

Informe Trabajo práctico Integrador DevOps. **DeliLunch.**



Integrantes: Magalí Sarmiento y Francisco Campora.

Profesor: Lucas Bonanni

Segundo Cuatrimestre 2025 - Universidad de Palermo.

Índice.

Índice.....	1
Objetivos del trabajo.....	2
Entregables.....	2
Fases del trabajo integrador.....	2
1. Desarrollo de la Aplicación.....	3
1.1.Desarrollo de la aplicación.....	3
1.1.1. Descripción de la aplicación.....	3
1.1.2.Funcionalidades requeridas de la aplicación.....	3
CRUD de productos.....	3
Endpoint básico de health check.....	4
Endpoints implementados.....	5
1.1.3. Lenguaje y Frameworks.....	5
1.2.Pruebas Unitarias sobre los EndPoints.....	6
1.3.Contenerización con Docker.....	8
1.3.1.Creación del archivo Dockerfile.....	8
Multi-stage build.....	8
Buenas prácticas aplicadas en el TP.....	8
1.4.Orquestación con Docker Compose.....	11
2. Configuración del Repositorio y CI.....	12
2.1.Gestión del Código Fuente.....	12
2.2.Configuración de Integración Continua (CI).....	13
Archivo de Configuración de CI.....	13
Workflow del backend.....	14
Workflow del frontend.....	15
Mejoras y ajustes en la Integración Continua (CI) propuestas por el profesor.....	16
Protección de la Rama Principal.....	17
2.3. Construcción y Publicación de la Imagen Docker.....	18
3. Despliegue Continuo.....	23
3.1. Hook de Render.....	23
3.2. Implementación del Webhook de Despliegue.....	24
3.3. Modificación del Workflow de GitHub Actions.....	26
3.4. Flujo completo de CI/CD.....	26
3.5. Resultados obtenidos.....	28
4. Monitoreo y Análisis.....	30
Procedimiento para agregar un widget al dashboard de New Relic.....	37
5. Conclusión.....	39

Objetivos del trabajo.

El objetivo del trabajo consiste en aplicar los conceptos y herramientas aprendidos en el curso de DevOps para crear un proceso de integración continua, entrega continua y despliegue automatizado de una aplicación utilizando herramientas y prácticas de DevOps.

Entregables.

- Link al repositorio (público)
- El código en un archivo zip
- Informe del trabajo práctico integrador.
- PPT

Fases del trabajo integrador.

Parte 1: Creación de la Aplicación. Desarrollar una aplicación web básica y preparar su entorno de contenedores.

Parte 2: Configuración del Repositorio y CI. Establecer un sistema de integración continua (CI) para asegurar la calidad y la coherencia del código.

Parte 3: Despliegue Continuo (CD) (Opcional). Configurar un proceso de despliegue automatizado que se active al realizar merges en el repositorio

Parte 4: Monitoreo y Análisis. Implementar herramientas de monitoreo y análisis para supervisar el rendimiento y la salud de la aplicación.

Parte 5: Documentación y Presentación. Documentar todo el proceso y presentar los resultados y aprendizajes

1. Desarrollo de la Aplicación

1.1.Desarrollo de la aplicación.

1.1.1. Descripción de la aplicación.

“Deli Lunch” es una aplicación web pensada para optimizar el manejo de stock en emprendimientos familiares, especialmente en aquellos donde no se cuenta con conocimientos técnicos. La idea surge con el fin de reemplazar el control de inventario manual que genera confusión, pérdida de información y una carga administrativa innecesaria. Deli Lunch es una herramienta digital simple, pero poderosa y amigable, que permite:

- Visualizar rápidamente qué productos están disponibles o agotados.
- Administrar el stock, agregar, modificar o eliminar productos del stock con unos pocos clics.
- Tener un mejor control del negocio y reducir errores humanos.
- Agendar los pedidos del negocio
- Ver un reporte con los ingresos mensuales.

La app transforma una gestión lenta y propensa a errores en una solución moderna, accesible y fácil de usar en el día a día. Ayuda a cualquier emprendedor que quiera organizar mejor su producción sin depender de herramientas complejas y costosas.

1.1.2.Funcionalidades requeridas de la aplicación.

CRUD de productos.

Se desarrolló un conjunto de operaciones CRUD (Create, Read, Update, Delete) aplicadas a la gestión de comidas y pedidos dentro del stock de Deli Lunch.

Este módulo permite **crear** nuevos productos y pedidos, **leer** (consultar) la lista de productos y pedidos registrados desde una tabla, **actualizar** los datos de un producto o pedido existente y **eliminar** aquellos que ya no se requieran.

Estos CRUDs son accesibles desde la pantalla principal Home.js y la pantalla Pedidos.js y permite una gestión sencilla y completa de los productos y pedidos del emprendimiento.

Las operaciones se exponen mediante la API del backend en la ruta /api/comidas y /api/pedidos y se integran con el frontend en React, brindando a la usuaria una interfaz sencilla para administrar el inventario.

Listado de productos disponibles

ID	Nombre	Descripción	Stock	Acciones
1	Empanadas	Empanadas de carne	20	<button>Editar</button> <button>Eliminar</button>
2	Tarta de Verdura	Tarta casera	4	<button>Editar</button> <button>Eliminar</button>
3	Pizza	Mozzarella grande	10	<button>Editar</button> <button>Eliminar</button>
4	Milanesa de ternera	Guarnición a elección (puré / papas)	3	<button>Editar</button> <button>Eliminar</button>

Gestión de Pedidos

Nuevo Pedido Cliente: Fecha de Entrega: Tipo de Envío: Estado: Precio: Guardar Pedido

Lista de Pedidos

Cliente	Fecha de Entrega	Tipo de Envío	Estado	Precio	Acciones
Joaquín	2025-04-23	Retira	Pendiente	\$40000.00	<button>Marcar como Terminado</button> <button>Eliminar</button>
Milagros Longarela	2025-05-23	Entrega	Pendiente	\$60000.00	<button>Marcar como Terminado</button> <button>Eliminar</button>

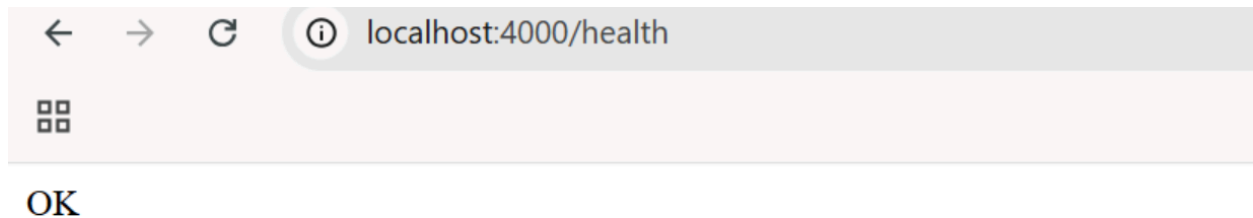
Capturas de pantalla de la aplicación DeliLunch.

Endpoint básico de health check.

Se implementó un endpoint de verificación denominado **/health**, cuya finalidad es confirmar el estado de ejecución del sistema. Este recurso responde a una petición HTTP con un código de estado 200 OK y el mensaje "OK", garantizando que la

aplicación se encuentra disponible y operativa. Su propósito principal es facilitar una comprobación rápida de la salud del servicio, tanto para desarrolladores como para posibles herramientas de monitoreo.

En pocas palabras, el endpoint `/health` es una práctica DevOps común para monitoreo: los sistemas de CI/CD y las herramientas como New Relic (en nuestro caso) lo usan para saber si el servicio está disponible.



Captura de pantalla del health funcionando.

Endpoints implementados

- `/api/comidas`: GET, POST, PUT, DELETE (gestión de stock).
- `/api/pedidos`: GET, POST, PUT, DELETE (gestión de pedidos).
- `/api/reportes/ingresos`: GET (reporte de ingresos mensuales).
- `/api/health`: GET (endpoint básico de health check).

1.1.3. Lenguaje y Frameworks.

- **JavaScript** fue el lenguaje elegido por su portabilidad y facilidad de ejecución en distintos sistemas. Al ser orientado a objetos, facilita la organización y el mantenimiento del código. Además, su flexibilidad y el ecosistema de tecnologías como Node.js, Express y React lo convierten en una opción confiable para el desarrollo de Deli Lunch.
- Para el frontend se escoge **React** que se encarga exclusivamente de la presentación y la interacción con el usuario, consumiendo una API REST desarrollada en el backend.

- Se selecciona **Node.js con Express** para el backend por su compatibilidad con JavaScript del frontend, su bajo nivel de complejidad para proyectos simples y su excelente soporte para construir APIs RESTful de forma rápida y ordenada.
- Se elige **SQLite** como base de datos por su portabilidad y compatibilidad con Node.js. Es perfecta para prácticas académicas donde el foco está en aplicar conceptos de arquitectura web, APIs REST y CRUD. No requiere instalar ni configurar un servidor externo y permite trabajar con persistencia de datos en un único archivo.

1.2.Pruebas Unitarias sobre los EndPoints.

Las pruebas unitarias tienen el objetivo de verificar una funcionalidad básica de la aplicación y asegurar que el código cumpla con los requisitos esperados.

Para garantizar el correcto funcionamiento de la API, se implementan pruebas automatizadas sobre los endpoints utilizando **Jest** como framework de testing junto con **Supertest** para simular peticiones HTTP reales sin tener que levantar el servidor completo. Estas pruebas permiten verificar que cada operación de la API responde correctamente ante diferentes escenarios.

En el backend se desarrollaron dos archivos de pruebas: `comida.test.js` y `pedido.test.js`, ubicados dentro de la carpeta `tests`. En ellos se testean los principales endpoints CRUD de comidas y pedidos (GET, POST, PUT, DELETE), así como también un endpoint especial de reporte para los ingresos mensuales.

Cada test evalúa tanto el código de estado HTTP como el contenido de la respuesta, validando que los datos sean coherentes y que el servidor actúe conforme a lo esperado. Las pruebas se ejecutan con el comando `npx jest`, el cual está configurado en el `package.json` mediante el script "test": "jest". Esta estrategia de testing ayuda a detectar errores temprano y aporta confiabilidad al desarrollo de la API.

```

Found 0 vulnerabilities

PS C:\Users\magui\DeliLunch\backend> npx jest tests/comida.test.js
PASS tests/comida.test.js
  Test de API de Comidas
    ✓ GET /api/comidas debe devolver todas las comidas (27 ms)
    ✓ POST /api/comidas debe crear una nueva comida (37 ms)
    ✓ GET /api/comidas/:id debe devolver una comida específica (6 ms)
    ✓ PUT /api/comidas/:id debe actualizar una comida (17 ms)
    ✓ DELETE /api/comidas/:id debe eliminar una comida (15 ms)

Test Suites: 1 passed, 1 total
Tests: 5 passed, 5 total
Snapshots: 0 total
Time: 1.446 s
Ran all test suites matching /tests\\comida.test.js/i.
PS C:\Users\magui\DeliLunch\backend> |

```

Capturas de pantalla de los tests de DeliLunch corriendo.

```

PS C:\Users\magui\DeliLunch\backend> npx jest tests/pedido.test.js
console.log
  Datos recibidos para crear pedido: {
    cliente: 'Juan Perez',
    fecha_entrega: '2025-05-30',
    tipo_envio: 'Retira',
    estado: 'Pendiente',
    precio: 2500
  }

    at log (controllers/pedidoController.js:12:11)

PASS tests/pedido.test.js
  Test de API de Pedidos
    ✓ POST /api/pedidos debe crear un nuevo pedido (74 ms)
    ✓ GET /api/pedidos debe devolver todos los pedidos (8 ms)
    ✓ GET /api/pedidos/ingresos-mensuales debe devolver reporte de ingresos (5 ms)
    ✓ DELETE /api/pedidos/:id debe eliminar un pedido (14 ms)
    ✓ PUT /api/pedidos/:id debe actualizar un pedido (6 ms)

Test Suites: 1 passed, 1 total
Tests: 5 passed, 5 total
Snapshots: 0 total
Time: 0.974 s, estimated 1 s
Ran all test suites matching /tests\\pedido.test.js/i.
PS C:\Users\magui\DeliLunch\backend> |

```

Cabe destacar que en DevOps, el testing se automatiza dentro del pipeline (CI), lo cual será implementado más adelante en el práctico.

1.3.Contenerización con Docker.

1.3.1.Creación del archivo Dockerfile.

La contenerización con **Docker** permite que la aplicación Deli Lunch se ejecute de la misma forma en cualquier entorno, sin depender de configuraciones locales ni sistemas operativos específicos.

El Dockerfile es el archivo donde se definen las instrucciones para crear la imagen de la aplicación: qué base se utiliza, qué dependencias instalar, qué archivos copiar y cómo iniciar el servicio. Gracias a esto, se obtiene un contenedor único, reproducible, portable y fácil de desplegar.

Multi-stage build

En este trabajo se implementó la técnica de multi-stage build en los Dockerfile del backend y del frontend con el objetivo de optimizar el tamaño de la imagen final.

- Backend: en el primer stage, basado en node:18-alpine, se instalan todas las dependencias utilizando npm install y se copian los archivos para preparar el proyecto. En el segundo stage, también basado en node:18-alpine, se copian únicamente los archivos necesarios desde el stage anterior y se configura el entorno de producción con las variables. Finalmente se ejecuta el comando llamado “node server.js” que va a llevar a la ejecución de la API.
- Frontend: en el primer stage (con node:18-alpine) se instalan dependencias y se genera el build de la aplicación React; en el segundo stage, basado en nginx:alpine, se copian únicamente los archivos estáticos generados, que son servidos por Nginx.

Esta separación clara entre **compilación y ejecución** permite que las imágenes finales no incluyan herramientas ni librerías de desarrollo, lo que resulta en *imágenes más livianas, seguras y rápidas de desplegar*. En simples palabras, primero se arma un entorno para instalar dependencias y preparar el entorno, y después otro más pequeño solo para correrlos.

Buenas prácticas aplicadas en el TP

- **Uso de imágenes base ligeras:** node:18-alpine y nginx:alpine.

- **Minimización de capas:** agrupando instrucciones y evitando dependencias innecesarias.
- **Aprovechamiento de la caché:** copiando primero package*.json en el frontend para que la instalación de dependencias no se repita si el código no cambia.
- Inclusión únicamente de los artefactos finales en la imagen de producción (código backend listo para correr o build estático de React)

En síntesis, el uso de **multi-stage builds** más estas prácticas de Docker permitió obtener imágenes finales más pequeñas, reproducibles y listas para producción, alineadas con los objetivos del trabajo práctico de DevOps.

```
# Stage 1: Construcción
FROM node:20-alpine AS build

WORKDIR /app

# Copiamos todo el proyecto al contexto de construcción del stage 1
COPY . .

# Instalamos las dependencias
RUN npm install

# Stage 2: Producción
FROM node:20-alpine

WORKDIR /app

#Argumentos para newrelic
ARG NEW_RELIC_APP_NAME
ARG NEW_RELIC_LICENSE_KEY
# Copiamos solo los archivos necesarios del stage 1 al stage 2
COPY --from=build /app ./

#Convertimos los argumentos en variables de entorno para que node los lea
ENV NEW_RELIC_APP_NAME=$NEW_RELIC_APP_NAME
    #nombre de la app
ENV NEW_RELIC_LICENSE_KEY=$NEW_RELIC_LICENSE_KEY
    #license key de la app

# Configuración de new relic solo con variables de entorno
ENV NEW_RELIC_NO_CONFIG_FILE=true
    #significa que va a correr el agente sin un file de config - parametro dado por new relic
ENV NEW_RELIC_DISTRIBUTED_TRACING_ENABLED=true
    #tracing activado - parametro dado por new relic
ENV NEW_RELIC_LOG=stdout
    #logs a la consola - parametro dado por new relic

EXPOSE 4000

CMD ["node", "server.js"]
```

*Captura de
pantalla del
Dockerfile
correspondiente al
servicio backend.*

```
# Stage 1: Compilación de la aplicación de React
FROM node:18-alpine AS build

WORKDIR /app

# Copiamos package.json y package-lock.json para cachear la instalación de dependencias
COPY package*.json ./

# Instalamos las dependencias
RUN npm install

# Copiamos el resto de los archivos de la aplicación
COPY . .

# Compilamos la aplicación de React
RUN npm run build

# Stage 2: Servir la aplicación con Nginx
FROM nginx:alpine

# Copiamos los archivos estáticos (el resultado de 'npm run build') al directorio de Nginx
COPY --from=build /app/build /usr/share/nginx/html

# Expone el puerto por defecto de Nginx
EXPOSE 80

# Comando para iniciar Nginx
CMD ["nginx", "-g", "daemon off;"]
```

Captura de pantalla del Dockerfile correspondiente al servicio frontend

1.4.Orquestación con Docker Compose

El uso de Docker Compose simplifica el despliegue de la aplicación, ya que *permite construir y ejecutar todos los servicios definidos (backend y frontend) con un único comando*, en lugar de ejecutar manualmente múltiples builds y runs por separado. Cabe destacar que el archivo utiliza el Dockerfile previamente creado para construir la imagen de la aplicación.

```
version: '3.8'

services:
  frontend:
    # Construye la imagen del frontend desde la carpeta 'frontend'
    build:
      context: ./frontend
      dockerfile: Dockerfile
    container_name: deli-lunch-frontend
    # Mapea el puerto 80 del contenedor al puerto 3000 del host
    ports:
      - "3000:80"
    # El frontend depende del backend, así que lo iniciará primero
    depends_on:
      - backend

  backend:
    # Construye la imagen del backend desde la carpeta 'backend'
    build:
      context: ./backend
      dockerfile: Dockerfile
    container_name: deli-lunch-backend
    ports:
      - "4000:4000"
    volumes:
      - ./backend/db:/app/db

# Definimos una red para que ambos servicios puedan comunicarse
networks:
  default:
    name: deli-lunch-network
```

Captura de pantalla del Docker Compose

En pocas palabras, el archivo *docker-compose* permite levantar ambos contenedores con un único comando *docker compose up*. Cada servicio usa la imagen que construye su Dockerfile siendo útil tanto en desarrollo local como en staging o producción.

2. Configuración del Repositorio y CI

Link al repositorio público de GitHub: <https://github.com/TpDevops2025/DeliLunch.git>

La finalidad del capítulo es establecer un sistema de integración continua (CI) para asegurar la calidad y la coherencia del código.

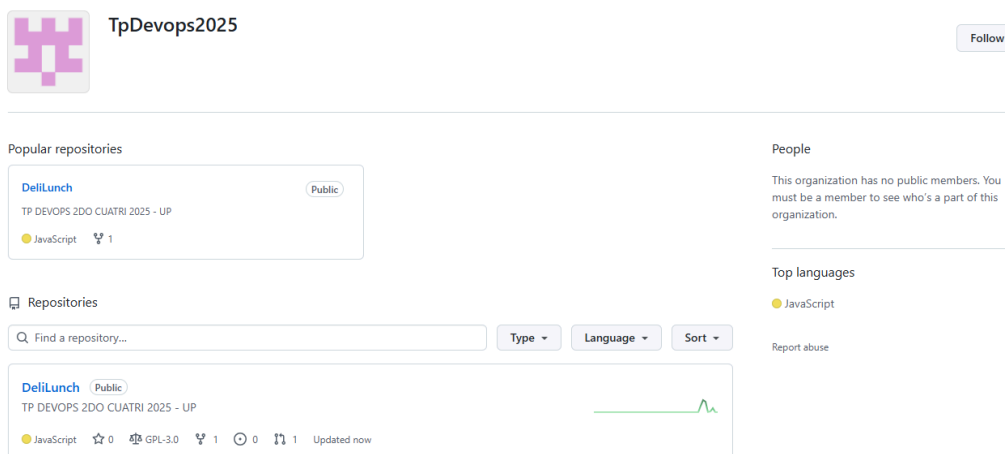
2.1. Gestión del Código Fuente

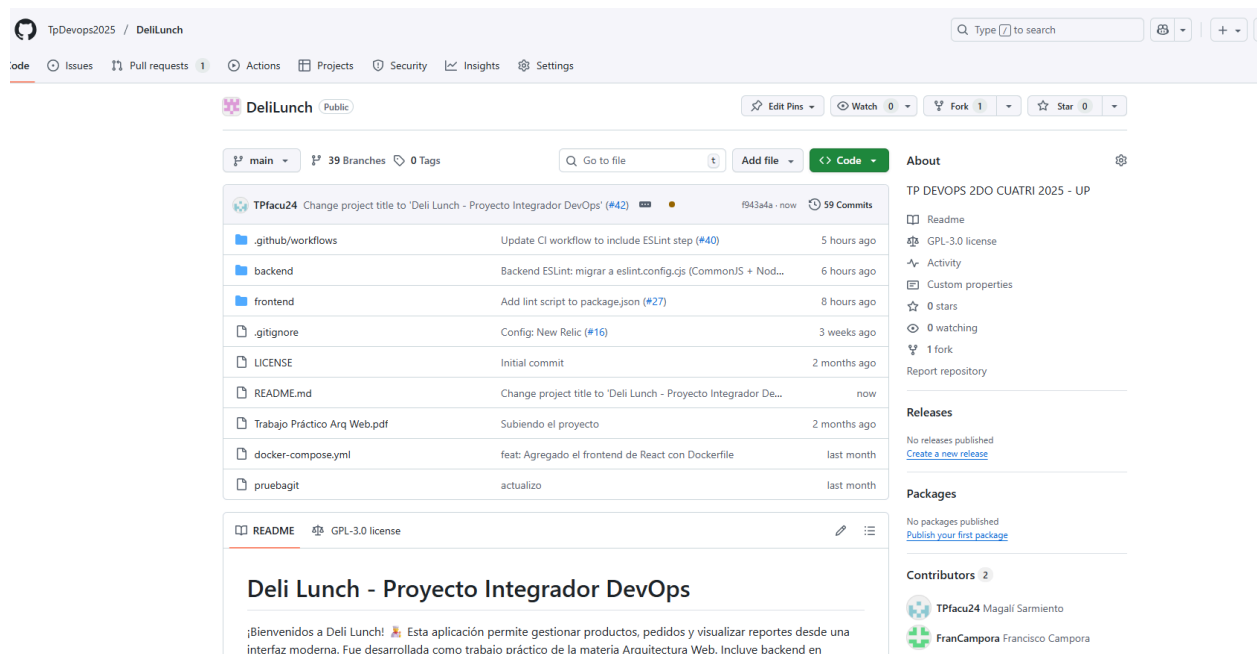
En pos de garantizar una efectiva gestión del código fuente, se creó un repositorio en GitHub para alojar y centralizar el código de la aplicación. Permite mantener una sincronización con el entorno local y trabajar en equipo. Cabe destacar que se creó el repositorio como organización siendo éste el punto central de colaboración, donde todos los cambios se registran y versionan.

Acciones realizadas:

- Creación del repositorio remoto en GitHub.
- Inicialización de Git en el proyecto local.
 - `git init`
 - `git add`
 - `git commit -m "Primer commit del proyecto"`
 - `git remote add origin https://github.com/TpDevops2025/DeliLunch.git`
 - `git push -u origin main`

Resultado: El proyecto queda alojado en GitHub y listo para CI.





Capturas de pantalla del repositorio en GitHub.

2.2. Configuración de Integración Continua (CI)

Archivo de Configuración de CI.

El flujo de CI asegura que cada cambio en el proyecto es verificado automáticamente, evitando errores antes de integrarlos a la rama principal, lo que garantiza calidad y consistencia en el desarrollo del proyecto.

En este caso específico se generan dos archivos de configuración específicos para el sistema de CI, dos GitHub Actions, uno correspondiente al frontend (React) y otro correspondiente al backend (Node).

Name	Last commit message
..	
backend-node.yml	Config: New Relic (#16)
docker-image.yml	Cambios en yml Docker por new relic (#21)
frontend-react.yml	Update frontend-react.yml para solucionar errores. (#12)

Captura de pantalla de los workflows generados en el repositorio en GitHub.

Workflow del backend.

El **workflow del backend** automatiza todo el proceso de integración continua del servidor en Node.js. Cada vez que se realiza un *push* o *pull request*, GitHub Actions ejecuta una serie de pasos: instala las dependencias, corre los tests configurados con Jest y Supertest, y valida que la API responda correctamente.

Si todas las verificaciones pasan, se continúa con la etapa de construcción de la imagen Docker del backend. Este flujo asegura que ningún cambio con errores pueda llegar a la rama principal y que la imagen generada esté lista para ser publicada en Docker Hub.

```
# CI del backend: instala deps, corre ESLint (flat config) y ejecuta tests.
name: Backend (Node)

on:
  pull_request:
    branches: [ "main" ]           # corre SIEMPRE en PR
  push:
    branches: [ "main" ]           # corre en main
    paths: [ "backend/**", ".github/workflows/backend-node.yml" ]

jobs:
  test:
    runs-on: ubuntu-latest
    defaults:
      run:
        working-directory: backend

    steps:
      - name: Checkout
        uses: actions/checkout@v4

      - name: Setup Node
        uses: actions/setup-node@v4
        with:
          node-version: "20"
          cache: npm
          cache-dependency-path: backend/package-lock.json

      - name: Install dependencies
        run: npm ci

      # ESLint v9 usando flat config (eslint.config.cjs en /backend)
      - name: ESLint (backend)
        run: npm run lint

      - name: Run tests (Jest)
        run: npm test -- --ci
        env:
          NEW_RELIC_APP_NAME: ${ secrets.NEW_RELIC_APP_NAME }
          NEW_RELIC_LICENSE_KEY: ${ secrets.NEW_RELIC_LICENSE_KEY }
          NEW_RELIC_ENABLED: false # evita enviar métricas desde los tests
```

*Captura de pantalla
del archivo
backend-node.yml*

Workflow del frontend.

El **workflow del frontend** verifica automáticamente que la aplicación React pueda compilarse sin errores y que todas las dependencias estén correctamente instaladas. Al igual que el backend, se ejecuta ante cada *push* o *pull request*, validando que el código sea funcional antes de fusionarse a la rama principal.

Además, en las etapas de build se genera una versión optimizada del sitio, lo que permite comprobar que el proceso de empaquetado y despliegue funcionen correctamente. Este flujo contribuye a mantener la coherencia entre desarrollo y producción, aplicando buenas prácticas de CI/CD en la capa visual de DeliLunch.

```
# CI del frontend: instala dependencias y compila el build de React.
name: Frontend (React)

on:
  pull_request:
    branches: [ "main" ]          # corre SIEMPRE en PR → el ruleset no queda "Expected"
  push:
    branches: [ "main" ]
    paths: [ "frontend/**", ".github/workflows/frontend-react.yml" ]

jobs:
  build:
    runs-on: ubuntu-latest
    defaults:
      run:
        working-directory: frontend
    steps:
      - name: Checkout
        uses: actions/checkout@v4

      - name: Setup Node
        uses: actions/setup-node@v4
        with:
          node-version: "20"
          cache: npm
          cache-dependency-path: frontend/package-lock.json

      - name: Install dependencies
        run: npm ci

      # 🟡 NUEVO: Lint del frontend
      - name: ESLint (frontend)
        run: npm run lint

      - name: Build React
        run: npm run build
```

*Captura de
pantalla del archivo
frontend-node.yml*

Cada vez que alguien hace un push o PR, GitHub Actions:

1. Descarga el repositorio.
2. Instala dependencias (npm ci o npm install).
3. Ejecuta pruebas (npm test).
4. Si todo pasa, marca el commit como válido.

Luego de crear ambos *workflows* verificamos que efectivamente estén corriendo correctamente. Para esto creamos un PR que cambie una porción mínima tanto del backend como del frontend para evaluar qué sucede luego de los cambios.

32 workflow runs

✓

Update index.html para probar CI

Docker Image CI #18: Pull request #6 opened by TPfacu24

Prueba-CI-frontend

🕒

1 minute ago

🕒

1m 13s

...

✓

Update index.html para probar CI

Frontend (React) #2: Pull request #6 opened by TPfacu24

Prueba-CI-frontend

🕒

1 minute ago

🕒

33s

...

✓

Update server.js

Docker Image CI #17: Pull request #5 synchronize by TPfacu24

test-backend-ci

🕒

6 minutes ago

🕒

1m 10s

...

✓

Update server.js

Backend (Node) #3: Pull request #5 synchronize by TPfacu24

test-backend-ci

🕒

6 minutes ago

🕒

14s


...

Capturas de pantalla de cambios éxitos en el repositorio de GitHub.

Mejoras y ajustes en la Integración Continua (CI) propuestas por el profesor.

Luego de las correcciones dadas por el profesor, se procedió a ajustar los workflows de backend y frontend con el fin de mejorar la calidad del código y asegurar la correcta ejecución del pipeline de Integración Continua.

En primer lugar, se incorporó un paso de análisis estático con ESLint en ambos workflows, lo que permite detectar errores de sintaxis y estilo antes de ejecutar los tests (en el backend) o la compilación (en el frontend). Además, se creó el archivo `eslint.config.cjs` permitiendo integrar la configuración en un único archivo compatible con entornos CommonJS, Node y Jest, y eliminando advertencias previas durante la ejecución del linting.



Durante el proceso de corrección se utilizó la herramienta GitHub Codespaces, que permitió realizar las modificaciones, instalar dependencias y probar los cambios directamente desde un entorno remoto en la nube, sin necesidad de entornos locales.

Con estos ajustes, el flujo de CI quedó conformado por tres etapas principales:

1. Instalación de dependencias (npm ci),
2. Ejecución del linting (npm run lint),
3. Validación mediante pruebas (backend) o compilación (frontend).

Estas mejoras fortalecen la práctica de Integración Continua (CI) en el proyecto, garantizando que ningún cambio se fusione a la rama principal sin haber superado previamente los controles de calidad y las pruebas automatizadas.

Protección de la Rama Principal.

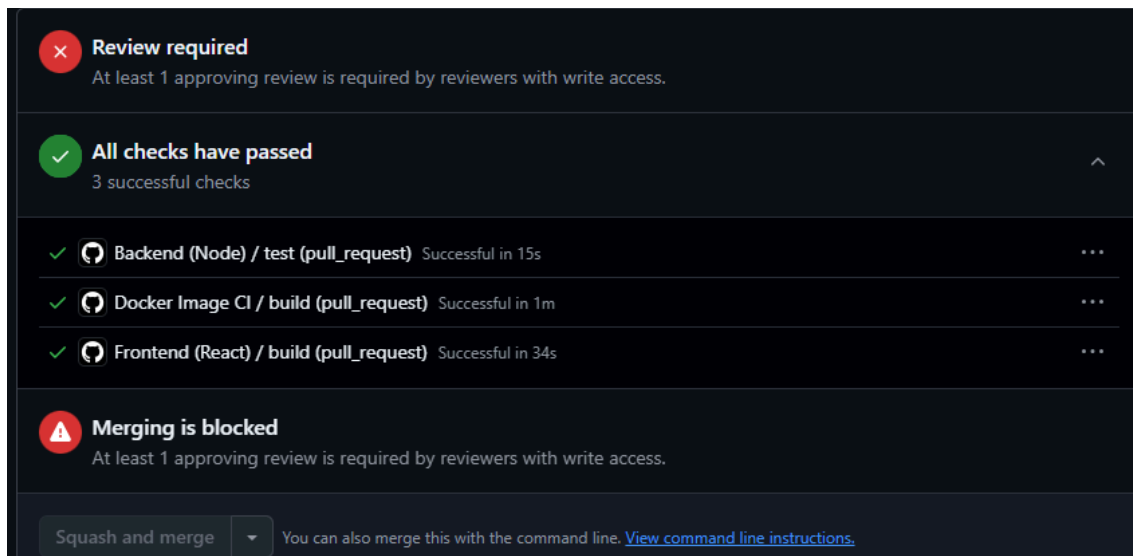
Con el objetivo de garantizar la calidad y estabilidad del proyecto, se configuró en GitHub la protección de la rama principal (main) mediante rulesets. Esta configuración impide que se realicen modificaciones directas sobre la rama y exige que todos los cambios ingresen a través de un Pull Request.

Dentro de la política definida, se establecieron los siguientes requisitos:

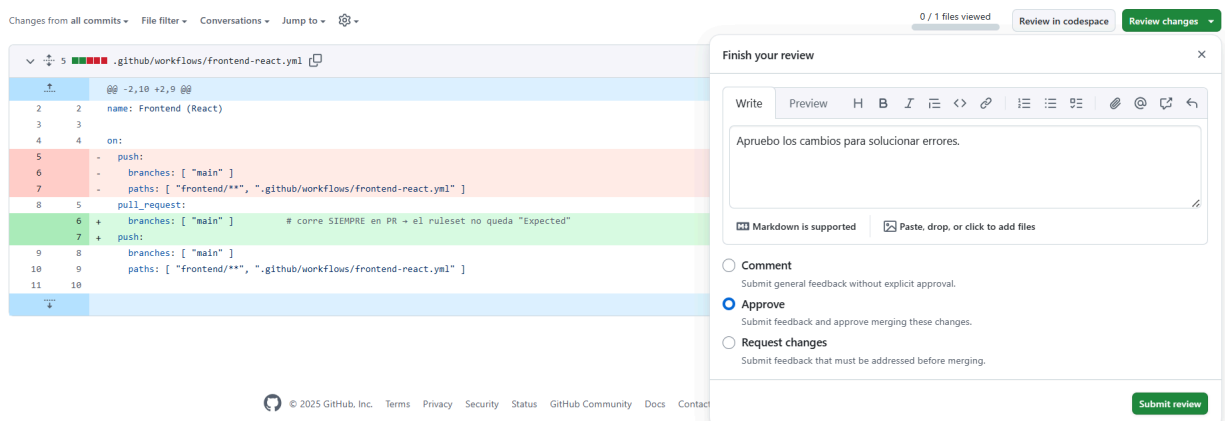
- **Revisión obligatoria de al menos un colaborador** antes de poder realizar el merge, asegurando así la validación de los cambios por otra persona del equipo.
- **Ejecución exitosa de los workflows de CI** como condición indispensable para fusionar ramas, de modo que ningún cambio se integre si las pruebas o la compilación fallan.
- **Bloqueo de pushes forzados y eliminaciones de la rama principal**, reforzando la integridad del historial de commits.

Gracias a esta configuración, la rama main se mantiene siempre en un estado estable, con código probado y validado, cumpliendo con las mejores prácticas de DevOps y con lo requerido en el trabajo práctico.

Creamos una rama → hacemos cambios → push → PR → revisión → merge a main.



Capturas de pantalla de solicitud de “Review Required” por existencia de política.



En conclusión, la protección de rama implementa la práctica “*main siempre estable*”, que es clave en DevOps.

2.3. Construcción y Publicación de la Imagen Docker.

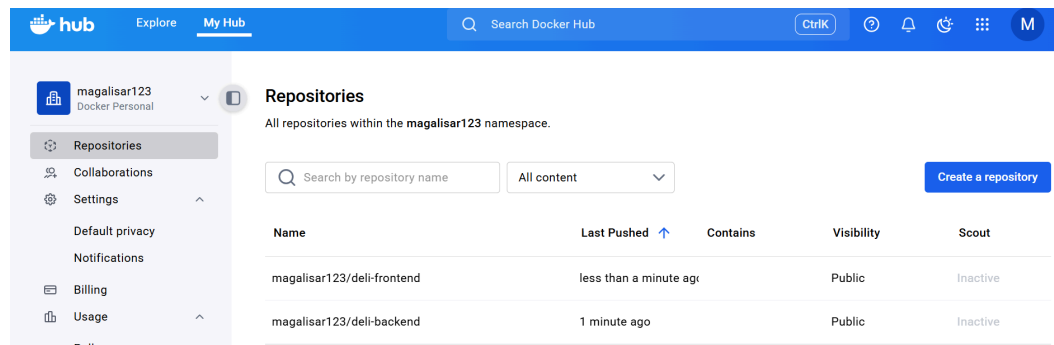
Se configura un workflow en GitHub Actions para la construcción y publicación automática de imágenes Docker. El flujo se activa en cada push a la rama principal (main) y realiza la construcción de las imágenes de backend y frontend a partir de los Dockerfile del proyecto. Posteriormente, las publica en un registro remoto de Docker Hub, bajo el repositorio del equipo, con dos etiquetas: latest (última versión estable) y el identificador único del commit (SHA) (trazabilidad)

Esta estrategia asegura portabilidad, consistencia y disponibilidad de las imágenes para todo el equipo de trabajo, permitiendo desplegar el sistema en cualquier entorno mediante un simple docker pull, cumpliendo con las mejores prácticas de integración continua (CI/CD).

1. Se crean dos repositorios públicos en Docker Hub con el fin de que las imágenes construidas en CI puedan ser descargadas y ejecutadas en cualquier máquina sin necesidad de reconstruirlas. Esto asegura **portabilidad y distribución** del sistema.

<https://hub.docker.com/repository/docker/magalisar123/deli-frontend/general>

<https://hub.docker.com/repository/docker/magalisar123/deli-backend/general>



Captura de pantalla de los repositorios creados en DockerHub.

2. Se genera un **token personal** en Docker Hub con permisos *Read, Write, Delete*. Este token permite a GitHub Actions **loguearse automáticamente en Docker Hub** para subir las imágenes, sin necesidad de usar contraseñas en texto plano. Se trata de una **mejor práctica de seguridad**.

Create access token

A personal access token is similar to a password except you can have many tokens and revoke access to each one at any time. [Learn more](#)

Access token description
github-actions-dellunch

Expiration date
90 days

Optional
Access permissions
Read, Write, Delete








Read, Write, Delete tokens allow you to manage your repositories.

Cancel Generate

Captura de pantalla de la creación del Token en DockerHub.

3. En el repositorio de GitHub se cargaron los secrets que permiten que los workflows usen credenciales de Docker Hub de forma segura y automática durante la publicación de imágenes. Así se protege la información sensible y se habilita la integración CI/CD.

Repository secrets New repository secret

Name 	Last updated
 DOCKERHUB_TOKEN	now  
 DOCKERHUB_USERNAME	now  

Captura de pantalla de la carga de los secrets en GitHub.

4. Se crea el archivo `.github/workflows/docker-image.yml` con dos comportamientos:
- En PR: construye las imágenes de backend y frontend para validar que los Dockerfile no tengan errores, pero no publica.
 - En push a main: construye las imágenes, inicia sesión en Docker Hub y publica dos versiones de cada una:
 - latest → última versión estable.
 - <sha> → identificador único del commit. Esto garantiza que cada cambio aprobado en main genere automáticamente una nueva versión de la aplicación lista para desplegar. Los tags latest permiten acceder rápido a la última versión, mientras que los tags sha aportan trazabilidad para saber qué commit generó cada imagen.

```
name: Docker Image CI

on:
  pull_request:
    branches: [ "main" ] # En PR: construye pero NO publica
  push:
    branches: [ "main" ] # En main: construye y publica
  workflow_dispatch:      # Permite lanzarlo manualmente desde la UI

env:
  IMAGE_BACKEND: ${ secrets.DOCKERHUB_USERNAME }/deli-backend
  IMAGE_FRONTEND: ${ secrets.DOCKERHUB_USERNAME }/deli-frontend
```

```

jobs:
  build_pr:
    if: github.event_name == 'pull_request'
    name: Build images (no push)
    runs-on: ubuntu-latest
    steps:
      - name: Checkout repository
        uses: actions/checkout@v6

      - name: Set up Docker Buildx
        uses: docker/setup-buildx-action@v3

      - name: Build backend image (no push)
        uses: docker/build-push-action@v6
        with:
          context: ./backend
          file: ./backend/Dockerfile
          push: false
          build-args: |
            NEW_RELIC_APP_NAME=${{secrets.NEW_RELIC_APP_NAME}}
            NEW_RELIC_LICENSE_KEY=${{secrets.NEW_RELIC_LICENSE_KEY}}
            # cuando se contruye la imagen del back se hace con las mismas config de new relic que en main

      - name: Build frontend image (no push)
        uses: docker/build-push-action@v6
        with:
          context: ./frontend
          file: ./frontend/Dockerfile
          push: false

  publish_main:
    if: github.event_name == 'push' && github.ref == 'refs/heads/main'
    name: Build & Push images
    runs-on: ubuntu-latest
    steps:
      - name: Checkout repository
        uses: actions/checkout@v6

      - name: Set up Docker Buildx
        uses: docker/setup-buildx-action@v3

      - name: Login to Docker Hub
        uses: docker/login-action@v3
        with:
          username: ${ secrets.DOCKERHUB_USERNAME }
          password: ${ secrets.DOCKERHUB_TOKEN }

      - name: Build & push backend
        uses: docker/build-push-action@v6
        with:
          context: ./backend
          file: ./backend/Dockerfile
          push: true
          build-args: |
            NEW_RELIC_APP_NAME=${{secrets.NEW_RELIC_APP_NAME}}
            NEW_RELIC_LICENSE_KEY=${{secrets.NEW_RELIC_LICENSE_KEY}}
          tags: |
            ${ env.IMAGE_BACKEND }:latest
            ${ env.IMAGE_BACKEND }:${{ github.sha }}

      - name: Build & push frontend
        uses: docker/build-push-action@v6
        with:
          context: ./frontend
          file: ./frontend/Dockerfile
          push: true
          tags: |
            ${ env.IMAGE_FRONTEND }:latest
            ${ env.IMAGE_FRONTEND }:${{ github.sha }}

      - name: Enviar a Render
        run: curl -X POST "${ secrets.RENDER_DEPLOY_HOOK_BACKEND }" &imageUri=${{ secrets.DOCKERHUB_USERNAME }}/${{ github.sha }}
        # curl viene con ubuntu es para abrir http osea links
        # -X es para que sea una peticion
        # POST es de http que es para enviar datos (chekear)
        # cambiamos de la imagen latest a -> github.sha que es el hash que se crea cuando commit

```

*Captura de
pantalla del
archivo
docker.image.yml*

5. Se verifica el flujo CI/CD asegurando que ningún cambio se fusione a la rama principal sin haber pasado por pruebas, build y verificación de imágenes. Además, la publicación automática reduce errores humanos y acelera los despliegues.


67 workflow runs		Event ▾	Status ▾	Branch ▾	Actor ▾
✓ chore: update docker-image.yml for build & push to Docker Hub (#15)	main	Docker Image CI #36: Commit b82385c pushed by magalisarmiento2003-cmyk	2 minutes ago 1m 33s	...	
✓ chore: update docker-image.yml for build & push to Docker Hub	TPFacu24-patch-11	Backend (Node) #12: Pull request #15 opened by TPFacu24	4 minutes ago 14s	...	
✓ chore: update docker-image.yml for build & push to Docker Hub	TPFacu24-patch-11	Docker Image CI #35: Pull request #15 opened by TPFacu24	4 minutes ago 1m 3s	...	
✓ chore: update docker-image.yml for build & push to Docker Hub	TPFacu24-patch-11	Frontend (React) #12: Pull request #15 opened by TPFacu24	4 minutes ago 36s	...	

Captura de pantalla de cambios correctamente integrados.





6. Verificación de las imágenes publicadas generadas (evidencia de portabilidad). Cualquier miembro del equipo (o incluso el docente) puede levantar el proyecto en segundos con un simple docker pull, sin necesidad de instalar Node, React o dependencias locales. Esto simplifica el despliegue y asegura que todos trabajen sobre la misma versión.

General Tags Image Management BETA Collaborators Webhook

Tags


 DOCKER SCOUT INACTIVE [Activate](#)

This repository contains 2 tag(s).

Tag	OS	Type	Pulled	Pushed
 b82385c9ba772dc2b...		Image	less than 1 day	3 minutes
 latest		Image	less than 1 day	3 minutes

[See all](#)

Captura de pantalla de imagen publicadas en DockerHub



En síntesis, cada push a main dispara el workflow docker; si los tests pasan, GitHub Actions hace login con los secrets y publica las imágenes en Docker Hub con tags latest y SHA. Así garantizamos portabilidad y reproducibilidad del despliegue.

Gracias a esta integración, cualquier persona puede acceder a la última versión de los contenedores oficiales del proyecto sin necesidad de reconstruirlos localmente. Sin embargo, el hecho de que las imágenes estén disponibles no implica que la aplicación se despliegue o ejecute automáticamente, ya que ese proceso se realiza desde la plataforma **Render**, que es la encargada del despliegue continuo.

3. Despliegue Continuo.

Para completar la automatización del flujo DevOps, se configuró un proceso de Despliegue Continuo (CD) que permite publicar automáticamente la aplicación DeliLunch cada vez que se aprueba un merge en la rama principal main.

El despliegue se gestiona mediante la plataforma Render, que actúa como entorno de producción y se integra directamente con el pipeline de GitHub Actions a través de un Webhook, permitiendo así una actualización inmediata de la aplicación sin intervención manual.

3.1. Hook de Render.

Se creó un servicio de tipo Web Service dentro de Render, utilizando la opción Deploy from Docker, de manera que el sistema tome las imágenes generadas y publicadas automáticamente en Docker Hub.

De esta forma, Render descarga siempre la versión más reciente (latest) de las imágenes del backend y del frontend.

Configure and deploy your new Web Service

Choose service > **2** Configure > **3** Deploy

Need help? Docs [?](#)

Source Code

Git Provider Public Git Repository **Existing Image**

Image URL

Deploy an image from a Docker registry

 docker.io/magalisar123/dellilunch-backend-sarmiento:latest ✓

Credential (Optional)

No credential ▼

Connect →

Captura de pantalla de la configuración del nuevo servicio en Render.

Durante la configuración del servicio se cargaron las variables de entorno necesarias para garantizar la correcta ejecución dentro del contenedor, incluyendo:

- **PORT:** Puerto donde se expone el servidor backend.
- **NEW_RELIC_APP_NAME y NEW_RELIC_LICENSE_KEY:** Claves necesarias para el monitoreo de la aplicación desde la plataforma New Relic.

Estas variables aseguran que tanto el backend como el frontend funcionen correctamente en el entorno cloud y que las métricas sean enviadas a la herramienta de observabilidad configurada.

3.2. Implementación del Webhook de Despliegue

Render provee una URL de Webhook que puede invocarse desde integraciones externas para ejecutar automáticamente un nuevo despliegue del servicio. Dicha URL fue guardada en el repositorio de GitHub como un secret bajo el nombre `RENDER_DEPLOY_HOOK`, garantizando así la seguridad y confidencialidad del dato.

Este Webhook se utiliza al final del pipeline de CI/CD, una vez que las imágenes Docker fueron correctamente construidas y publicadas en Docker Hub.

Deploy

Image
The image URL and credential used for your Web Service. [Edit](#)

Image URL
docker.io/magalisar123/deli-backend-latest

Credential (Optional)
No credential

Docker Command
Optionally override your Dockerfile's `CMD` and `ENTRYPOINT` instructions with a different command to start your service. [Edit](#)

Pre-Deploy Command Optional
Render runs this command before the start command. Useful for database migrations and static asset uploads. [Edit](#)

















Deploy Hook
Your private URL to trigger a deploy for this server. Remember to keep this a secret

.....

Captura de pantalla del Deploy Hook generado en Render.

Repository secrets

New repository secret

Name 	Last updated
 DOCKERHUB_TOKEN	last week  
 DOCKERHUB_USERNAME	last week  
 NEW_RELIC_APP_NAME	2 days ago  
 NEW_RELIC_LICENSE_KEY	2 days ago  
 RENDER_DEPLOY_HOOK_BACKEND	2 days ago  

Captura de pantalla de los secrets generados en GitHub.

3.3. Modificación del Workflow de GitHub Actions

En el archivo `.github/workflows/docker-image.yml` se añadió un nuevo paso al final del flujo de publicación, con el objetivo de notificar a Render que debe iniciar el despliegue.

El bloque agregado fue el siguiente:

```
- name: Enviar a Render
  run: curl -X POST "${{ secrets.RENDER_DEPLOY_HOOK_BACKEND }}"&imageUrl=${{secrets.DOCKERHUB_USERNAME}}/deli-backend:latest"
      # curl viene con ubuntu es para abrir http osea links
      # -X es para que sea una petición
      # POST es de http que es para enviar datos (chekear)
```

Captura de pantalla del bloque agregado al archivo `docker-image.yml`

De esta manera, GitHub Actions ejecuta el comando `curl` solo si las etapas previas (instalación de dependencias, pruebas unitarias, construcción y publicación de imágenes Docker) finalizaron exitosamente.

Si alguna de esas fases falla, el despliegue no se ejecuta, asegurando que solo versiones estables y probadas lleguen al entorno productivo.

3.4. Flujo completo de CI/CD

Cada vez que se aprueba un pull request o se realiza un merge hacia la rama `main`, el pipeline de CI/CD ejecuta las siguientes tareas de forma automática:

1. *Ejecución de pruebas unitarias y verificación de build:* garantiza la calidad y estabilidad del código antes de su publicación.
2. *Construcción de las imágenes Docker:* genera las versiones del backend y frontend listas para producción.
3. *Publicación en Docker Hub:* sube las imágenes con los tags `latest` y el identificador del commit (SHA) para asegurar trazabilidad.
4. *Despliegue en Render:* invoca el Webhook `RENDER_DEPLOY_HOOK`, lo que hace que Render descargue la nueva imagen `latest` y actualice el servicio en línea.

Gracias a esta integración, el proceso completo de `build` → `test` → `push` → `deploy` se realiza sin intervención manual, cumpliendo con los principios de Integración Continua (CI) y Despliegue Continuo (CD) propuestos en la consigna del trabajo práctico.

```

1  ▶ Run curl -X POST ""***&imageUrl=docker.io/***/delilunch-backend-sarmiento:latest"
8  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
9                      Dload  Upload  Total  Spent    Left  Speed
10
11  0      0    0     0     0     0     0     0     0  --:--:--  --:--:--  --:--:--    0
12  0      0    0     0     0     0     0     0     0  --:--:--  --:--:--  --:--:--    0
13  0      0    0     0     0     0     0     0     0  --:--:--  0:00:01  --:--:--    0
14  0      0    0     0     0     0     0     0     0  --:--:--  0:00:02  --:--:--    0
15 100    44 100    44     0     0    14     0  0:00:03  0:00:03  --:--:--   14
16 {"deploy":{"id":"dep-d3h9g7ruibrs73aopcug"}}

```

Captura de pantalla de la correcta ejecución de los cambios.



Deploy live for 4d52b5e  latest

October 5, 2025 at 1:10 PM

Deploy started for 4d52b5e  latest



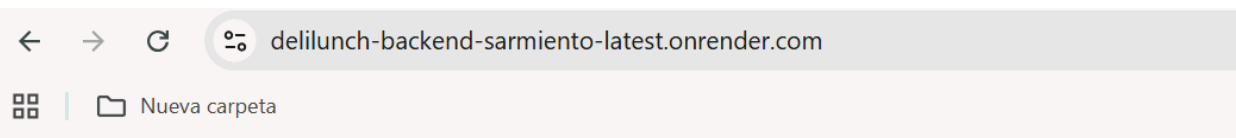
Triggered via Deploy Hook

October 5, 2025 at 1:09 PM

```

Oct 5 01:09:05 PM @ --> Starting service...
Oct 5 01:10:53 PM @ qxv7 Servidor corriendo en puerto 3000
Oct 5 01:10:53 PM @ qxv7 Ruta / llamada
Oct 5 01:10:57 PM @ --> Your service is live 🎉
Oct 5 01:10:57 PM @ -->
Oct 5 01:10:57 PM @ --> //////////////////////////////////////
Oct 5 01:10:58 PM @ -->
Oct 5 01:10:58 PM @ --> Available at your primary URL https://delilunch-backend-sarmiento-latest.onrender.com
Oct 5 01:10:58 PM @ -->
Oct 5 01:10:58 PM @ --> //////////////////////////////////////
Oct 5 01:11:00 PM @ qxv7 Ruta / llamada

```



¡Bienvenido a la API de DeliLunch! La API está corriendo.

Capturas de pantalla de la aplicación corriendo correctamente.

3.5. Resultados obtenidos

Con la implementación del despliegue automatizado, la aplicación DeliLunch queda disponible en su entorno de producción minutos después de aprobar un cambio, garantizando:

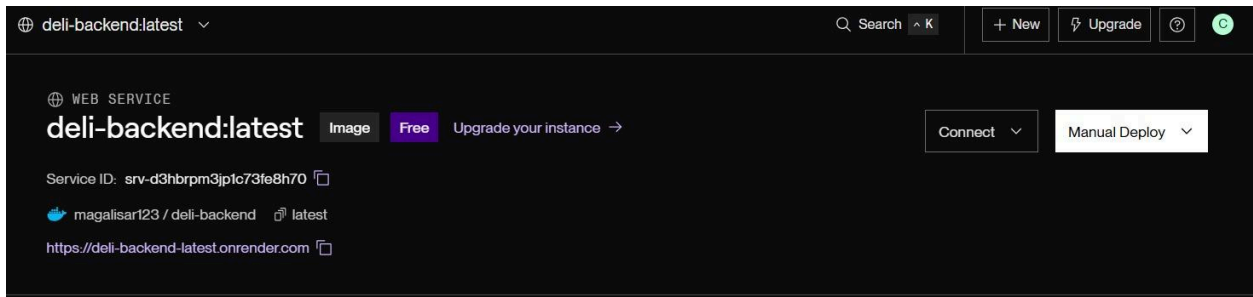
- **Velocidad de entrega:** la nueva versión se publica automáticamente sin necesidad de intervención manual.
- **Seguridad y trazabilidad:** el pipeline solo se ejecuta si los tests y la construcción fueron exitosos.
- **Reproducibilidad:** cualquier integrante del equipo puede verificar el proceso revisando los logs del workflow en GitHub Actions.
- **Estabilidad:** el entorno Render siempre ejecuta la versión más reciente y estable del proyecto.

En síntesis, esta etapa permitió cerrar el ciclo completo de CI/CD, logrando que la aplicación DeliLunch se mantenga actualizada, estable y lista para ser monitoreada mediante las herramientas implementadas en la siguiente fase del trabajo práctico.

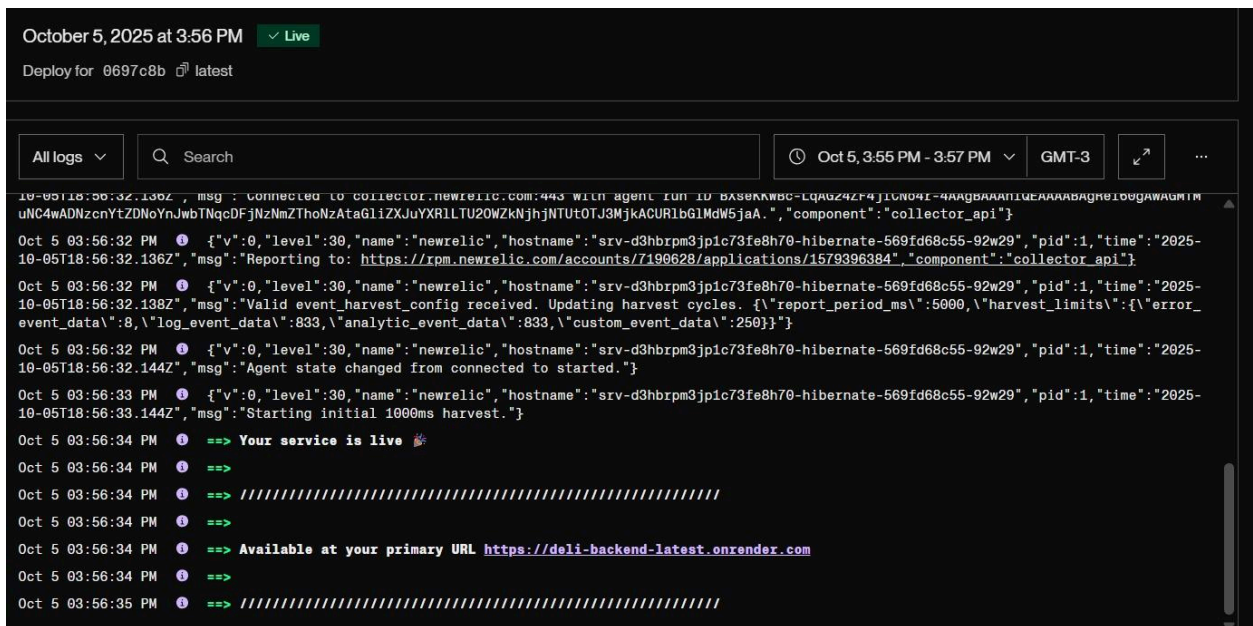
El pipeline CI/CD integra GitHub Actions, Docker Hub y Render. Cada push a main construye las imágenes Docker del backend y frontend y las publica en Docker Hub. Luego, mediante un Deploy Hook, Render despliega automáticamente la imagen actualizada del backend. Esto permite que cada cambio se refleje en producción sin intervención manual. Cabe destacar y aclarar que **el despliegue automático se aplica sólo al backend, ya que es el servicio que ejecuta la lógica y endpoints de la API.**

SERVICE NAME 1	STATUS	RUNTIME	REGION	DEPLOYED ↓
🌐 deli-backend:latest	✓ Deployed	Image	Oregon	2d ...

Status general del proyecto en Render



Preview del proyecto en Render



Último estado de los logs con el servicio en línea.

4. Monitoreo y Análisis.

Dentro del proyecto DeliLunch, el monitoreo cumple un rol clave para garantizar que tanto el backend como el frontend se mantengan disponibles y funcionando de manera correcta. Esta etapa permite supervisar el estado de los contenedores desplegados, validar que los endpoints de salud respondan adecuadamente y detectar posibles fallas de rendimiento o interrupciones en el servicio.

La incorporación de mecanismos de monitoreo en el flujo DevOps del trabajo práctico asegura una visión en tiempo real del sistema, aportando datos que permiten anticipar problemas, mantener la aplicación estable y mejorar la experiencia de los usuarios.

El servicio elegido para la etapa de monitoreo y análisis fue **New Relic** (<https://newrelic.com>), una herramienta que permite observar en detalle el rendimiento y el estado de los servicios en ejecución.

Para su implementación se siguieron los siguientes pasos:

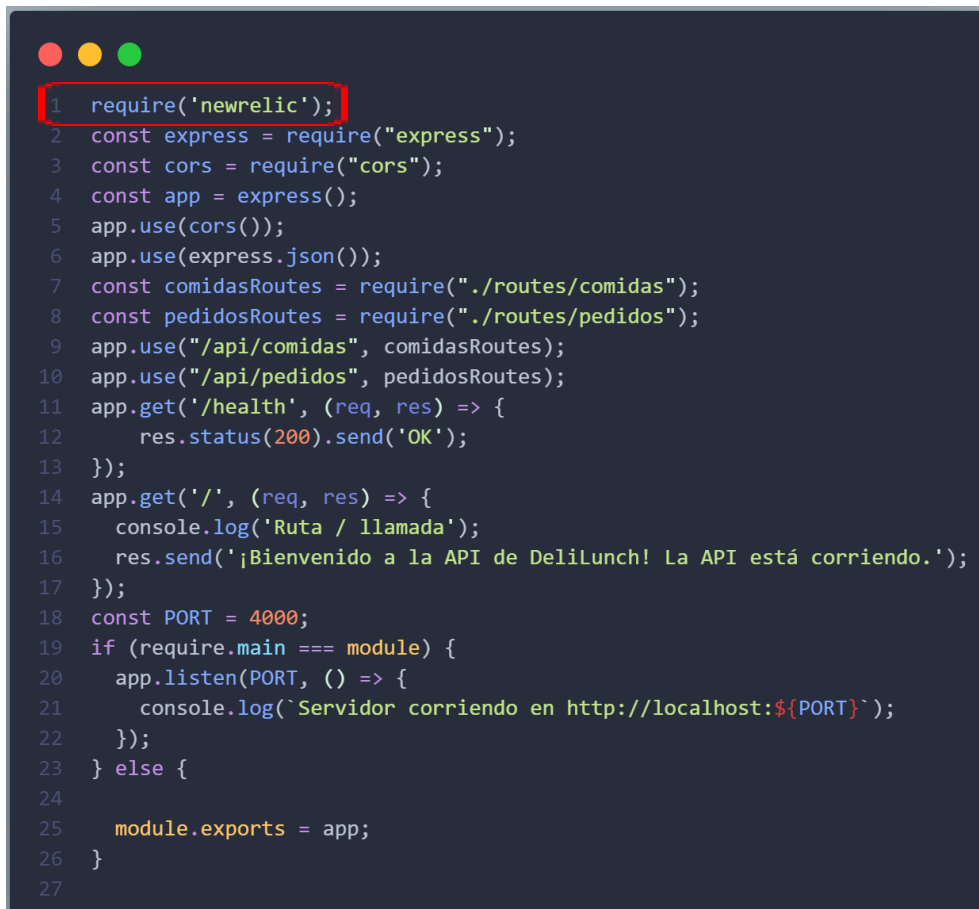
1. **Creación de una cuenta en New Relic**, lo que habilitó el acceso al panel de monitoreo.
2. En el menú principal se accedió a la sección **“Integration & Agents”** para configurar la integración.
3. De las opciones disponibles para la instalación del agente, se seleccionó la alternativa **Docker**, adaptada al despliegue de la aplicación.
4. Se generó una **License Key**, necesaria para autenticar la comunicación entre la aplicación y la plataforma de New Relic.
 - A partir de este punto, se documentan y adjuntan capturas con las **modificaciones realizadas en el código y configuración** del proyecto, con el fin de permitir la correcta instrumentación y recolección de métricas en la aplicación *DeliLunch*.
5. En la carpeta backend, dentro del archivo package.json, agregamos la librería New Relic como dependencia. Esto sirve para que el backend pueda enviar datos de cómo está funcionando a la plataforma de monitoreo. La configuración fue bastante sencilla: solo se instaló la dependencia y se agregó la referencia a New

Relic en la última línea del código del servidor, de forma que cada vez que la app arranca el agente se activa y empieza a mandar métricas a New Relic.



```
1  {
2    "name": "backend",
3    "version": "1.0.0",
4    "main": "index.js",
5    "scripts": {
6      "init-db": "node initDB.js",
7      "test": "jest",
8      "dev": "nodemon server.js",
9      "start": "node server.js"
10   },
11   "keywords": [],
12   "author": "Magali Sarmiento",
13   "license": "ISC",
14   "description": "- Node.js instalado",
15   "dependencies": {
16     "axios": "^1.9.0",
17     "cors": "^2.8.5",
18     "express": "^5.1.0",
19     "sqlite3": "^5.1.7"
20   },
21   "devDependencies": {
22     "jest": "^29.7.0",
23     "nodemon": "^3.0.0",
24     "supertest": "^7.1.0",
25     "newrelic": "latest"
26   }
27 }
28
```

6. Para que el monitoreo se inicie cada vez que arranca la aplicación, se inserta la referencia a New Relic en la primera línea del módulo principal, que en nuestro caso es el archivo `server.js`. De esta forma, apenas se ejecuta el servidor, el agente de New Relic ya empieza a mandar métricas y datos de rendimiento a la plataforma.



```
1 require('newrelic');
2 const express = require("express");
3 const cors = require("cors");
4 const app = express();
5 app.use(cors());
6 app.use(express.json());
7 const comidasRoutes = require("./routes/comidas");
8 const pedidosRoutes = require("./routes/pedidos");
9 app.use("/api/comidas", comidasRoutes);
10 app.use("/api/pedidos", pedidosRoutes);
11 app.get('/health', (req, res) => {
12   res.status(200).send('OK');
13 });
14 app.get('/', (req, res) => {
15   console.log('Ruta / llamada');
16   res.send('¡Bienvenido a la API de DeliLunch! La API está corriendo.');
```

7. En el backend/Dockerfile se configuró el agente de New Relic para que el monitoreo se active automáticamente dentro del contenedor.
- Se declaran los argumentos (ARG) NEW_RELIC_APP_NAME y NEW_RELIC_LICENSE_KEY, que permiten recibir los valores de configuración enviados desde el pipeline de GitHub Actions.
 - Luego, estos argumentos se convierten en variables de entorno (ENV) para que Node.js pueda acceder a ellos en tiempo de ejecución.
 - También se añaden variables específicas de configuración de New Relic:
 - i. NEW_RELIC_NO_CONFIG_FILE=true: ejecuta el agente sin requerir un archivo de configuración externo.

- ii. `NEW_RELIC_DISTRIBUTED_TRACING_ENABLED=true`: habilita el tracing distribuido para seguir las transacciones de extremo a extremo.
- iii. `NEW_RELIC_LOG=stdout`: redirige los logs del agente a la consola del contenedor, permitiendo visualizarlos directamente desde Render.

Gracias a esta configuración, el agente de New Relic se inicializa automáticamente en cada despliegue, enviando métricas, trazas y logs a la plataforma para un monitoreo en tiempo real del rendimiento de la aplicación.

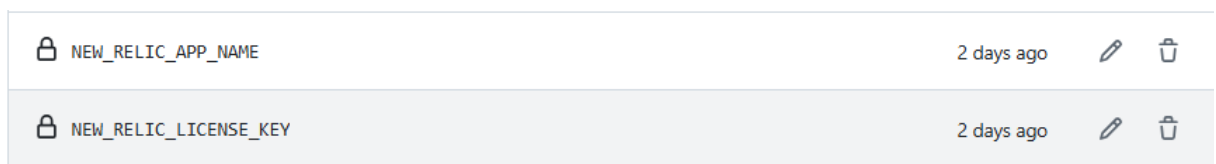
```
1 # Stage 1: Construcción
2 FROM node:18-alpine AS build
3
4 WORKDIR /app
5
6 # Copiamos todo el proyecto al contexto de construcción del stage 1
7 COPY . .
8
9 # Instalamos las dependencias
10 RUN npm install
11
12 # Stage 2: Producción
13 FROM node:18-alpine
14
15 WORKDIR /app
16
17 #Argumentos para newrelic
18 ARG NEW_RELIC_APP_NAME
19 ARG NEW_RELIC_LICENSE_KEY
20
21 # Copiamos solo los archivos necesarios del stage 1 al stage 2
22 COPY --from=build /app ./
23
24 #Convertimos los argumentos en variables de entorno para que node los lea
25 ENV NEW_RELIC_APP_NAME=$NEW_RELIC_APP_NAME
26 #nombre de la app
27 ENV NEW_RELIC_LICENSE_KEY=$NEW_RELIC_LICENSE_KEY
28 #license key de la app
29
30
31 # Configuración de new relic solo con variables de entorno
32 ENV NEW_RELIC_NO_CONFIG_FILE=true
33 #significa que va a correr el agente sin un file de config - parametro dado por new relic
34 ENV NEW_RELIC_DISTRIBUTED_TRACING_ENABLED=true
35 #tracing activado - parametro dado por new relic
36 ENV NEW_RELIC_LOG=stdout
37 #logs a la consola - parametro dado por new relic
38
39 EXPOSE 4000
40
41 CMD ["node", "server.js"]
```

8. Finalmente, se le asignó el nombre “DeliLunch” a la aplicación dentro de la configuración de New Relic. Esto se hace para poder identificar fácilmente el servicio en el panel de la plataforma y distinguirlo de otras posibles aplicaciones.

De esta manera, cada vez que se generan métricas, logs o trazas, estos aparecen asociados con el nombre “DeliLunch”, lo que facilita su monitoreo y análisis.

9. Dentro del repositorio también se crearon dos nuevos **secrets** en la sección *Settings* → *Secrets and variables* → *Actions*:

- NEW_RELIC_APP_NAME: cuyo valor es el nombre de la aplicación configurada en la plataforma, en este caso “DELILUNCH”.
- NEW_RELIC_LICENSE_KEY: cuyo valor corresponde a la license key generada en el paso de configuración inicial de New Relic.



Captura de pantalla de los secrets generados en GitHub.

10. Dentro del archivo YAML ubicado en `.github/workflows`, se realizaron ajustes para integrar correctamente New Relic en el proceso de construcción y despliegue del contenedor Docker.

- En el workflow del backend, se modificó el step correspondiente a la ejecución de tests, con el objetivo de excluir los datos generados por las pruebas del monitoreo. De esta forma, las métricas reflejadas en New Relic representan únicamente el comportamiento real de la aplicación en ejecución, sin interferencias del entorno de testing.

```

1  # CI del backend: instala dependencias y ejecuta los tests de la API.
2  name: Backend (Node)
3
4  on:
5    pull_request:
6      branches: [ "main" ]           # corre SIEMPRE en PR → el ruleset no queda "Expected"
7    push:
8      branches: [ "main" ]
9      paths: [ "backend/**", ".github/workflows/backend-node.yml" ]
10
11  jobs:
12    test:
13      runs-on: ubuntu-latest
14      defaults:
15        run:
16          working-directory: backend
17      steps:
18        - name: Checkout
19          uses: actions/checkout@v4
20
21        - name: Setup Node
22          uses: actions/setup-node@v4
23          with:
24            node-version: "20"
25            cache: npm
26            cache-dependency-path: backend/package-lock.json
27
28        - name: Install dependencies
29          run: npm ci
30
31        - name: Run tests
32          run: npm test -- --ci
33          env:
34            NEW_RELIC_APP_NAME: ${secrets.NEW_RELIC_APP_NAME}
35            NEW_RELIC_LICENSE_KEY: ${secrets.NEW_RELIC_LICENSE_KEY}
36            NEW_RELIC_ENABLED: false
37            #Así la data de los test no queda en las métricas
38

```

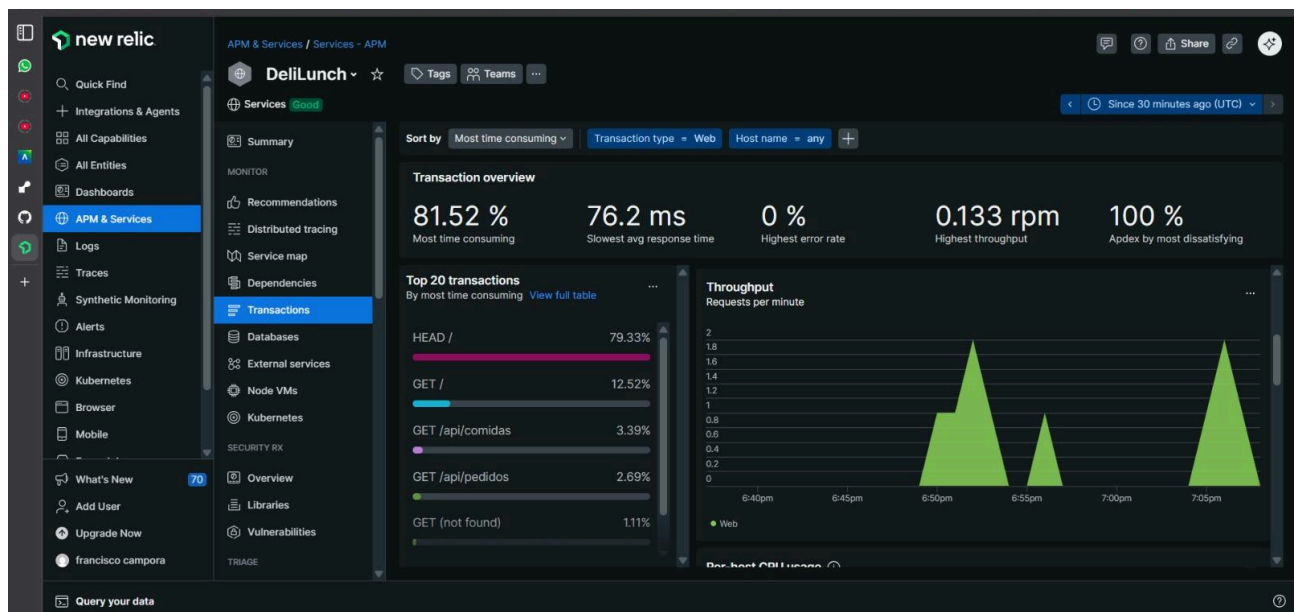
- En el workflow de Docker, se efectuó el cambio más relevante para la implementación del monitoreo. En el step denominado “Build & push backend”, se añadieron los parámetros build-args, los cuales permiten pasar variables de entorno al momento de construir la imagen. Allí se declararon las claves NEW_RELIC_APP_NAME y NEW_RELIC_LICENSE_KEY, tomando sus valores desde los secrets configurados previamente en GitHub Actions. Esto garantiza que el agente de New Relic quede correctamente inicializado dentro del contenedor al momento del despliegue.

```

1 - name: Build & push backend
2   uses: docker/build-push-action@v6
3   with:
4     context: ./backend
5     file: ./backend/Dockerfile
6     push: true
7     build-args: |
8       NEW_RELIC_APP_NAME=${{secrets.NEW_RELIC_APP_NAME}}
9       NEW_RELIC_LICENSE_KEY=${{secrets.NEW_RELIC_LICENSE_KEY}}
10    tags: |
11      ${ env.IMAGE_BACKEND } :latest
12      ${ env.IMAGE_BACKEND } :${ github.sha }

```

Pasando al panel se confirma que el agente de New Relic está funcionando correctamente y recibe métricas del backend de *DeliLunch*. La aplicación muestra un tiempo de respuesta promedio estable (76 ms), sin errores y con trazas activas en todos los endpoints. Esto demuestra que el monitoreo permite evaluar el rendimiento en tiempo real y detectar posibles cuellos de botella en la API.



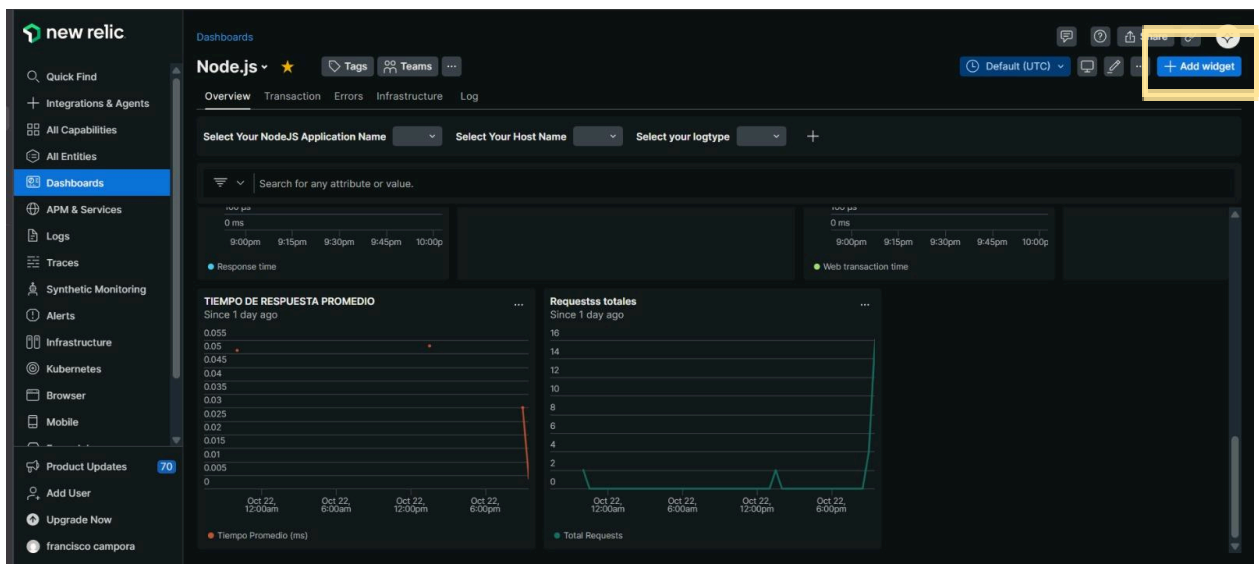
Métricas en New Relic.

Para optimizar el uso de New Relic, incorporamos diferentes widgets en el dashboard con el objetivo de facilitar la interpretación de los datos y el monitoreo del rendimiento de la aplicación. Los widgets seleccionados fueron:

- **Tasa de errores**, para visualizar la proporción de fallos en las solicitudes.
- **Tiempo de respuesta promedio**, que permite evaluar la eficiencia del servicio.
- **Cantidad total de requests**, útil para analizar el volumen de tráfico recibido.

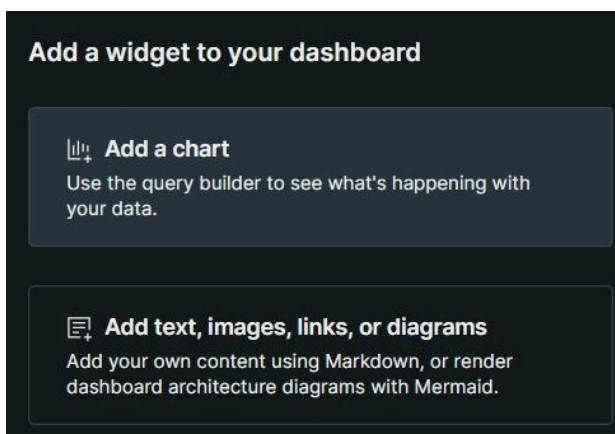
Procedimiento para agregar un widget al dashboard de New Relic.

1. Dentro del dashboard en la esquina derecha superior clickeamos en “add widget”.



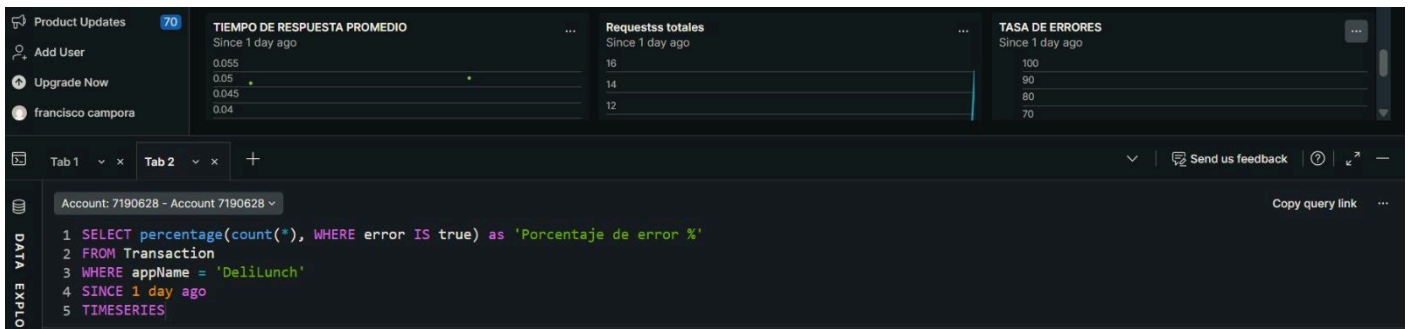
Captura de pantalla de la plataforma de New Relic para agregar un widget.

2. Elegimos la opción de “Add a chart”



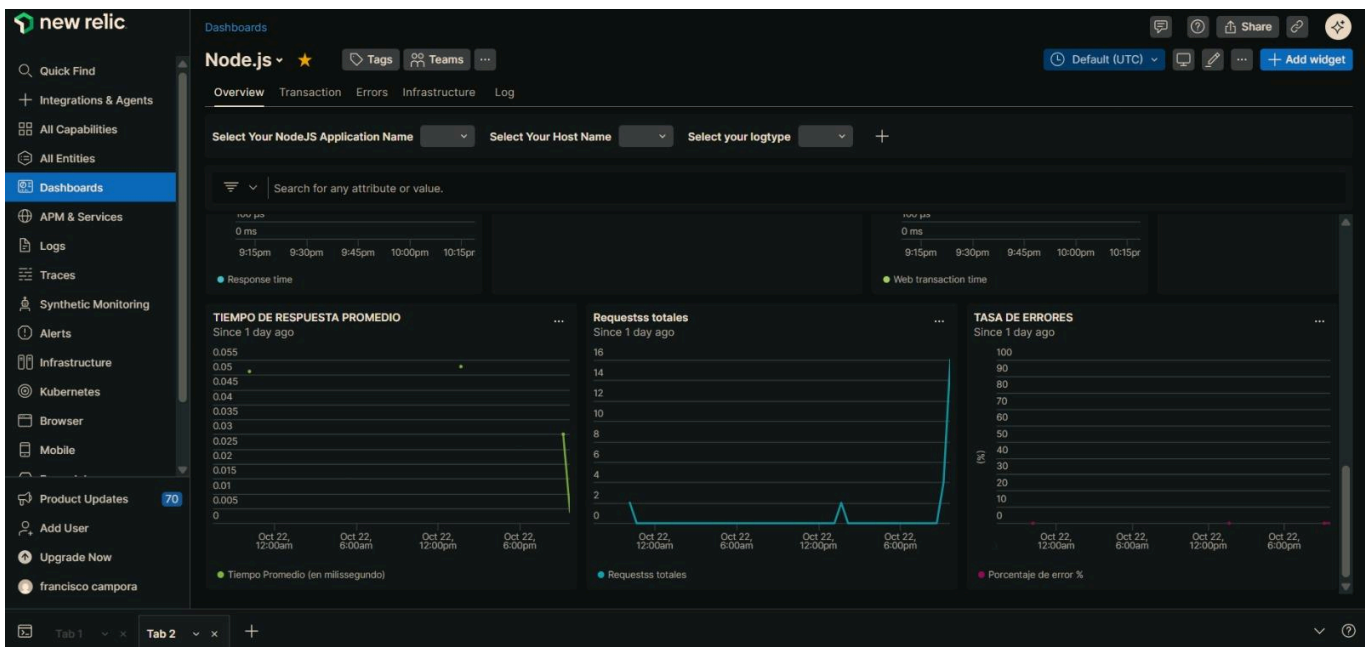
Captura de pantalla de la plataforma de New Relic para configurar un widget.

3. Colocar la query NRQL que se desea ejecutar para obtener los datos.



Captura de pantalla de la query de New Relic para configurar un widget.

4. Luego de ingresar la query NRQL, se presiona “Run” para ejecutar el código y establecer la conexión con los datos. New Relic mostrará automáticamente un gráfico con los resultados obtenidos. Finalmente, se asigna un título descriptivo al widget y se incorpora al dashboard mediante la opción “Add to dashboard”.
5. Visualizamos en el dashboard los widgets agregados.



Captura de pantalla de los widgets generados en New Relic.



5. Conclusión.

Documentar lecciones aprendidas + desafíos que tuvimos que afrontar durante el tp.