

«Talento Tech»

Back-End

Node JS

Clase 07



Clase N° 7- Cómo funciona la web

Temario:

1. ¿Cómo funciona Internet?
 - Protocolo TCP/IP
2. **Modelo Cliente/Servidor**
3. **Protocolo de comunicación**
 - Introducción a HTTP
 - HTTP Seguro
 - Certificados de seguridad
4. **HTTP en profundidad**
 - Métodos HTTP
 - Headers
 - Body
 - Códigos de estado
5. **URI: URL + URN**

Objetivos de la Clase

En esta clase, los estudiantes entenderán el modelo cliente-servidor y su importancia en la arquitectura de Internet. Aprenderán cómo las redes se comunican utilizando protocolos estándar y explorarán el funcionamiento del protocolo TCP/IP y su rol en la transferencia de datos. Además, diferenciarán entre HTTP y HTTPS, comprendiendo sus características y beneficios. También conocerán la estructura del protocolo HTTP en profundidad y descompondrán las partes que componen una URI, analizando las diferencias entre URL y URN.

¿Cómo funciona Internet?



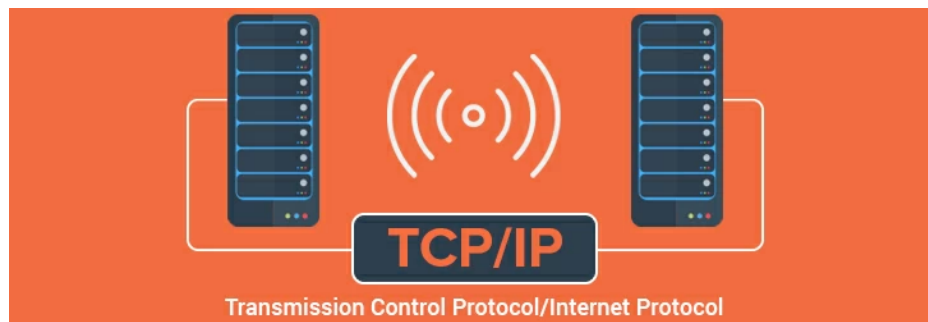
Aunque a veces se perciba como una "nube" que almacena datos y aplicaciones, **Internet** es en realidad una gigantesca "red de redes".

Una red conecta diferentes puntos (como computadoras o dispositivos) para que puedan comunicarse, incluso si no están directamente conectados entre sí. Internet funciona de la misma forma, permitiendo que todos los

dispositivos estén interconectados globalmente.

Para que estas redes colaboren, utilizan un "idioma común" llamado TCP/IP, un conjunto de reglas que garantiza que los datos puedan viajar de un dispositivo a otro, incluso a través de varias redes intermedias.

Protocolo TCP/IP



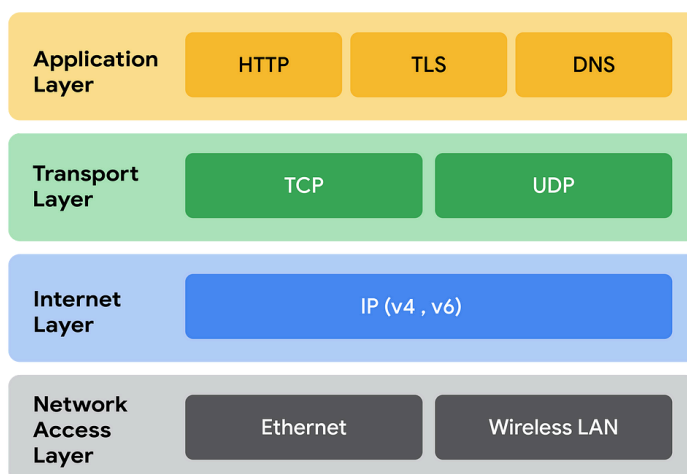
El protocolo TCP/IP es el que asegura la comunicación en Internet a través de este protocolo es posible transmitir información entre dispositivos conectados a la red. Para lograrlo se utiliza la conocida **IP (Internet Protocol)** que encuentra y define la dirección del destino, como si fuera una dirección o código postal y el **TCP (Transmission Control Protocol)** que garantiza que los datos lleguen completos y en el orden correcto

La **IP** es el identificador único a un dispositivo conectado a la red, sin embargo, no apunta a un recurso en específico de ese dispositivo, es decir, no es la dirección a la casa de un amigo, más bien es el código del barrio donde él vive, entendiendo al barrio como un servidor o un PC que brinda o recibe información y la casa de tu amigo como la porción de ese servidor donde se encuentra guardada dicha información o el lugar donde debe entregarse.

Por otra parte, el **TCP** funciona como el servicio de mensajería o de correo encargado de la entrega de la información. Antes de enviar algo, llama al destinatario para confirmar que está listo. Divide un paquete grande en partes manejables, les coloca etiquetas para que no se pierdan ni se desordenen, y luego supervisa cada paso del proceso para garantizar que todo llegue como se espera. Si algo se pierde, no se preocupa: lo reenvía hasta que esté seguro de que todo está completo y en su lugar.

Ambos trabajan juntos para que la información viaje de manera confiable. Por ejemplo, si envías un correo electrónico o visitas una página web, el protocolo TCP/IP gestiona el envío y recepción de esos datos.

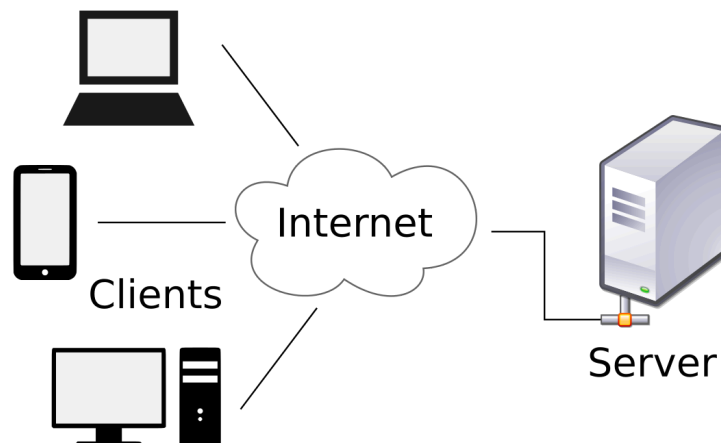
El protocolo TCP/IP es algo más que solo dos elementos trabajando juntos, sino más bien una herramienta multicapa donde cada una de ellas tiene una función específica:



Por el momento ya conocimos la **Transport Layer** o capa de transporte y la **Internet Layer** o capa de Internet, además existen las **Network Access Layer** o capa de acceso a la red que es la capa física con la cual realizamos la conexión de forma inalámbrica (WIFI) o cableada (Ethernet) y la **Application Layer** o capa de aplicación, encargada de generar la información y requerir las conexiones. Hablaremos de esta última más adelante.

Modelo Cliente/Servidor.

Ahora que entendemos mejor qué es internet y cómo conecta millones de dispositivos a través del mundo, profundicemos en un concepto clave para todos aquellos que trabajan en el desarrollo de servicios y aplicaciones web.



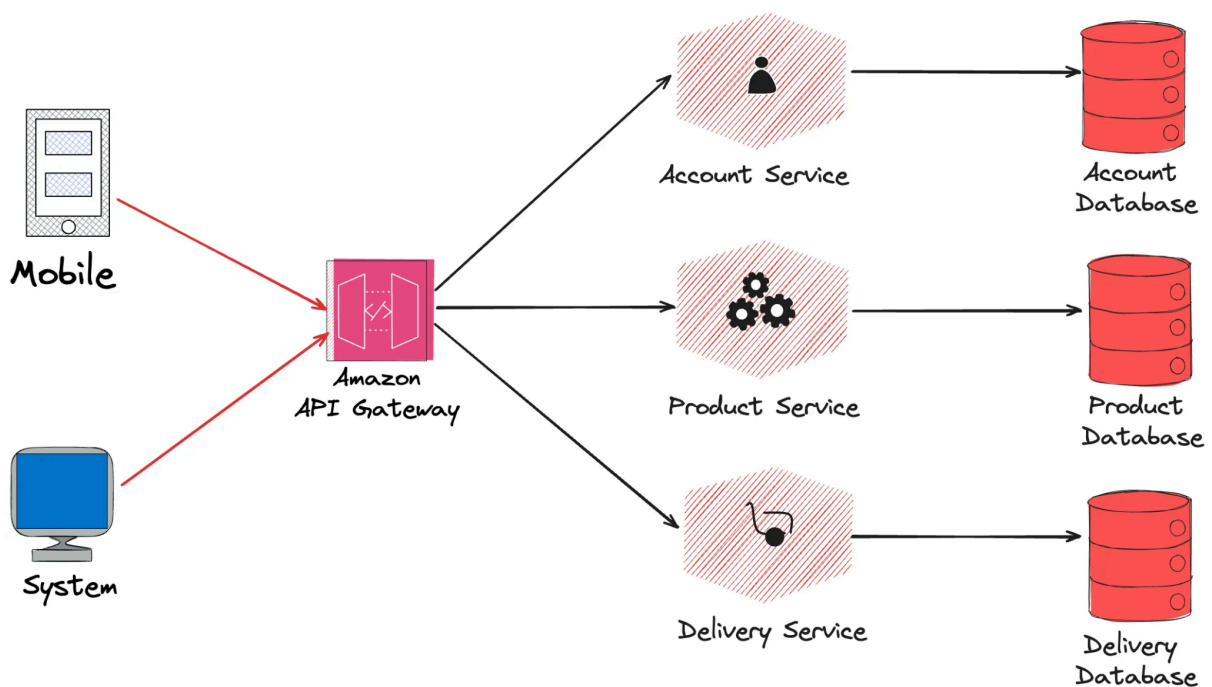
El modelo cliente/servidor describe cómo interactúan dos dispositivos para intercambiar información y recursos. En este modelo, un dispositivo toma el rol de cliente y solicita servicios o datos, mientras que otro dispositivo actúa como servidor, respondiendo a esas solicitudes. Por ejemplo, al navegar por internet, tu navegador (cliente) solicita una página web al servidor, que procesa esa solicitud y devuelve el contenido para que lo veas. Este modelo es escalable, ya que múltiples clientes pueden conectarse simultáneamente a un servidor, y es común en aplicaciones como correo electrónico, juegos en línea o servicios en la nube.

Este modelo es la base de la comunicación entre servicios en la web pero además ayuda a diferenciar 2 conceptos bastante difundidos, que son el **frontend** y el **backend**. Cuando hablamos de **frontend** lo entendemos como la capa visual de un servicio web que es

ejecutado dentro de un navegador o de un dispositivo móvil como aplicación, pero a su vez, también toma el rol de cliente ya que es quien solicita información constantemente con el fin de mostrarla al usuario, entonces sabemos que es quien hace las request. Por otra parte, cuando hablamos de **backend**, lo entendemos como la capa lógica y que tiene acceso a los datos que a su vez es la que se ejecuta dentro de un servidor, siendo la que responde con recursos o datos al cliente.

Si bien en el modelo cliente/servidor es común que sea código "frontend" quien ocupe el rol de cliente y código "backend" el rol de servidor, también es común que un **backend** pueda requerir datos a otro **backend** (como sucede en lo que se conoce como la arquitectura de microservicios), por lo cual el concepto de cliente no es exclusivo de la capa frontend de la aplicación, simplemente es una forma más de diferenciar ambos entornos.

Microservicios



Por ejemplo, en la imagen anterior existen varios elementos ocupando al mismo tiempo el rol de **cliente** y de **servidor**. En el caso de **Mobile** y **System** además de ser el lugar donde se ejecuta el código desarrollado por perfiles **frontend**, toman el rol de “clientes” ya que solicitan datos. Luego, el **API Gateway** toma el rol de “servidor” ya que responde a los requerimientos de los clientes, pero a su vez también toma el rol de “cliente” ya que para entregar los datos solicitados, necesita consultar a otros servidores web que brindan determinados servicios como la capa de autenticación o la capa de producto. Finalmente, estos servicios operan en el mismo rol dual ya que son “servidores” de API Gateway pero a su vez son “clientes” de los “servidores” de bases de datos que contienen la información necesaria. Todos los actores en este flujo de información que no sean **Mobile** y **System**, forman parte del código desarrollado por perfiles de programación **backend**. Normalmente, un programador con los conocimientos necesarios para desenvolverse en ambos roles, es conocido como un programador **FullStack**.

Finalmente, es importante mencionar que para que la interacción entre cada cliente y cada servidor sea exitosa es necesario que estos se comuniquen a través de un lenguaje común. Anteriormente hablamos de **TCP/IP**, sin embargo, la encargada de manejar esta parte de la comunicación es la capa conocida como **Application Layer** donde nos encontramos con el protocolo de comunicación llamado **HTTP**, el cual conoceremos a continuación.

Protocolos de comunicación

Ahora conoceremos el lenguaje que utilizan clientes y servidores para comunicarse. Este parte de un estándar de la web y si bien existen otros como onion, **HTTP** es el lenguaje utilizado por defecto.

Introducción a HTTP (Hypertext Transfer Protocol)

Como podemos observar en la siguiente imagen, un cliente realiza una **request** o petición bajo el estándar de comunicación **HTTP** y el servidor utiliza el mismo protocolo para enviar una **response** o respuesta.



HTTP funciona como un "idioma común" para intercambiar información. Por ejemplo, al escribir "google.com" en tu navegador, éste envía una solicitud al servidor de Google, que responde enviando la página web para que la veas.

Además es un protocolo sin estado, por lo que no guarda ninguna información sobre conexiones anteriores. Esto en algunos casos puede ser un problema ya que en el desarrollo de aplicaciones web frecuentemente se necesita mantener estado, por ejemplo, para el uso de carritos de compra en páginas de comercio electrónico. Para esto se usan las cookies, que es información que un servidor puede almacenar en el sistema cliente (navegador).

Este protocolo se compone de varias partes clave que facilitan la comunicación entre clientes y servidores en la web. Estas incluyen:

Métodos HTTP: Son las acciones que el cliente solicita al servidor, como obtener información (**GET**), enviar datos (**POST**), actualizar recursos (**PUT** o **PATCH**), o eliminarlos (**DELETE**), entre otros.

URLs (Uniform Resource Locators): Identifican de manera única los recursos en la web, especificando la ubicación y cómo acceder a ellos.

Cabeceras (Headers): Contienen metadatos sobre la solicitud o respuesta, como información de autenticación, tipo de contenido, tamaño de datos y más.

Cuerpo del mensaje (Body): Es el contenido enviado en las solicitudes o respuestas, como datos en formato JSON, HTML, archivos u otros recursos.

Códigos de estado (Status Codes): Indican el resultado de la solicitud, como éxito (**200 OK**), redirección (**301 Moved Permanently**), error del cliente (**404 Not Found**) o error del servidor (**500 Internal Server Error**).

HTTP Seguro.

HTTPS (Hypertext Transfer Protocol Secure) es una versión segura de HTTP que encripta los datos transmitidos entre el navegador y el servidor, protegiendo la privacidad del usuario. Cuando visitas un sitio con HTTPS, se establece una conexión segura mediante un certificado digital que verifica la identidad del sitio y cifra la información, evitando que pueda ser interceptada.



Certificados SSL/TLS.



Los certificados SSL (Secure Sockets Layer) o TLS (Transport Layer Security, su versión mejorada) son documentos digitales que verifican la identidad de un sitio web y aseguran la comunicación entre el navegador del usuario y el servidor. Estos certificados encriptan los datos transmitidos, evitando que terceros los intercepten. Aunque TLS es la tecnología más reciente, el término SSL

sigue siendo comúnmente usado.

Un certificado SSL/TLS es emitido por una Autoridad de Certificación (CA), que verifica la autenticidad del sitio. También hay opciones gratuitas, como las ofrecidas por Let's Encrypt, que facilitan la adopción de conexiones seguras en los sitios web.

HTTP en profundidad.

Métodos HTTP.

Los métodos HTTP definen las acciones específicas que los clientes desean realizar sobre un recurso en el servidor. Funcionan como instrucciones claras para que el servidor entienda lo que se espera de él.

store Access to Petstore orders	
GET	/store/inventory Returns pet inventories by status
POST	/store/order Place an order for a pet
GET	/store/order/{orderId} Find purchase order by ID
DELETE	/store/order/{orderId} Delete purchase order by ID

Los métodos más utilizados incluyen GET, que solicita un recurso sin modificarlo, como al cargar una página web; POST, que envía datos al servidor, por ejemplo, al completar un formulario; PUT y PATCH, que actualizan recursos existentes, siendo el primero para reemplazos

completos y el segundo para modificaciones parciales; y DELETE, que elimina un recurso identificado.

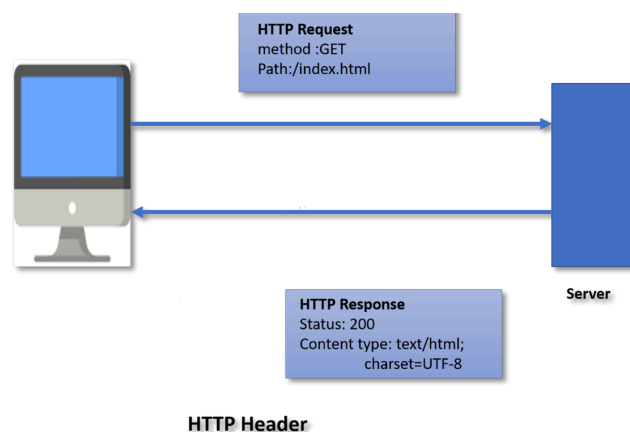
Más adelante, utilizaremos estos métodos dentro de nuestro código para definir cómo se va a realizar la comunicación entre clientes con nuestro servidor pero es importante destacar que cada uno de ellos forma parte de un estándar dentro del protocolo, es decir, no existe nada que impida que puedas borrar un recurso usando PUT en lugar de DELETE, o que puedas enviar datos usando GET en lugar de POST, por eso es vital conocer el rol que cumple cada uno para poder adaptarse a este estándar de forma correcta.

Finalmente, cabe mencionar que hacemos énfasis en estos métodos ya que son los más utilizados, sin embargo existen otros dentro del protocolo HTTP que puedes buscar si deseas profundizar en el tema.

Headers.

Las cabeceras HTTP mayormente conocidas como **headers**, son componentes clave de las solicitudes y respuestas, ya que contienen información adicional sobre la operación que se realiza. Están formadas por pares clave-valor y pueden incluir datos como el tipo de contenido esperado o enviado (**Content-Type**), la longitud de los datos (**Content-Length**), o detalles sobre la autenticación requerida (**Authorization**). También permiten al cliente y al servidor intercambiar datos de control, como especificar si una conexión debe mantenerse abierta o cerrarse después de completar la comunicación.

Los headers, son considerados los metadatos de una petición ya que terminan siendo invisibles para el usuario final, sin embargo, son esenciales para definir cómo se procesan las solicitudes y respuestas ya que la comunicación que contienen puede determinar si una solicitud es resuelta de manera correcta o no.



Por otra parte, si analizamos un poco más a fondo, llegaremos a la conclusión de que puede existir cabeceras en dos (2) sentidos, al pedir información y al enviar información, pero además existen datos compartidos y algunos específicos:

1. Cabeceras de solicitud

- **User-Agent:** Describe el software que realiza la petición (navegador, aplicación, etc.).
- **Accept:** Especifica el tipo de contenido que el cliente puede manejar (por ejemplo, `text/html`, `application/json`).
- **Authorization:** Envía credenciales para autenticación.
- **Cookie:** Incluye información de sesión almacenada en el cliente.

2. Cabeceras de respuesta

- **Content-Type:** Indica el tipo de contenido del cuerpo de la respuesta (por ejemplo, `text/html`).
- **Set-Cookie:** Envía cookies desde el servidor al cliente para futuras solicitudes.
- **Cache-Control:** Especifica reglas sobre el almacenamiento en caché de la respuesta.
- **Content-Length:** Indica el tamaño del cuerpo de la respuesta, en bytes.

3. Cabeceras generales

- **Connection:** Define si la conexión debe mantenerse abierta (`keep-alive`) o cerrarse.
- **Date:** Especifica la fecha y hora en que se generó el mensaje.
- **Transfer-Encoding:** Indica si la respuesta utiliza codificación como `chunked`.
- **Upgrade:** Solicita una actualización de protocolo (por ejemplo, a WebSocket).

4. Cabeceras específicas para control y seguridad

- **Access-Control-Allow-Origin:** Permite definir los orígenes autorizados para acceder a los recursos (CORS).
- **Strict-Transport-Security:** Fuerza el uso de HTTPS para las conexiones futuras.
- **X-Content-Type-Options:** Evita que los navegadores interpreten incorrectamente los tipos de contenido.

- **Content-Security-Policy:** Permite definir políticas de seguridad que restringen los recursos cargados en una página.

Para entender un poco más de que se tratan los headers, podemos resumir que son metadatos que se envían en cada petición y en cada respuesta, algunos de ellos son seteados por defecto por los navegadores y otros son establecidos explícitamente al momento de escribir el código sabiendo que nuestra petición o nuestra respuesta necesitará llevar esa determinada información.

Body

El cuerpo o **body** de un mensaje HTTP es el espacio donde se transmiten los datos reales entre el cliente y el servidor. En las solicitudes, puede incluir información como el contenido de un formulario enviado o un archivo subido al servidor. En las respuestas, el cuerpo puede contener datos como un documento HTML, una imagen, o un archivo JSON con datos estructurados.

POST /?id=1 HTTP/1.1

Request line

```
Host: www.swingvy.com
Content-Type: application/json; charset=utf-8
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10.12; rv:53.0)
Gecko/20100101 Firefox/53.0
Connection: close
Content-Length: 136
```

Header

```
{
  "status": "ok",
  "extended": true,
  "results": [
    {"value": 0, "type": "int64"},
    {"value": 1.0e+3, "type": "decimal"}
  ]
}
```

Body message

No todas las solicitudes HTTP tienen un cuerpo; por ejemplo, los métodos GET y DELETE suelen funcionar sin este componente, mientras que métodos como POST y PUT casi siempre lo incluyen. El formato y contenido del cuerpo están determinados por las cabeceras, particularmente Content-Type, que define cómo deben interpretarse los datos transmitidos.

Cabe destacar que mientras en el método GET la información viaja en la URL como parámetros visibles en todos los momentos, en los métodos POST y PUT que se encuentran habilitados a utilizar el body como canal de transporte de la información, estos datos se encuentran “ocultos” a la vista común del usuario pero es posible consultarlos mediante el uso de herramientas del navegador como las devtools, por eso es que si bien es el canal correcto para enviar datos sensibles como contraseñas, estas siempre deben viajar encriptadas u ofuscadas.

Códigos de estado.

Ahora la pregunta es, ¿qué sucede cuando se resuelve una solicitud HTTP y qué pasa si esta se encontró con algún tipo de error que no le permitió devolver la información solicitada.

Para ello existe lo que se conocen como los códigos de estado o códigos de respuesta, que no son más que respuestas numéricas emitidas por el servidor para indicar el resultado de una solicitud.

Estos códigos están organizados en categorías que reflejan el tipo de resultado. Por ejemplo, los códigos en el rango 200 indican éxito, como **200 OK**, que señala que la solicitud fue procesada correctamente. Los códigos 300 están relacionados con redirecciones, como **301 Moved Permanently**. Los códigos 400 representan errores del cliente, como **404 Not Found**, cuando el recurso solicitado no existe, **403 Forbidden**, indicando que el usuario no posee acceso al recurso o debe autenticarse y finalmente, los códigos 500 que indican errores del servidor, como **500 Internal Server Error**, que indica situaciones donde por ejemplo no se pudo hacer contacto con el servidor debido, por ejemplo, a que este no se encuentre operativo.

Estos códigos no solo informan sobre el estado de la interacción, sino que también guían la forma en que los desarrolladores deben responder a problemas en sus aplicaciones, por ejemplo, en el caso de **404 Not Found** que se presenta cuando un recurso solicitado no se encuentra, el desarrollador backend puede “capturarlo” y devolver otra cosa en su lugar.

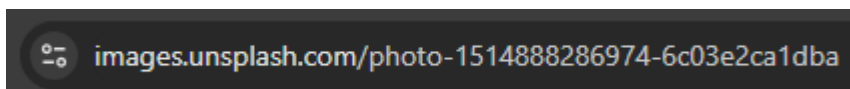
En el caso de un sitio web, si intentamos entrar a miweb.com/nosotros y esta página no existe, devolveremos una página prediseñada que indique que no se pudo encontrar el recurso solicitado.


URI: URL + URN.

Ya conocimos elementos importantes de la comunicación cliente/servidor, sin embargo nos falta mencionar uno de suma importancia ya que es la representación explícita de la información a la cual deseamos acceder.

El objetivo al que apunta una solicitud HTTP se lo conoce como “recurso” y dicho recurso puede ser de cualquier tipo, ya sea un documento o archivo, una imagen, datos en formato JSON o XML, entre otros.

Imaginemos que estamos buscando imágenes sobre un tema en particular y realizamos una búsqueda desde google, todos esos resultados son recursos que están alojados o guardados en algún servidor dentro de toda la web y ya sea que abramos el sitio web donde está esa imagen o que abramos la imagen en una nueva pestaña, la barra de búsqueda siempre tendrá un bloque de texto con un formato parecido al siguiente:



 images.unsplash.com/photo-1514888286974-6c03e2ca1dba

Posiblemente te resulte familiar, ya que este formato es lo que normalmente se conoce como **URL**, pero en realidad se les llama **URI** y dependiendo su contenido nos darán acceso a todos los recursos de la web.

URI

A cada recurso se lo identifica con un **URI** (identificador uniforme de recursos) que no es más que un bloque de texto utilizado en peticiones (request) que se realicen ya sea desde la barra de direcciones de un navegador o desde cualquier otro método que permita realizar una petición como fetch o ajax.

Si intentamos descomponer una URI nos encontraremos con determinadas propiedades que la conforman, por ejemplo consideremos la siguiente estructura:

<https://www.ejemplo.com:443/articulos?categoria=tecnologia#introduccion>

- **Protocolo (scheme):** [https](#)
- **Dominio (authority):** [www.ejemplo.com](#)
- **Puerto (port):** [443](#)
- **Ruta (path):** [/articulos](#)
- **Parámetros de consulta (query params):** [categoria=tecnologia](#)
- **Fragmento (fragment):** [#introduccion](#)

Las propiedades que componen una **URI** y se agrupan en 2 (dos) partes, la **URL** y la **URN**, hagamos un poco de zoom y veamos qué contiene cada una..

URL

Quizás la más nombrada, normalmente se la confunde con la URI, pensando que la URL o localizador uniforme de recursos, es toda la dirección que apunta a un recurso web, sin embargo, es tan solo una parte de la estructura que dirige a dicho recurso.

La URL no solo identifica un recurso, sino que también proporciona información sobre cómo acceder a él, como su ubicación en la red. Ejemplo:

<https://www.ejemplo.com/index.html>.

De la URL forman parte el **scheme** y el **authority** compuesto a su vez por **userinfo** (opcional), **host** (conocido normalmente como dominios y subdominios) y el **port** (dato normalmente no visible al usuario final ya que los hostings lo tienen definido por defecto).



URN

También forma parte de la URI, pero en este caso define la parte de acceso a los recursos de forma detallada. Tomando el ejemplo anterior

`https://www.ejemplo.com/index.html`, consideremos que queremos acceder a un elemento HTML dentro de `index.html` que tiene el `id` “contact-form”, para ello podemos utilizar el `#` que nos permite identificar una parte del recurso accedido. De esta manera `https://www.ejemplo.com/index.html#contact-form` nos llevará directamente a la posición de la web donde se encuentra ese elemento. A este identificador se le llama fragment y es parte de la **URN**, pero también existen otros:

- **path:** `/articulos`, identifican un recurso a través de una ruta predefinida por el servidor, en lugar de apuntar a un documento `.html`, `.jpeg`, las rutas permiten “enmascarar” los archivos y hacerlos accesibles mediante un path predefinido.
- **query params:** `categoria=tecnologia`, los parámetros utilizados por defecto en peticiones a través de método GET, nos ayudan a enviar información adicional al servidor siempre que utilicemos este método como método HTTP de consulta. Vale mencionar que los query params son visibles en todo momento para el usuario final por lo que no es recomendable enviar datos sensibles y su uso particularmente se aplica en filtros que refinan la experiencia.
- **fragment:** `#introduccion`, apuntan a una sección HTML identificada mediante el identificador proporcionado.

Cada una de estas herramientas y el uso del método HTTP para requerir y responder información mediante solicitudes a través de la web, lo trabajaremos al momento de crear nuestro propio servidor.

¡No te pierdas las siguientes clases!



Materiales y Recursos Adicionales:

- Comunicación Cliente/Servidor - [documentos de la web](#)
- Códigos de estado de respuesta HTTP - [MDN Web Docs - HTTP Status](#)
- Métodos de petición HTTP - [MDN Web Docs - HTTP Methods](#)

Preguntas para Reflexionar:

- ¿Por qué es importante que los dispositivos conectados a Internet hablen un "idioma común" como TCP/IP?
- ¿Cómo se relacionan las partes de una URI (como el dominio, el puerto o los parámetros de consulta) con el acceso y localización de recursos en la web?
- ¿Qué ventajas tiene el modelo cliente/servidor para la escalabilidad y flexibilidad de aplicaciones web modernas?
- ¿Por qué crees que es relevante diferenciar entre métodos HTTP como GET y POST al desarrollar aplicaciones web?

Próximos Pasos:

Servidores Web y Patrones de Arquitectura: Aprenderemos sobre cómo funcionan los servidores que dan vida a internet y conoceremos los cimientos de los proyectos de programación.

Creando un Servidor Web: Daremos nuestros primeros pasos en la creación de servidores web con Node JS.

Modelando una API Rest: Nos iniciaremos en la metodología para crear nuestra primera API Rest.



Buenos Aires
aprende
Agencia de Políticas para el Futuro

BA Buenos
Aires
Ciudad