***TRABAJO PRÁCTICO***

***Materia***

DEVOPS

**Integrantes del Grupo: Bernardo Napoleon Luscher**

**Turno: Noche**

**Docente: Rodrigo Ariel Herera**

**Fecha de entrega: 22/11/2023**

Objetivo

El objetivo de este trabajo práctico es que los alumnos apliquen los conceptos y herramientas aprendidas en el curso de DevOps para crear un proceso de integración continua, entrega continua y despliegue automatizado[opcional] de una aplicación utilizando herramientas y prácticas de DevOps.

**Parte 1: Creación de la Aplicación**

Crea una aplicación web simple en el lenguaje de programación de tu elección (por ejemplo, Node.js, Python, Go, etc). La aplicación debe tener al menos una página que muestre "Hola Mundo" o información similar. Asegúrate de que la aplicación tenga un sistema de pruebas unitarias (1 test unitario al menos). Armar Dockerfile para el lenguaje de programación elegido. Armar docker-compose y que levante el Dockerfile previo.

**Parte 2: Configuración del Repositorio y CI:**

Crea un repositorio en GitHub (de preferencia GitHub) para la aplicación. Genera un archivo de configuración -dependiendo del CI que hayan elegido- que realice lo siguiente: Instale las dependencias de la aplicación. Ejecuta las pruebas unitarias. Ejecutar un build (caso que sea un lenguaje que no se buildee, puede ser un lint en vez de build) Correr el build del Dockerfile y subir la imagen a Docker Hub.

Desarrollo del Trabajo.

# Parte 1

Se eligió para el desarrollo del a aplicación web, Spring Boot, un popular framework de desarrollo de aplicaciones en Java que simplifica la creación de aplicaciones Java empresariales.

Algunos beneficios de Spring Boot en general vamos a enumerar:

1. Facilita la configuración: Spring Boot simplifica la configuración de aplicaciones al proporcionar valores predeterminados sensatos y al permitir la configuración mediante archivos de propiedades o YAML.
2. Desarrollo rápido: Spring Boot incluye un conjunto de características y bibliotecas preconfiguradas que permiten a los desarrolladores crear aplicaciones de forma más rápida y eficiente, reduciendo la necesidad de escribir código repetitivo.
3. Integración sencilla: Spring Boot se integra fácilmente con otras tecnologías y frameworks de Spring, lo que facilita la creación de aplicaciones completas y coherentes.
4. Gestión de dependencias: Spring Boot utiliza el sistema de gestión de dependencias de Maven o Gradle, lo que simplifica la administración de las bibliotecas y dependencias de un proyecto.
5. Monitorización y métricas: Spring Boot incluye herramientas para la monitorización y la recopilación de métricas de aplicaciones, lo que facilita la supervisión y el mantenimiento de aplicaciones en producción.
6. Soporte para contenedores: Spring Boot es compatible con la creación de aplicaciones empaquetadas como contenedores Docker, lo que simplifica la implementación y escalabilidad en entornos de contenedorización.
7. Compatibilidad con múltiples fuentes de datos: Spring Boot permite conectar fácilmente aplicaciones a diversas fuentes de datos, como bases de datos SQL y NoSQL.

Adicionalmente adoptamos en nuestro start up a Spring Boot Initializer, una herramienta en línea que ayuda a los desarrolladores a crear proyectos de Spring Boot personalizados. Algunos de los beneficios de Spring Boot Initializer incluyen:

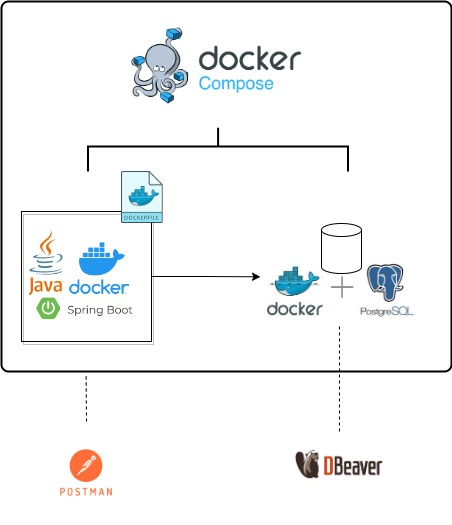
1. Inicio rápido: Spring Boot Initializer permite a los desarrolladores crear un proyecto de Spring Boot con la configuración inicial en cuestión de minutos, lo que acelera el proceso de desarrollo.
2. Personalización: Los desarrolladores pueden personalizar su proyecto seleccionando las dependencias y tecnologías específicas que desean incluir en su aplicación.
3. Plantillas predefinidas: Spring Boot Initializer ofrece una variedad de plantillas predefinidas para proyectos comunes, como aplicaciones web, aplicaciones de datos, aplicaciones de microservicios, lo que facilita la creación de proyectos específicos.
4. Integración con IDE: La herramienta se integra con varios entornos de desarrollo integrados (IDE) populares, como Eclipse y IntelliJ IDEA, lo que simplifica aún más el proceso de desarrollo.

Objetivo Parte1:

Vamos a desarrollar para este prototipo, una aplicación básica con 5 endpoints para operaciones CRUD de una tabla de empleado.

* Create
* Read all
* Read one
* Update
* Delete

Arquitectura de la implementación:



Pasos de ejecución:

1. Crear una aplicación Java utilizando Spring Boot, Spring Web, Spring Data e Hibernate.
   1. Tendrá es una clase Java que representa una entidad de empleado y se usa junto con una base de datos que utiliza Jakarta Persistence (anteriormente conocida como Java Persistence API o JPA) para el mapeo relacional de objetos (ORM).
   2. Tendrá una interfaz Java que define un repositorio para la entidad Usuario utilizando Spring Data JPA.
   3. Tendrá una clase Java Controller, que define una API RESTful para realizar operaciones CRUD (Crear, Leer, Actualizar, Eliminar) en la entidad Usuario utilizando Spring Boot y Spring Data JPA. Hay varios métodos de endpoints definidos en esta clase, cada uno asignado a un método HTTP específico (GET, POST, PUT, DELETE) y una ruta URL.
   4. El archivo application.properties es un archivo de configuración utilizado por una aplicación Spring Boot. Define propiedades que configuran la fuente de datos de la aplicación y Hibernate, el marco ORM (Object Relational Mapping) utilizado para la persistencia de la base de datos.

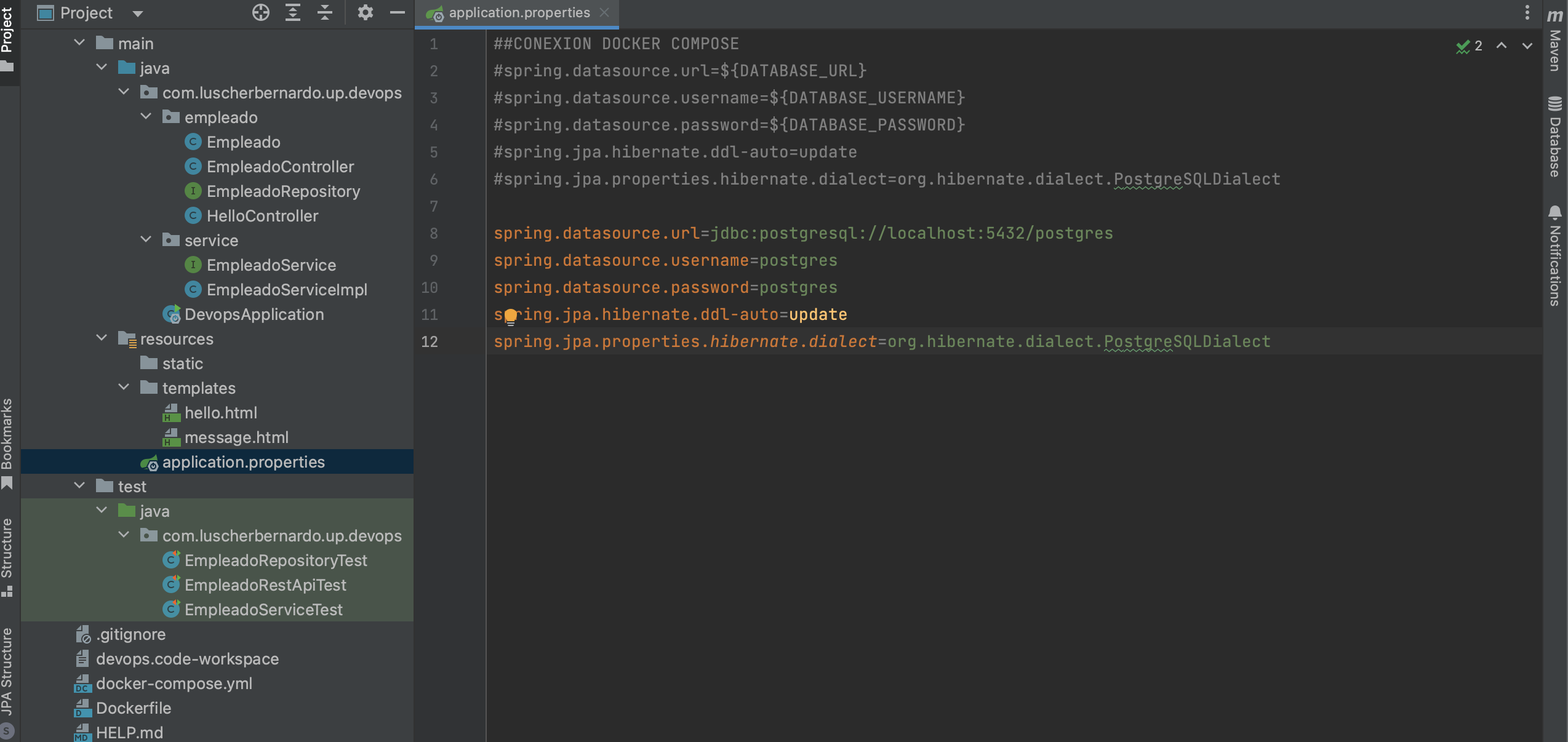
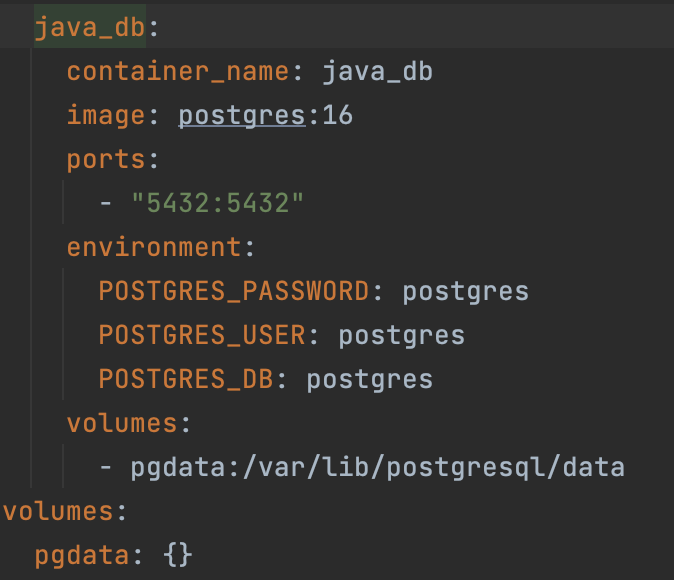


Fig.1 application.properties

1. Ejecutar la base de datos de Postgres en un contenedor usando Docker Compose y probarlo con DBeaver.
2. Para hacer eso, se debe crear un archivo llamado "docker-compose.yml" en el nivel raíz y llénarlo de la siguiente manera:



El resultado, el motor de base de dato funcionando en un contenedor fig.2:

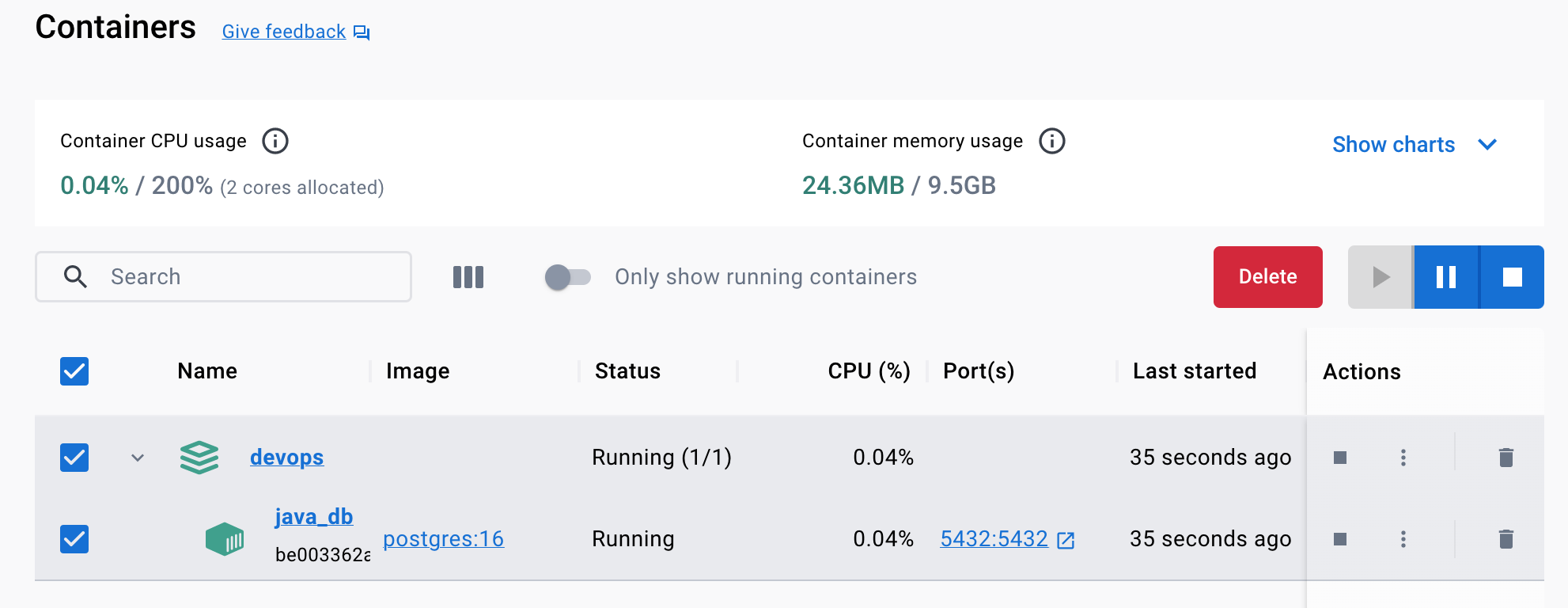
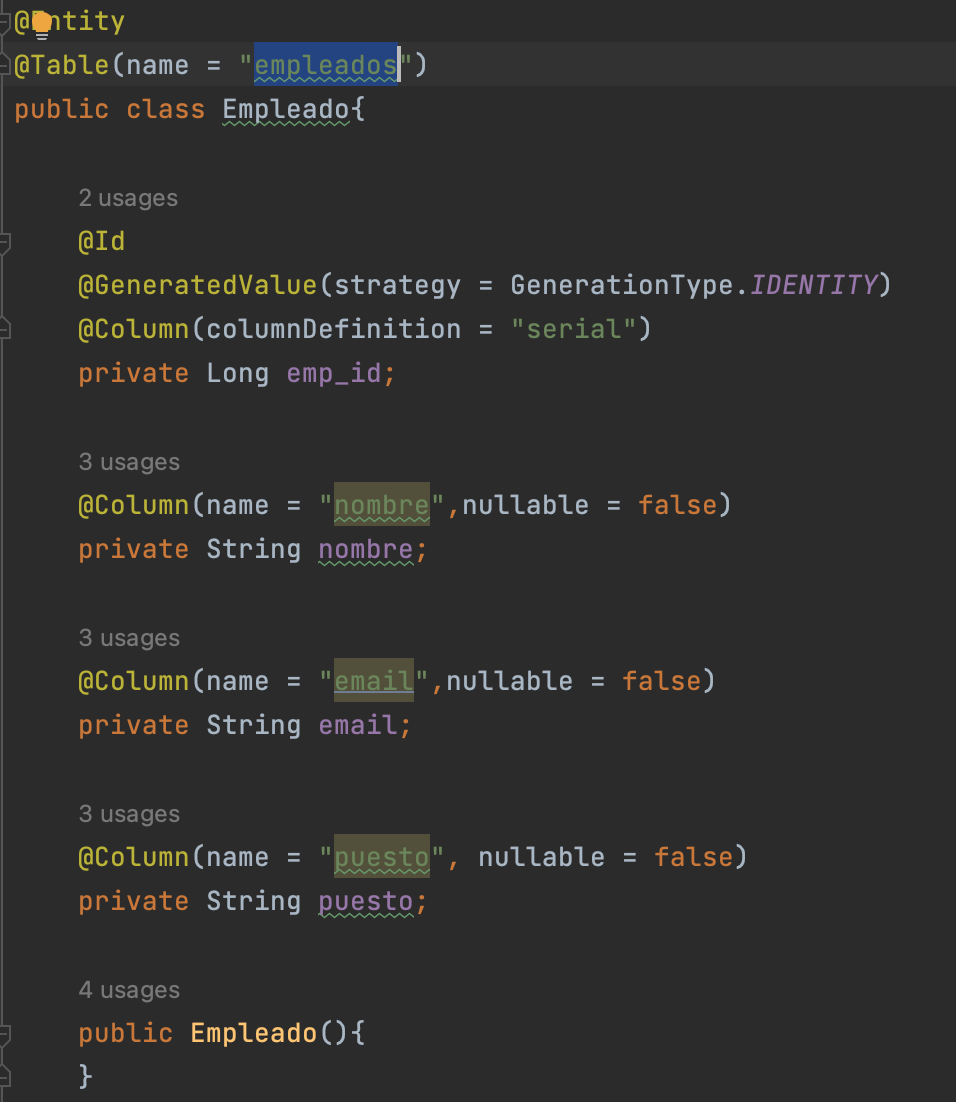


Fig.2 Postgresql dockerizado

1. Comprobamos la conexión con el cliente de base de dato DBeaver y la estructura de datos creadas para la aplicación.



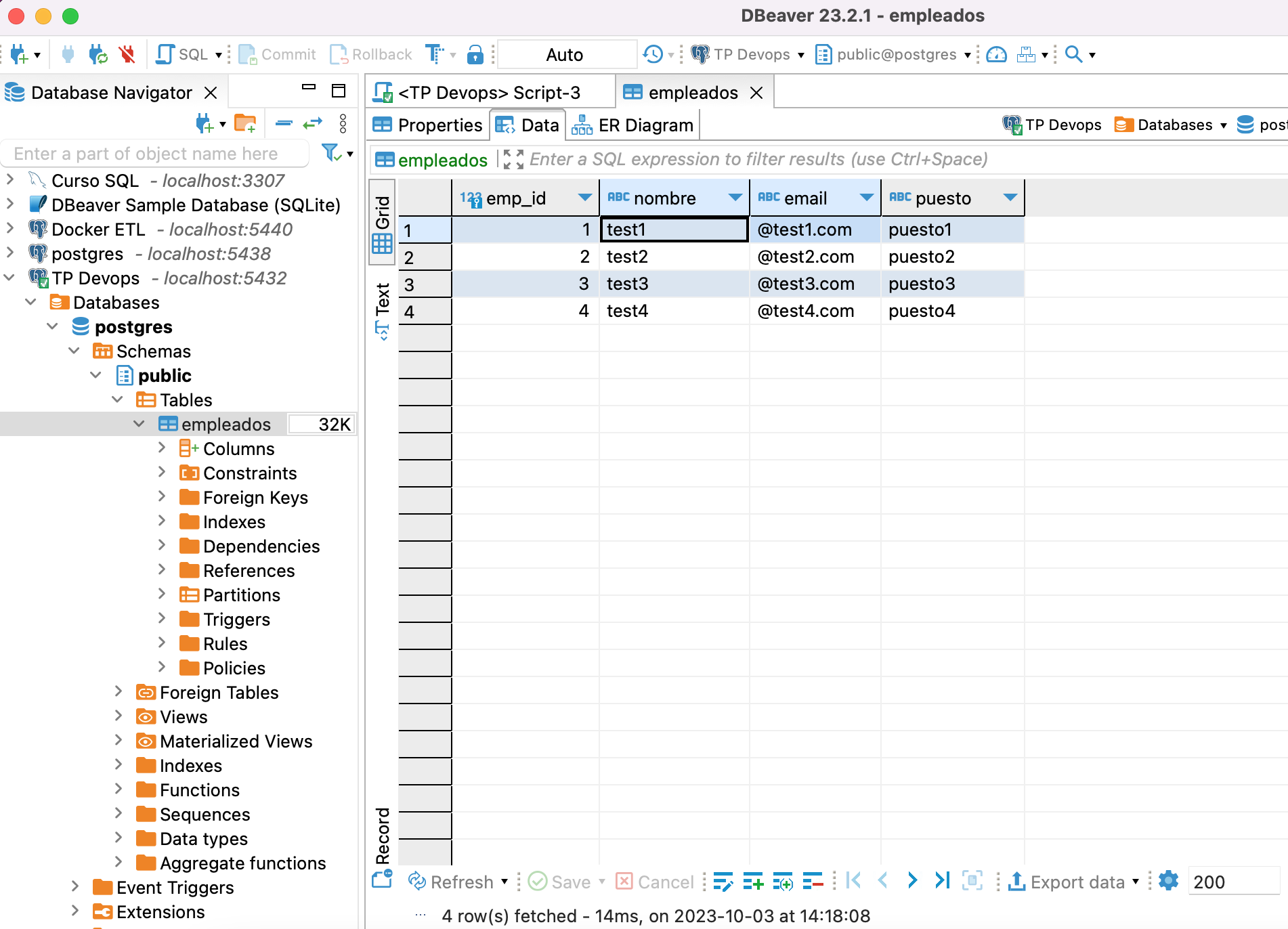
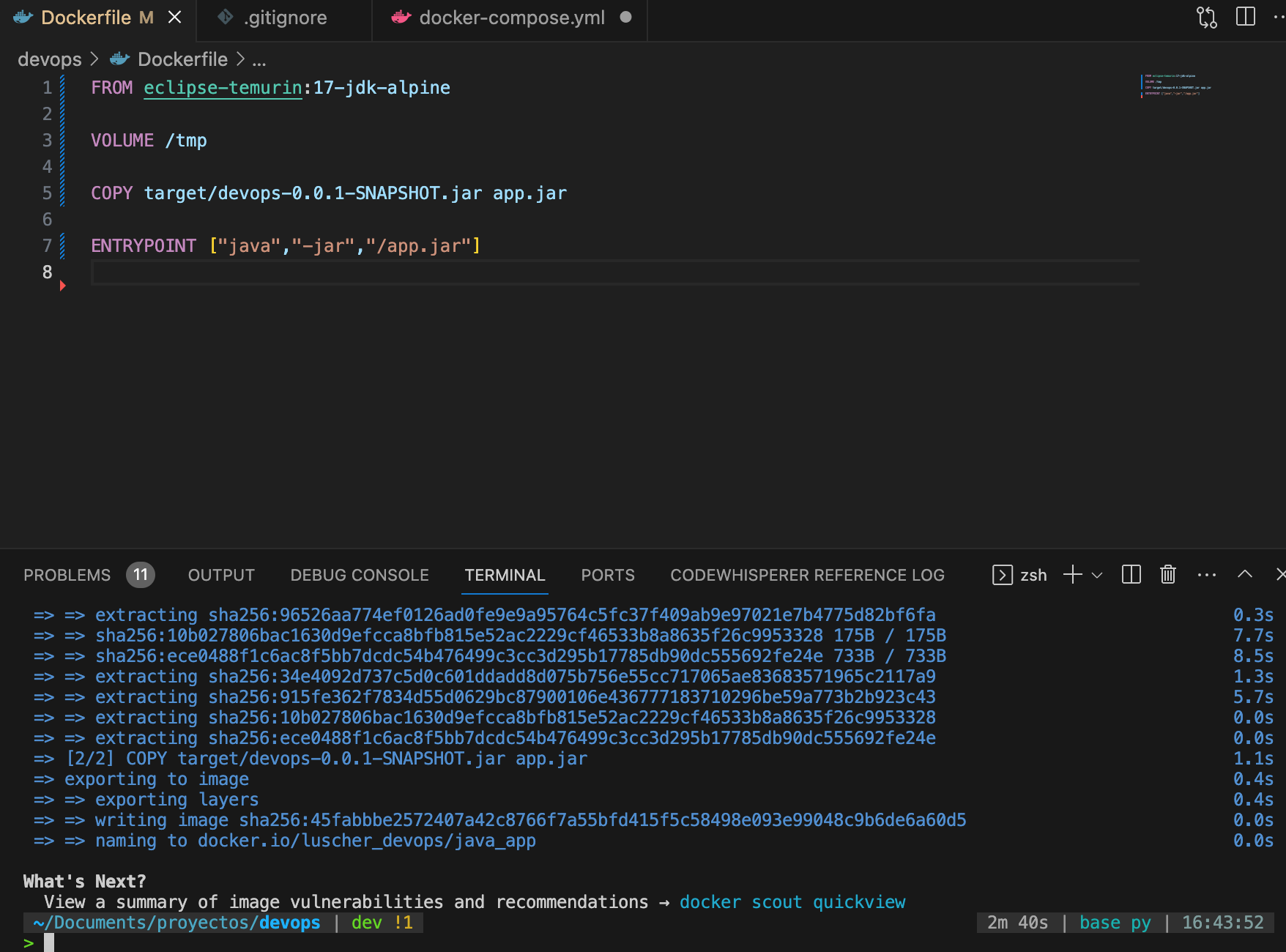
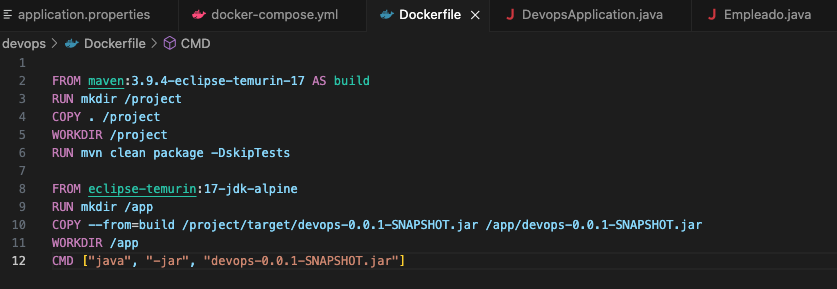


Fig.3 Base de dato Empleado Postgresql

1. Dockerizar la aplicación Java escribiendo un Dockerfile y un archivo docker-compose.yml para ejecutar la aplicación y la base de datos.
2. Dockerfile default



1. Dockerfile multi-stage build



El uso de Dockerfile con Multi-Stage Build ofrece los siguientes beneficios clave:

1. Imágenes más pequeñas: Permite crear imágenes finales más pequeñas, lo que ahorra espacio y tiempo de descarga.
2. Mayor seguridad: Reduce el riesgo de vulnerabilidades de seguridad al incluir solo archivos esenciales en la imagen final.
3. Optimización de caché: Aprovecha el caché de construcción para acelerar el proceso y reutilizar capas de caché.
4. Separación de entornos: Facilita la gestión de entornos de desarrollo y producción al dividir la construcción en etapas.
5. Mejora del flujo de trabajo: Simplifica el flujo de trabajo de desarrollo y la implementación de CI/CD.
6. Crear la aplicación Java, crear la imagen de Docker y ejecutar el contenedor usando Docker Compose, luego probarlo con Postman.

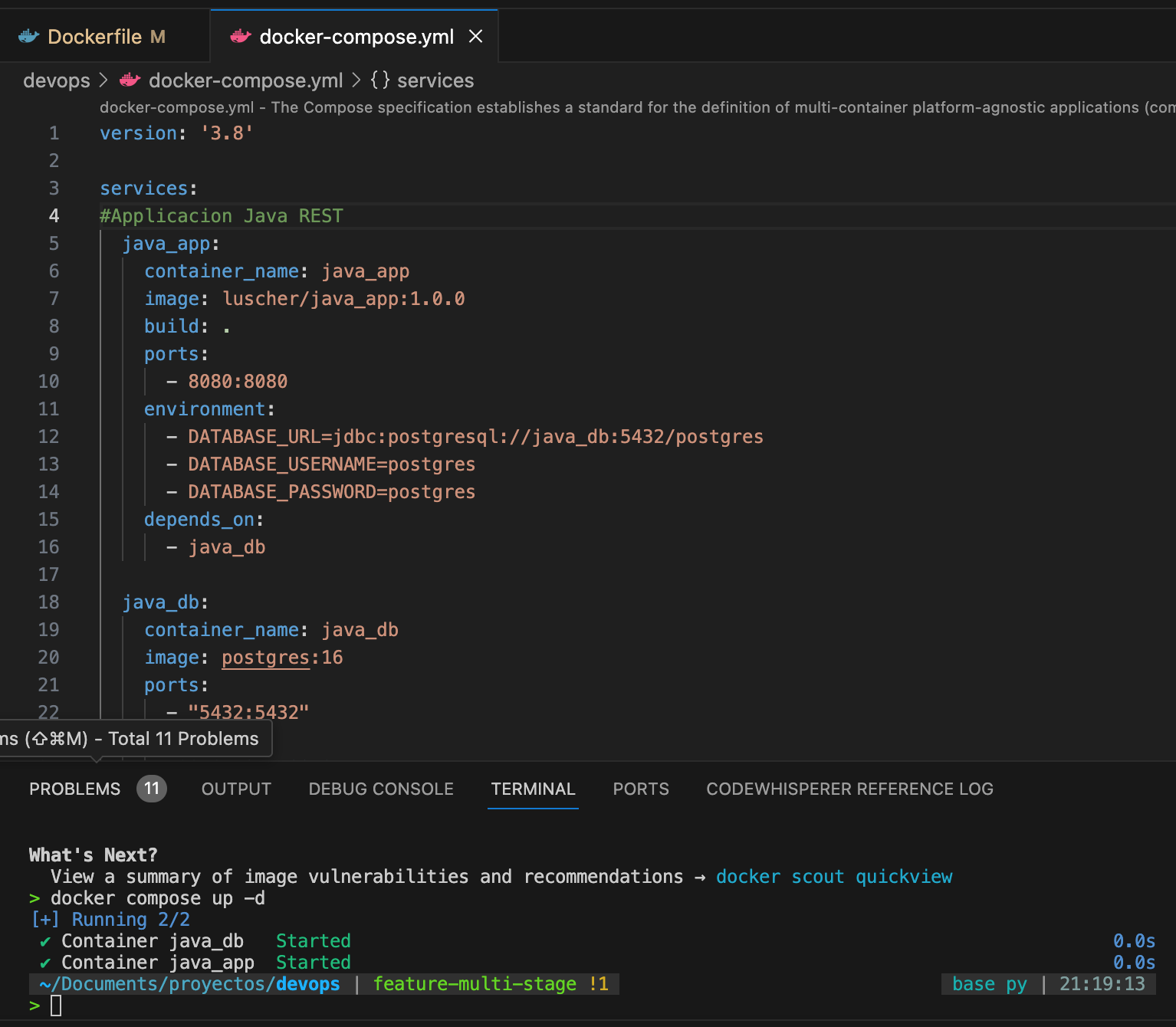


Fig.4 ejecutando el docker-compose

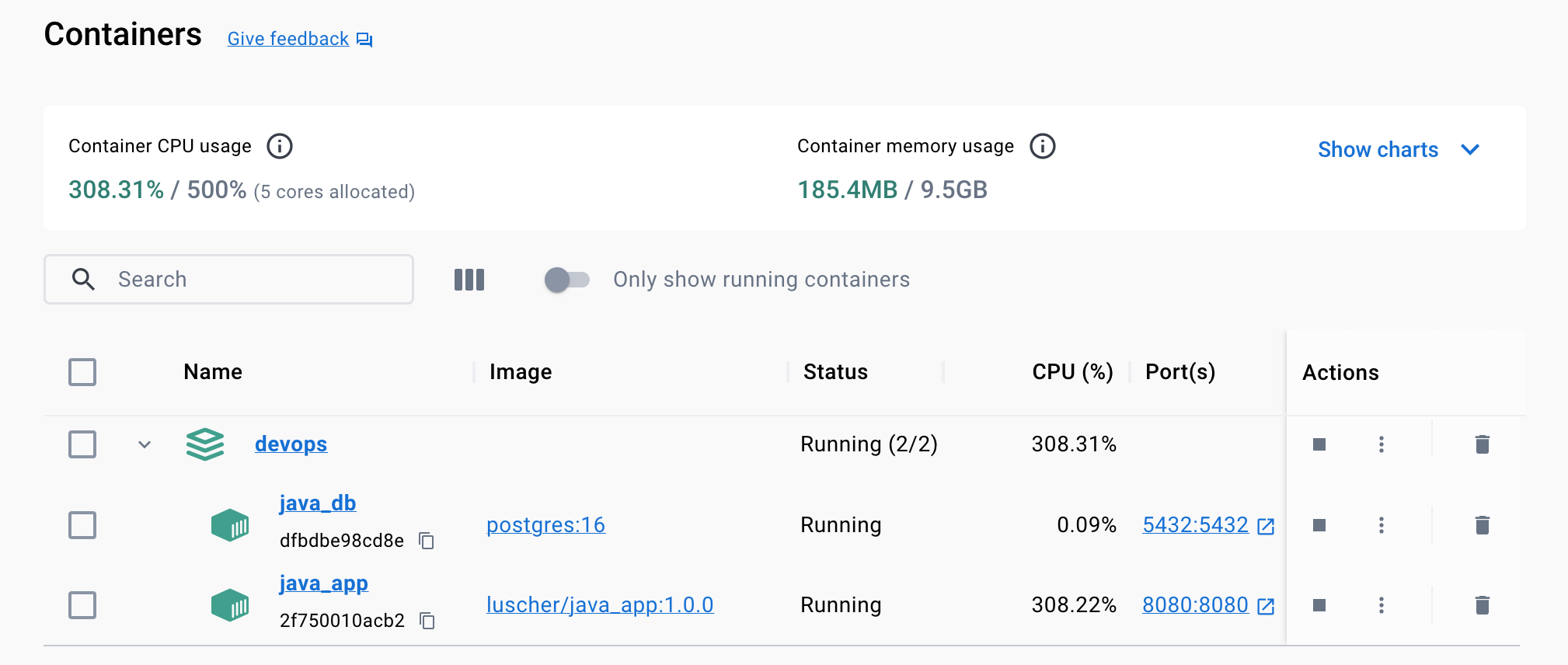
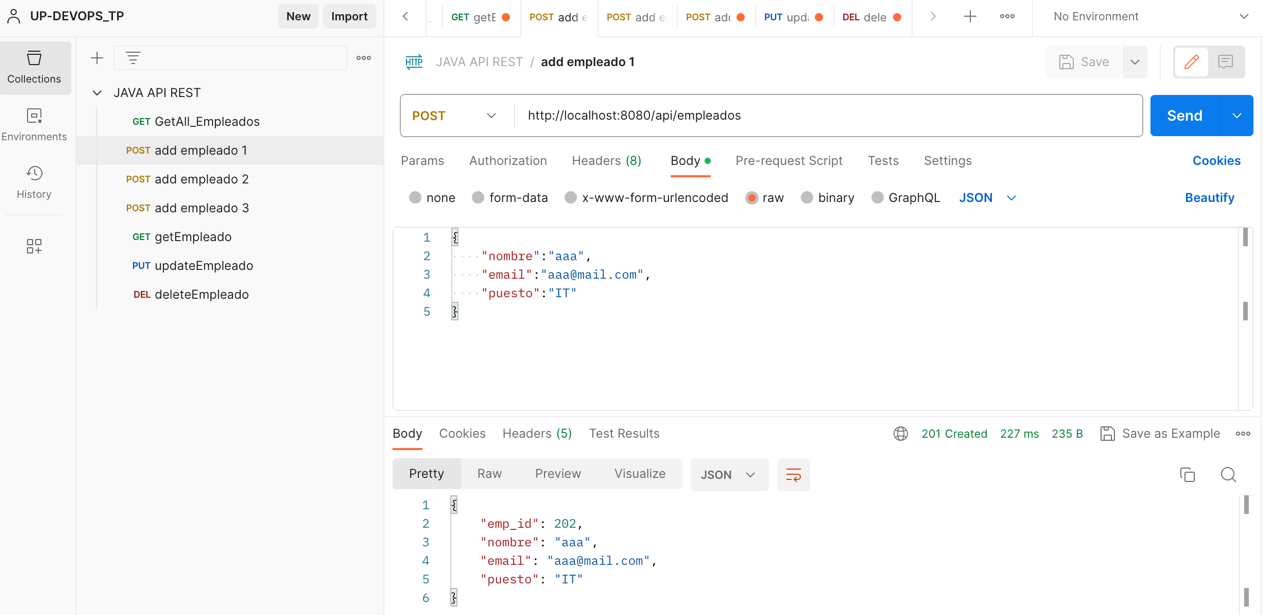


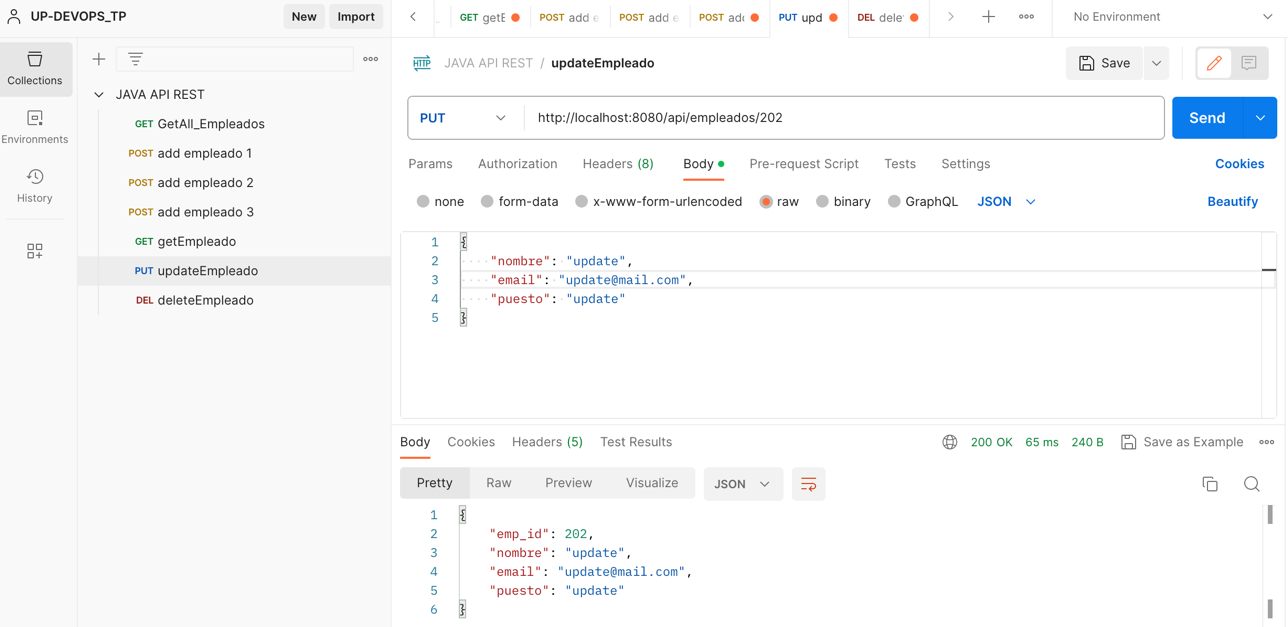
Fig.5 Docker Desktop, evidenciando los conteiner iniciados con el docker-compose

Test del API, algunos ejemplos:

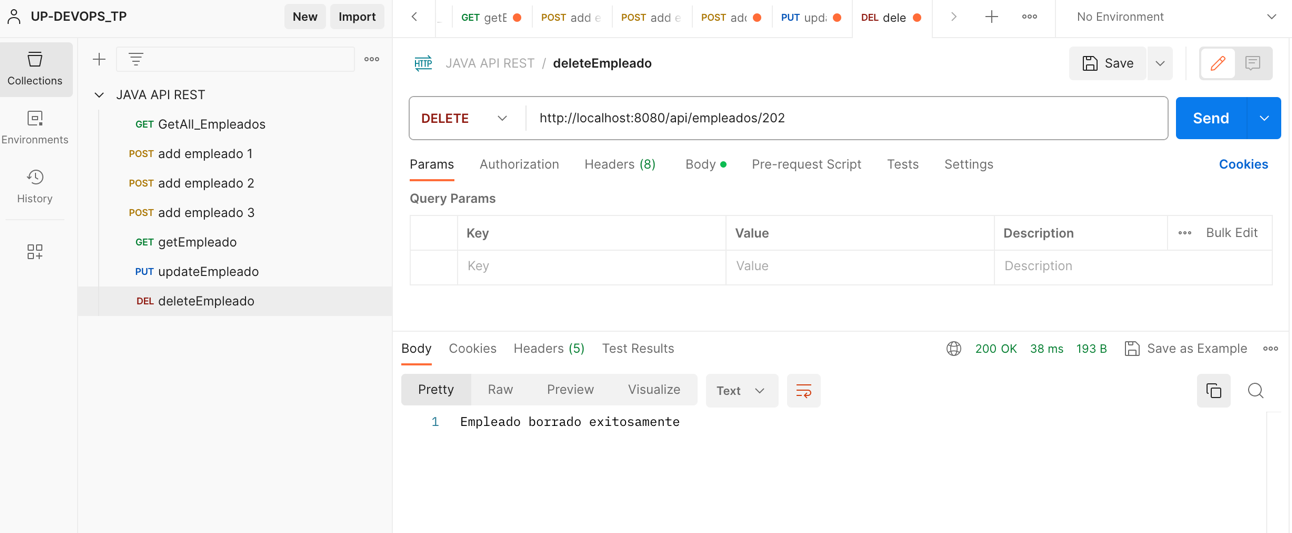
// alta de un empleado



//modificación de un empleado

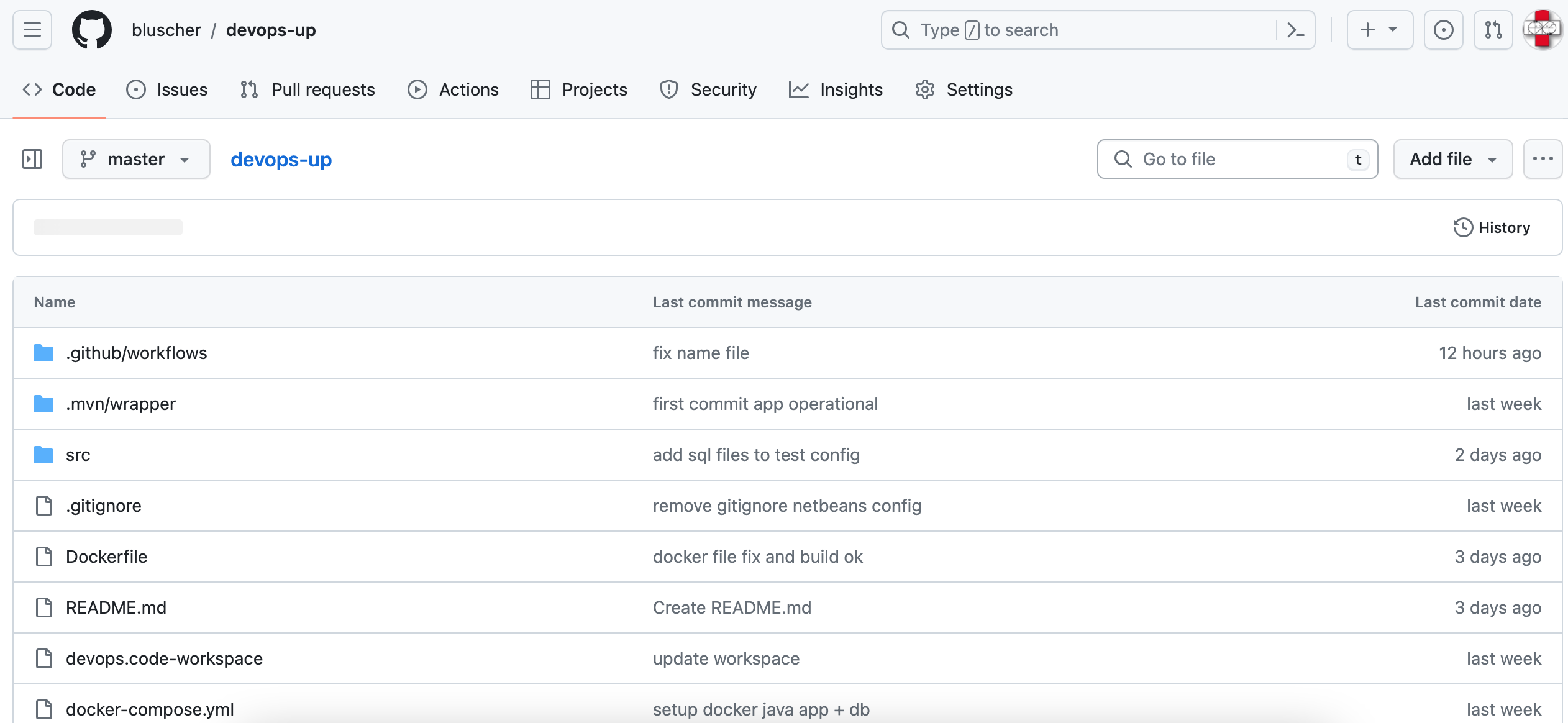


//borrado de un empleado

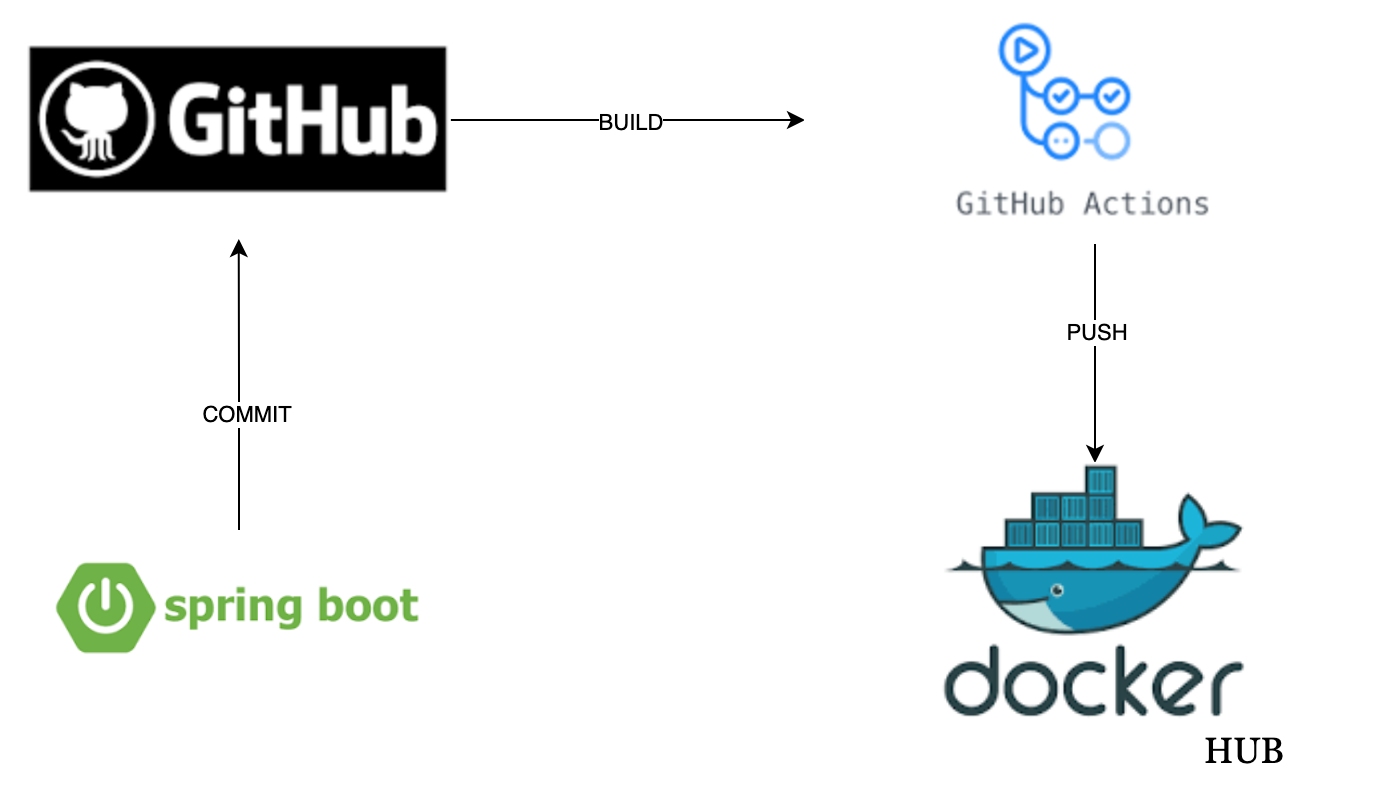


# Parte 2

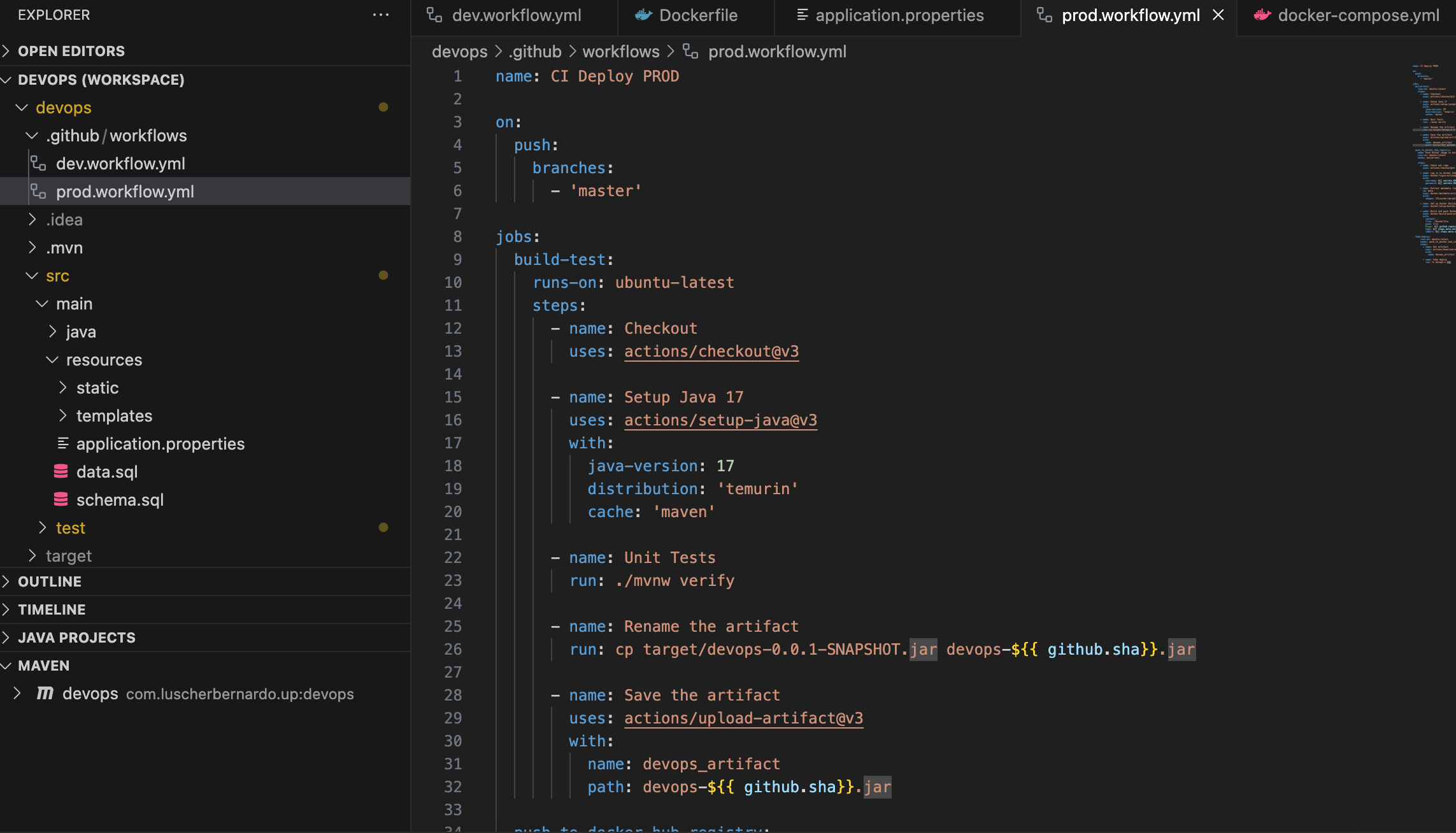
Se eligió para la segunda parte del trabajo un repositorio en GitHub. Se utilizaron varias ramas para las distintas funcionalidades que se fueron probando, siendo la “default”, la rama master.



Arquitectura del flujo CI



El archivo de configuración para el workflow productivo en github Actions es el siguiente: *prod.workflow.yml*



Este archivo se encarga de automatizar el proceso de implementación de una aplicación en un entorno de producción. A continuación, se explica cada parte del código:

1. #Nombre del flujo de trabajo:

"CI Deploy PROD". Esto es simplemente un nombre descriptivo para el flujo de trabajo.

2. #Eventos desencadenantes:

- ‘on’: Define cuándo debe ejecutarse el flujo de trabajo. En este caso, el flujo de trabajo se activará cuando haya un ‘push’ (confirmación) en la rama ‘master’ del repositorio.

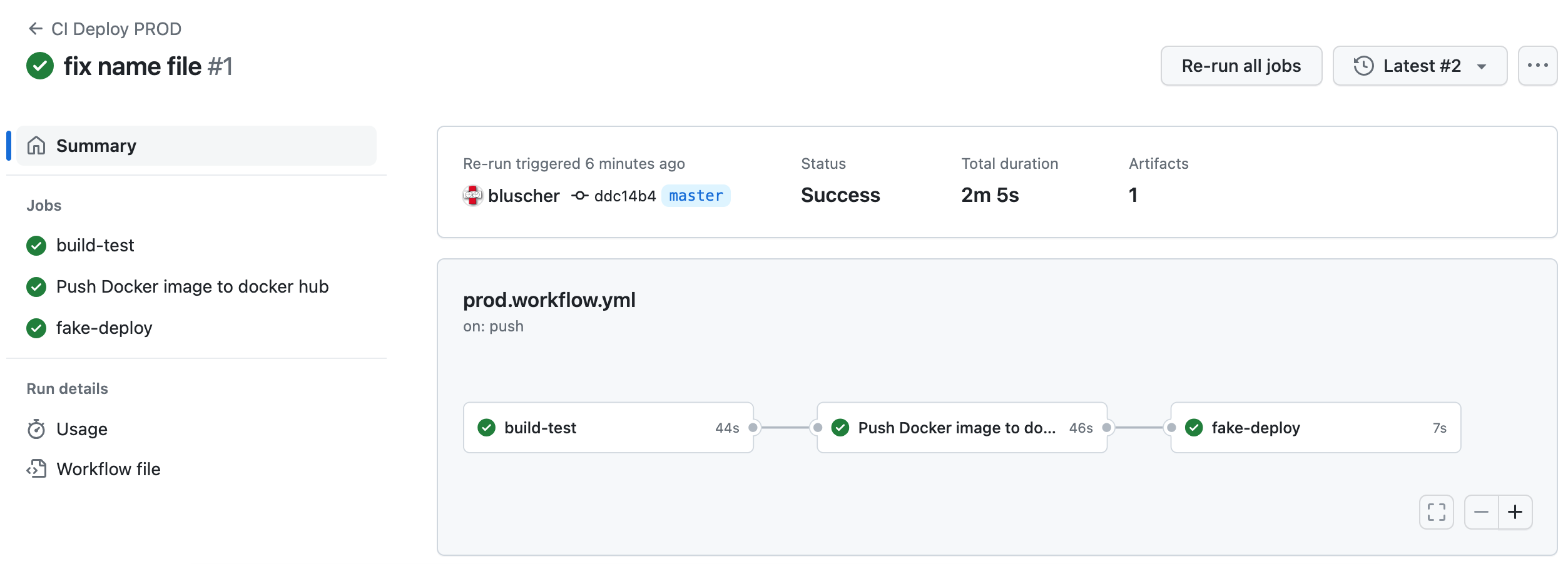


Fig.6 grafo de proceso - GitActions

3. #Trabajos (jobs):

**‘build-test’: Este es el primer trabajo que se ejecuta.**

- ‘runs-on’: Especifica que este trabajo se ejecutará en un entorno basado en Ubuntu.

- ‘steps’: Lista de pasos que se deben realizar en este trabajo.

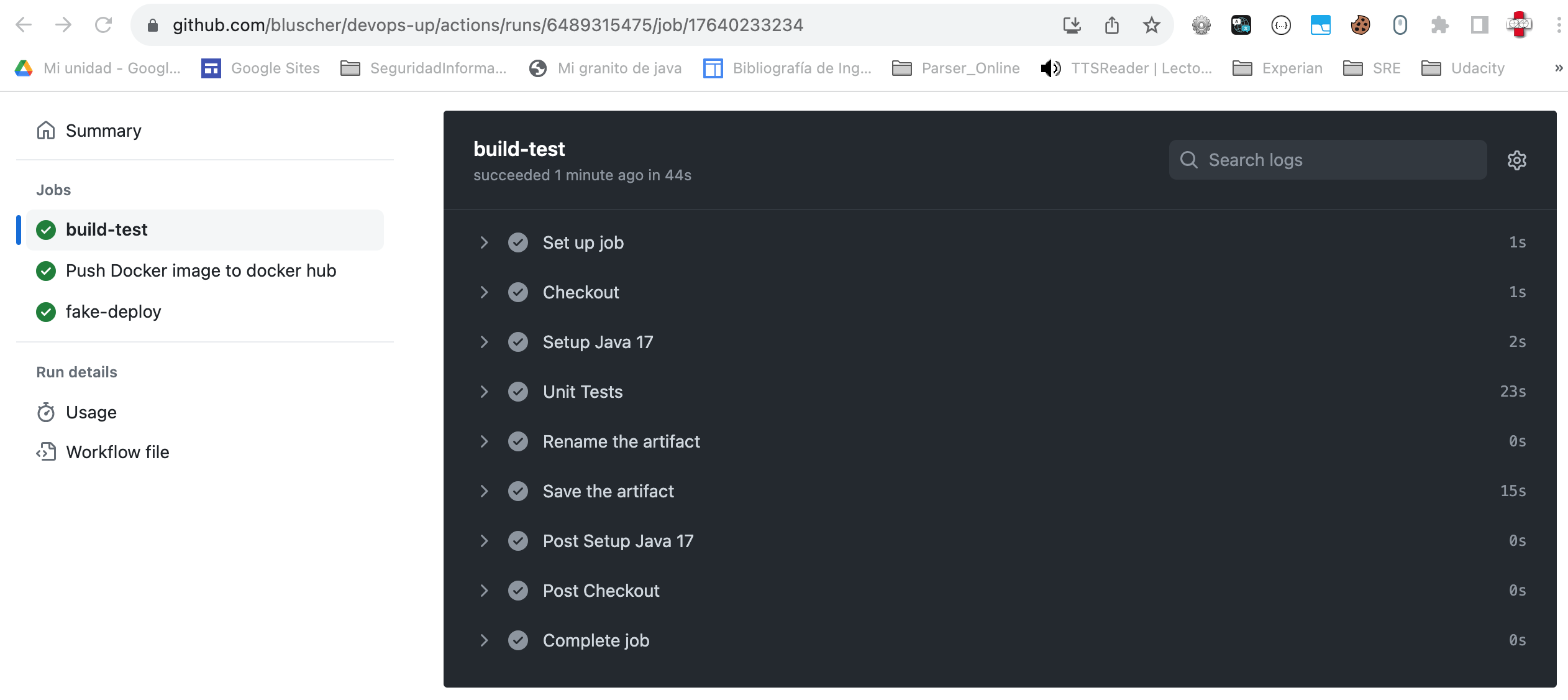
- ‘Checkout’: Se utiliza el paso ‘actions/checkout’ para clonar el repositorio.

- ‘Setup Java 17’: Configura Java 17 para el trabajo.

- ‘Unit Tests’: Ejecuta las pruebas unitarias con el comando ‘./mvnw verify’.

- ‘Rename the artifact’: Renombra el archivo JAR generado con el SHA de GitHub.

- ‘Save the artifact’: Utiliza el paso ‘actions/upload-artifact’ para guardar el artefacto en GitHub Actions.



**‘push\_to\_docker\_hub\_registry’: Este es el segundo trabajo**.

- ‘name’: Un nombre descriptivo para el trabajo.

- ‘runs-on’: Se ejecuta en un entorno de Ubuntu.

- ‘needs’: Indica que este trabajo depende del trabajo anterior (‘build-test’).

- ‘steps’: Lista de pasos que se deben realizar en este trabajo.

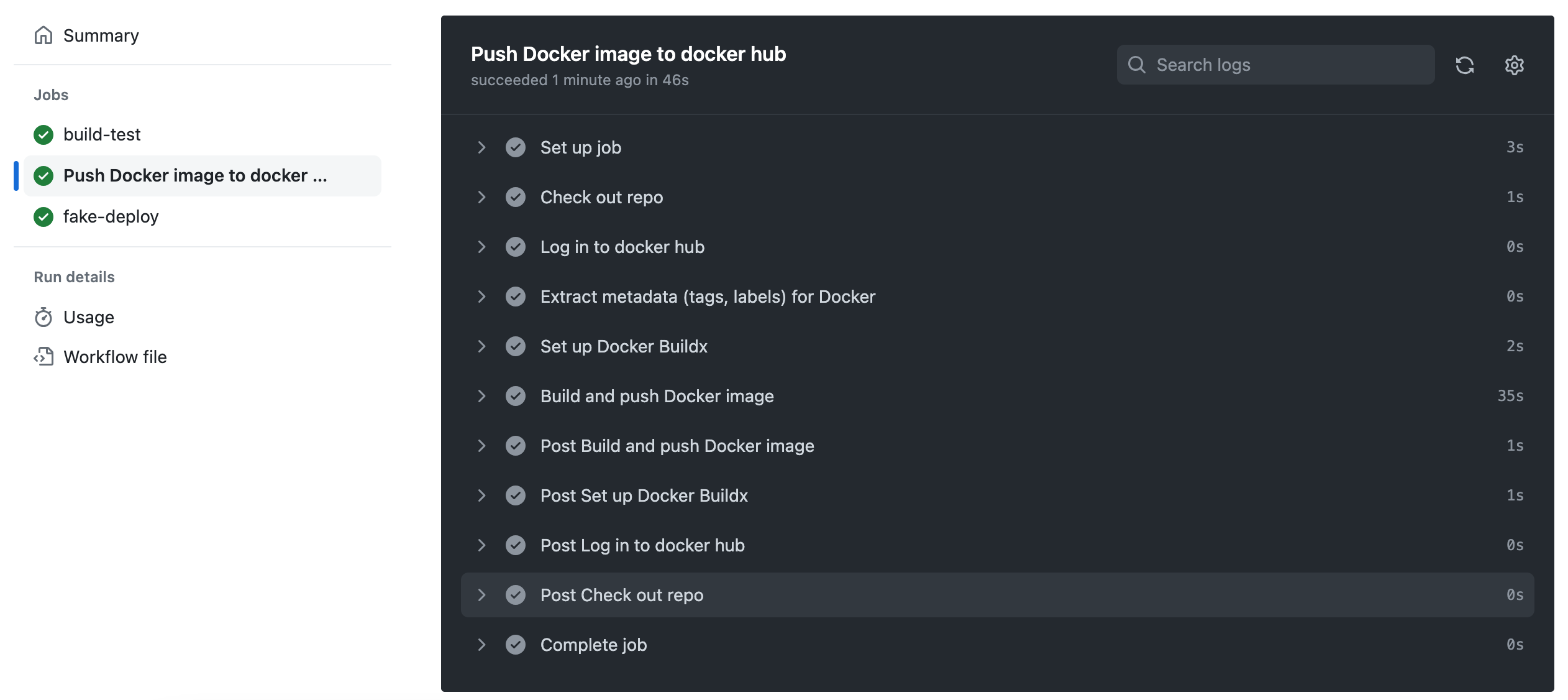
- ‘Check out repo’: Clona el repositorio nuevamente en este trabajo.

- ‘Log in to docker hub’: Inicia sesión en Docker Hub utilizando las credenciales almacenadas en secretos.

- ‘Extract metadata (tags, labels) for Docker’: Utiliza el paso ‘docker/metadata-action’ para obtener metadatos de una imagen Docker.

- ‘Set up Docker Buildx’: Configura Docker Buildx.

- ‘Build and push Docker image’: Utiliza ‘docker/build-push-action’ para construir y subir una imagen Docker al registro de Docker Hub.



**‘fake-deploy’: Este es el tercer trabajo**.

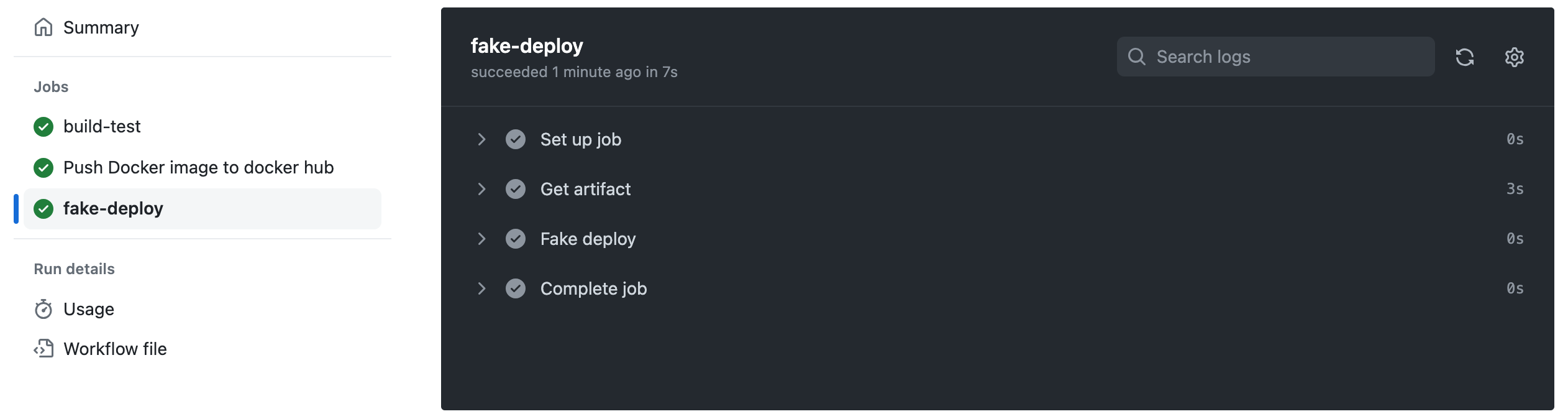
- ‘runs-on’: Se ejecuta en un entorno de Ubuntu.

- ‘needs’: Indica que este trabajo depende del trabajo anterior (‘push\_to\_docker\_hub\_registry’).

- ‘steps’: Lista de pasos que se deben realizar en este trabajo.

- ‘Get artifact’: Descarga el artefacto generado en el primer trabajo.

- ‘Fake deploy’: Ejecuta el comando ‘ls devops-\*.jar’ como una simulación de implementación.



# Conclusión

La automatización en los ciclos de Integración Continua (CI) proporciona una serie de beneficios significativos que contribuyen a la eficiencia, la calidad y la agilidad en el desarrollo de software. Algunas de las principales conclusiones sobre los beneficios de la automatización en los ciclos de CI incluyen estos 3 principales puntos:

1. Eficiencia Mejorada: La automatización de CI elimina tareas manuales repetitivas, lo que ahorra tiempo y recursos, permitiendo un desarrollo más rápido y consistente.
2. Calidad del Software Incrementada: La detección temprana de errores, pruebas rigurosas y despliegues controlados garantizan un software de mayor calidad y confiabilidad.
3. Entrega Continua y Colaboración Eficaz: La automatización facilita la entrega continua de software, permitiendo la colaboración eficiente entre los miembros del equipo y la retroalimentación rápida, lo que acelera el ciclo de desarrollo.