Informe Técnico de la API

Área de Desarrollo Backend
Centro De Electricidad Y Automatización Industrial, SENA
3067594: Análisis y Desarrollo de Software
Cómite Evaluador
4 de julio del 2026.

Cali, Valle del Cauca

Tabla de contenido

Introducción

La API TAM fue desarrollada siguiendo una arquitectura backend basada en Node.js y TypeScript. Su propósito es centralizar la lógica del negocio, administrar la comunicación con la base de datos MySQL y exponer servicios REST para que aplicaciones web o móviles del proyecto TAM consuman la información de usuarios, productos, pedidos y demás procesos del sistema.

Este documento describe en detalle la arquitectura, los módulos implementados, los endpoints disponibles, el modelo de datos y las prácticas de seguridad y manejo de errores utilizadas en el desarrollo actual de la API, con el fin de servir como referencia técnica para el equipo evaluador y para futuros desarrolladores que continúen el proyecto.

Objetivos

Construir una API REST escalable, segura y mantenible que centralice la lógica de negocio del proyecto TAM.

Separar responsabilidades del sistema en capas de rutas, controladores, modelos y configuración.
Facilitar la autenticación de usuarios mediante JWT y el cifrado de contraseñas con bcrypt.
Exponer operaciones CRUD completas para las 19 entidades del negocio (usuarios, productos, pedidos, carritos, proveedores, etc.).

Permitir la carga de imágenes de productos y fotos de perfil de usuario mediante Multer.
Dejar una base de código ordenada que facilite pruebas, documentación (Postman) y mejoras futuras como el versionado o la validación centralizada.

Descripción general

La aplicación recibe solicitudes HTTP desde el frontend (web o móvil). El enrutador de Express dirige cada petición al controlador correspondiente según el método HTTP y la ruta; el controlador valida los datos recibidos, ejecuta la consulta SQL necesaria contra la base de datos MySQL a través del módulo db.ts, y devuelve la respuesta en formato JSON junto con el código de estado HTTP correspondiente.
Actualmente la API expone 21 módulos de recursos más el módulo de autenticación, todos montados directamente sobre la raíz del servidor Express (sin prefijo de versión).

Arquitectura

El proyecto sigue una arquitectura por capas típica de aplicaciones Express + TypeScript:

- **Rutas (src/api/routes):** Definen los endpoints y los asocian a funciones del controlador.
- **Controladores (src/controllers):** contienen la lógica de negocio, las validaciones manuales y las consultas a la base de datos.

- **Modelos (src/models):** interfaces de TypeScript que describen la forma de cada entidad (no son modelos de un ORM, ya que el proyecto no utiliza Sequelize, TypeORM ni Prisma).

- **Middlewares (src/middlewares):** configuración de Multer para la carga de imágenes de productos y fotos de perfil.

- **Configuración (src/config):** variables de entorno, conexión a base de datos y envío de correos (config.ts, mailer.ts, helper.ts).

Esta separación facilita el mantenimiento y permite ubicar rápidamente en qué capa corregir o ampliar una funcionalidad.

Tecnologías utilizadas

- **Node.js:** entorno de ejecución del servidor.
- **Express 5:** framework web utilizado para el enrutamiento y el manejo de peticiones/respuestas HTTP.
- **TypeScript:** tipado estático sobre todo el código fuente del backend.

- **MySQL (mysql2/promise):** motor de base de datos relacional, con consultas SQL parametrizadas.
- **JWT (jsonwebtoken):** generación y verificación de tokens de sesión.
- **bcrypt:** cifrado (hash) de contraseñas de usuario.
- **Multer:** manejo de carga de archivos (imágenes de productos y fotos de perfil).
- **Nodemailer:** envío de correos con el código de verificación de cuenta.
- **dotenv:** carga de variables de entorno desde el archivo .env.

Lenguajes

El proyecto combina tres lenguajes con roles claramente diferenciados dentro del flujo de desarrollo:

TypeScript

Es el lenguaje principal del backend. Todo el código fuente en src/ (rutas, controladores, modelos y configuración) está escrito en TypeScript. Cada entidad del negocio cuenta con su propia interfaz en src/models (por ejemplo Product, Order, User, Cart, Calification), lo que permite tipar tanto el cuerpo de las peticiones (req.body as Product) como los resultados devueltos por la base de datos. El proyecto se compila con el compilador oficial (tsc) según la configuración de tsconfig.json, generando JavaScript en la carpeta dist/.

SQL (dialecto MySQL)

Se utiliza para todas las operaciones de lectura y escritura sobre la base de datos. Las consultas se escriben de forma manual (no hay ORM ni query builder) dentro de cada controlador, usando

parámetros preparados con el driver mysql2/promise para evitar inyección SQL, por ejemplo: SELECT * FROM productos WHERE id = ? o INSERT INTO pedidos (...) VALUES (?, ?, ?, ?, ?, ?, ?).

JavaScript (generado)

Es el resultado de la compilación de TypeScript. El script de arranque en producción (npm run start) ejecuta directamente node dist/src/index.js, mientras que en desarrollo se utiliza ts-node-dev src/index.ts para recompilar y reiniciar el servidor automáticamente ante cada cambio.

Estructura del proyecto

La carpeta src concentra todo el código fuente del backend, organizada de la siguiente manera:

Funcionamiento de la API

Cada endpoint recibe los parámetros de la petición (body, params o query), valida manualmente los datos obligatorios, ejecuta la consulta correspondiente contra la base de datos y devuelve un código HTTP adecuado junto con un cuerpo JSON. En la sección "Códigos de estado HTTP" de este documento se detalla el criterio usado para cada código.

Flujo de una petición

El recorrido típico de una petición dentro de la API es el siguiente:

Módulos

La API implementa 21 módulos de recursos, cada uno con su propio conjunto de operaciones CRUD (crear, consultar, actualizar y eliminar):

- Autenticación (login, registro, cambio de contraseña, verificación por correo)
- Usuarios (incluye validación de username, correo, documento y teléfono, y carga de foto de perfil)
- Roles
- Productos (incluye carga de imagen)
- Categorías y Subcategorías
- Proveedores, Categorías de proveedor y Teléfonos de proveedor
- Carrito de compras y Detalle de carrito
- Pedidos y Detalle de pedido
- Direcciones de usuario
- Métodos de pago de usuario
- Teléfonos de usuario
- Calificaciones de productos
- Historial de comparaciones de productos
- Soporte (tickets de usuario)
- Control de acciones (registro de auditoría de usuario)

Seguridad

La API implementa los siguientes mecanismos de seguridad:

Autenticación basada en JWT: al iniciar sesión se firma un token que incluye id, email, user_name y rol_id, con una vigencia de 8 horas. Cifrado de contraseñas con bcrypt (10 rondas de sal) tanto en el registro como en la verificación de contraseña. Verificación de cuenta por correo: se genera un código numérico de 6 dígitos, válido durante 3 minutos, que se envía mediante Nodemailer.

Control de estado del usuario: un usuario con estado inactivo (estado = 0) no puede iniciar sesión (respuesta 403). Uso de consultas parametrizadas en todas las operaciones SQL, lo que previene inyección SQL.

Como punto a reforzar: la clave usada para firmar los JWT (JWT_SECRET) y las credenciales del correo de envío (mailer.ts) están escritas directamente en el código fuente en lugar de leerse desde variables de entorno. Se recomienda moverlas al archivo .env antes de pasar a un entorno productivo, tal como ya se hace con las credenciales de la base de datos.

Base de datos

La API utiliza MySQL como motor de base de datos relacional, accedido mediante el driver mysql2 en su variante con promesas (mysql2/promise). No se utiliza ningún ORM ni query builder (no hay Sequelize, TypeORM ni Prisma en el proyecto): todas las consultas SQL se escriben de forma manual dentro de cada controlador y se ejecutan a través de la función query() definida en src/database/db.ts.

Cada llamada a query() abre una nueva conexión con mysql.createConnection(config.db) en lugar de reutilizar un pool de conexiones. Esto es funcional para el volumen actual del proyecto, pero se recomienda migrar a mysql.createPool() para mejorar el rendimiento bajo mayor carga concurrente.

La base de datos está compuesta por 19 tablas principales que corresponden una a una con las entidades descritas en la sección "Modelo de datos y entidades" de este documento: usuarios, roles, productos, categorías, subcategorías, proveedores y sus tablas de relación, carritos y su detalle, pedidos y su detalle, direcciones y métodos de pago de usuario, teléfonos de usuario y de proveedor, calificaciones, historial de comparaciones, soporte y control de acciones.

La configuración de conexión (host, puerto, usuario, contraseña y nombre de la base de datos) se obtiene desde variables de entorno definidas en el archivo .env y centralizadas en src/config/config.ts.

Ventajas de TypeScript

En el contexto específico de este proyecto, el uso de TypeScript aporta beneficios concretos:
Contratos claros por entidad: cada una de las 19 entidades del negocio tiene su propia interfaz (por ejemplo, User define 20 campos, incluyendo campos opcionales como foto_perfil o telefono), lo que deja explícito qué datos espera y devuelve cada endpoint.

Detección temprana de errores: al tipar req.body as Order o req.body as Product, el compilador avisa si en un controlador se intenta leer o insertar un campo que no existe en el modelo, antes de llegar a producción.
Autocompletado y refactorización más segura: con 21 controladores y 19 modelos, el editor puede sugerir los campos exactos de cada entidad (id, estado_actual, subcategoria_id, etc.), reduciendo errores de tipeo en nombres de columnas.

Tipado de Express: las funciones de los controladores están tipadas como (req: Request, res: Response) => Promise<Response>, lo que ayuda a detectar rutas de código que no siempre devuelven una respuesta.
Mejor mantenimiento a largo plazo: al ser un proyecto con 22 archivos de rutas y decenas de endpoints, el tipado estático facilita que un desarrollador nuevo entienda la forma de los datos sin depender únicamente de la documentación.

Endpoints por módulo

Todas las rutas se montan directamente sobre la raíz del servidor (por ejemplo, http://localhost:3000/products/all), ya que actualmente la API no utiliza un prefijo de versión como /api o /api/v1 (ver sección "Versionado de la API").

Autenticación (/auth)

Método	Ruta
POST	/auth/login
POST	/auth/register
PUT	/auth/change-password/:id
POST	/auth/send-code
POST	/auth/verify-code

Usuarios (/users)

Método	Ruta
GET	/users/all
GET	/users/check-username
GET	/users/check-email
GET	/users/check-document
GET	/users/check-phone
GET	/users/:id
POST	/users/create
PUT	/users/update/:id
DELETE	/users/delete/:id

Método	Ruta
PUT	/users/upload-photo/:id

Roles (/roles)

Método	Ruta
GET	/roles/all
GET	/roles/:id
POST	/roles/create
PUT	/roles/update/:id
DELETE	/roles/delete/:id

Productos (/products)

Método	Ruta
GET	/products/all
GET	/products/:id
POST	/products/create
PUT	/products/update/:id
DELETE	/products/delete/:id

Categorías (/categories)

Método	Ruta
GET	/categories/all
GET	/categories/:id
POST	/categories/create
PUT	/categories/update/:id
DELETE	/categories/delete/:id

Subcategorías (/subcategories)

Método	Ruta
GET	/subcategories/all
GET	/subcategories/:id
POST	/subcategories/create
PUT	/subcategories/update/:id
DELETE	/subcategories/delete/:id

Proveedores (/suppliers)

Método	Ruta
GET	/suppliers/all
GET	/suppliers/:id
POST	/suppliers/create
PUT	/suppliers/update/:id
DELETE	/suppliers/delete/:id

Categorías de proveedor (/supplierCategories)

Método	Ruta
GET	/supplierCategories/all
GET	/supplierCategories/:proveedor_id/:categoria_id
POST	/supplierCategories/create
PUT	/supplierCategories/update/:proveedor_id/:categoria_id
DELETE	/supplierCategories/delete/:proveedor_id/:categoria_id

Teléfonos de proveedor (/supplierPhones)

Método	Ruta
GET	/supplierPhones/all
GET	/supplierPhones/:id
POST	/supplierPhones/create
PUT	/supplierPhones/update/:id
DELETE	/supplierPhones/delete/:id

Carrito (/cart)

Método	Ruta
GET	/cart/all
GET	/cart/:id
POST	/cart/create
PUT	/cart/update/:id
DELETE	/cart/delete/:id

Detalle de carrito (/cartDetails)

Método	Ruta
GET	/cartDetails/all
GET	/cartDetails/:id
POST	/cartDetails/create
PUT	/cartDetails/update/:id
DELETE	/cartDetails/delete/:id

Pedidos (/orders)

Método	Ruta
GET	/orders/all
GET	/orders/:id
POST	/orders/create
PUT	/orders/update/:id
DELETE	/orders/delete/:id

Detalle de pedido (/orderDetails)

Método	Ruta
GET	/orderDetails/all
GET	/orderDetails/:id
POST	/orderDetails/create
PUT	/orderDetails/update/:id
DELETE	/orderDetails/delete/:id

Direcciones de usuario (/userAddresses)

Método	Ruta
GET	/userAddresses/all
GET	/userAddresses/:id
POST	/userAddresses/create
PUT	/userAddresses/update/:id
DELETE	/userAddresses/delete/:id

Métodos de pago (/userPaymentMethods)

Método	Ruta
GET	/userPaymentMethods/all

Método	Ruta
GET	/userPaymentMethods/:id
POST	/userPaymentMethods/create
PUT	/userPaymentMethods/update/:id
DELETE	/userPaymentMethods/delete/:id

Teléfonos de usuario (/phoneUsers)

Método	Ruta
GET	/phoneUsers/all
GET	/phoneUsers/:id
POST	/phoneUsers/create
PUT	/phoneUsers/update/:id
DELETE	/phoneUsers/delete/:id

Calificaciones (/califications)

Método	Ruta
GET	/califications/all
GET	/califications/:id
POST	/califications/create
PUT	/califications/update/:id
DELETE	/califications/delete/:id

Historial de comparaciones (/comparisonHistories)

Método	Ruta
GET	/comparisonHistories/all
GET	/comparisonHistories/:id
POST	/comparisonHistories/create
PUT	/comparisonHistories/update/:id
DELETE	/comparisonHistories/delete/:id

Soporte (/support)

Método	Ruta
GET	/support/all
GET	/support/:id
POST	/support/create

Método	Ruta
PUT	/support/update/:id
DELETE	/support/delete/:id

Control de acciones (/actionControls)

Método	Ruta
GET	/actionControls/all
GET	/actionControls/:id
POST	/actionControls/create
PUT	/actionControls/update/:id
DELETE	/actionControls/delete/:id

Ejemplos de Request / Response

A continuación se muestran ejemplos reales de uso de tres endpoints clave, construidos a partir de la lógica implementada en los controladores correspondientes. Los valores de ejemplo son ilustrativos.

1. **Iniciar sesión — POST /auth/login**

2. **Crear un pedido — POST /orders/créate**

3. Listar productos — GET /products/all

Códigos de estado HTTP

La API utiliza los siguientes códigos de respuesta de forma consistente en todos sus controladores:

Código	Cuándo se usa en la API
200 OK	Operación de consulta o actualización exitosa (login correcto, obtener un recurso, actualizar/eliminar un registro existente).
201 Created	Se creó un nuevo recurso en la base de datos (registro de usuario, creación de producto, pedido, categoría, etc.).
400 Bad Request	Faltan campos obligatorios en el body o no hay campos para actualizar. Se valida manualmente al inicio de cada controlador.
401 Unauthorized	Credenciales inválidas: contraseña incorrecta en login o en cambio de contraseña.
404 Not Found	El recurso solicitado (usuario, producto, pedido, etc.) no existe en la base de datos.
500 Internal Server Error	Error inesperado del servidor o de la conexión a la base de datos, capturado en el bloque catch de cada controlador.

Adicionalmente, en módulos puntuales se usan otros códigos: 403 Forbidden (usuario inactivo al iniciar sesión), 409 Conflict (correo ya registrado) y 410 Gone (código de verificación expirado).

Variables de entorno (.env)

El proyecto usa dotenv para cargar la configuración sensible desde un archivo .env que no se versiona en el repositorio. Las variables definidas son:

Variable	Uso
SERVER_URL	Host base usado para construir la URL con la que se anuncia el servidor (por defecto http://localhost).
PORT	Puerto en el que Express levanta el servidor HTTP.
DB_HOST	Host del servidor MySQL al que se conecta la API.
DB_PORT	Puerto de conexión al servidor MySQL.
DB_DATABASE	Nombre de la base de datos utilizada por la API.
DB_USERNAME	Usuario con el que la API se autentica en MySQL.
DB_PASSWORD	Contraseña del usuario de la base de datos.

Nota: *la clave de firma de JWT y las credenciales del correo de envío (Nodemailer) no están actualmente en el .env, sino escritas directamente en el código fuente (src/controllers/auth.ts y src/config/mailer.ts). Se recomienda migrarlas a variables de entorno (por ejemplo JWT_SECRET, MAIL_USER, MAIL_PASSWORD) como mejora de seguridad.*

Modelo de datos y entidades

La base de datos está compuesta por 19 entidades. A continuación se listan los modelos (interfaces TypeScript en src/models) con sus campos principales y su relación con otras entidades.

Entidad	Campos principales	Relación
Usuario (User)	id, nombre_completo, tipo_documento, identificacion, fecha_nacimiento, email, password, verificado, token_verificacion, user_name, foto_perfil, rol_id, estado, telefono, created_at, updated_at	Pertenece a un Rol (rol_id).
Rol (Role)	nombre, created_at, updated_at	Un rol puede tener muchos usuarios.
Producto (Product)	nombre, precio, marca, descripcion, caracteristicas_tecnicas, imagen, stock, costo, subcategoria_id, created_at, updated_at	Pertenece a una Subcategoría.
Categoría (Category)	descripcion, created_at, updated_at	Tiene muchas Subcategorías.
Subcategoría (Subcategory)	descripcion, categoria_id, created_at, updated_at	Pertenece a una Categoría; tiene muchos Productos.
Pedido (Order)	cod_factura, fecha, canal_venta, valor, estado_actual, historial_estados, usuario_id, created_at, updated_at	Pertenece a un Usuario; tiene muchos Detalles de pedido.

Entidad	Campos principales	Relación
Detalle de pedido (OrderDetail)	cantidad, valor_producto, iva_porcentaje, subtotal, iva, total, pedido_id, producto_id, created_at, updated_at	Pertenece a un Pedido y a un Producto.
Carrito (Cart)	usuario_id, activo, created_at, updated_at	Pertenece a un Usuario; tiene muchos Detalles de carrito.
Detalle de carrito (CartDetail)	carrito_id, producto_id, cantidad, subtotal, created_at, updated_at	Pertenece a un Carrito y a un Producto.
Calificación (Calification)	id, estado, puntuacion, comentario, imagen, producto_id, usuario_id, created_at, updated_at	Pertenece a un Producto y a un Usuario.
Proveedor (Supplier)	nombre, correo, created_at, updated_at	Relación muchos-a-muchos con Categoría (vía SupplierCategory); tiene muchos Teléfonos de proveedor.
Categoría de proveedor (SupplierCategory)	proveedor_id, categoria_id, created_at, updated_at	Tabla intermedia entre Proveedor y Categoría (llave compuesta).
Teléfono de proveedor (SupplierPhone)	proveedor_id, telefono, created_at, updated_at	Pertenece a un Proveedor.
Dirección de usuario (UserAddress)	usuario_id, direccion, barrio, ciudad, departamento, pais, created_at, updated_at	Pertenece a un Usuario.
Método de pago (UserPaymentMethod)	usuario_id, tipo, numero_parcial, titular, fecha_expiracion, created_at, updated_at	Pertenece a un Usuario.
Teléfono de usuario (phoneUser)	usuario_id, telefono, created_at, updated_at	Pertenece a un Usuario.
Historial de comparación (ComparisonHistory)	detalle_comparacion, producto_id_1, producto_id_2, producto_id_3, usuario_id, created_at, updated_at	Referencia hasta tres Productos y un Usuario.
Soporte (Support)	usuario_id, tipo, descripcion, estado, created_at, updated_at	Pertenece a un Usuario.
Control de acciones (ActionControl)	usuario_id, accion, observaciones, created_at, updated_at	Pertenece a un Usuario; registro de auditoría.

Resumen de relaciones clave: Usuario 1—N Pedido, Pedido 1—N Detalle de pedido, Producto 1—N Detalle de pedido, Usuario 1—1 Carrito activo, Carrito 1—N Detalle de carrito, Categoría 1—N Subcategoría, Subcategoría 1—N Producto, y Proveedor N—N Categoría a través de la tabla intermedia SupplierCategory.

Validaciones

El proyecto no utiliza una librería de validación como Joi o express-validator (no aparecen en las dependencias del package.json). Las validaciones se realizan de forma manual al inicio de cada función

del controlador, comprobando con condicionales que los campos obligatorios estén presentes antes de continuar. Por ejemplo, en el login se valida que existan user_name y password, y en la creación de un pedido se valida que existan cod_factura, valor y usuario_id, devolviendo 400 Bad Request en caso contrario.

Como complemento, TypeScript aporta una validación en tiempo de compilación (tipos de los campos), y el uso de consultas parametrizadas en mysql2 evita que datos maliciosos alteren la consulta SQL. No obstante, no hay validación de formato (por ejemplo, formato de correo, longitud de contraseña o rangos numéricos) más allá de la presencia del campo, lo cual queda registrado como oportunidad de mejora.

Manejo de errores

La API no cuenta con un middleware de errores centralizado (no existe un app.use con firma de cuatro parámetros registrado en src/api/api.ts). En su lugar, el manejo de errores se realiza por controlador: cada función envuelve su lógica en un bloque try/catch, registra el error en consola con console.error y responde con res.status(500).json({ error: 'Error interno del servidor' }).

Esto mantiene un formato de respuesta de error consistente en toda la API, aunque implica que el mismo patrón de manejo se repite en decenas de funciones. Centralizar este manejo en un middleware de errores de Express permitiría reducir la duplicación y estandarizar el registro de errores (logging).

Versionado de la API

Actualmente la API no implementa versionado. Todas las rutas se registran directamente sobre la raíz del servidor (this.router.use('/products', product), etc., montado luego en expressApp.use('/', this.router)), por lo que un endpoint se consume como http://host:puerto/products/all y no como /api/v1/products.
Se recomienda introducir un prefijo de versión (por ejemplo /api/v1) antes de exponer la API a consumidores externos, de forma que futuras versiones con cambios incompatibles (por ejemplo /api/v2) puedan convivir con la actual sin romper integraciones existentes.

Documentación de la API (Postman / Swagger)

El proyecto no cuenta con documentación Swagger/OpenAPI. Sí incluye una colección de Postman ubicada en la carpeta postman/collections/APIS/TAM del repositorio, organizada en subcarpetas por módulo (Auth, Usuarios, Productos, Subcategorías, Roles, Direccion_usuarios, Metodo_pago_usuarios, Historial_comparaciones, Detalle_pedido, Pedidos, entre otras), con cada endpoint definido como un archivo de solicitud individual.

Esta colección se adjunta comprimida junto con este informe para que el equipo evaluador pueda importarla directamente en Postman. Como mejora futura, se recomienda complementar esta colección con documentación Swagger/OpenAPI generada a partir del código, lo que permitiría exponer una interfaz interactiva de prueba de los endpoints.

Posibles mejoras

Introducir un prefijo de versión (/api/v1) en todas las rutas.
Centralizar el manejo de errores en un middleware único de Express.
Incorporar una librería de validación (Joi o express-validator) para validar formato y tipo de datos, no solo su presencia.

Mover el JWT_SECRET y las credenciales de correo a variables de entorno. Usar un pool de conexiones (mysql.createPool) en lugar de abrir una conexión nueva por cada consulta.

Agregar documentación Swagger/OpenAPI complementaria a la colección de Postman existente. Añadir pruebas unitarias y de integración, registro de logs estructurado, paginación en los listados, y un pipeline de CI/CD.

Conclusiones

La API TAM implementa una arquitectura por capas ordenada, con 21 módulos de recursos y un módulo de autenticación, todos escritos en TypeScript sobre Express y MySQL. El uso de JWT y bcrypt cubre las necesidades básicas de autenticación y protección de contraseñas, y la tipificación de cada entidad facilita el mantenimiento del código.

Al mismo tiempo, el análisis del código evidencia oportunidades concretas de mejora: introducir versionado de la API, centralizar el manejo de errores, formalizar las validaciones de entrada, mover credenciales sensibles a variables de entorno y complementar la colección de Postman existente con documentación Swagger/OpenAPI. Abordar estos puntos dejaría la API en una posición sólida para escalar hacia un entorno de producción.