



UNIVERSIDAD CENTRAL DEL ECUADOR
FACULTAD DE INGENIERIAS Y CIENCIAS APLICADAS



INGENIERIA EN SISTEMAS DE INFORMACIÓN

Arquitectura de Software

Tema: Manual de Implementación

Integrantes

Emily Guerrón

Paralelo: SI7-001

Fecha: 18-02-2026

QUITO – ECUADOR

Índice

1. Introducción	3
2. Objetivos.....	4
2.1 Objetivo General.....	4
2.2 Objetivos Específicos	4
3. Alcance del Sistema	4
4. Arquitectura del Sistema	5
5. Flujo de Funcionamiento	6
5.1 Flujo de Autenticación.....	6
5.2 Flujo de Creación de Pedido	7
5.3 Flujo de Consulta con Caché (Redis)	8
5.4 Flujo Polling + WebHook.....	9
6. Base de Datos	10
7. Pruebas de Carga (k6).....	11
8. Problemas Encontrados y Soluciones	13
9. Conclusiones.....	13
10. Recomendaciones.....	14

1. Introducción

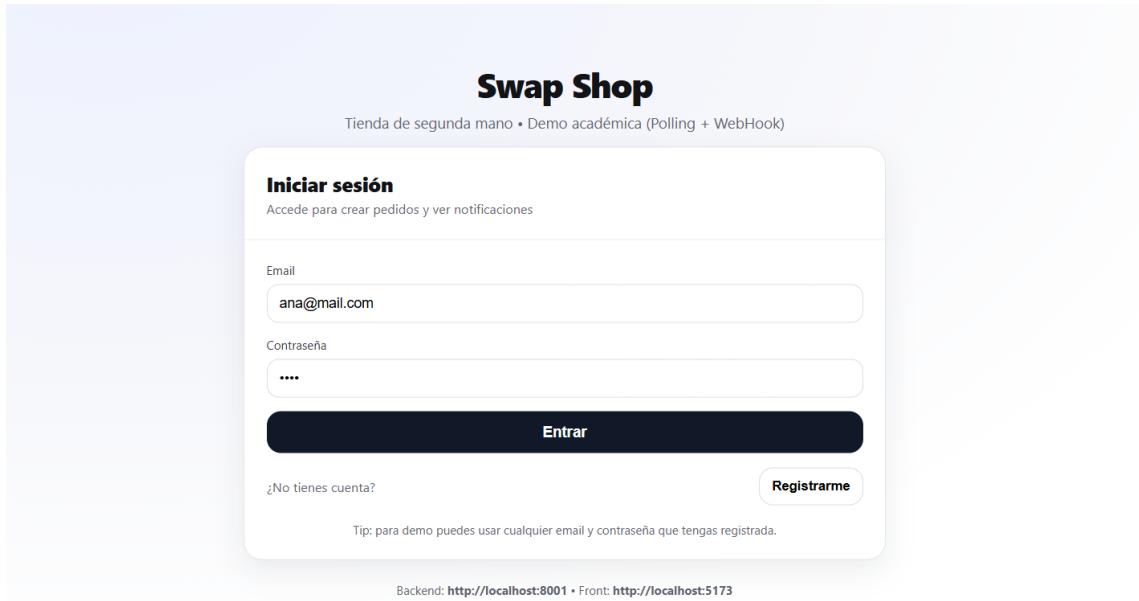
El sistema Swap Shop es una aplicación web desarrollada bajo una arquitectura de microservicios, diseñada para simular la gestión de pedidos en una tienda de segunda mano. La solución integra tecnologías modernas como FastAPI, MySQL, Redis, Docker y React, permitiendo demostrar conceptos de persistencia, caché, comunicación entre servicios y pruebas de carga.

El sistema implementa un flujo completo de negocio que incluye autenticación, creación de pedidos, almacenamiento en base de datos, procesamiento automático mediante polling y envío de notificaciones a través de WebHooks.

Se priorizó una arquitectura desacoplada y escalable, donde cada servicio cumple una función específica, favoreciendo mantenibilidad y separación de responsabilidades.

Además, se incorporó Redis como mecanismo de optimización para reducir la carga sobre la base de datos en consultas repetitivas.

Finalmente, se realizaron pruebas de carga con k6 para validar el comportamiento del sistema bajo múltiples usuarios concurrentes.



2. Objetivos

2.1 Objetivo General

Diseñar e implementar un sistema distribuido basado en microservicios que gestione pedidos de una tienda virtual, integrando persistencia relacional, caché en memoria y comunicación interservicios mediante polling y WebHooks.

2.2 Objetivos Específicos

1. Implementar una API REST en FastAPI para manejar autenticación y pedidos.
2. Diseñar una base de datos relacional en MySQL para almacenar información estructurada.
3. Integrar Redis como sistema de caché para optimizar consultas frecuentes.
4. Implementar un servicio de polling que procese pedidos pendientes automáticamente.
5. Desarrollar un frontend interactivo en React para interacción del usuario.

Cada objetivo responde a una capa específica de la arquitectura, garantizando modularidad y claridad estructural.

3. Alcance del Sistema

El sistema permite a los usuarios registrarse e iniciar sesión mediante un mecanismo de autenticación basado en tokens. Una vez autenticado, el usuario puede crear pedidos compuestos por múltiples items, los cuales son almacenados en la base de datos.

El sistema permite consultar pedidos pendientes, visualizar detalles específicos por ID y observar notificaciones generadas tras el procesamiento automático de pedidos.

No se incluyen funcionalidades avanzadas como pasarela de pagos real, sistema de inventario ni panel administrativo completo.

El alcance está orientado principalmente a demostrar integración tecnológica y comunicación entre microservicios.

El proyecto tiene fines académicos y de demostración técnica

4. Arquitectura del Sistema

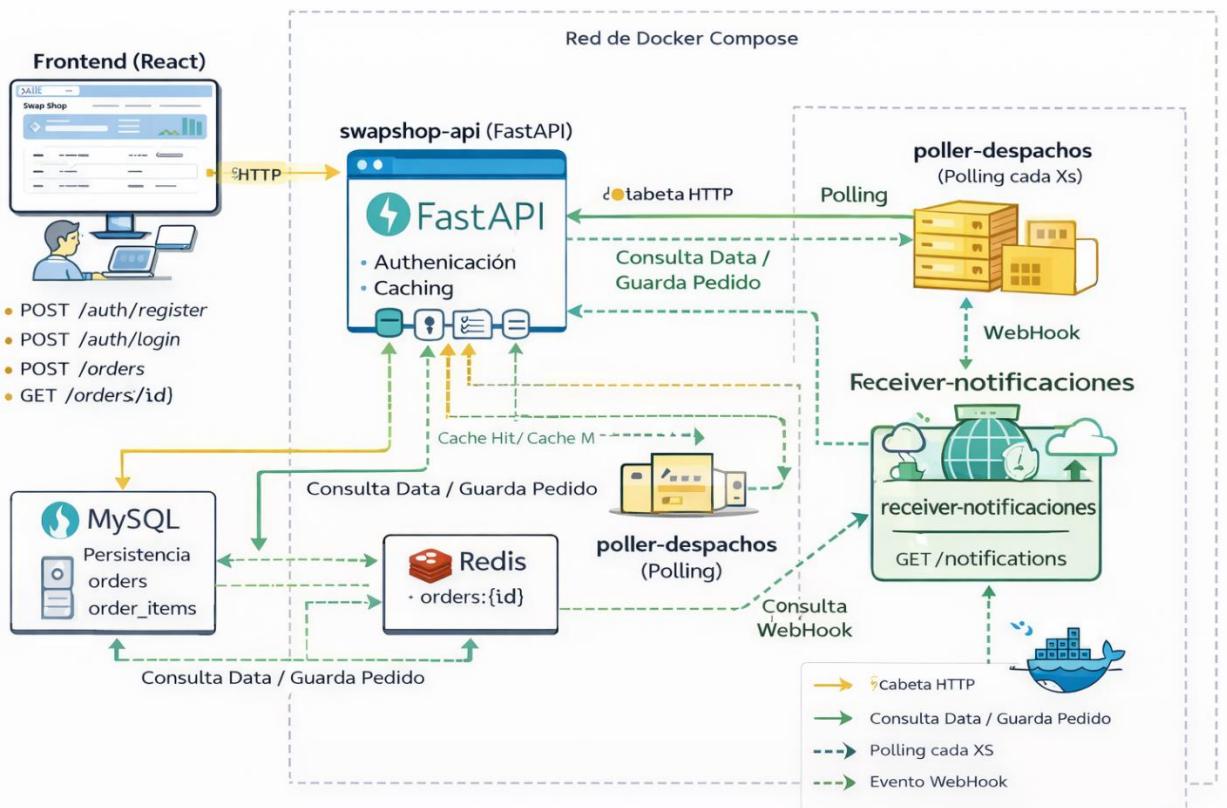
El sistema está compuesto por múltiples servicios independientes que se comunican entre sí dentro de una red Docker.

La API principal (swapshop-api) gestiona autenticación y lógica de negocio. MySQL se encarga del almacenamiento persistente, mientras que Redis funciona como capa de caché para acelerar consultas específicas.

El servicio poller-despachos consulta periódicamente pedidos pendientes y ejecuta procesamiento automático. El receiver-notificaciones recibe eventos mediante WebHook.

El frontend desarrollado en React actúa como cliente, permitiendo interacción del usuario con los servicios backend.

Esta arquitectura favorece escalabilidad horizontal y desacoplamiento entre componentes.



5. Flujo de Funcionamiento

5.1 Flujo de Autenticación

El usuario accede al sistema mediante un formulario de registro o inicio de sesión. Las credenciales son enviadas a la API, la cual valida los datos y genera un token de autenticación.

Este token es almacenado en el navegador y se utiliza en cada solicitud protegida mediante el header Authorization.

La autenticación permite restringir el acceso a endpoints sensibles como la creación de pedidos.

Este mecanismo garantiza seguridad básica y separación entre usuarios autenticados y no autenticados.

El uso de tokens permite una arquitectura stateless en el backend.

The screenshot shows a login interface with the following elements:

- Title:** Iniciar sesión
- Text:** Accede para crear pedidos y ver notificaciones
- Email Input:** emi@gmail.com
- Password Input:** (Redacted)
- Enter Button:** Entrar
- Links:** ¿No tienes cuenta? and Registrarme

5.2 Flujo de Creación de Pedido

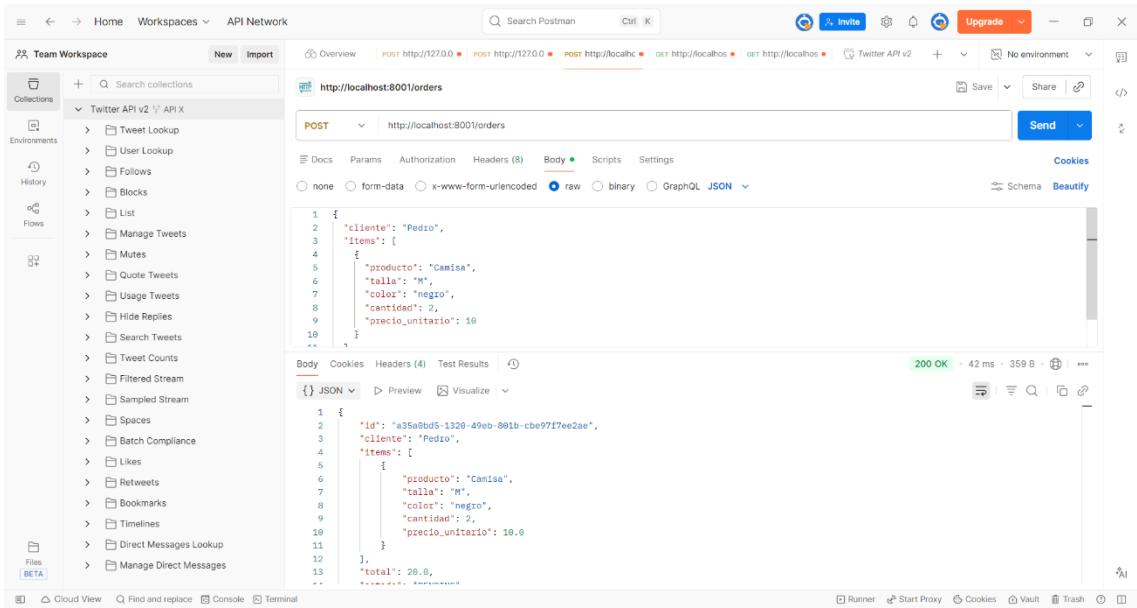
Cuando el usuario crea un pedido, el frontend envía la información estructurada hacia la API.

La API calcula el total automáticamente, genera un identificador único (UUID) y almacena los datos en MySQL.

Posteriormente, guarda una copia del pedido en Redis para optimizar futuras consultas.

El sistema devuelve el pedido creado al frontend, confirmando la operación.

Este flujo demuestra la interacción coordinada entre persistencia relacional y caché en memoria.



5.3 Flujo de Consulta con Caché (Redis)

Cuando se consulta un pedido por ID, el sistema primero verifica si existe en Redis.

Si el dato está en caché (CACHE HIT), la respuesta es inmediata y no se consulta MySQL.

Si no existe (CACHE MISS), se consulta la base de datos y luego se almacena el resultado en Redis.

Este mecanismo reduce carga en la base de datos y mejora rendimiento bajo alta concurrencia.

Redis funciona como acelerador de lectura para operaciones frecuentes.

```

swap-shop-redis-1
redis/7
0b90b3b65b5f
6379:6379

Logs Inspect Bind mounts Exec Files Stats

is-server /path/to/redis.conf
2026-02-18 19:32:01 1:M 19 Feb 2026 00:32:01.273 * monotonic clock: POSIX clock_gettime
2026-02-18 19:32:01 1:M 19 Feb 2026 00:32:01.278 * Running mode=standalone, port=6379.
2026-02-18 19:32:01 1:M 19 Feb 2026 00:32:01.280 * Server initialized
2026-02-18 19:32:01 1:M 19 Feb 2026 00:32:01.281 * Ready to accept connections tcp
2026-02-18 20:14:14 1:signal-handler (1771463654) Received SIGTERM scheduling shutdown...
2026-02-18 20:14:14 1:M 19 Feb 2026 01:14:14.871 * User requested shutdown...
2026-02-18 20:14:14 1:M 19 Feb 2026 01:14:14.872 * Saving the final RDB snapshot before exiting.
2026-02-18 20:14:14 1:M 19 Feb 2026 01:14:14.896 * DB saved on disk
2026-02-18 20:14:14 1:M 19 Feb 2026 01:14:14.897 # Redis is now ready to exit, bye bye...
2026-02-18 20:18:30 1:C 19 Feb 2026 01:18:30.138 * o0000000000 Redis is starting o0000000000
2026-02-18 20:18:30 1:C 19 Feb 2026 01:18:30.138 * Redis version=7.4.7, bits=64, commit=00000000, modified=0, pid=1, just started
2026-02-18 20:18:30 1:C 19 Feb 2026 01:18:30.138 # Warning: no config file specified, using the default config. In order to specify a config file use red
is-server /path/to/redis.conf
2026-02-18 20:18:30 1:M 19 Feb 2026 01:18:30.140 * monotonic clock: POSIX clock_gettime
2026-02-18 20:18:30 1:M 19 Feb 2026 01:18:30.146 * Running mode=standalone, port=6379.
2026-02-18 20:18:30 1:M 19 Feb 2026 01:18:30.149 * Server initialized
2026-02-18 20:18:30 1:M 19 Feb 2026 01:18:30.151 * Loading RDB produced by version 7.4.7
2026-02-18 20:18:30 1:M 19 Feb 2026 01:18:30.152 * RDB age 256 seconds
2026-02-18 20:18:30 1:M 19 Feb 2026 01:18:30.152 * RDB memory usage when created 0.98 Mb
2026-02-18 20:18:30 1:M 19 Feb 2026 01:18:30.152 * Done loading RDB, keys loaded: 0, keys expired: 0.
2026-02-18 20:18:30 1:M 19 Feb 2026 01:18:30.152 * DB loaded from disk: 0.002 seconds
2026-02-18 20:18:30 1:M 19 Feb 2026 01:18:30.153 * Ready to accept connections tcp
2026-02-18 20:48:00 1:M 19 Feb 2026 01:48:00.178 # Possible SECURITY ATTACK detected. It looks like somebody is sending POST or Host: commands to Redis.
This is likely due to an attacker attempting to use Cross Protocol Scripting to compromise your Redis instance. Connection from 172.18.0.1:44388 aborted.

```

RAM 1.89 GB CPU 0.25%

BETA > Terminal ⓘ New version available ⚡ 9

5.4 Flujo Polling + WebHook

El servicio poller ejecuta consultas periódicas a /orders/pending.

Cuando detecta pedidos pendientes, puede marcarlos como procesados.

Luego envía un evento mediante WebHook al servicio receiver-notificaciones.

El receiver almacena la notificación y la expone al frontend.

Este mecanismo simula comunicación asíncrona entre microservicios.

Panel de notificaciones

Aquí llegan los eventos que el poller envía al receiver-notificaciones.

● Receiver: :8002

ORDER_READY

Cliente: **ana**

Total: **\$20,00**

Color: —

► Ver payload

● WebHook recibido

ORDER_READY

Cliente: **k6-user**

Total: **\$30,00**

Color: —

► Ver payload

● WebHook recibido

6. Base de Datos

La base de datos MySQL almacena información estructurada en dos tablas principales: orders y order_items.

La tabla orders contiene la cabecera del pedido, incluyendo estado y fecha de creación.

La tabla order_items almacena el detalle de productos asociados a cada pedido mediante una clave foránea.

Este diseño relacional garantiza integridad referencial y normalización básica.

MySQL proporciona persistencia confiable y transacciones seguras.

```

20     FOREIGN KEY (order_id) REFERENCES orders(id)
21     ON DELETE CASCADE
22   );
23 •   SHOW TABLES;
24 •   USE swapshop;
25 •   SELECT * FROM order_items;
26 •   SELECT * FROM orders;

```

id	cliente	total	estado	created_at
00200457-9615-40db-9cdd-bec9ae1a8c00	k6-user	30.00	PROCESSED	2026-02-19 01:05:58.004000
0078a56a-848e-4e57-b851-ce99e716aa59	k6-user	30.00	PROCESSED	2026-02-19 01:05:56.666238
008c3e9e-9cb4-4b6b-b649-928f9ad71125	k6-user	30.00	PROCESSED	2026-02-19 01:06:10.849763
008f165f-72d2-4d8b-9b61-e7fb8c2137e9	k6-user	30.00	PROCESSED	2026-02-19 01:06:06.500284
00e0cd0b-782d-4a78-b35b-cfea5535ab0	k6-user	30.00	PROCESSED	2026-02-19 01:06:32.091445
015ad65b-d5e3-4efb-a44c-da417fa050c6	k6-user	30.00	PROCESSED	2026-02-19 01:06:09.610829
01dad39f-455d-4729-835c-9ef875f6b50	k6-user	30.00	PROCESSED	2026-02-19 01:06:17.278906
021a6dba-6d18-433b-ac2e-9e4586f9bd3	k6-user	30.00	PROCESSED	2026-02-19 01:06:38.882686
022a4b7e-a481-4ffb-91ab-3fe65e1aaeac	k6-user	30.00	PROCESSED	2026-02-19 01:06:00.453450
029f95e58-9841-4d3d-b007-76a49fa7b5b1	k6-user	30.00	PROCESSED	2026-02-19 01:06:09.224012
0305a461-a76e-4cee-a206-6c3399340851	k6-user	30.00	PROCESSED	2026-02-19 01:06:04.596551
0328e349-4d5d-41ec-8f4d-7dd389d0873b	k6-user	30.00	PROCESSED	2026-02-19 01:06:04.896496

7. Pruebas de Carga (k6)

Se utilizó la herramienta k6 para simular múltiples usuarios concurrentes realizando peticiones al sistema.

La prueba permitió evaluar tiempos de respuesta, estabilidad de la API y comportamiento bajo estrés.

Se verificó que Redis mejora significativamente el rendimiento en consultas repetidas.

Las pruebas demostraron que la arquitectura soporta múltiples conexiones simultáneas sin colapsar.

El uso de pruebas de carga es fundamental para validar escalabilidad.

```
PS C:\Users\USER\Desktop\patrones\swap-shop> docker run --rm -v $PWD:/scripts -w /scripts grafana/k6 run k6
js -e BASE_URL=http://host.docker.internal:8001
latest: Pulling from grafana/k6
486dd99fd2f4: Pull complete
127d8294fa37: Pull complete
4f4fb700ef54: Pull complete
4b5b6854a7b5: Download complete
Digest: sha256:5e937f439684142ba7803722b42e3c9ac9233cfa01d561de0596c1c2794fd680
Status: Downloaded newer image for grafana/k6:latest

      /\      Grafana      /_/
     / \      \  _ / / \
    /   \      | | ( ) |
   /     \      | | \_ \_|
  /       \      | | \_ \_|
 /         \      | | \_ \_|

```

execution: local
script: k6.js

ORDER_READY

● WebHook recibido

ORDER_READY

● WebHook recibido

ORDER_READY

● WebHook recibido

8. Problemas Encontrados y Soluciones

- Durante el desarrollo surgieron errores relacionados con resolución de servicios en Docker, autenticación MySQL y validación de respuestas.
- Se solucionaron ajustando redes Docker, agregando dependencias necesarias y corrigiendo modelos de respuesta en FastAPI.
- También se presentó un problema de pantalla en blanco en el frontend, resuelto mediante control de errores y validación de sesión.
- Estos incidentes permitieron fortalecer la estabilidad general del sistema.
- Cada error representó una oportunidad de aprendizaje técnico.

9. Repositorio Git

https://github.com/Emilianah/PROYECTO_FINAL.git

10. Conclusiones

- El sistema cumple con los objetivos planteados, integrando múltiples tecnologías modernas en una arquitectura funcional.
- La combinación de MySQL y Redis demuestra cómo se puede equilibrar persistencia y rendimiento.
- El uso de microservicios permite modularidad y fácil mantenimiento.
- Las pruebas de carga validaron estabilidad bajo condiciones exigentes.
- El proyecto evidencia dominio de backend, frontend y despliegue con contenedores.

11. Recomendaciones

- Se recomienda implementar persistencia de notificaciones en base de datos para evitar pérdida de información al reiniciar servicios.
- Agregar monitoreo con herramientas como Prometheus o Grafana mejoraría la observabilidad.
- Implementar expiración de tokens y refresh tokens reforzaría la seguridad.
- Optimizar manejo de errores en frontend evitaría bloqueos de interfaz.
- Escalar horizontalmente servicios críticos permitiría mayor rendimiento en producción.