Web Services Python

Het Flask micro web framework (2)

Kristof Michiels

Flask API's

Flask API's

- Vorige week hebben we gezien dat Flask een veelzijdig micro framework is om webtoepassingen te maken
- Met Flask kan je ook snel en veilig API's bouwen
- We bekijken dit vandaag aan de hand van een voorbeeldtoepassing
- Web en API kunnen gecombineerd worden in één en dezelfde toepassing...
- Ook kan er geopteerd worden om de zaken volledig te scheiden

Een eerste API-toepassing

Een eerste API-toepassing

- Onze app heet Webfaves API: een API die interessante weblinks verzamelt
- ! We bouwen hier enkel een application programming interface, geen user interface
- Verschillende types van toepassingen kunnen er dan in een latere fase mee interfacen, zodat de API-laag volledig gescheiden blijft van de presentatie

Features

- Alle webfaves oplijsten
- Nieuwe gebruikers registreren
- Bestaande gebruikers authenticeren
- Nieuwe webfaves toevoegen
- Bestaande webfaves updaten
- Webfaves verwijderen wanneer nodig

De appplicatie aanmaken

- Door een nieuw Flask project aan te maken in Pycharm
- Enkel professionele versie van Pycharm heeft deze handige feature (gratis voor studenten)
- Zoals de app van vorige week => Pycharm doet het werk voor ons
- Eventueel: de applicatie in Debug mode (automatisch herladen bij wijzigingen)

De applicatie aanmaken

```
from flask import Flask

app = Flask(__name__)

@app.route("/")
def start():
    return "Welkom bij de Webfaves API!"

@app.route("/api")
def api():
    return "De webfaves API zegt Hallo!"

if __name__ == '__main__':
    app.run()
```

Postman gebruiken om onze endpoints te testen

- We gaan 4 soorten requests gebruiken: GET, POST, PUT en DELETE
- GET is eenvoudig met de browser te testen, voor de andere hebben we een tool nodig
- Beste tool om API's te testen is Postman: https://www.postman.com/downloads/
- Download Postman zodat je calls kunt maken naar je lokale machine
- Test je eerste requests met Postman
- Bekijk ook de run window in Pycharm

JSON teruggeven

- Werken met JSON: deze resource goed doornemen als opdracht!
- Key-value-paren
- jsonify-functie gebruiken

```
from flask import Flask, jsonify

@app.route("/api")
def api():
    return jsonify(message="De webfaves API zegt hallo!")
```

HTTP Status Codes

- Herinner je het request-response mechanisme uit de eerste les?
- Requests en responses hebben headers met meta-data. Ze bevatten o.m. een status code, bvb 200 OK, 404 Not found, 401 Unauthorized

```
@app.route("/api")
def api():
    return jsonify(message="De webfaves API zegt hallo!"), 200
```

URL parameters

Laten ons toe om dynamische content te presenteren op basis van de URL-parameters die werden meegegeven. Hier een voorbeeld met querystrings zoals we ook vorige week hebben gezien

```
from flask import Flask, jsonify, request

@app.route("/url-parameters")
def parameters():
    naam = request.args.get("naam")
    url = request.args.get("url")
    if naam and naam not in ["mdn","w3schools"]:
        return jsonify(message="Blij met deze nieuwe URL: '" + url + "'."), 200
    else:
        return jsonify(message="We hebben eigenlijk al genoeg van deze resources.
        Just kidding ;-)"), 401
```

URL-variabelen

Een voorbeeld met URL-variabelen als onderdeel van de URL

```
@app.route("/url-variabelen/<string:naam>")
def parameters_bis(naam):
    if naam and naam not in ["mdn","w3schools"]:
        return jsonify(message="Blij met deze nieuwe naam: '" + naam + "'."), 200
    else:
        return jsonify(message="We hebben eigenlijk al genoeg van deze resources.
        Just kidding ;-)"), 401
```

Clean URLs zijn belangrijk: https://stackoverflow.blog/2020/03/02/best-practices-for-rest-api-design/

Onze data vasthouden

Onze data vasthouden

- We moeten data die onze API genereert of ontsluit ergens kunnen vasthouden
- We hebben in het web-voorbeeld van vorige week gezien hoe we een eenvoudige file-datastore kunnen bouwen in de vorm van een json-bestand
- Deze week werken we met SQLite, een file-gebaseerde databank
- We gaan ook een object-relational mapper (ORM) gebruiken: SQLAlchemy
- Werkt met Python objecten, geen SQL
- Je beheert je database vanuit je code

SQLAlchemy configuratie

- Heel eenvoudig om in latere fase van databank te veranderen. Ondersteunt heel wat databanken
- Flask-SQLAlchemy package installeren (2.5.1)

```
from flask_sqlalchemy import SQLAlchemy
from sqlalchemy import Column, Integer, String, Float
import os

basedir = os.path.abspath(os.path.dirname(__file__))
app.config["SQLALCHEMY_DATABASE_URI"] = "sqlite:///" + os.path.join(basedir,
"webfaves.db")
```

ORM models klasses

Je beschrijft je databank tabellen als klasses. Wij hebben 2 tabellen: users en webfaves

```
class User(db.Model):
    __tablename__ = 'users'
    id = Column(Integer, primary_key=True)
    voornaam = Column(String)
    familienaam = Column(String)
    email = Column(String, unique=True)
    wachtwoord = Column(String)
```

ORM models klasses

```
class Webfave(db.Model):
    __tablename__ = 'webfaves'
    id = Column(Integer, primary_key=True)
    naam = Column(String)
    type = Column(String)
    categorie = Column(String)
    rating = Column(Float)
    moeilijkheidsgraad = Column(Float)
    originaliteit = Column(Float)
```

Database aanmaken en voorzien van data

We maken hiervoor met Flask CLI 3 scripts aan, allemaal binnen onze toepassing:

```
db = SQLAlchemy(app)

@app.cli.command("db_create")
def db_create():
    db.create_all()
    print("Databank aangemaakt!")


@app.cli.command('db_drop')
def db_drop():
    db.drop_all()
    print("Databank verwijderd!")
```

Database aanmaken en voorzien van data

```
@app.cli.command('db_seed')
def db_seed():
    mdn_json = Webfave(
        naam="JSON - MDN Web Docs",
        url="https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference...",
        type="Link",
        categorie="technisch document",
        rating="4.0",
        moeilijkheidsgraad="3.0")
    ...
```

Database aanmaken en voorzien van data

```
db.session.add(mdn_json)
db.session.add(realpython_sqlalchemy)
db.session.add(csstricks_database_queries)

testgebruiker = User(voornaam="Kristof",
 familienaam="Michiels",
 email="kristof.michiels01@ap.be",
 wachtwoord="test")

db.session.add(testgebruiker)
db.session.commit()
```

Scripts runnen in de terminal + database checken

- export FLASK_APP=app-v2.py / set FLASK_APP=app-v2.py
- flask db_create
- flask db_seed
- DB Browser for SQLite: https://sqlitebrowser.org/

Lijst van Webfaves uit de database halen

```
@app.route("/webfaves", methods=['GET'])
def webfaves():
    webfaves_list = Webfave.query.all()
    return jsonify(data=webfaves_list)
```

Maar: "Object of type Webfave is not JSON serializable" => Marshmallow...

Marshmallow

- We gebruiken Marshmallow om objecten te (de-)serializeren
- https://flask-marshmallow.readthedocs.io/en/latest/: deze resource goed doornemen als opdracht!
- Integreert met SQLAlchemy
- Package flask-marshmallow installeren: 0.14.0
- We moeten Marshmallow toevoegen aan onze app en vertellen welke velden het moet zoeken

```
from flask_marshmallow import Marshmallow

ma = Marshmallow(app)
```

Marshmallow

Flask-marshmallow laat je een aantal schema-klasses definiëren

```
class UserSchema(ma.Schema):
class Meta:
    fields = ("id", "voornaam", "familienaam", "email", "wachtwoord")

class WebfaveSchema(ma.Schema):
    class Meta:
        fields = ("id", "naam", "url", "type", "categorie", "rating",
        "moeilijkheidsgraad", "originaliteit")

user_schema = UserSchema()
users_schema = UserSchema(many=True)
```

Marshmallow

```
webfave_schema = WebfaveSchema()
webfaves_schema = WebfaveSchema(many=True)
```

```
@app.route("/webfaves", methods=["GET"])
def webfaves():
    webfaves_list = Webfave.query.all()
    result = webfaves_schema.dump(webfaves_list)
    return jsonify(result)
```

API beveiliging

API: beveiliging met JWT

- JSON Web Tokens: https://jwt.io: bekijk dit!
- Flask-Login, Flask-User: web toepassingen, sessies
- Flask-JWT-Extended package installeren: 4.3.1
- Decorators toevoegen aan de routes die je wil beveiligen

Registreren van gebruikers

```
def registreren():
    email = request.form["email"]
    test = User.query.filter_by(email=email).first()
    if test:
        return jsonify(message="Dit email-adres bestaat reeds"), 409
    else:
        voornaam = request.form["voornaam"]
        familienaam = request.form["familienaam"]
        wachtwoord = request.form["wachtwoord"]
        user = User(voornaam=voornaam, familienaam=familienaam, email=email,
        wachtwoord=wachtwoord)
        db.session.add(user)
        db.session.commit()
        return jsonify(message="Gebruiker met succes aangemaakt"), 201
```

Gebruikers laten inloggen

```
from flask_jwt_extended import JWTManager, jwt_required, create_access_token

app.config["JWT_SECRET_KEY"] = "deodkoorjf42sdzzzdcd"

jwt = JWTManager(app)
```

Gebruikers laten inloggen

```
@app.route("/login", methods=["POST"])
def login():
    if request.is_json:
        email = request.json["email"]
        wachtwoord = request.json["wachtwoord"]
    else:
        email = request.form["email"]
        wachtwoord = request.form["wachtwoord"]

    test = User.query.filter_by(email=email, wachtwoord=wachtwoord).first()
    if test:
        access_token = create_access_token(identity=email)
        return jsonify(message="Login succesvol!", access_token=access_token)
    else:
        return jsonify(message="Fout email of wachtwoord"), 401
```

CRUD

CRUD

- CRUD = Create, Read, Update, Delete
- Een webfave opvragen
- Een webfave toevoegen (POST)
- Endpoint voor toevoegen beveiligen
- Een bestaande webfave wijzigen (PUT)
- Een webfave verwijderen (DEL)

Een webfave opvragen

```
@app.route('/webfave/<int:id>', methods=["GET"])
def webfave(id: int):
    webfave = Webfave.query.filter_by(id=id).first()
    if webfave:
        result = webfave_schema.dump(webfave)
        return jsonify(result)
    else:
        return jsonify(message="Deze webfave bestaat niet"), 404
```

Een webfave toevoegen (POST)

```
@app.route("/nieuwe_webfave", methods=["POST"])
def nieuwe_webfave():
    naam = request.form["naam"]
    test = Webfave.query.filter_by(naam=naam).first()
    if test:
        return jsonify("Er bestaat reeds een webfave met deze naam"), 409
    else:
        url = request.form["url"]
        webfave_type = request.form["type"]
        categorie = request.form["categorie"]
        rating = float(request.form["rating"])
        moeilijkheidsgraad = float(request.form["moeilijkheidsgraad"])
        originaliteit = float(request.form["originaliteit"])
```

Een webfave toevoegen (POST)

```
    nieuwe_webfave = Webfave(naam=naam,
    type=webfave_type,
    url=url,
    categorie=categorie,
    rating=rating,
    moeilijkheidsgraad=moeilijkheidsgraad,
    originaliteit=originaliteit)

    db.session.add(nieuwe_webfave)
    db.session.commit()
    return jsonify(message="De nieuwe webfave is toegevoegd"), 201
```

Endpoint voor toevoegen beveiligen

- In Postman: token toevoegen dat je verkrijgt na een succesvolle login
- Ga naar tabblad Authorization en selecteer "Bearer token"
- Plak in het veld "token" het verkregen token

```
@app.route("/nieuwe_webfave", methods=["POST"])
@jwt_required()
def nieuwe_webfave():
    ...
```

Een bestaande webfave wijzigen (PUT)

```
@app.route("/update_webfave", methods=["PUT"])
@jwt_required()
def update_webfave():
    id = int(request.form["id"])
    webfave = Webfave.query.filter_by(id=id).first()
    if webfave:
        webfave.naam = request.form["naam"]
        webfave.type = request.form["type"]
        webfave.url = request.form["url"]
        webfave.categorie = request.form["categorie"]
        webfave.rating = float(request.form["rating"])
        webfave.moeilijkheidsgraad = float(request.form["moeilijkheidsgraad"])
        webfave.originaliteit = float(request.form["originaliteit"])
```

Een bestaande webfave wijzigen (PUT)

```
        db.session.commit()
        return jsonify(message="Je hebt een webfave aangepast"), 202
    else:
        return jsonify(message="De webfave die ..."), 404
```

Een webfave verwijderen (DEL)

```
@app.route("/verwijder_webfave/<int:id>", methods=["DELETE"])
@jwt_required()
def verwijder_webfave(id: int):
    webfave = Webfave.query.filter_by(id=id).first()
    if webfave:
        db.session.delete(webfave)
        db.session.commit()
        return jsonify(message="De webfave werd met succes verwijderd"), 202
    else:
        return jsonify(message="De webfave die je wilde verwijderen bestaat niet")
```

Labo

Labo 3

- Je exploreert de mogelijkheden van Flask voor API's zoals ze hier zijn meegegeven
- Je bestudeert de bronnen die bij de slides staan vermeld
- Je maakt een eigen API-toepassing, met voldoende uitdaging
- Het thema kies je zelf en je vult het zelf in
- Zorg voor beveiliging met JWT
- Zorg voor een achterliggende SQLite databank
- Je documenteert je toepassing via de resource "/home"
- Je test je endpoints met behulp van Postman