الجمهورية العربية السورية

وزارة التعليم العالي

جامعة تشرين

كلية الهندسة الميكانيكية والكهربائية

هندسة الاتصالات والإلكترونيات – السنة الخامسة

برمجة الشبكات

Designing a RESTful API Services using FLASK

إعداد

ولاء محمد اليونس حياة صالح شبيب

إشراف

د.م مهند عيسى

فهرس المحتويات

ملخص البحث

سنبين في بحثنا طريقة تصميم RESTful API للتفاعل مع قاموس بسيط لتخزين البيانات وإجراء عمليات CRUD مع الإشعارات، لاستخدامها كأساس لخدمة مصغرة، وتحديد متطلبات API الخاصة بنا وتوضيح المهام التي تؤديها كل طريقة من طرق HTTP. وإنشاء بيئة افتراضية مع Flask و Flask-RESTful.

سنوضح الية إنشاء Model لتمثيل الإشعارت وعرضها وآلية إنشاء أصناف لتمثيل مصادر وعمليات مختلف أوامر HTTP وكيفية إنشاء Flask Server باستخدام CMD الذي يقوم بإنشاء وارسال أوامر HTTP إلى RESTFUL API وتحليل كل أمر ونتائجه.

كلمات مفتاحية:

API – Flask – Server- HTTP Request

المقدمة

واجهة التطبيق البرمجية (Application Programming Interface وتختصر إلى API) وصفٌ العناصر البرمجيّة حسب وظائفها، ومدخلاتها ومخرجاتها. ويتمثل الهدف الرئيسيّ منها في توفير قائمة من الوظائف المستقلّة تمامًا عن الآلية التي نفّذت بها، لتتيح للآخرين التواصل معها من خلال أيّ آلية أخرى.

تسهل Python علينا إنشاء RESTful Web APIs. تقدم Python إطار العمل (FrameWorks) Flask الذي يعد مثاليا لإنشاء خدمات توفر RESTful APIs. سنقوم في مشروعنا بإنشاء RESTful Web API الذي يقوم بعمليات CRUD (الإنشاء والقراءة والتحديث والحذف). سنعمل على توضيح التفاعل مع RESTful API لإجراء عمليات CRUD مع سلسلة رسائل تمثل إشعارات.

قمنا باختيار Flask FrameWork لأنه إطار عمل خفيف. بالإضافة إلى ذلك ، يعد Flask خيارًا مناسبًا لإنشاء خدمة صغيرة يمكنها تشغيل RESTful API على الخدمات السحابية.

١- المشكلة المقترحة : نريد إعداد رسالة تنبيه تعرض على شاشة المستخدم بحيث يكون لدينا كود برمجي يعيد الرسائل المختلفة المخزنة في قاعدة بيانات. يجب أن نحدد متطلبات ظهور الإشعار. وهي بشكل مختصر: تعريف عدد صحيح – محارف لتشكيل الرسالة – TTL (time to live) المدة الزمنية بالثواني لظهور الإشعار – زمن وتاريخ إنشاء الإشعار – تحديد طبيعة الإشعار هل هو تحذير أو معلومة ما.

٢- Micro services: بدأت العديد من التطبيقات الكبيرة في التحول من المعمارية المعقدة للبرامج إلى بنية الخدمات المصغرة التي تقوم على تنفيذ جميع الميزات التي تتطلبها التطبيقات الكبيرة بطريقة تضمن العمل الجيد والمستمر. تعد RESTful API جزءًا أساسيًا من بنية الخدمات المصغرة . يمكن لكل خدمة مصغرة أن تغلف RESTful API تحقق هدفا معينا. الخدمة المصغرة مستقلة ، ومن السهل صيانتها هناك عدة طرق لتنفيذ بنية الخدمات المصغرة. سوف نقدم كيفية تغليف RESTful API باستخدام الخدمات المصغرة بواسطة Flask و Python.

٣- تجهيز البيئة الافتراضية: نفتح القرص C ثم نختار مجلد users ونفتح مجلد المستخدم الأساسي وونشئ ملف ةتسميه flaskRestProject وننشئ بداخله ملف نسميه Flask01. ثم نقوم بفتح ال CMD ونكتب الأمر التالي:

python -m venv %USERPROFILE%\flaskRestProject\Flask01

يقوم الأمر السابق بإعداد البيئة الافتراضية في الملف الهدف من أجل التطبيق

ثم نقوم بتفعيل البيئة الافتراضية من خلال الأمر التالي:

%USERPROFILE%\flaskRestProject\Flask01\Scripts\activate.bat

فيتم تفعيل البيئة ونلاحظ أن المسار أصبح يبدأ ب Flask01 الذي يدل على أننا تعمل من داخل البيئة

ثم نقوم بإنشاء ملف نصي requirements.txt ونكتب به مايلي:

```
Flask==1.0.2
flask-restful==0.3.6
httpie==1.0.0
```

كل سطر من السطور السابقة يدل على الحزمة التي سنحتاجها لبناء التطبيق

ثم نقوم بكتابة سطر الأوامر التالي في CMD:

pip install -r requirements.txt

الذي يقوم بتنزيل وتثبيت كل الحزم المطلوبة.

ثم نقوم بإنشاء مجلد نسميه service في المجلد الجذر ونقوم بإنشاء ملف http_status.py ونكتب بداخله كود يصف حالة اتصال HTTP ويبين وضع الاتصال مثلا الحالة 404 تعني أن الصفحة غير موجودة ونكتب الكود التالي لوصف ماسبق:

```python
from enum import Enum
class HttpStatus(Enum):
    continue_100 = 100
    switching_protocols_101 = 101
    ok_200 = 200
    created_201 = 201
    accepted_202 = 202
    non_authoritative_information_203 = 203
    no_content_204 = 204
    reset_content_205 = 205
    partial_content_206 = 206
    multiple_choices_300 = 300
    moved_permanently_301 = 301
    found_302 = 302
    see_other_303 = 303
    not_modified_304 = 304
    use_proxy_305 = 305
    reserved_306 = 306
    temporary_redirect_307 = 307
    bad_request_400 = 400
    unauthorized_401 = 401
    payment_required_402 = 402
    forbidden_403 = 403
    not_found_404 = 404
    method_not_allowed_405 = 405
    not_acceptable_406 = 406
    proxy_authentication_required_407 = 407
    request_timetout_408 = 408
    conflict_409 = 409
    gone_410 = 410
    length_required_411 = 411
    precondition_failed_412 = 412
    request_entity_too_large_413 = 413
    request_uri_too_long_414 = 414
    unsupported_media_type_415 = 415
    requested_range_not_satisfiable_416 = 416
    expectation_failed_417 = 417
    precondition_required_428 = 428
    too_many_requests_429 = 429
    request_header_fields_too_large_431 = 431
    unavailable_for_legal_reasons_451 = 451
    internal_server_error_500 = 500
    not_implemented_501 = 501
    bad_gateway_502 = 502
    service_unavailable_503 = 503
    gateway_timeout_504 = 504
    http_version_not_supported_505 = 505
    network_authentication_required_511 = 511

    @staticmethod
    def is_informational(cls, status_code):
        return 100 <= status_code.value <= 199

    @staticmethod
    def is_success(status_code):
        return 200 <= status_code.value <= 299

    @staticmethod
    def is_redirect(status_code):
```

```
        return 300 <= status_code.value <= 399

    @staticmethod
    def is_client_error(status_code):
        return 400 <= status_code.value <= 499

    @staticmethod
    def is_server_error(status_code):
        return 500 <= status_code.value <= 599
```

نلاحظ أن الكود يحوي الصنف HttpStatus الذي يرث الصنف Enum ويحدد الصنف HttpStatus مجموعات فريدة من الأسماء والقيم التي تمثل رموز حالة HTTP المختلفة. تستخدم الأسماء وصف ورقم رمز حالة HTTP . على سبيل المثال ، يتم تحديد قيمة ٢٠٠ لرمز الحالة HTTP 200 OK في اسم HttpStatus.ok_200 ، ويتم تحديد رمز الحالة HTTP 404 غير موجود في اسم HttpStatus.not_found_404. بالإضافة إلى ذلك ، يحدد HttpStatus خمس تعيد أحد النتائج: نجاح، إعادة توجيه، خطأ في العميل، خطأ في الخادم .

٤- إنشاء MODEL:

الآن ننشئ ملف بايثون جديد ونسميه models.py ونكتب فيه الكود التالي:

```
class NotificationModel:
    def __init__(self, message, ttl, creation_date, notification_category):
        # We will automatically generate the new id
        self.id = 0
        self.message = message
        self.ttl = ttl
        self.creation_date = creation_date
        self.notification_category = notification_category
        self.displayed_times = 0
        self.displayed_once = False
```

يقوم الصنف بالتصريح NotificationModel تابع باني __init__. يستقبل العديد من المدخلات تستخدمها لتهيئة الخصائص بنفس الأسماء: message و ttl و create_date و alert_category. يتم تعيين الخاصية ID على 0 ، والخاصية displayed-times على 0.

٥- إنشاء Class :

ثم ننشئ الصنف NotificationManager في ملف جديد نسميه service.py كما يبين الكود التالي:

```
from flask import Flask
from flask_restful import abort, Api, fields, marshal_with, reqparse,
Resource
from datetime import datetime
from models import NotificationModel
from http_status import HttpStatus
from pytz import utc
```

```
class NotificationManager():
    last_id = 0
    def __init__(self):
        self.notifications = {}

    def insert_notification(self, notification):
        self.__class__.last_id += 1
        notification.id = self.__class__.last_id
        self.notifications[self.__class__.last_id] = notification

    def get_notification(self, id):
        return self.notifications[id]

    def delete_notification(self, id):
        del self.notifications[id]


notification_fields = {
    'id': fields.Integer,
    'uri': fields.Url('notification_endpoint'),
    'message': fields.String,
    'ttl': fields.Integer,
    'creation_date': fields.DateTime,
    'notification_category': fields.String,
    'displayed_times': fields.Integer,
    'displayed_once': fields.Boolean
}


notification_manager = NotificationManager()


class Notification(Resource):
    def abort_if_notification_not_found(self, id):
        if id not in notification_manager.notifications:
            abort(
                HttpStatus.not_found_404.value,
                message="Notification {0} not found".format(id))

    @marshal_with(notification_fields)
    def get(self, id):
        self.abort_if_notification_not_found(id)
        return notification_manager.get_notification(id)

    def delete(self, id):
        self.abort_if_notification_not_found(id)
        notification_manager.delete_notification(id)
        return '', HttpStatus.no_content_204.value

    @marshal_with(notification_fields)
    def patch(self, id):
        self.abort_if_notification_not_found(id)
        notification = notification_manager.get_notification(id)
        parser = reqparse.RequestParser()
        parser.add_argument('message', type=str)
        parser.add_argument('ttl', type=int)
        parser.add_argument('displayed_times', type=int)
        parser.add_argument('displayed_once', type=bool)
        args = parser.parse_args()
```

^

```
        print(args)
        if 'message' in args and args['message'] is not None:
            notification.message = args['message']
        if 'ttl' in args and args['ttl'] is not None:
            notification.ttl = args['ttl']
        if 'displayed_times' in args and args['displayed_times'] is not
None:
            notification.displayed_times = args['displayed_times']
        if 'displayed_once' in args and args['displayed_once'] is not None:
            notification.displayed_once = args['displayed_once']
        return notification


class NotificationList(Resource):
    @marshal_with(notification_fields)
    def get(self):
        return [v for v in notification_manager.notifications.values()]

    @marshal_with(notification_fields)
    def post(self):
        parser = reqparse.RequestParser()
        parser.add_argument('message', type=str, required=True,
help='Message cannot be blank!')
        parser.add_argument('ttl', type=int, required=True, help='Time to
live cannot be blank!')
        parser.add_argument('notification_category', type=str,
required=True, help='Notification category cannot be blank!')
        args = parser.parse_args()
        notification = NotificationModel(
            message=args['message'],
            ttl=args['ttl'],
            creation_date=datetime.now(utc),
            notification_category=args['notification_category']
            )
        notification_manager.insert_notification(notification)
        return notification, HttpStatus.created_201.value


app = Flask(__name__)
service = Api(app)
service.add_resource(NotificationList, '/service/notifications/')
service.add_resource(Notification, '/service/notifications/<int:id>',
endpoint='notification_endpoint')


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

يصرح الصنف NotificationManager عن الخاصية last_id ويعطيها القيمة 0. تخزن خاصية الصنف هذه آخر ID تم إنشاؤه وإضافته إلى قاموس. يُنشئ التابع الباني __init__ صفة notifications ويعينها كقاموس فارغ.

تم إنشاء الميثود الثلاثة التالية :

insert_notification: يتلقى هذا الميثود NotificationModel تم إنشاؤه مؤخرًا. فيزيد الكود من قيمة الصفة last_id ثم تقوم بإسناد القيمة الناتجة إلى ID للإشعار المستلم ثم يضيف إشعارًا كقيمة إلى key المحدد بالID المُنشأ ، last_id ، في قاموس notifications.

get_notification: يتلقى هذا الميثود ID الإشعار الذي يجب أخذه من القاموس ويعيد القيمة المتعلقة بالkey الذي يطابق الID المستلم في القاموس الذي نستخدمه كمصدر للبيانات.

delete_notification: يتلقى هذه الميثود ID الإشعار الذي يجب إزالته من القاموس. يحذف الكود زوج ال key والقيمة الموافقة.

ثم يتم إنشاء قاموس باسم notification_fields نستخدمه للتحكم بالبيانات التي نريد من Flask-Restful أن يقدم ردود عندما يستقبل حالات NotificationModel ويضم القاموس الKeys والقيم التالية:

fields.Integer: يعطي خرج بقيمة عددية

fields.Url: يقوم بتوليد سلسلة من المحارف تمثل عنوان URL

fields.DateTime: يعطي التاريخ والوقت.

fields.Boolean: يعطي سلسلة من المحارف تمثل قيمة منطقية true or false

في الكود السابق تم تعريف الصنف Notification الذي يرث الصنف resource ويصرح عن ثلاث ميثودات يتم استدعاءها عندما تصل قيمة ميثود HTTP وهي get و delete و patch
تقوم الميثودات الثلاثة تلك باستدعاء الميثود abort_if_notification_not_found التي تستقبل ID لإشعار موجود مسبقا.

تستخدم كل من طريقتي get و patch ملحق @marshal_with الذي يأخذ قائمة من البيانات وينظمها. يمكن أن يعمل التنظيم أيضًا مع القواميس.

عندما نستخدم @marshal_with فإن حالة HTTP هي OK ويعيد الرقم 200

ثم عرفنا الصنف NotificationList الذي سنستخدمه لتمثيل مجموعة الإشعارات ويرث أيضا الصنف . resource

ثم نقوم بعمليات التوجيه الضرورية لاستدعاء الميثودات المناسبة وتمرير جميع الوسائط الضرورية عن طريق تحديد عناوين URL.

يتم ذلك من خلال إنشاء نقطة الدخول الرئيسية إلى تطبيق Flask وتهيئتها حيث يقوم الكود بإنشاء نموذج عن صنف flask_restful.Api وحفظه في متغيرات الخدمات حيث في كل استدعاء للميثود service.add_resource يقوم بتوجيه عنوان URL إلى المصدر.Flask سيقوم بتمرير ال URL إلى الميثود الذي يتم استدعاؤه.

٦- إجراء طلب HTTP إلى Flask API:

نقوم بتشغيل الكود الذي يعمل على تشغيل سيرفر flask لإنشاء وارسال طلبات HTTP إلى الAPI فنقوم بكتابة وتنفيذ السطر التالي في CMD :

```
python service/service.py
```

فيكون الناتج كما هو موضح في الشكل التالي حيث أن السيرفر يستمع إلى البوابة 5000:

```
 * Serving Flask app "service" (lazy loading)
 * Environment: production
   WARNING: Do not use the development server in a production
environment.
   Use a production WSGI server instead.
 * Debug mode: on
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 122-606-712
```

من خلال التعليمة السابقة يبدأ سيرفر التطوير الخاص ب Flask بالعمل ويأخذ الIP الخاص بالlocal host الذي هو 127.0.0.1 ومن غير الممكن الوصول إلى هذا ال IP من جهاز آخر موصول على شبكة LAN لذا في حال كنا نريد تشغيل أمر طلب http أو API من جهاز اخر نضبط السيرفر على عنوان IP 0.0.0.0

سيرفر التطوير الخاص بالFlask يعمل على العنوان 127.0.0.1ويستمع إلى البوابة 5000 وينتظر منا HTTP Request.

HTTPie: عبارة عن سطر أوامر HTTP Client مكتوب بلغة بايثون يسمح بارسال HTTP request بسهولة ومن أهم ميزاته أنه يعطي خرج ملون ويعطي أكثر من سطر لتوضيح الاستجابة. نفتح نافذة CMD جديدة ونكتب سطر الأوامر التالي من أجل إنشاء HTTP Request وانتج إشعار جديد:

```
http POST ":5000/service/notifications/" message='eSports competition
starts in 2 minutes' ttl=20 notification_category='Information'
```

ال Request السابق يحدد /service/notifications/ ويقوم بتشغيل الميثود NotificationList.post هذا الميثود لا يأخذ أي متغيرات لأن URL الموجه لا يحتوي أي بارامتر. بما أن أمر HTTP هو Post فإن Flask تستدعي الميثود Post فإذا تم إدارج الإشعار الجديد في القاموس في NotificationModel فإن التابع يعيد حالة HTTP ذات القيمة: 201 Created.

نقوم الآن بتشكيل وارسال HTTP Request لاسترداد كل الإشعارات فنكتب سطر الأوامر التالي:

```
http ":5000/service/notifications/"
```

الأمر السابق يقوم بارسال الأمر GET فيقوم Flask الميثود GET الذي يقوم بعرض كل الأغراض الموجودة في NotificationModel وتوليد استجابة JSON مع جميع الاشعارات التي قمنا بتوليدها ويكون لها الشكل التالي:

```
HTTP/1.0 200 OK
Content-Length: 648
Content-Type: application/json
Date: Wed, 16 june 2021 21:09:43 GMT
Server: /0.14.1 Python/3.8
[
{
"creation_date": "Fri, 16 june 2021 21:07:31 -0000",
"displayed_once": false,
"displayed_times": 0,
"id": 1,
"message": "eSports competition starts in 2 minutes",
"notification_category": "Information",
"ttl": 20,
"uri": "/service/notifications/1"
}]
```

إذا قمنا باستدعاء إشعار بقيمة غير موجودة مسبقا فإن الاستجابة هي:

```
HTTP/1.0 404 NOT FOUND
```

٧- الخاتمة

لقد صممنا RESTful Api للتفاعل مع قاموس بسيط لتخزين البيانات وأجرى عمليات CRUD مع الإشعارات، لاستخدامها كأساس لخدمة مصغرة. حددنا متطلبات واجهة برمجة التطبيقات الخاصة بنا وفهمنا المهام التي تؤديها كل طريقة من طرق HTTP. أنشأنا بيئة افتراضية مع Flask و Flask-RESTful.

المراجع المستخدمة

Gaston C. Hillar. ***Developing RESTful APIs and Microservices with Flask 1.0.2***

www.github.com/Pact / Hands-On-RESTful-Python-Web-Services-Second-Edition-master