IIC2233 • PROGRAMACIÓN AVANZADA

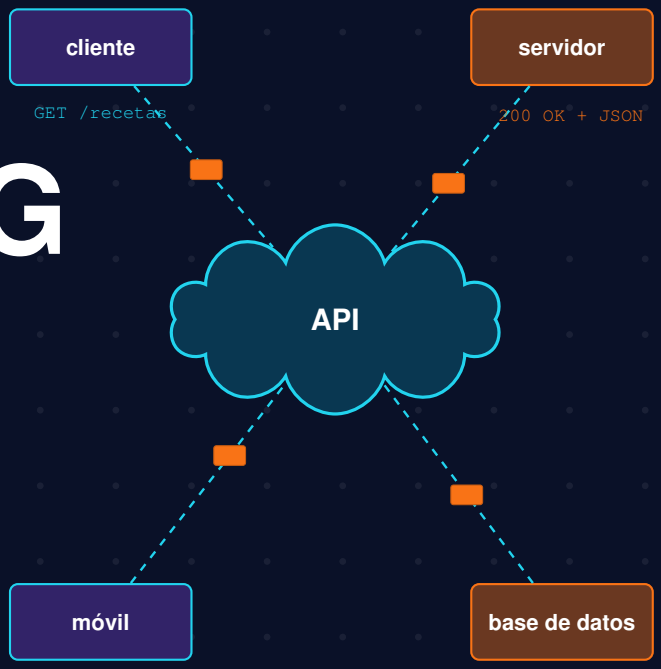
NETWORKING

SERIALIZACIÓN

Y APIs

`json` • HTTP • `requests` • Flask • hilos

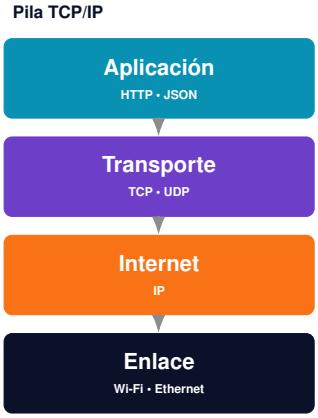
de objetos Python a bytes, de bytes a la red, y de vuelta





Felipín quiso hacer una app con API. Mandó los datos como un `str(objeto)` que nadie más sabe leer, hizo un `requests.get` sin `timeout` que quedó esperando para siempre, y su servidor `Flask` deja que cualquier JSON mal formado lo tire con un `500`. Para peor, dos clientes compran el último producto *al mismo tiempo…* y se lo venden a ambos.

No seas como Felipín. En este resumen aprenderás cómo viaja un dato desde un objeto de Python hasta otro computador: **serializarlo**, **enviarlo** por la red, **consumir** y **construir** APIs, y evitar los peores problemas de **concurrencia**.





Resumen de Contenidos

IIC2233 — Programación Avanzada • Edición Red de datos

Resumen elaborado por el equipo de ayudantía, en base al material oficial del curso

Tabla de Contenidos

1 Introducción 2

2 Serialización de objetos 3

3 Conceptos básicos de redes 8

4 Servicios web, APIs y HTTP 11

5 Cliente de un servicio web con `requests` 14

6 Servidor de un servicio web con `Flask` 18

7 Bonus: concurrencia y condiciones de carrera 22

8 Hoja de referencia 27



1 Introducción

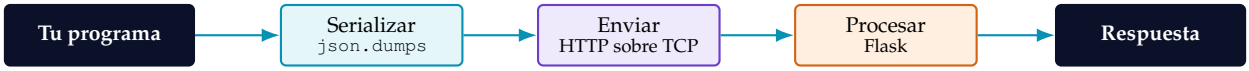
Este resumen reúne el contenido de la semana de **Networking** de IIC2233: cómo se transforma la información de un programa en algo que se pueda guardar o enviar (**serialización**), cómo viaja por una red (**TCP/IP**, **HTTP**) y cómo dos programas se conversan mediante una **API web**, tanto desde el lado del **cliente** (`requests`) como del **servidor** (`Flask`). Como bonus, vemos qué pasa cuando muchas solicitudes llegan *a la vez*.

Los cuatro bloques de este resumen

- **Serialización y JSON** (Sección 2): convertir objetos de Python a texto (y de vuelta), incluyendo clases propias con `JSONEncoder` y `object_hook`.
- **Redes y web** (Secciones 3 y 4): direcciones IP, DNS, puertos, TCP vs. UDP, encapsulamiento, cliente-servidor, URL, métodos y códigos HTTP, y TLS.
- **Cliente y servidor** (Secciones 5 y 6): consumir APIs con `requests` (parámetros, JSON, autenticación) y construir una API con `Flask` (rutas, validación, errores).
- **Concurrencia** (Sección 7, bonus): `ThreadPoolExecutor`, condiciones de carrera, solicitudes duplicadas, *rate limiting* y `Lock`.

★ Cómo leer este resumen

Cada bloque de código va acompañado de una explicación *"¿Qué hace este código?"*. Los recuadros **rojos** marcan errores típicos y trampas; los **naranjos** son definiciones importantes; los **violetas** son **analogías postales** para fijar ideas. El material tiene una idea central que se repite: *un dato siempre viaja como bytes, y ambos lados deben ponerse de acuerdo en cómo interpretarlos.*



el **ciclo completo** que recorre una consulta a una API web

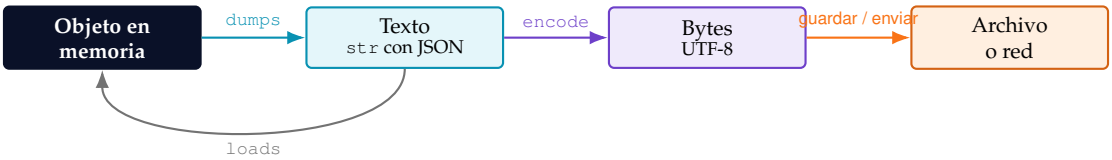
2 Serialización de objetos

2.1 ¿Qué es serializar?

Todo lo que un computador guarda o transmite termina como **bits y bytes**. Sin embargo, nuestros programas piensan en estructuras de más alto nivel: listas, diccionarios, instancias de clases. Para guardarlas en un archivo o enviarlas por una red necesitamos una **representación** que se pueda almacenar o transmitir.

■ Serialización y deserialización

Serializar es transformar el *estado* de un objeto a una representación que se puede guardar o transmitir. **Deserializar** es el proceso inverso: reconstruir el objeto a partir de esa representación. La representación puede ser **texto** (como JSON) o una **secuencia binaria**, según el formato.



serializar → (izquierda a derecha) **deserializar** ← (flecha curva y decodificación)

! Serializar no es lo mismo que persistir

Serializar un objeto **no lo vuelve persistente por sí solo**: la persistencia aparece cuando la representación se *guarda* en un archivo, una base de datos u otro medio de almacenamiento. Además, para enviar texto por una red hay que **codificarlo** a bytes (por ejemplo, con UTF-8).

★ Analogía postal

Serializar es **desarmar un mueble para el despacho**: lo desmontas en piezas planas que caben en una caja (texto o bytes), las mandas, y quien las recibe lo *vuelve a armar* siguiendo las instrucciones (deserializar). Si le mandas las piezas sin instrucciones, o con instrucciones en otro idioma, nadie arma nada.

2.2 JSON

JSON (*JavaScript Object Notation*) es un formato de **texto estándar** para intercambiar datos entre programas. Es independiente del lenguaje y relativamente fácil de leer para una persona. Su sintaxis **se parece** a la de los diccionarios y listas de Python, pero **no es idéntica**: por ejemplo, usa `true`, `false` y `null` (minúsculas), y los *strings* siempre van con comillas dobles.

El módulo `json` transforma por defecto los siguientes tipos:

Python	JSON
`dict`	objeto `{...}`
`list, tuple`	arreglo `[...]`
`str`	*string*
`int, float`	número
`True, False`	`true, false`
`None`	`null`

! Ojo: la ida y la vuelta no siempre dan lo mismo

`tuple` y `list` comparten el tipo *arreglo* en JSON, así que una **tupla se serializa bien, pero al deserializar vuelve como lista**. Además, por defecto **no se pueden serializar** funciones, conjuntos (`set`) ni instancias de clases personalizadas.

2.2.1 Las cuatro funciones del módulo `json`

Hay dos pares de funciones. La clave para no confundirlas: la **s** final significa *string*.

Función	Dirección	Trabaja con	Retorna
`dumps(obj)`	serializa	un `str`	el `str` con el JSON
`loads(texto)`	deserializa	un `str`	el objeto de Python
`dump(obj, archivo)`	serializa	un archivo abierto	nada: escribe en el archivo
`load(archivo)`	deserializa	un archivo abierto	el objeto de Python

Probemos con una lista de valores mixtos:

```python
import json

lista = ["a", 1, 3, "hola"]
lista_serializada = json.dumps(lista)
print(lista_serializada)               # ["a", 1, 3, "hola"]
print(type(lista_serializada))         # <class 'str'>

lista_deserializada = json.loads(lista_serializada)
print(lista_deserializada)             # ['a', 1, 3, 'hola']
print(lista == lista_deserializada)    # True  (mismo contenido)
print(lista is lista_deserializada)    # False (objetos distintos)
```

¿Qué hace este código? `dumps` entrega un `str` (no un objeto `list`). Al deserializar con `loads` obtenemos una lista **igual** a la original, pero es un **objeto nuevo** en memoria: `==` compara contenido, `is` compara identidad.

Ahora el caso que sorprende: una tupla con `None`, números y un diccionario.

```python
tupla = (None, 1, 22.33, {"a": True, "b": False})
serializada = json.dumps(tupla)
print(serializada)          # [null, 1, 22.33, {"a": true, "b": false}]

deserializada = json.loads(serializada)
print(deserializada)        # [None, 1, 22.33, {'a': True, 'b': False}]
print(type(deserializada)) # <class 'list'>
print(tupla == deserializada)    # False  (una tupla nunca es igual a una lista)
```

¿Qué hace este código? Cada valor cambió de "dialecto": `None` → `null`, `True` → `true`, y la tupla se volvió arreglo. Al volver, el arreglo es una **lista**, por eso la comparación con la tupla original da `False`.

2.2.2 Guardar y leer archivos con `dump` y `load`

`dump(objeto, archivo)` escribe el JSON en un archivo ya abierto y `load(archivo)` lo lee. Como JSON usa texto Unicode, conviene abrir los archivos **explícitamente con UTF-8**.

```python
import json
from os import path

lista = [1, 2, 3, 7, 8, 3]

with open(path.join("data", "mi_lista.json"), "w", encoding="utf-8") as archivo:
    json.dump(lista, archivo)

with open(path.join("data", "mi_lista.json"), "r", encoding="utf-8") as archivo:
    lista_cargada = json.load(archivo)

print(lista == lista_cargada)    # True
print(lista is lista_cargada)    # False
```

★ Dos opciones útiles de dumps/dump

`indent=2` imprime el JSON con sangría, ideal para leerlo a ojo. Y por defecto `json` escapa los caracteres no ASCII (ñ sale como `\u00f1`); si prefieres verlos tal cual, usa `ensure_ascii=False` y abre el archivo en UTF-8.

2.3 Serialización de objetos personalizados

¿Qué pasa si la información está en instancias de **nuestras propias clases**? Usaremos como ejemplo una mascota:

```python
class Mascota:
    def __init__(self, nombre: str, edad: int, especie: str, id_: int) -> None:
        self.nombre = nombre
        self.edad = edad
        self.especie = especie
        self.id_ = id_
```

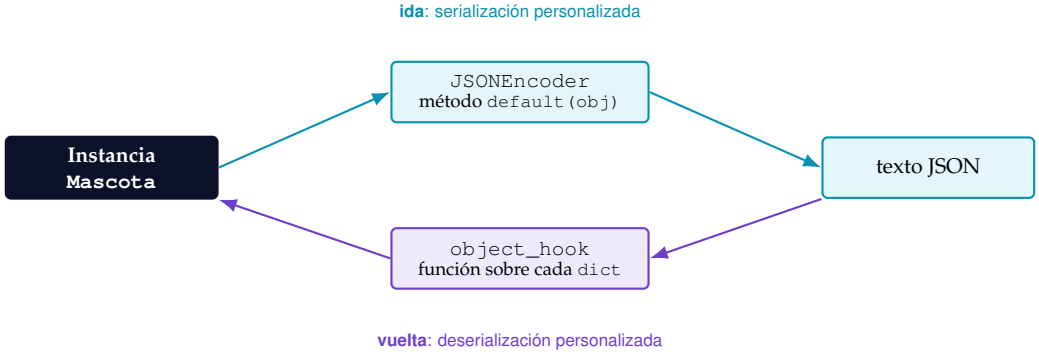
Un primer intento: serializar el diccionario de atributos, `__dict__`.

```python
m = Mascota("Luna", 8, "Gato", 0)

json_string = json.dumps(m.__dict__)
print(json_string)      # {"nombre": "Luna", "edad": 8, "especie": "Gato", "id_": 0}

json_deserializado = json.loads(json_string)
print(type(json_deserializado))    # <class 'dict'>
```

Los atributos viajan bien, pero al volver tenemos un **diccionario**, no una nueva `Mascota`. Para controlar ambos sentidos, el módulo ofrece dos herramientas:



2.3.1 Personalizar la serialización con `JSONEncoder`

Creamos una subclase de `json.JSONEncoder` y sobrescribimos `default(obj)`. Este método se llama **solo cuando `json` no sabe serializar** un objeto. Si reconocemos el tipo, **retornamos una estructura compatible con JSON** (un diccionario). Si no, llamamos `super().default(obj)`, que levanta `TypeError` avisando que ese tipo no es serializable.

```python
import json
from datetime import datetime

class MascotaEncoder(json.JSONEncoder):

    def default(self, obj):
        if isinstance(obj, Mascota):
            return {
                "tipo": "Mascota",
                "Mascota_id": obj.id_,
                "nombre": obj.nombre,
                "edad": obj.edad,
                "especie": obj.especie,
                # Aproximacion: el ano exacto depende del cumpleanos.
                "ano_nacimiento_aproximado": datetime.now().year - obj.edad,
            }
        return super().default(obj)

m1 = Mascota("Pepa", 70, "Tortuga", 1)
print(json.dumps(m1, cls=MascotaEncoder))
```

Salida

```
{"tipo": "Mascota", "Mascota_id": 1, "nombre": "Pepa", "edad": 70, "especie": "Tortuga", "
    ano_nacimiento_aproximado": 1956}
```

¿Qué hace este código? Pasamos nuestra clase con el argumento `cls=MascotaEncoder`. Cuando `dumps` encuentra algo que no entiende (la `Mascota`), invoca `default`, que la convierte en un diccionario **con un campo extra** `"tipo"` que nos servirá para reconocerla al volver. (El año depende de cuándo ejecutes el código; aquí asumimos 2026.)

! No llames al método de la superclase con el diccionario

Basta con **retornar** el diccionario ya transformado. Llamar a `super().default(...)` solo corresponde para el caso final, cuando **no reconocemos** el tipo y queremos que se levante el **`TypeError`**.

Un encoder para varias clases. Se agregan más ramas `isinstance` dentro de `default`. Cualquier objeto no reconocido termina en `super().default(obj)` y levanta **`TypeError`**:

```
class Veterinario:
    def __init__(self, nombre: str, especialidad: str) -> None:
        self.nombre = nombre
        self.especialidad = especialidad

class MascotaVeterinarioEncoder(json.JSONEncoder):
    def default(self, obj):
        if isinstance(obj, Mascota):
            return {"tipo": "Mascota", "nombre": obj.nombre,
                    "edad": obj.edad, "especie": obj.especie}
        if isinstance(obj, Veterinario):
            return {"tipo": "Veterinario", "nombre": obj.nombre,
                    "especialidad": obj.especialidad}
        return super().default(obj)      # tipo desconocido -> TypeError

v = Veterinario("Dra. Torres", "Exoticos")
m2 = Mascota("Cachirulo", 5, "Gato", 2)
m3 = Mascota("Guillermo", 0, "Gato", 3)

print(json.dumps([m2, v, m3], cls=MascotaVeterinarioEncoder, indent=2))

try:
    json.dumps(object(), cls=MascotaVeterinarioEncoder)
except TypeError as e:
    print(f"Tipo no soportado -> {e}")
```

Salida

```
Tipo no soportado -> Object of type object is not JSON serializable
```

¿Qué hace este código? La lista mixta se serializa sin problema porque el encoder sabe manejar ambas clases (`indent=2` la imprime ordenada). En cambio, un `object()` cualquiera no coincide con ninguna rama y recibe el **`TypeError`** del `super()`.

2.3.2 Personalizar la deserialización con `object_hook`

`object_hook` es una **función** que `loads`/`load` llama con **cada objeto JSON** (`{...}`) ya convertido a `dict`. Lo que esa función **retorna reemplaza** al diccionario. Importante: **no se ejecuta** para arreglos, *strings*, números, booleanos ni `null`; solo para objetos.

Un ejemplo simple: cargar un objeto JSON como lista de tuplas en vez de diccionario.

```
def funcion_lista_hook(diccionario: dict):
    return [(key, value) for key, value in diccionario.items()]

json_string = '{"nombre": "Luna", "edad": 8, "especie": "Gato", "ano_nacimiento": 2017}'
datos = json.loads(json_string, object_hook=funcion_lista_hook)
print(datos)
# [('nombre', 'Luna'), ('edad', 8), ('especie', 'Gato'), ('ano_nacimiento', 2017)]
```

Para reconstruir `Mascota` usamos el **discriminador** `"tipo"` que agregó el encoder. El *hook* reconoce **solo** los diccionarios que le corresponden y **devuelve sin cambios** todos los demás (esto es lo que permite que funcione

incluso con objetos anidados):

```
def funcion_mascota_hook(diccionario: dict):
    if diccionario.get("tipo") != "Mascota":
        return diccionario                          # no es nuestro: lo dejamos igual
    return Mascota(
        nombre=diccionario["nombre"],
        edad=diccionario["edad"],
        especie=diccionario["especie"],
        id_=diccionario["Mascota_id"],
    )

texto = json.dumps(m1, cls=MascotaEncoder)
pepa = json.loads(texto, object_hook=funcion_mascota_hook)
print(type(pepa), pepa.nombre)     # <class '__main__.Mascota'> Pepa
```

¿Qué hace este código? Cerramos el círculo: `Mascota` → JSON (con `MascotaEncoder`) → `Mascota` (con `funcion_mascota_hook`). El campo `"tipo"` actúa como una **etiqueta de aduana**: el hook sabe qué diccionarios convertir y cuáles dejar pasar.

El *hook* se aplica de adentro hacia afuera

`object_hook` se llama primero con los objetos **más internos**, y el resultado de cada llamada pasa a formar parte del diccionario que reciba la llamada siguiente. Veámoslo con una función que reemplaza cada diccionario por un número (33 si tiene más de una llave, 42 si no):

```
def hook(diccionario):
    return 33 if len(diccionario) > 1 else 42

texto = '{"a": {"b": 1, "c": [2, 3, {}], "d": null, "e": {"f": true}}}'
print(json.loads(texto, object_hook=hook))      # 42
```

Llamada	Diccionario recibido	Llaves	Retorna
1	`{} (dentro de la lista c)`	0	`42`
2	`{"f": True}`	1	`42`
3	`{"b": 1, "c": [2, 3, 42], "d": None, "e": 42}`	4	`33`
4	`{"a": 33}`	1	`42` (resultado final)

Fíjate en la llamada 3: los objetos internos ya fueron *reemplazados* (`{}` y `{"f": true}` se volvieron `42`) antes de que el hook vea al objeto que los contiene. Y el arreglo `[2, 3, ...]` no pasa por el hook: solo sus elementos que sean objetos.

2.4 Errores al deserializar JSON

Si el texto no cumple la sintaxis JSON, `loads` y `load` levantan **`json.JSONDecodeError`** (una subclase de **`ValueError`**). Podemos capturarla para informar el problema sin detener el programa. El mensaje incluye la **línea, columna y posición aproximada** del error.

```
casos = [
    '["Arreglo que no es cerrado"',                          # falta el ]
    '{"mascota_mas_vieja": "Pepa", "mascota_mas_nueva" "Guillermo"}',  # falta el :
    '["Luna", "Cachirulo" "Guillermo"]',                     # falta una coma
]
for texto_json in casos:
    try:
        json.loads(texto_json)
    except json.JSONDecodeError as error:
        print(f"JSON invalido: {error}")
```

¿Qué hace este código? Los tres textos tienen un error distinto (arreglo sin cerrar, falta de dos puntos entre llave y valor, falta de coma entre elementos), y en los tres casos se levanta **`JSONDecodeError`**, se imprime el mensaje y el programa **continúa**. El texto exacto del mensaje puede variar un poco según la versión de Python.

! Dónde aparece esto en la vida real

Cuando un servidor responde algo que no es JSON válido (por ejemplo, una página de error en HTML) y tu cliente llama a `.json()`, obtendrás un **`JSONDecodeError`**. Por eso, en la Sección 5, antes de interpretar una respuesta revisaremos su `Content-Type`.

✓ Resumen express: serialización

- **Serializar** = objeto → representación guardable/enviable; **deserializar** = al revés. Enviar por red exige además **codificar a bytes** (UTF-8).
- JSON es texto estándar. `dumps`/`loads` trabajan con `str`; `dump`/`load` con archivos abiertos (en UTF-8).
- Tuplas → arreglos → **listas**; `None` ↔ `null`; sin soporte por defecto para `set`, funciones ni clases propias.
- Clases propias: `JSONEncoder.default` (ida, con `cls=`) y `object_hook` (vuelta). Un campo `"tipo"` sirve de discriminador.
- `object_hook` se aplica a cada *objeto* JSON, de adentro hacia afuera. Sintaxis inválida → **`JSONDecodeError`**.

3 Conceptos básicos de redes

Las redes permiten que **dos procesos se comuniquen**, tanto si se ejecutan en la misma máquina como en dispositivos distintos. Para que eso ocurra se necesitan varias piezas que trabajan en conjunto.

3.1 ¿Cómo se comunican los computadores?



Necesitamos, entonces: (1) un **medio** para transmitir señales; (2) una forma de identificar la interfaz de destino, la **dirección IP**; (3) una forma de identificar el proceso de destino, el **puerto**; (4) un **protocolo de transporte** (TCP o UDP); y (5) un **protocolo de aplicación**, que define el significado y el formato de los mensajes.

3.2 Direcciones IP, DNS y puertos

3.2.1 Dirección IP

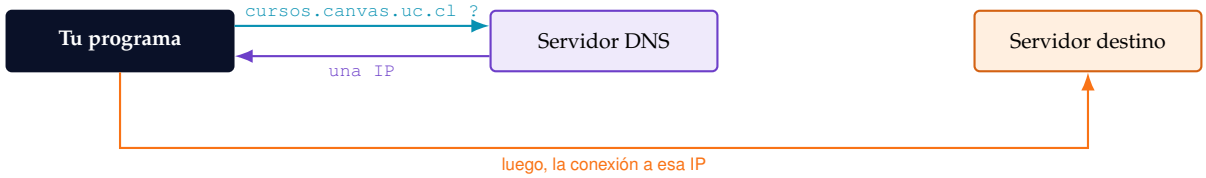
Una dirección IP identifica una **interfaz de red** dentro de un ámbito. Una dirección *privada* debe ser única dentro de su red local; una dirección *pública* enrutable debe ser única en Internet. Un dispositivo puede tener varias interfaces y, por lo tanto, varias direcciones.

Versión	Tamaño	Ejemplo	Combinaciones
IPv4	32 bits	`192.0.2.10`	2^{32}
IPv6	128 bits	`2001:db8:85a3::8a2e:370:7334`	2^{128}

No todas las combinaciones se asignan directamente a dispositivos.

3.2.2 DNS

DNS (*Domain Name System*) traduce nombres como `cursos.canvas.uc.cl` a una o más direcciones IP, mediante un sistema **distribuido y jerárquico**.



3.2.3 Puertos

Los puertos identifican **puntos de comunicación de la capa de transporte**. Tienen 16 bits, por lo que sus valores van de `0` a `65535`. Su asignación depende del protocolo y de la dirección local; TCP y UDP pueden usar el mismo número de puerto simultáneamente.

Puerto	Servicio	Puerto	Servicio
21	FTP (control)	80	HTTP
22	SSH	110	POP3
25	SMTP	143	IMAP
53	DNS	443	HTTPS

★ Un detalle de FTP

En FTP el puerto `21` se usa para **control**; el `20` puede utilizarse para **datos** en modo activo. Para contactar un servicio normalmente necesitamos **un host, un protocolo de transporte y un puerto**.

★ Analogía postal

Una dirección IP es la **dirección de un edificio** y el puerto es el **número de departamento**. La carta llega al edificio correcto gracias a la IP, pero solo la recibe el residente adecuado gracias al puerto. DNS es la **guía telefónica** que te dice la dirección del edificio a partir del nombre.

3.3 Protocolos de transporte: TCP y UDP

La información se **encapsula por capas**: TCP usa *segmentos*, UDP usa *datagramas*, y la capa IP los transporta en *paquetes IP*. Los **puertos** pertenecen a TCP o UDP; las **direcciones** pertenecen a IP.

3.3.1 TCP (*Transmission Control Protocol*)

TCP ofrece a la aplicación un **flujo ordenado y confiable de bytes**. Importante: **no conserva los límites** de los mensajes de la aplicación (si envías dos mensajes, el receptor podría recibirlos pegados o partidos). Establece una conexión mediante un *handshake* y usa números de secuencia, ACK acumulativos, retransmisiones y mecanismos de control de flujo y congestión. En el *handshake* pueden negociarse opciones como MSS, escalamiento de ventana y SACK.

3.3.2 UDP (*User Datagram Protocol*)

UDP envía **datagramas independientes** sin establecer conexión previa. **Conserva sus límites** (un mensaje enviado es un datagrama recibido), pero **no garantiza** entrega, orden ni retransmisión.

	TCP	UDP
Conexión	Sí, con *handshake*.	No hay conexión previa.
Entrega y orden	Garantizados (flujo confiable y ordenado).	No garantizados.
Límites de mensaje	**No** se conservan (flujo de bytes).	**Sí** se conservan (datagramas).
Unidad	Segmento.	Datagrama.
Usos típicos	HTTP/1.1, HTTP/2, SMTP.	DNS, telefonía y video en tiempo real, juegos, QUIC.

! HTTP/3 es la excepción

HTTP/1.1 y HTTP/2 suelen ir sobre TCP, pero **HTTP/3 usa QUIC, que corre sobre UDP**. Y tampoco todo audio o video usa UDP: también existen soluciones sobre TCP y HTTP.

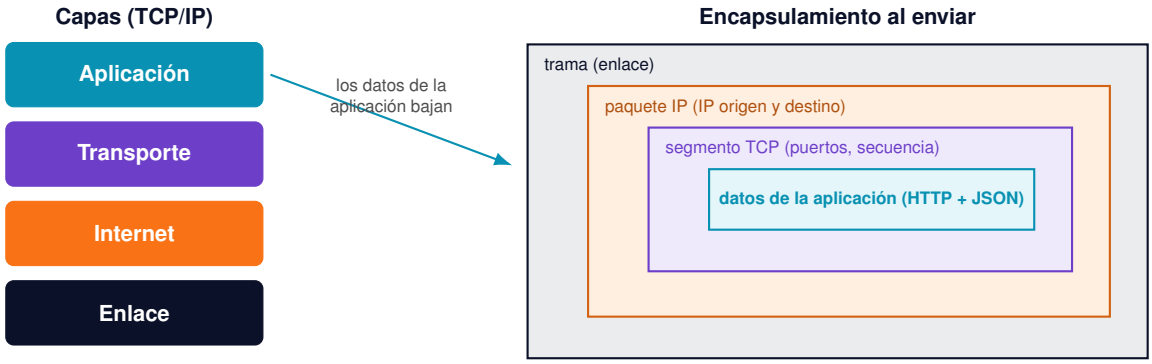


★ Analogía postal

TCP es una llamada telefónica: primero marcas y esperan que contesten (*handshake*); luego conversan en orden, y si se corta una palabra la repiten. **UDP es mandar postales sueltas**: cada una llega por su cuenta (o no llega), quizás en otro orden, pero es rapidísimo y no hay que "marcar" antes.

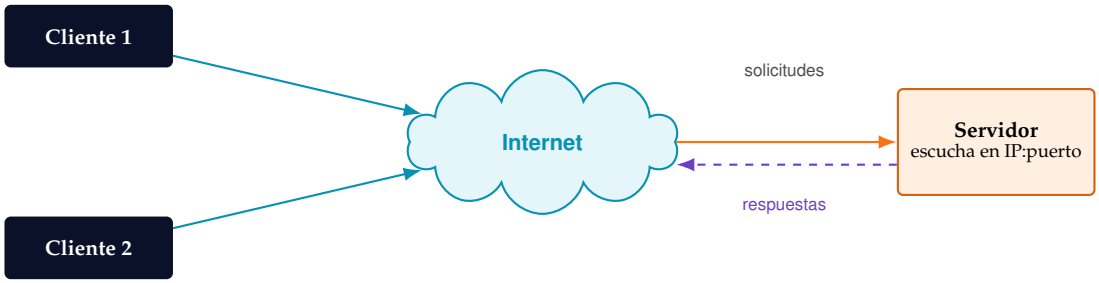
3.4 Encapsulamiento

Cada capa ofrece servicios a la capa superior y **agrega información de control**. Los modelos OSI y TCP/IP son distintos, aunque tienen correspondencias aproximadas. Nuestros programas operan en la **capa de aplicación** y usan servicios del sistema operativo, como *sockets* TCP o UDP. Los datos se **encapsulan** al bajar por la pila y se **desencapsulan** en el destino.



3.5 Arquitectura cliente-servidor

Un proceso **servidor** ofrece una funcionalidad y uno o más procesos **cliente** la consumen. Con TCP, el cliente **inicia una conexión** hacia la dirección y el puerto donde escucha el servidor. Con UDP, el cliente envía datagramas al puerto del servidor, sin una conexión TCP previa. Ambos pueden ejecutarse en computadores distintos o en la misma máquina mediante `localhost`.



✓ Resumen express: redes

- Para comunicar procesos: medio + **IP** (interfaz) + **puerto** (proceso) + transporte + protocolo de aplicación.
- **DNS** traduce nombres a IPs; los puertos van de 0 a 65535 (HTTP 80, HTTPS 443, SSH 22, DNS 53).
- **TCP**: flujo confiable y ordenado, con conexión, sin límites de mensaje. **UDP**: datagramas independientes, sin garantías. HTTP/3 corre sobre QUIC/UDP.
- **Encapsulamiento**: cada capa agrega su encabezado al bajar y se retira al subir en el destino.
- **Cliente-servidor**: el servidor escucha, el cliente inicia la comunicación.

4 Servicios web, APIs y HTTP

4.1 Servicios web

Un **servicio web** permite que programas intercambien información mediante tecnologías de la web, habitualmente HTTP. Hay dos formas típicas de usarlo:

- Un **sitio o aplicación web** suele entregar HTML, CSS y JavaScript para *una persona* que usa un navegador.
- Una **API web** expone operaciones y datos para *otras aplicaciones*, frecuentemente en JSON.
- Un mismo servidor puede ofrecer páginas web *y* endpoints de una API.

Cliente y servidor deben conocer el **protocolo**, el **formato de los datos** y el **significado de cada operación**.

4.2 API (*Application Programming Interface*)

Una **API** define cómo otros programas interactúan con un componente o servicio. Puede estar formada por funciones, métodos o *endpoints* web. Una API web documenta: la **URL base**, los *endpoints*, los **métodos HTTP**, los parámetros, los cuerpos, la autenticación, las respuestas y los códigos de estado.

★ Un navegador no es "una API"

Un navegador es un **cliente HTTP**, pero no todo servidor que visita es "una API": puede pedir documentos HTML y, además, ejecutar código que consume APIs.

★ Analogía postal

Una API es **el menú de un restaurante**: no necesitas saber cómo funciona la cocina, solo qué puedes pedir (*endpoints*), cómo pedirlo (método y parámetros) y qué te van a traer (la respuesta). Si pides algo que no está en el menú, te responden con un error.

4.3 URL (*Uniform Resource Locator*)

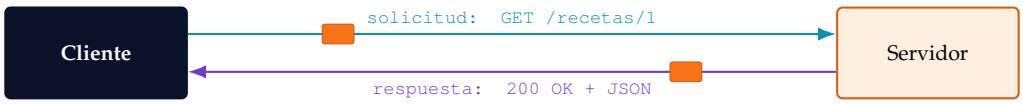
Una **URL** indica cómo localizar un recurso. Distintas URLs pueden conducir al mismo recurso. Estructura de una URL completa:



Componente	Ejemplo	Descripción
Esquema	`https`	Mecanismo de acceso.
Host	`api.ejemplo.com`	Dominio o dirección IP.
Puerto	`8080`	Opcional; HTTP usa 80 y HTTPS 443 por defecto.
Ruta	`/users/42/posts`	Recurso dentro del servicio.
Query string	`page=2&limit=10`	Parámetros después de `?`.
Fragmento	`#comentarios`	Sección del documento; **no se envía al servidor**.

4.4 HTTP (*Hypertext Transfer Protocol*)

HTTP es una familia de protocolos de **solicitud y respuesta**. Es ***stateless***: cada solicitud tiene significado por sí misma, y el servidor no "recuerda" las anteriores por obligación del protocolo. La *aplicación* sí puede guardar usuarios, compras y sesiones.



4.4.1 Métodos HTTP

La semántica principal está en el RFC 9110 (y `PATCH` se define en el RFC 5789). Los más importantes:

Método	Qué solicita
`GET`	Una **representación** del recurso. Debe ser **seguro** (no cambia el estado del servidor).
`HEAD`	Igual que `GET`, pero **sin contenido** de respuesta.
`POST`	**Procesar** el contenido enviado. Frecuentemente crea recursos, pero no se limita a eso.
`PUT`	**Crear o reemplazar** el estado del recurso identificado.
`PATCH`	Aplicar cambios **parciales**.
`DELETE`	Eliminar la asociación con el recurso.
`OPTIONS`, `CONNECT`, `TRACE`	Consultar opciones; crear un túnel; realizar diagnóstico.

4.4.2 Códigos de estado



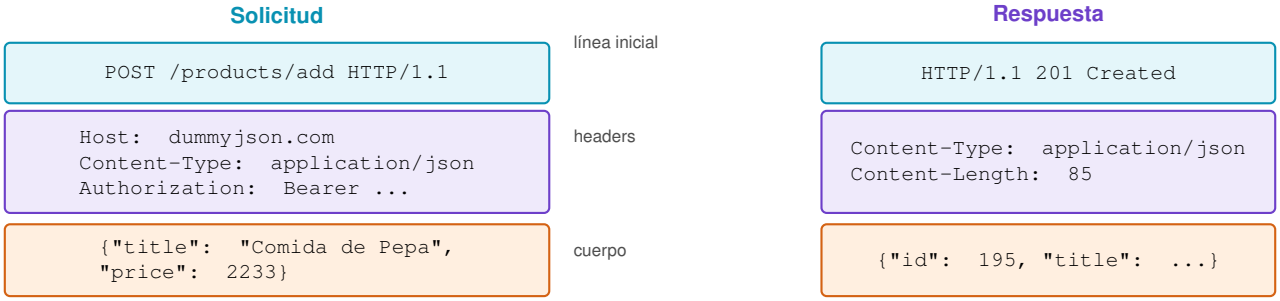
Códigos frecuentes: `200 OK`, `201 Created`, `202 Accepted`, `204 No Content`, `304 Not Modified`, `400 Bad Request`, `401 Unauthorized`, `403 Forbidden`, `404 Not Found`, `405 Method Not Allowed`, `409 Conflict`, `415 Unsupported Media Type`, `429 Too Many Requests`, `500 Internal Server Error` y `503 Service Unavailable`.

★ Regla práctica: ¿de quién es la culpa?

Un **4xx** dice "el *cliente* hizo algo mal" (pidió algo que no existe, no se autenticó, mandó datos inválidos). Un **5xx** dice "falló el *servidor*". En la Sección 6 veremos por qué un servidor bien hecho debe responder `4xx` ante entradas inválidas y no dejar que se convierta en un `500`.

4.4.3 Mensajes HTTP

Solicitudes y respuestas contienen una **línea inicial**, **campos de encabezado** (*headers*) y **contenido** opcional. El encabezado `Content-Type` describe el formato del cuerpo: por ejemplo `application/json`, `text/html; charset=utf-8` o `image/png`.



¿Qué muestra este diagrama? Ambos mensajes tienen la misma anatomía de tres partes. La solicitud dice *qué se quiere* (método + ruta) y manda el cuerpo; la respuesta dice *cómo salió* (código de estado) y manda el resultado. (Los valores son ilustrativos.)

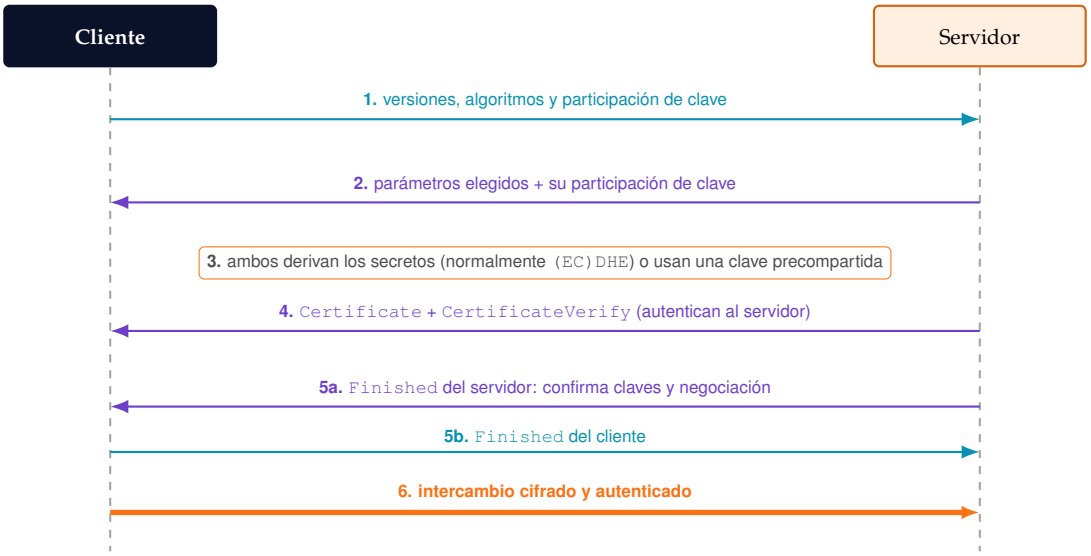
Referencias para profundizar: MDN (`developer.mozilla.org/docs/Web/HTTP`), el registro de códigos de IANA y el RFC 9110.

4.5 Bonus: TLS (*Transport Layer Security*)

TLS crea un **canal seguro**. En `https://`, HTTP se comunica mediante TLS. HTTP/1.1 y HTTP/2 normalmente usan TLS sobre TCP; HTTP/3 integra TLS 1.3 dentro de QUIC sobre UDP. SSL es el antecesor obsoleto de TLS.

TLS proporciona tres cosas: **confidencialidad**, **integridad** y **autenticación del servidor** (la autenticación del cliente es opcional).

4.5.1 *Handshake* TLS 1.3 simplificado



El intercambio `(EC)DHE` **establece las claves**; el certificado y la firma **autentican** al servidor. Un certificado se valida por su **cadena de confianza, vigencia, dominio y firmas**. La interfaz visual del navegador cambia con el tiempo, por lo que no conviene depender de un ícono específico (como el candado).

Característica	HTTP sin TLS	HTTPS
Cifrado	✗ No	✓ Sí
Puerto habitual	80	443
Autenticación del servidor	✗ No	✓ Sí
Uso recomendado	Ejercicios locales	Producción y datos reales

! "HTTP sin TLS es más rápido": no siempre

TLS agrega trabajo criptográfico, pero que HTTP sin TLS sea más rápido **no es una regla**: también influyen la reutilización de conexiones y el uso de HTTP/2 o HTTP/3. Para datos reales, usa siempre HTTPS.

✓ Resumen express: servicios web

- **API web** = operaciones y datos para otros programas (casi siempre HTTP + JSON); un sitio web entrega HTML a personas.
- **URL** = esquema + host + puerto + ruta + *query string* + fragmento (este último no llega al servidor).
- **HTTP** es de solicitud/respuesta y *stateless*. Métodos: `GET`, `POST`, `PUT`, `PATCH`, `DELETE`…; códigos: 2xx éxito, 4xx error del cliente, 5xx error del servidor.

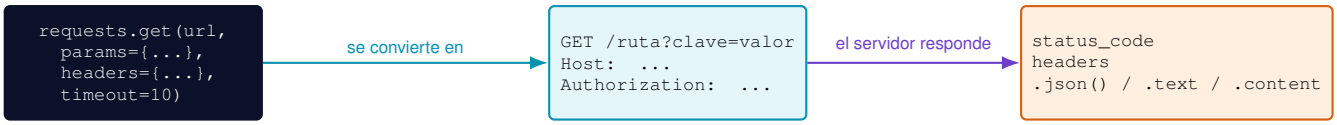
- Los mensajes tienen línea inicial, *headers* y cuerpo; `Content-Type` dice cómo leer el cuerpo.
- **TLS** da confidencialidad, integridad y autenticación del servidor; `https://` = HTTP sobre TLS.

5 Cliente de un servicio web con `requests`

La biblioteca `requests` permite enviar solicitudes HTTP desde Python. Se instala en el entorno del *kernel* con `%pip install requests` (en un notebook) o declarándola en un archivo de dependencias.

Dos hábitos que se aplican a *toda* solicitud

1. Incluir siempre un `timeout`: sin él, tu programa podría esperar indefinidamente si el servidor no responde.
2. Comprobar el estado antes de usar la respuesta: `respuesta.raise_for_status()` levanta una excepción ante un `4xx` o `5xx`.



cada argumento de `requests` controla una parte del mensaje HTTP

5.1 Consumir una API con `GET`

Para hacer una consulta con `GET` usamos `requests.get(url)`. El atributo `.status_code` entrega el código de estado, y `.json()` deserializa el cuerpo (¡el mismo JSON de la Sección 2!).

```python
import requests

url = "https://api.github.com/repos/IIC2233/Syllabus/contents/Actividades"
respuesta = requests.get(url, timeout=10)
respuesta.raise_for_status()
print(f"Status: {respuesta.status_code}")

data = respuesta.json()
for archivo in data:
    print(archivo["name"], archivo["html_url"])
```

¿Qué hace este código? Pide a la API de GitHub el listado de archivos de una carpeta de un repositorio. Si todo sale bien, `.json()` entrega una **lista de diccionarios** (uno por archivo), de la que imprimimos el nombre y el enlace.

5.1.1 Un ejemplo completo con DummyJSON

Usaremos la API educativa **DummyJSON**, que entrega datos ficticios. Una API suele documentar una **URL base** y distintos *endpoints*: aquí `products/` (productos), `recipes/` (recetas), `carts/` (carros de compra) e `image/` (imágenes generadas).

```python
BASE = "https://dummyjson.com/"
endpoint_1 = "recipes/"

respuesta = requests.get(BASE + endpoint_1, timeout=10)
respuesta.raise_for_status()
print(respuesta.status_code)          # 200

data = respuesta.json()
print("Llaves: ", data.keys())        # incluye 'recipes', 'total', ...
print(data["total"])                  # cuantas recetas hay en total
print(data["recipes"][0])             # la informacion completa de una receta
```

La respuesta es un diccionario cuya llave `"recipes"` contiene la lista de recetas. Con el *endpoint* `"products/"` funciona igual:

```python
respuesta = requests.get(BASE + "products/", timeout=10)
respuesta.raise_for_status()
data = respuesta.json()
print([producto["title"] for producto in data["products"][:2]])
```

! Los datos de un servicio externo pueden cambiar

Los resultados dependen de un servidor que no controlas: pueden cambiar el contenido, el formato o incluso dejar de funcionar. Por eso **siempre se valida la respuesta** antes de usarla (estado, `Content-Type`, existencia de las llaves que esperas).

5.2 Parámetros de una consulta

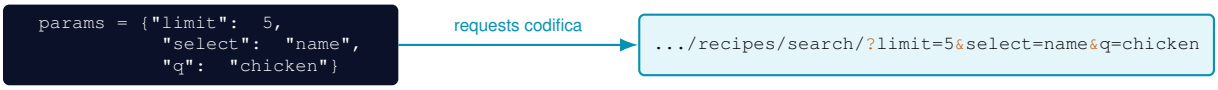
Los parámetros del *query string* se envían con el argumento `params`, que recibe un diccionario. `requests` se encarga de **codificar** espacios y caracteres especiales correctamente. DummyJSON acepta, entre otros, `limit` (cantidad de elementos), `select` (campos solicitados) y `q` (texto de búsqueda).

```python
endpoint_3 = "recipes/search/"
parametros = {"limit": 5, "select": "name", "q": "chicken"}

respuesta = requests.get(BASE + endpoint_3, params=parametros, timeout=10)
respuesta.raise_for_status()

print("URL codificada:", respuesta.url)
# https://dummyjson.com/recipes/search/?limit=5&select=name&q=chicken
print(respuesta.json()["recipes"])
```

¿Qué hace este código? Le pedimos buscar recetas con "chicken", limitar a 5 resultados y traer solo el campo `name`. Nota que `respuesta.url` muestra la **URL final** que `requests` armó a partir del diccionario: tú nunca concatenas el `?` ni los `&` a mano.



Ruta vs. *query string*. Algunos valores forman parte de la **ruta** y otros del *query string*. En el *endpoint* de imágenes de DummyJSON, el tamaño y los colores son **segmentos de ruta**, mientras que `text` y `fontFamily` son **parámetros de consulta**. Conviene distinguir ambos mecanismos y dejar que `requests` codifique los parámetros mediante `params`.

```python
consulta = "image/"
porte = "400x200/"            # ruta
color_fondo = "82944b/"       # ruta
color_fuente = "d4d2bc"       # ruta
endpoint_4 = consulta + porte + color_fondo + color_fuente

parametros = {"text": "Todos somos Pepa lovers", "fontFamily": "ubuntu"}
respuesta_imagen = requests.get(BASE + endpoint_4, params=parametros, timeout=10)
respuesta_imagen.raise_for_status()

content_type = respuesta_imagen.headers.get("Content-Type", "")
if not content_type.startswith("image/"):
    raise ValueError(f"Se esperaba una imagen, se recibio {content_type}")
```

5.3 Formatos de respuesta

! El código de estado NO dice qué formato tiene el cuerpo

Un `200 OK` solo dice que salió bien, no *qué* viene. Antes de interpretar el cuerpo hay que revisar `Content-Type`: puede ser JSON, texto, HTML, una imagen o **ningún contenido**.

Si el Content-Type es...	Usa	Obtienes
application/json	`respuesta.json()`	Un objeto de Python (`dict`, `list`...).
text/html, text/plain	`respuesta.text`	Un `str`.
image/png, etc.	`respuesta.content`	Un objeto `bytes` (datos binarios).

Para una imagen, en vez de imprimir todos los bytes se inspecciona su tipo y tamaño, y se guarda en un archivo:

```
from pathlib import Path

print("Content-Type:", respuesta_imagen.headers.get("Content-Type", ""))
print(f"Respuesta binaria de {len(respuesta_imagen.content)} bytes")

ruta_imagen = Path("data") / "img_personalizada.png"
ruta_imagen.parent.mkdir(exist_ok=True)
ruta_imagen.write_bytes(respuesta_imagen.content)
```

5.4 Enviar JSON con `POST`

`POST` solicita que un recurso **procese** el contenido enviado. Se usa frecuentemente para **crear recursos**, aunque esa no es su única semántica. Cuando la API espera JSON, usamos el argumento `json=payload`: `requests` **serializa** el diccionario por nosotros y agrega el encabezado `Content-Type: application/json`.

```
endpoint_5 = "products/add"
payload = {
    "title": "Comida de Pepa",
    "description": "Alimento esencial de una tortuga",
    "price": 2233,
    "rating": 5,
}

respuesta = requests.post(BASE + endpoint_5, json=payload, timeout=10)
respuesta.raise_for_status()

print(respuesta.status_code)    # 201
print(respuesta.reason)         # Created
print(respuesta.json())         # el mismo recurso, ahora con un 'id' asignado
```

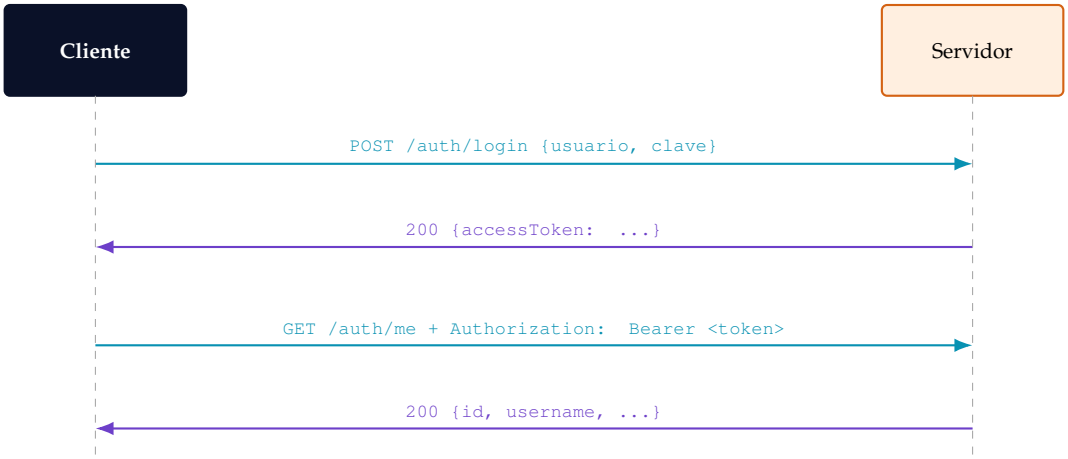
¿Qué hace este código? Enviamos un producto nuevo. La API responde `201 Created` y devuelve el recurso creado, al que le asignó un `id`. (DummyJSON simula la creación: no guarda el producto realmente.)

! `json=` y `data=` NO son lo mismo

`json=` serializa a JSON y ajusta `Content-Type`. `data=`, al recibir un diccionario, normalmente envía un **formulario** (`application/x-www-form-urlencoded`). Si la API espera JSON y le mandas `data=`, lo más probable es que responda con un error.

5.5 Autenticación mediante *headers*

Los *headers* transmiten **metadatos** de la solicitud. Un uso muy común es enviar credenciales, por ejemplo un **token** en el encabezado `Authorization`. Los nombres, el formato y los permisos dependen de cada API. El flujo típico: (1) iniciar sesión para obtener un token; (2) incluir ese token en las solicitudes siguientes.




```python
# 1) Iniciar sesion y obtener el token
url = "https://dummyjson.com/auth/login"
payload = {"username": "emilys", "password": "emilyspass", "expiresInMins": 30}

respuesta = requests.post(url, json=payload, timeout=10)
respuesta.raise_for_status()
TOKEN = respuesta.json()["accessToken"]
print("Token obtenido correctamente; no se muestra por seguridad.")

# 2) Usar el token en el header Authorization
url = "https://dummyjson.com/auth/me"
headers = {"Authorization": f"Bearer {TOKEN}"}

respuesta = requests.get(url, headers=headers, timeout=10)
respuesta.raise_for_status()
usuario = respuesta.json()
print({k: usuario[k] for k in ("id", "username", "email", "role")})

# 3) Sin token: la API rechaza la solicitud
respuesta = requests.get(url, timeout=10)
print(respuesta.status_code, respuesta.reason)    # un 4xx (401 o 403)
```

¿Qué hace este código? Primero hace `POST` con usuario y contraseña (de prueba) para obtener un `accessToken`. Luego lo manda con el esquema `Bearer` en el *header* `Authorization` para acceder a un recurso protegido. Sin ese *header*, el servidor responde con un código de error `4xx`.

! Un token es una contraseña

No lo imprimas, no lo escribas en el notebook y no lo subas a Git. Guárdalo en una variable de entorno o en un archivo ignorado por Git, y cuando escribas en la consola solo confirma que se obtuvo (como en el ejemplo).

5.6 Caso aplicado: crear una *issue* con la API de GitHub

El siguiente ejemplo prepara la creación de una *issue*. Requisitos: (1) definir propietario y repositorio; (2) crear un token con permisos específicos (`Issues: write`); (3) guardarlo en la variable de entorno `GITHUB_TOKEN`. La API espera un cuerpo JSON, que enviamos con `json=body`.

```python
import os
import requests

github_owner = "TU_USUARIO_O_ORGANIZACION"
github_repo = "TU_REPOSITORIO"
token = os.environ.get("GITHUB_TOKEN")
respuesta = None

body = {"title": "Creando una issue con la API",
        "body": "Issue creada desde Python."}

if token is None:
    print("Define la variable de entorno GITHUB_TOKEN antes de ejecutar.")
else:
    headers = {
        "Authorization": f"Bearer {token}",
```

```
        "Accept": "application/vnd.github+json",
        "X-GitHub-Api-Version": "2026-03-10",
    }
    url = f"https://api.github.com/repos/{github_owner}/{github_repo}/issues"
    respuesta = requests.post(url, json=body, headers=headers, timeout=10)

if respuesta is None:
    print("Solicitud no enviada.")
elif respuesta.status_code == 201:
    print("Issue creada:", respuesta.json()["html_url"])
else:
    print("No fue posible crear la issue.")
    print("Status:", respuesta.status_code)
    print("Detalle:", respuesta.text[:300])
```

¿Qué hace este código? Si el token no existe, **no envía nada** (y lo avisa) en vez de asumir que la solicitud salió. Si existe, hace el `POST` con tres *headers*: credenciales, formato de respuesta deseado y versión de la API. GitHub responde `201 Created` cuando la *issue* se crea; en otro caso se imprime el código y los primeros 300 caracteres del detalle.

★ Para seguir practicando

Explora otros *endpoints* de DummyJSON, consulta películas con OMDb API o busca información en PokéAPI. Revisa siempre la **documentación**, los **límites de uso** y los **requisitos de autenticación**. El envío de solicitudes en paralelo se ve en la Sección 7.

✓ Resumen express: cliente

`params=`	Parámetros del *query string* (los codifica `requests`).
`json=`	Cuerpo JSON: serializa y agrega `Content-Type: application/json`.
`headers=`	Metadatos y credenciales (`Authorization`, `Accept`...).
`timeout=`	Límite de espera; evita esperas indefinidas.
`raise_for_sta`	Levanta excepción ante `4xx/5xx`.

Antes de interpretar una respuesta: revisa el **estado** y el **`Content-Type`**. El cuerpo puede ser JSON, texto, HTML, binario o estar ausente.

6 Servidor de un servicio web con `Flask`

El lado del servidor **recibe solicitudes, ejecuta la lógica de la aplicación** y **genera respuestas**. Puede estar implementado en cualquier lenguaje, mientras respete el protocolo acordado con el cliente.

6.1 WSGI

WSGI es la **interfaz estándar** entre una aplicación web escrita en Python y un servidor WSGI. Flask implementa la *aplicación*; servidores como Waitress o Gunicorn pueden ejecutarla en producción.



La idea. El servidor WSGI se ocupa de **hablar HTTP** (recibir bytes, armar la solicitud, enviar la respuesta); tu aplicación solo define **qué responder**. Gracias a la interfaz común, puedes cambiar el servidor sin reescribir la aplicación.

Ejemplo sin Flask. El curso incluye un script (`servidor-API.py`) con una **aplicación WSGI mínima** que valida métodos y rutas para responder `404` y `405` cuando corresponde. Se ejecuta en una terminal y se prueba desde el cliente:

```
import requests
```

```python
BASE_URL = "http://localhost:4444/"
respuesta = requests.get(BASE_URL, timeout=5)
respuesta.raise_for_status()
print(respuesta.status_code, respuesta.json())

respuesta = requests.get(BASE_URL + "goodbye/", timeout=5)
print(respuesta.status_code, respuesta.json())
```

! Un solo servidor por puerto

Todos los ejemplos usan el puerto `4444`. Antes de iniciar los ejemplos con Flask, detén el servidor anterior con `Ctrl+C` en su terminal; si no, el puerto estará ocupado y el nuevo servidor no podrá iniciar.

6.2 Flask: ejemplo básico

Con Flask asociamos una **función** con una **ruta** y uno o más **métodos HTTP** mediante el decorador `@app.route`. Si la función retorna un diccionario, Flask lo convierte automáticamente a **JSON**.

```python
import random
from flask import Flask
from flask import request as flask_req

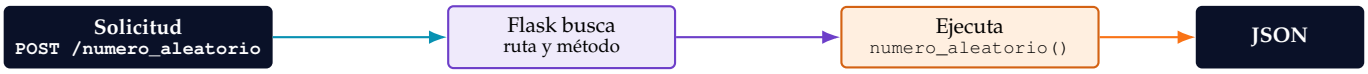
app = Flask("servidor_1")

@app.route("/", methods=["GET"])
def hello_world():
    return {"texto": "Hello, World!"}

@app.route("/numero_aleatorio", methods=["GET", "POST"])
def numero_aleatorio():
    if flask_req.method == "POST":
        numero = random.randint(0, 6)
        return {"texto": f"Tu numero es: {numero}", "metodo": "POST"}

    numero = random.randint(-4444, -11)
    return {"texto": f"Tu numero es: {numero}", "metodo": "GET"}

if __name__ == "__main__":
    app.run(host="localhost", port=4444)
```



si la ruta no existe → `404`; si el método no está permitido → `405`

¿Qué hace este código? Definimos tres posibilidades: `GET /` saluda; `GET /numero_aleatorio` devuelve un número entre -4444 y -11; `POST /numero_aleatorio` devuelve uno entre 0 y 6. La **misma ruta** se comporta distinto según el **método**, que leemos con `request.method`.

! `app.run` no es para producción

`app.run` inicia el **servidor de desarrollo** de Werkzeug: sirve para ejercicios locales, pero **no** para una aplicación pública. En producción se usa un servidor WSGI de producción (Waitress, Gunicorn) o una plataforma compatible. (En los *notebooks* del curso el servidor se levanta con `make_server` dentro de un hilo y se detiene después con `shutdown()`, para poder probarlo en la misma celda; en un script normal basta `app.run`.)

6.3 Flask: endpoints y parámetros

El ejemplo avanzado cubre cuatro mecanismos que aparecen en casi toda API: parámetros en la **ruta**, parámetros del *query string*, **JSON en el cuerpo** con validación, y **respuestas y errores**.

6.3.1 Caso 1: *endpoint* dinámico (parámetro en la ruta)

```python
@app.route("/hi/<string:username>", methods=["GET"])
def saludar(username):
    return {"texto": f"Hola {username}!"}
```

Cuando se escribe `<string:username>` en la ruta, esa sección de la URL **puede variar**: `/hi/Luna`, `/hi/Cachirulo` o `/hi/Guillermo` coinciden con este *endpoint*, y el valor se pasa a la función como argumento. También existen conversores como `<int:instruccion>`, que solo aceptan enteros.

6.3.2 Caso 2: argumentos en la URL (*query string*)

`request.args` es un `ImmutableMultiDict` con los parámetros del *query string*. Su método `get` permite indicar un **valor por defecto** y una **conversión de tipo**.

```python
@app.route("/argumentos", methods=["GET"])
def argumentos():
    nombre = flask_req.args.get("nombre", default="Guillermo", type=str)
    especie = flask_req.args.get("especie", default="Gato", type=str)
    edad = flask_req.args.get("edad", default=0, type=int)
    return {"texto": f"Hola {especie} {nombre} de {edad} anos!"}
```

Solicitud	Resultado
`/argumentos?nombre=Pepa&especie=Tortuga&`	Hola Tortuga Pepa de 70 años!
`/argumentos?nombre=Cachirulo&edad=5`	Hola Gato Cachirulo de 5 años! (especie por defecto)
`/argumentos`	Hola Gato Guillermo de 0 años! (todo por defecto)

Los parámetros que el servidor no usa (como un `informacion_extra` que el cliente envíe de más) **pueden ignorarse o rechazarse**, según el contrato de la API. En este ejemplo se ignoran. Desde el cliente, lo habitual es usar `params=`, igual que en la Sección 5.

6.3.3 Caso 3: datos JSON y validación

`request.get_json()` interpreta el cuerpo **únicamente si su tipo de contenido indica JSON**. Por eso el cliente debe enviar `Content-Type: application/json`, que `requests` agrega automáticamente al usar `json=body`.

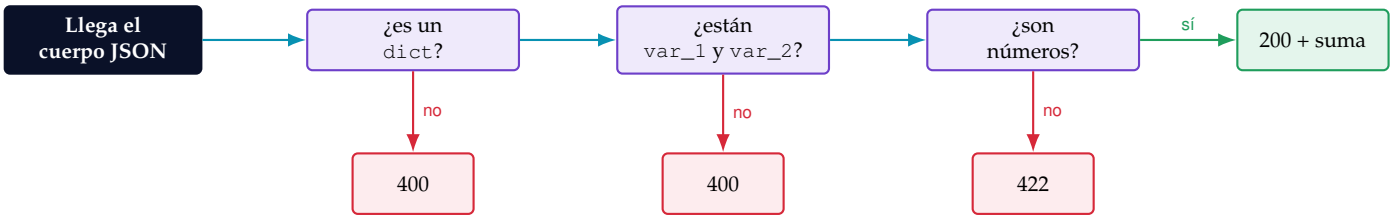
```python
@app.route("/body", methods=["POST"])
def datos_en_body():
    body_data = flask_req.get_json()

    if not isinstance(body_data, dict):
        return {"error": "El cuerpo debe ser un objeto JSON."}, 400
    if not {"var_1", "var_2"} <= body_data.keys():
        return {"error": "Faltan var_1 o var_2."}, 400

    numero_1 = body_data["var_1"]
    numero_2 = body_data["var_2"]
    if not isinstance(numero_1, (int, float)) or not isinstance(numero_2, (int, float)):
        return {"error": "Las variables deben ser numeros."}, 422

    return {"var_1": numero_1, "var_2": numero_2, "result": numero_1 + numero_2}
```

¿Qué hace este código? Valida **en capas**: primero que el cuerpo sea un objeto JSON (si no, `400`); luego que traiga las dos llaves (si no, `400`); y por último que los valores sean números (si no, `422`). Si todo está bien, devuelve la suma. Un diccionario seguido de un número (`return {...}, 400`) le dice a Flask qué **código de estado** usar.



■ Entrada inválida ⇒ error 4xx, nunca 500

Cuando una entrada del cliente es inválida, el servidor debe responder con un **error 4xx** claro, no dejar que una excepción termine convertida en `500 Internal Server Error`. Un `500` dice "yo fallé"; un `400/422` le dice al cliente *qué corregir*.

Desde el cliente, probamos con `json=`:

```
body = {"var_1": 11, "var_2": 13}
respuesta = requests.post("http://localhost:4444/body", json=body, timeout=5)
respuesta.raise_for_status()
print(respuesta.json())      # {'var_1': 11, 'var_2': 13, 'result': 24}
```

6.3.4 Caso 4: respuestas y errores

`abort(codigo)` **detiene** la vista levantando una `HTTPException`. `Response` permite controlar **cuerpo, código y `Content-Type`**. Para JSON, Flask convierte directamente un diccionario.

```
from flask import abort, Response

@app.route("/error/<int:instruccion>", methods=["DELETE"])
def error(instruccion):
    if instruccion == 0:
        abort(400)
    if instruccion == 1:
        return Response(status=418)
    if instruccion == 2:
        return Response(status=200, response="Podemos mandar textos",
                        content_type="text/plain; charset=utf-8")
    if instruccion == 3:
        return {"mensaje": "Tambien JSON"}
    abort(404)
```

Solicitud	Código	Qué entrega
`DELETE /error/0`	`400`	Error *Bad Request* (con `abort`).
`DELETE /error/1`	`418`	Respuesta sin cuerpo con un código personalizado.
`DELETE /error/2`	`200`	Texto plano (`text/plain`).
`DELETE /error/3`	`200`	JSON: {"mensaje": "Tambien JSON"}.
`DELETE /error/(otro)`	`404`	*Not Found* (instrucción desconocida).

★ ¿Por qué 200 y no 202 en el caso 3?

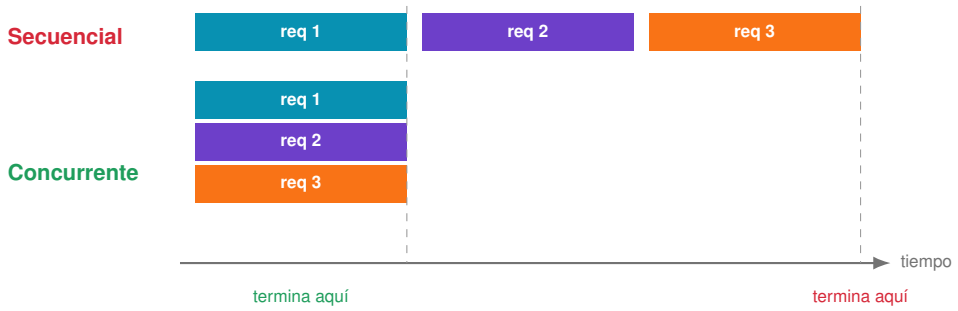
El código `202 Accepted` solo corresponde cuando el procesamiento **todavía no termina**. Como aquí la respuesta JSON es inmediata, lo correcto es `200 OK`.

✓ Resumen express: servidor

- **WSGI** conecta servidores (Werkzeug, Waitress, Gunicorn) con aplicaciones Python (Flask). `app.run` es solo para desarrollo.
- `@app.route(ruta, methods=[...])` asocia una función con ruta + método; retornar un `dict` entrega JSON; `return dict, codigo` fija el estado.
- Ruta dinámica: `<string:x>`, `<int:x>`. *Query string*: `request.args.get(clave, default=, type=)`. Cuerpo JSON: `request.get_json()`.
- **Valida** las entradas y responde `400/422`; usa `abort(codigo)` o `Response` para controlar códigos y `Content-Type`.
- Un servidor por puerto: libera el `4444` (Ctrl+C) antes de levantar otro.

7 Bonus: concurrencia y condiciones de carrera

Hasta ahora enviábamos una solicitud y **esperábamos su respuesta** antes de continuar. Cuando el trabajo está limitado por **entrada/salida** (en este caso, la espera por la red), podemos **superponer esperas** mediante hilos.



! Concurrencia no es lo mismo que paralelismo

Con hilos hablamos de **concurrencia**. En solicitudes HTTP el trabajo puede solaparse porque la mayor parte del tiempo se *espera* por la red. Esto **no** implica que todo el código Python se ejecute en paralelo sobre varios núcleos.

7.1 `ThreadPoolExecutor`

`ThreadPoolExecutor` (módulo `concurrent.futures`) administra un **grupo de hilos reutilizables**.

Elemento	Descripción
`ThreadPoolExecutor(max_wo`	Crea un *pool* de hasta `N` hilos.
`submit(fn, *args, **kwargs)`	Programa una llamada y retorna un `Future` **inmediatamente**.
`map(fn, iterable)`	Retorna un iterador; las llamadas corren concurrentemente y los resultados salen en **orden de entrada**. Pedir el siguiente puede bloquear.
`future.result()`	Espera el resultado, o **vuelve a levantar** la excepción que ocurrió en el hilo.
`as_completed(futuros)`	Entrega los `Future` **a medida que terminan**.

Al salir del bloque `with` se ejecuta `shutdown(wait=True)`, es decir, se **espera a que terminen** los trabajos pendientes.

```python
import time
from concurrent.futures import ThreadPoolExecutor, as_completed
import requests

BASE = "https://dummyjson.com/"

def obtener_receta(receta_id: int) -> dict:
    respuesta = requests.get(f"{BASE}recipes/{receta_id}", timeout=10)
    respuesta.raise_for_status()
    return {"id": receta_id, "status": respuesta.status_code,
            "nombre": respuesta.json().get("name")}

ids_recetas = list(range(1, 11))      # recetas 1 a 10

with ThreadPoolExecutor(max_workers=10) as executor:
    futuros = {executor.submit(obtener_receta, i): i for i in ids_recetas}
    resultados = [futuro.result() for futuro in as_completed(futuros)]

for r in sorted(resultados, key=lambda x: x["id"]):
    print(f"  Receta {r['id']:2d} [{r['status']}]: {r['nombre']}")
```

¿Qué hace este código? Lanza las 10 consultas **casi a la vez**. `submit` no espera: devuelve un `Future` (una "promesa" del resultado). `as_completed` entrega esos `Future` en el orden en que **terminan** (que no es el orden en que se enviaron), así que al final ordenamos por `id` para mostrar los resultados prolijos.

7.1.1 Comparación de tiempos

```python
inicio = time.perf_counter()
resultados_sec = [obtener_receta(i) for i in ids_recetas]
tiempo_sec = time.perf_counter() - inicio
print(f"Secuencial  -> {tiempo_sec:.2f}s")

inicio = time.perf_counter()
with ThreadPoolExecutor(max_workers=10) as executor:
    futuros = [executor.submit(obtener_receta, i) for i in ids_recetas]
    resultados_con = [f.result() for f in as_completed(futuros)]
tiempo_con = time.perf_counter() - inicio
print(f"Concurrente -> {tiempo_con:.2f}s")
print(f"Razon observada: {tiempo_sec / tiempo_con:.1f}x")
```

★ Medir bien

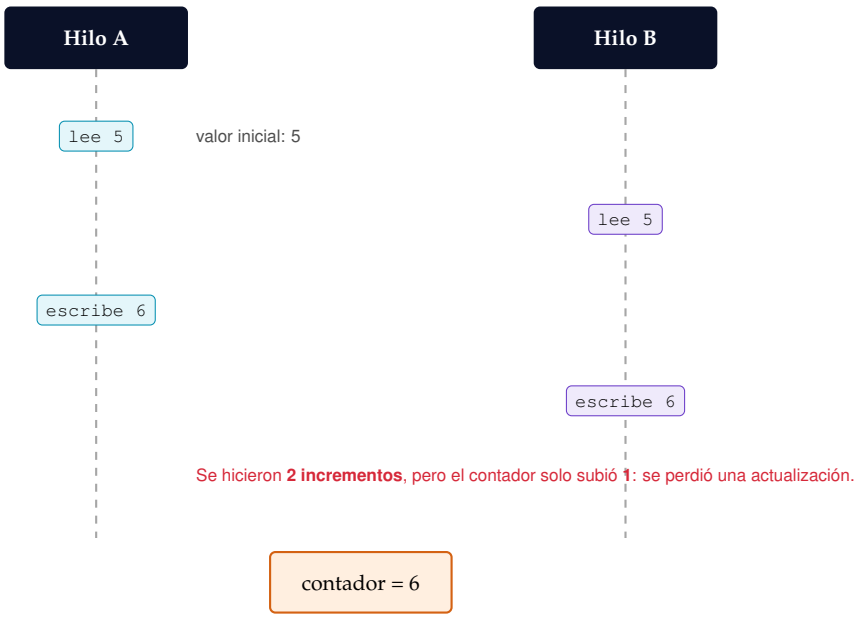
Se usa `time.perf_counter()`, apropiado para medir **intervalos**. El resultado depende de la red, el DNS, la carga del servicio, la caché y el número de hilos: es una **observación de esa ejecución**, no una aceleración garantizada.

7.2 ¿Qué es una condición de carrera?

■ Condición de carrera (*race condition*)

Ocurre cuando **el resultado depende de un orden de ejecución no coordinado** entre operaciones concurrentes. Pueden afectar memoria compartida, archivos, bases de datos y recursos modificados mediante APIs.

El caso clásico es la **actualización perdida** (*lost update*): dos hilos leen el mismo valor, cada uno calcula su nuevo valor y escribe, y una de las actualizaciones **se pisa**.



★ Analogía postal

Dos cajeros anotan en **la misma libreta** de ventas. Ambos ven que el total es $5.000, cada uno suma su venta de $1.000 en su cabeza y escribe $6.000. La libreta dice $6.000, pero se vendió $7.000. Nadie se equivocó al sumar: **el problema fue leer y escribir sin coordinarse**.

7.3 Riesgos de concurrencia en el cliente

7.3.1 Solicitudes duplicadas

Dos clics rápidos en "Comprar" pueden producir **dos `POST`**. Deshabilitar temporalmente el botón reduce el riesgo, pero **la garantía debe implementarse en el servidor**.

Un *header* como `Idempotency-Key` solo funciona si el **servidor lo reconoce**: guarda el resultado de la primera operación con esa clave y lo reutiliza ante repeticiones.

```
import uuid

clave = str(uuid.uuid4())       # una clave unica por intencion de compra
requests.post(
    "https://api.tienda.com/compras",
    json={"producto_id": 42, "cantidad": 1},
    headers={"Idempotency-Key": clave},
    timeout=10,
)
```

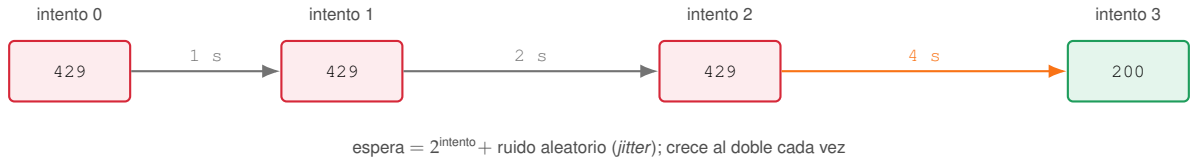
! Un *header* no basta por sí solo

Si el servidor **no implementa** `Idempotency-Key`, mandarlo no hace nada: la compra se duplicará igual. El cliente genera la clave; el servidor es quien la hace valer. (La URL de la tienda es solo un ejemplo.)

7.3.2 *Rate limiting* y reintentos

El código `429 Too Many Requests` indica que el cliente **excedió un límite** de solicitudes. No es una condición de carrera, pero puede aparecer al aumentar la concurrencia. Reglas:

- Si la respuesta incluye `Retry-After`, **respétalo**.
- Si no, usa ***backoff* exponencial** con un pequeño componente aleatorio (*jitter*), para que muchos clientes no reintenten exactamente al mismo tiempo.



espera $= 2^{\text{intento}}$ + ruido aleatorio (*jitter*); crece al doble cada vez

Nota. El gráfico muestra el *patrón* (1 s, 2 s, 4 s…; aquí sin el ruido aleatorio). En el código siguiente, con `max_reintentos=3`, hay un intento inicial y hasta tres reintentos: si el cuarto intento también devuelve `429`, la función **se rinde sin esperar más**.

```
import random

def obtener_producto_con_backoff(producto_id: int, max_reintentos: int = 3) -> dict:
    for intento in range(max_reintentos + 1):
        respuesta = requests.get(f"{BASE}products/{producto_id}", timeout=10)

        if respuesta.status_code != 429:        # solo reintentamos 429
            respuesta.raise_for_status()        # otros errores: excepcion
            return {"id": producto_id, "status": respuesta.status_code,
                    "titulo": respuesta.json().get("title")}

        if intento == max_reintentos:           # ya no quedan reintentos
            break

        retry_after = respuesta.headers.get("Retry-After")
        if retry_after is not None and retry_after.isdigit():
            espera = float(retry_after)          # el servidor manda
        else:
            espera = 2 ** intento + random.uniform(0, 0.5)

        print(f"[producto {producto_id}] 429; reintento en {espera:.2f}s")
        time.sleep(espera)

    return {"id": producto_id, "status": 429,
            "titulo": "Sin respuesta tras los reintentos"}
```

¿Qué hace este código? Reintenta **solo** ante un `429`. Prioriza el `Retry-After` del servidor; si no viene, espera 2^{intento} segundos más un ruido aleatorio de hasta medio segundo. Si agota los reintentos, entrega un resultado explícito de fracaso en vez de quedarse esperando para siempre.

Simulación con `unittest.mock`. DummyJSON no devuelve `429` fácilmente, así que la prueba **intercepta** `requests.get` con `patch` y finge respuestas: el producto 2 falla dos veces, el 4 una vez, y el resto responde `200`. El contador de llamadas del *mock* **también es estado compartido** entre hilos, por lo que se protege con su propio `Lock`.

```
import threading
from unittest.mock import MagicMock, patch

_llamadas = {}
_lock_llamadas = threading.Lock()

def mock_get(url, **kwargs):
    producto_id = int(url.split("/")[-1])

    with _lock_llamadas:                        # el contador tambien es compartido
        numero_llamada = _llamadas.get(producto_id, 0) + 1
        _llamadas[producto_id] = numero_llamada

    respuesta = MagicMock()
    respuesta.headers = {}
    if producto_id in (2, 4) and numero_llamada == 1:
        respuesta.status_code = 429
        return respuesta
    if producto_id == 2 and numero_llamada == 2:
        respuesta.status_code = 429
        return respuesta

    respuesta.status_code = 200
    respuesta.json.return_value = {"title": f"Producto simulado #{producto_id}"}
    return respuesta

with patch("requests.get", side_effect=mock_get):
    with ThreadPoolExecutor(max_workers=5) as executor:
        futuros = [executor.submit(obtener_producto_con_backoff, i) for i in range(1, 6)]
        resultados = [f.result() for f in as_completed(futuros)]

for resultado in sorted(resultados, key=lambda item: item["id"]):
    print(resultado)
```

7.4 Condiciones de carrera en servidores

La concurrencia en un servidor depende del servidor WSGI utilizado. En el experimento del curso, Werkzeug atiende **cada solicitud en un hilo** (`threaded=True`).

7.4.1 Estado compartido sin protección

Cien solicitudes incrementan un contador **global**. La pausa entre leer y escribir (`time.sleep(0.001)`) hace visible la pérdida de actualizaciones.

```
import time
from flask import Flask

app_sin_lock = Flask("contador_sin_lock")
contador_sin_lock = 0

@app_sin_lock.route("/incrementar", methods=["POST"])
def incrementar_sin_lock():
    global contador_sin_lock
    valor = contador_sin_lock          # 1) leer
    time.sleep(0.001)                  # (otro hilo puede colarse aqui)
    contador_sin_lock = valor + 1      # 2) escribir
    return {"ok": True}

@app_sin_lock.route("/contador", methods=["GET"])
def get_contador_sin_lock():
    return {"contador": contador_sin_lock}
```

El cliente lanza 100 `POST` con 20 hilos y luego consulta el contador:

```
N = 100
with ThreadPoolExecutor(max_workers=20) as executor:
    futuros = [executor.submit(requests.post, "http://localhost:4445/incrementar", timeout=5)
               for _ in range(N)]
```

```
    for futuro in as_completed(futuros):
        futuro.result().raise_for_status()

resultado = requests.get("http://localhost:4445/contador", timeout=5).json()["contador"]
print(f"Resultado esperado : {N}")
print(f"Resultado obtenido : {resultado}")       # menor que 100
print(f"Valores perdidos   : {N - resultado}")
```

¿Qué hace este código? Debería dar 100, pero como varios hilos leen el mismo valor antes de que alguno escriba, el contador **queda por debajo de 100**. Cuántos valores se pierden **varía en cada ejecución**: eso es justamente lo que vuelve tan difíciles de depurar las condiciones de carrera.

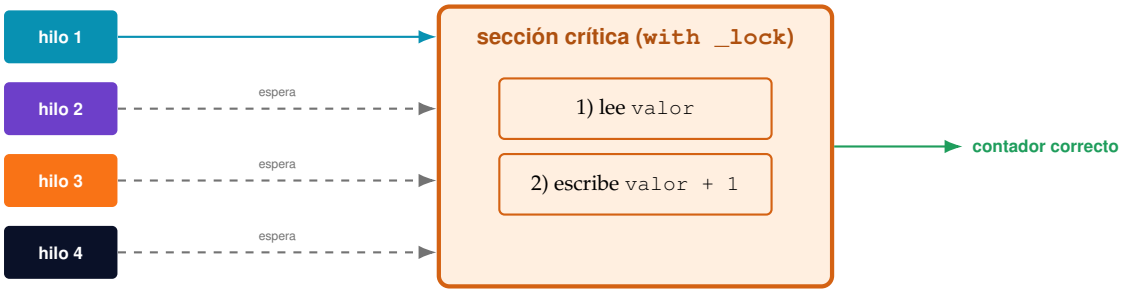
7.4.2 Proteger una sección crítica con `Lock`

`threading.Lock` permite que **un solo hilo por vez** ejecute la **sección crítica** (el trozo de código que lee y modifica el estado compartido).

```
import threading

app_con_lock = Flask("contador_con_lock")
_lock = threading.Lock()
contador_con_lock = 0

@app_con_lock.route("/incrementar", methods=["POST"])
def incrementar_con_lock():
    global contador_con_lock
    with _lock:                                  # solo un hilo a la vez
        valor = contador_con_lock
        time.sleep(0.001)
        contador_con_lock = valor + 1
    return {"ok": True}
```



¿Qué hace este código? Con `with _lock`, mientras un hilo está leyendo y escribiendo, los demás **esperan su turno**. Así cada incremento parte desde el valor que dejó el anterior, y las 100 solicitudes producen **exactamente 100**. La contrapartida: la sección crítica se ejecuta de a uno, por eso conviene mantenerla **lo más corta posible**.

! Los límites del `Lock`

Un `Lock` de Python resuelve este ejemplo **dentro de un proceso**. En producción pueden existir **varios procesos o varios servidores**, y un `Lock` **no se comparte entre ellos**. Para inventarios, reservas o saldos se requieren **transacciones, restricciones y operaciones atómicas en la base de datos**, o coordinación distribuida.

✓ Resumen express: concurrencia

- Para trabajo limitado por red, los **hilos superponen las esperas**: `ThreadPoolExecutor` + `submit`/`as_completed`. Es concurrencia, no necesariamente paralelismo.
- **Condición de carrera**: el resultado depende de un orden no coordinado (ej.: *lost update*).
- Cliente: evita **duplicados** (el servidor debe garantizarlo; `Idempotency-Key`) y respeta el ***rate limiting*** (`429`, `Retry-After`, *backoff* + *jitter*).
- Servidor: el estado compartido necesita `Lock`; pero un `Lock` solo cubre **un proceso**. Con varios procesos/servidores: base de datos transaccional y operaciones atómicas.

8 Hoja de referencia

Herramienta	Qué hace	Detalle clave
`json.dumps(obj)`	Objeto → `str` JSON	Con `cls=` usa un `JSONEncoder`.
`json.loads(texto)`	`str` JSON → objeto	Con `object_hook=` personaliza cada `dict`.
`json.dump / load`	Igual, con archivos abiertos	Abrir con `encoding="utf-8"`.
`JSONEncoder.default`	Convierte tipos desconocidos	Si no lo reconoce: `super().default(obj)`.
`JSONDecodeError`	JSON mal escrito	Subclase de `ValueError`.
`requests.get(url, params=)`	`GET` con *query string*	Siempre con `timeout=`.
`requests.post(url, json=)`	`POST` con cuerpo JSON	`json=` serializa y fija `Content-Type`.
`headers={...}`	Metadatos y credenciales	`Authorization: Bearer <token>`.
`respuesta.raise_for_status()`	Excepción ante 4xx/5xx	Úsalo antes de leer el cuerpo.
`.json() / .text / .content`	Cuerpo como objeto / texto / bytes	Según `Content-Type`.
`@app.route(ruta, methods=)`	Asocia función, ruta y método	Retornar `dict` → JSON.
`request.args.get()`	Parámetros del *query string*	`default=` y `type=`.
`request.get_json()`	Cuerpo JSON del cliente	Solo si el `Content-Type` es JSON.
`abort(cod) / Response`	Controlar código y cuerpo	Entrada inválida → 4xx.
`ThreadPoolExecutor`	Grupo de hilos	`submit` → `Future`.
`threading.Lock`	Exclusión mutua	Solo dentro de un proceso.

! Errores frecuentes

- Esperar que una **tupla** vuelva como tupla tras `dumps`/`loads` (vuelve como **lista**).
- Pasar una clase propia a `json.dumps` sin `cls=` (**`TypeError`**), o esperar que `loads` reconstruya la clase sin `object_hook`.
- Olvidar el **`timeout`** en `requests`, o usar la respuesta sin `raise_for_status()`.
- Llamar `.json()` sin revisar el `Content-Type` (**`JSONDecodeError`** ante un HTML de error).
- Usar `data=` cuando la API espera JSON (hay que usar `json=`); o concatenar a mano el *query string* en vez de `params=`.
- Imprimir o guardar **tokens** en el *notebook* o en Git.
- Dejar que una entrada inválida produzca un `500` en vez de un `400`/`422`; o usar `app.run` en producción.
- Levantar dos servidores en el mismo puerto (`4444`).
- Modificar una variable global desde varios hilos sin `Lock`; o creer que un `Lock` protege a varios procesos o servidores.

✓ Checklist de un cliente/servidor sano

1. ¿Mi solicitud tiene **`timeout`** y reviso el **estado** y el **`Content-Type`** antes de usar la respuesta?
2. ¿Mando **parámetros** con `params=`, **JSON** con `json=` y **credenciales** en `headers=` (sin exponer el token)?
3. ¿Mi servidor **valida** la entrada y responde el código correcto (`400`, `404`, `405`, `422`)?
4. ¿Hay estado compartido? ¿Qué pasa si dos solicitudes llegan **a la vez**? ¿Alcanza un `Lock` o necesito la base de datos?
5. ¿Mis clases propias tienen `JSONEncoder` y `object_hook` con un **discriminador**?

Fin del resumen • ¡Que tus paquetes lleguen siempre a destino!