

INFORME DE PRÁCTICA

INFORMACIÓN BÁSICA					
ASIGNATURA:	Programación web 2 - LAB A				
TÍTULO DE LA PRÁCTICA:	Laboratorio 4 -Ajax				
NÚMERO DE PRÁCTICA:	01	AÑO LECTIVO:	2025 - A	NRO. SEMESTRE:	III
FECHA DE PRESENTACIÓN	05/05/2025	HORA DE PRESENTACIÓN	11:59:00 PM		
INTEGRANTE(s): <ul style="list-style-type: none">Camargo Hilachoque Romina Giuliana				NOTA:	
DOCENTE:Carlo José Luis Corrales Delgado					

DESARROLLO
<p>A) Introducción:</p> <p>Este informe presenta el desarrollo de una serie de ejercicios prácticos enfocados en el uso de AJAX para la comunicación asincrónica entre cliente y servidor. El primer bloque se basa en ejemplos adaptados del sitio W3Schools, donde se aplican conceptos esenciales como XMLHttpRequest,.responseText, responseXML, cabeceras HTTP y conexión con PHP y bases de datos.</p> <p>En el segundo bloque, se implementaron ocho ejercicios que combinan AJAX con Google Charts, utilizando el archivo data.json para visualizar datos de casos confirmados de COVID-19 por región. Cada ejercicio resuelve un problema específico de análisis y se ejecuta en un servidor local con Python.</p> <p>El código completo de este trabajo se encuentra disponible en el repositorio de GitHub:</p> <p>https://github.com/romich1307/PWEB2-LAB_A/tree/main/LAB4</p> <p>Los primeros <i>commits</i> fueron realizados el 1 de mayo de 2025, marcando el inicio del desarrollo.</p>

B) Actividades

Como parte del desarrollo del laboratorio, se realizaron una serie de actividades prácticas destinadas a consolidar los conocimientos adquiridos sobre programación backend con Node.js y comunicación asíncrona entre cliente y servidor. A continuación, se detalla cada etapa del proceso:

1. Configuración inicial del entorno

Antes de iniciar la implementación, se consultaron las diapositivas teóricas proporcionadas por el docente, a fin de seguir una metodología estructurada. Se procedió a instalar Node.js, un entorno de ejecución para JavaScript del lado del servidor. Posteriormente, se creó un nuevo proyecto Node.js. Para ello, se utilizó el siguiente procedimiento:

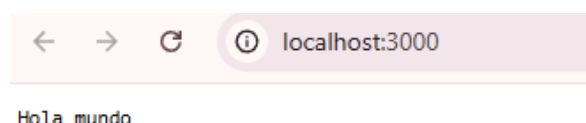
2. Inicialización del repositorio Git: *git init*

- I. Este comando permitió iniciar el control de versiones del proyecto desde sus primeras líneas de código, lo cual favorece el seguimiento del desarrollo y facilita la colaboración futura.
- II. Configuración de archivos ignorados:
Se creó un archivo .gitignore para excluir del repositorio aquellos archivos que no deben ser versionados (por ejemplo, node_modules/).

3. Inicialización del proyecto Node: *npm init -y*

- III. Este comando generó automáticamente el archivo package.json con los parámetros por defecto, permitiendo la gestión de dependencias y scripts del proyecto. Creación del archivo base del servidor (index.js): Se escribió el siguiente código para levantar un servidor HTTP básico:

```
const http = require('http');
const server = http.createServer((request, response) => {
  console.log(request.url);
  response.end('Hola mundo');
});
server.listen(3000);
console.log("Escuchando en: http://localhost:3000")
```



- IV. Este fragmento establece un servidor que responde con el mensaje “Hola mundo” a cualquier solicitud recibida. Al ejecutarlo con `node index.js`, se observó que el mensaje era mostrado correctamente en el navegador, y que cada petición generaba una salida en consola indicando la URL solicitada.

4. Observación sobre el encabezado “Content-Type”

Durante esta etapa se notó que el servidor no especifica explícitamente el tipo de contenido en su respuesta. Por defecto, el navegador asume que se trata de texto plano (`text/plain`). Si se desea enviar contenido en otro formato (por ejemplo, `text/html` o `application/json`), es necesario añadir un encabezado HTTP:

```
response.setHeader('Content-Type', 'text/plain');
```

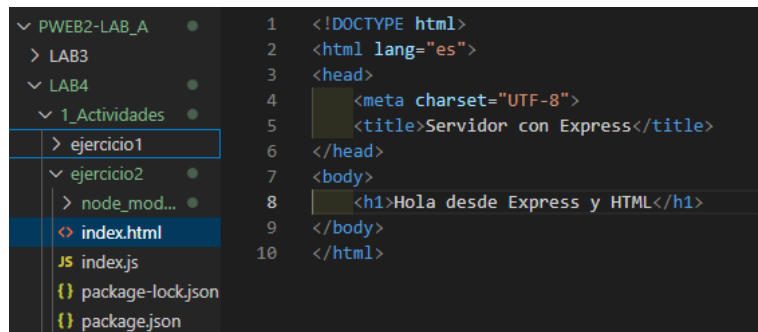
Aunque en este primer ejemplo el comportamiento por omisión no causa errores, en aplicaciones más complejas es fundamental declarar correctamente el `Content-Type` para asegurar la correcta interpretación de la respuesta por parte del cliente.

5. Introducción de Express y envío de archivos HTML

Para facilitar el desarrollo del servidor y mejorar su estructura, se introdujo el framework Express, el cual proporciona una interfaz más simple y elegante para definir rutas y gestionar peticiones. Su instalación se realizó mediante el gestor de paquetes npm:

```
npm install express
```

A continuación, se reescribió el archivo `index.js` para incluir Express y servir un archivo HTML como respuesta:



```
1 <!DOCTYPE html>
2 <html lang="es">
3 <head>
4   <meta charset="UTF-8">
5   <title>Servidor con Express</title>
6 </head>
7 <body>
8   <h1>Hola desde Express y HTML</h1>
9 </body>
10 </html>
```



Hola desde Express y HTML

Este nuevo servidor responde a una petición GET en la ruta raíz (/) devolviendo el archivo index.html, el cual debe estar ubicado en el directorio principal del proyecto. Se destaca que en este caso, la ruta solicitada no contiene explícitamente el nombre del archivo HTML, lo cual es gestionado internamente por el servidor.

C) Ejercicios Resueltos

1. Cree una aplicación web que ejecute javascript en el cliente (dentro de la carpeta pub) y nodejs en el servidor.

```
✓ Ejercicio1 ●
  > node_mod... ●
  ✓ pub
    JS app.js
    <> index.html
    {} package-lock.json
    {} package.json
    JS server.js
```

```
const path = require('path');
const express = require('express');
const app = express();

app.use(express.static('pub'));

app.listen(3000, () => {
  console.log("Escuchando en: http://localhost:3000");
});

app.get('/', (request, response) => {
  response.sendFile(path.resolve(__dirname, 'index.html'));
});
```

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Ejercicio1</title>
</head>
<body>
  <h1>Este es el ejercicio 1</h1>
  <button onclick="saludar()">Saludar</button>
  <script src="app.js"></script>
</body>
</html>
```

```
function saludar() {
  alert("Hola desde el cliente");
}
```

2. Cree una aplicación web que realice una petición ajax en el lado del cliente y responda usando nodejs en el lado del servidor.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Poema con AJAX</title>
</head>
<body>
  <h1>Poema</h1>
  <button onclick="recitar()">Recitar poemas</button>
  <div id="poema"></div>
  <script src="app.js"></script>
</body>
</html>
```

```
function recitar() {
  const url = 'http://localhost:3000/recitar';
  fetch(url)
    .then(response => response.json())
    .then(data => {
      document.querySelector("#poema").innerHTML = data.text;
    });
}
```

```
const fs = require('fs');
const path = require('path');
const express = require('express');
const app = express();

app.use(express.static('pub'));

app.listen(3000, () => {
  console.log("Escuchando en: http://localhost:3000");
});

app.get('/', (request, response) => {
  response.sendFile(path.resolve(__dirname, 'index.html'));
});

app.get('/recitar', (request, response) => {
  fs.readFile(path.resolve(__dirname, 'priv/poema.txt'), 'utf8', (err, data) => {
    if (err) {
      console.error(err);
      response.status(500).json({ error: 'message' });
      return;
    }
    response.json({ text: data.replace(/\n/g, '<br>') });
  });
});
```

¿En qué lugar debería estar el archivo poema.txt?

El archivo poema.txt debe colocarse en una carpeta llamada priv que esté en el mismo nivel que server.js.

¿Entiende la expresión regular en el código y se da cuenta de para qué es útil?

La expresión regular `\n` busca los saltos de línea en el texto (es decir, cuando pasas de una línea a otra en el archivo).

- `\n`: representa un salto de línea.
`/\n/g`: es una expresión regular que busca todas las apariciones de saltos de línea (gracias al modificador `g` de "global").
- `.replace(/\n/g, '
')`: reemplaza todos los saltos de línea por etiquetas `
`, que es el equivalente en HTML para un salto de línea.

¿Por qué es útil?

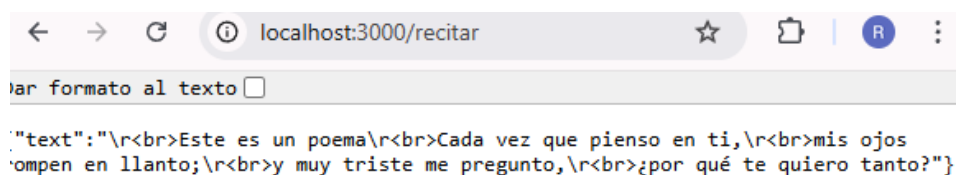
Porque si solo mostraras el texto tal como está en el archivo, el HTML ignoraría los saltos de línea y todo aparecería en una sola línea. Al reemplazar `\n` por `
`, haces que el poema se vea como en el archivo, línea por línea.

Note que la respuesta del servidor está en formato JSON, ¿Habría alguna forma de verla directamente?

Sí, es posible ver directamente la respuesta del servidor en formato JSON.

Para hacerlo, basta con ingresar en el navegador a la dirección:

<http://localhost:3000/recitar>



3. Cree una aplicación que haga peticiones AJAX usando POST a un servidor NodeJS

```
const fs = require('fs');
const path = require('path');
const express = require('express');
const bp = require('body-parser');
const MarkdownIt = require('markdown-it');
const md = new MarkdownIt();

const app = express();

app.use(express.static('pub'));
app.use(bp.json());
app.use(bp.urlencoded({ extended: true }));

app.listen(3000, () => {
  console.log("Escuchando en: http://localhost:3000");
});

app.get('/', (request, response) => {
  response.sendFile(path.resolve(__dirname, 'pub/index.html'));
});

app.post('/', (request, response) => {
  console.log(request.body);
  let markDownText = request.body.text;
  console.log(markDownText);

  let htmlText = md.render(markDownText);

  response.setHeader('Content-Type', 'application/json');
  response.end(JSON.stringify({
    text: htmlText
  }));
});
```

```
function recitar(markupText) {
  const url = 'http://localhost:3000/';
  const data = {
    text: markupText
  };

  const request = {
    method: 'POST',
    headers: {
      'Content-Type': 'application/json',
    },
    body: JSON.stringify(data),
  };

  fetch(url, request)
    .then(response => response.json())
    .then(data => {
      document.querySelector("#htmlCode").innerHTML = data.text;
    })
    .catch(error => {
      console.error('Error:', error);
    });
}

document.addEventListener('DOMContentLoaded', function () {
  const text = document.querySelector('#markupText');
  document.querySelector('#markupForm').onsubmit = () => {
    recitar(text.value);
    return false;
  };
});
```

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Convertemos Markdown a HTML</title>
</head>
<body>
  <h1>Markdown a HTML</h1>
  <form id="markupForm">
    <textarea id="markupText" rows="10" cols="50" placeholder="Escribe tu texto en Markdown aquí..."></textarea><br><br>
    <button type="submit">Convertir</button>
  </form>
  <h2>Resultado en HTML:</h2>
  <div id="htmlCode"></div>
  <script src="app.js"></script>
</body>
</html>
```

Markdown a HTML

```
# Título principal

Este es un párrafo de texto.

*Párrafo en itálica*
**Párrafo en negrita**

1. Primera lista numerada
2. Segunda lista numerada

- Primera lista de viñetas
- Segunda lista de viñetas

[Enlace](https://www.google.com)
```

Convertir

Resultado en HTML:

Título principal

Este es un párrafo de texto.

Párrafo en itálica **Párrafo en negrita**

1. Primera lista numerada
 2. Segunda lista numerada
- Primera lista de viñetas
 - Segunda lista de viñetas

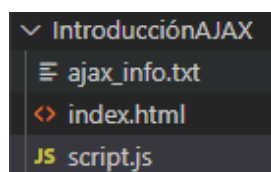
D) Ejercicios AJAX de W3Schools

Con el objetivo de reforzar los conocimientos fundamentales sobre AJAX, se realizó una revisión práctica de los ejemplos provistos por el sitio educativo W3Schools, replicando y adaptando cada uno en un servidor web local. La finalidad de esta actividad fue comprender el comportamiento de las peticiones asíncronas en distintas situaciones, utilizando tecnologías base como XMLHttpRequest, archivos .txt, .xml, .php, y simulaciones de bases de datos.

Ejemplo 1: Introducción a AJAX

Objetivo: Demostrar cómo usar el objeto XMLHttpRequest para cargar contenido externo (archivo .txt) sin recargar la página.

Archivos utilizados:



Funcionamiento:

Al hacer clic en el botón, se ejecuta `loadDoc()`, que usa `XMLHttpRequest` para solicitar `ajax_info.txt` y mostrar su contenido dentro del `<div id="demo">`.

Código principal (script.js):

```
function loadDoc() {  
    const xhttp = new XMLHttpRequest();  
    xhttp.onload = function() {  
        document.getElementById("demo").innerHTML = this.responseText;  
    };  
    xhttp.open("GET", "ajax_info.txt", true);  
    xhttp.send();  
}
```

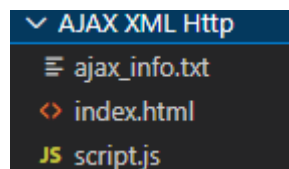
Resultado:

Se reemplaza dinámicamente el contenido del div con el texto cargado del archivo.

The XMLHttpRequest Object AJAX

[Change Content](#)**Ejemplo 2: AJAX con onreadystatechange****Objetivo:**

Ilustrar el uso de `XMLHttpRequest` mediante el evento `onreadystatechange`, controlando el flujo de la solicitud de forma manual.

Archivos utilizados:**Funcionamiento:**

Al hacer clic en el botón, se ejecuta `loadDoc()`, que crea un objeto `XMLHttpRequest` y define una función para el evento `onreadystatechange`. Cuando el estado de la solicitud es 4 (completo) y la respuesta del servidor es satisfactoria (`status == 200`), se inserta el contenido del archivo en el div con `id="demo"`.

Código principal (script.js):

```
function loadDoc() {  
    const xhttp = new XMLHttpRequest();  
    xhttp.onreadystatechange = function() {  
        if (this.readyState == 4 && this.status == 200) {  
            document.getElementById("demo").innerHTML =  
                this.responseText;  
        }  
    };  
    xhttp.open("GET", "ajax_info.txt");  
    xhttp.send();  
}
```

Diferencia respecto al ejemplo anterior:

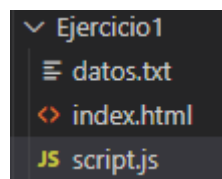
Aquí se utiliza `onreadystatechange` en lugar de `onload`, lo que permite un mayor control sobre los distintos estados de la solicitud.

The XMLHttpRequest Object - onreadystatechange[Change Content](#)

Este es el contenido del archivo AJAX cargado con `onreadystatechange`.

Ejercicio 3.1: Solicitud AJAX con archivo .txt

Objetivo: Realizar una solicitud AJAX sencilla utilizando XMLHttpRequest para cargar y mostrar el contenido de un archivo .txt.

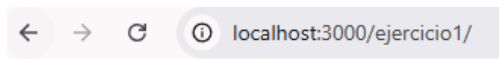
Archivos utilizados:

Funcionamiento:

Al hacer clic en el botón, se ejecuta la función loadDoc() que utiliza XMLHttpRequest para leer el archivo datos.txt y mostrarlo dentro del párrafo con id="demo".

Código principal (script.js):

```
function loadDoc() {  
    const xhttp = new XMLHttpRequest();  
    xhttp.onload = function() {  
        document.getElementById("demo").innerHTML = this.responseText;  
    };  
    xhttp.open("GET", "datos.txt");  
    xhttp.send();  
}
```



The XMLHttpRequest Object

Request data

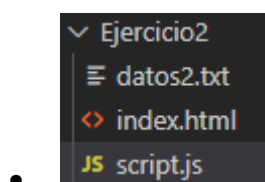
The XMLHttpRequest Object

Request data

Este es un archivo .txt cargado con AJAX desde Node.js.

Ejercicio 3.2: Evitar caché en solicitudes AJAX

Objetivo: Evitar que el navegador almacene en caché el contenido solicitado con XMLHttpRequest, garantizando que siempre se obtenga una versión actualizada del archivo.

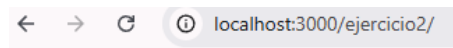
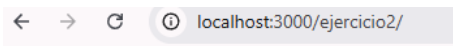
Archivos utilizados:

Funcionamiento:

Al hacer clic en el botón, se ejecuta la función `loadDoc()` que realiza una solicitud GET al archivo `datos2.txt`, agregando un parámetro aleatorio a la URL. Esto evita que el navegador sirva una versión antigua desde la caché.

Código principal (script.js):

```
function loadDoc() {  
  const xhttp = new XMLHttpRequest();  
  xhttp.onload = function() {  
    document.getElementById("demo").innerHTML = this.responseText;  
  };  
  xhttp.open("GET", "datos2.txt?t=" + Math.random());  
  xhttp.send();  
}
```

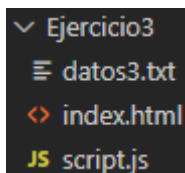
**The XMLHttpRequest Object****The XMLHttpRequest Object**

Este es el segundo ejercicio. No se está usando caché.

Ejercicio 3.3: Solicitud AJAX con parámetros en la URL

Objetivo: Enviar datos al servidor utilizando parámetros en la URL (GET), como si se tratara de un formulario simple.

Archivos utilizados:

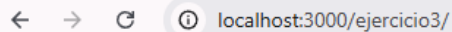
-  Ejercicio3
 - datos3.txt
 - index.html
 - script.js

Funcionamiento:

La función loadDoc() envía una solicitud con parámetros ?fname=Henry&lname=Ford incrustados en la URL. Aunque el archivo de respuesta es siempre el mismo (datos3.txt), esto simula cómo se podrían pasar datos al servidor para personalizar la respuesta.

Código principal (script.js):

```
function loadDoc() {  
  const xhttp = new XMLHttpRequest();  
  xhttp.onload = function() {  
    document.getElementById("demo").innerHTML = this.responseText;  
  };  
  xhttp.open("GET", "datos3.txt?fname=Henry&lname=Ford");  
  xhttp.send();  
}
```

localhost:3000/ejercicio3/**The XMLHttpRequest Object**

Request data

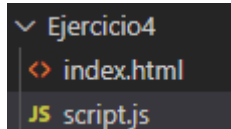
The XMLHttpRequest Object

Request data

Hello Henry Ford

Ejercicio 3.4: Solicitud AJAX con método POST

Objetivo: Realizar una solicitud AJAX utilizando el método POST, simulando el envío de datos al servidor.

Archivos utilizados:Ejercicio4
index.html
script.js

Funcionamiento: Al presionar el botón, se ejecuta la función `loadDoc()` que realiza una solicitud POST a la ruta `/datos4`. El contenido de la respuesta del servidor se inserta dinámicamente dentro del elemento `<p id="demo">`.

Código principal (script.js):

```
function loadDoc() {  
  const xhttp = new XMLHttpRequest();  
  xhttp.onload = function () {  
    document.getElementById("demo").innerHTML = this.responseText;  
  };  
  xhttp.open("POST", "/datos4", true);  
  xhttp.send();  
}
```

← → ↻ ⓘ localhost:3000/ejercicio4/

The XMLHttpRequest Object

Request data

The XMLHttpRequest Object

Request data

This content was requested using the POST method.

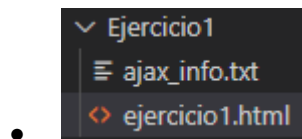
Requested at: 4/5/2025, 1:39:54 p. m.

Para la ejecución de los ejercicios de esta sección, se utilizó un servidor local implementado con Node.js y Express mediante un archivo `server.js`. Gracias a esta configuración, fue posible realizar pruebas completas en un entorno funcional similar a un servidor web real.

```
const express = require('express');  
const app = express();  
const port = 3000;  
  
app.use(express.urlencoded({ extended: true }));  
app.use(express.json());  
  
app.use(express.static('public'));  
  
app.post('/datos4', (req, res) => {  
  const now = new Date().toLocaleString();  
  res.send(`This content was requested using the POST method.<br><br>Requested at: ${now}`);  
});  
  
app.listen(port, () => {  
  console.log(`Servidor en http://localhost:${port}`);  
});
```

Ejercicio 4.1: Uso de responseText para cargar contenido

Objetivo: Demostrar cómo utilizar la propiedad responseText del objeto XMLHttpRequest para obtener y mostrar el contenido de un archivo .txt.

Archivos utilizados:

Funcionamiento: Al hacer clic en el botón, se ejecuta la función loadText(), que realiza una solicitud GET al archivo ajax_info.txt. El contenido del archivo es cargado en el navegador y mostrado dentro del párrafo con id="demo" usando this.responseText.

Código relevante:

```
<script>
function loadText() {
  const xhttp = new XMLHttpRequest();
  xhttp.onload = function() {
    document.getElementById("demo").innerHTML = this.responseText;
  };
  xhttp.open("GET", "ajax_info.txt");
  xhttp.send();
}
</script>
```

Usando responseText

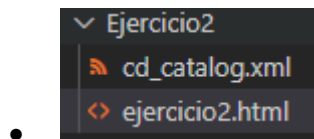
Usando responseText

AJAX

Ejercicio 4.2: Procesar XML con responseXML

Objetivo: Mostrar cómo acceder y recorrer un documento XML cargado vía AJAX utilizando la propiedad responseXML del objeto XMLHttpRequest.

Archivos utilizados:



Funcionamiento: Al hacer clic en el botón, se carga el archivo cd_catalog.xml. El contenido XML se interpreta como un objeto DOM mediante responseXML. Luego, se extraen todos los elementos <ARTIST> y se muestran sus nombres en el elemento con id="demo".

Fragmento de código relevante:

```
const xhttp = new XMLHttpRequest();
xhttp.onload = function() {
  const xmlDoc = xhttp.responseXML;
  const x = xmlDoc.getElementsByTagName("ARTIST");
  let txt = "";
  for (let i = 0; i < x.length; i++) {
    txt += x[i].childNodes[0].nodeValue + "<br>";
  }
  document.getElementById("demo").innerHTML = txt;
};
xhttp.open("GET", "cd_catalog.xml");
xhttp.send();
```

Usando responseXML

Cargar artistas

Usando responseXML

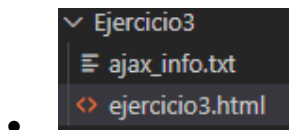
Cargar artistas

Bob Dylan
Bonnie Tyler

Ejercicio 4.3: Obtener cabeceras con `getAllResponseHeaders()`

Objetivo: Visualizar todas las cabeceras HTTP recibidas en una respuesta AJAX usando `getAllResponseHeaders()`.

Archivos utilizados:



Funcionamiento: Al hacer clic en el botón, se realiza una solicitud AJAX al archivo `ajax_info.txt`. En lugar de mostrar su contenido, se obtiene y muestra en pantalla el conjunto completo de cabeceras HTTP que devuelve el servidor.

Código principal:

```
function loadHeaders() {  
  const xhttp = new XMLHttpRequest();  
  xhttp.onload = function() {  
    document.getElementById("demo").innerHTML = this.getAllResponseHeaders();  
  };  
  xhttp.open("GET", "ajax_info.txt");  
  xhttp.send();  
}
```

`getAllResponseHeaders`

Ver headers

`getAllResponseHeaders`

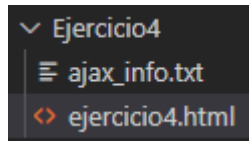
Ver headers

```
accept-ranges: bytes  
access-control-allow-credentials: true  
cache-control: public, max-age=0  
connection: keep-alive  
content-length: 4  
content-type: text/plain; charset=UTF-8  
date: Sun, 04 May 2025 18:50:10 GMT  
etag: W/"4-19699b3b87a"  
keep-alive: timeout=5  
last-modified: Sun, 04 May 2025 05:10:14 GMT  
vary: Origin
```

Ejercicio 4.4: Obtener cabecera específica con `getResponseHeader()`

Objetivo: Demostrar cómo obtener una cabecera HTTP específica (en este caso, Last-Modified) a partir de una respuesta AJAX.

Archivos utilizados:



Funcionamiento:

Al cargar la página, se realiza automáticamente una solicitud al archivo `ajax_info.txt`. Luego, usando `getResponseHeader("Last-Modified")`, se obtiene la fecha y hora en que el archivo fue modificado por última vez y se muestra en pantalla.

Código clave:

```
const xhttp = new XMLHttpRequest();
xhttp.onload = function() {
    const lastModified = this.getResponseHeader("Last-Modified");
    document.getElementById("demo").innerHTML = lastModified;
};
xhttp.open("GET", "ajax_info.txt", true);
xhttp.send();
```

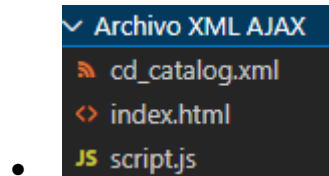
Ejemplo de `getResponseHeader()`

Última modificación del archivo:

Sun, 04 May 2025 05:10:14 GMT

Ejercicio 5: Leer y mostrar datos de un archivo XML

Objetivo: Leer un archivo XML con AJAX, extraer información estructurada (artista y título de CDs), y mostrarla en una tabla HTML con formato.

Archivos utilizados:

Funcionamiento: Al hacer clic en el botón, se ejecuta `loadDoc()`, que carga el archivo `cd_catalog.xml` mediante `XMLHttpRequest`. Luego, se extrae cada elemento `<CD>` del XML y se construye una tabla HTML con dos columnas: Artista y Título. Esta tabla se muestra dentro del elemento con `id="demo"`.

Fragmento clave del código (script.js):

```
function loadDoc() {
  const xhttp = new XMLHttpRequest();
  xhttp.onload = function() {
    myFunction(this);
  };
  xhttp.open("GET", "cd_catalog.xml", true);
  xhttp.send();
}

function myFunction(xml) {
  const xmlDoc = xml.responseXML;
  const x = xmlDoc.getElementsByTagName("CD");
  let table = "<tr><th>Artist</th><th>Title</th></tr>";
  for (let i = 0; i < x.length; i++) {
    table += "<tr><td>" +
      x[i].getElementsByTagName("ARTIST")[0].childNodes[0].nodeValue +
      "</td><td>" +
      x[i].getElementsByTagName("TITLE")[0].childNodes[0].nodeValue +
      "</td></tr>";
  }
  document.getElementById("demo").innerHTML = table;
}
```

The XMLHttpRequest Object

Get my CD collection

The XMLHttpRequest Object

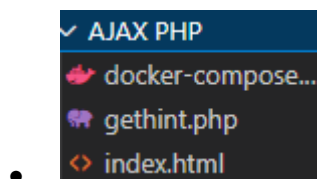
Get my CD collection

Artist	Title
Bob Dylan	Empire Burlesque
Bonnie Tyler	Hide your heart
Dolly Parton	Greatest Hits
Gary Moore	Still got the blues
Eros Ramazzotti	Eros
Bee Gees	One night only
Dr Hook	Sylvias Mother
Rod Stewart	Maggie May
Andrea Bocelli	Romanza
Percy Sledge	When a man loves a woman
Savage Rose	Black angel
Many	1999 Grammy Nominees
Kenny Rogers	For the good times
Will Smith	Big Willie style
Van Morrison	Tupelo Honey
Jorn Hoel	Soulsville
Cat Stevens	The very best of
Sam Brown	Stop
T'Pau	Bridge of Spies
Tina Turner	Private Dancer
Kim Larsen	Midt om natten
Luciano Pavarotti	Pavarotti Gala Concert

Ejercicio 6: Autocompletado con AJAX y PHP

Objetivo: Implementar un sistema básico de sugerencias (autocompletado) usando AJAX para enviar el texto ingresado por el usuario a un script PHP, que devuelve posibles coincidencias.

Archivos utilizados:



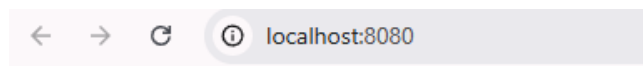
Funcionamiento: A medida que el usuario escribe en el campo de texto, se llama a `showHint()` con el valor ingresado. Esta función hace una solicitud AJAX al archivo `gethint.php` enviando el texto como parámetro `q`. El servidor busca coincidencias dentro de un arreglo de nombres y devuelve una lista de sugerencias, que se muestran bajo el campo de entrada.

Fragmento clave del script AJAX:

```
function showHint(str) {  
    if (str.length === 0) {  
        document.getElementById("txtHint").innerHTML = "";  
        return;  
    } else {  
        const xmlhttp = new XMLHttpRequest();  
        xmlhttp.onload = function () {  
            document.getElementById("txtHint").innerHTML = this.responseText;  
        };  
        xmlhttp.open("GET", "gethint.php?q=" + encodeURIComponent(str), true);  
        xmlhttp.send();  
    }  
}
```

Lógica del servidor (gethint.php):

```
$q = $_REQUEST["q"];  
$hint = "";  
  
// Buscar coincidencias  
if ($q !== "") {  
    $q = strtolower($q);  
    $len = strlen($q);  
    foreach($a as $name) {  
        if (strpos($q, substr($name, 0, $len)) !== false) {  
            if ($hint === "") {  
                $hint = $name;  
            } else {  
                $hint .= ", $name";  
            }  
        }  
    }  
}
```



Empieza a escribir un nombre en el campo de entrada:

Nombre:

Sugerencias: Nina

Para la ejecución del ejemplo con PHP, se utilizó un contenedor Docker configurado mediante Docker Compose, lo que facilitó el despliegue rápido de un servidor PHP sin necesidad de instalaciones adicionales en el sistema anfitrión. El proyecto se ejecutó localmente con el siguiente comando:

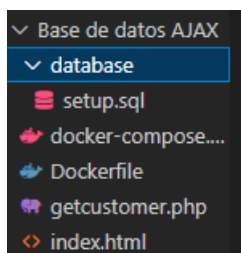
docker-compose up --build

Este comando construye la imagen definida en el archivo docker-compose.yml y lanza el contenedor, permitiendo acceder a index.html y al script gethint.php desde el navegador.

Ejercicio 7: Consulta a base de datos con AJAX + PHP + MySQL

Objetivo: Mostrar cómo se puede usar AJAX para realizar una consulta a una base de datos MySQL y obtener la información de un cliente sin recargar la página.

Archivos utilizados:



Funcionamiento

1. El usuario selecciona un cliente del menú desplegable en index.html.
2. Se llama a showCustomer(), que envía una solicitud AJAX con el ID del cliente (q) a getcustomer.php.
3. El script PHP consulta la base de datos MySQL y devuelve los datos del cliente en una tabla.
4. El resultado se muestra dinámicamente en la página, dentro del div#txtHint.

Código AJAX (resumen de index.html)

```
function showCustomer(str) {
    if (str == "") {
        document.getElementById("txtHint").innerHTML = "";
        return;
    }
    const xhttp = new XMLHttpRequest();
    xhttp.onload = function() {
        document.getElementById("txtHint").innerHTML = this.responseText;
    }
    xhttp.open("GET", "getcustomer.php?q=" + str);
    xhttp.send();
}
```

Script PHP (resumen de getcustomer.php)

```
$conn = new mysqli($servername, $username, $password, $dbname);

if ($conn->connect_error) {
    die("Connection failed: " . $conn->connect_error);
}

$q = $_REQUEST['q'];

$sql = "SELECT * FROM customers WHERE customerid = '$q'";
$result = $conn->query($sql);
```

Configuración del entorno (resumen de docker-compose.yml)

- Servicio db: MySQL 8.0 con base de datos customers_db, usando setup.sql como inicialización.
- Servicio web: servidor web PHP que sirve los archivos del proyecto.
- Ambos servicios se comunican por la red mynetwork.

Ejecución del entorno:

docker-compose up --build

The XMLHttpRequest Object

Alfreds Futterkiste

CustomerID	CompanyName	ContactName	Address	City	PostalCode	Country
ALFKI	Alfreds Futterkiste	Maria Anders	Obere Str. 57	Berlin	12209	Germany

CustomerID	CompanyName	ContactName	Address	City	PostalCode	Country
NORTS	North/South	Simon Crowther	South House 300 Queensbridge	London	SW7 1RZ	UK

A través del desarrollo de los ejemplos propuestos por W3Schools, se consolidaron los conocimientos fundamentales sobre el funcionamiento de AJAX, incluyendo el uso de XMLHttpRequest, el procesamiento de respuestas en texto y XML, y la integración con tecnologías del lado del servidor como PHP y MySQL. Además, se puso en práctica el uso de contenedores con Docker y servidores Express en Node.js, garantizando un entorno modular, portable y profesional.

E) Ejercicios AJAX con Google Charts

Ejercicio 1: Listar todas las regiones

Objetivo: Extraer y mostrar en una lista HTML todas las regiones únicas presentes en el archivo data.json, utilizando AJAX con fetch().

Archivos utilizados:

- **1.html:** contiene la estructura básica y el espacio para la lista.
- **ejercicio1.js:** realiza la carga del archivo y muestra las regiones.
- **data.json:** archivo con los datos COVID por región y fecha.

Funcionamiento: El script realiza una solicitud AJAX con fetch('data.json') y convierte la respuesta en un array de objetos. A partir de estos, se utiliza un Set para obtener las regiones sin duplicados y luego se construye dinámicamente una lista en la página.

Fragmento clave (ejercicio1.js):

```
fetch('data.json')
  .then(response => response.json())
  .then(data => {
    const regionesSet = new Set();

    data.forEach(item => {
      regionesSet.add(item.region);
    });

    const lista = document.getElementById('lista-regiones');
    regionesSet.forEach(region => {
      const li = document.createElement('li');
      li.textContent = region;
      lista.appendChild(li);
    });
  })
  .catch(error => {
    console.error('Error al cargar los datos:', error);
  });
```


Listado de Regiones

Amazonas
Ancash
Apurimac
Arequipa
Ayacucho
Cajamarca
Callao
Cusco
Huancavelica
Huanuco
Ica
Junin
La Libertad
Lambayeque
Lima
Loreto
Madre de Dios
Moquegua
Pasco
Piura
Puno
San Martin
Tacna
Tumbes
Ucayali

Ejercicio 2: Mostrar el número total de confirmados por región

Objetivo: Procesar el archivo data.json para calcular y mostrar la suma total de casos confirmados por cada región.

Archivos utilizados:

- **2.html:** estructura HTML con una tabla vacía donde se insertarán los resultados.
- **ejercicio2.js:** contiene la lógica para procesar los datos y generar las filas dinámicamente.
- **data.json:** contiene los datos diarios por región.

Funcionamiento: El script carga los datos desde data.json con `fetch()`. Luego, para cada objeto, accede al array `confirmed` y suma todos los valores de cada entrada. Se agrupan los resultados por región y se genera una tabla con el total de confirmados por cada una.

Fragmento clave (ejercicio2.js):

```
data.forEach(item => {  
  const region = item.region;  
  const confirmados = item.confirmed.reduce(  
    (total, entry) => total + parseInt(entry.value),  
    0  
  );  
  totalesPorRegion[region] = confirmados;  
});
```

Visualización:

Se genera dinámicamente una tabla `<table>` con dos columnas:

- Región
- Total Confirmados

Región	Total Confirmados
Amazonas	2985
Ancash	18753
Apurímac	1740
Arequipa	13817
Ayacucho	2540
Cajamarca	4506
Callao	78099
Cusco	6187
Huancavelica	2332
Huanuco	4716
Ica	11925
Junín	10171
La Libertad	18175
Lambayeque	51206
Lima	626744
Loreto	30242
Madre de Dios	2438
Moquegua	2443
Pasco	2449
Piura	29966
Puno	2078
San Martín	5445
Tacna	2457
Tumbes	7796
Ucayali	14488

Ejercicio 3: Top 10 de regiones con más casos confirmados

Objetivo: Mostrar una tabla con las 10 regiones que acumulan el mayor número de casos confirmados, ordenadas de mayor a menor.

Archivos utilizados:

- 3.html: estructura HTML con tabla y estilos personalizados.
- ejercicio3.js: lógica para calcular totales, ordenar y mostrar los resultados.
- data.json: fuente de datos con los casos diarios por región.

Funcionamiento: El script carga data.json y calcula el total de casos confirmados por región. Luego, utiliza `Object.entries()` para convertir el objeto en pares [región, total], los ordena descendientemente por total de confirmados y toma los primeros 10 con `.slice(0, 10)`. Finalmente, genera dinámicamente una tabla HTML con estos resultados.

Fragmento clave (ejercicio3.js):

```
const regionesOrdenadas = Object.entries(totalesPorRegion)
  .sort((a, b) => b[1] - a[1])
  .slice(0, 10);
```

Visualización:

Una tabla con columnas:

- Región
- Total Confirmados

Las filas corresponden a las 10 regiones con mayor número de casos.

Las 10 Regiones con más Confirmados

Región	Total Confirmados
Lima	626744
Callao	78099
Lambayeque	51206
Loreto	30242
Piura	29966
Ancash	18753
La Libertad	18175
Ucayali	14488
Arequipa	13817
Ica	11925

Ejercicio 4: Visualización en el tiempo para la región de Arequipa

Objetivo: Graficar la evolución diaria de casos confirmados en la región de Arequipa utilizando Google Charts.

Archivos utilizados:

- **4.html:** estructura del documento con un contenedor visual para el gráfico.
- **ejercicio4.js:** código que procesa data.json y construye el gráfico de líneas.
- **data.json:** archivo fuente con los datos por región y por fecha.

Funcionamiento:

Al cargarse la página, Google Charts se inicializa y ejecuta `drawChart()`.

Se filtran los datos correspondientes únicamente a la región Arequipa, y se construye una tabla de datos con pares "Fecha" y "Confirmados".

Con esta información, se genera un gráfico de líneas que muestra la evolución de los casos a lo largo del tiempo.

Fragmento clave (ejercicio4.js):

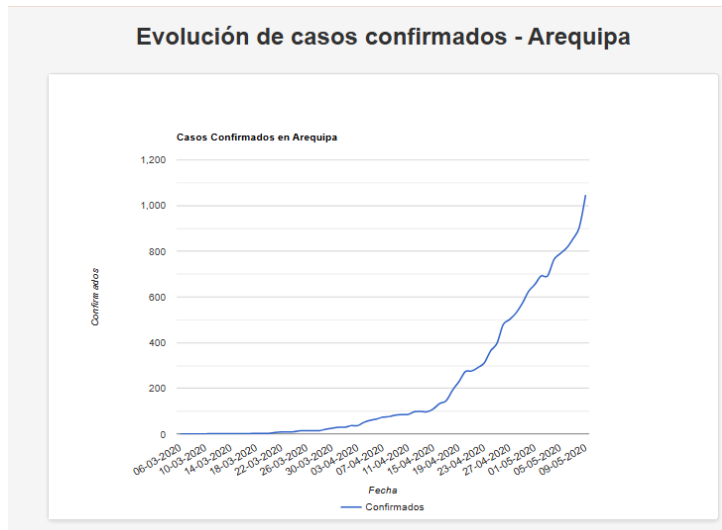
```
const chartData = [{"Fecha", "Confirmados"}]; // Encabezado

arequipaData.confirmed.forEach(entry => {
  chartData.push([entry.date, parseInt(entry.value)]);
});
```

Visualización esperada:

Un gráfico de línea con:

- Eje horizontal (X): Fechas
- Eje vertical (Y): Número de casos confirmados



Ejercicio 5: Gráfico comparativo entre regiones

Objetivo: Comparar visualmente la evolución de casos confirmados entre múltiples regiones (Arequipa, Cusco y Puno) usando un gráfico de líneas.

Archivos utilizados:

- 5.html: estructura visual con contenedor y estilo.
- ejercicio5.js: código que prepara y renderiza la comparación con Google Charts.
- data.json: contiene los datos históricos por región.

Funcionamiento:

1. Se define un conjunto de regiones a comparar: "Arequipa", "Cusco", "Puno".
2. Se obtiene la lista de fechas desde la primera región.
3. Para cada región, se extrae su lista de valores diarios de casos confirmados.
4. Se construye un arreglo de datos con encabezado ["Fecha", "Arequipa", "Cusco", "Puno"].
5. Se utiliza Google Charts para renderizar el gráfico.

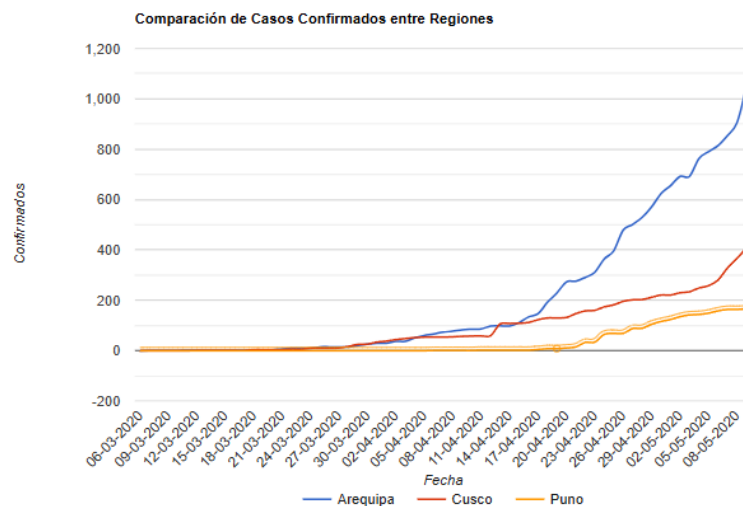
Fragmento clave (ejercicio5.js):

```
const chartData = [{"Fecha", ...regiones}];
fechas.forEach((fecha, index) => {
  const fila = [fecha];
  regiones.forEach(region => {
    fila.push(regionData[region][index]);
  });
  chartData.push(fila);
});
```

Visualización esperada:

- Gráfico de líneas con tres series, una por región.
- Eje X: Fechas
- Eje Y: Número de casos
- Leyenda en la parte inferior

Comparación de Confirmados entre Regiones



Ejercicio 6: Comparación de crecimiento en regiones (excepto Lima y Callao)

Objetivo: Visualizar el crecimiento diario de casos confirmados en todas las regiones excluyendo Lima y Callao, mediante un gráfico de líneas multiserie.

Archivos utilizados:

- 6.html: define el contenedor visual y carga el script al iniciar.
- ejercicio6.js: carga los datos, filtra las regiones y construye el gráfico.
- data.json: fuente de datos con series de confirmados por región.

Funcionamiento:

1. Se excluyen explícitamente las regiones "Lima" y "Callao".
2. Se obtiene la lista de fechas desde una región válida.
3. Se construye una tabla con formato Google Charts: ["Fecha", "Región1", "Región2", ...].
4. Se llena cada fila con los valores diarios correspondientes.
5. Se grafica con Google Charts como un gráfico de líneas con leyenda lateral.

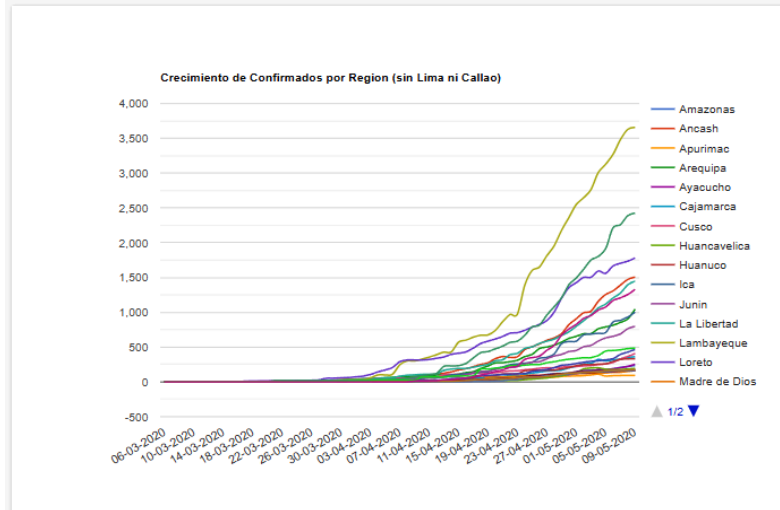
Fragmento clave (ejercicio6.js):

```
const regionesExcluidas = ['Lima', 'Callao'];  
  
const regionEjemplo = data.find(r => !regionesExcluidas.includes(r.region));
```

Visualización esperada:

- Eje X: Fechas
- Eje Y: Casos confirmados diarios
- Múltiples líneas (una por región), excluyendo Lima y Callao

Comparación del crecimiento por día (excepto Lima y Callao)



Ejercicio 7: Gráfico comparativo de regiones elegidas por el usuario

Objetivo: Permitir al usuario seleccionar libremente una o varias regiones y visualizar su evolución de casos confirmados en un gráfico comparativo.

Archivos utilizados:

- **7.html:** contiene los elementos visuales, estilos y un botón para generar el gráfico.
- **ejercicio7.js:** gestiona la carga de datos, creación de checkboxes y la construcción del gráfico.
- **data.json:** fuente de datos por región y por fecha.

Funcionamiento:

1. Al cargar la página, se ejecuta `loadRegions()` que genera un conjunto de checkboxes con los nombres de todas las regiones disponibles.
2. El usuario selecciona una o más regiones y hace clic en el botón "Comparar".
3. El script filtra los datos correspondientes, construye un `DataTable` para Google Charts y genera un gráfico de líneas con una serie por región seleccionada.

Fragmento clave (ejercicio7.js):


```
const selected = Array.from(document.querySelectorAll('#checkboxes input:checked'))  
  .map(cb => cb.value);  
  
if (selected.length === 0) {  
  alert('Selecciona al menos una región');  
  return;  
}  
  
const filteredRegions = jsonData.filter(region => selected.includes(region.region));
```

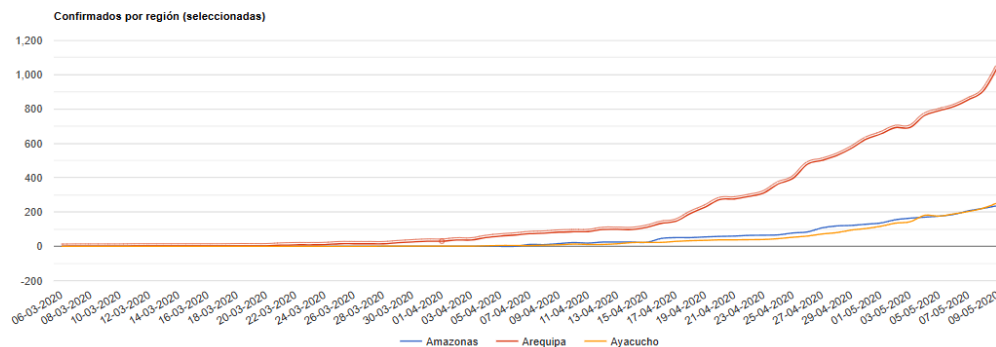
Visualización esperada:

- Gráfico de líneas dinámico
- Cada línea representa una región seleccionada
- Ejes: fechas (X) y confirmados (Y)
- Leyenda inferior

Comparar regiones seleccionadas

☒ Amazonas
☐ Ancash
☐ Apurímac
☒ Arequipa
☒ Ayacucho
☐ Cajamarca
☐ Callao
☐ Cusco
☐ Huancavelica
☐ Huanuco
☐ Ica
☐ Junín
☐ La Libertad
☐ Lambayeque
☐ Lima
☐ Loreto
☐ Madre de Dios
☐ Moquegua
☐ Pasco
☐ Piura
☐ Puno
☐ San Martín
☐ Tarma
☐ Tumbes
☐ Ucayali

Comparar



Ejercicio 8: Crecimiento diario en regiones (excluyendo Lima y Callao)

Objetivo: Visualizar el crecimiento diario de casos confirmados de COVID-19 en todas las regiones excepto Lima y Callao, mostrando una línea por región.

Archivos utilizados:

- 8.html: estructura de la página con estilos y el contenedor del gráfico.
- ejercicio8.js: código que filtra, estructura y grafica los datos.
- data.json: base de datos con valores diarios por región.

Funcionamiento:

1. Al cargarse la página, se ejecuta `loadData()`, que obtiene `data.json`.
2. Se filtran todas las regiones menos "Lima" y "Callao".
3. Se genera una estructura con fechas como eje X y valores diarios por región como columnas.
4. Se usa Google Charts para renderizar un gráfico de líneas con todas las regiones restantes.

Fragmento clave (ejercicio8.js):

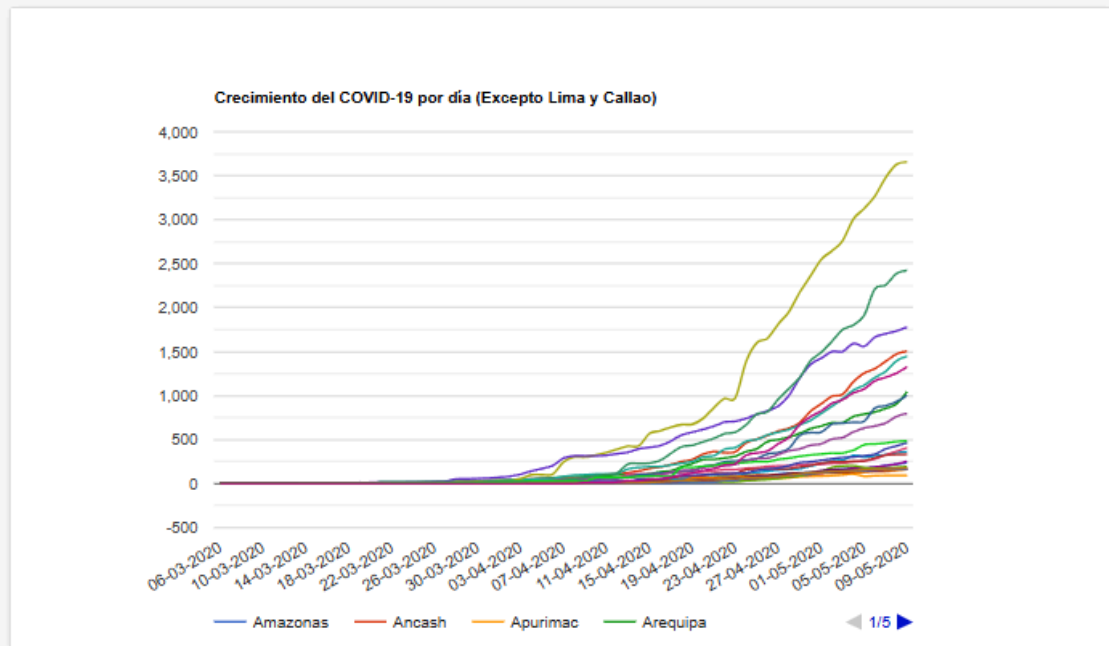
```
function drawChart() {
  const filteredRegions = jsonData.filter(region => region.region !== 'Lima' && region.region !== 'Callao');
```

Visualización esperada:

- Gráfico de líneas multiserie
- Eje X: fechas
- Eje Y: número de casos por día

- Cada línea representa una región (excepto Lima y Callao)

Crecimiento del COVID-19 por día (Excepto Lima y Callao)



F) Conclusión

La práctica permitió reforzar los fundamentos de AJAX y su integración con tecnologías como PHP, MySQL y Google Charts. Se desarrollaron soluciones completas para mostrar, comparar y visualizar datos reales, tanto en formato textual como gráfico, cumpliendo los objetivos de la actividad y fortaleciendo habilidades clave en desarrollo web moderno.

REFERENCIAS EN FORMATO APA7

W3Schools. (s.f.-a). Node.js Introduction. W3Schools. <https://www.w3schools.com/nodejs/>

W3Schools. (s.f.-b). Node.js Modules. W3Schools. https://www.w3schools.com/nodejs/nodejs_modules.asp

W3Schools. (s.f.-d). AJAX Introduction. W3Schools. https://www.w3schools.com/js/js_ajax_intro.asp

serverjs.io. (s.f.). Fast, modern, and lightweight Node.js web server. <https://serverjs.io/>



UNIVERSIDAD NACIONAL DE SAN AGUSTÍN DE AREQUIPA
FACULTAD DE INGENIERÍA DE PRODUCCIÓN Y SERVICIOS
ESCUELA PROFESIONAL DE INGENIERÍA DE SISTEMAS



Página: 36

RETROALIMENTACIÓN