DataCraft

Datenschnittstellen

2
Eigene APIs erstellen

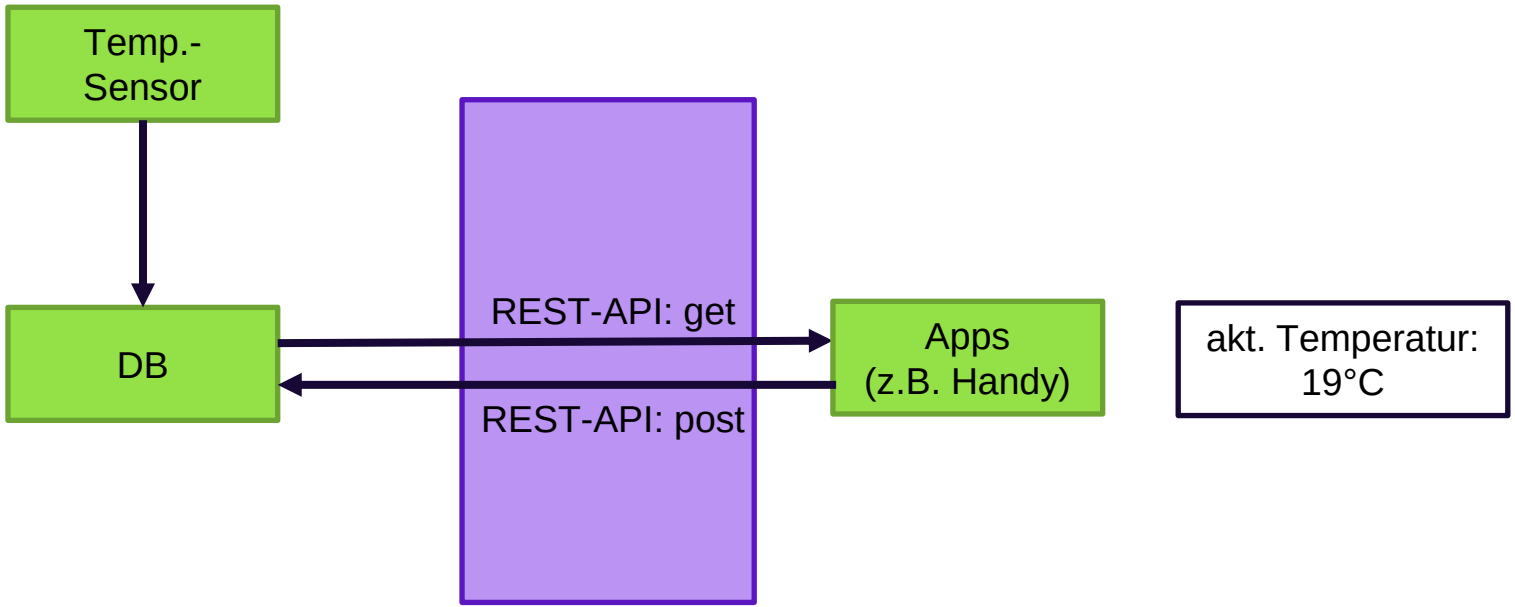
RESTful API: Eigene API-Schnittstelle

DataCraft

Es gibt mehrere Python-Tools, um APIs zu erstellen. Diese laufen erstmal auf dem lokalen PC. Um diese "öffentlich" zu machen bzw. diese auf einem Server zu konfigurieren, ist etwas aufwändiger.

- **Django**: Komplettlösung, auch für Website-Backends. Aufwändiger zu konfigurieren.
- **Flask**: Framework, um Web-Applikationen und REST-APIs zu bauen
- **FastAPI**: wie Flask, aber schnell und weniger fehleranfällig (typecheck), wird daher von vielen besser als Flask angesehen

Anwendung von REST-API



statt einer App den direkten Zugriff auf die Datenbank zu erlauben, wird eine Schnittstelle (API) eingebaut. Zudem ermöglicht es die REST-API auch, (Python-)Code auf dem Server auszuführen und damit komplexere Aufgaben zu erledigen als in der Datenbank möglich sind.

Installation von **fastapi** und **uvicorn** mittels conda (am besten eigene virtuelle Umgebung anlegen, siehe *2.1 Datenanalyse mit Python - conda*)

minimales Programm (fastapi01.py):

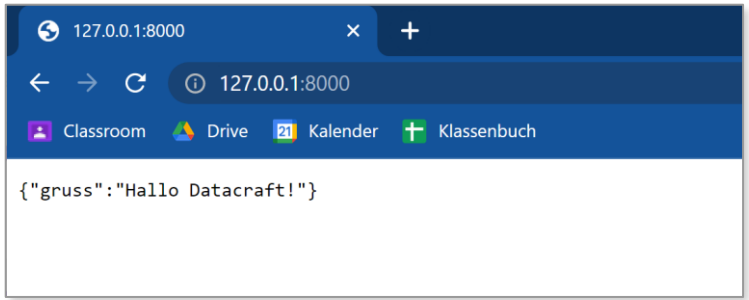
```
from fastapi import FastAPI

app = FastAPI()

@app.get("/")
def root():
    return {"gruss": "Hallo Datacraft!"}
```

Start mittels `uvicorn fastapi01:app`
Der Parameter `--reload` sorgt dafür, dass bei Änderung der py-Datei die App neu geladen wird. Abbruch mit STRG + C

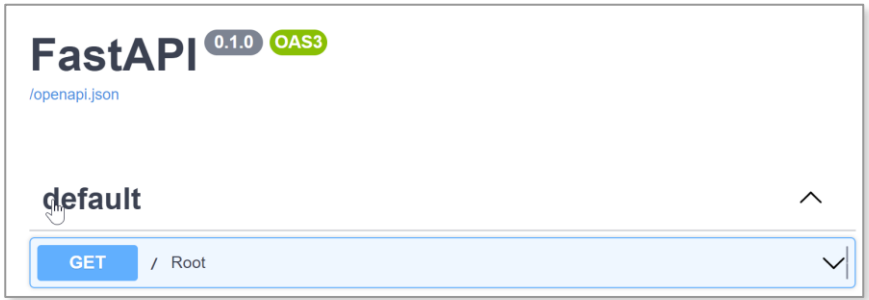
RESTful API: Eigene API-Schnittstelle mit fastAPI

DataCraft

Anzeige über: http://127.0.0.1:8000



Dokumentation: http://127.0.0.1:8000/docs



DataCraft

Neuen Pfad per Decorator hinzufügen:

```
users = [
    {"user_id": 1, "username": "AntMan", "forename": "Henry", "surname": "Pym"},
    {"user_id": 2, "username": "BlackCat", "forename": "Felicia", "surname": "Hardy"},
    {"user_id": 3, "username": "Electro", "forename": "Max", "surname": "Dillon"},
]


@app.get("/users")
async def get_users():
    return users


@app.get("/users/{id}")
async def get_user(id):
    for u in users:
        if u["user_id"] == int(id):
            return u
```

Parameter können einfach in die Funktion aufgenommen werden. Parameter werden mit ?PARAM1=WERT1 in der URL ergänzt, z.B. /users?start=1. Mehrere Parameter werden mit & verknüpft (/users?start=1&length=2)

```
users = [
    {"user_id": 1, "username": "AntMan", "forename": "Henry", "surname": "Pym"},
    {"user_id": 2, "username": "BlackCat", "forename": "Felicia", "surname": "Hardy"},
    {"user_id": 3, "username": "Electro", "forename": "Max", "surname": "Dillon"},
]

@app.get("/users")
async def get_users(start, length):
    return users[int(start):(int(start) + int(length)]
```

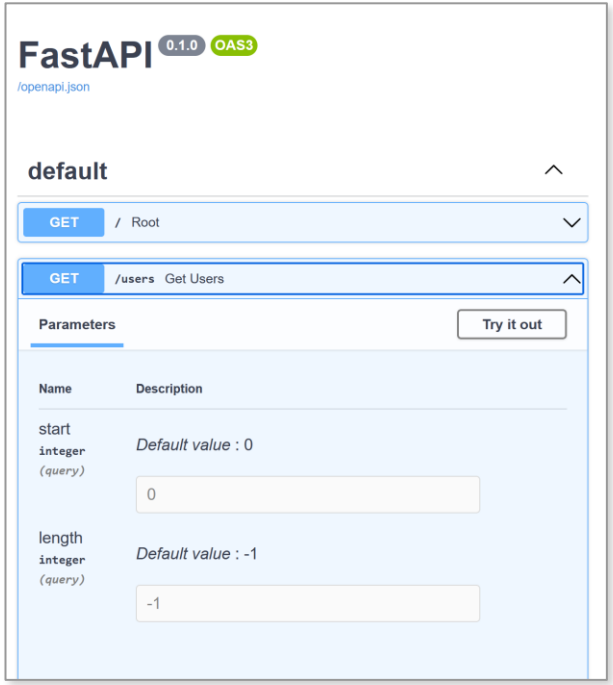
Achtung: Standardmäßig sind Parameter immer strings.
Besser ist es, den Typ direkt in der Funktion zu definieren (Python typehints)

```
@app.get("/users")
async def get_users(start:int, length:int):
    return users[ start:(start + length)]
```

Damit Parameter optional sind, müssen Defaultwerte angegeben werden

```
@app.get("/users")
async def get_users(start: int = 0, length: int = -1):
    if length == -1:
        length = len(users)
    return users[start : (start + length)]
```

Typ und Defaultwerte werden automatisch in der Dokumentation angegeben



DataCraft

DataFrames lassen sich mit to_dict umwandeln, damit sie ausgegeben werden können. Der Parameter orient="records" sorgt für das übliche JSON-Format, d.h. eine Liste von Einträgen, wobei jeder Eintrag einer Zeile entspricht.

```
@app.get("/penguins")
async def get_penguins():
    penguins = sns.load_dataset("penguins")
    return penguins.to_dict(orient="records")
```

Probleme machen jedoch NaN bei floats, da diese standardmäßig nicht richtig konvertiert werden. Entweder verzichtet man auf den Gebrauch von NaN (df.dropna()) oder man benutzt einen anderen JSON-Konverter:

```
import orjson
from fastapi.responses import JSONResponse

class ORJSONResponse(JSONResponse):
    media_type = "application/json"
    def render(self, content):
        return orjson.dumps(content)

# Initialisieren der App mit dem veränderten JSON-Konverter
app = FastAPI(default_response_class=ORJSONResponse)
```

Natürlich lassen sich auch POST-Abfragen realisieren. Dabei ist es ratsam, die Datenform vorher genau zu definieren. Das funktioniert über pydantic:

```
from pydantic import BaseModel

class User(BaseModel):
    user_id: int
    username: str
    forename: str
    surname: str

users = [
    User(user_id=1, username="AntMan", forename="Henry", surname="Pym"),
    User(user_id=2, username="BlackCat", forename="Felicia", surname="Hardy"),
    User(user_id=3, username="Electro", forename="Max", surname="Dillon"),
]

@app.post("/users", status_code=201)
async def add_user(new_user: User):
    users.append(new_user)
```