Universidad Nacional San Agustin de Arequipa

Facultad de Ingenieria de Produccion y Servicios

Escuela Profesional de Ingenieria de Sistemas

Curso:

Programacion Web 2

Profesor:

Carlo Jose Luis Corrales Delgado

Tema:

Laboratorio 4 - AJAX

Integrantes:

Cahui Benegas Anthony Ronaldo
Cuno Cahuari Armando Steven
Llacho Delgado Samir Jaren

Repositorio:

https://github.com/SteveArms/pwebAJAX.git

2024

Informe - AJAX

En la tarea actual, se han asignado 8 ejercicios relacionados con AJAX, así como otro ejercicio centrado en Markdown. Estos se distribuirán entre los integrantes del equipo para que cada uno se enfoque en una parte específica. La idea es evidenciar la aplicación de diversas funciones y métodos relacionados con AJAX, así como la comprensión de la sintaxis y funcionalidades de Markdown.
Para comenzar la actividad, se inició creando un repositorio en GitHub donde se alojarán los archivos pertinentes, como los códigos HTML, CSS, JS, entre otros. Este repositorio servirá como un espacio centralizado para colaborar y compartir el trabajo realizado en los ejercicios de AJAX y Markdown.

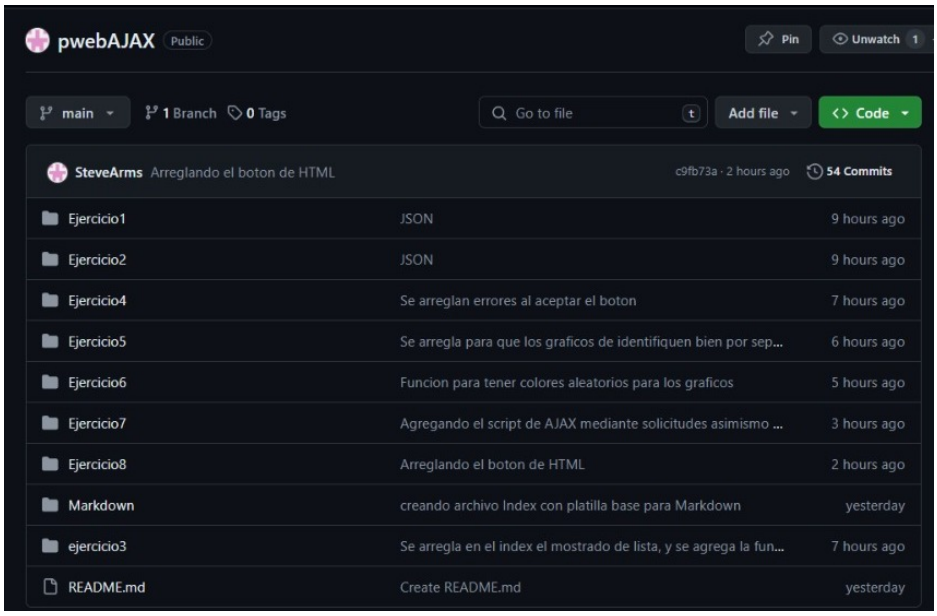


Figure 1: GitHub - Repositorio pwebAJAX

Para comprender mejor la programación necesaria para los ejercicios, primero probamos los ejemplos de AJAX de W3Schools en nuestro propio servidor. Utilizamos Python para abrir el servidor y ejecutar los ejemplos, lo que nos permitió experimentar con el código y entender su funcionamiento en un entorno práctico.
Para hacer prueba de los ejercicios de w3schools haremos prueba mediante el servidor por python.

AJAX XMLHttp

El objeto XMLHttpRequest se utiliza para intercambiar datos con el servidor sin necesidad de recargar la página.

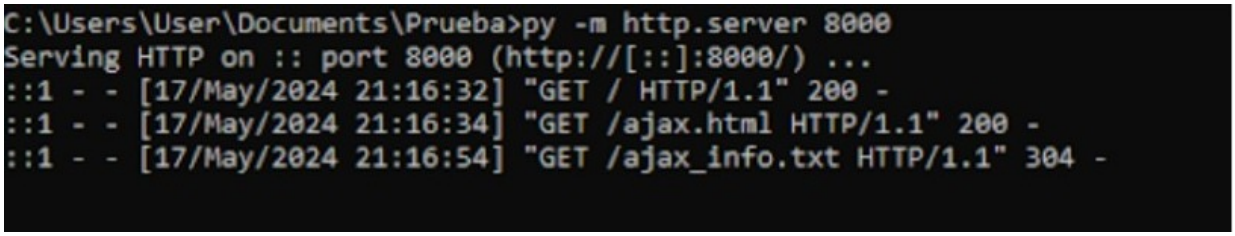


Figure 2: Servidor Python - XMLHttp

Figure 3: Pagina - XMLHttp

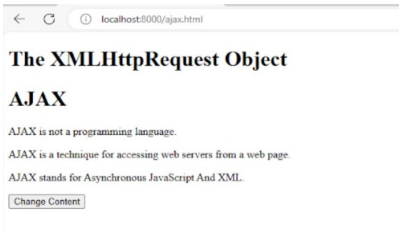


Figure 4: Pagina - XMLHttp

AJAX Request

Mediante el objeto XMLHttpRequest, enviamos una solicitud al servidor utilizando los métodos ‘open‘ y ‘send‘, lo que permite una interacción dinámica sin necesidad de recargar la página.

Figure 5: Servidor Python - AJAX Request

Figure 6: Pagina - AJAXRequest

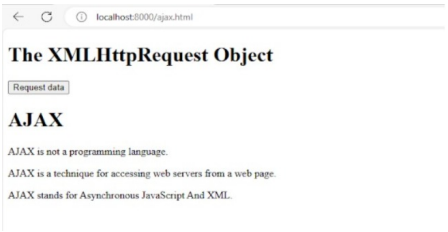


Figure 7: Pagina - AJAXRequest

AJAX Response

El objeto XMLHttpRequest tiene un analizador XML incorporado. La propiedad 'responseXML' devuelve la respuesta del servidor como un objeto XML DOM.

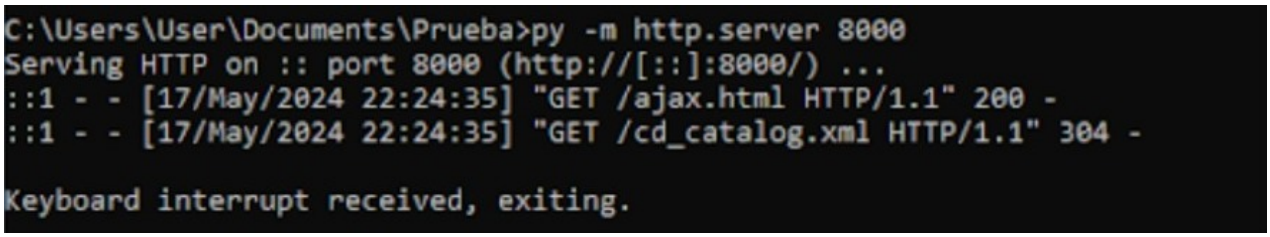


Figure 8: Servidor Python - AJAX Response

Figure 9: Pagina - AJAXResponse

AJAX XMLFile

Cuando un usuario hace clic en el botón "Obtener información del CD", se ejecuta la función 'loadDoc()'. Esta función crea un objeto.
XMLHttpRequest, define la función que se ejecutará cuando la respuesta del servidor esté lista y envía la solicitud al servidor.

Figure 10: Servidor Python - AJAX XMLFile

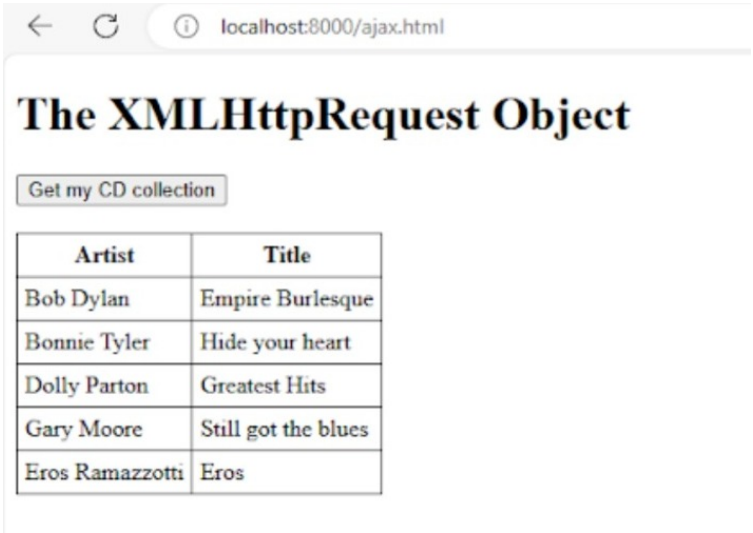


Figure 11: Pagina - XMLFile

Otros mas.

8 Ejercicios AJAX / GOOGLE CHART

1er Ejercicio: Listar todas las regiones

El código implementa una función llamada ‘loadTable‘ que envía una solicitud HTTP para obtener datos de un archivo JSON y mostrarlos en una tabla. La función crea un objeto ‘XMLHttpRequest‘, el cual ejecuta una función cuando la solicitud es exitosa, llamando a ‘myFunction()‘. Esta solicitud ”GET” asincrónica obtiene los datos del archivo JSON.
La función ‘myFunction‘ toma la respuesta JSON, la convierte en un objeto JavaScript mediante ‘JSON.parse‘, y construye una tabla HTML con los datos obtenidos. Finalmente, inserta esta tabla en el elemento HTML con el ID ”demo”.

```
JS script.js > ...
1   //Implementaremos la funcion loadTable para hacer una solicitud al JSON en el cual analizara si es exitosa
2   function loadTable(){
3       var xhttp = new XMLHttpRequest();
4       xhttp.onreadystatechange = function(){
5           if(this.status == 200 && this.readyState == 4){
6               // Ejecutar una función cuando la solicitud esté completa y sea exitosa
7               myFunction(this);
8           }
9       }
10      xhttp.open("GET" , "data.json", true);
11      xhttp.send();
12  }
13  //Empleamos una funcion la cual retornara los datos respectivos de una tabla
14  function myFunction(xhttp){
15      var responseJSON = JSON.parse(xhttp.responseText);
16      var table = "<tr><th> Region </th><th> Fallecidos </th></tr>"
17      for(var i = 0; i < responseJSON.length; i++){
18          var regiones = responseJSON[1];
19          var region = regiones.region;
20          var dateRegion = regiones.confirmed;
21          table += "<tr><td>" + region + "</td><td>";
22          for(var j = 0; j < dateRegion.length; j++){
23              var date = dateRegion[j].date ;
24              table += date + "<br>";
25          }
26          table += "</td></tr>";
27      }
28      document.getElementById("demo").innerHTML = table;
29  }
```

Figure 12: Codigo AJAX - Ejercicio 1

```
C:\Users\User\Documents\Prueba>py -m http.server 8000
Serving HTTP on :: port 8000 (http://[::]:8000/) ...
::1 - - [18/May/2024 08:46:24] "GET /pagina.html HTTP/1.1" 200 -
::1 - - [18/May/2024 08:46:24] "GET /script.js HTTP/1.1" 200 -
::1 - - [18/May/2024 08:46:40] "GET /data.json HTTP/1.1" 304 -

Keyboard interrupt received, exiting.
```

Figure 13: Servidor Python - Ejercicio 1

Figure 14: Pagina en Servidor - Ejercicio 1

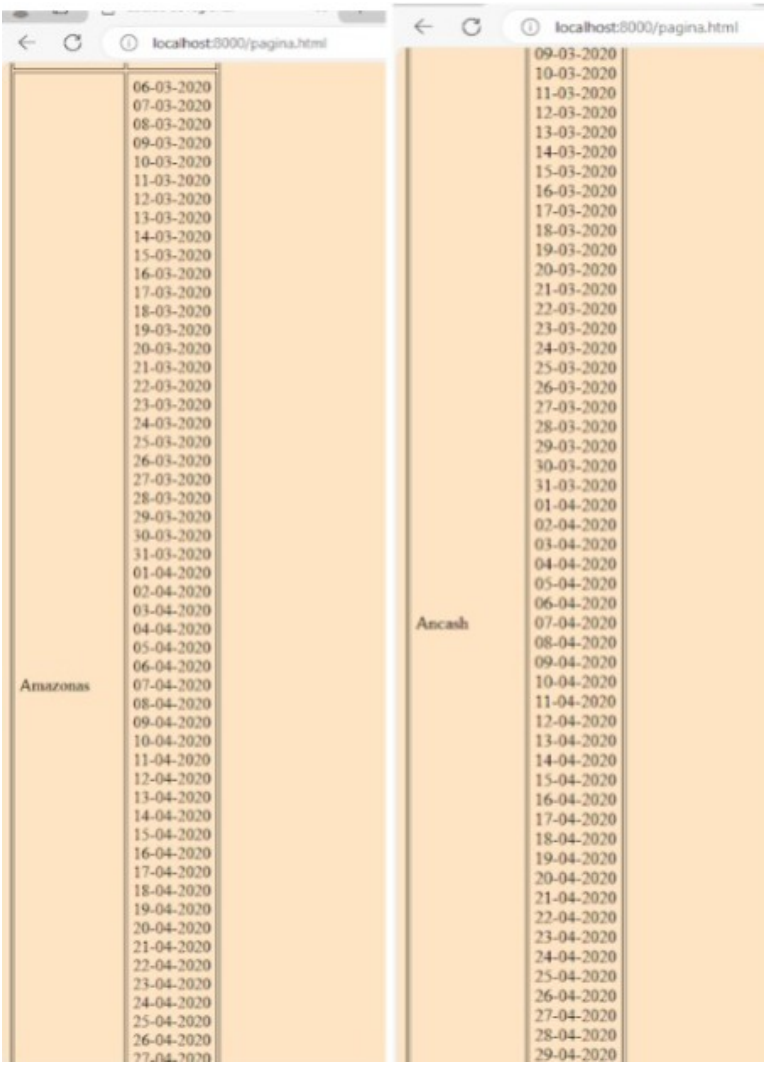


Figure 15: Pagina en Servidor - Ejercicio 1

2do Ejercicio: Total de confirmas por regiones

El siguiente código realiza una operación similar al ejercicio anterior, enviando una solicitud HTTP al servidor. Una vez que la solicitud es exitosa, se ejecuta la función 'myFunction()'. Esta función abre una solicitud "GET" y devuelve una tabla que se inserta en el elemento HTML correspondiente. La tabla se genera fila por fila, agregando el nombre de la región y el número de confirmados.

```
JS script.js > ...
1   function loadTable(){
2       var xhttp = new XMLHttpRequest();
3       xhttp.onreadystatechange = function(){
4           if(this.status == 200 && this.readyState == 4){
5               // Ejecutar una función cuando la solicitud esté completa y sea exitosa
6               myFunction(this);
7           }
8       }
9       xhttp.open("GET" , "data.json", true);
10      xhttp.send();
11  }
12  //Utilizaremos el mismo script del Ejercicio 1 pero modificaremos la entrada de datos por Nro Fallecidos
13  function myFunction(xhttp){
14      var responseJSON = JSON.parse(xhttp.responseText);
15      var table = "<tr><th> Region </th><th> Nro Fallecidos </th></tr>"
16      for(var i = 0; i < responseJSON.length; i++){
17          var regiones = responseJSON[i];
18          var region = regiones.region;
19          var dateRegion = regiones.confirmed;
20          table += "<tr><td>" + region + "</td><td>";
21          table += dateRegion.length + "</td></tr>";
22      }
23      document.getElementById("demo").innerHTML = table;
24  }
```

Figure 16: Codigo AJAX - Ejercicio 2

```
C:\Users\User>cd C:\Users\User\Documents\Prueba

C:\Users\User\Documents\Prueba>py -m http.server
Serving HTTP on :: port 8000 (http://[::]:8000/) ...
::1 - - [18/May/2024 09:34:17] "GET /pagina.html HTTP/1.1" 304 -
::1 - - [18/May/2024 09:34:17] "GET /style.css HTTP/1.1" 304 -
::1 - - [18/May/2024 09:34:17] "GET /script.js HTTP/1.1" 200 -
::1 - - [18/May/2024 09:34:21] "GET /data.json HTTP/1.1" 304 -

Keyboard interrupt received, exiting.
```

Figure 17: Servidor Python - Ejercicio 2

Figure 18: Pagina en Servidor - Ejercicio 2

Figure 19: Pagina en Servidor - Ejercicio 2

```
User@DESKTOP-567SEDE MINGW64 ~/Documents/pwebAJAX/Ejercicio2 (main)
$ git add script.js

User@DESKTOP-567SEDE MINGW64 ~/Documents/pwebAJAX/Ejercicio2 (main)
$ git commit -m "Utilizaremos el mismo script del Ejercicio 1 pero esta vez solo
 iteraremos una ves con tal que mediante un length hallaremos la cantidad de fal
lecidos"
[main 526c9d7] Utilizaremos el mismo script del Ejercicio 1 pero esta vez solo i
teraremos una ves con tal que mediante un length hallaremos la cantidad de falle
cidos
 1 file changed, 24 insertions(+)
 create mode 100644 Ejercicio2/script.js

User@DESKTOP-567SEDE MINGW64 ~/Documents/pwebAJAX/Ejercicio2 (main)
$ git push
Enumerating objects: 6, done.
Counting objects: 100% (6/6), done.
Delta compression using up to 12 threads
Compressing objects: 100% (4/4), done.
Writing objects: 100% (4/4), 950 bytes | 950.00 KiB/s, done.
Total 4 (delta 1), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (1/1), completed with 1 local object.
To https://github.com/SteveArms/pwebAJAX.git
   1ef2dfd..526c9d7  main -> main
```

Figure 20: Commits - Ejercicio 2

3er Ejercicio: 10 regiones con la mayor suma

Para implementar la funcionalidad de mostrar la lista de las 10 regiones con mayor número de casos confirmados, primero tenemos una función ‘mostrarLista()‘ que cambia el estilo del contenedor para hacerlo visible. Además, esta función llama a ‘cargarDatos()‘, que se encarga de solicitar los datos.
La función ‘cargarDatos()‘ realiza una solicitud usando ‘fetch‘, convierte la respuesta en JSON y pasa los datos obtenidos a la función ‘mostrarTop10Regiones()‘, capturando cualquier error que ocurra durante la solicitud.
La función ‘mostrarTop10Regiones()‘ inicia un objeto para almacenar el total de casos confirmados por región, luego recorre cada objeto ’region’, calculando el total de casos confirmados por cada región sumando los valores del array ‘confirmed‘ y almacenando el total con una clave correspondiente a cada región.
Luego, ordena las regiones por el número de casos confirmados convirtiéndolo en un array de pares clave-valor y ordenándolo de manera descendente. Una vez hecho esto, usando el ID de la lista ‘topList‘, se toman las primeras 10 regiones, creando un nuevo elemento ‘li‘ para cada una y agregando estos elementos al contenedor ‘topList‘.

```
function mostrarLista() {
    document.getElementById('topRegionsListContainer').style.display = 'block';
    cargarDatos();
}
function cargarDatos() {
    fetch('data.json')
        .then(response => response.json())
        .then(data => mostrarTop10Regiones(data))
        .catch(error => console.error('Error al cargar los datos:', error));
}
function mostrarTop10Regiones(data) {
    // Calcular la suma total para cada región
    const regionSum = {};
    data.forEach(region => {
        const totalConfirmed = region.confirmed.reduce((sum, entry) => sum + parseInt(entry.value), 0);
        regionSum[region.region] = totalConfirmed;
    });

    // Ordenar las regiones por su suma total en orden descendente
    const sortedRegions = Object.entries(regionSum).sort((a, b) => b[1] - a[1]);

    const top10List = document.getElementById('topList');
    top10List.innerHTML = ''; // Limpiar la lista antes de agregar los nuevos elementos
    sortedRegions.slice(0, 10).forEach(([region, total]) => {
        const listItem = document.createElement('li');
        listItem.textContent = `${region}: ${total}`;
        top10List.appendChild(listItem);
    });
}

document.addEventListener('DOMContentLoaded', cargarDatos);
```

Figure 21: Codigo AJAX - Ejercicio 3

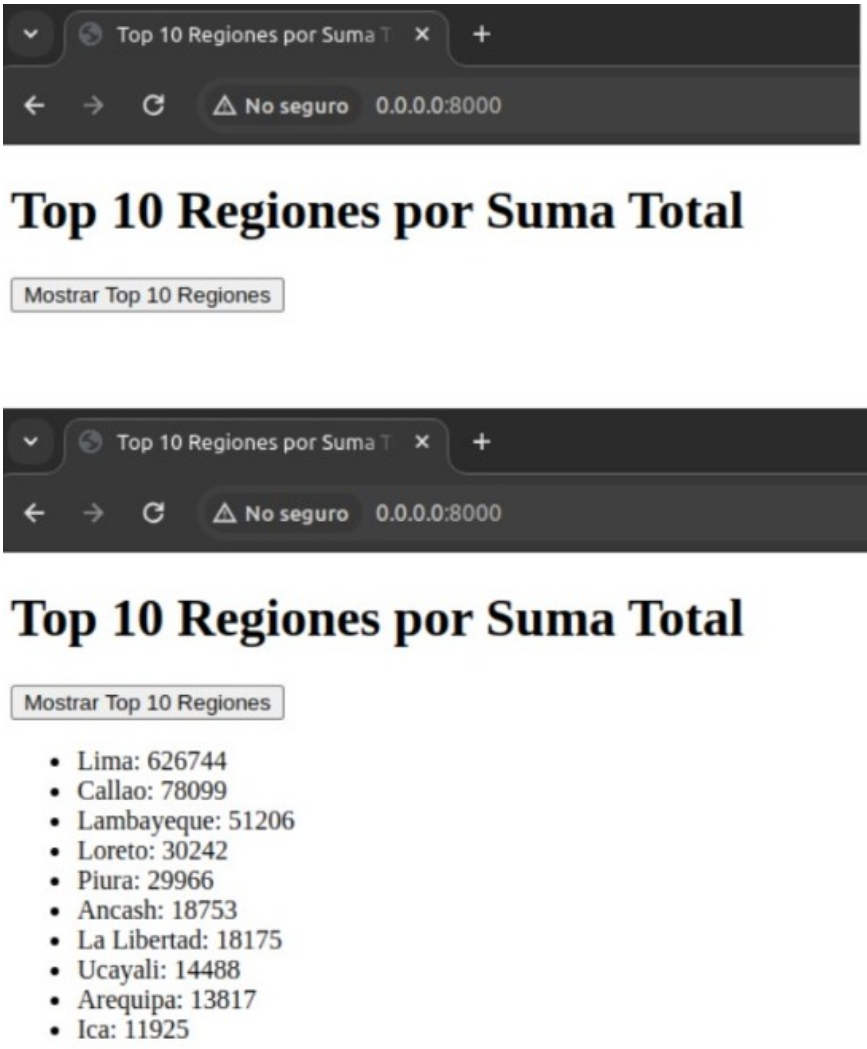


Figure 22: Pagina en Servidor - Ejercicio 3

Figure 23: Servidor - Ejercicio 3

4to Ejercicio: Gráfico en el tiempo de los valores para la región de Arequipa

Implementaremos un botón que disparará un evento llamando a la función 'cargarDatos()', verificando si los datosse cargan correctamente. Se crea una constante para almacenar los datos correspondientes a "Arequipa"; si no se encuentran, se mostrará un mensaje de error. Una vez hecho esto, se extraen las fechas y los valores confirmados, almacenándolos en dos arrays separados.
Para visualizar los datos en la gráfica, utilizaremos la función 'mostrarGrafico()', que recibirá los arrays de fechas y valores necesarios. La función 'cargarDatos()', que se trabajó en el ejercicio anterior, realizará la solicitud 'fetch'. La función 'mostrarGrafico()' se encargará de crear y configurar el gráfico, incluyendo el conjunto de datos correspondientes.

```
document.getElementById('mostrarGraficoBtn').addEventListener('click', function() {
    cargarDatos().then(data => {
        const arequipaData = data.find(region => region.region === 'Arequipa');
        if (!arequipaData) {
            console.error('No se encontraron datos para la región de Arequipa.');
            return;
        }

        const fechas = arequipaData.confirmed.map(entry => entry.date);
        const valores = arequipaData.confirmed.map(entry => parseInt(entry.value));

        mostrarGrafico(fechas, valores);
    }).catch(error => console.error('Error al cargar los datos:', error));
});

function cargarDatos() {
    return fetch('data.json')
        .then(response => response.json());
        console.log("SI funciona");
}

function mostrarGrafico(fechas, valores) {
    document.getElementById('chartContainer').style.display = 'block';

    const ctx = document.getElementById('myChart').getContext('2d');
    new Chart(ctx, {
        type: 'line',
        data: {
            labels: fechas,
            datasets: [{
                label: 'Valores para la Región de Arequipa',
                data: valores,
                borderColor: 'rgba(75, 192, 192, 1)',
                borderWidth: 1
            }]
        },
        options: {
            scales: {
                y: {
                    beginAtZero: true
                }
            }
        }
    });
}
```

Figure 24: Codigo AJAX - Ejercicio 4

Figure 25: Pagina en Servidor - Ejercicio 4

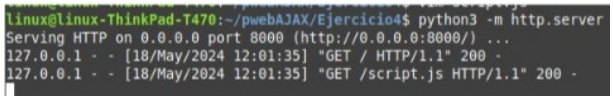


Figure 26: Servidor - Ejercicio 4

5to Ejercicio: Gráficos comparativos entre regiones usando líneas

En este ejercicio se implementará la funcionalidad para mostrar y ocultar un gráfico comparativo. Para ello, utilizamos un botón que añade un evento mediante un clic, el cual llama a la función 'toggleChartVisibility()'. Esta función se encarga de validar las entradas; si falta alguna, mostrará una alerta. Si las dos entradas son válidas, se llama a 'cargarDatos()', y luego se pasa la data a 'mostrarGrafico()'.
La función 'cargarDatos()' realiza una solicitud 'fetch' para obtener los datos del archivo JSON y filtra los datos especificados en las regiones. La función 'mostrarGrafico()' se encarga de gestionar el elemento del lienzo y verificar si existe un gráfico anterior. Además, prepara los datos extrayendo las fechas y los conjuntos de datos por cada región, creando un gráfico de línea.

```
let chartVisible = false;
let myChart;

document.getElementById('toggleChartBtn').addEventListener('click', toggleChartVisibility);

function toggleChartVisibility() {
    chartVisible = !chartVisible;
    const chartContainer = document.getElementById('chartContainer');
    chartContainer.style.display = chartVisible ? 'block' : 'none';

    if (chartVisible) {
        const region1 = document.getElementById('region1Input').value.trim();
        const region2 = document.getElementById('region2Input').value.trim();

        if (!region1 || !region2) {
            alert('Por favor, ingrese ambas regiones.');
            return;
        }

        cargarDatos([region1, region2]).then(data => {
            mostrarGrafico(data);
        });
    } else {
        if (myChart) {
            myChart.destroy();
        }
    }
}

function cargarDatos(regiones) {
    return fetch('data.json').then(response => response.json())
                             .then(data => data.filter(region => regiones.includes(region.region)));
}

function mostrarGrafico(data) {
    const ctx = document.getElementById('myChart').getContext('2d');

    if (myChart) {
        myChart.destroy();
    }

    const labels = data[0].confirmed.map(entry => entry.date);
    const datasets = data.map((regionData, index) => ({
        label: `Valores para ${regionData.region}`,
        data: regionData.confirmed.map(entry => parseInt(entry.value)),
        borderColor: index === 0 ? 'rgba(75, 192, 192, 1)' : 'rgba(192, 75, 75, 1)',
        borderWidth: 1
    }));

    myChart = new Chart(ctx, {
        type: 'line',
        data: {
            labels: labels,
            datasets: datasets
        },
        options: {
```

Figure 27: Codigo AJAX - Ejercicio 5

Figure 28: Pagina en Servidor - Ejercicio 5

Figure 29: Servidor - Ejercicio 5

6to Ejercicio: Gráfico comparativo del crecimiento en regiones excepto Lima y Callao

Este código tiene como objetivo cargar datos de un archivo JSON, filtrarlos excluyendo las regiones de Lima y Callao, y representarlos gráficamente.
La función ‘onloadGraphic()‘ se encarga de realizar una solicitud para obtener los datos del JSON y luego llama a la función ‘graficar()‘ pasando estos datos como argumento.
La función ‘graficar(regiones)‘ tiene la responsabilidad de cargar la biblioteca Google Charts y llamar a ‘drawChart()‘ una vez que la biblioteca está cargada. En ‘drawChart()‘, se preparan los datos para la visualización utilizando la función ‘google.visualization.arrayToDataTable()‘ y se configuran las columnas necesarias. Posteriormente, se crea una instancia de ‘BarChart‘ de Google Charts, la cual dibuja el gráfico en el elemento HTML con ID ”grafico”.
La función ‘valoresTabla(regiones)‘ se encarga de preparar los datos necesarios para la visualización del gráfico, excluyendo las regiones de Lima y Callao. Calcula la suma de casos confirmados para cada región y almacena estos datos junto con el nombre de la región y un color aleatorio para su posterior visualización en la gráfica.
En resumen, este código logra cargar datos desde un archivo JSON, filtrarlos y representarlos gráficamente, excluyendo las regiones de Lima y Callao, utilizando la biblioteca Google Charts.

```
function onloadGraphic(){
    var xhttp = new XMLHttpRequest();
    xhttp.onreadystatechange = function(){
        if(xhttp.status == 200 && xhttp.readyState == 4){
            var responseJSON = JSON.parse(xhttp.responseText);
            graficar(responseJSON);
        }
    }
    xhttp.open("GET", "data.json", true);
    xhttp.send();
}
function graficar(regiones){
    google.charts.load("current", {packages:["corechart"]});
    google.charts.setOnLoadCallback(drawChart);
    function drawChart() {
        var data = google.visualization.arrayToDataTable(valoresTabla(regiones));

        var view = new google.visualization.DataView(data);
        view.setColumns([0, 1,
                         { calc: "stringify",
                           sourceColumn: 1,
                           type: "string",
                           role: "annotation" },
                         2]);

        var options = {
            title: "Casos confirmados en todos menos Lima y Callao",
            width: 800,
            height: 600,
            bar: {groupWidth: "95%"},
            legend: { position: "none" },
        };
        var chart = new google.visualization.BarChart(document.getElementById("grafico"));
        chart.draw(view, options);
    }
}
```

```
function valoresTabla(regiones) {
    const valores = [];
    valores.push(["Region", "Confirmados", { role: "style" }]);

    const regionesFiltradas = regiones.filter(region => region.region != "Lima" && region.region != "Callao");

    for (let i = 0; i < regionesFiltradas.length; i++) {
        const nombreRegion = regionesFiltradas[i].region;
        const listaValores = regionesFiltradas[i].confirmed;
        let suma = 0;
        for (let j = 0; j < listaValores.length; j++) {
            suma += parseInt(listaValores[j].value); // Asumiendo que listaValores[j] es un número
        }
        console.log(`Región: ${regionesFiltradas[i].region}, Suma: ${suma}`);
        valores.push([regionesFiltradas[i].region, suma, colorRandom()]); // Utiliza regionesFiltradas[i] aquí
    }
    return valores;
}


function colorRandom() {
    var r = Math.floor(Math.random() * 256);
    var g = Math.floor(Math.random() * 256);
    var b = Math.floor(Math.random() * 256);
    return `rgb(${r}, ${g}, ${b})`;
}
```

Figure 30: Codigo AJAX - Ejercicio 6

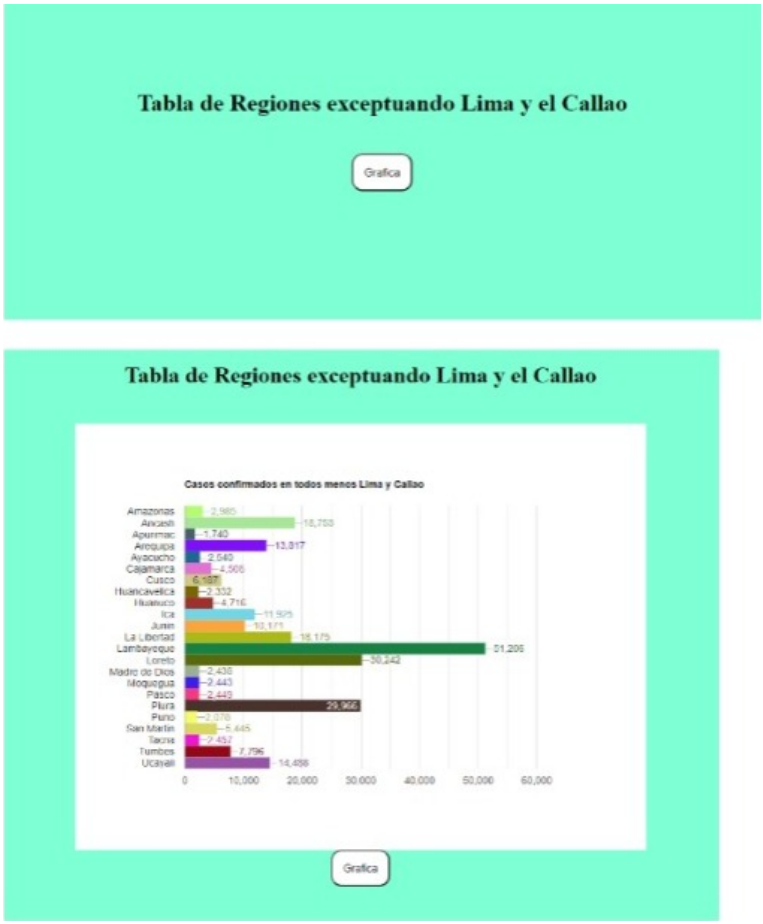


Figure 31: Pagina en Servidor - Ejercicio 6

```
C:\Users\User\Documents\pwebAJAX\Ejercicio8>py -m http.server 8000
Serving HTTP on :: port 8000 (http://[::]:8000/) ...
::1 - - [18/May/2024 17:00:04] "GET /pagina.html HTTP/1.1" 304 -
::ffff:127.0.0.1 - - [18/May/2024 17:00:05] "GET /pagina.html HTTP/1.1" 304 -
::ffff:127.0.0.1 - - [18/May/2024 17:00:06] "GET /pagina.html HTTP/1.1" 304 -

Keyboard interrupt received, exiting.
```

Figure 32: Servidor - Ejercicio 6

7mo Ejercicio: Gráficos comparativos entre regiones elegidas por el usuario.

Este ejercicio sigue una lógica similar al anterior en términos de la solicitud de datos, pero en esta ocasión se llaman a tres funciones una vez que la solicitud es exitosa, pasando como argumentos los datos y regiones obtenidos. Dentro de cada función ‘graficar()‘, se almacenan los datos correspondientes de las dos regiones seleccionadas, así como la suma de los casos confirmados para cada una. Estos dos valores se utilizan para generar gráficos que se dibujan en elementos HTML específicos.

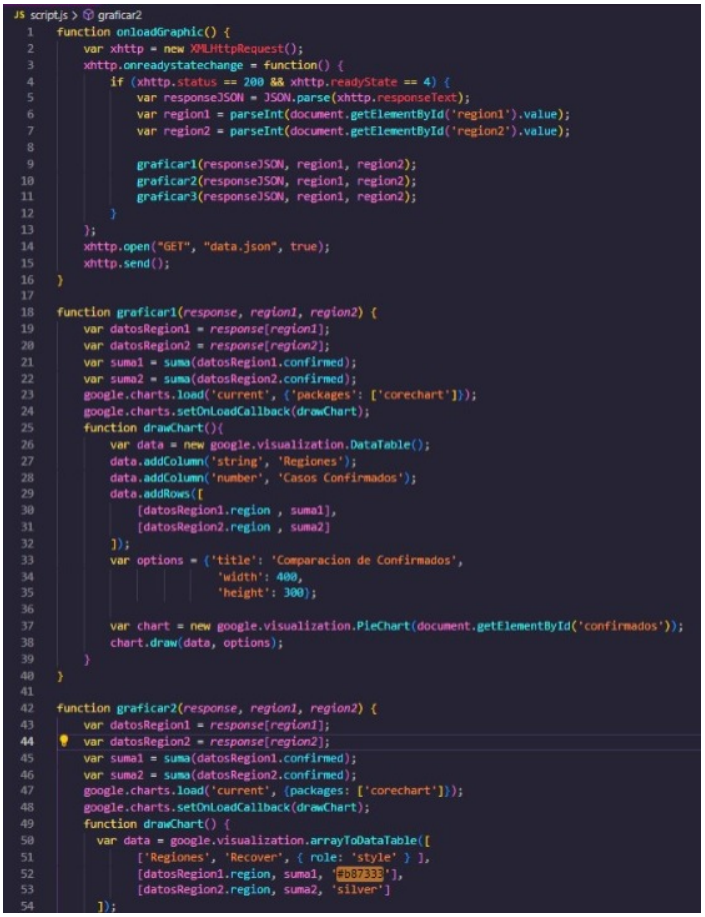
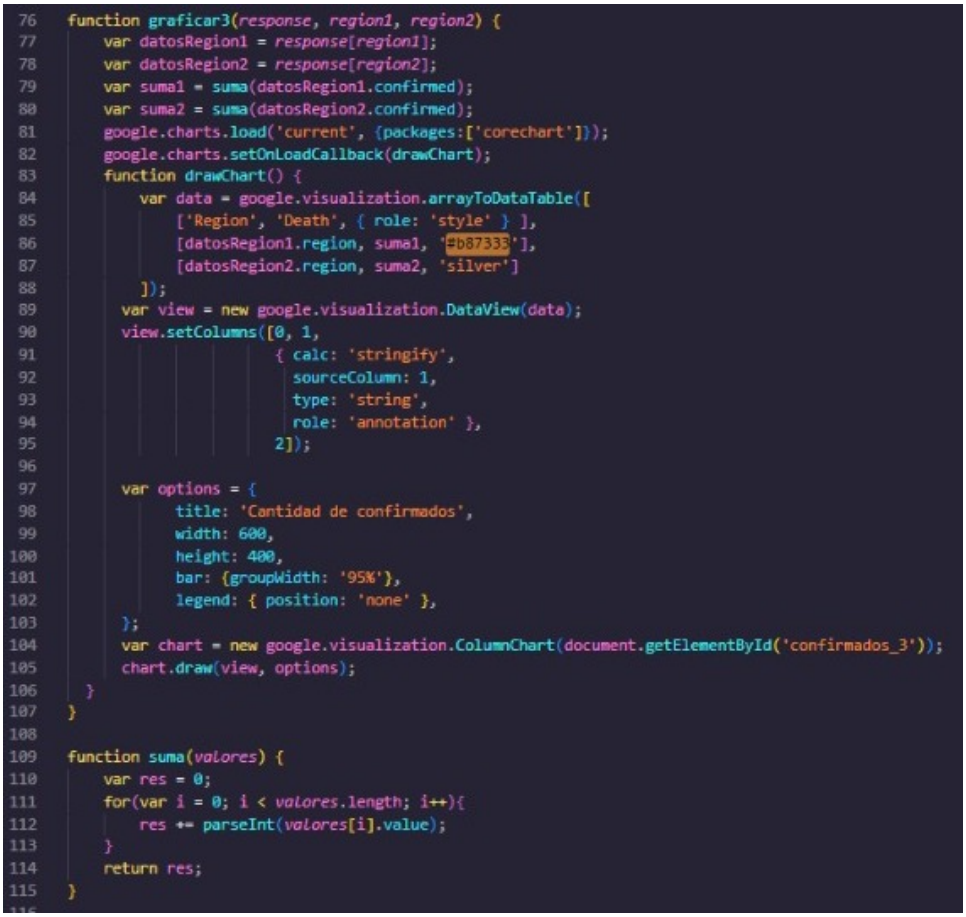


Figure 33: Codigo AJAX - Ejercicio 7

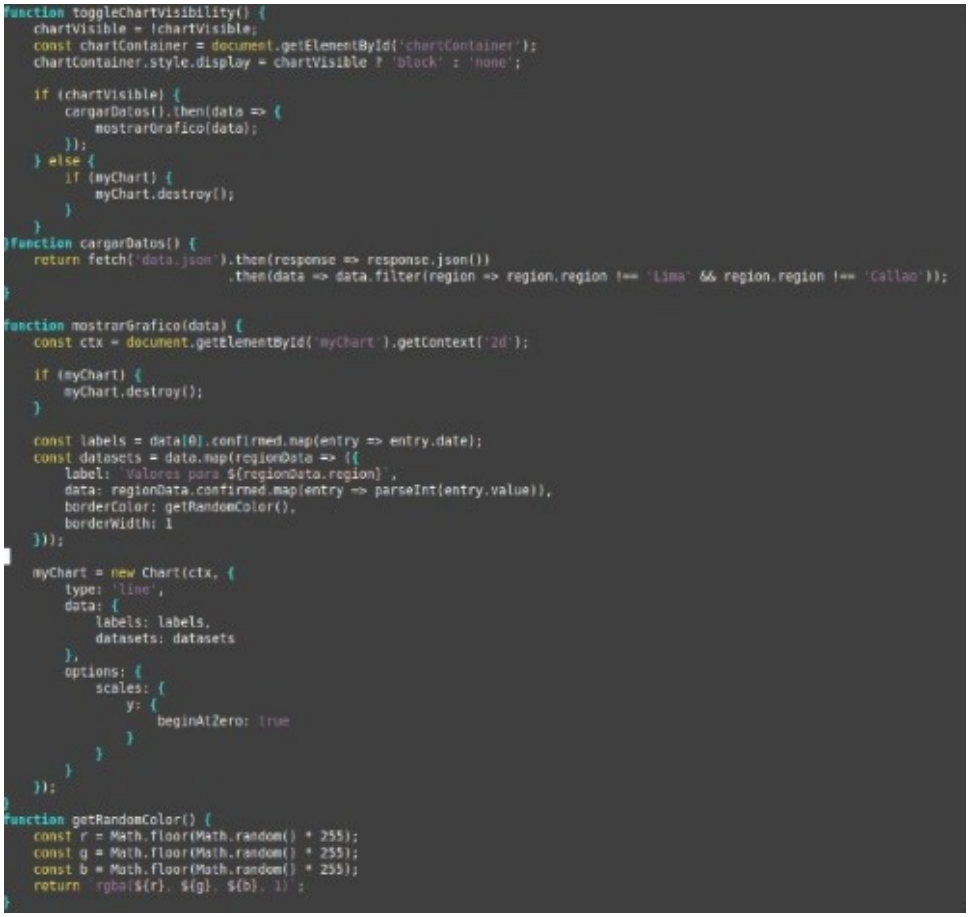


```
function graficar3(response, region1, region2) {
    var datosRegion1 = response[region1];
    var datosRegion2 = response[region2];
    var suma1 = suma(datosRegion1.confirmed);
    var suma2 = suma(datosRegion2.confirmed);
    google.charts.load('current', {packages:['corechart']});
    google.charts.setOnLoadCallback(drawChart);
    function drawChart() {
        var data = google.visualization.arrayToDataTable([
            ['Region', 'Death', { role: 'style' } ],
            [datosRegion1.region, suma1, '#b87333'],
            [datosRegion2.region, suma2, 'silver']
        ]);
      var view = new google.visualization.DataView(data);
      view.setColumns([0, 1,
                       { calc: 'stringify',
                         sourceColumn: 1,
                         type: 'string',
                         role: 'annotation' },
                       2]);

      var options = {
            title: 'Cantidad de confirmados',
            width: 600,
            height: 400,
            bar: {groupWidth: '95%'},
            legend: { position: 'none' },
      };
      var chart = new google.visualization.ColumnChart(document.getElementById('confirmados_3'));
      chart.draw(view, options);
  }
}

function suma(valores) {
    var res = 0;
    for(var i = 0; i < valores.length; i++){
        res += parseInt(valores[i].value);
    }
    return res;
}
```

Figure 34: Codigo AJAX - Ejercicio 7

8vo Ejercicio: Gráfico comparativo del crecimiento en regiones excepto Lima y Callao, mostrando el número de confirmados por cada día.

Este código permite alternar la visibilidad de un gráfico al hacer clic en un botón. Cuando se hace clic en el botón, la función ‘toggleChartVisibility()‘ alterna entre mostrar u ocultar el gráfico actualizando el estilo del contenedor del gráfico (‘chartContainer‘). Si el gráfico está visible, se cargan los datos desde un archivo JSON mediante la función ‘cargarDatos()‘ y luego se muestra el gráfico utilizando la función ‘mostrarGrafico()‘. Si el gráfico está oculto, se destruye la instancia del gráfico Chart.js.
La función ‘mostrarGrafico()‘ utiliza los datos recibidos como argumento para construir y mostrar un gráfico de líneas. Para cada conjunto de datos, se genera un color aleatorio utilizando la función ‘getRandomColor()‘, y luego se crea una instancia de Chart.js con los datos y opciones necesarios para visualizar el gráfico.
En resumen, este código proporciona una forma sencilla de mostrar u ocultar un gráfico interactivo al hacer clic en un botón, y utiliza la biblioteca Chart.js para generar y mostrar el gráfico con los datos proporcionados.



```
function toggleChartVisibility() {
    chartVisible = !chartVisible;
    const chartContainer = document.getElementById('chartContainer');
    chartContainer.style.display = chartVisible ? 'block' : 'none';

    if (chartVisible) {
        cargarDatos().then(data => {
            mostrarGrafico(data);
        });
    } else {
        if (myChart) {
            myChart.destroy();
        }
    }
}function cargarDatos() {
    return fetch('data.json').then(response => response.json())
                             .then(data => data.filter(region => region.region !== 'Lima' && region.region !== 'Callao'));
}

function mostrarGrafico(data) {
    const ctx = document.getElementById('myChart').getContext('2d');

    if (myChart) {
        myChart.destroy();
    }

    const labels = data[0].confirmed.map(entry => entry.date);
    const datasets = data.map(regionData => ({
        label: `Valores para ${regionData.region}`,
        data: regionData.confirmed.map(entry => parseInt(entry.value)),
        borderColor: getRandomColor(),
        borderWidth: 1
    }));

    myChart = new Chart(ctx, {
        type: 'line',
        data: {
            labels: labels,
            datasets: datasets
        },
        options: {
            scales: {
                y: {
                    beginAtZero: true
                }
            }
        }
    });
}
function getRandomColor() {
    const r = Math.floor(Math.random() * 255);
    const g = Math.floor(Math.random() * 255);
    const b = Math.floor(Math.random() * 255);
    return `rgba(${r}, ${g}, ${b}, 1)`;
}
```

Figure 35: Codigo AJAX - Ejercicio 8

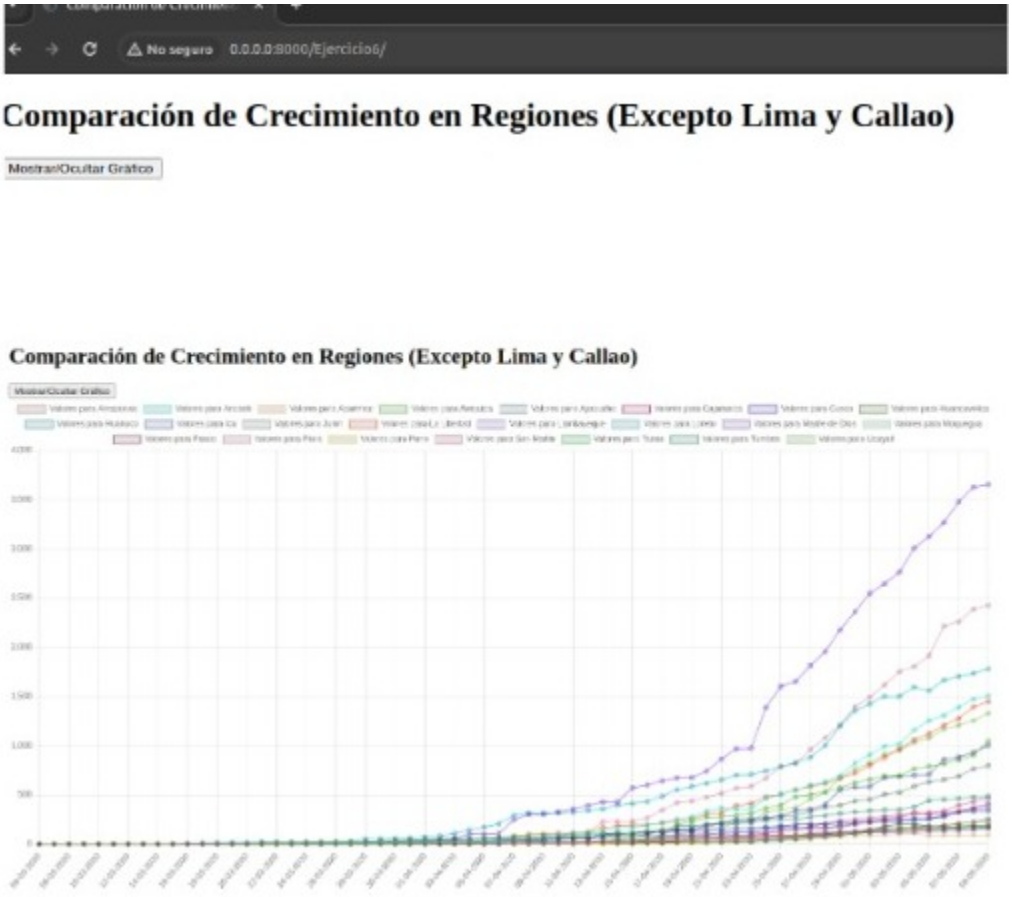


Figure 36: Pagina en Servidor - Ejercicio 8

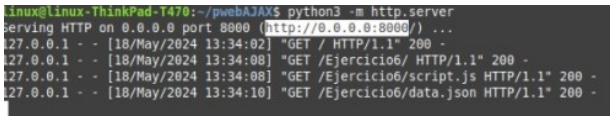


Figure 37: Servidor - Ejercicio 8

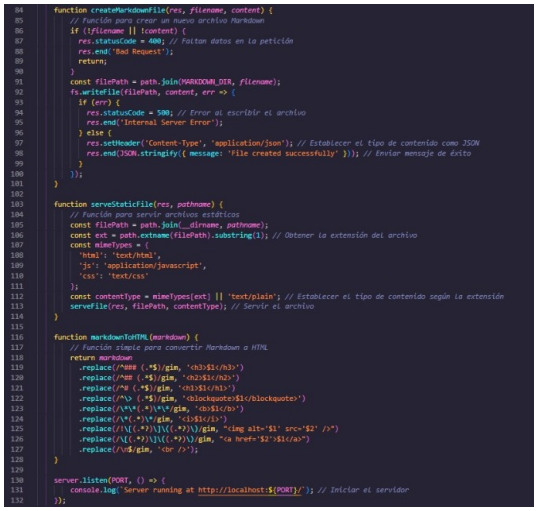
MarkDown

En esta segunda parte de la actividad, desarrollaremos una aplicación web que permitirá navegar por archivos Markdown. Esta aplicación tendrá la capacidad de listar los archivos Markdown disponibles, visualizar el contenido de un archivo Markdown traducido a HTML y crear nuevos archivos, almacenándolos en un servidor.

Para comenzar, necesitaremos configurar un servidor utilizando Node.js que brinde funcionalidades para interactuar con archivos Markdown. Para esto, inicialmente requeriremos los módulos 'http', 'fs', 'path' y 'url'. Estos módulos son esenciales para crear el servidor HTTP, interactuar con el sistema de archivos y manejar las rutas, respectivamente.
A continuación, definiremos un par de constantes importantes: 'PORT', que especificará el puerto en el cual el servidor estará a la escucha de las solicitudes, y 'MARKDOWNDIR', que indicará la ubicación donde se almacenarán los archivos Markdown.
Luego, procederemos a crear el servidor utilizando la función 'http.createServer()'. Esta función establecerá un servidor HTTP que ejecutará una función de callback cada vez que reciba una solicitud HTTP.
El manejo de estas solicitudes será diverso, principalmente para las solicitudes 'GET' a diferentes rutas. Estas solicitudes podrán listar los archivos Markdown disponibles, crear nuevos archivos Markdown y realizar otras operaciones relacionadas.
Además, se implementarán varias funciones de utilidad para manejar diferentes operaciones. Por ejemplo, 'serveFile()' servirá archivos al cliente, 'listMarkdownFiles()' listará los archivos disponibles en el directorio de Markdown, entre otras funciones esenciales para el funcionamiento del servidor.

Figure 38: Server Node.JS

Figure 39: Server Node.JS



```
function createMarkdownFile(res, filename, content) {
    // Función para crear un nuevo archivo Markdown
    if (!filename || !content) {
      res.statusCode = 400; // Faltan datos en la petición
      res.end('Bad Request');
      return;
    }
    const filePath = path.join(MARKDOWN_DIR, filename);
    fs.writeFile(filePath, content, err => {
      if (err) {
        res.statusCode = 500; // Error al escribir el archivo
        res.end('Internal Server Error');
      } else {
        res.setHeader('Content-Type', 'application/json'); // Establecer el tipo de contenido como JSON
        res.end(JSON.stringify({ message: 'File created successfully' })); // Enviar mensaje de éxito
      }
    });
}

function serveStaticFile(res, pathname) {
    // Función para servir archivos estáticos
    const filePath = path.join(__dirname, pathname);
    const ext = path.extname(filePath).substring(1); // Obtener la extensión del archivo
    const mimeTypes = {
      'html': 'text/html',
      'js': 'application/javascript',
      'css': 'text/css'
    };
    const contentType = mimeTypes[ext] || 'text/plain'; // Establecer el tipo de contenido según la extensión
    serveFile(res, filePath, contentType); // Servir el archivo
}

function markdownToHTML(markdown) {
    // Función simple para convertir Markdown a HTML
    return markdown
      .replace(/^### (.*$)/gim, '<h3>$1</h3>')
      .replace(/^## (.*$)/gim, '<h2>$1</h2>')
      .replace(/^# (.*$)/gim, '<h1>$1</h1>')
      .replace(/^\> (.*$)/gim, '<blockquote>$1</blockquote>')
      .replace(/\*\*(.*)\*\*/gim, '<b>$1</b>')
      .replace(/\*(.*)\*/gim, '<i>$1</i>')
      .replace(/!\[(.*?)\]\((.*?)\)/gim, "<img alt='$1' src='$2' />")
      .replace(/\[(.*?)\]\((.*?)\)/gim, "<a href='$2'>$1</a>")
      .replace(/\n$/gim, '<br />');
}

server.listen(PORT, () => {
    console.log(`Server running at http://localhost:${PORT}/`); // Iniciar el servidor
});
```

Figure 40: Server Node.JS

Este script implementa la lógica del lado del cliente para interactuar con el servidor y realizar operaciones como obtener, mostrar y crear archivos Markdown. El evento 'DOMContentLoaded' se activará una vez que el documento HTML haya sido completamente cargado y analizado, asegurando que el script pueda ejecutarse de manera adecuada.
Se seleccionan varios elementos del DOM utilizando el método 'getElementById'. Estos elementos representan la interfaz de usuario, como 'fileList' y 'fileContent', que se encargan de mostrar y representar el contenido de los archivos.
La función 'fetchFiles()' utiliza la API fetch para realizar una solicitud 'GET' al servidor y obtener una lista de archivos Markdown disponibles. Una vez recibida la respuesta del servidor, los archivos se muestran en la interfaz de usuario.
El evento 'submit' se activa cuando se envía un formulario para crear un nuevo archivo Markdown. Se recopilan los datos del formulario, como el nombre del archivo y su contenido, y se envían al servidor a través de una solicitud 'POST'. Esto permite la creación de nuevos archivos en el servidor.

```
public > JS main.js > ...
document.addEventListener('DOMContentLoaded', () => {
    const fileList = document.getElementById('fileList');
    const fileContent = document.getElementById('fileContent');
    const createFileForm = document.getElementById('createFileForm');
    const filenameInput = document.getElementById('filename');
    const fileContentInput = document.getElementById('fileContentInput');

    // Función para obtener la lista de archivos Markdown
    function fetchFiles() {
      fetch('/api/files')
        .then(response => response.json()) // Parsear la respuesta como JSON
        .then(files => {
          fileList.innerHTML = ''; // Limpiar la lista de archivos
          files.forEach(file => {
            const li = document.createElement('li');
            li.textContent = file;
            // Crear botón para ver el contenido del archivo
            const viewButton = document.createElement('button');
            viewButton.textContent = 'Ver contenido';
            viewButton.addEventListener('click', () => {
              fetchFileContent(file); // Añadir evento para cargar el contenido del archivo
            });
            li.appendChild(viewButton); // Añadir el botón al lado del nombre del archivo
            fileList.appendChild(li);
          });
        });
      }
      // Evento para crear un nuevo archivo Markdown
    createFileForm.addEventListener('submit', (e) => {
      e.preventDefault(); // Prevenir el envío del formulario por defecto
      const filename = filenameInput.value;
      const content = fileContentInput.value;

      fetch('/api/files', {
        method: 'POST',
        headers: {
          'Content-Type': 'application/json'
        },
        body: JSON.stringify({ filename, content }) // Enviar los datos como JSON
      })
        .then(response => response.json())
        .then(data => {
          alert(data.message); // Mostrar mensaje de éxito
          fetchFiles(); // Actualizar la lista de archivos
        });
      });

    fetchFiles(); // Cargar la lista de archivos al cargar la página
});
```

Figure 41: Script - Lado del cliente

```
public > index.html > html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Markdown</title>
</head>
<body>
    <h1>Markdown App</h1>
  <div>
    <h2>Archivos Markdown</h2>
    <ul id="fileList"></ul>
  </div>
  <div>
    <h2>Contenido del archivo</h2>
    <div id="fileContent"></div>
  </div>
  <div>
    <h2>Crear nuevo archivo</h2>
    <form id="createFileForm">
        <input type="text" id="filename" placeholder="Nombre del archivo (.md)" required>
        <textarea id="fileContentInput" placeholder="Contenido del archivo" required></textarea>
        <button type="submit">Crear Archivo</button>
      </form>
    </form>
  </div>
  <script src="main.js"></script>
</body>
</html>
```

Figure 42: Estructura HTML

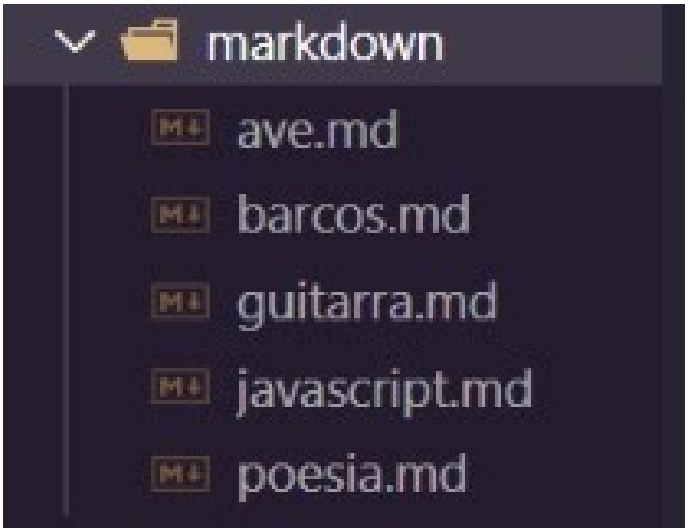


Figure 43: Archivos Markdown