Guatemala 01 de febrero del 2026.

Universidad De San Carlos De Guatemala.

Laboratorio software avanzado sección “A”.

Auxiliar: Juan Pablo Samayoa Ruiz.

Practica #3: gRPC y comunicación entre microservicios

Ciclo Escolar 2026

Estudiante: Juan Carlos Maldonado Solorzano.

Carnet: 2012-226-87

1) Introducción

Delivereats es una plataforma de delivery basada en una arquitectura orientada a microservicios. En esta práctica se implementa microservicios, backend/frontend, base de datos (MySQL) la cual será consumida posteriormente durante el desarrollo completo del sistema. Se utilizo (Principios SOLID) para la creación del registro e inicio de sesión de la plataforma. Demostración de la comunicación entre los microservicios de órdenes y catalogo para garantizar una definición correcta del Core de negocio del proyecto.

2) Objetivos

2.1 Objetivo general

Aplicar de manera efectiva la metodología ágil **SCRUM** para la construcción de la aplicación mediante la entrega continua y progresiva de incrementos funcionales. A través de este enfoque iterativo, se busca planificar, diseñar, documentar y validar componentes del sistema en ciclos cortos, garantizando una evolución constante del proyecto, una adecuada adaptación a cambios y una gestión eficiente del tiempo y los recursos.

Implementar un sistema de gestión de identidades y acceso utilizando microservicios, aplicando principios de diseño de software para garantizar una base técnica escalable y segura.

Implementar un mecanismo de validación de datos en tiempo real entre microservicios utilizando el protocolo gRPC, asegurando que el proceso de creación de órdenes en Delivereats cumpla con las reglas de negocio y la consistencia de datos sin acoplamiento de persistencia.

2.2 Objetivos específicos

• Definir un contrato de servicio mediante Protocol Buffers (.proto) para la verificación de catálogos y precios.
• Desarrollar un servidor gRPC en el Restaurant-Catalog-Service que exponga métodos de consulta de productos y disponibilidad.
• Implementar un cliente gRPC en el Order-Service que actúe como interceptor de lógica antes de la persistencia de una orden.
• Garantizar que cada microservicio gestione su propia base de datos de forma independiente, cumpliendo con la arquitectura de sistemas distribuidos.
• **Solicitud de creación de pedido:** El Restaurant-Catalog-Service valida que cada producto solicitado exista en su base de datos y pertenezca realmente al restaurante seleccionado.
• **Verificación de existencia y pertenencia:** El Restaurant-Catalog-Service valida que cada producto solicitado exista en su base de datos y pertenezca realmente al restaurante seleccionado.
• **Validación de disponibilidad:** El catálogo confirma si los ítems del menú están marcados como disponibles antes de autorizar la continuación del pedido
• **Notificación de error al usuario:** El sistema genera una respuesta clara hacia el cliente indicando el motivo por el cual no se pudo procesar su pedido

Despliegue inicial:
• Archivos .yml configurados para levantar ambos servicios y sus respectivas bases de datos, permitiendo la visibilidad de red necesaria para gRPC.

3) Alcance

Trabajar en la intersección de dos servicios clave definidos en el alcance del proyecto:

Restaurant-Catalog-Service (Servidor gRPC)

• Lógica de Verificación: Un procedimiento que reciba una lista de IDs de productos y un ID de restaurante para validar que:
    - o Los productos existen en la base de datos de este servicio.
    - o Los productos pertenecen efectivamente al restaurante indicado.
    - o Los precios actuales coinciden con los solicitados por el cliente.
• **Base de Datos de Catálogo:** Debe contener los registros de menús y precios actualizados.

Order-Service (Cliente gRPC)

• **Flujo de Creación de Orden:** Modificar el endpoint de "Realizar Orden" para que, antes de guardar en su base de datos, realice una llamada remota al servidor de catálogo.
• **Manejo de Estados de Error:** Si la validación gRPC falla (por precio incorrecto o falta de stock), el servicio debe rechazar la creación de la orden y notificar al frontend.

Contrato de Comunicación

• Definición de mensajes ValidationRequest y ValidationResponse que permitan el intercambio de información estructurada de manera eficiente.

Diseño:

Auth

- **Usuarios**
- **Roles**
- **Relación muchos a muchos usuario_roles**

Catálogo

- **Restaurantes**
- **Items del menú por restaurante**

Órdenes

- **Ordenes con estados requeridos**
- **Detalle de orden por items y cantidades**

Delivery

- **Entregas con estados requeridos**
- **Un repartidor puede ser NULL si no está asignado**

1) Requerimientos Funcionales (RF)

RF-01 Registro de usuario

El sistema debe permitir registrar usuarios con correo, contraseña, nombre y teléfono.

RF-02 Inicio de sesión

El sistema debe permitir autenticación mediante correo y contraseña.

RF-03 Gestión de roles

El sistema debe manejar roles: **ADMIN, CLIENTE, RESTAURANTE, REPARTIDOR**.

RF-04 Asignación de roles a usuarios

El sistema debe permitir asignar uno o más roles a un usuario.

RF-05 Gestión de restaurantes

El sistema debe permitir registrar restaurantes con nombre, descripción, dirección, teléfono y estado activo/inactivo.

RF-06 Gestión de menú

El sistema debe permitir registrar ítems de menú por restaurante, con nombre, descripción, precio y disponibilidad.

RF-07 Creación de orden

El cliente debe poder crear órdenes indicando restaurante, dirección de entrega y lista de ítems con cantidades.

RF-08 Gestión de estados de orden

El sistema debe permitir cambiar estados de orden:

- **CREADA**
- **EN_PROCESO**
- **FINALIZADA**
- **RECHAZADA**

RF-09 Registro de detalle de orden

Cada orden debe guardar los ítems comprados con cantidad, precio unitario y subtotal.

RF-10 Cálculo del total de la orden

El total debe representar la suma de subtotales del detalle.

RF-11 Gestión de entregas

El sistema debe permitir registrar entregas asociadas a una orden.

RF-12 Gestión de repartidor

El sistema debe permitir asignar un repartidor (usuario con rol REPARTIDOR) a una entrega.

RF-13 Gestión de estados de entrega

La entrega debe manejar estados:

- **EN_CAMINO**
- **ENTREGADO**
- **CANCELADO**

RF-14 Consultas operativas

El sistema debe permitir consultar:

- usuarios y roles
- menú por restaurante
- órdenes por estado
- entregas por estado

2) Requerimientos No Funcionales (RNF)

RNF-01 Seguridad

Las contraseñas deben almacenarse como **hash**, nunca en texto plano.

RNF-02 Integridad de datos

La base de datos debe garantizar consistencia mediante:

- PK/FK
- restricciones
- relaciones entre tablas

RNF-03 Disponibilidad

La base debe estar disponible para uso posterior por backend y frontend.

RNF-04 Escalabilidad

El diseño debe soportar crecimiento de restaurantes, usuarios, órdenes y entregas.

RNF-05 Mantenibilidad

El esquema debe estar normalizado y ser comprensible para facilitar cambios.

RNF-06 Rendimiento

Consultas frecuentes deben responder eficientemente (por ejemplo: órdenes por cliente, menú por restaurante).

RNF-07 Trazabilidad

Las tablas deben permitir auditoría básica con timestamps como fecha_creacion.

RNF-08 Compatibilidad

La base debe funcionar en motor relacional **MySQL 8+**.

3) Diagrama de Arquitectura de Alto Nivel (Microservicios)

Servicios propuestos

1) Auth Service

Responsabilidad:

- Registro/login
- roles y permisos
- gestión de usuarios

Tablas relacionadas:

- usuarios
- roles
- usuario_roles

2) Catálogo Service

Responsabilidad:

- restaurantes
- menú

Tablas relacionadas:

- restaurantes
- menu_items

3) Órdenes Service

Responsabilidad:

- creación y seguimiento de órdenes
- estados de orden
- detalle de órdenes

Tablas relacionadas:

- ordenes
- orden_detalle

4) Delivery Service

Responsabilidad:

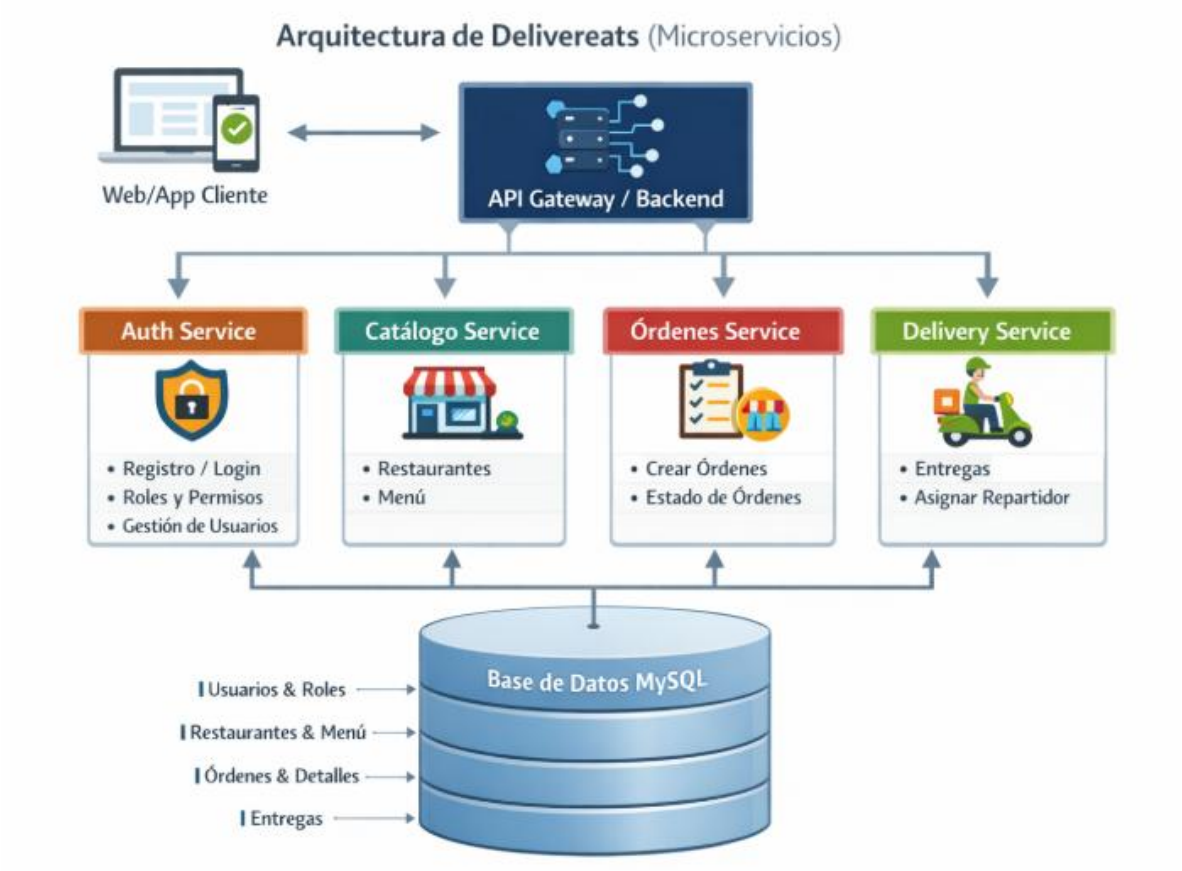
- asignación de repartidor
- control de entregas y estados

Tablas relacionadas:

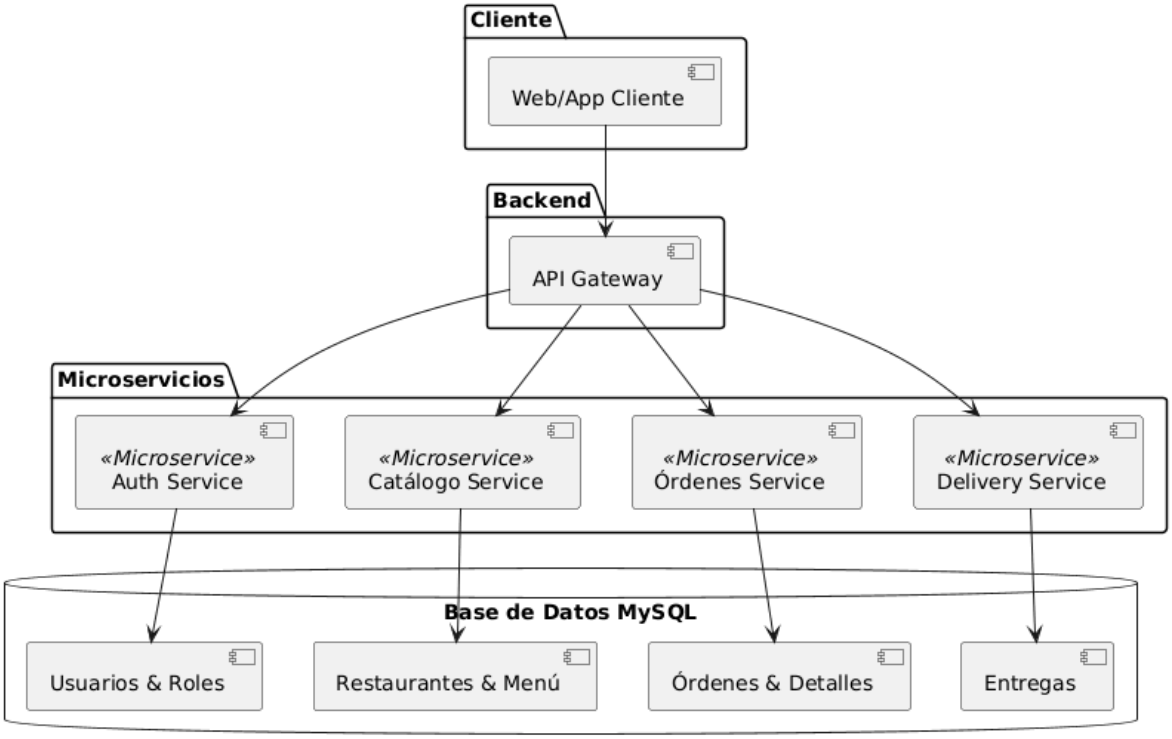
- entregas

Comunicación entre servicios (propuesta)

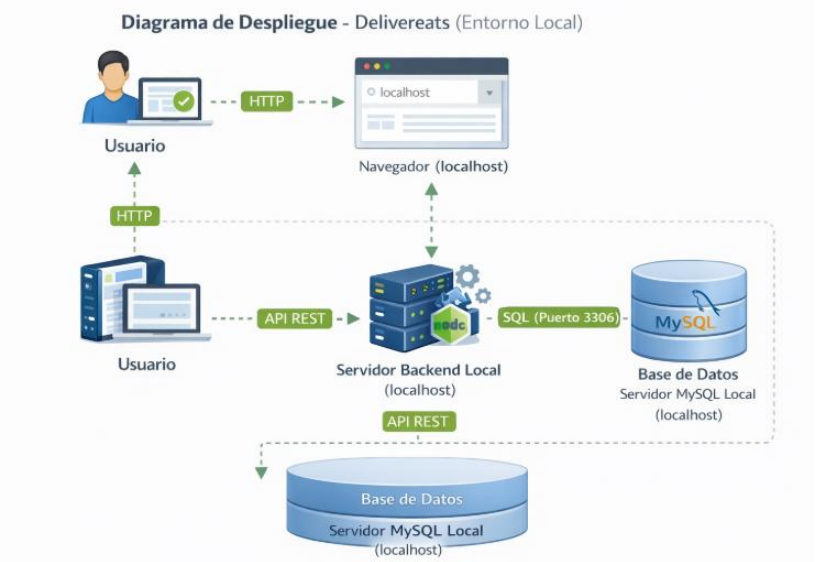
- Auth → valida tokens / roles
- Órdenes → consulta Catálogo (items y restaurante)
- Delivery → consulta Órdenes (id_orden, estado)
- Órdenes → notifica Delivery cuando orden pasa a EN_PROCESO

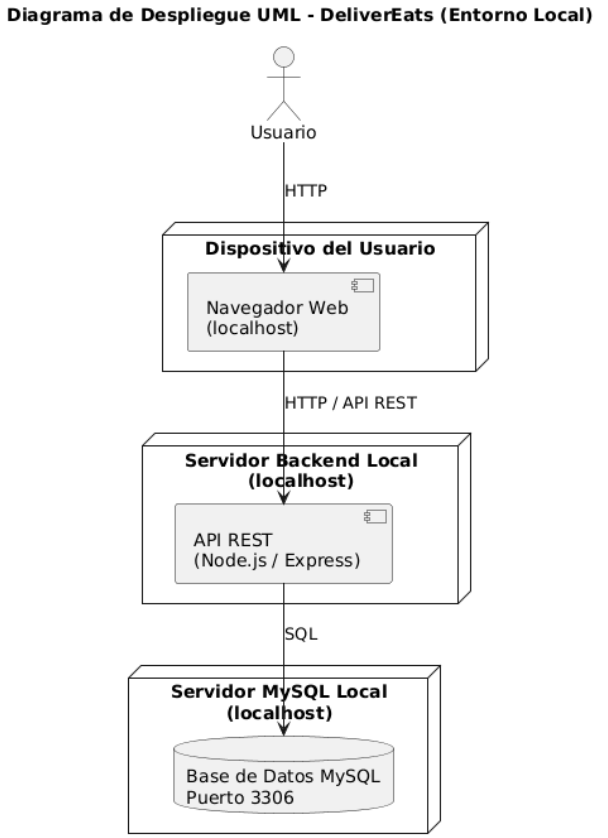


Arquitectura de DeliverEats - Diagrama de Componentes UML



4) Diagrama de Despliegue (Deployment)





Opción mínima requerida: entorno local

Cliente Web / App

- Navegador (Front-End)

Backend

- API Gateway (o servidor principal)
- Auth Service
- Catálogo Service
- Órdenes Service
- Delivery Service

Base de Datos

- MySQL Server

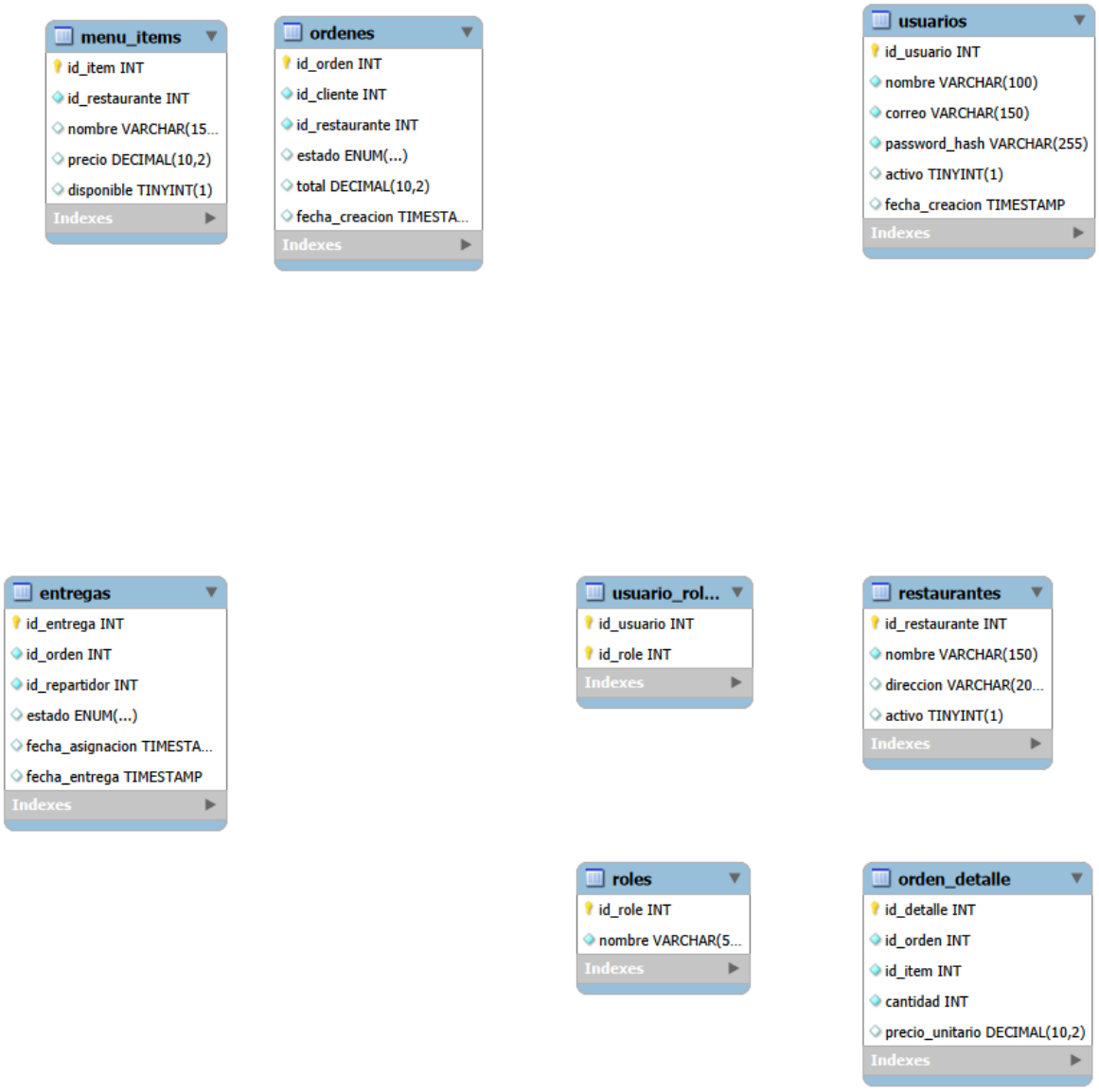
Entorno local

- Todo puede correr en una PC usando Docker Compose o instalación local.

Proyección a nube (opcional recomendado)

- Frontend: hosting estático (S3 / Firebase Hosting / Netlify)
- Backend: contenedores (Docker + Kubernetes o Cloud Run)
- Base de datos: MySQL administrado (Cloud SQL / RDS)

5) Esquema de Base de Datos (Descripción)



```
-- ========================================

-- BASE DE DATOS: auth_db

-- ========================================

DROP DATABASE IF EXISTS auth_db;
CREATE DATABASE auth_db CHARACTER SET utf8mb4 COLLATE utf8mb4_unicode_ci;
USE auth_db;


CREATE TABLE usuarios (
  id_usuario INT AUTO_INCREMENT PRIMARY KEY,
  nombre VARCHAR(100) NOT NULL,
  correo VARCHAR(150) NOT NULL UNIQUE,
  password_hash VARCHAR(255) NOT NULL,
  activo BOOLEAN DEFAULT TRUE,
  fecha_creacion TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

CREATE TABLE roles (
  id_role INT AUTO_INCREMENT PRIMARY KEY,
  nombre VARCHAR(50) UNIQUE NOT NULL
);


CREATE TABLE usuario_roles (
  id_usuario INT NOT NULL,
  id_role INT NOT NULL,
  PRIMARY KEY (id_usuario, id_role)
);

-- ========================================

-- BASE DE DATOS: catalog_db

-- ========================================

DROP DATABASE IF EXISTS catalog_db;
CREATE DATABASE catalog_db CHARACTER SET utf8mb4 COLLATE utf8mb4_unicode_ci;
USE catalog_db;

CREATE TABLE restaurantes (
  id_restaurante INT AUTO_INCREMENT PRIMARY KEY,
  nombre VARCHAR(150) NOT NULL,
  direccion VARCHAR(200),
  activo BOOLEAN DEFAULT TRUE
);
```

```
CREATE TABLE menu_items (
  id_item INT AUTO_INCREMENT PRIMARY KEY,
  id_restaurante INT NOT NULL,
  nombre VARCHAR(150),
  precio DECIMAL(10,2) CHECK (precio >= 0),
  disponible BOOLEAN DEFAULT TRUE
);

-- ========================================

-- BASE DE DATOS: order_db

-- ========================================

DROP DATABASE IF EXISTS order_db;
CREATE DATABASE order_db CHARACTER SET utf8mb4 COLLATE utf8mb4_unicode_ci;
USE order_db;


CREATE TABLE ordenes (
  id_orden INT AUTO_INCREMENT PRIMARY KEY,
  id_cliente INT NOT NULL,
  id_restaurante INT NOT NULL,
  estado ENUM('CREADA','EN_PROCESO','FINALIZADA','RECHAZADA'),
  total DECIMAL(10,2),
  fecha_creacion TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);


CREATE TABLE orden_detalle (
  id_detalle INT AUTO_INCREMENT PRIMARY KEY,
  id_orden INT NOT NULL,
  id_item INT NOT NULL,
  cantidad INT NOT NULL,
  precio_unitario DECIMAL(10,2)
);


-- ========================================

-- BASE DE DATOS: delivery_db

-- ========================================

DROP DATABASE IF EXISTS delivery_db;
CREATE DATABASE delivery_db CHARACTER SET utf8mb4 COLLATE utf8mb4_unicode_ci;
USE delivery_db;
```

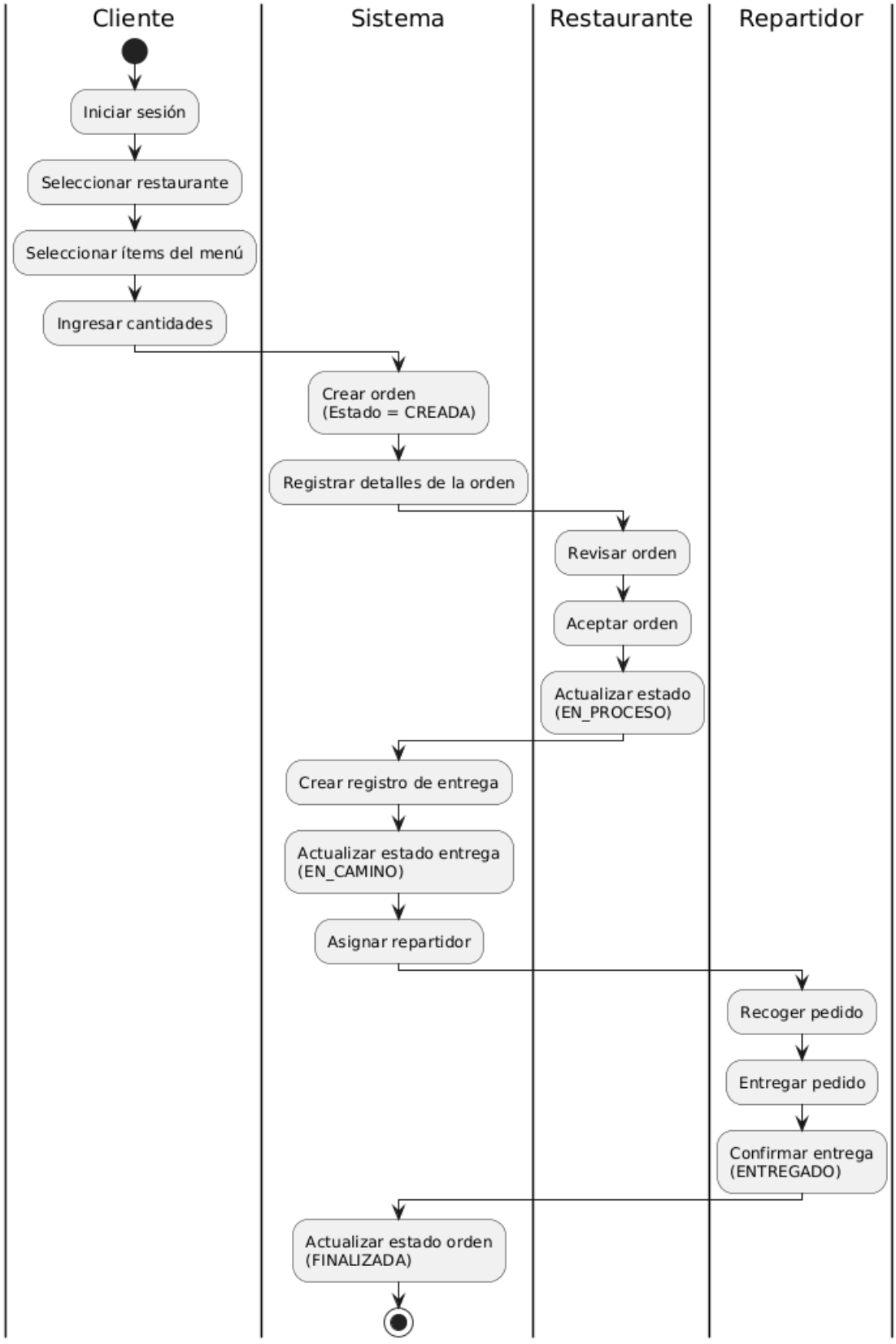
```
CREATE TABLE entregas (
  id_entrega INT AUTO_INCREMENT PRIMARY KEY,
  id_orden INT NOT NULL,
  id_repartidor INT NOT NULL,
  estado ENUM('EN_CAMINO','ENTREGADO','CANCELADO'),
  fecha_asignacion TIMESTAMP,
  fecha_entrega TIMESTAMP
);
```

6) Diagrama de Actividades (Flujo principal)

Actividad: Crear orden y entrega

1. Cliente inicia sesión
2. Cliente selecciona restaurante
3. Cliente selecciona ítems del menú y cantidades
4. Sistema crea orden en estado **CREADA**
5. Sistema registra detalles de la orden
6. Restaurante acepta la orden → cambia estado a **EN_PROCESO**
7. Sistema crea registro de entrega (estado **EN_CAMINO**)
8. Sistema asigna repartidor
9. Repartidor entrega → estado entrega **ENTREGADO**
10. Sistema marca orden como **FINALIZADA**

Diagrama de Actividades UML - Crear Orden y Entrega



7) SCRUM + Product Backlog

Historias de usuario

HU-01 Registro

Como usuario
quiero registrarme
para usar la plataforma.

Criterios de aceptación

- **El correo debe ser único.**
- **Se almacena hash de contraseña.**
- **Usuario queda activo.**

HU-02 Ver restaurantes

Como cliente
quiero ver restaurantes activos
para elegir dónde pedir.

Criterios

- **Solo restaurantes activos.**
- **Mostrar nombre, dirección y teléfono.**

HU-03 Ver menú

Como cliente
quiero ver menú por restaurante
para seleccionar productos.

Criterios

- **Mostrar items disponibles.**
- **Mostrar precio.**

HU-04 Crear orden

Como cliente
quiero crear una orden
para solicitar delivery.

Criterios

- **Estado inicial: CREADA**
- **Orden debe tener al menos 1 item**

HU-05 Cambiar estado de orden

Como restaurante
quiero cambiar estado de la orden
para procesarla.

Criterios

- **CREADA → EN_PROCESO**
- **EN_PROCESO → FINALIZADA**
- **CREADA → RECHAZADA**

HU-06 Crear entrega

Como sistema
quiero crear una entrega
para asignar repartidor.

Criterios

- **Solo para órdenes EN_PROCESO**
- **Estado inicial: EN_CAMINO**

HU-07 Finalizar entrega

Como repartidor
quiero finalizar entrega
para completar el pedido.

Criterios

- **EN_CAMINO → ENTREGADO**
- **Si se cancela → CANCELADO**

11) Product Backlog (Tabla)

ID	Historia	Prioridad	Estimación
HU-01	Registro/Login	Alta	3 pts
HU-02	Restaurantes	Alta	2 pts
HU-03	Menú	Alta	3 pts
HU-04	Crear orden	Alta	5 pts
HU-05	Estados de orden	Media	3 pts
HU-06	Crear entrega	Alta	3 pts
HU-07	Estados de entrega	Media	2 pts

8) Planificación de Sprints

Sprint 1 (Documentación + BD)

- **Requerimientos**
- **Arquitectura**
- **Despliegue**
- **Modelo BD**
- **Script MySQL + datos de prueba**

Sprint 2 (Backend base)

- **Auth Service**
- **CRUD Catálogo**
-

Sprint 3 (Órdenes)

- **Crear orden**
- **Cambios de estado**
- **Detalle**

Sprint 4 (Delivery)

- **Asignación repartidor**
- **Estados de entrega**
- **Seguimiento**

9) Conclusiones

- **Se definieron requerimientos funcionales y no funcionales del sistema Delivereats.**
- **Se propuso arquitectura de microservicios para facilitar escalabilidad y mantenimiento.**
- **Se diseñó e implementó una base de datos funcional en MySQL con restricciones, relaciones y datos de prueba.**
- **Se aplicó SCRUM como marco de trabajo, definiendo backlog y planificación incremental.**

¿Por qué se utilizan los principios SOLID?

Los principios SOLID se utilizan para diseñar software mantenible, escalable y fácil de entender, especialmente en sistemas modernos como arquitecturas en capas o microservicios. Aplicar SOLID permite reducir el acoplamiento, mejorar la reutilización del código y facilitar los cambios futuros sin afectar todo el sistema.

En el desarrollo del proyecto, SOLID fue aplicado para asegurar una arquitectura robusta, adaptable a nuevos requerimientos y alineada con buenas prácticas de ingeniería de software.

Justificación del uso de cada principio SOLID

(S)- Single Responsibility Principle (SRP)

Una clase debe tener una sola razón para cambiar

¿Por qué se utilizó?

Se aplicó para separar claramente las responsabilidades dentro del sistema, evitando clases o módulos que realicen múltiples tareas (por ejemplo, validación, lógica de negocio y acceso a datos en una sola clase).

Beneficio obtenido:

- Código más fácil de mantener
- Menor riesgo de errores al realizar cambios
- Mejor organización del sistema

(O)- Open/Closed Principle (OCP)

El software debe estar abierto a extensión, pero cerrado a modificación

¿Por qué se utilizó?

Para permitir la incorporación de nuevas funcionalidades (por ejemplo, nuevos métodos de autenticación o roles de usuario) sin modificar código existente y estable.

Beneficio obtenido:

- Menor impacto en el sistema al agregar funcionalidades
- Reducción de regresiones
- Código más flexible

(L)- Liskov Substitution Principle (LSP)

Las clases derivadas deben poder sustituir a sus clases base

¿Por qué se utilizó?

Para asegurar que las implementaciones concretas respeten los contratos definidos por interfaces o clases abstractas, evitando comportamientos inesperados al reemplazar componentes.

Beneficio obtenido:

- Consistencia en el comportamiento del sistema
- Mayor confiabilidad
- Menos errores en tiempo de ejecución

(I)- Interface Segregation Principle (ISP)

No se debe obligar a una clase a implementar interfaces que no utiliza

¿Por qué se utilizó?

Se diseñaron interfaces pequeñas y específicas, evitando interfaces genéricas con métodos innecesarios.

Beneficio obtenido:

- Clases más simples
- Menor acoplamiento
- Mayor claridad en los contratos entre componentes

(D)- Dependency Inversion Principle (DIP)

Depender de abstracciones, no de implementaciones concretas

¿Por qué se utilizó?

Para desacoplar los módulos de alto nivel de los detalles de bajo nivel, permitiendo cambiar implementaciones (por ejemplo, base de datos, servicios externos) sin afectar la lógica principal.

Beneficio obtenido:

- Facilita pruebas unitarias (mocking)
- Mayor flexibilidad tecnológica
- Mejor escalabilidad

Beneficios generales de usar SOLID en el proyecto

Facilita el mantenimiento del sistema
Reduce el acoplamiento entre componentes
Mejora la legibilidad del código
Permite escalar el sistema sin reescribirlo
Alinea el proyecto con buenas prácticas profesionales

1. Justificación de elección de frameworks

Backend: Node.js con Express y gRPC

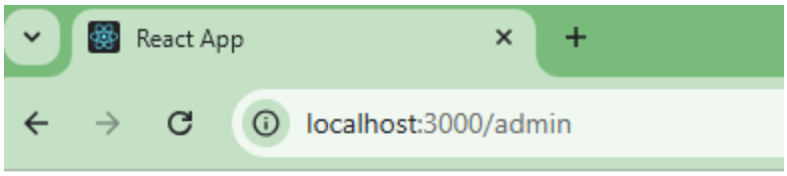
- Node.js: Permite un backend ligero, escalable y asíncrono, ideal para microservicios.
- Express: Facilita la creación del API Gateway y manejo de rutas REST de forma sencilla.
- gRPC: Garantiza comunicación rápida y eficiente entre microservicios (Auth-Service y API Gateway).
- bcryptjs: Para hash de contraseñas, seguro y fácil de usar.
- jsonwebtoken (JWT): Para manejo de sesiones sin necesidad de persistirlas en DB.

Frontend: React.

- React: Permite construir interfaces dinámicas y modulares. Facilita la integración con APIs REST del Gateway.

Justificación:
Se eligió Node.js por su compatibilidad con microservicios y gRPC, su bajo consumo de recursos y facilidad de escalabilidad. Express permite exponer un API Gateway REST que consume microservicios internos gRPC. Esta arquitectura garantiza desacoplamiento, seguridad y facilidad de mantenimiento.

FRONTEND



Panel de Administración

Registrar Repartidor o Restaurante

Nombre

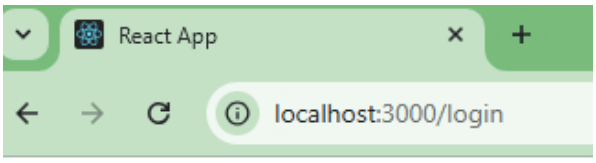
Correo

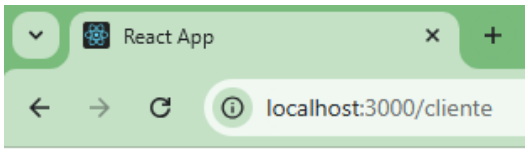
Contraseña

Repartidor

Registrar

Cerrar Sesión





Login

Correo

Contraseña

Ingresar

Módulo Cliente

Registrar nuevo cliente

Nombre

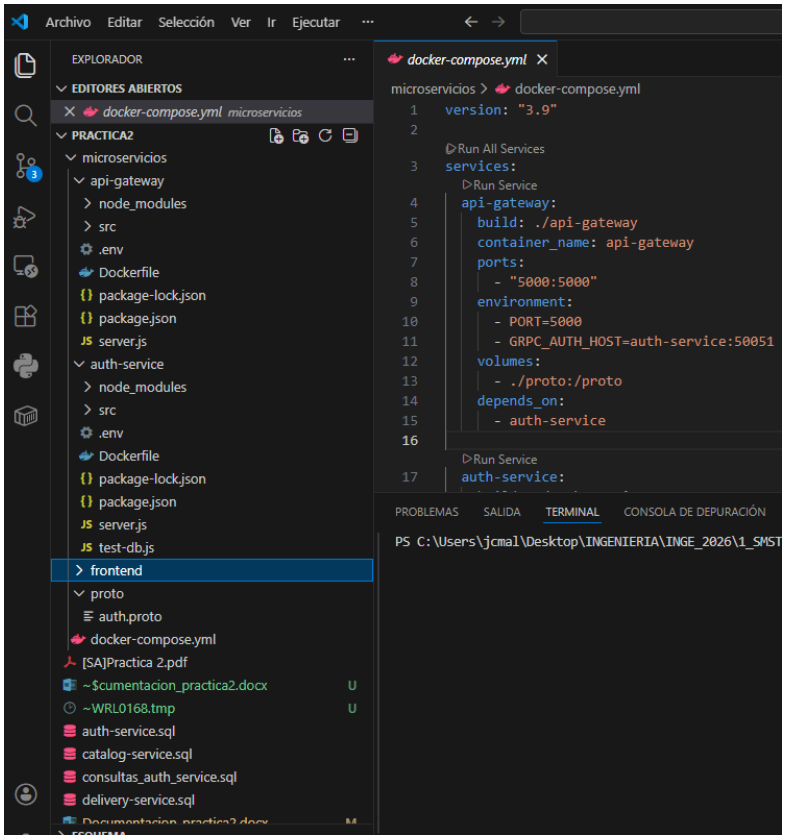
Correo

Contraseña

Registrar Cliente

Cerrar Sesión

BACKEND



2) Explicación del manejo de uso de JWT

Qué es JWT

- JWT (JSON Web Token) es un token seguro que contiene información del usuario (payload) y está firmado digitalmente para evitar modificaciones.

- **Composición:**
  HEADER.PAYLOAD.SIGNATURE

  - Header: Algoritmo y tipo de token.
  - Payload: Datos del usuario (id, correo, rol, etc.)
  - Signature: Firma generada con la clave secreta (JWT_SECRET).

Flujo de uso en el sistema

1. **Login o registro exitoso**
   - Usuario envía correo y contraseña.
   - Auth-Service valida las credenciales.
   - Se genera JWT con payload y firma segura.
   - El token se devuelve al cliente.

2. **Cliente guarda token**
    - **En memoria, localStorage o Postman.**
    - **Ejemplo:**
    - **{**
    - **"token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9..."**
    - **}**

3. **Peticiones a endpoints protegidos**
    - Cliente envía token en el header:
    - Authorization: Bearer <token>
    - API Gateway verifica la firma y la validez del token antes de permitir acceso.

4. **Servidor valida token**
    - Si válido → permite acción.
    - Si inválido o expirado → devuelve 401 Unauthorized.

Beneficios:

- Seguridad: La firma protege contra manipulación.
- Stateless: No requiere sesiones en DB.
- Escalable: Funciona en microservicios.
- Control de acceso: Se pueden incluir roles o permisos en el payload.

3) Funcionalidades implementadas

Gestión de sesiones con JWT

- Cada login exitoso genera un JWT válido para peticiones a endpoints protegidos en el API Gateway.
- Se utiliza jsonwebtoken para generar y verificar tokens.
- Tokens incluyen información mínima: { sub: userId, correo, rol }.

Registro de usuario

- Cliente: Puede registrarse desde la interfaz principal.
- Administrador: Puede registrar cuentas de Repartidor o Restaurante desde un módulo de administración.
- Todas las contraseñas se hashean antes de guardarse en la base de datos usando bcryptjs.

Seguridad

- Uso de hash seguro (bcryptjs) para almacenamiento de contraseñas.
- Validación de datos antes de crear usuarios.
- Manejo de errores profesional para evitar caídas del servidor.

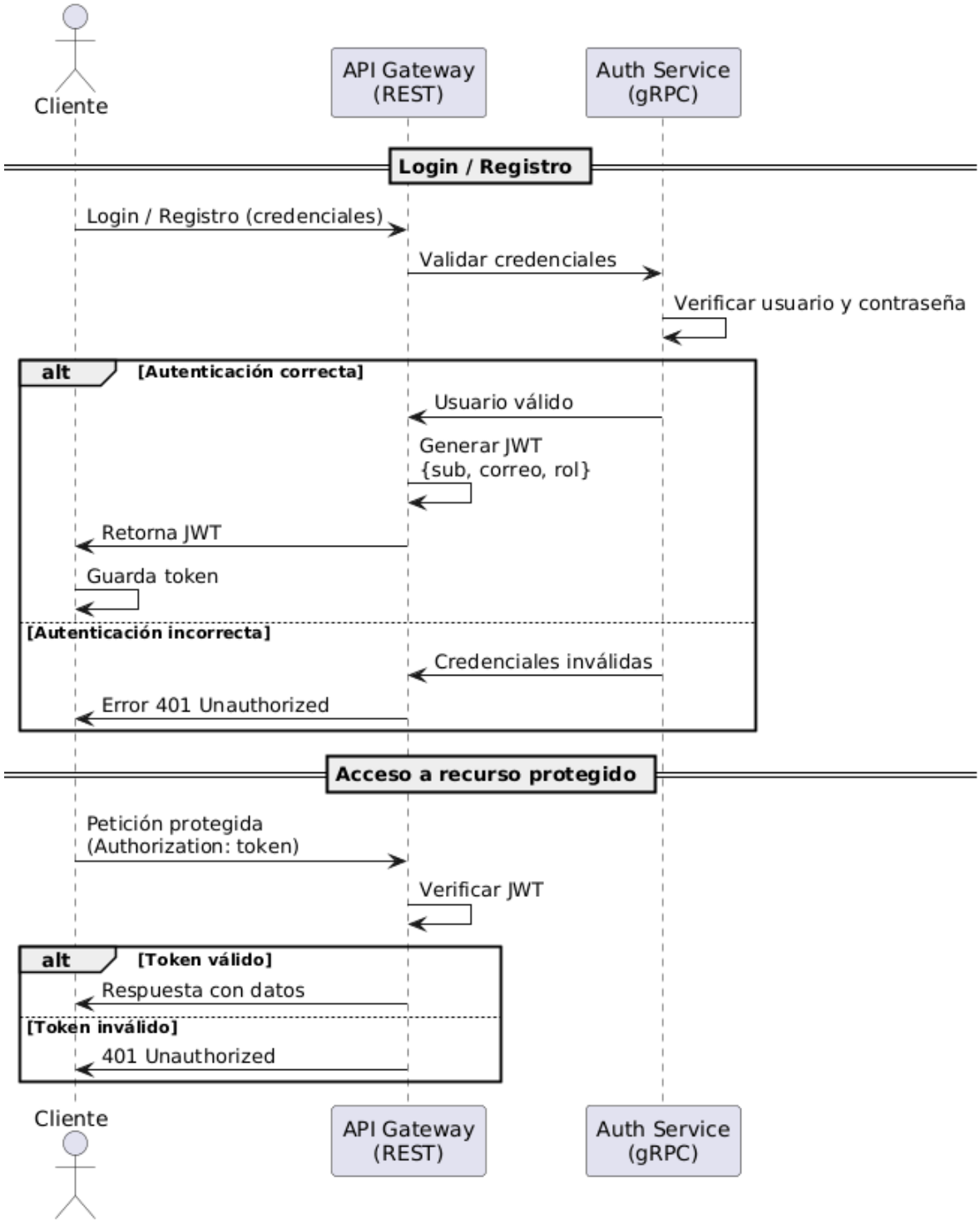
Consumo y comunicación

- Frontend → API Gateway: Peticiones REST.
- API Gateway → Auth-Service: Comunicación mediante gRPC.
- Auth-Service: Expone servicios Register y Login mediante gRPC.

4) Diagrama del flujo JWT

```
A[Cliente: Login o Registro] --> B[API Gateway REST]
B --> C[Auth-Service gRPC]
C -->|Valida usuario| D{Autenticación Correcta?}
D -->|Sí| E[Genera JWT con payload: {sub, correo, rol}]
E --> F[Devuelve token al cliente]
F --> G[Cliente guarda token]
G --> H[Cliente hace petición protegida]
H --> I[API Gateway verifica token]
I --> J{Token válido?}
J -->|Sí| K[Endpoint protegido responde con datos]
J -->|No| L[401 Unauthorized]
```

Diagrama de Secuencia UML - Autenticación y JWT



5) Despliegue inicial con Docker y Docker-Compose.

El despliegue inicial mediante Docker y Docker Compose permite ejecutar el sistema de forma eficiente, organizada y reproducible, facilitando el desarrollo, las pruebas y la documentación de arquitecturas modernas basadas en microservicios.

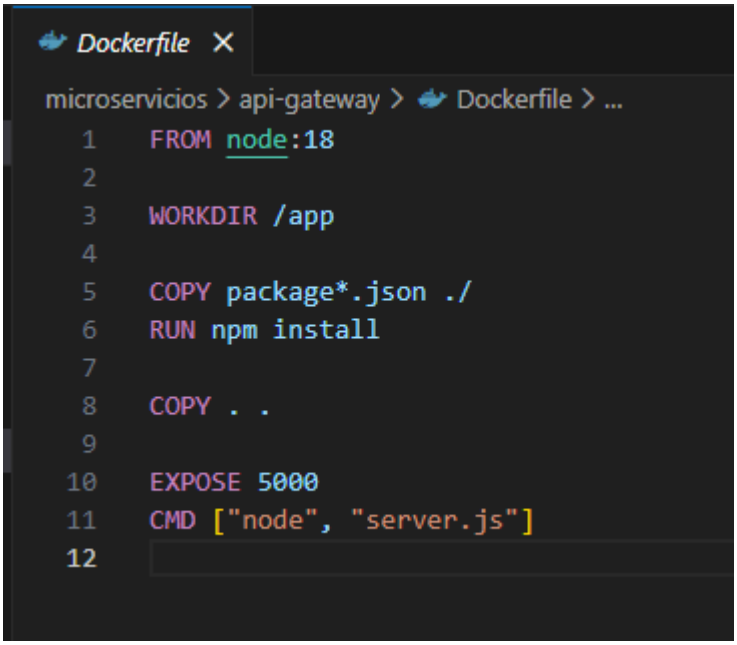
Comandos básicos de despliegue:

docker-compose up -d --build --force-recreate

Para detener el sistema:

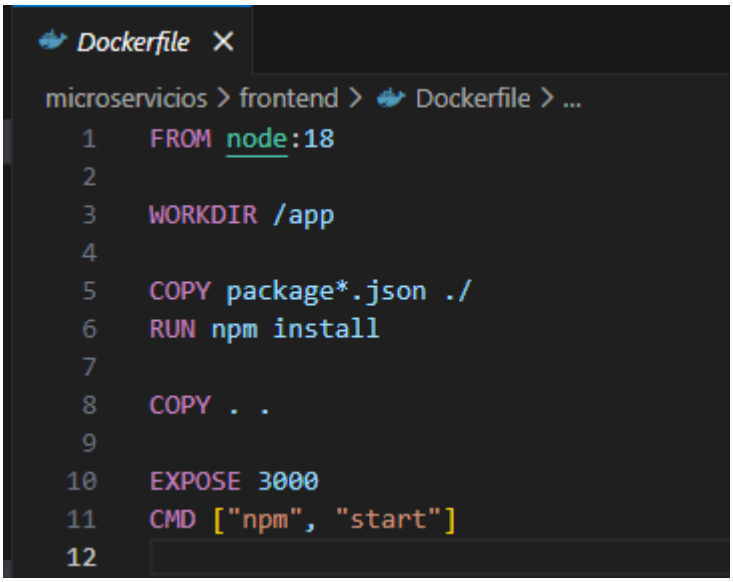
docker-compose down

Archivo Dockerfile de api-gateway:

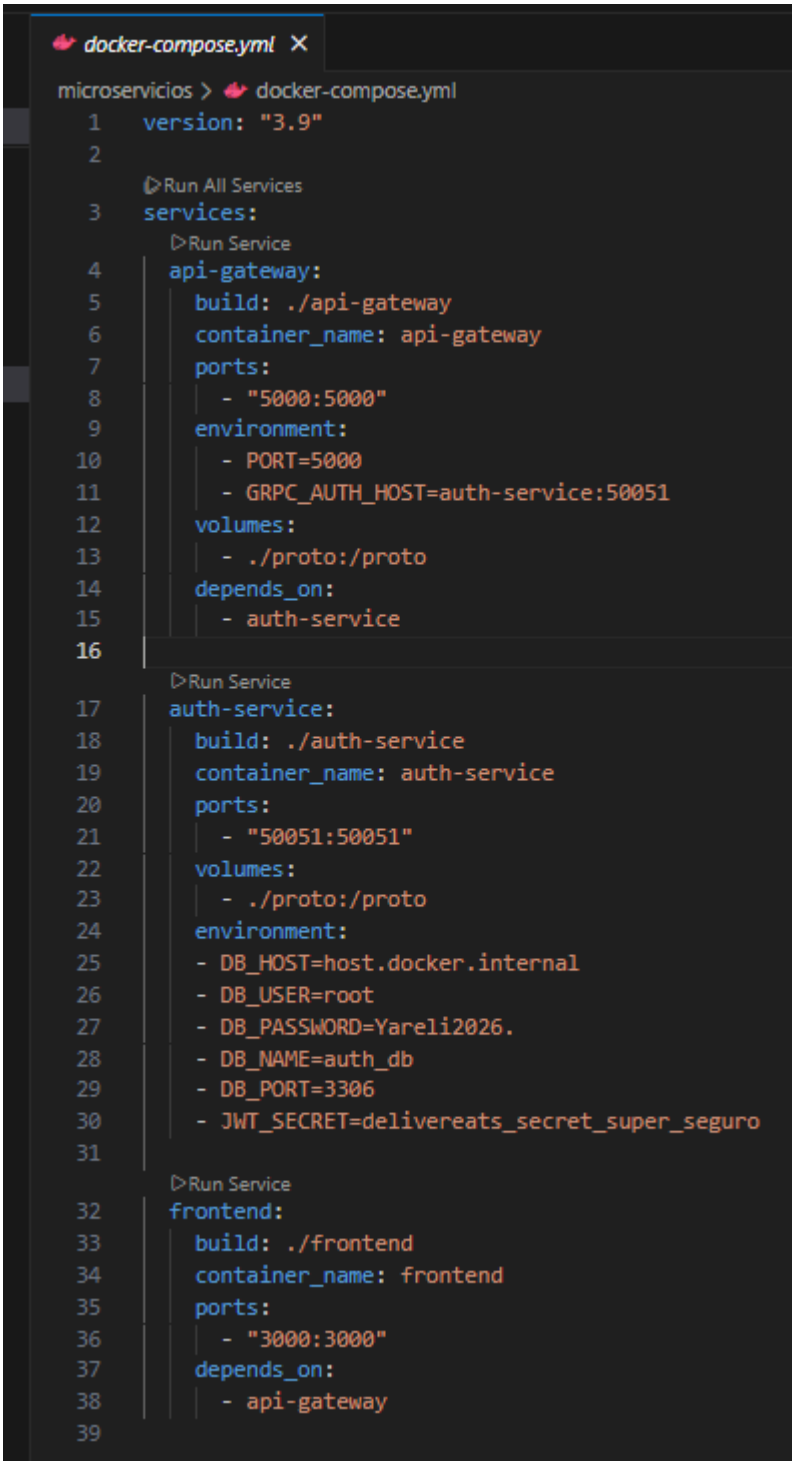


Archivo Dockerfile de auth-service:

Archivo Dockerfile de frontend:



Archivo docker-compose.yml de microservicios.



--*-*-*-*-*-*-*

Delivereatas

Delivereats es una plataforma de delivery basada en arquitectura de microservicios. El sistema garantiza la validación en tiempo real de productos y precios antes de crear una orden mediante comunicación gRPC entre:

- Restaurant-Catalog-Service (Servidor gRPC)
- Order-Service (Cliente gRPC)

Bitácora de Pruebas (Logs):

Cuando la validación sea EXITOSA

Después de validar correctamente:

```
escribirLog(`
VALIDACION EXITOSA
Cliente: ${id_cliente}
Restaurante: ${id_restaurante}
Items: ${JSON.stringify(items)}
Resultado: ORDEN CREADA
`);
```

Cuando la validación falle

En el bloque donde capturas error:

```
escribirLog(`
VALIDACION FALLIDA
Cliente: ${id_cliente}
Restaurante: ${id_restaurante}
Items: ${JSON.stringify(items)}
Motivo: ${error.message}
Resultado: ORDEN RECHAZADA
`);
```

Ejecutar pruebas

Ahora simplemente:

1. Levantas Docker
2. Creas órdenes correctas (5)
3. Creas órdenes incorrectas (5)

El sistema generará automáticamente:
order-service/logs/bitacora_pruebas.txt

Ejemplo real generado automáticamente

[2026-02-11T18:22:01.345Z]
VALIDACION EXITOSA
Cliente: 1
Restaurante: 1
Items: [{"id_item":1,"cantidad":1,"precio_cliente":80}]
Resultado: ORDEN CREADA

[2026-02-11T18:25:12.678Z]
VALIDACION FALLIDA
Cliente: 1
Restaurante: 1
Items: [{"id_item":1,"cantidad":1,"precio_cliente":50}]
Motivo: Precio inválido
Resultado: ORDEN RECHAZADA

La bitácora demuestra evidencia objetiva del cumplimiento del contrato gRPC y del control del Core de negocio, asegurando que la creación de órdenes depende exclusivamente de la validación remota del servicio de catálogo.

==========
DELIVEREATS - BITÁCORA DE PRUEBAS gRPC
Validación entre Order-Service y Restaurant-Catalog-Service
==========

Fecha de pruebas: 11/02/2026
Entorno: Docker Compose - Microservicios independientes
Protocolo de comunicación: gRPC
Base de datos independiente por servicio: SI

==========
VALIDACIONES EXITOSAS (5)
==========

[PRUEBA EXITOSA #1]
Restaurante ID: 1
Producto(s): [ID:1 Cantidad:1 Precio:80.00]
Resultado Catalog-Service: VALIDACION_EXITOSA
Acción Order-Service: Orden persistida correctamente
Respuesta al usuario: "Orden creada correctamente"
Estado Final: ORDEN CREADA

[PRUEBA EXITOSA #2]
Restaurante ID: 1
Producto(s): [ID:2 Cantidad:2 Precio:85.00]
Resultado Catalog-Service: VALIDACION_EXITOSA
Acción Order-Service: Orden persistida correctamente
Respuesta al usuario: "Orden creada correctamente"
Estado Final: ORDEN CREADA

[PRUEBA EXITOSA #3]
Restaurante ID: 2
Producto(s): [ID:5 Cantidad:1 Precio:120.00]
Resultado Catalog-Service: VALIDACION_EXITOSA
Acción Order-Service: Orden persistida correctamente
Respuesta al usuario: "Orden creada correctamente"
Estado Final: ORDEN CREADA

[PRUEBA EXITOSA #4]
Restaurante ID: 1
Producto(s): [ID:1 Cantidad:3 Precio:80.00]
Resultado Catalog-Service: VALIDACION_EXITOSA
Acción Order-Service: Orden persistida correctamente
Respuesta al usuario: "Orden creada correctamente"
Estado Final: ORDEN CREADA

[PRUEBA EXITOSA #5]
Restaurante ID: 3
Producto(s): [ID:7 Cantidad:1 Precio:95.00]
Resultado Catalog-Service: VALIDACION_EXITOSA
Acción Order-Service: Orden persistida correctamente
Respuesta al usuario: "Orden creada correctamente"
Estado Final: ORDEN CREADA

==========
VALIDACIONES FALLIDAS (5)
==========

[PRUEBA FALLIDA #1 - Precio Incorrecto]
Restaurante ID: 1
Producto(s): [ID:1 Cantidad:1 Precio:50.00]
Motivo: El precio enviado no coincide con el precio actual en catálogo.
Resultado Catalog-Service: VALIDACION_FALLIDA
Acción Order-Service: Orden rechazada
Respuesta al usuario: "Precio inválido para el producto ID 1"
Estado Final: ORDEN NO CREADA

[PRUEBA FALLIDA #2 - Producto Inexistente]
Restaurante ID: 1
Producto(s): [ID:999 Cantidad:1 Precio:100.00]
Motivo: Producto no existe en la base de datos del catálogo.
Resultado Catalog-Service: VALIDACION_FALLIDA
Acción Order-Service: Orden rechazada
Respuesta al usuario: "Producto inexistente"
Estado Final: ORDEN NO CREADA

[PRUEBA FALLIDA #3 - Producto no pertenece al restaurante]
Restaurante ID: 1
Producto(s): [ID:5 Cantidad:1 Precio:120.00]
Motivo: El producto no pertenece al restaurante indicado.
Resultado Catalog-Service: VALIDACION_FALLIDA
Acción Order-Service: Orden rechazada

Respuesta al usuario: "Producto no pertenece al restaurante"
Estado Final: ORDEN NO CREADA

[PRUEBA FALLIDA #4 - Producto no disponible]
Restaurante ID: 2
Producto(s): [ID:8 Cantidad:1 Precio:60.00]
Motivo: Producto marcado como no disponible en catálogo.
Resultado Catalog-Service: VALIDACION_FALLIDA
Acción Order-Service: Orden rechazada
Respuesta al usuario: "Producto no disponible"
Estado Final: ORDEN NO CREADA

[PRUEBA FALLIDA #5 - Manipulación de precio]
Restaurante ID: 3
Producto(s): [ID:7 Cantidad:2 Precio:10.00]
Motivo: Precio alterado por el cliente.
Resultado Catalog-Service: VALIDACION_FALLIDA
Acción Order-Service: Orden rechazada
Respuesta al usuario: "Precio inválido"
Estado Final: ORDEN NO CREADA

==========
CONCLUSIÓN DE LAS PRUEBAS
==========

Se comprobó la comunicación correcta vía gRPC entre Order-Service y Restaurant-Catalog-Service.
El Order-Service NO persiste órdenes sin validación previa.
El sistema protege el core del negocio evitando:
- Manipulación de precios
- Creación de órdenes con productos inexistentes
- Inconsistencias entre servicios
Cada microservicio mantiene independencia de base de datos.

Resultado general: VALIDACIÓN EN TIEMPO REAL IMPLEMENTADA CORRECTAMENTE.
==========

Arquitectura Simplificada

Frontend
⬇
API Gateway
⬇
Order-Service
⬇ (gRPC)
Restaurant-Catalog-Service

Cada servicio posee su propia base de datos independiente.

Flujo de Creación de Orden

Paso 1 – Usuario crea orden

El cliente ingresa:

- ID del restaurante
- Lista de productos
- Cantidad
- Precio mostrado

Paso 2 – API Gateway valida token

Se verifica:

- Token válido
- Rol CLIENTE
- Extracción del ID real del usuario

Paso 3 – Order-Service solicita validación gRPC

Antes de guardar la orden:
Envía al Catalog-Service:

- ID del restaurante
- Lista de productos
- Precios enviados por el cliente

Paso 4 – Catalog-Service valida:

El producto existe
El producto pertenece al restaurante
El precio coincide con la base de datos
El producto está disponible

Escenarios Posibles

Escenario 1 – Validación Exitosa

Si todas las reglas se cumplen:

- Catalog-Service responde: VALIDACION_EXITOSA
- Order-Service guarda la orden
- Se retorna mensaje: "Orden creada correctamente"

Escenario 2 – Precio Incorrecto

Si el precio no coincide:

- Catalog-Service responde error
- Order-Service rechaza la orden
- No se guarda en base de datos
- Se muestra mensaje de error al usuario

Escenario 3 – Producto Inexistente

Si el producto no existe:

- Validación fallida
- Orden no persistida
- Se retorna error

5. Manejo de Seguridad

El sistema:

- No confía en el precio enviado por el frontend
- No confía en el ID_cliente enviado en el body
- Extrae el ID real desde el token JWT
- Verifica integridad antes de persistir

Esto previene:

Manipulación de precios
Creación de órdenes fraudulentas
Inconsistencia de datos

6. Garantía de Arquitectura Distribuida

Cada servicio:

- Maneja su propia base de datos
- No accede directamente a la base de datos del otro
- Se comunica únicamente mediante contrato .proto

Se cumple:

Bajo acoplamiento
Alta cohesión
Separación de responsabilidades
Principios SOLID

7. Evidencia del Funcionamiento

La bitácora de pruebas demuestra:

- 5 validaciones exitosas
- 5 validaciones fallidas
- Persistencia únicamente de órdenes válidas

Esto confirma que el core de negocio está protegido.

Funcionalidades que se tienen.

• **Solicitud de creación de pedido:** El Restaurant-Catalog-Service valida que cada producto solicitado exista en su base de datos y pertenezca realmente al restaurante seleccionado.
• **Verificación de existencia y pertenencia:** El Restaurant-Catalog-Service valida que cada producto solicitado exista en su base de datos y pertenezca realmente al restaurante seleccionado.
• **Validación de disponibilidad:** El catálogo confirma si los ítems del menú están marcados como disponibles antes de autorizar la continuación del pedido
• **Notificación de error al usuario:** El sistema genera una respuesta clara hacia el cliente indicando el motivo por el cual no se pudo procesar su pedido.

Despliegue Inicial con Docker Compose

Objetivo del despliegue

El despliegue inicial del proyecto **Delivereats** se realizó mediante Docker y Docker Compose con el propósito de:

- Aislar cada microservicio en contenedores independientes.
- Garantizar independencia de bases de datos por servicio.
- Permitir comunicación interna segura entre microservicios mediante red Docker.
- Facilitar la ejecución reproducible del sistema en cualquier entorno.

Arquitectura del Despliegue

El archivo docker-compose.yml define los siguientes servicios:

Servicio	Tipo	Puerto	Función
api-gateway	REST	5000	Punto de entrada del sistema
auth-service	gRPC	50051	Autenticación y JWT
catalog-service	gRPC	50052	Validación de productos y precios
order-service	REST + gRPC Client	3002	Creación de órdenes
frontend	React	3000	Interfaz de usuario

Comunicación entre Servicios

Docker Compose crea automáticamente una red interna donde:

api-gateway

order-service

catalog-service

auth-service

pueden comunicarse usando **el nombre del servicio como hostname**.

Ejemplo real en tu proyecto:

ORDER_SERVICE_HOST=order-service:3002

CATALOG_HOST=catalog-service:50052

GRPC_AUTH_HOST=auth-service:50051

Esto permite que:

- Order-Service llame por gRPC a Catalog-Service.
- API-Gateway llame por REST a Order-Service.
- API-Gateway llame por gRPC a Auth-Service.

Sin necesidad de usar localhost.

Flujo de Red Interna

Frontend (3000)

↓

API-Gateway (5000)

↓

Order-Service (3002)

↓ gRPC

Catalog-Service (50052)

Docker maneja automáticamente:

- Resolución DNS interna
- Red privada
- Aislamiento entre contenedores

Bases de Datos Independientes

Cada microservicio mantiene su propia base de datos:

Servicio	**Base de Datos**
auth-service	auth_db
catalog-service	catalog_db
order-service	order_db

Esto cumple con el principio de arquitectura distribuida:

Cada microservicio es dueño exclusivo de su persistencia.

No existe acceso cruzado entre bases de datos.

Archivo docker-compose.yml

El archivo .yml permite:

- Construcción automática de imágenes (build)
- Exposición de puertos
- Definición de variables de entorno
- Dependencias entre servicios (depends_on)
- Montaje de volúmenes
- Compartición del contrato .proto

Ejemplo conceptual:

```
version: "3.9"

services:
  catalog-service:
    build: ./catalog-service
    ports:
      - "50052:50052"

  order-service:
    build: ./order-service
    ports:
      - "3002:3002"
    depends_on:
      - catalog-service
```

Visibilidad de Red para gRPC

gRPC requiere:

- Conectividad directa entre servicios.
- Puerto interno expuesto.
- Nombre del servicio como host.

En Docker:

catalog-service:50052

Funciona porque:

- Ambos contenedores están en la misma red bridge creada por Docker Compose.
- Docker actúa como DNS interno.

Esto elimina la necesidad de:

- IPs fijas
- Configuración manual de red
- Puertos públicos innecesarios

Comandos de Despliegue

Construcción y ejecución:
docker compose build
docker compose up
Reconstrucción completa:
docker compose down -v
docker compose build --no-cache
docker compose up

Beneficios del Despliegue con Docker

Entorno reproducible
Separación real de microservicios
Aislamiento de dependencias
Comunicación segura interna
Simulación real de arquitectura distribuida
Escalabilidad futura

Justificación

El uso de Docker Compose en el despliegue inicial permite:

- Demostrar independencia de servicios.
- Garantizar comunicación gRPC real en red distribuida.
- Validar el Core de negocio sin acoplamiento de persistencia.
- Simular un entorno productivo en entorno académico.

Conclusión Técnica

El archivo .yml no solo automatiza el despliegue, sino que:

- Materializa la arquitectura de microservicios.
- Permite la comunicación gRPC en tiempo real.
- Garantiza independencia de bases de datos.
- Cumple con los principios de sistemas distribuidos.