웹프로그래밍의 기초

Week10

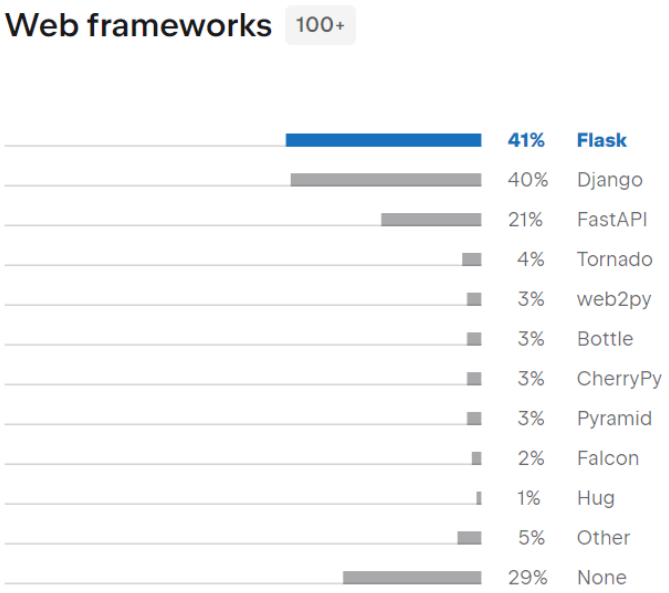
플라스크 설치, 설정, 실행; 템플릿 상속

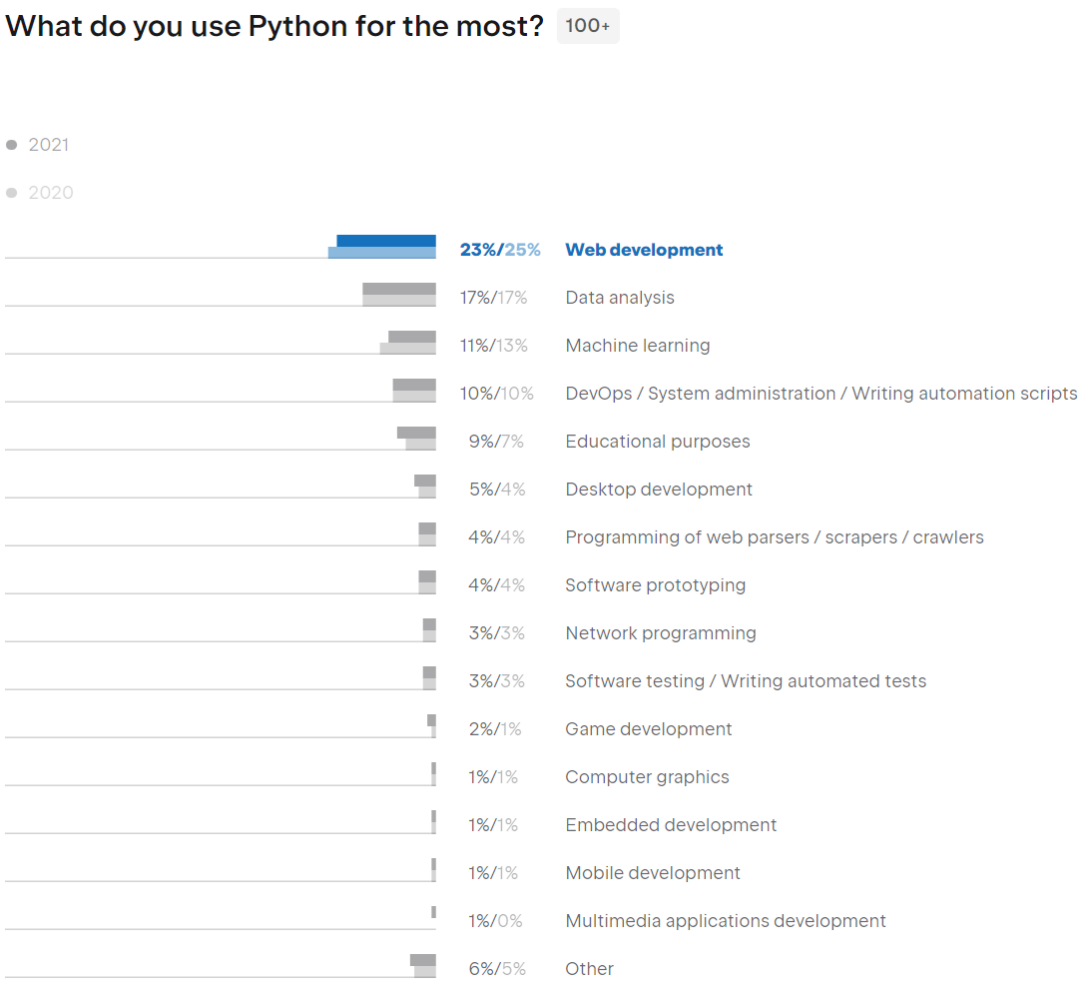
Introduction to Flask

Python for web development

Django	Flask
Django is probably best known for its *"batteries included"* approach, meaning that it comes prepackaged with a large number of useful tools (imagine a swiss army knife but its individual parts actually do the job quite well) that the developer can choose from. These include components like **web server, template system, caching framework,** and many others. https://www.djangoproject.com/	**Flask**, on the other hand, chooses a different approach: it's designed to be as light-weight as possible, only offering basic functionality. Should the developer wish to extend the functionality of a Flask app, they'd have to rely on third-party tools and extensions (as we'll find out later, this approach can pose certain security issues). Due to its small size (and a small set of features when compared to Django), Flask is often referred to as a *"microframework"*, i.e. a framework which, as the name suggests, offers only rather limited functionality. https://flask.palletsprojects.com

From the recent Python Developer Survey





https://lp.jetbrains.com/python-developers-survey-2021/#FrameworksLibraries

Flask 설치, 설정

1. Preparation for venv

- Python and Python3 executables are different.
- With Python3 installed on Linux system by default, a couple of packages need to be installed for the next step.

```
sudo apt update
sudo apt install python3-pip
sudo apt install python3-venv
```

2. Create venv

- Then create your venv home at '$HOME/venvs/webp'.
- You can create your venv home wherever you may want to create on, but for this class it conveniently presumes that we all use '$HOME/venvs/webp'.
- In case you already have Conda to manage your own virtual environments, Conda is the better option for you.

```
## 가상환경 루트 디렉토리는 ~/venvs
## 가상환경 디렉토리는 ~/venvs/webp
cd
mkdir venvs
cd venvs
python3 -m venv webp
## source ~/venvs/webp/bin/activate를 치면 가상환경이 활성화됨.
source ~/venvs/webp/bin/activate
```

3. Install Flask related packages

- PIP is the package installer for Python. Install the required packages for this class as follows.
- Run the following commands under ‘webp’ virtual environment.

```
# the new standard of Python distribution and are intended to replace eggs.
pip install wheel

# The main package for Flask
pip install -U Flask

# Provoding support for writing external scripts in Flask.
pip install -U Flask-Script

# Simple integration of Flask and WTForms
pip install -U Flask-WTF

# extension that handles SQLAlchemy database migrations for Flask applications using Alembic
pip install -U Flask-Migrate
```

4. Create Flask project directory

```
## 본인의 flask 실습 HOME은 ~/projects/webp
## 앞으로 flask 실습 HOME 디렉토리에서 실습과 과제를 진행하면 됩니다.
## 아무 위치에서나 deactivate를 치면 가상환경이 비활성화됨
deactivate

## 프로젝트 루트 디렉토리를 만든 뒤, 해당 위치에서 가상환경을 실행함.
cd
mkdir projects
cd projects
source ~/venvs/webp/bin/activate

## 프로젝트 루트 디렉토리 안에서 실제 작업할 프로젝트 디렉토리를 만든 뒤, 해당 위치에서 가상환경을 실행함.
mkdir webp
cd webp
```

5. Install IDE you want to use

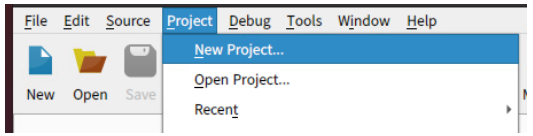
- If your option is Wing IDE, then follow the installation procedures that already introduced for you.

```
wget https://wingware.com/pub/wing-personal/8.3.3.0/wing-personal8_8.3.3-0_amd64.deb
```

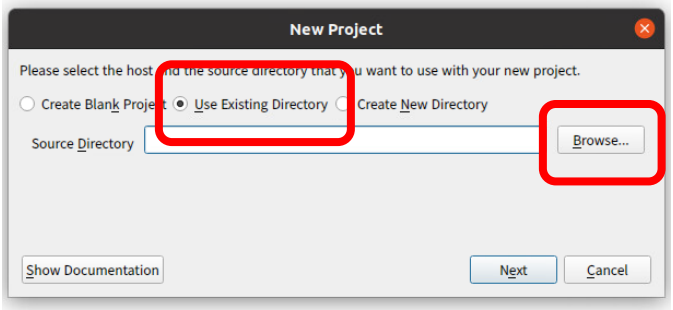
```
sudo apt install ./wing-personal8_8.3.3-0_amd64.deb
```

6. WING project setup for ENV

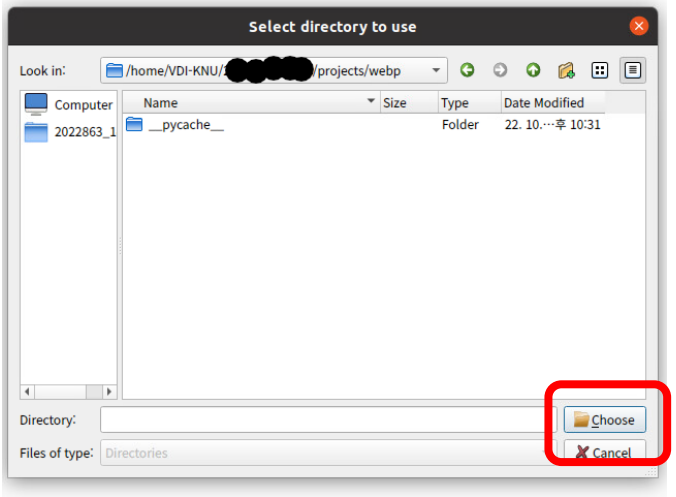
(1)



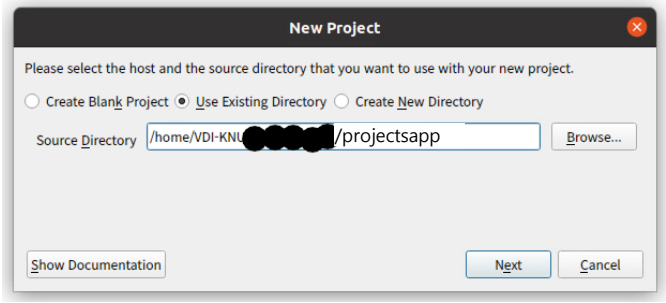
(2)



(3)

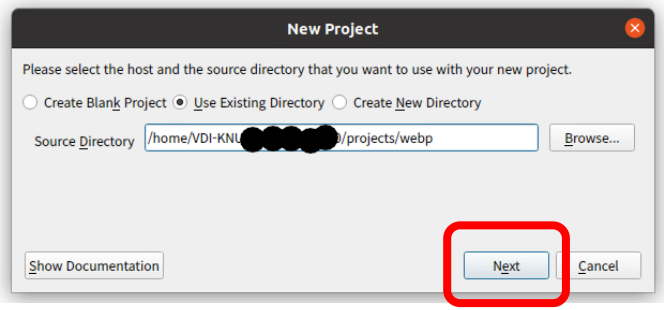


(4)

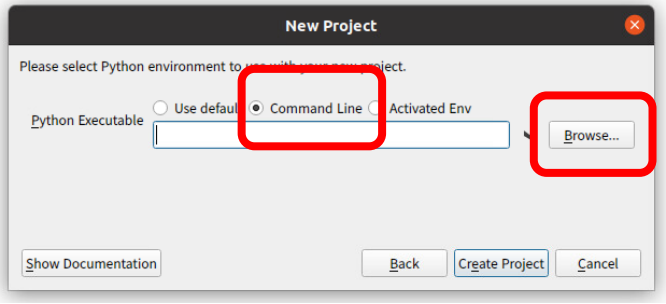


WING setup for ENV (Cont'd)

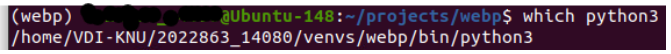
(4)



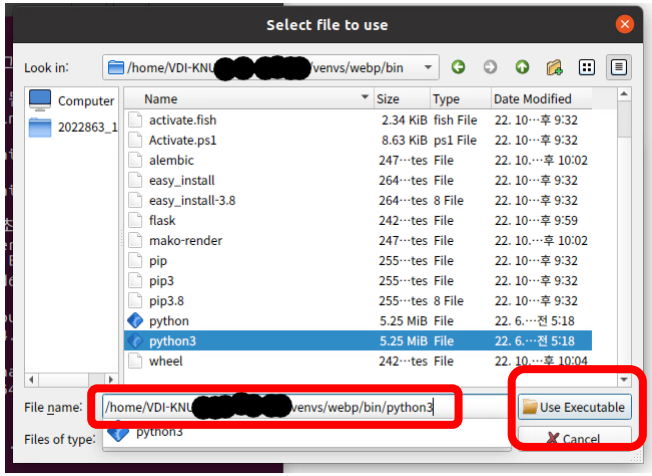
(5)



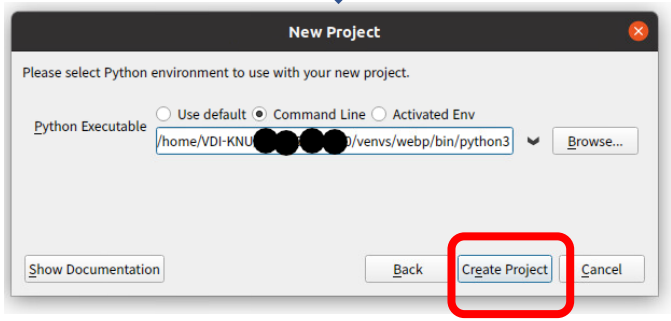
(6)



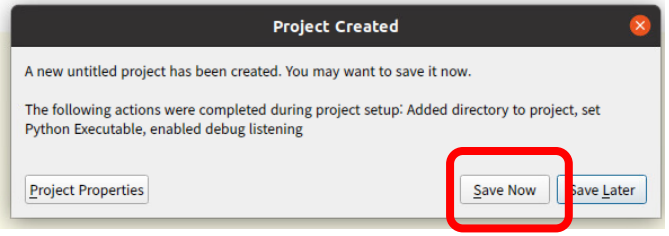
(7)



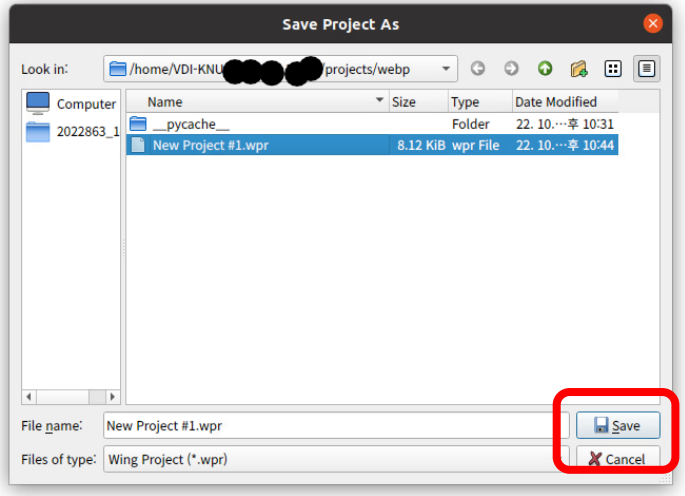
(8)



(9)



(10)



Flask 실행

Hello flask

- Make app.py file in the prescribed location.

```
~/projects/webp
 |
 └──── app.py
```

```
from flask import Flask

app = Flask(__name__)

@app.route('/')
def hello():
    return 'hello, flask'
```

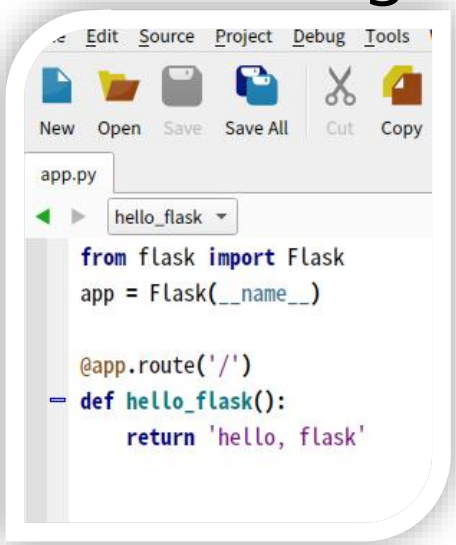
- Then, go to the directory where app.py resides then run flask.

```
cd ~/projects\webp
flask run
```

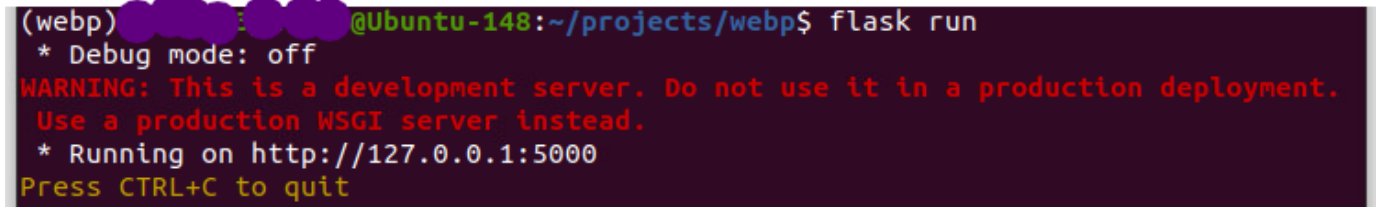
For application name, Flask looks into the environment variable of FLASK_APP. If it is null then Flask sees the application name as 'app', and finds 'app.py'

In real practice

- After saving app.py in your project folder

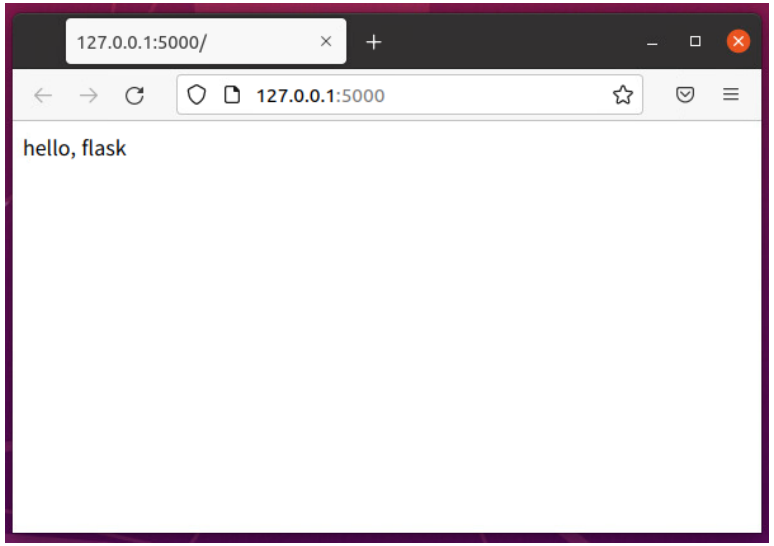


- Execute it with ‘flask run’ command to make it running on http:/127.0.0.1:5000 for default.



In real practice (Cont'd)

- Launch your web browser and type the URL into it, then you will see the test page you previously built.



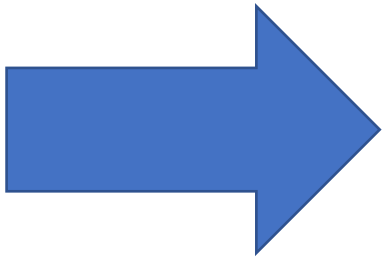
Flask application factories

Application Factories

- Well-structured web apps separate logic between files and modules, typically with separation of concerns in mind.
- This seems tricky with Flask at first glance because our app depends on an "app object" that we create via app = Flask(__name__)
- Separating logic between modules means we'd be importing this app object all over the place, eventually leading to problems like circular imports. The Flask Application Factory refers to a common "pattern" for solving this dilemma.

```
~/projects/webp
 |
 └──── app.py
```

```
from flask import Flask

app = Flask(__name__)

@app.route('/')
def hello():
    return 'hello, flask'
```

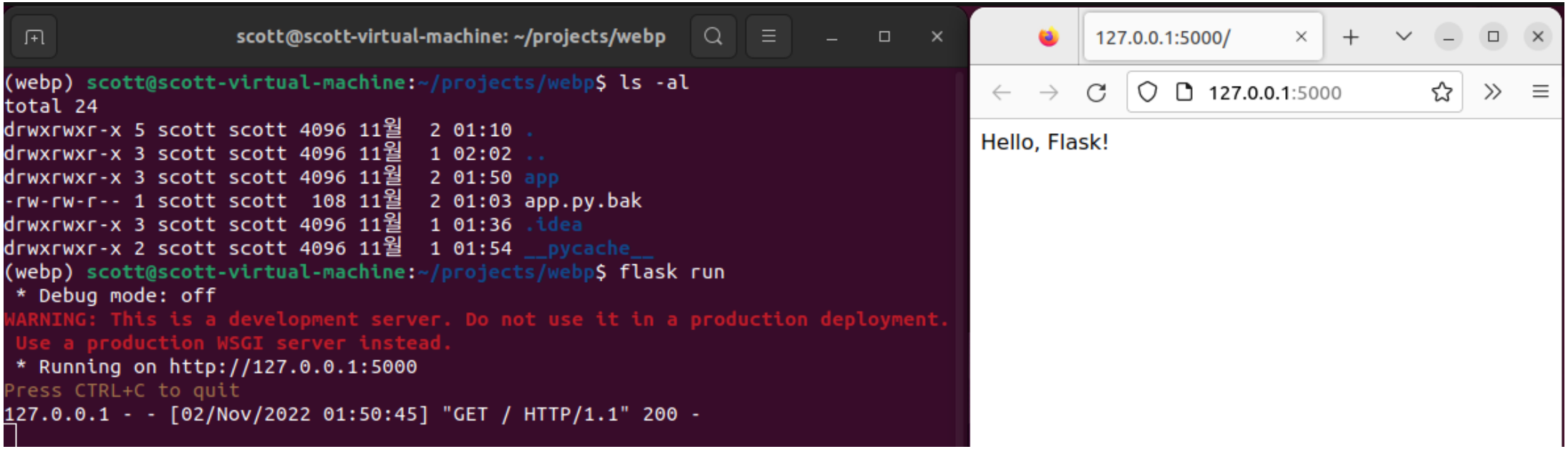


```
~/projects/webp/app
 |
 └──── __init__.py
```

```
from flask import Flask

def create_app():
    app = Flask(__name__)

    @app.route('/')
    def hello():
        return 'Hello, Flask!'

    return app
```

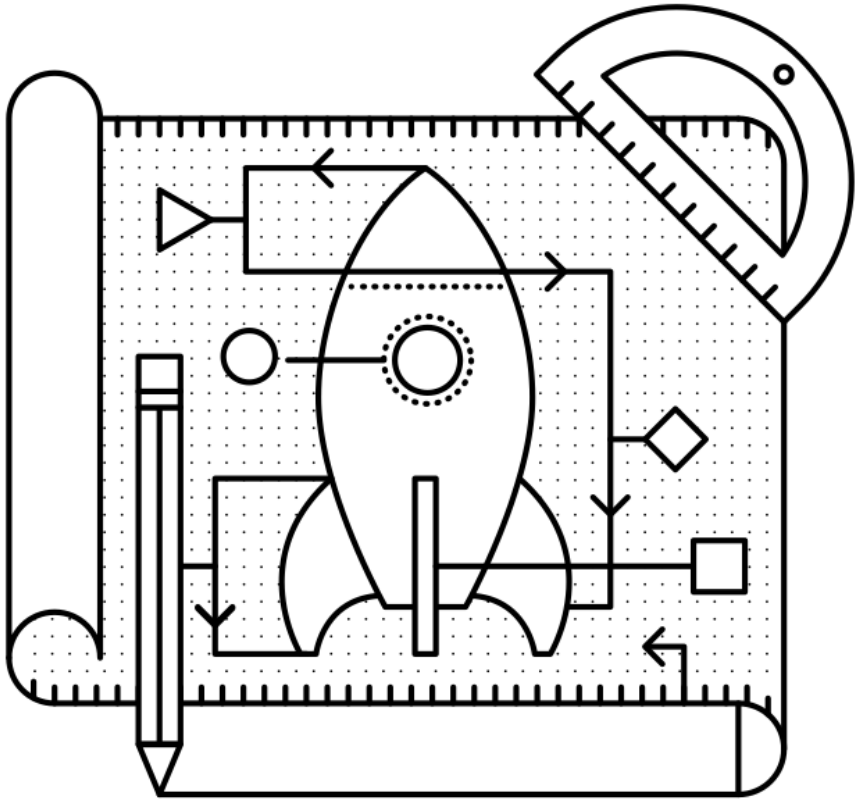
Application Factories (Cont'd)

- After doing backup your 'app.py' not to make available for flask anymore, execute 'flask run' command at the current locations.



Blueprint

- A blueprint defines a collection of views, templates, static files and other elements that can be applied to an application.
- For example, let’s imagine that we have a blueprint for an admin panel. This blueprint would define the views for routes like /admin/login and /admin/dashboard.
- It may also include the templates and static files that will be served on those routes



Blueprint (Cont'd)

- As your code grows, it can become harder for you to maintain everything in a single file. So, when your application grows in size or complexity, you may want to structure your code in a different way to keep it maintainable and clear to understand.
- Flask Blueprints encapsulate functionality, such as views, templates, and other resources.
- You can refactor the previous application by moving a view function into a Flask Blueprint. To do so, you have to create a Flask Blueprint that contains the view function and then use it in the application.

Flask project structure

- The killer use-case for blueprints is to organize our application into distinct components.
- Each distinct area of the site can be separated into distinct areas of the code as well.
- This lets us structure our app as several smaller “apps” that each do one thing.

```
~/projects₩webp
 |
 ├──── app/
 |       ├── __init__.py
 |       ├── models.py
 |       ├── forms.py
 |       ├── views/
 |       ├── static/
 |       └── templates/
 └──── config.py
```

- models.py file
  - defines models that require to mange database.
- forms.py file
  - defines Form classes from WTFoms library.
- **views directory**
  - **contains *.py files to show something to the user.**
- static directory
  - contains statics information for *.css, *.js, and image files such as JPG.
- template directory
  - contains HTML files to enable dynamic pages.
- config.py file
  - has project environment variables, database setups, and so on.

```
~/projects/webp/app/views
 |
 └──── main_views.py
```

1. Create a Blueprint object called testbp.
2. Add views to testbp using the route decorator.

```
from flask import Blueprint

testbp = Blueprint('testbp', __name__, url_prefix='/')

@testbp.route('/')
def hello():
    return 'Hello, Flask!'

@testbp.route('/steve')
def hello_steve():
    return 'Hello, Steve!'
```

```
~/projects/webp/app
 |
 └──── __init__.py
```

3. import the main_view.py containing Blueprint object
4. register testbp from main_view.py in the application using register_blueprint()

```
from flask import Flask

def create_app():
    app = Flask(__name__)

    from .views import main_views
    app.register_blueprint(main_views.testbp)

    return app
```

127.0.0.1:5000/ 127.0.0.1:5000/

127.0.0.1:5000

Hello, Flask!

127.0.0.1:5000/steve

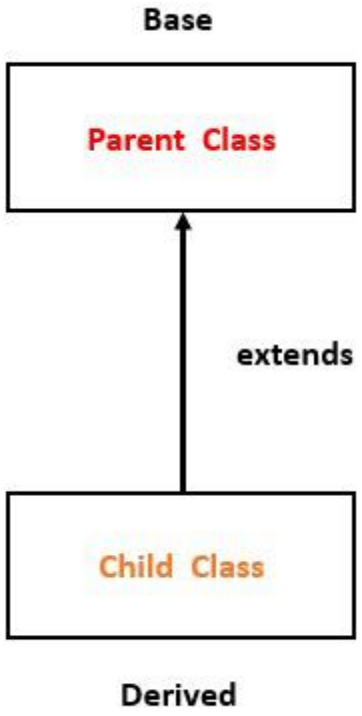
127.0.0.1:5000/steve

Hello, Steve!

Flask templates

Flask inheritace

- Template inheritance d a base “skeleton” template that contair elements of your site and defines blocks tr can override.



Template inheritance in a nutshell

~/projects/webp/app
└── __init__.py

```
from flask import Flask, render_template

def create_app():
    app = Flask(__name__)

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

    return app
```

~/projects/webp/app/templates
└── hello.html

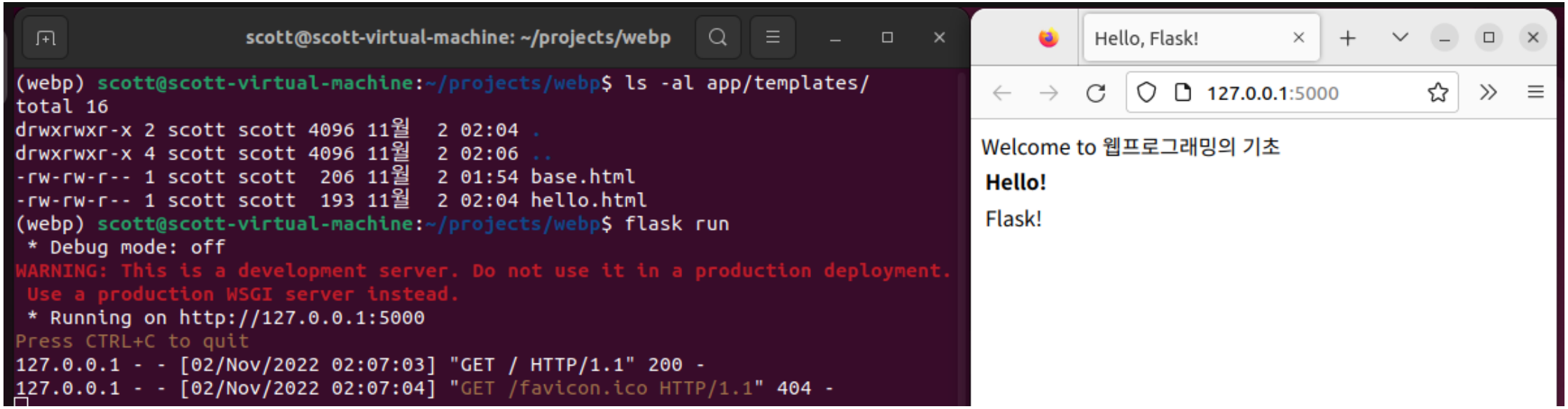
```
{% extends 'base.html' %}
{% block content %}
    <table>
        <th>
                                Hello!
        </th>
        <tr>
            <td>
                                Flask!
            </td>
        </tr>
    </table>
{% endblock %}
```

~/projects/webp/app/templates
└── base.html

```
<!doctype html>
<html lang="ko">
<head>
    <meta charset="utf-8">
    <title>Hello, Flask!</title>
</head>
<body>
Welcome to 웹프로그래밍의 기초
{% block content %}
{% endblock %}
</body>
</html>
```

Template inheritance in a nutshell (Cont'd)

- execute ‘flask run’ command at the current locations.



Flask blueprint + template

~/projects/webp/app/templates
└── base.html

```
<html lang="ko">
<head>
    <meta charset="utf-8">
    <title>Hello, Flask!</title>
</head>
<body>
<P>Welcome to 웹프로그래밍의 기초</P>
{% block content %}
{% endblock %}
</body>
</html>
```

~/projects/webp/app/templates
└── hello.html

```
{% extends 'base.html' %}
{% block content %}
    <table border="1" align="left" width=200>
        <th>
                                Hello!
        </th>
        <tr>
            <td align="center">
                                {{name.title()}}
            </td>
        </tr>
    </table>
{% endblock %}
```

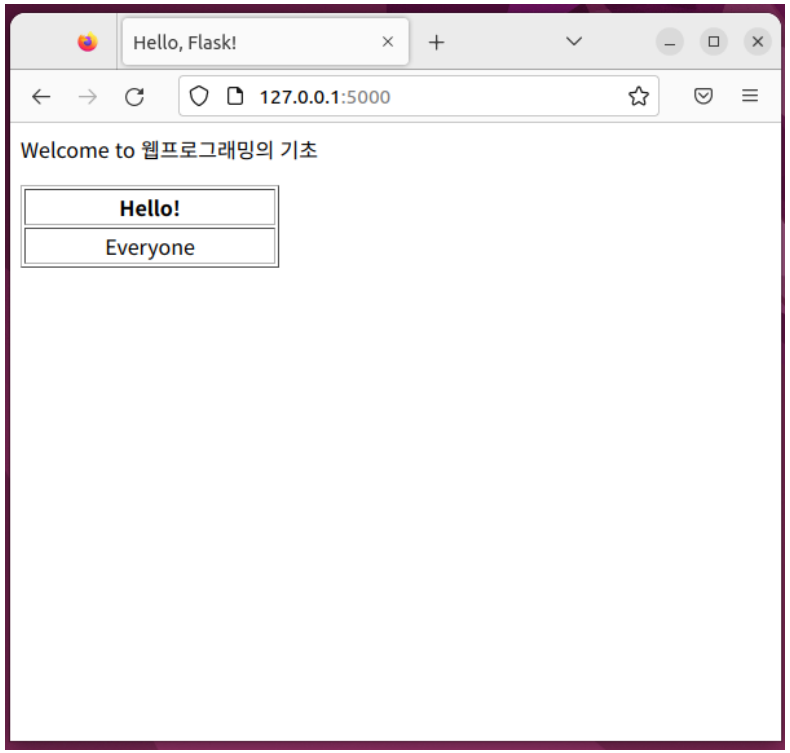
~/projects/webp/app
└── __init__.py

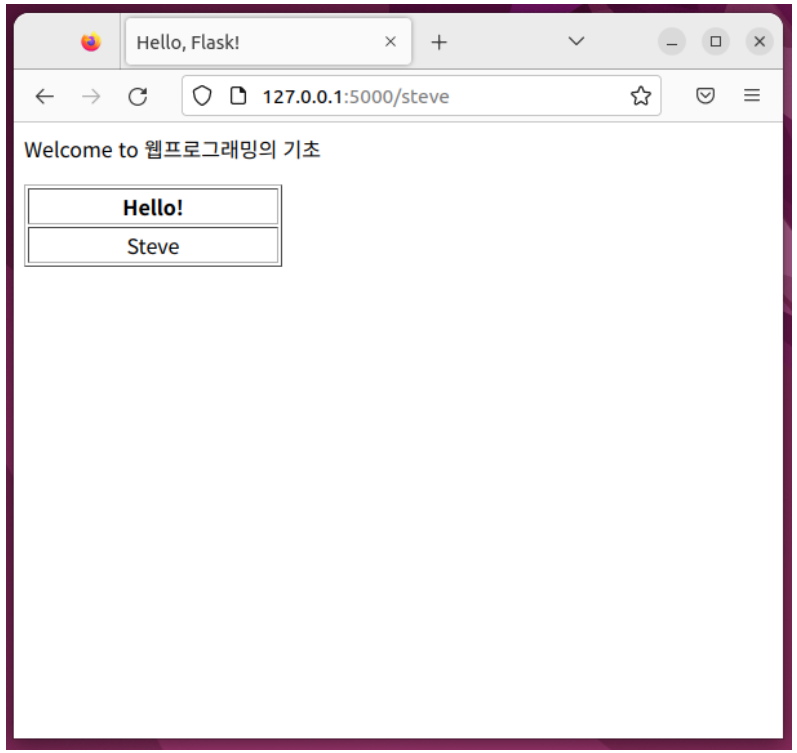
```
from flask import Flask

def create_app():
    app = Flask(__name__)

    from .views import main_views
    app.register_blueprint(main_views.bp)

    return app
```

~/projects/webp/app/views
└── main_views.py

```
from flask import Blueprint, render_template

bp = Blueprint('main', __name__, url_prefix='/')

@bp.route('/')
def hello():
    return render_template('hello.html', name='everyone')

@bp.route('/steve')
def hello_steve():
    return render_template('hello.html', name='steve')
```

Output


Hello, Flask!
127.0.0.1:5000
Welcome to 웹프로그래밍의 기초
Hello!
Everyone


Hello, Flask!
127.0.0.1:5000/steve
Welcome to 웹프로그래밍의 기초
Hello!
Steve