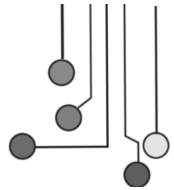

REST avanzado

Sistemas de Información Orientados a Servicios

RODRIGO SANTAMARÍA

OAuth

Flask

Apache

REST avanzado

OAuth

- DEFINICIÓN: ROLES Y MODOS
- OAUTH 2.0
- CLAVE Y PALABRA SECRETA
- COMPOSICIÓN Y CODIFICACIÓN
- TOKEN DE ACCESO
- USO

APIs con autenticación

- Hoy en día casi toda API requiere registro
  - Ligera: correo electrónico
  - Dura: DNI, tarjeta de crédito, proyecto, validación
- Tras registrarnos obtenemos una clave de acceso (**API key**) que debemos incluir en nuestras llamadas a los servicios de la API
- En autenticaciones más complejas, recibimos unas **credenciales** con las que podemos obtener un **token de acceso** (puede ser temporal, revocable)

APIs con autenticación: OAuth

- OAuth es un protocolo abierto para autorización
  - Permite que un **proveedor** de un recurso garantice el acceso a un **cliente**, previa autorización de su **propietario**
    - P. ej. permite que Facebook (*proveedor*) garantice el acceso a nuestros datos (*propietario*) a otra aplicación (*cliente*)
  - Sin que el propietario deba compartir su clave con el cliente
  - Diseñado sobre HTTP (intercambio de mensajes via REST)
  - En esencia implica la emisión de un 'permiso' (*access token*) para acceder a los datos propietarios del proveedor

OAuth: roles

Rol	“Sinónimo”	Descripción
Cliente	Aplicación	Aplicación que está intentando acceder a los recursos del usuario
Proveedor (del recurso)	API	El servidor que provee la API por la que se accede a los recursos del usuario
Propietario (del recurso)	Usuario	Propietario del recurso que debe dar acceso a una porción de su cuenta

OAuth: modalidades*

- Autenticación de 2 patas (aplicación=usuario)
  - Permite hacer peticiones al proveedor en nombre del cliente, pero sin hacerlo en nombre de ningún usuario
- OAuth de 3 patas
  - El modo normal, permite al cliente hacer peticiones al proveedor en nombre de un usuario (previo consentimiento)
  - *OAuth basado en PIN*: modalidad para clientes sin acceso a un navegador web (órdenes de consola, sistemas empotrados, etc.)
- OAuth Echo
  - OAuth de “4 patas”, añade el rol del delegador, un proveedor secundario que actúa a través del proveedor primario

* Twitter cuenta con una buena explicación de estas y otras modalidades

OAuth 2.0: Funcionamiento

- OAuth2: A Tale of Two servers
  - https://www.youtube.com/watch?v=tFYrq3d54Dc

Ejemplo: Twitter y OAuth de aplicación (1)

- **(1) Handshake** o ‘registra tu app’
  1. Conectarse con una cuenta de Twitter
  2. Crear una nueva aplicación (http://apps.twitter.com)
     - Se le asignará una clave y palabra secreta (credenciales)
     - No deben compartirse (especialmente la palabra secreta)
     - Clave y palabra secreta deben concatenarse con ‘:’ y codificarse en Base64* para compartirse mediante servicios web
       - Ejemplo
         - Clave: `xvz1evFS4wEEPTGEFPHBog`
         - Palabra secreta: `L8qq9PZyRg6ieKGEKhZolGC0vJWLw8iEJ88DRdyOg`
         - Codificación (p. ej. usando comando OpenSSL):

```
$ openssl enc -base64 <<<
xvz1evFS4wEEPTGEFPHBog:L8qq9PZyRg6ieKGEKhZolGC0vJWLw8iEJ88DRdyOg
eHZ6MWV2RlM0d0VFUFRHRUZQSEJvZzpMOHFxOVBaeVJnNmllS0dFS2hab2xHQzB2
SldMdzhpRUo4OERSZHlPZwo=
```

*¿Por qué Base64?

Ejemplo: Twitter (2)

- **(2) Petición de token de acceso**: POST a `oauth2/token`
  - Ejemplo 1 (`curl`)

```
curl --request 'POST' 'https://api.twitter.com/oauth2/token' --header
'Authorization: CREDENCIALES, Content-Type: application/x-www-form-
urlencoded;charset=UTF-8' --data "grant_type=client_credentials" –verbose –k
```

    - `CREDENCIALES` es la “clave:palabra” codificada en Base64
    - `-k` hace la conexión SSL sin certificado

  - Ejemplo 2 (`curl`, no hace falta codificación Base 64 previa)

```
curl -X POST --verbose "https://api.twitter.com/oauth2/token" -d
"grant_type=client_credentials" -u clave:palabraSecreta –k
```

  - Obtendremos una respuesta JSON con el token de acceso
    - {"token_type":"bearer","access_token":"AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA%2FAAAAAAAAAAAAAAAAAAAA%3DAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"}

Ejemplo: Twitter (3)

- **(3) Peticiones posteriores**: añadir una cabecera `Authorization` con el token a todas las invocaciones de servicios que soporten OAuth a nivel de aplicación
  - Ejemplo (curl):

```
curl --request 'GET'
'https://api.twitter.com/1.1/statuses/user_timeline.json?
count=100&screen_name=twitterapi' --header 'Authorization: Bearer
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA%2FAAAAAAAAAAAAAAAAAAAAA
%3DAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA' –k
```

  - Si el token se ve comprometido, podemos invalidarlo con una llamada POST a `oauth2/invalidate_token`
  - Para más detalles y FAQ, ver:
    - https://dev.twitter.com/docs/auth/application-only-auth
  - Para OAuth the 3 patas, ver:
    - https://dev.twitter.com/oauth/3-legged

OAuth y el mundo

- Prácticamente toda API bien establecida mantiene una política similar a la de Twitter
  - Facebook, Google, Instagram, Amazon, Reddit, Dropbox…
  - http://en.wikipedia.org/wiki/OAuth#List_of_OAuth_service_providers
- Si bien OAuth no es ni mucho menos perfecto*, se ha convertido en un estándar de facto
  - Cada implementación lidia con sus defectos

*Hammer, E. *Oauth 2.0 and the Road to Hell*. Recuperada con Internet Wayback Machine (originalmente en hueniverse.com) **2012**

Flask

13

INTRODUCCIÓN
GET, POST, PUT, DELETE
CORS
SUBIDA DE ARCHIVOS
MONTAJE EN SERVIDOR

Flask

- Entorno de desarrollo web muy simple para Python*
  - Permite implementar servicios REST
  - Similar a Jersey (Java)
    - Anotaciones en los métodos que implementan servicios
    - Ventaja: no necesita un soporte mediante Tomcat**
  - Instalación (ya instalado en el aula SUN):
    - `$pip install flask`
- Aprenderemos a:
  - Implementar operaciones CRUD (GET, POST, PUT, DELETE)
    - http://blog.miguelgrinberg.com/post/designing-a-restful-api-with-python-and-flask
  - Establecer cabeceras de seguridad CORS
    - http://mortoray.com/2014/04/09/allowing-unlimited-access-with-cors/
  - Subir archivos a un servidor
    - http://flask.pocoo.org/docs/patterns/fileuploads/

* Tutorial muy sencillo de python: https://www.codecademy.com/learn/python
Una chuleta con operaciones frecuentes en Python:
http://vis.usal.es/rodrigo/documentos/soa/additional/pythonCheatsheet.pdf

** Aunque para producción se recomienda un servidor tipo Apache o Nginx gestionado mediante wgsi

Flask: primera aplicación (*app.py*)


```
#!flask/bin/python
from flask import Flask

app = Flask(__name__)

@app.route('/')
def index():
    return "Hello, World!"

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

1) Instanciamos una aplicación de Flask

2) Definimos un servicio

3) Ejecutamos la aplicación

nos va a permitir realizar cambios en el código sin relanzar la aplicación, además de mostrar más mensajes informativos*

* También podemos indicarle host y port (api entera aquí)

Flask: primera aplicación (*app.py*)


```
$ chmod a+x app.py
$ ./app.py
 * Running on http://127.0.0.1:5000/
 * Restarting with reloader
```

1) Permisos de ejecución

2) Lanzamos la aplicación

cualquier mensaje adicional (peticiones aceptadas, errores, etc.) aparecerá aquí

- Testeo:
  - Navegador web: http://localhost:5000
  - Consola: `curl –i http://localhost:5000`

incluye la cabecera HTTP en la salida, a modo informativo

Flask: aumentando las opciones


```
#!flask/bin/python
from flask import Flask, jsonify

app = Flask(__name__)

tasks = [
    {
        'id': 1,
        'title': u'Buy groceries',
        'description': u'Milk, Cheese, Pizza, Fruit, Tylenol',
        'done': False
    },
    {
        'id': 2,
        'title': u'Learn Python',
        'description': u'Need to find a good Python tutorial on the
 web',
        'done': False
    }
]

@app.route('/todo/api/v1.0/tasks', methods = ['GET'])
def get_tasks():
    return jsonify( { 'tasks': tasks } )

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

1) Soporte para JSON

2) URI del servicio

3) Tipo de petición

Podemos usar variables globales (la implementación de REST en Flask es con estado)

Flask: múltiples URIs y errores

1) Soporte para errores

```
from flask import abort

@app.route('/todo/api/v1.0/tasks/<int:task_id>', methods = ['GET'])
def get_task(task_id):
    task = filter(lambda t: t['id'] == task_id, tasks)
    if len(task) == 0:
        abort(404)
    return jsonify( { 'task': task[0] } )
```

2) URI variable

3) Se recomienda usar el mismo nombre en la URI que como argumento

```
from flask import make_response

@app.errorhandler(404)
def not_found(error):
    return make_response(jsonify( { 'error': 'Not found' } ), 404)
```

`abort` *retorna el error mediante HTTP, para dar un error más informativo, usamos* `errorhandler` *y* `make_response`

Flask: argumentos y *request*

1) Soporte para peticiones

```
from flask import stream_with_context, request, Response

@app.route('/stream')
def streamed_response():
    @stream_with_context
    def generate():
        yield 'Hello '
        yield request.args['name']
        yield '!'
    return Response(generate())
```

2) Ni el URI ni la función tienen argumentos

3) `request.args` *es un diccionario con todos los parámetros pasados a la URL*

Flask: POST

1) Soporte para peticiones

```
from flask import request

@app.route('/todo/api/v1.0/tasks', methods = ['POST'])
def create_task():
    if not request.json or not 'title' in request.json:
        abort(400)
    task = {
        'id': tasks[-1]['id'] + 1,
        'title': request.json['title'],
        'description': request.json.get('description', ""),
        'done': False
    }
    tasks.append(task)
    return jsonify( { 'task': task } ), 201
```

2) tipo de método: POST

3) Comprobamos que la petición está codificada en JSON y tiene un campo 'title'

tomamos `title` (requerido) y `description` permitimos que quede en blanco si no está en el `request`

retornamos la tarea en JSON y el código HTTP que indica 'creado'

testeo con `curl`

```
$ curl -i -H "Content-Type: application/json" -X POST -d '{"title":"Read a book"}' h
ttp://localhost:5000/todo/api/v1.0/tasks
```

Flask: POST


```
$ curl -i -H "Content-Type: application/json" -X POST -d '{"title":"Read a book"}' h
ttp://localhost:5000/todo/api/v1.0/tasks
```

- Opciones de `curl`:
  - `-H` para añadir cabecera HTTP de la petición (p. ej. “el contenido es de tipo aplicación en formato JSON”)
    - http://en.wikipedia.org/wiki/List_of_HTTP_header_fields
  - `-X` para indicar el tipo de petición (GET, POST, PUT, DELETE)
  - `-d` para añadir datos a la petición en el formato indicado en la cabecera
  - `-k` para evitar el uso de certificados SSL (inseguro)

Flask: PUT


```
@app.route('/todo/api/v1.0/tasks/<int:task_id>', methods = ['PUT'])
def update_task(task_id):
    task = filter(lambda t: t['id'] == task_id, tasks)
    if len(task) == 0:
        abort(404)
    if not request.json:
        abort(400)
    if 'title' in request.json and type(request.json['title']) != unicode:
        abort(400)
    if 'description' in request.json and type(request.json['description']) is not un
icode:
        abort(400)
    if 'done' in request.json and type(request.json['done']) is not bool:
        abort(400)
    task[0]['title'] = request.json.get('title', task[0]['title'])
    task[0]['description'] = request.json.get('description', task[0]['description'])
    task[0]['done'] = request.json.get('done', task[0]['done'])
    return jsonify( { 'task': task[0] } )
```

Para modificar una tarea en la posición `task_id`

Obtenemos la tarea actual

Control exhaustivo de errores

testeo con `curl`

```
$ curl -i -H "Content-Type: application/json" -X PUT -d '{"done":true}' http://local
host:5000/todo/api/v1.0/tasks/2
```

Flask: DELETE


```
@app.route('/todo/api/v1.0/tasks/<int:task_id>', methods = ['DELETE'])
def delete_task(task_id):
    task = filter(lambda t: t['id'] == task_id, tasks)
    if len(task) == 0:
        abort(404)
    tasks.remove(task[0])
    return jsonify( { 'result': True } )
```

Obtenemos la tarea actual

Flask no es la única opción

- Han proliferado muchas bibliotecas para servicios web en Python
  - Flask no es la más rápida
  - Flask no es usable en producción directamente
    - (Ver más abajo: Flask y Apache)
  - Flask es sencilla y no requiere bibliotecas adicionales para funcionar
  - Otras opciones: **fastapi**, tornado, uvicorn...

CORS

- Política de Seguridad ‘Same Origin’
  - Un navegador web sólo permite que los scripts de una web A hagan peticiones a dicha web
    - Entendiendo origen como host+puerto+ruta
  - Evita que un software malicioso de una web haga peticiones sensibles a otra web
  - Es una política poco flexible y limitante en algunos casos
- Cross Origin Resource Sharing
  - Estándar para hacer peticiones o usar recursos en un origen distinto al que lo genera (cross origin)
  - Definición de nuevas cabeceras HTTP
    - En cabeceras de petición: `Origin`
    - En cabeceras de respuesta: `Access-Control-Allow-Origin`

CORS: Ejemplo


```
GET whatever
Origin: a.com
```

200 OK

a.com

Same origin

```
GET whatever
Origin: a.com
```

[1]

b.com

Cross origin

```
Access-Control-Allow-Origin: a.com

Access-Control-Allow-Origin: *
```

Información para que el navegador decida si continúa con la petición o abandona

[1] Adicionalmente, pueden agregarse a la cabecera los campos `Access-Control-Request-Method` y `Access-Control-Request-Headers` para indicar qué tipo de petición (GET,POST, etc) y cabeceras (Content-Type, Accept, etc) se realizará. El servidor b.com responderá con la lista de peticiones y cabeceras que permite, en los mismos campos.

CORS y Flask

- Añadimos un método que se ejecuta con cada una de las peticiones para añadir campos a la cabecera

```
#from http://mortoray.com/2014/04/09/allowing-unlimited-access-with-cors/
@app.after_request
def add_cors(resp):
    """ Ensure all responses have the CORS headers. This ensures any failures
        are also accessible by the client. """
    resp.headers['Access-Control-Allow-Origin'] = request.headers.get('Origin')
    resp.headers['Access-Control-Allow-Credentials'] = 'true'
    resp.headers['Access-Control-Allow-Methods'] = 'POST, OPTIONS, GET'
    resp.headers['Access-Control-Allow-Headers'] =
        request.headers.get('Access-Control-Request-Headers', 'Authorization' )
    # set low for debugging
    if app.debug:
        resp.headers['Access-Control-Max-Age'] = '1'
    return resp
```

CORS y Flask (ii)

- Biblioteca flask_cors*

```
$pip install -U flask-cors
```

```
from flask import Flask
from flask_cors import CORS

app = Flask(__name__)
CORS(app)

@app.route("/")
def helloWorld():
  return "Hello, cross-origin-world!"
```

* https://flask-cors.readthedocs.io/en/latest/

Blueprints

- Permiten definir servicios web en archivos distintos al que ejecuta app:

main.py

```
from flask import Flask
from metodos import metodos_blueprint

app=Flask(__name__)
app.register_blueprint(metodos_blueprint)

@app.route('/')
def index():
    return "Hola mundo  qué tal!"

app.run(debug=True)
```

metodos.py

```
from flask import Blueprint

metodos_blueprint=Blueprint("metodos", __name__)

@metodos_blueprint.route("otroHola")
def otroHola():
    return "Otro Hola!!"
```

* https://flask-cors.readthedocs.io/en/latest/

Subida de archivos

- En ciertos servicios, tratamos con gran cantidad de datos o archivos pesados propiedad del frontend
  - El frontend puede no ser tan potente como para procesarlos
  - Los datos se intercambian continuamente con el backend
    - Solución para evitar tráfico: subir archivos al backend

Subida de archivos en Flask


```
import os
from flask import Flask, request, redirect, url_for
from werkzeug.utils import secure_filename

UPLOAD_FOLDER = '/path/to/the/uploads'
ALLOWED_EXTENSIONS = set(['txt', 'pdf', 'png', 'jpg', 'jpeg', 'gif'])

app = Flask(__name__)
app.config['UPLOAD_FOLDER'] = UPLOAD_FOLDER
```

- `UPLOAD_FOLDER`: ruta en el backend donde se almacenan los archivos subidos
- `ALLOWED_EXTENSIONS`: formatos permitidos
  - Importante limitar los formatos, para evitar problemas de inyección (XML, SQL) o scripts maliciosos (PHP, archivos de código en general)

Subida de archivos en Flask


```
def allowed_file(filename):
    return '.' in filename and \
           filename.rsplit('.', 1)[1] in ALLOWED_EXTENSIONS

@app.route('/', methods=['GET', 'POST'])
def upload_file():
    if request.method == 'POST':
        file = request.files['file']
        if file and allowed_file(file.filename):
            filename = secure_filename(file.filename)
            file.save(os.path.join(app.config['UPLOAD_FOLDER'], filename))
            return redirect(url_for('uploaded_file',
                                    filename=filename))
    return '''
    <!doctype html>
    <title>Upload new File</title>
    <h1>Upload new File</h1>
    <form action="" method=post enctype=multipart/form-data>
      <p><input type=file name=file>
         <input type=submit value=Upload>
    </form>
    '''
```

retorna un valor válido si el fichero tiene una extensión permitida

convierte el nombre del archivo a un nombre seguro para evitar ataques basados en rutas

retorna la ruta en la que se guarda el fichero (POST) o un formulario para que se escoja un fichero a subir mediante POST

Nombres de archivo y seguridad

- Un ataque malicioso es enviar rutas relativas que acceden a niveles superiores del árbol de directorios
  - `filename = "../../../../home/username/.bashrc”`
  - ¡Si acierta con el número de `../` puede estar sobreescribiendo nuestro profile!
- La función `secure_filename` simplemente formatea este tipo de rutas:

```
>>> secure_filename('../../../../home/username/.bashrc')
'home_username_.bashrc'
```

Servir archivos y limitar tamaño

- Servicio para recuperar archivos subidos

```
from flask import send_from_directory

@app.route('/uploads/<filename>')
def uploaded_file(filename):
    return send_from_directory(app.config['UPLOAD_FOLDER'],
                               filename)
```

- Por defecto, se admiten archivos de cualquier tamaño, pero si queremos limitar sólo tenemos que usar una variable

```
from flask import Flask, Request

app = Flask(__name__)
app.config['MAX_CONTENT_LENGTH'] = 16 * 1024 * 1024
```

Barras de progreso

- No entraremos en detalle, pero hay muchas opciones en distintos lenguajes
  - Plupload (HTML5, Java, Flash)
  - SWFUpload (Flash)
  - JumpLoader (Java)

Flask y Apache*

- El servidor de Flask está bien para pruebas pero es poco fiable en producción.
- Tareas:

  1) Instalar Apache en un servidor UNIX
  2) Definir un archivo WSGI
  3) Configuración del sitio web en Apache
  4) Reiniciar Apache: `sudo apache2 restart`

*Existen otras opciones aparte de Apache, como Nginx. Recomiendo Apache porque es software libre y de probada solidez

Flask y Apache: WSGI

- Web Service Gateway Interface
  - Es una convención sencilla para redirigir servicios web a aplicaciones python
  - Simplemente es un archivo que indica dónde está mi aplicación python:

```
import sys
sys.path.insert(0, "/path/to/my/package")
from myPackage import myApp as application
```

  - Generalmente se ubican en `/var/www`

Flask y Apache: conf

- Archivo de configuración de Apache que indica dónde y cómo se escuchan y redirigen las peticiones de nuestro servicio web
- Ubicado en `/etc/apache2/sites-available`
  - Enlace simbólico en `/etc/apache2/sites-enabled` cuando queramos activarlo

Flask y Apache: conf (ii)


```
Listen 9090
<VirtualHost *:9090>
    WSGIDaemonProcess userX user=userX group=groupX threads=5
    WSGIScriptAlias / /var/www/userX/file.wsgi

    Options FollowSymLinks
    <Directory /var/www/userX>
        WSGIProcessGroup groupX
        WSGIApplicationGroup %{GLOBAL}
        WSGIScriptReloading On
        Order deny,allow
        Allow from all
       #Require all granted
       #CustomLog /home/userX/logs/access.log
       ErrorLog /home/userX/logs/error.log
    </Directory>
</VirtualHost>
```

Puerto de escucha en host virtual

Ruta al wsgi que enlaza con nuestro código python

Usuario y grupo en el que se ejecutan los procesos que sirven las peticiones, así como el número de hilos máximo que sirven de manera concurrente

Permite cambios on-fly en el script python sin necesidad de reiniciar Apache

Archivos de log para errores o salida estándar (cualquier línea se puede comentar con #)

Resumen

- **OAuth** es un método de autorización para que un servicio pueda tener identificado al consumidor que accede a sus servicios
- Consiste en un intercambio previo de mensajes que nos garantiza un **access token** que incluiremos en nuestras invocaciones de servicios
- Toda API actual ha migrado o está migrando hacia este tipo de autorización

- **Flask** es un entorno similar a Java-RX para desarrollar servicios REST en **python**
- Funciona a través de anotaciones o decoraciones (**@app.xxx**)
- Permite realizar cualquier operación **CRUD** para servicios RESTful (GET, POST, DELETE, UPDATE)
- Permite también una autenticación **CORS** a nivel de aplicación, y subida de archivos al servidor.

Referencias

- Aaron Parecki. OAuth 2 simplified. 2012.
  - https://aaronparecki.com/2012/07/29/2/oauth2-simplified
- Twitter. OAuth documentation
  - https://dev.twitter.com/oauth
- Eran Hammer. OAuth 2.0 and the road to hell. 2012
  - https://web.archive.org/web/20130325140509/http://hueniverse.com/2012/07/oauth-2-0-and-the-road-to-hell/
- OAuth2: A tale of two servers (vídeo)
  - https://www.youtube.com/watch?v=tFYrq3d54Dc
- Flask. Página oficial
  - http://flask.pocoo.org/
  - Otros recursos listados en la diapositiva 15