
 - 서버 로직, 라우팅
 - 화면 구조 (templates/)
 - HTML 뼈대
 - 디자인 (static/)
 - CSS, JS, 이미지

발전된 프로젝트 구조

```
project/
| app.py
├─ app/
|   ├── __init__.py
|   ├── routes.py      ← 라우팅만 담당
|   ├── models.py      ← 데이터베이스 모델
|   └─ templates/...
└─ static/...
```

숙제: Postman 설치

QnA