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.

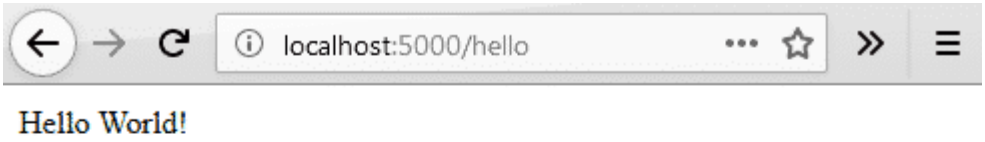
Listing 1: Einfacher Flask-Webservice (hello.py)

```
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

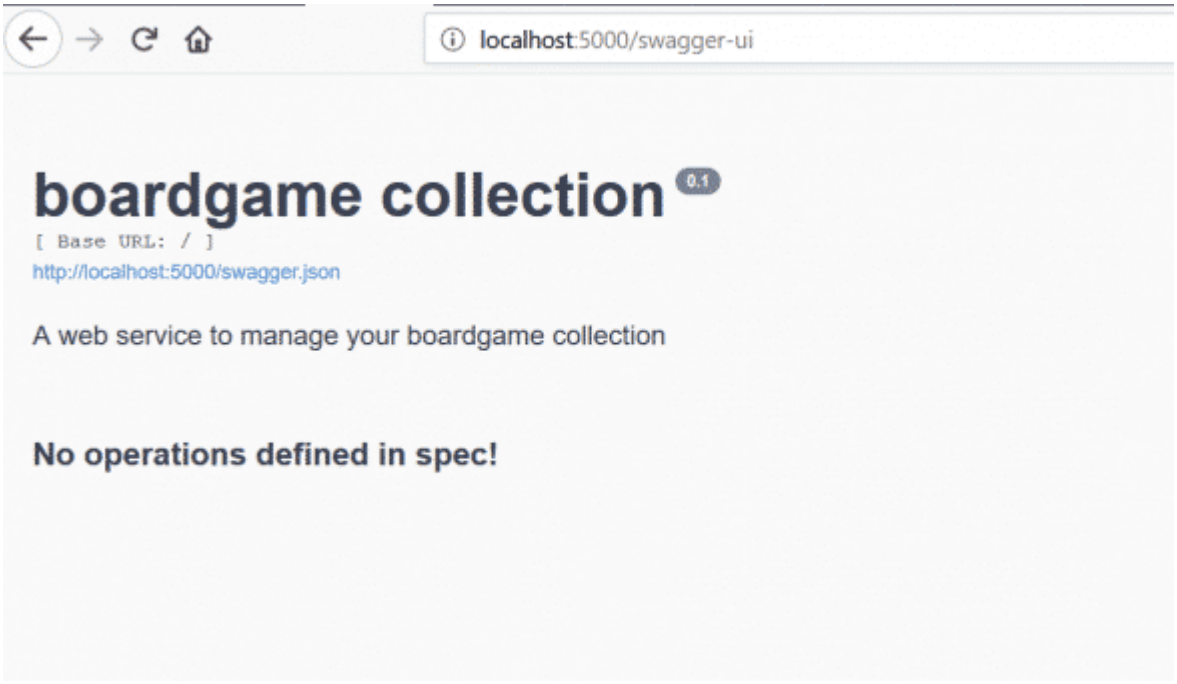
Flask-RESTPlus ist eine Erweiterung für das Flask-Framework, mit der sich REST-APIs einfach entwickeln lassen. Sie stellt eine Reihe von Werkzeugen zur Beschreibung der API bereit und generiert automatisch einen Endpunkt für Swagger UI – ein Framework für die interaktive Dokumentation und Visualisierung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

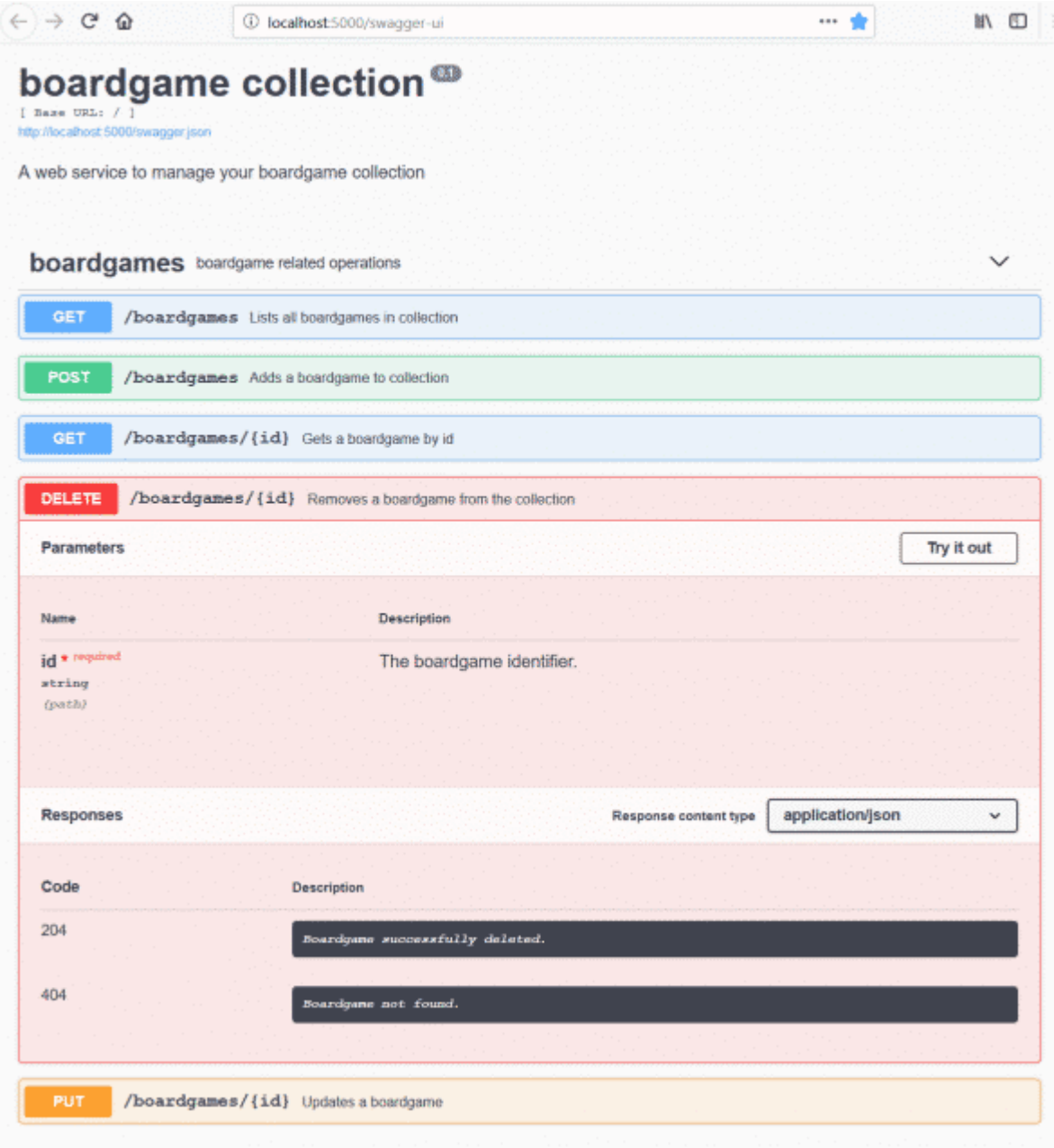
Kleine Helferlein

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]