Actividad : HTML, CSS, JavaScript

Consumo de API usando fetch

Parte 1: analizar el ejemplo en la carpeta ex0. Tenemos dos cajas de texto y dos botones que ejecutan una acción determinada.

1. Modificar el ejemplo para tener otro tipo de interacción con el usuario.
2. Modificar el archivo CSS y HTML para cambiar el aspecto visual de la página.

Parte 2: analizar el ejemplo proporcionado en la carpeta ex1. Usar la llave generada en la clase anterior para hacer peticiones a la API y poder visualizar los resultados.

1. Usando los elementos de la Parte 1, deben modificar el ejemplo proporcionado para que el usuario pueda digitar la ciudad y el lenguaje a usar
2. Modificar para que el usuario pueda seleccionar las unidades en que desea visualizar la temperatura: Centigrados °C o Farenheit °F
3. Crear una cuenta en la página: https://openweathermap.org generar una llave y consultar la documentación de la API. Consultar la documentación en https://openweathermap.org/current para adaptar la url y poder consultar el clima actual en una ciudad.
4. Copiar los archivos HTML, CSS y JS a una nueva carpeta y modificar el aplicativo para poder consumir datos desde **openweathermap.org**

Parte 3: analizar el ejemplo proporcionado en la carpeta ex2. Usar la llave de openweathermap para graficar el pronostico del clima.

1. Modificar el ejemplo de forma similar a como lo hizo en la parte 2 para que el usuario pueda ingresar la ciudad.
2. Copiar los archivos a una nueva carpeta y modificar el ejemplo para usar la api de weather-api.

Parte 4: API REST usando JAVA – Analizar el ejemplo presentado en la carpeta ex3.

El ejemplo se compone de dos partes principales:

1. **Servidor en Java**: Implementa la API REST y maneja las solicitudes HTTP.
2. **Cliente Web**: Consume la API utilizando HTML, CSS y JavaScript.

Servidor en Java

Archivo: `SimpleServer.java`

- **Creación del Servidor**: consultar sobre la clase `HttpServer`

```
HttpServer server = HttpServer.create(new InetSocketAddress(8000), 0);
server.createContext("/api/data", new DataHandler());
server.setExecutor(null);
server.start();
System.out.println("Server started on port 8000");
```

- Aquí creamos un servidor HTTP que escucha en el puerto 8000.
- `createContext("/api/data", new DataHandler())` define el endpoint `/api/data` y lo asigna a `DataHandler`.

Clase `DataHandler`:

- Esta clase maneja las solicitudes GET y POST.
- También maneja las solicitudes OPTIONS necesarias para CORS.

Método `handle`:

- **CORS**: Se configuran los encabezados para permitir solicitudes desde cualquier origen (`Access-Control-Allow-Origin: *`), métodos GET, POST y OPTIONS, y encabezados de tipo `Content-Type`.
- **OPTIONS**: Responde con un 204 (No Content) para las solicitudes OPTIONS, necesarias para CORS.
- **GET y POST**: Llama a métodos específicos para manejar estas solicitudes.

Método `handleGetRequest`:

Recopila todos los datos almacenados en `dataStore` y los envía como una respuesta JSON.

Método `handlePostRequest`:

Lee el cuerpo de la solicitud POST, lo guarda en `dataStore` y responde con un 201 (Created).

Cliente Web

Archivo: `index.html`

- **Formulario**: Permite al usuario ingresar datos y enviarlos al servidor.
- **Div `dataList`**: Muestra los datos recibidos del servidor.

Archivo: `styles.css`

- Estilos básicos para la página.

Archivo: `script.js`

- **Evento `submit`:**

    - Captura el evento de envío del formulario, previene la acción predeterminada y envía los datos a la API utilizando `fetch`.
    - Configura `fetch` para realizar una solicitud POST, incluyendo el encabezado `Content-Type` y el cuerpo de la solicitud como JSON.
    - Si la respuesta es exitosa (201), limpia el campo de entrada y actualiza la lista de datos llamando a `fetchData`.
- **Función `fetchData`**:
    - Realiza una solicitud GET a la API para obtener los datos almacenados.
    - Actualiza la lista de datos en el DOM.

Actividad para Estudiantes: Ampliando una API REST y su Cliente Web

Objetivo: Los estudiantes aprenderán sobre el desarrollo y ampliación de una API REST básica utilizando Java y cómo consumirla mediante un cliente web con HTML, CSS y JavaScript.

Tarea 1: Agregar funcionalidad DELETE a la API

1. **Modificar el servidor en Java para soportar DELETE**:
    - Agregar soporte para el método HTTP DELETE en `SimpleHttpServer.java`.
    - Permitir eliminar un dato específico del `dataStore` usando un ID.
2. **Modificar el cliente web para soportar DELETE**:
    - Agregar un botón de eliminación junto a cada dato en la lista.
    - Implementar una función JavaScript para enviar solicitudes DELETE a la API.

Tarea 2: Mejora de la interfaz de usuario

1. **Estilizar los botones de eliminación y el formulario**:

Consultar:

Express es un framework web minimalista para Node.js que simplifica enormemente el desarrollo de aplicaciones web y APIs REST. En el contexto del ejemplo anterior, el uso de Express podría haber facilitado significativamente la implementación del servidor. Express proporciona una interfaz más sencilla y estructurada para manejar rutas y métodos HTTP, permitiendo definir rutas para GET, POST, DELETE y otros métodos con menos código y de manera más legible. Además, Express gestiona automáticamente muchos aspectos del manejo de solicitudes, como el análisis del cuerpo de las solicitudes (body parsing) y la configuración de CORS (Cross-Origin Resource Sharing). Esto reduce la cantidad de código necesario y minimiza la posibilidad de errores, permitiendo a los desarrolladores centrarse en la lógica de la aplicación en lugar de los detalles de la infraestructura.

Temas clave para continuar el desarrollo de APIs usando Express

1. **Instalación y Configuración de Express**: Cómo instalar Express y configurar un proyecto básico.
2. **Manejo de Rutas y Métodos HTTP**: Definición de rutas para diferentes métodos HTTP (GET, POST, PUT, DELETE).
3. **Middleware en Express**: Uso de middleware para manejar autenticación, autorización, y otras tareas comunes.
4. **Gestión de Errores**: Cómo manejar y responder a errores en una aplicación Express.
5. **Body Parsing y Manejo de Datos**: Uso de `body-parser` y otros middleware para manejar datos de solicitud.
6. **Configuración de CORS**: Cómo configurar CORS para permitir o restringir el acceso a la API desde diferentes orígenes.
7. **Conexión a Bases de Datos**: Integración de Express con bases de datos como MongoDB, MySQL o PostgreSQL.
8. **Autenticación y Autorización**: Implementación de autenticación (JWT, OAuth) y autorización en aplicaciones Express.
9. **Testing de APIs**: Pruebas unitarias y de integración para asegurar la calidad de la API.
10. **Despliegue de Aplicaciones Express**: Buenas prácticas para desplegar aplicaciones Express en producción (uso de servicios en la nube como AWS, Heroku, etc.).