PythonでWebアプリを作ってみよう！ 全4回で学ぶFlask入門講座 4回目

2025/10/21

Kazuma Sekiguchi

前回のAgenda

- フォームデータの受け取り
- WTFormsを利用したフォームの作成
- バリデーションを行なう
- セッションを利用する

今回のAgenda

- データベースの利用（SQLiteの利用）
- データベースの作成
- ORMの利用
- データの保存と取りだし
- データの更新
- 課題管理アプリの作成

データベースの利用

- データベースはWebアプリケーションを作るときにはほぼ必須になっている
  - データの保存や必要なデータの読み出しはデータベースに対して行なうことで実現している
  - データベースは、ある規定した構造で数字や文字列を保存しておける仕組みのこと
    - 挿入、更新、削除、取得が可能で、高速に動作するほか、取り出す際に条件を付けて取り出すことができる

データベースの利用

- 今回利用するのはSQLiteと呼ばれるデータベースソフトウェア
  - 比較的簡易なデータベースであるが、機能が少ない分高速であり、利用はしやすい
  - 大規模なサイトでは向いていないため、別のデータベースソフトウェアを使う方が無難
  - 特に大人数による同時書き込みに弱い
- Pythonでは標準でSQLite3が利用可能になっている

データベースの利用

- データベースを使う場合、別ソフトウェアであるため、接続をする必要がある
  - 接続をした上で、データベースに対して命令を出して結果を取得する
  - 命令はSQL文を使ってデータベースに対して命令を行なう
    - SQL文はデータベースで動作するプログラミング言語の一種
    - データの取得や挿入などもSQL文で記述して実現する
- データベースにデータを入れるときには、入れるためのテーブルが必要
  - テーブルを作成してそこにデータを入れていく

ORMの利用

- 最近のWebアプリケーション開発では、SQL文を直接記述しない
  - SQL文を書かなくて済むため、バグが減ることと、セキュリティ的にも向上する
  - ORMを使うことで、SQL文を書かずにデータベースを操作できる
    - O：オブジェクト、R：リレーショナルデータベース、M：マッピング
  - ORMを使うことで、他のデータベースソフトを使うことも容易に可能になる
- ORMを使うとSQLインジェクションのリスクは軽減されるが、完全ではない
  - 特に.text()など生SQLを使う場合は注意が必要
- Flaskの場合、flask_sqlalchemyライブラリーを使うとORMを使うことが可能

SQLite上にデータベースを作成する

```
from flask_sqlalchemy import SQLAlchemy
import os

app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///items.db'
app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False
db = SQLAlchemy(app)

class Item(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(50), nullable=False)
    type = db.Column(db.String(50), nullable=False)
    price = db.Column(db.Integer, nullable=False)
```

必要なライブラリをimport

作成する、利用するデータベース名

オブジェクトへの変更検知を行なうかどうか。ほとんどの場合、False設定

db変数で扱えるように指定

Itemテーブルを定義
フィールドタイプや長さなどを指定する

- class Itemでテーブル構造を定義しておく
  - 自動的にテーブルが作成される
- importでosを読み込んでおくことで、items.dbが定義されていないときだけ定義させるなどの判別が可能

データベースへの初期コード挿入

```
def initialize_database():
    db_path = os.path.join(app.instance_path, 'items.db')
    if not os.path.exists(db_path):
        with app.app_context():
            db.create_all()
            print("Database tables created.")
            initial_items = [
                {"name": "リンゴ", "type": "果物", "price": 300},
                {"name": "トマト", "type": "野菜", "price": 200},
                {"name": "鮭", "type": "魚", "price": 400}
            ]
            for item in initial_items:
                for item in initial_items:
                    db.session.add(Item(
                        name=item['name'],
                        type=item['type'],
                        price=item['price']
                    ))
                db.session.commit()
if __name__ == '__main__':
    initialize_database()
```

DBが存在するかチェック
存在しなければ挿入処理を行なう

DB接続情報などを利用する

本番では使用を避けるべき

投入するデータ

ループで回して

itemテーブルを指定

1つずつadd（挿入）していく

挿入を確定する

起動時に実行する

db.create_all()

- SQLAlchemyが管理しているすべてのモデル（db.Model のクラス）に対応したテーブルをデータベース上に作成
  - 関数テーブルがまだ存在しない場合に限り作成する（既存のテーブルは変更しない）
- テーブル構造が変更されたときに自動更新されないため、データ不整合が起きる危険性がある
  - モデルを変更しても既存のDB構造は自動更新されない
  - 変更履歴が管理されず開発チームでDBがバラバラに
  - 本番環境ではFlask-Migrateを使うのが一般的

SQLALCHEMY_TRACK_MODIFICATIONS = False

- SQLAlchemyがオブジェクトの変更検知のために内部イベントを監視する機能
  - Flaskアプリでは不要であり、Trueにするとメモリ消費と警告が出る
  - Flask公式もFalseを推奨
- SQLALCHEMY_TRACK_MODIFICATIONSはデータ変更を追跡する機能
  - Flaskでは使わないので、False指定をしておく

トランザクション

- SQLAlchemyでは、データベースへの変更はすぐに反映されず、一度「セッション」に仮保存される
- commit()をすることで初めてデータベースに反映される
- 逆にエラーが起きたときはrollback()で変更を取り消すことも可能
  - 途中でエラーが起きたときにこれまでのDB処理を全て取り消すことができる仕組みをトランザクションと呼ぶ

データベースへの処理

- 全件取得

```
items = Item.query.all()#Itemテーブルから全件取得する
```

- 条件指定取得

```
item = item = Item.query.get_or_404(item_id)#Itemテーブルから指定したitem_idのものを取得する
```

  - idは主キーなので、そのままget_or_404()で取得が可能
  - 他のフィールドで取得したい場合は異なる

```
@app.route('/items/price/<int:price>', methods=['GET'])
def get_item_by_price(price):
  item = Item.query.filter_by(price=price).first_or_404()
  return render_template('item.html', item=item)
```

filter_byを利用する

データベースへの処理

- 新規追加

```
new_item = Item(name=name, type=item_type, price=price)
db.session.add(new_item)
db.session.commit()
```

session.add()を利用

- 更新

```
item = Item.query.get_or_404(item_id)
item.price = request.form['price']
db.session.commit()
```

データを取得しておく

データを置換える

コミットして更新する

データベースへの処理

- 削除

```
item = Item.query.get_or_404(item_id)
db.session.delete(item)
db.session.commit()
```

データを取得しておく

削除するデータを指定

コミットして削除

.first_or_404()と.get_or_404()

- first_or_404()は最初の1件だけを返す
  - 複数の結果がある場合は.all()を使う
  - 例えば同じ価格の商品を一覧で表示したい場合は、

```
Item.query.filter_by(price=price).all()
```

  のようにする
- .get_or_404()は主キー（PRIMARY KEY）でデータを1件だけ取得するためのメソッド
  - 主キー専用で高速検索ができる
  - .first_or_404()は検索条件があるときに使う

データベースの安全性

- データベースは多量のデータを保存可能
  - 個人情報なども含まれる可能性もある
  - 外部からSQL文を入れられた場合に実行されることがあれば、データを全て抜き取られる可能性も出てくる
    - SQLインジェクション攻撃
- flask_sqlalchemyを使い、ORMで記述をしている限り、ユーザーがSQL文を送ってきたとしても適切に処理される
  - こちらがSQL文を使うように記述しない限りは安全
  - ユーザーが正しいデータを送ってくるかは分からないので、バリデーション処理は必要
    - これはデータベースとはまた違った話

課題管理アプリの作成

- 課題名、期日、課題内容、状態を入力して登録すると、一覧画面に登録した内容が表示される
  - 状態は初期状態では未完了
  - 全てが必須扱い
- 状態変更ボタンを押すと、状態が完了状態になり、完了日として状態変更を押したときの日にちが登録される

ありがとうございました。

Flaskはデータベースと
組み合わせると、幅が拡がります。
いろいろなものを作成してみてください