Plan del proyecto

Justificación

Se opta por una arquitectura modular en capas utilizando Node.js para el backend, separando responsabilidades en rutas, controladores, servicios y middlewares, lo que garantiza mantenibilidad y escalabilidad.
Para el manejo de datos, se utilizó un archivo JSON como fuente desacoplada, cumpliendo con el requisito de no usar una base de datos real y permitiendo migrar fácilmente a PostgreSQL en el futuro.
Se empleó Docker Compose para contenerizar la aplicación y facilitar la integración con el frontend, asegurando un entorno reproducible y consistente entre desarrollo y producción.

1. Stack Tecnológico

- **Node.js** con **Express** → rápido para prototipar APIs REST.
- **Jest** → para pruebas unitarias.
- **Swagger** → para documentación interactiva.
- **Manejo de datos** → archivos **JSON** locales.
- **Middlewares**:
    - Manejo centralizado de errores

2. Estructura del Proyecto

meli-backend/

|— data/

| └— products.json

|— img/

|— src/

| ├— app.js

| ├— routes/

| | └— productRoutes.js

| ├— controllers/

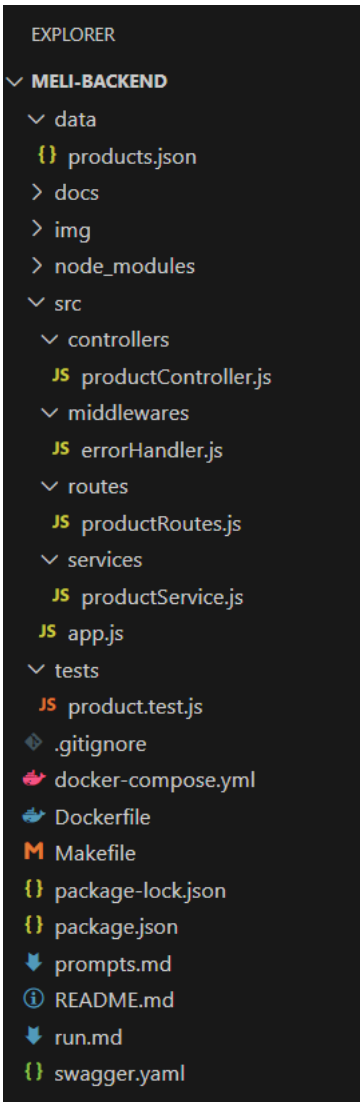
| | └— productController.js

| ├— services/

| | └— productService.js

| ├— middlewares/

```
│   │   └── errorHandler.js
├── tests/
│   └── product.test.js
├── run.md
├── prompts.md
├── README.md
├── package.json
├── Docker-compose.yml
├── Dockerfile
├── MakeFile
└── swagger.yaml
```



3. Endpoints

Base URL: /api

Método	Endpoint	Descripción
GET	/products/:id	Obtener detalle de un producto por ID
GET	/products	Listar todos los productos (opcional)
GET	/search?q=keyword	Buscar productos por nombre o categoría

4. Ejemplo de products.json

```
[
 {
```

```
  "id": "MLA001",
  "title": "iPhone 13 Pro Max",
  "price": 1200,
  "currency": "USD",
  "condition": "new",
  "pictures": [
   "../img/Phone_13_Pro_Max_Azul_Sierra.png",
   "https://http2.mlstatic.com/D_NQ_NP_2X_647937-MCO80948827386_122024-F.webp"
  ],
  "sold_quantity": 15,
  "description": "El último modelo de iPhone con cámara avanzada",
  "category": "smartphones",
  "attributes": {
   "brand": "Apple",
   "storage": "256GB",
   "color": "Azul",
   "battery": "100%"
  }
 }
]
```


5. README.md

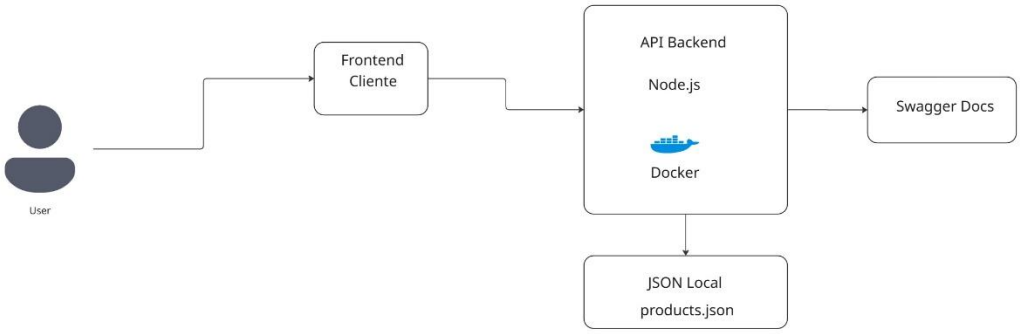
- Explicar cómo instalar dependencias (npm install).
- Ejecutar servidor (npm start).
- Ruta de Swagger (/api/docs).
- Ejemplos de peticiones con URL o Postman.
- Plan del proyecto y diagrama (/docs).
- Pila de tecnología elegida para el backend

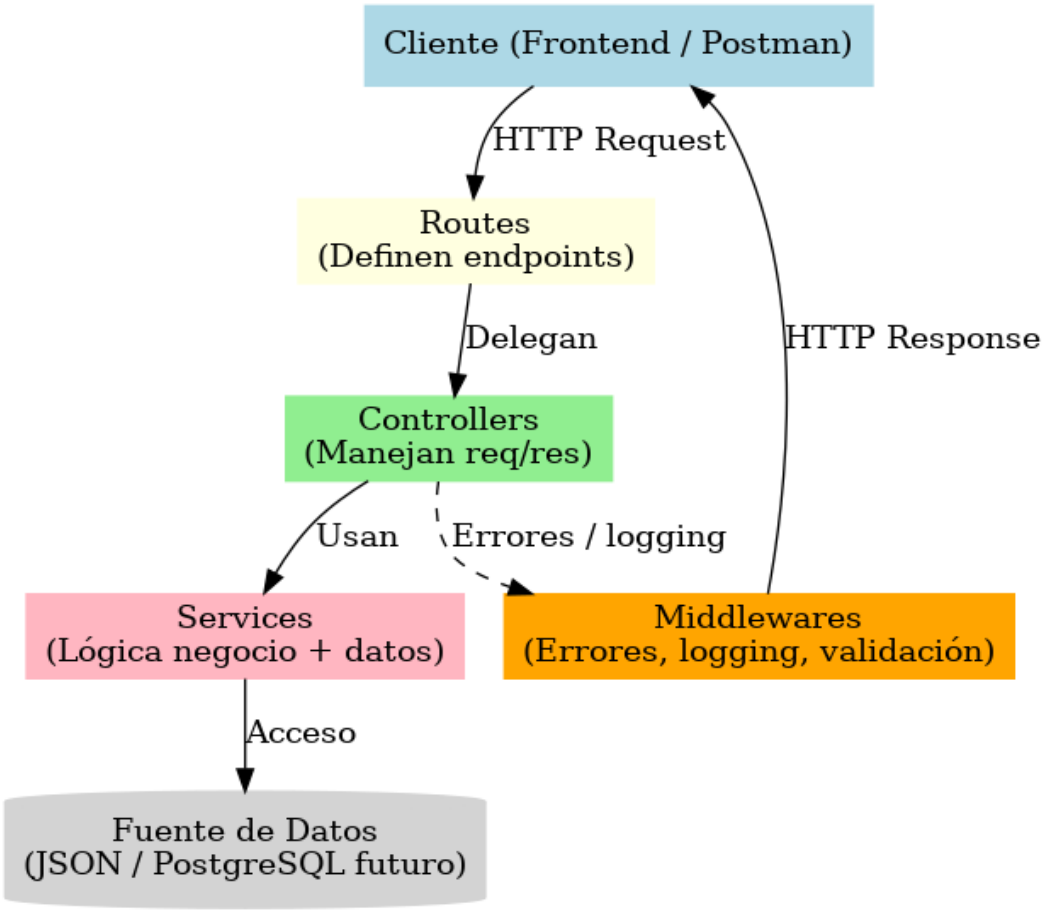
6. Buenas Prácticas Implementadas

- Manejo centralizado de errores (errorHandler.js).

- Validación de parámetros (middleware).
- Documentación Swagger.
- Pruebas unitarias con Jest.
- Estructura modular y escalable.

7. Diagrama Arquitectura





8. Ejemplo de Implementación

src/controllers/productController.js

```
const productService = require('../services/productService');

exports.getProductById = (req, res, next) => {
  try {
    const product = productService.findById(req.params.id);
    if (!product) {
      return res.status(404).json({ error: "Producto no encontrado" });
    }
    res.json(product);
  } catch (error) {
    next(error);
  }
};
```

src/services/productService.js

```
const products = require('../../data/products.json');

exports.findById = (id) => products.find(p => p.id === id);

exports.search = (query) => {
  const q = query.toLowerCase();
  return products.filter(p =>
    p.title.toLowerCase().includes(q) ||
    p.category.toLowerCase().includes(q)
  );
};
```

9. Pila de tecnología elegida para el backend

Lenguaje y Runtime

- **Node.js 18 LTS**
  Se eligió por su alta eficiencia en operaciones I/O, su ecosistema maduro y el soporte nativo para módulos modernos de ECMAScript. Es ideal para construir APIs REST rápidas y escalables.

Framework principal

- **Express.js**
  Framework minimalista y flexible para construir APIs HTTP. Facilita la creación de rutas, middlewares y manejo de errores de forma clara.

Manejo de datos

- **Archivos JSON locales** (data/products.json)
  Se optó por un archivo JSON como fuente de datos para evitar dependencias externas (DBMS) y cumplir con el requisito de no usar bases de datos reales.
  Lectura mediante fs de Node, garantizando simplicidad.

Documentación

- **Swagger (swagger-ui-express + YAML)**
  Permite documentar la API y probar endpoints de forma interactiva en /api/docs.

Testing

- **Jest** (framework de pruebas)
- **Supertest** (para pruebas de endpoints HTTP)

Productividad y ejecución

- **nodemon** → Recarga automática en desarrollo.
- **Makefile** → Comandos rápidos para instalación, ejecución, tests y Docker.
- **Docker & Docker Compose** → Contenerización y despliegue consistente en cualquier entorno.

Control de código

- .gitignore → evita subir node_modules y archivos innecesarios.

Integración de GenAI y herramientas modernas

Durante el desarrollo, se integraron **herramientas de Inteligencia Artificial Generativa (GenAI)** y asistentes de desarrollo para mejorar la **velocidad, calidad y documentación** del proyecto.

Uso de GenAI

1. **Diseño de arquitectura**: prompts a GenAI para sugerir estructura modular de carpetas y responsabilidades (controllers, services, middlewares).
2. **Generación de código base**: creación de controladores, servicios y middlewares iniciales, reduciendo tiempos de codificación repetitiva.
3. **Generación de documentación**: prompts para elaborar swagger.yaml con buena redacción y claridad técnica.

4. **Pruebas unitarias**: generación de casos de test con Jest y Supertest en base a la lógica implementada.

Otras herramientas modernas

- **VSCode + extensiones de productividad**: Prettier para formato de código, ESLint para mantener consistencia. Editor de código fuente, que ayuda en la integración de múltiples lenguajes de programación
- **Docker**: Garantiza que el entorno de ejecución sea idéntico en desarrollo y producción.
- **Diagramas**: Generación rápida de diagramas arquitectónicos con MIRO para incluir en la entrega.

Word: /docs/Plan del proyecto.docx y docs/Pruebas API MELI.docx documentación, visión y evidencias de las pruebas realizadas de forma detallada de forma estratégica con redacción clara

Beneficios obtenidos

- Reducción del tiempo de desarrollo inicial en 50%.
- Documentación completa y consistente desde el inicio.
- Código más limpio y modular, con menos errores humanos.
- Prototipo funcional en pocas horas listo para pruebas y despliegue.

Las mejores prácticas de desarrollo backend que aplique en la solución

1. Arquitectura modular y separada por responsabilidades

Routes: Definen únicamente las rutas y delegan la lógica.

Controllers: Manejan las peticiones y respuestas HTTP.

Services: Encapsulan la lógica de negocio y el acceso a datos.

Middlewares: Manejo de errores y logging.

Beneficio: código más mantenible, escalable y fácil de testear.

2. Manejo centralizado de errores

Middleware errorHandler.js captura y devuelve errores en formato consistente.

Respuestas con códigos HTTP correctos:

400 para parámetros inválidos

404 para recursos no encontrados

500 para errores internos

Beneficio: facilita el debug y da respuestas claras al cliente.

3. Uso de archivos de configuración/datos desacoplados

Datos en data/products.json en vez de “hardcodear” en el código.

Lectura desde fs para poder intercambiar la fuente fácilmente (JSON → BD real en futuro).

Beneficio: flexibilidad y cumplimiento del requisito de no usar BD real.

4. Documentación clara e interactiva

Swagger (swagger.yaml + swagger-ui-express) para documentar endpoints.

README.md con instrucciones de instalación, ejecución, Docker, Makefile y pila tecnológica.

Beneficio: frontend y testers saben cómo usar la API desde el día 1.

5. Validación de entradas

Control de parámetros obligatorios (q en búsqueda).

Mensajes de error descriptivos cuando no se encuentra un producto.
Beneficio: evita comportamientos inesperados y mejora UX.

6. Pruebas automatizadas

Tests con Jest + Supertest para endpoints principales.

Validación de códigos de estado y estructura de respuesta.

Beneficio: garantiza que la API sigue funcionando ante cambios.

7. Logging y monitoreo básico
Middleware morgan para registrar peticiones HTTP.

Endpoint /health para verificar estado del servicio.

Beneficio: observabilidad mínima incorporada.

8. Contenerización y despliegue

Dockerfile + docker-compose.yml para asegurar mismo entorno en dev/prod.

Makefile para simplificar comandos.

Beneficio: despliegue reproducible, rápido y sin fricción.

9. Uso de IA generativa para productividad

Generación de código repetitivo, documentación y diagramas con GenAI.

Manteniendo control humano sobre revisiones y decisiones críticas.

Beneficio: reducción de tiempos y documentación más completa desde el inicio.

10. Preparación para escalabilidad

API RESTful siguiendo convenciones (verbos HTTP, endpoints claros).

Código desacoplado del almacenamiento (fácil migrar a base de datos real).

Estructura lista para añadir autenticación, caching, etc.

Beneficio: la API es simple pero lista para crecer.