Proyecto de Diseño y Documentación de Arquitectura: Plataforma de Comercio Electrónico Basada en Microservicios

Roger Fabian Moreno Ruiz

Universidad Cesuma

Maestría en Ingeniería de Software

Profesor: Jose Oswaldo Perez Cruz

Colombia, Julio 2025

Contenido

1. Introducción

La rápida evolución de la tecnología y la digitalización de los procesos comerciales han impulsado la necesidad de desarrollar sistemas de software robustos, escalables y adaptables. En este contexto, el presente documento expone de manera integral el desarrollo y la documentación de una arquitectura de software orientada a una plataforma básica de comercio electrónico. El objetivo es abordar las demandas actuales del sector, como la flexibilidad, la alta disponibilidad y la capacidad de integración con diversos componentes y servicios externos.

Para lograrlo, se ha optado por la adopción de una arquitectura basada en microservicios, un enfoque que permite descomponer la aplicación en módulos independientes, cada uno responsable de una funcionalidad específica. Esta elección responde a buenas prácticas modernas de ingeniería de software y permite una mejor administración del ciclo de vida del producto, facilita la escalabilidad horizontal, el despliegue continuo y la resiliencia frente a fallos de componentes individuales.

El diseño y la documentación se fundamentan en conceptos, principios y metodologías obtenidos de los textos de estudio sugeridos, los cuales proporcionan un marco teórico sólido para la toma de decisiones arquitectónicas. Adicionalmente, se ha enriquecido el trabajo integrando información y tendencias actuales provenientes de fuentes académicas y especializadas encontradas en Internet, asegurando así la pertinencia y actualidad de la propuesta. De esta manera, el documento no solo cumple con los lineamientos académicos, sino que también incorpora las mejores prácticas y recomendaciones vigentes en el ámbito de software arquitectónico.

2. Descripción del Sistema

La plataforma de comercio electrónico se compone de los siguientes microservicios independientes, cada uno con responsabilidades bien definidas:

2.1. Gestión de Productos

- Funcionalidad: Creación y administración de productos, incluyendo atributos como cantidad en inventario y precio unitario.
- Capacidades clave: Actualización en tiempo real de stock y precios.

2.2. Gestión de Usuarios

- Funcionalidad: Altas, bajas y modificaciones de perfiles de usuarios.
- Capacidades clave: Mantenimiento seguro de datos personales y roles.

2.3. Procesamiento de Pedidos

- Funcionalidad: Creación de pedidos vinculando productos seleccionados y usuarios registrados.
- Capacidades clave: Cálculo automático de costos, gestión de despacho y seguimiento.
- ***2.2. Gestión de Usuarios***
- Funcionalidad: Altas, bajas y modificaciones de perfiles de usuarios.
- Capacidades clave: Mantenimiento seguro de datos personales y roles.

2.2. Gestión de Usuarios

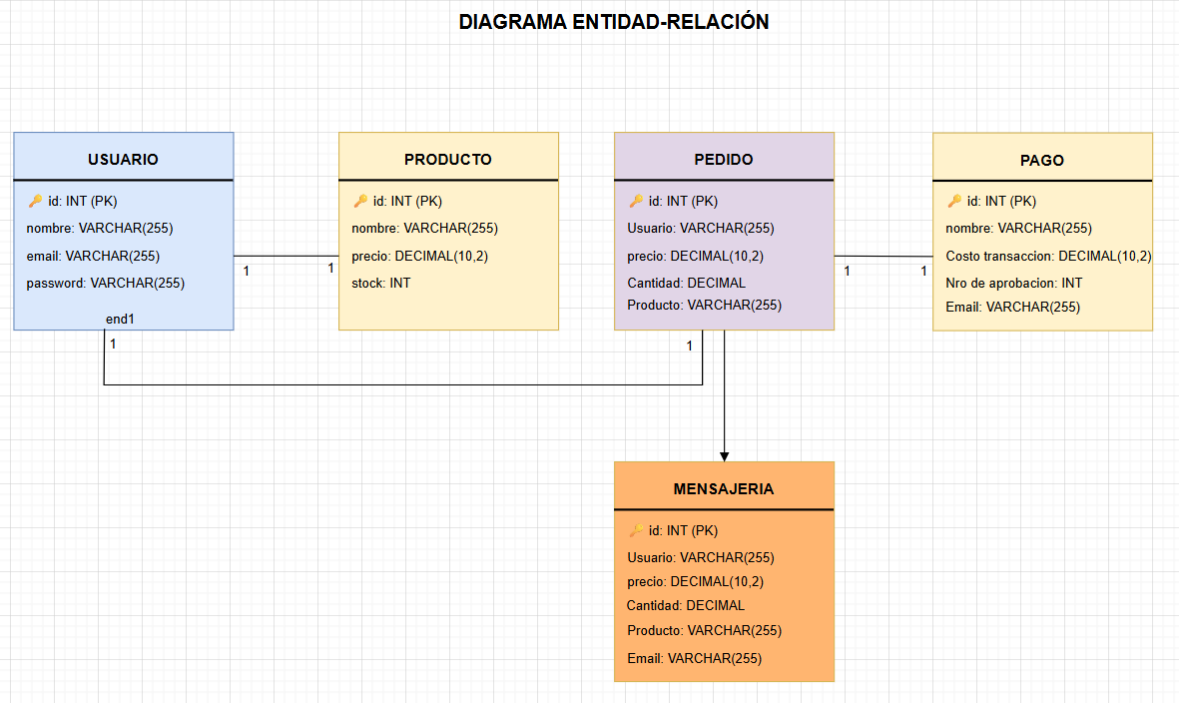
- Funcionalidad: Altas, bajas y modificaciones de perfiles de usuarios.
- Capacidades clave: Mantenimiento seguro de datos personales y roles**.**

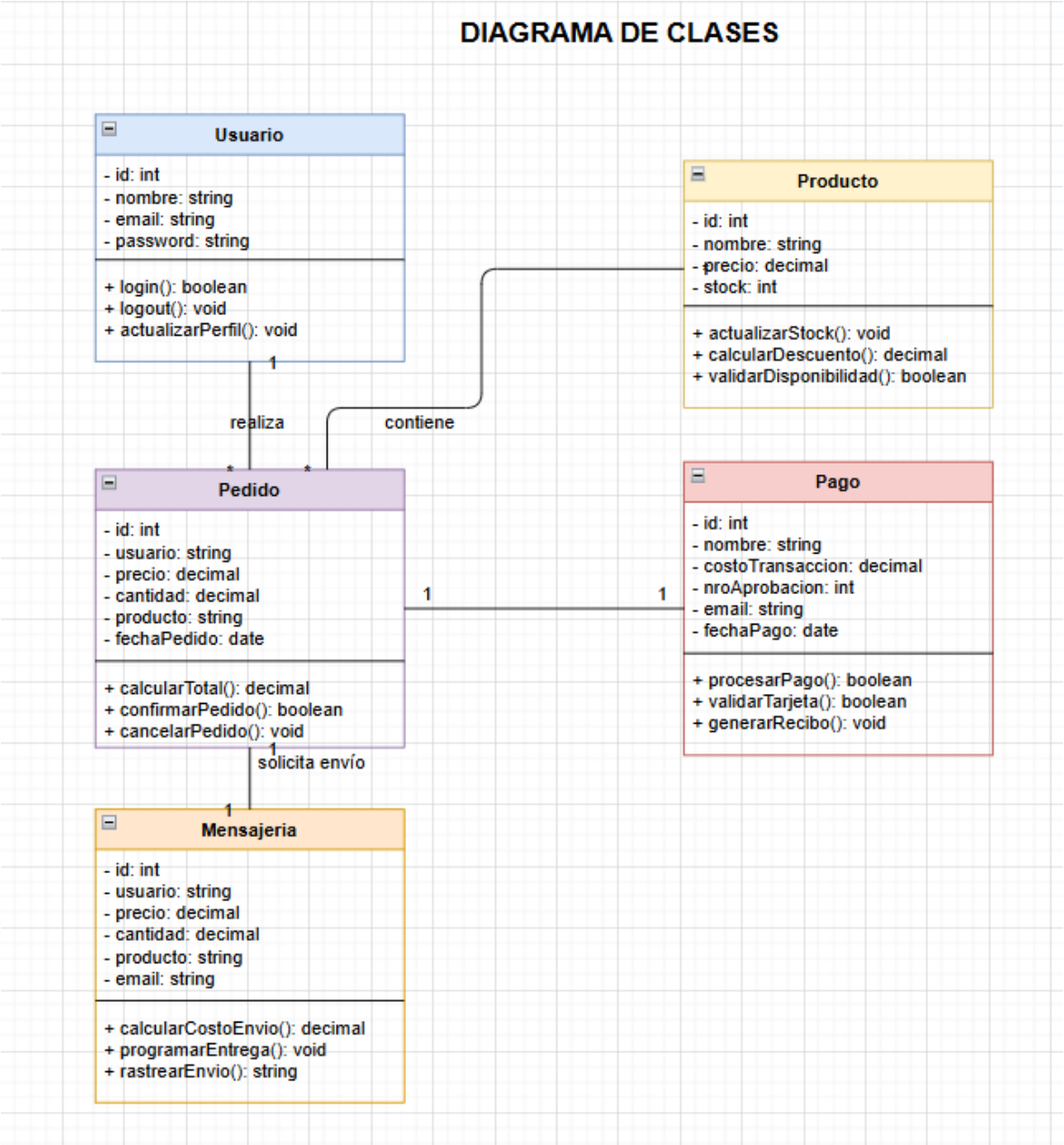
Cada microservicio posee su propia base de datos y lógica de negocio, y se comunica mediante APIs REST y mensajería asíncrona.

3. Modelado y Diseño

3.1. Diagrama de Arquitectura

La arquitectura propuesta para la plataforma de comercio electrónico se ilustra utilizando diagramas UML (Unified Modeling Language). Esta metodología facilita la visualización de los componentes, relaciones y flujos de interacción entre los microservicios, aportando claridad y formalismo a la documentación técnica del sistema.





3.1.1. Componentes Principales del Sistema

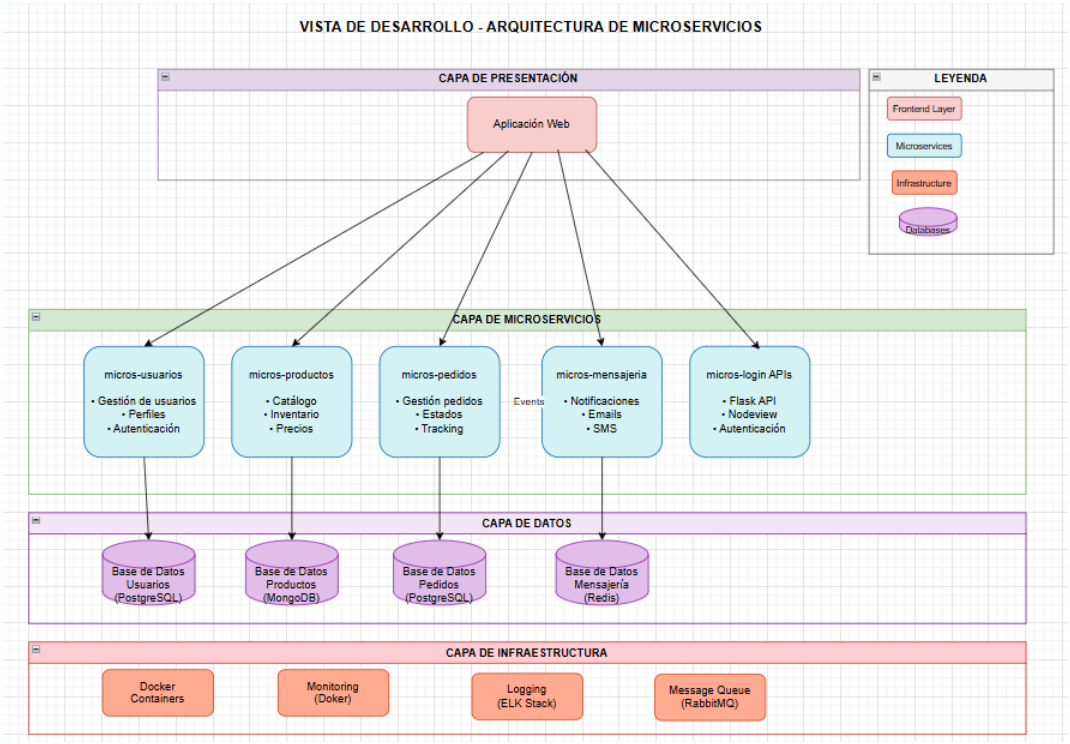
- **Microservicio de Gestión de Productos:** Administra el catálogo de productos. Incluye clases y operaciones relacionadas con el registro, modificación, desactivación y consulta de dichos productos.
- **Microservicio de Gestión de Usuarios:** Controla el ciclo de vida de usuarios, abarcando registro, validación, autenticación y actualización de

perfiles, modelado a través de clases como Usuario, Perfil y Token de sesión.

- **Microservicio de Procesamiento de Pedidos:** Orquesta la recepción, validación, procesamiento y seguimiento de los pedidos. Interactúa con los microservicios de productos para validar stock y con usuarios para identificar el cliente.
- **Microservicio de Notificaciones:** Suscribe eventos generados por los pedidos (mediante mecanismos asíncronos como RabbitMQ), y gestiona el envío de alertas por correo electrónico o SMS simulados.

3.1.2. Detalle Ampliado de la Interacción (Modelado UML y Comunicación)

- **Diagrama de Componentes (UML)**

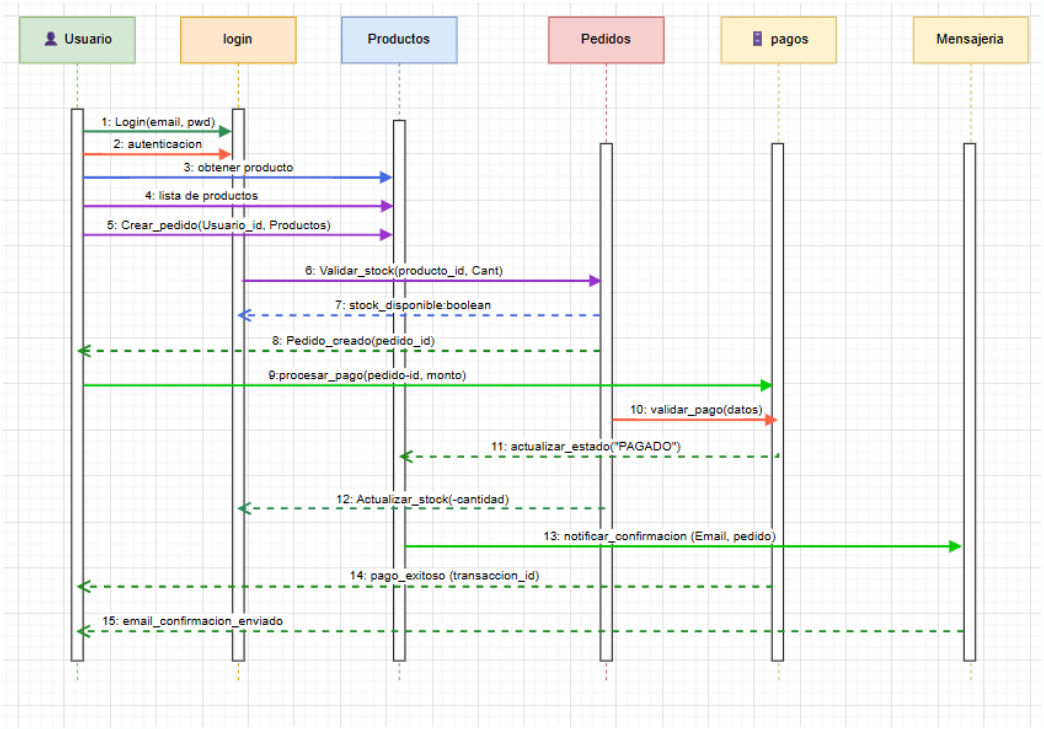


El diagrama de componentes incluye los siguientes elementos:

- **Clientes**: Representan usuarios finales que interactúan mediante una interfaz web o móvil.
- **Servicio de API Gateway**: Opcional según diseño, centraliza las solicitudes y distribuye hacia los microservicios.

- **Microservicios**: Cada uno modelado como un componente independiente con su propia base de datos y lógica.
- **Bus de Mensajes (RabbitMQ)**: Facilita la comunicación asíncrona entre los servicios, especialmente útil para integración de notificaciones y procesamiento de eventos.

3.1.3. Detalle de Interacciones (Secuencia UML)



- **Inicio del Flujo**

  El usuario ejecuta una acción de compra sobre la interfaz de la tienda (cliente).
- **Solicitud de Pedido**

  El cliente envía una solicitud al microservicio de Procesamiento de Pedidos.
- **Validación de Inventario**

  El microservicio de pedidos consulta de forma síncrona al microservicio de productos utilizando HTTP/REST, aplicando patrones UML de dependencias y asociaciones entre servicios para validar existencia y disponibilidad del producto.
- **Procesamiento y Registro**

Si hay stock, el microservicio de pedidos procesa el pago (real o simulado) y crea el registro del pedido. La lógica se modela mediante clases “Pedido”, “DetallePedido” y “Pago”.

- **Emisión de Evento**
  Una vez confirmado el pedido, el microservicio de pedidos publica un evento en el bus de mensajes (según el patrón Publish/Subscribe modelado en UML con señales y eventos).
- **Consumidor de Eventos**

El microservicio de notificaciones, suscrito a este canal, recibe el evento, extrae datos relevantes y procede al envío de notificación por los canales configurados (correo/SMS).

- **Respuesta al Usuario**
  El usuario es notificado del estado del pedido a través de la interfaz, tras la ejecución satisfactoria del flujo.

3.1.4. Aspectos Técnicos de Comunicación

- **HTTP/REST:** Usado para consultas directas y validaciones (por ejemplo, pedidos consulta a productos).
- **Mensajería Asíncrona (RabbitMQ):** Utilizada para desacoplar la generación y consumo de eventos, permitiendo escalabilidad y tolerancia a fallos en los servicios de notificaciones.

2E3.2 Justificación del Patrón Arquitectónico

Se opta por microservicios porque:

- *Facilitan la escalabilidad y el despliegue independiente.*
- *Permiten que cada servicio evolucione con tecnologías distintas.*
- *Mejoran la resiliencia ante fallos y soportan demandas variables.*

Según textos de referencia, la arquitectura de microservicios responde a necesidades de alta escalabilidad, despliegue continuo y autonomía modular.

.

3.3 Documentación de Microservicios

Microservicio	Propósito	Lenguaje y Framework	Base de Datos	Comunicación
Login, Productos, Usuarios, Pedidos	Gestión CRUD de productos, Usuarios y pedidos.	Node.js + Express + Python + Flask	MySQL	REST API
Notificaciones	Envío de notificaciones ante eventos (pedido creado, etc.)	Python + Flask	MySQL	Asíncrona/Eventos

4. Implementación

Cada microservicio es implementado de forma independiente, por ejemplo:

- **Node.js + Express** para Usuarios, login, productos y pedidos rápido desarrollo en la parte grafica visual de cara al usuario por tener buena integración con mysql y sistemas de mensajería.
- **Python + Flask** para para usuarios, login, productos y pedidos y notificaciones en el modelo y el controlador por ser simple y robusto para manejo de autenticación.

Se utilizan contenedores (Docker) para despliegue y aislamiento.

4.1 Comunicación entre Microservicios

- **HTTP/REST** para la mayoría de las operaciones CRUD y autenticación.

- **RabbitMQ** (o equivalente) para eventos: por ejemplo, cuando un pedido es creado, el microservicio de pedidos publica un mensaje; el servicio de notificaciones lo consume y envía la notificación correspondiente.
- La comunicación asíncrona mejora la resiliencia y la escalabilidad bajo cargas altas

4.2 Mecanismos de Resiliencia

- **Retry y fallback**: en caso de fallos de comunicación o respuesta lenta, los microservicios implementan mecanismos de reintentos y degradación controlada.
- **Desacoplamiento**: los errores en un microservicio no afectan directamente a otros.

5. Calidad y Evaluación de la Arquitectura

5.1 Pruebas Unitarias

Cada microservicio incluye pruebas unitarias para su lógica principal, validando rutas, operaciones de base de datos y manejo de errores.

5.2 Métricas de Calidad

- **Latencia de respuesta**: tiempo promedio de procesamiento de una solicitud (objetivo <30ms).
- **Tasa de errores**: porcentaje de solicitudes fallidas sobre el total de operaciones.
- Las métricas son registradas y revisadas periódicamente para garantizar el cumplimiento de los niveles de servicio establecidos.

5.3 Evaluación de Escalabilidad

- La arquitectura basada en microservicios implementada en este proyecto permite abordar la escalabilidad de manera flexible y eficiente. Cada

microservicio corre de forma aislada dentro de un contenedor Docker, lo que facilita su despliegue, actualización y manejo independiente.

- **Escalabilidad horizontal:**
  Cada microservicio podria escalarse horizontalmente aumentando el número de instancias que ejecutan el mismo servicio. Este enfoque permite soportar incrementos de carga o de usuarios distribuyendo el tráfico entre varias réplicas idénticas del microservicio, optimizando el uso de recursos y mejorando la tolerancia a fallos.
- **Orquestación y alta disponibilidad:**
  En la fase actual del proyecto, la gestión y despliegue de instancias se realiza mediante Docker Compose, adecuado para entornos de desarrollo y pruebas. Para producción o ambientes de alta disponibilidad, es recomendable agregar una capa de orquestación—mediante herramientas como **Kubernetes** o **Docker Swarm**—que permite:
  - Automatizar el escalado horizontal de instancias según la demanda (autoescalado).
  - Balancear la carga entre réplicas.
  - Recuperar automáticamente servicios ante caídas o fallos en alguna instancia.
  - Realizar actualizaciones sin interrupciones (rolling updates).
- **Ventajas del enfoque:**
  - Permite que cada componente crítico del sistema (como API de usuarios, productos, pedidos, login, mensajería, etc.) responda de manera independiente al crecimiento en el uso.
  - Facilita la asignación de recursos específicos para los servicios más demandados, optimizando costos.
  - Sienta las bases para el despliegue en infraestructura de nube moderna, donde la escalabilidad y la resiliencia son prioritarias.

En resumen, la solución está preparada para escalar horizontalmente gracias al aislamiento de microservicios en contenedores. Si la demanda lo requiere, basta

con ajustar la configuración del orquestador para que despliegue más instancias de aquel microservicio cuya capacidad deba aumentar, asegurando así el crecimiento gradual, seguro y automatizado de toda la plataforma.

6. Desafíos y Soluciones

- A lo largo del desarrollo y despliegue de la arquitectura de microservicios, se presentaron diversos desafíos técnicos y de diseño, que fueron afrontados y resueltos mediante decisiones arquitectónicas y tecnológicas. Se resumen a continuación los principales retos y las estrategias aplicadas:

- **Consistencia de datos:** Debido a la naturaleza distribuida de los microservicios y sus respectivas bases de datos, se optó por un enfoque de **consistencia eventual** en vez de consistencia estricta. Se eligieron patrones como **SAGA** para el manejo de transacciones distribuidas, permitiendo coordinar operaciones entre servicios de manera robusta y asíncrona. Así, cada servicio gestiona sus propios datos, y las compensaciones de errores entre pasos de una transacción se resuelven a través de mensajes o eventos, brindando resiliencia y tolerancia ante fallos parciales.
- **Definición de lenguajes de programación:** La selección de lenguajes priorizó aquellos **ligeros y ampliamente compatibles con contenedores**. Para la lógica backend y APIs, se utilizó **Python con Flask**, dada su sencillez, bajo consumo de recursos y amplia comunidad en el ámbito de microservicios. Para las vistas y frontends individuales, se eligió **Node.js**, permitiendo rapidez en el desarrollo, escalabilidad horizontal y facilidad para empaquetar cada servicio en su propio contenedor, además de mayor afinidad con tecnologías web modernas.
- **Motor de bases de datos:** En la etapa inicial, se utilizó **SQLite** por su simplicidad y facilidad de configuración en entornos de desarrollo. Sin embargo, al aumentar la complejidad de las relaciones entre entidades y requerirse operaciones concurrentes o multiusuario, SQLite presentó limitaciones (principalmente en la creación de nuevas tablas y relaciones

foráneas). Por ello, se decidió migrar a **MySQL**, aprovechando su robustez, soporte de transacciones, integridad referencial y la familiaridad del equipo con esta tecnología, lo cual facilitó la evolución de la estructura de datos y el manejo eficiente de múltiples servicios conectados a la misma base.

- **Autenticación:** En una primera iteración, los passwords se guardaban en texto plano, lo que representaba un serio riesgo de seguridad. Se implementó el **hash MD5** para almacenar de manera irreversible las contraseñas, mejorando la privacidad de los usuarios y evitando que una vulneración de la base de datos expusiera las claves reales. Aunque se optó por MD5 por simplicidad y compatibilidad en la fase prototipo, para entornos productivos se recomienda avanzar a hashes más robustos como bcrypt o Argon2.
- **Código de Front end:** El desarrollo frontend fue inicialmente abordado con **Spring Boot**, una poderosa herramienta para aplicaciones Java, pero surgieron limitaciones de conocimiento técnico y dificultades para la contenerización modular de cada interfaz. Por estos motivos, se migró a **Node.js** para los frontends de cada microservicio, facilitando la creación de contenedores independientes, acelerando los ciclos de desarrollo y permitiendo mayor flexibilidad en la integración con APIs REST y tecnologías JAvaScript modernas.
- **Monitorización:** Durante el despliegue se planificó la integración de soluciones como **Prometheus** y **Grafana** para la observabilidad y el monitoreo de servicios. Sin embargo, se encontraron desafíos para su montaje completo en esta fase del proyecto, principalmente por el tiempo y complejidad implicados en el despliegue inicial. Como solución temporal, se recurre al monitoreo mediante **logs de contenedores Docker** y los mecanismos de **health check** definidos en docker-compose, lo que permite verificar el estado operativo básico de cada microservicio. Está previsto que en fases futuras, especialmente con la migración a orquestadores como Kubernetes, se incorpore Prometheus y Grafana en una arquitectura de monitoreo plenamente automatizada y escalable.

- **En síntesis**, cada desafío abordado durante el proyecto ha reforzado la arquitectura para lograr una mayor independencia, seguridad y escalabilidad de los servicios, sentando las bases para futuros avances y mejoras en observabilidad, calidad y automatización del ciclo de vida de los microservicios.

7. Conclusiones

La arquitectura de microservicios ofrece una solución flexible, escalable y resiliente para aplicaciones de comercio electrónico, permitiendo adaptarse fácilmente a los cambios y crecimiento del sistema. Este enfoque modular no solo soporta demandas variables y picos de tráfico —escalando únicamente los servicios necesarios— sino que también garantiza una mayor disponibilidad, ya que los fallos en un servicio particular no afectan la totalidad de la plataforma

El diseño desacoplado facilita la evolución tecnológica y la introducción de nuevas funciones con menor riesgo, ya que los componentes pueden actualizarse y desplegarse de manera independiente sin interrumpir otros servicios críticos.Esta autonomía acelera los ciclos de desarrollo e integración, reduce el tiempo de llegada al mercado de nuevas funcionalidades y promueve la innovación continua.

La mantenibilidad y la flexibilidad mejoran notablemente: equipos pequeños y especializados pueden gestionar cada servicio de manera independiente, aplicando tecnologías, lenguajes y herramientas más apropiadas para cada contexto.

No obstante, el éxito de una solución basada en microservicios requiere el cumplimiento riguroso de buenas prácticas de pruebas, documentación exhaustiva y un sistema de monitoreo continuo, debido a la mayor complejidad operativa inherente (gestión de interdependencias, pruebas de integración, monitorización de múltiples servicios, versionado de contratos API, etc.), resulta imprescindible contar con procesos de automatización, herramientas de observabilidad, y capacitación constante de los equipos técnicos.

En suma, una arquitectura de microservicios bien implementada —junto a un gobierno de servicios eficiente, prácticas de CI/CD, y monitoreo efectivo— asegura una plataforma preparada para el crecimiento, la mejora continua y la resiliencia frente a cambios del negocio o del entorno tecnológico

Esta base modular contribuye no solo a la robustez técnica, sino también a la agilidad estratégica y a la sostenibilidad a largo plazo de la solución de comercio electrónico.

8. EJECUCIÓN Y DESARROLLO

8.1Codificación y estructura del código

Lenguajes de programación utilizados.

Para el desarrollo del sistema de marketplace se utilizaron los siguientes lenguajes:

- Node js
- Python
- Flask
- Express +

El proyecto esta en git en el siguiente link:
https://github.com/rfmoreno/arquitectura_microservicios

Arquitectura:

- Microservicios independientes en contenedores Docker.
- Comunicación vía HTTP/REST (entre servicios) y RabbitMQ (para mensajería).
- escalamiento horizontal para manejar alta carga de notificaciones.
- un entorno de desarrollo con recarga automática (hot-reload).

la aplicación de marketplace está corriendo en los siguientes contenedores Docker, cada uno con un propósito específico:

Servicio de Autenticación (Login)

- **Contenedor:** flaskapi-usuarios.1
  Función:
  - Gestiona autenticación de usuarios (login/registro).

- Genera tokens JWT para autorización.
- Almacena datos de usuarios en PostgreSQL.

Servicio de Productos

- **Contenedor:** flaskapi-productos-1
  Función:
  - Maneja el catálogo de productos (CRUD).
  - Calcula precios y descuentos.

Servicio de Pedidos

- **Contenedor:** flaskapi-pedidos-1
  Función:
  - Procesa la creación de pedidos.
  - Valida disponibilidad con el servicio de productos.
  - Almacena pedidos en PostgreSQL.

Servicio de Mensajería

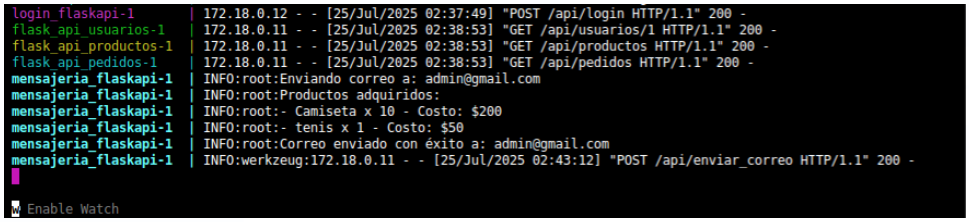
- **Contenedores:** flaskapi-mensajeria-1, mensajeria-flaskapi-1
  Función:
  - Gestiona notificaciones asíncronas (emails/SMS).
  - Usa RabbitMQ para colas de mensajes.
  - Envía confirmaciones de pedidos/pagos.

Servicio de Pagos

Función:

- Procesa transacciones con pasarelas de pago.
- Actualiza el estado de pedidos tras pago exitoso.

Contenedores Docker



Estructura del proyecto

```
micros/
├── docker-compose.yml
├── login
│   ├── flaskapi
│   │   ├── app.py
│   │   ├── Dockerfile
│   │   └── requirements.txt
│   └── nodeview
│       ├── Dockerfile
│       ├── index.js
│       ├── package.json
│       └── views
│           ├── login.pug
│           └── login_success.pug
├── mensajeria
│   └── flaskapi
│       ├── app.py
│       ├── Dockerfile
│       └── requirements.txt
├── mysql
│   └── init
│       └── docker-entrypoint-initdb.d
├── pedidos
│   ├── flaskapi
│   │   ├── app.py
│   │   ├── Dockerfile
│   │   ├── model_py
│   │   │   └── model.py
│   │   └── requirements.txt
│   └── nodeview
│       ├── Dockerfile
│       ├── index.js
│       ├── package.json
│       └── views
│           ├── pago_fake.pug
│           └── pedidos.pug
├── productos
│   ├── flaskapi
│   │   ├── app.py
│   │   ├── Dockerfile
│   │   ├── model_py
│   │   │   ├── _init_.py
│   │   │   └── model.py
│   │   └── requirements.txt
│   ├── model_py
│   │   ├── Dockerfile
│   │   ├── _init_.py
│   │   └── model.py
│   └── nodeview
│       ├── Dockerfile
│       ├── index.js
│       ├── package.json
│       ├── package-lock.json
│       └── views
│           └── productos.pug
├── README.md
└── usuarios
    ├── flaskapi
    │   ├── app.py
    │   ├── Dockerfile
    │   ├── model_py
    │   │   ├── _init_.py
    │   │   └── model.py
    │   └── requirements.txt
    └── nodeview
        ├── Dockerfile
        ├── index.js
        ├── package.json
        └── views
            ├── usuario_edit.pug
            └── usuarios.pug
```

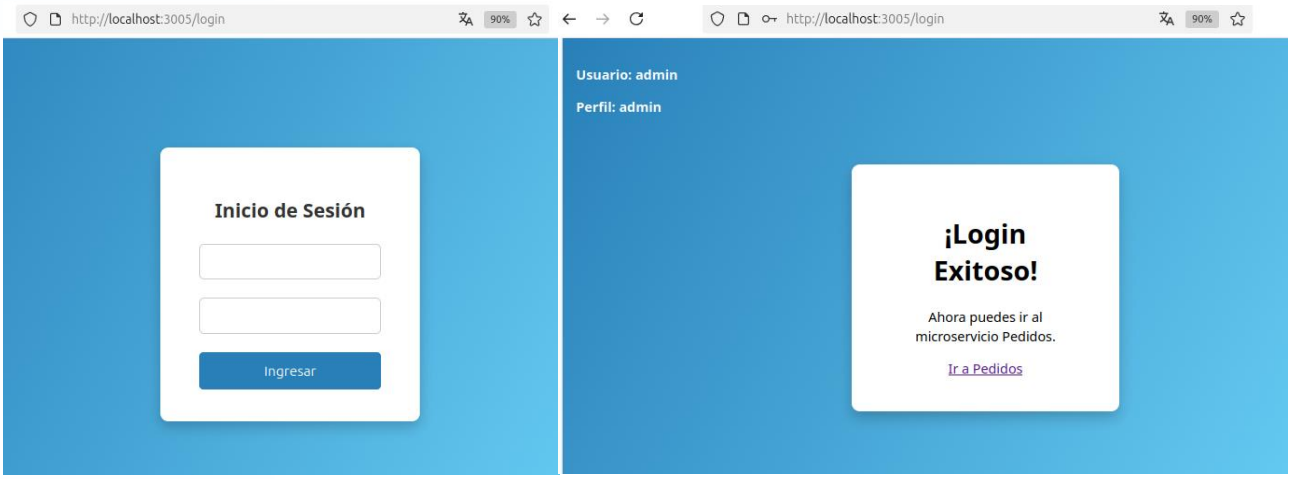
Base de datos

```
mysql> show databases
    -> ;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| mysql              |
| pedidos_db         |
| performance_schema |
| productos_db       |
| sys                |
| usuarios_db        |
+--------------------+
7 rows in set (0.01 sec)
```

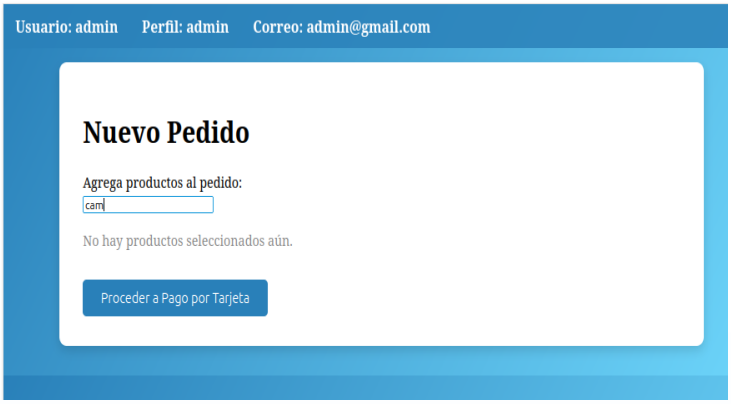
Pantalla de Login :

Modal centrado con formulario de autenticación

Campos: Email y password



Pedidos



Pasarela de pago

Consumo de servicios via api:

```
vlruz@ss:~/Documents/micros$ #Listar todos los usuarios
vlruz@ss:~/Documents/micros$ curl http://localhost:5002/api/usuarios
[{"email":"user@gmail.com","id":2,"id_perfil":1,"login":"user","nombre_completo":"user","perfil":"user"},{"email":"client@gmail.com","id":3,"id_perfil":3,"lo
gin":"client","nombre_completo":"client","perfil":"client"},{"email":"juanperez@email.com","id":4,"id_perfil":1,"login":"jperez","nombre_completo":"Juan Pere
z","perfil":"user"}]
vlruz@ss:~/Documents/micros$ #Consultar usuario por ID
vlruz@ss:~/Documents/micros$ curl http://localhost:5002/api/usuarios/1
{"error":"Usuario no encontrado"}
vlruz@ss:~/Documents/micros$ curl http://localhost:5002/api/usuarios/2
{"email":"user@gmail.com","hash_password":"4297f44b13955235245b2497399d7a93","id":2,"id_perfil":1,"login":"user","nombre_completo":"user","perfil":"user"}
vlruz@ss:~/Documents/micros$ #Crear un usuario
vlruz@ss:~/Documents/micros$ curl -X POST http://localhost:5002/api/usuarios \
  -H "Content-Type: application/json" \
  -d '{
        "nombre_completo": "Juan Perez",
        "login": "jperez",
        "email": "juanperez@email.com",
        "password": "123456",
        "id_perfil": 1
      }'
<!doctype html>
<html lang=en>
<title>500 Internal Server Error</title>
<h1>Internal Server Error</h1>
<p>The server encountered an internal error and was unable to complete your request. Either the server is overloaded or there is an error in the application.
</p>
vlruz@ss:~/Documents/micros$ #Modificar un usuario
vlruz@ss:~/Documents/micros$ curl -X PUT http://localhost:5002/api/usuarios/1 \
  -H "Content-Type: application/json" \
  -d '{
        "nombre_completo": "Juan Pérez Camacho",
        "login": "jperez",
        "email": "juanperez@email.com",
        "password": "nuevaClave456",
        "id_perfil": 2
      }'
{"status":"updated"}
vlruz@ss:~/Documents/micros$ #Eliminar un usuario
vlruz@ss:~/Documents/micros$ curl -X DELETE http://localhost:5002/api/usuarios/2
{"status":"deleted"}
vlruz@ss:~/Documents/micros$ #Listar perfiles
vlruz@ss:~/Documents/micros$ curl http://localhost:5002/api/perfiles
[{"id_perfil":1,"perfil":"user"},{"id_perfil":2,"perfil":"admin"},{"id_perfil":3,"perfil":"client"}]
vlruz@ss:~/Documents/micros$ #Listar todos los productos
vlruz@ss:~/Documents/micros$ curl http://localhost:5001/api/productos
[{"id":2,"nombre":"tenis","precio":"50.00","stock":10},{"id":3,"nombre":"Laptop XYZ","precio":"750.00","stock":30}]
```

```
vlruz@ss:~/Documents/micros$ #Crear un producto
vlruz@ss:~/Documents/micros$ curl -X POST http://localhost:5001/api/productos \
  -H "Content-Type: application/json" \
  -d '{
        "nombre": "Laptop XYZ",
        "precio": 750.0,
        "stock": 30
      }'
{"status":"ok"}
vlruz@ss:~/Documents/micros$ #Simular el envío de correo
vlruz@ss:~/Documents/micros$ curl -X POST http://localhost:5006/api/enviar_correo \
  -H "Content-Type: application/json" \
  -d '{
        "correo": "juanperez@email.com",
        "productos": [
          {"nombre": "Laptop Pro 16\"", "cantidad": 1, "costo_total": 950 },
          {"nombre": "Mouse USB", "cantidad": 2, "costo_total": 40 }
        ]
      }'
{"status":"correo enviado (simulado)"}
vlruz@ss:~/Documents/micros$
```

9. Bibliografía

- *Bass, L., Clements, P., & Kazman, R. (2013). Software Architecture in Practice (3rd ed.). Addison-Wesley.*
- *Cervantes, H., Velasco-Elizondo, P., & Castro Careaga, L. (2016). Arquitectura de software. Conceptos y ciclo de desarrollo. Cengage Learning.*
- Kubernetes Authors. (2024). Production best practices. Kubernetes Documentation. https://kubernetes.io/docs/setup/best-practices/
- Microservices. (2024, 20 de julio). En *Wikipedia, la enciclopedia libre*. https://en.wikipedia.org/wiki/Microservices