TRABAJO DE GRADO
Opción Seminario-Diplomado.

Plataforma Web DeliApp Basada en Microservicios para Optimizar la Entrega de Comida a Domicilio

Corporación Universitaria Remington.
Facultad de Ingeniería
Ingeniería en Sistemas

Erika Vanesa Rincón Quiñones
Andrés Hincapié Salazar
Tutor: Diego Fernando Marín Lozano

Opción de Trabajo de grado Seminario-Diplomado
2025

Dedicatoria

Dedicamos este proyecto a nuestra familia, nuestros padres, hijos, que nos han apoyado y brindado el respaldo durante este periodo de desarrollo profesional.

Agradecimientos

Queremos expresar nuestro más profundo agradecimiento a nuestras familias, quienes nos han acompañado con su apoyo incondicional y confianza, especialmente en los momentos más retadores de este proyecto.

También extendemos un agradecimiento muy especial al ingeniero Diego Fernando Marín, por su paciencia, comprensión y guía, que fueron fundamentales para nuestro crecimiento profesional y para ampliar nuestros conocimientos.

Tabla de Contenido

Resumen

En este proyecto se describe la idea y el desarrollo de DELIAPP, una plataforma web creada para gestionar pedidos de comida a domicilio en el municipio colombiano de Barbacoas. Mediante la integración eficiente de restaurantes, clientes y repartidores, la aplicación busca resolver la necesidad de soluciones digitales sólidas que mejoren la logística y maximicen las entregas.

Gracias al enfoque de desarrollo incremental utilizado, se pudieron entregar pronto características importantes y validar continuamente los prototipos. Se prevé que la experiencia del usuario final y la eficiencia operativa de los restaurantes locales sean los resultados esperados. Esta investigación promueve soluciones técnicas para las economías rurales emergentes.

Palabras clave

Plataforma web, microservicios, desarrollo incremental, soluciones digitales, economías rurales.

Problema

La pandemia de COVID-19 aumentó exponencialmente la cantidad de usuarios y la oferta de aplicaciones de entrega de comida a domicilio. Sin embargo, este aumento no se convirtió en un acierto en el tema financiero para las empresas del sector.

Un desafío fundamental radica en la limitada oferta de restaurantes asociados en numerosas plataformas, ya que muchos establecimientos operan con modelos de negocio independientes o se hallan en zonas geográficas no cubiertas, lo que impide la implementación de una experiencia de usuario verdaderamente integral (Navarro, Berbek, & Sanchez, 2024). Asimismo, debe considerarse que numerosos consumidores enfrentan barreras de acceso debido a factores como el costo elevado del servicio, restricciones de pago digital, la falta de tiempo para realizar pedidos y la limitada conectividad en ciertas áreas (Unesco, 2024).

Una limitación significativa de estas plataformas de entrega es su implementación en arquitecturas monolíticas, donde todos los procesos —desde la gestión del menú y el carrito de compras hasta la asignación de repartidores— se ejecutan dentro de un mismo servicio, comprometiendo su rendimiento durante horas pico. Esta estructura dificulta su escalabilidad, ya que cualquier modificación o incorporación de funcionalidades (como pedidos programados o seguimiento en tiempo real avanzado) puede afectar procesos existentes, propagando fallos que deriven en la interrupción total del servicio y comprometiendo su disponibilidad para usuarios y restaurantes. (Ortega, 2020).

Con base a lo que afirma Ortega se argumenta que las plataformas diseñadas con una base de datos únicas son más complejas en el manejo de los servicios, ya que estas para escalar o solicitar cambios se debe realizar un despliegue en toda la aplicación, situación que perjudica el desarrollo en la funcionalidad para los usuarios.

A continuación se enumeran algunas de sus disposiciones más esenciales:

• Utilizar vehículos y contenedores fabricados con materiales fáciles de limpiar y desinfectar.

• Un embalaje adecuado reduce la contaminación cruzada y evita el contacto directo con el suelo del vehículo.

• Mantener condiciones que eviten la contaminación y el deterioro durante el almacenamiento y el transporte.

• Mantener la cadena de frío para los productos que lo requieran, con control de la temperatura en cada etapa.

Metodología

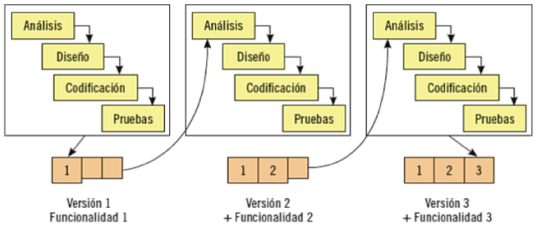
El desarrollo inicia con una base sólida que satisface un conjunto inicial de requisitos. Las iteraciones posteriores incorporan las funcionalidades pendientes, guiando una evolución progresiva del sistema por etapas. Este enfoque se emplea frecuentemente en entornos complejos que admiten lanzamientos iterativos del producto.

Entre sus ventajas se encuentra la posibilidad de experimentar con prototipos funcionales intermedios. Asimismo, la animación gráfica característica de este paradigma permite comprender su funcionamiento sin depender de documentación extensa, lo que en muchos casos facilita descartar versiones incrementales tempranas (Areba, 2001). De acuerdo con esta referencia, la metodología incremental se concibe como un mecanismo que gestiona procesos de cambio continuo en sistemas de aplicaciones..

Según (Lehman 1984) el modelo incremental supera la necesidad de una secuencia lineal en el desarrollo mediante un enfoque modular. En este enfoque, el sistema se construye progresivamente mediante la incorporación de módulos o capacidades funcionales, denominados incrementos. Cada fase sucesiva amplía el sistema con nuevas funcionalidades o requisitos, de modo que cada versión se fundamenta en la anterior e incorpora mejoras específicas.

(Mario G Piattini, 2018). Desacuerdo con Mario esta metodología restablece las asignaciones de tarea a cada rol que se va implementando en el software.

Figura 1 Metodología incremental y sus fases



Nota: Esta imagen nos muestra la funcionalidad del modelo incremental tomado de (Romero, 2023)

Desarrollo

Sugerimos un sistema basado en microservicios para abordar los problemas mencionados anteriormente, incluyendo la evaluación, el rendimiento, la gestión de restaurantes, repartidores y pedidos solicitados por el cliente, la autenticación y la gestión de perfiles. Estos se crean utilizando interfaces modulares y API, cada una de las cuales funciona de forma independiente de las demás.

Además, se utilizarán contenedores Docker para aislarlos, lo que nos permitirá incorporar todas las herramientas necesarias para un funcionamiento fiable y seguro en cualquier ordenador. Contenedores Docker Este marco integra FastAPI con Python para minimizar la duplicación de código y garantizar tiempos de respuesta óptimos, lo que facilita la identificación de errores y la adopción de medidas en el microservicio afectado sin poner en peligro otros servicios.

Gracias al desarrollo incremental, pudimos mantener un ritmo sostenible de entrega funcional, identificar errores desde el principio y validar periódicamente las decisiones técnicas. El producto final es un sistema modular y escalable que está preparado para futuras incorporaciones.

Planificación y comunicación.

Como se muestra en la siguiente tabla, se elaboran esquemas y se establecen especificaciones para iniciar el desarrollo mediante iteraciones una vez que se han

estudiado la plataforma mencionada anteriormente para la optimización de la entrega de pedidos.

Tabla 1 Requisitos Funcionales

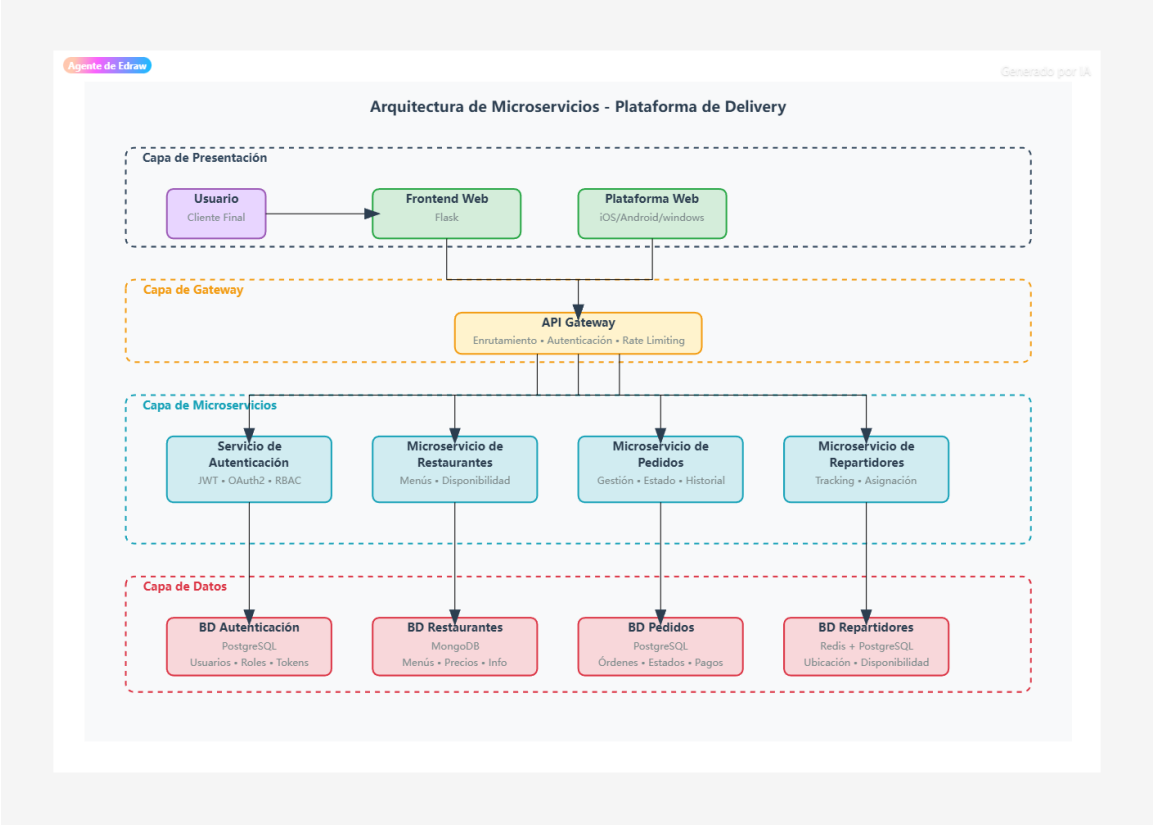
ID	Requisito	Descripción	Prioridad
RF-01	Registro de usuarios	Permitir registro de usuarios con roles	Alta
RF-02	Autenticación JWT	Login con generación de *access token*	Alta
RF-18	Logout	Revocación de *refresh token*	Media
RF-19	Refresh token	Renovación del *access token*	Media
RF-03	Gestión de restaurantes	CRUD de restaurantes	Alta
RF-04	Gestión de menú	Agregar, editar y eliminar ítems del menú	Alta
RF-05	Búsqueda de restaurantes	Búsqueda por nombre con filtro de texto.	Media
RF-06	Visualización de menú	Ver menú completo de un restaurante	Alta
RF-15	Subida de imágenes	Restaurantes pueden subir fotos	Media
RF-07	Creación de pedidos	Cliente crea pedidos seleccionando ítems	Alta
RF-08	Reserva atómica de stock	Sistema reserva stock automático	Alta
RF-09	Asignación de repartidor	Asignación automática de repartidor	Alta

ID	Requisito	Descripción	Prioridad
RF-14	Estados del pedido	Flujo: creado → asignado → completado.	Alta
RF-16	Cache con Redis	Cacheo de consultas	Media
RF-10	Dashboard de cliente	Ver historial de pedidos, estados y detalles.	Media
RF-11	Dashboard de restaurante	Ver pedidos recibidos y estadísticas	Alta
RF-12	Dashboard de repartidor	Ver pedido asignado y ganancias	Alta
RF-13	Completar entrega	Repartidor marca pedido completado	Alta
RF-17	API Gateway	Punto único de entrada con validación JWT	Alta
RF-20	Health checks	Endpoints de verificación de estado en todos los servicios.	Baja

Diagrama de arquitectura de proyecto.

Este diagrama muestra la arquitectura de una plataforma de reparto de comida a domicilio basada en microservicios: el usuario interactúa con la capa de presentación, que envía solicitudes al API Gateway para autenticación y enrutamiento, y este distribuye las peticiones a microservicios independientes (autenticación, restaurantes, pedidos y repartidores), cada uno con su propia base de datos. (ver Figura 2).

Figura 3 Diagrama de arquitectura de proyecto

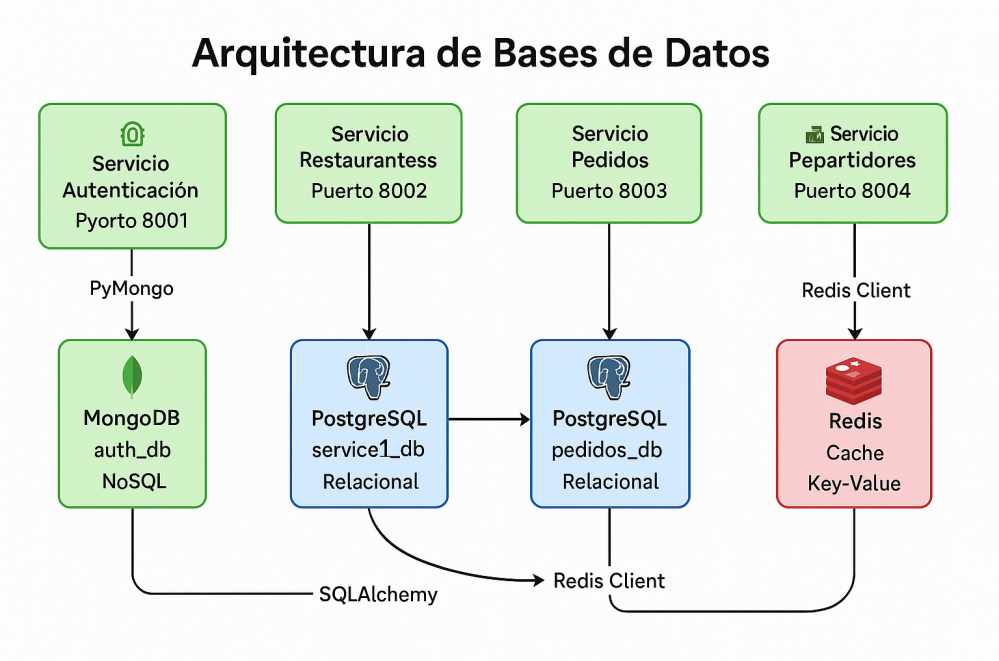


Nota. Figura elaborada por el autor usando EdrawMax (https://www.edrawmax.com/).

Diagrama de bases de datos.

A continuación, se presenta el diagrama de bases de datos la imagen describe cómo los microservicios se comunican con sus respectivas bases de datos usando diferentes tecnologías, y cómo Redis se integra como sistema de cache para optimizar el rendimiento del servicio de pedidos. (ver figura 3)

Figura 3 Diagrama de bases de datos



Nota. Figura elaborada por el autor usando EdrawMax (https://www.edrawmax.com/).

Incremento 1: Autenticación

Las siguientes tareas se construyen utilizando tecnologías como FastAPI, MongoDB y JWT para el siguiente microservicio de autenticación, que solo permite iniciar sesión de forma segura a los usuarios registrados.

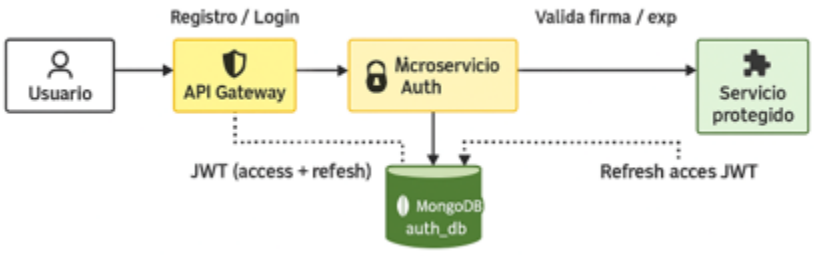
Desarrollos.

- ✓ Registro de usuarios
- ✓ Inicio de sesión
- ✓ Generador de jwt
- ✓ Verificación de credenciales

Diagrama de autenticación:

A continuación, se presenta diagrama de autenticación Diagrama de flujo de autenticación: el microservicio de autenticación verifica las credenciales en MongoDB y crea tokens JWT para un acceso seguro y una actualización una vez que el usuario se registra o inicia sesión a través de la API Gateway. (ver figura 4)

Figure 4 Diagrama de autenticación.



Nota. Figura elaborada por el autor usando EdrawMax (https://www.edrawmax.com/).

La creación de la autenticación de este código fuente explica: El endpoint POST /register valida que el email no exista y que la contraseña tenga mínimo 8 caracteres, luego hashea la contraseña con PBKDF2-SHA256 para seguridad, y lo inserta en la colección users de MongoDB, retornando {"message": "user created"} en caso de éxito o errores HTTP 409 (email duplicado) o 400 (contraseña corta) en caso contrario.

```
@app.post("/register", response_model=dict)
def register(user: UserCreate):
    """Registrar un nuevo usuario"""
    # Verificar si el email ya existe
    if users.find_one({"email": user.email}):
        raise HTTPException(status_code=409, detail="Email already
registered")

    # Validar longitud de contraseña
    if not user.password or len(user.password) < 8:
        raise HTTPException(
            status_code=400,
            detail="Password must be at least 8 characters long"
        )

    # Hashear contraseña y crear documento
    hashed = get_password_hash(user.password)
    user_doc = {
        "email": user.email,
        "password": hashed,
        "role": user.role,
        "created_at": datetime.utcnow(),
    }
    users.insert_one(user_doc)
    return {"message": "user created"}
```

Incremento 2: Microservicios de dominio

Se utilizan las tecnologías Python y FastAPI para crear los microservicios que gestionan los pedidos y la comunicación entre los repartidores y los clientes de los restaurantes. (ver figura 5).

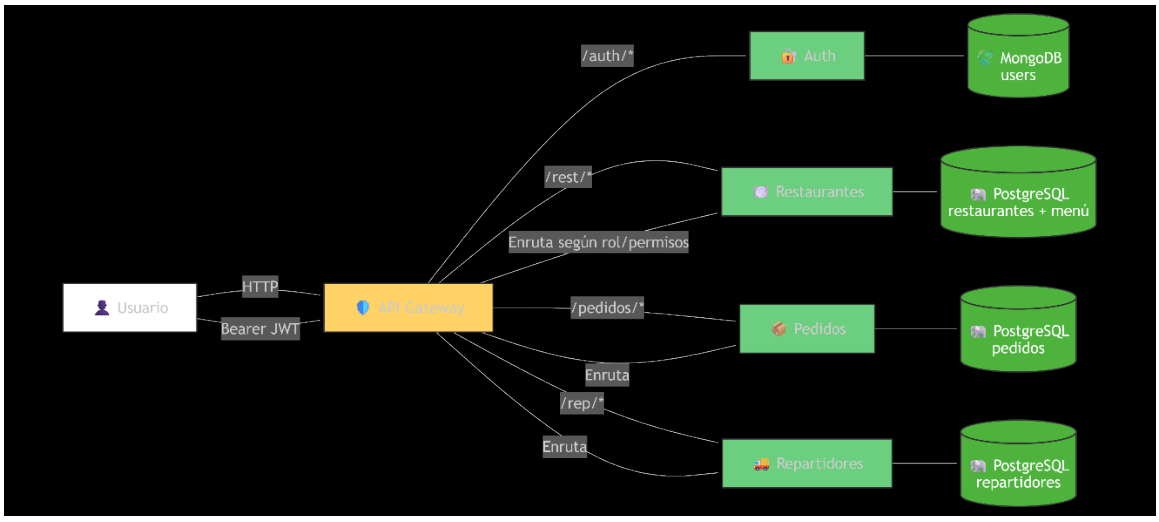
Desarrollos.

- ✓ Microservicio de Restaurantes.
- ✓ Microservicio Pedidos.
- ✓ Microservicio Repartidores.

Diagrama de microservicios.

El diagrama explica cómo el API Gateway recibe las solicitudes del usuario, valida el JWT y enruta las peticiones hacia los microservicios de Autenticación, Restaurantes, Pedidos y Repartidores, cada uno independiente.

Figura 5 Diagrama de Microservicios



Nota. Figura elaborada por el autor usando EdrawMax (https://www.edrawmax.com/).

El código fuente del microservicio Restaurants muestra cómo crearlo utilizando Python , El archivo Dockerfile crea una imagen Docker que inicia uvicorn main:app en el puerto 8002, descarga el código fuente (main.py, models.py, database_sql.py) e instala las dependencias de requirements.txt (FastAPI, SQLAlchemy, PostgreSQL). Esto utiliza SQLAlchemy para crear automáticamente las tablas restaurantes y menu_items en PostgreSQL, y luego las rellena con tres restaurantes de prueba y sus menús.

Código fuente para la creación del microservicio restaurantes

```
# 1. Docker construye la imagen desde el Dockerfile
docker build -t restaurantes-service .

# 2. Docker Compose levanta el contenedor
docker-compose up restaurantes-service

# 3. Uvicorn inicia FastAPI (main.py)
# 4. FastAPI ejecuta @app.on_event("startup")
# 5. Se crean tablas en PostgreSQL
# 6. Se insertan datos de prueba (seed)
# 7. Microservicio listo en puerto 8002
```

El procedimiento para crear el Microservicio Pedidos que muestra en el código fuente donde el Dockerfile crea una imagen Docker con Python 3.12 que instala FastAPI, SQLAlchemy y requests, ejecuta uvicorn main:app en puerto 8003, el cual al iniciar crea las tablas órdenes y ordenes_items en PostgreSQL, y expone el endpoint POST /api/v1/pedidos que valida stock en el servicio Restaurantes

Código Fuente indica creación del microservicio pedido

```
# Construir solo el servicio de pedidos
docker-compose build pedidos-service

# Construir e iniciar el servicio con sus dependencias (PostgreSQL, Redis)
docker-compose up --build pedidos-service

# Construir todos los servicios del proyecto
docker-compose build

# Iniciar todos los servicios
docker-compose up -d
```

El archivo Dockerfile genera una imagen Docker con Python 3.12 que instala FastAPI y SQLAlchemy, y luego ejecuta uvicorn main:app en el puerto 8004, lo que crea la tabla de controladores de entrega en PostgreSQL con campos de estado (disponible/ocupado).

Código Fuente indica creación del Microservicio de repartidores.

```
# Construir solo el servicio de repartidores
docker-compose build repartidores-service

# Construir e iniciar el servicio con sus dependencias (PostgreSQL)
docker-compose up --build repartidores-service

# Construir todos los servicios del proyecto
docker-compose build

# Iniciar todos los servicios
docker-compose up -d
```

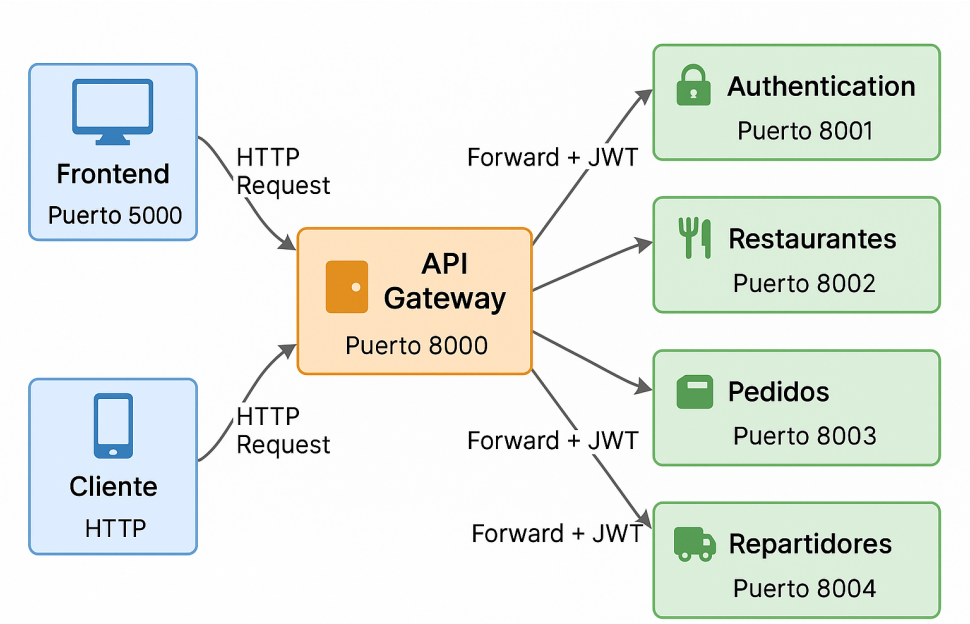
Incremento 3: API Gateway

Las rutas genéricas protegidas con JWT se aplican a través de API Gateway para controlar el acceso seguro a los servicios.

Diagrama de API Gateway.

A continuación, se presenta la arquitectura de un sistema orientado a microservicios se muestra en el diagrama la API Gateway (FastAPI en el puerto 8000) que sirve como punto de entrada, gestionando las solicitudes HTTP del cliente y del frontend y reenviándolas de forma segura mediante JWT a los servicios internos (autenticación, restaurantes, pedidos y personal de reparto). (ver figura 6).

Figura 6 Diagrama de Api Gateway



Nota. Figura elaborada por el autor usando EdrawMax (https://www.edrawmax.com/).

En el siguiente código de creación de API Gateway crea una puerta de enlace API con FastAPI importando dependencias, validando JWT, reenviando solicitudes HTTP a microservicios mediante solicitudes, habilitando CORS con CORSMiddleware y configurando el registro en el nivel INFO para documentar los procesos de Docker.

Código Fuente indica creación de API Gateway.

```
from fastapi import FastAPI, APIRouter, Request, HTTPException
from fastapi.responses import JSONResponse
from jose import JWTError, jwt
import os
from fastapi.middleware.cors import CORSMiddleware
import requests
import logging

# Define la instancia de la aplicación FastAPI.
app = FastAPI(title="API Gateway Taller Microservicios")

# configure basic logging to stdout so container logs show our debug
prints
logging.basicConfig(level=logging.INFO)
logger = logging.getLogger("api-gateway")
```

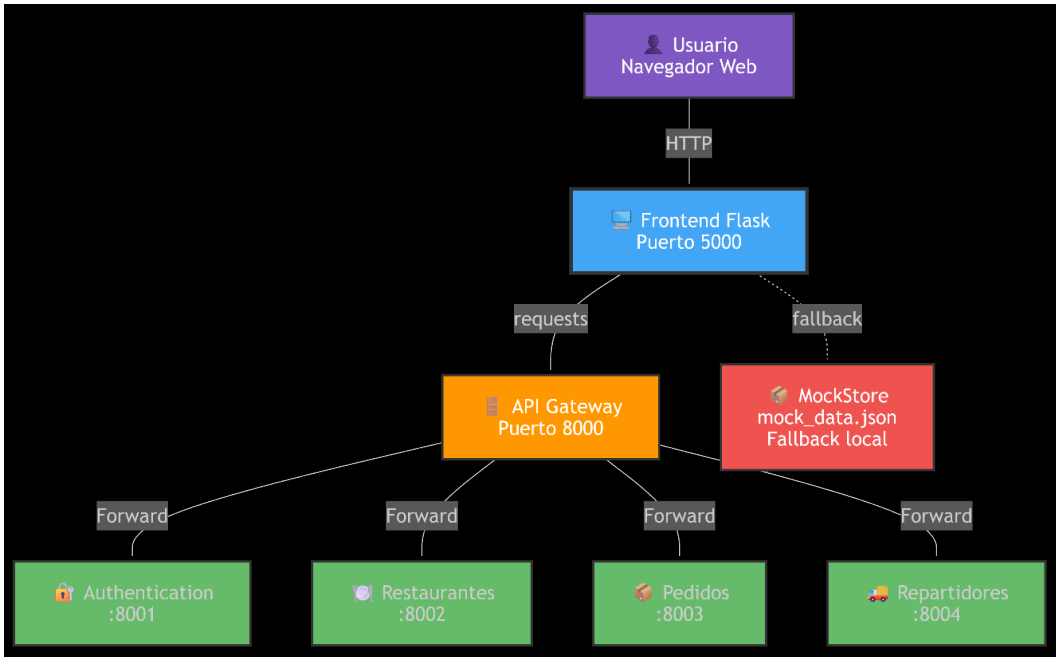
Incremento 4: Frontend

El framework Flask se utiliza para el desarrollo frontend, ya que es ligero y adaptable, lo que permite a los usuarios iniciar sesión y navegar rápida y fácilmente por los distintos microservicios que ofrece la plataforma.

Diagrama de Frontend.

A continuación, se presenta diagrama del Frontend este diagrama muestra cómo el usuario interactúa con el Frontend Flask, que envía solicitudes al API Gateway para distribuirlas a los microservicios (autenticación, restaurantes, pedidos y repartidores), mientras mantiene un fallback local con MockStore en caso de falla en la conexión (ver figura 7)

Figura 7 Diagrama de Frontend.



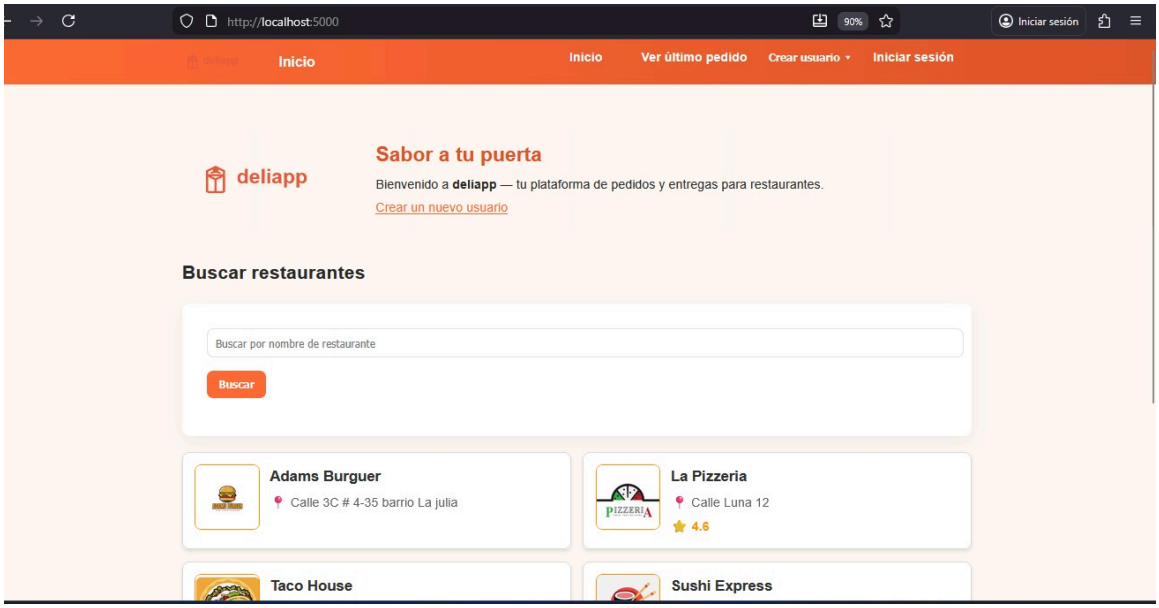
Nota. Figura elaborada por el autor usando EdrawMax (https://www.edrawmax.com/)

En el siguiente código para realizar la creación del fronted importa dependencias Flask para crear la aplicación web (templates, sesiones, requests HTTP), configura la instancia Flask, obtiene la URL del API Gateway desde variable de entorno (default http://localhost:8000), y establece una secret key para firmar las sesiones Flask que almacenarán el token JWT del usuario

Código Fuente indica creación de Fronted.

```
# /frontend/app.py

from flask import Flask, render_template, request, redirect, url_for,
session, jsonify
import os
import requests
from flask import flash
import json
import uuid
import threading
from datetime import datetime

app = Flask(__name__)

# Obtén la URL del API Gateway desde las variables de entorno.
# Esta variable debe estar configurada en el docker-compose.yml.
API_GATEWAY_URL = os.getenv("API_GATEWAY_URL", "http://localhost:8000")

# Secret key for session (only for dev). In production set a strong secret
via env.
app.secret_key = os.getenv("FLASK_SECRET", "dev-secret-change-me")
```

Figure 8 Pagina de inicio frontend



fuente: Andres Hincapie, Erika Rincón

Incremento 5: Despliegue y orquestación

Este aumento se implementa utilizando contenedores Docker con las especificaciones requeridas para el desarrollo de la plataforma.

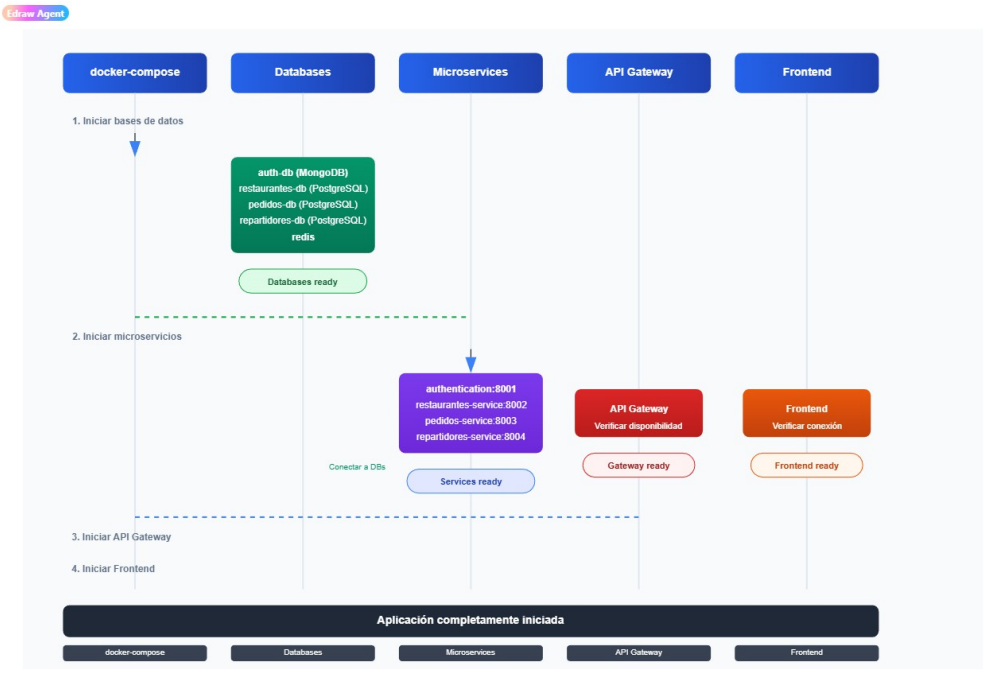
Desarrollos

- ✓ Contenedorizacion con Docker.
- ✓ Documentación

Orquestación con Contendores usando Docker.

El siguiente diagrama ilustra la arquitectura en contenedores donde cada capa (Frontend, Gateway, Microservicios, Bases de Datos y Almacenamiento) se ejecuta dentro de la **red Docker**, permitiendo comunicación interna segura y aislada (ver Figura 9)

Figura 9 Diagrama de despliegue y orquestación



Nota. Figura elaborada por el autor usando EdrawMax (https://www.edrawmax.com/

Código Fuente de orquestación de contenedores.

```
  repartidores-service:
    build: ./services/repartidores
    container_name: repartidores-service
    env_file:
      - .env
    ports:
      - "8004:8004"
    environment:
      - DATABASE_URL=postgresql://user:password@repartidores-
db:5432/repartidores_db
    depends_on:
      - repartidores-db
```

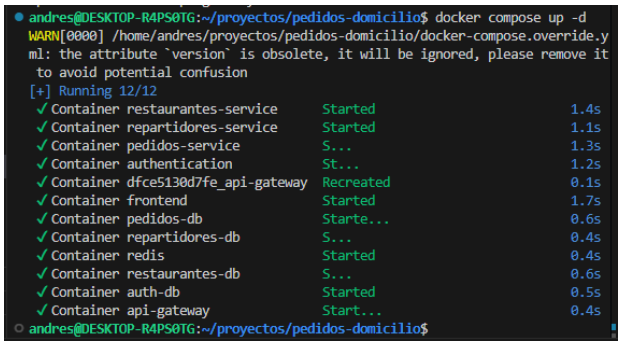
fuente: Andres Hincapie, Erika Rincón

Figura 40 Creación de contenedores

```
537   #39 [repartidores-service] exporting to image
538   #39 exporting layers
539   #39 exporting layers 0.5s done
540   #39 writing image sha256:b2ef468523ea13d0dbf3daa01e3e3ee2ebd6
541   #39 naming to docker.io/library/pedidos-domicilio-repartidore
542   #39 DONE 0.5s
543
544   #40 [repartidores-service] resolving provenance for metadata
545   #40 DONE 0.0s
```

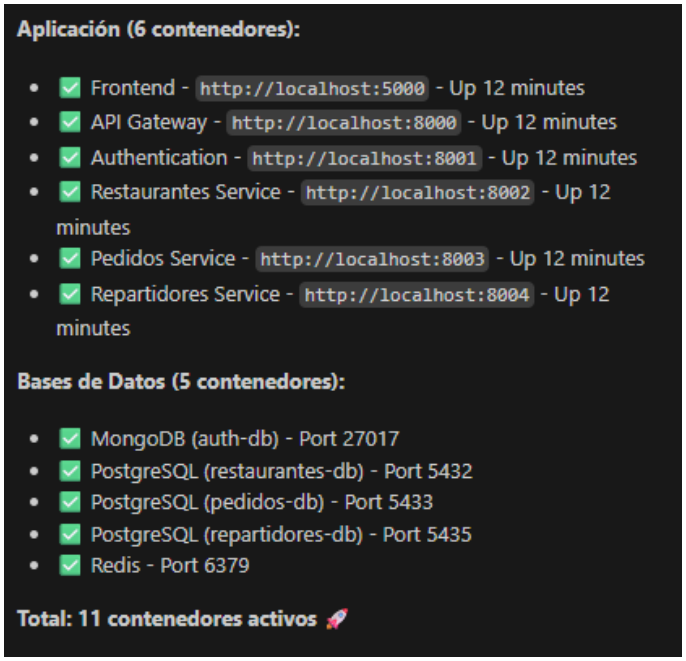
fuente: Andres Hincapie, Erika Rincón

Figura 51 En esta imagen muestra el proceso de levantar servicios de contenedores con Docker compose up -d



fuente: Andres Hincapie, Erika Rincón

Figura 12 Contenedores en Docker se verifica estado con: docker compose ps



Fuente: Aplicación Deliapp vista desde visual Studio Code

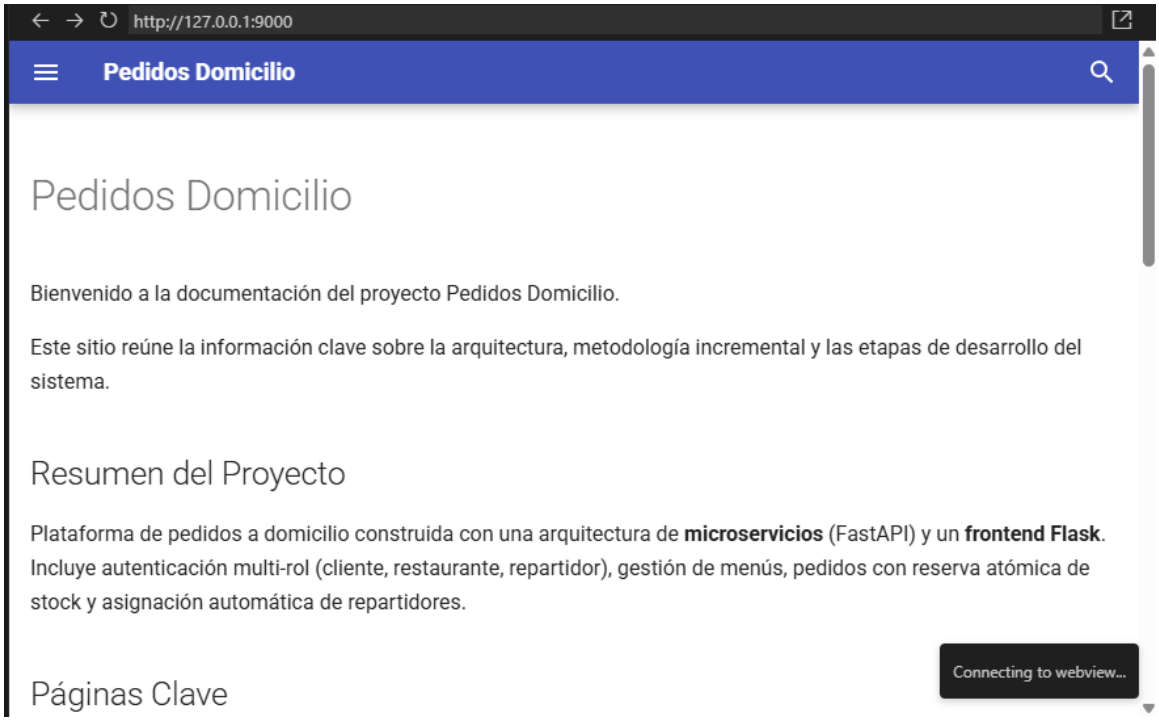
Documentación.

A continuación, se muestra documentación del código mediante MKDocs, (ver figura 13).

Código indica la creación de documentación en MKDocs.

```
site_name: Pedidos Domicilio
site_description: Plataforma de pedidos a domicilio basada en
microservicios.
site_url: https://github.com/andreshi40/pedidos-domicilio
repo_name: andreshi40/pedidos-domicilio
repo_url: https://github.com/andreshi40/pedidos-domicilio
edit_uri: ''

nav:
  - Inicio: index.md
  - Metodología:
      - Flujo Incremental: INCREMENTAL.md
      - Etapas Incrementales: ETAPAS_INCREMENTALES.md
      - Estado del Arte: estado-del-arte.md
  - Arquitectura:
      - Descripción: architecture.md
      - Diagrama Mermaid: architecture.mmd
      - ASCII Layout: architecture-ascii.txt
      - Caso de Uso (Pedidos): use-case.md
  - Diagramas Técnicos:
      - Bases de Datos: diagrama-bases-datos.md
      - Arquitectura del Sistema: diagrama-arquitectura.md
      - Autenticación: diagrama-autenticacion.md
      - API Gateway: diagrama-api-gateway.md
      - Frontend: diagrama-frontend.md
      - Orquestación Docker: diagrama-docker-orquestacion.md

theme:
  name: material
```

Figura 13 Documentación de código



fuente: Andres Hincapie, Erika Rincón

Conclusiones

Con el diseño basado en microservicios que garantiza la escalabilidad, la eficiencia y una mejor experiencia de usuario, DELIAPP se ofrece como solución tecnológica viable para optimizar los servicios de entrega de comida a domicilios en el municipio de Barbacoas-Nariño. La capacidad del sistema para adaptarse y crecer se ve reforzada por el uso de tecnologías contemporánea como Python y bases de datos hibridas.

Al digitalizar los procedimientos y aumentar los canales de ventas, la plataforma no solo mejora la gestión operativa de los restaurantes y los repartidores, sino que estimula la economía local permitiendo tener un sistema tecnológico que se ajuste a las necesidades del municipio. Se recomienda realizar pruebas pilotos para verificar el funcionamiento y evaluar métricas como los tiempos de respuesta, la satisfacción de los clientes y la estabilidad del sistema

Referencias

Areba, J. B. (2001). *Metodologia del analisis estructurado de sistemas.* Madrid: Belen Recio.

Benavides, C. (2024). *Contexto Municipio de Barbacoas -Nariño.* PASTO.

Mario G Piattini, F. G. (2018). *Calidad de Sistema de Informacion 4ta Edicion.* Jarama-Madrid: Ra-ma Editorial.

Muñoz, P. C. (2009). *Plataformas de teleformación y herramientas telemáticas.* España.

Ortega, J. M. (2020). *Tecnologías para arquitecturas basadas en microservicios.* (J. M. Ortega, Ed.)

Rocio , E. (2005). *Trabajo colaborativo en educación universitaria:.* (N. E. Educativa, Ed.) Mexico.

Romero, J. L. (2023). *Desarollo de Componentes Software para Tareas administrativas de Sistemas.* Antequera- Malaga: IC Editorial.

Social, M. d. (2013). *Resolucion 2674.* Bogota.

Sommerville, I. (2005). *Ingenieria del Software.* Madrid- España: Pearson Educacion S.A.

Unesco. (2024). (UNESCO, Ed.) Obtenido de https://www.google.com.co/books/edition/Informe_de_seguimiento_de_la_educaci%C3%B3n/gNINEQAAQBAJ?hl=es&gbpv=0

Usaola, M. (2015). (M. P. Usaola, Ed.) España. Obtenido de https://www.google.com.co/books/edition/_/UiAvCwAAQBAJ?hl=es-419&gbpv=0