Desarrollo CRM

1. INSTALACIÓN MySQL + Crear BD

2. MODELO DE DATOS (tablas básicas)

3. SERVIDOR EXPRESS + Conexión BD

4. CRUD USUARIOS (GET, POST, PUT, DELETE)

5. CRUD CLIENTES (GET, POST, PUT, DELETE)

6. CRUD VENTAS (GET, POST)

7. RUTAS ESPECÍFICAS (clientes por usuario, etc.)

8. EXTENSIONES (contactos, direcciones, etc.)

Diagrama de casos uso …
Diagrama de flujo …
Diagrama de clases …

Entidades y relaciones
Usuario: (empleados) id_usuario, nombre, correo, psw

- Usuarios gestionan clientes 1:N

Cliente: id_cliente, nombre, correo, direccion, id_usuario (FK)

- clientes adquieren productos N:M
- Relación_Comercial: id_cliente (FK/PK), id_producto (FK/PK), fecha de alta, fecha de baja, precio_venta
- Dirección: Tabla adicional por atributo compuesto:

Proveedores: id_prov, nombre, correo, NIF

- proveedores suministran productos 1:N

Servicio/Productos: id_producto, nombre, precio_compra, id_prov (FK)

BASE DE DATOS
-- Conectar a MySQL como root → Si no estás ya en MySQL workbench
mysql -u root -p

-- Crear base de datos
CREATE DATABASE crm_db;

-- Crear usuario específico (opcional pero recomendado, tendréis que cambiar los datos en la configuración de la base de datos)
CREATE USER 'crm_user'@'localhost' IDENTIFIED BY 'password_segura';
GRANT ALL PRIVILEGES ON crm_db.* TO 'crm_user'@'localhost';
FLUSH PRIVILEGES;

- Si no por defecto el usuario es root y la psw queda vacía.

USE crm_db;

Pegar los CREATE TABLE en el orden correcto. Es decir, no puedes crear una tabla que hace referencia a otra que aún no haya sido creada.

TABLAS COMPLETAS

Tabla de Usuarios (empleados)

```
CREATE TABLE Usuario (
    id_usuario INT PRIMARY KEY AUTO_INCREMENT,
    nombre VARCHAR(100) NOT NULL,
    correo VARCHAR(150) UNIQUE NOT NULL,
    psw VARCHAR(255) NOT NULL,
    rol ENUM('admin', 'vendedor', 'gestor') DEFAULT 'vendedor',
    fecha_alta DATETIME DEFAULT CURRENT_TIMESTAMP,
    activo BOOLEAN DEFAULT TRUE
);
```

Tabla de Clientes

```
CREATE TABLE Cliente (
    id_cliente INT PRIMARY KEY AUTO_INCREMENT,
    nombre VARCHAR(100) NOT NULL,
    correo VARCHAR(150),
    telefono VARCHAR(20),
    id_usuario_asignado INT,
    tipo_cliente ENUM('potencial', 'activo', 'inactivo') DEFAULT 'potencial',
    fecha_registro DATETIME DEFAULT CURRENT_TIMESTAMP,
    ingresos_anuales DECIMAL(12,2) NULL,
    FOREIGN KEY (id_usuario_asignado) REFERENCES Usuario(id_usuario)
);
```

Tabla de Direcciones

```
CREATE TABLE Direccion (
    id_direccion INT PRIMARY KEY AUTO_INCREMENT,
    id_cliente INT NOT NULL,
    tipo_direccion ENUM('fiscal', 'envio', 'oficina') DEFAULT 'fiscal',
    calle VARCHAR(100),
    ciudad VARCHAR(50),
    codigo_postal VARCHAR(10),
    pais VARCHAR(50) DEFAULT 'España',
    FOREIGN KEY (id_cliente) REFERENCES Cliente(id_cliente) ON DELETE CASCADE
);
```

Tabla de Contactos (personas de contacto en cada cliente)

```
CREATE TABLE Contacto (
    id_contacto INT PRIMARY KEY AUTO_INCREMENT,
    id_cliente INT NOT NULL,
    nombre VARCHAR(100) NOT NULL,
    apellidos VARCHAR(100),
    correo VARCHAR(150),
    telefono VARCHAR(20),
    puesto VARCHAR(100),
    es_contacto_principal BOOLEAN DEFAULT FALSE,
```

```
    fecha_creacion DATETIME DEFAULT CURRENT_TIMESTAMP,
    activo BOOLEAN DEFAULT TRUE,
    FOREIGN KEY (id_cliente) REFERENCES Cliente(id_cliente) ON DELETE CASCADE
);
```

Tabla de Proveedores

```
CREATE TABLE Proveedor (
    id_prov INT PRIMARY KEY AUTO_INCREMENT,
    nombre VARCHAR(100) NOT NULL,
    correo VARCHAR(150),
    nif VARCHAR(20) UNIQUE,
    telefono VARCHAR(20),
    direccion TEXT,
    activo BOOLEAN DEFAULT TRUE
);
```

Tabla de Productos/Servicios

```
CREATE TABLE Producto (
    id_producto INT PRIMARY KEY AUTO_INCREMENT,
    nombre VARCHAR(100) NOT NULL,
    descripcion TEXT,
    categoria ENUM('producto', 'servicio') DEFAULT 'producto',
    precio_compra DECIMAL(10,2),
    precio_venta_sugerido DECIMAL(10,2),
    id_prov INT,
    stock INT DEFAULT 0,
    activo BOOLEAN DEFAULT TRUE,
    FOREIGN KEY (id_prov) REFERENCES Proveedor(id_prov)
);
```

Tabla de Ventas (relación comercial)

```
CREATE TABLE Venta (
    id_venta INT PRIMARY KEY AUTO_INCREMENT,
    id_cliente INT NOT NULL,
    id_producto INT NOT NULL,
    id_usuario_venta INT NOT NULL,
    fecha_contratacion DATE NOT NULL,
    fecha_finalizacion_prevista DATE,
    fecha_finalizacion_real DATE NULL,
    precio_final DECIMAL(10,2) NOT NULL,
    cantidad INT DEFAULT 1,
    estado ENUM('cotizacion', 'contratado', 'en_proceso', 'completado', 'cancelado') DEFAULT
'cotizacion',
    notas TEXT,
    FOREIGN KEY (id_cliente) REFERENCES Cliente(id_cliente),
    FOREIGN KEY (id_producto) REFERENCES Producto(id_producto),
    FOREIGN KEY (id_usuario_venta) REFERENCES Usuario(id_usuario)
);
```

TABLA DE SEGUIMIENTOS (Muy importante para CRM)

```
CREATE TABLE Seguimiento (
    id_seguimiento INT PRIMARY KEY AUTO_INCREMENT,
    id_cliente INT NOT NULL,
    id_usuario INT NOT NULL,
    tipo_seguimiento ENUM('llamada', 'email', 'reunion', 'tarea', 'nota') DEFAULT 'nota',
    titulo VARCHAR(200) NOT NULL,
    descripcion TEXT,
    fecha_seguimiento DATETIME DEFAULT CURRENT_TIMESTAMP,
    fecha_proximo_seguimiento DATE NULL,
    prioridad ENUM('baja', 'media', 'alta') DEFAULT 'media',
    completado BOOLEAN DEFAULT FALSE,
    FOREIGN KEY (id_cliente) REFERENCES Cliente(id_cliente) ON DELETE CASCADE,
    FOREIGN KEY (id_usuario) REFERENCES Usuario(id_usuario)
);
```

SERVIDOR NODE CON EXPRESS

Creamos el proyecto NODE en una carpeta: npm init -y

Instalamos el framework EXPRESS: npm install express

- Se instala como dependencia del servidor en node

Creamos el archivo que será el corazón del servidor: app.js, index.js o server.js

```
// 1. Importar Express y mysql2
const express = require('express');
const mysql = require('mysql2');  //Driver para conectar con MySQL

// 2. Crear una instancia de la aplicación Express
const app = express();
```

app →

- Origen: Creado por Express
- Qué es: Es el objeto principal de tu aplicación Express
- Función: Contiene todos los métodos para configurar rutas, middleware, etc.

```
// 3. Definir el puerto en el que escuchará el servidor
const port = 3000;

// 4. MIDDLEWARE PARA PARSEAR JSON (IMPORTANTE para recibir datos)
app.use(express.json());
```

- Origen: Método de Express
- Qué es: Middleware (software intermedio)
- Función: Convierte automáticamente JSON en objetos JavaScript
- Ejemplo: Si recibes {"nombre": "Juan"}, lo convierte en req.body.nombre

```
//5.1  CONFIGURACIÓN DE LA BASE DE DATOS (cuidado si cambiaste los datos)
const connection = mysql.createConnection({
   host: 'localhost',          // MySQL está en esta misma computadora
   user: 'root',               // Tu usuario de MySQL (cambiar si es diferente)
   password: '',               // Tu contraseña de MySQL
   database: 'crm_escuela'     // Nombre de la base de datos (debe existir)
});
```

- Origen: Método del paquete mysql2
- Qué es: Configuración para la conexión (NO conecta todavía)
- Parámetros:
    - host: Donde está MySQL (localhost = misma computadora/'192.168.1.100' = dirección externa)
    - user: Usuario de MySQL (root por defecto)
    - password: Contraseña del usuario (vacía por defecto)
    - database: Base de datos específica a usar, ej: crm_db

```
// 5.2 CONECTAR A LA BASE DE DATOS
connection.connect((error) => {
   if (error) {
      console.error('Error conectando a MySQL:', error.message);
      return;
   }
   console.log('Conectado a la base de datos MySQL');
});
```

- Origen: Método del objeto connection
- Qué es: Intenta establecer la conexión real con MySQL
- Cómo funciona:
  - Es ASÍNCRONO (puede tardar segundos)
  - Cuando termina, ejecuta la función callback
  - error: Contiene el error si falla, es null si tiene éxito

```
// 6. Definir una ruta de prueba
// Cuando un cliente (navegador) haga una petición GET a la ruta '/',
// el servidor ejecutará esta función.
app.get('/', (req, res) => {
  // Envía una respuesta de texto plano al cliente.
  res.send('¡El servidor del CRM está funcionando!');
});
```

.get →

- Origen: Método proporcionado por Express
- Qué es: Define un manejador para peticiones HTTP GET
- Alternativas:
  - app.post() - Para crear datos
  - app.put() - Para actualizar datos (hace falta incluir todos los campos)*
  - app.delete() - Para eliminar datos
  - app.patch() - Para actualizaciones parciales

'/' →

- Origen: Definido por el desarrollador (tú)
- Qué es: El endpoint o URL que los clientes usarán
- Ejemplos:
  - '/clientes' → http://localhost:3000/clientes
  - '/api/usuarios' → http://localhost:3000/api/usuarios
  - '/productos/:id' → Rutas con parámetros

Función callback

(req, res) => { ... } - La función callback
En nuestro ejemplo es la función que se ejecuta cuando alguien hace una petición GET a '/'

req - El objeto Request (Petición)
posibles parámetros y métodos:

```
(req, res) => {
   console.log(req.url);       // URL solicitada
```

```
    console.log(req.method);     // Método HTTP (GET, POST, etc.)
    console.log(req.headers);    // Cabeceras de la petición
    console.log(req.query);      // Parámetros de query string (?nombre=valor)
    console.log(req.params);     // Parámetros de ruta (/users/:id)
    console.log(req.body);       // Cuerpo de la petición (para POST/PUT)
}
```

- Origen: Creado automáticamente por Express basado en la petición HTTP entrante
- Contiene: Toda la información que el cliente envía al servidor

res - El objeto Response (Respuesta)
posibles métodos:

```
(req, res) => {
    res.send('Texto plano');                    // Enviar texto
    res.json({mensaje: 'Hola'});                // Enviar JSON
    res.status(404).send('No encontrado');      // Enviar código de estado + mensaje
    res.redirect('/otra-ruta');                 // Redireccionar
    res.sendFile('/ruta/archivo.html');         // Enviar archivo
}
```

- Origen: Proporcionado por Express
- Función: Métodos para enviar respuesta al cliente

```
// 7. RUTA DE PRUEBA CON BASE DE DATOS
app.get('/test-db', (req, res) => {
    connection.query('SELECT "Conexión exitosa" as mensaje', (error, results) => {
        if (error) {
            return res.status(500).json({ error: 'Error en la base de datos' });
        }
        res.json({
            mensaje: 'Base de datos funcionando',
            resultado: results[0]
        });
    });
});
```

- connection.query(): Ejecuta una consulta SQL en la base de datos
- Callback: Se ejecuta cuando MySQL responde
- error: Información del error si la consulta falla
- results: Resultados de la consulta SQL

```
// 8. Poner el servidor a escuchar en el puerto definido
app.listen(port, () => {
  // Esta función se ejecuta cuando el servidor se inicia correctamente.
  console.log(`Servidor ejecutándose en: http://localhost:${port}`);
});
```

Ejecutamos el servidor: node app.js

- podemos instalar nodemon: npm install -D nodemon (arrancamos con “nodemon”)

- Si todo ha ido bien, verás el mensaje: "Servidor ejecutándose en: http://localhost:3000"
- detenemos el proyecto con ctrl + C

EJEMPLO DE FLUJO COMPLETO:

```
const express = require('express');
const app = express();
const port = 3000;

// Ruta de ejemplo más detallada
app.get('/saludo', (req, res) => {
    console.log('=== INFORMACIÓN DE LA PETICIÓN (req) ===');
    console.log('URL:', req.url);           // '/saludo'
    console.log('Método:', req.method);     // 'GET'
    console.log('Headers:', req.headers);   // Información del navegador, etc.

    // Podemos acceder a parámetros de query string
    // Ejemplo: http://localhost:3000/saludo?nombre=Juan
    const nombre = req.query.nombre || 'Visitante';

    console.log('=== CONSTRUYENDO RESPUESTA (res) ===');
    // Configuramos la respuesta
    res.setHeader('Content-Type', 'text/html; charset=utf-8');
    res.status(200);

    // Enviamos la respuesta
    res.send(`
        <html>
            <body>
                <h1>¡Hola, ${nombre}!</h1>
                <p>Esta es una respuesta HTML personalizada</p>
            </body>
        </html>
    `);
});

app.listen(port, () => {
    console.log(`Servidor en http://localhost:${port}`);
});
```

INICIALIZAR LA BASE DE DATOS
Ejecutar directamente en MySQL workbench de momento.

Crear: database/01_create_database.sql. **Opcional para scripts automáticos**

```
-- ============================================
-- SCRIPT 1: CREAR LA BASE DE DATOS Y USUARIO
-- ============================================

-- Conectar a MySQL como administrador (root) y ejecutar este script

-- 1. Crear la base de datos (si no existe)
CREATE DATABASE IF NOT EXISTS crm_db;

-- 2. Seleccionar la base de datos para usar
USE crm_db;

-- 3. (OPCIONAL) Crear un usuario específico para el CRM
-- CREATE USER IF NOT EXISTS 'crm_user'@'localhost' IDENTIFIED BY
'password_segura';
-- GRANT ALL PRIVILEGES ON crm_db.* TO 'crm_user'@'localhost';
-- FLUSH PRIVILEGES;

-- 4. Mostrar confirmación
SELECT 'Base de datos crm_db creada/existe' as Mensaje;
```

Crear: database/02_create_tables.sql. **Opcional para scripts automáticos.**

```
-- ============================================
-- SCRIPT 2: CREAR TODAS LAS TABLAS DEL CRM
-- ============================================

-- Asegúrate de que la base de datos esté seleccionada
USE crm_db;

-- ========================
-- TABLA 1: USUARIOS (EMPLEADOS)
-- ========================
CREATE TABLE IF NOT EXISTS Usuario (
    id_usuario INT PRIMARY KEY AUTO_INCREMENT,
    nombre VARCHAR(100) NOT NULL,
    correo VARCHAR(150) UNIQUE NOT NULL,
    psw VARCHAR(255) NOT NULL,
    rol ENUM('admin', 'vendedor', 'gestor') DEFAULT 'vendedor',
    fecha_alta DATETIME DEFAULT CURRENT_TIMESTAMP,
    activo BOOLEAN DEFAULT TRUE
);

-- ========================
-- TABLA 2: CLIENTES
-- ========================
CREATE TABLE IF NOT EXISTS Cliente (
    id_cliente INT PRIMARY KEY AUTO_INCREMENT,
    nombre VARCHAR(100) NOT NULL,
    correo VARCHAR(150),
    telefono VARCHAR(20),
    id_usuario_asignado INT,
    tipo_cliente ENUM('potencial', 'activo', 'inactivo') DEFAULT 'potencial',
    fecha_registro DATETIME DEFAULT CURRENT_TIMESTAMP,
    ingresos_anuales DECIMAL(12,2) NULL,
    FOREIGN KEY (id_usuario_asignado) REFERENCES Usuario(id_usuario)
);

-- ========================
-- TABLA 3: DIRECCIONES
-- ========================
CREATE TABLE IF NOT EXISTS Direccion (
    id_direccion INT PRIMARY KEY AUTO_INCREMENT,
    id_cliente INT NOT NULL,
    tipo_direccion ENUM('fiscal', 'envio', 'oficina') DEFAULT 'fiscal',
    calle VARCHAR(100),
    ciudad VARCHAR(50),
    codigo_postal VARCHAR(10),
    pais VARCHAR(50) DEFAULT 'España',
    FOREIGN KEY (id_cliente) REFERENCES Cliente(id_cliente) ON DELETE CASCADE
```

```
);

-- ========================
-- TABLA 4: CONTACTOS
-- ========================
CREATE TABLE IF NOT EXISTS Contacto (
    id_contacto INT PRIMARY KEY AUTO_INCREMENT,
    id_cliente INT NOT NULL,
    nombre VARCHAR(100) NOT NULL,
    apellidos VARCHAR(100),
    correo VARCHAR(150),
    telefono VARCHAR(20),
    puesto VARCHAR(100),
    es_contacto_principal BOOLEAN DEFAULT FALSE,
    fecha_creacion DATETIME DEFAULT CURRENT_TIMESTAMP,
    activo BOOLEAN DEFAULT TRUE,
    FOREIGN KEY (id_cliente) REFERENCES Cliente(id_cliente) ON DELETE CASCADE
);

-- ========================
-- TABLA 5: PROVEEDORES
-- ========================
CREATE TABLE IF NOT EXISTS Proveedor (
    id_prov INT PRIMARY KEY AUTO_INCREMENT,
    nombre VARCHAR(100) NOT NULL,
    correo VARCHAR(150),
    nif VARCHAR(20) UNIQUE,
    telefono VARCHAR(20),
    direccion TEXT,
    activo BOOLEAN DEFAULT TRUE
);

-- ========================
-- TABLA 6: PRODUCTOS/SERVICIOS
-- ========================
CREATE TABLE IF NOT EXISTS Producto (
    id_producto INT PRIMARY KEY AUTO_INCREMENT,
    nombre VARCHAR(100) NOT NULL,
    descripcion TEXT,
    categoria ENUM('producto', 'servicio') DEFAULT 'producto',
    precio_compra DECIMAL(10,2),
    precio_venta_sugerido DECIMAL(10,2),
    id_prov INT,
    stock INT DEFAULT 0,
    activo BOOLEAN DEFAULT TRUE,
    FOREIGN KEY (id_prov) REFERENCES Proveedor(id_prov)
);
```

```
-- ========================
-- TABLA 7: VENTAS
-- ========================
CREATE TABLE IF NOT EXISTS Venta (
    id_venta INT PRIMARY KEY AUTO_INCREMENT,
    id_cliente INT NOT NULL,
    id_producto INT NOT NULL,
    id_usuario_venta INT NOT NULL,
    fecha_contratacion DATE NOT NULL,
    fecha_finalizacion_prevista DATE,
    fecha_finalizacion_real DATE NULL,
    precio_final DECIMAL(10,2) NOT NULL,
    cantidad INT DEFAULT 1,
    estado ENUM('cotizacion', 'contratado', 'en_proceso', 'completado', 'cancelado') DEFAULT
'cotizacion',
    notas TEXT,
    FOREIGN KEY (id_cliente) REFERENCES Cliente(id_cliente),
    FOREIGN KEY (id_producto) REFERENCES Producto(id_producto),
    FOREIGN KEY (id_usuario_venta) REFERENCES Usuario(id_usuario)
);

-- ========================
-- TABLA 8: SEGUIMIENTOS
-- ========================
CREATE TABLE IF NOT EXISTS Seguimiento (
    id_seguimiento INT PRIMARY KEY AUTO_INCREMENT,
    id_cliente INT NOT NULL,
    id_usuario INT NOT NULL,
    tipo_seguimiento ENUM('llamada', 'email', 'reunion', 'tarea', 'nota') DEFAULT 'nota',
    titulo VARCHAR(200) NOT NULL,
    descripcion TEXT,
    fecha_seguimiento DATETIME DEFAULT CURRENT_TIMESTAMP,
    fecha_proximo_seguimiento DATE NULL,
    prioridad ENUM('baja', 'media', 'alta') DEFAULT 'media',
    completado BOOLEAN DEFAULT FALSE,
    FOREIGN KEY (id_cliente) REFERENCES Cliente(id_cliente) ON DELETE CASCADE,
    FOREIGN KEY (id_usuario) REFERENCES Usuario(id_usuario)
);

-- ============================================
-- CONFIRMACIÓN DE TABLAS CREADAS
-- ============================================
SELECT
    TABLE_NAME as 'Tabla creada',
    TABLE_ROWS as 'Filas'
FROM information_schema.TABLES
WHERE TABLE_SCHEMA = 'crm_db';
```

INSERTAR DATOS DE PRUEBA. (psw sin hashear)

```
-- database/03_sample_data.sql
USE crm_db;

-- Insertar usuarios de prueba
INSERT INTO Usuario (nombre, correo, psw, rol) VALUES
('Ana García', 'ana@empresa.com', '123456', 'admin'),
('Carlos López', 'carlos@empresa.com', '123456', 'vendedor'),
('Maria Rodriguez', 'maria@empresa.com', '123456', 'gestor');

-- Insertar clientes de prueba
INSERT INTO Cliente (nombre, correo, telefono, id_usuario_asignado) VALUES
('Empresa ABC', 'info@abc.com', '912345678', 1),
('Tienda XYZ', 'contacto@xyz.com', '934567890', 2),
('Consultora QRS', 'admin@qrs.com', '956789012', 1);

-- Resto de tablas …
```

EJEMPLOS DE PETICIONES

1. Listar Empleados (Usuarios)

```
// GET /api/usuarios - Listar todos los empleados
app.get('/api/usuarios', (req, res) => {
    /**
     * req: No necesita parámetros para listar todos
     * res: Devuelve array con todos los usuarios
     */

    const query = `
        SELECT
            id_usuario,
            nombre,
            correo,
            rol,
            fecha_alta,
            activo
        FROM Usuario
        WHERE activo = TRUE
        ORDER BY nombre
    `;

    connection.query(query, (error, results) => {
        if (error) {
            console.error('Error en la consulta:', error);
            return res.status(500).json({
                error: 'Error interno del servidor',
                detalles: error.message
            });
        }

        // Enviamos la respuesta con formato JSON
        res.json({
            success: true,
            count: results.length,
            usuarios: results
        });
    });
});
```

Explicación:

- Método HTTP: GET (solo lectura)
- Ruta: /api/usuarios
- Query SQL: Selecciona solo usuarios activos, excluyendo la contraseña por seguridad

Respuesta: Array de objetos JSON con información de usuarios

Prueba: GET http://localhost:3000/api/usuarios

2. Listar clientes por empleado:

```
// GET /api/usuarios/:id/clientes - Clientes asignados a un empleado específico
app.get('/api/usuarios/:id/clientes', (req, res) => {
    /**
     * req.params.id: ID del usuario que viene en la URL
     * res: Devuelve clientes asignados a ese usuario
     */

    const userId = req.params.id; // Extraemos el parámetro de la URL

    // Validación básica
    if (!userId || isNaN(userId)) {
        return res.status(400).json({
            error: 'ID de usuario inválido'
        });
    }

    const query = `
        SELECT
            c.id_cliente,
            c.nombre,
            c.correo,
            c.telefono,
            c.tipo_cliente,
            c.fecha_registro,
            c.ingresos_anuales,
            d.ciudad,
            d.calle
        FROM Cliente c
        LEFT JOIN Direccion d ON c.id_cliente = d.id_cliente AND d.tipo_direccion = 'fiscal'
        WHERE c.id_usuario_asignado = ?
        ORDER BY c.nombre
    `;

    connection.query(query, [userId], (error, results) => {
        if (error) {
            console.error('Error en la consulta:', error);
            return res.status(500).json({ error: 'Error en la consulta' });
        }

        res.json({
            success: true,
            usuario_id: userId,
            count: results.length,
            clientes: results
        });
    });
```

});
Explicación:

- :id: Parámetro de ruta que captura el ID del usuario
- ?: Placeholder para prevenir SQL injection
- [userId]: Array de parámetros que reemplaza el ? en la query
- JOIN: Une con la tabla Dirección para obtener información adicional

Prueba: GET http://localhost:3000/api/usuarios/1/clientes

3. Cambiar el estado de un contacto de un cliente de true a false

```
// PATCH /api/contactos/:id/estado - Cambiar estado activo/inactivo de un contacto
app.patch('/api/contactos/:id/estado', (req, res) => {
    /**
     * req.params.id: ID del contacto a modificar
     * req.body.activo: Nuevo estado (true/false) que viene en el cuerpo de la petición
     * res: Confirma la actualización
     */

    const contactoId = req.params.id;
    const { activo } = req.body; // Destructuring del body

    // Validaciones
    if (!contactoId || isNaN(contactoId)) {
        return res.status(400).json({ error: 'ID de contacto inválido' });
    }

    if (typeof activo !== 'boolean') {
        return res.status(400).json({
            error: 'El campo "activo" debe ser true o false'
        });
    }

    const query = `
        UPDATE Contacto
        SET activo = ?
        WHERE id_contacto = ?
    `;

    connection.query(query, [activo, contactoId], (error, results) => {
        if (error) {
            console.error('Error actualizando contacto:', error);
            return res.status(500).json({ error: 'Error en la actualización' });
        }

        if (results.affectedRows === 0) {
            return res.status(404).json({ error: 'Contacto no encontrado' });
        }

        res.json({
            success: true,
            message: `Contacto ${activo ? 'activado' : 'desactivado'} correctamente`,
            cambios: results.affectedRows
        });
    });
});
```

Explicación:

- Método HTTP: PATCH (actualización parcial)
- req.body: Datos que vienen en el cuerpo de la petición (JSON)
- Destructuring: { activo } = req.body extrae la propiedad activo
- affectedRows: Número de filas afectadas por el UPDATE

Prueba:

Desactivar un contacto
PATCH http://localhost:3000/api/contactos/5/estado
Body: { "activo": false }

Activar un contacto
PATCH http://localhost:3000/api/contactos/5/estado
Body: { "activo": true }

4. Crear un nuevo registro de venta:

```
// POST /api/ventas - Crear un nuevo registro de venta
app.post('/api/ventas', (req, res) => {
    /**
     * req.body: Contiene todos los datos de la nueva venta
     * res: Confirma la creación y devuelve el ID generado
     */

    const {
        id_cliente,
        id_producto,
        id_usuario_venta,
        fecha_contratacion,
        fecha_finalizacion_prevista,
        precio_final,
        cantidad = 1,
        estado = 'cotizacion',
        notas = ''
    } = req.body;

    // Validaciones básicas
    const camposRequeridos = ['id_cliente', 'id_producto', 'id_usuario_venta', 'precio_final'];
    const faltantes = camposRequeridos.filter(campo => !req.body[campo]);

    if (faltantes.length > 0) {
        return res.status(400).json({
            error: `Campos requeridos faltantes: ${faltantes.join(', ')}`
        });
    }

    if (precio_final <= 0) {
        return res.status(400).json({
            error: 'El precio final debe ser mayor a 0'
        });
    }

    const query = `
        INSERT INTO Venta (
            id_cliente, id_producto, id_usuario_venta,
            fecha_contratacion, fecha_finalizacion_prevista,
            precio_final, cantidad, estado, notas
        ) VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?)
    `;

    const parametros = [
        id_cliente, id_producto, id_usuario_venta,
        fecha_contratacion, fecha_finalizacion_prevista,
        precio_final, cantidad, estado, notas
```

```
    ];

    connection.query(query, parametros, (error, results) => {
        if (error) {
            console.error('Error creando venta:', error);
            return res.status(500).json({
                error: 'Error creando la venta',
                detalles: error.message
            });
        }

        res.status(201).json({
            success: true,
            message: 'Venta creada exitosamente',
            venta_id: results.insertId, // ID auto-generado por MySQL
            datos: {
                id_cliente,
                id_producto,
                precio_final,
                cantidad,
                estado
            }
        });
    });
});
```

Explicación:

- Método HTTP: POST (creación de recursos)
- req.body: Todos los datos necesarios para crear la venta
- Valores por defecto: cantidad = 1, estado = 'cotizacion'
- results.insertId: ID auto-incremental generado por MySQL
- Status 201: Created - indica creación exitosa

Prueba:

POST http://localhost:3000/api/ventas
Body:

```
{
  "id_cliente": 1,
  "id_producto": 3,
  "id_usuario_venta": 2,
  "fecha_contratacion": "2024-01-15",
  "fecha_finalizacion_prevista": "2024-12-31",
  "precio_final": 1500.00,
  "cantidad": 1,
  "estado": "contratado",
  "notas": "Venta inicial del proyecto CRM"
}
```

CRUD MONOLÍTICO

```
// crud_users_monolithic.js
// CRUD COMPLETO DE USUARIOS - VERSIÓN MONOLÍTICA (TODO EN UN ARCHIVO)

const express = require('express');
const mysql = require('mysql2');

// 1. CREAR LA APLICACIÓN EXPRESS
const app = express();
const port = 3000;

// 2. MIDDLEWARE PARA ENTENDER JSON
app.use(express.json());

// 3. CONFIGURACIÓN DE LA BASE DE DATOS
const connection = mysql.createConnection({
    host: 'localhost',
    user: 'root',
    password: '',
    database: 'crm_escuela'
});

// 4. CONECTAR A LA BASE DE DATOS
connection.connect((error) => {
    if (error) {
        console.error('Error conectando a MySQL:', error.message);
        return;
    }
    console.log('Conectado a la base de datos MySQL');
});

//
==========================================================================
======
// CRUD COMPLETO - USUARIOS
//
==========================================================================
======

// MIDDLEWARE DE LOGGING (Intercepta todas las peticiones)
app.use((req, res, next) => {
    console.log(`${new Date().toISOString()} - ${req.method} ${req.url}`);
    next(); // Pasa al siguiente middleware o ruta
});

// CREATE - Crear un nuevo usuario
app.post('/api/usuarios', (req, res) => {
    /**
```

```
 * FUNCIÓN: Crear un nuevo usuario en la base de datos
 * MÉTODO HTTP: POST (Crear recursos)
 * RUTA: /api/usuarios
 * DATOS NECESARIOS: Vienen en el body (req.body)
 */

console.log('Creando nuevo usuario...');

// 1. EXTRAER DATOS del cuerpo de la petición
const { nombre, correo, psw, rol = 'vendedor' } = req.body;
// Destructuring: Extrae propiedades del objeto req.body
// Valor por defecto: Si 'rol' no viene, usa 'vendedor'

// 2. VALIDACIONES BÁSICAS
if (!nombre || !correo || !psw) {
    // return: Detiene la ejecución aquí y envía respuesta
    return res.status(400).json({
        success: false,
        error: 'Faltan campos obligatorios: nombre, correo, psw'
    });
}

// 3. PREPARAR CONSULTA SQL
const query = `
    INSERT INTO Usuario (nombre, correo, psw, rol)
    VALUES (?, ?, ?, ?)
`;
// ?: Placeholders para prevenir SQL Injection

// 4. PARÁMETROS para la consulta (en el mismo orden que los ?)
const parametros = [nombre, correo, psw, rol];

// 5. EJECUTAR CONSULTA
connection.query(query, parametros, (error, results) => {
    // Esta función se ejecuta CUANDO MySQL responde

    if (error) {
        console.error('Error en INSERT:', error);

        // Manejar errores específicos
        if (error.code === 'ER_DUP_ENTRY') {
            return res.status(409).json({
                success: false,
                error: 'El correo electrónico ya existe'
            });
        }

        return res.status(500).json({
```

```
            success: false,
            error: 'Error interno del servidor'
        });
    }

    // 6. RESPUESTA EXITOSA
    console.log('Usuario creado con ID:', results.insertId);

    res.status(201).json({
        success: true,
        message: 'Usuario creado exitosamente',
        data: {
            id: results.insertId,  // ID auto-generado por MySQL
            nombre: nombre,
            correo: correo,
            rol: rol
        }
    });
  });
});

// READ - Obtener todos los usuarios
app.get('/api/usuarios', (req, res) => {
  /**
   * FUNCIÓN: Obtener lista de todos los usuarios
   * MÉTODO HTTP: GET (Leer recursos)
   * RUTA: /api/usuarios
   * PARÁMETROS: Ninguno (o opcionales por query string)
   */

  console.log('Obteniendo todos los usuarios...');

  // 1. PARÁMETROS OPCIONALES de query string
  const { activo = '1' } = req.query;
  // req.query: Parámetros de URL ?activo=0

  // 2. CONSULTA SQL
  const query = `
    SELECT
        id_usuario,
        nombre,
        correo,
        rol,
        fecha_alta,
        activo
    FROM Usuario
    WHERE activo = ?
    ORDER BY nombre
```

```
    `;

    // 3. EJECUTAR CONSULTA
    connection.query(query, [activo], (error, results) => {
        if (error) {
            console.error('Error en SELECT:', error);
            return res.status(500).json({
                success: false,
                error: 'Error obteniendo usuarios'
            });
        }

        // 4. RESPUESTA EXITOSA
        res.json({
            success: true,
            count: results.length,
            usuarios: results
        });
    });
});

// READ - Obtener un usuario específico por ID
app.get('/api/usuarios/:id', (req, res) => {
    /**
     * FUNCIÓN: Obtener un usuario específico
     * MÉTODO HTTP: GET
     * RUTA: /api/usuarios/:id
     * PARÁMETROS: id viene en la URL (req.params.id)
     */

    const userId = req.params.id;
    console.log(`Obteniendo usuario con ID: ${userId}`);

    // 1. VALIDACIÓN
    if (!userId || isNaN(userId)) {
        return res.status(400).json({
            success: false,
            error: 'ID de usuario inválido'
        });
    }

    // 2. CONSULTA SQL
    const query = `
        SELECT
            id_usuario,
            nombre,
            correo,
            rol,
```

```
            fecha_alta,
            activo
        FROM Usuario
        WHERE id_usuario = ?
    `;

    // 3. EJECUTAR CONSULTA
    connection.query(query, [userId], (error, results) => {
        if (error) {
            console.error('Error en SELECT:', error);
            return res.status(500).json({ error: 'Error en la consulta' });
        }

        // 4. VERIFICAR SI SE ENCONTRÓ EL USUARIO
        if (results.length === 0) {
            return res.status(404).json({
                success: false,
                error: 'Usuario no encontrado'
            });
        }

        // 5. RESPUESTA EXITOSA
        res.json({
            success: true,
            usuario: results[0]  // results es un array, tomamos el primer elemento
        });
    });
});

// UPDATE - Actualizar un usuario completo (PUT)
app.put('/api/usuarios/:id', (req, res) => {
    /**
     * FUNCIÓN: Actualizar TODOS los campos de un usuario
     * MÉTODO HTTP: PUT (Actualización completa)
     * RUTA: /api/usuarios/:id
     */

    const userId = req.params.id;
    console.log(`Actualizando usuario ID: ${userId}`);

    // 1. EXTRAER Y VALIDAR DATOS
    const { nombre, correo, rol, activo } = req.body;

    if (!nombre || !correo || !rol || typeof activo !== 'boolean') {
        return res.status(400).json({
            success: false,
            error: 'Faltan campos obligatorios: nombre, correo, rol, activo (boolean)'
        });
```

```
    }

    // 2. CONSULTA SQL
    const query = `
        UPDATE Usuario
        SET nombre = ?, correo = ?, rol = ?, activo = ?
        WHERE id_usuario = ?
    `;

    // 3. EJECUTAR CONSULTA
    connection.query(query, [nombre, correo, rol, activo, userId], (error, results) => {
        if (error) {
            console.error('Error en UPDATE:', error);
            return res.status(500).json({ error: 'Error actualizando usuario' });
        }

        // 4. VERIFICAR SI SE ACTUALIZÓ ALGUNA FILA
        if (results.affectedRows === 0) {
            return res.status(404).json({
                success: false,
                error: 'Usuario no encontrado'
            });
        }

        // 5. RESPUESTA EXITOSA
        res.json({
            success: true,
            message: 'Usuario actualizado completamente',
            cambios: results.affectedRows
        });
    });
});

// UPDATE - Actualización parcial (PATCH)
app.patch('/api/usuarios/:id', (req, res) => {
    /**
     * FUNCIÓN: Actualizar SOLO algunos campos del usuario
     * MÉTODO HTTP: PATCH (Actualización parcial)
     * RUTA: /api/usuarios/:id
     */

    const userId = req.params.id;
    const camposActualizar = req.body;

    console.log(`Actualizando parcialmente usuario ID: ${userId}`, camposActualizar);

    // 1. VALIDAR QUE HAYA CAMPOS PARA ACTUALIZAR
    if (Object.keys(camposActualizar).length === 0) {
```

```
        return res.status(400).json({
            success: false,
            error: 'No hay campos para actualizar'
        });
    }

    // 2. CONSTRUIR CONSULTA DINÁMICAMENTE
    const setClause = Object.keys(camposActualizar)
        .map(campo => `${campo} = ?`)
        .join(', ');

    const valores = Object.values(camposActualizar);
    valores.push(userId); // Agregar el ID al final

    const query = `UPDATE Usuario SET ${setClause} WHERE id_usuario = ?`;

    // 3. EJECUTAR CONSULTA
    connection.query(query, valores, (error, results) => {
        if (error) {
            console.error('Error en PATCH:', error);
            return res.status(500).json({ error: 'Error actualizando usuario' });
        }

        if (results.affectedRows === 0) {
            return res.status(404).json({ error: 'Usuario no encontrado' });
        }

        res.json({
            success: true,
            message: 'Usuario actualizado parcialmente',
            campos_actualizados: Object.keys(camposActualizar),
            cambios: results.affectedRows
        });
    });
});

// 🔹 DELETE - Eliminar (desactivar) un usuario
app.delete('/api/usuarios/:id', (req, res) => {
    /**
     * FUNCIÓN: Desactivar un usuario (eliminación lógica)
     * MÉTODO HTTP: DELETE
     * RUTA: /api/usuarios/:id
     */

    const userId = req.params.id;
    console.log(`Desactivando usuario ID: ${userId}`);

    // 1. VALIDACIÓN
```

```
    if (!userId || isNaN(userId)) {
        return res.status(400).json({
            success: false,
            error: 'ID de usuario inválido'
        });
    }

    // 2. CONSULTA SQL (eliminación lógica - no borrar físicamente)
    const query = `UPDATE Usuario SET activo = false WHERE id_usuario = ?`;

    // 3. EJECUTAR CONSULTA
    connection.query(query, [userId], (error, results) => {
        if (error) {
            console.error('Error en DELETE:', error);
            return res.status(500).json({ error: 'Error desactivando usuario' });
        }

        if (results.affectedRows === 0) {
            return res.status(404).json({
                success: false,
                error: 'Usuario no encontrado'
            });
        }

        res.json({
            success: true,
            message: 'Usuario desactivado correctamente'
        });
    });
});

// INICIAR SERVIDOR

app.listen(port, () => {
    console.log(`Servidor CRUD Usuarios ejecutándose en: http://localhost:${port}`);
    console.log('Endpoints disponibles:');
    console.log('POST   /api/usuarios     - Crear usuario');
    console.log('GET    /api/usuarios     - Listar usuarios');
    console.log('GET    /api/usuarios/:id - Obtener usuario específico');
    console.log('PUT    /api/usuarios/:id - Actualizar usuario completo');
    console.log('PATCH  /api/usuarios/:id - Actualizar usuario parcial');
    console.log('DELETE /api/usuarios/:id - Desactivar usuario');
});
```

ARCHIVO DE PRUEBAS

```
### 1. CREAR USUARIO
POST http://localhost:3000/api/usuarios
Content-Type: application/json

{
  "nombre": "Ana García",
  "correo": "ana@empresa.com",
  "psw": "123456",
  "rol": "admin"
}

### 2. CREAR OTRO USUARIO
POST http://localhost:3000/api/usuarios
Content-Type: application/json

{
  "nombre": "Carlos López",
  "correo": "carlos@empresa.com",
  "psw": "123456",
  "rol": "vendedor"
}

### 3. LISTAR TODOS LOS USUARIOS
GET http://localhost:3000/api/usuarios

### 4. OBTENER USUARIO ESPECÍFICO
GET http://localhost:3000/api/usuarios/1

### 5. ACTUALIZACIÓN COMPLETA (PUT)
PUT http://localhost:3000/api/usuarios/1
Content-Type: application/json

{
  "nombre": "Ana García Actualizada",
  "correo": "ana.actualizada@empresa.com",
  "rol": "admin",
  "activo": true
}

### 6. ACTUALIZACIÓN PARCIAL (PATCH)
PATCH http://localhost:3000/api/usuarios/2
Content-Type: application/json

{
  "rol": "gestor"
}
```

7. DESACTIVAR USUARIO
DELETE http://localhost:3000/api/usuarios/2

CRUD USUARIOS MODULARIZADO - MVC

ESTRUCTURA DE ARCHIVOS

```
crm_project/
├── app.js                  # Punto de entrada principal
├── config/
│   └── database.js         # Configuración de la base de datos
├── routes/
│   └── usuarios.routes.js  # Rutas de usuarios
│   └── clientes.routes.js  # Rutas de clientes
├── controllers/
│   └── usuarios.controller.js # Lógica de negocio de usuarios
├── models/
│   └── usuario.model.js    # Acceso a datos de usuarios
├── services/
│   └── usuarios.service.js # Lógica de negocio compleja (opcional)
└── package.json
```

PASO 0: ACTUALIZAR app.js

Archivo: `app.js` (versión modular)

```
// app.js - VERSIÓN MODULAR
// RESPONSABILIDAD: Configurar y arrancar el servidor

const express = require('express');
const app = express();
const port = 3000;

// 1. IMPORTAR RUTAS
const usuariosRoutes = require('./routes/usuarios.routes');

// 2. MIDDLEWARE GLOBALES
app.use(express.json()); // Parsear JSON
app.use(express.urlencoded({ extended: true })); // Parsear formularios

// 3. MIDDLEWARE DE LOGGING (opcional)
app.use((req, res, next) => {
    console.log(`${new Date().toISOString()} - ${req.method} ${req.url}`);
    next();
});

// 4. CONFIGURAR RUTAS
app.use('/api/usuarios', usuariosRoutes);
// Todas las rutas en usuarios.routes.js tendrán el prefijo /api/usuarios

// 5. RUTA RAÍZ
app.get('/', (req, res) => {
    res.json({
        message: 'API CRM Escuela - Bienvenido',
        version: '1.0.0',
        endpoints: {
            usuarios: '/api/usuarios'
        }
    });
});

// 6. MANEJO DE RUTAS NO ENCONTRADAS (404)
app.use((req, res) => {
    res.status(404).json({
        success: false,
        error: 'Ruta no encontrada'
    });
});

// 7. MANEJO DE ERRORES GLOBAL
```

```
app.use((err, req, res, next) => {
    console.error('Error global:', err);
    res.status(500).json({
        success: false,
        error: 'Error interno del servidor'
    });
});

// 8. INICIAR SERVIDOR
app.listen(port, () => {
    console.log(`Servidor modular ejecutándose en: http://localhost:${port}`);
    console.log('Endpoints disponibles:');
    console.log('   GET    /               - Documentación API');
    console.log('   POST   /api/usuarios   - Crear usuario');
    console.log('   GET    /api/usuarios   - Listar usuarios');
    console.log('   GET    /api/usuarios/:id - Obtener usuario');
    console.log('   PUT    /api/usuarios/:id - Actualizar usuario');
    console.log('   DELETE /api/usuarios/:id - Desactivar usuario');
});
```

PASO 1: CONFIGURACIÓN DE BASE DE DATOS SEPARADA

Archivo: `config/database.js`

```
// config/database.js
// RESPONSABILIDAD ÚNICA: Solo manejar la conexión a la base de datos

const mysql = require('mysql2');

// 1. Crear el pool de conexiones (MEJOR que una sola conexión)
const pool = mysql.createPool({
    host: 'localhost',
    user: 'root',
    password: '',
    database: 'crm_db',
    waitForConnections: true,
    connectionLimit: 10,          // Máximo de conexiones simultáneas
    queueLimit: 0
});

// 2. Convertir a Promesas (async/await)
const promisePool = pool.promise();

// 3. Probar la conexión
promisePool.getConnection()
    .then(connection => {
        console.log('Conectado a MySQL (pool de conexiones)');
        connection.release(); // Liberar la conexión de prueba
    })
    .catch(err => {
        console.error('Error conectando a MySQL:', err.message);
    });

// 4. Exportar para usar en otros archivos
module.exports = promisePool;
```

EXPLICACIÓN:

- `pool` vs `createConnection`: El pool maneja múltiples conexiones automáticamente
- `promise()`: Convierte callbacks en Promesas (más fácil con async/await)
- `module.exports`: Hace que este objeto sea accesible desde otros archivos

PASO 2: MODELO DE DATOS

Archivo: `models/usuario.model.js`

```
// models/usuario.model.js
// RESPONSABILIDAD: Acceso directo a la base de datos (CRUD básico)

const db = require('../config/database');

class UsuarioModel {
    // CREATE - Crear nuevo usuario
    static async create(nombre, correo, psw, rol = 'vendedor') {
        const query = `INSERT INTO Usuario (nombre, correo, psw, rol) VALUES (?, ?, ?, ?)`;
        const [result] = await db.execute(query, [nombre, correo, psw, rol]);
        return result.insertId; // Retorna el ID del nuevo usuario
    }

    // READ - Obtener todos los usuarios
    static async getAll(activo = 1) {
        const query = `SELECT id_usuario, nombre, correo, rol, fecha_alta, activo
                FROM Usuario WHERE activo = ? ORDER BY nombre`;
        const [rows] = await db.execute(query, [activo]);
        return rows; // Retorna array de usuarios
    }

    // READ - Obtener usuario por ID
    static async getById(id) {
        const query = `SELECT id_usuario, nombre, correo, rol, fecha_alta, activo
                FROM Usuario WHERE id_usuario = ?`;
        const [rows] = await db.execute(query, [id]);
        return rows[0] || null; // Retorna el usuario o null si no existe
    }

    // READ - Obtener usuario por email
    static async getByEmail(correo) {
        const query = `SELECT * FROM Usuario WHERE correo = ?`;
        const [rows] = await db.execute(query, [correo]);
        return rows[0] || null;
    }

    // UPDATE - Actualizar usuario completo
    static async update(id, nombre, correo, rol, activo) {
        const query = `UPDATE Usuario SET nombre=?, correo=?, rol=?, activo=?
                WHERE id_usuario=?`;
        const [result] = await db.execute(query, [nombre, correo, rol, activo, id]);
        return result.affectedRows; // Retorna número de filas afectadas
    }

    // UPDATE - Actualización parcial
```

```
    static async partialUpdate(id, campos) {
        // Construir SET dinámicamente
        const setClause = Object.keys(campos)
            .map(key => `${key}=?`)
            .join(', ');

        const valores = Object.values(campos);
        valores.push(id);

        const query = `UPDATE Usuario SET ${setClause} WHERE id_usuario=?`;
        const [result] = await db.execute(query, valores);
        return result.affectedRows;
    }

    // DELETE - Desactivar usuario (eliminación lógica)
    static async deactivate(id) {
        const query = `UPDATE Usuario SET activo=false WHERE id_usuario=?`;
        const [result] = await db.execute(query, [id]);
        return result.affectedRows;
    }
}

module.exports = UsuarioModel;
```

EXPLICACIÓN:

- `static async`: Métodos que se pueden usar sin crear instancia
- `await db.execute()`: Ejecuta consultas SQL de forma asíncrona
- `[result]` y `[rows]`: Destructuring del resultado de la promesa
- Clase vs Objeto: Usamos clase para organizar métodos relacionados

PASO 3: CONTROLADOR

Archivo: `controllers/usuarios.controller.js`

```
// controllers/usuarios.controller.js
// RESPONSABILIDAD: Manejar la lógica de negocio y las respuestas HTTP

const UsuarioModel = require('../models/usuario.model');

class UsuariosController {
    // CREATE - Crear nuevo usuario
    static async crearUsuario(req, res) {
        try {
            const { nombre, correo, psw, rol } = req.body;

            // Validaciones
            if (!nombre || !correo || !psw) {
                return res.status(400).json({
                    success: false,
                    error: 'Faltan campos obligatorios: nombre, correo, psw'
                });
            }

            // Verificar si el correo ya existe
            const usuarioExistente = await UsuarioModel.getByEmail(correo);
            if (usuarioExistente) {
                return res.status(409).json({
                    success: false,
                    error: 'El correo electrónico ya está registrado'
                });
            }

            // Crear usuario en la base de datos
            const nuevoId = await UsuarioModel.create(nombre, correo, psw, rol);

            // Respuesta exitosa
            res.status(201).json({
                success: true,
                message: 'Usuario creado exitosamente',
                data: { id: nuevoId, nombre, correo, rol }
            });

        } catch (error) {
            console.error('Error en crearUsuario:', error);
            res.status(500).json({
                success: false,
                error: 'Error interno del servidor'
            });
        }
```

```
}

// READ - Obtener todos los usuarios
static async obtenerUsuarios(req, res) {
    try {
        const { activo = '1' } = req.query;
        const usuarios = await UsuarioModel.getAll(parseInt(activo));

        res.json({
            success: true,
            count: usuarios.length,
            usuarios
        });

    } catch (error) {
        console.error('Error en obtenerUsuarios:', error);
        res.status(500).json({
            success: false,
            error: 'Error obteniendo usuarios'
        });
    }
}

// READ - Obtener usuario por ID
static async obtenerUsuario(req, res) {
    try {
        const { id } = req.params;

        if (!id || isNaN(id)) {
            return res.status(400).json({
                success: false,
                error: 'ID de usuario inválido'
            });
        }

        const usuario = await UsuarioModel.getById(parseInt(id));

        if (!usuario) {
            return res.status(404).json({
                success: false,
                error: 'Usuario no encontrado'
            });
        }

        res.json({
            success: true,
            usuario
        });
```

```
    } catch (error) {
        console.error('Error en obtenerUsuario:', error);
        res.status(500).json({
            success: false,
            error: 'Error obteniendo usuario'
        });
    }
}

// UPDATE - Actualizar usuario completo
static async actualizarUsuario(req, res) {
    try {
        const { id } = req.params;
        const { nombre, correo, rol, activo } = req.body;

        // Validaciones
        if (!nombre || !correo || !rol || typeof activo !== 'boolean') {
            return res.status(400).json({
                success: false,
                error: 'Faltan campos obligatorios'
            });
        }

        const filasAfectadas = await UsuarioModel.update(
            parseInt(id), nombre, correo, rol, activo
        );

        if (filasAfectadas === 0) {
            return res.status(404).json({
                success: false,
                error: 'Usuario no encontrado'
            });
        }

        res.json({
            success: true,
            message: 'Usuario actualizado correctamente'
        });

    } catch (error) {
        console.error('Error en actualizarUsuario:', error);
        res.status(500).json({
            success: false,
            error: 'Error actualizando usuario'
        });
    }
}
```

```
    // DELETE - Desactivar usuario
    static async desactivarUsuario(req, res) {
        try {
            const { id } = req.params;

            if (!id || isNaN(id)) {
                return res.status(400).json({
                    success: false,
                    error: 'ID de usuario inválido'
                });
            }

            const filasAfectadas = await UsuarioModel.deactivate(parseInt(id));

            if (filasAfectadas === 0) {
                return res.status(404).json({
                    success: false,
                    error: 'Usuario no encontrado'
                });
            }

            res.json({
                success: true,
                message: 'Usuario desactivado correctamente'
            });

        } catch (error) {
            console.error('Error en desactivarUsuario:', error);
            res.status(500).json({
                success: false,
                error: 'Error desactivando usuario'
            });
        }
    }
}

module.exports = UsuariosController;
```

EXPLICACIÓN:
- `try/catch`: Manejo de errores centralizado
- `static async`: Métodos de clase que no necesitan instancia
- Validaciones: Lógica de negocio separada de la base de datos
- Respuestas HTTP: Formato consistente para todas las respuestas

PASO 4: RUTAS

Archivo: `routes/usuarios.routes.js`

```
// routes/usuarios.routes.js
// RESPONSABILIDAD: Definir las rutas y asignarlas a controladores

const express = require('express');
const router = express.Router(); // Crea un enrutador específico
const UsuariosController = require('../controllers/usuarios.controller');

// RUTAS DE USUARIOS

// POST /api/usuarios - Crear nuevo usuario
router.post('/', UsuariosController.crearUsuario);

// GET /api/usuarios - Obtener todos los usuarios
router.get('/', UsuariosController.obtenerUsuarios);

// GET /api/usuarios/:id - Obtener usuario por ID
router.get('/:id', UsuariosController.obtenerUsuario);

// PUT /api/usuarios/:id - Actualizar usuario completo
router.put('/:id', UsuariosController.actualizarUsuario);

// DELETE /api/usuarios/:id - Desactivar usuario
router.delete('/:id', UsuariosController.desactivarUsuario);

// Exportar el router para usarlo en app.js
module.exports = router;
```

EXPLICACIÓN:
- `express.Router()`: Crea un mini-enrutador para agrupar rutas relacionadas
- Métodos HTTP claros: POST, GET, PUT, DELETE
- Parámetros de ruta: `:id` captura el ID de la URL
- Controladores asignados: Cada ruta llama a un método específico del controlador

PASO 5: ARCHIVO package.json

- Comprobamos que están las dependencias que queremos instalar y modificamos los scripts de arranque a nuestro gusto

```
{
  "name": "crm-escuela",
  "version": "1.0.0",
  "description": "CRM educativo con arquitectura MVC",
  "main": "app.js",
  "scripts": {
    "start": "node app.js",
    "dev": "nodemon app.js",
    "test": "echo \"No hay tests configurados\" && exit 1"
  },
  "dependencies": {
    "express": "^4.18.0",
    "mysql2": "^3.0.0"
  },
  "devDependencies": {
    "nodemon": "^3.0.0"
  }
}
```

COMPARACIÓN: MONOLÍTICO vs MODULAR

VERSIÓN MONOLÍTICA (TODO EN app.js)
VENTAJAS:
- Más simple para empezar
- Todo en un solo archivo
- Fácil de entender inicialmente

DESVENTAJAS:
- Se vuelve gigante rápidamente
- Difícil de mantener
- No se puede reutilizar código
- Todo el equipo trabaja en el mismo archivo

VERSIÓN MODULAR (MVC)
VENTAJAS:
-Separación de responsabilidades: Cada archivo tiene una tarea clara
- Mantenibilidad: Fácil encontrar y arreglar bugs
- Reutilización: Los modelos se usan en múltiples controladores
- Trabajo en equipo: Cada persona trabaja en archivos diferentes
- Escalabilidad: Fácil añadir nuevas funcionalidades

EJEMPLO PRÁCTICO:
Si mañana queremos añadir "Productos":
1. Crear models/producto.model.js
2. Crear controllers/productos.controller.js
3. Crear routes/productos.routes.js
4. Añadir en app.js: app.use('/api/productos', productosRoutes);

No habría que tocar el código de usuarios

PARA PROBAR LA VERSIÓN MODULAR:

1. Instalar dependencias si no lo has hecho
npm install

2. Ejecutar con nodemon (reinicia automáticamente)
npm run dev

3. Probar con Thunder Client o Postman:
   POST http://localhost:3000/api/usuarios
   GET http://localhost:3000/api/usuarios