**Respuesta**

**Buenas prácticas y estructura de código**

* Código modularizado: Separar rutas, servicios y controladores.
* Uso explícito de tipos con TypeScript para evitar errores.
* Manejo adecuado de errores con try/catch y respuestas HTTP claras.
* Validaciones robustas (ejemplo: formato y longitud del token).
* No exponer información sensible (como el CVV) en las respuestas.
* Código limpio y legible: nombres descriptivos, comentarios puntuales.
* Tests automatizados con Jest y supertest para asegurar la funcionalidad.

**2. Patrón de diseño utilizado**

**Patrón MVC (Model - View - Controller)**

Aunque no hay vistas (como en frontend), el patrón **MVC** se usa así:

* **Model**: models/ define la estructura de los datos y cómo se almacenan.
* **Controller**: controllers/ maneja las peticiones HTTP y envía las respuestas.
* **Service**: services/ actúa como la lógica de negocio (separa el controller del acceso a datos).
* **Repository**: repositories/ implementa el patrón **DAO** (Data Access Object) para desacoplar el acceso a la base de datos.

Este enfoque **mejora el desacoplamiento, la escalabilidad y la mantenibilidad** del sistema.

**3. Patrón de arquitectura utilizado**

**Arquitectura en capas (Layered Architecture)**

Las capas identificables son:

1. **Capa de presentación** (Express Controllers + Rutas)
2. **Capa de negocio** (Services)
3. **Capa de persistencia** (Repositories + Sequelize Models)
4. **Capa de configuración e infraestructura** (config/, Redis, DB)

Cada capa tiene su responsabilidad, lo que **favorece el testing unitario**, la reutilización de código y la separación de preocupaciones.

1.- **Modelo de arquitectura y estructura de carpetas**

Tu proyecto sigue una **arquitectura en capas (layered architecture)**, común en aplicaciones backend con Node.js y TypeScript. Esta arquitectura divide las responsabilidades en capas bien definidas.

|  |
| --- |
| src/  │  ├── app.ts → Punto de entrada de la aplicación  │  ├── config/ → Configuración (DB, variables de entorno)  │ ├── db.config.ts → Parámetros de conexión  │ └── db.init.ts → Inicialización de conexión  │  ├── controllers/ → Lógica de control de rutas (request/response)  │ ├── card.controller.ts  │ └── token.controller.ts  │  ├── middlewares/ → Middlewares personalizados  │ ├── auth.middleware.ts  │ └── validation.middleware.ts  │  ├── models/ → Modelos Sequelize que representan tablas  │ ├── card.model.ts  │ ├── Merchant.ts  │ └── token.model.ts  │  ├── repositories/ → Capa de acceso a datos (DAO)  │ └── token.repository.ts  │  ├── routes/ → Define rutas de endpoints REST  │ ├── card.router.ts  │ └── token.router.ts  │  ├── services/ → Lógica de negocio y reglas específicas  │ ├── card.service.ts  │ ├── token.service.ts  │ └── validation.service.ts  │  ├── utils/ → Utilidades auxiliares reutilizables  │ ├── email.validator.ts  │ ├── luhn.algorithm.ts  │ ├── luhnValidator.ts  │ └── tokenGenerator.ts |