

2023 / 2025

GESTIÓN Y ANÁLISIS DE APLICACIONES EN KUBERNETES: MONITOREO DE MÉTRICAS, COBERTURA DE CÓDIGO Y DESPLIEGUE CONTINUO

ASIR



GitHub Actions



kubernetes



Prometheus



Grafana



Grafana loki



sonarqube

Raúl Herrera Ruiz



ÍNDICE

1. Descripción del proyecto.	3
1.1 Tecnologías que se van a utilizar.	3
1.2 Resultados que se esperan obtener.	3
1.3 Escenario necesario para la realización del proyecto	4
2. Fundamentos teóricos y conceptos	5
2.1 Implantación de aplicaciones web en contenedores	5
2.2 El surgimiento de las aplicaciones contenedoras	5
2.2.1 Docker: La revolución de los contenedores	6
2.2.2 Podman: Contenedores sin daemon central	6
2.2.3 LXC (Linux Containers): Contenedores nativos de Linux	7
2.3 Orquestación de contenedores y el nacimiento de Kubernetes	7
2.4 La necesidad de la Observabilidad	9
2.5 Prometheus: La solución para métricas	10
2.5.1 Funcionamiento de Prometheus	10
2.5.2 Ventajas de Prometheus	10
2.6 Grafana: Visualización clara y personalizable	11
2.6.1 Funcionamiento de Grafana	11
2.6.2 Ventajas de Grafana	11
2.7 Loki: Gestión centralizada de Logs	12
2.7.1 Funcionamiento de Loki	12
2.7.2 Ventajas de Loki	12
2.8 Tests Unitarios en Aplicaciones Web	13
2.8.1 ¿Qué es un test unitario?	13
2.8.3 Jest, framework de testing	13
2.9 La importancia de la calidad de código	14
2.9.1 Beneficios del Análisis de Código	14
2.9.2 SonarQube, el estándar para análisis de código	14



2.10 Integración Continua y Despliegue Continuo (IC/DC)	15
2.10.1 ¿Qué es el IC/DC?	15
2.10.2 Ventajas de usar IC / DC	15
2.11 GitHub Actions: automatización desde el repositorio	16
2.11.1 Funcionamiento de GitHub Actions	16
2.11.2 Ventajas de GitHub Actions	16
3. Instalación del software.	17
3.1 Requisitos previos	17
3.2 Instalación de Docker	17
3.3 Instalación de Minikube	18
3.4 Instalación de Kubernetes	19
3.5 Instalación de SonarQube	20
3.5.1 Instalación de Helm	20
3.5.2 Instalación de SonarQube	21
3.6 Instalación de Prometheus y Grafana	23
3.7 Instalación de Loki	25
3.8 Instalación de Cloudflared	26
4. Creación del escenario y configuración del software.	27
4.1 Preparación del entorno	30
4.2 Tests unitarios	30
4.3 Workflow Github Actions	38
4.4 Despliegue de la App	45
4.5 Grafana, prometheus y loki	51
4.6 Demostraciones	59
5. Conclusiones y propuestas para seguir trabajando sobre el tema	60
6. Bibliografía	61



1. Descripción del proyecto.

Este proyecto se centra en el despliegue de un completo Stack de Observabilidad en un entorno Kubernetes utilizando Debian 12. El objetivo principal es desplegar y monitorear aplicaciones complejas, teniendo en cuenta los tests unitarios de dichas aplicaciones y la cobertura de código.

Se demostrará cómo en un entorno Kubernetes puede configurarse no sólo para ejecutar aplicaciones, sino también para garantizar su calidad, rendimiento y accesibilidad. Los análisis de métricas serán precisos, identificando los puntos críticos que afectan el rendimiento y la estabilidad de las aplicaciones.

1.1 Tecnologías que se van a utilizar.

Las principales tecnologías utilizadas en este proyecto serán:

- Kubernetes para la gestión de las apps
- Prometheus para la recolección de métricas
- Grafana para la visualización de datos
- Loki para la agregación y visualización de logs
- SonarQube para analizar y asegurar una alta cobertura de código.
- Jest para realizar tests unitarios en las aplicaciones.
- GitHub Actions para el despliegue de las aplicaciones.

El hardware usado será un portátil MSI GP72-7RD.

1.2 Resultados que se esperan obtener.

Al finalizar este proyecto, se obtendrá un clúster de Kubernetes completamente monitorizado y analizado, con aplicaciones generando métricas y logs relevantes. Se configurarán dashboards personalizados en Grafana para visualizar métricas clave como el uso de CPU, memoria y errores de las aplicaciones.

Las aplicaciones pasarán análisis de cobertura de código con SonarQube y tests unitarios con Jest, garantizando su calidad de código. Y se desplegará el entorno con GitHub Actions.



Se presentarán ejemplos prácticos de cómo utilizar estas herramientas para detectar problemas de rendimiento, fallos de cobertura y errores de configuración.

1.3 Escenario necesario para la realización del proyecto

Para llevar a cabo este proyecto, será necesario desplegar un stack de aplicaciones en Kubernetes que incluya tanto un frontend como un backend. Además, se deberán integrar todas las herramientas descritas anteriormente dentro del entorno del clúster. Por último, se requerirá un archivo YAML que defina el workflow de GitHub Actions, el cual actuará como orquestador del proceso de integración y despliegue continuo.



2. Fundamentos teóricos y conceptos

2.1 Implantación de aplicaciones web en contenedores

Tradicionalmente, las aplicaciones web se desplegaban en servidores físicos o máquinas virtuales, cada una configurada con su propio sistema operativo y dependencias. Esta forma de hacerlo resultaba ineficiente, ya que cualquier actualización o cambio en la aplicación podría requerir modificaciones complejas en el servidor haciendo que el servicio estuviese detenido un tiempo determinado.

La necesidad de un método más eficaz llevó al desarrollo de los **contenedores**, un enfoque que permite empaquetar aplicaciones con todas sus dependencias (bibliotecas, configuraciones) en una única imagen que puede ejecutarse de manera estable en cualquier entorno.

2.2 El surgimiento de las aplicaciones contenedoras

A medida que las aplicaciones crecieron en complejidad, se hizo evidente que los métodos tradicionales de despliegue, como instalar aplicaciones directamente en servidores físicos o máquinas virtuales, eran ineficientes. Cada aplicación podía tener dependencias específicas, configuraciones únicas y necesitar diferentes versiones de bibliotecas, lo que complicaba su gestión y despliegue.

Esto llevó al desarrollo del concepto de contenedores.

A diferencia de las máquinas virtuales, las contenedoras son ligeras porque comparten el kernel del sistema operativo, lo que reduce significativamente el consumo de recursos.



2.2.1 Docker: La revolución de los contenedores

Docker fue la primera plataforma en popularizar el uso de contenedores, ofreciendo una interfaz intuitiva para crear, gestionar y ejecutar contenedores de manera sencilla. Introdujo conceptos clave como:

Imágenes: Plantillas inmutables que contienen una aplicación y sus dependencias.

Contenedores: Instancias en ejecución de una imagen.

Docker Hub: Un repositorio público para compartir imágenes.

Docker se convirtió en el estándar para contenedores debido a su facilidad de uso y a su amplia adopción.

2.2.2 Podman: Contenedores sin daemon central

Podman es una alternativa a Docker diseñada para mejorar la seguridad y flexibilidad.

A diferencia de Docker, Podman no depende de un daemon central, lo que significa que los contenedores se ejecutan directamente bajo el control del usuario, sin requerir privilegios de root.

Las características de Podman son las siguientes:

- Sin Daemon: No necesita un servicio en segundo plano para funcionar.
- Compatibilidad: Utiliza el mismo formato de imágenes que Docker.
- Mayor Seguridad: Permite ejecutar contenedores sin permisos elevados, mejorando la seguridad del sistema.



2.2.3 LXC (Linux Containers): Contenedores nativos de Linux

LXC fue una de las primeras tecnologías de contenedores disponibles en Linux, diseñada para crear entornos aislados y ligeros que se comportan como sistemas completos. A diferencia de Docker y Podman, LXC ofrece un control más detallado sobre el entorno del contenedor.

Las características de LXC son :

- Aislamiento completo: Permite definir recursos específicos para cada contenedor.
- Control granular: Configuración avanzada de redes, almacenamiento y procesos.
- Flexibilidad: Ideal para entornos en los que se necesita un mayor control.

2.3 Orquestación de contenedores y el nacimiento de Kubernetes

Para gestionar aplicaciones complejas compuestas de múltiples contenedores, surgió la necesidad de crear orquestadores de contenedores. Un orquestador gestiona el ciclo de vida de los contenedores, asegurando que se desplieguen correctamente, se escalen automáticamente según la carga y se recuperen en caso de fallo.

Inicialmente, Docker ofreció su propio orquestador llamado Docker Swarm, pero Kubernetes se destacó rápidamente como el estándar debido a su flexibilidad, escalabilidad y soporte por parte de grandes empresas como Google (su creador).

Kubernetes proporciona funcionalidades avanzadas como:

Distribución y escalado inteligente: Ubica automáticamente los contenedores en los nodos adecuados según sus necesidades de recursos, permitiendo un escalado automático basado en el uso de CPU o manualmente según las necesidades del usuario.



Gestión integrada de servicios y balanceo de carga: Asigna direcciones IP y nombres DNS únicos a los servicios, distribuyendo de manera equilibrada las solicitudes entre los contenedores asociados.

Implementación segura y controlada: Permite realizar despliegues progresivos, monitoreando su estado y revirtiendo automáticamente si se detectan errores, asegurando la estabilidad de las aplicaciones.

Planificación y optimización de recursos: Selecciona de manera eficiente los nodos donde se ejecutarán los contenedores, equilibrando las cargas y las tareas de menor prioridad para maximizar el rendimiento.

Protección de datos sensibles: Gestiona de manera segura la configuración y los secretos (contraseñas, claves API) sin exponerlos en las imágenes de las aplicaciones.

Almacenamiento flexible: Facilita el montaje automático de volúmenes de almacenamiento, ya sea local, en la nube (AWS, GCP) o en red (NFS, Ceph), adaptándose a las necesidades de las aplicaciones.

Recuperación automática y gestión de errores: Supervisa el estado de los contenedores, reiniciando aquellos que fallen y reprogramando automáticamente los nodos en caso de problemas.



2.4 La necesidad de la Observabilidad

A medida que las aplicaciones en Kubernetes se volvieron más complejas, se hizo esencial contar con herramientas para observar su comportamiento. Esto dio lugar al concepto de observabilidad, que se centra en recopilar y analizar métricas, logs y trazas para comprender el estado y rendimiento de las aplicaciones.

Métricas: Datos numéricos que reflejan el estado del sistema (uso de CPU, memoria, latencia).

Logs: Registros textuales que detallan eventos y errores de las aplicaciones.

Trazas: Información detallada del flujo de ejecución entre componentes (útil en microservicios).

La observabilidad permite:

- Detectar problemas antes de que afecten a los usuarios.
- Identificar cuellos de botella de rendimiento.
- Realizar análisis forenses en caso de incidentes.



2.5 **Prometheus: La solución para métricas**

Prometheus es una herramienta de monitoreo desarrollada inicialmente por SoundCloud y adoptada como un proyecto oficial de la Cloud Native Computing Foundation (CNCF). Está diseñada específicamente para recolectar métricas en tiempo real, principalmente en entornos de Kubernetes, pero también se puede usar con otros servicios y aplicaciones.

Su arquitectura está basada en un modelo de recolección de métricas de tipo pull, donde Prometheus consulta activamente los endpoints de las aplicaciones para obtener información.

2.5.1 **Funcionamiento de Prometheus**

Scraping (Recolección de Métricas): Prometheus utiliza trabajos de scraping para conectarse periódicamente a los endpoints de las aplicaciones que exponen métricas. Estos endpoints suelen estar formateados en el estándar Prometheus (formato texto en /metrics).

Almacenamiento en series temporales: Cada métrica recopilada se almacena en una base de datos interna basada en series temporales (time series), lo que permite analizar su evolución a lo largo del tiempo.

Lenguaje de consultas (PromQL): Proporciona un lenguaje de consultas llamado PromQL, que permite realizar cálculos avanzados sobre las métricas recopiladas, como promedios, máximos, mínimos y tasas de crecimiento.

Alertas y notificaciones: Integra un sistema de alertas que permite definir reglas basadas en las métricas. Cuando se cumplen las condiciones configuradas, se generan alertas que pueden ser enviadas a diferentes canales (correo electrónico, Slack, Webhooks, etc.).

2.5.2 **Ventajas de Prometheus**

- Escalabilidad: Puede monitorizar miles de métricas en tiempo real.
- Independencia: No requiere una base de datos externa, lo que simplifica su despliegue.
- Flexibilidad: Compatible con múltiples fuentes de métricas, no solo aplicaciones en Kubernetes.



2.6 Grafana: Visualización clara y personalizable

Grafana es una plataforma de visualización de datos de código abierto que se ha convertido en el estándar para la creación de dashboards dinámicos. Permite transformar métricas en gráficos, tablas y paneles visuales que facilitan la interpretación de datos complejos. Aunque se integra perfectamente con Prometheus, también puede utilizar otras fuentes de datos como InfluxDB, MySQL, Elasticsearch y más.

2.6.1 Funcionamiento de Grafana

Dashboards dinámicos: Los usuarios pueden crear dashboards personalizables para visualizar métricas clave, con gráficos que se actualizan en tiempo real.

Paneles compartibles: Los dashboards se pueden compartir fácilmente entre equipos, lo que facilita la colaboración entre desarrollo y operaciones.

Alertas visuales: Grafana permite configurar alertas directamente en los gráficos, resaltando valores críticos. Por ejemplo, una línea que cambia de color al superar un umbral específico.

Consultas interactivas: Los usuarios pueden aplicar filtros, cambiar el rango de tiempo y modificar las consultas para adaptar los gráficos a sus necesidades.

2.6.2 Ventajas de Grafana

- Interfaz intuitiva: Su diseño gráfico permite visualizar métricas de manera clara.
- Compatibilidad multifuente: Puede conectarse a múltiples fuentes de datos simultáneamente.
- Seguridad: Soporte para autenticación de usuarios y control de acceso a dashboards.



2.7 **Loki: Gestión centralizada de Logs**

Loki es una solución de logging ligera y escalable creada por Grafana Labs, diseñada específicamente para Kubernetes. A diferencia de otras herramientas de logging, Loki no indexa el contenido completo de los logs, sino solo sus etiquetas (como nombre del contenedor, namespace, pod, etc.). Esto reduce significativamente el consumo de recursos.

2.7.1 **Funcionamiento de Loki**

Recolección ligera: Solo almacena las etiquetas de los logs y el contenido en bruto, lo que optimiza el rendimiento.

Integración con Grafana: Permite visualizar y analizar logs directamente en los dashboards de Grafana, junto a las métricas de Prometheus.

Filtros avanzados: Se pueden aplicar filtros para buscar patrones específicos en los logs (errores, advertencias, eventos críticos).

Escalabilidad Horizontal: Loki puede desplegarse como un servicio distribuido, ideal para entornos grandes de Kubernetes.

2.7.2 **Ventajas de Loki**

- Eficiencia: Utiliza menos recursos que otras soluciones de logging al no indexar el contenido completo.
- Búsqueda rápida: Gracias a su modelo basado en etiquetas, las búsquedas son rápidas y eficientes.
- Visualización integrada: Compatible directamente con Grafana, permitiendo una vista unificada de métricas y logs.



2.8 Tests Unitarios en Aplicaciones Web

Los tests unitarios permiten verificar que cada unidad de código (funciones, servicios, componentes, etc.) funcione correctamente de forma aislada. Son una práctica fundamental en el desarrollo moderno, ya que ayudan a prevenir errores, facilitan el mantenimiento del software y fomentan la confianza al modificar el código.

2.8.1 ¿Qué es un test unitario?

Un test unitario es una verificación automatizada que evalúa una unidad mínima de código. Sus principales características son:

- Aisladas: No dependen de otras partes del sistema.
- Repetibles: Ejecutan siempre con los mismos resultados.
- Rápidas: Ideales para la integración continua.

2.8.3 Jest, framework de testing

Jest es un framework de testing desarrollado por Facebook, muy utilizado en entornos JavaScript y TypeScript. Este framework ofrece lo siguiente:

- Sintaxis clara y expresiva.
- Soporte para mocks, pruebas asíncronas y temporizadores.
- Generación automática de cobertura de código.
- Integración sencilla en pipelines CI/CD.

Jest permite validar el comportamiento esperado del código antes de que llegue a producción, contribuyendo a una base sólida y confiable.



2.9 La importancia de la calidad de código

En el desarrollo de aplicaciones modernas, asegurarse que el código sea de alta calidad es fundamental para evitar errores, mejorar el rendimiento y garantizar su mantenimiento a largo plazo. El análisis de calidad de código permite identificar vulnerabilidades, código duplicado, errores potenciales y evaluar la cobertura de pruebas.

2.9.1 Beneficios del Análisis de Código

Prevención de errores: Detecta problemas antes de que lleguen a producción.

Mejora continua: Facilita el seguimiento de la calidad del código a lo largo del tiempo.

Seguridad: Identifica posibles vulnerabilidades de seguridad en el código.

2.9.2 SonarQube, el estándar para análisis de código

SonarQube es una plataforma de código abierto que permite analizar automáticamente la calidad del código de las aplicaciones, generando informes detallados que indican:

- Cobertura de código: Porcentaje de líneas cubiertas por pruebas.
- Vulnerabilidades: Problemas de seguridad identificados.
- Duplicados: Bloques de código que se repiten.
- Buenas prácticas: Asegura que el código sigue estándares de calidad.



2.10 Integración Continua y Despliegue Continuo (IC/DC)

La Integración Continua (IC) y el Despliegue Continuo (DC) son prácticas fundamentales en el desarrollo moderno de software, especialmente en entornos basados en contenedores como Kubernetes. Estas prácticas permiten automatizar pruebas, validaciones y despliegues de aplicaciones, asegurando mayor calidad, agilidad y reducción de errores en producción.

2.10.1 ¿Qué es el IC/DC?

- Integración Continua (IC): Consiste en integrar de manera automática los cambios realizados por los desarrolladores en un repositorio común. Cada cambio desencadena una cadena de pruebas automatizadas para validar que el nuevo código no rompe funcionalidades existentes.
- Despliegue Continuo (DC): Permite que, una vez validadas las pruebas, la aplicación se despliegue automáticamente en entornos de desarrollo, pruebas o producción, sin intervención manual.

2.10.2 Ventajas de usar IC / DC

- Automatización del ciclo de vida del software: Desde que se realiza un commit hasta que la aplicación se despliega.
- Detección temprana de errores: Gracias a las pruebas automatizadas.
- Despliegues más seguros y frecuentes: Se reduce el riesgo de errores en producción.
- Retroalimentación rápida: Los equipos de desarrollo reciben información inmediata sobre el estado del código.



2.11 GitHub Actions: automatización desde el repositorio

GitHub Actions es la herramienta de automatización de flujos de trabajo integrada en GitHub. Permite crear pipelines de IC/DC directamente desde los repositorios del proyecto, lo que facilita la integración con el código fuente y su despliegue automatizado.

2.11.1 Funcionamiento de GitHub Actions

- Workflows: Definidos en archivos YAML dentro del repositorio, especifican cuándo y cómo se deben ejecutar las acciones (por ejemplo, al hacer push, pull request, o manualmente).
- Jobs y Steps: Cada workflow puede contener varios jobs, que a su vez contienen pasos específicos como instalación de dependencias, ejecución de pruebas, construcción de contenedores y despliegue.
- Integración con Kubernetes y Docker: GitHub Actions permite construir imágenes Docker, subirlas a un registry y desplegarlas en clústeres de Kubernetes de forma automatizada.

2.11.2 Ventajas de GitHub Actions

- Entorno gestionado por GitHub: No es necesario mantener servidores para el pipeline.
- Alta integración: Compatible con contenedores, Helm, kubectl, y otros servicios de DevOps.
- Modularidad: Gran ecosistema de acciones reutilizables desde el marketplace de GitHub.
- Transparencia: Toda la ejecución queda registrada en el repositorio.



3. *Instalación del software.*

3.1 Requisitos previos

Lo primero que necesitaremos será tener el sistema operativo (Debian 12) actualizado.

```
sudo apt update && sudo apt upgrade -y
```

```
raulhr@tfg:~$ sudo apt update && sudo apt upgrade -y
```

Una vez hecho esto, instalaremos las dependencias necesarias

```
sudo apt install curl wget apt-transport-https gnupg2  
ca-certificates lsb-release -y
```

```
raulhr@tfg:~$ sudo apt install curl wget apt-transport-https gnupg2 ca-certificates lsb-release -y
```

3.2 Instalación de Docker

Para poder usar minikube y kubernetes vamos a necesitar instalarnos Docker, para ello haremos lo siguiente

```
sudo apt install docker.io
```

```
raulhr@tfg:~$ sudo apt install docker.io
```

Para poder usar docker con un usuario sin privilegios usaremos el siguiente comando, una vez ejecutado el comando tendremos que salir y volver a entrar de él para que se actualicen los permisos.

```
sudo usermod -aG docker (usuario)
```

```
raulhr@tfg:~$ sudo usermod -aG docker raulhr
```

Comprobamos que se ha instalado de forma correcta

```
raulhr@tfg:~$ docker --version  
Docker version 20.10.24+dfsg1, build 297e128
```



3.3 Instalación de Minikube

Lo primero que haremos será instalar Minikube antes que Kubernetes.

```
curl -LO
https://storage.googleapis.com/minikube/releases/latest/minikube\_1
atest\_amd64.deb

sudo dpkg -i minikube_latest_amd64.deb
```

```
raulhr@tfg:~$ curl -LO https://storage.googleapis.com/minikube/releases/latest/minikube_latest_amd64.deb
% Total    % Received % Xferd Average Speed   Time    Time     Time  Current
           Dload  Upload   Total   Spent    Left   Speed
100 39.0M  100 39.0M    0     0  26.6M    0  0:00:01  0:00:01  --:--:-- 26.6M
raulhr@tfg:~$ sudo dpkg -i minikube_latest_amd64.deb
Seleccionando el paquete minikube previamente no seleccionado.
(Leyendo la base de datos ... 159694 ficheros o directorios instalados actualmente.)
Preparando para desempaquetar minikube_latest_amd64.deb ...
Desempaquetando minikube (1.36.0-0) ...
Configurando minikube (1.36.0-0) ...
```

Una vez hecho esto comprobamos que se ha instalado correctamente

```
raulhr@tfg:~$ minikube version
minikube version: v1.36.0
commit: f8f52f5de11fc6ad8244afac475e1d0f96841df1-dirty
```

Y lo iniciamos con el driver que queramos, en mi caso será docker

```
minikube start --driver=docker
```

```
raulhr@tfg:~$ minikube start --driver=docker
🐹 minikube v1.36.0 en Debian 12.11 (kvm/amd64)
🌟 Using the docker driver based on user configuration
👉 Using Docker driver with root privileges
👉 Starting "minikube" primary control-plane node in "minikube" cluster
📦 Pulling base image v0.0.47 ...
📦 Descargando Kubernetes v1.33.1 ...
> preloaded-images-k8s-v18-v1...: 347.04 MiB / 347.04 MiB 100.00% 20.62 M
> gcr.io/k8s-minikube/kicbase...: 502.26 MiB / 502.26 MiB 100.00% 22.81 M
🔥 Creating docker container (CPUs=2, Memory=2200MB) ...
🐹 Preparando Kubernetes v1.33.1 en Docker 28.1.1...
  • Generando certificados y llaves
  • Iniciando plano de control
  • Configurando reglas RBAC...
🔗 Configurando CNI bridge CNI ...
🔗 Verifying Kubernetes components...
  • Using image gcr.io/k8s-minikube/storage-provisioner:v5
🌟 Complementos habilitados: default-storageclass, storage-provisioner
💡 kubectl not found. If you need it, try: 'minikube kubectl -- get pods -A'
🎉 Done! kubectl is now configured to use "minikube" cluster and "default" namespace by default
```



3.4 Instalación de Kubernetes

Una vez tenemos Docker y Minikube pasamos a la instalación de Kubernetes, como hemos comprobado anteriormente la versión de kubernetes que necesitamos es la 1.33.1 porque es la que está usando minikube

```
raulhr@tfg:~$ minikube start --driver=docker
🌻 minikube v1.36.0 en Debian 12.11 (kvm/amd64)
🌟 Using the docker driver based on user configuration
🌸 Using Docker driver with root privileges
👍 Starting "minikube" primary control-plane node in "minikube" cluster
📦 Pulling base image v0.0.47 ...
📦 Descargando Kubernetes v1.33.1 ...
> preloaded-images-k8s-v18-v1...: 347.04 MiB / 347.04 MiB 100.00% 20.62 M
> gcr.io/k8s-minikube/kicbase...: 502.26 MiB / 502.26 MiB 100.00% 22.81 M
🔥 Creating docker container (CPUs=2, Memory=2200MB) ...
🐳 Preparando Kubernetes v1.33.1 en Docker 28.1.1...
```

Por ello vamos a instalarnos ese cliente específico

```
curl -LO
"https://dl.k8s.io/release/v1.33.1/bin/linux/amd64/kubectl"

chmod +x kubectl

sudo mv kubectl /usr/local/bin/
```

```
raulhr@tfg:~$ curl -LO "https://dl.k8s.io/release/v1.33.1/bin/linux/amd64/kubectl"
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left   Speed
100 138 100 138    0    0    509      0  --:--:--  --:--:--  --:--:--   509
100 57.3M 100 57.3M    0    0 30.5M      0  0:00:01  0:00:01  --:--:-- 39.4M
raulhr@tfg:~$ chmod +x kubectl
raulhr@tfg:~$ sudo mv kubectl /usr/local/bin/
```

Ahora pasamos a comprobar que se ha instalado de forma correcta

```
raulhr@tfg:~$ kubectl version
Client Version: v1.33.1
Kustomize Version: v5.6.0
Server Version: v1.33.1
```

Se ha instalado correctamente la versión que necesitamos.



3.5 Instalación de SonarQube

Para instalar Sonar necesitaremos más recursos, para ello haremos lo siguiente

```
minikube config set memory 4096
minikube config set cpus 2
minikube delete && minikube start
```

```
raulhr@tfgr:~$ minikube config set memory 4096
! These changes will take effect upon a minikube delete and then a minikube start
raulhr@tfgr:~$ minikube config set cpus 2
! These changes will take effect upon a minikube delete and then a minikube start
raulhr@tfgr:~$ minikube delete && minikube start
🔥 Eliminando "minikube" en docker...
🔥 Eliminando contenedor "minikube" ...
🔥 Eliminando /home/raulhr/.minikube/machines/minikube...
💀 Removed all traces of the "minikube" cluster.
😄 minikube v1.36.0 en Debian 12.11 (kvm/amd64)
🌟 Controlador docker seleccionado automáticamente
👉 Using Docker driver with root privileges
👍 Starting "minikube" primary control-plane node in "minikube" cluster
📦 Pulling base image v0.0.47 ...
🔥 Creating docker container (CPUs=2, Memory=4096MB) ...
🐳 Preparando Kubernetes v1.33.1 en Docker 28.1.1...
  • Generando certificados y llaves
  • Iniciando plano de control
  • Configurando reglas RBAC...
🔗 Configurando CNI bridge CNI ...
🔍 Verifying Kubernetes components...
  • Using image gcr.io/k8s-minikube/storage-provisioner:v5
🌟 Complementos habilitados: storage-provisioner, default-storageclass
🏁 Done! kubectl is now configured to use "minikube" cluster and "default" namespace by default
```

Y ahora instalaremos SonarQube usando Helm

3.5.1 Instalación de Helm

Helm es una herramienta que simplifica la administración de aplicaciones en Kubernetes

```
wget https://get.helm.sh/helm-v3.8.0-linux-amd64.tar.gz
tar -zxvf helm-v3.8.0-linux-amd64.tar.gz
sudo mv linux-amd64/helm /usr/local/bin/helm
```

```
raulhr@tfgr:~$ wget https://get.helm.sh/helm-v3.8.0-linux-amd64.tar.gz
--2025-05-27 19:23:02-- https://get.helm.sh/helm-v3.8.0-linux-amd64.tar.gz
Resolviendo get.helm.sh (get.helm.sh)... 13.107.246.77, 2620:1ec:bdf::77
Conectando con get.helm.sh (get.helm.sh)[13.107.246.77]:443... conectado.
Petición HTTP enviada, esperando respuesta... 200 OK
Longitud: 13626774 (13M) [application/x-tar]
Grabando a: «helm-v3.8.0-linux-amd64.tar.gz»

helm-v3.8.0-linux-amd64.tar.gz 100%[=====] 13,00M 36,0MB/s en 0,4s
2025-05-27 19:23:02 (36,0 MB/s) - «helm-v3.8.0-linux-amd64.tar.gz» guardado [13626774/13626774]

raulhr@tfgr:~$ tar -zxvf helm-v3.8.0-linux-amd64.tar.gz
linux-amd64/
linux-amd64/helm
linux-amd64/LICENSE
linux-amd64/README.md
raulhr@tfgr:~$ sudo mv linux-amd64/helm /usr/local/bin/helm
raulhr@tfgr:~$ helm version
version.BuildInfo{Version:"v3.8.0", GitCommit:"d14138609b01886f544b2025f500351c9eb092e", GitTreeState:"clean", GoVersion:"go1.17.5"}
```

Una vez instalado Helm, continuamos con la instalación de SonarQube.



3.5.2 Instalación de SonarQube

Añadimos el repositorio donde se encuentra Sonar a helm y crearemos un namespace específico para Sonar.

```
helm repo add sonarqube
https://SonarSource.github.io/helm-chart-sonarqube
helm repo update
```

Ahora creamos el siguiente archivo de configuración para Sonar

```
nano sonarqube-values.yaml
community:
  enabled: true

jdbcOverwrite:
  enable: false
  oracleJdbcDriver:
    url: ""
    username: ""
    password: ""

postgresql:
  enabled: true
  postgresqlPassword: sonarpass
  postgresqlUsername: sonar
  postgresqlDatabase: sonarqube

sonarqube:
  web:
    contextPath: "/"
  jvmOpts: "-Xmx512m -Xms512m"

service:
  type: ClusterIP

persistence:
  enabled: false

resources:
  requests:
    cpu: "500m"
```



```
memory: "1Gi"
limits:
  cpu: "1"
  memory: "2Gi"
```

```
monitoringPasscode: "sonar-monitoring-pass"
```

Ejecutamos el siguiente comando para instalarlo

```
helm install sonarqube sonarqube/sonarqube -f
sonarqube-values.yaml --create-namespace sonarqube -n sonarqube
```

```
raulhr@tfq:~$ helm install sonarqube sonarqube/sonarqube -f sonarqube-values.yaml --create-namespace
sonarqube -n sonarqube
```

Y ya tendríamos SonarQube instalado y totalmente funcional.

```
raulhr@tfq:~$ kubectl get all -n sonarqube
NAME                                READY   STATUS    RESTARTS   AGE
pod/sonarqube-postgresql-0         1/1     Running   0           18m
pod/sonarqube-sonarqube-0         1/1     Running   0           18m

NAME                                TYPE          CLUSTER-IP   EXTERNAL-IP   PORT(S)    AGE
service/sonarqube-postgresql       ClusterIP    10.110.180.4 <none>        5432/TCP    18m
service/sonarqube-postgresql-headless ClusterIP     None         <none>        5432/TCP    18m
service/sonarqube-sonarqube        ClusterIP    10.109.132.205 <none>        9000/TCP    18m

NAME                                READY   AGE
statefulset.apps/sonarqube-postgresql 1/1     18m
statefulset.apps/sonarqube-sonarqube   1/1     18m
```

(La app se muestra ahora mismo por un port-forward de kubectl, más tarde veremos cómo configurar el ingress para acceder a él por una url)

The screenshot shows the SonarQube Community web interface. The main heading is "How do you want to create your project?". Below this, there is a question: "Do you want to benefit from all of SonarQube Community Build's features (like repository import and Pull Request decoration)?" followed by a "Create your project from your favorite DevOps platform." section. This section lists several options with "Setup" buttons: "Import from Azure DevOps", "Import from Bitbucket Cloud", "Import from Bitbucket Server", "Import from GitHub", and "Import from GitLab". At the bottom, there is a section for "Are you just testing or have an advanced use-case?" with a "Create a local project" button.



3.6 Instalación de Prometheus y Grafana

Para la instalación de estas dos herramientas necesitaremos usar Helm.

Añadimos el repositorio de prometheus a helm y actualizamos los repositorios de helm

```
helm repo add prometheus-community
https://prometheus-community.github.io/helm-charts
helm repo update
```

```
raulhr@tfg:~$ helm repo add prometheus-community https://prometheus-community.github.io/helm-charts
"prometheus-community" has been added to your repositories
raulhr@tfg:~$ helm repo update
Hang tight while we grab the latest from your chart repositories...
..Successfully got an update from the "prometheus-community" chart repository
Update Complete. *Happy Helming!*
```

Creamos un nuevo namespace donde instalaremos las aplicaciones de monitoreo

```
helm install prometheus prometheus-community/kube-prometheus-stack
--create-namespace --namespace monitoreo
```

```
raulhr@tfg:~$ helm install prometheus prometheus-community/kube-prometheus-stack --namespace monitoreo
NAME: prometheus
LAST DEPLOYED: Tue May 27 19:25:15 2025
NAMESPACE: monitoreo
STATUS: deployed
REVISION: 1
NOTES:
kube-prometheus-stack has been installed. Check its status by running:
  kubectl --namespace monitoreo get pods -l "release=prometheus"

Get Grafana 'admin' user password by running:

  kubectl --namespace monitoreo get secrets prometheus-grafana -o jsonpath="{.data.admin-password}" | base64 -d ; echo

Access Grafana local instance:

  export POD_NAME=$(kubectl --namespace monitoreo get pod -l "app.kubernetes.io/name=grafana,app.kubernetes.io/instance=prometheus" -oname)
  kubectl --namespace monitoreo port-forward $POD_NAME 3000

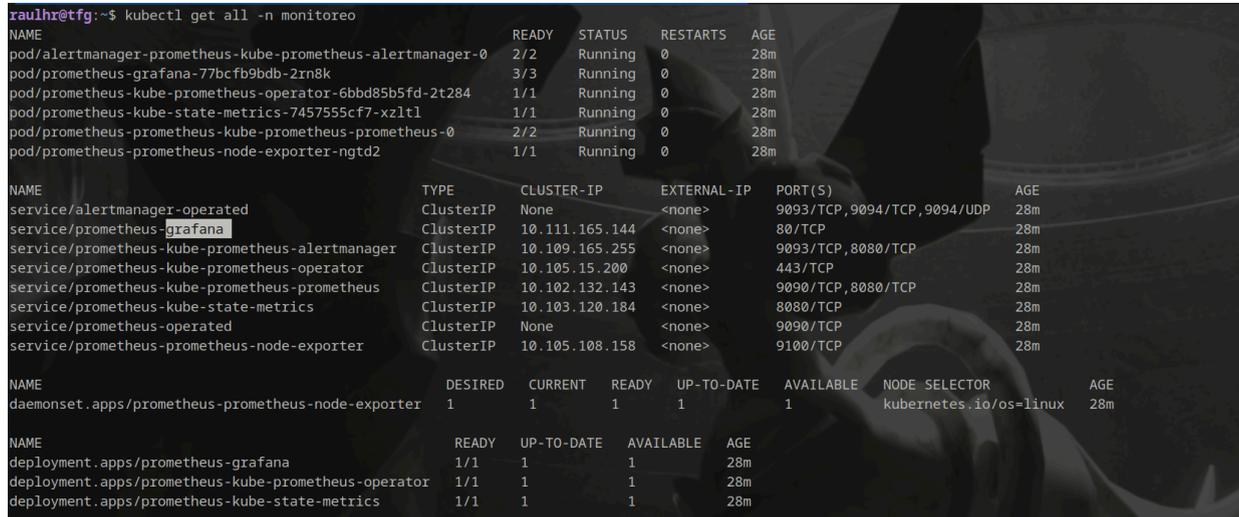
Visit https://github.com/prometheus-operator/kube-prometheus for instructions on how to create & configure Alertmanager and Prometheus instances using the Operator.
```

En la misma instalación nos muestra una guía de lo que puedes hacer para comenzar, pero nosotros lo configuraremos más tarde.



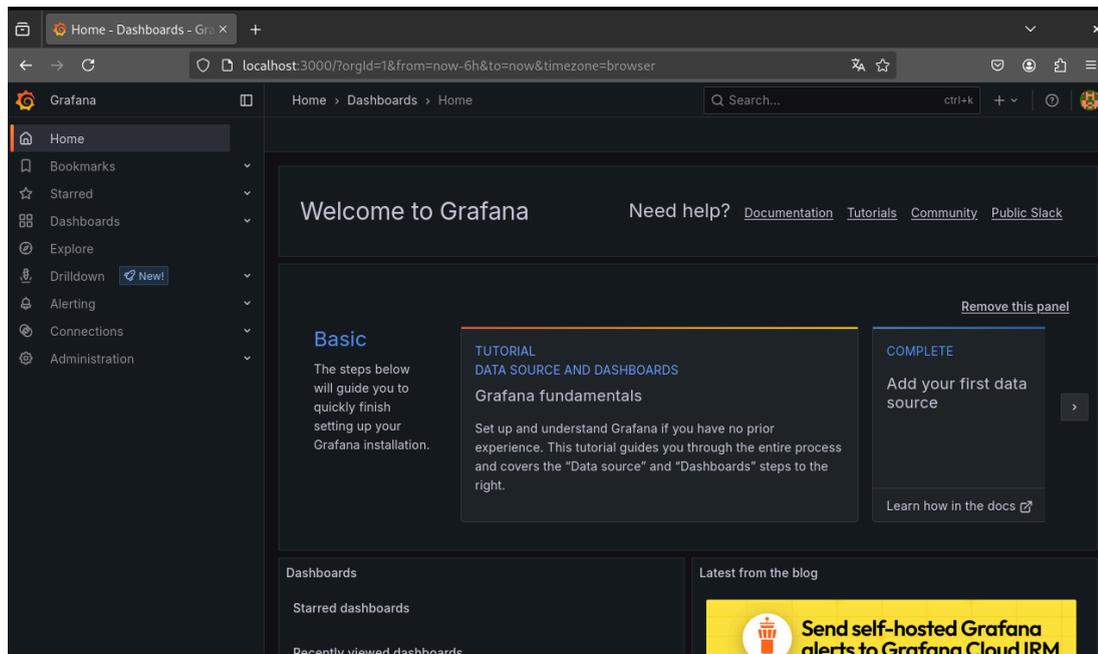
Podremos ver que está todo corriendo correctamente usando el siguiente comando

```
kubectl get all -n monitoreo
```



Esto nos mostrará todo lo que ha generado la instalación previa que hemos hecho con helm en el namespace monitoreo.

Observamos que también aparece grafana de forma correcta.





3.7 Instalación de Loki

Una vez instalado Grafana, pasamos con la instalación de Loki, para ello agregaremos el repositorio de grafana a helm e instalaremos loki usando helm en el namespace monitoreo creado previamente.

```
helm repo add grafana https://grafana.github.io/helm-charts
helm repo update
```

```
helm install loki grafana/loki-stack --namespace monitoreo
```

```
raulhr@tfgr:~$ helm repo add grafana https://grafana.github.io/helm-charts
"grafana" has been added to your repositories
raulhr@tfgr:~$ helm repo update
Hang tight while we grab the latest from your chart repositories...
...Successfully got an update from the "prometheus-community" chart repository
...Successfully got an update from the "grafana" chart repository
Update Complete. *Happy Helming!*
raulhr@tfgr:~$ helm install loki grafana/loki-stack --namespace monitoreo
NAME: loki
LAST DEPLOYED: Wed May 28 16:21:28 2025
NAMESPACE: monitoreo
STATUS: deployed
REVISION: 1
NOTES:
The Loki stack has been deployed to your cluster. Loki can now be added as a datasource in Grafana.
See http://docs.grafana.org/features/datasources/loki/ for more detail.
```

Comprobamos que esté instalado correctamente usando

```
kubectl get all -n monitoreo | grep loki
```

```
raulhr@tfgr:~$ kubectl get all -n monitoreo | grep loki
pod/loki-0                                1/1      Running    0          2m35s
pod/loki-promtail-js92m                  1/1      Running    0          2m35s
service/loki                             ClusterIP 10.111.88.165 <none> 3100/TCP 2m36s
service/loki-headless                    ClusterIP None <none> 3100/TCP 2m36s
service/loki-memberlist                  ClusterIP None <none> 7946/TCP 2m36s
daemonset.apps/loki-promtail             1         1          1          1          1          <none> 2m36s
statefulset.apps/loki                     1/1      2m36s
```

Vemos que está todo corriendo sin ningún error.



3.8 Instalación de Cloudflare

Necesitamos exponer las apps que tenemos en local a la red pública, para ello usaremos Cloudflared. Lo instalamos de la siguiente manera

```
curl -L
https://github.com/cloudflare/cloudflared/releases/latest/download
/cloudflared-linux-amd64 -o cloudflared
chmod +x cloudflared
sudo mv cloudflared /usr/local/bin/
```

```
raulhr@tfg:~$ curl -L https://github.com/cloudflare/cloudflared/releases/latest/download/cloudflared-linux-amd64 -o cloudflared
chmod +x cloudflared
sudo mv cloudflared /usr/local/bin/
```

Ahora ya podemos exponer nuestra app local a la red pública.



4. Creación del escenario y configuración del software.

Hemos observado haciendo

```
kubectl get all -n monitoreo
```

Que el pod de grafana está fallando

```
raulhr@tfgr:~$ kubectl get all -n monitoreo
```

NAME	READY	STATUS	RESTARTS	AGE
pod/alertmanager-prometheus-kube-prometheus-alertmanager-0	2/2	Running	2 (5m59s ago)	4d6h
pod/loki-0	1/1	Running	1 (5m59s ago)	4d6h
pod/loki-promtail-snnss	1/1	Running	1 (5m59s ago)	4d6h
pod/prometheus-grafana-77bcfb9bdb-jxgf5	2/3	CrashLoopBackOff	17 (2m15s ago)	4d6h

Comprobando los logs usando

```
kubectl logs -n monitoreo deployment/prometheus-grafana
```

Observamos que hay un fallo en un archivo de configuración llamado **datasource.yaml**

```
logger=provisioning t=2025-06-02T19:47:40.41603514Z level=error msg="Failed to provision data sources"
error="Datasource provisioning error: datasource.yaml config is invalid. Only one datasource per organi
zation can be marked as default"
logger=provisioning t=2025-06-02T19:47:40.416064028Z level=error msg="Failed to provision data sources"
error="Datasource provisioning error: datasource.yaml config is invalid. Only one datasource per organ
ization can be marked as default"
```

Una vez hemos identificado el fallo, vamos a buscar dicho archivo

```
sudo find / -name *datasource.yaml*
```

```
raulhr@tfgr:~$ sudo find / -name *datasource.yaml*
find: '/run/user/1000/gvfs': Permiso denegado
find: '/run/user/112/gvfs': Permiso denegado
/var/lib/docker/volumes/minikube/_data/lib/kubelet/pods/453c209d-61b2-4be4-b6ec-afdb4a126ba1/volumes/kubernetes.io~empty-dir/sc-datasources-volume/loki-stack-datasource.yaml
/var/lib/docker/volumes/minikube/_data/lib/kubelet/pods/453c209d-61b2-4be4-b6ec-afdb4a126ba1/volumes/kubernetes.io~empty-dir/sc-datasources-volume/datasource.yaml
```



Una vez localizados los archivos que están fallando vamos a comprobar el contenido que tienen

```
raulhr@tfgr:~$ sudo cat /var/lib/docker/volumes/minikube/_data/lib/kubelet/pods/453c209d-61b2-4be4-b6ec-afdb4a126ba1/volumes/kubernetes.io-empty-dir/sc-datasources-volume/loki-stack-datasource.yaml
apiVersion: 1
datasources:
- name: Loki
  type: loki
  access: proxy
  url: "http://loki:3100"
  version: 1
  isDefault: true
  jsonData:
    {}
raulhr@tfgr:~$ sudo cat /var/lib/docker/volumes/minikube/_data/lib/kubelet/pods/453c209d-61b2-4be4-b6ec-afdb4a126ba1/volumes/kubernetes.io-empty-dir/sc-datasources-volume/datasource.yaml
apiVersion: 1
datasources:
- name: "Prometheus"
  type: prometheus
  uid: prometheus
  url: http://prometheus-kube-prometheus-prometheus.monitorio:9090/
  access: proxy
  isDefault: true
  jsonData:
    httpMethod: POST
    timeInterval: 30s
- name: "Alertmanager"
  type: alertmanager
  uid: alertmanager
  url: http://prometheus-kube-prometheus-alertmanager.monitorio:9093/
  access: proxy
  jsonData:
    handleGrafanaManagedAlerts: false
```

Observamos que ambos tienen la opción `isDefault` en `true` por lo que Grafana intenta obtener la información de forma predeterminada tanto de Loki como de Prometheus, por ello da fallo ya que solo puede tener uno de los dos configurado de manera predeterminada.

Lo más lógico es poner como predeterminado a Prometheus ya que es el principal recolector de métricas, para ello modificamos el archivo de loki y le ponemos el parámetro **`isDefault`** a **`false`**

```
raulhr@tfgr:~$ sudo cat /var/lib/docker/volumes/minikube/_data/lib/kubelet/pods/453c209d-61b2-4be4-b6ec-afdb4a126ba1/volumes/kubernetes.io-empty-dir/sc-datasources-volume/loki-stack-datasource.yaml
apiVersion: 1
datasources:
- name: Loki
  type: loki
  access: proxy
  url: "http://loki:3100"
  version: 1
  isDefault: false
  jsonData:
    {}
```

Ahora aplicamos los cambios para que el pod de Grafana comience a funcionar de manera correcta

```
kubectl rollout restart deployment prometheus-grafana -n monitorio
```

```
raulhr@tfgr:~$ kubectl rollout restart deployment prometheus-grafana -n monitorio
deployment.apps/prometheus-grafana restarted
```

Y comprobamos que ya funciona correctamente



```
raulhr@tfg:~$ kubectl get pod/prometheus-grafana-6c96d68598-6wwvn -n monitoreo
NAME                                READY   STATUS    RESTARTS   AGE
prometheus-grafana-6c96d68598-6wwvn 3/3     Running   0           65s
```

Para que este cambio sea persistente vamos a modificar el chart usado para instalar Loki

```
helm upgrade loki grafana/loki-stack \
  -n monitoreo \
  --set grafana.datasources."datasources\.yaml".apiVersion=1 \
  --set
grafana.datasources."datasources\.yaml".datasources[0].name=Loki \
  --set
grafana.datasources."datasources\.yaml".datasources[0].type=loki \
  --set
grafana.datasources."datasources\.yaml".datasources[0].url=http://loki
:3100 \
  --set
grafana.datasources."datasources\.yaml".datasources[0].access=proxy \
  --set
grafana.datasources."datasources\.yaml".datasources[0].isDefault=false
```

```
raulhr@tfg:~$ helm upgrade loki grafana/loki-stack \
  -n monitoreo \
  --set grafana.datasources."datasources\.yaml".apiVersion=1 \
  --set grafana.datasources."datasources\.yaml".datasources[0].name=Loki \
  --set grafana.datasources."datasources\.yaml".datasources[0].type=loki \
  --set grafana.datasources."datasources\.yaml".datasources[0].url=http://loki:3100 \
  --set grafana.datasources."datasources\.yaml".datasources[0].access=proxy \
  --set grafana.datasources."datasources\.yaml".datasources[0].isDefault=false
Release "loki" has been upgraded. Happy Helming!
NAME: loki
LAST DEPLOYED: Tue Jun 3 20:19:53 2025
NAMESPACE: monitoreo
STATUS: deployed
REVISION: 2
NOTES:
The Loki stack has been deployed to your cluster. Loki can now be added as a datasource in Grafana.
See http://docs.grafana.org/features/datasources/loki/ for more detail.
```



4.1 Preparación del entorno

Pasamos a la creación del escenario. Lo primero que necesitaremos es una aplicación con un frontend y un backend para poder realizar tanto el monitoreo como la cobertura de código y los tests.

Para ello crearemos un nuevo namespace en kubernetes y ahí lanzaremos la aplicación

```
kubectl create namespace app
```

```
raulhr@tfg:~$ kubectl create namespace app
namespace/app created
```

Una vez creado el namespace procedemos a lanzar la aplicación. Para esta demostración me ha ayudado mi compañero de trabajo **Alejandro Salas** a crear la aplicación ya que es algo secundario al trabajo en sí. Dicha aplicación te solicita un registro y un login que irán a una base de datos externa. La app simula una votación con dos equipos de fútbol, Real Betis Balompié y Sevilla Fútbol Club.

La aplicación cuenta con tests unitarios, los podemos observar en los siguientes ficheros :

4.2 Tests unitarios

En nuestra aplicación podremos encontrar los tests unitarios en los siguientes ficheros :

backend/src/auth/auth.service.spec.ts
backend/src/user/user.service.spec.ts
backend/src/vote/vote.service.spec.ts
frontend/src/App.test.jsx
frontend/src/components/AuthModal.test.jsx
frontend/src/components/Popup.test.jsx



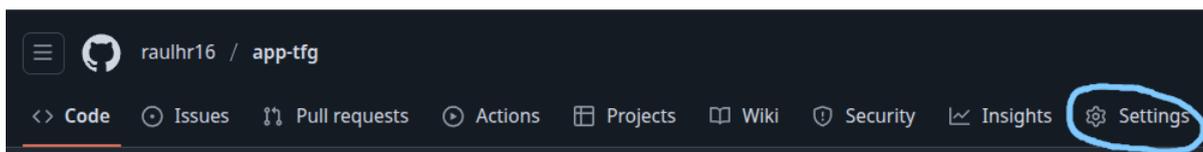
En cada uno de los anteriores ficheros podremos encontrar todos los tests unitarios para el código de nuestra aplicación, estos tests se lanzarán con jest en la ejecución del workflow de github actions y se guardará su porcentaje de cobertura en un archivo que posteriormente analizará sonarqube y nos mostrará el porcentaje de cobertura de código.

Encontramos la app en el siguiente repositorio de github : [App Votos](#)

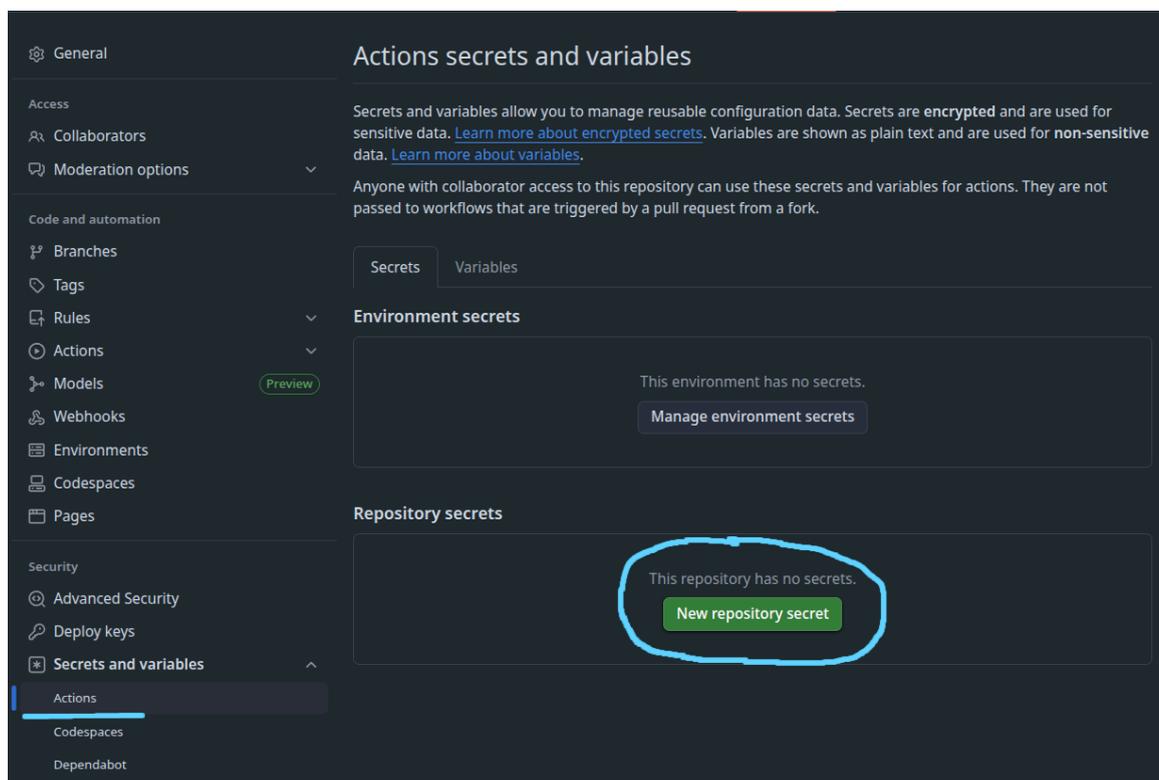
Una vez tenemos la app vamos a tener que seguir unos pasos antes de poder desplegarla y hacer todo lo que queremos con github actions.

Primero debemos crear en el repositorio de GitHub donde tenemos alojada la app una serie de secretos que serán necesarios luego en el despliegue con github actions.

Para ello vamos a Ajustes/Settings dentro del repositorio donde está alojada la app



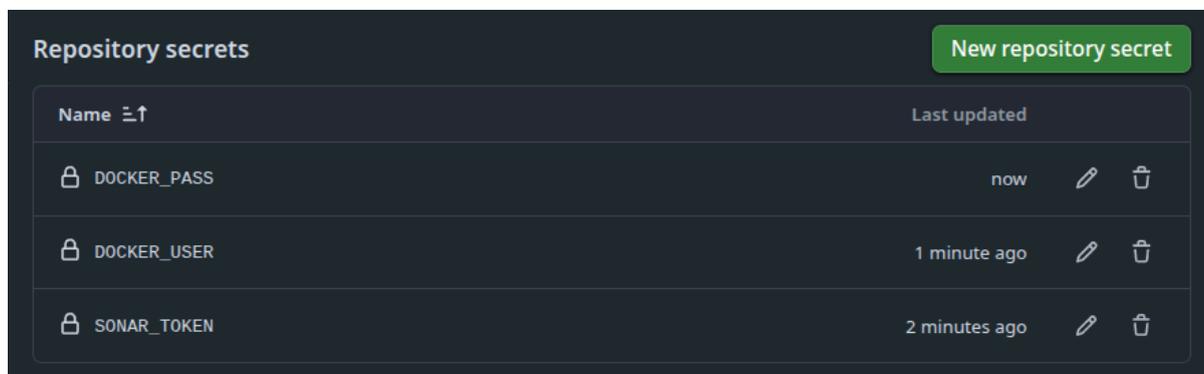
Y en la parte izquierda accedemos al apartado Actions dentro de Secrets and variables





Ahora creamos los secretos necesarios

```
SONAR_TOKEN -> En SonarQube My Account > Security > Generate Token
DOCKER_USER -> Nombre usuario docker (raulhr16)
DOCKER_PASS -> Contraseña de dicho usuario
```



Una vez tenemos los secretos creados en el repositorio de github y antes de seguir con el workflow vamos a configurar Cloudflared para poder acceder a grafana y sonar desde fuera.

Hay que asegurarnos previamente que tenemos configurado los servicios de grafana y sonarqube con la opción NodePort en vez de ClusterIP para poder acceder desde el exterior.

```
raulhr@tfg:~$ kubectl get service prometheus-grafana -n monitoreo
NAME                TYPE        CLUSTER-IP    EXTERNAL-IP    PORT(S)          AGE
prometheus-grafana  NodePort    10.100.59.249 <none>         80:31653/TCP     9d
raulhr@tfg:~$ kubectl get service sonarqube-sonarqube -n sonarqube
NAME                TYPE        CLUSTER-IP    EXTERNAL-IP    PORT(S)          AGE
sonarqube-sonarqube NodePort    10.109.132.205 <none>         9000:30703/TCP   3d23h
```

En caso de no tenerlos configurados de dicha manera, haremos un kubectl edit del servicio para modificarlo.

En mi caso prefiero usar como editor de texto Nano antes que vi por lo que ejecuto el siguiente export

```
export KUBE_EDITOR="nano"
```



Una vez hecho esto ya podemos ejecutar el siguiente comando para editar el servicio que queramos usando nano

```
kubectl edit service (nombre del servicio) -n (nom namespace)
```

Ahora ya podemos exponer tanto sonarqube como grafana a la red externa con los siguiente comandos, primero creamos dos archivos donde se guardarán los logs y la dirección donde estarán alojadas nuestras apps

```
touch /tmp/sonar_tunnel.log
touch /tmp/grafana_tunnel.log
```

```
raulhr@tfgr:~$ touch /tmp/sonar_tunnel.log
raulhr@tfgr:~$ touch /tmp/grafana_tunnel.log
```

Una vez creado los archivos donde se guardarán los logs exponemos las apps

```
nohup cloudflared tunnel --url http://192.168.49.2:30703/
--no-autoupdate > /tmp/sonar_tunnel.log 2>&1 &
```

Y ahora consultamos la url con el siguiente comando

```
head -n 6 /tmp/sonar_tunnel.log
```

```
raulhr@tfgr:~$ head -n 6 /tmp/sonar_tunnel.log
nohup: no se tendrá en cuenta la entrada
2025-06-07T18:17:18Z INF Thank you for trying Cloudflare Tunnel. Doing so, without a Cloudflare account, is a quick way to exper
However, be aware that these account-less Tunnels have no uptime guarantee, are subject to the Cloudflare Online Services Terms
cloudflare.com/website-terms/), and Cloudflare reserves the right to investigate your use of Tunnels for violations of such term
se Tunnels in production you should use a pre-created named tunnel by following: https://developers.cloudflare.com/cloudflare-on
-apps
2025-06-07T18:17:18Z INF Requesting new quick Tunnel on trycloudflare.com...
2025-06-07T18:17:24Z INF +-----+
2025-06-07T18:17:24Z INF | Your quick Tunnel has been created! Visit it at (it may take some time to be reachable): |
2025-06-07T18:17:24Z INF | https://buttons-mlb-corn-peterson.trycloudflare.com |
```



Keep your instance current and get the latest SonarQube Community Build, available now.

There's a new version of SonarQube Server available. Upgrade to SonarQube Server and get access to enterprise features. [Learn More](#)

How do you want to create your project?

Do you want to benefit from all of SonarQube Community Build's features (like repository import and Pull Request decoration)?

Create your project from your favorite DevOps platform.

First, you need to set up a DevOps platform configuration.

- Import from Azure DevOps [Setup](#)
- Import from Bitbucket Cloud [Setup](#)
- Import from Bitbucket Server [Setup](#)
- Import from GitHub [Setup](#)
- Import from GitLab [Setup](#)

Are you just testing or have an advanced use-case?

[Create a local project](#)

Y ya tendríamos acceso a SonarQube desde el exterior.

Para grafana seguimos los mismos pasos

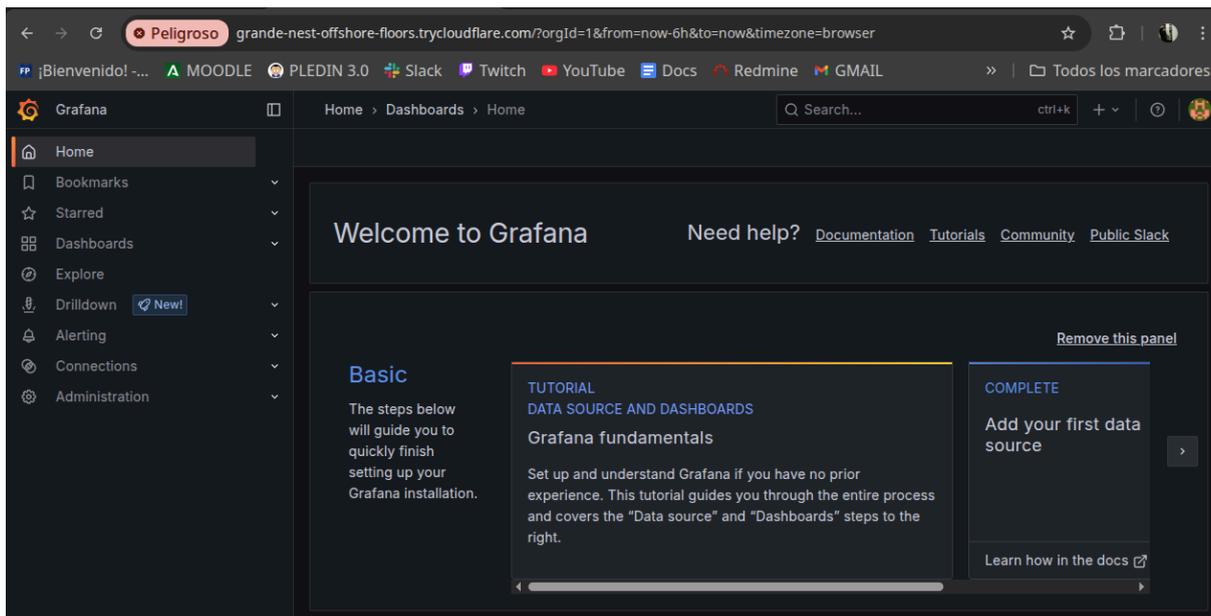
```
nohup cloudflared tunnel --url http://192.168.49.2:31653/
--no-autoupdate > /tmp/grafana_tunnel.log 2>&1 &

head -n 6 /tmp/grafana_tunnel.log
```

```
raulhr@tfg:~$ nohup cloudflared tunnel --url http://192.168.49.2:31653/ --no-autoupdate > /tmp/grafana_tunnel.log 2>&1 &
[2] 25936
raulhr@tfg:~$ head -n 6 /tmp/grafana_tunnel.log
nohup: no se tendrá en cuenta la entrada
2025-06-07T18:21:43Z INF Thank you for trying Cloudflare Tunnel. Doing so, without a Cloudflare account, is a quick way to
However, be aware that these account-less Tunnels have no uptime guarantee, are subject to the Cloudflare Online Services
cloudflare.com/website-terms/), and Cloudflare reserves the right to investigate your use of Tunnels for violations of such
se Tunnels in production you should use a pre-created named tunnel by following: https://developers.cloudflare.com/cloudfla
-apps
2025-06-07T18:21:43Z INF Requesting new quick Tunnel on trycloudflare.com...
2025-06-07T18:21:47Z INF +-----+
2025-06-07T18:21:47Z INF | Your quick Tunnel has been created! Visit it at (it may take some time to be reachable): |
2025-06-07T18:21:47Z INF | https://grande-nest-offshore-floors.trycloudflare.com |
raulhr@tfg:~$
```

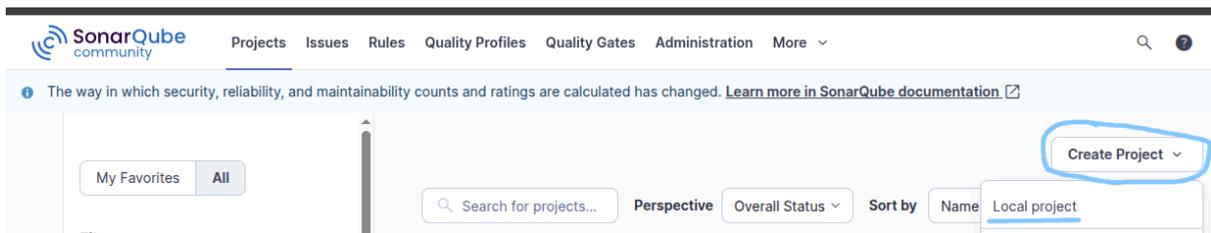


Y también tendremos ya acceso a grafana desde el exterior



Una vez podemos acceder a sonarqube y grafana desde el exterior, debemos crear un proyecto en sonarqube donde subamos el resultado de la cobertura a nuestra app, para ello una vez logueados en la web haremos lo siguiente:

En la página principal de SonarQube una vez conectado con la cuenta de administrador pulsaremos en Create Project > Local project





Ponemos el nombre que queramos y le damos a next

1 of 2

Create a local project

Project display name* ⓘ

Project key* ⓘ

Main branch name*

The name of your project's default branch [Learn More](#) ⓘ

Pulsamos sobre Use the global setting y pulsamos en Create project

2 of 2 ×

Set up project for Clean as You Code

The new code definition sets which part of your code will be considered new code. This helps you focus attention on the most recent changes to your project, enabling you to follow the Clean as You Code methodology. Learn more: [Defining New Code](#) ⓘ

Choose the baseline for new code for this project

Use the global setting

Previous version
Any code that has changed since the previous version is considered new code.
Recommended for projects following regular versions or releases.

Y ya tendríamos el proyecto creado en Sonar

The screenshot shows the SonarQube community interface. At the top, there is a navigation menu with links for Projects, Issues, Rules, Quality Profiles, Quality Gates, Administration, and More. Below the menu, the breadcrumb path is 'app-back-raulhr / main'. Underneath, there are tabs for Overview, Issues, Security Hotspots, Measures, Code, and Activity. The main heading is 'Analysis Method', followed by a sub-heading: 'Use this page to manage and set-up the way your analyses are performed.'

Ahora repetimos los mismos pasos para crear otro proyecto para el frontend.



Una vez creados ambos proyectos comenzamos con la creación del workflow de github actions para automatizarlo todo, para ello clonamos la aplicación en un repositorio local usando el siguiente comando

```
git clone git@github.com:raulhr16/app-tfg.git
```

Y ya lo tendríamos, ahora pasamos a crear el archivo de configuración del workflow, para ello crearemos la carpeta `.github/workflows` y dentro de ella el archivo `main.yml`

```
mkdir -p .github/workflows  
touch .github/workflows/main.yml
```

```
raulhr@tfg:~/app-votos$ mkdir -p .github/workflows  
raulhr@tfg:~/app-votos$ touch .github/workflows/main.yml
```



4.3 Workflow Github Actions

Una vez hecho esto creamos el flow que queremos, voy a ir explicando paso por paso que hace dicho flow.

```
name: CI/CD Pipeline

on:
  push:
    branches: [ "main" ]
  pull_request:
    branches: [ "main" ]

env:
  SONAR_HOST_URL: (url de sonar)
  SONAR_TOKEN: ${ secrets.SONAR_TOKEN }
  DOCKER_USER: ${ secrets.DOCKER_USER }
  DOCKER_PASS: ${ secrets.DOCKER_PASS }
  BACKEND_IMAGE: raulhr16/backend
  FRONTEND_IMAGE: raulhr16/frontend
```

Lo primero que hemos hecho ha sido especificar el nombre de la pipeline, indicamos que se ejecute cuando se hace un push en la rama main o en una pull request y especificamos las variables de entorno.

```
jobs:
  test-y-analisis:
    runs-on: ubuntu-latest

    steps:
      - name: Clonando repositorio
        uses: actions/checkout@v3

      - name: Instalando node
        uses: actions/setup-node@v3
        with:
          node-version: 20

      - name: Instalando dependencias de backend
        run: npm ci
        working-directory: backend
```



- name: Lanzando tests de backend
run: npm test -- --coverage
working-directory: backend
- name: Guardar cobertura backend
uses: actions/upload-artifact@v4
with:
 - name: backend-coverage
 - path: backend/coverage
- name: Instalando dependencias de frontend
run: npm ci
working-directory: frontend
- name: Lanzando tests de frontend
run: npm test -- --coverage
working-directory: frontend
- name: Guardar cobertura frontend
uses: actions/upload-artifact@v4
with:
 - name: frontend-coverage
 - path: frontend/coverage

Comenzamos con los pasos que ejecuta la pipeline, el primero es el test y análisis, en este paso vamos a clonar el repositorio en el agente, instalamos la versión 20 de node que es la que usa nuestra app, instalamos las dependencias y comenzamos a lanzar los tests. Los tests hay que lanzarlos con la opción `--coverage` para que cree un archivo con el resultado de la cobertura y así luego sonar podrá leer dicho archivo. Una vez creado dicho fichero hay que guardarlo como un artefacto para que luego en el siguiente paso el agente que se ejecute pueda descargarse el informe.



```
analysis-frontend:
  needs: test-y-analysis
  runs-on: ubuntu-latest
  steps:
    - name: Clonando repositorio
      uses: actions/checkout@v3

    - name: Descargar cobertura frontend
      uses: actions/download-artifact@v4
      with:
        name: frontend-coverage
        path: frontend/coverage

    - name: Escanear frontend
      uses: SonarSource/sonarcloud-github-action@master
      with:
        args: >
          -Dsonar.host.url=${{ env.SONAR_HOST_URL }}
          -Dsonar.token=${{ env.SONAR_TOKEN }}
          -Dsonar.projectBaseDir=frontend

    - name: Esperar quality gate frontend
      uses: warchant/setup-sonar-scanner@v7
      with:
        sonar-token: ${{ env.SONAR_TOKEN }}
```

En este paso analizamos la cobertura del frontend, añadimos la condición de que el primer paso tiene que salir en verde para que se ejecute este paso y luego la configuración de sonar necesaria.



```
analysis-backend:
  needs: test-y-analysis
  runs-on: ubuntu-latest
  steps:
    - name: Clonando repositorio
      uses: actions/checkout@v3

    - name: Descargar cobertura backend
      uses: actions/download-artifact@v4
      with:
        name: backend-coverage
        path: backend/coverage

    - name: Escaneando el backend
      uses: SonarSource/sonarcloud-github-action@master
      with:
        args: >
          -Dsonar.host.url=${{ env.SONAR_HOST_URL }}
          -Dsonar.token=${{ env.SONAR_TOKEN }}
          -Dsonar.projectBaseDir=backend

    - name: Esperar quality gate backend
      uses: warchant/setup-sonar-scanner@v7
      with:
        sonar-token: ${{ env.SONAR_TOKEN }}
```

Lo mismo que hemos explicado anteriormente pero en este caso para el backend.

```
quality-gate:
  needs: [analysis-frontend, analysis-backend]
  runs-on: ubuntu-latest
  steps:
    - run: echo "Ambos quality gates completados"
```

Con este paso nos aseguramos de que pasen bien ambos análisis.



```
build-and-push:
  needs: quality-gate
  runs-on: ubuntu-latest

  steps:
    - name: Clonando repositorio
      uses: actions/checkout@v3

    - name: Iniciando sesión en docker
      run: echo "${{ env.DOCKER_PASS }}" | docker login -u "${{
env.DOCKER_USER }}" --password-stdin

    - name: Buildeando y pusheando la última versión del backend
      run: |
        docker build -t $BACKEND_IMAGE:$GITHUB_RUN_NUMBER
        ./backend
        docker push $BACKEND_IMAGE:$GITHUB_RUN_NUMBER

    - name: Buildeando y pusheando la última versión del
frontend
      run: |
        docker build -t $FRONTEND_IMAGE:$GITHUB_RUN_NUMBER
        ./frontend
        docker push $FRONTEND_IMAGE:$GITHUB_RUN_NUMBER
```

Y en este último paso creamos la imagen de nuestra app después de habernos asegurado que pasa bien la cobertura con sonar y subimos la nueva versión de la app a docker-hub.

The screenshot displays the GitHub Actions interface for a repository named 'raulhr16 / app-votos'. The top navigation bar includes 'Code', 'Pull requests', 'Actions', 'Projects', 'Wiki', 'Security', 'Insights', and 'Settings'. The main content area shows a 'CI/CD Pipeline' with a green checkmark and the title 'fixed coverage exceptions #12'. A 'Re-run all jobs' button is visible. Below this, a 'Summary' section indicates the pipeline was 'Triggered via push 1 hour ago' and has a 'Success' status. A table provides details: 'raulhr16 pushed' (commit ad069a0 on main) with a 'Total duration' of '3m 7s' and '2' artifacts. The 'Jobs' list on the left shows five completed jobs: 'test-y-analysis', 'analysis-frontend', 'analysis-backend', 'quality-gate', and 'build-and-push'. The 'Run details' section shows a workflow diagram for 'main.yml' triggered on 'push', with a sequence of jobs: 'test-y-analysis' (40s), 'analysis-frontend' (42s), 'analysis-backend' (49s), 'quality-gate' (2s), and 'build-and-push' (1m 21s).



Y comprobamos que han pasado bien ambos proyectos de sonar, tanto del frontend como del backend.

Frontend

The screenshot shows the SonarQube interface for the 'app-front-raulhr' project. The main quality gate status is 'Passed' with a green checkmark. A warning message states: 'The last analysis has warnings. See details'. The overall code quality is 'A'. The metrics are as follows:

Metric	Value	Quality
Security	0 Open issues	A
Reliability	6 Open issues	B
Maintainability	5 Open issues	A
Accepted issues	0 (Valid issues that were not fixed)	A
Coverage	97.0% (On 69 lines to cover)	A
Duplications	0.0% (On 420 lines)	A

Backend

The screenshot shows the SonarQube interface for the 'app-back-raulhr' project. The main quality gate status is 'Passed' with a green checkmark. A warning message states: 'The last analysis has warnings. See details'. The overall code quality is 'A'. The metrics are as follows:

Metric	Value	Quality
Security	0 Open issues	A
Reliability	0 Open issues	A
Maintainability	8 Open issues	A
Accepted issues	0 (Valid issues that were not fixed)	A
Coverage	100% (On 52 lines to cover)	A
Duplications	0.0% (On 681 lines)	A



Y comprobamos que se ha subido la última versión de la imagen a docker-hub

Frontend

raulhr16/frontend 🔒
 Last pushed about 1 hour ago · Repository size: 23.1 MB

Add a description ✎ 🔒
 Add a category ✎ 🔒

Docker commands Public view
 To push a new tag to this repository:

```
docker push raulhr16/frontend:tag name
```

General | Tags | Image Management BETA | Collaborators | Webhooks | Settings

Tags 🔒 DOCKER SCOUT INACTIVE [Activate](#)
 This repository contains 4 tag(s).

Tag	OS	Type	Pulled	Pushed
12		Image	less than 1 day	about 1 hour

buildcloud
 Build with **Docker Build Cloud**
 Accelerate image build times with access to cloud-based builders and shared cache.

Backend

raulhr16/backend 🔒
 Last pushed about 1 hour ago · Repository size: 326.8 MB

Add a description ✎ 🔒
 Add a category ✎ 🔒

Docker commands Public view
 To push a new tag to this repository:

```
docker push raulhr16/backend:tag name
```

General | Tags | Image Management BETA | Collaborators | Webhooks | Settings

Tags 🔒 DOCKER SCOUT INACTIVE [Activate](#)
 This repository contains 4 tag(s).

Tag	OS	Type	Pulled	Pushed
12		Image	less than 1 day	about 1 hour

buildcloud
 Build with **Docker Build Cloud**
 Accelerate image build times with access to cloud-based builders and shared cache.

Vemos que efectivamente ambas imágenes se han subido de forma correcta y con el tag de la run de github actions.



4.4 Despliegue de la App

Ahora comenzamos con el despliegue de la aplicación en local y con la conexión con grafana, prometheus y loki.

Para el despliegue de la app en local usaremos kubernetes, crearemos los siguiente ficheros de configuración para poder desplegarla

secreto-db.yaml

```
apiVersion: v1
kind: Secret
metadata:
  name: secreto-db
  namespace: app
type: Opaque
data:
  POSTGRES_DB: dGZn
  POSTGRES_USER: cmF1bGhy
  POSTGRES_PASSWORD: cmF1bGhy
```

postgres-deployment.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: postgres
  namespace: app
spec:
  replicas: 1
  selector:
    matchLabels:
      app: postgres
  template:
    metadata:
      labels:
        app: postgres
    spec:
      containers:
        - name: postgres
          image: postgres:15
          ports:
```



```
- containerPort: 5432
env:
  - name: POSTGRES_DB
    valueFrom:
      secretKeyRef:
        name: secreto-db
        key: POSTGRES_DB
  - name: POSTGRES_USER
    valueFrom:
      secretKeyRef:
        name: secreto-db
        key: POSTGRES_USER
  - name: POSTGRES_PASSWORD
    valueFrom:
      secretKeyRef:
        name: secreto-db
        key: POSTGRES_PASSWORD
volumeMounts:
  - name: postgres-data
    mountPath: /var/lib/postgresql/data
volumes:
  - name: postgres-data
    emptyDir: {}
```

postgres-service.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: postgres
  namespace: app
spec:
  selector:
    app: postgres
  ports:
    - port: 5432
      targetPort: 5432
```



backend-deployment.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: backend
  namespace: app
spec:
  replicas: 1
  selector:
    matchLabels:
      app: backend
  template:
    metadata:
      labels:
        app: backend
    spec:
      containers:
        - name: backend
          image: raulhr16/backend:12
          ports:
            - containerPort: 3000
          env:
            - name: DB_HOST
              value: postgres
            - name: DB_PORT
              value: "5432"
            - name: DB_USER
              valueFrom:
                secretKeyRef:
                  name: secreto-db
                  key: POSTGRES_USER
            - name: DB_PASS
              valueFrom:
                secretKeyRef:
                  name: secreto-db
                  key: POSTGRES_PASSWORD
            - name: DB_NAME
              valueFrom:
                secretKeyRef:
                  name: secreto-db
                  key: POSTGRES_DB
```



backend-service.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: backend
  namespace: app
spec:
  selector:
    app: backend
  ports:
    - port: 3000
      targetPort: 3000
```

frontend-deployment.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: frontend
  namespace: app
spec:
  replicas: 1
  selector:
    matchLabels:
      app: frontend
  template:
    metadata:
      labels:
        app: frontend
    spec:
      containers:
        - name: frontend
          image: raulhr16/frontend:12
          ports:
            - containerPort: 80
```



frontend-service.yaml

```

apiVersion: v1
kind: Service
metadata:
  name: frontend
  namespace: app
spec:
  type: NodePort
  selector:
    app: frontend
  ports:
    - port: 80
      targetPort: 80
      nodePort: 30080

```

Una vez creado todos los archivos de configuración los desplegamos usando

```
kubectl apply -f .
```

```

raulhr@tfg:~/app-k8s$ kubectl apply -f .
deployment.apps/backend created
service/backend created
deployment.apps/frontend created
service/frontend created
deployment.apps/postgres created
service/postgres created
secret/secreto-db created

```

Y comprobamos que está funcionando bien

```

raulhr@tfg:~/app-k8s$ kubectl get all -n app
NAME                                READY   STATUS    RESTARTS   AGE
pod/backend-8476fd7847-fxzkj        1/1     Running   1 (55s ago) 2m14s
pod/frontend-69f855f85c-rsq8m      1/1     Running   0           2m14s
pod/postgres-7d979b96c8-4gwxd      1/1     Running   0           2m13s

NAME                                TYPE           CLUSTER-IP   EXTERNAL-IP   PORT(S)          AGE
service/backend                     ClusterIP      10.107.60.229 <none>        3000/TCP        2m14s
service/frontend                     NodePort       10.106.226.53 <none>        80:30080/TCP    2m14s
service/postgres                     ClusterIP      10.102.17.129 <none>        5432/TCP        2m13s

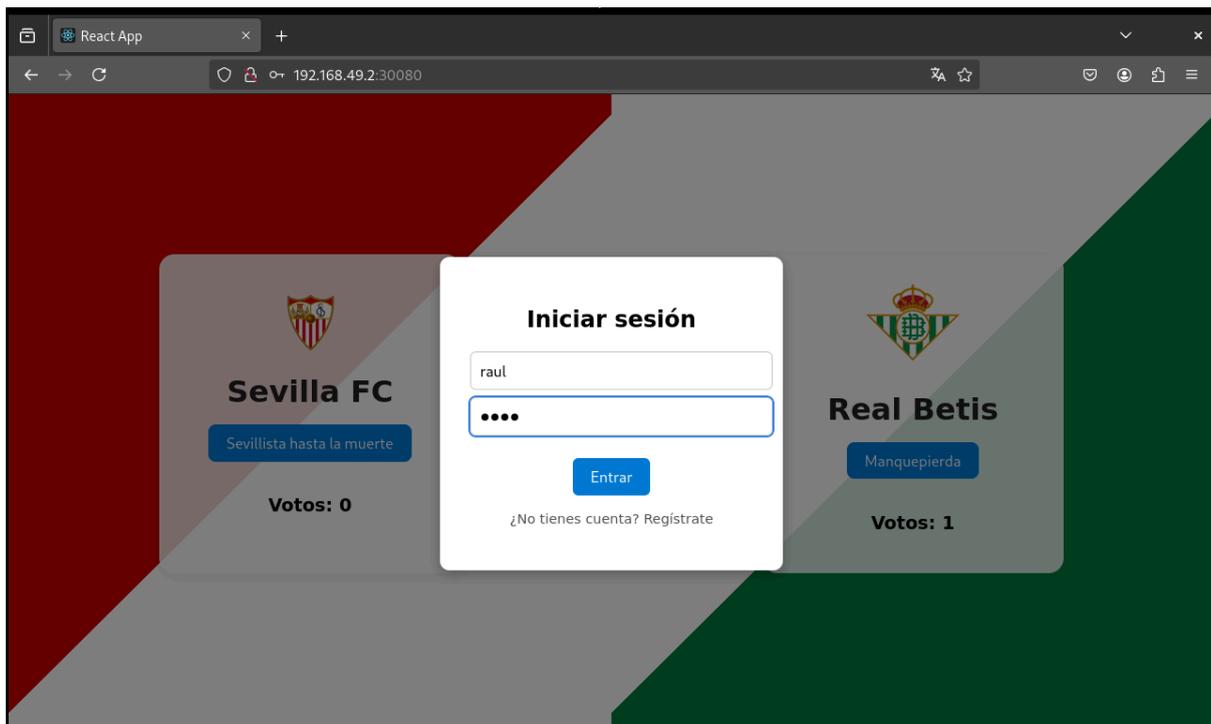
NAME                                READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/backend              1/1     1             1           2m14s
deployment.apps/frontend              1/1     1             1           2m14s
deployment.apps/postgres              1/1     1             1           2m13s

NAME                                DESIRED   CURRENT   READY   AGE
replicaset.apps/backend-8476fd7847  1         1         1       2m14s
replicaset.apps/frontend-69f855f85c  1         1         1       2m14s
replicaset.apps/postgres-7d979b96c8  1         1         1       2m13s

```



Ahora vamos a probar la app

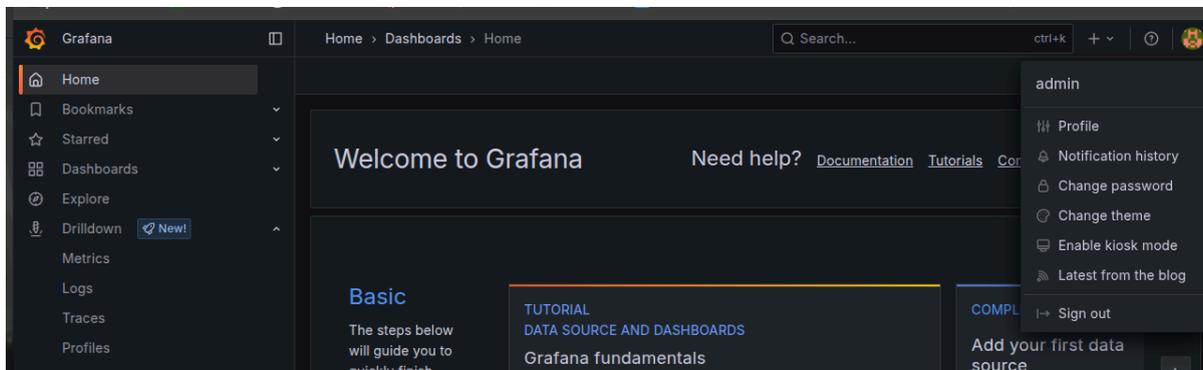


Y efectivamente funciona bien.

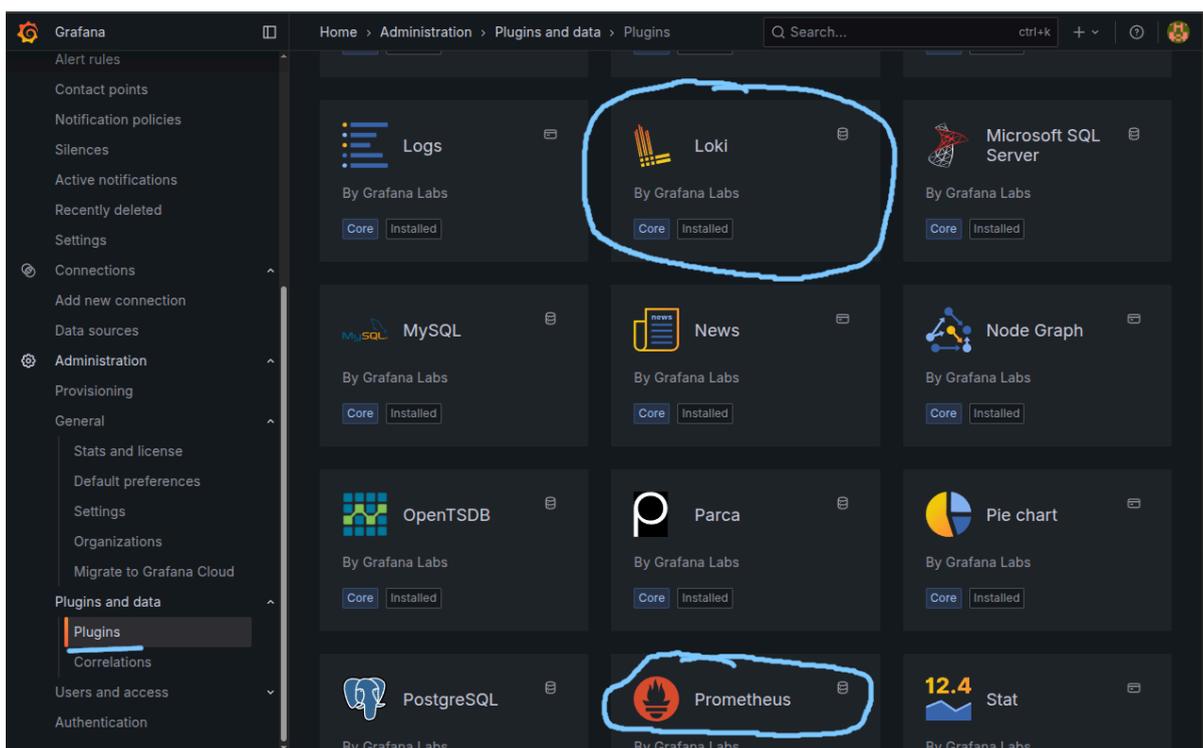


4.5 Grafana, prometheus y loki

Ya podemos comenzar con la configuración de los observadores. Para ello lo primero que haremos será iniciar sesión como administrador en Grafana.



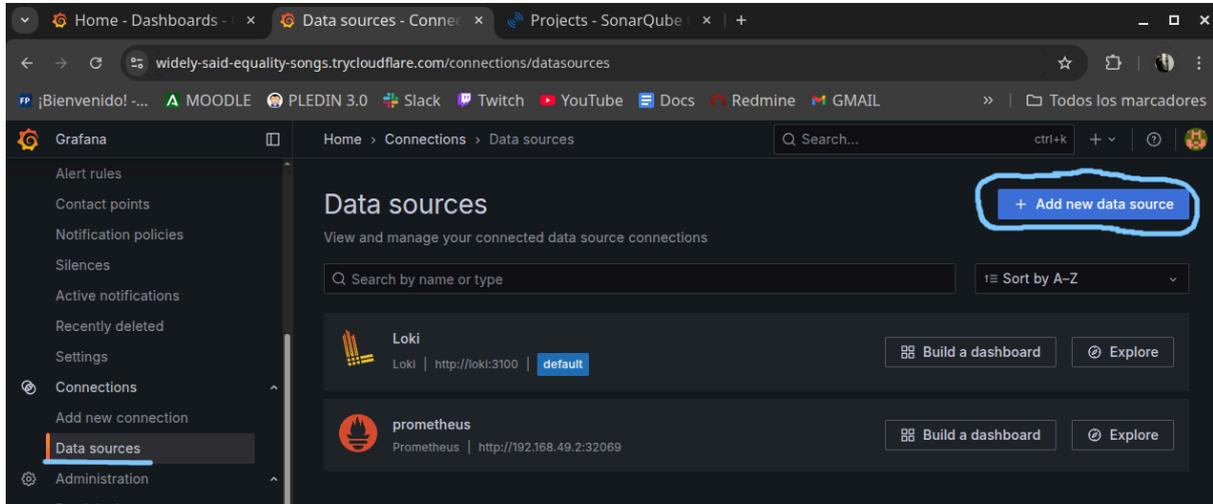
Una vez dentro de grafana vamos a asegurarnos de que tenemos los plugins necesarios para usar loki y prometheus instalados



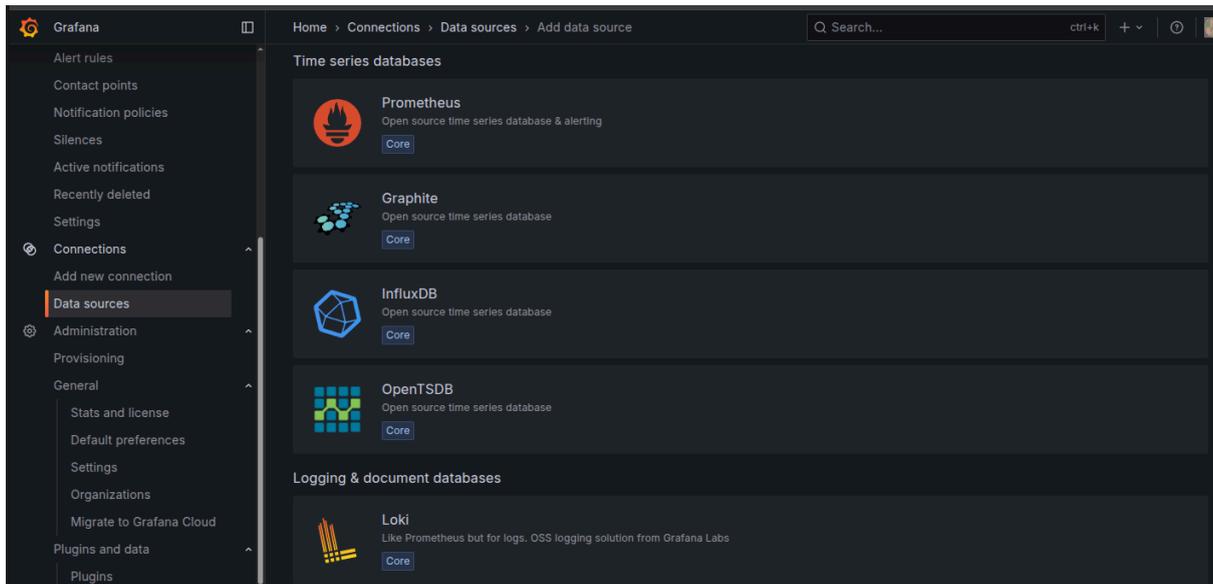
Si no tuviéramos alguno de esos dos nos lo instalaremos introduciendo el nombre en el buscador y luego pulsando el botón de instalar.



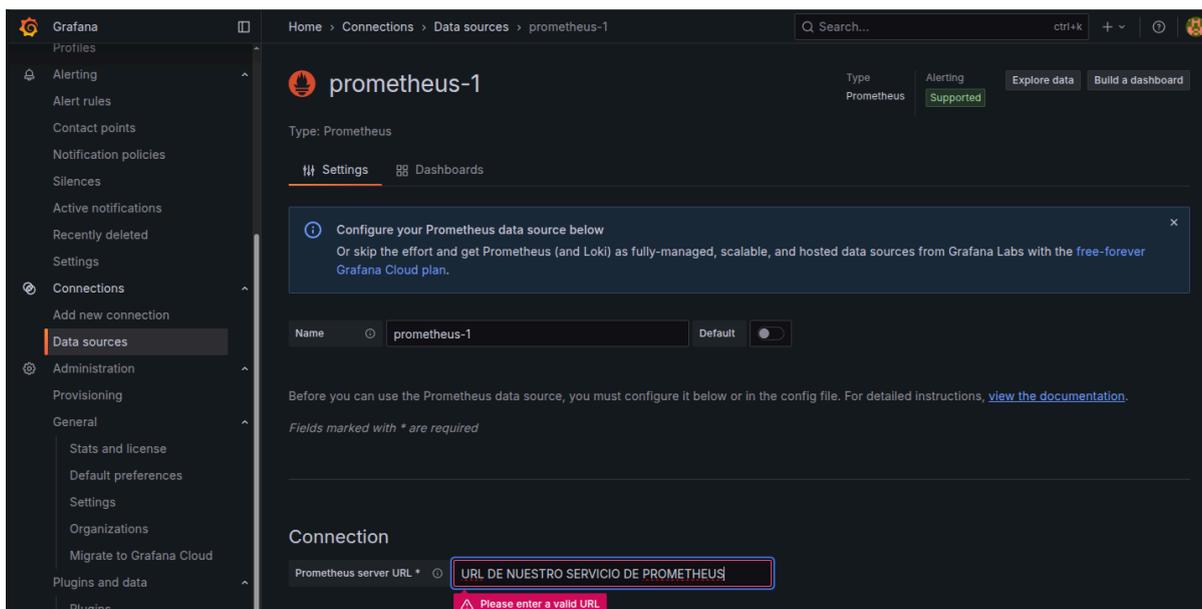
Una vez tenemos ambos plugins instalados pasamos a añadir en grafana ambas fuentes de datos, para ello iremos a la pestaña Data sources > add new datasource



Una vez pulsamos dicha opción nos saldrá el siguiente menú



Aquí seleccionaremos la fuente de datos que queremos añadir, recordad que tenemos que añadir tanto Loki como Prometheus, como ejemplo usaré Prometheus



Rellenamos la URL para que Grafana se pueda conectar con prometheus y le daríamos a ‘save & test’

Para saber qué URL hay que poner ahí podemos comprobar el servicio de prometheus en nuestro kubernetes, en mi caso es el siguiente

```
raulhr@tfg:~/app-k8s$ kubectl get service prometheus-kube-prometheus-prometheus -n monitoreo
NAME                                TYPE        CLUSTER-IP      EXTERNAL-IP      PORT(S)          AGE
prometheus-kube-prometheus-prometheus NodePort    10.100.84.199   <none>           9090:32069/TCP,8080:32225/TCP 11d
```

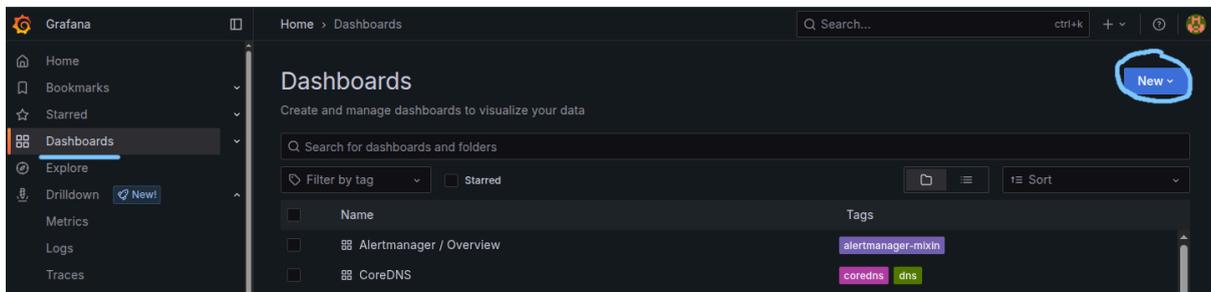
Mi servicio es de tipo NodePort por si necesito entrar en prometheus para cambiar algo, debido a esto yo me conectaré a la siguiente url : <http://192.168.49.2:32069>

La ip que sale es la ip de minikube. Una vez añadido eso ya tendríamos Prometheus y Loki en Grafana con capacidad de mirar la información de la app que tenemos en el namespace App.



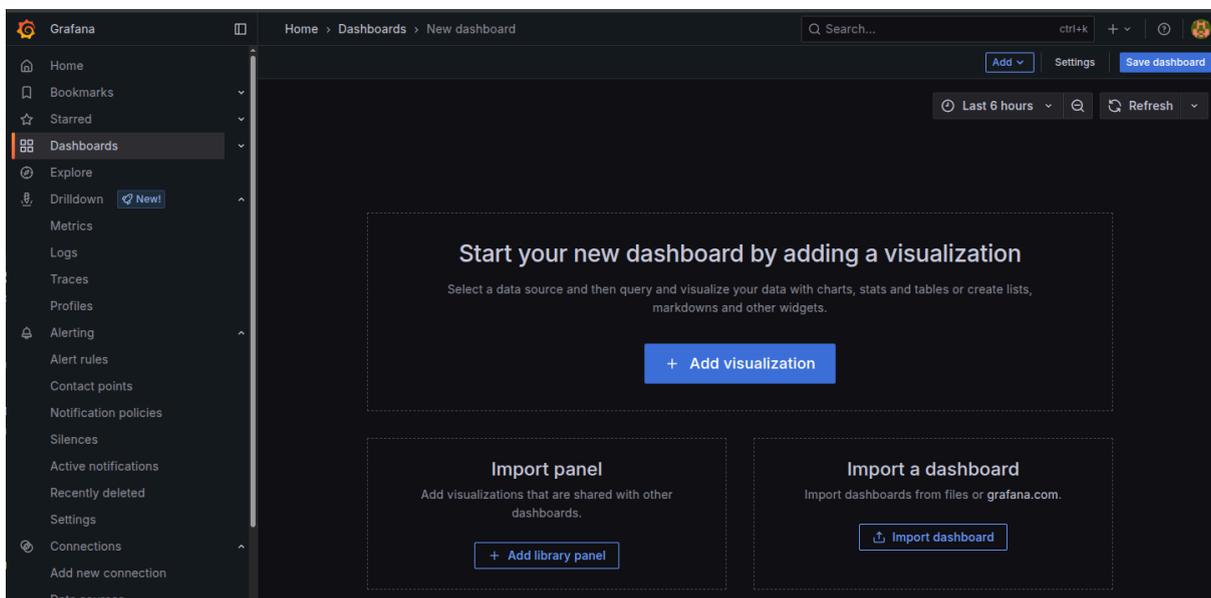
Vamos a comenzar con las pruebas en Grafana para comprobar el correcto funcionamiento del stack.

Primero iremos a la pestaña de dashboards,



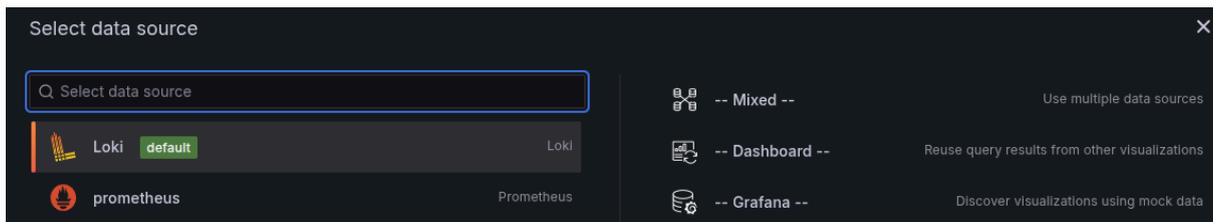
En Grafana hay varios dashboards por defecto y también hay dashboards de la comunidad que podemos usar mediante el id de dichos dashboards, pero en mi caso vamos a crear uno propio, para ello pulsaremos en el botón de new que aparece en la esquina superior derecha.

Una vez ahí le daremos a new dashboard y nos traerá a la siguiente ventana

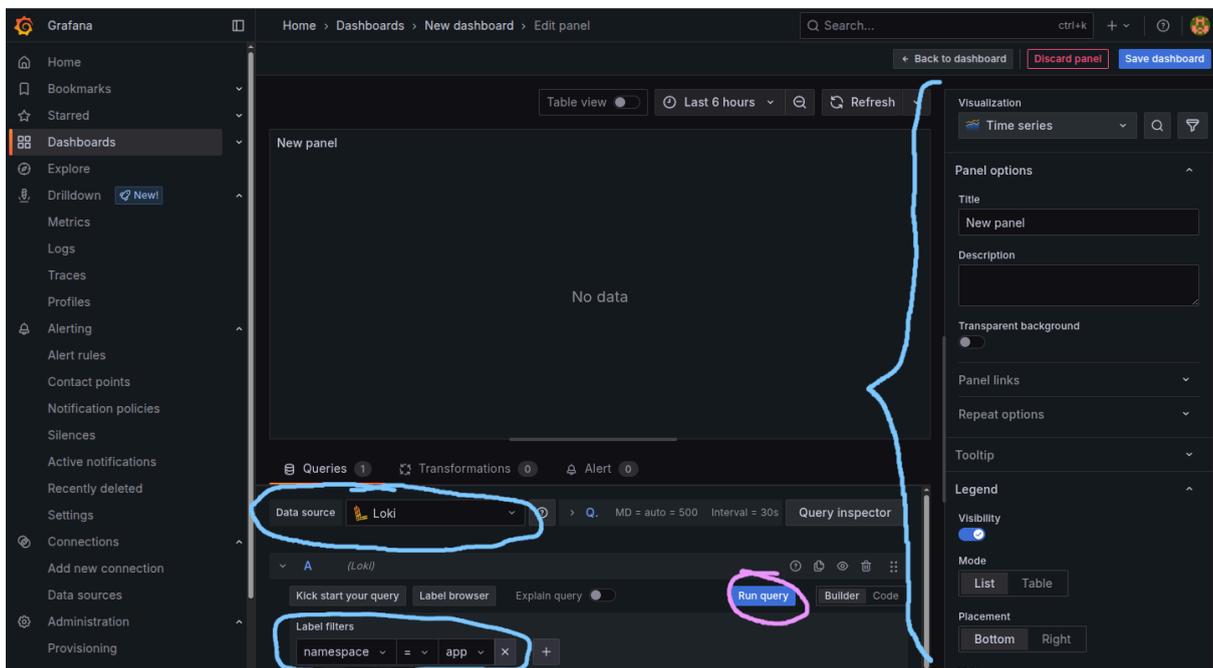




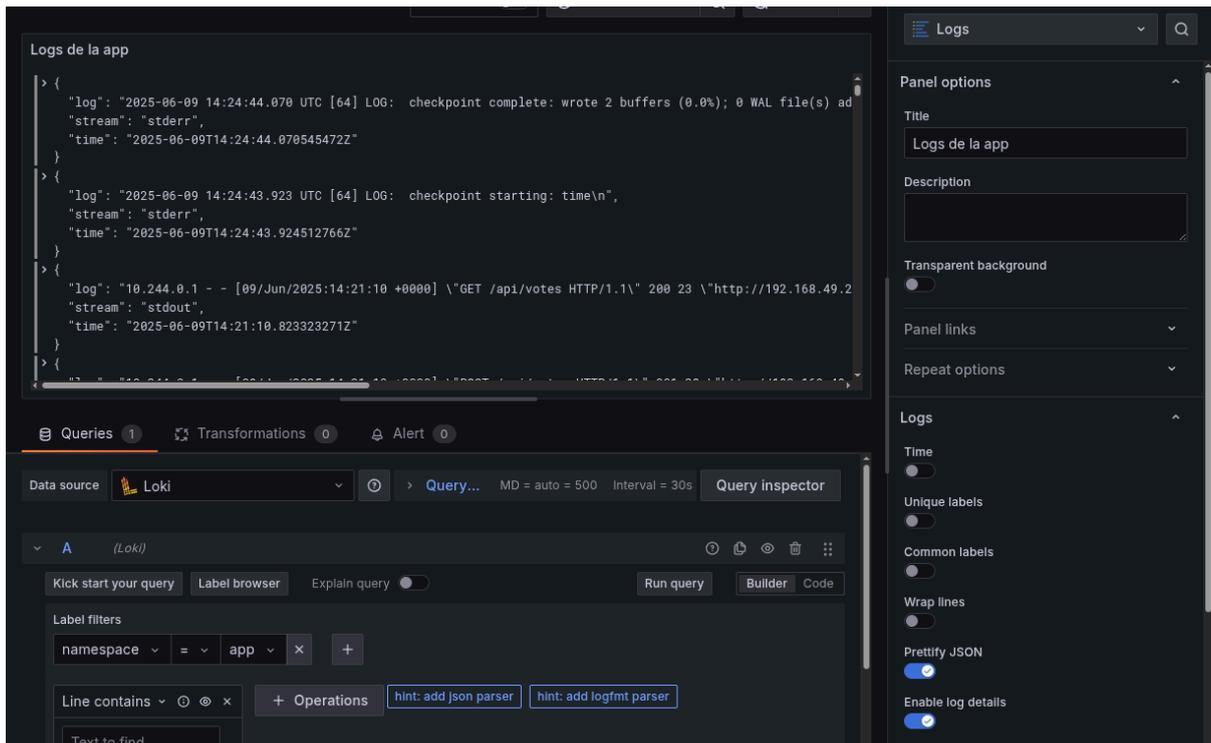
Aquí le daremos a Add visualization



Y te pedirá que introduzcas una data source para este panel, en mi caso vamos a empezar con Loki para que el primer panel sea de logs, una vez seleccionado Loki nos aparecerá la siguiente ventana

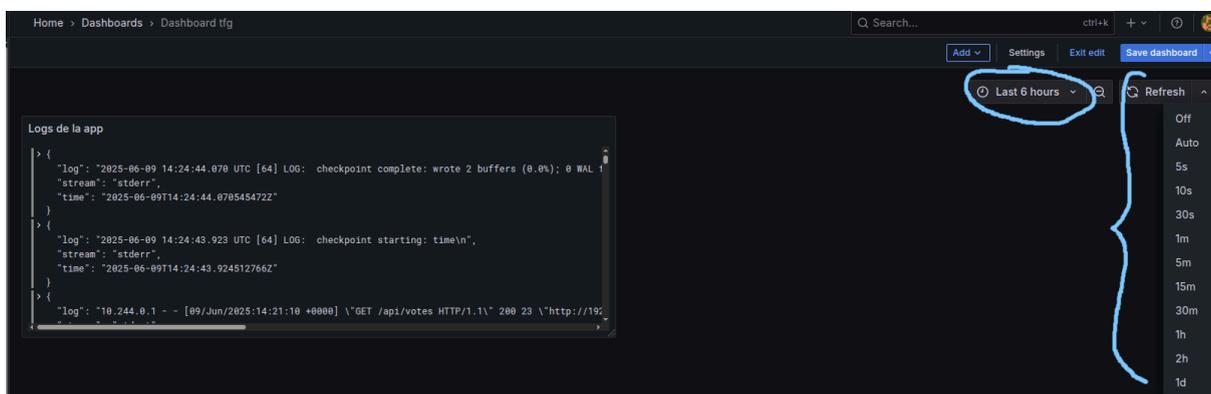


En esta ventana ya podremos editar todo lo que necesitemos de Loki, en mi caso quiero que me muestre los logs del namespace app, y editamos las opciones de la derecha para dejar el panel a nuestro gusto



Queremos que me aparezcan los logs con el pretty json activado para que sean más legibles.

Guardamos los datos y ya tendríamos nuestro primer dashboard con nuestro primer panel.



También podremos elegir el rango de tiempo del cual queremos obtener información y cada cuanto queremos que se actualice dicha información.



Una vez entendiendo el funcionamiento de Loki podemos usar filtros para obtener lo que queremos en la query, por ejemplo:

Ver errores de backend

```
{namespace="app"} |~ `(?!i)ERROR`
```

Ver advertencias

```
{namespace="app"} |~ `(?!i)WARN`
```

Logs que contengan una palabra específica

```
{namespace="app"} |= "login"
```

Logs del frontend

```
{namespace="app", container="frontend"}
```

The screenshot shows a Loki query interface with four panels:

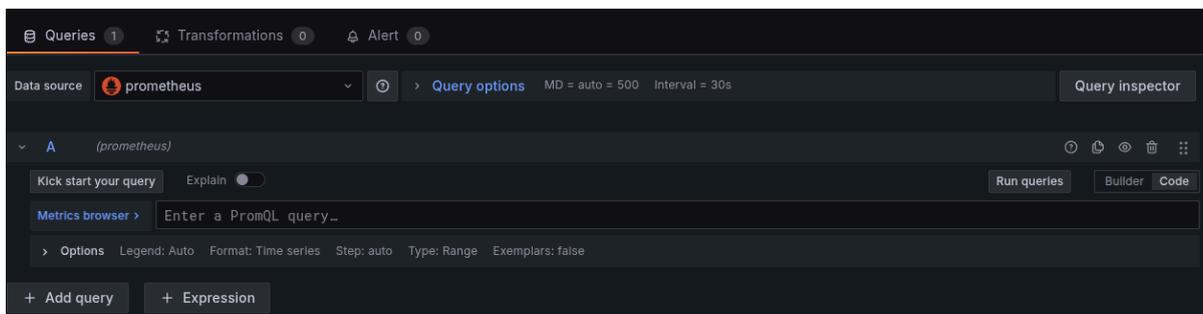
- Logs frontend:** Displays a list of log entries for the 'frontend' container, including timestamps, log levels, and message snippets.
- Errores en la app:** Shows log entries filtered for errors, with a table of labels, time, line numbers, and IDs.
- Contiene login:** Shows log entries filtered for the word 'login', with a table of labels, time, line numbers, and IDs.
- Warning:** Shows log entries filtered for warnings, with a table of labels, time, line numbers, and IDs.



Ahora vamos a ver cómo podemos crear un panel usando prometheus. Pulsamos en add > visualization



Y esta vez cogemos prometheus como data source



Y ya podemos poner las query que queramos, aquí están algunas de las más útiles

Porcentaje de CPU por pod

```
100 * rate(container_cpu_usage_seconds_total{namespace="app"}[1m])
```

Uso de memoria por pod

```
container_memory_usage_bytes{namespace="app"}
```

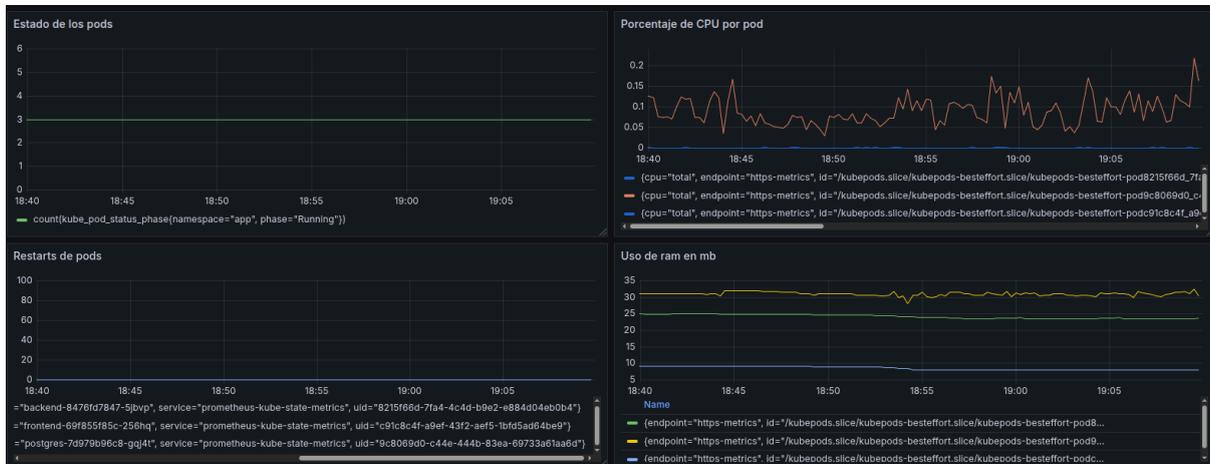
Restarts de pods

```
increase(kube_pod_container_status_restarts_total{namespace="app"} [5m])
```



Estado de los pods

```
count(kube_pod_status_phase{namespace="app", phase="Running"})
```



Y ya tendríamos configurado Grafana con logs de Loki, métricas de Prometheus bajo un cluster de kubernetes donde se aloja una app desplegada, dicha app se puede actualizar de manera constante gracias a que obtiene el frontend y el backend de un despliegue continuo realizado con github actions, en dicho despliegue se comprueban los tests unitarios con jest y la cobertura de código con sonarqube.

4.6 Demostraciones

Video demostración del correcto funcionamiento del proyecto

[demo1 tfg Raúl Herrera Ruiz](#)

Repositorio de GitHub con los documentos

[Repositorio con aplicación y archivos de despliegue k8s de dicha app](#)



5. Conclusiones y propuestas para seguir trabajando sobre el tema

Este proyecto me ha resultado muy útil para entender mejor herramientas que uso a diario en el trabajo. Una de las más destacadas ha sido SonarQube, que nunca había utilizado ni conocido hasta ahora. Me ha sorprendido lo útil que es para detectar problemas en el código relacionados con la seguridad y la cobertura, aspectos que normalmente no se revisan con mucho detalle.

También me ha parecido muy buena idea el centrarme un poco más en Github Actions y el despliegue continuo ya que pienso que en la actualidad tiene mucha importancia por la gran versatilidad que tiene y en la cantidad de sitios que se está empezando a usar o que ya lleva tiempo siendo usado. Además ahora vamos a comenzar una migración de Bitbucket y Jenkins a Github y Github Actions en mi trabajo.

Otro aspecto importante ha sido el uso de logs y métricas. Gracias a ellos es posible detectar errores y prevenir caídas en la aplicación antes de que ocurran, lo cual es muy útil para mantener la estabilidad del sistema.

Para seguir trabajando sobre este tema profundizaría un poco más en las métricas y en los tests. En especial, los tests end-to-end y de accesibilidad, que no he llegado a usar y creo que pueden aportar mucho. Pero en general, estoy bastante satisfecho con el trabajo realizado.



6. Bibliografía

<https://github.com/josedom24/> (Para parte de la teoría)

<https://kubernetes.io/es/docs/home/> Teoría y configuración de Kubernetes.

<https://prometheus.io/docs/introduction/overview/> Teoría y configuración de Prometheus.

<https://grafana.com/docs/> Teoría y configuración de Grafana.

<https://grafana.com/docs/loki/latest/?pg=oss-loki&plcmt=quick-links> Teoría y configuración de Loki.

https://es.wikipedia.org/wiki/Prueba_unitaria Test unitarios

<https://jestjs.io/es-ES/docs/getting-started> Teoría y configuración de Jest.

<https://docs.sonarsource.com/sonarqube-server/latest/> Teoría y configuración de Sonarqube.

<https://docs.github.com/es/actions> Teoría y configuración de Github Actions.

<https://helm.sh/docs/> Instalación del software dentro de kubernetes.

<https://minikube.sigs.k8s.io/docs/> Instalación de minikube

<https://developers.cloudflare.com/cloudflare-one/connections/connect-networks/>
Configuración e instalación cloudflared.