**5 Szkolenie z Flaska**

**SQLAlchemy, integracja z bazą danych**

Nadeszła wiekopomna chwila! W szkoleniu tym przedstawię Ci, w jaki sposób zintegrować tworzoną aplikację z zewnętrzną bazą danych. Nie muszę chyba tłumaczyć Ci, czemu może służyć takie połączenie. Bez miejsca na trwałe przechowywanie informacji, nie obejdzie się obecnie żadna praktyczna aplikacja. Chcąc dostarczać jak największy UX, musimy, np. zapamiętywać wybory użytkowników czy podawanie przez nich informacje.

Szczególnie łatwą i intuicyjną pracę z danymi umożliwi zewnętrzna biblioteka **SQLAlchemy**, czyli sposób na wprowadzenie wyższego poziomu operacji na bazie danych!

Importując bowiem do tworzonej aplikacji wyżej wymienioną bibliotekę, zmieniamy tradycyjne wykonywanie operacji na bazie danych - czyli przy użyciu SQLa - na **obiektowy odpowiednik**.

Jednym słowem, nie jest już konieczne ręczne pisanie poleceń, a korzystanie jedynie **z wbudowanych metod**, które operować będą na, np. stworzonych tabelach.   
  
A to tylko jedno z wielu ułatwień. Możemy również:

* Automatycznie tworzyć i usuwać określone tabele
* Odczytywać, zapisywać, modyfikować bazy i tabele bez znajomości SQL-a, przy wykorzystaniu podstawowych metod w postaci .update(), .add() etc.
* Wszelkie struktury traktować jak klasy i obiekty

Tak więc, aby umieścić w tabeli określone dane, standardowo musielibyśmy wykonać polecenie:

| INSERT INTO Students(Name, Surname, Age) VALUES(‘Devs’, ‘Mentoring’, 2); |
| --- |

Tymczasem przy wykorzystaniu SQLAlchemy, kod możemy znacznie uprościć do poniższej postaci:

| student = Student(name='Devs', surname='Mentoring', age=2) database.session.add(st) |
| --- |

I to właśnie dzięki tak wykorzystanej technice **ORM** (Object Relational Mapping), możemy w intuicyjny sposób transferować dane. Kod staje się znacznie łatwiejszy do debugowania, elastyczny i nie musimy martwić się trudne do wyłapania literówki, które to często mogą się pojawiać przy pisaniu SQL-owych poleceń.

**Wstęp do SQLAlchemy**

Zanim przejdziemy do ulepszenia stworzonego we wcześniejszych szkoleniach projektu i dodania do niego integracji z bazą, musimy zaznajomić się z pojedynczymi poleceniami i podstawowym workflow służącym do obsługiwania bazy danych i jej tabel przy użyciu omawianego frameworka.

**Krok I:**

Aby móc efektywnie korzystać SQLAlchemy, należy do Naszego venv-a doinstalować moduł SQLAlchemy.

| pip install Flask-SQLAlchemy |
| --- |

oraz zaimportować go w samym już projekcie:

| from flask\_sqlalchemy import SQLAlchemy |
| --- |

**Krok II:**

Skonfiguruj aplikację, aby nawiązała połączenie z dowolną bazą danych (jeżeli nie będzie znaleziona, to zostanie automatycznie utworzona).

| app = Flask(\_\_name\_\_) app.config['SQLALCHEMY\_DATABASE\_URI'] = 'sqlite:///nazwa\_bazy.sqlite3' db = SQLAlchemy(app) |
| --- |

Podając ścieżkę sqlite://nazwa\_bazy.sqlite3 dla wartości klucza **SQLALCHEMY\_DATABASE\_URI**, program odniesie się do katalogu projektowego i odczyta (lub utworzy) plik nazwa\_bazy.sqlite3.   
  
To miejsce, w którym lokalnie przechowywana będzie Nasza baza danych i jej rekordy (więcej za chwilę).

**Krok III:**

Utworzenie odpowiednich tabeli. Przykładowa tabela Users:

Plik main.py

| app = Flask(\_\_name\_\_) app.config['SQLALCHEMY\_DATABASE\_URI'] = 'sqlite://nazwa\_bazy.sqlite3' db = SQLAlchemy(app)  class Users(db.Model):  \_id = db.Column(db.Integer, primary\_key=True)  name = db.Column(db.String(80), unique=True)  email = db.Column(db.String(100), unique=True)   def \_\_init\_\_(self, name, email):  self.name, self.email = name, email |
| --- |

W gwoli ścisłości - \_id jest polem prywatnym klasy Users (stąd podkreślnik w formie prefixu) oraz z racji, iż jest on kluczem głównym, następuje jego automatycznie inkrementacji podczas tworzenia nowych rekordów użytkowników. Dlatego jesteśmy zwolnieni z ręcznego nadawania mu wartości w konstruktorze klasy.

**UWAGA:**

SQLAlchemy umożliwia również tworzenie relacji między tabelami. Poniżej przykład **relacji jeden do wielu**.

| app = Flask(\_\_name\_\_) app.config['SQLALCHEMY\_DATABASE\_URI'] = 'sqlite://nazwa\_bazy.sqlite3' db = SQLAlchemy(app)  class Parent(db.Model):  id = db.Column(db.Integer, primary\_key=True)  children = relationship("Child")  class Child(db.Model):  id = db.Column(db.Integer, primary\_key=True)  parent\_id = db.Column(db.Integer, ForeignKey('parent.id')) |
| --- |

**Krok IV:**

Utworzenie tabeli.

Aby móc operować na stworzonej tabeli Users, musimy dodać ją do istniejącej bazy danych. Następuje to przez polecenie **db.create\_all()**, które powinniśmy wywołać **jednorazowo**.

W tym celu, skorzystamy z terminala i wykonamy następujące polecenia:

* Przejdź do odpowiedniego katalogu (powinien to być katalog nadrzędny dla Twojej paczki projektowej)
* Uruchom interpreter Pythona poleceniem:

**py** na systemie Windows

**python** na systemach Unix-owych

* Zaimportuj instancje db oraz app z odpowiedniego pliku (załóżmy, że kod z przykładu zapisaliśmy w pliku project.py)

from project import db

db.create\_all()

**lub gdy korzystasz z metody create\_app()**:

from project import db, create\_app

db.create\_all(app=create\_app())

**Przykładowe operacje DML, DQL, DDL:**

Dodawanie do tabeli odpowiednich rekordów. Odpowiednik **INSERT INTO**:

| from main import db for i in range(10):  new\_user = Users(f'Jan {i}', f'jan\_{i}@devs-mentoring.pl')  db.session.add(new\_user) # dodanie rekordu  db.session.commit() # zatwierdzenie zmian |
| --- |

Wybór rekordów z tabeli. Odpowiednik **SELECT**:

| from main import db found\_user = Users.query.filter\_by(name='Jan 1').first()  print(found\_user.email) # prints jan\_1@devs-mentoring.pl |
| --- |

| from main import db for user in Users.query.order\_by(Users.name):  print(user.name, user.email) # prints users in ascending order by their names |
| --- |

| from main import db found\_user = Users.query.get(1) # gets user by primary\_key (id) print(found\_user.email) # prints jan\_1@devs-mentoring.pl |
| --- |

Usuwanie rekordów w tabeli. Odpowiednik **DELETE FROM**:

| from main import db Users.query.filter\_by(name='Jan 2').delete() db.session.commit() |
| --- |

Aktualizacja danych w tabeli. Odpowiednik **UPDATE**:

| from main import db found\_user = Users.query.filter\_by(name='Jan 2').first() found\_user.email = 'new\_email@gmail.com'  db.session.commit() |
| --- |

**Ulepszenie dotychczasowego projektu**

Po zapoznaniu się z podstawami SQLAlchemy, Naszym zadaniem będzie zmodyfikowanie projektu tak, aby podawane dane były zapamiętywane w bazie oraz z daną nazwą użytkownika kojarzony był konkretny adres e-mail.   
Będzie to więc prosta symulacja procesu rejestracji/logowania do konta w serwisie.

Jeżeli użytkownik w formularzu poda po raz pierwszy swoją nazwę, to zostanie zarejestrowany i zapamiętany w bazie. Z poziomu dashboarda będzie miał również możliwość uzupełnienia swojego profilu o adres email.

Następnie, gdy przy kolejnym podejściu do logowania, poda tę samą (zarejestrowaną już) nazwę, to zostanie przekierowany do profilu, na którym widnieć będzie podany już wcześniej adres mailowy. A to wszystko dzięki integracji aplikacji Flaskowej z bazodanową technologią!

**Implementacja w praktyce**

Bazową wiedzę już mamy! Teraz czas na wykorzystanie wszystkich wiadomości w praktyce.

**Plik \_\_init\_\_.py**

| from flask import Flask from flask\_sqlalchemy import SQLAlchemy from hashlib import md5 from datetime import timedelta  db = SQLAlchemy()  def create\_app():  app = Flask(\_\_name\_\_)  encryptor = md5()   app.permanent\_session\_lifetime = timedelta(minutes=30)  app.secret\_key = encryptor.digest()  app.config['SQLALCHEMY\_DATABASE\_URI'] = 'sqlite:///db.sqlite3'  db.init\_app(app)   app.debug = True   from .main import login\_blueprint, dashboard\_blueprint, logout\_blueprint  app.register\_blueprint(login\_blueprint)  app.register\_blueprint(dashboard\_blueprint)  app.register\_blueprint(logout\_blueprint)    return app |
| --- |

**Wyjaśnienie**:

W powyższym pliku, bazę danych zintegrowaliśmy z aplikacją przy użyciu metody **init\_app()**. A to wszystko na rzecz stworzenia **globalnego obiektu bazy (db)**, którego możliwe będzie importowanie między plikami (np. w momencie, gdy będziemy chcieli odwołać się do niej z poziomu tabeli znajdującej się w innym pliku niż **\_\_init\_\_.py**).

Równocześnie pamiętaj, aby przed pierwszym uruchomieniem wywołać polecenie:

from nazwa\_aplikacji import db, create\_app

db.create\_all(app=create\_app())

z poziomu interpretera Pythona (uruchamianego przez polecenie py na systemie Windows lub python na systemach Unixowych).

**Plik models.py** (na tabele bazy danych):

| from . import db  class Users(db.Model):  \_id = db.Column(db.Integer, primary\_key=True)  name = db.Column(db.String(80), unique=True)  email = db.Column(db.String(100), unique=True)  def \_\_init\_\_(self, name, email):  self.name, self.email = name, email |
| --- |

**Wyjaśnienie:**

Zauważ, że to właśnie tutaj przydało się globalne utworzenie obiektu db w **\_\_init\_\_.py** w celu zaimportowania obiektu (from . import db) i użycia go w ustaleniu hierarchii dziedziczenia klasy Users.

**Plik main.py**

| from flask import render\_template, redirect, url\_for, Blueprint, request, session, flash from .models import Users from . import db  login\_blueprint = Blueprint("login", \_\_name\_\_) dashboard\_blueprint = Blueprint("dashboard", \_\_name\_\_) logout\_blueprint = Blueprint("logout", \_\_name\_\_)   @login\_blueprint.route('/', methods=["POST", "GET"]) def login():  if request.method == "POST":  nickname = request.form['nickname']  found\_user = Users.query.filter\_by(name=nickname).first()   if not found\_user:  new\_user = Users(name=nickname, email=None)  db.session.add(new\_user)  db.session.commit()  session.update({"nick" : nickname, "email" : None})   flash("You've been successfully registered.", "success")  else:  session.update({"nick" : nickname, "email" : found\_user.email})  flash("You've been successfully logged in.", "success")   elif request.method == "GET" and "nick" not in session:  return render\_template("login.html")  elif request.method == "GET" and "nick" in session:  flash("Already logged in!", "warning")   return redirect(url\_for("dashboard.dashboard"))   @logout\_blueprint.route('/logout') def logout():  if "nick" in session:  session.pop("nick", None)  session.pop("email", None)  flash("You have been logged out!", "success")  else:  flash("You are not logged in!", "warning")   return redirect(url\_for("login.login"))   @dashboard\_blueprint.route('/dashboard', methods=["POST", "GET"]) def dashboard():  if "nick" in session:  nickname = session["nick"]  email = session["email"]   if request.method == "POST":  found\_user = Users.query.filter\_by(name=nickname).first()  email = request.form["email"]  found\_user.email = email  db.session.commit()  session.update({"email" : email})   return render\_template("dashboard.html", nickname=nickname, email=email)  else:  flash("You are not logged in!", "warning")   return redirect(url\_for("login.login")) |
| --- |

**Wyjaśnienie:**

Plik main.py operuje zarówno na sesji jak i utworzonej bazie danych. Sesja służy temu, by zapewnić łatwiejszy dostęp do informacji o zalogowanym użytkowniku (nie musimy za każdym razem odczytywać informacji z bazy danych).

**Plik dashboard.html:**

| {% extends "base.html" %} {% block title %} Dashboard {% endblock %} {% block content %}  <**h1**>You're logged as: {{nickname}}</**h1**>  <**form** action = "#" method = "POST">  <**input** type = "email" name = "email" placeholder = "Enter you email" value = "{{ email if email }}" />  <**input** type = "submit" value = "save" />  </**form**> {% endblock %} |
| --- |

Pliki base.html, index.html pozostaje bez zmian w stosunku do wcześniejszych szkoleń.

**Dodatek**

W dodatku dołączonym do tego szkolenia chciałbym zaprezentować Ci jednolity szablon (z którego sam korzystam w pracy komercyjnej) na tworzenie mniej i bardziej rozbudowanych projektów we Flasku. Dodatkowo szablon ten będzie się opierał tylko na operacjach w kodzie, bez dokładania sobie konieczności wykonywania poleceń w terminalu (np. w celu utworzenia bazy danych).

Takie rozwiązanie jest owocem mojego doświadczenia oraz faktu, że jestem zwolennikiem pisania kodu bez konieczności co rusz przeskakiwania z IDE do terminala. Dlatego też opracowałem tak ogólny wzorzec postępowania i opracowywania projektów we Flasku.

**Dlaczego nie pokazałem tego wcześniej?**

Jeżeli zastanawiasz się, dlaczego nie pokazałem Ci do tej pory tak rozchwytywanego przeze mnie sposobu na podział plików w projekcie, to wiedz, że nie uczyniłem tego z uzasadnionego powodu. Chciałem, abyś doświadczył na własnej skórze różnych rozwiązań na tworzenie Flaskowej struktury plików. Dodatkowo, moim celem było danie Ci możliwości zetknięcia się z powszechnymi problemami, gdy próbujesz wykonać pewne funkcjonalności w terminalu etc.

Wspomniany wzorzec pokazujący uniwersalny rozkład plików w projekcie znajdziesz tutaj: [Projekt Wzorcowy](https://github.com/Devs-Mentoring/Backend-Path-Materials/tree/master/Flask/flask-project-template)

Omówmy przedstawioną strukturę projektu wraz z opisem katalogów i plików, które się w obrębie takiego projektu znajdują.

Struktura wzorca:

| |  │ run.py  │  ───app  │ constants.py  │ main.py  │ models.py  │ my\_database.db  │ \_\_init\_\_.py  │  ─── static  │ base.css  │  ─── templates  │ base.html  │ dashboard.html  │ index.html |
| --- |

Opis katalogów i plików:

* W katalogu projektowym znajduje się podkatalog app/
* Bezpośrednio w katalogu projektowym nad app/, znajduje się plik run.py. Służy on uruchamianiu aplikacji Flaskowej i tworzeniu pliku bazy danych
* W katalogu app/ znajduje się całe “mięcho aplikacji”. To tutaj umieszczamy znaną Ci strukturę projektu Flask, czyli pliki takie jak: \_\_init\_\_.py, main.py, models.py, constants.py
* Dodatkowo w app/ znajdują się takie katalogi jak static/ jak i html/

Omówienie nowych funkcjonalności:

* W zasadzie w plikach tak utworzonego projektu nie wprowadzono żadnych nowych konceptów w stosunku do tych, które poznałeś uprzednio. Jedyną nowością może być funkcja create\_db umieszczona w pliku run.py.

| def create\_db(app):  with app.app\_context():  if not os.path.exists(DB\_PATH):  db.create\_all() |
| --- |

Wykorzystuje ona znany już Ci zapis db.create\_all(), który odpowiada za tworzenie pliku bazodanowego. Aby dobrze zrozumieć logikę działania omawianej funkcji, rozbijmy ją na czynniki pierwsze:

with app.app\_context() - zapis mówiący, że kod wykonywany w obrębie takiego bloku with, należy do kontekstu aplikacji. Jest to niezbędny zapis, aby cała aplikacja mogła po prostu poprawnie działać, a plik bazodanowy mógł być tworzony poza plikiem, w którym definiowana jest instancja Flask.

if not os.path.exists(DB\_PATH) - linia sprawdzająca, czy istnieje plik bazy danych pod wskazaną ścieżką DB\_PATH (która została umieszczona w pliku constants.py. Cel umieszczenia takiej linii w kodzie jest prosty - nie chcemy nadpisywać zawartości bazy danych i wywoływać polecenia db.create\_all() przy każdym jednorazowym uruchomieniu aplikacji. Linia ta po prostu sprawdza, czy istnieje zasób pod wskazaną ścieżką - jeżeli nie - tworzy go, a w przeciwnym razie pozostawia w nienaruszonym stanie.