Documentación Técnica

Automatización CI/CD con Docker y GitHub Actions

Autor: John Arenales

Asignatura: Afondamiento nas Competencias Profesionais

Actividad 1_4: Docker

Fecha: December 7, 2025

Contents

1 Introducción

1.1 Objetivo y Alcance del Documento

El propósito de este documento es proporcionar una **guía técnica completa y profesional** para la implementación, despliegue y mantenimiento de una solución de **Integración Continua y Despliegue Continuo (CI/CD)** utilizando Docker, GitHub Actions y notificaciones por Telegram.

Este documento cubre:

1. **Arquitectura del Sistema:** Descripción de la aplicación API REST y su estructura tecnológica.
2. **Contenerización:** Empaquetamiento de la aplicación Node.js/Express en contenedores Docker portables.
3. **Orquestación Local:** Definición y ejecución de servicios múltiples (API y MongoDB) mediante Docker Compose.
4. **Automatización CI/CD:** Configuración de flujos de trabajo en GitHub Actions para construcción y publicación automática de imágenes.
5. **Notificaciones:** Integración de alertas en tiempo real mediante Telegram.

El alcance incluye la documentación técnica, códigos ejecutables y procedimientos operacionales para desarrolladores y equipos de DevOps.

1.2 Visión General del Proyecto

Este proyecto implementa una **solución integral de DevOps** para una API RESTful de gestión de usuarios y grupos. La aplicación principal está desarrollada en **Node.js con Express** (versión 5.1.0) y utiliza **MongoDB** (mongo:latest) como base de datos NoSQL.

1.2.1 Objetivos Clave

- **Portabilidad:** La aplicación funciona de manera idéntica en cualquier máquina que tenga Docker instalado, eliminando la fricción entre entornos de desarrollo, prueba y producción.
- **Automatización:** Cada cambio en el repositorio desencadena automáticamente un flujo de construcción, prueba y publicación de la imagen Docker sin intervención manual.
- **Trazabilidad:** El equipo de desarrollo recibe notificaciones inmediatas en Telegram sobre el estado de cada despliegue, mejorando la comunicación y la respuesta ante fallos.

1.2.2 Flujo del Proyecto

1. El desarrollador realiza un `git push` al repositorio en GitHub en las ramas `main` o `master`.
2. GitHub Actions detecta el cambio y dispara el flujo de trabajo `build-and-push`.
3. El flujo construye una nueva imagen Docker de la API y la sube a Docker Hub.
4. Se envía una notificación a Telegram confirmando el éxito (o reporte del error).
5. La imagen está lista para ser desplegada en cualquier entorno que necesite la aplicación.

2 Arquitectura y Stack Tecnológico

2.1 Stack Resumen

La siguiente tabla presenta todos los componentes tecnológicos utilizados en este proyecto:

Table 1: Componentes del Stack Tecnológico

Categoría	Tecnología/Herramienta	Versión/Especificación	Rol en el Proyecto
Runtime	Node.js	v25.2.1	Entorno de ejecución de la API RESTful.
Framework Web	Express.js	v5.1.0	Marco de trabajo para construir y servir la API REST de gestión de usuarios y grupos.
Base de Datos	MongoDB	mongo:latest	Base de datos NoSQL para la persistencia de datos de usuarios y grupos.
Contenedorización	Docker	Dockerfile	Empaquetamiento de la API Node.js en una imagen portable e inmutable.
Orquestación Local	Docker Compose	v3.8	Definición y ejecución de la aplicación multi-contenedor (API + MongoDB) en desarrollo.
Control de Versiones	GitHub	Repositorio	Alojamiento del código fuente y plataforma base para CI/CD.
CI/CD	GitHub Actions	docker-push.yml	Automatización del pipeline: checkout, build, push y notificación.
Registro de Imágenes	Docker Hub	johny050824/api_docker_compose	Repositorio centralizado para almacenar y distribuir imágenes Docker.
Notificaciones	Telegram	appleboy/telegram-action	Canal de comunicación en tiempo real para alertas de despliegue.

2.2 Componentes Clave del Sistema

2.2.1 API de Gestión

La API es una aplicación Node.js con Express que proporciona endpoints RESTful para operaciones CRUD (Create, Read, Update, Delete) sobre dos entidades principales:

- **Usuarios:** Registros con campos como nombre, apellido, edad y teléfono.
- **Grupos:** Colecciones de usuarios que pueden organizarse bajo un nombre común.

La API establece una conexión con MongoDB mediante la URI especificada en la variable de entorno `MONGO_URI`. Esta variable permite que la configuración se adapte dinámicamente según el entorno (desarrollo, prueba, producción) sin cambiar el código.

2.2.2 Archivos de Configuración (Infrastructure as Code - IaC)

Los siguientes archivos actúan como la "infraestructura como código" (IaC) y definen completamente cómo debe construirse, ejecutarse y desplegarse la aplicación:

- **Dockerfile:** Define la imagen Docker de la API especificando la imagen base, dependencias, puertos expuestos y comandos de inicio.
- **docker-compose.yml:** Orquesta los servicios de MongoDB y API localmente, definiendo redes, volúmenes y dependencias.
- **docker-push.yml (Workflow):** Automatiza el pipeline de integración continua en GitHub Actions.

- **.Dockerignore:** Especifica archivos que no deben incluirse en la imagen Docker (node_modules, .env, .git).

3 Requisitos del Sistema

3.1 Requisitos Funcionales (RF)

La siguiente tabla define los requisitos funcionales que el sistema debe cumplir:

Table 2: Requisitos Funcionales del Sistema

ID	Requisito	Descripción
RF1	API de Gestión	La aplicación debe exponer endpoints REST para ejecutar operaciones CRUD completas sobre Usuarios y Grupos almacenados en MongoDB.
RF2	Ejecución Unificada	El desarrollador debe poder iniciar toda la infraestructura (API + MongoDB) con un único comando: `docker-compose up -d --build`.
RF3	Despliegue Automático	Cada `git push` a las ramas `main` o `master` desencadena automáticamente la construcción y publicación de la imagen en Docker Hub.
RF4	Notificación de Estado	Al completar el despliegue automático, se envía un mensaje de Telegram indicando éxito o fracaso del proceso.
RF5	Portabilidad	La API debe ejecutarse sin cambios dentro de su contenedor Docker, sin depender de software instalado directamente en el equipo anfitrión.

3.2 Requisitos No Funcionales (RNF)

La siguiente tabla define los atributos de calidad y restricciones que el sistema debe satisfacer:

Table 3: Requisitos No Funcionales del Sistema

ID	Requisito	Descripción
RNF1	Seguridad	Las credenciales sensibles (Docker Hub, Telegram) se almacenan como **GitHub Secrets**, nunca expuestas en código fuente ni archivos de configuración.
RNF2	Disponibilidad	La imagen Docker en Docker Hub debe ser accesible 24/7 para descargas y despliegues en cualquier momento.
RNF3	Documentación	El flujo de trabajo debe estar completamente documentado y comentado para permitir mantenimiento futuro.
RNF4	Configurabilidad	Los parámetros de conexión a MongoDB (URI, puerto, base de datos) se cargan mediante variables de entorno, permitiendo adaptación a diferentes contextos.
RNF5	Mantenibilidad	El código LaTeX y los scripts deben seguir estándares profesionales de formato e indentación.

4 Instrucciones de Uso Local

4.1 Prerrequisitos

Antes de ejecutar el proyecto localmente, asegúrate de tener instalados los siguientes componentes:

- **Docker:** https://www.docker.com/products/docker-desktop - Versión 20.10 o superior.
- **Docker Compose:** Incluido en Docker Desktop. Verifica con `docker-compose --version`.

- **Git:** https://git-scm.com/ - Para clonar el repositorio.
- **Acceso al Repositorio:** Permisos de lectura en https://github.com/JohnPV894/practica_express.js_docker.

4.2 Ejecución de la Aplicación

4.2.1 Paso 1: Clonar el Repositorio

```
git clone https://github.com/JohnPV894/practica_express.js_docker.git
cd practica_express.js_docker
```

4.2.2 Paso 2: Construir e Iniciar los Servicios

```
docker-compose up -d --build
```

Este comando:

1. Construye la imagen Docker de la API usando el `Dockerfile`.
2. Descarga la imagen oficial de MongoDB.
3. Crea y inicia ambos contenedores en segundo plano (`-d`).
4. Expone la API en `http://localhost:3000` y MongoDB en `localhost:27017`.

4.2.3 Paso 3: Verificar el Estado

```
docker-compose ps
```

Deberías ver dos contenedores en ejecución: `api_docker_compose` y `mongodb_gestion`.

4.3 Prueba de la API

La API expone los siguientes endpoints para gestionar Usuarios y Grupos. A continuación se presentan ejemplos de uso con `curl`:

4.3.1 Crear un Usuario

```
curl -X POST http://localhost:3000/usuarios \
  -H "Content-Type: application/json" \
  -d '{
    "nombre": "Juan",
    "apellido": "P rez",
    "edad": 28,
    "telefono": "555-1234"
  }'
```

Respuesta esperada:

```
{
  "_id": "507f1f77bcf86cd799439011",
  "nombre": "Juan",
  "apellido": "P rez",
  "edad": 28,
  "telefono": "555-1234"
}
```

4.3.2 Obtener Todos los Usuarios

```
curl http://localhost:3000/usuarios
```

4.3.3 Obtener un Usuario por ID

```
curl http://localhost:3000/usuarios/507f1f77bcf86cd799439011
```

4.3.4 Actualizar un Usuario

```
curl -X PUT http://localhost:3000/usuarios/507f1f77bcf86cd799439011 \
  -H "Content-Type: application/json" \
  -d '{
    "telefono": "555-5678"
  }'
```

4.3.5 Eliminar un Usuario

```
curl -X DELETE http://localhost:3000/usuarios/507f1f77bcf86cd799439011
```

5 Implementación Técnica y CI/CD

5.1 Dockerfile

5.1.1 Código del Dockerfile

```
FROM node:20
WORKDIR /app
COPY package*.json ./
RUN npm install
COPY . .
EXPOSE 3000
CMD ["npm","start"]
```

5.1.2 Explicación Detallada de Cada Instrucción

- `FROM node:20`: Define la imagen base. Utiliza Node.js versión 20 como sistema operativo base con Node.js y npm preinstalados. Este es el punto de partida para la imagen.
- `WORKDIR /app`: Establece el directorio de trabajo dentro del contenedor. Todos los comandos subsiguientes se ejecutan en este contexto. Si el directorio no existe, Docker lo crea automáticamente.
- `COPY package*.json ./`: Copia los archivos `package.json` y `package-lock.json` (si existe) desde el anfitrión al contenedor. El asterisco (*) actúa como comodín. Esta se hace **antes** de copiar el código fuente para aprovechar el caché de Docker.
- `RUN npm install`: Instala las dependencias del proyecto listadas en `package.json`. Esto crea el directorio `node_modules` dentro del contenedor.
- `COPY . .`: Copia el resto del código fuente del proyecto (excepto lo listado en `.Dockerignore`) al directorio de trabajo del contenedor.
- `EXPOSE 3000`: Documenta que el contenedor escucha en el puerto 3000. Esta instrucción es informativa y no publica automáticamente el puerto; es necesario usar `-p` en `docker run` o mapear puertos en Docker Compose.
- `CMD ["npm","start"]`: Define el comando predeterminado cuando se inicia el contenedor. Ejecuta `npm start`, que a su vez ejecuta `node main.js` según se define en `package.json`.

5.1.3 Archivo .Dockerignore

```
node_modules
.env
.git
npm-debug.log
```

Este archivo especifica qué archivos y directorios **no** deben incluirse en la imagen Docker:

- `node_modules`**:** Se reinstalará durante el `RUN npm install`, por lo que no es necesario copiarlo.
- `.env`**:** Archivo con variables de entorno sensibles que no deben estar en la imagen.
- `.git`**:** Historial de Git innecesario en la imagen de producción.
- `npm-debug.log`**:** Archivos de depuración que no aportan valor.

5.2 docker-compose.yml

5.2.1 Código de Docker Compose

```
version: '3.8'

services:
  mongo:
    image: mongo:latest
    container_name: mongodb_gestion
    ports:
      - "27017:27017"
    volumes:
      - mongo_data:/data/db

  api:
    image: johny050824/api_docker_compose:latest
    build: .
    container_name: api_docker_compose
    ports:
      - "3000:3000"
    depends_on:
      - mongo

volumes:
  mongo_data:
```

5.2.2 Explicación de la Configuración

Sección `version` `version: '3.8'` especifica la versión del formato Docker Compose. La versión 3.8 es compatible con Docker Engine 19.03.0+.

Servicio `mongo`

- `image: mongo:latest`**:** Usa la imagen oficial más reciente de MongoDB desde Docker Hub.
- `container_name: mongodb_gestion`**:** Nombre amigable para el contenedor.
- `ports`**:** Mapea el puerto 27017 del contenedor al puerto 27017 del anfitrión, permitiendo acceso local a la BD.
- `volumes`**:** Monta el volumen nombrado `mongo_data` en `/data/db`, asegurando la persistencia de datos incluso si el contenedor se elimina.

Servicio `api`

- `image:` Referencia a la imagen en Docker Hub. Se actualiza cuando se ejecuta `docker-compose pull`.
- `build: .:` Indica que Docker Compose debe construir la imagen usando el Dockerfile en el directorio actual si la imagen local no existe.
- `container_name:` Nombre identificable para el contenedor de la API.
- `ports:` Mapea puerto 3000 (interno) a puerto 3000 (anfitrión) para acceso a la API.
- `depends_on: [mongo]:` Especifica que el servicio `api` depende del servicio `mongo`. Docker Compose inicia MongoDB primero, luego la API. **Nota:** Esto solo controla el orden de inicio, no garantiza que MongoDB esté completamente listo. La aplicación debe reintentar la conexión si es necesario.

Sección `volumes` `mongo_data:` Define un volumen nombrado que persiste los datos de MongoDB incluso cuando se detienen los contenedores. Los volúmenes se almacenan en `/var/lib/docker/volumes/` en el anfitrión.

5.3 Flujo de Trabajo de GitHub Actions

5.3.1 Código Completo del Workflow

```
name: Construir y Subir Imagen Docker de la API

on:
  push:
    branches:
      - main
      - master
    paths:
      - 'package.json'
      - 'main.js'
      - 'Dockerfile'
      - '.github/workflows/docker-push.yml'
  workflow_dispatch:

env:
  DOCKER_USERNAME: johny050824
  DOCKER_IMAGE_NAME: johny050824/api_docker_compose
  DOCKER_TAG: latest

jobs:
  build-and-push:
    runs-on: ubuntu-latest
    environment: docker

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

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

      - name: Login a Docker Hub
        uses: docker/login-action@v3
        with:
          username: ${{ env.DOCKER_USERNAME }}
          password: ${{ secrets.DOCKER_PASSWORD }}

      - name: Construir y Subir Imagen Docker
```

```
      uses: docker/build-push-action@v5
      with:
        context: .
        file: ./Dockerfile
        push: true
        tags: ${{env.DOCKER_IMAGE_NAME}}:${{ env.DOCKER_TAG }}
        cache-from: type=registry,ref=${{ env.DOCKER_IMAGE_NAME }}:buildcache
        cache-to: type=inline

    - name: Mostrar informacion de la imagen
      run: |
        echo "Imagen construida y subida exitosamente:"
        echo "  - Imagen: ${{ env.DOCKER_IMAGE_NAME }}:${{ env.DOCKER_TAG }}"
        echo "  - Docker Hub: https://hub.docker.com/r/johny050824/
            api_docker_compose/tags"

    - name: Notificacion de Exito en Telegram
      uses: appleboy/telegram-action@master
      with:
        to: ${{ secrets.TELEGRAM_ID_CHAT }}
        token: ${{ secrets.TELEGRAM_API_TOKEN }}
        message: |
          [Exito] Imagen Docker Subida
          Repositorio: ${{ github.repository }}
          Rama: ${{ github.ref_name }}
          Autor: ${{ github.actor }}
          Imagen: ${{ env.DOCKER_IMAGE_NAME}}:${{ env.DOCKER_TAG }}
        format: markdown
        disable_web_page_preview: true
```

5.3.2 Descripción del Proceso CI/CD

El flujo de trabajo implementa un pipeline automatizado con los siguientes componentes:

Triggers (`on`)

- **`push` a `main` o `master`:** El workflow se activa solo cuando se realizan cambios específicos en archivos críticos (`package.json`, `main.js`, `Dockerfile`, el workflow mismo). Esto evita ejecuciones innecesarias.
- **`workflow_dispatch`:** Permite ejecutar el workflow manualmente desde la interfaz de GitHub sin necesidad de un commit.

Variables de Entorno (`env`)

- `DOCKER_USERNAME`: Usuario de Docker Hub.
- `DOCKER_IMAGE_NAME`: Nombre completo de la imagen en Docker Hub.
- `DOCKER_TAG`: Etiqueta de la imagen (latest, v1.0, etc.).

Pasos del Job `build-and-push`

1. **Checkout código (Paso 1):**
   - **Acción:** `actions/checkout@v4`
   - **Función:** Descarga el código del repositorio al ejecutor de GitHub Actions.
   - **Resultado:** El directorio de trabajo contiene todo el código fuente, Dockerfile y configuraciones.
2. **Configurar Docker Buildx (Paso 2):**

- **Acción:** `docker/setup-buildx-action@v3`
- **Función:** Inicializa Docker Buildx, una herramienta avanzada para construir imágenes Docker con soporte para múltiples arquitecturas (amd64, arm64, etc.).
- **Ventaja:** Permite construcciones más rápidas y cachés más eficientes.

3. **Login a Docker Hub (Paso 3):**
   - **Acción:** `docker/login-action@v3`
   - **Función:** Autentica contra Docker Hub usando el usuario y contraseña/token almacenados en GitHub Secrets.
   - **Seguridad:** Las credenciales nunca se exponenen en los logs del workflow.
   - **Parámetros:**
     - `username`: `${{ env.DOCKER_USERNAME }}`
     - `password`: `${{ secrets.DOCKER_PASSWORD }}` (secreto cifrado en GitHub)
4. **Construir y Subir Imagen Docker (Paso 4):**
   - **Acción:** `docker/build-push-action@v5`
   - **Función:** Construye la imagen Docker usando el Dockerfile y la sube inmediatamente a Docker Hub.
   - **Parámetros clave:**
     - `context`: Raíz del repositorio (donde reside el Dockerfile).
     - `file`: Ruta del Dockerfile.
     - `push: true`: Especifica que la imagen debe subirse a Docker Hub.
     - `tags`: Etiqueta de la imagen (`johny050824/api_docker_compose:latest`).
     - `cache-from` y `cache-to`: Optimizan construcciones futuras reutilizando capas.
5. **Mostrar Información de la Imagen (Paso 5):**
   - **Acción:** `run` (script de shell)
   - **Función:** Imprime información sobre la imagen construida en los logs del workflow para verificación.
   - **Salida:** URLs directas a la imagen en Docker Hub para acceso rápido.
6. **Notificación en Telegram (Paso 6):**
   - **Acción:** `appleboy/telegram-action@master`
   - **Función:** Envía un mensaje de Telegram notificando el éxito del despliegue.
   - **Información incluida:**
     - Estado: [Éxito]
     - Nombre del repositorio
     - Rama desde la que se disparó
     - Usuario que realizó el commit
     - Imagen completa en Docker Hub
   - **Parámetros:**
     - `to`: ID del chat de Telegram (secreto).
     - `token`: Token de la API del bot de Telegram (secreto).
     - `format: markdown`: Formatea el mensaje con soporte Markdown.

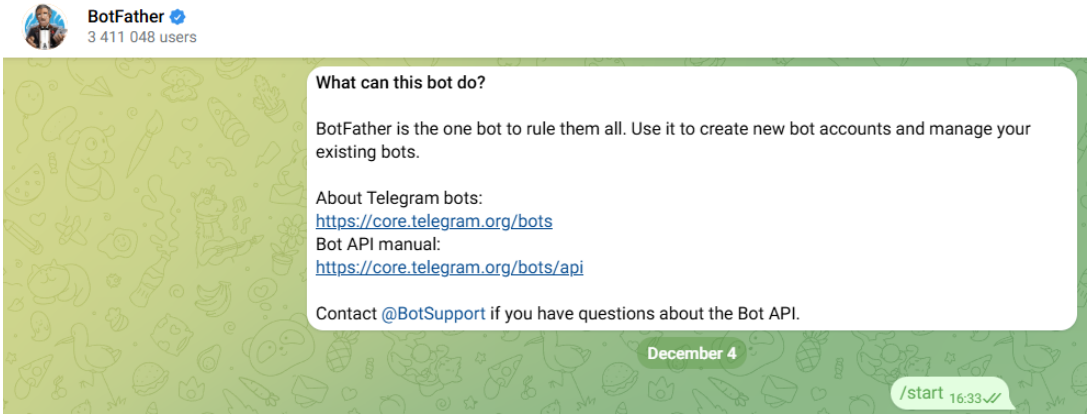
6 Configuración de Notificaciones

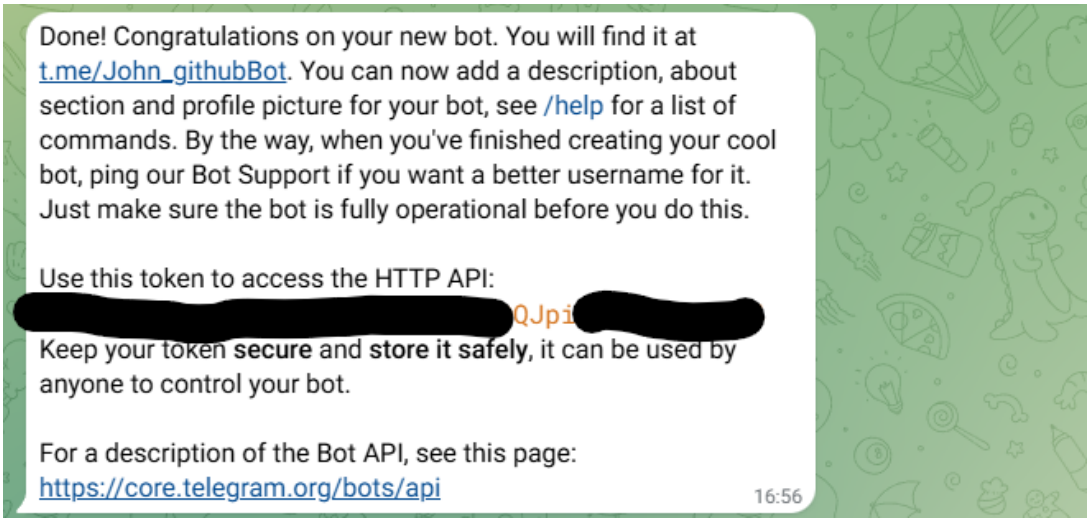
6.1 Integración con Telegram

Telegram proporciona un mecanismo confiable y en tiempo real para notificar al equipo de desarrollo sobre el estado de los despliegues. Los pasos a continuación detallan cómo configurar un bot de Telegram y vincularlo con GitHub Actions.

6.1.1 Paso 1: Crear el Bot en BotFather

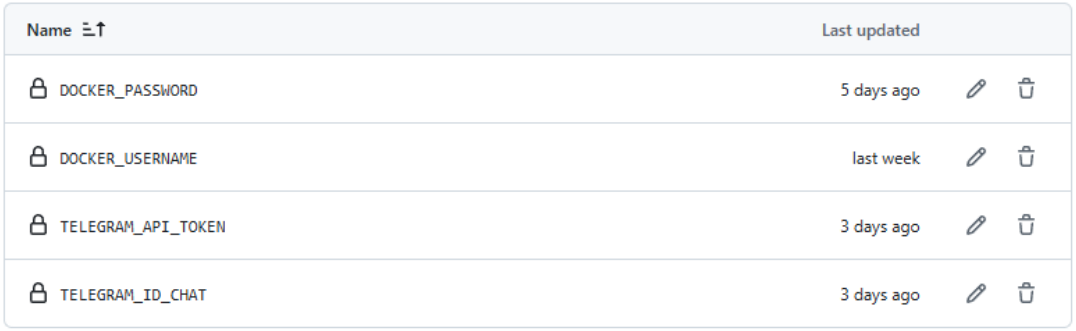
BotFather es el bot oficial de Telegram para crear y gestionar otros bots. Sigue estos pasos:



1. Abre Telegram y busca el usuario `@BotFather`.
2. Inicia una conversación escribiendo `/start`.
3. Escribe el comando `/newbot` para crear un nuevo bot.
4. BotFather solicitará un nombre para el bot. Ingresa algo como `API Docker Deploy Bot`.
5. Luego solicitará un nombre de usuario único. Ingresa algo como `api_docker_bot_tu_nombre`.
6. BotFather responderá con un mensaje que contiene el **TOKEN del bot**. Ejemplo:
7. 


```
Done! Congratulations on your new bot. You will find it at
https://t.me/api_docker_bot. You can now add a description, about section and
profile picture for your bot, see /help for a list of commands.
```

8. **Copia y guarda el TOKEN en un lugar seguro. COMO LOS SECRECTS** Tendrá un formato como: `123456:ABC-DEF1234ghIkl-zyx57W2v1u123ew11`



9.

6.1.2 Paso 2: Obtener el ID del Chat

Para que el bot envíe mensajes al equipo, necesita el ID del chat. Sigue estos pasos:

1. En Telegram, busca tu bot recién creado (el nombre de usuario que especificaste en BotFather).
2. Abre el chat con el bot.
3. **Envía cualquier mensaje** al bot, como por ejemplo `/hola` o simplemente `Hola`.
4. **Importante:** El bot solo procesa mensajes después de recibir al menos uno del usuario.
5. Abre tu navegador y accede a la siguiente URL, reemplazando `<TOKEN>` con el token obtenido en el Paso 1:

   `https://api.telegram.org/bot<TOKEN>/getUpdates`

6. Se mostrará una respuesta JSON. Busca el campo `"chat":{"id": XXXXX` dentro del contenido. El número **XXXXX** es tu **ID del chat**.
7. Ejemplo de respuesta:

```
{
  "ok": true,
  "result": [
    {
      "update_id": 123456789,
      "message": {
        "message_id": 1,
        "date": 1607000000,
        "chat": {
          "id": 987654321,
          "type": "private"
        },
        "text": "Hola",
        "from": {
          "id": 987654321,
          "is_bot": false,
          "first_name": "Tu Nombre"
        }
      }
    }
  ]
}
```

8. En este ejemplo, el ID del chat es **987654321**.

6.1.3 Paso 3: Almacenar Credenciales en GitHub Secrets

Los tokens y IDs sensibles deben almacenarse como **GitHub Secrets** para que el workflow pueda acceder a ellos de forma segura:

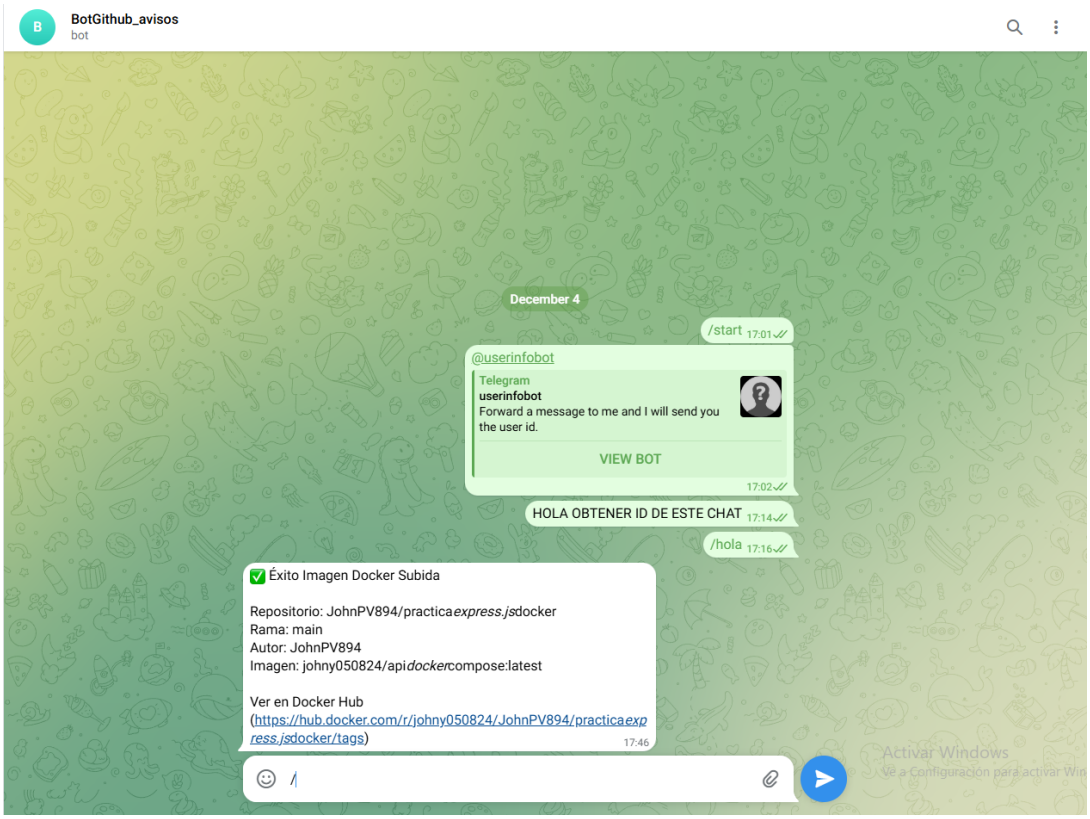
1. Dirígete al repositorio en GitHub: https://github.com/JohnPV894/practica_express.js_docker
2. Haz clic en la pestaña **Settings** (Configuración).
3. En el menú lateral izquierdo, selecciona **Secrets and variables > Actions**.
4. Haz clic en el botón **New repository secret**.
5. Crea los siguientes secrets:
   - **Nombre:** `DOCKER_PASSWORD`
     - **Valor:** Tu token de acceso personal de Docker Hub (puedes generarlo en https://hub.docker.com/settings/security).
     - Este secret se usa en el paso de login a Docker Hub del workflow.
   - **Nombre:** `TELEGRAM_API_TOKEN`
     - **Valor:** El token del bot obtenido en el Paso 1 (ej: `123456:ABC-DEF1234ghIkl-zyx57W2v1u123ew11`).
     - Este secret se usa para la acción de notificación de Telegram.
   - **Nombre:** `TELEGRAM_ID_CHAT`
     - **Valor:** El ID del chat obtenido en el Paso 2 (ej: `987654321`).
     - Este secret especifica a qué chat de Telegram enviar las notificaciones.
6. Para cada secret, ingresa el nombre, el valor, y haz clic en **Add secret**.
7. Los secrets aparecerán en la lista con una descripción de cuándo fueron actualizados.

6.1.4 Verificación

Una vez configurado todo:

1. Realiza un `git push` a la rama `main` con cambios en `main.js`, `Dockerfile`, o `package.json`.
2. Dirígete a la pestaña **Actions** en GitHub.
3. Observa el workflow ejecutándose. Si todo es correcto, debe completarse con éxito en pocos minutos.
4. Verifica tu chat de Telegram. Deberías recibir un mensaje como:

5. **[Éxito] Imagen Docker Subida**
Repositorio: JohnPV894/practica_express.js_docker
Rama: main
Autor: tu_usuario
Imagen: johny050824/api_docker_compose:latest



7 Conclusión

Este documento ha presentado una arquitectura completa de DevOps que integra:

- **Contenerización:** Encapsulación de la aplicación en Docker para portabilidad.
- **Orquestación:** Gestión de múltiples servicios con Docker Compose.
- **Automatización:** Pipeline CI/CD basado en GitHub Actions que construye y despliega automáticamente.
- **Monitoreo:** Notificaciones en tiempo real mediante Telegram.

Esta solución proporciona un flujo de trabajo eficiente, seguro y auditable para el desarrollo y despliegue de aplicaciones modernas.