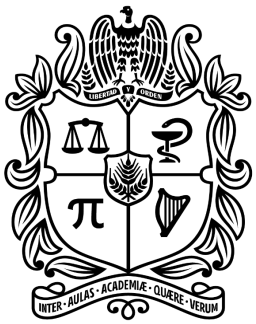
Informe de Implementación y Resultados del Proyecto DevOps



UNIVERSIDAD
NACIONAL
DE COLOMBIA

Repositorio: https://github.com/Luchito0703/proyectoDevOps

Luis Miguel López Uribe
lulopezu@unal.edu.co
David Moreno Gaviria
davmorenoga@unal.edu.co
Santiago Ortiz Cardona
sanortizca@unal.edu.co

Estudiantes de Administración de Sistemas Informáticos
Universidad Nacional de Colombia
Sede Manizales

Introducción a DevOps
Juan Camilo Escobar Naranjo
2025

Contenido

1. Introducción

El presente informe detalla el proceso de implementación de un ciclo de vida DevOps completo para una aplicación de gestión de ventas desarrollada con el framework FastAPI. El objetivo principal del proyecto fue aplicar las metodologías y herramientas DevOps para automatizar la integración, el despliegue y la operación de la aplicación, garantizando la eficiencia, la fiabilidad y la escalabilidad del sistema.

Partiendo de una aplicación base, se implementó un flujo de trabajo que abarca desde el control de versiones con Git hasta el monitoreo en tiempo real con Prometheus y Grafana. El proyecto se estructuró para cumplir con los siguientes entregables clave: un repositorio versionado bajo la estrategia Git Flow, la contenerización de la aplicación y sus servicios dependientes, la configuración de un pipeline de Integración Continua y Despliegue Continuo (CI/CD) con GitHub Actions, la gestión de artefactos con un repositorio Nexus autoalojado y el despliegue final en un entorno basado en Docker Compose.

2. Arquitectura y Tecnologías

2.1. Arquitectura de la Solución

La arquitectura se centra en un modelo de servicios contenerizados, orquestados por Docker Compose. Los componentes principales son:

- **Servicio de Aplicación (fastapi-app):** El núcleo de la solución. Es una API RESTful construida en FastAPI que expone endpoints para gestionar clientes, productos y ventas. Además, ofrece un endpoint /metrics para el monitoreo.
- **Servicio de Monitoreo (prometheus):** Se encarga de recolectar periódicamente las métricas expuestas por la aplicación FastAPI a través del endpoint /metrics.
- **Servicio de Visualización (grafana):** Se conecta a Prometheus como fuente de datos para presentar las métricas en dashboards interactivos y visualmente comprensibles.
- **Servicio de Artefactos (nexus):** Funciona como un repositorio privado para almacenar los artefactos generados, en este caso, las imágenes Docker de la aplicación.

Todos los servicios se comunican entre sí a través de una red virtual (app-network) definida en el archivo docker-compose.yml, lo que garantiza un entorno aislado y seguro.

2.2. Tecnologías Utilizadas

- **Python y FastAPI:** Para el desarrollo de una API de alto rendimiento.
- **Git y Git Flow:** Para el control de versiones y la gestión colaborativa del código.
- **Docker y Docker Compose:** Para la contenerización de la aplicación y la orquestación de los servicios.
- **GitHub Actions:** Como plataforma de CI/CD para automatizar las fases de prueba, construcción y despliegue.

- **Nexus Repository Manager:** Para el almacenamiento y versionamiento de las imágenes Docker.
- **Prometheus y Grafana:** Para la recolección y visualización de métricas de rendimiento y estado de la aplicación.
- **Pytest:** Para la implementación y ejecución de pruebas unitarias automatizadas.

3. Control de Versiones con Git y Git Flow

Se adoptó la estrategia de ramificación **Git Flow** para mantener un control de versiones organizado y predecible. Este modelo nos permitió separar el trabajo en desarrollo (develop) del código en producción (master), además de gestionar nuevas funcionalidades (feature), lanzamientos (release) y correcciones urgentes (hotfix) de manera aislada y controlada.

Como se puede observar en la siguiente figura, el repositorio refleja fielmente esta estructura, con ramas específicas para cada propósito, lo que demuestra la aplicación práctica de la metodología en nuestro proyecto.

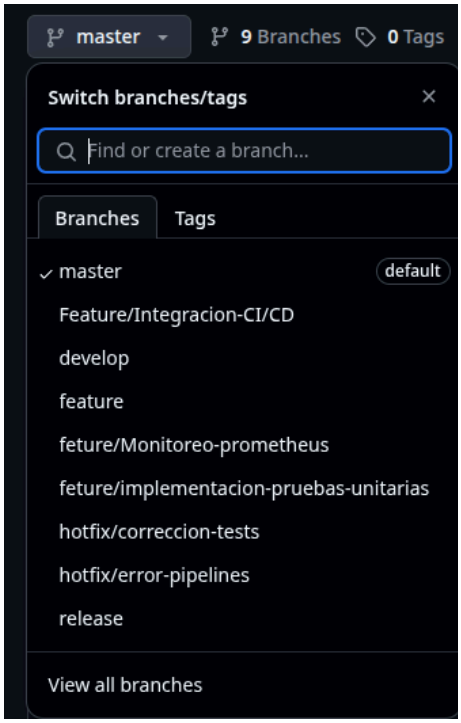


Figura 1: Estructura de ramas del proyecto implementando la estrategia Git Flow.

Para el desarrollo de nuevas funcionalidades, se siguió el flujo estándar:

1. Se inicializa una nueva rama de característica a partir de develop con el comando git flow feature start <nombre-feature>.
2. Una vez completado el desarrollo en la rama de característica, esta se fusiona de nuevo en develop mediante git flow feature finish <nombre-feature>.

Es importante destacar que, por diseño de Git Flow, las ramas feature son eliminadas después de ser fusionadas en develop. Por esta razón, en el historial final del repositorio no se observan las ramas de características que fueron fusionadas, ya que su propósito es temporal y su ciclo de vida termina al integrarse en la rama principal de desarrollo.

4. Contenerización con Docker

4.1. Dockerfile: Se creó un Dockerfile para empaquetar la aplicación FastAPI con todas sus dependencias. El proceso de construcción de la imagen sigue los siguientes pasos:

1. Utiliza una imagen base de Python.
2. Establece el directorio de trabajo dentro del contenedor.
3. Copia el archivo requirements.txt e instala las dependencias para aprovechar el cache de capas de Docker.
4. Copia el resto del código fuente de la aplicación.
5. Expone el puerto **5000**, que es el puerto en el que la aplicación escucha las peticiones.
6. Define el comando de inicio (CMD) para ejecutar la aplicación con **uvicorn**.

4.2. Docker Compos: Para orquestar el despliegue de la aplicación junto con sus servicios auxiliares, se utilizó el archivo *docker-compose.yml*. Este archivo define cuatro servicios principales: app, prometheus, grafana y nexus.

- El servicio **app** se construye a partir del Dockerfile local.
- Los servicios **prometheus**, **grafana** y **nexus** utilizan imágenes oficiales de Docker Hub.
- Se definieron volúmenes persistentes para grafana-storage y nexus-data con el fin de que los datos de configuración y los artefactos no se pierdan al reiniciar los contenedores.
- Todos los servicios están conectados a la red app-network, permitiendo la comunicación entre ellos mediante los nombres de servicio. Por ejemplo, Prometheus puede encontrar la aplicación en app:5000.

5. Integración y Despliegue Continuo (CI/CD)

Se diseñó y configuró un pipeline de CI/CD utilizando **GitHub Actions** para automatizar el ciclo de vida de la aplicación. El flujo de trabajo se configuró para activarse en cada push a las ramas develop y master, garantizando que cada cambio sea validado y desplegado de forma automática.

La siguiente imagen muestra el historial de ejecuciones de nuestros workflows en GitHub Actions, donde se aprecian los pipelines para pruebas, publicación de artefactos y despliegues automáticos.

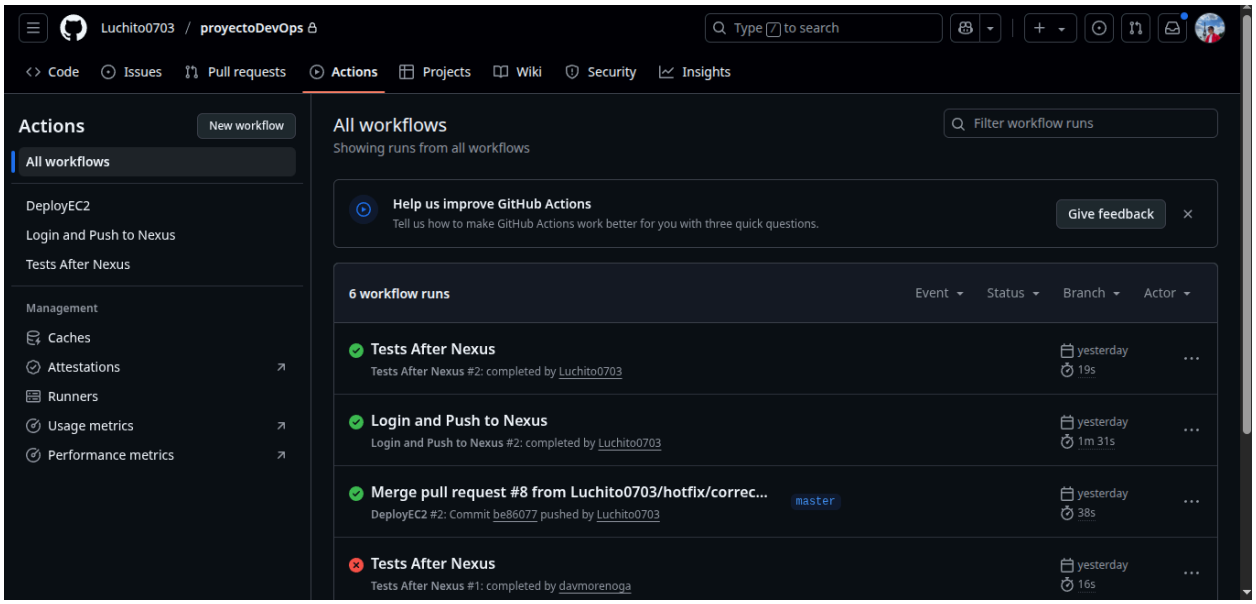


Figura 2: Historial de ejecución de los workflows de CI/CD en GitHub Actions.

El pipeline se estructuró en las siguientes fases:

Fase de CI (Integración Continua):

1. **Checkout:** El pipeline clona el repositorio.
2. **Configuración del Entorno:** Se configura la versión de Python requerida.
3. **Instalación de Dependencias:** Se instalan las librerías listadas en requirements.txt.
4. **Pruebas Automatizadas:** Se ejecutan las pruebas unitarias con pytest (workflow "Tests After Nexus"). Si alguna prueba falla, el pipeline se detiene.

Fase de CD (Despliegue Continuo):

1. **Construcción y Publicación de Artefacto:** Si las pruebas son exitosas, el pipeline construye la imagen Docker, se autentica en Nexus y publica la nueva versión (workflow "Login and Push to Nexus").
2. **Despliegue:** Finalmente, el pipeline de despliegue (workflow "DeployEC2") se conecta al servidor de producción, descarga la imagen actualizada desde Nexus y reinicia los servicios con docker-compose.

6. Gestión de Artefactos con Nexus

Se desplegó una instancia de Nexus Repository Manager como parte de nuestra pila de servicios en Docker Compose. El propósito de Nexus en este proyecto fue centralizar la gestión de artefactos de software, específicamente las imágenes Docker.

El pipeline de CD fue configurado para autenticarse y enviar (push) las imágenes Docker construidas y validadas a un repositorio docker (hosted) dentro de Nexus. Esto nos permite

tener un registro versionado y seguro de cada compilación estable de la aplicación, desacoplando el proceso de construcción del de despliegue y facilitando los rollbacks en caso de ser necesario.

7. Monitoreo con Prometheus y Grafana

Para garantizar la observabilidad de la aplicación, se implementó una solución de monitoreo basada en Prometheus y Grafana.

7.1. Exposición de Métricas La aplicación FastAPI fue instrumentada utilizando la librería prometheus_client. En el archivo app/metrics.py, se configuraron dos métricas principales:

- **http_requests_total (Counter):** Un contador que registra el número total de peticiones HTTP, etiquetado por método y endpoint.
- **http_request_duration_seconds (Histogram):** Un histograma que mide la latencia de las peticiones para cada endpoint.

Un middleware se encarga de interceptar cada petición para actualizar estas métricas, las cuales son expuestas a través de un endpoint /metrics en un formato que Prometheus puede procesar.

7.2. Recolección y Visualización

El archivo de configuración prometheus.yml define un job de recolección (scrape) que apunta al servicio de la aplicación (app:5000), permitiendo a Prometheus obtener las métricas de forma periódica. Grafana se configuró con Prometheus como fuente de datos, y se crearon paneles para visualizar en tiempo real la tasa de peticiones, la latencia y otros indicadores clave de rendimiento, permitiendo un monitoreo proactivo de la salud del sistema.

8. Pruebas Unitarias

Las pruebas automatizadas son esenciales para asegurar la calidad del código y la fiabilidad de la aplicación. Se utilizaron pytest y TestClient para crear pruebas unitarias para los endpoints de la API.

8.1. Casos de Prueba Implementados En el archivo tests/test_main.py, se desarrollaron las siguientes pruebas:

- **test_root:** Verifica que el endpoint raíz / responde con un código de estado 200.
- **test_metrics:** Confirma que el endpoint /metrics es accesible y devuelve métricas de Prometheus.
- **test_add_sale_success:** Simula un flujo completo creando un cliente y un producto, para luego registrar una venta exitosamente.

8.2. Análisis de Resultados

La ejecución de las pruebas arrojó el siguiente resultado, como se muestra en la figura a continuación.

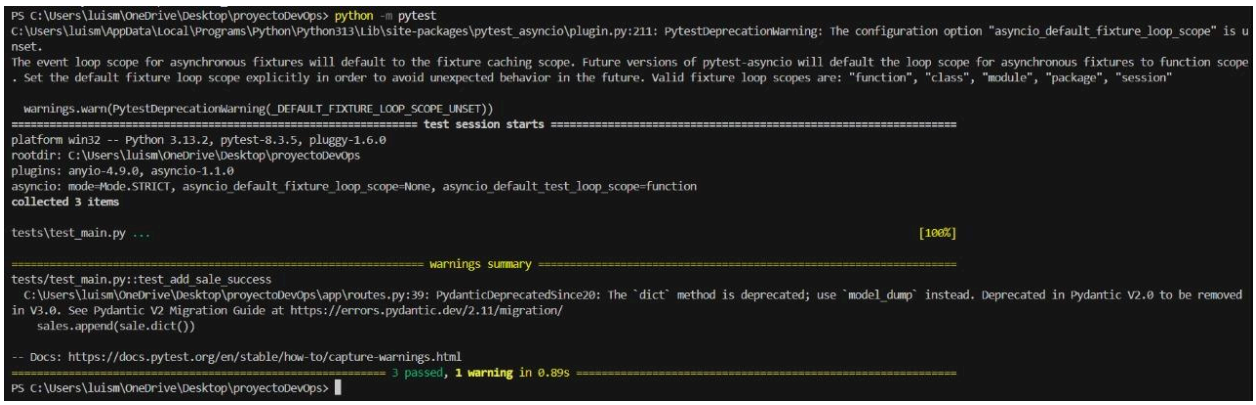


Figura 3: Captura de pantalla de los resultados de las pruebas unitarias.

El resumen **"3 passed, 1 warning in 0.89s"** indica que **todos los casos de prueba se ejecutaron con éxito**, validando la funcionalidad básica de la API. Sin embargo, se identificaron dos advertencias (warnings) que merecen ser analizadas:

1. **PytestDeprecationWarning: ... asyncio_default_fixture_loop_scope**: Esta advertencia es emitida por el plugin pytest-asyncio. Nos informa que el comportamiento predeterminado para el scope del loop de eventos asíncronos cambiará en futuras versiones. Para asegurar la compatibilidad futura y eliminar la advertencia, se recomienda configurar explícitamente el scope en el archivo de configuración de pytest.
2. **PydanticDeprecatedSince20: The 'dict' method is deprecated; use 'model_dump' instead**: Esta advertencia proviene de la librería Pydantic v2. Nos indica que el método .dict() utilizado en nuestro código, específicamente en la línea sales.append(sale.dict()) del archivo app/routes.py, ha quedado obsoleto. La práctica recomendada es reemplazarlo por el método .model_dump(), que ofrece una funcionalidad mejorada. Aunque el código actual funciona, es crucial atender esta advertencia para mantener la compatibilidad con futuras versiones de Pydantic.

9. Conclusiones y Lecciones Aprendidas

La implementación de este proyecto nos permitió consolidar nuestros conocimientos en la cultura y las herramientas DevOps. Logramos construir un pipeline automatizado que integra el control de versiones, las pruebas, la construcción de artefactos, el despliegue y el monitoreo, cumpliendo con todos los objetivos planteados, además nos permitió aprender los siguientes conceptos y valores del ciclo de desarrollo de software:

- **La importancia de la automatización:** La automatización de pruebas y despliegues reduce drásticamente el error humano y acelera el tiempo de entrega de nuevas funcionalidades.

- **El poder de la contenerización:** Docker y Docker Compose demostraron ser herramientas invaluables para crear entornos de desarrollo y producción consistentes, eliminando el problema de "en mi máquina sí funciona".
- **La observabilidad es clave:** Implementar monitoreo desde el inicio no es un lujo, sino una necesidad para entender el comportamiento de la aplicación en producción y reaccionar rápidamente ante cualquier problema.
- **El valor de las buenas prácticas:** Seguir estrategias como Git Flow y atender las advertencias de las librerías son prácticas que, aunque requieren un esfuerzo inicial, mejoran la mantenibilidad y la calidad del proyecto a largo plazo.