**6 Szkolenie z Flaska**

**REST API**

**Wstęp**

Chcąc tworzyć efektywne i funkcjonalne rozwiązania, które będą realizowały podstawowe założenia designu i będą na tyle użyteczne, iż użytkownik będzie mógł z łatwością wykonywać operacje CRUD (Create, Read, Update, Delete), na pewno konieczna jest integracja tworzonej aplikacji z bazą danych. Jak to zrealizować, poznaliśmy już w poprzednim szkoleniu dotyczącym Flaska...

Programiści jednak często chcą iść krok dalej i tworzyć uniwersalne i wieloplatformowe aplikacje, z którymi łączyć się będą z poziomu różnych platform - np. aplikacji mobilnej czy strony internetowej. Przyznaj, że głupotą byłoby tworzenie dedykowanych baz danych dla każdej aplikacji z osobna. Raczej wolelibyśmy utworzyć jedną bazę i umożliwić dostęp do niej z poziomu niezależnych instancji aplikacji.

Poniższe szkolenie opisywać właśnie będzie, w jaki sposób tworzyć własne API (czyli pośrednika między naszą aplikacją a bazą), przez które będziemy mogli pobierać, zmieniać, dodawać dane do bazy danych. Tak więc dowolna aplikacja mobilna czy strona internetowa, będzie po prostu komunikowała się z takim rozwiązaniem, wysyłała do niego żądania i odpowiednio operowała na postawionej bazie.

Bez potrzeby pisania oddzielnej logiki dla poszczególnych platform, na których działa aplikacja.

Oczywiście to tylko jeden z wielu przykładów, gdzie wykorzystanie znajduje implementacja aplikacji typu REST API. Tego typu rozwiązanie jest również szeroko wykorzystywane w momencie, gdy chcemy integrować frontend aplikacji z backendem (np. gdy potrzebujemy z poziomu frontu triggerować polecenia dynamicznie modyfikujące zasobami w bazie). Temat jest dość szeroki i wymaga dogłębnego zrozumienia, dlatego czytaj dalej, a w razie konieczności przedyskutuj wątpliwości ze swoim mentorem.

**REST API**

Zanim przejdziemy do konkretów, musimy zagłębić się w teorię i wyjaśnić, z czym jest związane choćby pojęcie REST i API.

**REST - Representational State Transfer** **-** to styl architektury, który określa, w jaki sposób definiowane są zasoby (np. w bazie danych) oraz w jaki sposób uzyskiwać do niej dostęp.

**API - Application Programming Interface -** to zbiór reguł definiujących, w jaki sposób klient może komunikować się z serwerem i uzyskiwać dostęp do określonych zasobów.

Tak więc REST API oznacza całokształt systemu, w którym w określony sposób (przez API) user może manipulować na zasobach (zbudowanych zgodnie ze stylem REST).

**Więcej o REST**

Jak już się zapewne domyślasz, opisywane rozwiązanie jest niezwykle elastyczne i powszechne w użyciu. Przy jego tworzeniu jednak należy mieć szczególną uwagę na kilka zasad, które oceniają, jak bardzo skuteczne jest REST API.   
  
Zdefiniowane one zostały przez Roy’a Fielding’a i brzmią:

* **Client-Server -** w komunikacji powinniśmy bazować na połączeniu Klient-Serwer. Czyli, np. użytkownik korzystający z front-endu sklepu internetowego, wysyła przez API żądania do serwera, który je przetwarza i dodaje zakupiony produkt do koszyka.

Dzięki temu klient jest niezależny od serwera, a serwer niezależny od klienta. Oznacza to, iż klient nie może wpływać na działanie serwera i vice versa. Jedynie mogą się komunikować przez stworzone API.

* **Stateless -** każde żądanie wykonane przez klienta jest niezależne od pozostałych żądań. Serwer nie przechowuje żądań klienta, które mogłyby wpływać na dalsze operacje. Jeżeli będzie potrzeba ich przechowywania, to będą one zapamiętywane po stronie klienta (np. w pamięci cache).
* **Cache -** pamięć podręczna powinna również być wykorzystywana w momencie, gdy chcemy poprawić wydajność aplikacji i, np. nie odwoływać się za każdym razem do bazy danych w celu pobrania pewnych informacji. Przykładem może być sesja i zapamiętywanie w niej pobranego z bazy emailu użytkownika po poprawnym zalogowaniu.
* **Layered system** - wielowarstwowość polega na tym, iż Nasz system powinien działać tak samo wydajnie, niezależnie od tego, czy API dotyczy tylko jednego serwera, czy napotyka po drodze również warstwy pośredniczące, np. load balancer.
* **Code on Demand** - umożliwia przesyłanie od strony serwera pewnych fragmentów kodu, np. JS dla klienta. W praktyce rzadko wykorzystywane.
* **Uniform Interface -** dotyczy tego, iż możemy w różny sposób reprezentować, np. zasoby (API może zwracać pobrane dane z bazy w postaci JSON-a, XML-a czy HTML-a). Jednolitość danych polega też na tym, iż choćby w nagłówku Content-Type protokołu HTTP, możemy odczytywać, w jaki sposób zostały zakodowane dane.

**Więcej o API**

Aby móc lepiej wyobrazić sobie, jak wygląda praca z interfejsem aplikacji, przeanalizujmy cykl komunikacji:

1. Klient preparuje zapytanie w postaci odpowiedniego adresu URI (endpoint), np. devs-mentoring.pl/students
2. Klient wysyła przygotowane zapytanie (request),
3. System otrzymuje zapytanie klienta i przygotowuje odpowiedź (response),
4. System zwraca odpowiedź na zapytanie klienta,
5. Klient otrzymuje i przetwarza odpowiedź.

Jednak jest jeszcze jedna rzecz, w zasadzie nieodzowna dla powyższych kroków. Otóż, aby użytkownik mógł skutecznie komunikować się z serwerem - czyli wysyłać żądania i je odbierać, musi przecież korzystać z jakichś zestawu reguł umożliwiających wymianę informacji. Taki zestaw reguł jest definiowany przez powszechny protokół HTTP. Przeanalizujmy jego budowę.

**Budowa zapytania HTTP (request)**

Żądanie w formie HTTP powinno składać się z następujących elementów:

* **Request method** - za jej pomocą określamy, jakie żądania HTTP chcemy wykonać, np. GET, POST, PUT, DELETE.
* **Header** - w nagłówku przechowujemy dodatkowe informacje o żądaniu, np. informacje dotyczące plików cookie, kodowania danych.
* **Resource Path** - ścieżka dostępowa do danego zasobu, czyli, tzw. URI
* **Body** - pole opcjonalne. Miejsce do umieszczania dodatkowych informacji, np. danych użytkownika, którego chcemy utworzyć w bazie danych po stronie systemu.

Przykładowy HTTP Request z metodą GET i nagłówkami User-Agent, Host, Accept-Language, Accept-Encoding, Connection, bez pola body:

| GET /devs-mentoring.pl/articles User-Agent: Mozilla/4.0 (compatible; MSIE5.01; Windows NT) Host: www.tutorialspoint.com Accept-Language: en-us Accept-Encoding: gzip, deflate Connection: Keep-Alive |
| --- |

**HTTP Request Method:**

Do najbardziej podstawowych metod wykorzystywanych w zapytaniu HTTP należą:

* GET - pobieranie danych z serwera
* POST - wysyłanie nowych danych do serwera
* PUT - podmienianie istniejących danych na serwerze
* DELETE - usunięcie danych z serwera
* PATCH - częściowa modyfikacja danych na serwerze

Na ten moment nie będę rozpisywał się nad szczegółowym rozpisaniem funkcjonalności wymienionych metod. Dogłębniejsze poznanie ich przyjdzie z czasem, szczególnie wtedy gdy będziemy tworzyli pierwsze aplikacje RESTowe.

**HTTP Header**

Tak już zauważyłeś na powyższych przykładzie, nagłówki zapytania mają postać par klucz - wartość i definiują one dodatkowe atrybuty zapytania. Umożliwiają więc one zarówno klientowi, jak i serwerowi na przekazanie dodatkowych informacji. Wyróżniamy wiele rodzajów nagłówków, a ich zastosowanie zależy wyłącznie od kreatywności i potrzeb, np. twórcy aplikacji.   
  
Do najpopularniejszych należy zaliczyć te odpowiadające za uwierzytelnianie, pamięć tymczasową, elementy związane z kodowaniem, bezpieczeństwem itd. Do wyboru do koloru!

**HTTP Resource Path**

URI podawany jest tuż po metodzie HTTP i określa on, do którego zasobu chcemy się odwołać. Pamiętaj, aby podczas projektowania RESTful API, ustalać endpointy (właśnie te wspomniane adresy URI), w taki sposób, aby użyte słowa jasno wyrażały przeznaczenie.

Tak więc w momencie, gdy chcesz pobrać, np. artykuł o ID 1234, pierwszą linię zapytania skonstruujesz tak jak poniżej: **GET devs-mentoring.pl/articles/1234**

**HTTP Message Body**

I na koniec wypadałoby wspomnieć o polu opcjonalnym - ciele zapytania HTTP. Możemy przekazywać w nim dodatkowe wartości, np. dane dla tworzonego rekordu w bazie danych. Przykładowo tak mogłoby wyglądać zapytanie POST z zakodowanymi danymi w postaci JSON-a.

| POST newswebsite.com/articles  Body: {  "article": {  "title": "Example",  "author": "Devs-Mentoring.pl",  "text": "It's working!"  } } |
| --- |

Tak jak widzisz - pole Body często jest wypełniane w momencie, gdy chcemy modyfikować lub dodawać nowe dane do choćby serwerowej bazy danych.

**HTTP Response**

Nie zapominajmy jeszcze o tym, iż oprócz żądań wysyłanych w stronę serwera, otrzymujemy również odpowiedzi, tzw. responses, które również bazują na protokole HTTP. Od strony struktury, nie ma dużych różnic między zapytaniem a odpowiedzią.

Jedyna substelna zmiana to brak HTTP Resource Path - serwer bowiem zastępuje to pole, tzw. Response Code, mówiącym o tym, czy powiodło się dostarczenie zapytania, albo czy serwer funkcjonuje prawidłowo.

I tak na przykład:

Jeżeli wszystko poszło zgodnie z planem, to serwer może odpowiedzieć standardowym kodem **200 (OK)** lub **201 (CREATED)** w przypadku pomyślnego utworzenia nowego rekordu w bazie danych. Jeżeli żądanie klienta zostało błędnie sformułowane (niewłaściwa struktura, szukanie zasobu, który nie istnieje) serwer może odpowiedzieć kodem **400 (BAD REQUEST)** lub **404 (NOT FOUND).**

Wykonując więc choćby takie zapytanie:

| GET devs-mentoring.pl/articles/1234 Accept: application/json |
| --- |

Możemy otrzymać:

| HTTP/1.1 200 (OK) Content-Type: application/json Body:  {  Title: 'Python TOP językiem roku 2020',  Date: 01/01/2021,  Author: Devs-Mentoring.pl } |
| --- |

Jeżeli chodzi o wszystkie rodzaje błędów oraz informacje, z jakim oznaczeniami są one identyfikowane, to odsyłam Cię do dokumentacji, z której sam korzystam na co dzień: [100 Continue - HTTP | MDN](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/100)

**Na zakończenie teorii...**

To wszystko jeżeli chodzi o teorię na temat procesu komunikacji Klient - Serwer. Nie bez znaczenia stworzyłem tak obszerny wstęp do zagadnienia związanego z tworzeniem własnego REST API we Flasku. Myślę, że dzięki takiemu zapoznaniu się z wyżej przedstawionymi informacjami, szybciej i łatwiej zrozumiesz logikę działania aplikacji, którą za chwilę będziemy budowali. Także czytaj dalej!

**Pomysł na REST API**

Zanim przejdziemy do implementacji, musimy zastanowić się nad tym, jak mniej więcej działać będzie Nasze REST API, z jakich metod i endpointów będziemy korzystali oraz jakie technologie wykorzystamy.

Jako że oprócz programowania, jestem zapalonym sportowcem, a szczególnie bardzo cenię sobie treningi crossfitowe, to stworzymy proste API umożliwiające wykonywanie podstawowych operacji CRUD na bazie danych.

Co więcej, wiem, iż będę chciał za jakiś czas stworzyć aplikację mobilną, która umożliwiać mi będzie zarządzanie moimi treningami przez przyjazny GUI, stąd też tak stworzone REST API idealnie nada się w przyszłości na integrację ze stworzoną apką!

Oczywiście będzie to mocno uproszczona aplikacja, głównie przez to, iż celem tego szkolenia, jest zapoznanie Cię z REST-ową ideą tworzenia API. Projektując bowiem mikroserwisy, które faktycznie miałyby być wykorzystywane przez funkcjonalną aplikację mobilną, konieczne by było zapewnienie choćby procesu uwierzytelnienia (tak aby nikt niepowołany nie był w stanie operować na Naszym API). Do tego tematu wrócę jeszcze na koniec - po przedstawieniu już docelowego kodu źródłowego apki.

Od strony założeń technologicznych - w aplikacji Flaskowej skorzystamy również z biblioteki **flask\_marshmallow**, która umożliwi łatwą serializację danych. Dane z obiektów będziemy bowiem zwracali do użytkownika w formie JSON-a. Informacje o treningach będziemy natomiast zapisywali w tabeli Trainings.

Jej kolumny to: id (unikalny zasób identyfikujący trening), date (pole reprezentujące datę odbytego treningu w formacie DD/MM/YY), duration (czas trwania treningu podany w minutach), note (notatka na umieszczenie opisu przebiegu treningu).

**Spojrzenie na aplikację pod kątem funkcjonalności**

Zanim przejdziemy do implementacji REST API, musimy zastanowić się, z jakich metod zapytań HTTP będziemy korzystali oraz przede wszystkim, jak będziemy formułowali dane endpointy.

Tak jak zostało więc wspomniane, Naszym celem będzie stworzenie aplikacji obsługującej zapytania CRUD. Czyli użytkownik będzie mógł:

* Dodawać pojedyncze rekordy do tabeli Trainings
* Wyświetlać wszystkie dodane treningi z tabeli.
* Wyszukiwać określony trening po ID
* Usuwać trening po określonym ID

Operować będziemy na informacjach w formacie JSON-a. Tak jak zostało wspomniane - aby móc łatwo serializować przetwarzane dane, wykorzystamy nieznaną dotąd bibliotekę Marshmallow. To właśnie od jej omówienia rozpoczniemy budowę projektu.

**Czym jest Marshmallow?**

Zasadniczo Flask-Marshmallow posłuży Nam do realizacji jednego celu - łatwiejszej konwersji obiektów SQLAlchemy do JSON-a. To wszystko po to, aby móc łatwo porozumiewać się z serwerem przez odpowiednio sformułowane zapytania.   
  
Jak wiesz zapewne, nie jest możliwe bezpośrednie przesyłanie obiektów w zapytaniu, które chcemy, np. utworzyć i przesłać z poziomu aplikacji do API, tylko musimy wykorzystać odpowiedni format zapisu danych (np. JSON, XML).   
I właśnie to jest główna przyczyna powstania omawianej biblioteki.

Zdejmuje ona bowiem z programisty konieczność ręcznej zamiany jednego typu na drugi. Dzięki niej możemy automatycznie konwertować, np. obiekt reprezentujący rekord z bazy na JSON-a i dawać możliwość jego pobrania przez aplikację mobilną.

Tak więc przykładowo, mając utworzony rekord reprezentujący pojedynczy trening:

Trainings(name=”Push Up”, date=01/01/2020, duration=120, note=”Everyting went well”)

Możemy zamienić go do postaci JSON-a (czyli serializować):

| { name: “Push Up”, date: “01/01/20”, duration: 120, note: ”Everything went well” } |
| --- |

I zwrócić jako rezultat danego endpointa (dzięki czemu, np. aplikacja mobilna może łatwo łączyć się z takim endpointem, pobierać tak skonwertowany text, odpowiednio go obrabiać i wyświetlać na GUI telefonu).

Identycznie sytuacja się ma w przypadku metody POST i chęci dodania nowego treningu do tabeli w bazie. Wystarczy bowiem spreparować z poziomu aplikacji mobilnej odpowiednią strukturę JSON-a z podanymi przez użytkownika danymi, a następnie wysłać ją na serwer. Tam wówczas odbędzie się deserializacja tych danych i zostanie utworzony obiekt.

Zacznijmy od serializacji. Aby zapewnić w programie właśnie taką funkcjonalność, należy utworzyć klasę, która będzie schematem umożliwiającym sprawne manipulowanie JSON-em.

Wystarczy więc utworzyć klasę dziedziczącą po Schema oraz dodać do niej dokładnie takie kolumny, jakie posiadać będzie utworzona przez Nas tabela reprezentująca treningi.

| class TrainingSchema(ma.Schema):  \_id = fields.fields.Integer()  name = fields.fields.Str()  date = fields.fields.DateTime(format='%Y-%m-%d')  duration = fields.fields.Integer()  note = fields.fields.Str() |
| --- |

Oczywiście, aby to było możliwe, należy zainstalować i zaimportować odpowiednie biblioteki:

| **Instalacja**  pip install flask\_marshmallow |
| --- |

**Import**

| from flask\_marshmallow import Marshmallow, fields |
| --- |

I utworzyć obiekt:

| ma = Marshmallow() |
| --- |

Gdybyś natomiast zapytał, jak konkretnie wygląda już procedura serializacji, to wystarczy zapisać to w ten sposób:

| training\_schema = TrainingSchema() new\_training = Trainings(name=name, date=date, duration=duration, note=note) training\_schema.jsonify(new\_training) |
| --- |

Oczywiście powyższe linie przedstawiają czysto koncepcyjne podejście do problemu. Implementacją w projekcie zajmiemy się za chwilę…

**Zacznijmy od podstaw…**

Pierwszym krokiem przy tworzeniu Flaskowego API, na pewno będzie skonfigurowanie obiektu aplikacji i utworzenie odpowiedniej bazy danych.

Struktura projektu:

---Trainings\_REST

¦ \_\_init\_\_.py - konfigurator projektu

¦ main\_api.py - plik z funkcjonalnością endpointów

¦ models.py - plik z bazodanowymi tabelami, schematami Marshmallow

¦ simulation.py - plik, w którym zasymulujemy zapytania do aplikacji

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

| from flask import Flask from flask\_sqlalchemy import SQLAlchemy from flask\_marshmallow import Marshmallow  db = SQLAlchemy() ma = Marshmallow()   def create\_app():  app = Flask(\_\_name\_\_)  app.config['SQLALCHEMY\_DATABASE\_URI'] = 'sqlite:///trainings.sqlite3'   db.init\_app(app)  ma.init\_app(app)  return app |
| --- |

Jeżeli przerobiłeś poprzednie szkolenia z Flaska, to myślę, że kod ten nie wymaga szczegółowego wytłumaczenia. Jedynie nową rzeczą może być obiekt ma, który powstał przez zapis:

ma = Marshmallow(),

a następnie jego zintegrowanie z app:

ma.init\_app(app)

Obiekt ma posłuży Nam do tworzenia schematów (o których wspomniałem już wcześniej przy omawianiu serializacji danych JSON-owych). Spokojnie, wszystko zobaczysz na przykładach w pliku models.py!

**Plik models.py**

W następnym kroku musimy przejść do utworzenia odpowiednich tabel wykorzystywanych przez bazę oraz klasy Schema, które będą niejako wzorem, z jakich kluczy ma składać się JSON po serializacji.

| from \_\_future\_\_ import annotations from . import db, ma from flask\_marshmallow import fields from datetime import datetime   class Trainings(db.Model):  \_id = db.Column(db.Integer, primary\_key=True)  name = db.Column(db.String(length=50), nullable=False)  date = db.Column(db.DateTime, default=datetime.utcnow, nullable=False)  duration = db.Column(db.Integer, nullable=False)  note = db.Column(db.String(length=5000), nullable=True)   def \_\_init\_\_(self, name: str, date: datetime, duration: int, note: str):  self.name = name  self.date = date  self.duration = duration  self.note = note   def update(self, modified\_training: Trainings) -> None:  self.name = modified\_training.name  self.date = modified\_training.date  self.duration = modified\_training.duration  self.note = modified\_training.note   @staticmethod  def create\_from\_json(json\_body: dict) -> Trainings:  if 'date' not in json\_body:  date = datetime.utcnow()  else:  date = datetime.strptime(json\_body['date'], '%d/%m/%y')   return Trainings(  name=json\_body['name'], date=date,  duration=json\_body['duration'], note=json\_body['note']  ) |
| --- |

Tabela Trainings będzie składała się z 5 kolumn: name (typ str), date (typ datetime w formacie DD/MM/YY), duration (typ int), note (typ str).

Dodatkowo do tabeli dodaliśmy metodę - update oraz statyczną funkcjonalność - create\_from\_json.

Pierwsza służyć nam będzie do modyfikowania zawartości istniejącego już rekordu przez nadpisanie go nowym, natomiast create\_from\_json przyda się, gdy będziemy chcieli utworzyć rekord ze znanego Nam formatu JSON-a (deserializować dane). Zwróć też uwagę na to, co się dzieje, gdy użytkownik nie prześle w swoim zapytaniu pola date. Wówczas jest ono domyślnie ustawiane na aktualną datę.

Na zakończenie jeszcze - mała dygresja. Wiesz, dlaczego pierwsza linia rozpatrywanego pliku to: from \_\_future\_\_ import annotations?

Służy Nam ona temu, aby móc statycznie zaadnotować typ zwracany przez metodę create\_from\_json. Wiele osób o tym zapomina, ale gdybyśmy nie dodali tejże biblioteki, program zwróciłby błąd, mówiący o tym, iż typ klasy Trainings jest nieznany (a to dlatego, iż nie jesteśmy w stanie typować klasę, w której dana metoda się znajduje).

**Tworzenie schematu**

Wiedząc już, jak wygląda struktura tabeli Trainings, możemy utworzyć odpowiadający jej schemat. Oczywiście operujemy na tym samym pliku, co do tej pory, czyli models.py

I to właśnie w tej części programu wykorzystamy utworzony w pliku \_\_init\_\_.py obiekt ma (uprzednio importując go zapisem from . import db, ma).

| class TrainingSchema(ma.Schema):  \_id = fields.fields.Integer()  name = fields.fields.Str()  date = fields.fields.DateTime(format='%d-%m-%y')  duration = fields.fields.Integer()  note = fields.fields.Str() |
| --- |

Jeżeli przeoczyłeś gdzieś we wcześniejszych sekcjach, do czego wykorzystamy tę klasę, to posłuży Nam ona do łatwej serializacji danych. Mając bowiem utworzony obiekt TrainingSchema, możemy bez problemu “przekonwertować” obiekt Trainings na format JSON-a. W schemacie jedynie wystarczy umieścić odpowiednie nazwy pól z tabeli, które chcemy, aby znalazły się w docelowym formacie.

**Plik main\_api.py**

Omawiany plik w zasadzie jest gwoździem programu. To tutaj będziemy realizowali założenia API.

Pozwól, że wkleję już na początku, jak będzie wyglądała finalna wersja tego pliku, a kolejne funkcjonalności będziemy omawiali progresywnie.

| from flask import Blueprint, request, jsonify from . import db from .models import Trainings, TrainingSchema  add\_training\_blueprint = Blueprint('add\_training', \_\_name\_\_) get\_trainings\_blueprint = Blueprint('get\_trainings', \_\_name\_\_) get\_training\_blueprint = Blueprint('get\_training', \_\_name\_\_) update\_training\_blueprint = Blueprint('update\_training', \_\_name\_\_) delete\_training\_blueprint = Blueprint('delete\_training', \_\_name\_\_)  training\_schema = TrainingSchema() trainings\_schema = TrainingSchema(many=True)   def add\_to\_db(new\_training: Trainings) -> None:  db.session.add(new\_training)  db.session.commit()   def delete\_from\_db(training\_to\_delete: Trainings) -> None:  db.session.delete(training\_to\_delete)  db.session.commit()   @add\_training\_blueprint.route('/training', methods=['POST']) def add\_training() -> str:  body = request.json   new\_training = Trainings.create\_from\_json(json\_body=body)   add\_to\_db(new\_training)   return training\_schema.jsonify(new\_training)   @get\_trainings\_blueprint.route('/trainings', methods=['GET']) def get\_trainings() -> str:  all\_trainings = Trainings.query.all()  return trainings\_schema.jsonify(all\_trainings)   @get\_training\_blueprint.route('/training/<int:id>', methods=['GET']) def get\_training\_by\_id(id: int) -> str:  found\_training = Trainings.query.get(id)  return training\_schema.jsonify(found\_training)   @update\_training\_blueprint.route('/training/<int:id>', methods=['PUT']) def update\_training(id: int) -> str:  found\_training = Trainings.query.get(id)   body = request.json  found\_training.update(Trainings.create\_from\_json(json\_body=body))   db.session.commit()   return training\_schema.jsonify(found\_training)   @delete\_training\_blueprint.route('/training/<int:id>', methods=['DELETE']) def delete\_training(id: int) -> str:  training\_to\_delete = Trainings.query.get(id)  delete\_from\_db(training\_to\_delete)   return training\_schema.jsonify(training\_to\_delete) |
| --- |

Stworzymy w sumie 5 głównych metod (add\_training, get\_trainings, get\_training\_by\_id, update\_training, delete\_training), które determinować będą funkcjonalność API.

Zacznijmy od podstawowej - dodawania nowego treningu do tabeli w bazie.

Metoda add\_training będzie wykorzystywać metodę POST. Tak więc, aby móc dodać nowy trening, użytkownik będzie musiał wysłać zapytanie HTTP pod endpoint /training z wykorzystaniem metody POST. Tak skonstruowany request będzie ponadto zawierał w swoim ciele json\_body z kluczami name, date, duration, note.

Odbieranie JSON-owego ciała zawartego w requestcie HTTP, możliwe jest dzięki wykorzystaniu modułu request zaimportowanego z biblioteki flask:

| from flask import request |
| --- |

oraz dzięki poniższemy zapisowi umieszczonego bezpośrednio już w metodzie add\_training:

| body = request.json |
| --- |

Dalsze kroki umożliwiające dodanie rekordu do tabeli, to wykorzystanie statycznej metody Trainings.create\_from\_json i… uwaga, uwaga zserializowanie utworzonego obiektu do postaci JSON-a.

A to właśnie dzięki pięknu biblioteki Marshmallow i utworzonemu obiektowi training\_schema.

| training\_schema = TrainingSchema() trainings\_schema = TrainingSchema(many=True) |
| --- |

Jak już jesteśmy przy tych liniach kodu - gdybyś się zastanawiał, dlaczego utworzyliśmy dwa obiekty training\_schema oraz trainings\_schema, to wiedz, że ten drugi umożliwi nam serializowanie wielu obiektów Trainings naraz. Przyda to Nam się w momencie, gdy będziemy chcieli odczytać listę wszystkich treningów zapisanych w tabeli…

Powtórzę się, ale zauważ, jak prosto możemy serializować dane przy użyciu Marshmallow. Wystarczy tak naprawdę jedna linia kodu, aby łatwo tworzyć format JSON-a.

| training\_schema.jsonify(training\_to\_delete) |
| --- |

Gdybyśmy nie mieli możliwości tak łatwej konwersji danych, musielibyśmy mocno się napracować, aby utworzyć właściwego JSON-a. Konieczne bowiem by było utworzenie słownika, ręczne dodanie do niego odpowiednich kluczy i wyekstrahowanie wartości pól z obiektu. Stanowczy overkill!

Mając już tak utworzony plik main\_api.py, ostatnim krokiem będzie zarejestrowanie Blueprintów w aplikacji. Musimy zatem wrócić do \_\_init\_\_.py, zaimportować i zarejestrować utworzone Blueprinty, tj:

| add\_training\_blueprint = Blueprint('add\_training', \_\_name\_\_) get\_trainings\_blueprint = Blueprint('get\_trainings', \_\_name\_\_) get\_training\_blueprint = Blueprint('get\_training', \_\_name\_\_) update\_training\_blueprint = Blueprint('update\_training', \_\_name\_\_) delete\_training\_blueprint = Blueprint('delete\_training', \_\_name\_\_) |
| --- |

Tak więc po modyfikacji, funkcja create\_app z \_\_init\_\_.py będzie miała następującą postać:

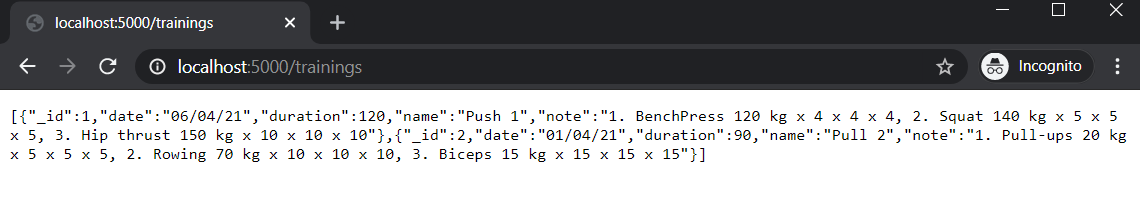
| def create\_app():  app = Flask(\_\_name\_\_)  app.config['SQLALCHEMY\_DATABASE\_URI'] = 'sqlite:///trainings.sqlite3'   db.init\_app(app)  ma.init\_app(app)   from .main\_api import add\_training\_blueprint, get\_trainings\_blueprint, \  get\_training\_blueprint, update\_training\_blueprint, delete\_training\_blueprint   app.register\_blueprint(add\_training\_blueprint)  app.register\_blueprint(get\_trainings\_blueprint)  app.register\_blueprint(get\_training\_blueprint)  app.register\_blueprint(update\_training\_blueprint)  app.register\_blueprint(delete\_training\_blueprint)   return app |
| --- |

**Funkcjonalność API**

Voilà! Kod gotowy. Przyszło jeszcze sprawdzenie funkcjonalności aplikacji. W tym celu zasymulujemy interakcję użytkownika z API przez plik simulation.py. Wykorzystamy bibliotekę requests, która umożliwia łatwe wysyłanie HTTP Requestów na określone adresy URI wraz z wybranymi metodami (np. GET, POST, PUT, DELETE).

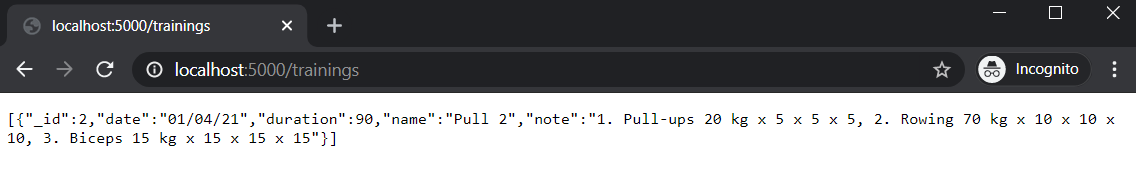
1. Dodanie do bazy 2 nowych treningów.

| import requests  requests.post('http://localhost:5000/training',  json={  "name": "Push 1",  "duration": 120,  "note": "1. BenchPress 120 kg x 4 x 4 x 4, "  "2. Squat 140 kg x 5 x 5 x 5, "  "3. Hip thrust 150 kg x 10 x 10 x 10"  })  requests.post('http://localhost:5000/training',  json={  "name": "Pull 2",  "date": "01/04/21",  "duration": 90,  "note": "1. Pull-ups 20 kg x 5 x 5 x 5, "  "2. Rowing 70 kg x 10 x 10 x 10, "  "3. Biceps 15 kg x 15 x 15 x 15"  }) |
| --- |

.

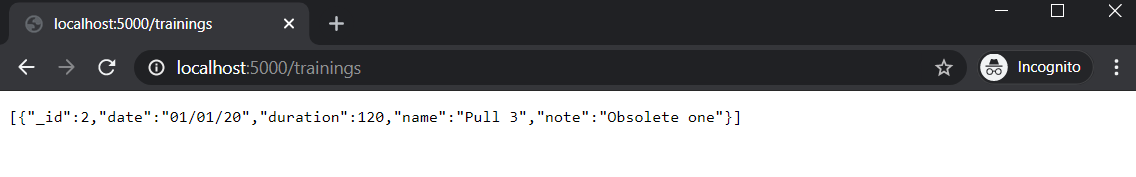
1. Usunięcie z bazy treningu o ID 1

| import requests  requests.delete('http://localhost:5000/training/1') |
| --- |

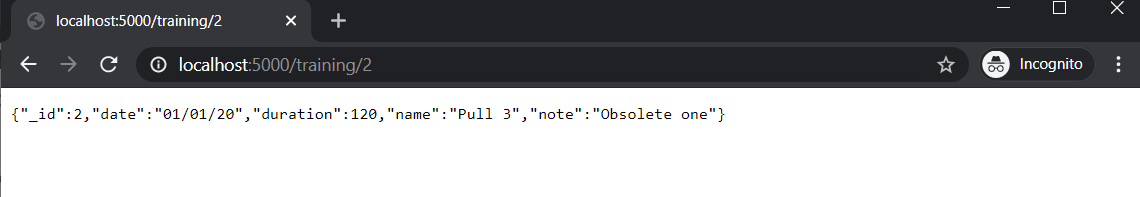


1. Modyfikowanie istniejącego rekordu

| import requests  requests.put('http://localhost:5000/training/2',  json={  "name": "Pull 3",  "date": "01/01/20",  "duration": 120,  "note": "Obsolete one"  }) |
| --- |



1. Wyświetlanie treningu o określonym ID.



**Jak zabezpieczyć API?**

Na koniec jeszcze - zastanowiłeś się może, w jaki sposób zabezpieczyć swoją aplikację, aby nie miał do niej dostępu żaden nieautoryzowany użytkownik? Gdybyśmy bowiem w tym momencie zdeployowali tak stworzoną Flaskową aplikację gdzieś w sieci, to dowolna osoba, mogłaby wysyłać POST czy DELETE request i manipulować danymi w bazie.

A przecież tę funkcjonalność chcemy jedynie zapewnić zarejestrowanym użytkownikom i tylko w odniesieniu do własnoręcznie utworzonych rekordów (nie chcemy, aby Jan mógł modyfikować treningów Stanisława; Jan może jedynie zmieniać własne treningi).

Po pierwsze, aby zrealizować powyższe założenia, na pewno musielibyśmy zmodyfikować nieco tabelę Training. Konieczne bowiem by było dodanie do niej kolumny informującej o tym, użytkownik o jakim ID dodał dany trening. Jeżeli więc, Stanisław zarejestrował się w Naszej aplikacji, to został mu przydzielony ID, np. o wartości 5. Dzięki temu, dodając kolejne treningi, pozostawia on niejako swój “odcisk” (poprzez dodanie wartości 5 do odpowiedniej kolumny w tabeli) i dzięki temu jesteśmy w stanie identyfikować treningi z danym userem. Tak więc, jeżeli Stanisław będzie chciał zmienić trening o ID 1, to w pierwszej kolejności odnajdziemy wszystkie treningi należące do użytkownika o ID 5, a dopiero później sprawdzimy, czy jest na nich szukana pozycja.

Jest jeszcze tylko jedna rzecz do rozpatrzenia. A dokładnie problem związany z tym, w jaki sposób nadawać prawa użytkownikom? Otóż z pomocą przychodzi, tzw. JWT (JSON Web Token), czyli przyjazny sposób na autoryzowanie dostępu do pewnych zasobów. W celu łatwej integracji aplikacji z tą technologią, wystarczy tak naprawdę zainstalować bibliotekę pyjwt i odpowiednio skonfigurować tworzony projekt.

Od tego momentu, moglibyśmy zapewnić realizację poniższych kroków:

* Klient dostarcza swój adres mailowy i hasło, które jest wysyłana na serwer
* Serwer weryfikuje email i hasło (porównuje z danymi zapisanymi w bazie) i zwraca wygenerowany token
* Klient zachowuje token i wysyła go wraz z kolejnymi zapytaniami do API
* Serwer waliduje przekazany token i odpowiednio autoryzuje użytkownika

Dzięki takiemu zabiegowi, korzystanie z aplikacji staje się w pełni bezpieczne. Żaden niepowołany użytkownik bez odpowiednio wygenerowanego tokenu, nie będzie w stanie zmienić zasobów przechowywanych w bazie po stronie serwera.