

1. Introducción

En los apuntes anteriores construimos aplicaciones web con Flask (**apunte 11**) y aprendimos a persistir objetos con SQLAlchemy (**apunte 12**). En ambos casos, la aplicación estaba pensada para que un **navegador** la consumiera: el usuario ingresaba una URL, el framework generaba una página HTML y el navegador la mostraba. Este modelo se conoce como *renderizado del lado del servidor* y sigue siendo muy utilizado.

Sin embargo, existe otro modelo igualmente importante: aplicaciones que no devuelven HTML, sino **datos estructurados**. Estas aplicaciones no están pensadas para un navegador, sino para que **otras aplicaciones** las consuman. Una aplicación móvil que muestra el ranking de jugadores, un cliente de escritorio que sincroniza partidas, o incluso otra aplicación web que necesita datos de la primera: todos ellos necesitan una interfaz programática para comunicarse.

A esta interfaz se la denomina **API** (*Application Programming Interface*). En este apunte vamos a estudiar un estilo particular de APIs web, conocido como **REST**, que es el dominante en la industria. Veremos qué es un recurso, cómo se usan los métodos HTTP, qué significan los códigos de estado, cómo se estructura la información en JSON y cómo se autentican los clientes. También mencionaremos el concepto de **microservicio** y las herramientas más utilizadas para probar y documentar APIs.

2. ¿Qué es una API?

Una **API** es una interfaz que permite que dos programas se comuniquen entre sí. El término es amplio: incluye desde funciones de una biblioteca hasta sistemas operativos completos. Una API define **qué operaciones** están disponibles y **cómo** invocarlas, sin exponer los detalles de cómo están implementadas por dentro.

Esta idea ya la vimos, aunque no con este nombre, cuando estudiamos las interfaces en POO. Una interfaz define un contrato: qué mensajes se pueden enviar a un objeto, sin importar cómo responde. Una API es exactamente lo mismo, pero a nivel de sistemas: define qué peticiones se pueden hacer a un servicio, sin que el cliente necesite saber cómo está construido.

Existen varios tipos de APIs:

- **APIs de biblioteca:** funciones y clases que exponen las bibliotecas que usamos (por ejemplo, `math.sqrt()` o `random.randint()`).
- **APIs de sistema operativo:** llamadas al sistema que permiten interactuar con archivos, procesos o redes.
- **APIs web:** servicios accesibles a través de HTTP, que devuelven datos en algún formato (habitualmente JSON).

En este apunte nos centramos en las **APIs web**, que son las que se construyen con frameworks como Flask y las que se consumen a través de la red.

3. El estilo REST

REST (*Representational State Transfer*) es un estilo arquitectónico para diseñar APIs web, descrito por Roy Fielding en su tesis doctoral del año 2000¹. No es un protocolo ni una tecnología concreta: es una **forma de diseñar** aplicaciones distribuidas que aprovecha las capacidades del protocolo HTTP.

REST se apoya en varios principios:

- **Recursos:** todo lo que la API expone es un *recurso*, identificado por una URL. Un jugador, una partida, una lista de puntajes: todos son recursos.
- **Sin estado (*stateless*):** cada petición debe contener toda la información necesaria para ser entendida y procesada. El servidor no guarda el estado de la conversación entre peticiones.
- **Representaciones:** el cliente y el servidor intercambian *representaciones* de los recursos. Un mismo recurso puede representarse en JSON, XML, HTML, etc. Lo habitual es JSON.

¹Fielding, R. T. (2000). *Architectural Styles and the Design of Network-based Software Architectures*. Tesis doctoral, University of California, Irvine. En el capítulo 5 se describe REST como un estilo arquitectónico derivado de las restricciones del HTTP original.

- **Interfaz uniforme:** se usan los métodos HTTP estándar (GET, POST, PUT, DELETE) con una semántica consistente en toda la API.

3.1. REST vs RESTful

Estos dos términos suelen confundirse, y conviene distinguirlos:

- **REST** es el estilo arquitectónico, el conjunto de principios.
- **RESTful** es la calificación de un servicio que *respet*a esos principios.

No toda API que usa HTTP es REST. Por ejemplo, una API que expone rutas como `/obtenerJugador?id=3` no es RESTful, porque el verbo está en la URL en lugar de estar en el método HTTP. Una API RESTful expondría ese mismo recurso como `/jugadores/3`, y el método GET indicaría que se está consultando.

3.2. El modelo de madurez de Richardson

Leonard Richardson propuso un modelo que clasifica las APIs en cuatro niveles según cuán fielmente respetan los principios de REST². Los niveles son:

1. **Nivel 0:** un solo endpoint, que recibe todo por POST (por ejemplo, XML-RPC o SOAP).
2. **Nivel 1:** múltiples recursos, identificados por URLs.
3. **Nivel 2:** uso correcto de los métodos HTTP y los códigos de estado.
4. **Nivel 3:** hipermedia (HATEOAS), donde las respuestas incluyen enlaces a otras acciones posibles.

La mayoría de las APIs en producción se ubican en el **nivel 2**. El nivel 3 es más ambicioso y menos común. En este curso apuntamos al nivel 2, que es lo que se espera en la industria.

²Richardson, L. & Amundsen, M. (2013). *RESTful Web APIs*. O'Reilly. El modelo de madurez se describe en los primeros capítulos.

4. Recursos, URLs y métodos HTTP

4.1. Recursos y URLs

Un **recurso** es cualquier entidad que la API expone. Se identifica por una URL, y por convención las URLs usan **sustantivos en plural**, no verbos:

Bien (RESTful)	Mal
GET /jugadores	GET /obtenerJugadores
GET /jugadores/3	GET /jugador?id=3
POST /jugadores	POST /crearJugador
GET /jugadores/3/partidas	GET /partidasDeJugador?jugador=3

La URL identifica *qué* recurso se está manipulando; el método HTTP indica *qué se hace* con él. Esto es lo que se conoce como interfaz uniforme.

4.2. Métodos HTTP

Los métodos HTTP (también llamados *verbos*) tienen una semántica bien definida en el contexto de REST:

Método	Acción	Ejemplo
GET	Obtener un recurso o una colección	GET /jugadores
POST	Crear un nuevo recurso	POST /jugadores
PUT	Reemplazar un recurso completo	PUT /jugadores/3
PATCH	Modificar parcialmente un recurso	PATCH /jugadores/3
DELETE	Eliminar un recurso	DELETE /jugadores/3

Cada método tiene dos propiedades importantes desde el punto de vista arquitectónico:

- **Seguridad (*safety*):** un método seguro no modifica el estado del servidor. GET es seguro: consultar no cambia nada. POST, PUT, PATCH y DELETE no lo son.
- **Idempotencia:** un método es idempotente si ejecutarlo varias veces produce el mismo resultado que ejecutarlo una vez. GET, PUT y DELETE son idempotentes. POST no lo es: dos POST con el mismo cuerpo crean dos recursos distintos.

Estas propiedades no son un capricho teórico: los clientes HTTP, los proxies y los navegadores las asumen y actúan en consecuencia. Una API que no las respeta puede comportarse de manera inesperada.

5. Códigos de estado

Toda respuesta HTTP incluye un **código de estado**, un número de tres dígitos que indica el resultado de la petición. Los códigos se agrupan en cinco clases:

Rango	Significado
1xx	Información (rara vez se usan en APIs REST).
2xx	Éxito. La petición se procesó correctamente.
3xx	Redirección. El recurso está en otra URL.
4xx	Error del cliente. La petición está mal formada o no autorizada.
5xx	Error del servidor. Algo falló del lado del servicio.

Los códigos más utilizados en una API REST son:

Código	Cuándo usarlo
200 OK	La petición fue exitosa. Se usa para GET, PUT y PATCH.
201 Created	Se creó un recurso nuevo. Se usa para POST exitosos.
204 No Content	La petición fue exitosa pero no hay nada que devolver. Se usa para DELETE.
400 Bad Request	La petición está mal formada (por ejemplo, JSON inválido).
401 Unauthorized	El cliente no está autenticado.
403 Forbidden	El cliente está autenticado pero no tiene permiso.
404 Not Found	El recurso solicitado no existe.
500 Internal Server Error	Error inesperado del servidor.

Una respuesta REST completa incluye:

- El **código de estado**, que indica el resultado general.
- Los **encabezados HTTP**, que aportan metadatos (tipo de contenido, longitud, etc.).

- El **cuerpo**, que contiene la representación del recurso en formato JSON (o vacío, en el caso de 204).

Un error bien manejado no es solo un 500 genérico. Una API RESTful devuelve códigos específicos que permiten al cliente entender qué falló y cómo reaccionar.

6. JSON como formato de intercambio

El formato más utilizado para las representaciones en una API REST es **JSON** (*JavaScript Object Notation*). Es un formato de texto que representa datos estructurados de manera simple y legible.

JSON admite seis tipos de valores:

- **Objetos:** pares clave-valor entre llaves (`{"nombre": "Arturo", "nivel": 5}`).
- **Arreglos:** listas ordenadas entre corchetes (`[1, 2, 3]`).
- **Cadenas:** texto entre comillas dobles.
- **Números:** enteros o decimales.
- **Booleanos:** `true` o `false`.
- **Nulo:** `null`.

Un recurso como un jugador podría representarse así:

```
{
  "id": 3,
  "nombre": "Arturo",
  "nivel": 5
}
```

Y una colección de jugadores sería un arreglo de objetos:

```
[
  {"id": 1, "nombre": "Arturo", "nivel": 5},
  {"id": 2, "nombre": "Lancelot", "nivel": 4},
  {"id": 3, "nombre": "Merlin", "nivel": 6}
]
```

6.1. Marshalling y unmarshalling

Los dos procesos centrales al trabajar con JSON son:

- **Marshalling** (serialización): convertir objetos del programa a JSON para enviarlos por la red.
- **Unmarshalling** (deserialización): convertir JSON recibido de la red a objetos del programa.

En **Python** estas conversiones son casi triviales porque los diccionarios y las listas se serializan directamente:

```
1 import json
2
3 # Marshalling: de dict a JSON
4 jugador = {"id": 3, "nombre": "Arturo", "nivel": 5}
5 texto = json.dumps(jugador)
6 # texto = '{"id": 3, "nombre": "Arturo", "nivel": 5}'
7
8 # Unmarshalling: de JSON a dict
9 datos = json.loads(texto)
10 # datos = {"id": 3, "nombre": "Arturo", "nivel": 5}
```

En Flask, la función `jsonify()` hace el marshalling automáticamente en las respuestas, y `request.get_json()` hace el unmarshalling del cuerpo recibido. Esto conecta directamente con la decisión de diseño que tomamos en el apunte 10: las filas se devuelven como diccionarios porque son la estructura natural para serializar a JSON.

7. Microservicios

El término **microservicio** se popularizó en la última década para describir un estilo arquitectónico en el que una aplicación se construye como un conjunto de **servicios pequeños, independientes y desplegados por separado**. Cada servicio se ocupa de una responsabilidad acotada y se comunica con los demás a través de la red, típicamente mediante APIs REST.

Contrasta con el estilo **monolítico**, donde toda la aplicación es un único programa que se despliega como una unidad. En un monolito, el ranking de jugadores, la autenticación

y el catálogo de ítems viven en el mismo proceso. En una arquitectura de microservicios, cada uno de esos componentes podría ser un servicio independiente con su propia base de datos y su propio ciclo de despliegue.

Las ventajas más citadas de los microservicios son:

- **Escalabilidad selectiva:** si el servicio de ranking recibe más tráfico que el de autenticación, se puede escalar solo ese servicio.
- **Independencia tecnológica:** cada servicio puede estar escrito en un lenguaje distinto y usar su propia base de datos.
- **Despliegue independiente:** cambiar un servicio no requiere redespargar toda la aplicación.
- **Aislamiento de fallos:** si un servicio falla, los demás pueden seguir funcionando.

También tiene desafíos importantes: la comunicación por red es más lenta y menos confiable que una llamada en memoria, la coordinación entre servicios es compleja, y el monitoreo y la depuración son más difíciles. La decisión entre monolito y microservicios es un trade-off que depende del tamaño y las necesidades del proyecto. Para aplicaciones pequeñas, un monolito bien estructurado suele ser la opción correcta; los microservicios se justifican cuando la escala o la organización del equipo lo requieren.

8. Autenticación con tokens y JWT

Hasta ahora, todas las operaciones que describimos asumen que cualquiera puede invocarlas. Esto no es realista: en una aplicación real, algunas operaciones requieren que el cliente esté autenticado, y otras requieren que tenga ciertos permisos.

En el apunte 10 vimos cómo almacenar contraseñas de manera segura usando hashing. Ese es el primer paso: cuando un usuario se registra, se guarda el hash de su contraseña. Ahora veamos el segundo paso: cómo se autentica ese usuario en cada petición posterior.

8.1. Sesiones vs. tokens

Existen dos grandes enfoques:

- **Sesiones:** el servidor guarda el estado del usuario autenticado en memoria o en una base de datos, y le asigna un identificador de sesión al cliente. En cada petición, el cliente envía ese identificador y el servidor busca el estado correspondiente. Es el enfoque de las aplicaciones web tradicionales.
- **Tokens:** el servidor no guarda estado. Cuando el cliente se autentica correctamente, el servidor le entrega un **token** firmado. En cada petición, el cliente envía el token, y el servidor lo verifica. Este enfoque es más natural para APIs REST, que por definición son *stateless*.

8.2. JSON Web Tokens (JWT)

El estándar más utilizado para tokens en APIs REST es **JWT** (*JSON Web Token*). Un JWT es un token **autocontenido** y **firmado**: contiene información sobre el usuario (por ejemplo, su id y su rol) y una firma que garantiza que fue emitido por el servidor y no fue alterado.

Un JWT tiene tres partes separadas por puntos:

```
header.payload.signature
```

- **Header:** indica el algoritmo de firma (habitualmente HS256 o RS256).
- **Payload:** contiene los *claims*, es decir, los datos del usuario (id, rol, expiración, etc.).
- **Signature:** firma criptográfica que el servidor calcula con una clave secreta. Si alguien modifica el payload, la firma deja de coincidir y el servidor lo rechaza.

Un JWT real se ve así:

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.  
eyJzdWIiOiIzIiwicm9sIjoianVnYWVvcyJ9.  
TjVA95OrM7E2cBab30RMHrHDcEfxjoYZgeFONFh7HgQ
```

8.3. Cómo se envía el token

El cliente envía el token en el encabezado HTTP **Authorization**, usando el esquema **Bearer**:

```
Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...
```

En el servidor, la API verifica la firma del token. Si es válida, sabe que el usuario es quien dice ser, y puede autorizar la operación. Si no es válida, devuelve **401 Unauthorized**.

Nota: JWT y contraseñas no son lo mismo

Un JWT **no contiene la contraseña** del usuario, ni siquiera hasheada. Contiene los datos que el servidor decide incluir en el payload (por ejemplo, el id y el rol), y una firma que garantiza su autenticidad. La contraseña solo se usa una vez, en el momento de autenticarse, para emitir el token. En el resto de las peticiones, la contraseña ya no viaja por la red.

Esta separación entre *autenticación inicial* (con contraseña) y *autenticación continua* (con token) es una buena práctica de seguridad: reduce la cantidad de veces que las credenciales sensibles viajan por la red.

9. Herramientas del ecosistema

Existen varias herramientas que facilitan el trabajo con APIs REST, tanto para desarrollarlas como para consumirlas.

9.1. curl

curl es un cliente HTTP de línea de comandos. Está disponible en prácticamente todos los sistemas operativos y es la forma más directa de hacer una petición HTTP desde la terminal:

```
# Consultar todos los jugadores
curl http://localhost:5000/jugadores

# Consultar un jugador específico
curl http://localhost:5000/jugadores/3
```

```
# Crear un jugador nuevo
curl -X POST http://localhost:5000/jugadores \
  -H "Content-Type: application/json" \
  -d '{"nombre": "Arturo", "nivel": 5}'
```

9.2. Postman

Postman es un cliente gráfico para probar APIs. Permite construir peticiones con una interfaz visual, guardar colecciones de peticiones, organizarlas por proyecto y compartirlas con el equipo. Es muy útil durante el desarrollo, cuando se prueba el mismo endpoint muchas veces con distintas combinaciones de datos.

9.3. Swagger y OpenAPI

OpenAPI es una especificación estándar para describir APIs REST. Un archivo OpenAPI describe cada endpoint, sus parámetros, los formatos de petición y respuesta, y los códigos de estado posibles. A partir de esa descripción se pueden generar:

- Documentación interactiva (habitualmente con la herramienta **Swagger UI**).
- Clientes para distintos lenguajes.
- Tests automáticos.

En el ecosistema Python existen bibliotecas que generan automáticamente la especificación OpenAPI a partir del código de la API. FastAPI, por ejemplo, lo hace por defecto. En Flask se puede integrar a través de extensiones.

10. Mini ejemplo en Flask

Veamos cómo se ve un endpoint REST en Flask, retomando el modelo **Jugador** del apunte anterior. Solo mostramos el mínimo necesario para ver la sintaxis; el detalle completo va en las guías prácticas.

```
1 from flask import Flask, jsonify, request
2 from flask_sqlalchemy import SQLAlchemy
3 from sqlalchemy.orm import Mapped, mapped_column
4
5 app = Flask(__name__)
6 app.config["SQLALCHEMY_DATABASE_URI"] = "mysql+pymysql://..."
7 db = SQLAlchemy(app)
8
9 class Jugador(db.Model):
10     __tablename__ = "jugadores"
11
12     id: Mapped[int] = mapped_column(primary_key=True)
13     nombre: Mapped[str]
14     nivel: Mapped[int]
15
16     def to_dict(self) -> dict:
17         return {"id": self.id, "nombre": self.nombre,
18                 "nivel": self.nivel}
19
20 # --- GET: devuelve todos los jugadores ---
21 @app.route("/jugadores", methods=["GET"])
22 def listar_jugadores():
23     jugadores = db.session.execute(
24         db.select(Jugador).order_by(Jugador.nivel.desc())
25     ).scalars().all()
26     return jsonify([j.to_dict() for j in jugadores])
27
28 # --- POST: crea un jugador nuevo ---
29 @app.route("/jugadores", methods=["POST"])
30 def crear_jugador():
31     datos = request.get_json()
32     jugador = Jugador(nombre=datos["nombre"], nivel=datos["nivel"])
33     db.session.add(jugador)
34     db.session.commit()
35     return jsonify(jugador.to_dict()), 201
```

Analicemos los puntos clave:

- El decorador `@app.route("/jugadores", methods=["GET"])` registra la función para responder a peticiones GET en esa URL. Para POST se registra con `methods=["POST"]`.
- La función `listar_jugadores()` devuelve un arreglo JSON. Flask convierte el resultado de `jsonify()` en una respuesta HTTP con el encabezado **Content-Type: application/json**.
- La función `crear_jugador()` usa `request.get_json()` para leer el cuerpo de la pe-

tición. El cuerpo viene en formato JSON y se convierte automáticamente a un diccionario de Python.

- El código de estado **201** se pasa como segundo elemento de la tupla devuelta por la vista. Esto es lo que le indica al cliente que el recurso fue creado y no simplemente consultado.

Con solo estos dos endpoints ya se ve el patrón REST en acción: un mismo recurso (/jugadores) que responde a distintos métodos HTTP con distintas operaciones. A partir de aquí, el resto de las operaciones (GET por id, PUT, PATCH, DELETE) siguen exactamente la misma estructura.

Nota: diferencias con apunte de ORM

Los métodos y clases que en el apunte anterior se accedían directamente desde SQLAlchemy (`Session`, `select`, `DeclarativeBase`), acá aparecen como atributos del objeto `db` que inicializa Flask-SQLAlchemy. Es la misma tecnología, integrada al framework.

11. Referencias

Para más información puede consultar los siguientes enlaces:

- Fielding, R. T. (2000). *Architectural Styles and the Design of Network-based Software Architectures*. Tesis doctoral, University of California, Irvine.
- Richardson, L. & Amundsen, M. (2013). *RESTful Web APIs*. O'Reilly.
- Documentación oficial de Flask:
<https://flask.palletsprojects.com/>
- Especificación JWT:
<https://datatracker.ietf.org/doc/html/rfc7519>
- OpenAPI Specification:
<https://spec.openapis.org/oas/latest.html>