

Webアプリ動作環境

...

Python(uWSGI/Flask)

Nginx + Python(uWSGI)

サービス構成

Pythonで動作するWebアプリのサービス構成は

- Webサーバー：Nginx
- Webアプリ動作方法(アプリケーション・サーバー)：uWSGI
- データベース：MySQL or SQLite

これらのサービスをDockerのコンテナで動作させます。Pythonをコンテナで利用できるようにして、WSGI準拠のuWSGIモジュールを介してWebアプリのフレームワークを使ってWebアプリを動作させます。WebアプリのフレームワークはFlaskを使用します。

*SQLiteはPython3の標準で利用できます。MySQLを使用する場合はコンテナを作成してください。
(後述)

ファイル構成

環境構築のためのファイル構成は次のようになります。

docker-uwsgi	# 開発環境ディレクトリ
├── app	# Webアプリ格納ディレクトリ
│ ├── Dockerfile	# Pythonコンテナを生成するための情報
│ ├── app.py	# 動作確認用アプリ
│ ├── reload.txt	# uWSGI再起動用ファイル(中身は空白)
│ ├── requirements.txt	# Pythonパッケージリスト
│ ├── static	# Webアプリが扱う静的ファイルを格納
│ │ └── style.css	# 動作確認用CSSファイル
│ ├── templates	# HTMLのテンプレートファイルを格納
│ │ ├── base.html	# HTMLファイルすべてのテンプレート
│ │ └── sample.html	# 動作確認用テンプレート
│ └── uwsgi.ini	# uwsgiモジュールの設定ファイル
├── docker-compose.yml	# Dockerイメージとコンテナを生成・起動するための情報
├── nginx	# Nginxサーバーの設定を格納するディレクトリ
│ └── nginx.conf	# Nginxの設定ファイル

docker-compose.yml ①

YAML形式のファイルを作成し「docker-compose」コマンドでファイルに記述されたコンテナのイメージ作成・起動を行います。(docker-compose.yml)

```
version: '3.8'
services:
  uwsgi:
    build: ./app
    container_name: uwsgi
    volumes:
      - ./app:/var/www
    ports:
      - "3031:3031"
    restart: always
```

version:	Compose ファイルのバージョン
services:	生成・起動するサービス(コンテナ)を設定
uwsgi:	サービス名(任意の名前)
build:	Dockerfileのあるディレクトリを指定
container_name:	コンテナ名(任意の名前)
volumes:	コンテナ内とホストOSのディレクトリを関連付ける Webアプリのディレクトリを指定(ホスト:コンテナ)
ports:	公開用のポート番号(ホスト:コンテナ)
restart:	コンテナが停止すると常に再起動する(always) 例えばコンテナ動作中にDockerが終了するとコンテナは停止するが、Dockerを起動すると自動で再起動する

docker-compose.yml ②

nginx:

image: nginx:latest

container_name: nginx

ports:

- "8000:80"

volumes:

- ./nginx/nginx.conf:/etc/nginx/conf.d/default.conf

depends_on:

- uwsgi

restart: always

nginx:	サービス名(任意の名前)
image:	DockerHubにあるイメージをそのまま利用するとき指定
container_name:	コンテナ名(任意の名前)

volumes:	コンテナ内とvolumes:を関連付ける この設定でホストOSにあるNginx設定を使う
depends_on:	コンテナの起動順序（指定したサービスの後に起動）
restart:	コンテナが停止すると常に再起動する(always) 例えばコンテナ動作中にDockerが終了するとコンテナは停止するが、Dockerを起動すると自動で再起動する

Dockerfile

Pythonイメージを生成・起動するためのDockerfileを作成します。PythonはモジュールにFlaskとuwsgiをインストールし生成・起動します。(app/Dockerfile)

- タイムゾーンを日本にする
- Flaskとuwsgiモジュールをインストールする
- ホストOSで作成したuwsgi.ini(uwsgi設定)を有効にする

```
FROM python:3.10 # Docker Hubにあるイメージ
WORKDIR /var/www # 作業ディレクトリ
ENV TZ='Asia/Tokyo' # タイムゾーンを日本に設定
COPY requirements.txt ./ # ホストOSのファイル
RUN pip install -r requirements.txt # モジュールのインストール
CMD ["uwsgi", "--ini", "/var/www/uwsgi.ini"] # uwsgi設定の読み込み
```

uwsgi.ini

uwsgiの設定ファイル(app/uwsgi.ini)

```
[uwsgi]
wsgi-file = app.py           # ロードするWebアプリ
callable = app               # WSGIで呼び出せる名前(Flaskオブジェクト名)
processes = 4                # 実行するプロセス数
threads = 2                  # 実行するスレッド数
socket = uwsgi:3031          # uWSGIと接続するソケット(アドレス:ポート)
                              # >アドレスにはサービス名を使用
vacuum = true                # プロセス終了時にファイル/ソケットを削除
touch-reload = reload.txt    # 指定されたファイルが変更されるとuWSGIをリロード
```


requirements.txt

インストールするPythonモジュールを記述します。次の2つは必須です。
(app/requirements.txt)

```
flask  
uwsgi
```

reload.txt

Webアプリ修正(Pythonコードを変更・削除)を行ったときに、サーバーを再起動せずにuWSGIだけをリロードするための空ファイル(app/reload.txt)

nginx.conf

Nginxの設定ファイル(nginx/nginx.conf)

```
server {  
    listen 80;  
    server_name _;  
  
    root /var/www;  
    index index.html;  
  
    location / {  
        include uwsgi_params;  
        uwsgi_pass uwsgi:3031;  
    }  
}
```

uWSGIとNginxを接続する

app.py

Webアプリのルートコード(app/app.py)

```
from flask import Flask, render_template
import datetime
```

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

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

```
    now = datetime.datetime.now().strftime('%Y年%m月%d日%H:%M:%S.%f')
    return render_template('sample.html', page_name='トップページ！！', time=now)
```

```
@app.route('/sample')
```

```
def sample():
```

```
    now = datetime.datetime.now().strftime('%Y年%m月%d日%H:%M:%S')
    return render_template('sample.html', page_name='サンプルページ！！', time=now)
```

base.html

HTMLファイルすべてのテンプレート(templates/base.html)

```
<!DOCTYPE html>
<html lang="ja">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>{% block title %}{% endblock %}</title>
  {% block head %}
  <link rel="stylesheet" href="/static/style.css">
  {% endblock %}
</head>
<body>
  {% block content %}
  {% endblock %}
</body>
</html>
```

sample.html

動作確認用サンプルページのHTMLテンプレート(templates/sample.html)

「localhost:8000/sample」のアクセスで使用されます。

```
{% extends "base.html" %}
{% block title %}サンプル{% endblock %}
{% block content %}
    <h1>{{page_name}}</h1>
    <p>{{time}}</p>
{% endblock %}
```

※テンプレートとなるHTMLファイルはデフォルトでtemplatesディレクトリ配下に配置します。

style.css

スタイルシート(static/style.css)

```
h1 {  
    color: red;  
}
```

※staticディレクトリはFlaskを利用するときに静的なファイルI/Oを行う際のデフォルトディレクトリとなります。JavaScriptファイルなど読み込みが必要なファイルがある場合はstaticディレクトリ配下に配置します。

サーバーの生成・起動

これで一通り必要な設定ファイルなどが完成したのでDocker Composeを使ってサーバーを生成・起動します。

Docker Desktopを起動し、次のコマンドをdocker-uwsgi配下で実行します。

【生成・起動】

```
docker-compose up -d
```

このコマンド実行でディレクトリにあるdocker-compose.ymlファイルに基づいてサービスが生成・起動されます。はじめての実行時にはイメージが生成されます。

【終了】

```
docker-compose down
```

このコマンドで起動しているサービスを停止します。この場合イメージは残り、DBは保存されます。

【終了（イメージとDBの削除）】

```
docker-compose down --rmi all --volumes
```

このコマンドでサービスを停止すると共に、イメージとDBも削除されます。

動作確認

Webブラウザで次のURLにアクセスしサンプルコードで作成したページが表示されることを確認します。

localhost:8080

トップページ！！

2022年12月12日16:47:51.452238

localhost:8080/sample

サンプルページ！！

2022年12月12日16:49:07

※現在日時が日本の日時になっていることを確認しましょう。

uWSGI再起動の方法

Webアプリ開発中の間は、「ソースコード修正→動作確認」の作業を頻繁に繰り返すと思います。そこで、サーバーを起動したままでuWSGIだけを再起動(リロード)する設定をuwsgi.iniに記述しています。(touch-reload = reload.txt)

この設定により指定したファイルを更新するとuWSGIだけをリロードします。

ここではreload.txtファイルを更新するとuWSGIをリロードする設定です。

【Linux/Mac】

```
$ touch reload.txt
```

【Windows(Powershell)】

```
> Set-ItemProperty reload.txt LastWriteTime $(Get-Date)  
もしくは  
> sp reload.txt LastWriteTime $(Get-Date)
```

* コンテナを再起動することも可能です。「`docker-compose restart`」

Nginx + Python(uWSGI) + SQLite

SQLiteの追加

次の流れで環境を構築します。(Python3はSQLiteのモジュールを標準搭載)

単独コードサンプルを作成し動作確認 (uWSGIコンテナ内かホストOS内どちらでも可能)



Webアプリサンプルを作成 (PythonコードとHTMLテンプレート) し、Webブラウザを使って動作確認

DBファイルの格納場所

SQLiteのデータベースとなるファイルをdbディレクトリ配下に格納するため、dbディレクトリを作成しておきます。

```
docker-uwsgi
├── app
│   └── db (sample.db)
```

SQLiteのデータベースファイルは、接続時にファイルが存在しなければ作成されます。

なので、格納するためのディレクトリだけを用意しておきます。

単独コードサンプル作成①

Python3は標準でSQLiteを利用することが可能です。次のコードは単独で動作するサンプルです。（app/sqlite_sample.py）

標準モジュールのcontextlibのclosingを使ってwithを記述すると自動的に接続を閉じます。

```
import sqlite3
from contextlib import closing
try:
    with closing(sqlite3.connect('./db/sample.db')) as con:
        print('接続成功')

        cur = con.cursor()
        cur.execute('''CREATE TABLE IF NOT EXISTS users (
                        id INTEGER PRIMARY KEY,
                        name TEXT,
                        score INTEGER
                    )''')
        print('テーブル作成')
```

単独コードサンプル作成②

```
data = [(1, 'Yamada', 85), (2, 'Tanaka', 79)]
cur.executemany("INSERT INTO users VALUES(?, ?, ?)", data)
cur.execute("INSERT INTO users VALUES(:id, :name, :score)", {'id':3, 'name':'Suzuki', 'score':63})
#cur.execute("INSERT INTO users VALUES(1, 'Yamada', 85)")
#cur.execute("INSERT INTO users VALUES(2, 'Tanaka', 79)")
#cur.execute("INSERT INTO users VALUES(3, 'Suzuki', 63)")
print('データ挿入')
```

```
cur.execute("SELECT * FROM users WHERE score >= ?", (70,))
#cur.execute("SELECT * FROM users WHERE score >= 70")
result = cur.fetchall()
print('70点以上選択')
for id,name,score in result:
    print(f'{id}\t{name}\t{score}')
```

```
cur.execute("DROP TABLE users")
print('テーブル削除')
con.commit()
```

```
except sqlite3.Error as e:
    print(e)
```

このサンプルでは次の処理をします。

- テーブル作成
- テーブルにデータ挿入
- テーブルからデータを抽出
- テーブル削除

動作確認

コンソールに次のように表示されれば正常に動作しています。

【実行結果】

接続成功

テーブル作成

データ挿入

70点以上選択

1	Yamada	85
---	--------	----

2	Tanaka	79
---	--------	----

テーブル削除

Webアプリサンプル作成①

次のコードはWebアプリのサンプルです。

(app/sqlite_sample_web.py)

```
from flask import Flask, render_template
import datetime
import sqlite3
from contextlib import closing
```

```
dbname = './db/sample.db'
app = Flask(__name__)
```

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

```
    now = datetime.datetime.now().strftime('%Y年%m月%d日%H:%M:%S.%f')
    return render_template('sample.html', page_name='トップページ！！', time=now)
```


Webアプリサンプル作成②

```
@app.route('/sqlitesample')
def sample():
    now = datetime.datetime.now().strftime('%Y年%m月%d日%H:%M:%S')
    res = ''
    try:
        with closing(sqlite3.connect('./db/sample.db')) as con:
            res += '接続成功<br>'
            cur = con.cursor()
            cur.execute('''
CREATE TABLE IF NOT EXISTS users (
    id INTEGER PRIMARY KEY,
    name TEXT,
    score INTEGER
)''')
            res += 'テーブル作成<br>'
```

Webアプリサンプル作成③

```
cur.execute("INSERT INTO users VALUES(1, 'Yamada', 85)")
cur.execute("INSERT INTO users VALUES(2, 'Tanaka', 79)")
cur.execute("INSERT INTO users VALUES(3, 'Suzuki', 63)")
res += 'データ挿入<br>'
```

```
cur.execute("SELECT * FROM users WHERE score >= 70")
result = cur.fetchall()
res += '70点以上選択<br>'
for id,name,score in result:
    res += f'{id}\t{name}\t{score}<br>'
```

```
cur.execute("DROP TABLE users")
res += 'テーブル削除<br>'
con.commit()
```

Webアプリサンプル作成④

```
except sqlite3.Error as e:  
    app.logger.error(e)
```

エラー出力をFlaskのロギングに変更

```
return render_template('sqlitesample.html',  
                        page_name='SQLiteサンプルページ！！', time=now, result=res)
```


```
if __name__ == '__main__':  
    app.run(port=8000, debug=True)
```

デバッグ用にFlask単体で動作できるようにする。
FlaskでWebアプリを作成する場合は、単体で動作確認を行ってからサーバーに導入する場合がある。

テンプレートHTML作成

次の内容でテンプレートHTML「sqlitesample.html」をtemplatesディレクトリ配下に作成します。

```
{% extends "base.html" %}
{% block title %}SQLiteサンプル{% endblock %}
{% block content %}
    <h1>{{page_name}}</h1>
    <p>{{time}}</p>
    <p>{{result | safe}}</p>
{% endblock %}
```



テンプレートに渡された文字列のHTMLタグをタグとして機能させる

他の方法として、flaskのMarkupを使う例)

```
from flask import Markup
tx = Markup('<div>TEST</div>')
#文字列中のタグがタグとして機能する
```

動作確認

デバッグ用にFlask単体動作できるようにしているので、まずは単体動作で確認します。コンソールでコードを直接実行「python sqlite_sample_web.py」しWebブラウザで「localhost:8000/sqlitesample」にアクセスして次のように画面に表示されれば正常に動作しています。

正常に動作すればコンソールに次のように表示されます。

```
127.0.0.1 - - [15/Feb/2023 22:57:32] "GET /sqlitesample HTTP/1.1" 200 -  
127.0.0.1 - - [15/Feb/2023 22:57:32] "GET /static/style.css HTTP/1.1" 304 -
```

もしエラーが発生したときは次のように表示されます。

```
[2023-02-15 22:55:51,076] ERROR in sqlite_sample_web: UNIQUE constraint failed: users.id  
127.0.0.1 - - [15/Feb/2023 22:55:51] "GET /sqlitesample HTTP/1.1" 200 -  
127.0.0.1 - - [15/Feb/2023 22:55:51] "GET /static/style.css HTTP/1.1" 304 -
```

SQLiteサンプルページ！！

2022年12月25日17:54:08

接続成功
テーブル作成
データ挿入
70点以上選択
1 Yamada 85
2 Tanaka 79
テーブル削除

uWSGIで動作を確認

uwsgi.iniファイルに以下の修正をしてコンテナを起動します。

```
[uwsgi]  
wsgi-file = sqlite_sample_web.py
```

```
$ docker-compose up -d
```

Webブラウザで「localhost:8000/sqlitesample」にアクセスし右のように表示されれば正常に動作しています。テーブルに格納する値を変更して動作させてもよいでしょう。

SQLiteサンプルページ！！

2022年12月25日21:26:21

接続成功

テーブル作成

データ挿入

70点以上選択

1 Yamada 85

2 Tanaka 79

テーブル削除

SQLiteのTIPS

【AUTOINCREMENT】

カラムにINTEGER PRIMARY KEYを指定している場合、そのカラム以外だけで値を挿入した場合は自動的にそのカラムに値が「現在の最大値+1」で挿入されます。以前に削除されたレコードがあったとしてもその数値と同じものが設定されます。

例えば4が最大値のときに、4のレコードを削除し3が最大となる場合に次に自動で設定する値は4となります。（4のレコードを削除してもまた割り当てる）

これを回避するには、制約にAUTOINCREMENTを設定します。この場合、例えば4が最大値のときに、4のレコードを削除し3が最大となる場合に次に自動で設定する値は5となります。過去に最大であった値を記憶し、自動的にその値+1を割り当てます。

Nginx + Python(uWSGI) + MySQL

MySQLの追加

次の流れで環境を構築します。

MySQLとphpMyAdminコンテナの追加



uWSGIコンテナのPythonにmysql-connector-pythonをインストール



コンテナの起動、およびデータベースの作成



単独コードサンプルを作成し、uWSGIコンテナ内で動作確認



Webアプリサンプルを作成（PythonコードとHTMLテンプレート）し、Webブラウザを使って動作確認

MySQLコンテナの追加

MySQLとphpMyAdminツールのコンテナを追加します。docker-compose.ymlファイルに以下を追記します。

```
mysql:
  image: mysql:latest
  container_name: mysql
  volumes:
    - db_data:/var/lib/mysql
  restart: always
  environment:
    MYSQL_ROOT_PASSWORD: 'password'
  ports:
    - "3306:3306"
```

```
uwsgi:
  .....
  depends_on:
    - mysql
```

mysqlコンテナ起動後に
uwsgiコンテナ起動

```
phpmyadmin:
  image: phpmyadmin:latest
  container_name: phpmyadmin
  depends_on:
    - mysql
  environment:
    PMA_HOST: mysql
    MEMORY_LIMIT: 128M
    UPLOAD_LIMIT: 64M
  restart: always
  ports:
    - "8080:80"
  volumes:
    db_data: {}
```

データベースを永続化

MySQLモジュールのインストール

PythonでMySQLと接続するには外部モジュールを使用します。複数種類が存在しますが、ここではMySQL公式が提供しているモジュールを使用します。

requirements.txtに以下を追記します。

```
flask
uwsgi
mysql-connector-python
```

(参考)

<https://dev.mysql.com/doc/connector-python/en/connector-python-installation-binary.html>

コンテナの起動

コンテナを起動します。

【生成・起動】

```
docker-compose up -d
```

このコマンド実行でディレクトリにあるdocker-compose.ymlファイルに基づいてサービスが生成・起動されます。はじめての実行時にはイメージが生成されます。

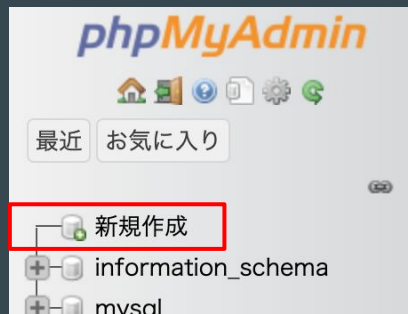
データベース作成

phpMyAdminを使ってデータベースを作成します。WebブラウザでURLを「localhost:8080」に接続するとログイン画面が表示されます。

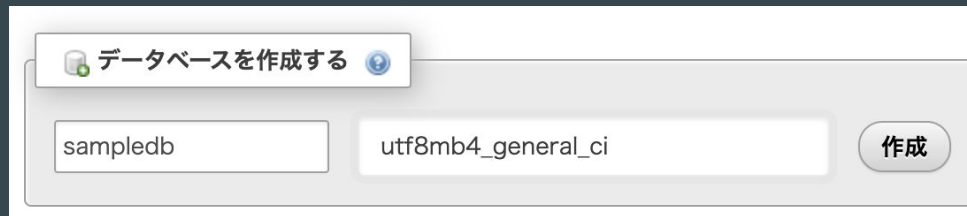


The login screen of phpMyAdmin. At the top is the phpMyAdmin logo and the text "phpMyAdmin へようこそ". Below this is a "言語 (Language)" section with a dropdown menu set to "日本語 - Japanese". Underneath is a "ログイン" section with input fields for "ユーザ名:" and "パスワード:", and a "ログイン" button at the bottom right.

ユーザ名：root
パスワード：password



データベース名：sampedb
照合順序
：utf8mb4_general_ciで作成



The screen for creating a new database. It has a title bar that says "データベースを作成する". Below this are two input fields: the first contains "sampedb" and the second contains "utf8mb4_general_ci". To the right of these fields is a button labeled "作成" (Create).

単独コードサンプル作成①

次のコードは単独で動作するサンプルです。

(app/mysql_sample.py)

Dockerコンテナ間で接続する場合、ホスト名がサービス名orコンテナ名となります。

```
import mysql.connector
from mysql.connector import errorcode
from contextlib import closing
try:
    with closing(mysql.connector.connect(user='root', password='password',
                                         host='mysql', database='sampledb')) as cnx:
        cur = cnx.cursor(prepared=True)
        print('接続成功')
        cur.execute('''CREATE TABLE IF NOT EXISTS users (
                        id INTEGER PRIMARY KEY,
                        name VARCHAR(20),
                        score INTEGER)''')
        print('テーブル作成')
```

単独コードサンプル作成②

```
cur.execute("INSERT INTO users VALUES(1, 'Yamada', 85)")
cur.execute("INSERT INTO users VALUES(2, 'Tanaka', 79)")
cur.execute("INSERT INTO users VALUES(3, 'Suzuki', 63)")
print('データ挿入')
```

```
cur.execute("SELECT * FROM users WHERE score >= 70")
result = cur.fetchall()
print('70点以上選択')
for id,name,score in result:
    print(f'{id}\t{name}\t{score}')
```

```
cur.execute("DROP TABLE users")
print('テーブル削除')
cnx.commit()
```

```
except mysql.connector.Error as err:
    print(err)
```

(参考) MySQL公式サイトの
コードサンプル
<https://dev.mysql.com/doc/connector-python/en/connector-python-examples.html>

このサンプルでは次の処理をします。

- テーブル作成
- テーブルにデータ挿入
- テーブルからデータを抽出
- テーブル削除

動作確認

Dockerコマンドを使いuwsgiコンテナ内に入って動作確認します。

【コンテナ内コマンド実行】

```
docker exec -it <コンテナ名> bash
```

uWSGIコンテナに入る場合

```
docker exec -it uwsgi bash  
root@fb6c495e7028:/var/www#
```

抜けるときは

```
root@fb6c495e7028:/var/www# exit
```

サンプル実行

```
root@fb6c495e7028:/var/www# python mysql_sample.py
```

【実行結果】

接続成功

テーブル作成

データ挿入

70点以上選択

1 Yamada 85

2 Tanaka 79

テーブル削除

Webアプリサンプル作成①

次のコードはWebアプリのサンプルです。Webアプリをサーバーで起動しFlask単体での動作確認は行いません。そのため標準出力でエラーなどの情報を確認できないので、情報をファイルに出力するようにします。

(app/mysql_sample_web.py)

Flaskのログレベルを設定し出力するログの種類を決めます。<setLevel(>

ログのレベルは DEBUG -> INFO -> WARNING -> ERROR -> CRITICAL の順で高くなります。logging.DEBUGに設定したので、全てのログが出力されます。FileHandlerで出力先をファイルにしてログレベルに応じて異なるファイルへ出力しています。

```
from flask import Flask, render_template
import datetime
import mysql.connector
from mysql.connector import errorcode
import logging
from contextlib import closing
```

```
app = Flask(__name__)
debug_handler = logging.FileHandler('debug.log')
debug_handler.setLevel(logging.DEBUG)
app.logger.addHandler(debug_handler)
error_handler = logging.FileHandler('error.log')
error_handler.setLevel(logging.ERROR)
app.logger.addHandler(error_handler)
app.logger.setLevel(logging.DEBUG)
```

Webアプリサンプル作成②

```
@app.route('/')
def index():
    now = datetime.datetime.now().strftime('%Y年%m月%d日%H:%M:%S.%f')
    return render_template('sample.html', page_name='トップページ!!', time=now)

@app.route('/mysqlsample')
def sample():
    now = datetime.datetime.now().strftime('%Y年%m月%d日%H:%M:%S')
    try:
        res = ''
        with closing(mysql.connector.connect(user='root', password='password',
                                             host='mysql', database='sampledb')) as cnx:
            cur = cnx.cursor(prepared=True)
            res += '接続成功<br>'
            cur.execute('''CREATE TABLE IF NOT EXISTS users (
                           id INTEGER PRIMARY KEY,
                           name VARCHAR(20),
                           score INTEGER)''')
            res += 'テーブル作成<br>'
```

Webアプリサンプル作成③

```
cur.execute("INSERT INTO users VALUES(1, 'Yamada', 85)")
cur.execute("INSERT INTO users VALUES(2, 'Tanaka', 79)")
cur.execute("INSERT INTO users VALUES(3, 'Suzuki', 63)")
res += 'データ挿入<br>'
```

```
cur.execute("SELECT * FROM users WHERE score >= 70")
result = cur.fetchall()
res += '70点以上選択<br>'
for id,name,score in result:
    res += f'{id}\t{name}\t{score}<br>'
```

```
cur.execute("DROP TABLE users")
res += 'テーブル削除<br>'
cnx.commit()
```

Webアプリサンプル作成④

```
except mysql.connector.Error as err:
```

```
    app.logger.error(err)
```

```
    return render_template('mysqlsample.html',
```

```
                           page_name='MySQLサンプルページ！！', time=now, result=res)
```

```
if __name__ == '__main__':
```

```
    app.run(port=8000, debug=True)
```

Flaskでログを出力するときはレベルごとに分けて出力できます。

例)

```
app.logger.debug('debug message')
```

```
app.logger.info('info message')
```

```
app.logger.warning('warning message')
```

```
app.logger.error('error message')
```

```
app.logger.critical('critical message')
```


このサンプルの設定では、debug,info,warning
を実行したときは「debug.log」に出力し、
error,criticalを実行したときは「error.log」に
出力します。

なのでサンプルでは例外が発生したときは
「error.log」に出力します。

テンプレートHTML作成

次の内容でテンプレートHTML「mysqlsample.html」をtemplatesディレクトリ配下に作成します。

```
{% extends "base.html" %}
{% block title %}MySQLサンプル{% endblock %}
{% block content %}
    <h1>{{page_name}}</h1>
    <p>{{time}}</p>
    <p>{{result | safe}}</p>
{% endblock %}
```



テンプレートに渡された文字列のHTMLタグをタグとして機能させる（エスケープを解除）

他の方法として、flaskのMarkupを使う例)

```
from flask import Markup
tx = Markup('<div>TEST</div>')
#文字列中のタグがタグとして機能する
```

*テンプレートで扱うデータはデフォルトでHTMLエスケープされる

動作確認

uwsgi.iniファイルを修正後、コンテナを再起動するか、reload.txtを更新してuWSGIを再起動します。

```
[uwsgi]  
wsgi-file =
```

```
-mysql_sample_web.py  
  【reload.txt更新】
```

< Windows(Powershell) >

```
> Set-ItemProperty reload.txt LastWriteTime $(Get-Date)
```

もしくは

```
> sp reload.txt LastWriteTime $(Get-Date)
```

< Linux/Mac >

```
$ touch reload.txt
```

【コンテナの再起動】

```
$ docker-compose restart
```

停止するときにイメージやデータベースも削除した場合

```
$ docker-compose down --rmi all --volumes
```

もしくは、停止->起動

```
$ docker-compose down
```

```
$ docker-compose up -d
```

データベースを作成してから動作確認を実施

phpMyAdminからDBを作成「localhost:8080」

動作確認

Webブラウザで「localhost:8000/msqlsample」にアクセスし、次のように画面に表示されていれば正常に動作しています。

MySQLサンプルページ！！

2022年12月29日09:45:58

接続成功

テーブル作成

データ挿入

70点以上選択

1 Yamada 85

2 Tanaka 79

テーブル削除

プレースホルダー

SQL文を実行するコードを記述する際、通常はSQLインジェクション対策としてプレースホルダーを使用します。ここではMySQLとSQLiteでの使用方法を説明します。

Pythonの場合使用するモジュールによってコードの書き方が少し変わります。

MySQLは公式が提供しているコネクタ(`mysql.connector`)を利用しています。

SQLiteは標準モジュール(`sqlite3`)を利用しています。

mysql.connectorでプレースホルダーを使用する

mysql_sample.pyを修正します。

```
import mysql.connector
from mysql.connector import errorcode
from contextlib import closing
try:
    with closing(mysql.connector.connect(user='root', password='password',
                                         host='mysql', database='sampledb')) as cnx:
        cur = cnx.cursor(prepared=True)
        print('接続成功')
        cur.execute('''CREATE TABLE IF NOT EXISTS users (
                        id INTEGER PRIMARY KEY,
                        name VARCHAR(20),
                        score INTEGER)''')
        print('テーブル作成')
```

prepared引数をTrueにすることで、プレースホルダーが使用できるようになります。

mysql.connectorでプレースホルダーを使用する

--省略--

```
data = [(1, 'Yamada', 85), (2, 'Tanaka', 79), (3, 'Suzuki', 63)]
```

```
for d in data:
```

```
    cur.execute("INSERT INTO users VALUES(?, ?, ?)", d)
```

```
#cur.executemany("INSERT INTO users VALUES(?, ?, ?)", data)
```

```
print('データ挿入')
```

```
cur.execute("SELECT * FROM users WHERE score >= ?", (70,))
```

```
result = cur.fetchall()
```

```
print('70点以上選択')
```

--省略--

executemanyを使ってリストの内容をまとめて置き換えることもできます。

プレースホルダーは、executeで第1引数に指定するSQLステートメント文字列中の?を第2引数で指定するタプルの情報に置き換えます。

文字列中の?を先頭から検知された順に、指定したタプルの先頭から置き換わります。

置き換えるデータが1つであってもタプルに変換して渡す必要があります。

sqlite3でプレースホルダーを使用する

Pythonの標準で搭載されているsqlite3は、「prepared=True」などの引数の指定は無く、デフォルトでexecuteがプレースホルダーを使用できるようになっています。方法はmysql.connectと同じですが、以下はexecutemanyの使用例と辞書の使用例を記述しています。(sqlite_sample.pyを修正し試してください。)

--省略--

```
data = [(1, 'Yamada', 85),(2, 'Tanaka', 79)]
cur.executemany("INSERT INTO users VALUES(?, ?, ?);", data)
cur.execute("INSERT INTO users VALUES(:id, :name, :score)", {'id':3,'name':'Suzuki','score':63})
print('データ挿入')
```

```
cur.execute("SELECT * FROM users WHERE score >= ?", (70,))
result = cur.fetchall()
print('70点以上選択')
--省略--
```

* sqlite3ではcursor実行時に「prepared=True」の記述を取り除くのを忘れずに
con.cursor(~~prepared=True~~)