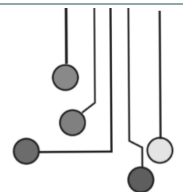

REST avanzado

Sistemas de Información Orientados a Servicios

RODRIGO SANTAMARÍA

OAuth

Flask

REST avanzado

Objetivo

- En Sistemas Distribuidos vimos cómo:
  - invocar un servicio REST desde un navegador/consola/Java
  - implementar un servicio REST en Java mediante Jersey
- Ahora veremos cómo
  - Autenticar aplicaciones para usar APIs con seguridad OAuth
  - Implementar un servicio REST desde Python con Flask

OAuth

DEFINICIÓN
CLAVE Y PALABRA SECRETA
COMPOSICIÓN Y CODIFICACIÓN
TOKEN DE ACCESO
USO

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 del propietario del recurso
    - P. ej. permite que Facebook (*proveedor*) garantice el acceso a nuestros datos de usuario (*recurso*) a otra aplicación (*cliente*)
  - En la práctica, generalmente implica un intercambio de mensajes y la emisión de un 'permiso' (*access token*) previamente al uso de servicios REST
  - **Objetivo**: identificar a toda aplicación que use la API
- OAuth2: A Tale of Two servers
  - https://www.youtube.com/watch?v=tFYrq3d54Dc

Obteniendo autorización en Twitter

- Instrucciones para el acceso mediante OAuth
  1. Obtener una clave y palabra secreta
     - *De aplicación*: buscar tweets, acceder a amigos de cualquier cuenta, obtener timelines de usuario
     - *De usuario*: postear tweets, acceder a detalles de la cuenta, obtener datos de geolocalización
  2. Componer y codificar la credencial con la clave+palabra
  3. Obtener el token de acceso
  4. Autenticar las llamadas a la API con el token de acceso

1) obtener clave/palabra secreta

- https://dev.twitter.com/docs/auth/obtaining-access-tokens
  1. Conectarse con una cuenta de Twitter
  2. Crear una nueva aplicación
     - Se le asignará una clave y palabra secreta
     - Deben tratarse como una contraseña:
       - No son legibles
       - No deben compartirse (especialmente la palabra secreta)
- Siguientes pasos
  - Nivel de usuario:
    - https://dev.twitter.com/docs/auth/tokens-devtwittercom
  - Nivel de aplicación (lo más normal):
    - https://dev.twitter.com/docs/auth/application-only-auth

2) Componer y codificar

- Clave y palabra secreta deben concatenarse con ‘:’ y codificarse en Base64
  - Ejercicio: ¿Por qué Base64?
  - Ejemplo
    - Clave: `xvz1evFS4wEEPTGEFPHBog`
    - Palabra secreta: `L8qq9PZyRg6ieKGEKhZolGC0vJWLw8iEJ88DRdyOg`
    - Composición:
      - `xvz1evFS4wEEPTGEFPHBog:L8qq9PZyRg6ieKGEKhZolGC0vJWLw8iEJ88DRdyOg`
    - Codificación (p. ej. usando comando UNIX):

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

3) Obtener el token de acceso

- Realizar la una petición POST a `oauth2/token`
  - Ejemplo 1 (`curl`)

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

    - `TOKEN` es la composición codificada en Base64 de clave: palabra secreta
    - `-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":"AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA%2FAAAAAAAAAAAAAAAAAAAA%3DAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"}

4) Utilizar el token de acceso

- 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
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA%2FAAAAAAAAAAAAAAAAAAAAA
%3DAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA' -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

OAuth y el mundo

- Prácticamente toda API bien establecida mantiene una política como la que hemos visto para Twitter
  - Facebook, Google, Instagram, Amazon, Reddit, Dropbox…
  - http://en.wikipedia.org/wiki/OAuth#List_of_OAuth_service_providers

Flask

12

INTRODUCCIÓN
GET, POST, PUT, DELETE
CORS
SUBIDA DE ARCHIVOS

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:
    - `$pip install flask`
- Aprenderemos:
  - 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/

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

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: argumentos 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) Argumentos en la URI

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: POST

1) Soporte para peticiones

2) tipo de método: POST

```
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
```

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

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

CORS

- Cross Origin Resource Sharing
  - Estándar para usar recursos de un dominio en un dominio distinto al que lo creó (cross origin)
  - Alternativa a
    - All origins → permite todas las comparticiones entre dominios
      - Poco seguro
    - Same origin → permite sólo usar en el dominio en que se creó
      - Poco flexible
- Definición de nuevas cabeceras HTTP
  - En cabeceras de petición: `Origin`
  - En cabeceras de respuesta: `Access-Control-Allow-Origin`

CORS: Ejemplo

- Sea una página http://www.socialBooks.com que intenta acceder a los datos en http://www.socialNetwork.com
  - Si el navegador del usuario implementa CORS (lo normal), añadirá a la petición de `socialBooks.com` un campo:
    - `Origin: http://www.socialBooks.com`
  - Si `socialNetwork.com` acepta la petición retornará la respuesta con el siguiente campo en la cabecera:
    - `Access-Control-Allow-Origin: http://www.socialBooks.com`
    - También podría garantizar acceso a cualquier origen (no recomendado)
      - `Access-Control-Allow-Origin: *`
- Evita que, a nivel de navegador, una página se haga pasar por otra para acceder a servicios ~ OAuth a otro nivel

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
```

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

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)

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

- Nicolai M. Josuttis. *SOA in practice. The Art of Distributed System Design*. O'Reilly, **2007**. Ch 1/2.

- Grady Booch. *Avoid the 'stupid' SOA aproach*. GCN, **2006**.
  - http://gcn.com/Articles/2006/07/12/Grady-Booch--Avoid-the-stupid-SOA-approach.aspx?Page=1

- OASIS. *Reference model for service oriented architectures*. Commitee Draft 1.0, **2007**
  - http://xml.coverpages.org/SOA-RM-ReferenceModel200602-CD.pdf