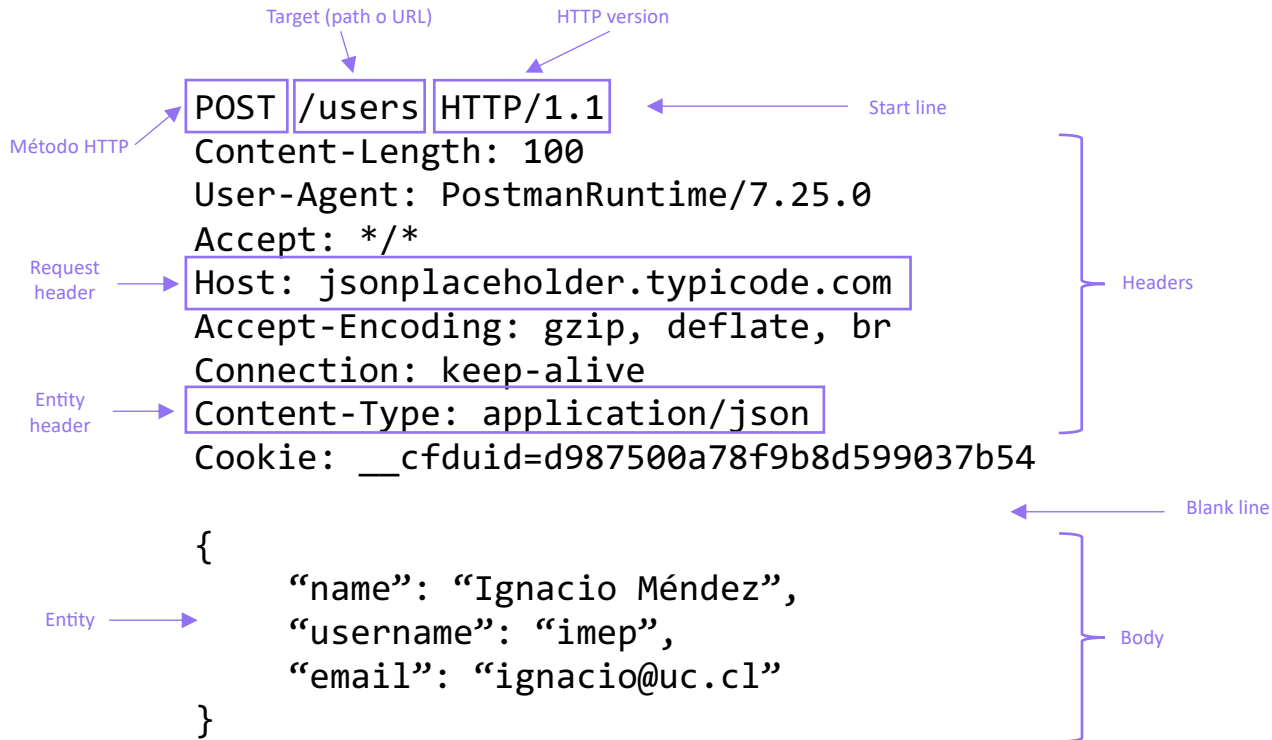


Resumen Tecnologías y Aplicaciones Web

Request HTTP estructura



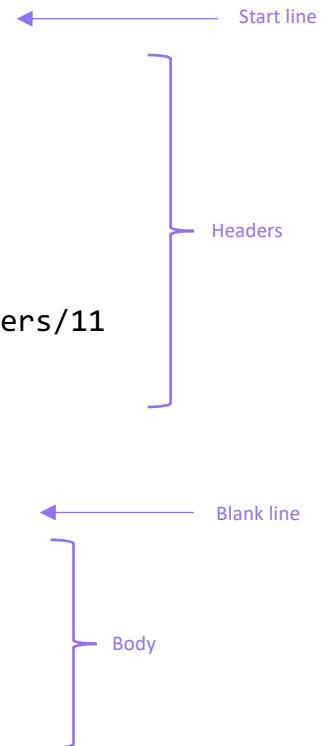
Métodos HTTP

- **GET:** Obtener representación de un recurso específico.
- **POST:** Crear un nuevo recurso en base a entidad adjunta.
- **PUT:** Reemplazar recurso específico con entidad adjunta.
- **PATCH:** Modificar recurso específico en base a entidad adjunta.
- **DELETE:** Eliminar recurso específico.

Response HTTP estructura

```
HTTP/1.1 201 Created
Date: Thu, 03 Sep 2020 06:13:27 GMT
Content-Length: 106
Content-Type: application/json; charset=utf-8
Connection: keep-alive
Access-Control-Allow-Credentials: true
Cache-Control: no-cache
Access-Control-Expose-Headers: Location
Location: http://jsonplaceholder.typicode.com/users/11
X-Content-Type-Options: nosniff
Etag: W/"6a-MIZeGYXbkV9UzaMzb/OuKeiqb00"
Via: 1.1 vegur
Server: cloudflare
```

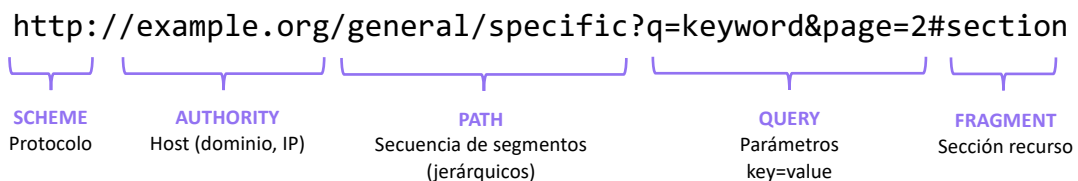
```
{
  "name": "Ignacio Méndez",
  "username": "imep",
  "email": "ignacio@uc.cl",
  "id": 11
}
```



Status codes

- **1xx:** Informative → *Request* recibido y continuando proceso.
- **2xx:** Éxito → *Request* recibido, entendido y aceptado.
- **3xx:** Redirección → Acciones necesario para completar request.
- **4xx:** Error de cliente → *Request* con sintaxis errónea o no válido.
- **5xx:** Error de servidor → Servidor falló en completar *request*.

URI: Texto que permite **identificar** de forma única un **recurso web**.



URL: Tipo de URI que incorpora información de la **ubicación del recurso web**.

HTML (*HyperText Markup Language*)

Lenguaje de marcado utilizado para crear y estructurar el contenido de una página web. Utiliza etiquetas y elementos para definir la forma en que se deben mostrar los diferentes elementos de una página, como encabezados, párrafos, imágenes, enlaces, listas y más.

CSS (*Cascading Style Sheets*)

Lenguaje de hojas de estilo que se utiliza para controlar la apariencia y el diseño de una página web. Permite definir reglas y estilos para los elementos HTML, como colores, fuentes, márgenes, tamaños, posicionamiento y otros atributos visuales.

JavaScript

- **Constructores y Clases:**

```
// Definición del constructor para un objeto "Persona"
function Persona(nombre, edad) {
  this.nombre = nombre;
  this.edad = edad;
  this.saludar = function() {
    console.log('Hola, mi nombre es ${this.nombre} y tengo ${this.edad} años.');
```

```
// Definición de una clase "Persona"
class Persona {
  constructor(nombre, edad) {
    this.nombre = nombre;
    this.edad = edad;
  }
  saludar() {
    console.log('Hola, mi nombre es ${this.nombre} y tengo ${this.edad} años.');
```

- **Callback:** Función que se pasa como argumento a otra función y que se ejecuta después de que la función principal ha terminado de hacer su trabajo.

```
function realizarTareaAsincrona(callback) {
  // Simulamos una operación asíncrona con un retraso de 2 segundos
  setTimeout(function() {
    console.log("La tarea asíncrona ha sido completada.");
    callback();
  }, 2000);
}

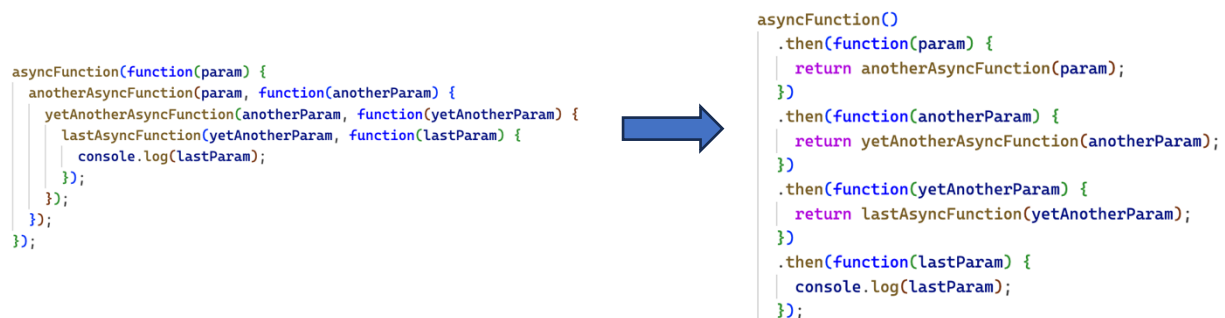
function miCallback() {
  console.log("El callback ha sido llamado!");
}

// Llamamos a la función con el callback
realizarTareaAsincrona(miCallback);
console.log('Este es el primer log que aparece en consola ... ')
```

- **Promesas:** Una promesa representa un valor que puede estar disponible ahora, en el futuro o nunca. En lugar de utilizar *callbacks* directamente, las promesas proporcionan una interfaz más clara y encadenable para trabajar con operaciones asíncronas. Una promesa puede estar en uno de estos tres estados:
 1. Pendiente: La promesa se ha creado, pero aún no se ha cumplido o rechazado.
 2. Completada: La promesa se ha cumplido, lo que significa que la operación asíncrona se ha completado con éxito y devuelve un valor.
 3. Rechazada: La promesa se ha rechazado, lo que indica que la operación asíncrona ha fallado y devuelve un motivo (error).

```
const miPromesa = new Promise((resolve, reject) => {  
  // Código para realizar la operación asíncrona  
  // Si la operación es exitosa, llamamos a "resolve" con el resultado  
  // Si la operación falla, llamamos a "reject" con el motivo del error  
});
```

Esta estructura nos permite salir del “*callback hell*”:



The diagram illustrates the transition from a nested callback structure to a promise chain. On the left, a function `asyncFunction` calls `anotherAsyncFunction`, which calls `yetAnotherAsyncFunction`, which in turn calls `lastAsyncFunction`. Each function takes a parameter and passes it to the next function in the chain. On the right, the same logic is shown using promises. The `asyncFunction` calls `.then` on `anotherAsyncFunction`, which calls `.then` on `yetAnotherAsyncFunction`, which calls `.then` on `lastAsyncFunction`. The `.then` methods return the next function in the chain, creating a linear sequence of calls. A blue arrow points from the nested callback version to the promise chain version.

```
asyncFunction(function(param) {  
  anotherAsyncFunction(param, function(anotherParam) {  
    yetAnotherAsyncFunction(anotherParam, function(yetAnotherParam) {  
      lastAsyncFunction(yetAnotherParam, function(lastParam) {  
        console.log(lastParam);  
      });  
    });  
  });  
});
```

```
asyncFunction()  
  .then(function(param) {  
    return anotherAsyncFunction(param);  
  })  
  .then(function(anotherParam) {  
    return yetAnotherAsyncFunction(anotherParam);  
  })  
  .then(function(yetAnotherParam) {  
    return lastAsyncFunction(yetAnotherParam);  
  })  
  .then(function(lastParam) {  
    console.log(lastParam);  
  });
```

Node.js

Entorno de tiempo de ejecución (*runtime*) de código abierto y multiplataforma que permite ejecutar JavaScript fuera del navegador. A diferencia de JavaScript, que se ejecuta principalmente en navegadores web para interactuar con la página y el contenido del usuario, Node.js permite ejecutar JavaScript en el servidor, lo que lo convierte en una tecnología ampliamente utilizada en el desarrollo de aplicaciones web y servidores.

- Construido sobre el motor de JavaScript V8.
- Permite realizar operaciones I/O de forma asíncrona (*non-blocking*) gracias al *event loop*.
- Incluye una librería estándar (*standard library*) con muchos módulos que permiten realizar operaciones I/O.
- Tiene soporte para *networking* incluyendo el módulo `http`.
- Plataforma de bajo nivel sobre la cual han surgido múltiples *frameworks web*.

HTTP Stateless: No tiene estado, y por ende no guarda información. Las peticiones funcionan de manera independiente.

Cookies:

Pequeños archivos de texto que los sitios web colocan en el navegador del usuario cuando este visita una página web.

Estos archivos se utilizan para almacenar información específica sobre la interacción del usuario con el sitio web y se envían de vuelta al servidor con cada solicitud que realiza el navegador.

Aseguramiento de calidad (QA)

¿Qué podemos hacer?

- | | | |
|--|--|--|
| <ul style="list-style-type: none">• Minimizar errores en el software que construimos.• Verificar que el software funcione.• Verificar que el software es mantenible. |  | <ul style="list-style-type: none">• Análisis estático de código.• <i>Code Review</i>• <i>Testing</i> |
|--|--|--|

- Análisis estático de código: Proceso automatizado en el que se examina el código fuente del programa sin ejecutarlo. Se utiliza una herramienta de análisis estático para buscar problemas y violaciones de reglas de codificación específicas. Estas herramientas analizan el código en busca de errores sintácticos, problemas de estilo, malas prácticas y posibles vulnerabilidades de seguridad.
- Code review: Proceso manual en el que otro desarrollador revisa el código escrito por un compañero antes de ser “mergeado” en el repositorio principal. El revisor examina el código en busca de errores, inconsistencias, problemas de diseño y oportunidades de mejora.
- Testing: Proceso fundamental en el desarrollo de software que consiste en verificar y validar que el software funcione correctamente y cumpla con los requisitos establecidos. El objetivo principal del *testing* es identificar y corregir errores (bugs) en el código, asegurando que el software sea confiable, seguro, y que funcione de acuerdo con las expectativas del usuario.

API (*Application Programming Interface*)

Es una **interfaz** o intermediario entre una **aplicación** o software y quien desee **interactuar** con la aplicación.

- Se dice que una aplicación “**expone**” funcionalidades mediante una API.
- Quien desee acceder a estas funcionalidades tiene que “**consumir**” la API.

API Web

Es una **API que se accede utilizando** la infraestructura Web, es decir, el **protocolo HTTP**.

- En su forma más genérica, el cliente hace un request y recibe **datos como respuesta**.
- No necesariamente un documento HTML.

Arquitectura RPC (*Remote Procedure Call*)

Paradigma de programación que permite que dos procesos en diferentes computadoras se comuniquen entre sí como si estuvieran en la misma computadora. Esto se logra mediante la encapsulación de una llamada a función en un mensaje que se envía a través de la red.

El proceso cliente envía una solicitud al proceso servidor, que luego ejecuta la función solicitada y devuelve los resultados al proceso cliente. La comunicación entre el cliente y el servidor se realiza a través de un protocolo, que define el formato de los mensajes que se intercambian.

La arquitectura RPC es una herramienta poderosa que se puede utilizar para construir aplicaciones distribuidas. Se utiliza en una variedad de aplicaciones, incluyendo sistemas de archivos distribuidos, bases de datos distribuidas y aplicaciones de comercio electrónico.

Arquitectura REST (*REpresentational State Transfer*)

Estilo arquitectónico para el diseño de sistemas de comunicación en red, especialmente en el contexto de aplicaciones web. **REST** define un conjunto de principios y restricciones que se centran en la simplicidad, escalabilidad, independencia del estado y el uso adecuado de los verbos HTTP.

Un servicio web que sigue los principios de la arquitectura REST se denomina **RESTful**. Un servicio web RESTful es aquel que implementa una API web que cumple con las restricciones y principios de REST.

DOM (*Document Object Model*)

Representa la página web como un árbol de objetos, donde cada elemento HTML se convierte en un nodo en ese árbol, y cada nodo puede ser accedido y modificado mediante código JavaScript u otros lenguajes de programación.

El objeto **document** es nuestro **punto de entrada**

- **document.querySelector**: Devuelve el primer elemento que coincide con el selector CSS especificado.

```
// Selecciona el primer elemento con la clase "mi-clase"
const elemento = document.querySelector('.mi-clase');
```

- **document.querySelectorAll**: Devuelve una lista (*NodeList*) de todos los elementos que coinciden con el selector CSS especificado.

```
// Selecciona todos los elementos <p> en el documento
const elementos = document.querySelectorAll('p');
```

- **document.getElementById**: Permite seleccionar un único elemento por su atributo "id". El valor del "id" debe ser único en todo el documento, y este método devuelve el elemento correspondiente.

```
// Selecciona el elemento con el ID "mi-elemento"
const elemento = document.getElementById('mi-elemento');
```

- **document.getElementsByTagName**: Devuelve una lista (*HTMLCollection*) de todos los elementos que tienen la etiqueta especificada. Puede seleccionar múltiples elementos con este método.

```
// Selecciona todos los elementos <div> en el documento
const elementos = document.getElementsByTagName('div');
```

JQuery

Librería de JavaScript de código abierto que simplifica la interacción con el DOM y proporciona un conjunto de utilidades para realizar tareas comunes en el desarrollo web. Provee una interfaz simple, y es por lejos la librería más popular de JavaScript.

```
<button id="miBoton">Haz clic</button>
<div id="mensaje"></div>

$(document).ready(function() {
  $('#miBoton').click(function() {
    $('#mensaje').text('Se hizo clic en el botón!');
  });
});
```

AJAX (Asynchronous JavaScript and XML)

Técnica de programación que permite realizar solicitudes y recibir respuestas del servidor de manera asíncrona, sin tener que recargar toda la página web. El **DOM es actualizado** cuando llega la respuesta, la aplicación **NO se bloquea** y se siente **más responsiva**.

```
const xhr = new XMLHttpRequest();
xhr.open('GET', url);
xhr.onload = function() {
  // Code to execute when response is available
};
xhr.send();
```

React

- Funcionamiento de `useState` en el ejemplo:

- **count**: será la variable que representa el estado actual del componente.
- **setCount**: Es una función que se utiliza para actualizar el valor de `count`. Cuando `setCount` se llama con un nuevo valor, React re-renderiza el componente con el nuevo estado y actualiza la interfaz de usuario en consecuencia.
- **0**: Será el valor inicial del estado del componente.

```
import React, { useState } from 'react';

function Counter() {
  const [count, setCount] = useState(0);

  const increment = () => {
    setCount(count + 1);
  };

  return (
    <div>
      <p>Contador: {count}</p>
      <button onClick={increment}>Incrementar</button>
    </div>
  );
}
```

- **useEffect**: El primer parámetro que recibe es una función (en el ejemplo es una simple *arrow function* que no recibe parámetros). Esta función se ejecutará después de que el componente se haya renderizado, y **siempre que se actualicen las dependencias** especificadas.

```
useEffect(() => {
  // Código para el efecto secundario
}, [dependencias]);
```

```
useEffect(() => {
  // Code run after every render
});
```

```
useEffect(() => {
  // Code run after mounting
}, []);
```

```
useEffect(() => {
  // Code run after a specific prop or state changes
}, [propOrState]);
```

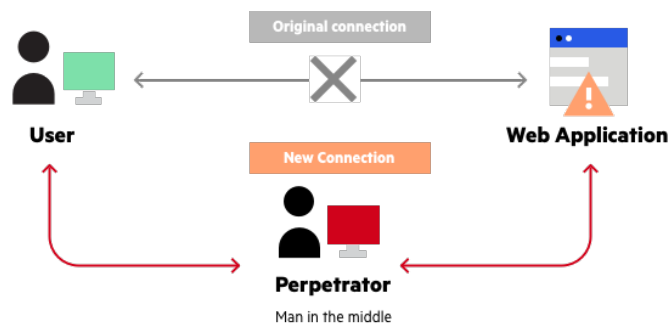
Seguridad Web: Principales Vulnerabilidades

- **Cross-Site Scripting (XSS)**: Se produce cuando un atacante puede inyectar y ejecutar *scripts* maliciosos en páginas web vistas por otros usuarios. Hay 3 tipos:
 - **XSS Reflejado**: El *script* malicioso es **inyectado** en un enlace, formulario o URL y enviado al servidor. El servidor procesa la solicitud y devuelve el contenido, incluyendo el *script*, que se ejecuta en el navegador del usuario.
 - **XSS Persistente**: El *script* malicioso es **almacenado permanentemente en el servidor**, como en una base de datos o en un foro público. Cuando un usuario visita la página que contiene el *script* almacenado, el navegador ejecuta el *script* sin que el usuario lo note.
 - **XSS Basado en DOM**: Ocurre en el lado del cliente, donde el *script* malicioso se ejecuta en el navegador del usuario manipulando el DOM (Document Object Model) de la página. No se comunica con el servidor, por lo que puede ser más difícil de detectar y prevenir.

- **Cross-Site Request Forgery (CSRF):** Ocurre cuando un atacante engaña a un usuario autenticado para que realice una acción involuntaria en una aplicación web. Esto puede ser un link engañoso, o algo por el estilo.
- **SQL injection:** Se produce cuando un atacante puede insertar código SQL malicioso en campos de entrada de una aplicación web, y ese código se ejecuta directamente en la base de datos. Esto puede suceder a través de un input mal programado.

```
<input type="text" name="name" /> "SELECT * FROM users WHERE name = ' " + name + "';"  
  
name = "'; DROP TABLE users;"
```

- **Denial of Service (DoS):** Tipo de ataque cibernético diseñado para impedir que un servicio, recurso o sistema en línea esté disponible y responda a las solicitudes legítimas de los usuarios. El objetivo de un ataque DoS es **abrumar los recursos de un sistema para que no pueda atender las solicitudes legítimas** ("spamerlo"), lo que provoca una denegación del servicio.
 - **Sobrecarga de recursos:** Se enfoca en agotar los recursos del sistema objetivo, como ancho de banda, memoria, capacidad de procesamiento o capacidad de almacenamiento. El atacante envía una gran cantidad de tráfico malicioso o solicitudes maliciosas al sistema con el objetivo de saturar sus recursos y hacer que no pueda responder a las solicitudes legítimas.
 - **Sobrecarga de conexiones:** El objetivo es agotar el número máximo de conexiones que el sistema puede manejar simultáneamente.
- **Man in the middle:** Ataque de seguridad en el que un atacante intercepta y manipula la comunicación entre dos partes que creen estar comunicándose directamente entre sí. El atacante se sitúa entre el cliente y el servidor (o entre dos usuarios), y puede leer, modificar o incluso suplantar los mensajes enviados entre ellos sin que ninguna de las partes sea consciente de la presencia del atacante.



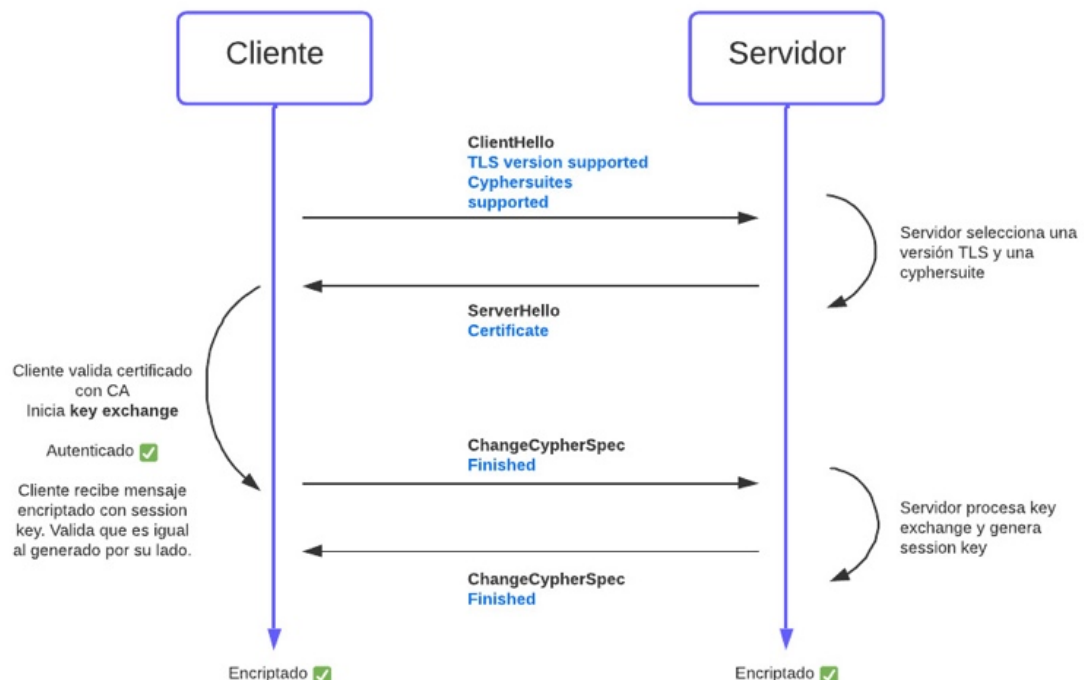
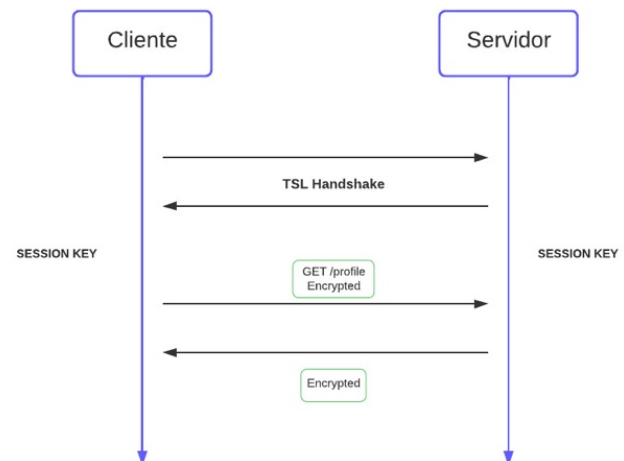
¿Por qué HTTP NO es seguro?

Porque es un protocolo de comunicación que transmite datos en texto claro, lo que significa que la información se envía **sin cifrar**. Esto hace que los datos transmitidos a través de HTTP sean vulnerables a ser interceptados y leídos por cualquier persona que tenga acceso a la red entre el cliente (navegador) y el servidor.

HTTPS (Hypertext Transfer Protocol Secure)

Versión segura del protocolo HTTP utilizado para la transferencia de datos entre un navegador web y un servidor. HTTPS utiliza el **cifrado de extremo a extremo** mediante **TLS** para proteger la privacidad e integridad de los datos durante su transmisión.

- **TLS (Transport Layer Security)**: Protocolo de intercambio de información seguro entre dos partes. Provee 3 cosas:
 - Autenticación
 - Encriptación
 - Integridad
- **Certificado TLS (SSL)**: Servidor debe obtener un certificado (archivo) y guardarlo. Entidad que entrega certificados: **Certificate Authority**. Entidad **certifica X dominio es dueño de Y public key**.



CORS (*Cross-Origin Resource Sharing*)

Mecanismo de seguridad implementado en los navegadores web para controlar las solicitudes de recursos (como datos, fuentes, imágenes, scripts, etc.) realizadas por un cliente web desde un dominio (origen) hacia otro dominio diferente. Específicamente, CORS se utiliza para permitir o restringir solicitudes entre dominios con el fin de prevenir ataques de seguridad, como el acceso no autorizado a recursos de otros sitios web.

- **Same origin policy:** La política de mismo origen establece que un script en un sitio web solo puede acceder a recursos del mismo origen que el sitio web que lo contiene. Esto significa que los recursos, como cookies, datos de sesión, almacenamiento local y recursos de otras páginas, solo pueden ser accedidos por scripts en el mismo dominio y puerto del sitio web que los originó.
- **Cross origin resource sharing:** Mecanismo que permite que un servidor especifique qué dominios (*origins*) tienen permiso para acceder a los recursos (como datos, fuentes, imágenes, scripts, etc.) que se encuentran en él. Está basado en la utilización de **header HTTP**, que permiten este acceso.
2 escenarios:
 - **Requests simples:** Son solicitudes HTTP **GET** o **POST** que solo incluyen encabezados permitidos y no incluyen otros encabezados personalizados o credenciales. Estas solicitudes no requieren una pre-comprobación (*preflight*) CORS y son más simples de manejar.
 - **Requests preflight:** Son solicitudes HTTP que incluyen encabezados personalizados o usan métodos HTTP no seguros (como **PUT**, **DELETE**, etc.) o envían credenciales (como *cookies* de autenticación). Antes de realizar la solicitud real, el navegador primero envía una solicitud de pre-comprobación CORS (**OPTIONS**) al servidor para verificar si la solicitud real es permitida.
- **Access-Control-Allow-Origin:** Indica qué dominios pueden acceder a los recursos. Si se establece en "*", permite a cualquier origen acceder. Si se establece en un dominio específico, solo ese dominio tiene acceso.
- **Access-Control-Allow-Methods:** Enumera los métodos HTTP permitidos para la solicitud CORS.
- **Access-Control-Allow-Headers:** Enumera los encabezados HTTP permitidos para la solicitud CORS.

Dominio

Dirección única y legible por humanos que identifica un conjunto de recursos relacionados en Internet.

DNS (*Domain Name System*)

Sistema utilizado en Internet para traducir nombres de dominio legibles por humanos en direcciones IP numéricas que identifican de manera única los dispositivos y recursos en la red.