

## Flask REST API

1. Zainstaluj odpowiednie moduły z poziomu File->Settings:

flask

connexion

flask-sqlalchemy

flask-marshmallow

flask-Bootstrap

marshmallow-sqlalchemy

marshmallow

swagger-ui-bundle

2. W folderze Day13 załóż nowy folder o nazwie server, utwórz w nim plik o nazwie app.py
3. Uzupełnij kod pliku app.py:

```
import os
import connexion
from flask_sqlalchemy import SQLAlchemy
from flask_marshmallow import Marshmallow
from flask_bootstrap import Bootstrap

main_directory = os.path.abspath(os.path.dirname(__file__))

app_connexion = connexion.App(__name__, specification_dir=main_directory)

app = app_connexion.app
Bootstrap(app)

# database path
sqlite_path = "sqlite:/// " + os.path.join(main_directory, "music_store.db")

# Configure the SQLAlchemy part of the app instance
app.config["SQLALCHEMY_ECHO"] = True
app.config["SQLALCHEMY_DATABASE_URI"] = sqlite_path
app.config["SQLALCHEMY_TRACK_MODIFICATIONS"] = False

# sql alchemy orm to communicate with sqlite db
db = SQLAlchemy(app)

# marshmallow init
ma = Marshmallow(app)
```

Plik app.py będzie służył jako plik z główną konfiguracją aplikacji, jak aplikacja serwera – flask/connection oraz dostęp do bazy danych

4. Utwórz plik index.py w folderze server – plik ten będzie służył jako główny skrypt uruchamiający naszą aplikację. Poniżej kod, który należy umieścić w pliku:

```
from flask import render_template
from Day13.server.app import app_connexion
from Day13.server.models import Album, AlbumSchema

# Get the application instance
connex_app = app_connexion

# create a URL route in our application for "/"
@connex_app.route("/")
def home():
    """
    This function just responds to the browser URL
    Localhost:8080/
    :return: the rendered template
    """
    return "<h1>Hello, iSA Python World!</h1>"

if __name__ == "__main__":
    connex_app.run(port=8080, debug=True)
```

5. Uruchom plik index.py, otwórz przeglądarkę i otwórz adres <http://localhost:8080/>
6. Zmień kod w pliku index.py, aby w przeglądarce wyświetlił się napis: „That is REST API example” – zapisz plik index.py po zmianach, przejrzyj logi konsoli i odśwież stronę przeglądarki – czy tekst się zmienił?
7. Zatrzymaj uruchomiony skrypt. W folderze server dodaj plik swagger.yml, zawartość pliku:

```
swagger: "2.0"
info:
  description: This is the swagger file with our REST API spec
  version: "1.0.0"
  title: Swagger Rest Documentation
consumes:
  - application/json
produces:
  - application/json

basePath: /api

# Paths supported by the server application
paths:
  /album:
    get:
      operationId: album.read_all
      tags:
        - Albums
      summary: Gets all the albums, sorted by artist
      description: Gets all the albums, sorted by artist
      responses:
        200:
          description: Successfully returned albyms
```

```

schema:
  type: array
  items:
    properties:
      album_id:
        type: string
        description: Id of the albym
      artist:
        type: string
        description: Title of the album
      title:
        type: string
        description: Title of the album
      image_url:
        type: string
        description: Image URL of the album
      last_edited_at:
        type: string
        description: Creation/Update timestamp of the album

```

8. W folderze server stwórz plik album.py:

```

import json
from flask import jsonify

albums = [
    {
        "album_id": 1,
        "artist": "Michael Jackson",
        "title": "Thriller",
        "image_url":
"https://upload.wikimedia.org/wikipedia/en/thumb/5/55/Michael_Jackson_-_Thriller.png/220px-Michael_Jackson_-_Thriller.png"
    },
    {
        "album_id": 2,
        "artist": "Metallica",
        "title": "Kill'em all",
        "image_url":
"https://upload.wikimedia.org/wikipedia/en/thumb/5/55/Michael_Jackson_-_Thriller.png/220px-Michael_Jackson_-_Thriller.png"
    }
]

def read_all():
    """
    This function responds to a request for /api/album
    with the lists of available albums
    :return:      json string of list of albums
    """
    # Create the list of albums from our data
    return jsonify(albums)

```

9. W kodzie pliku index.py za linią 6 dodaj konfigurację servera z wcześniej stworzonego pliku YAML:

```
# Read the swagger.yml file to configure the endpoints
connex_app.add_api("swagger.yml")
```

10. Uruchom plik index.py, a następnie przejdź w przeglądarce do adresu

<http://localhost:8080/api/ui>

Rozwiń dokumentację do sekcji Albums i wypróbuj wywołanie metody GET /album

11. Uzupełnij konfigurację pliku swagger.yml o pozostałe metody do sekcji Albums:

Plik powinien wyglądać następująco:

```
swagger: "2.0"
info:
  description: This is the swagger file with our REST API spec
  version: "1.0.0"
  title: Swagger Rest Documentation
consumes:
  - application/json
produces:
  - application/json

basePath: /api

# Paths supported by the server application
paths:
  /album:
    get:
      operationId: album.read_all
      tags:
        - Albums
      summary: Gets all the albums, sorted by artist
      description: Gets all the albums, sorted by artist
      responses:
        200:
          description: Successfully returned albysms
          schema:
            type: array
            items:
              properties:
                album_id:
                  type: string
                  description: Id of the albym
                artist:
                  type: string
                  description: Title of the album
                title:
                  type: string
                  description: Title of the album
                image_url:
                  type: string
                  description: Image URL of the album
                last_edited_at:
                  type: string
                  description: Creation/Update timestamp of the album

    post:
      operationId: album.create
```

```
tags:
  - Albums
summary: Create an album
description: Create an new album
parameters:
  - name: album
    in: body
    description: Album to create
    required: True
    schema:
      type: object
      properties:
        artist:
          type: string
          description: Artist of album to create
        title:
          type: string
          description: Title of album to create
        image_url:
          type: string
          description: Image URL of album to create
responses:
  201:
    description: Successfully created album
    schema:
      properties:
        album_id:
          type: string
          description: Id of the album
        artist:
          type: string
          description: Artist of album to create
        title:
          type: string
          description: Title of album to create
        image_url:
          type: string
          description: Image URL of album to create
        timestamp:
          type: string
          description: Creation/Update timestamp of the album record
```

/album/{album\_id}:

```
get:
  operationId: album.read_one
  tags:
    - Albums
  summary: Read one album data
  description: Get one album
  parameters:
    - name: album_id
      in: path
      description: Id of the album
      type: integer
      required: True
  responses:
    200:
      description: Successfully read album
      schema:
```

```

    type: object
    properties:
      album_id:
        type: string
        description: Id of the album
      artist:
        type: string
        description: Artist of album
      title:
        type: string
        description: Title of album
      image_url:
        type: string
        description: Image URL of album
      timestamp:
        type: string
        description: Creation/Update timestamp of the album record

put:
  operationId: album.update
  tags:
    - Albums
  summary: Update an album
  description: Update an album
  parameters:
    - name: album_id
      in: path
      description: Id of the album to update
      type: integer
      required: True
    - name: album
      in: body
      schema:
        type: object
        properties:
          artist:
            type: string
            description: Artist of album
          title:
            type: string
            description: Title of album
          image_url:
            type: string
            description: Image URL of album
  responses:
    200:
      description: Successfully updated album
      schema:
        properties:
          album_id:
            type: string
            description: Id of the album
          artist:
            type: string
            description: Artist of album
          title:
            type: string
            description: Title of album
          image_url:

```

```

        type: string
        description: Image URL of album
    timestamp:
        type: string
        description: Creation/Update timestamp of the album record

delete:
    operationId: album.delete
    tags:
        - Albums
    summary: Delete an album from the db
    description: Delete an album
    parameters:
        - name: album_id
          in: path
          type: integer
          description: Id of the album to delete
          required: true
    responses:
        200:
            description: Successfully deleted an album!

```

12. Po uzupełnieniu sekcji konfiguracji, należy dodać właściwy kod do obsługi albumów – najpierw jednak skorzystamy z biblioteki SQLAlchemy i stworzymy model naszego albumu, przystosowany do przechowywania w bazie danych. W folderze server stwórz plik models.py:

```

from Day13.server.app import db, ma
import datetime

class Album(db.Model):
    __tablename__ = "album"
    album_id = db.Column(db.Integer, primary_key=True)
    artist = db.Column(db.String(64))
    title = db.Column(db.String(128))
    image_url = db.Column(db.String(255))
    last_edited_at = db.Column(
        db.DateTime, default=datetime.datetime.utcnow,
        onupdate=datetime.datetime.utcnow
    )

class AlbumSchema(ma.Schema):
    class Meta:
        # model = Album,
        fields = ("album_id", "artist", "title", "image_url", "last_edited_at")
        sqla_session = db.session

```

13. Następnie zmień kod pliku album.py, tak aby korzystać tym razem z danych dostępnych w bazie danych:

```

from Day13.server.app import db
from Day13.server.models import Album, AlbumSchema
from flask import make_response, abort

```

```

def read_all():
    """
    This function responds to a request for /api/album
    with the lists of available albums
    :return:      json string of list of albums
    """
    # Create the list of people from our data
    albums = Album.query.order_by(Album.artist).all()

    # Serialize the data for the response
    album_schema = AlbumSchema(many=True)
    dump = album_schema.dump(albums)
    data = dump.data
    return data

def read_one(album_id):
    """
    This function responds to a request for /api/album/{album_id}
    with one matching album from albums
    :param album_id:  Id of album to find
    :return:          album with passed id (if founded)
    """
    # Get the album from db
    album = Album.query.filter(Album.album_id == album_id).one_or_none()

    if album is not None:

        # Serialize the data for the response
        album_schema = AlbumSchema()
        data = album_schema.dump(album).data
        return data

    # Otherwise, nope, didn't find that album
    else:
        abort(
            404,
            "Album not found for Id: {album_id}".format(album_id=album_id),
        )

def create(album):
    """
    This function creates a new album
    based on the passed data
    :param album:  album to create in people structure
    :return:       201 on success, 406 on album exists
    """
    artist = album.get("artist")
    title = album.get("title")
    image_url = album.get("image_url")

    existing_album = (
        Album.query.filter(Album.artist == artist)
        .filter(Album.title == title)
        .one_or_none()
    )

    if existing_album is None:

```



```

        # Create a album instance using the schema
        schema = AlbumSchema()
        new_album = Album(artist=album["artist"], title=album["title"],
image_url=album["image_url"])

        # Add the album to the database
        db.session.add(new_album)
        db.session.commit()

        # Serialize and return the newly created album in the response
        data = schema.dump(new_album).data

        return data, 201

# Album exists already
else:
    abort(
        409,
        f"Album {artist}: {title} exists already"
    )

def update(album_id, album):
    """
    This function updates an existing album
    Throws an error if an album
    already exists in the database.
    :param album_id: Id of the album to update
    :param album: new data of album
    :return: updated album data
    """

    album_to_update = Album.query.filter(
        Album.album_id == album_id
    ).one_or_none()

    # Check if we're trying to create duplicates in db
    artist = album.get("artist")
    title = album.get("title")
    image_url = album.get("image_url")

    existing_album = (
        Album.query.filter(Album.artist == artist)
        .filter(Album.title == title)
        .one_or_none()
    )

    if album_to_update is None:
        abort(
            404,
            "Album not found for Id: {album_id}".format(album_id=album_id),
        )

    # Would our update create a duplicate?
    elif (
        existing_album is not None and existing_album.album_id != album_id
    ):
        abort(

```

```

        409,
        f"Album {artist}: {title} exists already"
    )

    # Otherwise go ahead and update!
    else:

        # turn the passed data into a db object
        schema = AlbumSchema()
        update_album_data = Album(artist=artist, title=title, image_url=image_url)

        # Set the id to the album we want to update
        update_album_data.album_id = album_to_update.album_id

        # merge the new object into the old and commit it to the db
        db.session.merge(update_album_data)
        db.session.commit()

        # return updated album
        data = schema.dump(update_album_data).data

        return data, 200

def delete(album_id):
    """
    This function deletes an album
    :param album_id: Id of the album to delete
    :return: 200 on successful delete, 404 if not found
    """
    # Get the album from db
    album = Album.query.filter(Album.album_id == album_id).one_or_none()

    if album is not None:
        db.session.delete(album)
        db.session.commit()
        return make_response(
            f"Album with id {album_id} deleted", 200
        )

    # Didn't find that album
    else:
        abort(
            404,
            f"Album with id {album_id} not found"
        )

```

14. Aby aplikacja działała prawidłowo, należy jeszcze zainicjować naszą bazę danych SQLite – stwórz plik `init_database.py`:

```

import os
from Day13.server.app import db
from Day13.server.models import Album

# Data to initialize database with
Albums = [

```

```

        {"artist": "Michael Jackson", "title": "Thriller", "image_url":
"https://upload.wikimedia.org/wikipedia/en/thumb/5/55/Michael_Jackson_-_
_Thriller.png/220px-Michael_Jackson_-_Thriller.png"},
        {"artist": "Linkin Park", "title": " Meteora", "image_url":
"https://image.ceneostatic.pl/data/products/48665992/i-linkin-park-meteora-album-
cover-sticker.jpg"}],

# Delete database file if it exists currently
if os.path.exists("music_store.db"):
    os.remove("music_store.db")

# Create the database
db.create_all()

# populate the database
for album in Albums:
    a = Album(artist=album.get("artist"), title=album.get("title"),
image_url=album.get("image_url"))
    db.session.add(a)

db.session.commit()

```

15. Uruchom plik index.py, przejdź do strony <http://localhost:8080/api/ui/#/> i przetestuj działanie API – spróbuj dodać nowe albumy, zaktualizować istniejące:

POST

/album

Create an album

Implementation Notes

Create an new album

Response Class (Status 201)

Successfully created album

Model

Example Value

```
{
  "album_id": "string",
  "artist": "string",
  "image_url": "string",
  "timestamp": "string",
  "title": "string"
}
```

Response Content Type

application/json

Parameters

Parameter	Value	Description	Parameter Type	Data Type
album	<pre>{   "artist": "Some Artist",   "image_url": "Some image URL",   "title": "Some title" }</pre>	Album to create	body	<div>Model</div> <div>Example Value</div> <pre>{   "artist": "string",   "image_url": "string",   "title": "string" }</pre>

Parameter content type:

application/json

Try it out!

Hide Response

16. Stwórz folder templates w folderze server i umieść w nowo utworzonym folderze templates plik base.html:

```
{# ``base.html`` is the template all our other templates derive from. While
Flask-Bootstrap ships with its own base, it is good form to create a custom
one for our app, as it allows customizing some aspects.
```

Deriving from bootstrap/base.html gives us a basic page scaffolding.

You can find additional information about template inheritance at

<http://jinja.pocoo.org/docs/templates/#template-inheritance>

```
{#}
{%- extends "bootstrap/base.html" %}

{# We also set a default title, usually because we might forget to set one.
   In our sample app, we will most likely just opt not to change it #}
{% block title %}Music Store{% endblock %}

{# While we are at it, we also enable fixes for legacy browsers. First we
   import the necessary macros: #}
{% import "bootstrap/fixes.html" as fixes %}

{# Then, inside the head block, we apply these. To not replace the header,
   ``super()`` is used: #}
{% block head %}
{{super()}}

{#- Docs: http://pythonhosted.org/Flask-Bootstrap/macros.html#fixes
   The sample application already contains the required static files. #}
{{fixes.ie8()}}
{%- endblock %}

{# Adding our own CSS files is also done here. Check the documentation at
   http://pythonhosted.org/Flask-Bootstrap/basic-usage.html#available-blocks
   for an overview. #}
{% block styles -%}
    {{super()}} {# do not forget to call super or Bootstrap's own stylesheets
                  will disappear! #}
    <link rel="stylesheet" type="text/css"
        href="{{url_for('static', filename='main.css')}}">
{% endblock %}
```

17. Do folderu templates dodaj jeszcze plik index.html:

```
{# This simple template derives from ``base.html``. See ``base.html`` for
   more information about template inheritance. #}
{%- extends "base.html" %}

{# Loads some of the macros included with Flask-Bootstrap. We are using the
   utils module here to automatically render Flask's flashed messages in a
   bootstrap friendly manner #}
{% import "bootstrap/utils.html" as utils %}
```



```

        

        <div class="caption">
            <h3>Thriller</h3>
            <p>
                This is album of Michael Jackson - Thriller
            </p>
        </div>
    </div>
</div>

<ul>
    {% for album in albums %}
    <li>{{ album.artist }} {{ album.title }}</li>
    {% endfor %}
</ul>

</div>

</div>
{%- endblock %}

```

18. Zaktualizuj plik index.py:

```

from flask import render_template
from Day13.server.app import app_connexion
from Day13.server.models import Album, AlbumSchema

# Get the application instance
connex_app = app_connexion

# Read the swagger.yml file to configure the endpoints
connex_app.add_api("swagger.yml")

# create a URL route in our application for "/"
@connex_app.route("/")
def home():
    """
    This function just responds to the browser URL
    localhost:5000/
    :return: the rendered template "index.html"
    """
    albums = Album.query.order_by(Album.artist).all()

    # Serialize the data for the response
    # album_schema = AlbumSchema(many=True)
    # dump = album_schema.dump(albums)
    # data = dump.data

    return render_template("index.html", albums=albums)

if __name__ == "__main__":
    connex_app.run(port=8080, debug=True)

```

19. Uruchom plik index.py i przejdź do strony <http://localhost:8080/>
20. Zmień kod w pliku index.htm, aby albumy nie były wyświetlane jako prosta lista, ale jako „karty” – użyj istniejącego kodu z elementem

```
<div class="col-md-4">  
  <div class="thumbnail">
```