



FORMULARIO DE PRODUCCIÓN DE CONTENIDOS

NOMBRE DE LA ASIGNATURA: Desarrollo en plataformas

NOMBRE COMPLETO DEL DOCENTE CONTENIDISTA: Damián Aníbal Nicolalde Rodríguez

DESARROLLO

1. INTRODUCCIÓN DE LA CLASE

La Clase 9 marca un punto de inflexión en el desarrollo del curso, pues constituye la transición definitiva desde los fundamentos del frontend y la lógica cliente-servidor hacia la construcción profesional del back-end moderno. Hasta este momento, los estudiantes han comprendido cómo una SPA se comunica con un servidor mediante peticiones HTTP, y han interactuado con APIs básicas. Sin embargo, en esta clase se profundiza de manera sistemática en los elementos que estructuran la arquitectura interna de un servicio web real.

El propósito central de la clase es brindar al estudiante una comprensión sólida de cómo funcionan las APIs REST, cómo se organizan mediante rutas, controladores y modelos, cómo se implementan las operaciones CRUD, y cómo se asegura su correcto funcionamiento mediante técnicas de middleware y manejo estructurado de errores. Estas herramientas representan la base sobre la cual se construyen plataformas digitales contemporáneas —desde comercio electrónico y redes sociales hasta sistemas empresariales de misión crítica— y permiten garantizar escalabilidad, mantenimiento y consistencia en los datos.

Clase 9: Back-end moderno y APIs básicas.

RDA 2: Programar aplicaciones web basadas en el estilo arquitectónico cliente-servidor hasta su publicación, garantizando la eficiencia y la seguridad en la comunicación entre el cliente y el servidor, complementando su desarrollo con IA.

Reto # 2

9.1. Introducción a Node.js y Express

El desarrollo de aplicaciones modernas —especialmente plataformas digitales, sistemas distribuidos, SPAs, aplicaciones móviles e integraciones de servicios— exige arquitecturas que permitan alto rendimiento, escalabilidad y la capacidad de manejar miles de solicitudes concurrentes. En ese contexto, Node.js y su framework más popular, Express, se han convertido en una de las herramientas fundamentales del ecosistema JavaScript para construir back-end eficientes, flexibles y fáciles de mantener.



Este apartado introduce en profundidad ambos elementos, su arquitectura interna, sus ventajas, sus desafíos y su papel dentro de un entorno cliente-servidor. Además, se contextualiza cómo se usará en esta asignatura para la construcción del proyecto CaféCat y para futuros módulos como bases de datos, integración y microservicios.

9.1.1. ¿Qué es Node.js y por qué es tan importante?

Node.js es un entorno de ejecución de JavaScript del lado del servidor, construido sobre el motor V8 de Google Chrome. En otras palabras, permite ejecutar JavaScript fuera del navegador, usando funciones del sistema operativo como acceso a archivos, redes, procesos y timers.

Node.js cambia totalmente la forma tradicional de construir servidores porque utiliza un modelo denominado:

✓ Modelo de ejecución “single-threaded, event-driven, non-blocking I/O”

Este modelo se basa en:

- Un solo hilo para ejecutar JavaScript.
- I/O no bloqueante: operaciones de lectura/escritura en disco y red se delegan a hilos internos del sistema, mientras el hilo principal mantiene la capacidad de procesar más solicitudes.
- Event loop: mecanismo interno que procesa eventos, callbacks y promesas.

El resultado práctico es que un servidor Node.js puede atender miles de conexiones simultáneas sin necesidad de usar múltiples hilos para cada cliente. Esto lo hace ideal para APIs REST, microservicios, integraciones, aplicaciones en tiempo real (chat, streaming) y operaciones intensivas de red.

Ventajas principales de Node.js

1. **Un solo lenguaje para todo el stack:** front-end y back-end comparten JavaScript.
2. **Excelente rendimiento en aplicaciones I/O intensivas:** APIs que consultan datos, llaman servicios, acceden a archivos, etc.
3. **Gran ecosistema (npm):** miles de paquetes disponibles.
4. **Arquitectura ligera:** perfecta para microservicios y despliegue en contenedores.



5. **Facilita la asincronía:** gracias a promesas, async/await y el event loop.

Desafíos comunes

- No es ideal para tareas CPU-heavy (procesamiento masivo, encriptación pesada).
- Requiere comprender la asincronía para evitar errores lógicos.
- npm contiene muchos paquetes no mantenidos; se deben seleccionar con criterio.

Tabla 1. *Comparación conceptual entre servidores tradicionales y Node.js*

Fuente: elaboración propia.

Característica	Servidor tradicional (multi-hilo)	Node.js (event-driven)
Hilos de ejecución	Un hilo por cliente	Un hilo procesa miles de clientes
Rendimiento	Bueno para operaciones CPU	Excelente para I/O intensivo
Escalabilidad	Costosa, requiere más hilos	Fácil con cluster o contenedores
Lenguaje	C#, Java, PHP, Ruby	JavaScript
Consumo de memoria	Alto	Bajo
Curva de aprendizaje	Media	Media con asincronía

9.1.2. Express: el framework más utilizado en Node.js

Express es un framework minimalista que provee herramientas para:

- Crear servidores HTTP de forma rápida.
- Manejar rutas y métodos HTTP.
- Implementar middleware.
- Procesar cuerpos JSON.
- Manejar errores.
- Organizar un back-end modular (rutas, controladores, modelos).

Su diseño es intencionalmente simple: Express no impone una arquitectura estricta, lo cual lo vuelve flexible para proyectos pequeños y grandes.

Ejemplo mínimo de Express

```
import express from "express";
const app = express();

app.get("/saludo", (req, res) => {
  res.json({ mensaje: "Hola desde Express" });
});

app.listen(3000);
```

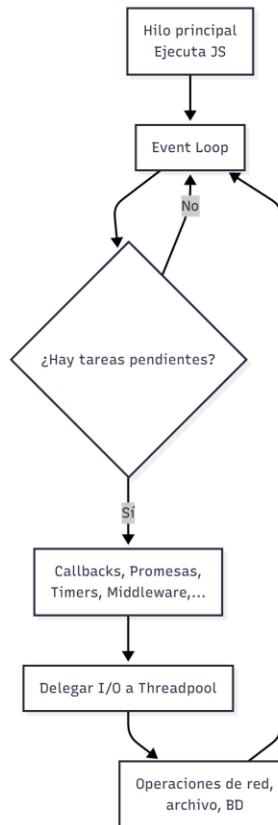
Solo tres líneas crean un servidor funcional. A partir de aquí, se organizan rutas, controladores y modelos de forma profesional.

9.1.3. Arquitectura interna: event loop, threadpool y callbacks

El funcionamiento interno de Node.js se representa conceptualmente en la siguiente figura.

Figura 1. Event loop y arquitectura de ejecución de Node.js

Fuente: elaboración propia.



Explicación:



- El event loop es el corazón de Node.js.
- Cuando Express recibe una petición HTTP:
 - El event loop ejecuta el middleware y el controlador.
 - Si el controlador debe realizar una operación pesada de I/O, la delega al threadpool.
 - Mientras tanto, sigue atendiendo otras solicitudes.

9.1.4. Node.js dentro del ecosistema del proyecto CaféCat

En esta asignatura, Node.js es el *motor* que soportará el back-end de la plataforma CaféCat.

Express se encargará de:

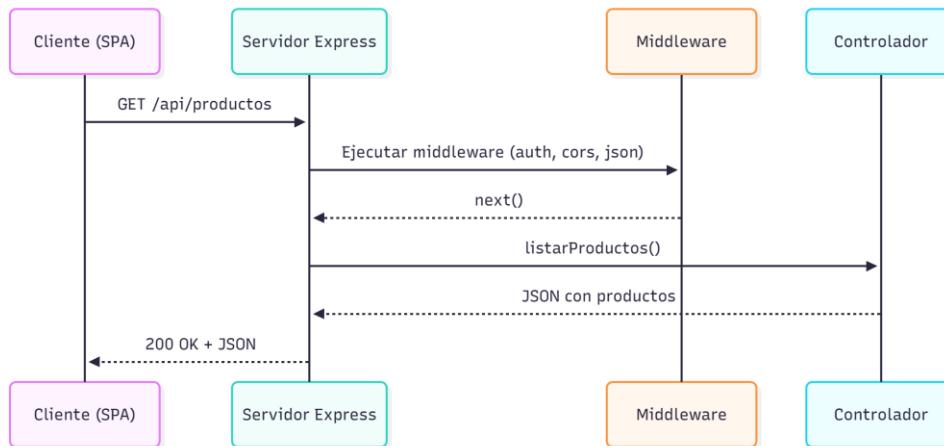
- Exponer productos (GET /api/productos).
- Registrar pedidos (POST /api/pedidos).
- Administrar usuarios y roles (POST /api/usuarios).
- Manejar autenticación básica por cabeceras.
- Proveer endpoints consumidos por la SPA (Single Page Application).

9.1.5. Flujo de una petición en Express

El flujo completo de una petición HTTP en Express puede representarse con un diagrama de secuencia.



Figura 2. Flujo de petición en Express con middleware y controlador
 Fuente: elaboración propia.



Este flujo explica:

1. El cliente (navegador o SPA) envía una petición.
2. Express ejecuta middleware (CORS, logging, autenticación).
3. La petición llega al controlador.
4. El controlador consulta la fuente de datos.
5. Se genera una respuesta en formato JSON.

9.1.6. Casos de uso de Node.js + Express en plataformas modernas

Node.js con Express es ampliamente utilizado en:

1. APIs REST y GraphQL

- E-commerce, redes sociales, banca digital.

2. Microservicios

- Arquitecturas distribuidas basadas en contenedores (Docker, Kubernetes).

3. Integración entre sistemas

- Adaptadores, gateways, sistemas de colas.

4. Aplicaciones en tiempo real

- Chats, notificaciones, videojuegos.

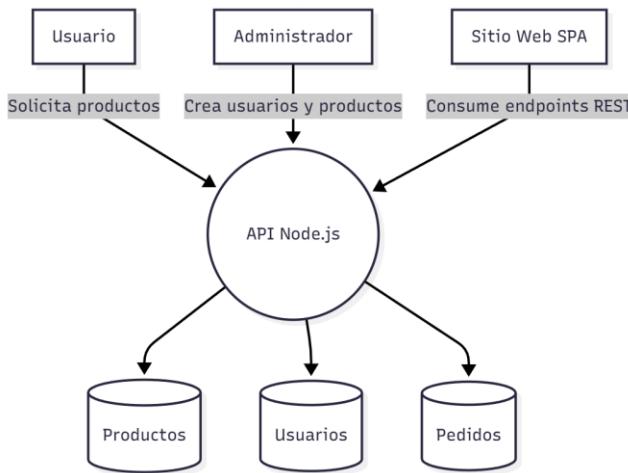


5. Backend para SPAs

- React, Angular, Vue.

Figura 3. Diagrama de casos de uso de una API Express

Fuente: elaboración propia.



9.1.7. Manejo de JSON y transporte de datos

Uno de los pilares de Express es la capacidad de procesar JSON, el formato estándar para comunicar clientes web con APIs.

Express incluye un middleware integrado:

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

Este middleware convierte automáticamente el cuerpo de la petición (`req.body`) en un objeto JavaScript.

9.1.8. Rutas, métodos y controladores

Express facilita la creación de endpoints, por ejemplo:

```
app.get("/api/productos", controladorListarProductos)
```

```
app.post("/api/productos", controladorCrearProducto)
```

Cada método HTTP tiene un propósito:

- **GET**: recuperar datos.
- **POST**: crear recursos.



- **PUT:** actualizar completamente.
- **PATCH:** actualizar parcialmente.
- **DELETE:** eliminar un recurso.

9.1.9. Tabla: comparación de métodos HTTP y su uso

Tabla 2. *Métodos HTTP y su propósito en APIs REST*

Fuente: elaboración propia.

Método	Descripción	Ejemplo
GET	Obtener información	/api/productos
POST	Crear un recurso	/api/usuarios
PUT	Actualizar completamente	/api/productos/3
PATCH	Actualización parcial	/api/usuarios/5
DELETE	Eliminar	/api/pedidos/10

9.1.10. ¿Por qué Express es ideal para aprender y construir plataformas?

- **Es minimalista:** solo incluye lo esencial.
- **Es flexible:** cada proyecto puede organizarse según las necesidades.
- **Es ampliamente adoptado:** millones de proyectos reales.
- **Está alineado con microservicios:** módulos pequeños, independientes.

9.2. Estructura del proyecto: rutas, controladores y modelos

La correcta organización de un proyecto backend es un pilar fundamental en el desarrollo de plataformas modernas. No se trata únicamente de escribir código que funcione, sino de construir sistemas mantenibles, escalables, modulares y alineados a buenas prácticas arquitectónicas. En este contexto, Express no impone una estructura rígida, lo cual ofrece flexibilidad, pero también demanda disciplina. Por eso, en esta sección se estudia la arquitectura recomendada basada en rutas, controladores y modelos, un estándar ampliamente adoptado en APIs REST y microservicios modernos.



9.2.1. La importancia de una estructura modular

En un proyecto pequeño, todo podría escribirse en un solo archivo (server.js). Sin embargo, esto es inviable para plataformas reales. Separar el proyecto en capas permite:

1. Mantener el código organizado y legible.
2. Asegurar cohesión interna y bajo acoplamiento.
3. Facilitar pruebas unitarias y mantenimiento.
4. Permitir escalabilidad horizontal y vertical.
5. Delegar responsabilidades a diferentes miembros del equipo.

Las plataformas modernas (como Amazon, Netflix, Uber o Mercado Libre) adoptan arquitecturas modulares como:

- MVC (Modelo–Vista–Controlador)
- Capa de servicios
- Repositorios
- Modelos de datos
- Microservicios independientes

Dentro de esta asignatura, adoptaremos una estructura profesional simplificada: rutas → controladores → modelos, suficiente para APIs REST y escalable a proyectos reales.

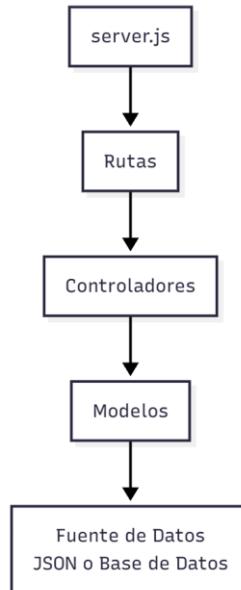
9.2.2. Estructura base recomendada

A continuación, se presenta una estructura típica para un proyecto Express moderno.



Figura 4. Estructura del proyecto Express recomendado

Fuente: *Elaboración propia*



La arquitectura en capas permite controlar el flujo de información y el orden lógico del proyecto:

1. **Rutas:** definen los endpoints y métodos HTTP.
2. **Controladores:** contienen la lógica de negocio.
3. **Modelos:** manejan datos y la interacción con la fuente persistente.
4. **Middleware:** se ejecuta antes de llegar a controladores (CORS, logs, auth, etc.)

9.2.3. La carpeta “routes/”: la interfaz externa de la API

La responsabilidad de una ruta es mapear un endpoint HTTP a su controlador correspondiente.

Ejemplo típico:

```

router.get("/", listarProductos);

router.post("/", crearProducto);

router.get("/:id", detalleProducto);
  
```

Una ruta bien diseñada debe:

- ser simple, sin lógica



- declarar únicamente la estructura del endpoint
- permanecer desacoplada de la lógica de negocio

Tabla 3. Estructura conceptual de una ruta REST

Fuente: *Elaboración propia*

Endpoint	Método	Controlador	Descripción
/api/productos	GET	listarProductos	Devuelve todos los productos
/api/productos	POST	crearProducto	Crea un nuevo producto
/api/productos/:id	GET	detalleProducto	Obtiene un producto
/api/productos/:id	PUT	actualizarProducto	Actualiza
/api/productos/:id	DELETE	eliminarProducto	Elimina

La función de la ruta es entonces de “puente” entre el cliente y el controlador.

9.2.4. La carpeta “controllers/”: la lógica de negocio

Los controladores contienen la lógica que da sentido al endpoint, por ejemplo:

- validar entradas
- llamar al modelo
- aplicar reglas de negocio
- construir la respuesta final

Una ruta nunca debe contener lógica, y un controlador nunca debe contener acceso directo a almacenamiento.

Ejemplo conceptual:

```
export function listarProductos(req, res) {
  const data = ProductosModel.getAll();
  res.status(200).json({ status: "success", data });
}
```

Esto permite probar el controlador sin involucrar Express y facilita que la lógica sobreviva incluso si en el futuro cambiamos:



- Express por Fastify
- JSON por SQL
- un monolito por microservicios

9.2.5. La carpeta “models”: capa de acceso a datos

En esta clase (Semana 9), los modelos trabajarán con:

- archivos JSON
- o estructuras en memoria

En la Semana 11, los modelos serán reemplazados por:

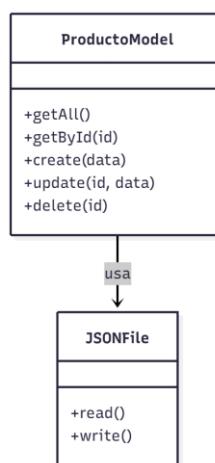
- Sequelize / Prisma
- MySQL / SQL Server / MongoDB

La función del modelo es:

- leer datos
- escribir datos
- ejecutar consultas
- aplicar validaciones de consistencia

Figura 5. Diagrama conceptual de Modelo

Fuente: Elaboración propia



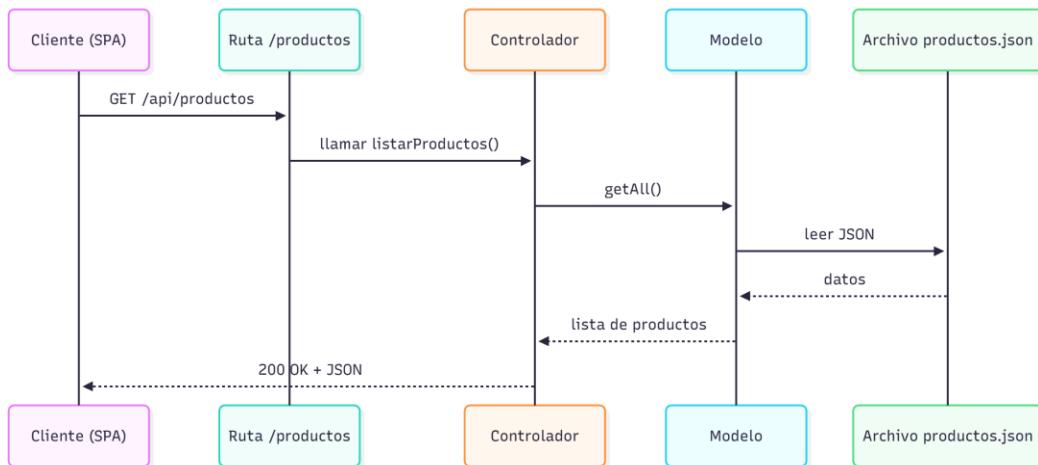


9.2.6. Ejemplo completo: flujo de una petición

Analicemos el flujo de GET /api/productos.

Figura 6. Diagrama de secuencia del flujo de “Listar Productos”

Fuente: Elaboración propia



9.2.7. Organización de carpetas recomendada

Ejemplo de organización para CaféCat:

```

server/
├ server.js
├ routes/
│ ├ productos.js
│ ├ pedidos.js
│ ├ usuarios.js
│ └ auth.js
└ controllers/
  ├ productos.controller.js
  ├ pedidos.controller.js
  ├ usuarios.controller.js
  └ auth.controller.js
└ models/
  ├ productos.model.js
  ├ usuarios.model.js
  └ pedidos.model.js
└ data/
  ├ productos.json
  ├ usuarios.json
  └ pedidos.json
  
```

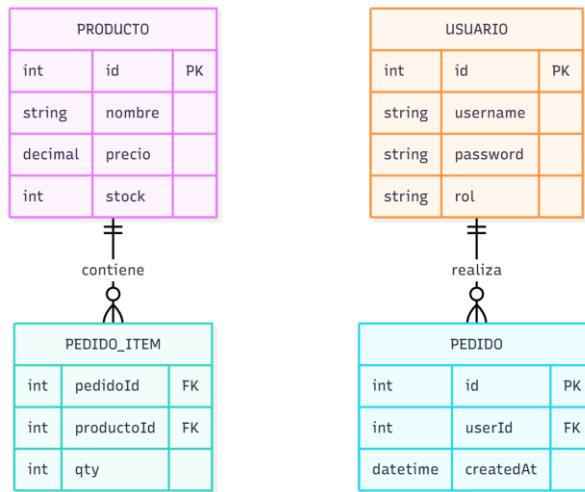
9.2.8. Modelo conceptual de datos (ER simplificado)

Aunque la base de datos se implementará en la Semana 11, es importante conocer desde ya la estructura semántica.



Figura 7. Diagrama ER básico del proyecto CaféCat

Fuente: *Elaboración propia*



Este ER es suficiente para una primera versión del CRUD (Semana 9) y para evolucionarlo a SQL con un ORM en la Semana 11.

9.2.9. Patrón de diseño aplicado: “Separation of Concerns”

La separación correcta entre rutas, controladores y modelos es un ejemplo del principio SoC (Separation of Concerns).

Beneficios:

- Facilita pruebas unitarias
- Facilita refactorización
- Permite escalar módulos de forma independiente
- Permite reemplazar capas sin romper el sistema
- Reduce duplicación de código

Tabla 4. Responsabilidades de cada capa

Fuente: *Elaboración propia*

Capa	Responsabilidad	No debe hacer
Ruta	Declarar endpoints	Lógica de negocio, acceso a datos



Controlador	Aplicar reglas	Conocer infraestructura o almacenamiento
Modelo	Leer / escribir datos	Responder peticiones Express

9.2.10. Ejemplo completo integrado

Ruta

```
router.get("/", listarProductos);
```

Controlador

```
export function listarProductos(req, res) {
  const data = ProductoModel.getAll();
  res.status(200).json({ status: "success", data });
}
```

Modelo

```
export const ProductoModel = {
  getAll() {
    const raw = fs.readFileSync("./data/productos.json", "utf8");
    return JSON.parse(raw);
  }
};
```

9.2.11. Conclusión

La estructura del proyecto define la calidad futura del software. En esta asignatura, usar rutas, controladores y modelos permitirá construir una API modular, profesional y escalable. Aunque en esta clase la persistencia será JSON o memoria, la arquitectura ya queda preparada para incorporar:

- SQL
- NoSQL
- ORM
- Microservicios
- Integración de terceros
- Seguridad y despliegues

Es decir, construimos desde ahora las bases de una plataforma real.



9.3. CRUD básico (Create, Read, Update, Delete)

El CRUD (Create, Read, Update y Delete) representa la base funcional de cualquier sistema de información. Sin operaciones CRUD no puede existir interacción significativa con datos persistentes, por lo que constituye el corazón de las plataformas digitales modernas. En este módulo se explica a profundidad qué es un CRUD, cómo se estructura en una API REST, cómo se implementa en Node.js con Express y cómo se relaciona con los modelos de datos y la arquitectura vista en la clase 9.2.

9.3.1. Qué es un CRUD y por qué es fundamental en plataformas

En un sistema digital, los datos no son estáticos: deben crearse, consultarse, actualizarse y eliminarse. Por ello se diseñó una convención universal:

- Create → Crear
- Read → Leer / Consultar
- Update → Actualizar
- Delete → Eliminar

Las operaciones CRUD son fundamentales en:

- E-commerce
- Sistemas de gestión
- Redes sociales
- Plataformas de contenido
- Apps móviles
- Microservicios
- APIs internas y externas

Prácticamente todo software que maneje datos implementa CRUD, por lo que esta sección constituye el bloque fundamental sobre el que se construirá el proyecto CaféCat.

9.3.2. Relación entre CRUD y métodos HTTP

En una API REST, cada operación CRUD se define de forma estándar usando métodos HTTP. Esto permite que cualquier cliente (SPA, móvil, microservicio) interactúe de manera consistente.



Tabla 5. Mapeo CRUD ↔ Métodos HTTP

Fuente: *Elaboración propia*

Operación	Acción	Método HTTP	Ejemplo
Create	Crear recurso	POST	POST /api/productos
Read	Leer recursos	GET	GET /api/productos
Read	Obtener uno	GET	GET /api/productos/5
Update	Actualizar todo	PUT	PUT /api/productos/5
Update	Actualizar parcial	PATCH	PATCH /api/productos/5
Delete	Eliminar	DELETE	DELETE /api/productos/5

Este estándar permite construir APIs interoperables y coherentes.

9.3.3. Flujo arquitectónico de un CRUD en Express

El flujo de cualquier operación CRUD sigue esta estructura:

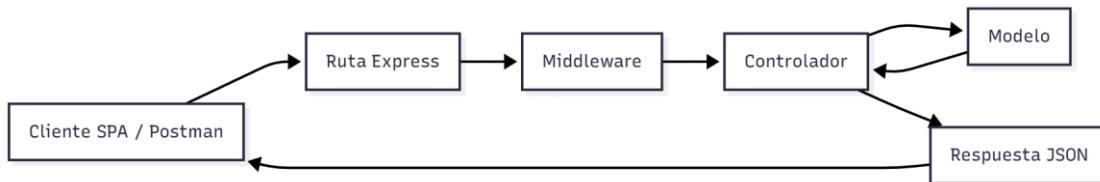
1. **Cliente** envía solicitud HTTP.
2. **Ruta** identifica el endpoint y el método.
3. **Middleware** valida autentificación / roles / formato.
4. **Controlador** ejecuta lógica del negocio.
5. **Modelo** accede a los datos (JSON en clase 9, BD en semana 11).
6. **Controlador** construye la respuesta.
7. **Servidor Express** responde al cliente.

Esto se representa en el siguiente diagrama.



Figura 8. Diagrama general de flujo CRUD

Fuente: *Elaboración propia*



9.3.4. Diagrama ER aplicado al CRUD

Para comprender CRUD, es importante relacionarlo con las entidades de datos. Usaremos como ejemplo la entidad Producto, parte del proyecto CaféCat.

Figura 9. ER básico para Producto

Fuente: *elaboración propia*

PRODUCTO		
int	id	PK
string	nombre	
decimal	precio	
int	stock	
string	descripcion	

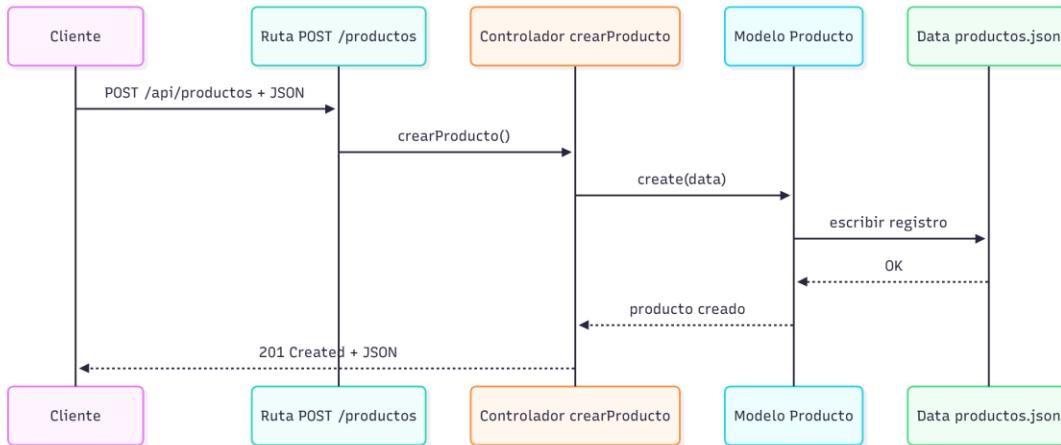
Cada operación CRUD modificará este conjunto de atributos.

9.3.5. Caso práctico: CRUD de productos

A continuación, se describen las 4 operaciones CRUD para la entidad Producto.

A) CREATE (POST /api/productos)

Esta operación crea un nuevo producto.



Validaciones típicas

- nombre no vacío
- precio > 0
- categoría válida
- stock ≥ 0

Ejemplo de respuesta

```
{
  "status": "created",
  "data": {
    "id": 5,
    "nombre": "Café Espresso",
    "precio": 9.90
  }
}
```

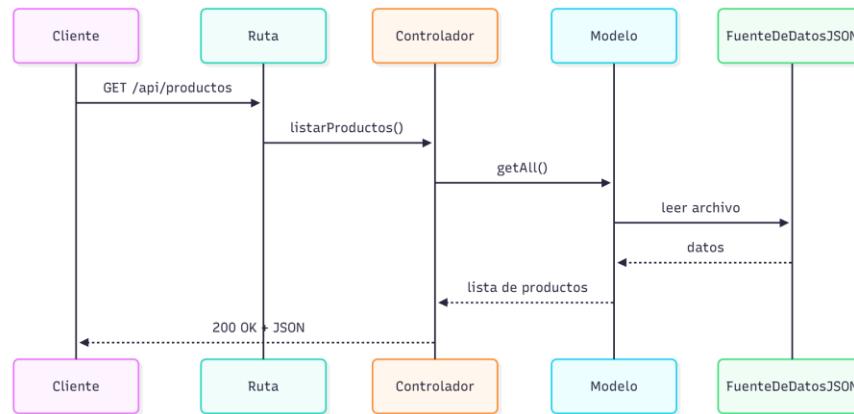
B) READ (GET /api/productos y GET /api/productos/:id)

El “Read” de CRUD tiene dos variantes:

1. Leer todos
2. Leer uno por ID



Figura 10. Flujo del “Read All”



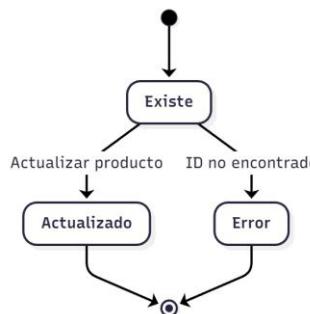
Ejemplo de respuesta:

```
{
  "status": "success",
  "data": [
    { "id": 1, "nombre": "Café Premium", "precio": 22.9 },
    { "id": 2, "nombre": "Té Matcha", "precio": 15.4 }
  ]
}
```

C) UPDATE (PUT /api/productos/:id)

El update reemplaza totalmente el recurso.

Figura 11. Diagrama de estados para “Update”



Reglas comunes:

- validar que el ID exista
- validar que los campos tengan formato correcto
- sobrescribir el registro completo



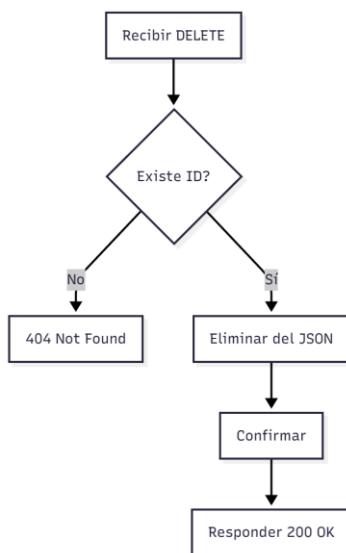
Ejemplo de respuesta:

```
{
  "status": "updated",
  "data": {
    "id": 3,
    "nombre": "Café Latte",
    "precio": 12.5
  }
}
```

D) DELETE (DELETE /api/productos/:id)

Elimina el recurso permanentemente.

Figura 12. Diagrama de actividad para “Delete”





9.3.6. Tabla resumen del CRUD

Tabla 6. Resumen de operaciones CRUD

Fuente: Elaboración propia

Operación	Endpoint	Entrada	Salida	Estado
Create	POST /productos	JSON	Recurso creado	201
Read All	GET /productos	—	JSON lista	200
Read One	GET /productos/:id	ID	JSON	200 / 404
Update	PUT /productos/:id	JSON	Recurso actualizado	200
Delete	DELETE /productos/:id	ID	Confirmación	200 / 404

9.3.7. Patrones de validación en CRUD

Es fundamental validar:

- tipos de datos
- campos obligatorios
- duplicados
- rangos válidos
- existencia del ID
- consistencia semántica

Esto evita errores, corrupción de datos y ataques.

9.3.8. Buenas prácticas al diseñar CRUD

1. Mantener rutas limpias:
`/api/productos` y no `/api/obtenerProductos.php`
2. Usar códigos HTTP correctos:
 201 (created), 404 (not found), 400 (bad request)



3. Validar todo input del cliente
4. No mezclar lógica entre capas
5. Respuestas siempre en JSON
6. No exponer datos sensibles
Ej.: contraseñas deben omitirse o encriptarse
7. Implementar paginación para listas grandes
8. Evitar sobrecargar la API
(limitar tamaño de payloads)
9. Hacer logging del CRUD
para auditoría

9.3.9. Ejemplo completo de CRUD para Producto (archivo JSON)

Ruta

```
router.put("/:id", actualizarProducto);
router.delete("/:id", eliminarProducto);
```

Controlador

```
export function actualizarProducto(req, res) {
  const id = Number(req.params.id);
  const data = req.body;
  const updated = ProductoModel.update(id, data);

  if (!updated) {
    return res.status(404).json({ status: "fail", message: "Producto no encontrado" });
  }

  res.status(200).json({ status: "updated", data: updated });
}
```

Modelo

```
update(id, data) {
  const productos = load();
  const idx = productos.findIndex(p => p.id === id);
  if (idx === -1) return null;

  const nuevo = { id, ...data };
  productos[idx] = nuevo;
  save(productos);
  return nuevo;
}
```



9.3.10. Conclusión

El CRUD constituye el eje funcional de cualquier API REST y plataforma digital. Entender su flujo, semántica HTTP, estructura modular y buenas prácticas permitirá al estudiante implementar sistemas robustos y totalmente alineados a estándares reales de la industria.

En esta clase, los modelos utilizan JSON como fuente de datos; sin embargo, la arquitectura está diseñada para que en la Semana 11 podamos reemplazarla de forma transparente por:

- MySQL
- SQL Server
- MongoDB
- ORM como Sequelize o Prisma

9.4. Gestión de errores y middleware

La gestión de errores y el uso correcto de middleware representan dos pilares fundamentales de un backend profesional. Aunque muchas veces se consideran aspectos “secundarios”, son precisamente los mecanismos que diferencian un API casero de una API robusta, mantenible y lista para producción. En esta sección analizaremos cómo funciona el manejo de errores en Node.js y Express, cómo se construyen middlewares personalizados, cuál es su rol dentro del pipeline de ejecución y cómo deben implementarse siguiendo buenas prácticas y estándares industriales.

9.4.1. Concepto de error en una API REST

Un error en el backend no necesariamente significa un fallo del servidor. En realidad, dentro de un API REST existen varios tipos de errores:



Tabla 7. Tipos de errores comunes en un backend

Fuente: elaboración propia

Tipo de error	Descripción	Ejemplo
Error del cliente (400)	La solicitud está mal construida	Parámetros faltantes, JSON inválido
No autorizado (401)	El cliente no está autenticado	Falta token o sesión
Prohibido (403)	El cliente está autenticado, pero sin permisos	Cliente intentando acceder a zona admin
No encontrado (404)	El recurso no existe	Producto inexistente
Error de servidor (500)	Fallo interno	Excepción no controlada, crash
Conflicto (409)	No se puede completar por regla del sistema	Usuario duplicado

En un proyecto profesional, controlar estos errores garantiza:

- seguridad
- predictibilidad
- facilidad de depuración
- buena experiencia del cliente (SPA, móvil, microservicios)
- consistencia semántica

9.4.2. ¿Qué es un middleware en Express?

Un middleware es una función que se ejecuta entre la solicitud del cliente y la respuesta del servidor. Normalmente recibe tres parámetros:



```
function (req, res, next) { }
```

Este patrón permite:

- interceptar solicitudes
- validar, transformar o rechazar datos
- registrar logs
- aplicar seguridad
- manejar errores globales
- extender comportamientos del servidor

9.4.3. Pipeline de middlewares

El flujo de ejecución de middlewares se visualiza como una cadena (pipeline) que Express recorre de manera secuencial.

Figura 13. Diagrama de flujo del pipeline de middlewares



Cada middleware puede:

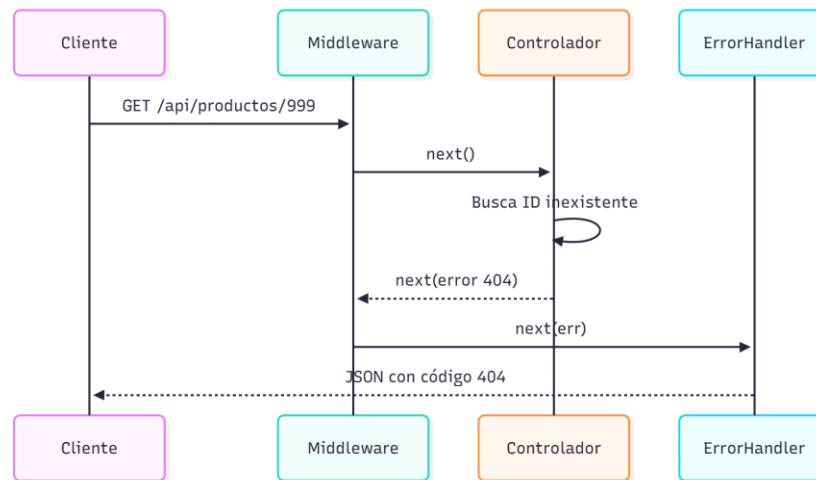
- **llamar next()** → continúa la ejecución
- **retornar una respuesta** → detiene la cadena
- **llamar next(err)** → envía el flujo al manejador global de errores

9.4.4. Ciclo de secuencia: Middleware + Controlador + Error

A continuación, se muestra el flujo de cómo viaja un error desde el controlador hasta el manejador global.



Figura 14. Diagrama de secuencia del manejo de errores



9.4.5. Diseño de un middleware de manejo de errores

El middleware global de errores es el último en ejecutarse y debe cumplir tres reglas:

1. Recibir cuatro parámetros: (err, req, res, next)
2. Nunca lanzar excepciones sin capturarlas
3. Retornar un JSON estándar para el cliente

Ejemplo:

```
// server/middleware/errorHandler.js
export function errorHandler(err, req, res, next) {
  console.error("🔥 Error capturado:", err);

  const status = err.status || 500;
  const msg = err.message || "Error interno del servidor";

  res.status(status).json({
    status: "error",
    code: status,
    message: msg
  });
}
```

Este middleware permitirá centralizar toda la gestión de errores del backend.

9.4.6. Generación de errores desde los controladores

Los controladores no deben construir respuestas 500 manualmente.

En cambio, deben crear errores estructurados y pasarlo con next(err).



Ejemplo correcto:

```
export function detalleProducto(req, res, next) {
  try {
    const id = Number(req.params.id);
    const productos = load();

    const prod = productos.find(p => p.id === id);
    if (!prod) {
      const err = new Error("Producto no encontrado");
      err.status = 404;
      return next(err);
    }

    res.json({ status: "success", data: prod });
  } catch (err) {
    next(err);
  }
}
```

Esto permite que:

- el controlador se mantenga limpio
- el errorHandler gestione la respuesta
- el frontend reciba respuestas uniformes

9.4.7. Tipos de middlewares y su propósito

En el proyecto CaféCat se usan tres middlewares principales:

1. Middleware de CORS

Controla qué cliente puede acceder a la API.

```
res.header("Access-Control-Allow-Origin", origin);
```

2. Middleware de correlación

Genera un ID único por solicitud útil para debugging.

```
req.correlationId = "cid-abc123";
```

3. Middleware de contexto de usuario

Inserta req.user basado en encabezados enviados por el frontend.

```
req.user = { id: Number(id), rol };
```



Tabla 8. Clasificación de middlewares según su rol

Fuente: Elaboración propia

Tipo	Función	Ejemplo
De aplicación	Se ejecuta en toda la API	CORS, logger, JSON parser
De seguridad	Restringen acceso	authContext
De validación	Validan datos del cliente	validateCreateProduct
De negocio	Preparan datos para lógica	parsePagination
De error	Capturan fallos globales	errorHandler

9.4.8. Middleware de validación con esquema

Aun cuando la base de datos se implementará en semana 11, es útil introducir validaciones.

Ejemplo de validador:

```
export function validarProducto(req, res, next) {
  const { nombre, precio, stock } = req.body;

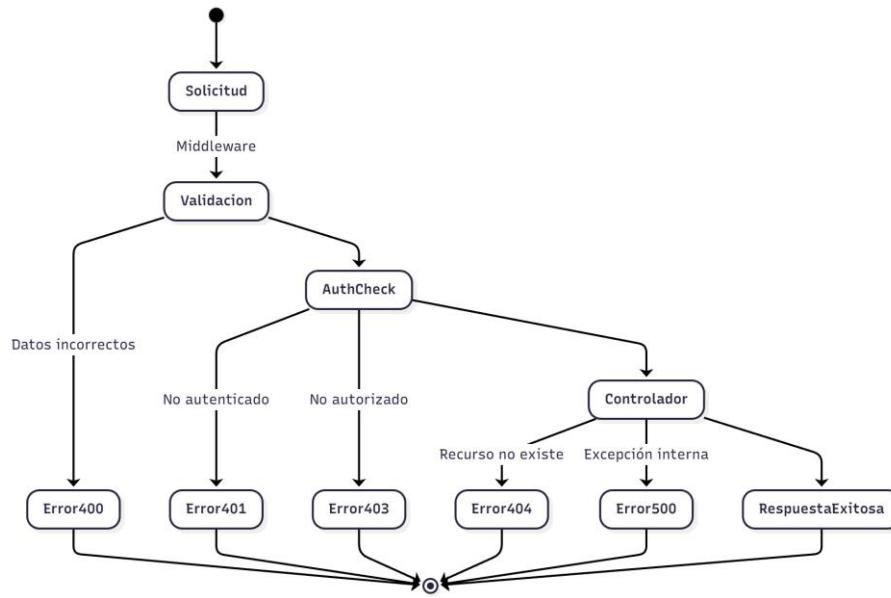
  if (!nombre || precio == null || stock == null) {
    const err = new Error("Datos incompletos");
    err.status = 400;
    return next(err);
  }

  next();
}
Uso:
router.post("/", validarProducto, crearProducto);
```



9.4.9. Diagrama de estados del error HTTP

Figura 15. Diagrama de estados de errores HTTP



9.4.10. Logging y trazabilidad con correlation-id

La trazabilidad es esencial para depurar APIs complejas.

Ejemplo de log:

[cid-hsg3f78] GET /api/productos

[cid-hsg3f78] Respuesta 200 - 3 productos

Esto permite:

- rastrear solicitudes específicas
- identificar fallos en microservicios
- registrar accesos de usuarios



9.4.11. Tabla de buenas prácticas en manejo de errores

Tabla 9. Buenas prácticas sugeridas (APA)

Fuente: elaboración propia

Práctica	Descripción
No devolver stacktraces al cliente	Evita filtración de información sensible
Estandarizar formato JSON	status, message, code, details
Registrar errores en consola o archivo	Facilita debugging
No detener el servidor por errores	Usar try/catch + next(err)
Manejar errores asíncronos correctamente	Siempre capturar en async/await
Responder códigos HTTP correctos	400, 401, 403, 404, 500

9.4.12. Conclusiones

El manejo adecuado de errores y el uso correcto de middlewares constituyen una competencia esencial para construir plataformas digitales profesionales. Permiten garantizar seguridad, robustez, mantenibilidad y trazabilidad del sistema, y evitan que errores no controlados generen fallos masivos o inconsistencias de datos.

En esta clase, el estudiante no solo comprende la teoría, sino que reconoce cómo estos mecanismos ya están implementados en la arquitectura del proyecto CaféCat (CORS, correlation-id, authContext y validación de solicitudes).

9.5. Buenas prácticas en la implementación de APIs REST

La construcción de una API REST moderna no se limita únicamente a implementar rutas y CRUD. Las organizaciones tecnológicas que operan a gran escala, como Google, Meta, Amazon o Netflix, aplican un conjunto de normas, recomendaciones y patrones que garantizan que sus APIs sean seguras, escalables, mantenibles, predecibles y fáciles de consumir. En esta sección se presentan las buenas prácticas fundamentales que todo desarrollador debe aplicar al diseñar una API REST profesional utilizando Node.js y Express.



9.5.1. Diseño limpio de endpoints

El diseño de endpoints es esencial para garantizar que la API sea legible e intuitiva.

Regla 1 — Usar sustantivos, no verbos

Incorrecto

/crearProducto

/eliminarProductoPorID

Correcto

GET /productos

POST /productos

DELETE /productos/:id

Regla 2 — Consistencia en plural y minúsculas

- Siempre usar **minúsculas**
- Siempre usar **plural** para colecciones

✓ /usuarios, /productos, /pedidos

Regla 3 — Estructura jerárquica

Ejemplos adecuados:

- GET /usuarios/5/pedidos
- GET /productos/10/reseñas
- DELETE /categorias/3/productos/20

9.5.2. Uso correcto de los métodos HTTP

Los verbos HTTP existen para reflejar acciones semánticamente correctas.



Tabla 10. Mapping de métodos HTTP recomendados

Fuente: Elaboración propia

Método	Uso adecuado	Ejemplo
GET	Leer recursos	/productos
POST	Crear recurso	/usuarios
PUT	Reemplazar recurso	/productos/5
PATCH	Actualización parcial	/usuarios/3
DELETE	Eliminar recurso	/productos/7

El uso erróneo causa APIs confusas y muy difíciles de mantener.

9.5.3. Estructura estándar de respuestas

Las respuestas de la API deben ser consistentes.

Cada respuesta debe seguir un formato unificado como:

```
{
  "status": "success",
  "data": {
    "id": 1,
    "nombre": "Café Orgánico"
  }
}
```

Para errores:

```
{
  "status": "error",
  "code": 404,
  "message": "Producto no encontrado"
}
```

Esto permite que:

- cualquier cliente procese respuestas sin ambigüedad
- los desarrolladores conozcan el formato previamente
- se logre compatibilidad con SPA, mobile y microservicios



9.5.4. Versionamiento de la API

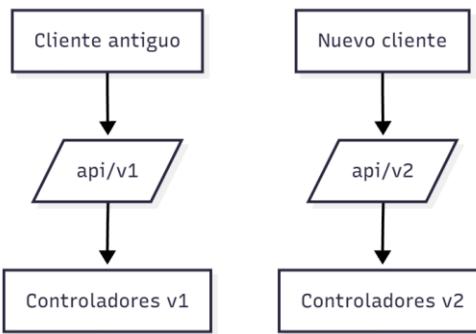
El versionamiento evita romper el código de clientes externos cuando se realizan cambios mayores.

Ejemplo:

- /api/v1/productos
- /api/v2/productos

Figura 16. Diagrama de flujo del versionamiento API

Fuente: Elaboración propia



Ventajas:

- permite actualizar la API sin interrumpir sistemas existentes
- evita errores por cambios estructurales
- facilita migraciones graduales

9.5.5. Buenas prácticas de seguridad

La seguridad es crítica en cualquier plataforma digital.

1. No exponer datos sensibles

Nunca enviar:

- contraseñas
- tokens internos
- secretos
- rutas internas del servidor



2. Validar todos los datos recibidos del cliente

Ejemplo:

```
if (!precio || precio <= 0) {
    throw new ValidationError("Precio inválido");
}
```

3. Uso obligatorio de HTTPS (en producción)

HTTPS garantiza:

- confidencialidad
- autenticidad
- integridad

4. Sanitización para evitar ataques (XSS, Injection, CSRF)

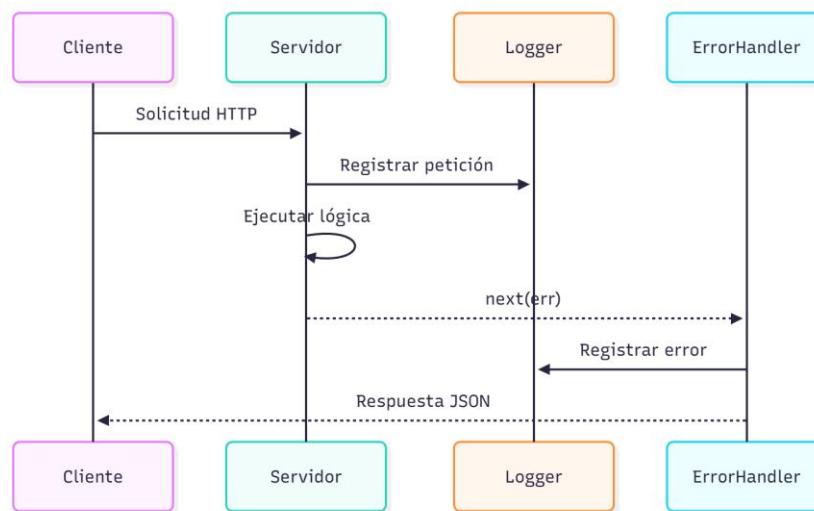
Esto será profundizado en semanas posteriores.

9.5.6. Manejo adecuado de errores y logs

Debe existir un middleware global de errores y un logger.

Ambos fueron explicados en 9.4, pero aquí se resumen como buenas prácticas.

Figura 17. Diagrama de secuencia: Logging + Error handler



El logging es el “rastro digital” que permite reconstruir lo que ocurrió dentro del servidor.



9.5.7. Paginación, filtros y ordenamiento

En listas grandes, una API debe permitir:

- paginar
- ordenar
- filtrar

Ejemplo estándar:

```
GET /productos?page=2&limit=20&categoria=bebidas&sort=precio
```

Beneficios:

- evita que la API responda con miles de registros
- mejora rendimiento
- reduce uso de memoria
- aumenta velocidad en el cliente

9.5.8. Documentación de API

Toda API profesional debe tener documentación clara.

Las herramientas más usadas son:

- Swagger / OpenAPI
- Postman Collections
- Redoc

Ejemplo en Swagger de un endpoint GET:

```
paths:  
/productos:  
  get:  
    summary: Lista todos los productos  
    responses:  
      200:  
        description: Lista completa
```

La documentación:

- estandariza
- facilita onboarding de nuevos desarrolladores



- evita errores por mal uso de la API

9.5.9. Uso de controladores y modelos separados

No se debe escribir lógica compleja directamente en la ruta.

Incorrecto:

```
router.post("/productos", (req, res) => {
  const prod = JSON.parse(fs.readFileSync("productos.json"));
  // lógica mezclada, caótica
});
```

Correcto:

- **ruta** → recibe solicitud
- **middleware** → valida
- **controlador** → ejecuta reglas
- **modelo** → manipula datos

Figura 18. Arquitectura limpia de API



Este patrón sostiene toda la arquitectura profesional de APIs REST.

9.5.10. Control de concurrencia y atomicidad

En sistemas con muchos usuarios simultáneos, es fundamental evitar:

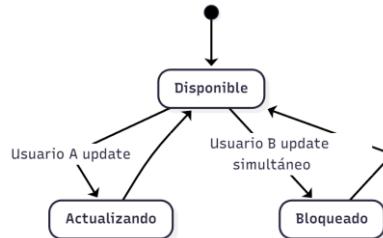
- estados inconsistentes
- corrupción de datos
- actualizaciones simultáneas fallidas

Esto se logra con:

- operaciones atómicas
- bloqueos de escritura
- validaciones antes de actualizar
- ORMs que manejan transacciones (Semana 11: Sequelize/Prisma)



Ejemplo conceptual:



9.5.11. HATEOAS (opcional pero recomendado)

HATEOAS significa Hypermedia As The Engine Of Application State

Consiste en incluir enlaces dentro de las respuestas.

Ejemplo:

```
{
  "id": 5,
  "nombre": "Café Premium",
  "links": {
    "self": "/api/productos/5",
    "delete": "/api/productos/5"
  }
}
```

Beneficios:

- autodescripción de la API
- navegación sin necesidad de documentación
- estandarización de flujos

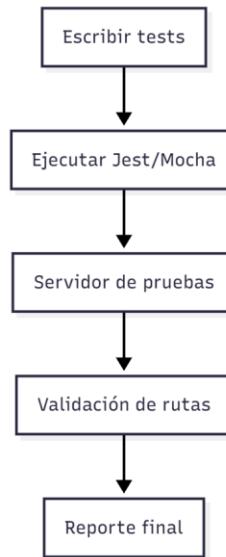
9.5.12. API testing y automatización

Buenas prácticas:

- probar rutas con Postman, SOAP UI
- automatizar tests con Jest o Mocha
- usar mocks para modelos
- validar respuestas y códigos HTTP



Figura 19. Flujo de tests automáticos



Esto garantiza confiabilidad y agilidad en el desarrollo.

9.5.13. Conclusión

Las buenas prácticas en el diseño e implementación de APIs REST no son simples recomendaciones; constituyen un conjunto de principios que sustentan la calidad, mantenibilidad y seguridad de cualquier plataforma digital moderna. Una API bien diseñada permite:

- escalar en usuarios
- integrar nuevos clientes fácilmente
- facilitar la depuración
- reducir errores de operación
- mantener un código sostenible a largo plazo

Referencias citadas en la Clase 9.

- Fielding, R. (2000). *Architectural Styles and the Design of Network-based Software Architectures*. University of California.
- Flanagan, D. (2020). *JavaScript: The Definitive Guide* (7th ed.). O'Reilly Media.
- Haverbeke, M. (2018). *Eloquent JavaScript* (3rd ed.). No Starch Press.
- Holmes, A. (2022). *Node.js Design Patterns* (3rd ed.). Packt Publishing.



- Microsoft. (2024). *Node.js Guidance*. <https://learn.microsoft.com/en-us/javascript/nodejs/>
- Mozilla Developer Network. (2024). *HTTP Overview*. <https://developer.mozilla.org>
- O'Reilly Media. (2021). *RESTful Web APIs*. O'Reilly.
- Tilkov, S., & Vinoski, S. (2010). Node.js: Using JavaScript to Build High-Performance Network Programs. *IEEE Internet Computing*, 14(6), 80–83.

Definición de términos

- **API (Application Programming Interface)**. Conjunto de reglas y especificaciones que permiten que dos sistemas interactúen mediante solicitudes y respuestas predeterminadas.
- **Back-end**. Parte del sistema encargada del procesamiento interno, la lógica de negocio, la gestión de datos y la comunicación con recursos del servidor.
- **CRUD (Create, Read, Update, Delete)**. Conjunto básico de operaciones utilizado para manipular datos dentro de sistemas de información.
- **Controlador (Controller)**. Componente de una API encargado de manejar la lógica de respuesta ante una solicitud del cliente.
- **Middleware**. Función intermedia en Express que intercepta solicitudes para ejecutar validaciones, seguridad, transformación de datos o manejo de errores.
- **Modelo (Model)**. Representación estructurada de los datos y reglas que definen cómo interactuar con ellos.
- **Node.js**. Entorno de ejecución de JavaScript que permite ejecutar código del lado del servidor utilizando una arquitectura basada en eventos.
- **Express**. Framework minimalista para Node.js que simplifica la creación de APIs REST mediante rutas, controladores y middleware.
- **HTTP (Hypertext Transfer Protocol)**. Protocolo estándar utilizado para la comunicación entre cliente y servidor en la web.
- **REST (Representational State Transfer)**. Arquitectura que utiliza HTTP y recursos identificados por URLs para construir servicios web escalables.
- **Recursos de profundización**

Recurso: Profundizacion1_clase9.docx

ACTIVIDAD DE EVALUACIÓN:

INSTRUCCIONES	<p><i>Responde las siguientes preguntas de opción múltiple relacionadas con los temas abordados en la Clase.</i></p> <p><i>Selecciona la alternativa correcta en cada caso. Al finalizar, revisa la retroalimentación para reforzar tu comprensión.</i></p>
----------------------	---



Titulo	Back-end moderno y APIs básicas					
Dificultad		Baja	X	Media	Alta	
RDAs Evaluados:		RDA 1				
	X	RDA 2				
		RDA 3				
Secciones Evaluadas		Clase: 9	Back-end moderno y APIs básicas			
PREGUNTA 1	¿Cuál de las siguientes afirmaciones describe mejor el propósito de un <i>middleware</i> en Express?					
CORRECTA	Es una función que intercepta solicitudes para procesarlas antes de llegar al controlador.					
Incorrecta	Es un módulo encargado de almacenar todos los datos del servidor.					
Incorrecta	Es una herramienta externa para depurar errores en Node.js.					
Incorrecta	Es un componente que sirve únicamente para manejar sesiones de usuario.					
Retroalimentación de las respuestas	El <i>middleware</i> es una pieza clave en la arquitectura de Express. Su función principal es interceptar y procesar las solicitudes HTTP antes de que lleguen al controlador , permitiendo validar datos, autenticar usuarios, registrar logs, manejar CORS o capturar errores. No almacena datos (opción a), no es una herramienta de depuración externa (c) ni se limita a gestionar sesiones (d).					
PREGUNTA 2	¿Cuál es el método HTTP estándar utilizado para actualizar completamente un recurso en una API REST?					
CORRECTA	PUT					



Incorrecta	GET
Incorrecta	POST
Incorrecta	PATCH
Retroalimentación de las respuestas	El método PUT se utiliza cuando se desea reemplazar por completo un recurso existente, enviando una nueva versión que sustituye totalmente a la anterior. POST se usa para crear (no para reemplazar), GET para consultar, y PATCH para actualizaciones parciales. En APIs REST profesionales, la semántica del método HTTP es fundamental para mantener consistencia y escalabilidad.
PREGUNTA 3	En un proyecto bien estructurado con Express, ¿qué responsabilidad principal tiene un controlador?
CORRECTA	Implementar la lógica de negocio asociada a una solicitud.
Incorrecta	Definir las URLs y los métodos HTTP.
Incorrecta	Almacenar y recuperar datos directamente del archivo JSON o la base de datos.
Incorrecta	Configurar la seguridad del servidor.
Retroalimentación de las respuestas	Los controladores representan la capa encargada de ejecutar la lógica de negocio: validar reglas, coordinar operaciones, procesar datos y decidir qué respuesta enviar al cliente. Las rutas definen las URLs (a), los modelos manejan la persistencia (c), y la seguridad es responsabilidad demiddlewares o servicios especializados (d). Por ello, la separación de responsabilidades es un pilar de la arquitectura Express.
PREGUNTA 4	¿Cuál de las siguientes características corresponde a una buena práctica en el diseño de APIs REST?
CORRECTA	Usar sustantivos en plural para representar recursos, como /productos.
Incorrecta	Usar verbos dentro de los endpoints, por ejemplo: /crearProducto.



Incorrecta	Retornar respuestas en múltiples formatos para mayor flexibilidad.
Incorrecta	Evitar usar códigos HTTP para simplificar la API.
Retroalimentación de las respuestas	En las APIs REST modernas, los endpoints deben representar recursos, no acciones. Por ello, se emplean sustantivos en plural, como /usuarios, /pedidos, /productos. Usar verbos (a) rompe la semántica REST, múltiples formatos complican la interoperabilidad (b) y omitir códigos HTTP (d) limita la capacidad del cliente para interpretar correctamente las respuestas del servidor. El uso de sustantivos pluralizados es un estándar universalmente aceptado.

Enlaces externos recomendados

Título del enlace relacionado: Node.js Official Documentation

Descripción del enlace relacionado:

Sitio oficial de Node.js que proporciona documentación completa del entorno de ejecución. Incluye guías detalladas sobre módulos nativos, arquitectura del event loop, manejo de callbacks y promesas, configuración del servidor, herramientas de depuración, seguridad y ejemplos prácticos para la creación de aplicaciones backend modernas.

Enlace: <https://nodejs.org/en/docs>

Título del enlace relacionado: Express.js Documentation

Descripción del enlace relacionado:

Documentación oficial del framework Express, ampliamente utilizado para construir APIs REST. Contiene guías sobre rutas, middlewares, solicitudes y respuestas HTTP, manejo de errores, patrones de estructura del proyecto y ejemplos prácticos para servidores web de producción.

Enlace: <https://expressjs.com/>

Título del enlace relacionado: MDN Web Docs – HTTP Overview

Descripción del enlace relacionado:

Recurso completo de Mozilla Developer Network que explica el funcionamiento del protocolo HTTP, los métodos estándar (GET, POST,



PUT, DELETE), códigos de estado, cabeceras y patrones de comunicación cliente-servidor esenciales para APIs REST.

Enlace: <https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview>

Título del enlace relacionado: RESTful API Tutorial – Restfulapi.net

Descripción del enlace relacionado: Sitio especializado en arquitectura REST con explicaciones claras sobre principios REST, diseño de recursos, mejores prácticas, versionamiento, seguridad, ejemplo de endpoints y patrones avanzados para APIs escalables.

Enlace: <https://restfulapi.net/>

Título del enlace relacionado: Postman Learning Center

Descripción del enlace relacionado: Centro de aprendizaje oficial de Postman que ofrece tutoriales, guías y laboratorios interactivos para probar APIs, documentarlas y crear colecciones de pruebas automatizadas. Recurso ideal para estudiantes en prácticas de desarrollo de backend.

Enlace: <https://learning.postman.com/>