Guatemala 01 de febrero del 2026.

Universidad De San Carlos De Guatemala.

Laboratorio software avanzado sección “A”.

Auxiliar: Juan Pablo Samayoa Ruiz.

FASE DOS

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.

El presente proyecto tiene como finalidad el diseño e implementación de una plataforma de gestión de pedidos y entregas para restaurantes, denominada **DELIVEREATS**. En la Fase 1, el sistema fue desarrollado bajo una arquitectura de microservicios básica con enfoque funcional, permitiendo la gestión de usuarios, restaurantes, menús, órdenes y entregas mediante operaciones CRUD y comunicación relativamente directa entre servicios.

Sin embargo, ante la necesidad de mejorar la escalabilidad, desacoplamiento, resiliencia y preparación para entornos productivos reales, se plantea la transición hacia una **Fase 2**, donde el sistema evoluciona hacia una arquitectura orientada a eventos (Event-Driven Architecture). Esta nueva fase incorpora patrones modernos como el **Patrón Outbox**, autenticación distribuida mediante **JWT**, integración con un **Message Broker**, automatización mediante **CI/CD**, y despliegue orquestado en **Kubernetes**.

Esta transición no representa un cambio funcional del sistema, sino una evolución estructural y arquitectónica que lo convierte en una solución más robusta, escalable y alineada con estándares actuales de desarrollo de software empresarial.

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.

Evolucionar la arquitectura del sistema DELIVEREATS desde un modelo tradicional de microservicios hacia una arquitectura distribuida orientada a eventos, incorporando mecanismos de seguridad, automatización y despliegue en la nube que garanticen escalabilidad, resiliencia y buenas prácticas DevOps.

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.

- Implementar comunicación asíncrona entre microservicios mediante un Message Broker.
- Incorporar el Patrón Outbox para garantizar consistencia en la publicación de eventos.
- Integrar autenticación basada en JWT para seguridad distribuida y sin estado.
- Automatizar el proceso de integración y despliegue continuo mediante un pipeline CI/CD.
- Orquestar el sistema utilizando Kubernetes para permitir escalabilidad horizontal y alta disponibilidad.
- Implementar estrategias de despliegue como Rolling Update y Rollback para minimizar riesgos en producción.
- Mejorar la observabilidad y control del sistema mediante monitoreo y validaciones de estado.

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

****

ACTUALIZACIÓN DE REQUERIMIENTOS FUNCIONALES

Los requerimientos funcionales deben reflejar:

- Introducción de mensajería asíncrona
- Preparación para infraestructura Kubernetes
- Validación de comunicación desacoplada
- Preparación para CI/CD
- Flujo JWT claro

RF-01 – Creación de Orden

El sistema debe permitir que un usuario autenticado cree una orden a través del Order-Service.

Cambio respecto a Fase 1:
Ahora la creación de orden debe generar un evento asíncrono.

RF-02 – Publicación de Evento de Orden

Al crearse una orden, el Order-Service debe publicar un mensaje en una cola de mensajería (RabbitMQ o Kafka).

El mensaje debe incluir como mínimo:

- **order_id**
- **restaurant_id**
- **total**
- **timestamp**

RF-03 – Consumo del Evento por Restaurant-Service
El Restaurant-Service debe estar suscrito a la cola correspondiente y debe consumir el mensaje publicado por Order-Service.
Para la Prueba de Concepto, el servicio debe:

- **Recibir el mensaje correctamente**
- **Deserializarlo**
- **Registrarlo en consola**

RF-04 – Despliegue en Clúster Kubernetes
El sistema debe poder ejecutarse en un clúster Kubernetes utilizando:

- **Deployments**
- **Services**
- **Ingress Controller**
- **ConfigMaps**
- **Secrets**

RF-05 – Autenticación mediante JWT
El sistema debe validar todas las solicitudes protegidas mediante JWT emitido por Auth-Service.
El token debe:

- **Estar firmado**
- **Incluir claims de usuario**
- **Ser validado por API Gateway o microservicio correspondiente**

RF-06 – Versionado del Proyecto
El sistema debe permitir versionar la entrega mediante la creación del tag:
v0.4.0

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+**.

ACTUALIZACIÓN DE REQUERIMIENTOS NO FUNCIONALES

(Fase 2 – Orientado a resiliencia y escalabilidad)

Aquí es donde realmente se demuestra evolución arquitectónica.

RNF-01 – Alta Disponibilidad

El sistema debe garantizar alta disponibilidad mediante:

- Orquestación en Kubernetes
- Reinicio automático de pods ante fallos

RNF-02 – Escalabilidad Horizontal

Cada microservicio debe poder escalar horizontalmente mediante réplicas en Kubernetes sin afectar la disponibilidad del sistema.

RNF-03 – Desacoplamiento

La comunicación entre Order-Service y Restaurant-Service debe realizarse mediante mensajería asíncrona para evitar dependencia directa.

Esto debe garantizar:

- Tolerancia a fallos
- Procesamiento diferido
- Resiliencia ante caídas parciales

RNF-04 – Tolerancia a Fallos

Si Restaurant-Service no está disponible, el mensaje debe permanecer en la cola hasta que pueda ser consumido.

RNF-05 – Seguridad de Configuración

Las credenciales sensibles deben almacenarse en:

- Kubernetes Secrets
- Variables de entorno seguras en CI/CD

No deben almacenarse en código fuente.

RNF-06 – Automatización de Despliegue

El sistema debe contar con un pipeline CI/CD que permita:

- Construcción automática
- Ejecución de pruebas
- Publicación de imagen
- Despliegue automático en Kubernetes

RNF-07 – Observabilidad Básica

Los microservicios deben registrar eventos importantes en consola, incluyendo:

- Publicación de mensajes
- Consumo de mensajes
- Errores de conexión

RNF-08 – Versionamiento Controlado

Cada incremento funcional debe estar asociado a un tag de versión en el repositorio.

Diferencia Clave con Fase 1

Fase 1	**Fase 2 (Actualización)**
Comunicación síncrona	Comunicación asíncrona
Docker Compose	Kubernetes
Sin escalado real	Escalado horizontal
Sin resiliencia	Tolerancia a fallos
Despliegue manual	CI/CD automatizado

Con esta actualización se demuestra:

Evolución arquitectónica
Pensamiento de sistemas distribuidos
Comprensión de resiliencia
Enfoque DevOps
Aplicación real de Scrum

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

Fase Uno

Actualización del Diagrama de Arquitectura de Alto Nivel

La arquitectura evoluciona hacia un modelo desacoplado basado en eventos.
Se introduce un broker de mensajería (RabbitMQ) que permite separar el flujo de creación de órdenes del procesamiento por parte del restaurante.

El Order-Service ya no depende directamente del Restaurant-Service, sino que publica eventos en una cola.
El Restaurant-Service consume dichos eventos cuando se encuentra disponible.

Esta modificación mejora:

- Tolerancia a fallos
- Escalabilidad
- Desacoplamiento
- Resiliencia

Además, la arquitectura está diseñada para ejecutarse dentro de un clúster Kubernetes, donde cada microservicio se despliega como un Deployment independiente.

¿Qué demuestra este diagrama?

Separación por dominios
Independencia de bases de datos
Desacoplamiento mediante broker
Infraestructura dentro de Kubernetes
Rol de cada servicio

Actualización del Diagrama de Arquitectura de Alto Nivel (UML – Componentes)

Objetivo

Reflejar la transición desde una arquitectura básica hacia:

- Microservicios desacoplados
- Comunicación híbrida (REST + Mensajería)
- Infraestructura containerizada
- CI/CD integrado

Componentes Principales

Cliente

- **Web App / Mobile App**
- **Consume API vía HTTPS**

API Gateway

- **Punto único de entrada**
- **Routing**
- **Seguridad (JWT)**
- **Rate limiting**

Auth-Service

- **Registro/Login**
- **Generación de JWT**
- **Gestión de roles**

Order-Service

- **Gestión de órdenes**
- **Publica eventos:**

OrderCreated
OrderCancelled

- **Base de datos propia**

Restaurant-Service

- **Gestión de restaurantes**
- **Consume eventos de órdenes**
- **Actualiza disponibilidad**

RabbitMQ

- **Broker de mensajería**
- **Cola: order.created**
- **Cola: order.status**

Bases de Datos

- **Base independiente por microservicio**
- **PostgreSQL por servicio**

⚙️ **CI/CD**

- **GitHub / GitLab**
- **Pipeline:**
  - **Build**
  - **Test**
  - **Dockerize**
  - **Push registry**
  - **Deploy Kubernetes**

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)

Actualización del Diagrama de Despliegue (UML – Deployment)

Objetivo

Representar:

- Cluster Kubernetes
- Pods
- Servicios
- Ingress
- RabbitMQ
- Bases de datos

Elementos Clave

Cluster Kubernetes

- Node 1
- Node 2

Pods

- Pod Auth-Service
- Pod Order-Service
- Pod Restaurant-Service

- Pod RabbitMQ
- Pod PostgreSQL (por servicio)

Ingress Controller

- NGINX Ingress
- Exposición externa

5) Esquema de Base de Datos (Descripción)

```sql
-- ============================================
-- 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
);
```

```sql
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
);
```

Actualización del Esquema de Base de Datos (ER + Script SQL)

Cambios en Fase 2

- **Separación por microservicio**
- **Tabla de eventos**
- **Estados de orden**
- **Soporte para trazabilidad**

Soporte para eventos asíncronos
Trazabilidad (auditoría)
Idempotencia en mensajería
Desacoplamiento total entre servicios
Mejora en estados y control transaccional
Índices y claves foráneas bien definidas

ACTUALIZACIÓN – auth_db

- **Mejoras realizadas**
  - **Se agregan claves foráneas**
  - **Se agrega auditoría**
  - **Se agrega control de actualización**

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

CREATE TABLE roles (
  id_role INT AUTO_INCREMENT PRIMARY KEY,
  nombre VARCHAR(50) UNIQUE NOT NULL,
  descripcion VARCHAR(255),
  fecha_creacion TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

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,
  fecha_actualizacion TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE
CURRENT_TIMESTAMP
);

CREATE TABLE usuario_roles (
  id_usuario INT NOT NULL,
  id_role INT NOT NULL,
  PRIMARY KEY (id_usuario, id_role),
  CONSTRAINT fk_usuario FOREIGN KEY (id_usuario)
    REFERENCES usuarios(id_usuario)
    ON DELETE CASCADE,
  CONSTRAINT fk_role FOREIGN KEY (id_role)
    REFERENCES roles(id_role)
    ON DELETE CASCADE
);
```

ACTUALIZACIÓN – catalog_db

- **Mejoras**
  - **Índices**
  - **Auditoría**
  - **Preparación para eventos**

```
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,
  fecha_creacion TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
  fecha_actualizacion TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE
CURRENT_TIMESTAMP
);

CREATE TABLE menu_items (
  id_item INT AUTO_INCREMENT PRIMARY KEY,
  id_restaurante INT NOT NULL,
  nombre VARCHAR(150) NOT NULL,
```

```
   precio DECIMAL(10,2) NOT NULL CHECK (precio >= 0),
   disponible BOOLEAN DEFAULT TRUE,
   fecha_creacion TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
   CONSTRAINT fk_menu_restaurante
      FOREIGN KEY (id_restaurante)
      REFERENCES restaurantes(id_restaurante)
      ON DELETE CASCADE
);

CREATE INDEX idx_menu_restaurante ON menu_items(id_restaurante);
```

ACTUALIZACIÓN PRINCIPAL – order_db (Fase 2 Clave)
Aquí es donde realmente adaptamos a mensajería asíncrona.

Nuevos elementos
- **Tabla de eventos (event sourcing ligero)**
- **Tabla outbox (patrón Outbox para RabbitMQ)**
- **Control de estado más robusto**
- **Idempotencia**

```
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',
      'VALIDADA',
      'EN_PROCESO',
      'LISTA',
      'ENVIADA_A_DELIVERY',
      'FINALIZADA',
      'RECHAZADA',
      'CANCELADA'
   ) NOT NULL DEFAULT 'CREADA',
   total DECIMAL(10,2),
   version INT DEFAULT 1,
   fecha_creacion TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
   fecha_actualizacion TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE
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) NOT NULL,
   CONSTRAINT fk_detalle_orden
      FOREIGN KEY (id_orden)
      REFERENCES ordenes(id_orden)
      ON DELETE CASCADE
);

CREATE INDEX idx_orden_cliente ON ordenes(id_cliente);
CREATE INDEX idx_orden_restaurante ON ordenes(id_restaurante);

-- ============================
-- TABLA DE EVENTOS (Fase 2)
-- ============================

CREATE TABLE order_events (
   id_evento BIGINT AUTO_INCREMENT PRIMARY KEY,
   id_orden INT NOT NULL,
   tipo_evento VARCHAR(100) NOT NULL,
   payload JSON NOT NULL,
   procesado BOOLEAN DEFAULT FALSE,
   fecha_creacion TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
   CONSTRAINT fk_evento_orden
      FOREIGN KEY (id_orden)
      REFERENCES ordenes(id_orden)
      ON DELETE CASCADE
);

-- ============================
-- PATRÓN OUTBOX
-- ============================

CREATE TABLE outbox_events (
   id BIGINT AUTO_INCREMENT PRIMARY KEY,
   aggregate_type VARCHAR(50) NOT NULL,
   aggregate_id INT NOT NULL,
   event_type VARCHAR(100) NOT NULL,
   payload JSON NOT NULL,
   enviado BOOLEAN DEFAULT FALSE,
```

```
   fecha_creacion TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

CREATE INDEX idx_outbox_enviado ON outbox_events(enviado);
```

ACTUALIZACIÓN – delivery_db

Mejoras

- **Estados ampliados**
- **Control de trazabilidad**
- **Relación con orden**

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

CREATE TABLE repartidores (
   id_repartidor INT AUTO_INCREMENT PRIMARY KEY,
   nombre VARCHAR(100) NOT NULL,
   activo BOOLEAN DEFAULT TRUE,
   fecha_creacion TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

CREATE TABLE entregas (
   id_entrega INT AUTO_INCREMENT PRIMARY KEY,
   id_orden INT NOT NULL,
   id_repartidor INT NOT NULL,
   estado ENUM(
      'ASIGNADA',
      'EN_CAMINO',
      'ENTREGADO',
      'CANCELADO'
   ) NOT NULL,
   fecha_asignacion TIMESTAMP,
   fecha_entrega TIMESTAMP,
   fecha_actualizacion TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE
CURRENT_TIMESTAMP
);

CREATE INDEX idx_entrega_orden ON entregas(id_orden);
```

¿Qué Cambió Realmente en Fase 2?

Elemento	Fase 1	Fase 2
Eventos	No	Sí
Outbox Pattern	No	Sí
Versionado	No	Sí
Auditoría	Básica	Completa
Estados ampliados	No	Sí
Índices	Parcial	Optimizado
Preparado para RabbitMQ	No	Sí

Esto Permite Ahora

- Order-Service crea orden
- Inserta evento en outbox_events
- Worker publica en RabbitMQ
- Restaurant-Service consume evento
- Sistema resiliente ante fallos
- Idempotencia controlada
- Escalable en Kubernetes

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 – FASE 2 (DETALLADO Y PROFESIONAL)

Este incluye:

- Swimlanes (particiones)
- ORDER_SERVICE
- Message Broker
- CATALOG_SERVICE

- DELIVERY_SERVICE
- NOTIFICATION_SERVICE
- Patrón Outbox
- Procesamiento asíncrono

GUÍA DE DESPLIEGUE EN KUBERNETES

Proyecto: DELIVEREATS – Fase 2

Prerrequisitos

Antes de iniciar necesitas:

- Docker instalado
- Kubernetes (Minikube, Docker Desktop o cluster real)
- kubectl configurado
- Imágenes Docker de cada microservicio

Microservicios:

- auth-service
- catalog-service
- order-service
- delivery-service
- notification-service
- message-broker (Kafka o RabbitMQ)
- Bases de datos MySQL separadas

Verificar cluster:

```
kubectl cluster-info
kubectl get nodes
```

Crear Namespace

```
kubectl create namespace delivereats
```

Verificar:

```
kubectl get namespaces
```

Desplegar Bases de Datos

Ejemplo para AUTH_DB:

auth-db-deployment.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: auth-db
  namespace: delivereats
spec:
  replicas: 1
  selector:
    matchLabels:
      app: auth-db
```

```
  template:
    metadata:
      labels:
        app: auth-db
    spec:
      containers:
      - name: mysql
        image: mysql:8
        env:
        - name: MYSQL_ROOT_PASSWORD
          value: root
        - name: MYSQL_DATABASE
          value: auth_db
        ports:
        - containerPort: 3306
```

Aplicar:

```
kubectl apply -f auth-db-deployment.yaml
```

Crear Service:

```
apiVersion: v1
kind: Service
metadata:
  name: auth-db-service
  namespace: delivereats
spec:
  selector:
    app: auth-db
  ports:
  - port: 3306
  type: ClusterIP
```

Repetir mismo patrón para:

- catalog-db
- order-db

- delivery-db

Desplegar Message Broker (RabbitMQ ejemplo)

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: rabbitmq
  namespace: delivereats
spec:
  replicas: 1
  selector:
    matchLabels:
      app: rabbitmq
  template:
    metadata:
      labels:
        app: rabbitmq
    spec:
      containers:
      - name: rabbitmq
        image: rabbitmq:3-management
        ports:
        - containerPort: 5672
        - containerPort: 15672
```

Service:

```
apiVersion: v1
kind: Service
metadata:
  name: rabbitmq-service
  namespace: delivereats
spec:
  selector:
    app: rabbitmq
```

```
  ports:
  - port: 5672
  - port: 15672
  type: ClusterIP
```

Desplegar Microservicios

Ejemplo: order-service

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: order-service
  namespace: delivereats
spec:
  replicas: 2
  selector:
    matchLabels:
      app: order-service
  template:
    metadata:
      labels:
        app: order-service
    spec:
      containers:
      - name: order-service
        image: tuusuario/order-service:latest
        ports:
        - containerPort: 8080
        env:
        - name: DB_HOST
          value: order-db-service
        - name: BROKER_HOST
          value: rabbitmq-service
```

Service:

```yaml
apiVersion: v1
kind: Service
metadata:
  name: order-service
  namespace: delivereats
spec:
  selector:
    app: order-service
  ports:
  - port: 80
    targetPort: 8080
  type: ClusterIP
```

Repetir para:

- auth-service
- catalog-service
- delivery-service
- notification-service

Exponer API Gateway o Servicio Principal

API Gateway:

```yaml
apiVersion: v1
kind: Service
metadata:
  name: api-gateway
  namespace: delivereats
spec:
  selector:
    app: api-gateway
  ports:
  - port: 80
```

```
  targetPort: 8080
 type: NodePort
```

Ver puerto:

```
kubectl get svc -n delivereats
```

Verificar Despliegue

```
kubectl get pods -n delivereats
kubectl get services -n delivereats
```

Logs:

```
kubectl logs <pod-name> -n delivereats
```

Escalado Horizontal

```
kubectl scale deployment order-service --replicas=3 -n delivereats
```

Verificación Final

Probar endpoints:

http://<node-ip>:<nodeport>/api/orders

Verificar eventos:

- Crear orden
- Revisar logs
- Confirmar publicación en broker
- Confirmar reacción en delivery-service

Limpieza del Entorno

```
kubectl delete namespace delivereats
```

Limpieza del Entorno

kubectl delete namespace delivereats

Guia de kubernetes.

Despliegue de microservicios en Kubernetes
Separación por base de datos
Comunicación por broker
Escalabilidad
Arquitectura cloud-ready

Descripción del Pipeline CI/CD

Objetivo

Automatizar:

- Build de cada microservicio
- Ejecución de tests
- Construcción de imagen Docker
- Push al Container Registry
- Despliegue automático en Kubernetes

Arquitectura del Pipeline

Repositorio → CI (Build & Test) → Docker Build → Registry → CD → Kubernetes

Puede implementarse con:

- GitHub Actions
- GitLab CI
- Jenkins
- Azure DevOps

(Ejemplo asumido: GitHub Actions)

Flujo del Pipeline

Trigger

Se activa cuando:

- Se hace push a main
- Se crea un Pull Request
- Se crea un tag de versión

Stage: Build

Para cada microservicio:

```
mvn clean package
# o
npm install && npm run build
```

Stage: Test

```
mvn test
# o
npm test
```

Si falla → se detiene el pipeline.

Stage: Docker Build

```
docker build -t delivereats/order-service:1.0.0 .
```

Push a Registry

```
docker push registry/delivereats/order-service:1.0.0
```

Deploy a Kubernetes

```
kubectl apply -f k8s/
kubectl rollout restart deployment order-service
```

Variables y Secrets Necesarios

Variables de entorno

Variable	Descripción
DB_HOST	Host de la base de datos
DB_PORT	Puerto DB
DB_NAME	Nombre base
DB_USER	Usuario
DB_PASSWORD	Contraseña
JWT_SECRET	Clave secreta JWT
BROKER_HOST	Host RabbitMQ/Kafka
BROKER_PORT	Puerto broker

Kubernetes Secrets

Ejemplo:

```
kubectl create secret generic db-secret \
  --from-literal=DB_PASSWORD=superpassword \
  -n delivereats
```

Ejemplo YAML:

```
apiVersion: v1
kind: Secret
metadata:
```

```
  name: jwt-secret
type: Opaque
data:
  JWT_SECRET: c3VwZXJzZWNyZXQ=
```

(Base64 encoded)

Flujo JWT en Fase 2

En arquitectura de microservicios, JWT permite:

- Autenticación centralizada
- Autorización distribuida
- Stateless

Flujo JWT Completo

Login

Cliente envía:

```
POST /auth/login
Content-Type: application/json

{
  "correo": "cliente@email.com",
  "password": "123456"
}
```

AUTH_SERVICE valida credenciales

Si son correctas → genera JWT:

Ejemplo payload:

```
{
  "sub": "123",
  "role": "CLIENTE",
```

"iat": 1710000000,
"exp": 1710003600
}

Respuesta:

{
"token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9..."
}

Cliente usa el token

POST /orders
Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...
Content-Type: application/json

ORDER_SERVICE valida JWT

- Verifica firma con JWT_SECRET
- Verifica expiración
- Extrae rol

Si es válido → procesa pedido.

Ventajas en Fase 2

No hay sesión en servidor
Cada microservicio valida token
Escalable horizontalmente
Compatible con Kubernetes

Estrategias de Rollout y Rollback

En Kubernetes se usan Deployment strategies.

Rolling Update (Recomendada)

Reemplaza pods gradualmente.

```
kubectl set image deployment/order-service \
order-service=registry/order-service:1.0.1
```

Kubernetes:

- Crea nuevo pod
- Espera que esté Ready
- Elimina uno viejo
- Repite

Sin downtime
Producción estable

Blue-Green Deployment

Se tienen dos versiones:

- v1 (Blue)
- v2 (Green)

Se cambia el Service para apuntar a la nueva.

Ventajas:

Rollback inmediato
Testing en producción

Canary Deployment

Se despliega:

- 10% tráfico a nueva versión
- 90% a antigua

Si funciona → aumentar tráfico.

Ideal con:

- Ingress + NGINX
- Istio

Rollback en Kubernetes

Ver historial:

kubectl rollout history deployment/order-service

Volver a versión anterior:

kubectl rollout undo deployment/order-service

Fase # 2:

DevOps real
Seguridad con JWT
Automatización CI/CD
Estrategias profesionales de despliegue
Preparado para producción

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

SCRUM – FASE 2 (Versión Evolucionada)

Historias de Usuario – Fase 2

HU-08 Autenticación con JWT distribuido

Como microservicio
Quiero validar tokens JWT
Para permitir acceso seguro sin sesión centralizada

Criterios de aceptación

- El token incluye id y rol
- Firma válida con JWT_SECRET
- Token expira correctamente
- Cada microservicio valida token de forma independiente

HU-09 Publicar evento OrderCreated

Como Order Service
Quiero publicar un evento cuando se crea una orden
Para notificar a otros microservicios

Criterios

- Se registra en tabla outbox_events
- Se publica al broker
- Se marca como enviado
- No se pierde evento si falla el envío

HU-10 Consumir evento en Delivery Service

Como Delivery Service
Quiero consumir evento OrderCreated
Para crear entrega automáticamente
Criterios

- Solo procesa órdenes EN_PROCESO
- Maneja reintentos
- No duplica entregas

HU-11 Actualización por eventos

Como Order Service
Quiero actualizar estado cuando reciba DeliveryCompleted
Para cerrar la orden automáticamente
Criterios

- Evento recibido correctamente
- Estado cambia a ENTREGADO
- Registro en tabla order_events

HU-12 Pipeline CI/CD automático

Como equipo DevOps
Quiero pipeline automatizado
Para desplegar sin intervención manual
Criterios

- Ejecuta tests
- Construye imagen Docker
- Hace push a registry
- Despliega en Kubernetes
- Soporta rollback

HU-13 Despliegue en Kubernetes

Como equipo técnico
Quiero desplegar en Kubernetes
Para escalar microservicios
Criterios

- Deployment por microservicio
- Services internos
- Secrets configurados
- Rolling Update activo

HU-14 Observabilidad básica

Como equipo técnico
Quiero monitorear servicios
Para detectar fallos
Criterios

- Logs centralizados
- readinessProbe y livenessProbe
- Métricas básicas

Product Backlog – Fase 2

ID	Historia	Prioridad	Estimación
HU-08	JWT distribuido	Alta	3 pts
HU-09	Publicar eventos (Outbox)	Alta	5 pts
HU-10	Consumidores de eventos	Alta	5 pts
HU-11	Actualización por eventos	Media	3 pts
HU-12	Pipeline CI/CD	Alta	5 pts
HU-13	Kubernetes deployment	Alta	5 pts
HU-14	Observabilidad	Media	3 pts

Planificación de Sprints – Fase 2

Sprint 1 – Seguridad y Base Event-Driven

- **Implementar JWT distribuido**
- **Agregar tabla order_events**
- **Agregar tabla outbox_events**

- **Publicador de eventos**
- **Configuración Message Broker**

Sprint 2 – Comunicación entre Microservicios

- **Consumer en Delivery Service**
- **Consumer en Catalog Service**
- **Manejo de reintentos**
- **Evitar duplicados**
- **Pruebas de integración**

Sprint 3 – DevOps

- **Dockerización completa**
- **Pipeline CI/CD**
- **Versionado de imágenes**
- **Rolling Update**
- **Rollback automático**

Sprint 4 – Kubernetes + Escalabilidad

- **Deploy en Kubernetes**
- **Configuración Secrets**
- **ConfigMaps**
- **Readiness & Liveness Probes**
- **Escalado horizontal**
- **Pruebas de carga básicas**

Diferencia Clara Fase 1 vs Fase 2

Fase 1	Fase 2
CRUD tradicional	Event-Driven

Fase 1	Fase 2
Comunicación directa	Message Broker
Arquitectura simple	Arquitectura distribuida
Sin CI/CD	Pipeline automatizado
Sin Kubernetes	Despliegue orquestado
Sin Outbox	Patrón Outbox

Resultado Final Fase 2

Tu proyecto ahora demuestra:

- **Microservicios reales**
- **Seguridad con JWT**
- **Comunicación asíncrona**
- **DevOps moderno**
- **Kubernetes**
- **Estrategias de despliegue**
- **Escalabilidad**

/*/*/*/*/*/*/*/*

Prueba de Concepto (PoC) – RabbitMQ

Arquitectura de Microservicios con Node.js

Introducción

El presente documento describe la implementación de una Prueba de Concepto (PoC) utilizando RabbitMQ como sistema de mensajería en una arquitectura basada en microservicios desarrollada en Node.js.

El objetivo principal es demostrar la comunicación asíncrona entre servicios mediante el patrón Productor–Consumidor, garantizando el desacoplamiento entre componentes del sistema.

Arquitectura Implementada

El sistema está compuesto por los siguientes microservicios:

- **Order-Service → Servicio productor**
- **Catalog-Service → Validación vía gRPC (comunicación síncrona)**
- **Restaurant-Service → Servicio consumidor**
- **RabbitMQ → Broker de mensajería**

Flujo General del Sistema

1. **El cliente crea una orden.**
2. **Order-Service valida la orden mediante gRPC con Catalog-Service.**
3. **Order-Service almacena la orden en base de datos.**
4. **Order-Service publica un mensaje en RabbitMQ en la cola orden_creada.**
5. **Restaurant-Service consume el mensaje de la cola.**
6. **Restaurant-Service procesa e imprime la orden en consola.**

Tecnologías Utilizadas

- **Node.js**
- **RabbitMQ**
- **amqplib (cliente RabbitMQ para Node.js)**
- **MySQL (para catálogo)**
- **gRPC**
- **Postman (para pruebas)**

Configuración del Entorno

4.1 Instalación de RabbitMQ

RabbitMQ se ejecuta localmente en:

amqp://localhost:5672

Panel de administración:

http://localhost:15672

Credenciales por defecto:

- **Usuario: guest**
- **Contraseña: guest**

Implementación del Productor (Order-Service)

5.1 Publicación del Evento

Cuando se crea una orden, el servicio publica un mensaje en la cola:

```
await channel.assertQueue("orden_creada", { durable: false });

channel.sendToQueue(
  "orden_creada",
  Buffer.from(JSON.stringify(orden))
);
```

Funcionalidad:

- **Se asegura la existencia de la cola.**
- **Se serializa la orden en formato JSON.**
- **Se envía el mensaje al broker RabbitMQ.**

Resultado Esperado:

En el panel de RabbitMQ se observa:

Queue: orden_creada
Ready: 1

Implementación del Consumidor (Restaurant-Service)

6.1 Creación del Servicio

Estructura:

```
restaurant-service/
│
├── package.json
└── server.js
```

Instalación de dependencia:

```
npm install amqplib
```

6.2 Código del Consumidor

```
const amqp = require("amqplib");

const QUEUE = "orden_creada";
const RABBIT_URL = "amqp://localhost:5672";

async function startServer() {
  const connection = await amqp.connect(RABBIT_URL);
  const channel = await connection.createChannel();

  await channel.assertQueue(QUEUE, { durable: false });

  console.log("Restaurant-Service escuchando órdenes...");

  channel.consume(
    QUEUE,
    async (msg) => {
      if (msg) {
        const orden = JSON.parse(msg.content.toString());

        console.log("Orden recibida:", orden);

        channel.ack(msg);
      }
    },
    { noAck: false }
  );
}

startServer();
```

6.3 Explicación Técnica

- **assertQueue() garantiza que la cola exista.**
- **consume() escucha mensajes entrantes.**
- **ack() confirma que el mensaje fue procesado correctamente.**
- **Se utiliza confirmación manual (noAck: false) para garantizar confiabilidad.**

Pruebas Realizadas

Paso 1

Ejecutar Order-Service.

Paso 2

Ejecutar Restaurant-Service:

node server.js

Paso 3

Crear orden desde Postman.

Resultado Observado

En consola del consumidor:

Orden recibida:
{ ...datos de la orden... }

En RabbitMQ:

Ready: 0
Unacked: 0

Lo que confirma que el mensaje fue consumido correctamente.

Validación del Cumplimiento de la PoC

Requisito	Estado
Order-Service publica mensaje	Cumplido
RabbitMQ recibe mensaje	Cumplido
Restaurant-Service consume mensaje	Cumplido
Mensaje registrado en consola	Cumplido
Confirmación manual del mensaje (ack)	Implementado

Conclusión

La Prueba de Concepto demuestra exitosamente:

- **Implementación del patrón Productor–Consumidor.**
- **Comunicación asíncrona mediante RabbitMQ.**
- **Desacoplamiento entre microservicios.**
- **Confirmación manual de mensajes para garantizar confiabilidad.**
- **Arquitectura basada en eventos (Event-Driven Architecture).**

El sistema cumple completamente con los requisitos establecidos para la PoC.

Consideraciones Técnicas

- **La configuración de la cola debe coincidir entre productor y consumidor.**
- **RabbitMQ no permite modificar propiedades de una cola existente.**
- **Se recomienda usar colas durables en ambientes productivos.**
- **El uso de ack() evita pérdida de mensajes.**

Archivo Editar Selección Ver Ir Ejecutar
SA_PROYECTO_201222687
EXPLORADOR
EDITORES ABIERTOS
server.js FASEUNO\microservicios\restaurante-service
package.json FASEUNO\microservicios\restaurante-s...
SA_PROYECTO_201222687
FASEDOS
FASEUNO
PRACTICA1
PRACTICA2
PRACTICA3
PRACTICA4
README.md
ESQUEMA
LÍNEA DE TIEMPO
FASEUNO > microservicios > restaurante-service > server.js > ...
async function startServer() {
} catch (error) {
console.error("❌ Error conectando a RabbitMQ:", error);
}
}
async function procesarOrden(orden) {
console.log("⏳ Procesando orden...");
await new Promise((resolve) => setTimeout(resolve, 2000));
console.log(`✅ Orden procesada correctamente\n`);
}
PROBLEMAS SALIDA TERMINAL CONSOLA DE DEPURACIÓN PUERTOS
PS C:\Users\jcmal\Desktop\INGENIERIA\INGE_2026\1_SMSTRE_2026\SOFT_AVANZADO\LAB_SOFT_AVANZADO\SA_PROYECTO_201222687\FASEUNO\microservicios\restaurante-service> node server.js
}
⏳ Procesando orden...
✅ Orden procesada correctamente
PS C:\Users\jcmal\Desktop\INGENIERIA\INGE_2026\1_SMSTRE_2026\SOFT_AVANZADO\LAB_SOFT_AVANZADO\SA_PROYECTO_201222687\FASEUNO\microservicios\restaurante-service> node server.js
Debugger attached.
🚀 Iniciando Restaurant-Service...
Restaurant-Service escuchando órdenes...
📦 Orden recibida:
{
id_orden: 19,
id_cliente: 31,
id_restaurante: 2,
total: 105,
estado: 'CREADA',
fecha: '2026-02-28T00:11:07.020Z'
}
⏳ Procesando orden...
✅ Orden procesada correctamente
node api-gateway
node auth-service
node catalog-service
node order-service
node frontend
powershell
JavaScript Debug Termina
Lín. 60, col. 15 Espacios: 4 UTF-8 CRLF

localhost:15672/#/queues
RabbitMQ
RabbitMQ 4.2.4
Erlang 28.3.3
Refreshed 2026-02-27 18:11:58
Refresh every 5 seconds
Virtual host All
Cluster rabbit@Maldonado
User guest
Log out
All stable feature flags must be enabled after completing an upgrade. [Learn more]
Deprecated features are being used. [Learn more]
Overview
Connections
Channels
Exchanges
Queues and Streams
Admin
Queues
All queues (1)
Pagination
Page 1 of 1 - Filter:
Regex ?
Displaying 1 item , page size up to: 100
Overview
Messages
Message rates
Virtual host
Name
Type
Features
State
Ready
Unacked
Total
incoming
deliver / get
ack
/
orden_creada
classic
Args
idle
0
0
0
0.00/s
0.00/s
0.00/s
Add a new queue
HTTP API
Documentation
Tutorials
New releases
Commercial edition
Commercial support
Discussions
Discord
Plugins
GitHub
localhost:15672/#/queues
ESP
LAA
18:11
27/02/2026

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.**

La transición de la Fase 1 a la Fase 2 representa un avance significativo en la madurez técnica del proyecto DELIVEREATS. Mientras que la Fase 1 permitió validar los requerimientos funcionales y establecer una base estructural del sistema, la Fase 2 introduce principios de arquitectura moderna que preparan la solución para entornos reales de producción.
La adopción de una arquitectura orientada a eventos mejora el desacoplamiento entre servicios y favorece la escalabilidad. La implementación del Patrón Outbox garantiza confiabilidad en la comunicación distribuida. La integración de JWT fortalece la seguridad, mientras que el uso de CI/CD y Kubernetes posiciona el sistema dentro de prácticas DevOps actuales.
En conjunto, esta evolución transforma el proyecto de un sistema académico funcional a una solución técnicamente alineada con estándares empresariales, capaz de operar en entornos distribuidos, escalables y resilientes.

¿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

Login

Correo

Contraseña

Ingresar

Módulo Cliente

Registrar nuevo cliente

Nombre

Correo

Contraseña

Registrar Cliente

Cerrar Sesión

React App
Archivo Editar Selección Ver Ir Ejecutar
EXPLORADOR
EDITORES ABIERTOS
docker-compose.yml microservicios
PRACTICA2
microservicios
api-gateway
node_modules
src
.env
Dockerfile
package-lock.json
package.json
server.js
auth-service
node_modules
src
.env
Dockerfile
package-lock.json
package.json
server.js
test-db.js
frontend
proto
auth.proto
docker-compose.yml
[SA]Practica 2.pdf
~$cumentacion_practica2.docx
~WRL0168.tmp
auth-service.sql
catalog-service.sql
consultas_auth_service.sql
delivery-service.sql
docker-compose.yml
microservicios > docker-compose.yml
version: "3.9"
Run All Services
services:
Run Service
api-gateway:
build: ./api-gateway
container_name: api-gateway
ports:
- "5000:5000"
environment:
- PORT=5000
- GRPC_AUTH_HOST=auth-service:50051
volumes:
- ./proto:/proto
depends_on:
- auth-service
Run Service
auth-service:
PROBLEMAS SALIDA TERMINAL CONSOLA DE DEPURACIÓN
PS C:\Users\jcmal\Desktop\INGENIERIA\INGE_2026\1_SMST

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
Cliente
API Gateway (REST)
Auth Service (gRPC)
Login / Registro
Login / Registro (credenciales)
Validar credenciales
Verificar usuario y contraseña
alt
[Autenticación correcta]
Usuario válido
Generar JWT {sub, correo, rol}
Retorna JWT
Guarda token
[Autenticación incorrecta]
Credenciales inválidas
Error 401 Unauthorized
Acceso a recurso protegido
Petición protegida (Authorization: token)
Verificar JWT
alt
[Token válido]
Respuesta con datos
[Token inválido]
401 Unauthorized
Cliente
API Gateway (REST)
Auth Service (gRPC)

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:

Dockerfile

microservicios > api-gateway > Dockerfile > ...

```
FROM node:18

WORKDIR /app

COPY package*.json ./
RUN npm install

COPY . .

EXPOSE 5000
CMD ["node", "server.js"]
```

Archivo Dockerfile de auth-service:

Dockerfile

microservicios > auth-service > Dockerfile > ...

```
FROM node:18

WORKDIR /app

COPY package*.json ./
RUN npm install

COPY . .

EXPOSE 50051
CMD ["node", "server.js"]
```

Archivo Dockerfile de frontend:

Dockerfile

microservicios > frontend > Dockerfile > ...

```
FROM node:18

WORKDIR /app

COPY package*.json ./
RUN npm install

COPY . .

EXPOSE 3000
CMD ["npm", "start"]
```

Archivo docker-compose.yml de microservicios.

docker-compose.yml

microservicios > docker-compose.yml

```yaml
version: "3.9"

services:
  api-gateway:
    build: ./api-gateway
    container_name: api-gateway
    ports:
      - "5000:5000"
    environment:
      - PORT=5000
      - GRPC_AUTH_HOST=auth-service:50051
    volumes:
      - ./proto:/proto
    depends_on:
      - auth-service

  auth-service:
    build: ./auth-service
    container_name: auth-service
    ports:
      - "50051:50051"
    volumes:
      - ./proto:/proto
    environment:
    - DB_HOST=host.docker.internal
    - DB_USER=root
    - DB_PASSWORD=Yareli2026.
    - DB_NAME=auth_db
    - DB_PORT=3306
    - JWT_SECRET=delivereats_secret_super_seguro

  frontend:
    build: ./frontend
    container_name: frontend
    ports:
      - "3000:3000"
    depends_on:
      - api-gateway
```

*_*_*_*_*_*_*_*

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.

--*-*-*

Manual de Usuario

Despliegue en Google Cloud Platform con Ubuntu

Plataforma de Gestión de Pedidos – Fase 1

1. Introducción

Este manual describe el proceso de implementación de una infraestructura en **Google Cloud Platform** utilizando una máquina virtual con **Ubuntu** para alojar una arquitectura basada en microservicios.

El objetivo es proporcionar un entorno escalable, seguro y accesible para el despliegue de la plataforma distribuida del proyecto.

2. FASE UNO – Descripción del problema

Pequeños comercios y emprendedores suelen administrar pedidos e inventarios mediante:

- Mensajes de WhatsApp
- Redes sociales
- Llamadas telefónicas
- Cuadernos o registros manuales

Problemas identificados

- Pérdida de información
- Errores en pedidos
- Falta de control de inventario
- Mala coordinación entre clientes, restaurantes y repartidores
- Dificultad para escalar el negocio
- Ausencia de automatización

Solución propuesta

Desarrollo de una **plataforma distribuida basada en microservicios** que permita:

- Gestión de usuarios
- Manejo de pedidos
- Catálogo de restaurantes
- Entregas
- Notificaciones
- Seguridad mediante JWT
- Comunicación REST y gRPC

3. Alcance del Proyecto – Microservicios

Microservicios mínimos

1. API Gateway / BFF
2. Auth-Service
3. Restaurant-Catalog-Service
4. Order-Service
5. Delivery-Service

6. Notification-Service

4. Arquitectura General

La plataforma utiliza:

- **API Gateway (REST)** para el frontend
- Comunicación interna mediante **gRPC**
- Autenticación con **JWT**
- Base de datos por servicio
- Despliegue en VM Ubuntu dentro de GCP

---

5. Creación de Máquina Virtual en GCP

4

Paso 1: Acceder a Compute Engine

1. Ingresar a la consola de GCP
2. Ir a **Compute Engine**
3. Seleccionar **Create Instance**

Paso 2: Configuración básica

- Nombre: microservices-vm
- Región: cercana al usuario
- Máquina: e2-medium (recomendado)
- Sistema operativo: **Ubuntu 22.04 LTS**

- Firewall:
    - Allow HTTP
    - Allow HTTPS

Paso 3: Crear la instancia

Una vez creada, se asignará una **IP externa temporal**.

6. Conexión a la VM (SSH)

Desde el navegador:

- Click en **SSH**

O desde terminal:

ssh usuario@IP_PUBLICA

Actualizar sistema:

sudo apt update && sudo apt upgrade -y

Instalar herramientas necesarias:

sudo apt install docker.io docker-compose git -y

7. ¿Por qué usar IP Estática?

Problema con IP temporal

Las IP externas por defecto son **efímeras**:

- Cambian al reiniciar la VM
- El frontend o clientes pierden conexión
- DNS deja de funcionar

Beneficios de una IP estática

- Dirección fija permanente
- Acceso confiable al servidor
- Permite:
    - Configurar dominios
    - Conectar frontend
    - Acceso a APIs
    - Integración con servicios externos

Cómo asignar IP estática

1. Ir a **VPC Network → External IP addresses**
2. Seleccionar la IP de la VM
3. Click en **Reserve Static Address**
4. Asignar nombre y guardar

8. Despliegue del Proyecto en Ubuntu

Clonar repositorio

```
git clone https://github.com/usuario/proyecto.git
cd proyecto
```

Levantar microservicios (Docker)

sudo docker-compose up -d --build
Ver contenedores:
sudo docker ps

---

9. Descripción de los Microservicios

9.1 API Gateway

- Expone endpoints REST
- Valida JWT
- Controla roles
- Enruta a servicios gRPC

9.2 Auth-Service

Funciones:

- Registro de usuarios
- Login
- Generación de JWT
- Validación de tokens

Modelo de usuario:

- Id
- Email
- Contraseña encriptada
- Rol:
    - CLIENTE
    - RESTAURANTE
    - REPARTIDOR
    - ADMIN

9.3 Restaurant-Catalog-Service

Funciones:

Administrador

- CRUD de restaurantes

Restaurante

- CRUD de menú

---

10. Acceso a la Plataforma

Desde navegador:
http://IP_ESTATICA:PUERTO
Ejemplo:
http://34.xxx.xxx.xxx:5000

---

11. Seguridad Implementada

- Autenticación con JWT
- Contraseñas encriptadas
- Validación de roles
- Firewall GCP
- Acceso controlado por API Gateway

---

12. Ventajas de usar GCP

- Escalabilidad
- Alta disponibilidad
- Pago por uso
- Acceso remoto global
- Integración con otros servicios cloud

---

13. Conclusión

La implementación en **Google Cloud Platform** permite desplegar una arquitectura moderna basada en microservicios utilizando una máquina virtual Ubuntu con IP estática, garantizando:

- Disponibilidad constante
- Escalabilidad futura
- Seguridad
- Facilidad de administración

Esta infraestructura constituye la base para el crecimiento de la plataforma en fases posteriores.