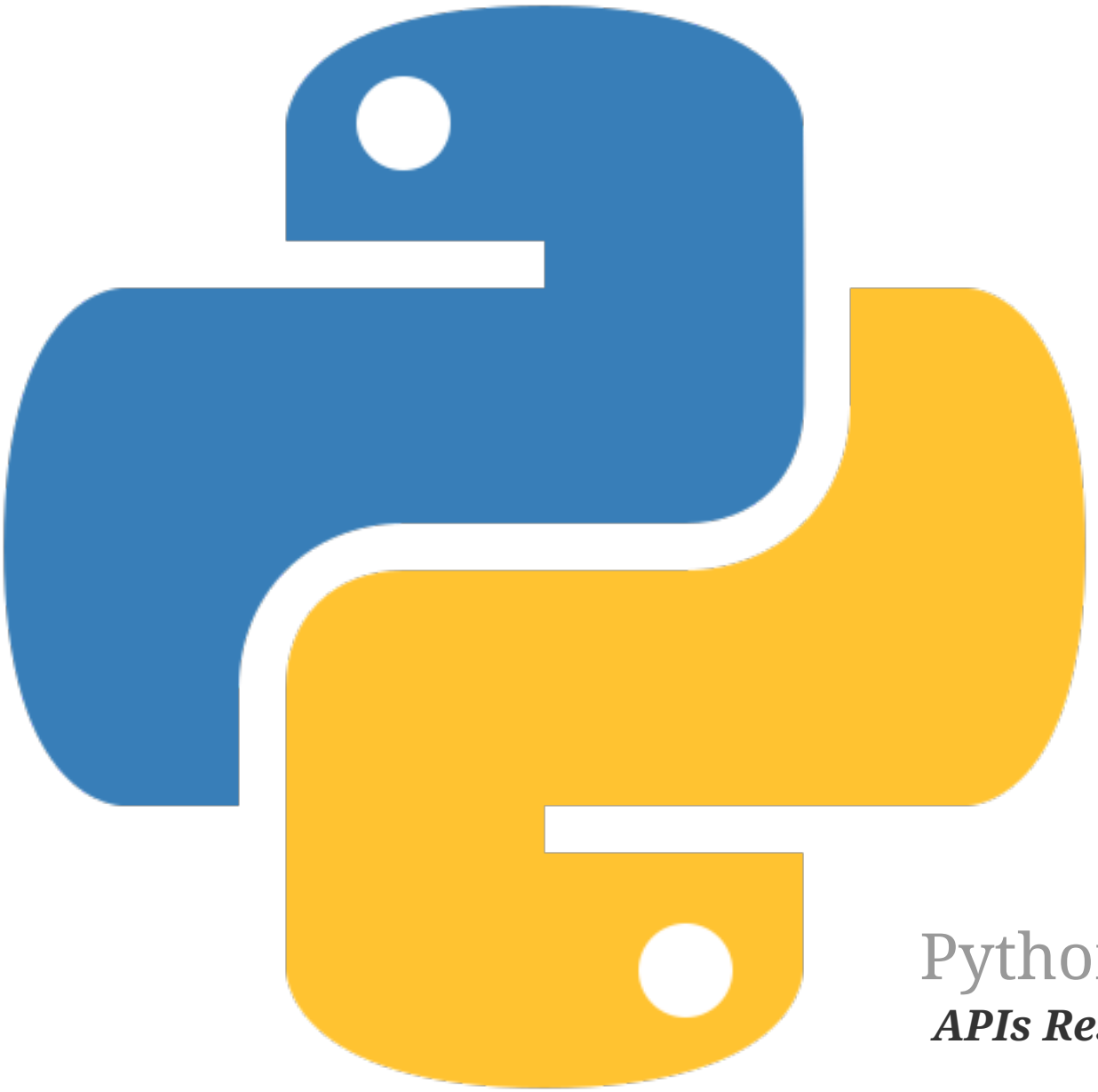


Python

APIs Rest

Pronoide

Version 2.3.0 2025-04-23

Contenidos

Chapter 1. Conceptos básicos de programación de redes

La **programación de redes** permite que las aplicaciones se comuniquen a través de una red para intercambiar datos y compartir recursos. Python nos ofrece módulos para crear aplicaciones de red, desde clientes y servidores simples hasta sistemas más complejos.

Las aplicaciones de red están en todas partes:

- Navegadores web que solicitan aplicaciones a los servidores.
- Clientes de correo electrónico que envían y reciben mensajes
- Aplicaciones de chat en tiempo real
- Plataformas de streaming de video
- …

1.1. Redes, capas e internet

El **modelo OSI** es un modelo conceptual creado por la Organización Internacional de Normalización (ISO) que sirve como referencia para entender como se comunican los dispositivos a través de una red. Divide el proceso de comunicación en capas independientes que se encargan de diferentes aspectos, desde el nivel físico hasta la interacción con el usuario.

Este modelo ayuda a diseñar y comprender las redes sin preocuparse pode los detalles de implementación específicos de cada fabricante o tecnología.

Este modelo se compone de 7 capas:

1. **Capa física**: define las especificaciones eléctricas, mecánicas y funcionales de los dispositivos físicos. Es la encargada de la transmisión de bits (0s y 1s) a través del medio físico (cable, fibra…).
2. **Capa de enlace de datos**: asegura la transferencia de datos entre dos nodos directamente conectados. Usa las direcciones MAC y protocolos como Ethernet.
3. **Capa de red**: se encarga del direccionamiento lógico (IP) y el enrutamiento de paquetes entre redes diferentes. Decide la mejor ruta para llegar desde un punto de origen hasta el punto de destino.
4. **Capa de transporte**: proporciona una transferencia de datos y con control de errores entre dos sistemas extremos (origen y destino). Aquí actúan los protocolos TCP y UDP.
5. **Capa de sesión**: es la capa que establece, gestiona y finaliza la sesión entre aplicaciones. Se encarga de controlar el dialogo (quien habla y cuando), y de la sincronización de los datos.
6. **Capa de presentación**: es la capa encargada de traducir los datos entre el formato que usa la aplicación y el formato que utiliza la red. Se encarga de la codificación de caracteres (ASCII, UTF-8…), cifrado/descifrado y compresión de datos.
7. **Capa de aplicación**: es la capa más cercana al usuario. Proporciona servicios de red a las aplicaciones de los usuarios (navegadores web, clientes de correo…). Aquí nos encontramos con protocolos como el HTTP, FTP, SMTP…

Aunque el modelo OSI es teórico y muy útil para aprender las bases, en la práctica, Internet funciona sobre el modelo TCP/IP, que tiene menos capas.

- **Capa de acceso a la red**: es la capa física más la de enlace de datos del modelo OSI.
- **Capa de internet**: es la capa de red del modelo OSI (protocolo IP).
- **Capa de transporte**: es la misma capa del modelo OSI (TCP / UDP).
- **Capa de aplicación**: es la capa de aplicación, presentación y de sesión del modelo OSI.

Este modelo es más simple y refleja cómo se implementan realmente los protocolos y servicios en las redes modernas.

Cada una de las capas solo se comunica con la capa superior e inferior:

- Con al superior para recibir datos que necesita procesar.

- Con la inferior para entregar los datos ya procesador para que los siga transmitiendo a las siguientes capas.

La forma de funcionar de forma simple, podemos decir que es:

1. La capa de aplicación genera los datos.
2. La capa de transporte añade información de control (número de puerto…).
3. La capa de red añade la dirección IP de origen y destino.
4. La capa de enlace añade la dirección MAC.
5. Por último, la capa física transmite los bits.

Este mismo proceso se realiza en el otro nodo, pero en orden contrario.

1.2. Dominios y DNS

Un **nombre de dominio** es un nombre fácil de recordar que se asocia a una dirección IP física en internet. Por ejemplo, python.org, google.com…

El **sistema de nombres de dominio** (DNS) es un servicio que traduce los nombres de dominio a direcciones IP. Cuando escribimos en nuestro navegador una URL, un servidor DNS se encarga de buscar la dirección IP correspondiente para poder conectarnos al servidor correcto.

El sistema DNS es jerárquico y distribuido, lo que significa que diferentes servidores participan en la resolución de nombres. Además, para mejorar el rendimiento, los sistemas operativos y navegadores suelen guardar en caché las respuestas DNS durante un tiempo limitado.

El DNS está estructurado como una jerarquía de niveles, de derecha a izquierda:

- .: raíz.
- .org: dominio de nivel superior (TLD).
- python: dominio de segundo nivel.

Cuando se resuelve un dominio, el proceso sigue está jerarquía:

- El servidor raíz sabe donde encontrar los TLD (.com, .es, .org…)
- El servidor TLD sabe dónde encontrar el servidor autoritativo del dominio (python.org, google.com…)
- El servidor autoritativo tiene el registro exacto que asocia el nombre con su dirección IP.

Esto es distribuido, pues no hay un solo servidor DNS que lo sepa todo. El sistema está repartido en millones de servidores por todo el mundo.

1.3. Direcciones IP

Una **dirección IP** es un identificador numérico único asignado a cada dispositivo que se conecta a la red y que se va a comunicar con otros dispositivos.

Existen dos versiones de direcciones IP:

- IPv4: compuesta por 4 números separados por puntos, como por ejemplo 127.0.0.1 o la 192.168.24.85.
- IPv6: es una versión más moderna con un espacio de direcciones mucho más grande para poder dar direcciones al número creciente de dispositivos de la época actual. Por ejemplo: 2001:0db8:85a3:0000:0000:8a2e:0370:7334.

La dirección **127.0.0.1** se conoce como **localhost** y se utiliza para referirse a nuestro dispositivo local. Es útil para probar servicios localmente sin necesidad de realizar una conexión externa.

1.4. Puertos

Un **puerto** es un número que identifica un proceso (aplicación, servicio…) específico que se está ejecutando en un dispositivo al que identifica la dirección IP. Cada aplicación que se ejecuta en un dispositivo tiene asignado un puerto para que el sistema operativo pueda identificarla y enrutar los datos correctamente.

Por ejemplo, el tráfico web (HTTP) utiliza comúnmente el puerto **80**, mientras que el tráfico de correo electrónico (SMTP) utiliza el puerto **25**.

Nosotros al levantar una aplicación en un servidor haremos que se ponga a escuchar las comunicaciones en un puerto específico. Cuando un cliente se conecte a ese puerto, el servidor se encargará de atender la petición.

Los puertos que van del **0 al 1023** son puertos que están reservados para servicios estándar. Luego podemos utilizar los puertos del **1024 al 49151** para levantar nuestras aplicaciones.

1.5. Protocolos

Los **protocolos** son conjuntos de reglas que definen cómo se transmiten los datos entre los dispositivos en una red. Algunos de los más importantes son:

- **IP**: se encarga del direccionamiento y enrutamiento de paquetes de datos.
- **TCP**: proporciona una comunicación fiable y ordenada, es decir, se asegura de que todos los paquetes lleguen a su destino sin errores y en el orden correcto. Este es el que se usa para las aplicaciones web, correo electrónico y transferencia de archivos.
- **UDP**: proporciona una comunicación más rápida pero no fiable. No garantiza que los paquetes lleguen o que lo hagan en orden. Es ideal para aplicaciones como el streaming de video o los juegos online, donde la velocidad es más importante que la precisión.

Como hemos visto antes, hay distintas capas, pues IP sería un protocolo de la capa de red, mientras que los otros dos son protocolos de la capa de transporte que trabaja encima de IP.

Podemos verlo de la siguiente forma:

- IP: es como correos, se encarga de entregar cartas, paquetes…
- TCP: es una carta certificada, por lo que te avisan de cuando llegó.
- UDP: es una carta sin certificado, por tanto puede llegar, puede no llegar, puede tardar mucho en llegar, incluso más tarde que otra carta que enviaste después.

1.6. Servicios

Los **servicios de red** son aplicaciones que se ejecutan en la capa de aplicación y que utilizan los protocolos de transporte (como TCP y UDP) para realizar la comunicación. Algunos de los ejemplos más comunes son:

- **HTTP/HTTPS**: para la navegación web.

- **FTP**: para la transferencia de archivos.
- **SMTP, POP3, IMAP**: para el correo electrónico.
- **DHCP**: para la asignación automática de direcciones IP a dispositivos en una red.
- **DNS**: para la resolución de nombres de dominio.
- **SSH**: para la conexión remota.
- **Telnet**: para la conexión a través de una terminal.

1.7. Clientes y servidores

En la programación de redes, el **modelo cliente-servidor** es una arquitectura de aplicación distribuida que divide las tareas o cargas de trabajo entre los proveedores de un recurso o servicio, llamados servidores, y los solicitantes del servicio, llamados clientes.

Python proporciona el módulo **socket** que ofrece una interfaz de bajo nivel para la comunicación en red. Esto permite crear aplicaciones cliente-servidor que se comunican a través de protocolos como TCP/IP.

Un **servidor** es un programa que espera conexiones entrantes de los clientes. Una vez que se establece una conexión, el servidor puede recibir y procesar las solicitudes del cliente y enviarle las respuestas que pedía.

Los pasos para crear un servidor son:

1. Crear un socket.
2. Vincular el socket a una dirección IP y un puerto.
3. Poner el socket en modo de escucha para aceptar conexiones.
4. Aceptar una conexión de un cliente.
5. Comunicarse con el cliente (recibir y enviar datos).
6. Cerrar la conexión.

Mientras que un **cliente** es un programa que se conecta a un servidor para enviar solicitudes y recibir respuestas a dichas solicitudes. El cliente siempre inicia la comunicación con el servidor.

Los pasos para crear un cliente son:

1. Crear un socket.
2. Conectarse a la dirección y puerto del servidor.
3. Comunicarse con el servidor (enviar y recibir datos).
4. Cerrar la conexión.

Chapter 2. Sockets

Los sockets son un mecanismo fundamental de comunicación entre procesos que permite el intercambio de datos a través de la red o dentro del mismo sistema. En Python, la librería **socket** nos proporciona una interfaz de bajo nivel para trabajar con conexiones de red, permitiendo crear tanto clientes como servidores.

Un socket puede entenderse como un punto final de comunicación bidireccional. Imagina que es como un teléfono: necesitas marcar un número (dirección IP y puerto) para establecer una conexión con otro teléfono (otro socket) y así poder intercambiar información.

Python incluye el módulo estándar **socket**, por lo que no necesitamos instalar ningún paquete adicional. Esta librería nos permite trabajar con diferentes tipos de sockets, siendo los más comunes los sockets TCP que garantizan la entrega ordenada y confiable de los datos.

2.1. Crear sockets

2.2. Conexión a un servidor

2.3. Solicitar documentos de un servidor

2.4. Cerrar conexiones

2.5. Clientes HTTP

2.6. La respuesta del servidor

2.7. Excepciones

2.8. Crear sockets

Para comenzar a trabajar con sockets en Python, primero necesitamos importar la librería y crear un objeto socket. El proceso es bastante directo y nos permite especificar el tipo de comunicación que queremos establecer.

Cuando creamos un socket, debemos indicar dos parámetros principales:

- La familia de direcciones: la más común es **AF_INET** que se usa para comunicaciones **IPv4**.
- El tipo de socket: **SOCK_STREAM** indica que queremos usar **TCP**.

```
import socket


cliente_tcp = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
print("Socket creado correctamente")
```

También podemos crear sockets UDP utilizando **SOCK_DGRAM**, aunque TCP es más común para aplicaciones que requieren confiabilidad en la transmisión de datos.

```
import socket

socket_udp = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
```

Es importante recordar que **crear un socket no establece ninguna conexión todavía**. Solo estamos preparando el mecanismo que usaremos para comunicarnos. Es como tener un teléfono listo para usar, pero aún no hemos marcado ningún número.

2.9. Conexión a un servidor

Una vez que tenemos nuestro socket creado, podemos usarlo para conectarnos a un servidor remoto. Para esto utilizamos el método **connect()** que recibe una tupla con:

- La dirección IP del servidor.
- El puerto al que queremos conectarnos.

El proceso de conexión puede fallar por varias razones:

- El servidor puede estar apagado.
- El puerto puede estar cerrado.
- Hay problemas de red.

Por eso es importante manejar estas situaciones con bloques try-except.

```
import socket


cliente_tcp = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
print("Socket creado correctamente")

try:
    servidor = "www.google.com"
    puerto = 80
    cliente_tcp.connect((servidor, puerto))
    print(f"Conectado correctamente a {servidor}:{puerto}")
except socket.error as e:
    print(f"Error al conectar: {e}")
```

2.10. Cerrar conexiones

Cerrar las conexiones es muy importante para liberar recursos del sistema y evitar problemas como sockets huérfanos. Python nos proporciona varias formas de cerrar conexiones, siendo la más básica llamar a la función **close()** del cliente creado.

Los bloques **try-finally** o el uso de los **context managers** son útiles para cerrar los sockets tanto cuando hay un error como cuando hay que cerrar el socket de forma normal.

```
import socket


cliente_tcp = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
print("Socket creado correctamente")

try:
    servidor = "www.google.com"
    puerto = 80
    cliente_tcp.connect((servidor, puerto))
    print(f"Conectado correctamente a {servidor}:{puerto}")
except socket.error as e:
    print(f"Error al conectar: {e}")
finally:
    cliente_tcp.close()
```

Una forma más elegante y pythonica es usar el **with** de los context managers. Esto garantiza que el socket se cierre automáticamente, incluso si ocurre una excepción.

```
import socket


cliente_tcp = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
print("Socket creado correctamente")

with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as cliente_tcp:
    try:
        servidor = "www.google.com"
        puerto = 80
        cliente_tcp.connect((servidor, puerto))
        print(f"Conectado correctamente a {servidor}:{puerto}")
    except socket.error as e:
        print(f"Error al conectar: {e}")
```

2.11. Solicitar documentos de un servidor

Ahora vamos a ver como obtener datos del servidor, para lo que primero vamos a configurar una API rápidamente con un paquete de Python basado en la librería **json-server** de NPM.

Vamos a instalar **json-server.py** con el siguiente comando:

```
$ pip install json-server.py
```

Una vez instalado, vamos a crear un archivo **db.json** con unos datos en formato JSON a partir de los cuales se va a crear una API Rest.

```
{
  "productos": [
    {
      "id": 1,
      "nombre": "Auriculares Bluetooth Pro",
      "descripcion": "Auriculares inalámbricos con cancelación de ruido activa y estuche de carga.",
      "precio": 59.99,
      "cantidad_vendida": 320,
      "stock": 45
    },
    {
      "id": 2,
      "nombre": "Teclado mecánico RGB",
      "descripcion": "Teclado mecánico con retroiluminación RGB y switches azules para una experiencia de escritura
precisa.",
      "precio": 74.50,
      "cantidad_vendida": 210,
      "stock": 27
    },
    {
      "id": 3,
      "nombre": "Smartwatch FitBand 2",
      "descripcion": "Reloj inteligente con monitor de frecuencia cardíaca, contador de pasos y pantalla AMOLED.",
      "precio": 89.00,
      "cantidad_vendida": 405,
      "stock": 60
    }
  ]
}
```

Y ahora ya podemos ejecutar el siguiente comando para levantar la API en el puerto 3000:

```
$ json-server db.json -b :3000
```

Si probarmos a entrar en la url http://localhost:3000/productos deberíamos de ver los productos que habíamos puesto en el archivo JSON.

Ahora ya podemos continuar con el envio de datos al servidor y la obtención de la respuesta a la petición enviada. Para esto vamos a utilizar los métodos:

- **send()**: para enviar datos.

- **recv()**: para recibir datos.

Es importante recordar que los sockets trabajan con bytes, no con strings, por lo que debemos codificar y decodificar los datos.

Cuando trabajamos con servidores HTTP, necesitamos seguir el protocolo HTTP para realizar peticiones válidas. Una petición HTTP básica, incluye:

- Un método: GET, POST, PUT...
- La ruta del recurso.
- La versión del protocolo.
- Y las cabeceras necesarias.

Vamos a crear un método que se va a encargar de realizar una petición al servidor para obtener los datos que hemos puesto en el archivo **db.json** de arriba.

```
import socket
import json


def get_recurso(servidor, puerto, ruta):
    cliente_tcp = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

    try:
        cliente_tcp.connect((servidor, puerto))

        peticion = f"GET {ruta} HTTP/1.1\r\n"
        peticion += f"Host: {servidor}\r\n"
        peticion += "Connection: close\r\n"
        peticion += "\r\n"

        cliente_tcp.send(peticion.encode("utf-8"))

        respuesta = b""
        while True:
            datos = cliente_tcp.recv(1024)
            if not datos:
                break
            respuesta += datos

        return respuesta.decode("utf-8")

    except socket.error as e:
        return f"Error en la conexión con el servidor: {e}"
    finally:
        cliente_tcp.close()

respuesta_productos = get_recurso("localhost", 3000, "/productos")
```

```
print(respuesta_productos)
```

2.12. Clientes HTTP

Los **clientes HTTP** son aplicaciones que realizan peticiones a servidores web siguiendo el protocolo HTTP.

Clientes HTTP que nos pueden venir a la mente inmediatamente tenemos los propios navegadores, Postman o curl. Mientras que a nivel de Python podemos hablar de la librería **requests** que veremos más adelante.

Un cliente HTTP debe ser capaz de construir peticiones válidas, manejar diferentes códigos de respuesta, y procesar las cabeceras y el cuerpo de la respuesta de la forma correcta.

Al final el trabajo que hace es:

1. Establece la conexión TCP con el servidor.
2. Construye la petición HTTP según el protocolo.
3. Envía la petición al servidor.
4. Recibe la respuesta.
5. Cierra la conexión (salvo que se mantenga viva con la cabecera **Connection: keep-alive**).

Para construir correctamente una petición HTTP, hay que seguir la siguiente estructura:

- En la primera línea se pone el método, la ruta y la versión del protocolo.
- Luego van las cabeceras necesarias.
- Dejamos una línea en blanco entre las cabeceras y el cuerpo de la petición.
- Al final va el cuerpo de la petición cuando hay que enviar unos datos al servidor.

Por aquí tenemos un ejemplo de una petición **GET**:

```
GET /productos/2 HTTP/1.1
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8
Accept-Encoding: gzip, deflate, br, zstd
Accept-Language: en,es-ES;q=0.9,es;q=0.8
Cache-Control: no-cache
Connection: keep-alive
Host: localhost:3000
Pragma: no-cache
```

Aquí tenemos un ejemplo de petición **POST** que envia unos datos al servidor:

```
POST /productos HTTP/1.1
Content-Type: application/json
Accept: */*
Cache-Control: no-cache
Host: localhost:3000
Accept-Encoding: gzip, deflate, br
Connection: keep-alive
Content-Length: 204
```

```
{
"nombre": "Producto prueba",
"descripcion": "Auriculares inalámbricos con cancelación de ruido activa y estuche de carga.",
"precio": 89.32,
"cantidad_vendida": 120,
"stock": 490
}
```

2.13. La respuesta del servidor

Cuando un servidor recibe una petición HTTP, responde con un mensaje que sigue un formato específico. Esta respuesta contiene información sobre el resultado de nuestra petición, incluyendo un código de estado, cabeceras con metadatos, y el cuerpo con los datos solicitados.

Cuando hemos pedido los datos de los productos anteriormente, hemos visto que obteniamos una respuesta como esta.

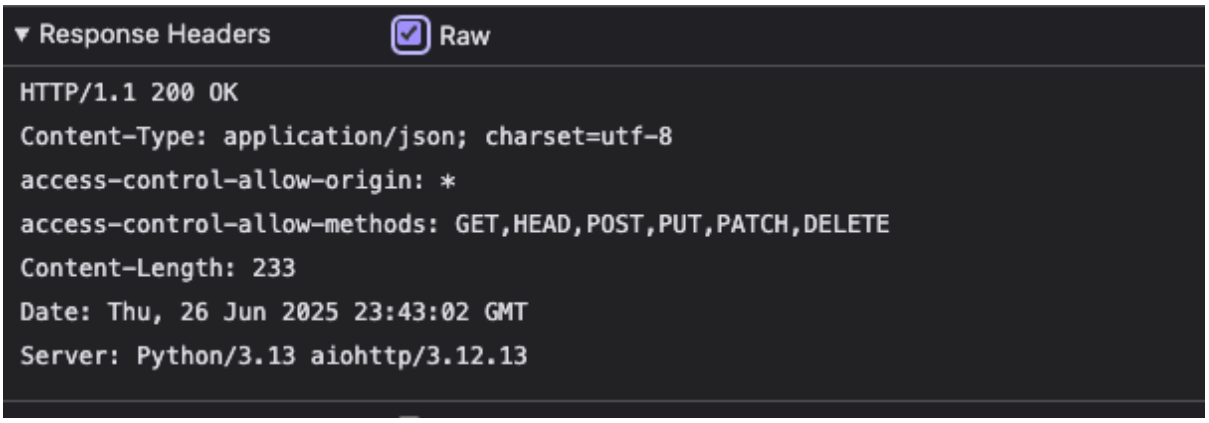
En ella podemos ver todos estos datos que se comentan:

- En la primera línea la versión del protocolo, el código de estado y el mensaje de estado asociado a dicho código.
- En las siguientes líneas nos encontramos con las cabeceras (hasta que llegamos a un doble salto de línea)
- Tras el doble salgo de línea, podemos encontrar el cuerpo de la respuesta con los datos.

```
HTTP/1.1 200 OK
Content-Type: application/json; charset=utf-8
access-control-allow-origin: *
access-control-allow-methods: GET,HEAD,POST,PUT,PATCH,DELETE
Content-Length: 233
Date: Thu, 26 Jun 2025 23:37:03 GMT
Server: Python/3.13 aiohttp/3.12.13
Connection: close

{"id": 2, "nombre": "Teclado mec\u00e1nico RGB", "descripcion": "Teclado mec\u00e1nico con retroiluminaci\u00f3n RGB y
switches azules para una experiencia de escritura precisa.", "precio": 74.5, "cantidad_vendida": 210, "stock": 27}
```

Además, si hacemos esta misma petición desde el navegador y la vemos desde la pestaña **Network** de las herramientas del desarrollador, nos encontramos lo mismo como se puede ver en la siguiente imagen.



2.14. Excepciones

Cuando trabajamos con sockets, deberíamos de manejar diferentes excepciones que pueden ocurrir durante la comunicación de red. Python tiene varias excepciones específicas para sockets que nos permiten identificar y responder a diferentes tipos de errores.

La gestión adecuada de estas excepciones previene que nuestro programa acabe cerrandose inesperadamente además que podríamos implementar ciertas estrategias para intentar solventar estos errores como por ejemplo, realizando algunos reintentos.

Entre las excepciones que nos podemos encontrar están:

- **socket.timeout**: se lanza cuando se hace una petición y se excede el tiempo límite establecido para obtener la respuesta.
- **socket.gaiaerror**: se lanza cuando se hace una petición a un dominio y este no se puede resolver.
- **ConnectionRefusedError**: se lanza cuando intentamos conectarnos a un servidor, y este rechaza la conexión, quizás porque no hay ningún servicio escuchando en el puerto.
- **BrokenPipeError**: este se da si intentamos escribir datos en un socket que ya ha sido cerrado desde el otro extremo del canal.

Ahora vamos a modificar el código de la función **get_recurso** para hacer la petición a un puerto en el que no hay nadie escuchando peticiones y vamos a controlar el error **ConnectionRefusedError** y el error **socket.gaiaerror** para mostrar un mensaje por consola.

```
import socket
import json


def get_recurso(servidor, puerto, ruta):
    cliente_tcp = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

    try:
        cliente_tcp.connect((servidor, puerto))

        peticion = f"GET {ruta} HTTP/1.1\r\n"
        peticion += f"Host: {servidor}\r\n"
        peticion += "Connection: close\r\n"
        peticion += "\r\n"

        cliente_tcp.send(peticion.encode("utf-8"))

        respuesta = b""
        while True:
            datos = cliente_tcp.recv(1024)
            if not datos:
                break
            respuesta += datos
```

```
        return respuesta.decode("utf-8")

    except ConnectionRefusedError:
        return f"No se ha podido conectar a {servidor}:{puerto} (conexión rechazada)"
    except socket.gaierror:
        return f"Error al resolver el nombre del host: {servidor}"
    except socket.error as e:
        return f"Error en la conexión con el servidor: {e}"
    finally:
        cliente_tcp.close()

respuesta_productos = get_recurso("localhost", 3000, "/productos")
print(respuesta_productos)

respuesta_producto_2 = get_recurso("localhost", 3000, "/productos/2")
print(respuesta_producto_2)

respuesta_connection_refused = get_recurso("localhost", 8375, "/productos/2")
print(respuesta_connection_refused)

respuesta_gaiaerror = get_recurso("noexiste.estehost", 3000, "/productos")
print(respuesta_gaiaerror)
```

Chapter 3. JSON

JSON es un formato para el intercambio de datos basado en texto. Por lo que es fácil de leer para tanto para una persona como para una maquina. El nombre es un acrónimo de las siglas en inglés de JavaScript Object Notation.

Los datos en los archivos JSON son pares de propiedad valor separados por dos puntos. Estos pares se separan mediante comas y se encierran entre llaves. Las claves van entre comillas dobles y el valor de una propiedad puede ser otro objeto JSON, lo que ofrece una gran flexibilidad a la hora de estructurar información. Esta estructura de datos recuerda mucho a los diccionarios de Python.


```
[
  {
    "id": 1,
    "name": "Leanne Graham",
    "username": "Bret",
    "email": "Sincere@april.biz",
    "address": {
      "street": "Kulas Light",
      "suite": "Apt. 556",
      "city": "Gwenborough",
      "zipcode": "92998-3874",
      "geo": {
        "lat": "-37.3159",
        "lng": "81.1496"
      }
    },
    "phone": "1-770-736-8031 x56442",
    "website": "hildegard.org",
    "company": {
      "name": "Romaguera-Crona",
      "catchPhrase": "Multi-layered client-server neural-net",
      "bs": "harness real-time e-markets"
    }
  },
  {
    "id": 2,
    "name": "Ervin Howell",
    "username": "Antonette",
    "email": "Shanna@melissa.tv",
    "address": {
      "street": "Victor Plains",
      "suite": "Suite 879",
      "city": "Wisokyburgh",
      "zipcode": "90566-7771",
      "geo": {
        "lat": "-43.9509",
        "lng": "-34.4618"
      }
```

```
    },
    "phone": "010-692-6593 x09125",
    "website": "anastasia.net",
    "company": {
      "name": "Deckow-Crist",
      "catchPhrase": "Proactive didactic contingency",
      "bs": "synergize scalable supply-chains"
    }
  }
]
```

Python tiene un módulo estándar llamado **json** para trabajar con este formato de datos. Y este módulo nos proporciona las siguientes funciones:

- **dumps(objeto)**: convierte un objeto de Python a un texto en formato JSON.

```
producto = {
    "nombre": "Nothing Phone 2",
    "descripcion": "El Nothing Phone 2 es un smartphone Android de gama media□alta lanzado por Nothing en julio de 2023,
con un enfoque muy marcado en el diseño y la experiencia visual",
    "en_venta": True,
    "caracteristicas": ["wifi", "nfc", "bluetooth", "usb-c"],
    "modelos": {
        "128GB": {
            "colores": ["negro", "blanco"],
            "ram": ["8GB", "12GB"],
            "precio": 349.99,
            "stock": 198,
        },
        "256GB": {
            "colores": ["negro", "gris", "blanco"],
            "ram": ["8GB", "12GB"],
            "precio": 589.99,
            "stock": 121,
        }
    }
}

print(json.dumps(producto))
```

Se puede formatear el texto JSON para que sea más legible y no nos lo muestre en una línea utilizando el parámetro **indent** con el que vamos a controlar la indentación (número de espacios por nivel de anidamiento).

```
producto = {
    "nombre": "Nothing Phone 2",
    "descripcion": "El Nothing Phone 2 es un smartphone Android de gama media□alta lanzado por Nothing en julio de 2023,
con un enfoque muy marcado en el diseño y la experiencia visual",
    "en_venta": True,
    "caracteristicas": ["wifi", "nfc", "bluetooth", "usb-c"],
    "modelos": {
        "128GB": {
            "colores": ["negro", "blanco"],
            "ram": ["8GB", "12GB"],
            "precio": 349.99,
            "stock": 198,
        },
```

```
        "256GB": {
            "colores": ["negro", "gris", "blanco"],
            "ram": ["8GB", "12GB"],
            "precio": 589.99,
            "stock": 121,
        }
    }
}

print(json.dumps(producto, indent=4))
```

- **loads(texto)**: convierte un texto en formato JSON a un objeto de Python.

```
producto_json = '''
{
  "nombre": "Nothing Phone 2",
  "descripcion": "El Nothing Phone 2 es un smartphone Android de gama media\u2011alta lanzado por Nothing en julio de
2023, con un enfoque muy marcado en el dise\u00f1o y la experiencia visual",
  "en_venta": true,
  "caracteristicas": ["wifi", "nfc", "bluetooth", "usb-c"],
  "modelos": {
    "128GB": {
      "colores": ["negro", "blanco"],
      "ram": ["8GB", "12GB"],
      "precio": 349.99,
      "stock": 198
    },
    "256GB": {
      "colores": ["negro", "gris", "blanco"],
      "ram": ["8GB", "12GB"],
      "precio": 589.99,
      "stock": 121
    }
  }
}
'''

print(json.loads(producto_json))
```

- **dump(objeto, archivo)**: guarda un objeto de Python en un archivo en formato JSON.

```
with open("producto.json", "w") as f:
    json.dump(producto, f, indent=4)
```

- **load(archivo)**: carga un archivo en formato JSON en un objeto de Python.

```
with open("producto.json", "r") as f:
    producto = json.load(f)
    print(producto)
```

Cualquier tipo de objeto de Python se puede convertir a JSON. A continuación podemos ver las relaciones entre tipos que podemos encontrarnos:

JSON	Python
string	str
number	int, float
true/false	True/False
object	dict
array	list
null	None

Chapter 4. XML

XML (eXtensible Markup Language) es un formato de texto estructurado que se usa para almacenar datos y transportarlos. Es similar a HTML, pero su objetivo no es mostrar los datos sino describir como están estructurados.

Es un formato que utiliza etiquetas personalizadas (<**nombre**>, <**precio**>, <**usuario**>...), tiene una estructura jerárquica como un árbol donde hay una etiqueta superior y otras que se muestran como hijas de los nodos padres, es decir que las etiquetas van unas dentro de otras y de esta forma podemos ver como se relacionan los datos entre sí.

El XML es un formato legible, y auto descriptivo ya que el nombre de las etiquetas ya nos dicen que es lo que están representando.

Principalmente, se usa para el intercambio de datos entre sistemas (servicios SOAP), pero tiene otros usos como el de gestionar la configuración (por ejemplo de servidores) o incluso pode realizar traducciones de aplicaciones webs (aunque suele ser más común el JSON para esta tarea).

Cada par de etiquetas de apertura o cierre representa un elemento o nodo con el mismo nombre de la etiqueta. Cada elemento puede contener texto, atributos y otros elementos anidados. Si un elemento XML está vacío (no tiene contenido), puede representarse con una etiqueta auto-cerrada.

En general resulta útil pensar en un XML como en la estructura de un árbol,

```
<persona>
  <nombre>Charly</nombre>
  <apellido>Falco</apellido>
  <edad>45</edad>
  <ciudad>San Bernardino</ciudad>
  <email ocultar="true">cfalco@gmail.com</email>
</persona>
```

Para trabajar con este tipo de textos, vamos a utilizar el paquete estándar de **xml**, más concretamente el módulo **xml.etree.ElementTree** con el que podemos parsear archivos XML, crearlos, modificarlos, buscar elementos dentro de ellos y guardarlos como archivos.

Para parsear un XML como string, vamos a utilizar la función **fromstring(xml)** pasándole como parámetro el XML, y esta función lo que nos devuelve es una instancia de un elemento **Element** que es el nodo raíz del árbol.

Las instancias de **Element** nos proporcionan las siguientes propiedades:

- **tag**: nombre de la etiqueta.
- **text**: el texto que hay antes del primer subelemento.
- **attrib**: un diccionario con los atributos del elemento.

```
import xml.etree.ElementTree as ET
```

```
usuario_xml = """
  <usuario>
    <id>2</id>
    <name>Ervin Howell</name>
    <username>Antonette</username>
    <email>Shanna@melissa.tv</email>
    <address hide="true">
      <street>Victor Plains</street>
      <suite>Suite 879</suite>
      <city>Wisokyburgh</city>
      <zipcode>90566-7771</zipcode>
      <geo>
        <lat>-43.9509</lat>
        <lng>-34.4618</lng>
      </geo>
    </address>
    <phone hide="true">010-692-6593 x09125</phone>
    <website>anastasia.net</website>
    <company>
      <name>Deckow-Crist</name>
      <catchPhrase>Proactive didactic contingency</catchPhrase>
      <bs>synergize scalable supply-chains</bs>
    </company>
  </usuario>
"""

arbol = ET.fromstring(usuario_xml)
print(f"Etiqueta: {arbol.tag}")
```

Como la etiqueta **usuario** no tiene texto como tal, sino que tiene más etiquetas, entonces tendremos que acceder a estas otras etiquetas para poder sacar el texto o atributos a los que necesitemos acceder.

Para acceder a los subelementos, podemos iterar con un bucle for el nodo actual, o podemos utilizar la función **find** para buscar a través del árbol XML y recuperar el nodo que coincide con la etiqueta especificada.

```
for etiqueta in arbol:
    print(f"{etiqueta.tag}: {etiqueta.text.strip()}")

nombre = arbol.find("name").text
print(nombre)
```

Estas operaciones solo se realizan sobre el siguiente nivel al nodo que tenemos, es decir, en este caso sobre las etiquetas que hay un nivel dentro de **usuario**. Si queremos acceder a las que hay dos niveles o más hacia abajo en el árbol tendremos que ver como recorrer el árbol.

4.1. Desplazamiento a través de los nodos

Si nuestro XML tiene múltiples nodos anidados, habrá que escribir un bucle para procesarlos todos, pero primero tendremos que encontrar estos nodos que queremos procesar.

Aquí podemos hacer uso del método **findall()** al que le pasamos el nombre de la etiqueta que queremos buscar igual que hemos hecho con el método **find**, solo que esta vez en lugar de un nodo, obtendremos una lista de nodos.

```
usuarios_xml = """
    <usuarios>
      <usuario>
        <id>1</id>
        <name>Leanne Graham</name>
        <username>Bret</username>
        <email>Sincere@april.biz</email>
        <address>
          <street>Kulas Light</street>
          <suite>Apt. 556</suite>
          <city>Gwenborough</city>
          <zipcode>92998-3874</zipcode>
          <geo>
            <lat>-37.3159</lat>
            <lng>81.1496</lng>
          </geo>
        </address>
        <phone>1-770-736-8031 x56442</phone>
        <website>hildegard.org</website>
        <company>
          <name>Romaguera-Crona</name>
          <catchPhrase>Multi-layered client-server neural-net</catchPhrase>
          <bs>harness real-time e-markets</bs>
        </company>
      </usuario>
      <usuario>
        <id>2</id>
        <name>Ervin Howell</name>
        <username>Antonette</username>
        <email>Shanna@melissa.tv</email>
        <address>
          <street>Victor Plains</street>
          <suite>Suite 879</suite>
          <city>Wisokyburgh</city>
          <zipcode>90566-7771</zipcode>
          <geo>
            <lat>-43.9509</lat>
            <lng>-34.4618</lng>
          </geo>
        </address>
        <phone>010-692-6593 x09125</phone>
        <website>anastasia.net</website>
        <company>
```

```
          <name>Deckow-Crist</name>
          <catchPhrase>Proactive didactic contingency</catchPhrase>
          <bs>synergize scalable supply-chains</bs>
        </company>
      </usuario>
    </usuarios>
"""

arbol = ET.fromstring(usuarios_xml)

nodos_usuarios = arbol.findall("usuario")
print(len(nodos_usuarios))

for nodo_usuario in nodos_usuarios:
    print(f"Usuario (id: {nodo_usuario.find("id").text}): {nodo_usuario.find("name").text}")
```

Cuando queremos hacer la búsqueda a un nivel mayor, tenemos que pasarle el path o camino de como llegar hasta ese nivel para que pueda hacer la búsqueda que queremos. Por ejemplo, si queremos buscar los nombres de las empresas de los usuarios, está vez, tendremos que pasarle al **findall** como se llega hasta esos nodos.

```
nodos_nombres_empresas = arbol.findall("usuario/company/name")

for nodo_nombre in nodos_nombres_empresas:
    print(f"Nombre: {nodo_nombre.text}")
```

4.2. Crear un XML

Para crear un XML desde 0, tenemos que tener claro que hay 2 tipos de datos que vamos a necesitar:

- **Element**: es un nodo de un árbol.
- **ElementTree**: es el árbol completo que tiene todos los nodos.

Teniendo en cuenta esto, para crear un nodo, usaremos el constructor de **Element** al que le pasaremos el nombre de la etiqueta como parámetro. Luego tendremos que añadirle los subnodos con la clase **SubElement** que recibe el nodo o elemento donde queremos crear un nuevo nodo y el nombre de este nuevo nodo.

Para modificar el texto de estos nodos, vale con asignarle un valor a la propiedad **text** de los nodos.

Por último, una vez tenemos todos los nodos del árbol, hay que crear el árbol completo pasándole a **ElementTree** el nodo raíz a partir del cual crece el árbol de etiquetas XML.

```
raiz_arbol = ET.Element("usuarios")

usuarios = [
    { "nombre": "Octavia Blake", "email": "oblake@gmail.com" },
```

```
    { "nombre": "Charly Falco", "email": "falco@gmail.com" },
]

for usuario in usuarios:
    nodo_usuario = ET.SubElement(raiz_arbol, "usuario")
    ET.SubElement(nodo_usuario, "nombre").text = usuario["nombre"]
    ET.SubElement(nodo_usuario, "email").text = usuario["email"]

nuevo_arbol = ET.ElementTree(raiz_arbol)
ET.indent(nuevo_arbol, space="  ", level=0)

nuevo_arbol.write("usuarios.xml", encoding="utf-8", xml_declaration=True)
```

Ahora ya podemos formatear el XML que hemos creado para que se muestre de una forma legible y con las indentaciones correctas, para lo que vamos a utilizar la función **indent** pasándole el árbol y luego las opciones de:

- **space**: es un string que va a usarse como espacios de indentación que queremos que haya entre cada nivel.
- **level**: es el nivel a partir del cual queremos que se aplique el espaciado.

```
raiz_arbol = ET.Element("usuarios")

usuarios = [
    { "nombre": "Octavia Blake", "email": "oblake@gmail.com" },
    { "nombre": "Charly Falco", "email": "falco@gmail.com" },
]

for usuario in usuarios:
    nodo_usuario = ET.SubElement(raiz_arbol, "usuario")
    ET.SubElement(nodo_usuario, "nombre").text = usuario["nombre"]
    ET.SubElement(nodo_usuario, "email").text = usuario["email"]

nuevo_arbol = ET.ElementTree(raiz_arbol)

ET.indent(nuevo_arbol, space="  ", level=0)

nuevo_arbol.write("usuarios.xml", encoding="utf-8", xml_declaration=True)
```

Por último, para guardar el XML en un archivo usamos el método **write** del árbol, pasándole como parámetros el nombre del archivo, el encoding y si queremos que añada una línea con la declaración del XML.

```
raiz_arbol = ET.Element("usuarios")

usuarios = [
    { "nombre": "Octavia Blake", "email": "oblake@gmail.com" },
    { "nombre": "Charly Falco", "email": "falco@gmail.com" },
```

```
]

for usuario in usuarios:
    nodo_usuario = ET.SubElement(raiz_arbol, "usuario")
    ET.SubElement(nodo_usuario, "nombre").text = usuario["nombre"]
    ET.SubElement(nodo_usuario, "email").text = usuario["email"]

nuevo_arbol = ET.ElementTree(raiz_arbol)

ET.indent(nuevo_arbol, space="  ", level=0)

nuevo_arbol.write("usuarios.xml", encoding="utf-8", xml_declaration=True)
```

4.3. Cargar un XML desde un archivo

Para cargar un XML desde un archivo, podemos utilizar la función **parse()** del módulo **xml.etree.ElementTree**. Esta función recibe como parámetro el nombre del archivo XML y devuelve un árbol XML que podemos utilizar para acceder a los datos.

```
arbol_usuarios = ET.parse("usuarios.xml")

for nodo_email in arbol_usuarios.findall("usuario/email"):
    print(f"Email: {nodo_email.text}")
```

Chapter 5. Filosofía REST

REST (REpresentational State Transfer) es un tipo de arquitectura del desarrollo web basada en el protocolo HTTP que fue definida por **Roy Fielding**. Con REST describimos cualquier interfaz entre sistemas que use directamente el protocolo HTTP para acceder a datos o realizar operaciones sobre estos, pudiendo usar cualquier formato, aunque el más común es JSON.

5.1. Características

Las principales de características de REST son:

- Interfaz uniforme
- Cliente-Servidor
- Sin estado
- Cache
- Sistema de capas

5.1.1. Interfaz uniforme

Debe existir una interfaz uniforme entre los componentes que interactúan en el sistema. Esto significa que el cliente y el servidor deben de seguir una serie de reglas para que la comunicación entre ellos sea efectiva.

- Identificación de recursos por URIs: cada recurso se identifica con una URI única.
- Manipulación de recursos a través de representaciones: el cliente puede manipular los recursos a través de las representaciones que le envía el servidor, que suelen ser en formato JSON o XML.
- Mensajes auto-descriptivos: cada mensaje debe contener toda la información necesaria para saber como procesarlo.

```
GET /inventos/123
HTTP/1.1 200 OK
Content-Type: application/json
{
  "id": 123,
  "titulo": "REST",
  "autor": "Roy Fielding",
}
```

La definición del contrato de comunicación entre el cliente y el servidor nos ayudará a que la parte del frontend y la parte del backend se puedan desarrollar de forma independiente siempre que sigan lo que se ha definido en el contrato.

En dicho contrato aparecerá como acceder a los distintos recursos, los tipos de datos que obtendremos, los datos que hay que enviar... de tal forma que podemos crear algunos mocks para poder ir implementando el frontend sin necesidad de que el backend lo esté, y viceversa.

5.1.2. Cliente-Servidor

El protocolo cliente-servidor hace que podamos utilizar una arquitectura web poco acoplada y fácilmente escalable, ya que mientras las dos partes sigan la interfaz definida, no tienen porque preocuparse por la tecnología que está utilizando la otra parte.

Aquí cada parte se encarga de una cosa, el cliente es el responsable de la UI/UX, mientras que el

servidor gestiona los datos y la lógica de negocio.

Es decir, que al backend le da igual si el front se ha desarrollado con React, Vue o HTML/CSS/JS a pelo, mientras que al front no le importa si el backend está escrito con Ruby, Java o JavaScript. Lo único que importa aquí es que al hacer una petición X al servidor, si esta petición va con la información necesaria que se había definido en la interfaz entonces el servidor podrá gestionar la petición y enviarle una respuesta que también debería de estar definida en la interfaz.

5.1.3. Sin estado

Que Rest sea sin estado, significa que el servidor no tiene que almacenar información sobre las sesiones de los clientes entre peticiones.

La información importante debe ser enviada por el cliente en las peticiones al servidor en todas sus interacciones, pudiéndose enviar en cualquier parte de la petición, como la propia URI, un parámetro de query, en las cabeceras o en el body. De esta forma se podrá procesar correctamente.

```
GET /inventos
Authorization: Bearer mi-token
```

5.1.4. Caché

El almacenamiento en caché es otra característica que podemos encontrar en REST y que hace que aumente el rendimiento del servidor.

La caché puede estar en cualquier lugar entre el cliente y el servidor. Puede estar en el propio cliente, puede estar en el servidor, en un lugar externo (como un CDN)... Con esto reducimos la carga de trabajo de nuestros servidores, sobre todo cuando estos no se han diseñado para soportar grandes cargas de trabajo.

```
HTTP/1.1 200 OK
Cache-Control: max-age=3600
```

5.1.5. Sistema de capas

Podemos insertar capas entre el cliente y el servidor que hagan de intermediarias entre ellos sin que el cliente necesite saberlo. Como por ejemplo un **gateway** que se encargue de la seguridad, el almacenamiento en caché, balanceo de carga, la comunicación con varios microservicios...

Con esto podemos conseguir que aumente la escalabilidad del sistema.

5.2. Recursos

Los recursos en REST son los elementos de información a los que podemos acceder utilizando las URIs. Estos elementos de información se transmiten entre el cliente y el servidor a través del protocolo HTTP.

La tarea de nombrar los recursos es una de las más importantes ya que si les ponemos unos buenos nombres a los recursos de los cuales queremos la información, vamos a conseguir una API bien diseñada, intuitiva, fácil de entender y de usar por los clientes.

Para poner nombres a los recursos que hagan que la API sea más intuitiva y fácil de usar para los consumidores, tendríamos que pensar en los recursos desde el punto de vista de estos, y que les sería más fácil de entender a la hora de ver la documentación.

A la hora de poner nombres a los recursos, tenemos que evitar a toda costa utilizar verbos o acciones ya que de esta parte se encargan los distintos métodos de las peticiones HTTP.

Por ejemplo, si tenemos que diseñar una API para dar información sobre pisos en alquiler o venta, un buen nombre para uno de sus recursos sería **pisos** y algo que deberíamos de evitar poner como nombre del recurso sería **getPisos**.

En las APIs nos podemos encontrar con recursos que están anidados unos dentro de otros.

Imaginemos una API que nos permite obtener información sobre algún deporte, y se han definido como recursos:

- equipos
- jugadores

Si quisiéramos obtener la lista de jugadores del Borussia Donuts, la mejor forma de acceder hasta ellos sería **/equipos/borussia-donuts/jugadores**.

Diseñar las URIs para que sean predecibles y sigan un enfoque jerárquico para acceder a los recursos hace que sea mucho más fácil de utilizar y desarrollar.

También debemos de tener cuidado con el anidamiento de los recursos y evitar tener más de 2 o 3 niveles de anidamiento.

5.3. Uso correcto de URIs

Una **URI (Uniform Resource Identifier)** es una cadena de caracteres utilizada para identificar un recurso en la web de manera única. Puede identificar cualquier recurso, ya sea a través de su ubicación o su nombre.

Por ejemplo una URI puede ser:

- Una dirección URL que identifica un recurso por su localización.

```
https://www.example.com/path/to/resource
```

- Un URN que identifica un recurso sin especificar como localizarlo, como por ejemplo el ISBN de un libro.

```
urn:isbn:123-4-56-789012-3
```

Para nuestro caso usaremos URLs.

Sabiendo que las URLs se componen de las siguientes partes, vamos a ver que tenemos que tener en cuenta para darles unos buenos nombres.

```
{protocolo}://{dominio}{:puerto}/{recurso}?{parámetros de consulta}
```

A la hora de crear las URIs de nuestras APIs tenemos que tener en cuenta las siguientes reglas para que sean más intuitivas:

- No tiene que haber varias URIs para obtener un mismo recurso, sino que a un recurso específico solo podemos acceder desde una sola URI, y no más de una.

Mal	Bien
http://www.miapp.com/api/producto/3	http://www.miapp.com/api/productos/3
http://www.miapp.com/api/productos/3	http://www.miapp.com/api/productos/3

- Hay que utilizar nombres en plural para referirnos a una colección de recursos:

Mal	Bien
http://www.miapp.com/api/producto	http://www.miapp.com/api/productos

- Las URIs no deben de llevar nombres de acciones en los recursos, es decir, que en la propia URI no debería de poner nada del estilo crear, eliminar, editar..., sino que estas acciones deberían de representarse con los métodos de las peticiones HTTP.

Mal	Bien
PUT http://www.miapp.com/api/posts/100/editar	PUT http://www.miapp.com/api/posts/100
POST http://www.miapp.com/api/crear/posts	POST http://www.miapp.com/api/posts
GET http://www.miapp.com/api/getposts	GET http://www.miapp.com/api/posts

- No tienen que depender del formato en el que esperamos la respuesta, es decir, que si queremos obtener unos datos en JSON o XML, la URI no debería cambiar, sino que tendría que ser la misma.

Mal	Bien
http://www.miapp.com/api/coches.json	http://www.miapp.com/api/coches
http://www.miapp.com/api/coches.xml	http://www.miapp.com/api/coches

- Las URIs tienen que tener una jerarquía lógica.

Mal	Bien
http://www.miapp.com/api/jugadores/equipos/valencia	http://www.miapp.com/api/equipos/valencia/jugadores

- Los recursos de la URI no deberían de llevar el filtro, sino que este debería de ir en los parámetros de consulta. Por ejemplo, si queremos obtener todas las personas cuyo nombre contiene **ry**:

Mal	Bien
http://www.miapp.com/api/nombres/contienen/ry	http://www.miapp.com/api/nombres?contienen=ry

- Si la URI tiene un segmento con más de una palabra, estas deberían de separarse con **-** y no con espacios o _. También hay que evitar las mayúsculas.

Mal	Bien
http://www.miapp.com/api/sobre_nosotros	http://www.miapp.com/api/sobre-nosotros
http://www.miapp.com/api/sobreNosotros	http://www.miapp.com/api/sobre-nosotros

5.4. Protocolo HTTP

El protocolo HTTP es un protocolo de comunicación que permite transferir información a través de la web. Es un protocolo basado en cliente-servidor, es decir que un cliente iniciará la comunicación con el servidor al que le va a mandar una petición para pedir unos datos, y el servidor cuando la reciba tendrá que procesarla para responderle con aquellos datos que le han pedido.

A través de este protocolo se pueden mandar distintos tipos de datos, pueden ir desde un archivo HTML, un script, una imagen, un JSON…

Como hemos dicho, aquí destacan dos tipos de agentes:

- El cliente: normalmente es el navegador y es el que va a iniciar la comunicación mandando una petición, por ejemplo al entrar a una web, lo primero que pedirá será el documento HTML. Una vez que el servidor se lo mande y empiece a leer el documento, es muy probable que realice más peticiones al servidor, por ejemplo para pedirle los archivos CSS o los JS.
- El servidor: es el elemento que se encuentra al otro lado del canal y que se va a encargar de recibir las peticiones, procesarlas y enviar la respuesta al cliente.

5.4.1. Características

Algunas de las características de este protocolo son:

- Es un protocolo sencillo y legible pensado para que los mensajes que se transmiten sean fácilmente interpretados por las personas.
- Es extensible, permitiendo añadir más información en los mensajes a través de las cabeceras que van en ellos.
- Es un protocolo sin estado, por lo que no guarda ningún dato entre las peticiones haciendo que estas tengan que mandarse de forma ordenada. Aunque no guarda estado, se puede hacer uso de las cookies para ello.
- Es flexible ya que permite enviar distintos tipos de datos, como texto, imágenes, video, archivos no binarios… Según se especifique en la cabecera del tipo de contenido del mensaje.

5.4.2. Tipos de mensajes

Por último, el cliente y el servidor se comunican a través de mensajes donde podemos encontrarnos dos, las peticiones y las respuestas.

Las **peticiones** se forman por los siguientes campos:

- Un método: indica la acción a realizar (GET, POST, PUT, DELETE…).
- La URI: la ruta del recurso solicitado.
- La versión del protocolo HTTP.
- Cabeceras: aportan información adicional como el tipo de contenido que se envía, las cookies, la autorización…
- El cuerpo de la petición: contiene los datos que se envían al servidor, no todos los métodos lo usan.

Mientras que las **respuestas** se forman con los campos:

- La versión del protocolo HTTP.
- El código de estado: indica el resultado de la solicitud (200, 404…).
- Un mensaje de estado: es una descripción del código de estado (Ok, Not found…).
- Cabeceras: aportan información adicional como el tipo de contenido que se envía, la fecha…
- El cuerpo de la respuesta: contiene el recurso pedido.

5.4.3. Verbos HTTP

Los métodos más utilizados a la hora de realizar peticiones HTTP al servidor son:

- Get
- Post
- Put
- Patch
- Delete

5.4.3.1. GET

El verbo **GET** se usa para pedir información sobre el recurso al que identifica la URI a la que se realiza la petición. Estas peticiones solo deben de obtener información, nunca modificarla.

Si la petición se ejecuta correctamente, esta devolverá la información pedida normalmente en un formato como JSON o XML (se indica en la cabecera **Accept** de la petición), además de obtener un código de estado **200 OK**.

En caso de fallar la petición, normalmente obtendremos un error **404 Not found** cuando el recurso pedido no se ha encontrado.

5.4.3.2. POST

El método **POST** se utiliza para crear nuevos recursos, o mejor dicho un recurso simple del tipo de recursos al que se realiza la petición. Es decir, que si hacemos un post a /**coches**, estaríamos creando un solo coche, no una colección de coches.

La información necesaria para crear el recurso indicado normalmente se envía en el **body** de la petición, y al crearse se le asigna un identificador único que es el que usaremos en una petición GET para pedir la información del recurso creado.

Si el recurso se crea sin problemas, el servidor nos responderá con un **201 Created** donde recibiremos la información del recurso creado en el formato en el que lo está esperando el cliente, o un **400 Bad Request** en caso de que la petición enviada no lleve la información necesaria.

5.4.3.3. PUT

El método **PUT** se utiliza para actualizar recursos ya existentes y que identificamos con una petición a la URI que los identifica. En caso de existir el recurso, este se cambiará por el contenido del **body** de la petición que se ha realizado.

Cuando se actualiza un recurso, el servidor nos responderá con un código de estado **200 OK**. En caso de que el recurso no exista, según el servidor, este se creará o no, y se nos devolverá un **201 Created** o un **404 Not found**.

5.4.3.4. PATCH

El método **PATCH** es similar al **PUT**. La diferencia entre ellos es que el PATCH modifica

parcialmente el recurso con la información enviada en el cuerpo de la petición.

Es decir, que si tenemos un recurso **Persona** con las propiedades: nombre, apellido, email y dni, y se hace una petición PATCH que envía en el body un nuevo valor para email, lo único que se actualizará será el email.

5.4.3.5. DELETE

El verbo **DELETE**, como su propio nombre indica, se utiliza para eliminar recursos identificados por la URI a la que se realiza la petición.

Si la petición se realiza correctamente y el recurso se elimina, obtendremos una respuesta con el código de estado **200 OK**, y en el caso de que no vayamos a devolver ningún dato, podría devolverse un **204 No Content**.

En caso de que el recurso a eliminar no exista, entonces obtendremos como código de estado un **404 Not found**. Si por ejemplo el usuario no tiene permisos para eliminar el recurso, entonces obtendremos un **403 Forbidden**.

5.4.4. Códigos de estado

Las respuestas del servidor a las peticiones HTTP que ha realizado el cliente comunican el estado de dichas peticiones mediante los códigos de estado.

Estos códigos se agrupan en 5 grupos:

- 1xx: respuestas con carácter informativo.
- 2xx: indican que la petición realizada se ha recibido, entendido y aceptado, es decir, que todo ha ido bien.
- 3xx: indican que el cliente tiene que realizar una acción adicional, como por ejemplo una redirección.
- 4xx: indican que ha ocurrido un error por culpa del cliente.
- 5xx: indican que hay un error en el servidor y no se ha podido procesar la petición.

En el protocolo HTTP podemos encontrar muchos códigos de estado, pero a continuación vamos a ver algunos de los más utilizados.

- 200 OK: la petición se ha procesado correctamente.
  - Ejemplo: una petición GET a /users devuelve una lista de usuarios.
- 201 Created: la petición se ha procesado bien y se ha creado un recurso.
  - Ejemplo: una petición POST a /users crea un nuevo usuario.
- 204 No Content: la petición se ha procesado sin problemas y no se devuelve nada en el cuerpo de la respuesta.
  - Ejemplo: una petición DELETE a /users/1 elimina el usuario sin retornar datos.
- 301 Moved Permanently: indica que el recurso que se ha pedido se encuentra en una nueva URI.
  - Ejemplo: una antigua URL redirige permanentemente a una nueva ubicación.
- 400 Bad Request: la petición no se ha procesado correctamente porque el servidor no ha sido capaz de entenderla, quizás porque faltaba algún parámetro o se ha encontrado un error de sintaxis.
  - Ejemplo: una petición POST a /users sin el campo `email` puede devolver este error.
- 401 Unauthorized: la petición require autenticación y no se ha proporcionado o es inválida.
  - Ejemplo: una petición GET a /profile sin un token de autenticación válido devuelve este error.
- 403 Forbidden: el servidor reconoce la petición, pero se niega a autorizarla porque no cumplimos con alguna política de seguridad para poder acceder al recurso.
  - Ejemplo: intentar acceder a /admin sin los permisos necesarios devuelve este error.
- 404 Not Found: el servidor no encuentra el recurso pedido en la petición.
  - Ejemplo: una petición GET a /users/999 para un usuario inexistente devuelve este error.
- 500 Internal Server Error: indica que el servidor no puede procesar la petición por un problema en el propio servidor.

    - Ejemplo: un fallo inesperado en el servidor durante el procesamiento de una solicitud.
- 503 Service Unavailable: indica que el servidor está temporalmente inactivo quizás por tareas de mantenimiento o por una sobrecarga de peticiones.
    - Ejemplo: un servidor en mantenimiento puede devolver este error.

Podemos ver todos los códigos en https://developer.mozilla.org/es/docs/Web/HTTP/Status.

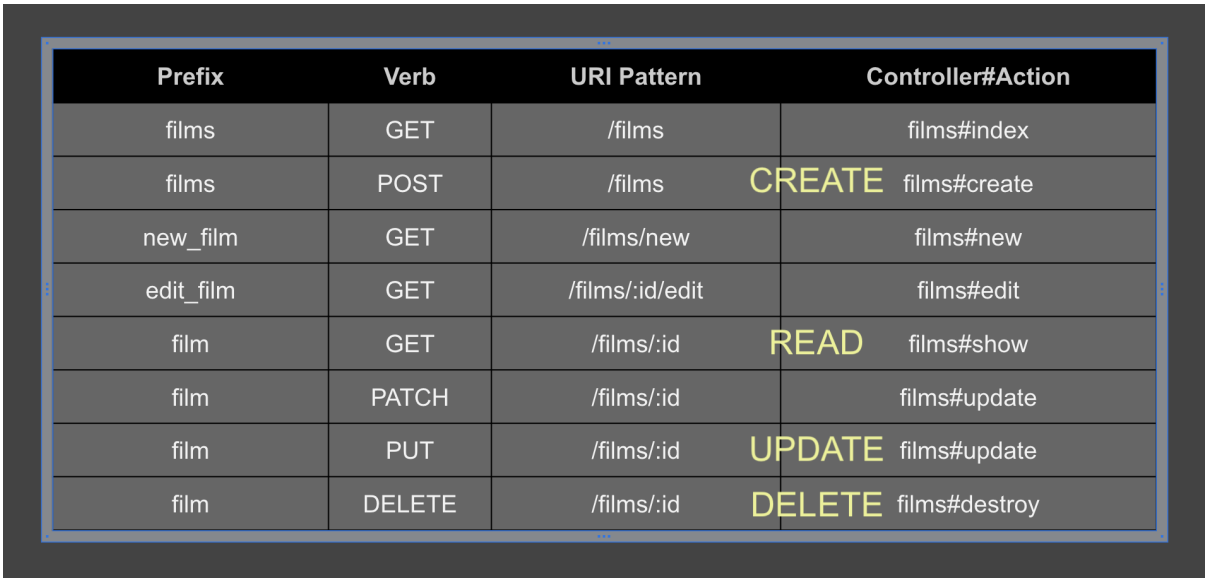
También hay códigos no oficiales que se utilizan en algunas situaciones por algunas empresas o aplicaciones: https://en.wikipedia.org/wiki/List_of_HTTP_status_codes#Unofficial_codes.

5.5. CRUD

CRUD es un acrónimo para Create, Read, Update y Delete. Estas son cuatro operaciones básicas que describen las operaciones que solemos realizar sobre las BBDD para persistir los datos.

Create	Crear un nuevo objeto
Read	Obtener información de un objeto
Update	Actualizar un objeto
Delete	Eliminar un objeto

Ahora imaginaos que tenemos una aplicación para gestionar peliculas, en la que podemos realizar distintas operaciones para obtener datos o persistirlos en la BBDD, pues podríamos tener una serie de recursos Rest como los de la siguiente imagen.



Si nos fijamos en la imagen anterior, normalmente estas operaciones CRUD se suelen asociar a unos métodos HTTP según os muestro en la siguiente tabla.

Operación CRUD	**Método HTTP**
Create	POST
Read	GET
Update	PUT y PATCH
Delete	DELETE

Por tanto, siguiendo con el ejemplo de la imagen, si quisieramos crear una nueva película tendríamos que realizar una petición **POST a http://www.peliculasdb.com/api/films**, mientras que si quisieramos obtener la información de la pelicula con id 62 tendríamos que realizar una petición **GET a http://www.peliculasdb.com/api/films/62**.

5.6. HATEOCAS

HATEOAS (Hypermedia As The Engine Of Application State) es un principio que asegura que cada vez que se hace una petición al servidor y este le responde, dicha respuesta llevará entre sus datos otros enlaces o hipervínculos que tienen relación con los que se han pedido.

Al llevar dichos enlaces, es más fácil para el cliente como interactuar con los recursos sin que este tenga que conocer todas las URIs, y es el propio servidor el que guía al cliente a través de la aplicación proporcionando enlaces relevantes a los recursos disponibles.

Supongamos que tenemos una API de series y hacemos una petición GET a **http://www.seriesdb.com/api/serie/game-of-thrones** para la que obtenemos la siguiente respuesta:

```
{
  "titulo": "Game of Thrones",
  "description": "...",
  "temporadas": [...],
  "actores": [14, 55, 89, 174, 230],
}
```

Parece que está bien, que obtenemos la información que queríamos, hemos recibido el título, la descripción, las temporadas con los capítulos, y actores y actrices que actúan en ella.

Pero ¿qué pasa ahora si queremos obtener la información del actor con el id 174?, pues que tenemos que ir a la documentación de la API para ver a que URL tenemos que realizar la siguiente petición GET que nos va a devolver los datos del actor.

Vamos a la documentación, y encontramos que la URL a la que hay que hacer la petición es **http://www.seriesdb.com/api/actor/:id**, pues podemos coger nosotros y añadir al endpoint el identificador del actor que queríamos y ya la tendríamos preparada para obtener sus datos.

¿Y si ahora deciden cambiar la API a **http://www.seriesdb.com/api/v2/actor/:id**?

Pues tendríamos que ir cambiando en el código de nuestra aplicación todas las peticiones y añadirles el **v2** a todas las URLs que ya habíamos construido.

Entonces, estos problemas no los tendríamos si la API siguiera el principio HATEOAS por el cual se indica que en lugar de mandar esos identificadores, mandemos las URLs ya formadas, de tal forma que aquellas aplicaciones que consuman la API no tengan que buscar entre la documentación, no tengan que crear ellos mismo las URLs y sobre todo que ante un cambio en la API no tengan que realizar cambios en las aplicaciones.

Por tanto, la petición anterior quedaría de la siguiente forma si la API cumpliera este principio.

```
{
  "titulo": "Game of Thrones",
  "description": "...",
  "temporadas": [...],
```

```
  "actores": [
    "http://www.seriesdb.com/api/v2/actor/14",
    "http://www.seriesdb.com/api/v2/actor/55",
    "http://www.seriesdb.com/api/v2/actor/89",
    "http://www.seriesdb.com/api/v2/actor/174",
    "http://www.seriesdb.com/api/v2/actor/230"
  ],
}
```

Podemos encontrar un ejemplo de este principio en algunas APIs como las siguientes:

- https://pokeapi.co/
- https://anapioficeandfire.com/

Por ejemplo, si entramos en el primer link, la API de Pokemon, podemos ver que cuando se realiza una petición de un personaje dado su identificador, este recibe entre todos sus datos algunos enlaces a otro tipo de información como sus habilidades, movimientos, tipos...

Chapter 6. Módulo requests

El módulo **requests** de Python permite realizar peticiones HTTP de una forma sencilla. Este módulo nos facilita la interacción con APIs REST y servicios web.

Al no ser un módulo estándar de Python, debemos instalarlo antes de poder utilizarlo. Para ello, podemos utilizar el gestor de paquetes **pip** y lanzar el siguiente comando:

```
$ pip install requests
```

Este módulo proporciona métodos para cada petición HTTP:

- GET
- POST
- PUT
- DELETE
- ...

Donde la estructura básica de una petición es **requests.<metodo>(<url>, <parametros>)**.

Esta llamada devuelve una respuesta que contiene información importante como:

- **status_code**: el código de estado de la respuesta HTTP.
- **json()**: convierte la respuesta JSON a un diccionario o lista de Python.

6.1. Petición GET

La petición **GET** se usa para obtener datos del servidor.

```
def get_posts():
  response = requests.get(url)
  data = response.json()
  print(data)
  print(response.status_code)


  posts = []
  for key, post in data.items():
    posts.append({
      "id": key,
      **post
    })

  return posts
```

6.2. Petición POST

La petición **POST** se utiliza para crear nuevos recursos.

```
def create_post(post):
  response = requests.post(url, json=post)
  print(response.status_code)

  data = response.json()
  return data


post = {
  "title": "Nuevo post",
  "body": "Contenido del post",
  "userId": 1
}

created_post = create_post(post)
```

6.3. Petición PUT

La petición **PUT** se utiliza para actualizar un recurso completo.

```
def update_post(post_id, post_to_update):
  response = requests.put(f"{url}/{post_id}", json=post_to_update)
  print(response.status_code)

  data = response.json()
  return data


otro_post = {
  "title": "Nuevo post 3000",
  "body": "Contenido del post 3000",
  "userId": 2
}

updated_post = update_post(3000, otro_post)
```

6.4. Petición DELETE

La petición **DELETE** se utiliza para eliminar un recurso.

```
def delete_post(post_id):
  response = requests.delete(f"{url}/{post_id}")
```

```
  return True


deleted = delete_post(created_post["name"])
```

6.5. Petición PATCH

La petición **PATCH** se utiliza para actualizar parcialmente un recurso.

```
def update_user_id(post_id, post_user_id):
  response = requests.patch(f"{url}/{post_id}", json=post_user_id)
  print(response.status_code)

  data = response.json()
  return data

post_user_id = {
  "userId": random.randint(1, 10)
}

updated_user_id = update_user_id("-O99H6SwWrU8dSv93U8l", post_user_id)
```

6.6. Query parameters

Los **query parameters** o parámetros de consulta se usan para mandar información al servidor con los cuales podemos hacer operaciones como filtrado, ordenación, paginación... y así obtener unos resultados más precisos.

Estos parámetros se añaden como valor de la clave **params** en la llamada a la petición, y es un diccionario con las claves y valores que definen el tipo de operaciones a realizar o de información necesaria que hay que enviar.

```
def get_posts_by_user(user_id):
  params = {
    "userId": user_id
  }

  response = requests.get(url, params=params)

  data = response.json()
  return data


posts = get_posts_by_user(2)
```

6.7. Gestión de errores

El módulo requests proporciona varias formas de manejar errores en las peticiones HTTP:

- Códigos de estado comunes de error:
  - 4xx: 400 (Bad Request), 401 (Unauthorized), 403 (Forbidden), 404 (Not Found)
  - 5xx: 500 (Internal Server Error), 502 (Bad Gateway), 503 (Service Unavailable)
- Excepciones principales:
  - HTTPError: para errores 4xx y 5xx.
  - ConnectionError: problemas de conexión.
  - Timeout: tiempo de espera excedido.
  - TooManyRedirects: demasiadas redirecciones.

Aquí para lanzar la excepción y poder capturarla tenemos que utilizar el método **raise_for_status()** de la respuesta.

```
def get_404_raise_error():
  response = requests.get("https://jsonplaceholder.typicode.com/posts/999")

  try:
    response.raise_for_status()
  except requests.exceptions.HTTPError as err:
    print(f"Mi HTTP error: {err}")

def post_400_raise_error():
  response = requests.post("https://reqres.in/api/register", headers={"x-api-key": "reqres-free-v1"}, json={"email":
"cfalco@gmail.com"})

  try:
    response.raise_for_status()
  except requests.exceptions.HTTPError as err:
    print(f"Mi HTTP error: {err}")


get_404_raise_error()
post_400_raise_error()
```

6.8. Parámetros adicionales

El módulo requests acepta varios parámetros adicionales para personalizar las peticiones:

- **headers**: para añadir cabeceras HTTP a las peticiones.
- **auth**: para enviar los datos para la autenticación.
- **timeout**: para establecer un tiempo límite de espera.
- **data**: para enviar datos de formulario o datos binarios.
- **files**: para enviar archivos.

```
# Headers y Timeout
```

```
def get_timeout_error():
  response = requests.get("https://reqres.in/api/users", params={"delay": 5}, headers={"X-API-Version": "2.0.1", "x-api-key": "reqres-free-v1"}, timeout=3)
  print(response.status_code)

get_timeout_error()

# AuthBasic
def post_with_basic_auth():
  response = requests.post("https://reqres.in/api/login", auth=requests.auth.HTTPBasicAuth("user", "password"))
  print(response.status_code)

post_with_basic_auth()
```

Chapter 7. Flask

Flask es un **microframework** web para Python creado por Armin Ronacher en 2010. Se caracteriza por su simplicidad, flexibilidad y facilidad de aprendizaje, convirtiéndolo en una opción tanto para principiantes como para desarrolladores experimentados.

El término **microframework** no significa que Flask sea limitado o que solo sirva para aplicaciones pequeñas. Se refiere a que mantiene un núcleo simple y extensible.

Este microframework se basa en:

- **Werkzeug**: es una librería WSGI (Web Server Gateway Interface) que nos proporciona herramientas para manejar las peticiones y respuestas HTTP, hacer el enrutamiento de URLs, manejar los datos de formularios…
- **Jinja2**: es un motor de plantillas que nos permite separar la lógica de la presentación, y nos permite reutilizar código HTML.

Flask es muy utilizado sobre todo en startups por tener una curva de aprendizaje baja, además de que tiene una comunidad muy activa y por tanto hay mucha documentación de donde aprender. Además te da libertad total para organizar las aplicaciones como uno quiera.

Si comparamos Flask con otros frameworks, nos encontramos que:

- A diferencia de **Django**, que es más estructurado y viene con muchas herramientas listas para usar, Flask te da libertad para elegir como implementar cada parte del proyecto.
- Y si lo comparamos con **FastAPI**, Flask es más antiguo y maduro y por tanto tiene más herramientas y documentación, mientras que este otro es más moderno y destaca en rendimiento.

7.1. Instalación y primeros pasos

Para instalar Flask, vamos a lanzar el siguiente comando:

```
$ pip install Flask
```

Una vez instalado, para crear una primera aplicación con Flask, vamos a crear un archivo **app.py** en el que importaremos Flask. Justo después vamos a crear la aplicación de Flask, para lo que tendremos que llamar a la función **Flask** que hemos importado previamente. A esta función le vamos a pasar la propiedad de Python ***name*** la cual necesita Flask para poder ubicar recursos como archivos estáticos (css, js, imágenes...), plantillas... o determinar rutas relativas de la aplicación.

Justo después vamos a poner nuestra primera ruta con el decorador **app.route(ruta)** que decorará una función. Esto indica que cuando se haga una petición a dicha ruta, se tiene que ejecutar la función decorada, la cual vamos a hacer que devuelva como respuesta un HTML con un título de bienvenida.

Por último, vamos a ejecutar la aplicación cuando se ejecute este archivo, y activaremos el modo **debug** mientras estamos en la fase de desarrollo que nos da algunas ventajas como:

- Recarga automática
- Debugger interactivo
- Mensajes de error detallados

También le vamos a indicar que utilice el puerto **8080** con el parámetro **port**.

```
from flask import Flask

app = Flask(__name__)

@app.route("/")
def inicio():
    return """
        <h1>Bienvenido a la aplicación de los saludos</h1>
    """

if __name__ == "__main__":
    app.run(port=8080, debug=True)
```

Ahora para ejecutar la aplicación solo tenemos que ejecutar el script, y el servidor se pondrá a escuchar peticiones en http://127.0.0.1:8080 por defecto.

Ahora ya podemos entrar desde el navegador en la url que nos indica el servidor para ver la aplicación.

7.2. Rutas dinámicas

Ahora vamos a añadir una ruta más, en la cual le vamos a pasar un nombre y la ruta utilizará dicho nombre para mostrar un mensaje personalizado. Esto se consigue utilizando **rutas dinámicas**, que son rutas que pueden contener variables. En este caso nuestra variable será **nombre**, y se le indica a Flask que es una variable y no un texto fijo porque lo ponemos entre los símbolos <**variable**>.

La función que decoramos con el decorador de la ruta va a recibir esta variable como parámetro.

```
from flask import Flask

app = Flask(__name__)

@app.route("/")
def inicio():
    return """
        <h1>Bienvenido a la aplicación de los saludos</h1>
    """

@app.route("/saludar/<nombre>")
def saludar(nombre):
    return f"""
        <h1>Hola {nombre}</h1>
    """

if __name__ == "__main__":
    app.run(port=8080, debug=True)
```

Ahora vamos a añadirles unos enlaces a los 2 HTMLs que devuelven estas rutas para no tener que escribir a mano las URLs en el navegador.

```
from flask import Flask

app = Flask(__name__)

@app.route("/")
def inicio():
    return """
        <h1>Bienvenido a la aplicación de los saludos</h1>
        <ul>
            <li><a href="/saludar/charly">Saludar a Charly</a></li>
            <li><a href="/saludar/Octavia">Saludar a Octavia</a></li>
        </ul>
    """

@app.route("/saludar/<nombre>")
def saludar(nombre):
    return f"""
        <h1>Hola {nombre}</h1>
```

```
        <ul>
            <li><a href="/">Volver al inicio</a></li>
            <li><a href="/saludar/charly">Saludar a Charly</a></li>
            <li><a href="/saludar/Octavia">Saludar a Octavia</a></li>
        </ul>
    """

if __name__ == "__main__":
    app.run(port=8080, debug=True)
```

Si probamos a entrar en la aplicación y pinchamos en los enlaces veremos los mensajes de saludo a Charly y Octavia y también la página inicial donde se da la bienvenida.

7.2.1. Tipos de conversores disponibles

Cuando usamos las rutas dinámicas, podemos utilizar unos elementos que llamamos conversores que se encargan de convertir los datos que se pasan en la URL a un tipo de dato específico. Si se indica un dato y el valor del parámetro no se puede parsear a dicho dato, entonces dará un error indicándonos que no encuentra dicha ruta en el servidor.

A continuación tenemos la tabla con los conversores disponibles:

Convertidor	Descripción	Ejemplo
string	Acepta cualquier texto sin barras (por defecto)	/usuario/<nombre> o /usuario/<string:nombre>
int	Acepta enteros positivos	/post/<int:id>
float	Acepta números de punto flotante	/precio/<float:valor>
path	Como string pero acepta barras	/archivo/<path:ruta>
uuid	Acepta UUID strings	/user/<uuid:id>

Vamos a utilizar el conversor de **int** para crear una ruta que pueda sumar 2 números y nos muestre el resultado.

```
from flask import Flask

app = Flask(__name__)

@app.route("/")
def inicio():
    return """
        <h1>Bienvenido a la aplicación de los saludos</h1>
        <ul>
            <li><a href="/saludar/charly">Saludar a Charly</a></li>
            <li><a href="/saludar/Octavia">Saludar a Octavia</a></li>
            <li><a href="/sumar/1/2">1 + 2</a></li>
            <li><a href="/sumar/2/4">2 + 4</a></li>
```

```
        </ul>
    """

@app.route("/saludar/<nombre>")
def saludar(nombre):
    return f"""
        <h1>Hola {nombre}</h1>
        <ul>
            <li><a href="/">Volver al inicio</a></li>
            <li><a href="/saludar/charly">Saludar a Charly</a></li>
            <li><a href="/saludar/Octavia">Saludar a Octavia</a></li>
        </ul>
    """

@app.route("/sumar/<int:n1>/<int:n2>")
def sumar(n1, n2):
    return f"""
        <h1>La suma de {n1} y {n2} es {n1 + n2}</h1>
        <ul>
            <li><a href="/">Volver al inicio</a></li>
            <li><a href="/sumar/1/2">1 + 2</a></li>
            <li><a href="/sumar/2/4">2 + 4</a></li>
        </ul>
    """

if __name__ == "__main__":
    app.run(port=8080, debug=True)
```

7.3. Peticiones y respuestas

Hasta ahora no hemos visto como diferenciar una petición de otra, por ejemplo, una peticion GET de una petición POST. Para indicar las funciones que tienen que ejecutarse según el tipo de petición, lo que vamos a hacer ahora es añadir en el decorador de las rutas, además de la propia ruta, un parámetro **methods** con una lista de métodos. Cuando se reciba una petición a la ruta y cuyo método coincida con alguno de los que están definidos en el parámetro entonces se ejecutará la función decorada.

Los métodos disponibles y más usados que podemos utilizar son:

- GET
- POST
- PUT
- PATCH
- DELETE

Vamos a empezar por añadir un método **GET** en una ruta /**usuarios** que nos tiene que devolver en formato JSON la lista de usuarios de la aplicación.

Para poder devolver una respuesta en formato JSON a la petición recibida, tenemos que utilizar la función **jsonify** de flask, la cual recibe como parámetro los datos que queremos retornar.

```
from flask import Flask, jsonify

app = Flask(__name__)

usuarios = [
    {"id": 1, "nombre": "Charly Falco", "email": "cfalco@gmail.com" },
    {"id": 2, "nombre": "Octavia Blake", "email": "oblake@gmail.com" },
    {"id": 3, "nombre": "Sara Molina", "email": "sara.molina@gmail.com" },
]

@app.route("/usuarios", methods=["GET"])
def get_usuarios():
    print("Recibida petición para obtener los usuarios")
    return jsonify({ "usuarios": usuarios })


if __name__ == "__main__":
    app.run(debug=True, port=8080)
```

Por ejemplo, ahora podríamos querer retornar un solo usuario, por lo que para ello, podemos poner en la ruta el identificador del usuario a devolver. Para ello, usaremos una ruta dinámica con el parámetro **usuario_id**, el cual vamos a utilizar para buscar entre la lista de usuarios el que tenga dicho identificador.

Para realizar la búsqueda hemos utilizado una expresión generadora que nos devuelve el usuario si lo encuentra, y en caso que no lo haga entonces devuelve None.

```
from flask import Flask, jsonify

app = Flask(__name__)

usuarios = [
    {"id": 1, "nombre": "Charly Falco", "email": "cfalco@gmail.com" },
    {"id": 2, "nombre": "Octavia Blake", "email": "oblake@gmail.com" },
    {"id": 3, "nombre": "Sara Molina", "email": "sara.molina@gmail.com" },
]

@app.route("/usuarios", methods=["GET"])
def get_usuarios():
    print("Recibida petición para obtener los usuarios")
    return jsonify({ "usuarios": usuarios })

@app.route("/usuarios/<int:usuario_id>", methods=["GET"])
def get_usuario_by_id(usuario_id):
    print(f"Recibida petición para obtener el usuario con id {usuario_id}")
    usuario = next((usuario for usuario in usuarios if usuario["id"] == usuario_id), None)
    if not usuario:
        return jsonify({ "error": f"Usuario con id {usuario_id} no encontrado" })
    return jsonify(usuario)


if __name__ == "__main__":
    app.run(port=8080, debug=True)
```

Ahora si probamos desde el propio navegador a entrar en las siguientes rutas, veremos que obtenemos la respuesta esperada:

- http://localhost:8080/usuarios/2: los datos de Octavia
- http://localhost:8080/usuarios/10: usuario no encontrado

Ahora mismo, para ambas rutas, tanto la que funciona como la que no, estamos devolviendo un código de estado en la respuesta de **200**, pero al no encontrar un usuario, o cualquier otro elemento que se esté buscando, deberíamos de devolver otro tipo de código, uno de error, el **404 Not found**. Esto lo conseguimos, devolviendo junto a la respuesta el código, como una tupla de 2 valores, solo que no ponemos los paréntesis que la envuelven por ser redundantes según Python.

Flask devuelve respuestas con el código de estado el 200 Ok, en caso de que no se indique otro código diferente.

```
from flask import Flask, jsonify
```

```python
app = Flask(__name__)

usuarios = [
    {"id": 1, "nombre": "Charly Falco", "email": "cfalco@gmail.com" },
    {"id": 2, "nombre": "Octavia Blake", "email": "oblake@gmail.com" },
    {"id": 3, "nombre": "Sara Molina", "email": "sara.molina@gmail.com" },
]

@app.route("/usuarios", methods=["GET"])
def get_usuarios():
    print("Recibida petición para obtener los usuarios")
    return jsonify({ "usuarios": usuarios })

@app.route("/usuarios/<int:usuario_id>", methods=["GET"])
def get_usuario_by_id(usuario_id):
    print(f"Recibida petición para obtener el usuario con id {usuario_id}")
    usuario = next((usuario for usuario in usuarios if usuario["id"] == usuario_id), None)
    if not usuario:
        return jsonify({ "error": f"Usuario con id {usuario_id} no encontrado" }), 404
    return jsonify(usuario)


if __name__ == "__main__":
    app.run(port=8080, debug=True)
```

Además de los datos y el código de estado en la respuesta, hay un tercer valor en la tupla, que son las **cabeceras**, las cuales se añaden en un diccionario de Python.

Python permite retornar 3 tipos de tuplas para definir las respuestas:

- (datos, código de estado)
- (datos, código de estado, cabeceras)
- (datos, cabeceras)

```python
from flask import Flask, jsonify

app = Flask(__name__)

usuarios = [
    {"id": 1, "nombre": "Charly Falco", "email": "cfalco@gmail.com" },
    {"id": 2, "nombre": "Octavia Blake", "email": "oblake@gmail.com" },
    {"id": 3, "nombre": "Sara Molina", "email": "sara.molina@gmail.com" },
]

@app.route("/usuarios", methods=["GET"])
def get_usuarios():
    print("Recibida petición para obtener los usuarios")
    return jsonify({ "usuarios": usuarios })

@app.route("/usuarios/<int:usuario_id>", methods=["GET"])
```

```
def get_usuario_by_id(usuario_id):
    print(f"Recibida petición para obtener el usuario con id {usuario_id}")
    usuario = next((usuario for usuario in usuarios if usuario["id"] == usuario_id), None)
    headers = { "x-mi-cabecera": "Una cabecera personalizada" }
    if not usuario:
        return jsonify({ "error": f"Usuario con id {usuario_id} no encontrado" }), 404, headers
    return jsonify(usuario), 200, headers

if __name__ == "__main__":
    app.run(port=8080, debug=True)
```

Tenemos otra forma de crear estas respuestas que nos da un poco más de flexibilidad, ya que con lo que acabamos de ver, si yo quiero devolver unas cabeceras, y el código de estado de la respuesta es un código 200, estoy obligado a poner **200** o **None** en la segunda posición de la tupla para poder pasarle después las cabeceras. Ahora vamos a cambiar el código de la primera ruta para utilizar **make_response** que hay que importar de flask.

Con **make_response** creamos un objeto **Response** el cual podemos ir configurando como queramos, aquí al llamar a la función le podemos pasar los datos de la respuesta, y luego podemos agregarle:

- **status_code**: el código de estado de la respuesta. También se le puede pasar como segundo parámetro a la función **make_response**.
- **headers**: las cabeceras de la respuesta.
- **set_cookie**: es una función que asigna una cookie a la respuesta y recibe el nombre de la cookie y el valor como parámetros.
- **delete_cookie**: es una función que elimina de la respuesta la cookie que se le pasa como parámetro.
- **render_template**: se le puede pasar como parámetro de **make_response** esta función con la ruta a una plantilla de HTML para devolverla.

```
from flask import Flask, jsonify, make_response

app = Flask(__name__)

usuarios = [
    {"id": 1, "nombre": "Charly Falco", "email": "cfalco@gmail.com" },
    {"id": 2, "nombre": "Octavia Blake", "email": "oblake@gmail.com" },
    {"id": 3, "nombre": "Sara Molina", "email": "sara.molina@gmail.com" },
]

@app.route("/usuarios", methods=["GET"])
def get_usuarios():
    print("Recibida petición para obtener los usuarios")
    response = make_response(jsonify({ "usuarios": usuarios }))
    response.headers = { "x-mi-cabecera": "Una cabecera personalizada" }
    response.status_code = 200
    return response
```

```python
@app.route("/usuarios/<int:usuario_id>", methods=["GET"])
def get_usuario_by_id(usuario_id):
    print(f"Recibida petición para obtener el usuario con id {usuario_id}")
    usuario = next((usuario for usuario in usuarios if usuario["id"] == usuario_id), None)
    headers = { "x-mi-cabecera": "Una cabecera personalizada" }
    if not usuario:
        return jsonify({ "error": f"Usuario con id {usuario_id} no encontrado" }), 404, headers
    return jsonify(usuario), None, headers

if __name__ == "__main__":
    app.run(port=8080, debug=True)
```

7.3.1. Objeto request

Flask nos proporciona un objeto **request** con el cual podemos obtener **los datos que van en el cuerpo de la petición** (normalmente una petición POST, PUT o PATCH), y también acceder a los **parámetros de query**.

Si el cuerpo de la petición es un JSON, entonces tenemos que utilizar la función **get_json()** del objeto **request** para obtener los datos. Estos datos nos los devuelve como un diccionario. Si no nos envian los datos correctamente, puede que obtengamos un None en lugar de los datos esperados, por lo que hay que asegurarse que nos envían la cabecera **Content-Type: application/json**.

```python
import random

from flask import Flask, jsonify, make_response, request

app = Flask(__name__)

# ...

@app.route("/usuarios", methods=["POST"])
def post_usuario():
    datos_usuario = request.get_json()
    print(f"Petición recibida para crear un nuevo usuario con los datos: {datos_usuario}")

    # Código para crear el usuario

    datos_usuario.update({"id": random.randint(0, 1000)})
    return jsonify(datos_usuario), 201


if __name__ == "__main__":
    app.run(port=8080, debug=True)
```

Otra forma de recibir los datos puede ser en formato **form-data** o **x-www-form-urlencoded**, entonces ya no usaríamos la función anterior, sino que ahora tendríamos que utilizar la propiedad **request.form** para acceder a ellos. En este caso no obtenemos un diccionario, sino un objeto de tipo **InmutableMultiDict**, que es parecido a un diccionario solo que no podemos modificar como tal, ya que es inmutable.

Para acceder a los datos del formulario, esta vez tenemos que utilizar el método **get(campo)**.

```
import random

from flask import Flask, jsonify, make_response, request

app = Flask(__name__)

# ...

@app.route("/usuarios", methods=["POST"])
def post_usuario():
    # datos_usuario = request.get_json()
    datos_usuario = {**request.form}
    print(f"Petición recibida para crear un nuevo usuario con los datos: {datos_usuario}")

    # Código para crear el usuario

    datos_usuario.update({"id": random.randint(0, 1000)})
    return jsonify(datos_usuario), 201


if __name__ == "__main__":
    app.run(port=8080, debug=True)
```

Por otro lado, si nos envían **parámetros de query**, podemos acceder a ellos con **request.args**, teniendo en cuenta que funciona igual que **request.form**, por tanto obtenemos un **InmutableMultiDict** y accedemos a los parámetros con el método **get(param)**.

Vamos a modificar la petición **GET /usuarios** para poder filtrar los usuario por ciudad con un parámetro de query.

```
import random

from flask import Flask, jsonify, make_response, request

app = Flask(__name__)

@app.route("/usuarios", methods=["GET"])
def get_usuarios():
    args = {**request.args}
    if args:
        print(f"Petición recibida para obtener los usuarios filtrados por: {args}")
    else:
        print(f"Petición recibida para obtener los usuarios")
    # return jsonify({ "usuarios": usuarios })
    usuarios_filtrados = usuarios if not args else [usuario for usuario in usuarios if usuario["ciudad"] == request.args
.get("ciudad")]
    response = make_response(jsonify({ "usuarios": usuarios_filtrados }))
    response.headers = { "x-mi-cabecera": "Una cabecera personalizada" }
    response.status_code = 200
    return response

# ...
```

```
if __name__ == "__main__":
    app.run(port=8080, debug=True)
```

7.4. Gestión de errores

Hemos visto que podemos devolver respuestas de error con Flask, pero esas respuestas al final estarán duplicadas por todas las rutas de nuestra API. Flask nos permite centralizar la gestión de los errores comunes (como el 404, 400 o 500) utilizando los manejadores de errores **@app.errorhandler** que como podemos ver es un decorador.

Este decorador recibe como parámetro un código de estado y decora la función que se va a encargar de devolver el error. El error se recibe como parámetro y tiene las siguientes propiedades:

- **name**: el nombre del error.
- **description**: el mensaje de error.
- **code**: el código de error.

Para lanzar un error y que lo gestione este decorador, tenemos que utilizar **abort(codigo, description="mensaje de error")** que se importa desde flask. Internamente esta función lanza las excepciones que nos proporciona **werkzeug.exceptions**.

```
import random

from flask import Flask, jsonify, make_response, request, abort

app = Flask(__name__)

usuarios = [
    {"id": 1, "nombre": "Charly Falco", "email": "cfalco@gmail.com", "ciudad": "madrid" },
    {"id": 2, "nombre": "Octavia Blake", "email": "oblake@gmail.com", "ciudad": "valencia" },
    {"id": 3, "nombre": "Sara Molina", "email": "sara.molina@gmail.com", "ciudad": "madrid" },
]

@app.errorhandler(404)
def gestionar_404(error):
    return jsonify({ "error": error.description }), 404

# ...

@app.route("/usuarios/<int:usuario_id>", methods=["GET"])
def get_usuario_by_id(usuario_id):
    print(f"Petición recibida para obtener el usuario con id {usuario_id}")
    usuario = next((usuario for usuario in usuarios if usuario["id"] == usuario_id), None)
    headers = { "x-mi-cabecera": "Una cabecera personalizada" }

    if not usuario:
        abort(404, description=f"Usuario con id {usuario_id} no encontrado")

    return jsonify(usuario), headers

# ...


if __name__ == "__main__":
```

```
    app.run(port=8080, debug=True)
```

Como decía antes, podemos lanzar excepciones de **werkzeug.exceptions** en lugar de utilizar la función de **abort**. De aquí podemos sacar algunos errores como:

Excepción	Código	Descripción
BadRequest	400	Petición incorrecta
Unauthorized	401	No autorizado
Forbidden	403	Prohibido
NotFound	404	Recurso no encontrado
InternalServerError	500	Error interno del servidor

Por tanto, el mismo caso que antes, pero con esta otra forma de lanzar las excepciones quedaría de la siguiente forma.

```
import random

from flask import Flask, jsonify, make_response, request, abort
from werkzeug.exceptions import NotFound

app = Flask(__name__)

usuarios = [
    {"id": 1, "nombre": "Charly Falco", "email": "cfalco@gmail.com", "ciudad": "madrid" },
    {"id": 2, "nombre": "Octavia Blake", "email": "oblake@gmail.com", "ciudad": "valencia" },
    {"id": 3, "nombre": "Sara Molina", "email": "sara.molina@gmail.com", "ciudad": "madrid" },
]

@app.errorhandler(404)
def gestionar_404(error):
    return jsonify({ "error": error.description }), 404

# ...

@app.route("/usuarios/<int:usuario_id>", methods=["GET"])
def get_usuario_by_id(usuario_id):
    print(f"Petición recibida para obtener el usuario con id {usuario_id}")
    usuario = next((usuario for usuario in usuarios if usuario["id"] == usuario_id), None)
    headers = { "x-mi-cabecera": "Una cabecera personalizada" }

    if not usuario:
        raise NotFound(f"Usuario con id {usuario_id} no encontrado")

    return jsonify(usuario), headers

# ...


if __name__ == "__main__":
```

```
    app.run(port=8080, debug=True)
```

Esto nos permite centralizar la gestion de errores comunes desde un único sitio, de tal forma que como deciamos antes, no hay que retornar en todas nuestras rutas las respuestas de error, sino que ahora esa respuesta se da desde estos manejadores.

Todas estas excepciones se heredan de **HTTPException**, por tanto, podríamos gestionar todas ellas desde un único decorador pasándole esa excepción como parámetro y utilizando los datos que vienen en el objeto de error para configurar nuestra respuesta.

```
import random

from flask import Flask, jsonify, make_response, request, abort
from werkzeug.exceptions import NotFound, HTTPException

app = Flask(__name__)

usuarios = [
    {"id": 1, "nombre": "Charly Falco", "email": "cfalco@gmail.com", "ciudad": "madrid" },
    {"id": 2, "nombre": "Octavia Blake", "email": "oblake@gmail.com", "ciudad": "valencia" },
    {"id": 3, "nombre": "Sara Molina", "email": "sara.molina@gmail.com", "ciudad": "madrid" },
]

@app.errorhandler(HTTPException)
def gestionar_errores_http(error):
    return jsonify({ "error": error.name, "message": error.description }), error.code

# ...

@app.route("/usuarios/<int:usuario_id>", methods=["GET"])
def get_usuario_by_id(usuario_id):
    print(f"Petición recibida para obtener el usuario con id {usuario_id}")
    usuario = next((usuario for usuario in usuarios if usuario["id"] == usuario_id), None)
    headers = { "x-mi-cabecera": "Una cabecera personalizada" }

    if not usuario:
        raise NotFound(f"Usuario con id {usuario_id} no encontrado")

    return jsonify(usuario), headers

# ...


if __name__ == "__main__":
    app.run(port=8080, debug=True)
```

7.5. Middlewares

Los **middlewares** son funciones que se ejecutan antes o después de ejecutar la función que se ejecuta con la llegada de las peticiones al servidor.

Los middlewares que nos da flask son:

- **before_request**: se ejecuta antes de ejecutar la función que se está decorando.
- **after_request**: se ejecuta después de ejecutar la función que se está decorando.
- **teardown_request**: se ejecuta después de ejecutar la función que se está decorando, incluso si ocurre un error.

Vamos a empezar con un código similar al de los anteriores capítulos:

```
import random

from flask import Flask, jsonify, request

app = Flask(__name__)

usuarios = [
    {"id": 1, "nombre": "Charly Falco", "email": "cfalco@gmail.com", "password": "1234", "ciudad": "madrid" },
    {"id": 2, "nombre": "Octavia Blake", "email": "oblake@gmail.com", "password": "1234", "ciudad": "valencia" },
    {"id": 3, "nombre": "Sara Molina", "email": "sara.molina@gmail.com", "password": "1234", "ciudad": "madrid" },
]

@app.errorhandler(401)
def gestionar_401(error):
    return jsonify({ "error": error.name, "message": error.description }), 401


@app.route("/usuarios", methods=["POST"])
def post_usuario():
    datos_nuevo_usuario = request.get_json()
    print(f"Petición recibida para crear un nuevo usuario con los datos: {datos_nuevo_usuario}")

    # Código para crear el usuario

    datos_nuevo_usuario.update({"id": random.randint(0, 1000)})
    return jsonify(datos_nuevo_usuario), 201


if __name__ == "__main__":
    app.run(port=8080, debug=True)
```

Lo primero de todo es que vamos a añadir un endpoint para realizar el **login**, con la ruta **POST /login** en la que vamos a recoger las credenciales de un usuario (email y password) y vamos a devolver un token de autenticación.

Lo primero que hacemos es utilizar el **request.get_json()** para sacar las credenciales, y vamos a buscar entre los usuarios registrados si hay alguno que tenga dichas credenciales. En caso de no haberlo, vamos a devolver un error 401.

```
import random
```

```
from flask import Flask, jsonify, request, abort

app = Flask(__name__)

usuarios = [
    {"id": 1, "nombre": "Charly Falco", "email": "cfalco@gmail.com", "password": "1234", "ciudad": "madrid" },
    {"id": 2, "nombre": "Octavia Blake", "email": "oblake@gmail.com", "password": "1234", "ciudad": "valencia" },
    {"id": 3, "nombre": "Sara Molina", "email": "sara.molina@gmail.com", "password": "1234", "ciudad": "madrid" },
]

@app.errorhandler(401)
def gestionar_401(error):
    return jsonify({ "error": error.name, "message": error.description }), 401

@app.route("/login", methods=["POST"])
def login():
    datos = request.get_json()

    usuario_registrado = next((usuario for usuario in usuarios if usuario["email"] == datos["email"] and usuario[
"password"] == datos["password"]), None)

    if not usuario_registrado:
        abort(401, description="Las credenciales no son válidas")


@app.route("/usuarios", methods=["POST"])
def post_usuario():
    datos_nuevo_usuario = request.get_json()
    print(f"Petición recibida para crear un nuevo usuario con los datos: {datos_nuevo_usuario}")

    # Código para crear el usuario

    datos_nuevo_usuario.update({"id": random.randint(0, 1000)})
    return jsonify(datos_nuevo_usuario), 201


if __name__ == "__main__":
    app.run(port=8080, debug=True)
```

Para poder generar el token, necesitamos instalar el paquete **pyjwt** con el siguiente comando:

```
$ pip install pyjwt
```

Ahora, ya podemos generar el token. Para ello, vamos a importar **jwt** y vamos a utilizar el método **encode** al cual le vamos a pasar un diccionario con los datos que queramos codificar en el token.

Los datos que se codifican en el payload del token se pueden leer si tenemos el token a mano y podemos decodificarlo de base 64.

El segundo parámetro es una clave secreta, que luego vamos a utilizar para verificar en las siguientes peticiones si el token es válido o no. Este secreto lo vamos a guardar en una variable **config** de la aplicación, donde podemos poner variables de configuración para nuestra aplicación.

Y por último, un tercer parámetro con el algoritmo que se va a utilizar para generar el JWT.

```
import random
```

```python
from flask import Flask, jsonify, request, abort
import jwt

app = Flask(__name__)

app.config["SECRET"] = "esto_es_un_secreto"

usuarios = [
    {"id": 1, "nombre": "Charly Falco", "email": "cfalco@gmail.com", "password": "1234", "ciudad": "madrid" },
    {"id": 2, "nombre": "Octavia Blake", "email": "oblake@gmail.com", "password": "1234", "ciudad": "valencia" },
    {"id": 3, "nombre": "Sara Molina", "email": "sara.molina@gmail.com", "password": "1234", "ciudad": "madrid" },
]

@app.errorhandler(401)
def gestionar_401(error):
    return jsonify({ "error": error.name, "message": error.description }), 401

@app.route("/login", methods=["POST"])
def login():
    datos = request.get_json()

    usuario_registrado = next((usuario for usuario in usuarios if usuario["email"] == datos["email"] and usuario[
"password"] == datos["password"]), None)

    if not usuario_registrado:
        abort(401, description="Las credenciales no son válidas")

    token = jwt.encode({
        "usuario": usuario_registrado["nombre"],
    }, app.config["SECRET"], algorithm="HS256")

    return jsonify({ "token": token })

@app.route("/usuarios", methods=["POST"])
def post_usuario():
    datos_nuevo_usuario = request.get_json()
    print(f"Petición recibida para crear un nuevo usuario con los datos: {datos_nuevo_usuario}")

    # Código para crear el usuario

    datos_nuevo_usuario.update({"id": random.randint(0, 1000)})
    return jsonify(datos_nuevo_usuario), 201


if __name__ == "__main__":
    app.run(port=8080, debug=True)
```

Con esto, ya podemos generar un token, cuando el usuario se autentique correctamente.

Ahora vamos a añadir un middleware que queremos ejecutar justo antes de cada petición para validar si el token que nos envían es correcto o no.

Para ello, vamos a utilizar **@app.before_request** para decorar una función que se va a ejecutar antes de cada petición.

Para no verificar el token en todas las peticiones que nos lleguen, vamos a poner una lista con los paths de las peticiones que no necesita validación, como por ejemplo, la de el propio **login**.

Después vamos a obtener el token para comprobar que nos llega, y en caso de que no lo haga devolveremos un error. Para ello tenemos que acceder a las cabeceras, más concretamente a la cabecera de **Authorization**.

```python
import random

from flask import Flask, jsonify, request, abort
import jwt

app = Flask(__name__)

app.config["SECRET"] = "esto_es_un_secreto"

usuarios = [
    {"id": 1, "nombre": "Charly Falco", "email": "cfalco@gmail.com", "password": "1234", "ciudad": "madrid" },
    {"id": 2, "nombre": "Octavia Blake", "email": "oblake@gmail.com", "password": "1234", "ciudad": "valencia" },
    {"id": 3, "nombre": "Sara Molina", "email": "sara.molina@gmail.com", "password": "1234", "ciudad": "madrid" },
]

@app.errorhandler(401)
def gestionar_401(error):
    return jsonify({ "error": error.name, "message": error.description }), 401


@app.before_request
def verificar_token():
    rutas_sin_verificacion = ["/login"]

    if request.path in rutas_sin_verificacion:
        return

    token = None

    if "Authorization" in request.headers:
        bearer, token = request.headers["Authorization"].split(" ")

    if not token:
        abort(401, description="El token es obligatorio")


@app.route("/login", methods=["POST"])
def login():
    datos = request.get_json()

    usuario_registrado = next((usuario for usuario in usuarios if usuario["email"] == datos["email"] and usuario[
"password"] == datos["password"]), None)

    if not usuario_registrado:
        abort(401, description="Las credenciales no son válidas")

    token = jwt.encode({
        "usuario": usuario_registrado["nombre"],
    }, app.config["SECRET"], algorithm="HS256")

    return jsonify({ "token": token })

@app.route("/usuarios", methods=["POST"])
def post_usuario():
    datos_nuevo_usuario = request.get_json()
    print(f"Petición recibida para crear un nuevo usuario con los datos: {datos_nuevo_usuario}")

    # Código para crear el usuario

    datos_nuevo_usuario.update({"id": random.randint(0, 1000)})
    return jsonify(datos_nuevo_usuario), 201


if __name__ == "__main__":
    app.run(port=8080, debug=True)
```

Ahora ya podemos verificar si el token es correcto, para lo que vamos a utilizar la función **jwt.decode()** a la que le vamos a pasar 3 parámetros:

- El token que hemos recibido del usuario.
- El secreto que teníamos guardado en la configuración y con el que se generó el token.
- El algoritmo que se había utilizado para generar el token.

A partir de estos datos, se puede verificar que el token es correcto, si es así vamos a obtener el **payload** que hay dentro del token. Pero si el token es incorrecto pueden lanzarse varias excepciones:

- **jwt.ExpiredSignatureError**: el token está expirado y ya no es válido.
- **jwt.InvalidTokenError**: el token no es válido.

Y ahora podríamos querer pasarle los datos que van incrustrados en el token a la aplicación para que puedan ser utilizados en las peticiones. Para ello, podemos utilizar el objeto **g** que flask nos ofrece para guardar datos en él. Este objeto lo vamos a poder leer en las funciones que hemos asociado a las rutas.

```
import random

from flask import Flask, jsonify, request, abort, g
import jwt

app = Flask(__name__)

app.config["SECRET"] = "esto_es_un_secreto"

usuarios = [
    {"id": 1, "nombre": "Charly Falco", "email": "cfalco@gmail.com", "password": "1234", "ciudad": "madrid" },
    {"id": 2, "nombre": "Octavia Blake", "email": "oblake@gmail.com", "password": "1234", "ciudad": "valencia" },
    {"id": 3, "nombre": "Sara Molina", "email": "sara.molina@gmail.com", "password": "1234", "ciudad": "madrid" },
]

@app.errorhandler(401)
def gestionar_401(error):
    return jsonify({ "error": error.name, "message": error.description }), 401


@app.before_request
def verificar_token():
    rutas_sin_verificacion = ["/login"]

    if request.path in rutas_sin_verificacion:
        return

    token = None

    if "Authorization" in request.headers:
        bearer, token = request.headers["Authorization"].split(" ")

    if not token:
        abort(401, description="El token es obligatorio")

    try:
        datos = jwt.decode(token, app.config["SECRET"], algorithms=["HS256"])
        g.usuario = datos["usuario"]
    except jwt.ExpiredSignatureError:
```

```
        abort(401, description="El token ha expirado")
    except jwt.InvalidTokenError:
        abort(401, description="El token no es válido")


@app.route("/login", methods=["POST"])
def login():
    datos = request.get_json()

    usuario_registrado = next((usuario for usuario in usuarios if usuario["email"] == datos["email"] and usuario[
"password"] == datos["password"]), None)

    if not usuario_registrado:
        abort(401, description="Las credenciales no son válidas")

    token = jwt.encode({
        "usuario": usuario_registrado["nombre"],
    }, app.config["SECRET"], algorithm="HS256")

    return jsonify({ "token": token })

@app.route("/usuarios", methods=["POST"])
def post_usuario():
    datos_nuevo_usuario = request.get_json()
    print(f"Petición recibida para crear un nuevo usuario con los datos: {datos_nuevo_usuario}")

    # Código para crear el usuario

    datos_nuevo_usuario.update({"id": random.randint(0, 1000)})
    return jsonify(datos_nuevo_usuario), 201


if __name__ == "__main__":
    app.run(port=8080, debug=True)
```

Finalmente vamos a leer el la información del usuario guardada en la variable global **g** dentro de la función de crear un nuevo usuario.

```
import random

from flask import Flask, jsonify, request, abort, g
import jwt

app = Flask(__name__)

app.config["SECRET"] = "esto_es_un_secreto"

usuarios = [
    {"id": 1, "nombre": "Charly Falco", "email": "cfalco@gmail.com", "password": "1234", "ciudad": "madrid" },
    {"id": 2, "nombre": "Octavia Blake", "email": "oblake@gmail.com", "password": "1234", "ciudad": "valencia" },
    {"id": 3, "nombre": "Sara Molina", "email": "sara.molina@gmail.com", "password": "1234", "ciudad": "madrid" },
]

@app.errorhandler(401)
def gestionar_401(error):
    return jsonify({ "error": error.name, "message": error.description }), 401


@app.before_request
def verificar_token():
    rutas_sin_verificacion = ["/login"]

    if request.path in rutas_sin_verificacion:
        return
```

```
    token = None

    if "Authorization" in request.headers:
        bearer, token = request.headers["Authorization"].split(" ")

    if not token:
        abort(401, description="El token es obligatorio")

    try:
        datos = jwt.decode(token, app.config["SECRET"], algorithms=["HS256"])
        g.usuario = datos["usuario"]
    except jwt.ExpiredSignatureError:
        abort(401, description="El token ha expirado")
    except jwt.InvalidTokenError:
        abort(401, description="El token no es válido")


@app.route("/login", methods=["POST"])
def login():
    datos = request.get_json()

    usuario_registrado = next((usuario for usuario in usuarios if usuario["email"] == datos["email"] and usuario[
"password"] == datos["password"]), None)

    if not usuario_registrado:
        abort(401, description="Las credenciales no son válidas")

    token = jwt.encode({
        "usuario": usuario_registrado["nombre"],
    }, app.config["SECRET"], algorithm="HS256")

    return jsonify({ "token": token })

@app.route("/usuarios", methods=["POST"])
def post_usuario():
    datos_nuevo_usuario = request.get_json()
    print(f"Petición recibida del usuario {g.usuario} para crear un nuevo usuario con los datos: {datos_nuevo_usuario}")

    # Código para crear el usuario

    datos_nuevo_usuario.update({"id": random.randint(0, 1000)})
    return jsonify(datos_nuevo_usuario), 201


if __name__ == "__main__":
    app.run(port=8080, debug=True)
```

7.5.1. Usando decoradores para la autenticación

La funcion de validación del token que hemos puesto con el middleware se está ejecutando con cada petición que llega, y puede no ser muy óptimo, por lo que vamos a utilizar un decorador que podemos poner en cada petición que necesita verificar el token y así evitar que se verifique el token en aquellas que no lo necesitan.

En este caso, vamos a partir del siguiente código.

```
import random

from flask import Flask, jsonify, request, abort
import jwt
```

```
app = Flask(__name__)

app.config["SECRET"] = "esto_es_un_secreto"

usuarios = [
    {"id": 1, "nombre": "Charly Falco", "email": "cfalco@gmail.com", "password": "1234", "ciudad": "madrid" },
    {"id": 2, "nombre": "Octavia Blake", "email": "oblake@gmail.com", "password": "1234", "ciudad": "valencia" },
    {"id": 3, "nombre": "Sara Molina", "email": "sara.molina@gmail.com", "password": "1234", "ciudad": "madrid" },
]

@app.errorhandler(401)
def gestionar_401(error):
    return jsonify({ "error": error.name, "message": error.description }), 401


@app.route("/login", methods=["POST"])
def login():
    datos = request.get_json()

    usuario_registrado = next((usuario for usuario in usuarios if usuario["email"] == datos["email"] and usuario[
"password"] == datos["password"]), None)

    if not usuario_registrado:
        abort(401, description="Las credenciales no son válidas")

    token = jwt.encode({
        "usuario": usuario_registrado["nombre"],
    }, app.config["SECRET"], algorithm="HS256")

    return jsonify({ "token": token })


@app.route("/usuarios", methods=["POST"])
def post_usuario():
    datos_nuevo_usuario = request.get_json()
    print(f"Petición recibida para crear un nuevo usuario con los datos: {datos_nuevo_usuario}")

    # Código para crear el usuario

    datos_nuevo_usuario.update({"id": random.randint(0, 1000)})
    return jsonify(datos_nuevo_usuario), 201


if __name__ == "__main__":
    app.run(port=8080, debug=True)
```

Lo primero que vamos a hacer es crearnos nuestro decorador encargado de realizar la verificación del token.

Por lo que vamos a crear una función **verificar_token** que reciba como parámetro una función. Dicha función será la que se va a ejecutar una vez que se haya verificado el token correctamente.

Dentro de esta función vamos a definir otra función que va a recibir unos args y kwargs a la que podemos llamar **wrapper** y que nos va a devolver el resultado de llamar a la función decorada **fn**. A su vez, la primera función va a retornar la función **wrapper**.

Otra cosa que vamos a aprovechar para hacer es utilizar el decorador **@wraps(fn)** justo delante del **wrapper**. Este decorador viene de **functools**, y se encarga de preservar los métadatos (nombre, docstrings...) de la función original cuando esta es decorada.

```python
import random
from functools import wraps

from flask import Flask, jsonify, request, abort
import jwt

app = Flask(__name__)

app.config["SECRET"] = "esto_es_un_secreto"

usuarios = [
    {"id": 1, "nombre": "Charly Falco", "email": "cfalco@gmail.com", "password": "1234", "ciudad": "madrid" },
    {"id": 2, "nombre": "Octavia Blake", "email": "oblake@gmail.com", "password": "1234", "ciudad": "valencia" },
    {"id": 3, "nombre": "Sara Molina", "email": "sara.molina@gmail.com", "password": "1234", "ciudad": "madrid" },
]

@app.errorhandler(401)
def gestionar_401(error):
    return jsonify({ "error": error.name, "message": error.description }), 401


def verificar_token(fn):
    @wraps(fn)
    def wrapper(*args, **kwargs):

        # Aquí va la verificación del token y llamada a fn (función decorada)

    return wrapper


@app.route("/login", methods=["POST"])
def login():
    datos = request.get_json()

    usuario_registrado = next((usuario for usuario in usuarios if usuario["email"] == datos["email"] and usuario[
"password"] == datos["password"]), None)

    if not usuario_registrado:
        abort(401, description="Las credenciales no son válidas")

    token = jwt.encode({
        "usuario": usuario_registrado["nombre"],
    }, app.config["SECRET"], algorithm="HS256")

    return jsonify({ "token": token })


@app.route("/usuarios", methods=["POST"])
def post_usuario():
    datos_nuevo_usuario = request.get_json()
    print(f"Petición recibida para crear un nuevo usuario con los datos: {datos_nuevo_usuario}")

    # Código para crear el usuario

    datos_nuevo_usuario.update({"id": random.randint(0, 1000)})
    return jsonify(datos_nuevo_usuario), 201


if __name__ == "__main__":
    app.run(port=8080, debug=True)
```

Ahora ya podemos poner la verificación del token que es igual que lo que ya vimos con el middleware. Lo que es nuevo en este caso es que vamos a llamar a la función decorada **fn**

pasándole como parámetro el usuario que hemos extraido del token junto a los args y kwargs por si luego son necesarios.

```
import random
from functools import wraps

from flask import Flask, jsonify, request, abort
import jwt

app = Flask(__name__)

app.config["SECRET"] = "esto_es_un_secreto"

usuarios = [
    {"id": 1, "nombre": "Charly Falco", "email": "cfalco@gmail.com", "password": "1234", "ciudad": "madrid" },
    {"id": 2, "nombre": "Octavia Blake", "email": "oblake@gmail.com", "password": "1234", "ciudad": "valencia" },
    {"id": 3, "nombre": "Sara Molina", "email": "sara.molina@gmail.com", "password": "1234", "ciudad": "madrid" },
]

@app.errorhandler(401)
def gestionar_401(error):
    return jsonify({ "error": error.name, "message": error.description }), 401


def verificar_token(fn):
    @wraps(fn)
    def wrapper(*args, **kwargs):
        token = None

        if "Authorization" in request.headers:
            bearer, token = request.headers["Authorization"].split(" ")

        if not token:
            abort(401, description="El token es obligatorio")

        try:
            datos = jwt.decode(token, app.config["SECRET"], algorithms=["HS256"])
            usuario = datos["usuario"]
        except jwt.ExpiredSignatureError:
            abort(401, description="El token ha expirado")
        except jwt.InvalidTokenError:
            abort(401, description="El token no es válido")

        return fn(usuario, *args, **kwargs)

    return wrapper


@app.route("/login", methods=["POST"])
def login():
    datos = request.get_json()

    usuario_registrado = next((usuario for usuario in usuarios if usuario["email"] == datos["email"] and usuario[
"password"] == datos["password"]), None)

    if not usuario_registrado:
        abort(401, description="Las credenciales no son válidas")

    token = jwt.encode({
        "usuario": usuario_registrado["nombre"],
    }, app.config["SECRET"], algorithm="HS256")

    return jsonify({ "token": token })
```

```
@app.route("/usuarios", methods=["POST"])
def post_usuario():
    datos_nuevo_usuario = request.get_json()
    print(f"Petición recibida para crear un nuevo usuario con los datos: {datos_nuevo_usuario}")

    # Código para crear el usuario

    datos_nuevo_usuario.update({"id": random.randint(0, 1000)})
    return jsonify(datos_nuevo_usuario), 201


if __name__ == "__main__":
    app.run(port=8080, debug=True)
```

Ahora que ya tenemos el decorador, solo nos falta utilizarlo. Para ello vamos a decorar la función **post_usuario** y vamos a hacer que dicha función reciba el **usuario** extraido del token que le estabamos pasando como argumento.

```
import random
from functools import wraps

from flask import Flask, jsonify, request, abort
import jwt

app = Flask(__name__)

app.config["SECRET"] = "esto_es_un_secreto"

usuarios = [
    {"id": 1, "nombre": "Charly Falco", "email": "cfalco@gmail.com", "password": "1234", "ciudad": "madrid" },
    {"id": 2, "nombre": "Octavia Blake", "email": "oblake@gmail.com", "password": "1234", "ciudad": "valencia" },
    {"id": 3, "nombre": "Sara Molina", "email": "sara.molina@gmail.com", "password": "1234", "ciudad": "madrid" },
]

@app.errorhandler(401)
def gestionar_401(error):
    return jsonify({ "error": error.name, "message": error.description }), 401


def verificar_token(fn):
    @wraps(fn)
    def wrapper(*args, **kwargs):
        token = None

        if "Authorization" in request.headers:
            bearer, token = request.headers["Authorization"].split(" ")

        if not token:
            abort(401, description="El token es obligatorio")

        try:
            datos = jwt.decode(token, app.config["SECRET"], algorithms=["HS256"])
            usuario = datos["usuario"]
        except jwt.ExpiredSignatureError:
            abort(401, description="El token ha expirado")
        except jwt.InvalidTokenError:
            abort(401, description="El token no es válido")

        return fn(usuario, *args, **kwargs)

    return wrapper
```

```
@app.route("/login", methods=["POST"])
def login():
    datos = request.get_json()

    usuario_registrado = next((usuario for usuario in usuarios if usuario["email"] == datos["email"] and usuario[
"password"] == datos["password"]), None)

    if not usuario_registrado:
        abort(401, description="Las credenciales no son válidas")

    token = jwt.encode({
        "usuario": usuario_registrado["nombre"],
    }, app.config["SECRET"], algorithm="HS256")

    return jsonify({ "token": token })


@app.route("/usuarios", methods=["POST"])
@verificar_token
def post_usuario(usuario):
    datos_nuevo_usuario = request.get_json()
    print(f"Petición recibida del usuario {usuario} para crear un nuevo usuario con los datos: {datos_nuevo_usuario}")

    # Código para crear el usuario

    datos_nuevo_usuario.update({"id": random.randint(0, 1000)})
    return jsonify(datos_nuevo_usuario), 201


if __name__ == "__main__":
    app.run(port=8080, debug=True)
```

Y con esto tendríamos el mismo ejemplo que antes, pero esta vez con un decorador.