«Talento Tech»

Desarrollo Web 4

Clase 04

Buenos Aires Ciudad

Buenos Aires aprende
Agencia de Habilidades para el Futuro

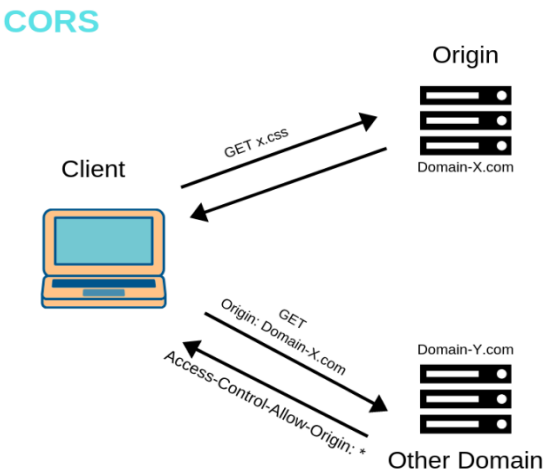
Clase N° 4 | Gestor XAMPP y CRUD

Temario:

- Implementación de CORS (¿Qué es y para qué sirve?)
- Instalación y configuración de XAMPP
- ¿Qué es SQL?
- Creación de un CRUD implementando MYSQL

Buenos Aires aprende
Agencia de Habilidades para el Futuro

Implementación de CORS



¿Qué es CORS (Cross-Origin Resource Sharing)?

CORS es una política de seguridad implementada por los navegadores web para controlar cómo las páginas web en un dominio (origen) pueden solicitar recursos (como datos o scripts) desde otro dominio.

La política de mismo origen (same-origin policy) restringe, por defecto, las solicitudes de recursos desde un origen diferente al de la propia página web.

Principales Características de CORS

Política de Mismo Origen
La política de mismo origen impide que una página web realice solicitudes a otro dominio diferente al de la propia página, por razones de seguridad.

Necesidad de Solicitar Recursos Externos

Buenos Aires aprende
Agencia de Habilidades para el Futuro

A menudo, las aplicaciones web necesitan solicitar recursos (como datos de una API) desde dominios diferentes al de la propia aplicación.

Permisos Controlados por el Navegador
CORS permite a los servidores especificar qué orígenes tienen permiso para acceder a sus recursos. Los navegadores controlan y aplican estas reglas.

Encabezados HTTP Específicos
En una solicitud CORS, el navegador agrega encabezados HTTP específicos, como Origin, para indicar el origen de la solicitud. El servidor puede responder con encabezados que permitan o denieguen el acceso (Access-Control-Allow-Origin, Access-Control-Allow-Methods, etc.).

¿Cómo funciona CORS?

Solicitud Simple
Si una solicitud HTTP es considerada "simple" (como GET o POST con ciertas condiciones), el navegador realiza la solicitud y espera que el servidor incluya el encabezado Access-Control-Allow-Origin en la respuesta.

Solicitud Prefabricada
Si la solicitud es considerada "no simple" (por ejemplo, si usa ciertos encabezados personalizados o métodos como DELETE), el navegador realiza una solicitud "prefabricada" (preflight) OPTIONS antes de la solicitud real. El servidor responde indicando si se permiten las solicitudes reales.

Manejo de Errores
Si el servidor no permite el acceso desde el origen de la página web, el navegador bloquea la solicitud y genera un error CORS

```
Access-Control-Allow-Origin: https://dominio-permitido.com
```

Buenos Aires aprende
Agencia de Habilidades para el Futuro

Implementando CORS en Express.js

Para poder utilizar CORS en express.js vamos a tener que instalar la librería con el siguiente comando:

```
npm i cors
```

Cors lo vamos a implementar como un Middleware y vamos a dejar su configuración para que cualquier cliente pueda acceder a nuestro servidor de la siguiente manera:

```
app.use(cors());
```

A continuación vamos a ver un ejemplo integrador de un servidor donde implementaremos cors:

```
const express = require('express');
const bodyParser = require('body-parser');
const cors = require('cors');

const app = express();

// Middleware para habilitar CORS para cualquier origen
app.use(cors());

// Middleware para analizar el cuerpo de las solicitudes en formato
JSON
app.use(bodyParser.json());

// Variable global que almacena un array de objetos
let datos = [
  { id: 1, nombre: 'Objeto 1' },
  { id: 2, nombre: 'Objeto 2' },
  // Puedes agregar más objetos según sea necesario
```

```
];

// Endpoint GET que devuelve el array de objetos en formato JSON
app.get('/', (req, res) => {
  res.json(datos);
});

// Endpoint POST para agregar un nuevo objeto al array
app.post('/', (req, res) => {
  const nuevoObjeto = req.body;
  nuevoObjeto.id = datos.length + 1;
  datos.push(nuevoObjeto);
  res.json(datos);
});

// Puerto en el que el servidor escucha las solicitudes
const puerto = 3000;
app.listen(puerto, () => {
  console.log(`Servidor escuchando en el puerto ${puerto}`);
});
```

A continuación crearemos un html que va a consumir la información de éste servidor. Recordá que es muy importante que para poder consumir desde el front esta información deberás tener el servidor ejecutándose con el comando node index.js en la consola de tu servidor.

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width,
initial-scale=1.0">
  <title>Consumir Datos del Servidor</title>
```

```
</head>
<body>

  <h1>Datos del Servidor:</h1>
  <ul id="datosList"></ul>

  <script>
    // URL del servidor
    const serverURL = 'http://localhost:3000';

    // Elemento en el que se mostrarán los datos
    const datosList = document.getElementById('datosList');

    // Función para obtener y mostrar los datos del servidor
    async function obtenerDatos() {
      try {
        // Realizar solicitud GET al servidor
        const respuesta = await fetch(`${serverURL}/`);

        // Verificar si la solicitud fue exitosa (código 200)
        if (respuesta.ok) {
          // Obtener datos en formato JSON
          const datos = await respuesta.json();

          // Mostrar los datos en la lista
          datos.forEach(dato => {
            const listItem = document.createElement('li');
            listItem.textContent = `${dato.id}: ${dato.nombre}`;
            datosList.appendChild(listItem);
          });
        } else {
          console.error('Error al obtener datos del servidor:',
respuesta.statusText);
        }
```

```
      } catch (error) {
        console.error('Error en la solicitud:', error);
      }
    }

    // Llamar a la función para obtener y mostrar los datos
    obtenerDatos();
  </script>
</body>
</html>
```

Al ejecutar el front desde localhost vas a poder ver la información proveniente del backend impresa en el frontend. ¿No es increíble?

Instalación y configuración de XAMPP

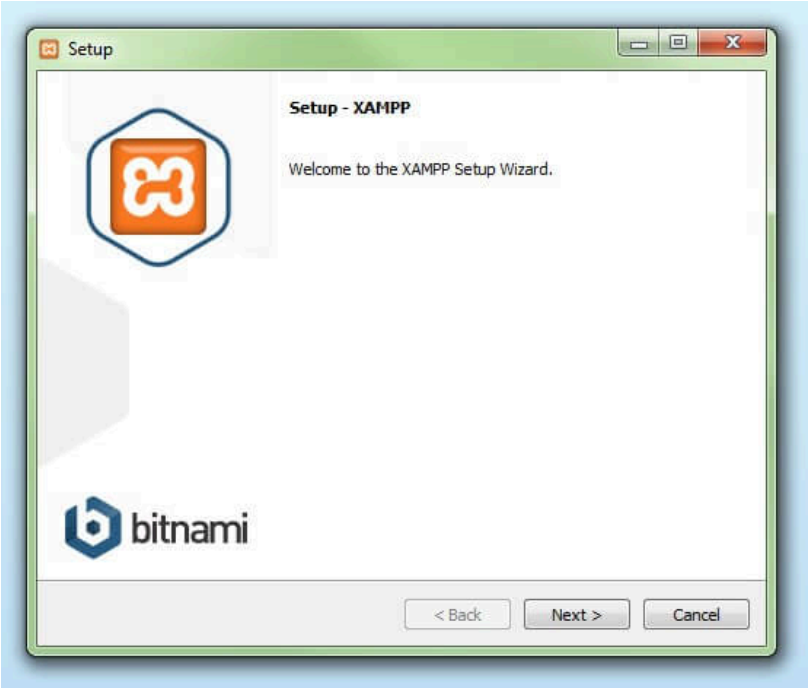
A la hora de trabajar con una **BD (Base de Datos/Database),** vamos a necesitar un gestor que nos permita interactuar con todos los datos. a su vez, necesitamos un servidor en el que corra la BD.

Buenos Aires aprende
Agencia de Habilidades para el Futuro

XAMPP es una distribución de **Apache** que incluye varios software libres. El nombre es un acrónimo compuesto por las iniciales de los programas que lo constituyen: el servidor web Apache, los sistemas relacionales de administración de bases de datos **MySQL y MariaDB**, así como los lenguajes de programación **Perl y PHP**. La inicial X se usa para representar a los sistemas operativos **Linux, Windows y Mac OS X.**
Lo que nos va a interesar a nosotros es trabajar con las **BD, MySQL y MariaDB.**
MySQL (se pronuncia MAY-ES-KYU-EL) es un sistema de gestión de bases de datos relacionales de código abierto. Es una de las más populares, junto a Oracle y Microsoft **SQL Server**, todo para entornos de desarrollo web.
MariaDB es un sistema de gestión de bases de datos que deriva de **MySQL.**

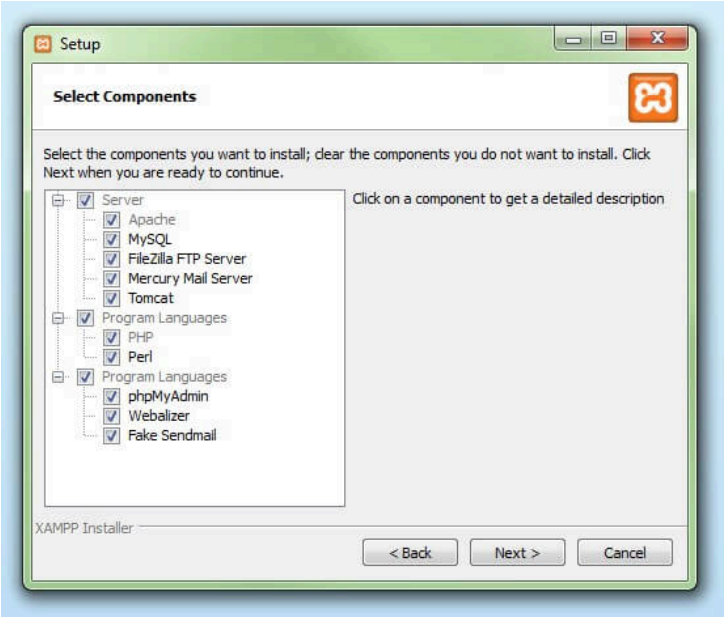
Integrando SQL en el proyecto con XAMPP

Lo primero que vamos a hacer es descargar el entorno de **XAMPP** en nuestro ordenador. Para esto, iremos a la siguiente url: **https://www.apachefriends.org/es/download.html**
Podemos descargarlo para **Windows, Linux y MAC OS.**

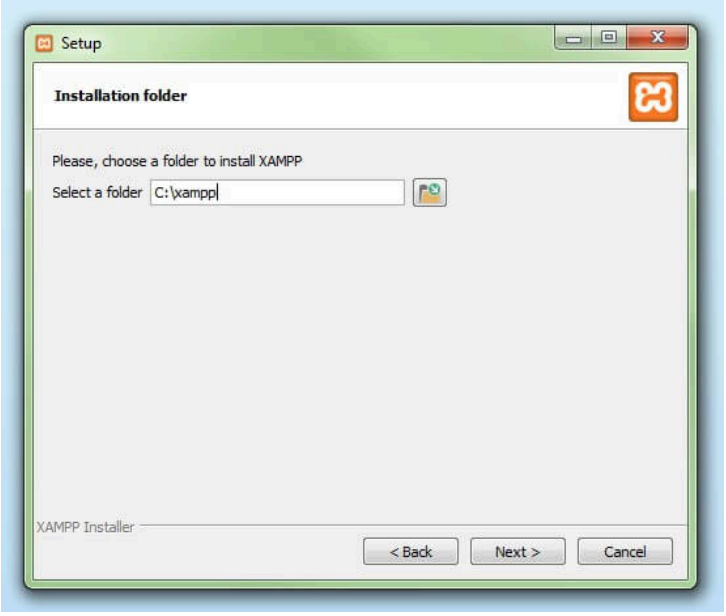


Paso 1:
Una vez descargado, ejecutamos el **.exe** , aparece la pantalla de inicio del asistente para instalar **XAMPP**. Para ajustar las configuraciones de la instalación se hace clic en **“Next”.**

Buenos Aires aprende
Agencia de Habilidades para el Futuro



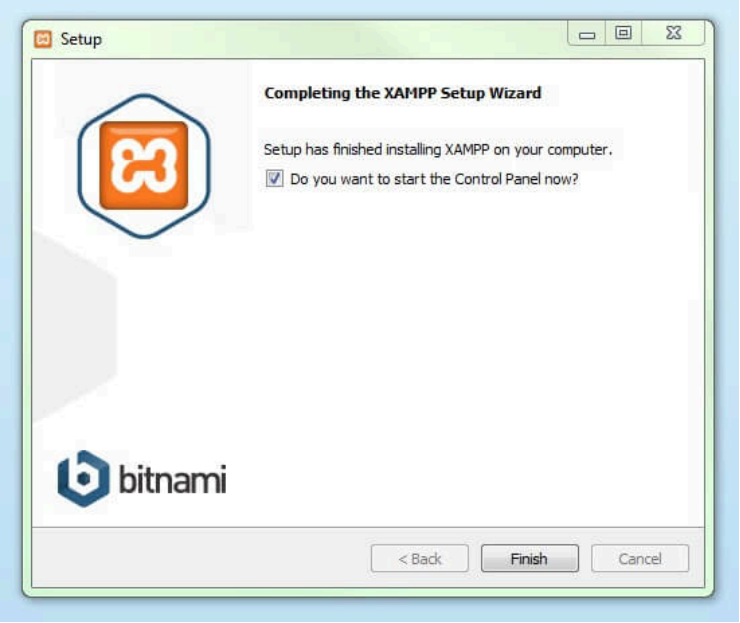
Paso 2:
Luego nos vamos a encontrar con los componentes que ya vienen por defecto. Los dejamos así y ponemos **Next.**



Paso 3:
En el próximo paso elegimos la carpeta dónde queremos instalarlo. Luego, ponemos Next.

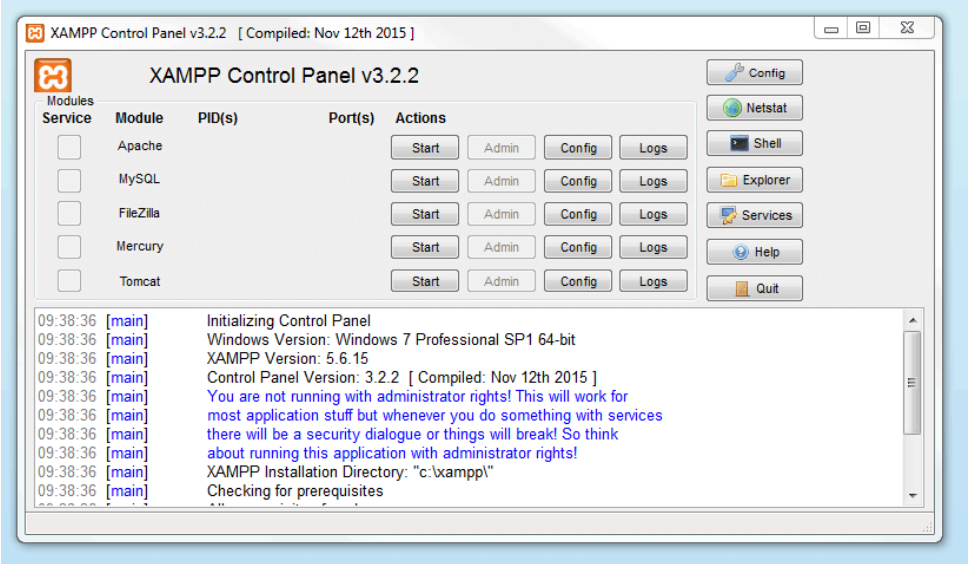
Buenos Aires aprende
Agencia de Habilidades para el Futuro

Paso 4:
Ahora comienza el proceso de instalación, que puede durar varios minutos.

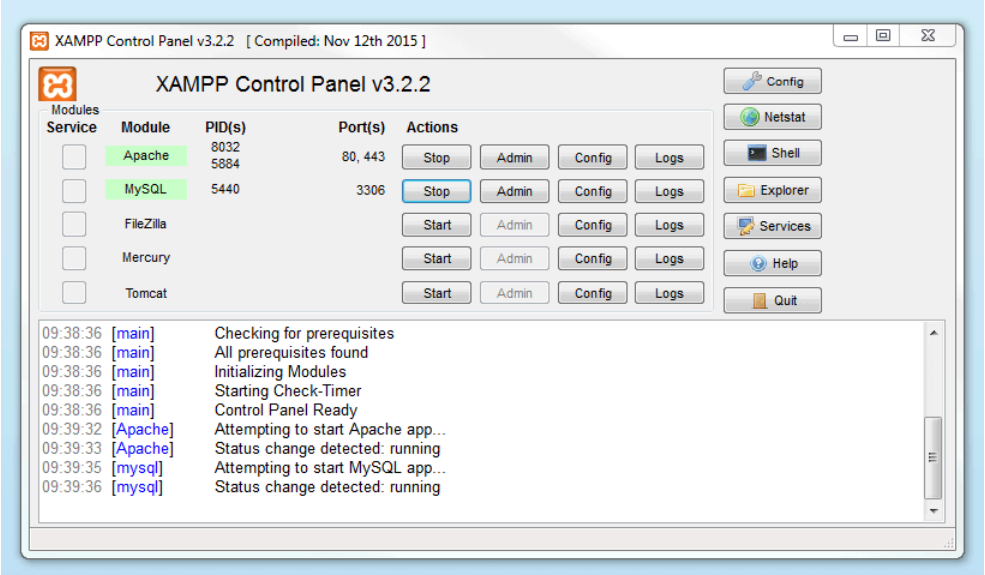


Paso 5:
Una vez finalizado, abrimos el XAMPP y nos encontraremos con la siguiente terminal:

Buenos Aires aprende
Agencia de Habilidades para el Futuro

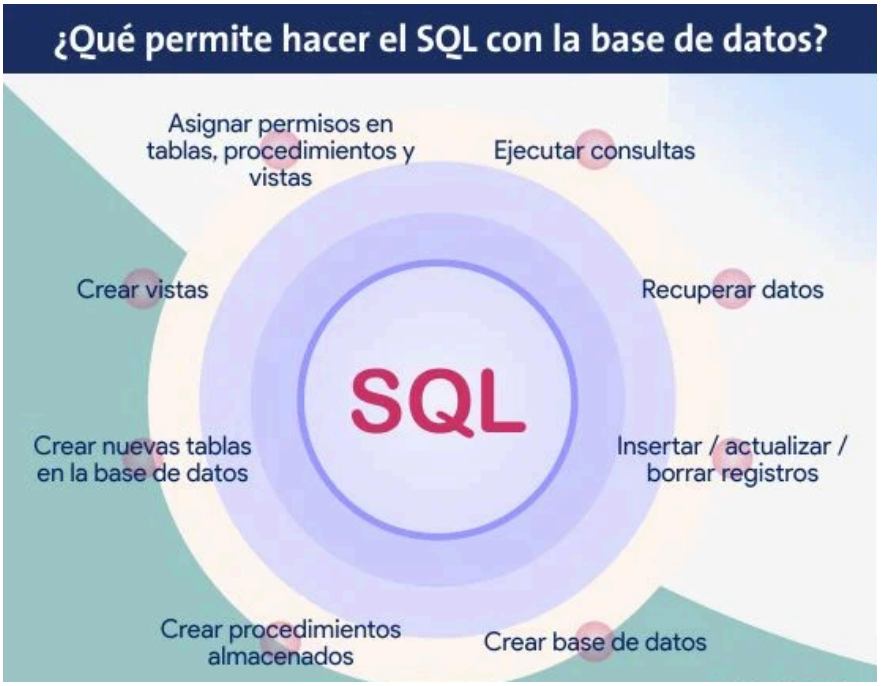


Paso 6:
Cada vez que trabajemos con el **Xampp**, dejaremos en la opción **“Start”** los módulos Apache y MySQL, como podemos ver en la siguiente imagen:



Buenos Aires aprende
Agencia de Habilidades para el Futuro

SQL (Structured Query Language)



Las bases de datos relacionales se basan en el modelo relacional, una forma intuitiva y directa de representar datos en tablas. En una base de datos relacional, cada fila de la tabla es un registro con un **ID** único llamado clave. Las columnas de la tabla contienen atributos de los datos, y cada registro generalmente tiene un valor para cada atributo, lo que facilita el establecimiento de las relaciones entre los puntos de datos. De tal manera que una de las principales características de la base de datos relacional es evitar la duplicidad de registros y a su vez garantizar la integridad referencial, es decir, que si se elimina uno de los registros, la integridad de los registros restantes no será afectada. Además, gracias a las claves se puede acceder de forma sencilla a la información y recuperarla en cualquier momento.

Dentro de este tipo de **BD**, se utiliza un lenguaje para interactuar con ellas, llamado **SQL (Structured Query Language o Lenguaje de Consultas Estructurado)**. Este lenguaje nos sirve para trabajar con los datos, ya sea crearlos, modificarlos o eliminarlos, entre muchas

otras cosas. Es importante entender que **SQL no es un lenguaje de programación**, ya que no nos permite, por ejemplo, crear variables, funciones, condicionales o repeticiones.

Comandos SQL Básicos

- **SELECT:** Utilizado para recuperar datos de una o varias tablas.
- **INSERT:** Permite agregar nuevos registros a una tabla.
- **UPDATE:** Utilizado para modificar registros existentes en una tabla.
- **DELETE:** Elimina registros de una tabla.
- **CREATE:** Se emplea para crear nuevas tablas, índices u otros objetos en la base de datos.
- **ALTER:** Modifica la estructura de una tabla existente.
- **DROP:** Elimina objetos de la base de datos, como tablas o índices.

Creación de un CRUD con SQL

En las clases anteriores habíamos aprendido a implementar verbos como **GET, POST y DELETE** para poder realizar distintas acciones mediante métodos **HTTP** y todas esas acciones las almacenamos en memoria. Lo cual la persistencia de datos se perdía cuando se reiniciaba el servidor.

Ahora vamos a aprender a realizar las acciones **PUT** implementando la persistencia de información mediante una base de datos relacional.

¿Qué es el método PUT?

El método **HTTP PUT** es utilizado para actualizar un recurso existente o crear uno nuevo si no existe. En el contexto de las aplicaciones web y las **API,** se utiliza para modificar la información de un recurso identificado por una URL específica. La solicitud **PUT** requiere que el cliente especifique la URL del recurso y proporcione los datos actualizados.

Buenos Aires aprende
Agencia de Habilidades para el Futuro

¿Qué es MYSQL?

MySQL, como lo mencionamos antes, es un sistema de gestión de bases de datos relacional o también conocido como **“RDBMS”,(por sus siglas en inglés “Relational Database Management System“)** que se utiliza para almacenar y recuperar datos. Es uno de los sistemas de gestión de bases de datos más populares y ampliamente utilizados en el mundo. Fue desarrollado inicialmente por **MySQL AB**, que luego fue adquirida por Sun Microsystems y, posteriormente, por **Oracle Corporation**.

MySQL es utilizado en una variedad de aplicaciones, desde sitios web y aplicaciones empresariales hasta sistemas integrados y más. Además, es comúnmente utilizado junto con otros componentes de software, como el servidor web Apache y el lenguaje de programación **PHP**, en la pila **LAMP (Linux, Apache, MySQL, PHP/Python/Perl).**

Instalación de MYSQL

Para tener instalado **MYSQL** vamos a tener que tener ya configurado e instalado nuestro servidor en **Express.js**

Para instalar MYSQL debemos hacer este comando:

```
npm install express mysql body-parser
```

Integración de MYSQL con Express.js

```
const express = require('express');
const mysql = require('mysql');
const bodyParser = require('body-parser');

const app = express();
const PORT = 3000;

// Configuración para analizar el cuerpo de las solicitudes
app.use(bodyParser.urlencoded({ extended: true }));
app.use(bodyParser.json());
```

Buenos Aires aprende
Agencia de Habilidades para el Futuro

```
// Configuración de la conexión a la base de datos
const connection = mysql.createConnection({
  host: 'localhost',
  user: 'root',
  password: '',
  database: 'phpmyadmin'
});

connection.connect(err => {
  if (err) {
    console.error('Error al conectar a la base de datos:', err);
  } else {
    console.log('Conexión exitosa a la base de datos');
  }
});


app.get('/crear-tabla', (req, res) => {
    const createTableQuery = `
      CREATE TABLE productos (
        id INT AUTO_INCREMENT PRIMARY KEY,
        nombre VARCHAR(255) NOT NULL,
        precio DECIMAL(10, 2) NOT NULL,
        descripcion TEXT
      )
    `;

    connection.query(createTableQuery, (err, results) => {
      if (err) {
        console.error('Error al crear la tabla:', err);
        res.status(500).send('Error al crear la tabla');
      } else {
        console.log('Tabla creada exitosamente');
```

```
        res.send('Tabla creada exitosamente');
      }
    });
  });

// Definir una ruta para crear la tabla
app.get('/productos', (req, res) => {
    const selectQuery = 'SELECT * FROM productos';

    connection.query(selectQuery, (err, results) => {
      if (err) {
        console.error('Error al seleccionar productos:', err);
        res.status(500).send('Error al seleccionar productos');
      } else {
        console.log('Productos seleccionados exitosamente');
        res.json(results);
      }
    });
  });

app.post('/agregar-producto', (req, res) => {
    const { nombre, precio, descripcion } = req.body;

    const insertQuery = `
      INSERT INTO productos (nombre, precio, descripcion)
      VALUES (?, ?, ?)
    `;

    connection.query(insertQuery, [nombre, precio, descripcion], (err,
results) => {
      if (err) {
        console.error('Error al agregar producto:', err);
        res.status(500).send('Error al agregar producto');
      } else {
```

```
        console.log('Producto agregado exitosamente');
        res.send('Producto agregado exitosamente');
      }
    });
  });

  app.put('/productos/:id', (req, res) => {
    const productId = req.params.id;
    const { nombre, precio, descripcion } = req.body;

    const updateQuery = `
      UPDATE productos
      SET nombre = ?, precio = ?, descripcion = ?
      WHERE id = ?
    `;

    connection.query(updateQuery, [nombre, precio, descripcion,
productId], (err, results) => {
      if (err) {
        console.error('Error al actualizar producto:', err);
        res.status(500).send('Error al actualizar producto');
      } else {
        console.log('Producto actualizado exitosamente');
        res.send('Producto actualizado exitosamente');
      }
    });
  });

  app.delete('/productos/:id', (req, res) => {
    const productId = req.params.id;

    const deleteQuery = 'DELETE FROM productos WHERE id = ?';

    connection.query(deleteQuery, [productId], (err, results) => {
```

```
      if (err) {
        console.error('Error al eliminar producto:', err);
        res.status(500).send('Error al eliminar producto');
      } else {
        console.log('Producto eliminado exitosamente');
        res.send('Producto eliminado exitosamente');
      }
    });
  });


  app.listen(PORT, () => {
  console.log(`Servidor Express escuchando en el puerto ${PORT}`);
});
```

Ahora vamos a explicar el código paso a paso:

1. **Creación de una conexión:**

```
const conexion = mysql.createConnection({
    host: 'localhost',
    user: 'root',
    password: '',
    database: 'phpmyadmin'
 });
```

- `mysql.createConnection` es un método proporcionado por la biblioteca `mysql` en Node.js para crear una nueva conexión a una base de datos MySQL.
- El objeto de configuración pasado como argumento contiene información necesaria para establecer la conexión, como el host de la base de datos, el nombre de usuario, la contraseña y el nombre de la base de datos a la que se va a conectar.

2. **Conexión a la base de datos:**

```
conexion.connect(error => {
    if (error) {
      console.error('Error al conectar a la base de datos:', error);
    } else {
      console.log('Conexión exitosa a la base de datos');
    }
  });
```

- `conexion.connect` es un método que intenta establecer la conexión a la base de datos **MySQL.**
- Toma una función de devolución de llamada (callback) que se ejecutará después de intentar establecer la conexión.
- Si hay un error durante la conexión, el callback recibe un objeto de error y se imprime un mensaje de error en la consola.
- Si la conexión se establece con éxito, se imprime un mensaje indicando que la conexión fue exitosa.

Este código es esencial al trabajar con MySQL en Node.js, ya que crea una conexión a la base de datos y verifica si la conexión se realizó correctamente. Después de establecer la conexión, puedes utilizar el objeto `conexión` para realizar consultas y operaciones en la base de datos. Es importante cerrar la conexión cuando ya no se necesite para liberar recursos, aunque en este código en particular, la conexión no se cierra explícitamente. En una aplicación más grande, se recomienda gestionar adecuadamente el ciclo de vida de la conexión y cerrarla cuando ya no sea necesaria.

Utilización de Endpoint

Los endpoints son puntos de acceso bien definidos que permiten la interacción entre aplicaciones y servicios a través de la web.

Endpoint para Crear una Tabla

```
app.get('/crear-tabla', (req, res) => {
    const createTableQuery = `
      CREATE TABLE productos (
        id INT AUTO_INCREMENT PRIMARY KEY,
```

```
        nombre VARCHAR(255) NOT NULL,
        precio DECIMAL(10, 2) NOT NULL,
        descripcion TEXT
      )
    `;

    connection.query(createTableQuery, (err, results) => {
      if (err) {
        console.error('Error al crear la tabla:', err);
        res.status(500).send('Error al crear la tabla');
      } else {
        console.log('Tabla creada exitosamente');
        res.send('Tabla creada exitosamente');
      }
    });
});
```

Esta ruta (/crear-tabla) está configurada para manejar peticiones **GET**. Cuando se accede a esta ruta, se ejecuta una consulta **SQL (createTableQuery)** para crear una tabla llamada "productos" con ciertos campos (id, nombre, precio, descripción). Luego, la conexión a la base de datos ejecuta esta consulta y devuelve la respuesta al cliente. Si hay un error, se envía una respuesta de error con un estado 500; de lo contrario, se envía un mensaje indicando que la tabla se creó exitosamente.

Endpoint para Obtener Productos

```
app.get('/productos', (req, res) => {
    const selectQuery = 'SELECT * FROM productos';

    connection.query(selectQuery, (err, results) => {
      if (err) {
        console.error('Error al seleccionar productos:', err);
        res.status(500).send('Error al seleccionar productos');
      } else {
```

```
        console.log('Productos seleccionados exitosamente');
        res.json(results);
      }
    });
});
```

Esta ruta (/productos) maneja peticiones **GET** y ejecuta una consulta **SQL (selectQuery)** para seleccionar todos los productos de la tabla "productos". Los resultados se envían como una respuesta **JSON** al cliente. Si hay un error, se envía una respuesta de error con un estado 500.

Endpoint para Agregar un Producto

```
app.post('/agregar-producto', (req, res) => {
    const { nombre, precio, descripcion } = req.body;

    const insertQuery = `
      INSERT INTO productos (nombre, precio, descripcion)
      VALUES (?, ?, ?)
    `;
    connection.query(insertQuery, [nombre, precio, descripcion], (err,
results) => {
      if (err) {
        console.error('Error al agregar producto:', err);
        res.status(500).send('Error al agregar producto');
      } else {
        console.log('Producto agregado exitosamente');
        res.send('Producto agregado exitosamente');
      }
    });
});
```

Este **endpoint** (/agregar-producto) maneja peticiones **POST** y permite agregar un nuevo producto a la tabla. Los datos del nuevo producto se obtienen del cuerpo de la solicitud

(req.body). Luego, se ejecuta una consulta **SQL (insertQuery)** para insertar el nuevo producto en la tabla. La respuesta se envía al cliente, indicando si la operación fue exitosa o si hubo un error.

Endpoint para Actualizar un Producto

```
app.put('/productos/:id', (req, res) => {
    const productId = req.params.id;
    const { nombre, precio, descripcion } = req.body;

    const updateQuery = `
      UPDATE productos
      SET nombre = ?, precio = ?, descripcion = ?
      WHERE id = ?
    `;

    connection.query(updateQuery, [nombre, precio, descripcion,
productId], (err, results) => {
      if (err) {
        console.error('Error al actualizar producto:', err);
        res.status(500).send('Error al actualizar producto');
      } else {
        console.log('Producto actualizado exitosamente');
        res.send('Producto actualizado exitosamente');
      }
    });
});
```

Este endpoint (/productos/:id) maneja peticiones **PUT** y permite actualizar un producto existente en la tabla. El **ID** del producto se obtiene de los parámetros de la URL (req.params.id), y los nuevos datos del producto se obtienen del cuerpo de la solicitud (req.body). Se ejecuta una consulta **SQL (updateQuery)** para realizar la actualización. La respuesta se envía al cliente, indicando si la operación fue exitosa o si hubo un error.

Buenos Aires aprende
Agencia de Habilidades para el Futuro

Endpoint para Eliminar un Producto

```
app.delete('/productos/:id', (req, res) => {
    const productId = req.params.id;

    const deleteQuery = 'DELETE FROM productos WHERE id = ?';

    connection.query(deleteQuery, [productId], (err, results) => {
      if (err) {
        console.error('Error al eliminar producto:', err);
        res.status(500).send('Error al eliminar producto');
      } else {
        console.log('Producto eliminado exitosamente');
        res.send('Producto eliminado exitosamente');
      }
    });
});
```

Este endpoint (**/productos/:id)** maneja peticiones **DELETE** y permite eliminar un producto de la tabla. El ID del producto se obtiene de los parámetros de la URL **(req.params.id)**. Se ejecuta una consulta **SQL (deleteQuery)** para realizar la eliminación. La respuesta se envía al cliente, indicando si la operación fue exitosa o si hubo un error.

Buenos Aires aprende
Agencia de Habilidades para el Futuro

Desafío #2

¿Te acordas que en la clase número 2 creamos un servidor con express js? 🤔

Ahora vamos a agregarle la dependencia de **mysql** para poder armar una base de datos que contenga una tabla llamada productos que contengan los siguientes campos : nombre **(VARCHAR(255))**, precio **(DECIMAL(10,2))**, descripcion **(TEXT)**.

El proyecto deberá tener:

- Instaladas las dependencias (express, mysql, bodyparser)
- Contar con un endpoint para crear la tabla (app.get('/crear-tabla')
- Contar con un endpoint para obtener la tabla (app.get('/productos')
- Un post que nos sirva para agregar los productos (app.post('/agregar-producto')


Buenos Aires
aprende
Buenos
Aires
Ciudad