



Zarządzanie Danymi

Flask REST API (część 1) – przykłady aplikacji

dr inż. Łukasz Piątek

Katedra Sztucznej Inteligencji

Wyższa Szkoła Informatyki i Zarządzania z siedzibą w Rzeszowie

Agenda



- Implementacja przykładowych aplikacji typu *Flask REST API*:
 - prosta (podstawowa) aplikacja *Flask REST API*,
 - aplikacja umożliwiająca obsługę *CRUD* (ang., *Create*, *Read*, *Update* oraz/lub *Delete*),
 - aplikacja z autoryzacją dostępu do danych (z *Flask-JWT*)
- Wszystkie w/w aplikacje zaimplementowano w:
 - środowisku *Microsoft VS Studio Code*,
 - w połączeniu z wykorzystaniem programu *Postman*.

Podstawowa aplikacja *Flask REST API* (1/3)



■ *Microsoft VS Studio Code*:

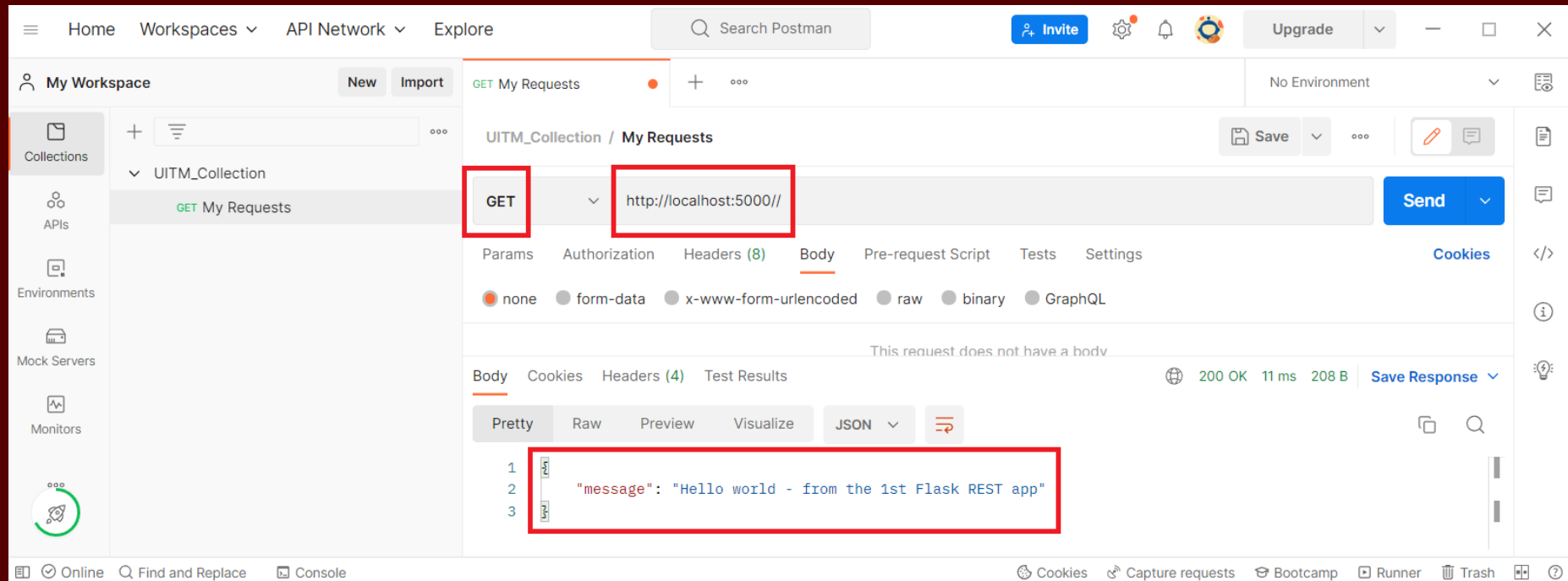
```
01_simple_api.py X
01_simple_api.py
1  from flask import Flask
2  from flask_restful import Resource, Api
3
4  app = Flask(__name__)
5  api = Api(app)
6
7  class HelloWorld(Resource):
8      def get(self):
9          return {'message': 'Hello world - from the 1st Flask REST app'}
10
11  api.add_resource(HelloWorld, '/')
12
13  if __name__ == '__main__':
14      app.run(debug=True)
15
```

Rys.1.A. Implementacja prostej aplikacji *Flask REST API* w środowisku *Microsoft VS Studio Code*

Podstawowa aplikacja *Flask REST API* (1/3)



■ *Postman*:



Rys.1.B. Wysłanie zapytania (oraz odpowiedź) z aplikacji klienckiej *Postman*

Aplikacja *Flask REST CRUD API* (2/3)



■ *Microsoft VS Studio Code* (część 1):

```
02_crud_api.py X
02_crud_api.py
1  from flask import Flask, request
2  from flask_restful import Resource, Api
3
4  app = Flask(__name__)
5  api = Api(app)
6
7  # We should call to the database!
8  # However, right now its just a list of dictionaries
9  # students = [{'name':'Andrusiv'},{'name':'Cyran'},.....]
10 # Keep in mind, its in memory, it clears with every restart!
11 students = []
```

Rys.2.A.1. Implementacja aplikacji *Flask REST CRUD API* w środowisku *Microsoft VS Studio Code* (1)

Aplikacja *Flask REST CRUD API* (2/3)



■ *Microsoft VS Studio Code* (część 2):

```
13 class StudentNames(Resource):
14     def get(self, name):
15         print(students)
16
17         # Cycle through list for students
18         for stud in students:
19             if stud['name'] == name:
20                 return stud
21
22         # If it's request of a student not yet in the students list
23         return {'name': None}, 404
24
25     def post(self, name):
26         # Add the dictionary to list
27         stud = {'name': name}
28         students.append(stud)
29         # Then return it back
30         print(students)
31         return stud
32
33     def delete(self, name):
34
35         # Cycle through list for students
36         for ind, stud in enumerate(students):
37             if stud['name'] == name:
38                 # don't really need to save this
39                 delted_stud = students.pop(ind)
40                 return {'note': 'delete successful'}
```

Rys.2.A.2. Implementacja aplikacji *Flask REST CRUD API* w środowisku *Microsoft VS Studio Code* (2)

Aplikacja *Flask REST CRUD API* (2/3)



■ *Microsoft VS Studio Code* (część 3):

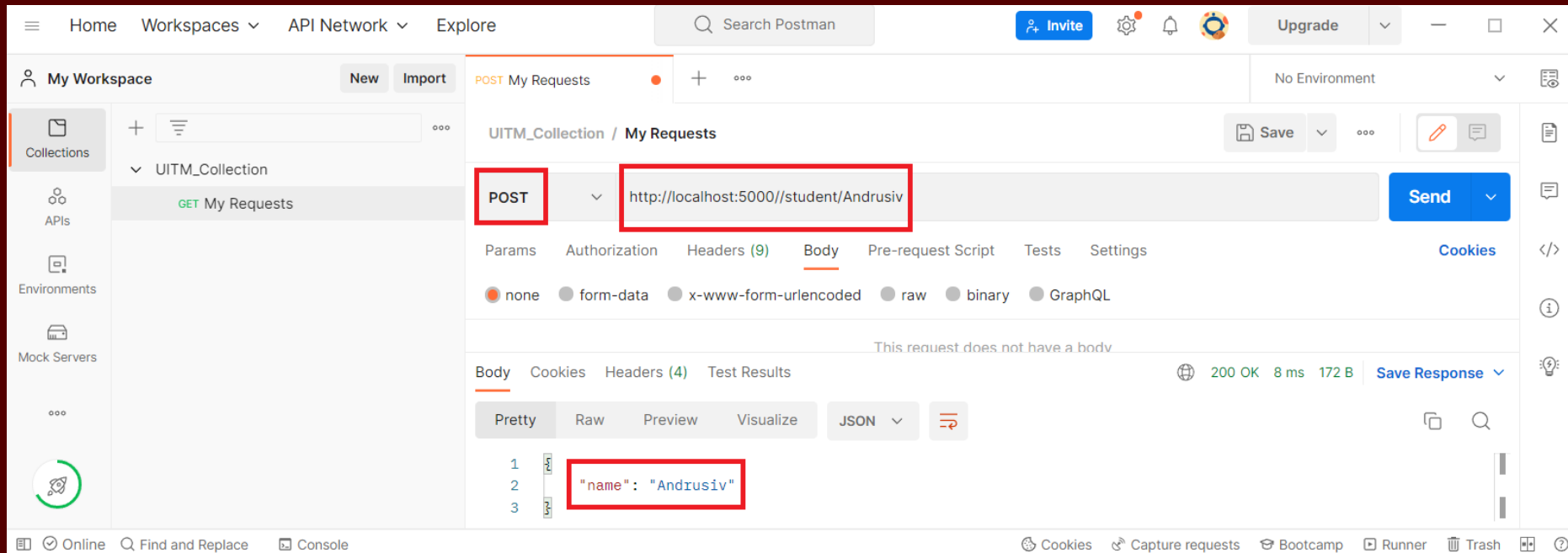
```
42 class AllNames(Resource):
43
44     def get(self):
45         # return all STUDENTS
46         return {'students': students}
47
48
49 api.add_resource(StudentNames, '/student/<string:name>')
50 api.add_resource(AllNames, '/students')
51
52 if __name__ == '__main__':
53     app.run(debug=True)
54
```

Rys.2.A.3. Implementacja aplikacji *Flask REST CRUD API* w środowisku *Microsoft VS Studio Code* (3)

Aplikacja *Flask REST CRUD API* (2/3)



■ *Postman* (1):



Rys.2.B.1. Wysłanie zapytania (oraz odpowiedź) z aplikacji klienckiej *Postman* – ***dodanie*** nowego obiektu (***studenta***)

Aplikacja *Flask REST CRUD API* (2/3)



■ *Postman* (2):

The screenshot shows the Postman interface with a GET request to `http://localhost:5000//students` in the `UITM_Collection / My Requests` collection. The response is a JSON array of student names, displayed in the `Body` tab. The response status is `200 OK` with a response time of `7 ms` and a size of `322 B`.

Request Details:

- Method: GET
- URL: `http://localhost:5000//students`
- Params: None
- Headers: 8
- Body: None

Response Details:

- Status: 200 OK
- Time: 7 ms
- Size: 322 B
- Format: JSON

Response Body (JSON):

```
1 {
2   "students": [
3     {
4       "name": "Andrusiv"
5     },
6     {
7       "name": "Cyran"
8     },
9     {
10      "name": "Wilk"
11    }
12  ]
13 }
```

Rys.2.B.2. Wysłanie zapytania (oraz odpowiedź) z aplikacji klienckiej *Postman* – wyświetlenie listy *wszystkich* studentów

Aplikacja *Flask REST CRUD API* (2/3)



■ *Postman* (3):

The screenshot displays the Postman interface with a DELETE request configured. The request is part of a collection named 'UITM_Collection' and is titled 'My Requests'. The URL is 'http://localhost:5000//student/Wilk'. The response is a 200 OK status with a body containing the JSON object: `{ "note": "delete successful" }`.

Request Details:

- Method: DELETE
- URL: http://localhost:5000//student/Wilk
- Params: None
- Headers: 8
- Body: None
- Pre-request Script: None
- Tests: None
- Settings: None

Response Details:

- Status: 200 OK
- Time: 6 ms
- Size: 181 B
- Body: `{ "note": "delete successful" }`

Rys.2.B.3. Wysłanie zapytania (oraz odpowiedź) z aplikacji klienckiej *Postman* – *usuwanie* wybranego obiektu (*studenta*)

Autoryzacja *Flask-JWT* (3/3)



■ *Microsoft VS Studio Code* (1):

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  JUPYTER

PS C:\Workspace\UITM_2022_2023> pip install Flask-JWT
```

Rys.3.A. Instalacja pakietu ***Flask-JWT***

```
user.py  X
03_flask_JWT_authorisation > user.py
1  # user.py
2
3  class User():
4
5      def __init__(self,id,username,password):
6
7          self.id = id
8          self.username = username
9          self.password = password
10
11      def __str__(self):
12          return f"User ID: {self.id}"
13
```

Rys.3.A.1. Implementacja aplikacji w ***MS VS Studio Code*** (plik ***user.py***)

Autoryzacja *Flask-JWT* (3/3)



■ *Microsoft VS Studio Code* (2):

```
secure_check.py X
03_flask_jwt_authorisation > secure_check.py
1  # secure_check.py
2
3  # In the real world/solution this should be a database table!
4  from user import User
5
6  users = [ User(1, 'Piątek', 'my_secret_pass'),
7            |   |   | User(2, 'Barabash', 'another_pass')]
8
9  username_table = {u.username: u for u in users}
10 userid_table = {u.id: u for u in users}
11
12
13 def authenticate(username, password):
14     user = username_table.get(username, None)
15     if user and password == user.password:
16         return user
17
18 def identity(payload):
19     user_id = payload['identity']
20     return userid_table.get(user_id, None)
21
```

Rys.3.A.2. Implementacja aplikacji w MS VS Studio Code (plik *secure_check.py*)

Autoryzacja *Flask-JWT* (3/3)



■ *Microsoft VS Studio Code* (3.1):

```
auth_api.py X
03_flask_JWT_authorisation > auth_api.py
1  from flask import Flask, request
2  from flask_restful import Resource, Api
3  from secure_check import authenticate, identity
4  from flask_jwt import JWT, jwt_required
5
6  app = Flask(__name__)
7  app.config['SECRET_KEY'] = 'mysecretkey'
8  api = Api(app)
9
10 jwt = JWT(app, authenticate, identity)
11
12 # We should call to the database!
13 # However, right now its just a list of dictionaries
14 # students = [{'name': 'Andrusiv'}, {'name': 'Cyran'}, .....]
15 # Keep in mind, its in memory, it clears with every restart!
16 students = []
```

Rys.3.A.3.1. Implementacja aplikacji w MS VS Studio Code (plik *auth_api.py*)

Autoryzacja *Flask-JWT* (3/3)



■ *Microsoft VS Studio Code* (3.2):

```
18 class StudentNames(Resource):
19     def get(self, name):
20         print(students)
21
22         # Cycle through list for students
23         for stud in students:
24             if stud['name'] == name:
25                 return stud
26
27         # If it's request of a student not yet in the students list
28         return {'name': None}, 404
29
30     def post(self, name):
31         # Add the dictionary to list
32         stud = {'name': name}
33         students.append(stud)
34         # Then return it back
35         print(students)
36         return stud
37
38     def delete(self, name):
39
40         # Cycle through list for students
41         for ind, stud in enumerate(students):
42             if stud['name'] == name:
43                 # don't really need to save this
44                 delted_stud = students.pop(ind)
45                 return {'note': 'delete successful'}
```

Autoryzacja *Flask-JWT* (3/3)



■ *Microsoft VS Studio Code* (3.3):

```
47 class AllNames(Resource):
48
49     @jwt_required()
50     def get(self):
51         # return all STUDENTS
52         return {'students': students}
53
54
55 api.add_resource(StudentNames, '/student/<string:name>')
56 api.add_resource(AllNames, '/students')
57
58 if __name__ == '__main__':
59     app.run(debug=True)
60
```

Rys.3.A.3.3. Implementacja aplikacji w *MS VS Studio Code* (plik *auth_api.py*)

Autoryzacja *Flask-JWT* (3/3)



■ *Postman* (1):

The screenshot shows the Postman interface with a GET request to `http://localhost:5000/students`. The response is a 401 UNAUTHORIZED status with a JSON body indicating that the request does not contain an access token.

Request: GET `http://localhost:5000/students`

Response: 401 UNAUTHORIZED 7 ms 327 B

Body (JSON):

```
{
  "description": "Request does not contain an access token",
  "error": "Authorization Required",
  "status_code": 401
}
```

Rys.3.B.1. Próba wyświetlenia listy studentów bez autoryzowanego dostępu w aplikacji klienckiej *Postman*

Autoryzacja *Flask-JWT* (3/3)



■ *Postman* (2):

1

POST

Params Authorization **Headers (10)** Body Pre-request Script Tests Settings Cookies

Headers 8 hidden

	KEY	VALUE	DESCRIPTION	...	Bulk Edit	Presets
<input checked="" type="checkbox"/>	Content-Type	application/json				

2

POST

Params Authorization Headers (10) **Body** Pre-request Script Tests Settings Cookies

none form-data x-www-form-urlencoded raw binary GraphQL JSON Beautify

```
1 {
2   "username": "Piątek",
3   "password": "my_secret_pass"
4 }
```

Body Cookies Headers (4) Test Results 200 OK 5 ms 339 B Save Response

Pretty Raw Preview Visualize JSON

```
1 {
2   "access_token": "eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJleHAiOjE2NjkxNTMxMzIsIm1hdCI6MTY2OTE1MjgzMiwibmJmIjojY5MTUyODMyLCJpZGVudGl0eSI6MX0.KHSDzP_-bX0Pwvf0sJs1levBRb2NTXxIAJKCdNf5r4g"
3 }
```

Rys.3.B.2. Autoryzacja dostępu dla wybranego użytkownika (*Piątek*) w aplikacji klienckiej *Postman*

Autoryzacja *Flask-JWT* (3/3)



■ *Postman* (3):

The screenshot shows the Postman interface with a GET request to `http://localhost:5000/students`. The request is configured with the following headers:

KEY	VALUE	DESCRIPTION
Content-Type	application/json	
Authorization	JWT eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1Ni...	

The response is a JSON array of student names:

```
1 {
2   "students": [
3     {
4       "name": "Rachwał"
5     },
6     {
7       "name": "Słonka"
8     },
9     {
10      "name": "Szkafarak"
11    }
12  ]
13 }
```

Rys.3.B.3. Wyświetlenie listy studentów po uzyskaniu autoryzacji dla wybranego użytkownika (*Piątek*) w aplikacji klienckiej *Postman*



**Wyższa Szkoła Informatyki i Zarządzania
ul. Sucharskiego 2, 35-225 Rzeszów, Polska**



Dziękuję za uwagę...

