



Construyendo una API REST con Node JS y MySQL

Facultad de Ingeniería y Ciencias Naturales, Universidad de Sonsonate

Jeffrey A. Argueta, Christopher B. Álvarez, Christopher R. Gutierrez,

Nayeli M. Guzmán, Carlos F. Chavarría

I04-2-08: Base de Datos I

Ing. Daro Cristian Arias Jaco

Ciclo 01 – 2025

Índice

Introducción.....	3
Objetivos Generales.....	4
Objetivos Específicos	4
¿Qué es una API?.....	5
Características clave de una API	5
¿Qué es una API REST?.....	5
Principios clave de una API REST	6
Ventajas de las API REST	6
APIs REST populares	7
Configuración de entorno	7
Modelado de la base de datos	8
Estructura inicial del proyecto API.....	8
Ejemplo básico de una API REST.....	9
Optimización de la API	11
Seguridad y Autenticación.....	14
Manejo de errores y validación	16

Introducción

En la era de las aplicaciones modernas, las **APIs REST** se han convertido en una pieza fundamental para la comunicación entre sistemas. Ya sea en aplicaciones web, móviles o servicios de terceros, contar con una API bien estructurada y eficiente permite la integración fluida y escalable entre diferentes plataformas.

Exploraremos desde los fundamentos hasta la implementación práctica de una **API REST utilizando Node.js y MySQL**. A través de ejemplos reales, aprenderemos a diseñar endpoints, gestionar bases de datos, implementar autenticación y aplicar buenas prácticas para construir una API segura y mantenible. Además de la teoría, pondremos énfasis en la **modularidad, documentación clara y seguridad**, aspectos clave para desarrollar soluciones robustas y escalables. Al finalizar, los participantes tendrán una comprensión sólida sobre cómo construir su propia API, integrarla con una base de datos relacional y prepararla para un entorno de producción.

Este taller está diseñado para desarrolladores que desean profundizar en el backend y mejorar sus habilidades en desarrollo de APIs, arquitectura de software y gestión de bases de datos.

Objetivos Generales

1. Comprender los fundamentos de las API REST
2. Configurar un entorno de desarrollo backend
3. Diseñar y gestionar una base de datos en MySQL
4. Implementar endpoints de una API REST
5. Probar y optimizar el funcionamiento de la API
6. Aplicar principios de seguridad en una API REST
7. Adquirir mejores prácticas de desarrollo backend

Objetivos Específicos

1. Definir qué es una API REST y cómo funciona
2. Conocer las ventajas de usar Node.js y MySQL en el desarrollo backend
3. Instalar y configurar Node.js, MySQL y herramientas complementarias
4. Establecer la estructura básica de un proyecto con Express
5. Usar Sequelize para la gestión de modelos en Node.js
6. Construir rutas (GET, POST, PUT, DELETE) siguiendo buenas prácticas
7. Aplicar middleware para seguridad y validaciones
8. Utilizar Postman para validar los endpoints
9. Manejar errores y optimizar consultas en MySQL para mejorar el rendimiento
10. Implementar autenticación con JWT
11. Prevenir vulnerabilidades comunes como SQL Injection y CORS
12. Entender modularidad y escalabilidad en proyectos backend

¿Qué es una API?

Una **API (Application Programming Interface)** es un conjunto de reglas y herramientas que permiten la comunicación entre diferentes software o sistemas. En términos simples, una API actúa como un intermediario que facilita el intercambio de datos y funcionalidades entre aplicaciones sin que estas tengan que conocer los detalles internos de cada una.

Características clave de una API

- **Interoperabilidad:** Permite que diferentes aplicaciones, incluso con tecnologías distintas, se comuniquen entre sí.
- **Estandarización:** Usa protocolos y formatos comunes, como REST, GraphQL, SOAP o RPC.
- **Seguridad:** Implementa autenticación y control de acceso para proteger la información.
- **Abstracción:** Oculta la complejidad interna de un sistema y expone solo lo necesario para su uso.

¿Qué es una API REST?

Una **API REST (Representational State Transfer)** es un conjunto de reglas y principios que permiten la comunicación entre sistemas utilizando el protocolo HTTP. Su diseño sigue principios específicos que aseguran que la interacción sea eficiente, escalable y fácil de consumir.

Principios clave de una API REST

1. **Recursos bien definidos:** Cada elemento de la API (usuarios, productos, pedidos) se representa mediante una URL única.
2. **Métodos HTTP estandarizados:** Se utilizan métodos como GET (obtener datos), POST (crear datos), PUT (actualizar datos) y DELETE (eliminar datos).
3. **Stateless:** Cada solicitud es independiente y contiene toda la información necesaria, sin depender de estados previos del servidor.
4. **Formato de respuesta ligero:** Se usan formatos como JSON o XML para intercambiar datos de forma estructurada y eficiente.
5. **Uso de códigos de estado HTTP:** La API devuelve códigos como 200 OK, 404 Not Found o 500 Internal Server Error, para indicar el resultado de la solicitud.

Ventajas de las API REST

- ✓ **Simplicidad y facilidad de uso:** Utilizan HTTP como protocolo de comunicación, lo que facilita su integración en aplicaciones sin necesidad de configuraciones complejas.
- ✓ **Estructura uniforme:** Siguen principios estándar, como recursos representados por URLs y operaciones CRUD (**Create, Read, Update, Delete**), lo que permite diseñarlas de forma intuitiva.
- ✓ **Interoperabilidad:** Pueden ser consumidas por cualquier tecnología o plataforma que soporte HTTP, desde **JavaScript en el frontend** hasta **Node.js, Python o Java en el backend**.

- ✓ **Escalabilidad:** Son adecuadas para sistemas distribuidos y aplicaciones que deben manejar grandes volúmenes de tráfico sin afectar el rendimiento.
- ✓ **Sin estado:** Cada solicitud es independiente, lo que simplifica la gestión de sesiones y permite un alto rendimiento en arquitecturas de microservicios.
- ✓ **Compatibilidad con caché:** Las respuestas pueden almacenarse para reducir la carga en el servidor y mejorar la velocidad de las solicitudes.

APIs REST populares

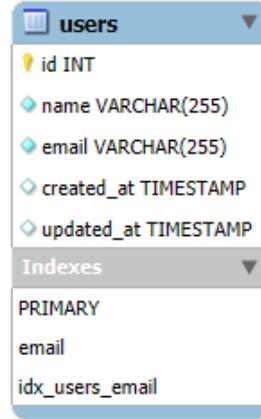
- **Google Maps API:** Permite integrar mapas interactivos y datos de geolocalización en aplicaciones.
- **Giphy API:** Permite buscar y compartir GIFs animados en aplicaciones y sitios web.
- **NASA API:** Ofrece acceso a datos astronómicos, imágenes y eventos espaciales.
- **AWS API Gateway:** Facilita la creación y gestión de APIs en la nube con Amazon Web Services.

Configuración de entorno

- **Node.js** (incluye npm) → [Descargar aquí](#)
- **MySQL** (Base de datos) → [Descargar aquí](#)
- **Postman** (Opcional, para probar las rutas HTTP) → [Descargar aquí](#)
- **MySQL Workbench** (Administrador) → [Descargar aquí](#)
- **VS Code** (Editor de código) → [Descargar aquí](#)

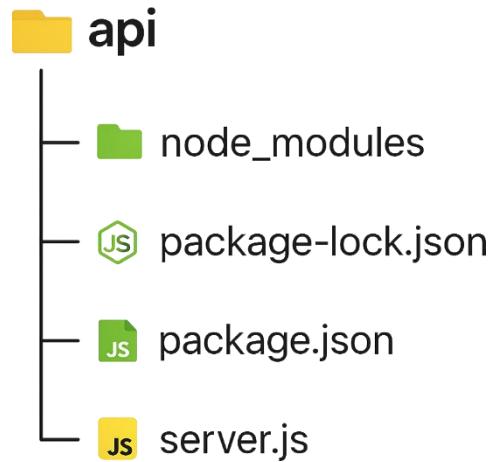
Modelado de la base de datos

Herramientas: **MySQL** y **MySQL Workbench**



Estructura inicial del proyecto API

Herramientas: **Node.js** y **VS Code**



Ejemplo básico de una API REST

Ejemplo básico de una API REST utilizando **Node.js** con **Express** y **MySQL** a través de **Sequelize**. Este código define una API REST para gestionar usuarios, permitiendo operaciones CRUD (**Create, Read, Update, Delete**).

1. Instala las dependencias necesarias:

```
Terminal
npm init -y
npm install express sequelize mysql2
```

2. Crea el archivo `server.js` con la configuración del servidor:

```
Javascript
const express = require("express");
const app = express();
const { Sequelize, DataTypes } = require("sequelize");

const sequelize = new Sequelize("prueba_api", "root", "admin", {
  host: "localhost",
  dialect: "mysql",
  logging: console.log
});

sequelize.authenticate()
  .then(() => console.log("✅ Connected to MySQL
successfully"))
  .catch(err => console.error("❌ Connection error:", err));

const User = sequelize.define("User", {
  id: { type: DataTypes.INTEGER, primaryKey: true,
  autoIncrement: true },
  name: { type: DataTypes.STRING, allowNull: false },
  email: { type: DataTypes.STRING, allowNull: false, unique:
true }
}, {
  timestamps: true,
  createdAt: "created_at",
  updatedAt: "updated_at",
  tableName: "users"
});

app.use(express.json());
```

```

app.get("/users", async (req, res) => {
  const users = await User.findAll();
  res.json(users);
});

app.post("/users", async (req, res) => {
  const user = await User.create(req.body);
  res.json(user);
});

app.put("/users/:id", async (req, res) => {
  const user = await User.findByPk(req.params.id);
  if (user) {
    await user.update(req.body);
    res.json(user);
  } else {
    res.status(404).json({ error: "Usuario no encontrado" });
  }
});

app.delete("/users/:id", async (req, res) => {
  const user = await User.findByPk(req.params.id);
  if (user) {
    await user.destroy();
    res.json({ message: "Usuario eliminado" });
  } else {
    res.status(404).json({ error: "Usuario no encontrado" });
  }
});

app.listen(3000, async () => {
  console.log("🌐 Servidor corriendo en  
http://localhost:3000");
});

```

Express: es un framework minimalista y flexible para Node.js que se utiliza para desarrollar aplicaciones web y APIs de manera eficiente. Es ampliamente adoptado en el ecosistema JavaScript debido a su facilidad de uso y su capacidad para manejar enrutamiento, middleware y peticiones HTTP de forma sencilla.

Algunas características clave de Express.js incluyen:

- **Sistema de enrutamiento** para gestionar diferentes rutas en una aplicación.
- **Middleware** para procesar solicitudes antes de enviarlas a la lógica principal.
- **Compatibilidad con plantillas** para generar contenido dinámico.

- **Integración con bases de datos** y otras herramientas de backend.

Sequelize: es un **ORM (Object-Relational Mapping)** para Node.js que facilita la interacción con bases de datos relacionales como **MySQL, PostgreSQL, SQLite y MSSQL**. Permite definir modelos de datos en JavaScript y ejecutar consultas sin necesidad de escribir SQL directamente.

Algunas características clave de Sequelize:

- **Definición de modelos** con validaciones y relaciones entre tablas.
- **Soporte para transacciones** y consultas avanzadas.
- **Eager y lazy loading** para optimizar el acceso a datos.
- **Migraciones** para gestionar cambios en la estructura de la base de datos.

CRUD: es un acrónimo que representa las cuatro operaciones fundamentales en la gestión de datos dentro de una aplicación:

- **Create (Crear):** Permite agregar nuevos registros a una base de datos.
- **Read (Leer):** Recupera y muestra datos almacenados.
- **Update (Actualizar):** Modifica registros existentes.
- **Delete (Eliminar):** Borra datos de la base de datos.

Estas operaciones son esenciales en el desarrollo de aplicaciones que manejan información persistente, como sistemas de gestión de usuarios, plataformas de comercio electrónico y APIs RESTful.

Optimización de la API

1. Optimización de consultas en Sequelize

- ✓ Carga selectiva de columnas

En tu consulta `findAll()`, estás trayendo todos los datos de la tabla. Si solo necesitas ciertos campos, usa `attributes`:

```
Javascript
app.get("/users", async (req, res) => {
  const users = await User.findAll({
    attributes: ["id", "name", "email"]
  });
  res.json(users);
});
```

Esto reducirá el tamaño de la respuesta y acelerará la consulta.

- ✓ Paginación eficiente

Si tu tabla crece, traer todos los registros puede ser costoso. Implementa paginación con `limit` y `offset`:

```
Javascript
app.get("/users", async (req, res) => {
  const { page = 1, limit = 10 } = req.query;
  const users = await User.findAll({
    limit: parseInt(limit),
    offset: (page - 1) * limit,
    attributes: ["id", "name", "email"]
  });
  res.json(users);
});
```

Esto evita cargar más datos de los necesarios y mejora el rendimiento.

2. Middleware para Manejo de Errores

Actualmente, los errores se manejan dentro de cada ruta. Para mayor claridad y mantenimiento, centraliza la captura de errores en un middleware:

```
Javascript
app.use((err, req, res, next) => {
  console.error("✗ Error:", err.message);

  res.status(err.status || 500).json({
    error: err.message || "Error interno del servidor"
  });
});
```

Dentro de cada controlador, usa `try/catch` y pasa el error a `next(err)`.

3. Indexación en Base de Datos

Si la tabla `users` tiene muchos registros, asegurarte de que `email` esté indexado acelerará la consulta:

Sql
<pre>CREATE INDEX idx_users_email ON users(email);</pre>

Esto hace que las búsquedas por `email` sean más rápidas.

4. Conexiones a Base de Datos con Pooling

Evita crear una nueva conexión en cada consulta. Usa **pooling** en Sequelize:

Javascript
<pre>const sequelize = new Sequelize("prueba_api", "root", "admin", { host: "localhost", dialect: "mysql", logging: false, pool: { max: 10, // Máximo de conexiones simultáneas min: 2, acquire: 30000, idle: 10000 } });</pre>

Esto reduce el número de conexiones innecesarias.

5. Validaciones en POST

Validaciones en `req.body` antes de `User.create(req.body)` Actualmente, si POST `/users` recibe datos erróneos, Sequelize puede fallar. Mejora la validación de entrada:

Javascript

```
app.post("/users", async (req, res, next) => {
  try {
    const { name, email } = req.body;
    if (!name || !email) return res.status(400).json({ error: "Nombre y email son obligatorios" });

    const user = await User.create({ name, email });
    res.json(user);
  } catch (err) {
    next(err);
  }
});
```

Seguridad y Autenticación

1. Instalar las dependencias necesarias:

Terminal
npm install jsonwebtoken cors bcryptjs

2. Configurar CORS

Javascript
const cors = require("cors"); app.use(cors());

3. Implementar autenticación con JWT

Javascript
const jwt = require("jsonwebtoken"); const bcrypt = require("bcryptjs"); const SECRET_KEY = "supersecreto"; app.post("/login", async (req, res) => { try { const { email, password } = req.body; if (!email !password) return res.status(400).json({ error: "Email y contraseña son obligatorios" }); const user = await User.findOne({ where: { email } }); if (!user !bcrypt.compareSync(password, user.password)) {

```

        return res.status(401).json({ error: "Credenciales
inválidas" });
    }

    const token = jwt.sign({ id: user.id, email: user.email },
SECRET_KEY, { expiresIn: "1h" });
    res.json({ token });
} catch (err) {
    res.status(500).json({ error: "Error interno del
servidor" });
}
);

```

4. Proteger rutas con middleware

Crea un middleware que valide los tokens antes de acceder a recursos protegidos:

Javascript

```

const verifyToken = (req, res, next) => {
    const token = req.headers["authorization"];

    if (!token) return res.status(403).json({ error: "Token
requerido" });

    jwt.verify(token.split(" ")[1], SECRET_KEY, (err, decoded) =>
{
        if (err) return res.status(401).json({ error: "Token
inválido" });

        req.user = decoded;
        next();
    });
};

```

5. Hashear contraseñas antes de guardarlas

Modifica la creación de usuarios para almacenar contraseñas encriptadas:

Javascript

```

app.post("/users", async (req, res, next) => {
  try {
    const { name, email, password } = req.body;
    if (!name || !email || !password) return
    res.status(400).json({ error: "Nombre, email y contraseña son obligatorios" });

    const hashedPassword = bcrypt.hashSync(password, 10);

    const user = await User.create({ name, email, password: hashedPassword });
    res.json(user);
  } catch (err) {
    next(err);
  }
});

```

Manejo de errores y validación

1. Instalar las dependencias necesarias:

Terminal
npm install express-validator winston

2. Configurar Winston para monitoreo

Javascript
<pre> const winston = require("winston"); const logger = winston.createLogger({ level: "info", format: winston.format.combine(winston.format.timestamp(), winston.format.json()), transports: [new winston.transports.Console(), new winston.transports.File({ filename: "logs/api.log" })] }); </pre>

Guarda logs en logs/api.log, Imprime en consola en desarrollo, Incluye marca de tiempo en cada log.

3. Implementar validación con express-validator

Modifica la ruta POST /users para validar los datos antes de procesarlos:

```
Javascript

const { body, validationResult } = require("express-validator");

app.post("/users",
[
    body("name").notEmpty().withMessage("El nombre es obligatorio"),
    body("email").isEmail().withMessage("Debe ser un email válido"),
    body("password").isLength({ min: 6 }).withMessage("La contraseña debe tener al menos 6 caracteres")
],
async (req, res, next) => {
    const errors = validationResult(req);
    if (!errors.isEmpty()) {
        res.status(400).json({ errors: errors.array() });
    }

    try {
        const { name, email, password } = req.body;
        const hashedPassword = bcrypt.hashSync(password, 10);

        const user = await User.create({ name, email, password: hashedPassword });

        logger.info(`Usuario creado: ${user.email}`);

        res.json(user);
    } catch (err) {
        logger.error(`Error al crear usuario: ${err.message}`);
        next(err);
    }
});
);
```

Verifica que el nombre no esté vacío. Valida el formato del email. Asegura que la contraseña tenga al menos 6 caracteres. Registra eventos de creación y errores con Winston.

4. Registrar errores globalmente

Modifica el middleware de errores para incluir logs con Winston:

```

Javascript
app.use((err, req, res, next) => {
  logger.error(`❌ Error en la API: ${err.message}`);
  res.status(err.status || 500).json({ error: err.message || "Error interno del servidor" });
});

```

Cada error se guarda en los logs (api.log). Facilita el monitoreo de fallos en producción.

Modularización y Escalabilidad

1. Importancia de la Modularización

- **Facilita el mantenimiento:** Al dividir el código en módulos independientes, es más fácil localizar y corregir errores sin afectar otras partes del sistema.
- **Mejora la reutilización:** Puedes usar los mismos componentes en múltiples partes del proyecto o incluso en otros proyectos.
- **Fomenta la claridad:** Un código modular es más legible, lo que facilita la colaboración con otros desarrolladores.
- **Reducción del acoplamiento:** Evita que los cambios en un módulo afecten a todo el sistema, haciendo que el desarrollo sea más flexible.

Ejemplo: En tu API, tener services/userService.js separado de controllers/userController.js permite modificar la **lógica de negocio** sin alterar cómo se manejan las solicitudes HTTP.

2. Importancia de la Escalabilidad

- **Soporta crecimiento del sistema:** Un código escalable permite agregar nuevas funcionalidades sin **romper** el sistema existente.

- **Optimiza el rendimiento:** Puedes mejorar el manejo de **cargas** al usar técnicas como optimización de consultas con **Sequelize pooling**, etc.
- **Preparado para expansión:** Si necesitas migrar a una arquitectura de **microservicios**, un código escalable lo hace más sencillo.

Ejemplo: Si tu API empieza con autenticación JWT y luego decides integrar **OAuth**, la modularización facilita la integración sin modificar la estructura principal.