README

Autor: Christian Grubmüller

Datum: 10.12.2021

In diesem Repository ist eine Aufgabe, die ich für die Schule gemacht habe. Die Aufgabe war eine *REST-Schnittstelle* zu implementieren, bei der man sich mit *Anfragen* **registrieren** und **anmelden** kann. Die Schnittstelle wurde mit *Python* und *Flask* umgesetzt. Die E-Mail wird bei der Registrierung auf die Gültigkeit überprüft und dann mit dem gehashten Passwort in einer *SQLite* Datenbank abgespeichert. Damit die Verbindung gesichert ist, wurde auch ein selbst signiertes *SSL-Zertifikat* hinzugefügt. Das Deployment erfolgt über einen *Gunicorn* in einem *Docker*-Container.

Ausführen

```
# Repository klonen
git clone git@github.com:TGM-HIT/syt5-gk961-cloud-datenmanagement-
cgrubmueller.git
cd syt5-gk961-cloud-datenmanagement-cgrubmueller

# Docker-Image builden
docker build -t cloud_interface .
# Docker Container ausführen
docker run -d -p 8001:8001 cloud_interface
```

Testen

```
# Registrieren
curl --location --request POST 'https://localhost:8001/register' --header
'Content-Type: application/json' --data-raw
'{"bname":"cgrubmueller","email":"cgrubmueller@student.tgm.ac.at",
"passwd":"123456"}'

# Einloggen -> Welcome
curl --location --request POST 'https://localhost:8001/login' --header 'Content-
Type: application/json' --data-raw
'{"bname":"cgrubmueller","email":"cgrubmueller@student.tgm.ac.at",
"passwd":"123456"}'

# Nochmal registrieren -> This email is already in use!
curl --location --request POST 'https://localhost:8001/register' --header
'Content-Type: application/json' --data-raw
'{"bname":"cgrubmueller","email":"cgrubmueller@student.tgm.ac.at",
"passwd":"123456"}'

# Invalide Email -> The email cgrubmuellerstudent.tgm.ac.at is invalid. Try again
with a valid email.
curl --location --request POST 'https://localhost:8001/login' --header 'Content-
Type: application/json' --data-raw
'{"bname":"cgrubmueller","email":"cgrubmuellerstudent.tgm.ac.at",
"passwd":"123456"}'

# Falsche Email -> There are no user registered with this email!
```

```
curl --location --request POST 'https://localhost:8001/login' --header 'Content-
Type: application/json' --data-raw
'{"bname":"cgrubmueller","email":"grubmueller@student.tgm.ac.at",
"passwd":"123456"}'

# Falsches Passwort
curl --location --request POST 'https://localhost:8001/login' --header 'Content-
Type: application/json' --data-raw
'{"bname":"cgrubmueller","email":"cgrubmueller@student.tgm.ac.at",
"passwd":"156"}'
```

Deployment

Da man Python an sich eigentlich nicht deployen kann, verwende ich Docker.

Dafür habe ich ein File namens `Dokerfile` erstellt. In diesem File wird definiert, dass *python:3.7-alpine* als "Grundlage" verwendet werden soll. Dann wird der Ordner `/app` erstellt und als Working-Directory definiert.

Anschließend werden die Dateien, die notwendig für die Ausführung des Programms sind hinzugefügt.

Danach werden alle Python-Dependencies geladen.

Zu guter Letzt werden wird festgelegt, dass der `ENTRYPOINT` das File `gunicorn.sh` ist.

Mit diesem File kann man ein Docker-Image erstellen, welches man dann laufen lassen kann.

```
FROM python:3.7-alpine

RUN mkdir /app
WORKDIR /app
ADD gunicorn.sh /app
ADD requirements.txt /app
ADD src/rest.py /app
ADD src/database.db /app

RUN pip3 install -r requirements.txt

ENTRYPOINT ["./gunicorn.sh"]
```

Flask & Gunicorn

Weil Flask nicht für Production gedacht ist, verwende ich stattdessen Gunicorn. [1]

Das ist ein HTTP Server für WSGI Applikationen. Oben sieht man schon das der `ENTRYPOINT` das `.sh`-File ist. In diesem kann man folgenden Code finden.

```
#!/bin/sh
gunicorn rest:app -w 2 --threads 2 -b 0.0.0.0:8001
```

`rest:app` ist quasi der Pfad, an dem Gunicorn die Flask-Applikation finden kann.

`-w 2` heißt das es 2 Worker-Threads gibt, die die Anfragen bearbeiten

`--threads` bedeutet das die 2 Worker Thread in weiter 2 Threads aufgeteilt sind????

`-b 0.0.0.0:8001` legt fest, dass sich gunicorn an eine frei IP bindet. Mit dem Port 8001

Implementierung

Die Aufgabe habe ich mit Flask und mit SQLite umgesetzt. SQLite habe ich verwendet, weil MongoDB für diese Aufgabe zu aufwändig wäre.

Flask

Flask is a lightweight WSGI web application framework. It is designed to make getting started quick and easy, with the ability to scale up to complex applications. It began as a simple wrapper around Werkzeug and Jinja and has become one of the most popular Python web application frameworks.

Flask offers suggestions, but doesn't enforce any dependencies or project layout. It is up to the developer to choose the tools and libraries they want to use. There are many extensions provided by the community that make adding new functionality easy. [2]

Da ein Flask Server nicht für das Deployment geeignet ist, verwende *gunicorn*. [1]

Gunicorn 'Green Unicorn' is a Python WSGI HTTP Server for UNIX. It's a pre-fork worker model ported from Ruby's Unicorn project. The Gunicorn server is broadly compatible with various web frameworks, simply implemented, light on server resources, and fairly speedy. [3]

SQLite

SQLite is a C-language library that implements a small, fast, self-contained, high-reliability, full-featured, SQL database engine. SQLite is the most used database engine in the world. SQLite is built into all mobile phones and most computers and comes bundled inside countless other applications that people use every day. [4]

Webserver starten

Mit folgendem Code wird das Programm und der Flask Server gestartet.

```
if __name__ == "__main__":
    app.run(debug=True, host='0.0.0.0', port=8001, ssl_context='adhoc')
```

`host='0.0.0.0'` bedeutet, das sich der Flask-Server auf irgendeine IP-Adresse bindet. Das ist notwendig, weil ansonsten kann man nicht darauf zugreifen, wenn er in dem Docker-Container läuft. [5]

`port=8001` bedeutet, dass Port 8001 verwendet wird.

Rest-Schnittstellen

Um mit dem Flask-Server ein Rest-Schnittstelle zu Verfügung zu stellen, kann man folgenden Code verwenden. Dort wird zuerst die URL definiert, wie man darauf zugreift und anschließend die *method* wie man darauf zugreifen kann. Wenn man nun auf `loalhost:8001/login` mittels *POST* zugreift, wird die Methode `loginPost()` ausgeführt. Das habe ich außerdem auch noch für `hello_world()`, `signupPOST()`, und `reset()` gemacht.

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

Datenübertragung mit JSON

Schicken

Um Daten im JSON-Format über Rest zu verschicken, muss man in im `curl`-Befehl definieren, dass man im Body diese Formatierung verwendet.

Zum Beispiel wird das hier mit `-H "Content-Type: application/json` gemacht.

```
curl -X POST -H "Content-Type: application/json" -d
'{"email":"cgrubmueller@student.tgm.ac.at", "bname":"cgrubmueller",
"passwd":"123456"}' http://127.0.0.1:8001/login
```

Empfangen

Wenn man das gemacht hat, kann man in Python auch darauf zugreifen. Das kann mit `request.json` machen. Dann kann man mit dem richtigen String als Key darauf zuggreifen.

```
credentials = request.json
...
bname = credentials['bname']
email = credentials['email']
passwd = credentials['passwd']
```

E-Mail check 6

Um zu überprüfen, ob die E-Mail valide ist, verwende ich Regex (Python-Modul: `re`).

Dafür wird global der String `email_regex` definiert, mit dem dann die Benutzereingaben verglichen werden.

Später im Code kann man dann mit der Methode `fullmatch()` vergleichen ob die E-Mail gültig ist. Wenn sie gültig ist, wird *true* zurückgeliefert, wenn nicht *false*.

```python
import re

# Regex-Pattern für E-Mail
email_regex = r'\b[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Z|a-z]{2,}\b'

# checking if the email is valid
re.fullmatch(email_regex, email) # return true or false
```

SQLite 7

Da MongoDB für diese Übung zu aufwendig ist, verwende ich hier jetzt SQLite.

Dafür muss man die Library `sqlite3` importieren. `pip install pysqlite3`

Anschließend kann man mit dem Pfad zu dem Datenbankfile auf die Datenbank mittels SQL zugreifen.

Lesen

In diesem Beispiel wird auf die Datenbank zugegriffen. Dann werden Daten mittels `SELECT` ausgelesen und in der Variable `users` als Array, das Tupels enthält, abgespeichert.

```python
with sql.connect("database.db") as con:
    cur = con.cursor()
    cur.execute(f"SELECT bname, email, passwd FROM benutzer WHERE email = ?",
[email])
    users = cur.fetchall()
```

Schreiben

Hier werden Daten in die Tabelle Benutzer abgespeichert. Falls diese Tabelle noch nicht existiert, wird sie erstellt. Nachdem man die Daten insertet hat, wird diese Transaktion commitet, damit die Daten auch wirklich abgespeichert sind. Natürlich werden sie mit einem *prepared statement* abgespeichert, da ansonsten SQL-Injections möglich sind.

```python
with sql.connect("database.db") as con:
    con.execute("CREATE TABLE IF NOT EXISTS benutzer (email TEXT PRIMARY KEY,
bname TEXT, passwd TEXT)")
    con.execute("INSERT INTO benutzer (bname,email,passwd) VALUES (?,?,?)",
(bname, email,hash_password(passwd)))
    con.commit()
```

Zurücksetzten

Um die Datenbank zurücksetzten, muss man einfach nur die Tabelle `benutzer` löschen.

```python
with sql.connect("database.db") as con:
    con.execute("DROP TABLE benutzer")
```

Passwort hashen [8]

Damit das Passwort nicht in Plaintext abgespeichert wird, habe ich es *gehasht* und mit einem *salt* versehen. Wenn man keinen *salt* verwenden würde, könnte man einfach mit einer Dictionary-Attack das Passwort herausfinden.

Ein *salt* ist eine Zufallszahl, die mit dem Hash addiert wird und mit einem `:` getrennt hinter dem Hash in der Datenbank abgespeichert wird. Wenn also ein Login-Request kommt wird das eingegeben Passwort gehasht und ebenfalls mit dem gleichen *salt* addiert. Wenn dann der Hash in der Datenbank und der Hash des eingegeben Passworts gleich sind, war es korrekt.

Das habe ich mit zwei Methoden in Python umgesetzt.

```
import uuid
import hashlib

# Hasht ein Passwort und generiert einen salt-Wert
def hash_password(password):
    # uuid is used to generate a random number
    salt = uuid.uuid4().hex
    return hashlib.sha256(salt.encode() + password.encode()).hexdigest() + ':' +
salt

# Überprüft, ob ein Passwort richtig ist.
def check_password(hashed_password, user_password):
    password, salt = hashed_password.split(':')
    return password == hashlib.sha256(salt.encode() +
user_password.encode()).hexdigest()
```

Verwendung

Einloggen

Um das eingegeben Passwort zu überprüfen, kann man die obere Methode in einem `if` verwenden.

```
if check_password(users[0][2] , passwd):
    msg = "\nWelcome!"
else:
    msg = "\nUsername or password was not correct!"
```

Registrieren

Hier kann man sehen, dass beim Abspeichern in die Datenbank das gehashte Passwort abgespeichert wird.

```
con.execute("INSERT INTO benutzer (bname,email,passwd) VALUES (?,?,?)", (bname,
email,hash_password(passwd)))
```

SSL-Zertifikat

Damit ich *https* verwenden kann brauche ich ein *SSL*-Zertifikat. Deshalb habe ich mir ein *self-signed* Zertifikat erstellt. Das kann man in Linux ganz einfach mit `openssl` machen. [9]

```
openssl req -newkey rsa:4096 -x509 -sha256 -days 365 -nodes -out
cloud_interface.crt -keyout cloud_interface.key
```

```
...
Country Name (2 letter code) [AU]:AT
State or Province Name (full name) [Some-State]:Niederoesterreich
Locality Name (eg, city) []:Bruck an der Leitha
Organization Name (eg, company) [Internet Widgits Pty Ltd]:SYT
Organizational Unit Name (eg, section) []:Cloud Interface Uebung
Common Name (e.g. server FQDN or YOUR name) []:Grubmueller
Email Address []:cgrubmueller@student.tgm.ac.at
...
```

Danach sind zwei Files erstellt. Ein Zertifikat (`cloud_interface.key` und `cloudinterface.crt`)

Anschließend muss ich diese Files bei *gunicorn* hinzufügen. [10]

```
gunicorn rest:app -w 2 -b 0.0.0.0:8001 --keyfile cloud_interface.key --certfile
cloud_interface.crt
```

Falls man das über Flask machen möchte, kann man das auch in den Parametern übergeben. [11]

```
app.run(debug=True, host='0.0.0.0', port=8001, ssl_context=
('../cloud_interface.crt', '../cloud_interface.key'))
```

Quellen

1. https://itnext.io/setup-flask-project-using-docker-and-gunicorn-4dcaaa829620 (10.12.2021) ↩ ↩
2. https://pypi.org/project/Flask/ (07.10.2021) ↩
3. https://docs.gunicorn.org/en/stable/run.html (11.12.2021) ↩
4. https://sqlite.org/index.html (07.10.2021) ↩
5. https://stackoverflow.com/questions/54776600/unable-to-connect-to-flask-while-running-on-docker-container#54776696 (10.12.2021) ↩
6. https://www.geeksforgeeks.org/check-if-email-address-valid-or-not-in-python/ (10.12.2021) ↩
7. https://www.tutorialspoint.com/sqlite/sqlite_python.htm (10.12.2021) ↩
8. https://www.pythoncentral.io/hashing-strings-with-python/ (10.12.2021) ↩
9. https://www.ryangeddes.com/how-to-guides/tech/how-to-create-a-self-signed-ssl-certificate-on-linux/ (10.12.2021) ↩
10. https://docs.gunicorn.org/en/stable/settings.html#ssl (10.12.2021) ↩
11. https://www.educba.com/flask-https/ (10.12.2021) ↩