

Servicios Rest Flask

Antonio Espín Herranz

Introducción

- El estilo REST *es una forma ligera de crear Servicios Web.*
- Se basan en las URLs.
- Proporcionan acceso a URLs para obtener información o realizar alguna operación.
- Son interesante para utilizar con **peticiones** de tipo **AJAX** y para acceder con **dispositivos con poco recursos.**

Características

- Sistema cliente / servidor.
- No hay estado → sin sesión.
- Soporta un sistema de caché
- ***Cada recurso tendrá una única dirección de red.***
- Sistema por capas.
- Variedad de formatos:
 - XML, HTML, text plain, JSON, etc.

Recursos

- Un recurso REST es cualquier cosa que sea direccionable a través de la Web.
- Algunos ejemplos de recursos REST son:
 - Una noticia de un periódico
 - La temperatura de Alicante a las 4:00pm

Algunos formatos soportados

Formato	Tipo MIME
Texto plano	text/plain
HTML	text/html
XML	application/xml
JSON	application/json

URI

- Una URI, o **Uniform Resource Identifier**, en un servicio web RESTful es un hiper-enlace a un recurso, y es la única forma de intercambiar representaciones entre clientes y servidores.
- Un servicio web RESTful expone un conjunto de recursos que identifican los objetivos de la interacción con sus clientes.

Formato de las peticiones

- Las peticiones REST tienen un formato con este:
- `http://localhost:8080/app/trabajadores/101`
- **trabajadores: representa un recurso.**
- **101:** El identificador del Trabajador, es el equivalente a `.../trabajadores?id=101`
- La URL de REST está orientada a recursos y localiza un recurso.

Verbos REST

- Los verbos nos permiten llevar a cabo acciones con los recursos.
- Se asocian con las operaciones **CRUD**.
 - **GET**: Obtener información sobre un recurso. El recurso queda identificado por su URL. **Operación read**.
 - **POST**: Publica información sobre un recurso. **Operación create**.
 - **PUT**: Incluye información sobre recursos en el Servidor. **Operación update**.
 - **DELETE**: Elimina un recurso en el Servidor. **Operación delete**.

REST vs SOAP

	REST	SOAP
Características	<p>Las operaciones se definen en los mensajes.</p> <p>Una dirección única para cada instancia del proceso.</p> <p>Cada objeto soporta las operaciones estándares definidas.</p> <p>Componentes débilmente acoplados.</p>	<p>Las operaciones son definidas como puertos WSDL.</p> <p>Dirección única para todas las operaciones.</p> <p>Múltiple instancias del proceso comparten la misma operación.</p> <p>Componentes fuertemente acoplados.</p>
Ventajas declaradas	<p>Bajo consumo de recursos.</p> <p>Las instancias del proceso son creadas explícitamente.</p> <p>El cliente no necesita información de enrutamiento a partir de la URI inicial.</p> <p>Los clientes pueden tener una interfaz “listener” (escuchadora) genérica para las notificaciones.</p> <p>Generalmente fácil de construir y adoptar.</p>	<p>Fácil (generalmente) de utilizar.</p> <p>La depuración es posible.</p> <p>Las operaciones complejas pueden ser escondidas detrás de una fachada.</p> <p>Envolver APIs existentes es sencillo</p> <p>Incrementa la privacidad.</p> <p>Herramientas de desarrollo.</p>
Posibles desventajas	<p>Gran número de objetos.</p> <p>Manejar el espacio de nombres (URIs) puede ser engorroso.</p> <p>La descripción sintáctica/semántica muy informal (orientada al usuario).</p> <p>Pocas herramientas de desarrollo.</p>	<p>Los clientes necesitan saber las operaciones y su semántica antes del uso.</p> <p>Los clientes necesitan puertos dedicados para diferentes tipos de notificaciones.</p> <p>Las instancias del proceso son creadas implícitamente.</p>

¿Dónde es útil REST?

- El servicio Web no tiene estado.
- Tanto el productor como el consumidor del servicio conocen el contexto y contenido que va a ser comunicado
- El ancho de banda es importante y necesita ser limitado.
 - REST es particularmente útil en dispositivos con escasos recursos como PDAs o teléfonos móviles
- Los desarrolladores pueden utilizar tecnologías como **AJAX**

¿Dónde es útil SOAP?

- Se establece un contrato formal para la descripción de la interfaz que el servicio ofrece → WSDL.
- La arquitectura necesita manejar procesamiento asíncrono e invocación.

Librerías python para REST

- Instalar la librería: **Flask**
 - **pip install flask**
 - **pip install flask-restful**
- Viene con un servidor para poder publicar los servicios.
- También suministra clientes para consumir los servicios.
- Es un **framework** para implementar **APIs de REST, ligero y eficiente**.
- Cumple con el **estándar WSGI**, se puede utilizar con servidores **compatibles** con WSGI, por ejemplo, **Apache**.
- Independiente de la BD.
- Arranca un servidor local en el puerto 5000.

Ejemplo

```
from flask import Flask
from flask_restful import Resource, Api

app = Flask(__name__)
api = Api(app)

class HelloWorld(Resource):
    def get(self):
        return {"hello": "world"}

api.add_resource(HelloWorld, '/')

if __name__ == '__main__':
    app.run(debug=True) # Petición: http://localhost:5000
```

Enrutamiento

- El componente principal proporcionado por Flask-RESTful son los **recursos**.
- Los recursos se construyen sobre las vistas conectables de Flask , lo que le brinda fácil acceso a múltiples métodos HTTP simplemente definiendo métodos en su recurso.
- Permite la implementación de recursos **CRUD**.

Ejemplo: enrutamiento GET/PUT

```
app = Flask(__name__)  
api = Api(app)
```

```
todos = {} # Diccionario global para simular la BD
```

```
class ApiSimple(Resource):
```

```
    def get(self, id):  
        return {id: todos[id]}
```

```
    def put(self, id):  
        todos[id] = request.form['data']  
        return {id: todos[id]}
```

```
api.add_resource(ApiSimple, '/<string:id>') # Definición del parámetro <tipo:nombre>
```

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

Testear el servicio

- Desde postman
 - <https://www.postman.com/downloads/>
- Desde la consola con el comando **curl**:
 - **Comando** de Windows / Linux
- Desde la librería requests de Python
 - **pip install requests**

Postman

- Para realizar peticiones POST, PUT, etc. en **postman**: enviar los parámetros como json.

The screenshot displays the Postman interface for a PUT request. The URL bar at the top shows `http://localhost:5000/todo4`. The method dropdown is set to **PUT**. The **Body** tab is selected, and the **raw** radio button is chosen. The body content is a JSON object: `{ "todo4": "ejemplo de put" }`. The **Send** button is visible in the top right. Below the body editor, the response is shown as `200 OK` with a status of `6 ms` and a size of `198 B`. The response body is displayed in the **Body** tab, showing the same JSON object.

http://localhost:5000/todo4

Save

PUT

http://localhost:5000/todo4

Send

Params Authorization Headers (8) **Body** Pre-request Script Tests Settings Cookies

none form-data x-www-form-urlencoded raw binary GraphQL

	KEY	VALUE	DESCRIPTION	...	Bulk Edit
<input checked="" type="checkbox"/>	data	ejemplo de put			
	Key	Value	Description		

Body Cookies Headers (5) Test Results

200 OK 6 ms 198 B Save Response

Pretty Raw Preview Visualize JSON

```
1 {
2   "todo4": "ejemplo de put"
3 }
```

Postman: post en json

The screenshot shows the Postman interface for a POST request to `http://localhost:5000/productos/`. The request is highlighted with a red box. The body is set to JSON and contains the following JSON object:

```
1 [{"id":-1,"nombre":"Video","precio":34.6,"exis":15}]
```

The response is also highlighted with a red box. It shows a 200 OK status with a response time of 6 ms and a body of 199 B. The response body is displayed in the 'Body' tab, showing a JSON object with a 'create' property set to 1.

```
1 [
2   {
3     "create": 1
4   },
5 ]
```

En consola

- Petición **PUT**:
- **curl** http://localhost:5000/**todo1** -d "**data**=ejemplo de put" -X PUT
- Petición **GET**:
- **curl** http://localhost:5000/todo1

Librería requests

```
>>> from requests import put, get
```

```
# Podemos recibir la respuesta en json
```

```
>>> put('http://localhost:5000/todo1', data={'data': 'datos de la pet.'}).json()  
{u'todo1': u'Remember the milk'}
```

```
>>> get('http://localhost:5000/todo1').json()  
{u'todo1': u'Datos de la pet.'}
```

```
>>> put('http://localhost:5000/todo2', data={'data': 'cambiar datos'}).json()  
{u'todo2': u'cambiar datos'}
```

```
>>> get('http://localhost:5000/todo2').json()  
{u'todo2': u'cambiar datos'}
```

Tipos de respuesta

- Flask-restful admite múltiples tipos de valores de respuesta en los métodos de la vista que devuelven los recursos.
- Por defecto el código de respuesta es 200 (todo ok!)
- Pero se puede indicar otro código (ver códigos de respuesta):
 - <https://developer.mozilla.org/es/docs/Web/HTTP/Status>
- Incluso las cabeceras

Ejemplo

```
class Test3(Resource):  
    def get(self):  
        return {"test3":"resp3"} # Por defecto Código 200  
  
class Test4(Resource):  
    def get(self):  
        return {"test4":"resp4"}, 201  
  
class Test5(Resource):  
    def get(self):  
        return {"test5":"resp5"},201, {'Etag': 'some'}
```

EndPoints: add_resource

- Al método **add_resource** del objeto Api se le pueden pasar múltiples URLs, asocian una clase (que hereda de **Resource**) a un endpoint.
- `api.add_resource(Clase, "/", "/hello")`
 - <http://localhost:8000/>
 - <http://localhost:8000/hello>

Paso de argumentos

- Flask-RESTful Permite la validación de los parámetros mediante la librería **reqparse**
- Se crea un parser y se indican los argumentos (con tipo) para poder validarlos después.
- Si un argumento no pasa la validación, el framework responderá con una solicitud incorrecta (código http: 400) y el mensaje de error.
 - `from flask_restful import reqparse`
 - `parser = reqparse.RequestParser()`
 - `parser.add_argument('rate', type=int, help='Rate to charge for this resource')`
 - `args = parser.parse_args()`

Ejemplo

- Objetos anidados:

Para parsear los argumentos:

```
parser_cat = reqparse.RequestParser()  
parser_cat.add_argument('id', type=int)  
parser_cat.add_argument('nombre', type=str)
```

```
parser = reqparse.RequestParser()  
parser.add_argument('id', type=int)  
parser.add_argument('nombre', type=str)  
parser.add_argument('cat', type=dict)  
parser.add_argument('precio', type=float)  
parser.add_argument('exis', type=int)
```

Error generado

- `curl -d 'rate=foo' http://127.0.0.1:5000/todos`
- `{'status': 400, 'message': 'foo cannot be converted to int'}`

Respuestas I

- Para devolver el objeto podemos utilizar la función: **make_response**
- Y **jsonify** para convertir el objeto.
 - Deberíamos aplicarlo a un objeto pasado a un diccionario de Python.
 - Utilizar la propiedad **__dict__** de los objetos.
- `from flask import jsonify, make_response`
- Ejemplo:
 - `return make_response(jsonify(producto.to_json()),200)`

Respuestas II

- Respuesta de **error**:
 - Función **abort** con el código de error 404: Recurso no encontrado
 - `abort(404, message=f"Todo {todo_id} no existe")`
- **DELETE**:
 - La petición se ha completado con éxito pero su respuesta no tiene ningún contenido: **return "", 204**
 - También podemos devolver algo de información con un código 200
 - Por ejemplo: el nombre del objeto eliminado.
- **PUT / UPDATE**:
 - La solicitud ha tenido éxito y se ha creado un nuevo recurso como resultado de ello. Ésta es típicamente la respuesta enviada después de una petición PUT
 - **return task, 201**
- **POST / CREATE**:
 - Devolver información en json si ha creado o no.

Formateo de datos

- La devolución de objetos de Python suele funcionar bien, pero cuando queremos devolver un objeto tendremos que definir la estructura del objeto.
- El framework proporciona el módulo **fields** y el decorador **marshal_with**.
- Utilizamos este decorador para serializar el objeto, y fields para definir la estructura de los campos del objeto.

Ejemplo

```
from flask_restful import fields, marshal_with
```

```
resource_fields = {  
    'task': fields.String,  
    'uri': fields.Url('todo_ep')  
}
```

```
class TodoDao(object):  
    def __init__(self, todo_id, task):  
        self.todo_id = todo_id  
        self.task = task  
  
    # This field will not be sent in the response  
    self.status = 'active'
```

```
class Todo(Resource):  
    @marshal_with(resource_fields)  
    def get(self, **kwargs):  
        return TodoDao(todo_id='my_todo', task='Remember the milk')
```

Enlaces

- Flask-RestFul

- <https://flask-restful.readthedocs.io/en/latest/quickstart.html>
- <https://blog.miguelgrinberg.com/post/designing-a-restful-api-using-flask-restful>

- Flask

- <https://flask-restful.readthedocs.io/en/latest/api.html>
- <https://flask.palletsprojects.com/en/2.2.x/>
- <https://flask.palletsprojects.com/en/2.2.x/views/>