CENTRO FP SUPERIOR
CAMPUS CÁMARA COMERCIO SEVILLA

Curso: **Desarrollo de Aplicaciones Aplicaciones Multiplataforma/Web**
Módulo Profesional: DWEC
Profesor: Carlos Basulto Pardo
GIT: https://github.com/CarlosBasulto/ApiRest_Node_ORM_Sequelize
2 ramas: Main y Modelado

Refactorización a MVC reducida y AutoCRUD con Sequelize (Node.js)

Resultados de Aprendizaje y Conceptos básicos

Resultados de aprendizaje y criterios de evaluación.

NODE

- Taller Api Rest con Express
- >Entender las ROUTE
- > Entender los verbos y API Rest

Contenidos básicos:

− Modelos de programación en entornos cliente/servidor.
− Mecanismos de ejecución de código en un navegador Web.
− Capacidades y limitaciones de ejecución. Compatibilidad con navegadores Web.
− Lenguajes de programación en entorno cliente.
− Características de los lenguajes de script. Ventajas y desventajas sobre la programación tradicional.
− Tecnologías y lenguajes asociados.
− Integración del código con las etiquetas HTML.
− Herramientas de programación.

Contenidos ampliados:
- Iniciación Angular.
- Iniciación Typescript
- Comprensión de principales arquitecturas Clientes.

Glosario

RA: Resultado de aprendizaje
CEV: Criterio de evaluación
AF: Actividades formativas
AEE: Actividades específicas evaluables

CENTRO FP SUPERIOR
CAMPUS CÁMARA COMERCIO SEVILLA

Curso: **Desarrollo de Aplicaciones Aplicaciones Multiplataforma/Web**
Módulo Profesional: DWEC
Profesor: Carlos Basulto Pardo
GIT: https://github.com/CarlosBasulto/ApiRest_Node_ORM_Sequelize
2 ramas: Main y Modelado

AEER:Actividades específicas evaluables de recuperación
AERF: Actividades específicas de refuerzo
AEEA: Actividades específicas evaluables de ampliación
AEFA: Actividades específicas formativas de ampliación
AC: Actividades complementarias

CENTRO FP SUPERIOR
CAMPUS CÁMARA COMERCIO SEVILLA

Curso: **Desarrollo de Aplicaciones Aplicaciones Multiplataforma/Web**
Módulo Profesional: DWEC
Profesor: Carlos Basulto Pardo
GIT: https://github.com/CarlosBasulto/ApiRest_Node_ORM_Sequelize
2 ramas: Main y Modelado

Práctica: Refactorización a MVC reducida y AutoCRUD con Sequelize (Node.js)

Contexto

Partiendo del taller realizado en clase (proyecto Node.js + Sequelize con generación de endpoints CRUD), se solicita aplicar una serie de cambios para evolucionar el proyecto hacia una **arquitectura MVC reducida**, manteniendo la idea clave del taller: **generación automática del CRUD (AutoCRUD)**.

Esta práctica se validará en clase el **viernes que viene** mediante una prueba real: se creará una **tabla nueva** y se ejecutará el AutoCRUD para comprobar que **genera todo automáticamente** (rutas + controladores + servicios).
Si no funciona en la prueba, la práctica no se aprueba.

Se facilitará un **proyecto base** para el alumnado que no realizó el taller, disponible en la **rama "Modelado"**.

Objetivo general

Modificar el AutoCRUD para que, además de generar el CRUD, genere el proyecto siguiendo una **estructura MVC reducida** basada en:

- **Rutas**
- **Controladores**
- **ControladoresBase**
- **Servicios**

Además, se incorpora una tabla extra para registrar logs.

Requisitos funcionales

1) Nueva tabla Log (hecha en clase)

Debe existir en el modelo una tabla **Log** con los campos:

- `id` (PK, autoincremental)
- `log` (texto / string)

Este punto se considera realizado en clase, pero debe estar presente y funcional en la entrega.

2) AutoCRUD con arquitectura MVC reducida (obligatorio)

El sistema AutoCRUD debe generar automáticamente (por cada tabla/modelo detectado):

1. **Ruta** (archivo dentro de `routes/`)
2. **Controlador específico** (archivo dentro de `controllers/`)
3. **ControladorBase** reutilizable (archivo/clase dentro de `controllers/base/` o `controllersBase/`)
4. **Servicio** (archivo dentro de `services/`)

El resultado debe permitir, como mínimo, operaciones CRUD típicas:

- `GET /recurso` (listar)
- `GET /recurso/:id` (detalle)
- `POST /recurso` (crear)
- `PUT /recurso/:id` (actualizar completo)
- `DELETE /recurso/:id` (borrar)

Requisitos técnicos y de arquitectura

Estructura mínima exigida (MVC reducida)

El proyecto debe quedar organizado de forma clara y consistente. Ejemplo orientativo:

- `routes/` → define endpoints y delega en controlador
- `controllers/` → contiene lógica de capa controlador (usa servicios)
- `controllersBase/` (o `controllers/base/`) → controlador genérico reutilizable
- `services/` → acceso a datos (Sequelize), lógica de negocio simple
- `models/` → modelos Sequelize

Importante: **las rutas no deben contener lógica de negocio**. Solo enrutar.

Prueba obligatoria en clase (criterio de aprobado)

El viernes en clase se realizará este test:

1. Se crea una **nueva tabla/modelo** (distinto de los existentes).
2. Se ejecuta el AutoCRUD.
3. Debe generarse automáticamente:
    - la ruta
    - el controlador
    - el servicio
    - y usar correctamente el controlador base
4. Se prueban endpoints CRUD y deben responder correctamente.

Si al ejecutar el AutoCRUD no se genera correctamente o fallan los endpoints, la práctica queda suspensa.

Entregables

1. **Repositorio Git** con el proyecto completo.
2. Carpeta y estructura final cumpliendo MVC reducida.
3. AutoCRUD funcionando y documentado.
4. Un archivo `README.md` con:
    - cómo instalar dependencias
    - cómo configurar `.env`
    - cómo ejecutar migraciones/seed (si aplica)
    - cómo lanzar el servidor
    - cómo ejecutar el AutoCRUD
    - ejemplos de endpoints (mínimo 1 recurso)

Criterios de evaluación (orientativos)

- **Funciona 10**
- No funciona 0

Condiciones

- Se permite partir del taller hecho en clase.
- Quien no tenga el taller parte del proyecto de la **rama "Modelado"**.
- La práctica debe quedar lista para la **prueba en clase**.

Anexo I. autocrud a modificar

```
// autocrud.js
import fs from "fs";
import path from "path";

const modelsPath = "./models";
const controllersPath = "./controllers";
const routesPath = "./routes";

fs.mkdirSync(controllersPath, { recursive: true });
fs.mkdirSync(routesPath, { recursive: true });

// Filtramos solo los modelos (sin incluir init-models.js)
const models = fs.readdirSync(modelsPath)
  .filter(f => f.endsWith(".js") && f !== "init-models.js");

for (const modelFile of models) {
  const modelName = path.basename(modelFile, ".js"); // ejemplo: productos
  const modelClass = modelName.charAt(0).toUpperCase() + modelName.slice(1); //
Productos
  const singular = modelName.replace(/s$/, ""); // producto, cliente, pedido, etc.

  // ---------- CONTROLADOR ----------
  const controllerContent = `// controllers/${modelName}Controller.js
import { sequelize } from "../config/db.js";
import ${modelName} from "../models/${modelFile}";
import { DataTypes } from "sequelize";

// 🔧 Inicializamos el modelo con la conexión activa
const ${modelClass.slice(0, -1)} = ${modelName}.init(sequelize, DataTypes);

// CREATE
export const crear${modelClass.slice(0, -1)} = async (req, res) => {
  try {
    const nuevo = await ${modelClass.slice(0, -1)}.create(req.body);
    res.status(201).json(nuevo);
  } catch (error) {
    console.error(error);
    res.status(500).json({ mensaje: "Error al crear ${singular}", error });
  }
};

// READ (todos)
```

```
export const obtener${modelClass} = async (req, res) => {
  try {
    const lista = await ${modelClass.slice(0, -1)}.findAll();
    res.json(lista);
  } catch (error) {
    console.error(error);
    res.status(500).json({ mensaje: "Error al obtener ${modelName}", error });
  }
};

// READ (uno)
export const obtener${modelClass.slice(0, -1)} = async (req, res) => {
  try {
    const item = await ${modelClass.slice(0, -1)}.findByPk(req.params.id);
    if (!item) return res.status(404).json({ mensaje: "No encontrado" });
    res.json(item);
  } catch (error) {
    console.error(error);
    res.status(500).json({ mensaje: "Error al obtener ${singular}", error });
  }
};

// UPDATE
export const actualizar${modelClass.slice(0, -1)} = async (req, res) => {
  try {
    const item = await ${modelClass.slice(0, -1)}.findByPk(req.params.id);
    if (!item) return res.status(404).json({ mensaje: "No encontrado" });
    await item.update(req.body);
    res.json(item);
  } catch (error) {
    console.error(error);
    res.status(500).json({ mensaje: "Error al actualizar ${singular}", error });
  }
};

// DELETE
export const eliminar${modelClass.slice(0, -1)} = async (req, res) => {
  try {
    const item = await ${modelClass.slice(0, -1)}.findByPk(req.params.id);
    if (!item) return res.status(404).json({ mensaje: "No encontrado" });
    await item.destroy();
    res.json({ mensaje: "${modelClass.slice(0, -1)} eliminado correctamente" });
  } catch (error) {
    console.error(error);
    res.status(500).json({ mensaje: "Error al eliminar ${singular}", error });
  }
```

```
};
`;

  fs.writeFileSync(`${controllersPath}/${modelName}Controller.js`, controllerContent);

  // ---------- RUTA ----------
  const routeContent = `// routes/${modelName}Routes.js
import express from "express";
import {
  crear${modelClass.slice(0, -1)},
  obtener${modelClass},
  obtener${modelClass.slice(0, -1)},
  actualizar${modelClass.slice(0, -1)},
  eliminar${modelClass.slice(0, -1)}
} from "../controllers/${modelName}Controller.js";

const router = express.Router();

router.get("/", obtener${modelClass});
router.get("/:id", obtener${modelClass.slice(0, -1)});
router.post("/", crear${modelClass.slice(0, -1)});
router.put("/:id", actualizar${modelClass.slice(0, -1)});
router.delete("/:id", eliminar${modelClass.slice(0, -1)});

export default router;
`;

  fs.writeFileSync(`${routesPath}/${modelName}Routes.js`, routeContent);
  console.log(`✅ CRUD generado para: ${modelName}`);
}

console.log("🎉 Todos los controladores y rutas han sido generados correctamente.");
```

Anexo II. Herencia

Template Method Pattern (Patrón del Método Plantilla)

🔹 **Definición:**
El patrón *Template Method* define el esqueleto de un algoritmo en una clase base (padre),
permitiendo que las subclases redefinan partes específicas del proceso sin cambiar su

estructura general.

- **Aplicación en tu caso:**
  - Tu **autocrud** genera los **métodos genéricos** (`create`, `findAll`, `update`, `delete`) en controladores base.
  - Los **controladores extendidos** (por ejemplo, `productosController.js`) **redefinen** o **amplían** esos métodos.

PROBLEMA→ OJO el autocrud resetea los controladores y rutas, por lo que hay que tenerlo en cuenta a la hroa de hacer cambios en esta clase. Lo ideal es definir un patrón de herencia para cambios posteriores.

Si lanzamos:

node autocrud.js

el script:

1. Recorre los modelos de la carpeta `models/`.
2. Crea (o sobrescribe) los controladores y rutas para cada modelo.
3. No distingue entre código nuevo o modificado.

Por tanto:

- Si añadiste lógica personalizada (por ejemplo, validaciones, includes, middlewares o filtros),
- al regenerar el autocrud, **todo se borra y se sustituye** por la plantilla base del generador.

Solución conceptual: herencia o extensión de controladores

Para mantener el poder del autocrud sin perder tus personalizaciones,
puedes adoptar un **modelo de herencia o capas**:

🧩 1. Mantén el autocrud como “controladores base”

Deja que `autocrud.js` genere archivos como:

controllers/base/productosBaseController.js
controllers/base/clientesBaseController.js

Estos controladores contendrán los CRUD genéricos que pueden regenerarse sin riesgo.

🧩 2. Crea controladores extendidos en una carpeta aparte

Luego, creas tus controladores reales extendiendo esos controladores base, por ejemplo:

📄 **controllers/productosController.js**

```
import * as Base from "./base/productosBaseController.js";

export const obtenerProductos = async (req, res) => {
  try {
    // ✅ Lógica personalizada antes de llamar al base
    console.log("🧠 Cargando productos con validación personalizada...");

    // Llamar a la versión base del método
    await Base.obtenerProductos(req, res);
  } catch (error) {
    res.status(500).json({ mensaje: "Error al obtener productos", error });
  }
};

// Si no necesitas personalización, simplemente reexportas:
export const crearProducto = Base.crearProducto;
export const obtenerProducto = Base.obtenerProducto;
export const actualizarProducto = Base.actualizarProducto;
export const eliminarProducto = Base.eliminarProducto;
```

De esta forma:

- El **autocrud** solo toca los archivos de `/controllers/base/`.
- Tus **controladores personalizados** viven en `/controllers/` y no se sobrescriben.

Ajusta tus rutas para usar los controladores extendidos

📄 **routes/productosRoutes.js**

```
import express from "express";
import {
  crearProducto,
  obtenerProductos,
  obtenerProducto,
  actualizarProducto,
  eliminarProducto
} from "../controllers/productosController.js"; // ← usa el extendido

const router = express.Router();

router.get("/", obtenerProductos);
router.get("/:id", obtenerProducto);
router.post("/", crearProducto);
router.put("/:id", actualizarProducto);
router.delete("/:id", eliminarProducto);

export default router;
```

4. Estructura recomendada del proyecto

```
controllers/
├── base/
│   ├── productosBaseController.js
│   ├── categoriasBaseController.js
│   └── ...
├── productosController.js
├── categoriasController.js
└── ...
```

El **autocrud** siempre escribe en `/controllers/base/`,
y tú solo trabajas en `/controllers/` (los extendidos).