API rest:

¿Qué es una API?

Una interfaz de programa de aplicación (API) define las reglas que se deben seguir para comunicarse con otros sistemas de software.

Se puede pensar en una API web como una puerta de enlace entre los clientes y los recursos de la Web

### Clientes

Los clientes son usuarios que desean acceder a información desde la Web. El cliente puede ser una persona o un sistema de *software* que utiliza la API.

### Recursos

Los recursos son la información que diferentes aplicaciones proporcionan a sus clientes. Los recursos pueden ser imágenes, videos, texto, números o cualquier tipo de datos. La máquina encargada de entregar el recurso al cliente también recibe el nombre de servidor

## ¿Qué es REST?

La transferencia de estado representacional (REST) es una arquitectura de software que impone condiciones sobre cómo debe funcionar una API. En un principio, REST se creó como una guía para administrar la comunicación en una red compleja como Internet.

Las API que siguen el estilo arquitectónico de REST se llaman API REST. Los servicios web que implementan una arquitectura de REST son llamados servicios web RESTful. El término API RESTful suele referirse a las API web RESTful.

A continuación, se presentan algunos de los principios del estilo arquitectónico de REST:

### Interfaz uniforme

La interfaz uniforme es fundamental para el diseño de cualquier servicio web RESTful. Ella indica que el servidor transfiere información en un formato estándar. El recurso formateado se denomina representación en REST. Este formato puede ser diferente de la representación interna del recurso en la aplicación del servidor. Por ejemplo, el servidor puede almacenar los datos como texto, pero enviarlos en un formato de representación HTML.

La interfaz uniforme impone cuatro limitaciones de arquitectura:

1. Las solicitudes deben identificar los recursos. Lo hacen mediante el uso de un identificador uniforme de recursos.
2. Los clientes tienen información suficiente en la representación del recurso como para modificarlo o eliminarlo si lo desean. El servidor cumple esta condición por medio del envío de los metadatos que describen el recurso con mayor detalle.
3. Los clientes reciben información sobre cómo seguir procesando la representación. El servidor logra esto enviando mensajes autodescriptivos que contienen metadatos sobre cómo el cliente puede utilizarlos de mejor manera.
4. Los clientes reciben información sobre todos los demás recursos relacionados que necesitan para completar una tarea. El servidor logra esto enviando hipervínculos en la representación para que los clientes puedan descubrir dinámicamente más recursos.

### Tecnología sin estado

En la arquitectura de REST, la tecnología sin estado se refiere a un método de comunicación en el cual el servidor completa todas las solicitudes del cliente independientemente de todas las solicitudes anteriores. Los clientes pueden solicitar recursos en cualquier orden, y todas las solicitudes son sin estado o están aisladas del resto. Esta limitación del diseño de la API REST implica que el servidor puede comprender y cumplir por completo la solicitud todas las veces.

### Sistema por capas

En una arquitectura de sistema por capas, el cliente puede conectarse con otros intermediarios autorizados entre el cliente y el servidor y todavía recibirá respuestas del servidor. Los servidores también pueden pasar las solicitudes a otros servidores. Es posible diseñar el servicio web RESTful para que se ejecute en varios servidores con múltiples capas, como la seguridad, la aplicación y la lógica empresarial, que trabajan juntas para cumplir las solicitudes de los clientes. Estas capas se mantienen invisibles para el cliente.

### Almacenamiento en caché

Los servicios web RESTful admiten el almacenamiento en caché, que es el proceso de almacenar algunas respuestas en la memoria caché del cliente o de un intermediario para mejorar el tiempo de respuesta del servidor. Por ejemplo, suponga que visita un sitio web que tiene imágenes comunes en el encabezado y el pie de página en todas las páginas. Cada vez que visita una nueva página del sitio web, el servidor debe volver a enviar las mismas imágenes. Para evitar esto, el cliente guarda en la memoria caché o almacena estas imágenes después de la primera respuesta y, luego, utiliza las imágenes directamente desde la memoria caché. Los servicios web RESTful controlan el almacenamiento en caché mediante el uso de respuestas de la API que se pueden guardar en la memoria caché o no.

### Código bajo demanda

En el estilo de arquitectura de REST, los servidores pueden extender o personalizar temporalmente la funcionalidad del cliente transfiriendo a este el código de programación del *software*. Por ejemplo, cuando completa un formulario de inscripción en cualquier sitio web, su navegador resalta de inmediato cualquier error que usted comete, como un número de teléfono incorrecto. El navegador puede hacer esto gracias al código enviado por el servidor.

¿Cómo funcionan las API RESTful?

La función básica de una API RESTful es la misma que navegar por Internet. Cuando requiere un recurso, el cliente se pone en contacto con el servidor mediante la API. Los desarrolladores de API explican cómo el cliente debe utilizar la API REST en la documentación de la API de la aplicación del servidor. A continuación, se indican los pasos generales para cualquier llamada a la API REST:

1. El cliente envía una solicitud al servidor. El cliente sigue la documentación de la API para dar formato a la solicitud de una manera que el servidor comprenda.
2. El servidor autentica al cliente y confirma que este tiene el derecho de hacer dicha solicitud.
3. El servidor recibe la solicitud y la procesa internamente.
4. Luego, devuelve una respuesta al cliente. Esta respuesta contiene información que dice al cliente si la solicitud se procesó de manera correcta. La respuesta también incluye cualquier información que el cliente haya solicitado.

Los detalles de la solicitud y la respuesta de la API REST varían un poco en función de cómo los desarrolladores de la API la hayan diseñado.

## ¿Qué contiene la solicitud del cliente de la API RESTful?

Las API RESTful requieren que las solicitudes contengan los siguientes componentes principales:

### Identificador único de recursos

El servidor identifica cada recurso con identificadores únicos de recursos. En los servicios REST, el servidor por lo general identifica los recursos mediante el uso de un localizador uniforme de recursos (URL). El URL especifica la ruta hacia el recurso. Un URL es similar a la dirección de un sitio web que se ingresa al navegador para visitar cualquier página web. El URL también se denomina punto de conexión de la solicitud y especifica con claridad al servidor qué requiere el cliente.

### Método

Los desarrolladores a menudo implementan API RESTful mediante el uso del protocolo de transferencia de hipertexto (HTTP). Un método de HTTP informa al servidor lo que debe hacer con el recurso. A continuación, se indican cuatro métodos de HTTP comunes:

***GET***

Los clientes utilizan GET para acceder a los recursos que están ubicados en el URL especificado en el servidor. Pueden almacenar en caché las solicitudes GET y enviar parámetros en la solicitud de la API RESTful para indicar al servidor que filtre los datos antes de enviarlos.

***POST***

Los clientes usan POST para enviar datos al servidor. Incluyen la representación de los datos con la solicitud. Enviar la misma solicitud POST varias veces produce el efecto secundario de crear el mismo recurso varias veces.

***PUT***

Los clientes utilizan PUT para actualizar los recursos existentes en el servidor. A diferencia de POST, el envío de la misma solicitud PUT varias veces en un servicio web RESTful da el mismo resultado.

***DELETE***

Los clientes utilizan la solicitud DELETE para eliminar el recurso. Una solicitud DELETE puede cambiar el estado del servidor. Sin embargo, si el usuario no cuenta con la autenticación adecuada, la solicitud fallará.

### Encabezados de HTTP

Los encabezados de solicitudes son los metadatos que se intercambian entre el cliente y el servidor. Por ejemplo, el encabezado de la solicitud indica el formato de la solicitud y la respuesta, proporciona información sobre el estado de la solicitud, etc.

***Datos***

Las solicitudes de la API REST pueden incluir datos para que los métodos POST, PUT y otros métodos HTTP funcionen de manera correcta.

***Parámetros***

Las solicitudes de la API RESTful pueden incluir parámetros que brindan al servidor más detalles sobre lo que se debe hacer. A continuación, se indican algunos tipos de parámetros diferentes:

* Los parámetros de ruta especifican los detalles del URL.
* Los parámetros de consulta solicitan más información acerca del recurso.
* Los parámetros de cookie autentican a los clientes con rapidez.

## ¿Qué son los métodos de autenticación de la API RESTful?

Un servicio web RESTful debe autenticar las solicitudes antes de poder enviar una respuesta. La autenticación es el proceso de verificar una identidad. Por ejemplo, puede demostrar su identidad mostrando una tarjeta de identificación o una licencia de conducir. De forma similar, los clientes de los servicios RESTful deben demostrar su identidad al servidor para establecer confianza.

La API RESTful tiene cuatro métodos comunes de autenticación:

### Autenticación HTTP

HTTP define algunos esquemas de autenticación que se pueden utilizar directamente cuando se implementa la API REST. A continuación, se indican dos de estos esquemas:

***Autenticación básica***

En la autenticación básica, el cliente envía el nombre y la contraseña del usuario en el encabezado de la solicitud. Los codifica con base64, que es una técnica de codificación que convierte el par en un conjunto de 64 caracteres para su transmisión segura.

***Autenticación del portador***

El término autenticación del portador se refiere al proceso de brindar el control de acceso al portador del *token*. El *token* del portador suele ser una cadena de caracteres cifrada que genera el servidor como respuesta a una solicitud de inicio de sesión. El cliente envía el *token* en los encabezados de la solicitud para acceder a los recursos.

### Claves de la API

Las claves de la API son otra opción para la autenticación de la API REST. En este enfoque, el servidor asigna un valor único generado a un cliente por primera vez. Cada vez que el cliente intenta acceder a los recursos, utiliza la clave de API única para su verificación. Las claves de API son menos seguras debido a que el cliente debe transmitir la clave, lo que la vuelve vulnerable al robo de red.

### OAuth

OAuth combina contraseñas y *tokens* para el acceso de inicio de sesión de alta seguridad a cualquier sistema. El servidor primero solicita una contraseña y luego solicita un *token* adicional para completar el proceso de autorización. Puede verificar el token en cualquier momento y, también, a lo largo del tiempo, con un alcance y duración específicos

## Qué contiene la respuesta del servidor de la API RESTful?

Los principios de REST requieren que la respuesta del servidor contenga los siguientes componentes principales:

### Línea de estado

La línea de estado contiene un código de estado de tres dígitos que comunica si la solicitud se procesó de manera correcta o dio error. Por ejemplo, los códigos 2XX indican el procesamiento correcto, pero los códigos 4XX y 5XX indican errores. Los códigos 3XX indican la redirección de URL.

A continuación, se enumeran algunos códigos de estado comunes:

* 200: respuesta genérica de procesamiento correcto
* 201: respuesta de procesamiento correcto del método POST
* 400: respuesta incorrecta que el servidor no puede procesar
* 404: recurso no encontrado

### Cuerpo del mensaje

El cuerpo de la respuesta contiene la representación del recurso. El servidor selecciona un formato de representación adecuado en función de lo que contienen los encabezados de la solicitud. Los clientes pueden solicitar información en los formatos XML o JSON, lo que define cómo se escriben los datos en texto sin formato. Por ejemplo, si el cliente solicita el nombre y la edad de una persona llamada John, el servidor devuelve una representación JSON como la siguiente:

'{"name":"John", "age":30}'

### Encabezados

La respuesta también contiene encabezados o metadatos acerca de la respuesta. Estos brindan más contexto sobre la respuesta e incluyen información como el servidor, la codificación, la fecha y el tipo de contenido.

Mysql2: npm i mysql2

Importar el método createPool: nos brinda una mejor performance a la hora de crear la conexión con la base de datos. Nos permite establecer un conjunto de conexiones que podemos reutilizar.

Este método recibe un objeto con los datos de la conexión.

Ejemplo: db.js

import {createPool} from 'mysql2/promise';

const pool = createPool({

    host:'localhost',

    user:'root',

    password:'',

    port:'3306',

    database:'node\_prueba'

});

export {pool};

Ejemplo: app.js

import express from 'express';

import {pool} from './db.js';

const app = express();

app.get('/', async (req,res) => {

    const [result] = await pool.query('SELECT \* FROM categorias');

    res.json(result);

})

app.listen(3000);

console.log('servidor corriendo');

Enrutador: vamos a modularizar los componentes de nuestra app, es decir, ordenamos en diferentes carpetas estos componentes, por ejemplo, las rutas.

Ejemplo: routes/empleados.routes.js

import {Router} from 'express';

const router = Router();

router.get('/empleados', (req,res) => {

    res.send('listado de empleados')

})

router.post('/empleados', (req,res) => {

    res.send('creacion de empleado')

})

router.put('/empleados', (req,res) => {

    res.send('modificacion de empleado')

})

router.delete('/empleados', (req,res) => {

    res.send('quitar empleado')

})

export default router;

ejemplo app.js:

import express from 'express';

import {pool} from './db.js';

import empleadosEnrutador from './routes/empleados.routes.js';

const app = express();

app.use(empleadosEnrutador);

app.listen(3000);

console.log('servidor corriendo');

Ejemplo para vincular un enrutador con la base de datos y poder ejecutar consultas:

import {Router} from 'express';

import {pool} from '../db.js';

const router = Router();

router.get('/', async (req,res) => {

    const [result] = await pool.query('SELECT \* FROM categorias');

    res.json(result);

})

export default router;

Luego llamamos al enrutador desde app:

import express from 'express';

import indexEnrutador from './routes/index.routes.js'

import empleadosEnrutador from './routes/empleados.routes.js';

const app = express();

app.use(empleadosEnrutador);

app.use(indexEnrutador);

app.listen(3000);

console.log('servidor corriendo');

**Controladores:**

En este caso lo utilizamos para modularizar las funciones a ejecutar por el enrutador.

Ejemplo: controllers/empleados.controller.js

export const listarEmpleados = (req,res) => {

    res.send('listado de empleados')

};

export const crearEmpleado = (req,res) => {

    res.send('creación de empleados')

};

export const modificarEmpleado = (req,res) => {

    res.send('actuaalizar de empleados')

};

export const eliminarEmpleado = (req,res) => {

    res.send('eliminar de empleados')

};

Ejemplo: routes/empleados.routes.js:

import {Router} from 'express';

import {listarEmpleados, crearEmpleado, modificarEmpleado, eliminarEmpleado}  from '../controllers/empleados.controllers.js';

const router = Router();

router.get('/empleados', listarEmpleados)

router.post('/empleados', crearEmpleado)

router.put('/empleados', modificarEmpleado)

router.delete('/empleados', eliminarEmpleado)

export default router;

Para poder modularizar una función que utiliza la conexión con la base de datos hacemos lo siguiente (importo el modulo de mysql2 en este nuevo controlador, para poder ejecutar las consultas a la base de datos.):

Ejemplo: controllers/index.controller.js:

import {pool} from '../db.js';

export const listarCategorias = async (req,res) => {

    const [result] = await pool.query('SELECT \* FROM categorias');

    res.json(result);

}

Ejemplo: routes/index.routes.js

import {Router} from 'express';

import { listarCategorias } from '../controllers/index.controller.js';

const router = Router();

router.get('/', listarCategorias)

export default router;

**Configuracion CRUD**

empleados.controllers.js:

import {pool} from '../db.js';

export const listarEmpleados = async (req,res) => {

    try{

        const [rows] = await pool.query('SELECT \* FROM empleados');

        res.json(rows);

    }catch(error){

        return res.status(500).json({message : 'Ha ocurrido un error'});

    }

};

export const mostrarEmpleado = async(req,res) => {

    const id\_cliente = req.params.id;

    try {

        const [rows] = await pool.query('SELECT \* FROM empleados WHERE id\_empleado = ?', [id\_cliente]);

        if(rows.length > 0){

            res.json(rows[0]);

        }else{

            res.status(404).send('No se han  encontrado empleados con ese ID');

        }

    } catch (error) {

        return res.status(500).json({message : 'Ha ocurrido un error'});

    }

}

export const crearEmpleado = async (req,res) => {

         // CAPTURAMOS LOS DATOS QUE NOS ENVIAN A TRAVES DEL REQUEST

    // LO IDEAL ES VALIDAR LOS DATOS ANTES DE INSERTARLOS.

    const {nombre,telefono,direccion,categoria} = req.body;

    try {

    const [rows] = await pool.query('INSERT INTO empleados (nombre\_completo, direccion,telefono,id\_categoria) VALUES (?, ? , ?, ?)',

    [nombre, direccion, telefono, categoria]);

    // DEVUELVO LOS DATOS DEL EMPLEADO CREEADO, MAS EL ID GENERADO PARA ESE REGISTRO.

    res.send({

        id: rows.insertId,

        nombre,

        direccion,

        telefono,

        categoria

    });

    } catch (error) {

        return res.status(500).json({message : 'Ha ocurrido un error'});

    }

};

export const modificarEmpleado = async (req,res) => {

    const {id} = req.params;

    const {nombre,telefono,direccion,categoria} = req.body;

    try {

       const [result] = await pool.query('UPDATE empleados SET nombre\_completo = IFNULL(?, nombre\_completo), direccion =  IFNULL(?, direccion), telefono =  IFNULL(?, telefono), id\_categoria = ? WHERE id\_empleado = ?', [nombre,direccion,telefono,categoria,id]);

        console.log(result);

        if(result.affectedRows > 0) {

            const [rows] = await pool.query('SELECT \* FROM empleados WHERE id\_empleado = ?', [id]);

            res.json(rows[0]);

        }else{

            res.status(404).json({message: "Empleado no encontrado"});

        }

    } catch (error) {

        return res.status(500).json({message : 'Ha ocurrido un error'});

    }

};

export const eliminarEmpleado = async (req,res) => {

    try {

        const id\_cliente = req.params.id;

        const [result] = await pool.query('DELETE FROM empleados WHERE id\_empleado = ?', [id\_cliente]);

        console.log(result);

        if(result.affectedRows > 0){

            res.send('Empleado eliminado');

        }else{

            res.status(404).send('Error:No eliminado');

        }

    } catch (error) {

        return res.status(500).json({message : 'Ha ocurrido un error'});

    }

};

empleados.routes.js

import {Router} from 'express';

import {listarEmpleados,mostrarEmpleado, crearEmpleado, modificarEmpleado, eliminarEmpleado}  from '../controllers/empleados.controllers.js';

const router = Router();

router.get('/empleados', listarEmpleados)

router.get('/empleados/:id', mostrarEmpleado)

router.post('/empleados', crearEmpleado)

// PUT ES PARA MODIFICAR EL REGISTRO COMPLETO

//router.put('/empleados/:id', modificarEmpleado)

//PATCH ES PARA MODIFICAR ALGUNOS DE LOS DATOS DEL REGISTRO.

router.patch('/empleados/:id', modificarEmpleado)

router.delete('/empleados/:id', eliminarEmpleado)

export default router;

app.js:

import express from 'express';

import indexEnrutador from './routes/index.routes.js'

import empleadosEnrutador from './routes/empleados.routes.js';

const app = express();

app.use(express.json());

// EJ. Para anteponer una ruta antes que otras. Ejemplo: api/empleados

app.use('/api',empleadosEnrutador);

app.use(indexEnrutador);

//PAGE NOT FOUND

app.use((req,res,next) => {

    res.status(404).json({message:'ENDPOINT NOT FOUND'})

})

app.listen(3000);

console.log('servidor corriendo');

Variables de entorno: son variables que puedo utilizar en cualquier parte dentro mi aplicación.

Para eso en la raíz del proyecto creo un archivo .env que es el cual va a contener estas variables de la siguiente manera:

PORT = 3000

DB\_HOST = localhost

DB\_PORT = 3306

DB\_USER = root

DB\_PASSWORD =

DB\_NAME = node\_prueba

Luego dentro de la carpeta src (donde está app.js), creo un archivo llamado config.js (previamente hay que instalar dotenv)

import {config} from 'dotenv';

config();

console.log(process.env.db\_host);

Ahora vamos a utilizar las variables de entorno para configurar la conexión con la base de datos, entonces:

.env:

PORT = 3000

DB\_HOST = localhost

DB\_PORT = 3306

DB\_USER = root

DB\_PASSWORD =

DB\_NAME = node\_prueba

Config.js:

import {config} from 'dotenv';

config();

export const PUERTO = process.env.port || 3000;

export const DB\_HOST = process.env.db\_host || 'localhost';

export const DB\_PORT = process.env.db\_port || 3306;

export const DB\_USER = process.env.db\_user || 'root';

export const DB\_PASS = process.env.db\_password || '';

export const DB\_NAME = process.env.db\_name || '';

db.js:

import {createPool} from 'mysql2/promise';

import {DB\_PORT,DB\_HOST,DB\_USER,DB\_NAME,DB\_PASS} from './config.js'

const pool = createPool({

    host:DB\_HOST,

    user:DB\_USER,

    password:DB\_PASS,

    port:DB\_PORT,

    database:DB\_NAME

});

export {pool};

Subir el proyecto en RENDER:

Build command: npm install

Start command: npm run start

Vincularlo con github

Crear una base de datos en la nube usar clever-cloud.com -> user tumbe.manu pass manu123123

Aclaracion: Al intentar colocar las variables de entorno dentro del servidor, no me permitia realizar la conexión con la base de datos. Al ponerlas de forma estática, funciono. Revisar.