Daten aus Social Media automatisiert herunterladen

Das Automatisieren von Downloads spielt bei der Auswertung von Daten eine große Rolle. Oftmals liegen riesige Datenmengen vor, die allerdings auf mehrere Dateien aufgeteilt wurden. Das betrifft vor allem Daten, die von diversen Diensten wie einem Fahrradverleih zur Verfügung gestellt werden. Andererseits sind Daten von sozialen Medien gefragt, anhand derer sich beispielsweise erkennen lässt, wie beliebt ein Post, Tweet oder Video ist.

Für etliche Anbieter von sozialen Medien existieren bereits Bibliotheken, die Zugriff auf das API eines sozialen Netzwerks ermöglichen [1]. Bei diversen Downloadproblemen ist allerdings die Verwendung der allgemeineren Requests-Bibliothek erforderlich [2]. Requests beschäftigt sich nämlich mit den *POST*– und *GET*-Anfragen, die an HTTP-Server übermittelt werden. Anschließend kann die gewünschte Seite oder Datei mittels geeigneter Funktion heruntergeladen werden (**Abb. 1**).

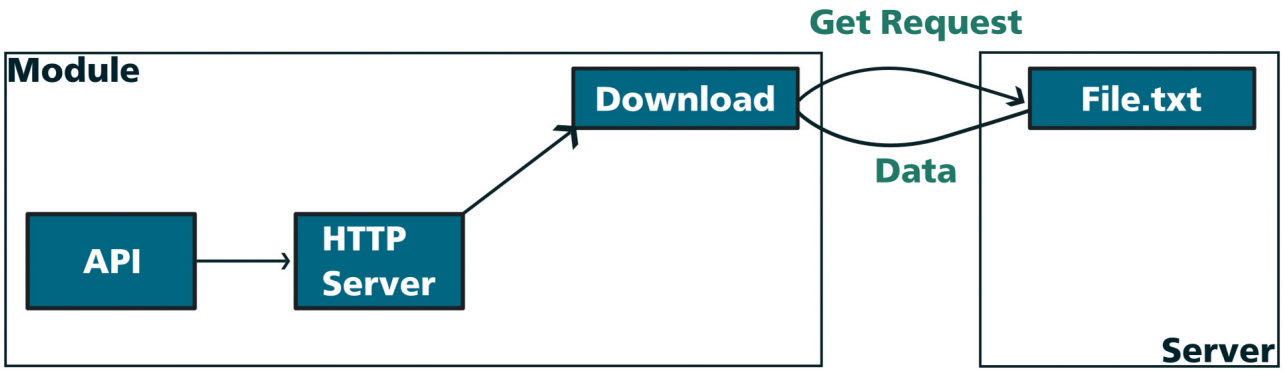


Abb. 1: Requests-Bibliothek

Alles in allem belegen diverse Statistiken, dass das Datenvolumen exponentiell wächst, womit es für Anwender unausweichlich wird, das Herunterladen von Daten beziehungsweise den Zugriff auf Daten zu automatisieren [3].

Soziale Medien

Üblicherweise platzieren Unternehmen beziehungsweise Organisationen Links zu den sozialen Netzwerken auf der eigenen Webseite, was vor allem auf die Kontaktseite zutrifft. So befinden sich die Icons der bekannten Anbieter von sozialen Medien eingebettet auf der Kontaktseite. Andererseits ist es üblich, die Icons im Footer einer x-beliebigen Seite der Organisation zu platzieren. Der dafür erforderliche HTML-Code, um das Icon eines Anbieters für soziale Medien auszugeben, könnte wie folgt aussehen:

```
<a                                        target="_blank"
href="https://www.facebook.com/GradeSaverLLC">
                <img        class="socialMedia__image"
src="/assets/footer/facebook-
beb8875d903a5a5d32d7d55667361d763ad2f0fb1c533b86afa19056eb4cbb
f8.png">
</a>
```

Sobald ein Anwender auf eines dieser Icons klickt, leitet der Browser den Anwender auf die Social-Media-Seite der Organisation weiter. Die Reichweite des Links umfasst dabei das komplette Social-Media-Icon.

Um Links aus Webseiten extrahieren zu können, eignet sich das Paket „Extract Social Media“ in der Version 0.4.0. Zusätzlich ist der Einsatz des Requests-Pakets erforderlich. Mit pip lassen sich die Pakete wie folgt installieren [4]:

```
pip install extract-social-media
pip install requests
```

Wird nun ein URL der selbstdefinierten Methode *get_links* übergeben, parst sie die Webseite nach möglichen Links und gibt diese aus (Listing 1).

Listing 1

```
import requests
```

```
from extract_social_media import find_links_tree
from html_to_etree import parse_html_bytes

def get_links(url):
  res = requests.get(url)
          tree     =     parse_html_bytes(res.content,
res.headers.get('content-type'))
  link_set = set(find_links_tree(tree))
  for s in link_set:
    print(s)

get_links('https://www.gradesaver.com/contact')
```

In der Methode *get_links* wird zunächst eine *GET*-Anfrage *requests.get* an die Webseite gesendet. Der daraufhin übermittelte Inhalt der Webseite wird in der Variable *res* gespeichert. Anschließend fischt die Methode *find_links_tree* die Links heraus und speichert sie in einem Set ab (**Abb. 2**).

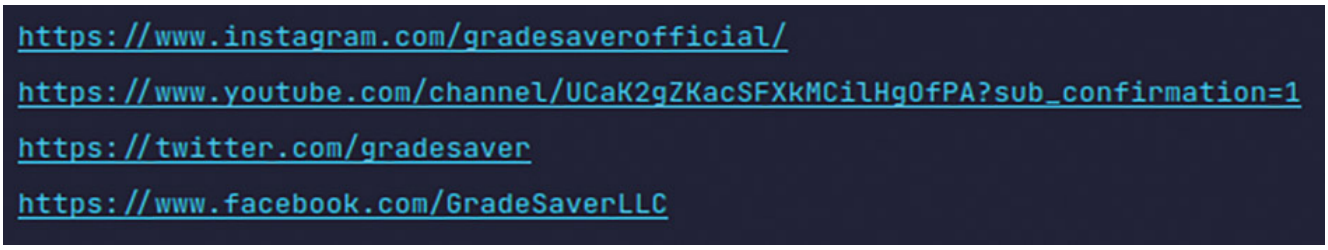


Abb. 2: Extrahierte Links einer Webseite

Dateidownload

Das Herunterladen von Dateien lässt sich unter Python in mehreren Schritten bewerkstelligen, wobei das Python-Skript davon ausgeht, dass die Links zu den Dateien bereits vorhanden sind. Vor allem die Auswertung von Dateien wird durch den automatisierten Download erleichtert, da sich heruntergeladene Text- oder CSV-Dateien im Anschluss in einen Dataframe importieren lassen. Für den Download mit Python werden die Importe aus Listing 2 gebraucht.

Listing 2

```
import numpy as np
```

```
import pandas as pd
import requests
import zipfile
import os
import glob
```

Das Python-Skript legt zunächst einen Ordner an, um die Dateien dort abzuspeichern. Der Ordner wird lediglich dann erstellt, wenn er noch nicht existiert:

```
def make_dir(folder):
  if not os.path.exists(folder):
    os.makedirs(folder)
```

Beim anschließenden Herunterladen der Dateien wird durch eine Liste mit Links *links* iteriert, wobei erneut von der Requests-Bibliothek in Version 2.28.2 Gebrauch gemacht wird (Listing 3). Um den Fortschritt des Downloads anzuzeigen, wird auf Python-Bordmittel zurückgegriffen. So wird der Index des aktuellen Links ermittelt, wobei die Anzeige zusätzlich die Gesamtzahl der Links zusammen mit der Antwort des Servers beinhaltet. In der Produktion sollte die *GET*-Anfrage *requests.get* unter Berücksichtigung eines Timeout ausgegeben werden. Dadurch wartet das Python-Skript lediglich für eine bestimmte Zeit in Sekunden auf eine Antwort des Servers, um anschließend weiterzumachen. Ansonsten besteht die Gefahr, dass sich das Programm aufhängt [5]. Mit den letzten zwei Zeilen werden die Dateien im Ordner abgespeichert (**Abb. 3**).

Listing 3

```
# downloads all zip files that are mentioned in the list:
def download(links, folder):
  size = len(links)
  for l in links:
    i = links.index(l)
    msg = '( '+str(i+1)+'/'+str(size)+' ) '+'downloading file
from: '+l
    response = requests.get(l,timeout=30)
    print(msg)
    print(response)
```

```
    print(response.elapsed)

      with open(os.path.join(folder, l.split('/')[-1]),
mode='wb') as file:
      file.write(response.content)
```

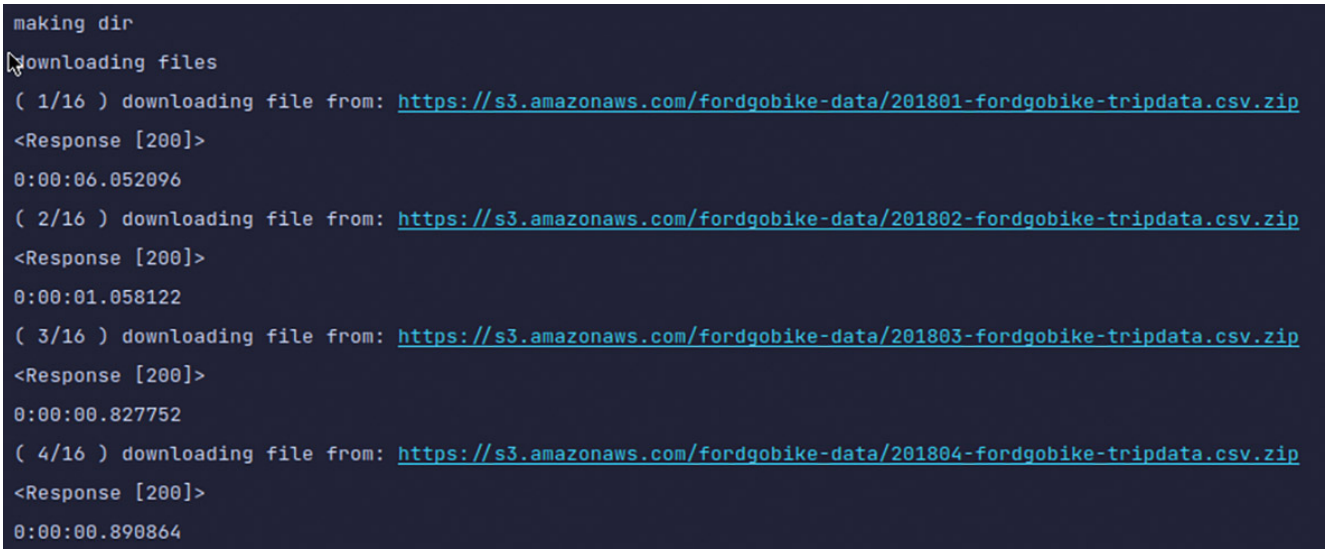


Abb. 3: Fortschrittsanzeige der Downloads

Das Programm könnte mit Listing 3 vorbei sein. Allerdings lässt sich das Programm noch weiter ausbauen, indem beispielsweise heruntergeladene ZIP-Archive automatisch entpackt werden (Listing 4). Die Methode *extract* erledigt genau das, indem sie zusätzlich die entpackten Dateien an einem beliebigen Ort auf der Festplatte ablegt.

Listing 4

```
# Extracts all contents from zip file
def extract(folder):
  all_files = glob.glob(folder + "/*.zip")
  archive = folder + '/' + working_path
  for f in all_files:
    with zipfile.ZipFile(f, 'r') as myzip:
      myzip.extractall(path=folder)
```

Handelt es sich bei den heruntergeladenen Dateien um Datenreihen, die beispielsweise als Textdatei oder im CSV-Format vorliegen, dann bietet es sich an, diese Daten in einem Dataframe zu importieren (Listing 5). So iteriert die Methode

merge durch alle Dateien vom Typ CSV. Danach werden die Datenreihen schrittweise in einen einzigen Dataframe geladen.

Listing 5

```
# merges all csv files into one dataframe
def merge(folder):
  all_files = glob.glob(folder + "/*.csv")
  li = []
  for filename in all_files:
    df = pd.read_csv(filename, index_col=None, header=0)
    li.append(df)
  fordgobike = pd.concat(li, axis=0, ignore_index=True)
  # displays the columns and their datatypes
  fordgobike.info()
```

Listing 6 beherbergt die Testdaten. So können Sie dort entnehmen, wie die Liste mit den Links definiert worden ist und wie sich die einzelnen Methoden aufrufen lassen (vgl. *test_files_download*).

Listing 6

```
folder_name = 'fordgobike'
working_path = 'archive'
# urls of zip files
urls                                                    =
['https://s3.amazonaws.com/fordgobike-data/201801-fordgobike-t
ripdata.csv.zip',
'https://s3.amazonaws.com/fordgobike-data/201802-fordgobike-tr
ipdata.csv.zip',
'https://s3.amazonaws.com/fordgobike-data/201803-fordgobike-tr
ipdata.csv.zip',
'https://s3.amazonaws.com/fordgobike-data/201804-fordgobike-tr
ipdata.csv.zip',
'https://s3.amazonaws.com/fordgobike-data/201805-fordgobike-tr
ipdata.csv.zip',
'https://s3.amazonaws.com/fordgobike-data/201806-fordgobike-tr
ipdata.csv.zip',
'https://s3.amazonaws.com/fordgobike-data/201807-fordgobike-tr
ipdata.csv.zip',
'https://s3.amazonaws.com/fordgobike-data/201808-fordgobike-tr
```

```
ipdata.csv.zip',
'https://s3.amazonaws.com/fordgobike-data/201809-fordgobike-tr
ipdata.csv.zip',
'https://s3.amazonaws.com/fordgobike-data/201810-fordgobike-tr
ipdata.csv.zip',
'https://s3.amazonaws.com/fordgobike-data/201811-fordgobike-tr
ipdata.csv.zip',
'https://s3.amazonaws.com/fordgobike-data/201812-fordgobike-tr
ipdata.csv.zip',
'https://s3.amazonaws.com/fordgobike-data/201901-fordgobike-tr
ipdata.csv.zip',
'https://s3.amazonaws.com/fordgobike-data/201902-fordgobike-tr
ipdata.csv.zip',
'https://s3.amazonaws.com/fordgobike-data/201903-fordgobike-tr
ipdata.csv.zip',
'https://s3.amazonaws.com/fordgobike-data/201904-fordgobike-tr
ipdata.csv.zip']
def test_files_download():
  print('making dir')
  make_dir(folder_name)
  print('downloading files')
  download(urls, folder_name)
  print('extracting files')
  extract(folder_name)
  print('merging files into to dataframe')
  merge(folder_name)
```

Vimeo-Bot

Vimeo ist ein beliebter Videodienst. Die gute Nachricht ist, dass es bereits eine Bibliothek gibt, die das Vimeo-API implementiert [6]. In Version 0.4.1 lässt sich die Bibliothek Vimeo Downloader wie folgt installieren:

```
pip install vimeo-downloader
```

Zusätzlich braucht Ihr Programm den folgenden Import, um auf die Methoden und Attribute des Vimeo-API zuzugreifen:

```
from vimeo_downloader import Vimeo
```

Um mehr über ein Video zu erfahren, können Sie zunächst die Metadaten eines Videos abrufen (Listing 7). Sobald ein Objekt vom Typ *Vimeo* durch Übergabe eines Links initialisiert wird, lassen sich der Titel sowie die Zahl der Likes und Views abrufen. Zusätzlich können Sie alle Kategorien des Metaobjekts anzeigen lassen (**Abb. 4**).

Listing 7

```
def print_info(url):
  v = Vimeo(url)
  meta = v.metadata
  print('Title:', meta.title)
  print('Number of likes: ', meta.likes)
  print('Number of views: ', meta.views)
  print(meta._fields)
  return v
```

```
Title: The Science Catalyst Live Show:Chemistry Experiment
Number of likes:  1
Number of views:  1242
('id', 'title', 'description', 'url', 'upload_date', 'thumbnail_small', 'thumbnail_medium', 'thumbnail_large', 'user_id', 'user_name', 'user_url', 'user_portrait_small', 'user_portrait_medium', 'user_portrait_large', 'user_por
```

Abb. 4: Metadaten eines Vimeo-Videos

Damit Sie nun einen Vimeo-Bot generieren können, ist es lohnenswert, den Bot auf eine Liste von Links anzuwenden. So initialisiert die Schleife jedes Mal das *Vimeo*-Objekt bei Übergabe des aktuellen Links. Dabei können Sie gegebenenfalls den Titel anpassen. So lassen sich beispielsweise alle Sonderzeichen sowie Leerzeichen mit geeigneten Methoden entfernen. Den Titel für einen Link rufen Sie zunächst über die Metadaten ab, indem Sie der Methode *get_title* das zuvor initialisierte Vimeo-Objekt übergeben:

```
def get_title(v):
  meta = v.metadata
  return meta.title
```

Danach können Sie den Titel eines Videos mittels der Methode *set_filename* anpassen (Listing 8). Bei dieser Methode werden alle Sonderzeichen einschließlich der Leerzeichen entfernt. Anschließend prüft die Methode die Länge des Strings und kürzt

ihn gegebenenfalls [7].

Listing 8

```
def set_filename(s):
  max_filename = 255
  normal_string = "".join(ch for ch in s if ch.isalnum())
  stripped_s = normal_string.strip()
  split_string = stripped_s[:max_filename]
  return split_string
```

Das eigentliche Herunterladen des Videos findet in der Methode *download* unter Übermittlung des *Vimeo*-Objekts, des *Download*-Ordners sowie eines neuen Dateinamens statt. Normalerweise beherbergt der Aufruf von *vimeo.streams* eine Liste voller Streams, die für den jeweiligen Titel verfügbar sind. Allerdings handelt es sich beim letzten Listenelement um den Stream mit der höchsten Auflösung:

```
def download(vim,path,filename):
  stream = vim.streams
  best_stream = stream[-1]
        best_stream.download(download_directory=path,
filename=filename)
```

Alternativ lässt sich über die Liste verfügbarer Streams iterieren, um den gewünschten Stream herauszufiltern. Sofern Sie beispielsweise einen Stream mit einer Auflösung von 720 herunterladen wollen, können Sie wie in Listing 9 vorgehen [8].

Listing 9

```
for s in stream:
  if s.quality == '720p':
              s.download(download_directory='video',
filename=v.metadata.title)
    break
  else:
    print('quality not found')
```

Die Methode *download_all* lädt schließlich alle Videos

herunter, indem auf die zuvor erwähnten Hilfsmethoden zurückgegriffen wird (Listing 10). Abgesehen davon, beherbergt diese Methode eine Fortschrittsanzeige, die den aktuellen Download samt Downloadraten ausgibt (**Abb. 5**).

Listing 10

```
vimeo_path = '/home/Videos/vimeo'

v_1 = 'https://vimeo.com/136491689'
v_2 = 'https://vimeo.com/509738789'
v_3 = 'https://vimeo.com/546042556'
v_4 = 'https://vimeo.com/58326464'
v_5 = 'https://vimeo.com/396053468'
v_6 = 'https://vimeo.com/560619626'
v_7 = 'https://vimeo.com/4510860'

videos = [v_1, v_2, v_3, v_4, v_5, v_6]

def download_all(path,urls):
  size = len(urls)
  for url in urls:
    v = Vimeo(url)
    title = get_title(v)
    new_title = set_filename(title)
    i = urls.index(url)
    print('Downloading video ('+str(i+1)+'/'+str(size)+')')
    download(v,path,new_title)
```

Abb. 5: Download von Vimeo-Videos

YouTube-Bot

Genauso wie für Vimeo existiert eine Bibliothek, die Zugriff auf das YouTube-API ermöglicht. Aktuell liegt pytube in der Version 12.1.2 vor und lässt sich wie folgt installieren [9]:

```
pip install pytube
```

Bei Bots können Sie abgesehen von Listen Dateien einlesen und durch diese zeilenweise iterieren, um aus dem aktuellen Link ein *youtube*-Objekt zu generieren. In der Regel verfügt ein YouTube-Link über etliche Streams, sodass es empfehlenswert ist, die Auswahl weiter einzuschränken. So lässt sich angeben, dass beispielsweise lediglich Streams mit der Dateiendung *.mp4* heruntergeladen werden. Die Einstellung *progressive=true* sorgt dafür, dass lediglich Streams ausgewählt werden, die sowohl eine Audio- als auch Videospur beinhalten. Die Anweisung *streams.order_by(‚resolution‘)* wählt schließlich unter allen noch verfügbaren Streams das Video mit der höchsten Auflösung aus. Abgesehen davon beinhaltet die *filter*-Methode weitere Parameter, die in Listing 11 jedoch nicht erscheinen [10].

Mit der Downloadmethode *stream.download()* aus dem YouTube-API wird der ausgewählte Stream heruntergeladen [11]. Dabei lässt sich der Download weiter einstellen, indem auf geeignete Parameter zurückgegriffen wird. So kann zusätzlich der Pfad angegeben werden. Der *Timeout*-Parameter hingegen gibt an, wie lange auf eine Antwort vom Server gewartet wird. Daneben lässt sich die Zahl der Downloadversuche weiter eingrenzen (**Abb. 6**).

Listing 11

```
def download_yt():
    link  =  open('/home/Videos/chemistry/links_file.txt',
mode='r')
  SAVE_PATH = '/home/Videos/chemistry'
  for i in link:
    try:
      youtubeObject = YouTube(i)
      d_video = youtubeObject.streams.filter(progressive=True,
file_extension='mp4').order_by('resolution').desc().first()
      print ('Stream: '+d_video)
      print('Downloading video: ' + d_video.title)
      d_video.download(SAVE_PATH, timeout=30, max_retries=3)
    except:
      print("An error has occurred")
  print("Download is completed successfully")
```

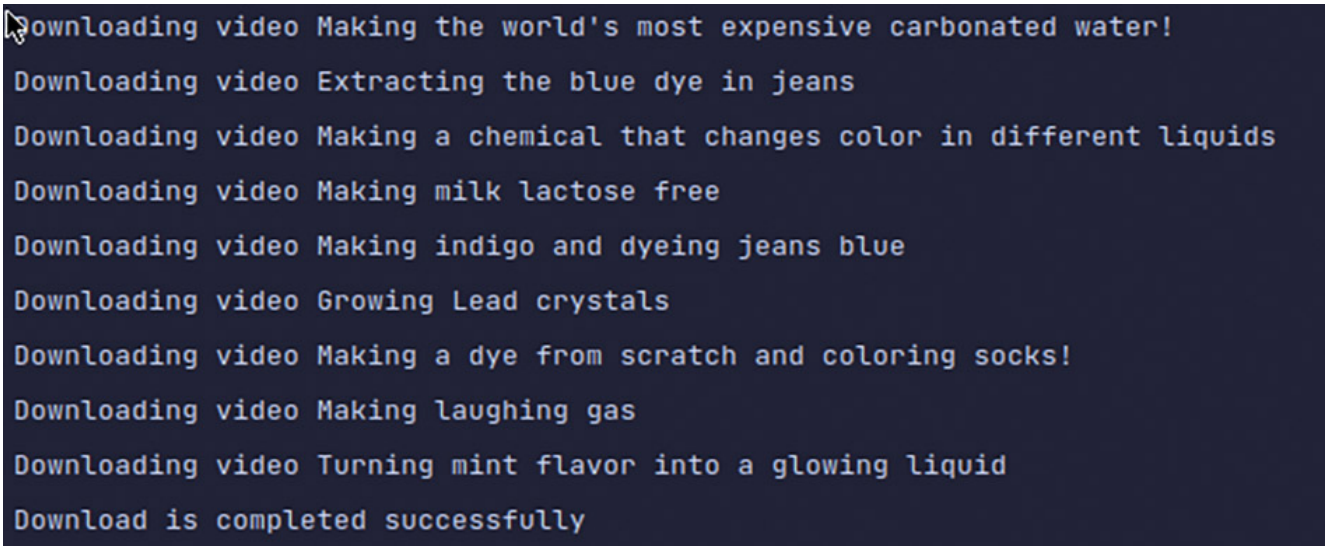


Abb. 6: Download von YouTube-Videos ohne Fortschrittsanzeige

Twitter-Bot

Twitter-Seiten erhalten eine Vielzahl an Informationen, angefangen von Tweets, Retweets, Followern, bis hin zu eingebetteten Videos etc. Dabei ermöglicht die Tweepy-Bibliothek 4.13.0 Zugriff auf das Twitter-API, sodass sich damit Etliches an Daten herunterladen lässt [12]. Um Zugriff auf das Twitter-API zu erhalten, brauchen Sie allerdings einen Entwickleraccount von Twitter. Anhand der folgenden Schritte können Sie einen Entwickleraccount bei Twitter beantragen:

- normales Benutzerkonto auf der Twitter-Seite erstellen [13]
- Benutzerkonto für Entwickler einrichten und sich für den erweiterten Zugang bewerben [14]
- Tokens, Secrets sowie Schlüssel generieren

Als Nächstes installieren Sie die Tweepy-Bibliothek:

```
pip install tweepy
```

Wenn Twitter Ihren Antrag genehmigt hat, sollten Sie über diverse Tokens, Schlüssel sowie Secrets verfügen. Sie können dafür sorgen, dass die Accountdaten griffbereit sind, indem Sie Ihre Zugangsdaten in einer Konfigurationsdatei speichern.

Diese Datei enthält Schlüssel-Wert-Paare und ist unter Python wie in **Abbildung 7** aufgebaut. Anschließend speichern Sie die Konfigurationsdatei mit der Dateiendung *.cfg* ab. Angenommen der Dateiname lautet *config.cfg*, dann lässt sich die Konfigurationsdatei wie in Listing 12 einlesen. Die Schlüssel-Wert-Paare eines Abschnitts (*section*) werden dabei in einem Dictionary abgelegt [15].

Listing 12

```
import configparser

def get_config_dict(section):
  config = configparser.RawConfigParser()
  config.read(r'config.cfg')
  if not hasattr(get_config_dict, 'config_dict'):
    get_config_dict.config_dict = dict(config.items(section))
  return get_config_dict.config_dict
```

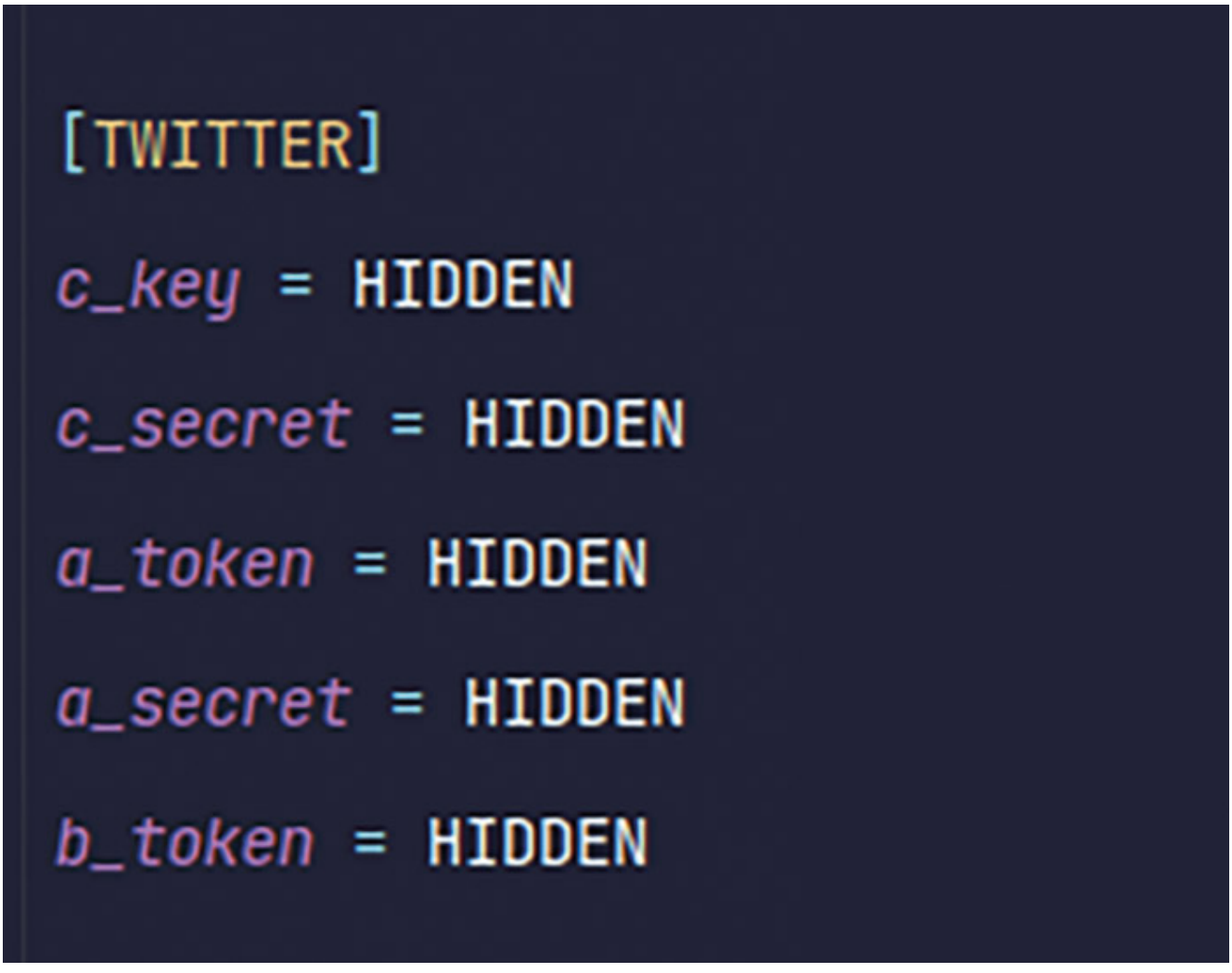


Abb. 7: Konfigurationsdatei unter Python

Um etwas bei Twitter automatisieren zu können, ist es erforderlich, sich erst einmal zu authentifizieren. So liest die Methode *create_api* zunächst die zuvor generierten Tokens, Schlüssel und Secrets unter Angabe des Abschnitts ein (Listing 13). Die eigentliche Authentifizierung beim Twitter-API erfolgt mittels der Methode *api.verify_credentials(),* wobei die zurückgegebene API-Variable für weitere Aktionen gebraucht wird [16].

Listing 13

```
import tweepy as tw
from tweepy import OAuthHandler

def create_api():
  config_details = get_config_dict('TWITTER')
  c_key = config_details['c_key']
  c_secret = config_details['c_secret']
  a_token = config_details['a_token']
  a_secret = config_details['a_secret']
  auth = OAuthHandler(c_key, c_secret)
  auth.set_access_token(a_token, a_secret)
  api = tw.API(auth, wait_on_rate_limit=True)
  try:
    api.verify_credentials()
  except Exception as e:
    print("Error creating API")
    raise e
  print("API created")
  return api
```

Bei Datenanalysen ist es relevant, Tweets zu einem bestimmten Hashtag zu analysieren. Die Suchergebnisse lassen sich dabei in einem Dataframe ablegen, um hinterher in der Lage zu sein, große Datenmengen zu analysieren. Bei dieser Suche erfolgt die Authentifizierung mit dem Bearer-Token (Inhabertoken), das ebenfalls über die Konfigurationsdatei abrufbar ist. So nimmt die Methode *get_tweets* zunächst die Authentifizierung vor

(Listing 14). Danach sucht die Methode *client.search_recent_tweets* nach Tweets der letzten sieben Tage, indem zuvor die Suchanfrage *query* definiert wird. Die Suchanfrage beinhaltet neben dem Hashtag auch die Sprache der in Frage kommenden Tweets, wobei die Suche lediglich aus *#<Suchwort>* bestehen kann. Dabei wird die maximale Zahl an Suchergebnissen auf *100* begrenzt. Außerdem lässt sich in der Suchmethode *client.search_recent_tweets* definieren, welche Felder eines Tweets gespeichert werden (**Abb. 8**). Zusätzlich lassen sich die Tweets in einem Dataframe ablegen [17].

Listing 14

```
def get_tweets(search):
  config_details = get_config_dict('TWITTER')
  token = config_details['b_token']
  client = tw.Client(bearer_token=token)
  # What to search for
  query = '#'+search+' -is:retweet lang:de'
  max_results = 100
  # We grab tweets + context annotations + create date + user name of tweeter
    tweets = client.search_recent_tweets(query=query, tweet_fields=['context_annotations', 'created_at'], user_fields=['name'], expansions='author_id', max_results=max_results)
  for tweet in tweets.data:
    print(tweet.text)
    if len(tweet.context_annotations) > 0:
      print(tweet.context_annotations)
  # Return the tweet data as a dataframe
  df2 = pd.DataFrame(tweets.data).astype(str)
  # Collect the user IDs
  df_users = pd.DataFrame(tweets.includes['users'])
```

Abb. 8: Tweets zum Hashtag #Mavropanos zusammen mit Annotationen

Abgesehen von Tweets basierend auf Hashtags ist es möglich, alle möglichen Tweets eines Twitter-Accounts herunterzuladen. Hierfür wird angenommen, dass die Tweet-ID eines Tweets zusammen mit weiteren Kategorien wie *Timestamp*, *Text* etc. in einer CSV-Datei gespeichert ist (**Abb. 9**). Dieser Twitter-Bot ist dann in der Lage, die Tweets von einem Account anhand der ID zu identifizieren und herunterzuladen.

Abb. 9: CSV-Datei zusammen mit den Tweet-IDs eines Twitter-Accounts

Die selbstdefinierte Methode *save_data_to_file* ermöglicht unter Angabe des API, dem Pfad zum Twitter-Archiv sowie dem Zielpfad für die Tweets das Herunterladen von Tweets zu automatisieren (Listing 15). Nach der Authentifizierung wird zunächst ein Twitter-Archiv namens *twitter-archive-enhanced.csv* eingelesen. Anschließend werden vom zuvor erzeugten Dataframe die Tweet-IDs extrahiert und in einer Liste abgelegt. Durch diese Liste wird anschließend iteriert, wobei pro Tweet eine JSON-Datei heruntergeladen wird. Die heruntergeladenen JSON-Dateien landen später alle in der Textdatei *tweet_json.txt*. Anhand einer selbstdefinierten Fortschrittsanzeige weiß der Benutzer sofort Bescheid, welche

Tweets sich herunterladen lassen und bei welchen Tweets der Download scheitert (**Abb. 10**). Besteht der Download hingegen aus mehreren Tausend Tweets, kommt das Rate Limit von Twitter zum Tragen und der Download wird in bestimmten Zeitintervallen für einige Minuten unterbrochen. Dadurch wird der Download einer großen Menge Tweets in die Länge gezogen.

Listing 15

```
import tweepy as tw
from tweepy import OAuthHandler
import json
from timeit import default_timer as timer
import pandas as pd

def save_data_to_file(api, sFrom, sTo):
  df = pd.read_csv(sFrom)
  tweet_ids = df.tweet_id.values
  len(tweet_ids)
  count = 0
  fails_dict = {}
  start = timer()
  # Save each tweet's returned JSON as a new line in a .txt file
  with open(sTo, 'w') as outfile:
    # This loop will likely take 20-30 minutes to run because of Twitter's rate limit
    for tweet_id in tweet_ids:
      count += 1
      print(str(count) + ": " + str(tweet_id))
      try:
                    tweet  =  api.get_status(tweet_id, tweet_mode='extended')
        print("Success")
        json.dump(tweet._json, outfile)
        outfile.write('\n')
      except tw.errors.TweepyException as e:
        print("Fail")
        fails_dict[tweet_id] = e
        pass
  end = timer()
```

```
print(end - start)
print(fails_dict)
```

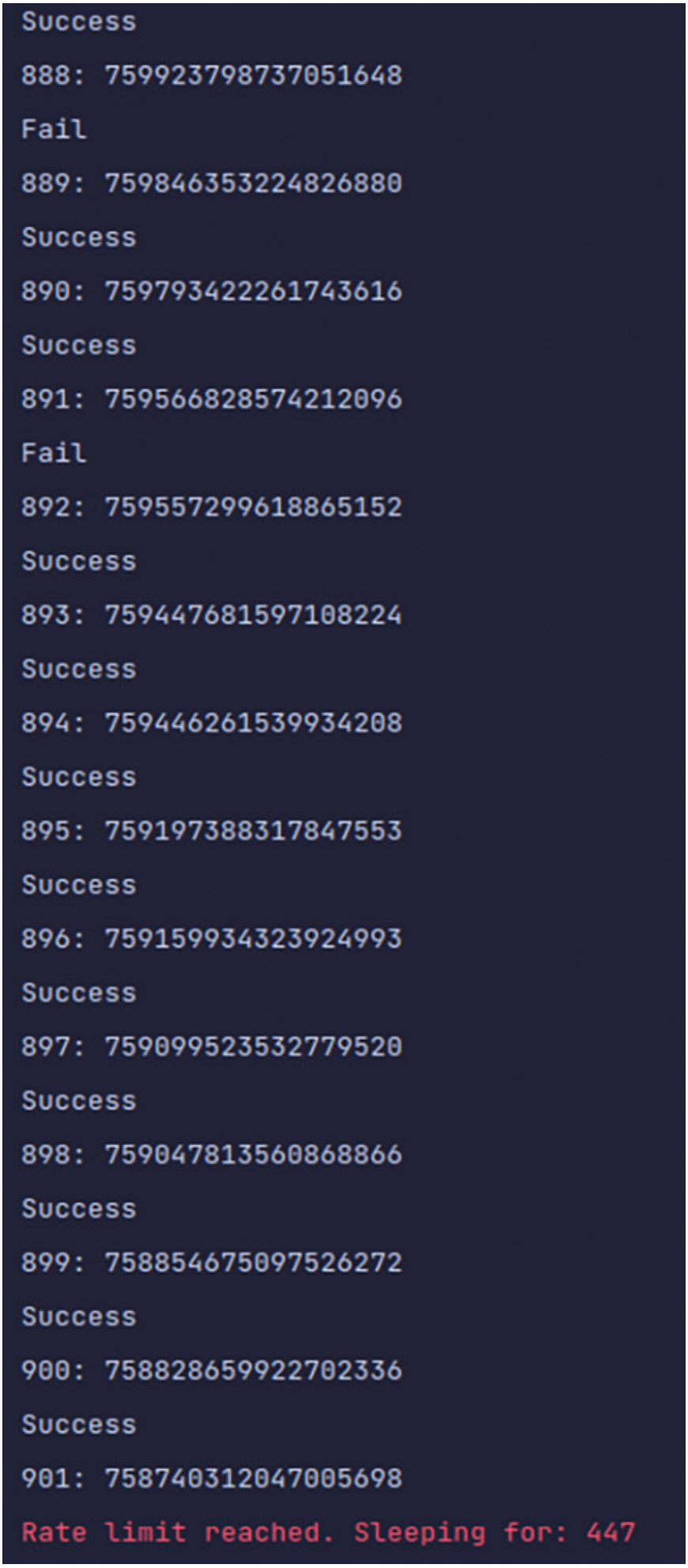


Abb. 10: Download von

Tweets

Abgesehen vom Schreiben bietet Anzela Minosi Dienstleistungen auf Legiit.com an. Dort erstellt Anzela Datenanalysen, Datenbanksoftware, Python-Skripte sowie Kommandozeilentools für den Raspberry Pi. Bevor Anzela sich selbständig gemacht hat, war sie zehn Jahre lang in den Automobil-, Bildungs- und Telekommunikationsbranchen tätig, wo sie diverse IT-Tätigkeiten ausübte: Support, Softwaretests sowie Webentwicklung. Anzela verbringt ihre Freizeit gerne an der ligurischen Küste, fährt Fahrrad und spielt Retrospiele auf dem Raspberry Pi. Sie steht für Redaktionsprojekte sowie für persönliche Beratungsgespräche zur Verfügung.

Links & Literatur

[1] PyPi: https://pypi.org/

[2] Requests: https://www.geeksforgeeks.org/get-post-requests-using-python/

[3] Datenvolumen: https://firstsiteguide.com/big-data-stats/

[4] Extract Social Media: https://pypi.org/project/extract-social-media/

[5] Requests: https://requests.readthedocs.io/en/latest/user/quickstart/

[6] Vimeo: https://pypi.org/project/vimeo-downloader/

[7] Sonderzeichen in Strings:

https://www.scaler.com/topics/remove-special-characters-from-string-python/

[8] Vimeo-Download: https://jakeroid.com/blog/how-to-download-vimeo-video-using-python/

[9] pytube: https://pytube.io/en/latest/

[10] Streams: https://pytube.io/en/latest/user/streams.html

[11] YouTube-Download: https://www.geeksforgeeks.org/pytube-python-library-download-youtube-videos/

[12] Tweepy: https://pypi.org/project/tweepy/

[13] Twitter: https://twitter.com

[14] Entwicklerportal: https://developer.twitter.com/en/support/twitter-api/developer-account

[15] Konfigurationsdatei: https://stackoverflow.com/questions/19379120/how-to-read-a-config-file-using-python

[16] Twitter-Bot: https://realpython.com/twitter-bot-python-tweepy

[17] Tweet-Suche: https://towardsdatascience.com/how-to-access-data-from-the-twitter-api-using-tweepy-python-e2d9e4d54978

Python im Web

Python im Web

Dynamisches HTML im Browser mit PyScript statt JavaScript

Browser führen nur JavaScript aus? Nicht mehr! PyScript tritt als Alternative zu JavaScript auf. Wir erklären, wie das möglich ist, und programmieren als Beispiel ein Spiel mit der neuen Technik.

Von Pina Merkert

kompakt

- PyScript nutzt die maschinennahe Sprache WebAssembly, um Python mit der JavaScript-Engine eines Browsers auszuführen.
- Über PyScript-Funktionen kann der Python-Code auf das DOM zugreifen, was dynamisches HTML ermöglicht.
- PyScript kann JavaScript ersetzen, es lädt aber wesentlich langsamer.

Beim Versuch, auf eine Objekteigenschaft zuzugreifen, wird die Eigenschaft nicht nur in dem Objekt selbst, sondern auch in seinem Prototyp und dem Prototyp des Prototyps gesucht. Wenn Ihnen Sätze wie dieser auch rätselhaft vorkommen, ist JavaScript wohl auch nicht Ihre Muttersprache. Python ist da oft zugänglicher, der Code hat weniger Zeilen und das Sprachdesign ist auf Lesbarkeit optimiert. Nur leider muss Python immer lokal installiert sein, die allgegenwärtigen Webbrowser verarbeiten nur JavaScript. Oder etwa nicht?

Dass Python nicht im Browser läuft, stimmt nicht mehr: Mit einer trickreichen Software namens „Pyodide“ interpretieren Webseiten auch Python-Code. Python-Entwickler können auf diesem Weg JavaScript durch Python ersetzen. Sie programmieren dann in Python mit PyScript-Funktionen statt in JavaScript. Wir zeigen, wie Sie mit PyScript loslegen.

Als Beispiel haben wir uns von dem Worträtsel „Wordle“ inspirieren lassen und ein Rätsel für Nerds mit dem Namen „Nerdle“ programmiert. Die Spielregeln sind einfach: PyScript wählt aus einer langen Liste mit Begriffen aus der Technikwelt mit fünf Zeichen (Nerdle ist schwieriger als Wordle, weil unser Rätsel Wörter mit Ziffern und Sonderzeichen enthält) einen zufälligen aus, den Sie erraten müssen. Dafür tippen Sie über die Bildschirmtastatur zunächst einen Begriff. Jeder geratene Begriff muss in der Liste stehen, damit das Spiel die Eingabe akzeptiert. Wenn ein Zeichen des geratenen Begriffs an der gleichen Stelle steht wie im gesuchten Begriff, wird es grün. Kommt ein Zeichen irgendwo im gesuchten Begriff vor, wird es gelb. Zeichen, die gar nicht vorkommen, werden grau. Die Tasten der Bildschirmtastatur verfärben sich genauso. Diese Farben geben Hinweise für den nächsten Begriff, sodass Sie mit zusätzlichen Versuchen immer mehr über den gesuchten Begriff erfahren. Wenn Sie spätestens beim sechsten Versuch richtig raten, gewinnen Sie das Spiel. Wenn Sie zu oft falsch raten, sollten Sie mehr c’t lesen *zwinkersmiley*. Ohne selbst zu programmieren, können Sie das Spiel sofort unter nerdle.pinae.net ausprobieren; den Code finden Sie als Open Source (GPLv3) über ct.de/ynk9.

WebAssembly

Browser integrieren weiterhin keinen Python-Interpreter. Sie führen aber in ihrer virtuellen Maschine schon länger WebAssembly aus. WebAssembly ist eine maschinennahe Sprache,

die der Browser innerhalb von Millisekunden in Maschinencode übersetzt. Das funktioniert so gut, dass WebAssembly meist nur wenige Prozent langsamer läuft als gleicher Code, den ein Compiler direkt in Maschinencode übersetzt hat. Da der Code aber in der Browser-Sandbox läuft, ist nicht jedes Programm WebAssembly-tauglich. Beispielsweise verbieten Browser direkten Zugriff aufs Dateisystem oder Hardware wie Netzwerkkarten.

Das Pyodide-Projekt hat die Python-Referenzimplementierung CPython so modifiziert, dass sie nach WebAssembly kompiliert. Damit läuft Python zwar im Browser, man sieht davon aber noch nichts. An dieser Stelle kommt PyScript ins Spiel: PyScript bringt Funktionen mit, um vom Browser-Python auf den DOM-Tree zuzugreifen, also auf die HTML-Struktur der Webseite. Außerdem stellt es eigene HTML-Tags bereit, die den Code aufnehmen, Module nachladen und Eingabefelder bereitstellen. Das Projekt steht noch am Anfang: Release-Nummern sind im Datumsformat, die Versionen auf GitHub als „Pre-release“ getaggt. Manchen Funktionen sieht man den frühen Entwicklungsstand noch an. Beispielsweise muss man per Hand Funktionen einkapseln, um sie ins Ereignissystem von JavaScript einzuklinken. Trotzdem funktioniert der Code bereits gut genug für ein Browserspiel.

Einbinden

PyScript bindet man wie ein JavaScript-Framework ein, indem man zwei Tags im <head> der Webseite ergänzt:

```
<link rel="stylesheet"
    href="https://pyscript.net/alpha/pyscript.css" />
<script                                          defer
src="https://pyscript.net/alpha/pyscript.js"></script>
```

Den Code lädt der Browser dann aus dem Content Delivery Network (CDN) der PyScript-Entwickler, man bekommt also immer die aktuellste Version. Um Probleme bei Updates zu umgehen, könnte man PyScript auch selbst übersetzen und hosten. Momentan raten wir aber noch davon ab, PyScript produktiv

einzusetzen, weil sich in den kommenden Monaten sicherlich noch einiges an den Funktionssignaturen ändern kann.

Danach kann man einfach irgendwo im HTML der Seite den <py-script>-Tag einfügen und in dem Python-Code platzieren. Für ein Hallo-Welt-Programm braucht man nur drei Zeilen:

```
<py-script>
  print("Hallo Welt.")
</py-script>
```

Module

Python funktioniert im Browser wie gewohnt. Das bezieht sich auch auf Module, die man wie üblich mit import ins Programm einbindet:

```
import random
wordlist = ["CT.DE", "RULEZ"]
word = random.choice(wordlist)
```

Da dem Browser der Python-Paketmanager pip fehlt, packt man externe Module mit Spiegelstrichen in den <py-env>-Tag und PyScript kümmert sich ums Nachladen:

```
<py-env>
  - numpy
  - matplotlib
</py-env>
```

Es stehen schon einige beliebte Module wie numpy und matplotlib zur Verfügung, solche mit Hardwarezugriff wie requests können aber nicht funktionieren. Gibt man ein Modul im <py-env>-Tag an, muss man es trotzdem im Code mit einem import einbinden.

Wortlisten per Ajax asynchron laden

Damit wir die Begriffe zentral verwalten können, wollten wir sie nicht als ellenlange Liste hardcoden, sondern lieber mit Ajax nachladen. In normalem Python-Code würde man für so etwas

eine Bibliothek wie requests nehmen. Die läuft aber im Browser nicht. Der kann jedoch von sich aus Daten laden, was die Funktion pyfetch() aus dem pyodide-Modul anstößt.

Ajax-Anfragen sind von Natur aus asynchron, weshalb pyfetch() nur in asynchronen Funktionen funktioniert. Die erzeugt man mit async def statt def. Da sie dem normalen Code nicht im Weg stehen, kann man in so einer Funktion problemlos mit await auf Antworten warten, ohne das ganze Programm auszubremsen:

```
from pyodide.http import pyfetch
from pyodide import JsException
from js import console

async def load_wordlist():
  try:
    response = await pyfetch(
url="https://raw.githubusercontent.com/pinae/Nerdle/main/nerdl
e-begriffe.txt",
      method="GET",
      headers={"Content-Type":
               "text/plain"})
    if response.ok:
      data = await response.string()
      console.log(data.split())
      return data.split()
  except JsException:
    return None
```

Die JsException ist die Basisklasse aller JavaScript-Fehler, die PyScript automatisch kapselt. Man könnte hier auch spezifische Exceptions fangen, um aussagekräftige Fehlermeldungen anzuzeigen.

Das Speichern der Liste und das Auswählen des zufälligen Worts übernimmt die Funktion pick_word(). Auch sie ist asynchron, weil sie mit await auf die Rückgabe von load_wordlist() warten muss:

```
async def pick_word():
  global wordlist, word, accept_input
```

```
  wordlist = await load_wordlist()
  word = random.choice(wordlist)
  console.log("Wort: ",
              " ".join(list(word)))
  accept_input = True
```

Um die asynchronen Funktionen so aufzurufen, dass sie den Code nicht blockieren, kann man einen alten JavaScript-Trick benutzen:

```
setTimeout(create_proxy(pick_word), 0)
```

Der Timeout von 0 zwingt den Code nicht zum Warten, bei 0 Millisekunden Verzögerung gibt es aber auch keine Wartezeit. Die Timeout-Funktion sorgt dabei automatisch für eine nebenläufige Ausführung.

Nerdle-HTML

Nerdle benutzt ein Spielbrett aus 30 Quadraten (6 Zeilen mit je 5), die je ein Zeichen aufnehmen. Das geht hervorragend mit einem Grid-Layout. Und da alle Felder gleich aussehen, kann man sie bequem im Quellcode erzeugen. Der fügt alle hintereinander in ein und merkt sich die Divs in einer Liste aus Listen (eine pro Zeile):

```
tiles=[]
board=document.getElementById("board")
for row in range(6):
  tiles.append([])
  for col in range(5):
    tile=document.createElement("div")
    board.appendChild(tile)
    tiles[-1].append(tile)
```

Form und Farbe legt die Datei nerdle-styles.css fest, die Sie zusammen mit dem Rest des Codes im Repository über ct.de/ynk9 finden.

Die Funktionen getElementById() und createElement() funktionieren genau wie in JavaScript, sodass Sie dort die

Dokumentation konsultieren können, solange PyScript noch keine eigene dafür hat. Die Funktionen gehören zum document-Objekt, das Sie mit from js import document laden.

Die Tasten der Bildschirmtastatur sind ganz ähnliche , allerdings von Anfang an im HTML. Es gibt einen Unterschied: Das umrahmende hat die ID "keyboard". Der Python-Code kann sich anhand der Beschriftung der Tasten dann selbst ein Dictionary zusammenbauen, das jedem Zeichen das passende DOM-Objekt zuordnet:

```
key_objects = {}
for row in document.getElementById(
               "keyboard").childNodes:
  for key in row.childNodes:
    key.addEventListener("click",
                         js_key_clicked)
    if len(key.textContent) == 1:
      key_objects[
                key.textContent] = key
```

childNodes ist dabei die JavaScript-Datenstruktur NodeList, die aber das Iterator-Interface implementiert, sodass sich damit fast wie mit einer Python-Liste arbeiten lässt. addEventListener() ist die von Python aufrufbare JavaScript-Funktion, die den JavaScript-Function-Pointer js_key_clicked annimmt. Den muss man allerdings etwas umständlich erzeugen.

Verpackte Funktionen

Funktionen definiert man im Python-Code wie üblich mit def. Heraus kommt dabei eine Python-Funktion, die im Python-Code ganz normal funktioniert. Will man aber mit dem DOM interagieren, muss man die von PyScript gekapselten JavaScript-Funktionen benutzen, die man an den Namen in CamelCase erkennt. Diese JavaScript-Funktionen sehen die Python-Funktionen nicht, weshalb man eine Funktionsreferenz beispielsweise nicht einfach an addEventListener() übergeben kann.

Die Sprachbarriere überwindet das pyodide-Modul, das PyScript standardmäßig mitbringt (kein Eintrag in <py-env> nötig).

```
from pyodide import create_proxy
def key_clicked(e):
  print(e.target.textContent)

js_key_clicked = create_proxy(
                           key_clicked)
```

Mit create_proxy() packt man die Python-Funktion so ein, dass JavaScript sie sehen kann. Die so erzeugte Referenz kann man wie eine JavaScript-Funktion an addEventListener() übergeben.

Dabei funktionieren wiederum alle von JavaScript bekannten Datenstrukturen, sodass die Python-Funktion über den Parameter e das Event-Objekt bekommt. Das verweist unter e.target auf das DOM-Objekt, von dem das Ereignis ausging, und das verrät mit e.target.textContent, was innerhalb des HTML-Tags steht.

Noch ein Hinweis auf eine mögliche Fehlerquelle beim Konvertieren von JavaScript-Code von StackOverflow: Die Python-Funktion muss alle Parameter nennen, die JavaScript übergibt. JavaScript erlaubt es, Parameter still und heimlich wegzulassen, während Python mindestens einen _ verlangt. Man muss die übergebenen Parameter in der Funktion aber nicht benutzen, wenn man sie nicht braucht.

Raten

Nachdem der Code schon ein Wort zum Erraten ausgewählt und das Dictionary mit den Tasten initialisiert ist, muss er nur noch die Eingaben verarbeiten und bei „Enter“ den geratenen Begriff auswerten.

Die aktuelle Eingabe speichert das Programm in der Variable guess. Fürs Verarbeiten der Eingaben macht sich die Funktion key_clicked() den Umstand zunutze, dass alle mit einem Zeichen einen textContent mit Länge 1 haben. Das Backspace-

Symbol ist ein SVG, weshalb textContent bei dieser Taste ein leerer String ist. Die Enter-Taste dagegen ist mit „Enter“ beschriftet, also mit fünf Zeichen. Die Fallunterscheidung ist eine gute Gelegenheit, das mit Python 3.10 eingeführte Pattern-Matching einzusetzen:

```
match len(e.target.textContent):
  case 0:
    guess = guess[:-1]
  case 1:
    guess += e.target.textContent
  case _:
    check_enter()
display_guess()
```

match…case funktioniert so ähnlich wie switch…case in C, erlaubt aber beispielsweise auch reguläre Ausdrücke hinter case. Statt default: gibt es case _:, der zum Tragen kommt, wenn keiner der anderen Fälle eintrifft. In allen Fällen kümmert sie die Funktion display_guess() darum, den Inhalt von guess auch anzuzeigen. Pattern Matching funktioniert auch mit Objekten, beispielsweise mit re (das Modul für reguläre Ausdrücke) erzeugte Tupel. Ein Beispiel dafür finden Sie im Code auf GitHub.

Die Funktion display_guess() konsultiert die Variable guess_no, die speichert, in welcher Zeile Spieler gerade raten. Zuerst füllt die Funktion überall Leerzeichen ein, um vorherige Eingaben zu löschen:

```
for i in range(5):
  tiles[guess_no][i].textContent = ""
```

Das Eintragen aller Buchstaben aus guess geht fast genauso einfach, weil Python klaglos mit list() Strings in Listen aus Einzelzeichen verwandelt:

```
for pos, c in enumerate(list(guess)):
  tiles[guess_no][pos].textContent = c
```

Python kann Tupel automatisch auspacken, wenn man der Anzahl

entsprechend viele Variablen mit Komma getrennt hintereinander schreibt. enumerate() gibt für jeden Iterator, also für alles, was sich wie eine Liste behandeln lässt, ein 2-Tupel aus der Nummer und dem Element zurück. Im Beispiel landet in pos also die Nummer des Zeichens und in char das Zeichen. Die Schreibweise, die dabei herauskommt, versteht man ganz intuitiv.

Vergleichen

Bei einem „Enter“ muss das Spiel zunächst prüfen, ob der geratene Begriff fünf Zeichen hat und ob er in der Wortliste steht. Wenn nicht, schreibt die Funktion in das mit der ID "info" eine Fehlermeldung:

```
pyscript.write("info", f"„{guess}“ " +
      "steht nicht in der Wortliste.")
```

Die Funktion pyscript.write() nimmt einem dabei die Arbeit ab, das DOM-Element mit der ID "info" herauszusuchen und seinen textContent zu ändern. Wir vermuten, dass PyScript in Zukunft noch weitere Funktionen ähnlicher Art bekommt, die DOM-Zugriffe mit weniger Code erlauben.

Außerdem nutzt die Zeile Pythons seit 3.6 verfügbare f-Strings. Die definiert man mit dem Buchstaben f vor den Anführungszeichen und Python ersetzt dann alle in geschweiften Klammern angegebenen Variablen durch deren Werte. Man kann die Ausgabe mit den gleichen Filtern wie in format() beeinflussen.

Steht in guess ein Begriff aus der Wortliste, vergleicht die Funktion evaluate_guess() den geratenen Begriff mit word, dem zu erratenden Begriff. Dafür macht sie den Begriff wieder zu einer Liste, holt sich mit enumerate() die Zeichennummer dazu, die sie dann benutzt, um das Zeichen aus dem zu erratenden Begriff zu ziehen und die Zeichen zu vergleichen:

```
for p, char in enumerate(list(guess)):
  if char == word[p]:
```

```
    tiles[guess_no][p].classList.add(
                            "nailedit")
    key_objects[char].classList.add(
                            "nailedit")
    correct_counter += 1
```

Damit Spieler Grün und Gelb sehen, ergänzt die Funktion über classList.add() die passende CSS-Klasse, die die Farbe festlegt. Das passiert sowohl bei den Quadraten im Spielfeld als auch bei der Bildschirmtastatur. Dass die Tasten ihre Farbe verändern, ist eine willkommene Hilfe beim Sinnieren über den nächsten Begriff, den man raten könnte.

Bei richtig geratenen Zeichen zählt die Funktion auch die lokale Variable correct_counter hoch. Steigt die nämlich beim Einfärben aller Zeichen-Quadrate auf 5, ist das Ratespiel gewonnen. Ist die Variable kleiner und zusätzlich guess_no >= 5, ist das Spiel verloren.

Usability

Der vollständige Code, den Sie über das Repository über ct.de/ynk9 finden, enthält noch einige zusätzliche Zeilen. Die dienen der besseren Bedienbarkeit, damit man beispielsweise statt per Bildschirmtastatur auch mit der normalen Tastatur tippen kann. Das geht mit einer key_down()-Funktion, die den onkeydown-Event-Handler von document überschreibt:

```
js_key_down = create_proxy(key_down)
document.onkeydown = js_key_down
```

Der Code ist mitsamt HTML kaum mehr als 200 Zeilen lang, die bis auf einen regulären Ausdruck leicht zu lesen sind.

Exceptions sind in der Entwicklerkonsole sichtbar, die Browser mit F12 öffnen. Die Darstellung ist leider nicht so übersichtlich, weil PyScript alles in JavaScript-Objekte einpacken muss.
Fürs Debugging lohnt es sich, mit F12 die Entwickler-Konsole zu öffnen. Dort landet auch alles, was Sie mit console.log() ausgeben. Die Fehlermeldungen sind leider nicht gut lesbar, weil es in JavaScript-Exceptions verpackte Python-Exceptions sind. Das müssen die Entwickler noch stark nachbessern.

Eine weitere Baustelle sind die Entwicklungsumgebungen. Beispielsweise erkannte die Entwicklungsumgebung PyCharm den Code in <py-script>-Tags nicht als Python-Quelltext, sodass

weder Farben noch Code-Vervollständigung funktionierten.

Spielen

Das selbst programmierte Nerdle starten Sie, indem Sie einfach die Datei index.html im Browser öffnen. Ein Webserver ist dafür nicht notwendig, zum Nachladen des PyScript-Codes und der Wortliste aber eine Internetverbindung. Installieren müssen Sie gar nichts. Falls Sie das Spiel doch mit einem Webserver hosten wollen, muss der nur eine statische Seite ausliefern können.

Unser Worträtsel „Nerdle“ ist wegen der Tech-Begriffe mit Abkürzungen deutlich schwerer als „Wordle“ von der New York Times.
Unsere Liste mit Begriffen finden Sie im Repository in der Datei nerdle-begriffe.txt. Momentan stehen da schon mehr als 1500 Begriffe zum Raten bereit. Mit dem Skript word_list_filter.py können Sie die Liste aber bequem erweitern. Es liest die Datei neue-begriffe.txt, filtert nach den richtigen Zeichen und fragt für alle neuen Begriffe einzeln, ob Sie die hinzufügen wollen. So können Sie mit wenig Arbeit ganze Listen aus Kreuzworträtsel-Datenbanken ergänzen.

Kritik

Wir sind von PyScript begeistert. Python-Code ohne Installation im Browser ausprobieren? Grandios!

Noch ist PyScript aber nicht an dem Punkt angelangt, darauf bestehenden Code zu portieren und über Experimente hinausgehende Projekte damit umsetzen zu wollen. Außerdem muss eine Webseite mit PyScript mehr als 14 Megabyte an Code laden, bevor sie überhaupt starten kann. Danach muss die JavaScript-Engine den WebAssembly-Code zunächst in Bytecode für die Prozessorarchitektur übersetzen, was selbst auf einer schnellen Desktop-CPU fast eine Sekunde dauert. Erst danach läuft der Code der Seite los. Bei normalem JavaScript lädt der Browser nur die winzige Skriptdatei und führt diese sofort aus, weil die JavaScript-Engine längst warm gelaufen ist.

PyScript hat trotz allem sinnvolle Anwendungen: Hat beispielsweise eine Statistikerin ihre Daten schon mit Python ausgewertet und mit Matplotlib ein Diagramm gezeichnet, müsste sie normalerweise alles in JavaScript nachprogrammieren, um das Diagramm in eine Webseite einzubinden. PyScript senkt die Hürde für Pythonisten, mal eben schnell aus einer Idee eine Webanwendung zu basteln – wie unser Beispiel zeigt. *(pmk@ct.de)*

Code bei GitHub, PyScript-Dokumentation: ct.de/ynk9

RESTful APIs mit Python und Flask entwickeln

RESTful APIs mit Python und Flask entwickeln

[expand title="mehr lesen…"]

RESTful APIs mit Python und Flask entwickeln

Kommunikationshelfer

Sebastian Bindick

Microservices kommunizieren über standardisierte APIs. Die Python-Frameworks Flask und Flask-RESTPlus ermöglichen Entwicklern, REST-APIs einfach zu erstellen.

-tract

- Unternehmen setzen verstärkt Microservices ein, die über APIs in Kontakt stehen.
- Für die Koordination der einzelnen Dienste untereinander verwenden Entwickler REST.

- Mit Flask und seiner Erweiterung Flask-RESTPlus lassen sich REST-APIs für Enterprise-Anwendungen entwickeln.
- Ein Anwendungsbeispiel zeigt die Funktionsweise von RESTful Webservices mit Flask-RESTPlus: Aufbau und Struktur der Anwendung, Ressourcen und Routing, Validieren von Modellen, Fehlerbehandlung, API-Testing, Umgebungen sowie Dokumentation.

Qualitativ hochwertige APIs sind so wichtig wie noch nie, denn immer mehr Unternehmen setzen auf Microservices, die über Schnittstellen miteinander kommunizieren. Das Zusammenspiel der einzelnen Dienste bestimmt dabei maßgeblich die Qualität des gesamten Systems. Für diesen Zweck nutzen Softwareentwickler mittlerweile vorwiegend den Architekturstil REST (Representational State Transfer). Dieser orientiert sich an den Prinzipien des World Wide Web und beschreibt die leichtgewichtige Kommunikation zwischen Services im Netzwerk.

Für die Programmiersprache Python stehen mit Flask und Flask-RESTPlus umfangreiche Frameworks zur Verfügung, mit denen das Entwickeln von REST-APIs leicht gelingt. Dieser Artikel stellt anhand einer Beispielanwendung zur Verwaltung von Brettspielsammlungen vor, wie RESTful Webservices auf Basis von Flask-RESTPlus funktionieren.

Flask ist ein schlankes, in Python geschriebenes Webframework, das einen einfachen Einstieg ermöglicht und sich zudem gut erweitern lässt. Im Kern enthält es Basisfunktionen, die sich durch Erweiterungen etwa für Authentifizierung, Object-Relational Mapping, Caching oder RESTful APIs ergänzen lassen. Flask macht hierbei und bei der Projektstruktur wenig Vorgaben und lässt Entwicklern viele Freiheiten. Es zählt mittlerweile zu den populärsten Python-Webframeworks und Unternehmen wie Pinterest, LinkedIn, Netflix und Red Hat setzen es ein (siehe ix.de/zhjd).

Die Installation von Flask erfolgt über das Python-Paketverwaltungsprogramm pip mit dem Kommando pip install

Flask. Listing 1 zeigt eine einfache Flask-Anwendung, die nach dem Import der Flask-Bibliothek eine Instanz der Applikation mit dem Namen der App als Argument erstellt. Der Decorator @app.route erzeugt die Route /hello. Jeder Routenaufruf führt die Funktion hello() aus, die den String "Hello World!" zurückgibt.

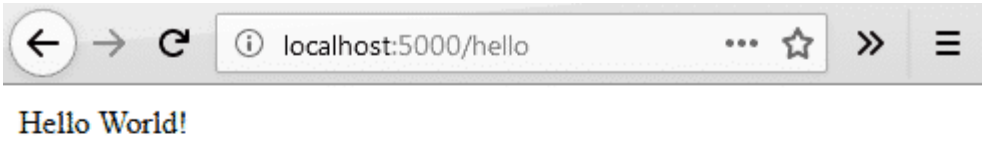
Listing 1: Einfacher Flask-Webservice (hello.py)

```
from flask import Flask

app = Flask("my hello app")

@app.route("/hello")
def hello():
    return "Hello World!"

app.run()
```

Das Ausführen der Datei hello.py auf der Konsole startet den Webservice mit folgender Ausgabe:

```
python hello.py
* Serving Flask app "my hello app"
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

Flask verwendet hierzu standardmäßig einen integrierten einfachen Entwicklungsserver, der nicht für produktive Umgebungen geeignet ist. Beim Aufruf des Links im Browser erscheint der String "Hello World!" (Abbildung 1).



Die Anwendung lässt sich im Browser unter http://localhost:5000/hello aufrufen (Abb. 1).

In Kooperation

Flask-RESTPlus ist eine Erweiterung für das Flask-Framework, mit der sich REST-APIs einfach entwickeln lassen. Sie stellt eine Reihe von Werkzeugen zur Beschreibung der API bereit und generiert automatisch einen Endpunkt für Swagger UI – ein Framework für die interaktive Dokumentation und Visualisierung von APIs.

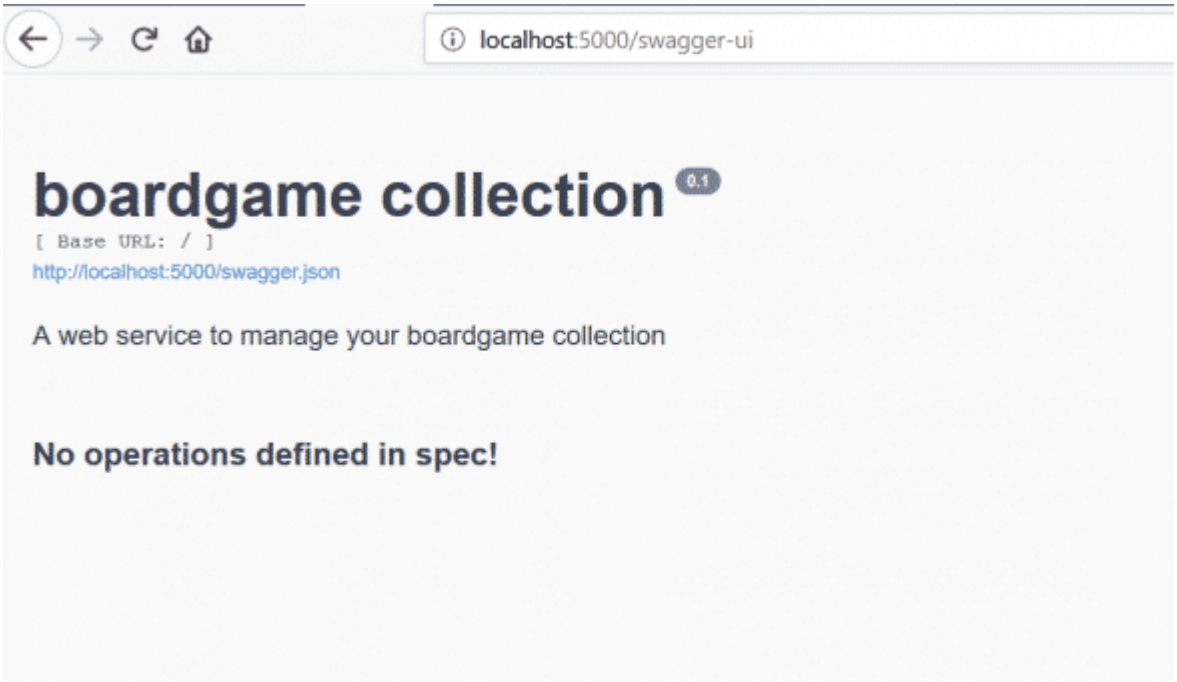
Zum Installieren startet man pip mit dem Kommando pip install flask-restplus. Der Haupteinstiegspunkt einer Flask-RESTPlus-Anwendung ist die Klasse Api, die sich mit einer Flask-Applikation initialisieren lässt (Listing 2). Die Konfiguration von Api wie Version, Titel, Beschreibung, Pfad zur Dokumentation, Kontaktinformationen und Lizenz erfolgt über den Konstruktor. Als Beispiel dient hier ein Webservice zum Verwalten einer Brettspielsammlung (boardgame collection), die in den folgenden Abschnitten um weitere Funktionalitäten ergänzt wird.

Listing 2: Haupteinstiegspunkt der API (main.py)

```
from flask import Flask
from flask_restplus import Api

app = Flask("boardgame collection")
```

```
api = Api(app,
          version='0.1',
          title='boardgame collection',
           description='A web service to manage your boardgame
collection',
          doc='swagger-ui')

if __name__ == '__main__':
    app.run(debug=True, port=5000)
```

Der Aufruf der run-Methode startet den Service. Die automatisch generierte API-Dokumentation lässt sich im Browser aufrufen, die Informationen hierzu stammen aus der Konfiguration (Abbildung 2).



Swagger UI zeigt die API-Dokumentation unter http://localhost:5000/swagger-ui (Abb. 2).
Zur besseren Übersicht ist die Anwendung in einer Baumstruktur entsprechend ihren funktionalen Aufgaben organisiert (Listing 3). Hierzu legt man für jede API-Ressource eine Datei im Paket resources an. Die zugehörigen fachlichen Modellbeschreibungen finden sich im Paket models in jeweils eigenen Dateien. Alle Tests der API sind im Paket tests abgelegt. Die folgenden Abschnitte erläutern Ressourcen, Modelle und Tests für die App boardgame collection.

Listing 3: Struktur und Aufbau der Anwendung

```
├── resources
│       └── __init__.py
│       └── boardgames.py
├── models
│       └── __init__.py
│       └── boardgame.py
│       └── boardgame-expansion.py
├── tests
│       └── __init__.py
│       └── test_boardgames_api.py
├── environments.py
└── main.py
```

Die spezifische Konfiguration verschiedener Zielumgebungen wie Produktiv- oder Entwicklungssystem erfolgt über environments.py (Listing 4).

Listing 4: Verschiedene Zielumgebungen konfigurieren (environments.py)

```
import os

class DevelopmentConfig:
    port = 5000
    debug = True
    log_path = "boardgames.log"
    documentation_path = "/swagger-ui"
    ...

class ProductionConfig:
    port = 8000
    debug = False
    log_path = "boardgames.log"
    documentation_path = None
    ...

configurations = {
    "dev":  DevelopmentConfig,
    "prod": ProductionConfig }
```

```
environment = os.environ.get("BG_CONFIG", "dev")
config = configurations[environment]
```

Hierdurch lässt sich das jeweilige Einstellungsset für Port oder Debug-Modus über eine Umgebungsvariable von außen für unterschiedliche Deployments steuern. Ein Python Dictionary speichert die Konfigurationen ProductionConfig und DevelopmentConfig. Das Programm liest beim Importieren der -environments.py die Umgebungsvariable BG_CONFIG ein und setzt die entsprechende Konfiguration für die Zielumgebung. Der Import der Konfiguration mit

```
from environments import config
```

stellt die Einstellungen in Komponenten wie der main-Methode bereit (Listing 5). Die Methode run erhält die Einstellungen für den Debug-Modus und den Port für die Zielumgebung. Außerdem übernimmt die Anwendung den Pfad zu Swagger UI aus config und initialisiert einen Python-Standard-Logger mit dem entsprechenden Pfad aus der Konfiguration. Vor dem Ausführen der Anwendung lässt sich die Zielumgebung über die Kommando-zeile setzen (unter Windows mit set BG_CONFIG=prod; unter Linux mit ENV BG_CONFIG=prod).

Für den produktiven Betrieb von Flask-Anwendungen eignen sich diverse Webserver wie Apache Web Server, nginx, Cherokee und Gunicorn, da Flask auf die WSGI-Standardschnittstelle (Web Server Gateway Interface) für die Kommunikation zwischen Webserver und Webframework setzt. Für Details zur Skalierung und zum Betrieb produktiver Flask-RESTPlus-Anwendungen siehe ix.de/zhjd.

Listing 5: Haupteinstiegspunkt der API um Konfiguration aus environments ergänzt (main.py)

```
from flask import Flask
from flask_restplus import Api
from environments import config
import logging
```

```
app = Flask("boardgame collection")
api = Api(app,
          ...
          doc=config.documentation_path)

if __name__ == '__main__':
          logging.basicConfig(filename=config.log_path,
level=logging.DEBUG)
    logging.info("start boardgame collection service ...")

    app.run(debug=config.debug, port=config.port)
```

Ressourcen und Routing

Zentraler Bestandteil jeder REST-API sind Ressourcen. Hierzu stellt Flask-RESTPlus eine entsprechende Basisklasse (Resource) zur Verfügung, die das Routing für verschiedene HTTP-Methoden (GET, PUT, DELETE) für einen gegebenen Endpunkt (URL) ermöglicht.

Die Ressource Boardgame in Listing 6, abgeleitet von Resource, stellt die Basis-CRUD-Operationen Create, Read, Update und Delete für ein einzelnes Brettspiel bereit. Der Decorator @api.route('/<id>') weist die Route zu. Mit der Ressource BoardgameList lässt sich auf Brettspiellisten zugreifen. Beide Ressourcen fasst man in einem gemeinsamen Namespace /boardgames zusammen. Namespaces gruppieren in Flask-RESTPlus Ressourcen unter einem Endpunkt; mit der Methode add_namespace(...) fügt man sie einer Api hinzu (Listing 7).

Listing 6: Boardgame-Ressource (resources/boardgames.py)

```
from flask_restplus import Namespace, Resource
from flask import abort

bg_collection = {} # boardgames stored in memory
api = Namespace('boardgames', description='boardgame related
operations')
```

```
@api.route('/<id>')
class Boardgame(Resource):
    def get(self, id):
      '''Gets a boardgame by id'''
      if id in bg_collection:
        return bg_collection[id], HTTPStatus.OK
      else:
            abort(HTTPStatus.NOT_FOUND, 'Boardgame {0} not
found.' .format(id))

    def delete(self, id):
      '''Removes a boardgame from the collection'''
      if id in bg_collection:
        del bg_collection[id]
        return '', HTTPStatus.NO_CONTENT
      else:
        abort(HTTPStatus.NOT_FOUND, 'Boardgame {0} not found.'
.format(id))

    def put(self, id):
      '''Updates a boardgame'''
      boardgame = request.json
      if id in bg_collection:
        bg_collection[id] = boardgame
        return boardgame, HTTPStatus.OK
      else:
        abort(HTTPStatus.NOT_FOUND, 'Boardgame {0} not found.'
.format(id))

@api.route('')
class BoardgameList(Resource):
    def get(self):
      '''Lists all boardgames in collection'''
      bg_collection_list = list(bg_collection.values())
      return bg_collection_list, HTTPStatus.OK

    def post(self):
      '''Adds a boardgame to collection'''
      boardgame = request.json
      bg_collection[boardgame['id']] = boardgame
      return boardgame, HTTPStatus.CREATED
```

Listing 7: Namespace boardgame registrieren (main.py)

```
from flask import Flask
from flask_restplus import Api
from app.resources.boardgames import api as boardgames_api_namespace

app = Flask("boardgame collection")
api = Api(...)

api.add_namespace(boardgames_api_namespace)
...
```

Somit ergeben sich die folgenden URLs für den Brettspiel-Webservice:

- GET http://localhost:5000/boardgames/1 gibt das Brettspiel mit der id 1 zurück.
- DELETE http://localhost:5000/boardgames/1 löscht das Brettspiel mit der id 1.
- PUT http://localhost:5000/boardgames/1 aktualisiert das Brettspiel mit der id 1.
- GET http://localhost:5000/boardgames gibt die Liste von Brettspielen zurück.
- POST http://localhost:5000/boardgames erzeugt ein neues Brettspiel in der Liste.

Der Aufruf dieser URLs etwa über curl http://localhost:5000/boardgames/1 -X GET triggert die entsprechende Methode in der Python-Ressource (wie get der Klasse Boardgame). Die Methode verarbeitet dann den Input, generiert eine Antwort und schickt diese als HTTP-Response an den Aufrufer zurück. Die Methode erhält die Inputparameter wie id zur Identifikation eines Objekts aus dem URL-Pfad als Parameter.

Für die Antwortnachricht wandelt die Methode die Rückgabewerte automatisch in ein Flask-Response-Objekt um. Sie kann bis zu drei Rückgabewerte enthalten. An erster Stelle steht der im

HTTP Response Body verwendete Content. Optional lassen sich über die weiteren Rückgabewerte der HTTP-Statuscode – als Default wird hier 200 OK verwendet – und eigene HTTP Response Header setzen. Die Hilfsmethode abort generiert im Fehlerfall eine HTTPException abhängig vom jeweiligen HTTP-Statuscode.

Das Python Dictionary bg_collection hält die Daten, hier Brettspiele, im Speicher. Auf das Anbinden einer Datenbank wird zugunsten der besseren Lesbarkeit verzichtet. Somit lassen sich jetzt Brettspiele über die API der Sammlung hinzufügen und abrufen (Listing 8).

Listing 8: Brettspiel erstellen und Sammlung über curl ausgeben

```
$ curl -X POST "http://localhost:5000/boardgames" -d "{ "id":
"1",
"name":  "Wingspan",  "designer":  "Elizabeth  Hargrave",
"playing_time": "60 Min",
"rating": 8.1,  "expansions": [{"name": "European Expansion",
"rating": "8.5"}]}"

$ curl -X GET "http://localhost:5000/boardgames"
[{
    "id": "1",
    "name": "Wingspan",
    "designer": "Elizabeth Hargrave",
    "playing_time": "60 Min",
    "rating": 8.1,
    "expansions":
      [{
        "name": "European Expansion",
        "rating": 8.5
      }]
}]
```

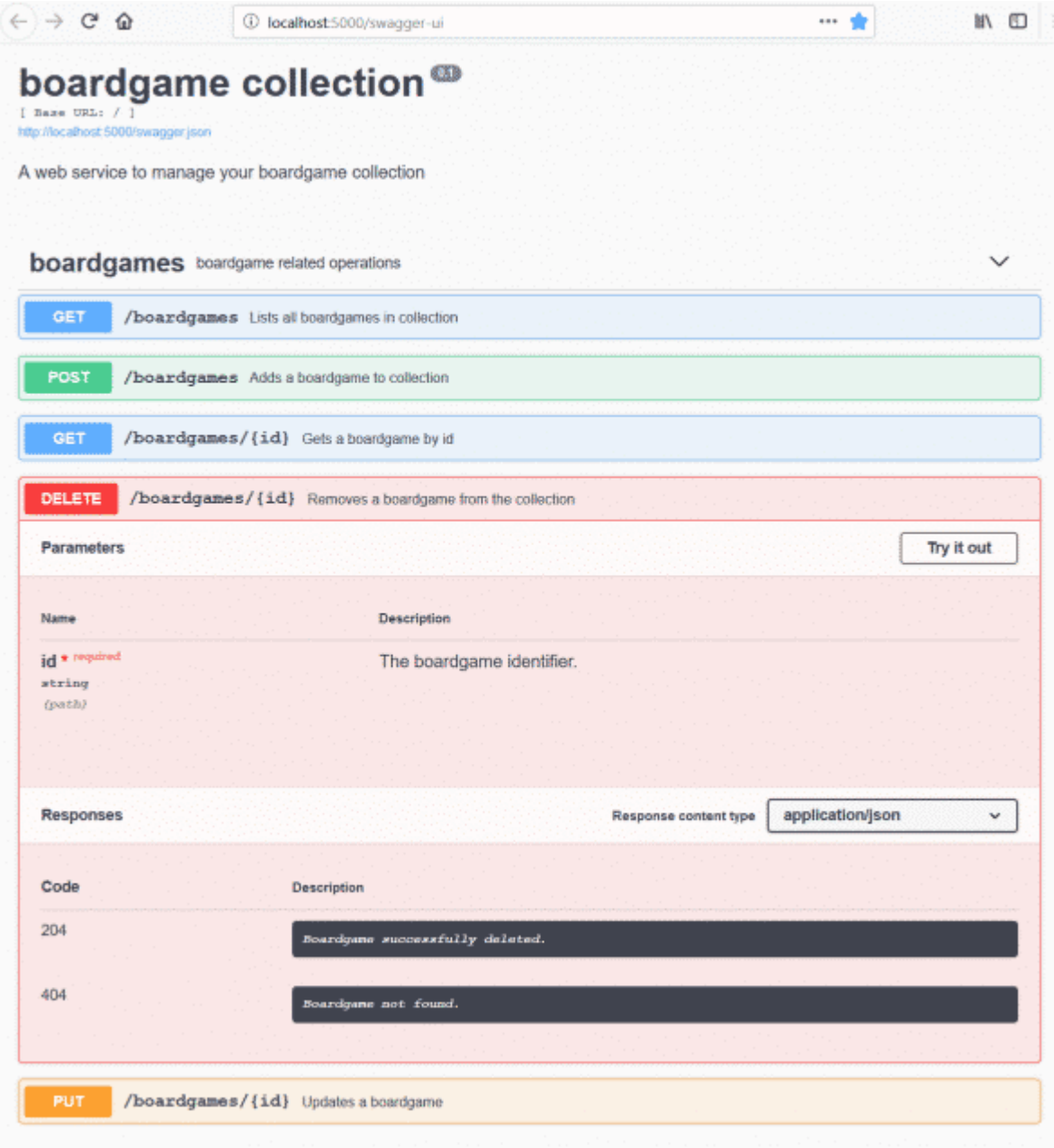
Kleine Helferlein

Verschiedene Werkzeuge von Flask-RESTPlus dienen dazu, die automatisch generierte Dokumentation auf Basis der OpenAPI

Specification individuell anzupassen. So ermöglicht es der Decorator @api.response(), mögliche Antworten einer Ressource oder Methode festzulegen und sie in der Dokumentation unter der Response der jeweiligen Route aufzuführen (Listing 9 und Abbildung 3). Alternativ kann die Dokumentation der Parameter aus dem URL-Pfad über den Decorator @api.param() erfolgen und damit global an der Ressource (gilt dann für alle Methoden) oder individuell je Methode.

Listing 9: Boardgame-Ressource um Dokumentation erweitert (resources/boardgames.py)

```
...
@api.route('/<id>')
@api.param('id', 'The boardgame identifier.')
@api.response(404, 'Boardgame not found.')
class Boardgame(Resource):
    ...
    @api.response(204, 'Boardgame successfully deleted.')
    def delete(self, id):
      ...

    @api.response(200, 'Boardgame successfully updated.')
    def put(self, id):
      ...

@api.route('')
class BoardgameList(Resource):
    @api.response(200, 'All boardgames successfully fetched!')
    def get(self):
      ...
    @api.response(201, 'Boardgame successfully created.')
    def post(self):
      ...
```



Swagger UI stellt das Routing für die Ressource boardgames dar (Abb. 3).
Flask-RESTPlus ermöglicht es, die Struktur der Payload von HTTP-Anfragen und Antworten durch Modelle zu beschreiben. Das Framework rendert auf Basis dieser Modelle die HTTP-Payload automatisch aus den internen Daten wie ORM-Modellen und eigenen Klassen. Über das als Marshalling bezeichnete Verfahren lässt sich steuern, welche Daten die API in welchem Format kommuniziert.

Mit der Factory model() lassen sich Modelle auf dem Namespace der API erstellen und registrieren. Listing 10 legt durch api_namespace.model('Boardgame',{...}) ein Modell für das

Objekt Brettspiel (boardgame) an. Ein Dictionary beschreibt das Modell. Jeder Schlüssel im Dictionary repräsentiert hierbei ein in der HTTP-Payload gerendertes Attribut. Der zugehörige Wert definiert das Format des zu rendernden Attributs, beschrieben durch das Modul fields. Flask-RESTPlus stellt unter diesem Modul verschiedene Klassen für Datentypen wie Zeichenketten (String), Fließkommazahlen (Float), Datum (DateTime) und Objektlisten (List) bereit. Die Datentypen besitzen eine Reihe optionaler Argumente, die das Feld detailliert beschreiben. So lassen sich zum Beispiel eine Ober- und Untergrenze für einen Wert festlegen, Defaultwerte vorgeben, eine Beschreibung (description) ergänzen und ein Wert als erforderlich (required) definieren.

Mit Modellen arbeiten

Listing 10: Boardgame-Modell (model/boardgame.py)

```
from flask_restplus import fields
from models.expansion import create_expansion_model

def create_boardgame_model(api):
    boardgame_model = api.model('Boardgame', {
        'id': fields.String(description='unique boardgame
identifier',
                            required=True),
      'name': fields.String(description='boardgame name',
                            min_length=3,
                            max_length=128,
                            required=True),
         'designer': fields.String(description='boardgame
designer',
                                   required=True),
        'playing_time': fields.String(description='aprox.
playing time',
                                      enum=["15 Min", "30 Min",
"60 Min"],
                                      required=True),
      'rating': fields.Float(description='rating [0 to 10]',
                             min=0.0,
```

```
                                max=10.0,
                                required=False),
                                                    'expansions':
fields.List(fields.Nested(create_expansion_model(api),
description='list of expansions',
                                                   required=False))
    })

    return boardgame_model
```

Die Modelle der hier vorgestellten Anwendung zur Verwaltung einer Brettspielsammlung sind im Paket models abgelegt. Listing 10 und 11 erstellen die Modelle Brettspiel (boardgame) und Brettspielerweiterung (expansion). Ein Brettspiel erhält das Attribut expansions, das eine Liste der zugehörigen Erweiterungen speichert. Flask-RESTPlus stellt die Klasse fields.Nested zum Anlegen verschachtelter Strukturen bereit, der sich zuvor angelegte Modelle wie expansion_model übergeben lassen. Die so definierten Modelle sind in der API-Dokumentation dargestellt (Abbildung 4).

Listing 11: Expansion-Modell (model/expansion.py)

```
from flask_restplus import fields

def create_expansion_model(api):
    expansion_model = api.model('Expansion', {
       'name': fields.String(description='boardgame expansion
name',
                              min_length=3,
                              max_length=128,
                              required=True),
      'rating': fields.Float(description='rating [0 to 10]',
                              min=0.0,
                              max=10.0,
                              required=False)
    })
    return expansion_model
```

Swagger UI zeigt die Modellbeschreibung für Boardgame und Expansion (Abb. 4).
Damit sich diese Modelle innerhalb einer Ressource verwenden lassen, stellt Flask-RESTPlus verschiedene Decorators zur Verfügung. Der Decorator @api.marshal_with(model) definiert für eine Ressourcenmethode, dass die Daten in der HTTP-Antwort automatisch auf Basis des entsprechenden Modells gerendert werden (Listing 12). @api.marshal_list_with(model) rendert die Rückgabe als Objektliste des Modells.

Listing 12: Boardgame-Ressource um Modelle ergänzt (resources/boardgames.py)

```
...
from models.boardgame import create_boardgame_model

bg_collection = {} # boardgames stored in memory
api = Namespace('boardgames', description='boardgame related operations')

boardgame = create_boardgame_model(api)

@api.route('/<id>')
```

```
@api.param('id', 'The boardgame identifier.')
@api.response(404, 'Boardgame not found.')
class Boardgame(Resource):
    @api.marshal_with(boardgame)
    def get(self, id):
      ...

    @api.response(200, 'Boardgame successfully updated.')
    @api.expect(boardgame, validate=True)
    def put(self, id):
      ...


@api.route('')
class BoardgameList(Resource):
    @api.response(200, 'All boardgames successfully fetched!')
    @api.marshal_list_with(boardgame)
    def get(self):
      ...

    @api.response(201, 'Boardgame successfully created.')
    @api.expect(boardgame, validate=True)
    def post(self):
      ...
```

Der Decorator @api.expect(model) erlaubt es, die Payload einer HTTP-Anfrage auf Basis des Modells zu spezifizieren. Das Setzen des optionalen Parameters validate auf True aktiviert die Inputvalidierung. So gleicht Flask-RESTPlus das in der Payload enthaltene Objekt mit der Modellspezifikation ab und generiert im Fall von Abweichungen eine Antwort mit dem Status 400 (Bad Request). Diese Antwort enthält außerdem Details zu den gefundenen Abweichungen (Listing 13).

Listing 13: Fehlermeldung aufgrund fehlgeschlagener Inputvalidierung für die post-Route

```
$ curl -X POST "http://localhost:5000/boardgames" -d "{ "id":
"2", "rating": 11.0
"name": "King of Tokyo", "playing_time": "30 Min",
```

```
"expansions": []}"

400 BAD REQUEST
{
    "errors": {
      "designer": "'designer' is a required property",
      "rating": "11.0 is greater than the maximum of 10.0"
    },
    "message": "Input payload validation failed"
}
```

Sortieren und testen

Für API-Endpunkte, die Listen von Elementen zurückliefern, ist es üblich, die maximale Anzahl abzurufender Elemente zu begrenzen. Dies bezeichnet man als Pagination. Andernfalls kann es bei großen Datenmengen zu Performanceproblemen kommen. Der API-Aufrufer übergibt bei der Pagination in der Regel ein Offset (Startelement in der Liste) und ein Limit (die maximale Anzahl abzurufender Elemente) als URL-Parameter. Darüber hinaus lässt sich über URL-Parameter auch das Sortieren und Filtern der Listenelemente realisieren.

Vor diesem Hintergrund implementiert Listing 14 entsprechende Mechanismen für die get-Route der Ressource BoardgameList, die eine Liste von Brettspielen ausliefert. Hierzu definiert der Decorator @api.param() die URL-Parameter für Pagination (offset und limit) und Sortierung (sort_by und sort_order). Der URL-Paramerter sort_by gibt vor, nach welchem Attribut die Liste der Brettspiele zu sortieren ist. Über den Decorator @api.param() lassen sich neben einer Beschreibung des Parameters auch der Datentyp und ein Defaultwert festlegen. So ist der URL-Parameter sort_order als Aufzählungstyp (enum) realisiert und definiert die Reihenfolge der Sortierung als auf- oder absteigend (asc oder desc).

Listing 14: Ressource BoardgameList um URL-

Parameter für Pagination und Sortierung erweitert (resources/boardgames.py)

```
@api.route('')
class BoardgameList(Resource):
    @api.param('offset',
               'The offset of the first boardgame in the list
to return.',
               type=int,
               default=0)
    @api.param('limit',
               'The maximum number of boardgames to return.',
               type=int,
               default=100)
    @api.param('sort_by',
               'Sort the returned boardgames by key.',
               default="id",
                enum=["id", "name", "designer", "rating"])
    @api.param('sort_order',
               'Ascending or descending sort order.',
               default="asc", enum=["asc", "desc"])
    @api.response(200, 'All boardgames successfully fetched!')
    @api.marshal_list_with(boardgame)
    def get(self):
        '''Lists all boardgames in collection'''
        offset = request.args.get('offset')
        ...
```

Über die Anfrageargumente (request.args) greift man auf die URL-Parameter innerhalb der get-Methode der Ressource zu. Der folgende curl-Aufruf ruft die ersten hundert Brettspiele aufsteigend nach der ID sortiert ab:

```
curl -X GET
"http://localhost:5000/boardgames?sort_
order=asc&sort_by=id&limit=100&offset=0"
```

Die vorgestellte Boardgame-Collection-API testet man über das Python-Unit-Testing-Framework unittest. Im Paket tests ist hierzu die Datei test_boardgames_api.py angelegt, die eine Klasse BoardgamesApiTest enthält – abgeleitet von

unittest.TestCase. Das Initialisieren der Tests findet in der Methode setUp statt, die automatisch vor jedem Test ausgeführt wird. Die Methode test_client() des Flask-Frameworks erstellt innerhalb von setUp einen Testclient für die API. Für die Kommunikation mit der API stellt der Client Schnittstellen für die verschiedenen HTTP-Methoden bereit. So lässt sich mit response = self.app.get('/boardgames') die Route boardgames get triggern und die HTTP-Antwort im Rahmen der Tests auswerten. Listing 15 zeigt exemplarisch einige Tests zur Absicherung der API-Grundfunktionen. Man startet die Tests über die Kommandozeile mit

```
python -m unittest tests/test_boardgames_api.py
```

Listing 15: Boardgame-API testen (tests/test_boardgames_api.py)

```
import unittest
from main import app

class BoardgamesApiTest(unittest.TestCase):
    def setUp(self):
      # 1. ARRANGE
      self.app = app.test_client()
      # initialize app with first boardgame
      self.app.post('/boardgames', json={
          "name": "Wingspan",
          "designer": "Elizabeth Hargrave",
          "playing_time": "60 Min",
          "rating": 8.1,
          "id": "1",
            "expansions": [{"name": "European Expansion",
"rating": 8.5}]
      })
    def test_get_boardgame_by_id(self):
      # 2. ACT
      response = self.app.get('/boardgames/1')
      # 3. ASSERT
      self.assertEqual('200 OK', response.status)
      self.assertEqual('Wingspan', response.json['name'])
```

```
    def test_get_boardgame_for_unknwon_id(self):
      # 2. ACT
      response = self.app.get('/boardgames/2')
      # 3. ASSERT
      self.assertEqual('404 NOT FOUND', response.status)
         self.assertTrue("Boardgame 2 not found." in
response.json['message'])

    def test_delete_boardgame_by_id(self):
      # 2. ACT
      response = self.app.delete('/boardgames/1')
      # 3. ASSERT
      self.assertEqual('204 NO CONTENT', response.status)
      response = self.app.get('/boardgames')
      self.assertEqual(0, len(response.json))

    def test_post_new_boardgame(self):
      # 2. ACT
      response = self.app.post('/boardgames', json={
          "name": "King of Tokyo",
          "designer": "Richard Garfield",
          "playing_time": "30 Min",
          "rating": 7.2,
          "id": "2",
          "expansions": []
      })
      # 3. ASSERT
      self.assertEqual('201 CREATED', response.status)
      self.assertEqual('King of Tokyo', response.json['name'])
```

Fazit

Flask und Flask-RESTPlus bieten einen beachtlichen Funktionsumfang und erlauben die Entwicklung robuster REST-APIs auch für Enterprise-Anwendungen. Hierbei ermöglicht der ressourcenorientierte Aufbau von Flask-RESTPlus, APIs nah am REST-Standard zu realisieren. Die automatische HTTP-Payload-Validierung durch das Framework auf Basis der Modellbeschreibungen reduziert die Fehleranfälligkeit der Anwendung. Mit wenigen Zeilen gut lesbarem Code entstehen

umfangreiche APIs, die sich einfach testen und weiterentwickeln lassen. API-Konsumenten hilft die automatisch generierte API-Dokumentation auf Basis der OpenAPI Specification und das Bereitstellen über Swagger UI.

Wer sich für die Sicherheit von Flask-Anwendungen interessiert, findet in der Flask-Dokumentation weitere Informationen (siehe ix.de/zhjd). (nb@ix.de)

1. Quellen
2. Listings, Informationen zu Flask im Unternehmen, Skalierung und Sicherheit: ix.de/zhjd

Dr. Sebastian Bindick

ist IT-Architekt in der Volkswagen-IT. Seine Schwerpunkte liegen in den Bereichen moderne Architekturen, Webtechnologien, Testautomatisierung und Computational Engineering.

[/expand]