UNIVERSIDAD COMPLUTENSE MADRID

Desarrollo de servicios web con Python: API REST

Enrique Martín - emartinm@ucm.es

Gestión de la Información en la Web (GIW)

Optativa de grados

Fac. Informática

Contenidos

Servicios web

¿Qué es un servicio web?

Servicio web (según la W3C)

A Web service is a software system designed to support interoperable machine-to-machine interaction over a network.

URL: `https://www.w3.org/TR/2004/NOTE-ws-gloss-20040211/#webservice`

- Simplificando, son sistemas que permiten que unos programas se comuniquen con otros.
- Existen muchas tecnologías disponibles y se usan en diversas circunstancias, que podríamos englobar en dos familias:
  - Llamadas a precedimientos remotos
  - Consulta y manipulación de datos alojados en servidores

Arquitecturas/Tecnologías para servicios web

- **REST (o RESTful)**
- SOAP: Simple Object Access Protocol
- GraphQL (Facebook)
- Apache Thrift (Facebook)
- JSON-RPC
- gRPC (Google)
- Falcor (Netflix)
- Protocol Buffers (Google)
- Apache Avro

Servicios web RESTful

- Una de las arquitecturas más utilizadas últimamente para la implementación de servicios web es REST (*Representational State Transfer*)
  - Se ejecuta completamente sobre HTTP
  - Está centrada en los recursos a través de su URL
  - Usa métodos de HTTP (GET, PUT, DELETE...) para representar las operaciones
- Un servicio web que se utiliza REST se denomina servicio web RESTful, pero muchas veces se usan estos términos como sinónimos
- A la interfaz que proporciona un servicio RESTful se la denomina **API REST**

Características de REST

- **No tiene estado**: el servidor sabe cómo responder a cada petición únicamente con los datos contenidos en esa petición
- Los **recursos** se determinan mediante su **URL** única
- La operación a realizar se representa con **métodos HTTP**
  - GET: consultar un recurso
  - POST: crear un recurso nuevo
  - PUT: reemplazar un recurso existente
  - PATCH: editar partes de un recurso existente
  - DELETE: borrar un recurso existente
- Sigue el principio **HATEOAS** (*Hypermedia As The Engine Of Application State*). Los resultados obtenidos en una operación pueden contener URL para acceder a otros recursos relacionados.
- Está basado en **JSON**, pero también suele soportar **XML**

JSON vs XML

JSON

```
{ "name"    : "Pepe",
  "age"     : 23,
  "citizen": true,
  "address": {
    "street": "Mayor",
    "number": 55
  },
  "cars": [ "/cars/89",
            "/cars/92"
  ]
}
```

XML

```
<client>
  <name>Pepe</name>
  <age>23</age>
  <citizen>true</citizen>
  <address>
    <number>55</number>
    <street>Mayor</street>
  </address>
  <cars>/cars/89</cars>
  <cars>/cars/92</cars>
</client>
```

- JSON es más compacto y se transforma fácilmente a diccionarios u objetos del lenguaje de programación
- XML es más extenso y su lectura es más compleja (puede requerir expresiones XPath o consultas XQuery)

Recordatorio de HTTP

HTTP

- HTTP (Hypertext Transfer Protocol) es un protocolo en la capa de aplicación de la pila TCP/IP
- Es el protocolo que se utiliza para navegar por Internet, siendo HTTPS su versión cifrada
- Es un protocolo que funciona mediante peticiones y respuestas de texto en un entorno cliente-servidor
- Versiones:
  - HTTP 0.9: año 1991
  - HTTP 1.0: año 1996
  - HTTP 1.1: año 1997
  - HTTP 2.0: año 2015
  - HTTP 3.0: año 2018 (todavía no se usa ampliamente)

Recursos

- Los recursos se identifican mediante su URL (*Uniform Resource Locator*). Ejemplo:

  `https://informatica.ucm.es/informatica/informacion-docente`
- Tiene tres partes:
  - El protocolo: **https**
  - El host: **informatica.ucm.es**
  - El identificador: **/informatica/informacion-docente**

Parámetros y fragmentos en URL

- El identificador del recurso puede incluir parámetros que serán usados por el servidor web. Estos van siempre al final
- Los parámetros se pasan utilizando el símbolo **?** tras el identificador y añadiendo parejas **nombre=valor** separadas por el símbolo **&**.
  Ejemplo:
  `http://web.fdi.ucm.es/Guia_Docente/Prog_asignatura.asp?fdicurso=2020-2021&titu=39`
- Esta URL contiene dos parámetros:
  - **fdicurso** con valor 2020-21
  - **titu** con valor 39

- El identificador del recurso también puede incluir un *identificador de fragmento* dentro del recurso. Para ello se utiliza el carácter **#**:
  `https://www.ucm.es/#carouselUcm`

Tipos de peticiones HTTP (métodos)

- Existen varios tipos de peticiones en el estándar HTTP, aunque para la navegación usual se utilizan principalmente **GET** y **POST**:
  - `GET`: solicita un recurso específico. **Las peticiones `GET` sólo deben recuperar datos, no cambiar el estado del servidor**
  - `HEAD`: igual que `GET` pero sin solicitar el contenido del recurso, solo las cabeceras
  - `POST`: enviar contenido a un recurso en específico, causando un cambio en el estado del servidor
  - `PUT`: reemplaza un recurso
  - `DELETE`: borra un recurso
  - `CONNECT`: establece un túnel hacia un servidor
  - `OPTIONS`: descubre qué opciones de comunicación tiene un recurso
  - `TRACE`: prueba el camino de ida y vuelta a un recurso
  - `PATCH`: modifica parcialmente un recurso

Peticiones y contestaciones HTTP

- Una petición o contestación HTTP tiene 4 partes:
  - Tipo de petición: `GET`, `POST`...
  - Recurso involucrado
  - Cabecera: información adicional de la petición
  - Cuerpo: contenido de la petición, si es necesario
- Ejemplo de petición realizada en navegador Firefox:

```
GET /estudiantes HTTP/1.1
Host: informatica.ucm.es
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:82.0) Gecko
    /20100101 Firefox/82.0
Accept: text/html,application/xhtml+xml
Accept-Language: es-ES,es;q=0.8,en-US;q=0.5,en;q=0.3
Accept-Encoding: gzip, deflate, br
Connection: keep-alive
Referer: https://informatica.ucm.es/
Upgrade-Insecure-Requests: 1
```

Peticiones y contestaciones HTTP

- Ejemplo de contestación:

```
HTTP/1.1 200 OK
Date: Tue, 27 Oct 2020 10:38:48 GMT
Server: Apache
Content-Type: text/html; charset=UTF-8
Strict-Transport-Security: max-age=63072000
Vary: Accept-Encoding
Content-Encoding: gzip
Content-Length: 4629

<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8" />
    <title>Facultad de Informática</title>
    ...
</html>
```

Estados HTTP

- Las contestaciones HTTP llevan un código de estado indicando el resultado. Estos códigos están organizados en 5 familias:
  - **1xx** → Información, p.ej. informar de que se ha recibido la petición y se está procesando
  - **2xx** → La petición ha tenido éxito: **200 OK**, **201 Created**, **204 No Content**, **206 Partial Content**...
  - **3xx** → Redirección, para informar que el recurso ha cambiado de dirección: **301 Moved Permanently**, **307 Temporary Redirect**...
  - **4xx** → Error del cliente como mala sintaxis, fichero no encontrado o acceso prohibido: **400 Bad Request**, **401 Unauthorized**, **403 Forbidden**, **404 Not Found**...
  - **5xx** → Error del servidor. La petición era correcta pero el servidor ha fallado por causas ajenas al cliente: **500 Internal Server Error**, **502 Bad Gateway**, **503 Service Unavailable**...

Aplicaciones web con Flask

Implementación de servicios web en Python

- Existen varios *frameworks* que permiten implementar servicios web con facilidad:
  - Bottle
  - Pyramid
  - **Flask**
  - Django
- En este tema nos centraremos en Flask porque es un *framework* pequeño, sencillo y potente, con el que es sencillo implementar servicios REST
- En un tema posterior trataremos **Django**, que sirve para implementar aplicaciones web más complejas

Flask

- Flask es un *micro-framework* en Python para el desarrollo de aplicaciones web
- Es *micro* en el sentido de que incluye lo mínimo imprescindible para implementar una aplicación web:
  - Definición de rutas
  - Acceso al contenido de las peticiones
  - Generación de respuestas
  - Manejo de sesiones
  - Soporte de plantillas para generar respuestas (*Jinja*)
- Existen bibliotecas que proporcionan capacidades adicionales no incluidas en el núcleo de Flask: manejo de formularios, generación de *tokens*, persistencia de datos, etc.

Uso de Flask

- A continuación se presentarán los aspectos más importantes de Flask mediante ejemplos de código
- Podéis encontrar el fichero completo test_flask.py en el Campus Virtual para ejecutarlo en vuestro ordenador
- Para ampliar información recomiendo visitar:
  - Página oficial de Flask
    `https://flask.palletsprojects.com/`
  - Documentación de Jinja (para definir plantillas)
    `https://jinja.palletsprojects.com/templates/`

Esqueleto básico de un servicio web con Flask

```
from flask import Flask, request, session, render_template
app = Flask(__name__)


class FlaskConfig:
    """Configuración de Flask"""
    # Activa depurador y recarga automáticamente
    ENV = 'development'
    DEBUG = True
    TEST = True
    # Imprescindible para usar sesiones
    SECRET_KEY = "giw_clave_secreta"
    STATIC_FOLDER = 'static'
    TEMPLATES_FOLDER = 'templates'


if __name__ == '__main__':
    app.config.from_object(FlaskConfig())
    app.run()
```

Ejecutar el servicio web desde la consola

- Para lanzar el servicio web hay que invocar a python y pasar como parámetro el nombre del fichero Python que queremos ejecutar (en nuestro caso test_python.py)

```
$ python test_flask.py
 * Serving Flask app "test_flask" (lazy loading)
 * Environment: development
 * Debug mode: on
 * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 258-973-077
```

Definición de rutas

```python
@app.route('/partidos', methods=['GET'])
def partidos():
    """Ruta fija"""
    return 'Datos de todos los partidos'


@app.route('/partidos/<int:numero>', methods=['GET'])
def partido_concreto(numero):
    """Ruta con una variable numérica"""
    return f'Datos del partido número {numero}'
```

Definición de rutas

```
@app.route('/partidos/<int:numero>/goles', methods=['GET'])
def partido_goles(numero):
    """Ruta con variable numérica en la mitad"""
    return f'Goles del partido número {numero}'


@app.route('/partidos/<int:num_partido>/goles/<int:num_gol>',
    methods=['GET'])
def partido_gol_concreto(num_partido, num_gol):
    """Ruta con varias variables numéricas"""
    return f'Información del gol {num_gol} del partido número {
    num_partido}'
```

Uso del log para mostrar información de depuración

```
@app.route('/logging', methods=['GET'])
def usar_log():
    """ Ejemplos de uso de log de Flask """
    app.logger.debug('Mensaje de depuración')
    app.logger.info('Mensaje informacion')
    app.logger.warning('Aviso de algo potencialmente malo')
    app.logger.error('Algo malo ha ocurrido')
    app.logger.critical('Situación crítica del servidor')
    return 'Se ha escrito en el log del servidor'
```

Uso de diferentes métodos HTTP

Objeto `request`

`request` es un objeto que se puede utilizar en cualquier función y representa la petición actual. Sirve para acceder al método (`method`), los argumentos (`args`), las cabeceras (`headers`)...

```
@app.route('/post', methods=['POST'])
def post():
    """Ruta para peticiones POST"""
    return 'Recibida peticion POST'


@app.route('/methods', methods=['GET', 'POST'])
def methods():
    """request.method indica el método HTTP utilizado"""
    if request.method == 'GET':
        return 'Recibida una petición GET'
    return 'Recibida una petición POST'
```

Uso de diferentes métodos HTTP

```python
@app.route('/methods', methods=['DELETE'])
def methods_delete():
    """Se pueden definir varias rutas para el mismo recurso.
    Si una petición encaja en varias rutas, se elegirá la
    primera que encaje"""
    return 'Recibida una petición DELETE'
```

Parámetros y cabeceras de la petición

```python
@app.route('/parametros', methods=['GET'])
def parametros():
    """request.args es un diccionario con los
    parámetros de la petición"""
    return f"Parámetros enviados: ini={request.args.get('ini')}
     fin={request.args.get('fin')}"
```

```python
@app.route('/cabeceras', methods=['GET'])
def cabeceras():
    """request.headers es un diccionario con las
    cabeceras de la petición"""
    return f'Cabeceras de la petición: Accept={request.headers
    ["Accept"]}'
```

Valores enviados en un formulario

```python
@app.route('/login', methods=['POST'])
def login():
    """request.form es un diccionario con los datos
    enviados por formulario"""
    username = request.form['username']
    password = request.form['password']
    return f'Recibido user={username} pass={password}'
```

Extraer JSON

Si la petición contiene un JSON, se puede extraer directamente con `request.get_json()`. Este método devuelve un diccionario.

Generación de respuestas: HTML y JSON

```python
@app.route('/respuesta_html', methods=['GET'])
def respuesta_html():
    """Si la funcion devuelve una cadena, se genera
    una contestación que contiene contenido HTML
    con código de estado 200"""
    return "<html><h1>Bienvenido!!</h1></html>"
```

```python
@app.route('/respuesta_json', methods=['GET'])
def respuesta_json():
    """Si se devuelve un diccionario, se generará
    una respuesta JSON de manera automática con
    código de estado 200"""
    dicc = {'nombre': 'pepe', 'edad': 55}
    return dicc
```

Generación de respuestas: estado y cabeceras

```python
@app.route('/error', methods=['GET'])
def error():
    """ Devolver código de error """
    return ("¡Todo mal!", 404)
```

```python
@app.route('/respuesta_cabeceras', methods=['GET'])
def respuesta_cabeceras():
    """Si se devuelve 3 valores, el tercer valor
    es un diccionario que representa las cabeceras
    que se añadirán en la contestación"""
    return (
        "<user><nombre>Pepe</nombre><edad>44</edad></user>",
        200, {'Content-Type': 'application/xml'})
```

Manejo de sesiones

Objeto `session`

`session` es un diccionario que representa la sesión actual. Permite acceder y modificar los valores almacenados en la sesión

```python
@app.route('/ver_sesion', methods=['GET'])
def ver_sesion():
    """session es un diccionario que almacena el
    estado de la sesión y que se puede modificar.
    Para usar sesiones es imprescindible establecer
    SECRET_KEY en la configuración de Flask"""
    valor_actual = session.get('valor', 0)
    session['valor'] = valor_actual + 1
    return f'Valor en la sesión: {session["valor"]}'
```

Plantilla simple

Plantilla `templates/saludo.html`

```
<html>
  <h1>Hola {{ name }}, bienvenido!!</h1>
</html>
```

Ruta que utiliza la plantilla

```
@app.route('/plantilla_simple', methods=['GET'])
def plantilla_simple():
    """Renderizado de plantilla simple"""
    return render_template('saludo.html', name='José')
```

Plantilla condicional

Plantilla `templates/saludo_admin.html`

```
<html>
  {% if admin %}
  <h1>Hola administrador {{ name }}!!</h1>
  {% else %}
  <h1>Hola usuario {{ name }}!!</h1>
  {% endif %}
</html>
```

Ruta que utiliza la plantilla

```
@app.route('/plantilla_condicional', methods=['GET'])
def plantilla_condicional():
    """Renderizado de plantilla condicional"""
    return render_template('saludo_admin.html',
                           admin=False, name='José')
```

Plantilla con bucles

Plantilla `templates/saludo_lista.html`

```
<html>
  <ul>
  {% for elem in productos %}
    <li>{{ elem }}</li>
  {% endfor %}
  </ul>
</html>
```

Ruta que utiliza la plantilla

```
@app.route('/plantilla_bucle', methods=['GET'])
def plantilla_bucle():
    """Renderizado de plantilla que recorre una lista.
    Más información sobre la sintaxis de plantillas en
    https://jinja.palletsprojects.com/en/2.11.x/templates/"""
    return render_template('saludo_lista.html',
                           productos=['pera', 'platano'])
```

Diseño de una API REST

Principios de diseño de una API REST

- A la hora diseñar una API REST tendremos que elegir bien las URL de los recursos, decidir qué datos almacenará cada recurso y qué operaciones están permitidas en cada recurso
- A continuación veremos una serie de consejos concretos para diseñar una buena API, pero deben seguirse dos «principios de diseño»:
  - *Keep it simple, stupid* (**KISS**): la API debe ser lo más sencilla posible. Idealmente se debería saber usar únicamente viendo ejemplos de llamadas, sin tener que consultar ninguna documentación
  - *Hypermedia as the Engine of Application State* (**HATEOAS**): los recursos pueden contener hiperenlaces a otros recursos

Consejos para diseñar una API REST (resumen)

1. Usar sustantivos en plural para los recursos
2. Ser consistentes con los nombres
3. Permitir operaciones CRUD con los métodos HTTP esperados
4. Devolver códigos de estado informativos
5. Permitir lecturas parciales para recursos grandes
6. Paginar, filtrar y ordenar listados de recursos
7. Soportar varias versiones de la API
8. Negociar el contenido de las respuestas

Nombre de los recursos

- Los recursos deben ser identificados con un **sustantivo**, nunca con un **verbo**. El verbo representa qué vamos a hacer con el recurso, y eso ya lo indicará el **método HTTP** elegido.
- Además, los sustantivos de las rutas deben estar en **plural** cuando pueda haber más de uno, nunca en singular.
- Ejemplos de recursos:
  - `/orders`: acceso a todos los pedidos realizados
  - `/orders/6`: acceso al pedido número 6
  - `/orders/6/elements`: acceso a los artículos que forman parte del pedido número 6
  - `/orders/6/address`: acceso a la dirección de envío del pedido número 6. **En este caso solo podrá haber una**.

Consistencia de los nombres

- A la hora de decidir el nombre de los recursos o de los campos que forman los recursos, es usual que aparezcan nombres compuestos
- Hay varias alternativas para estos casos, pero se pueden destacar 2:
  - **`snake_case`**: todo en minúsculas y cada palabra separa por barra baja (_). Es el estilo usado en Python. Ejemplos: `delivery_address`, `credit_card`, `product_number`.
  - **`camelCase`**: primera palabra en minúsculas y el resto de palabras con mayúscula inicial, con todas las palabras juntas. Es el estilo más usado en lenguajes OO como Java. Ejemplos: `deliveryAddress`, `creditCard`, `productNumber`.
- Es importante elegir una y seguir el mismo estilo en todos los nombres, tanto de **recursos** como de **campos**.

Operaciones CRUD

- Salvo excepciones justificadas, los recursos deben soportar las 4 operaciones **CRUD**: *Create, Read, Update & Delete*.
- Cada operación se debe invocar con un método HTTP concreto
  - **Create**: método `POST`
  - **Read**: método `GET`
  - **Update**: método `PUT` si se trata de una actualización completa, método `PATCH` si se trata de una modificación parcial de algunos campos internos
  - **Delete**: método `DELETE`
- Ejemplos de peticiones REST:
  - `<<POST /orders>>`: añade un nuevo pedido
  - `<<GET /orders>>`: obtiene el listado de todos los pedidos
  - `<<GET /orders/67>>`: obtiene el pedido número 67
  - `<<PUT /orders/8>>`: reemplaza el pedido número 8
  - `<<PATCH /orders/8>>`: modifica algunos campos del pedido número 8
  - `<<DELETE /orders/6>>`: elimina el pedido número 6

Códigos de estado informativos

- Al recibir una petición, se debe responder con un código de estado que indique claramente el resultado.
- Códigos de **éxito**:
  - **200 OK**: código de éxito para casos generales, por ejemplo al devolver un recurso completo o un listado *completo* (GET) o al aplicar una modificación (PUT/PATCH).
  - **201 Created**: respuesta al crear con éxito un recurso (POST)
  - **204 No Content**: la petición tuvo éxito pero no hay nada que devolver. Se usa al borrar recursos (DELETE)
  - **206 Partial Content**: el resultado que se devuelve es un fragmento de un recurso o un subconjunto de un listado (GET)

Códigos de estado informativos

- Dentro de los códigos de **error** se pueden destacar:
  - **400 Bad Request**: error genérico si la petición no se puede procesar porque está mal formada. Por ejemplo, si se han pasado parámetros no soportados (p.ej. `<<GET /orders?total=5>>` y el parámetro `total` no está soportado)
  - **404 Not Found**: el recurso solicitado no existe
  - **405 Method Not Allowed**: el recurso no admite el método HTTP utilizado

Permitir lecturas parciales

- Si los recursos contienen mucha información, es interesante permitir que el cliente indique qué campos quiere.
- Estos campos se indicarían mediante parámetros en la petición.
- Una posibilidad es permitir un parámetro `fields` que indique la lista de campos a devolver:
  - `<<GET /orders/8?fields=user,date,total>>`
    Campos `user`, `date` y `total` del pedido número 8
- Si la selección escogida incluye todos los campos, el código de estado deberá ser «**200 OK**»; si ha quedado alguno fuera deberá ser «**206 Partial Content**»

Paginación de resultados

- A la hora de obtener un listado de varios recursos, es posible que se pueda recibir un listado demasiado extenso
- En estos casos sería necesario permitir obtener fragmentos del listado mediante paginación
- Esto se puede obtener fácilmente con dos parámetros:
  - `per_page`: número de elementos a devolver en cada página
  - `page`: número de página a devolver
  - **Ejemplo**: `GET /orders?per_page=10&page=3`
    Tercera página de pedidos conteniendo 10 pedidos cada página
- Si con la paginación elegida se devuelven todos los elementos, el código de estado deberá ser «**200 OK**», si ha quedado alguno fuera deberá ser «**206 Partial Content**»

Filtrado de recursos

- A la hora de obtener un listado de varios recursos, es muy posible que estemos interesados en aquellos que cumplan ciertas propiedades
- Para permitir filtrados, habrá que soportar parámetros que permitan indicar qué recursos concretos nos interesan. Estas condiciones pueden involucrar igualdad estricta o comparaciones más complejas (orden, encaje de patrones, etc.)
- Ejemplos:
  - `GET /orders?status=paid`
    Listado de pedidos que han sido pagados
  - `GET /orders?date_gte=2020-09-01`
    Listado de pedidos realizados a partir del 1 de septiembre de 2020
  - `GET /users?name_like=*McGol*`
    Listado de usuarios cuyo nombre contiene la subcadena «`McGol`»

Ordenación de resultados

- De manera complementaria al paginado y selección de resultados al solicitar un listado de recursos, también suele ser útil permitir la ordenación de los mismos
- Esto se puede llevar a cabo con un parámetro adicional `sort` que indica los campos sobre los que se realizará la ordenación
- Ejemplo: «`GET /orders?sort=date,total`»
  Obtener todos los pedidos ordenados por fecha, y en caso de empate por importe total
- Se puede ampliar esta sintaxis si es necesario indicar el sentido de la ordenación (ascendente o descendente) o si se quiere permitir cualquier mezcla: «`GET /orders?sort=date-ASC,total-DESC)`»

Versionado de la API

- A lo largo del tiempo es posible que la API vaya evolucionando, ya sea para ampliar posibilidades o corregir errores
- Para manejar este proceso sin molestar a los clientes de la API se suelen tener varias versiones funcionando a la vez. La última versión será la recomendada, mientras que las anteriores se mantendrán durante un tiempo para que los clientes puedan seguir funcionando mientras se adaptan a la nueva versión
- Las peticiones deben indicar qué versión concreta de la API quieren utilizar. Esto se podría incluir como un parámetro o dentro de las cabeceras, pero lo más usual es incluir la versión elegida como un prefijo del recurso:
  - `GET /v1/orders/7`: pedido 7 usando la API v1
  - `GET /v2/orders/7`: pedido 7 usando la API v2
  - `GET /v3/orders/7`: pedido 7 usando la API v3

Negociación de contenido

- Lo más usual es que únicamente se soporte JSON, pero si se permite el uso de XML hay que permitir al usuario elegir cómo quiere recibir los resultados
- La manera adecuada de realizar esto es que el usuario indique en la **cabecera** `Accept` de su petición una lista de formatos que permite en la respuesta.
- Ejemplos:
  - `Accept: application/json, application/xml`
    Acepta contestaciones JSON y XML
  - `Accept: application/xml`
    Acepta únicamente contestaciones XML
- Si la API no soporta ninguno de los formatos aceptados por el usuario, la contestación debería tener el código de estado «**406 Not Acceptable**»