Programación Avanzada
IIC2233 2024-2

Hernán Valdivieso - Daniela Concha - Francisca Ibarra - Lucas Van Sint Jan - Francisca Cattan

Anuncios

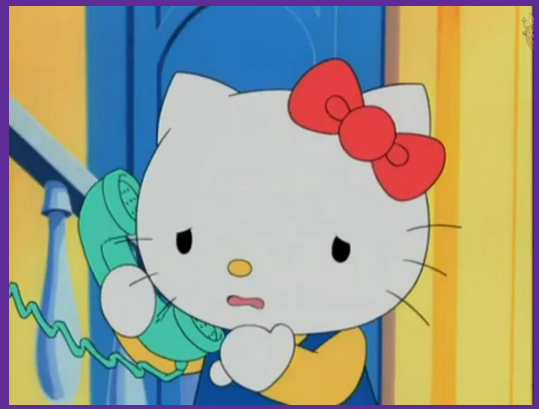
Jueves 17 de octubre 2024

1. Hoy tenemos la **tercera experiencia del curso**, enfocada en *networking*.
2. Hagan *push* de su T3 👀. Se entrega hoy y no se aceptará **ningún atraso**.

Networking 2

(...Ahora con código)

Arquitectura Cliente - Servidor y *Sockets*

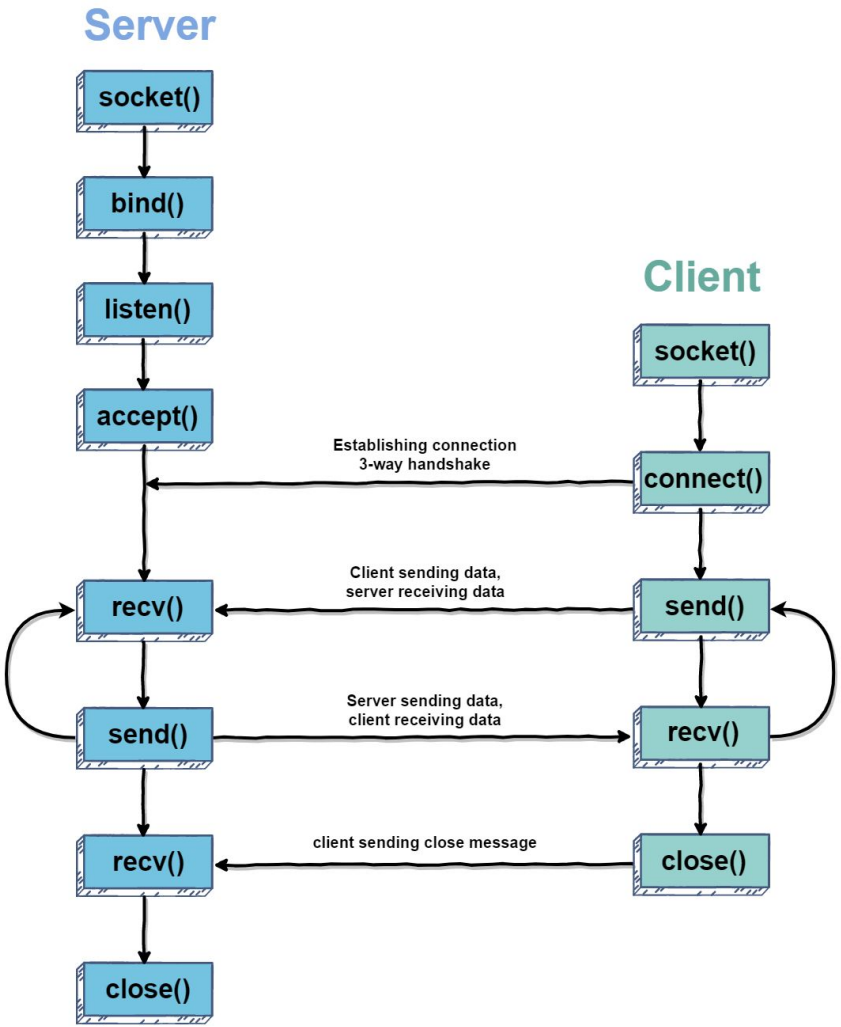


Socket

Es un **objeto del sistema operativo** que permite a un programa **transmitir y recibir datos** desde y hacia otro programa corriendo en otra máquina, o en la misma máquina pero en otro puerto.

```
import socket
mi_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

Flujo de comunicación con *sockets* entre Cliente y Servidor



Sockets: Servidor

Para crear el servidor es necesario **enlazarlo** con la dirección y el puerto deseado, y luego quedar **escuchando** clientes.

```
ServerAddress = "127.0.0.1" # También sirve poner “localhost”.
ServerPort = 9999           # Valores altos tienden a estar desocupados.

mi_socket.bind((ServerAddress, ServerPort)) # ServerAddress y ServerPort son
                                            # las variables def. previamente.

mi_socket.listen() # Habilitamos el socket para escuchar clientes.

socket_cliente, address = mi_socket.accept() # Acepta un cliente en particular.
                                             # Este método retorna cuando un
                                             # cliente en algún lugar se
                                             # conecta a este servidor.
```

Sockets: Cliente

Para conectar un cliente a un servidor debemos crear el *socket* y conectarlo al puerto y la dirección del **servidor**.

```
ServerAddress = "127.0.0.1" # Dirección del servidor anterior
ServerPort = 9999 # Puerto que el servidor anterior está usando

cliente = socket.socket()
cliente.connect((ServerAddress, ServerPort)) # Este connect es el que activa
                                             # y hace que el método ‘accept’
                                             # del servidor anterior retorne.
```

Enviando y recibiendo información

Para enviar y recibir información (***bytes***) desde el cliente al servidor (y viceversa), se utiliza el método ***send*** o ***sendall*** para enviar datos, y el método ***recv*** para recibir *bytes*.

```
# Este es un servidor que acepta una conexión, envía un mensaje
# y decodifica la respuesta del cliente
socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
socket.bind((ServerAddress, ServerPort))
socket.listen()
socket_cliente, address = socket.accept()
socket_cliente.send("Hola".encode("ascii"))
data = socket_cliente.recv(1024)
print(data.decode("ascii"))
```

Enviando y recibiendo información

Para enviar y recibir información (***bytes***) desde el cliente al servidor (y viceversa), se utiliza el método ***send*** o ***sendall*** para enviar datos, y el método ***recv*** para recibir *bytes*.

```
# Este es un cliente que solicita conectarse, recibe un mensaje
# y responde al servidor
socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
socket.connect((ServerAddress, ServerPort))
data = socket.recv(1024)
print(data.decode("ascii"))
socket.send("Hola de vuelta!".encode("ascii"))
```

Agregando *Threads* a nuestro programa

Podemos aprovechar los *threads* de Python para hacer que nuestro servidor y cliente **no se bloqueen** mientras esperan un mensaje. Para esto, podemos tener distintos *threads* encargados de distintos aspectos del programa: aceptar conexiones, escuchar y manejar los mensajes recibidos, entre otros.

En la experiencia de hoy veremos un ejemplo de esto 😎.

Diferentes modelaciones con *networking*

Ya sabemos que para establecer una **conexión** necesitamos al menos un **Servidor** y un **Cliente**. Sin embargo, esto no significa que solo podamos hacer programas con esta configuración. Podemos usar *sockets* para diseñar programas tales como:

- Un servidor con múltiples clientes
- Un cliente que se conecta a más de un servidor
- Clientes que se comunican entre sí sin un servidor intermedio (*Peer 2 Peer).*
- Y más.

Pregunta de Evaluación Escrita

Tema: Networking (Examen 2024-1)

4. ¿Cuál de las siguientes es **correcta** respecto a la modelación de Servidores y Clientes usando *sockets* en Python?

   A) Es posible tener un programa que sea Servidor y Cliente al mismo tiempo.
   B) Un Cliente puede estar conectado máximo a un Servidor al mismo tiempo.
   C) Mientras esté activo, un Servidor siempre aceptará a todos los Clientes que se conecten.
   D) Una conexión entre Servidor y Cliente solo puede ser cerrada cuando uno de los dos programas finaliza su ejecución.
   E) En caso de fallo en el Cliente, el Servidor automáticamente cierra la conexión a dicho Cliente.

Pregunta de Evaluación Escrita

Tema: Networking (Examen 2024-1)

4. ¿Cuál de las siguientes es **correcta** respecto a la modelación de Servidores y Clientes usando *sockets* en Python?

A) Es posible tener un programa que sea Servidor y Cliente al mismo tiempo.

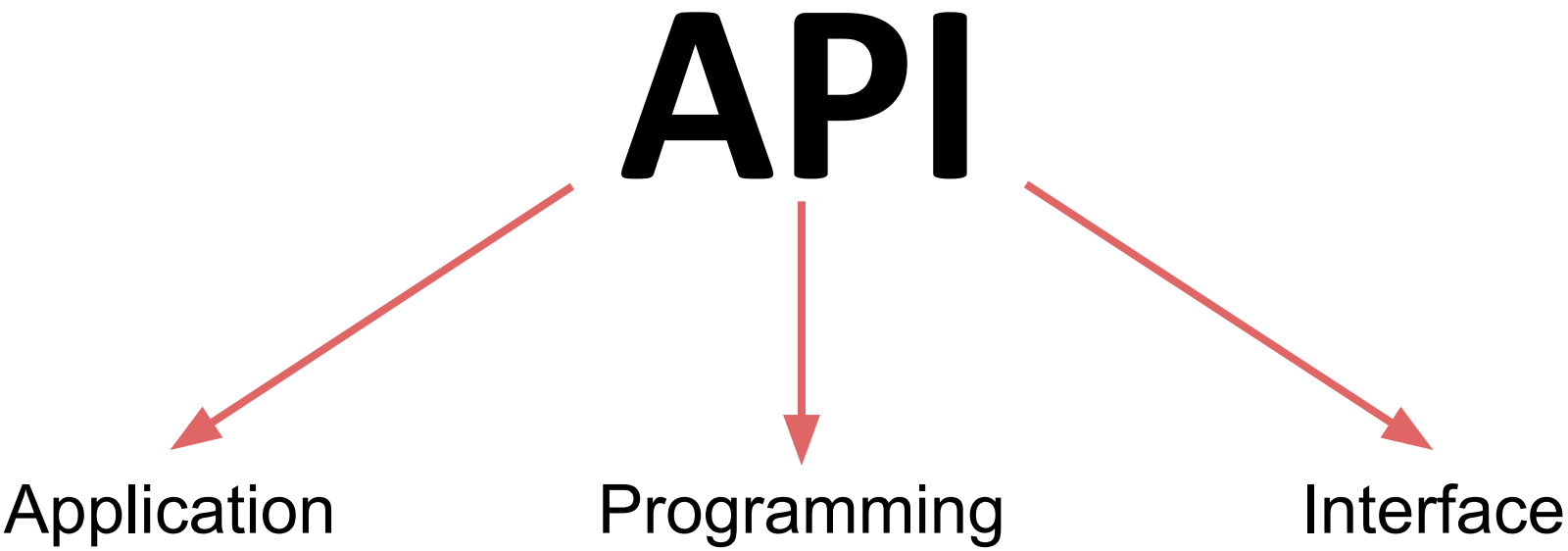
B) Un Cliente puede estar conectado máximo a un Servidor al mismo tiempo.

C) Mientras esté activo, un Servidor siempre aceptará a todos los Clientes que se conecten.

D) Una conexión entre Servidor y Cliente solo puede ser cerrada cuando uno de los dos programas finaliza su ejecución.

E) En caso de fallo en el Cliente, el Servidor automáticamente cierra la conexión a dicho Cliente.

Webservices (API)


API
Application
Programming
Interface

API

En general, API es un conjunto de funciones que son expuestas por un servicio para ser utilizadas por otros programas.

Nosotros nos enfocaremos específicamente en los servicios web o *web services.*

Por lo tanto, primero entendamos un poco cómo nos comunicamos en la web y luego retomaremos más el tema de la API.

Comunicándose a través de internet

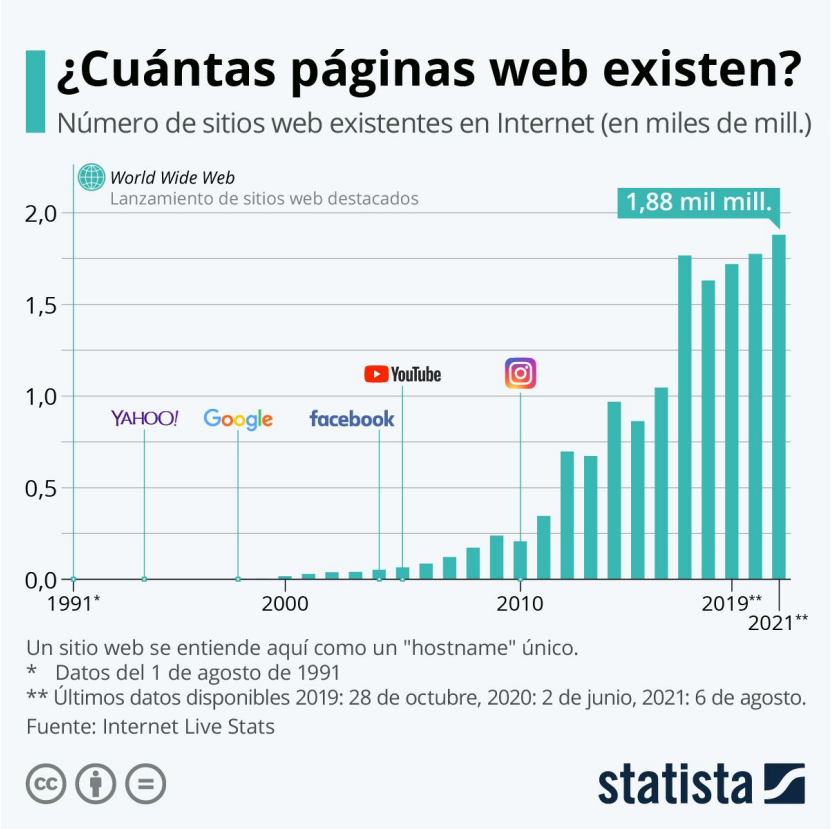
¿Cómo me conecto a un servidor?

- Para empezar, en internet hay millones de páginas, y cada página puede tener cientos de directorios o recursos.

Fuente:
Gráfico: ¿Cuántos sitios web hay en el mundo? | Statista

¿Cómo me conecto a un servidor?

- Para empezar, en internet hay millones de páginas, y cada página puede tener cientos de directorios o recursos.
- Para identificar cada página, directorio y recurso, existe el concepto de **URL**.



Fuente:
Gráfico: ¿Cuántos sitios web hay en el mundo? | Statista

URL *(Uniform Resource Locator)*

- Más de una vez hemos visto textos así:

https://es.aliexpress.com/item/1005006792016183.html?spm=a2g0o.productlist.main.1.3d1aeGRMeGRMwJ&algo_pvid=4a7a04ee-41fe-457b-8b5a-8e46d7554e61&algo_exp_id=4a7a04ee-41fe-457b-8b5a-8e46d7554e61-0&pdp_npi=4%40dis%21CLP%212457%211899%21%21%212.60%212.01%21%40210318c31717433108518167 5e3076%211200000383168394 42%21sea%21CL%21171744362%21&curPageLogUid=ZC08JbSCPYCs&utparam-url=scene%3Asearch%7Cquery_from%3A

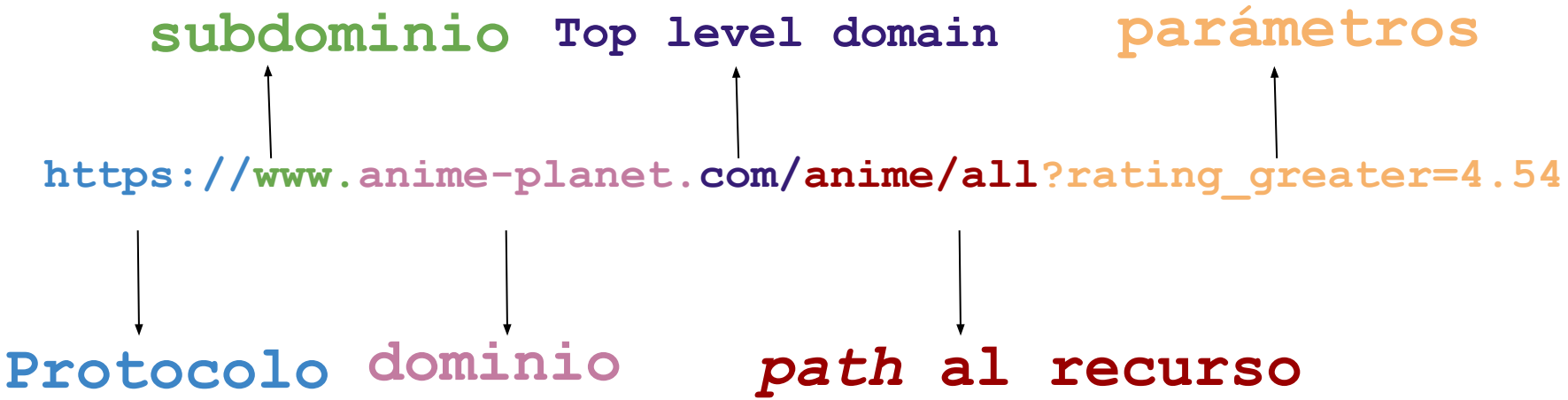
URL *(Uniform Resource Locator)*

Se definió un **formato que deben cumplir las URLs** para encontrar a lo que sea que deseen buscar en internet.

`https://www.anime-planet.com/anime/all?rating_greater=4.54`

URL *(Uniform Resource Locator)*

Se definió un **formato que deben cumplir las URLs** para encontrar a lo que sea que deseen buscar en internet.



URL *(Uniform Resource Locator)*

Todo enlace, por intimidante que se vea, lo cumple.

```
https://es.aliexpress.com/item/1005006792016183?spm=a2g0o.p
roductlist.main.1.3d1aeGRMeGRMwJ&algo_pvid=4a7a04ee-41fe-45
7b-8b5a-8e46d7554e61&algo_exp_id=4a7a04ee-41fe-457b-8b5a-8e
46d7554e61-0&pdp_npi=4%40dis%21CLP%212457%211899%21%21%212.
60%212.01%21%40210318c31717433108518 1675e3076%211200003831 6
839442%21sea%21CL%21171744362%21&curPageLogUid=ZC08JbSCPYCs
&utparam-url=scene%3Asearch%7Cquery_from%3A
```

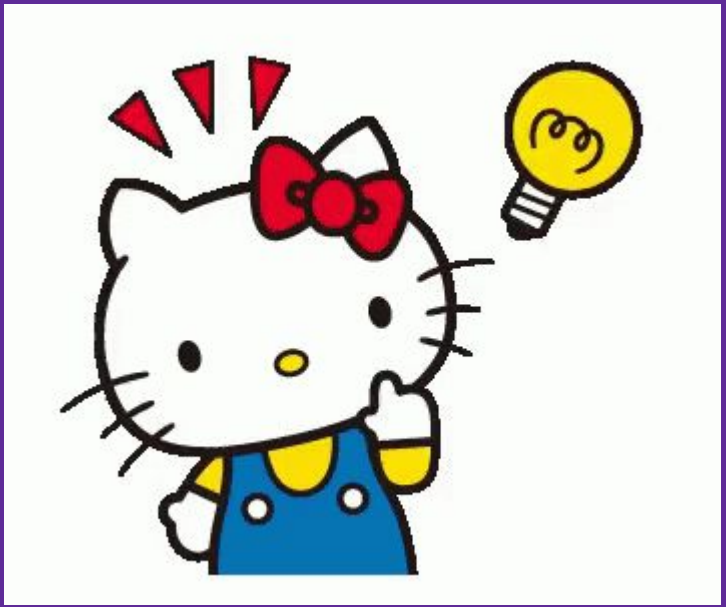
HTTP *(Hypertext Transfer Protocol)*

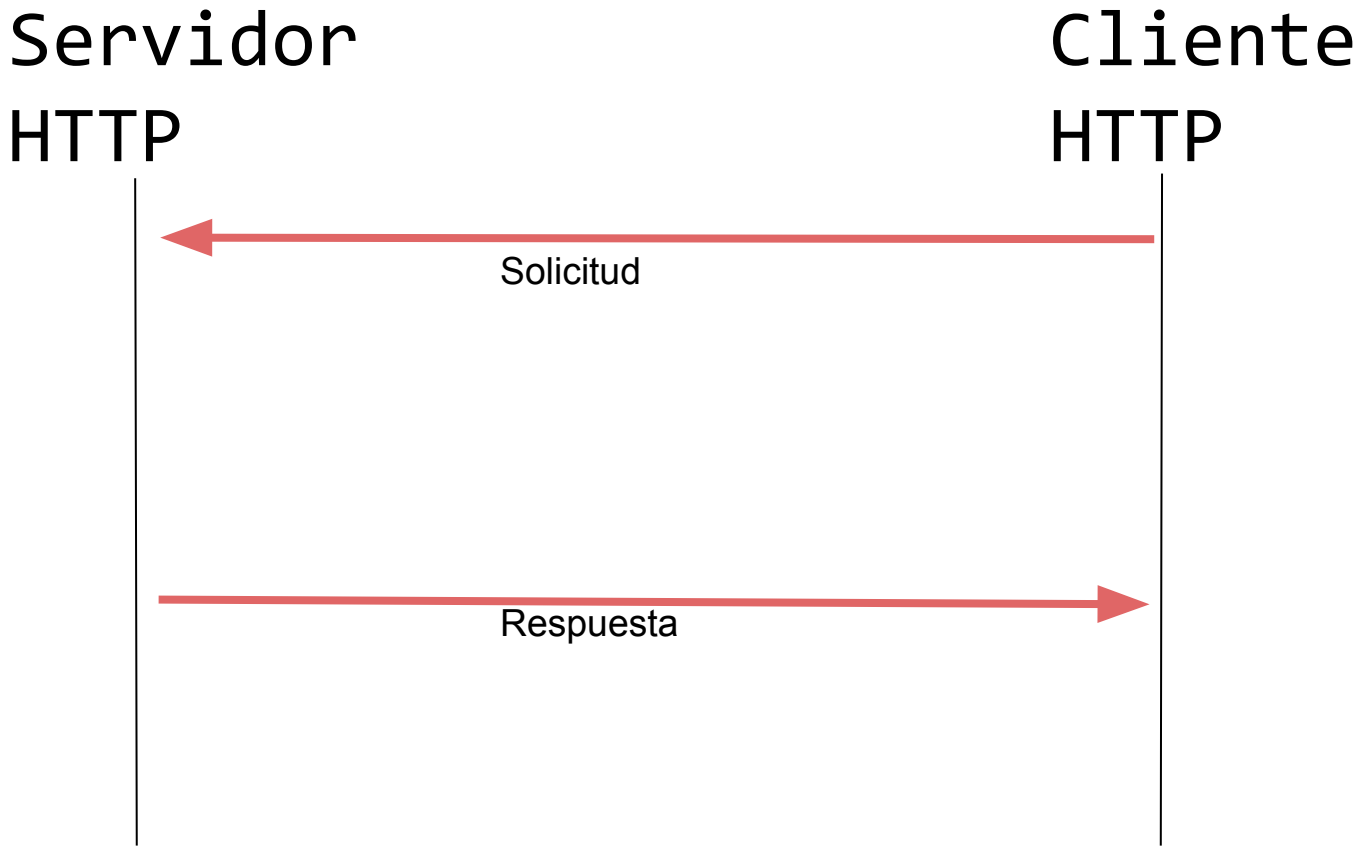
Así como en *networking* se define un protocolo para comunicarse entre cliente y servidor, en el mundo real existe un protocolo altamente utilizado para comunicarse a través de internet: **HTTP**

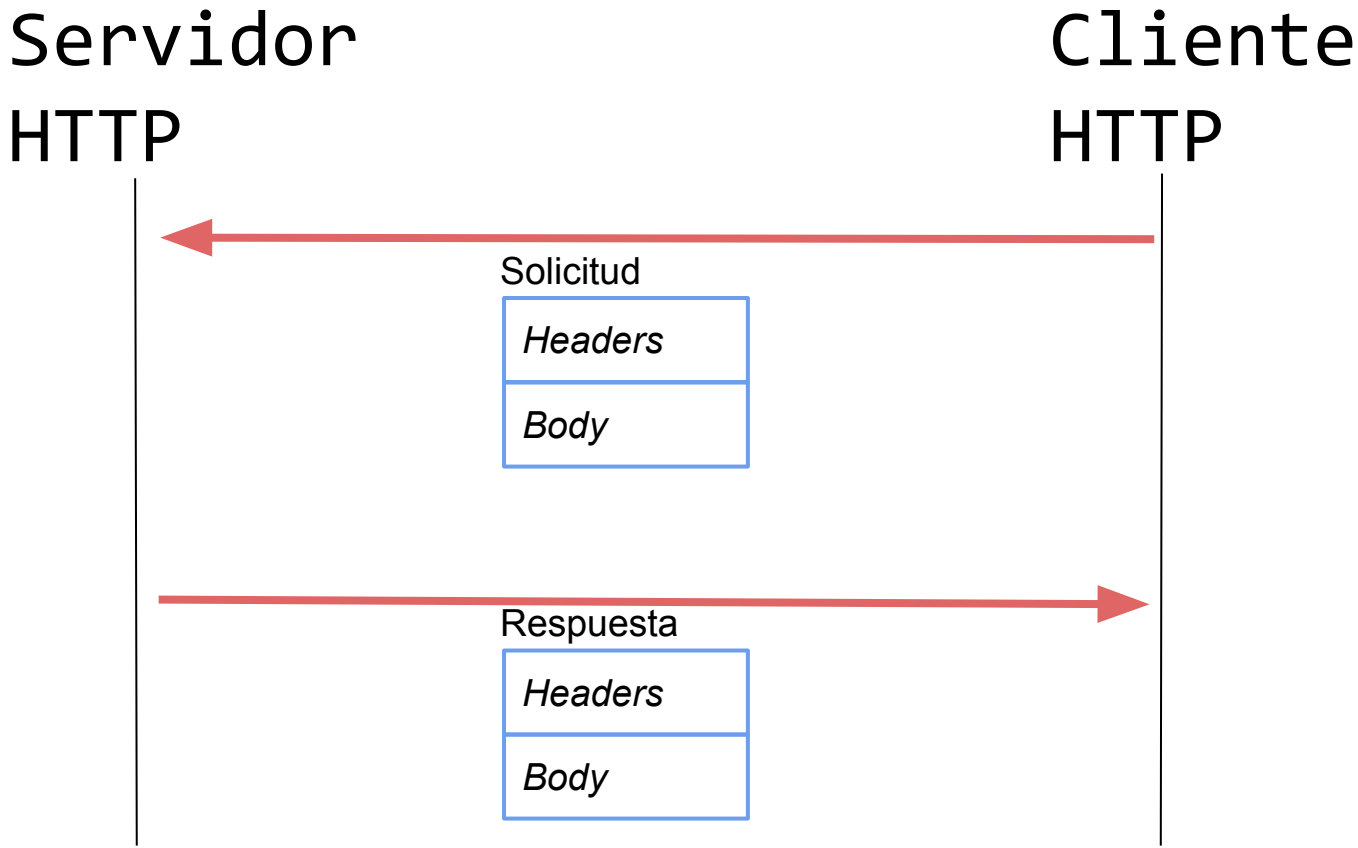
- Cliente envía una petición, y el servidor le envía una respuesta.
- Una vez enviada la respuesta, el servidor “olvida” dicha petición.

Entonces, ¿cómo nos comunicamos con el servidor que queremos? 🤔

¿Cómo funciona la comunicación una vez que accedo a una URL?




Servidor
HTTP
Cliente
HTTP
Solicitud
Respuesta


Servidor HTTP
Cliente HTTP
Solicitud
Headers
Body
Respuesta
Headers
Body

Tenemos el cómo comunicarnos.

Ahora…

¿Qué es lo que comunicamos?

Tipos de solicitudes: Informando al servidor lo que queremos hacer.

- **GET**: Pedimos la representación de un recurso **sin cambiar nada** en el servidor.
- **POST**: **Creamos** un recurso.
- **PATCH**: Aplica **modificaciones parciales** a un recurso.
- **PUT**: **Reemplaza completamente** un recurso existente.
- **DELETE**: **Elimina** un recurso.

Códigos de respuesta: Informando al cliente lo que ocurrió.

- **200**: Ok, solicitud exitosa.
- **403**: La solicitud es correcta, pero se rechaza dar una respuesta.
- **404**: El recurso solicitado no se encuentra en el servidor.
- **500**: Error interno del servidor

Y muchos más…

Una URL puede (o no) tener varias acciones asociadas

Ejemplo

Supongamos que tenemos la página web **http://www.paginaiic2233.com/**, que tiene un recurso estudiantes. Para poder interactuar con los datos de la página, se habilitaron las siguientes URLs:
http://www.paginaiic2233.com/estudiantes
http://www.paginaiic2233.com/estudiante/<id>

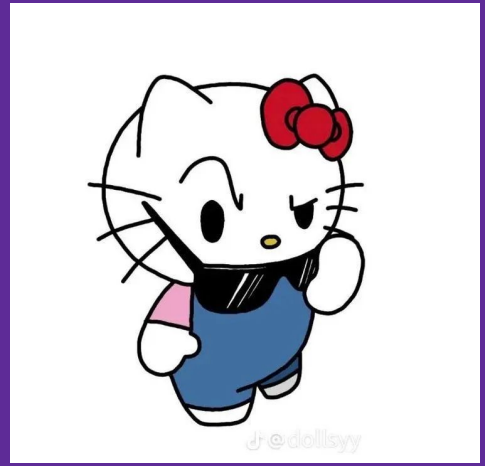
Donde el id es un número. Considerando esto, podríamos pensar que dicha página tiene las siguientes acciones:

Ejemplo

URL	GET	POST	PUT	PATCH	DELETE
/estudiantes	Obtiene una lista de todos los estudiantes	Crea un nuevo estudiante			
/estudiante/1/	Obtiene los datos del estudiante con id 1		Reemplaza todos los datos del estudiante con id 1	Reemplaza algunos atributos del estudiante con id 1	Elimina al estudiante con id 1

*Esto es solo un ejemplo, las funcionalidades dependerán de lo que decida quien implementó el *webservice*.

¿Cómo sabe una API quién soy?



Autenticación

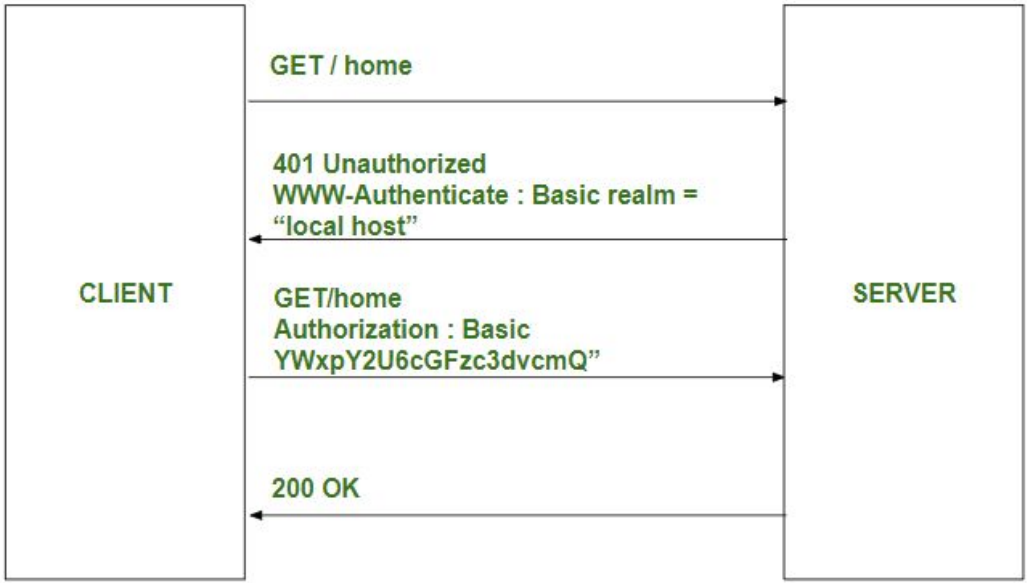
Problema

- Hay acciones, como editar la base de datos, que no todo usuario puede hacer.
- Generalmente las APIs se componen de funciones que “no memorizan entre una solicitud y otra”. Por lo tanto, no podemos hacer una solicitud de “login” y luego una de “modificar base de datos” en donde recuerde el login previo.
- Surge la necesidad de un mecanismo que permita, en cada solicitud, poder identificar al usuario.

Autenticación

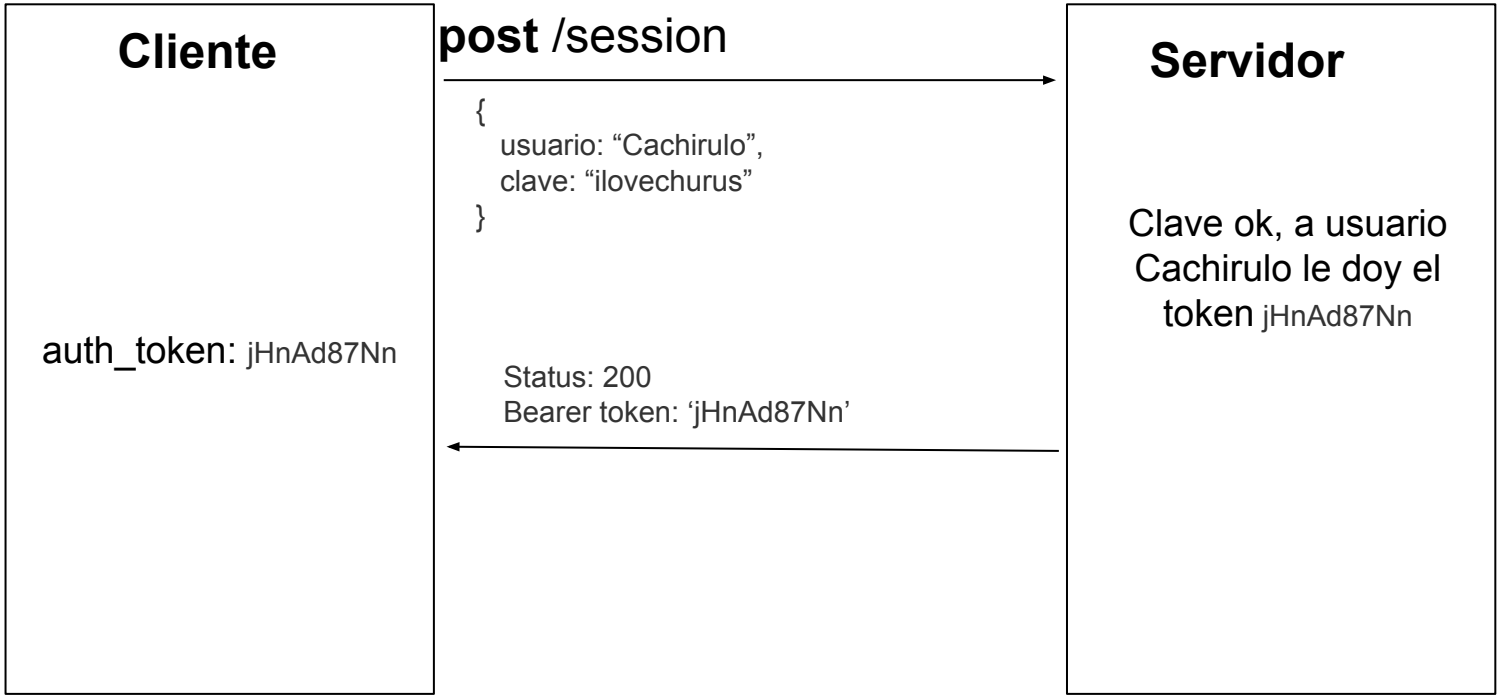
Solución

- En el *header* podemos inclui un **token especial** de acceso que a la API le sirve para identificar y verificar si permisos.



Autenticación - Ejemplo (obtener el token)

Autenticación - Ejemplo (obtener el token)



Autenticación - Ejemplo (usar el token)

Autenticación - Ejemplo (usar el token)

¿Y esto en Python cómo se ve?

requests

```python
import requests


url =  "https://fakestoreapi.com/products/1"
response = requests.get(url) # Método GET
print(response.status_code)  # Status 200
print(response.json())       # Respuesta del servidor
```

requests

```python
import requests
import json

url =  "https://fakestoreapi.com/products"
# Información adicional (datos + headers) para crear un producto
datos_a_subir = json.dumps({ "producto": "Manga", "precio": 7000 })
headers = { "Authorization": "token GjsdxYkdLSDandsGH" }

# Hacer la request
request.post(url, data=datos_a_subir, headers=headers) # Método POST
```

Levantamiento de una API

Flask

```
from flask import Flask

app = Flask(__name__) # Creamos la API


# Definimos endpoint “/” que acepta solo GET
@app.route("/", methods=["GET"])
def hello_world():
    return {"texto": "Holi"}


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

Flask

```
from flask import Flask, request

app = Flask(__name__) # Creamos la API

# Definimos endpoint “/dado” que acepta GET y POST
@app.route("/dado", methods=["GET", "POST"])
def dado():
    if request.method == "POST":
        numero = random.randint(0, 6)
        return {"resultado": numero", "método": "POST"}

    numero = random.randint(-4444, -11)
    return {"resultado": numero, "método": "GET"}
```

Temas adicionales de interés

Temas adicionales de interés

- Hay tokens que incluso pueden almacenar información: JWT.io
- Existen muchos más códigos HTTP
  - Códigos de estado de respuesta HTTP
  - Existe su explicación gatuna: HTTP Cats
  - El código 418 es cuando el servidor se rehúsa a preparar café porque es una tetera.
- Existen otras librerías para programar rápidamente una API como FastApi.

Programación Avanzada
IIC2233 2024-2

Hernán Valdivieso - Daniela Concha - Francisca Ibarra - Lucas Van Sint Jan - Francisca Cattan