

Guatemala 01 de febrero del 2026.

Universidad De San Carlos De Guatemala.

Laboratorio software avanzado sección "A".

Auxiliar: Juan Pablo Samayoa Ruiz.



Practica #2: Principios SOLID y JWT para sesiones de Delivereats.

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 utilizó (Principios SOLID) para la creación del registro e inicio de sesión de la plataforma.

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.

2.2 Objetivos específicos

- Desarrollar un servicio de autenticación que gestione el registro y login mediante JWT
- Asegurar la integridad de los datos sensibles mediante la encriptación de contraseñas
- Establecer comunicación técnica entre componentes utilizando el protocolo gRPC para servicios y REST para Gateway
- Garantizar la persistencia de datos y la protección de información sensible mediante contraseñas encriptadas.

3) Alcance

- **Frontend:** Interfaz de usuario para capturar los datos
- **API Gateway:** Encargado de exponer rutas REST, validar tokens JWT, enrutar peticiones vía gRPC.
- **Auth-Service:** Responsable de la lógica de usuarios y generación de JWT y almacenamiento seguro de credenciales.

Diseño:

Auth

- Usuarios
- Roles
- Relación muchos a muchos usuario_rol

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:

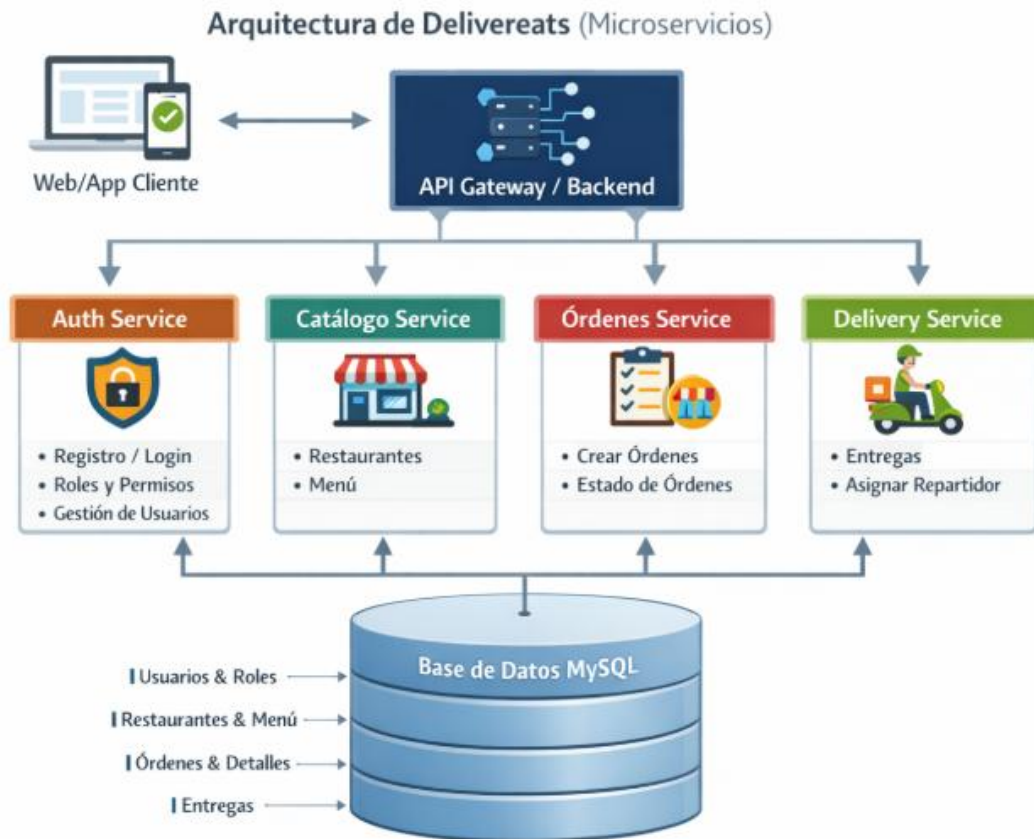
- asignación de repartidor
- control de entregas y estados

Tablas relacionadas:

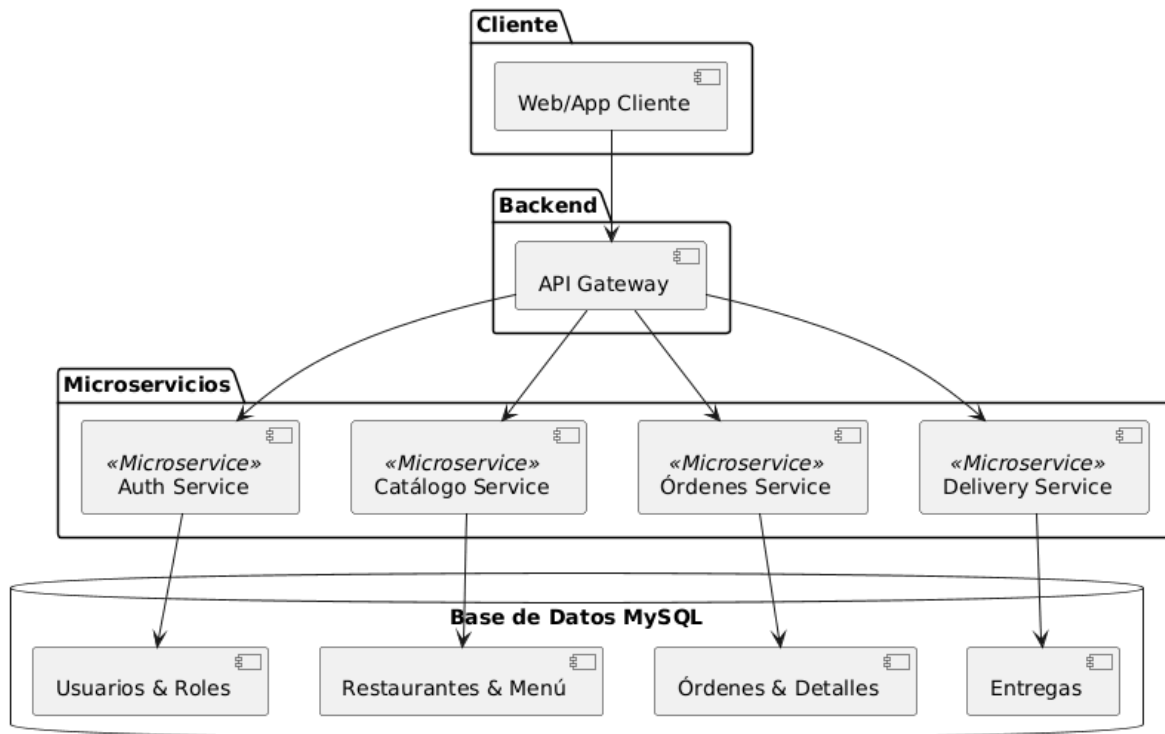
- entregas

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)

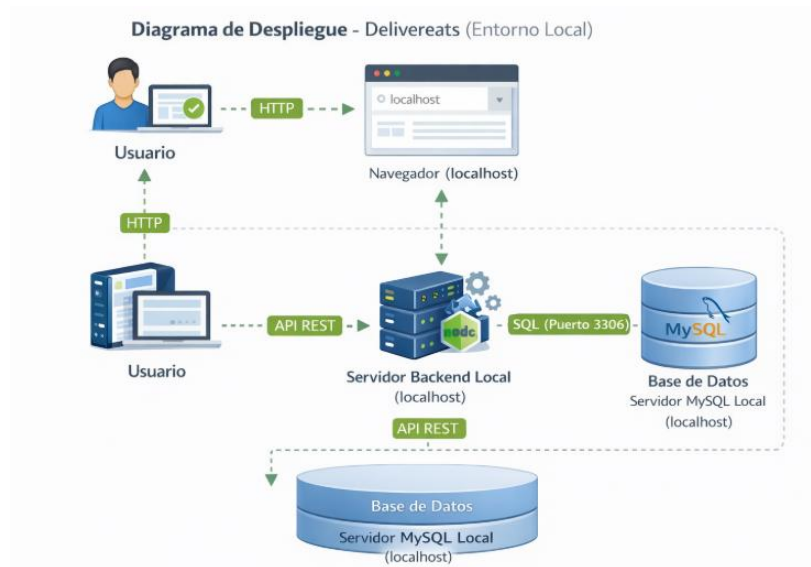
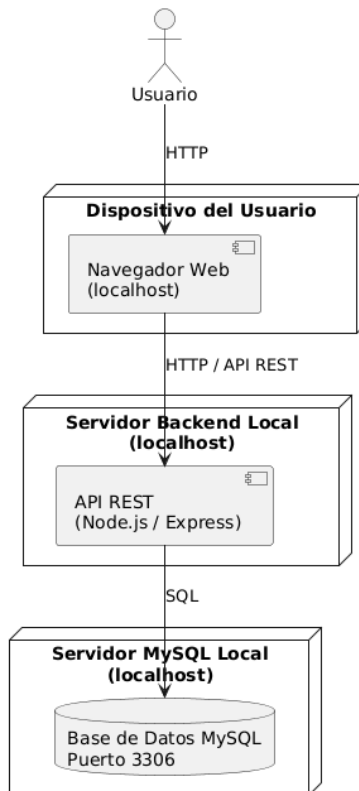


Diagrama de Despliegue UML - DeliverEats (Entorno Local)



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)

menu_items
id_item INT
id_restaurante INT
nombre VARCHAR(150)
precio DECIMAL(10,2)
disponible TINYINT(1)
Indexes

ordenes
id_orden INT
id_cliente INT
id_restaurante INT
estado ENUM(...)
total DECIMAL(10,2)
fecha_creacion TIMESTAM...
Indexes

usuarios
id_usuario INT
nombre VARCHAR(100)
correo VARCHAR(150)
password_hash VARCHAR(255)
activo TINYINT(1)
fecha_creacion TIMESTAMP
Indexes

entregas
id_entrega INT
id_orden INT
id_repartidor INT
estado ENUM(...)
fecha_asignacion TIMESTA...
fecha_entrega TIMESTAMP
Indexes

usuario_rol...
id_usuario INT
id_role INT
Indexes

restaurantes
id_restaurante INT
nombre VARCHAR(150)
direccion VARCHAR(200)
activo TINYINT(1)
Indexes

roles
id_role INT
nombre VARCHAR(50)
Indexes

orden_detalle
id_detalle INT
id_orden INT
id_item INT
cantidad INT
precio_unitario DECIMAL(10,2)
Indexes

-- =====

-- 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;

```

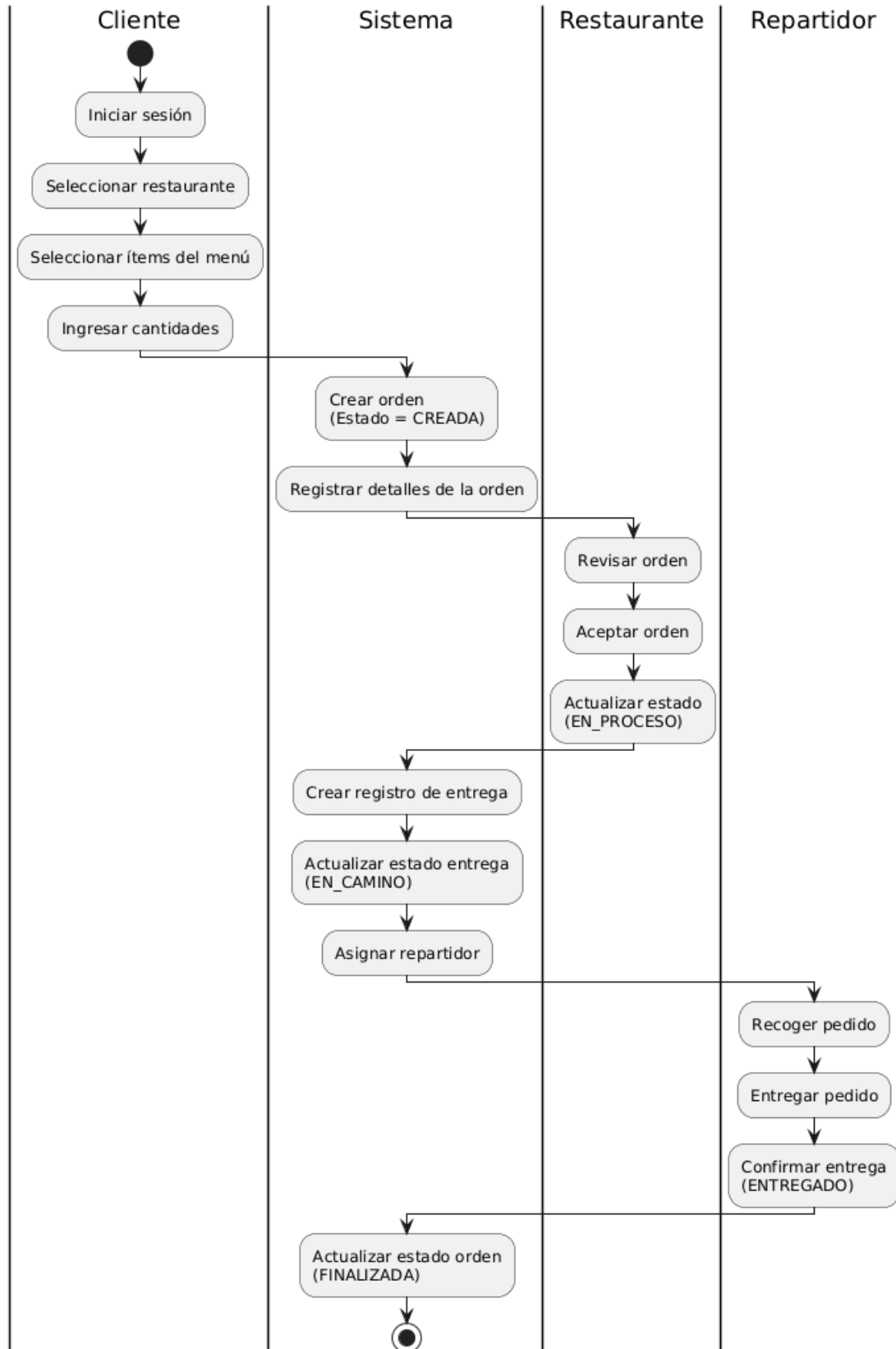
```
CREATE TABLE entregas (  
  id_entrega INT AUTO_INCREMENT PRIMARY KEY,  
  id_orden INT NOT NULL,  
  id_repartidor INT NOT NULL,  
  estado ENUM('EN_CAMINO','ENTREGADO','CANCELADO'),  
  fecha_asignacion TIMESTAMP,  
  fecha_entrega TIMESTAMP  
);
```

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 Deliverats.
 - 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.

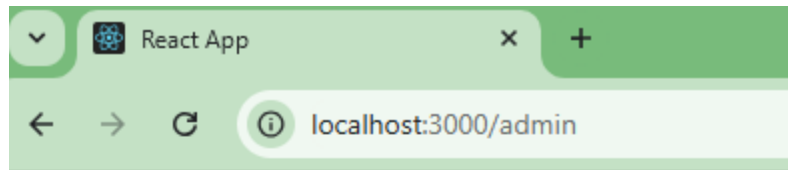
Frontend: React.

- 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



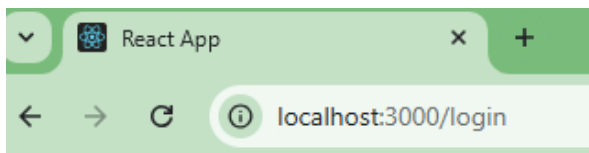
Panel de Administración

Registrar Repartidor o Restaurante

Repartidor ▼

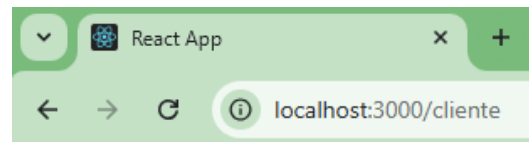
Registrar

Cerrar Sesión



Login

Ingresar



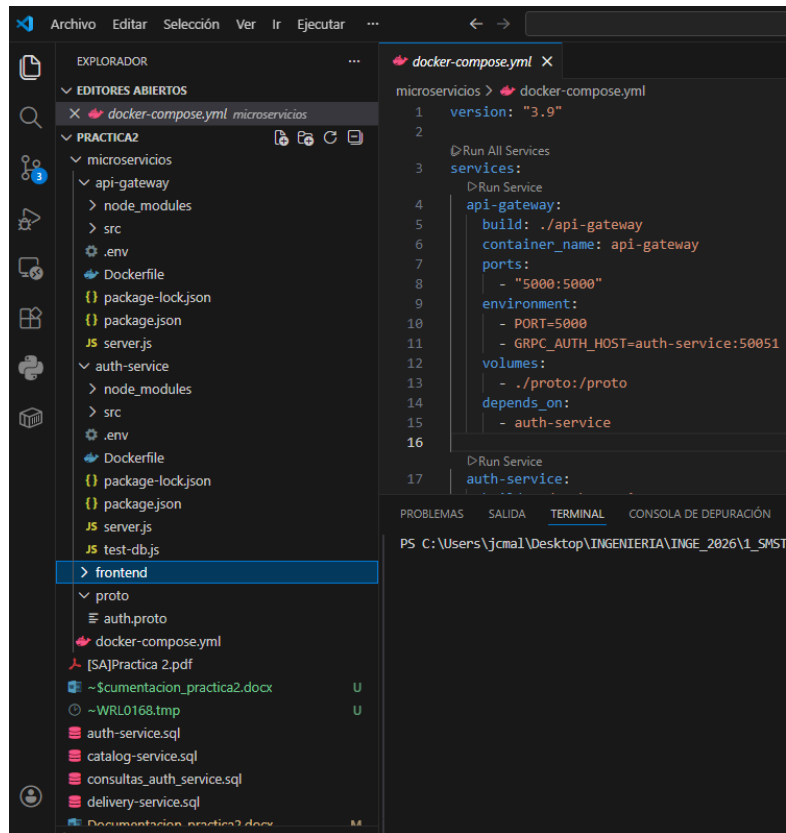
Módulo Cliente

Registrar nuevo cliente

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.

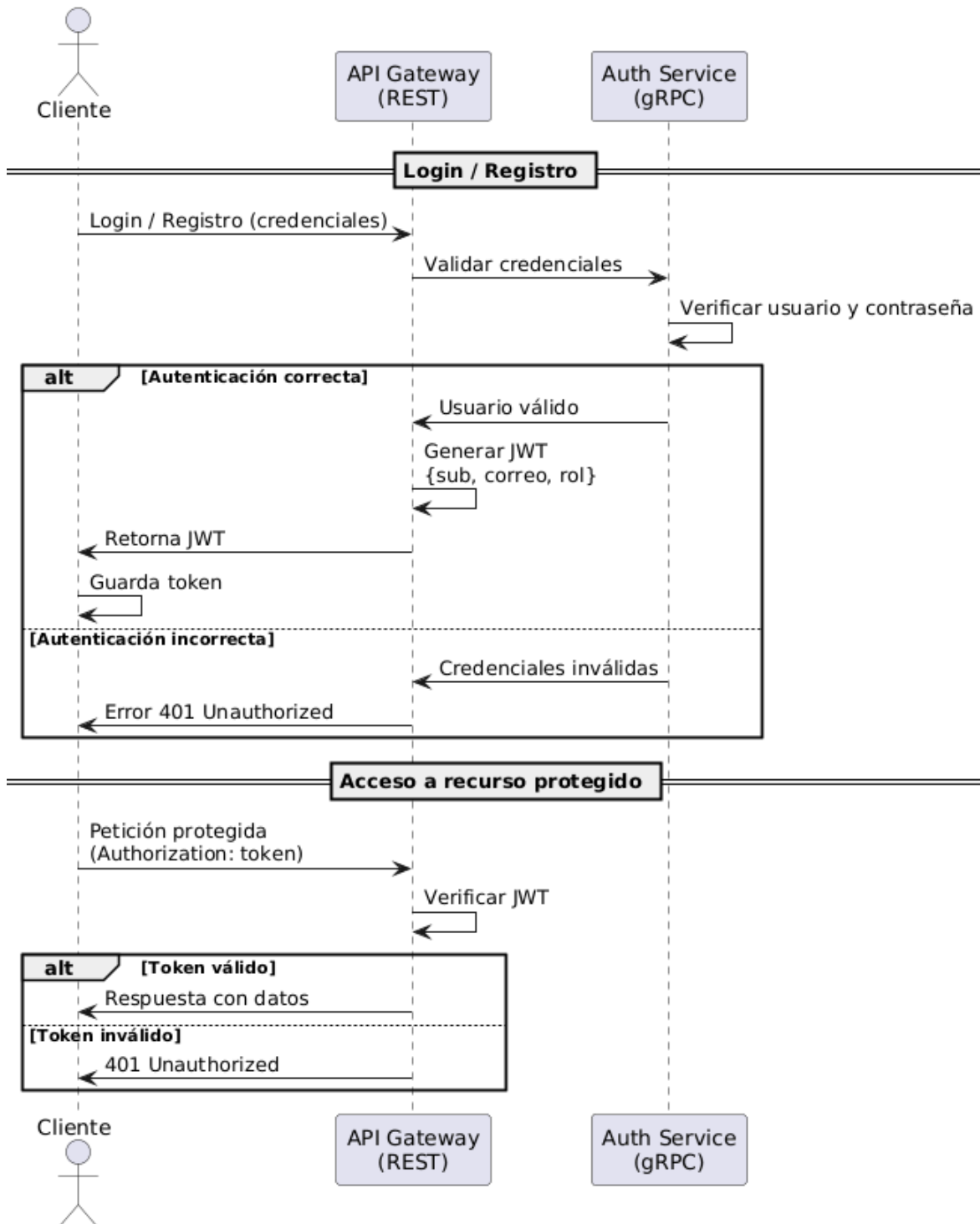
Consumo y comunicación

- 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.

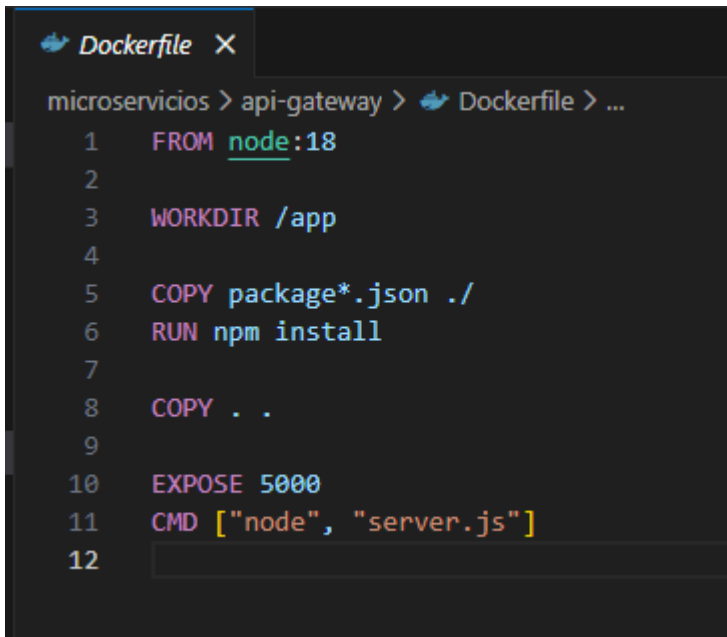
Comandos básicos de despliegue:

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

Para detener el sistema:

```
docker-compose down
```

Archivo Dockerfile de api-gateway:

A screenshot of a code editor showing a Dockerfile. The file is titled 'Dockerfile' with a close button. The breadcrumb navigation shows 'microservicios > api-gateway > Dockerfile > ...'. The code is as follows:

```
1 FROM node:18
2
3 WORKDIR /app
4
5 COPY package*.json ./
6 RUN npm install
7
8 COPY . .
9
10 EXPOSE 5000
11 CMD ["node", "server.js"]
12
```

Archivo Dockerfile de auth-service:

```
Dockerfile X
microservicios > auth-service > Dockerfile > ...
1  FROM node:18
2
3  WORKDIR /app
4
5  COPY package*.json ./
6  RUN npm install
7
8  COPY . .
9
10 EXPOSE 50051
11 CMD ["node", "server.js"]
12
```

Archivo Dockerfile de frontend:

```
Dockerfile X
microservicios > frontend > Dockerfile > ...
1  FROM node:18
2
3  WORKDIR /app
4
5  COPY package*.json ./
6  RUN npm install
7
8  COPY . .
9
10 EXPOSE 3000
11 CMD ["npm", "start"]
12
```

Archivo docker-compose.yml de microservicios.

```
docker-compose.yml X
microservicios > docker-compose.yml
1  version: "3.9"
2
3  services:
4    api-gateway:
5      build: ./api-gateway
6      container_name: api-gateway
7      ports:
8        - "5000:5000"
9      environment:
10       - PORT=5000
11       - GRPC_AUTH_HOST=auth-service:50051
12      volumes:
13       - ./proto:/proto
14      depends_on:
15       - auth-service
16
17    auth-service:
18      build: ./auth-service
19      container_name: auth-service
20      ports:
21       - "50051:50051"
22      volumes:
23       - ./proto:/proto
24      environment:
25       - DB_HOST=host.docker.internal
26       - DB_USER=root
27       - DB_PASSWORD=Yareli2026.
28       - DB_NAME=auth_db
29       - DB_PORT=3306
30       - JWT_SECRET=delivereats_secret_super_seguro
31
32    frontend:
33      build: ./frontend
34      container_name: frontend
35      ports:
36       - "3000:3000"
37      depends_on:
38       - api-gateway
39
```