

# INTEGRANDO HARBOR EN GITLAB CI/CD

Arantxa Fernández Morató



**Proyecto Integrado**

**2º Grado Superior en Administración de  
Sistemas Informáticos en Red**

**Diciembre 2023**

**IES Gonzalo Nazareno**

## ÍNDICE

1	Descripción del proyecto.....	2
1.1	Tecnologías que se van a utilizar.....	2
1.2	Resultados que se esperan obtener.....	2
2	Fundamentos.....	3
2.1	Introducción a Harbor.....	3
2.2	Introducción a Gitlab.....	4
2.3	¿Por qué se ha elegido este proyecto?.....	6
3	Preparación del escenario en Openstack.....	7
3.1	Creación de instancias.....	7
4	Instalación y configuración inicial.....	9
4.1	Gitlab.....	9
4.1.1	Instalación.....	9
4.1.2	Primer acceso.....	10
4.1.3	Cambio de URL.....	11
4.1.4	Creación de grupos, proyectos y usuarios.....	11
4.1.5	Roles y permisos.....	15
4.2	Harbor.....	17
4.2.1	Instalación.....	17
4.2.2	Primer acceso.....	19
4.2.3	Creación de un usuario.....	21
4.2.4	Acceso desde un servidor externo.....	22
5	Gitlab CI/CD.....	23
5.1	Conceptos Básicos.....	23
5.2	Ventajas de Gitlab CI/CD.....	24
5.3	Integraciones de GitLab.....	25
5.4	Instalación de Gitlab Runner.....	25
6	Integración de Harbor.....	29
6.1	Requisitos previos.....	29
6.2	Integración en GitLab.....	29
6.3	Ejemplo de pipeline usando las variables de Harbor.....	32
6.4	Uso de comandos docker en el pipeline. Ejecutores del runner.....	34
7	Demostración con aplicación Python.....	37
7.1	Pasos previos.....	37
7.2	Fichero .gitlab-ci.yml.....	38
7.3	Resultados obtenidos.....	40
8	Conclusiones.....	47
9	Bibliografía y enlaces de interés.....	49
10	Anexos.....	51
10.1	Problemas encontrados con Docker in Docker.....	51

# 1 Descripción del proyecto

El proyecto tiene como objetivo principal la integración exitosa de Harbor, un registro de contenedores, en el flujo de CI/CD de GitLab. Esta integración permitirá una gestión más eficiente de imágenes de contenedor y una mayor automatización en el proceso de construcción, prueba y despliegue de aplicaciones. Se proporcionará una comprensión básica de cómo estas herramientas pueden trabajar juntas de manera efectiva.

## 1.1 Tecnologías que se van a utilizar

Las tecnologías que se van a utilizar son las siguientes:

- Gitlab: es una plataforma de gestión de repositorios de código que se basa en Git. Tiene funcionalidades CI/CD.
- Harbor: es un registro de contenedores para el almacenamiento seguro de imágenes Docker.
- Docker: es un proyecto de código abierto que automatiza el despliegue de aplicaciones dentro de contenedores, proporcionando una capa adicional de abstracción y automatización de virtualización de aplicaciones en múltiples sistemas operativos.
- Máquinas virtuales de Openstack.
- Gitlab Runners: es un componente o instancia que trabaja con Gitlab CI/CD para ejecutar trabajos del pipeline.

## 1.2 Resultados que se esperan obtener

Con la realización de este proyecto se espera que Harbor se integre de forma exitosa en Gitlab CI/CD.

Se espera que cada vez que haya un cambio en el proyecto se lance el pipeline. Se mostrará en Gitlab la automatización de la construcción, prueba y despliegue de una aplicación en un contenedor Docker en otro servidor. Se comprobará la imagen Docker en nuestro registro de Harbor y el despliegue correcto de la aplicación.

Para entender mejor de lo que tratará este proyecto, a continuación, se explicarán brevemente las dos principales tecnologías que se trabajarán, Harbor y Gitlab.

## 2 Fundamentos

### 2.1 Introducción a Harbor

Harbor es un proyecto de código abierto de la Cloud Native Computing Foundation (CNCF). Es un Registro de Contenedores (Container Registry) y está diseñado para facilitar la gestión de contenedores en un entorno empresarial. Harbor ofrece un registro seguro y escalable para almacenar, distribuir y gestionar imágenes de contenedores. Este registro facilita la creación, almacenamiento y distribución de imágenes Docker, permitiendo a los desarrolladores compartir fácilmente sus aplicaciones y entornos. Harbor brinda una serie de ventajas y características importantes.

- 1 Seguridad y Control: Harbor se centra en la seguridad y permite implementar políticas de acceso y escaneo de seguridad en las imágenes de contenedor. Puedes definir quién tiene acceso a las imágenes y garantizar que las imágenes sean seguras antes de ser implementadas.
- 2 Almacenamiento Privado y Público: Harbor se puede utilizar para almacenar imágenes de contenedor de forma privada, lo que es esencial en entornos empresariales sensibles. Además, se puede configurar repositorios públicos si deseas compartir imágenes con la comunidad o equipos externos.
- 3 Gestión de Imágenes: Harbor facilita la gestión de imágenes de contenedor, lo que incluye etiquetas (tags) para versiones, control de políticas de retención y un historial de actividad para cada imagen. Al permitir el almacenamiento y la gestión eficiente de imágenes de contenedor, Harbor ayuda a reducir costos al eliminar la necesidad de almacenamiento redundante y mejorar la eficiencia en la gestión de imágenes.
- 4 Integración con GitLab y CI/CD: Harbor se integra de manera efectiva con sistemas de CI/CD, como GitLab, lo que permite la automatización del proceso de compilación, prueba y despliegue de imágenes de contenedor.
- 5 Seguridad y escaneo de vulnerabilidades: Harbor ofrece escaneo de seguridad para identificar vulnerabilidades en las imágenes de contenedor. Esto es crucial para garantizar que las aplicaciones desplegadas sean seguras.
- 6 Alta Disponibilidad: se puede configurar Harbor en un clúster para garantizar una alta disponibilidad y evitar tiempos de inactividad.

- 7 Proyectos: en Harbor, los proyectos son espacios lógicos que agrupan imágenes relacionadas dentro de un registro. Estos proyectos ayudan a organizar y gestionar las imágenes de manera más efectiva, proporcionando una estructura lógica para la colaboración y la administración.
- 8 Repositorio de proyectos: dentro de un proyecto en Harbor, un repositorio es una ubicación específica que contiene una colección de imágenes de contenedores relacionadas. Cada proyecto puede tener uno o más repositorios, lo que permite una organización más detallada de las imágenes.
- 9 Políticas de retención (Retention Policies): las políticas de retención en Harbor son reglas que determinan cuánto tiempo se retienen las imágenes y cuándo se eliminan automáticamente. Estas políticas ayudan a gestionar el almacenamiento y garantizan que solo se retengan las versiones necesarias, facilitando el mantenimiento y la conformidad con políticas internas.
- 10 Soporte comunitario y activo: Harbor es una herramienta de código abierto respaldada por una comunidad activa y con una base de usuarios creciente, lo que garantiza un soporte y desarrollo continuo.

En resumen, Harbor es una solución valiosa para gestionar imágenes de contenedor en entornos privados, brindando seguridad, control y automatización. Su integración con herramientas de CI/CD, como GitLab, facilita la construcción y despliegue de aplicaciones en contenedores de manera eficiente y segura.

## 2.2 Introducción a Gitlab

Gitlab es una plataforma de desarrollo de software basada en Git que proporciona una variedad de herramientas para la gestión del ciclo de vida del desarrollo de aplicaciones. Es ampliamente utilizado por equipos de desarrollo de software para colaborar en proyectos, realizar control de versiones, implementar integración continua y entrega continua (CI/CD), rastrear problemas y administrar el ciclo de vida de las aplicaciones. Algunas características de Gitlab son:

- 1 Control de versiones: el control de versiones de Gitlab permite a los equipos rastrear cambios en el código, lo que es fundamental para el desarrollo de software colaborativo.
- 2 Gestión de proyectos: Gitlab brinda un entorno centralizado para que los equipos trabajen juntos en proyectos de desarrollo de software, lo que facilita la colaboración y la

comunicación. Además facilita la revisión de código mediante feedback, solicitudes de fusión (Merge Requests), Pull Request...

- 3 Integración Continua (CI) y Despliegue Continuo (CD): las capacidades de CI/CD de Gitlab permiten automatizar tareas repetitivas, lo que ahorra tiempo y reduce errores humanos (eficiencia).
- 4 Registros y métricas: proporciona registros y métricas detalladas de todas las actividades relacionadas con los usuarios, el código, los proyectos, etc. para ayudar a los equipos a monitorear el rendimiento de sus aplicaciones.
- 5 Control de acceso: Gitlab ofrece una administración avanzada de roles y permisos de los usuarios en los proyectos, así como la gestión mediante grupos, lo que garantiza que solo las personas autorizadas tengan acceso a proyectos y recursos específicos. Esto se hace mediante la gestión de usuarios y roles que permite controlar el acceso a los proyectos. Los usuarios pueden tener diferentes niveles de permisos, desde lectura hasta administración completa del proyecto (Guest, Custom, Reporter, Developer, Maintainer, Owner y Minimal Access).
- 6 Seguridad: la plataforma incluye funciones de seguridad que ayudan a proteger las aplicaciones y los datos de amenazas potenciales. Funciones como análisis estático de código, escaneo de dependencias y protección contra ataques de seguridad.
- 7 Integración con herramientas externas: Gitlab se integra con muchas herramientas populares, como Kubernetes, Docker, Slack, JIRA, Maven, Harbor, etc., lo que facilita la construcción de un flujo de trabajo de desarrollo personalizado. Esto hace que la administración de proyectos sea sencilla, ya que se ofrece una variedad de herramientas para gestionar proyectos.
- 8 Flexibilidad y Escalabilidad: Gitlab se puede adaptar a las necesidades específicas de los equipos de desarrollo y es escalable para proyectos de cualquier tamaño.

En resumen, Gitlab es una plataforma de desarrollo de software integral que simplifica la colaboración, automatiza el desarrollo y despliegue de aplicaciones, y proporciona herramientas sólidas de control de versiones y gestión de proyectos.

## 2.3 ¿Por qué se ha elegido este proyecto?

Teniendo en cuenta las funcionalidades y ventajas de estas tecnologías, se ha elegido este proyecto porque la gestión de contenedores y la automatización de CI/CD son esenciales en entornos empresariales actuales.

La integración de Harbor en Gitlab ayuda a reducir los riesgos asociados con la implementación de aplicaciones, ya que permite la detección temprana de vulnerabilidades y facilita las actualizaciones y correcciones.

La automatización de Gitlab CI/CD permite una mayor eficiencia en el desarrollo y despliegue de aplicaciones, lo que reduce costos y tiempos de entrega.

La gestión centralizada de imágenes de contenedor con Harbor garantiza que los datos y las aplicaciones críticas estén seguras y protegidas, lo que es fundamental para la privacidad de los datos y el cumplimiento de regulaciones.

Todas estas ventajas son fundamentales para mantener la competitividad y la integridad de los activos de la empresa en un entorno empresarial sensible.

## 3 Preparación del escenario en Openstack

Los requerimientos mínimos para instalar Gitlab son los siguientes (se tendrá en cuenta al crear la instancia):

- Almacenamiento: 2.5 GB
- CPU: 4 cores es el mínimo recomendado y soporta hasta 500 usuarios.
- Memoria: 4 GB de RAM es el tamaño mínimo de memoria y soporta hasta 500 usuarios.

Gitlab usa como base de datos PostgreSQL, que tiene como requisito mínimo:

- Almacenamiento: 5-10 GB

Los requisitos mínimos para la instalación de Harbor son:

- Almacenamiento: 40 GB
- CPU: 2
- Memoria: 4 GB

Además, el servidor Harbor debe tener instalado Docker y Docker Compose.

### 3.1 Creación de instancias

Los servidores se crearán usando instancias de Openstack del IES Gonzalo Nazareno. Para ello me tendré que conectar a la VPN del instituto, activar el cliente openstack y acceder con mi contraseña. Las dos instancias que voy a crear se van a configurar con cloud-init. Para ello he creado dos ficheros de configuración que se pueden encontrar en mi repositorio: [cloud-config-gitlab.yaml](#) y [cloud-config-harbor.yaml](#). Estos ficheros harán lo siguiente:

- Actualizan los paquetes de cada instancia.
- Especifica que el dominio utilizado será arantxa.gonzalonazareno.org y, para cada instancia, se indica el hostname (gitlab y harbor) y el FQDN (gitlab.arantxa.gonzalonazareno.org y harbor.arantxa.gonzalonazareno.org).
- Crean dos usuarios en cada instancia:
  - El usuario “arantxa” sin privilegios con el que accederé a la máquina usando mi clave ssh privada.
  - El usuario profesor, que puede utilizar sudo sin contraseña. Se han copiado las claves públicas de los profesores del IES Gonzalo Nazareno en las instancias para que puedan



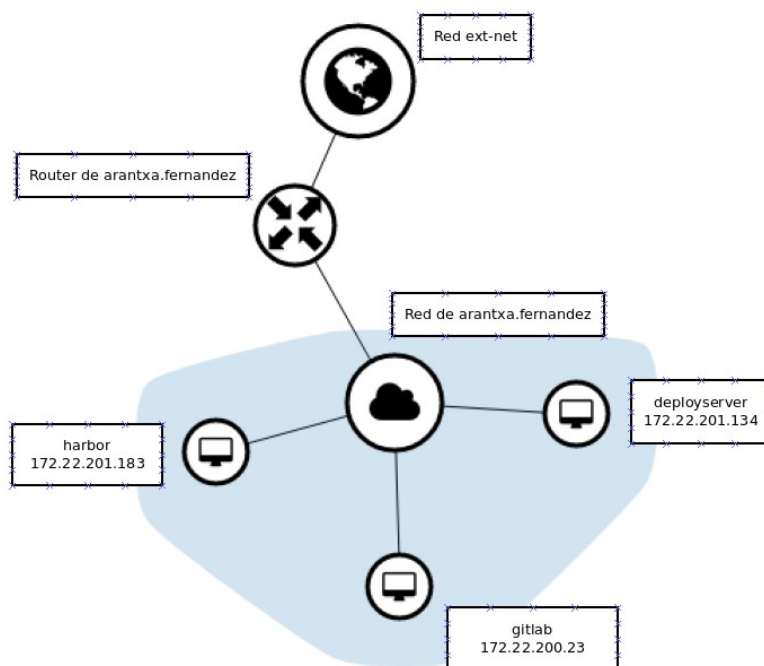
acceder con el usuario profesor. Este usuario se ha creado por si el profesorado quisiera ver la configuración en más detalle.

- Cambia la contraseña del usuario root.

Las instancias tendrán dos volúmenes asociados de 50GB booteables con imágenes Debian 11 Bullseye. Es un volumen por cada instancia y los he creado con los nombres vol-gitlab y vol-harbor.

A continuación, habrá que crear las instancias. La instancia con nombre gitlab usará el flavor m1.xlarge. Este flavor tiene 8GB de RAM, 20GB de disco y 4 VCPUs.<sup>1</sup> La instancia llamada harbor usará el flavor m1.large, que tiene 2 VCPUs, 20GB de disco y 4GB de RAM. También he creado una instancia extra llamada deployserver con un volumen asociado de 10GB con imagen Debian 11, con el flavor m1.normal (2 VCPUs, 10GB de disco y 1GB de RAM) que solo usaré para desplegar la aplicación Python de prueba cuando se ejecute el job correspondiente del pipeline.

Lo siguiente será asignar una IP flotante a cada instancia para poder acceder mediante ssh a cada una. El esquema que quedaría es el siguiente:



Todos los comandos usados para crear las instancias los dejo en el documento [instancias.md](#) de [mi repositorio](#).

<sup>1</sup> En un principio utilicé el flavor m1.large y, aunque Gitlab funcionaba, al lanzar los pipelines tardaba demasiado. Usando el flavor m1.xlarge funciona perfectamente.

## 4 Instalación y configuración inicial

### 4.1 Gitlab

#### 4.1.1 Instalación

Se va a proceder a continuación a realizar la instalación de Gitlab CE (Community-Edition) en su última versión, la 16.5.1, sobre la instancia Debian creada en Openstack, mediante la instalación automática de Omnibus.

Para empezar hacemos update y, a continuación, habrá que instalar el paquete Omnibus de GitLab. Para ello, tienes que añadir primero el “repositorio GitLab Package”.

```
sudo apt update
curl -s https://packages.gitlab.com/install/repositories/gitlab/gitlab-ce/
script.deb.sh | sudo bash
```

Para actualizar a la última versión se usa el siguiente comando:

```
sudo apt install gitlab-ce
```

Si quisiéramos instalar a una versión concreta primero vemos las versiones disponibles para nuestra distro y luego usamos el comando anterior especificando la versión.

```
sudo apt-cache madison gitlab-ce
sudo apt install gitlab-ce=15.8.0-ce.0
```

Para comprobar la versión instalada podemos usar:

```
sudo gitlab-rake gitlab:env:info
```

Para ver información de los servicios de Gitlab:

```
sudo gitlab-ctl status
```

Para comprobar que la configuración es correcta:

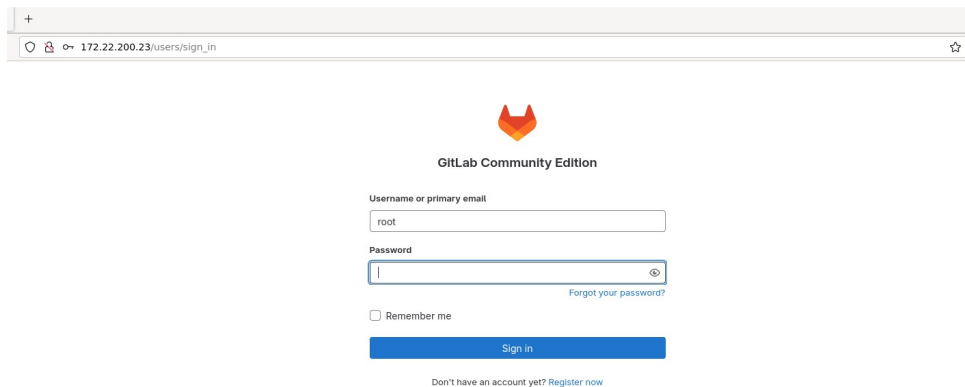
```
sudo gitlab-rake gitlab:check SANITIZE=true
```

Si quisiéramos acceder a la base de datos de Gitlab para versiones de la 14.2 en adelante:

```
sudo gitlab-rails dbconsole --database main
```

## 4.1.2 Primer acceso

Accedemos desde el navegador.



La primera vez que accedemos debemos poner como usuario “root” y la contraseña se adquiere de la siguiente forma:

```
sudo cat /etc/gitlab/initial_root_password
```

```
dtvj61/ruNzaRw6IVeWrxc8pdViqh4pj890tgDUjp4=
```

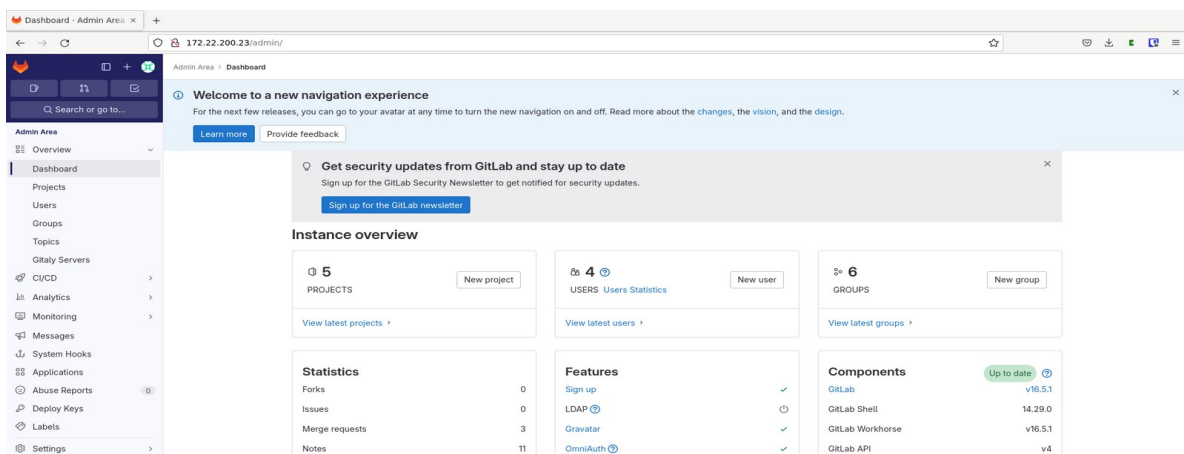
Una vez dentro debemos cambiarla porque la contraseña se borrará en las primeras 24h o al hacer “gitlab-ctl reconfigure” y no podremos volver a usarla.

Si ha habido algún problema y no funciona la contraseña seguir el siguiente enlace:

[https://docs.gitlab.com/ee/security/reset\\_user\\_password.html#reset-your-root-password](https://docs.gitlab.com/ee/security/reset_user_password.html#reset-your-root-password)

Mi nueva contraseña para acceso con el usuario root es:

```
g1t l4b-pr0y3ct0
```

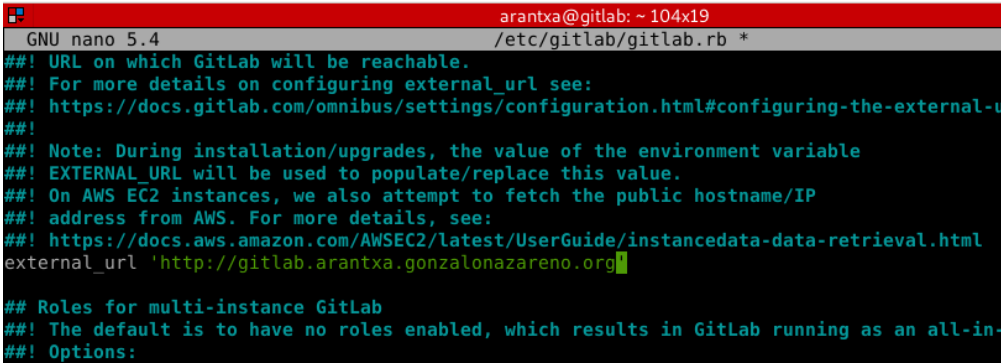


### 4.1.3 Cambio de URL

Para modificar la URL "https://gitlab.example.com" a la URL que se esté usando realmente habrá que editar el fichero de configuración gitlab.rb.

```
sudo nano /etc/gitlab/gitlab.rb
```

En el archivo gitlab.rb buscar la línea 9 ("external\_url") e introducir el URL deseada, que en mi caso sería *gitlab.arantxa.gonzalonazareno.org*, pero al no configurar el DNS he dejado la IP del servidor (<http://172.22.200.23>).



```

arantxa@gitlab: ~ 104x19
GNU nano 5.4 /etc/gitlab/gitlab.rb *
##! URL on which GitLab will be reachable.
##! For more details on configuring external_url see:
##! https://docs.gitlab.com/omnibus/settings/configuration.html#configuring-the-external-u
##!
##! Note: During installation/upgrades, the value of the environment variable
##! EXTERNAL_URL will be used to populate/replace this value.
##! On AWS EC2 instances, we also attempt to fetch the public hostname/IP
##! address from AWS. For more details, see:
##! https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/instancedata-data-retrieval.html
external_url 'http://gitlab.arantxa.gonzalonazareno.org'
## Roles for multi-instance GitLab
##! The default is to have no roles enabled, which results in GitLab running as an all-in-
##! Options:

```

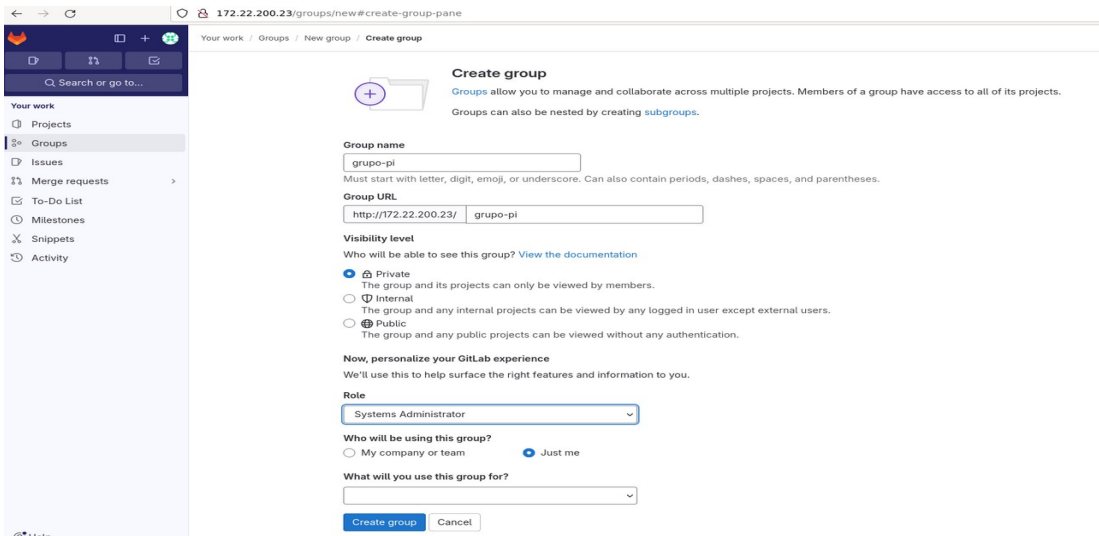
Gitlab se configurará con dicha URL tras su reinicio.

```
sudo gitlab-ctl restart
```

### 4.1.4 Creación de grupos, proyectos y usuarios

Primero creamos un grupo en Groups > New group > Create group.

Le damos un nombre, qué visibilidad tendrá, un rol de grupo y se especifica por quién será usado.



172.22.200.23/groups/new#create-group-pane

Your work / Groups / New group / Create group

**Create group**  
Groups allow you to manage and collaborate across multiple projects. Members of a group have access to all of its projects.  
Groups can also be nested by creating subgroups.

**Group name**  
grupo-pi  
Must start with letter, digit, emoji, or underscore. Can also contain periods, dashes, spaces, and parentheses.

**Group URL**  
http://172.22.200.23/grupo-pi

**Visibility level**  
Who will be able to see this group? [View the documentation](#)

Private  
The group and its projects can only be viewed by members.

Internal  
The group and any internal projects can be viewed by any logged in user except external users.

Public  
The group and any public projects can be viewed without any authentication.

**Now, personalize your GitLab experience**  
We'll use this to help surface the right features and information to you.

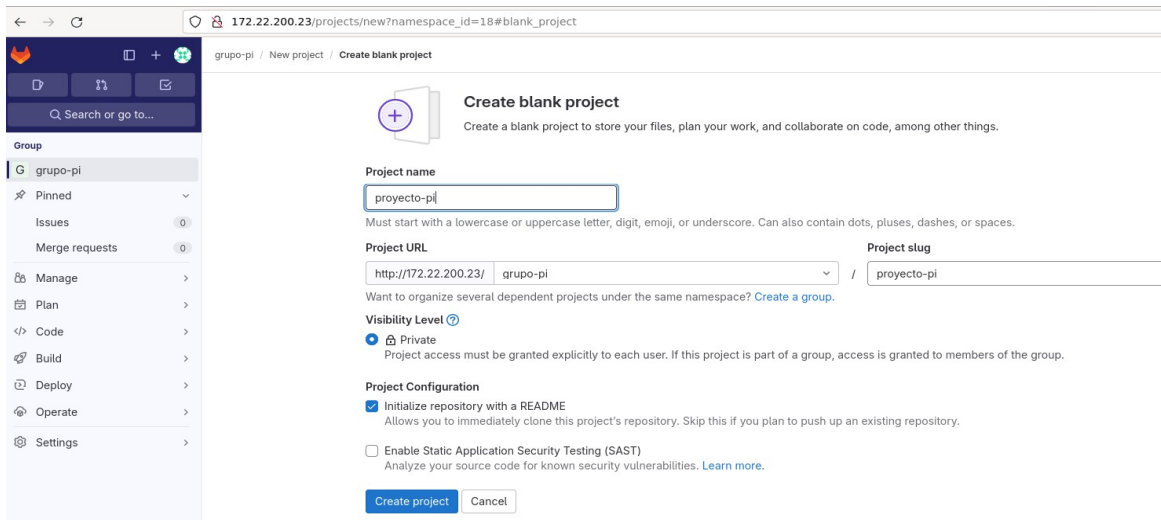
**Role**  
Systems Administrator

**Who will be using this group?**  
 My company or team  Just me

**What will you use this group for?**

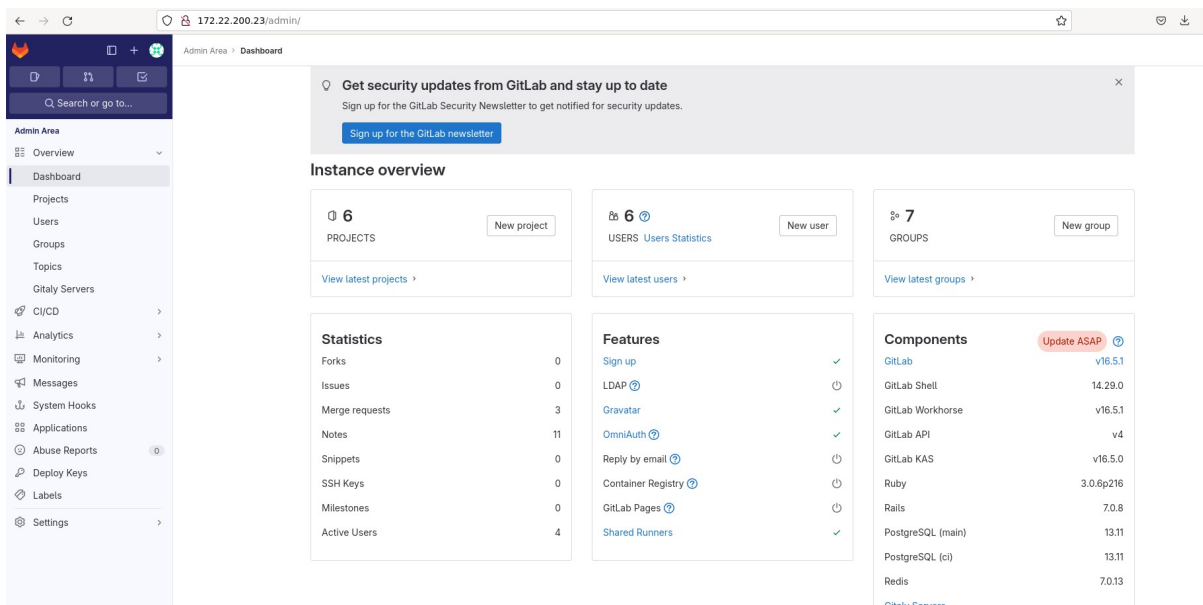
Create group Cancel

Una vez creado podemos crear un subgrupo o un proyecto en ese grupo. En mi caso paso a crear un proyecto en blanco. Le doy un nombre, establezco la visibilidad que tendrá y señalo la opción para que se cree un fichero Readme.

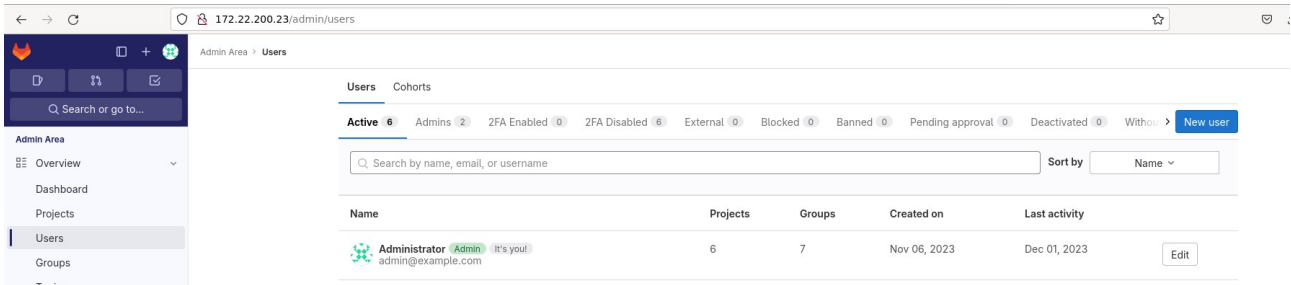


Una vez creado el proyecto vemos que en el fichero Readme vienen instrucciones con los pasos a seguir para clonar, añadir ficheros, integrarlo con otras herramientas, crear merge request, usar Gitlab CI/CD, etc.

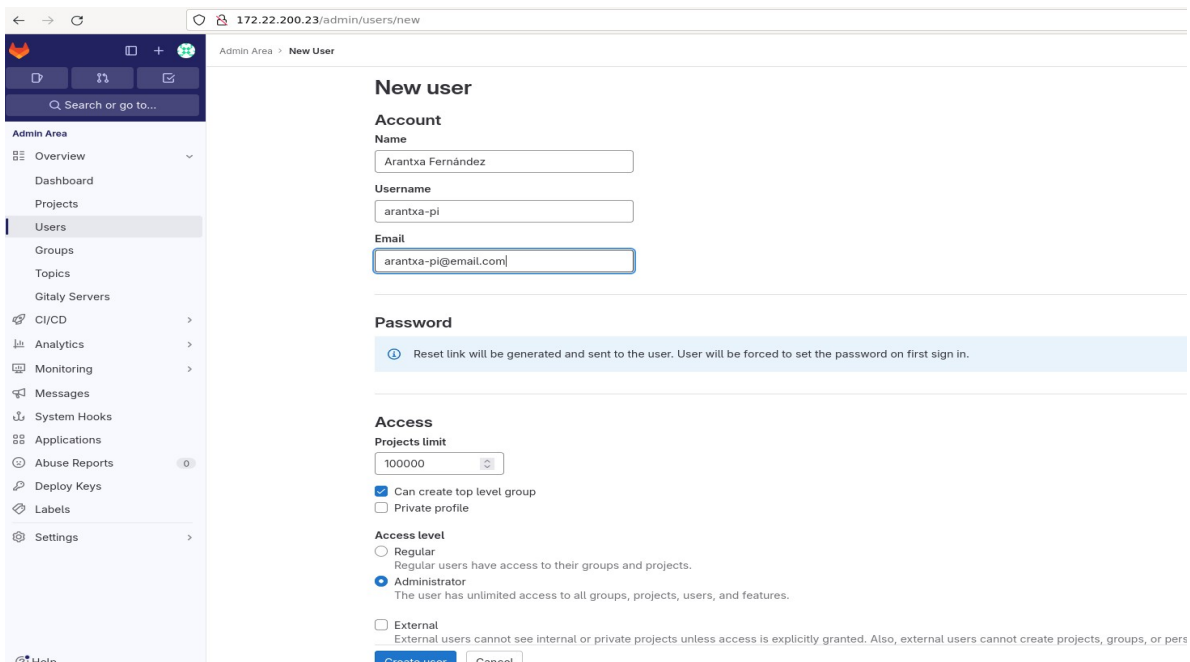
Para crear usuarios tendremos que ir al área del administrador. Desde aquí también se pueden crear grupos y proyectos.



Si vamos a Users podemos ver información de los usuarios activos, administradores, bloqueados, pendientes de aprobación...



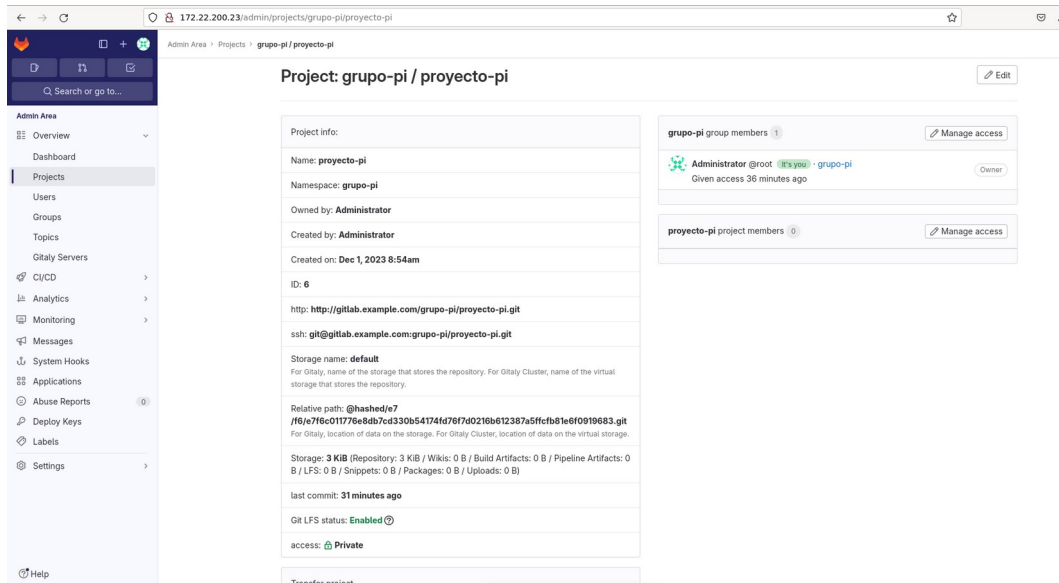
Hago click en New User para crear un nuevo usuario y relleno los campos.



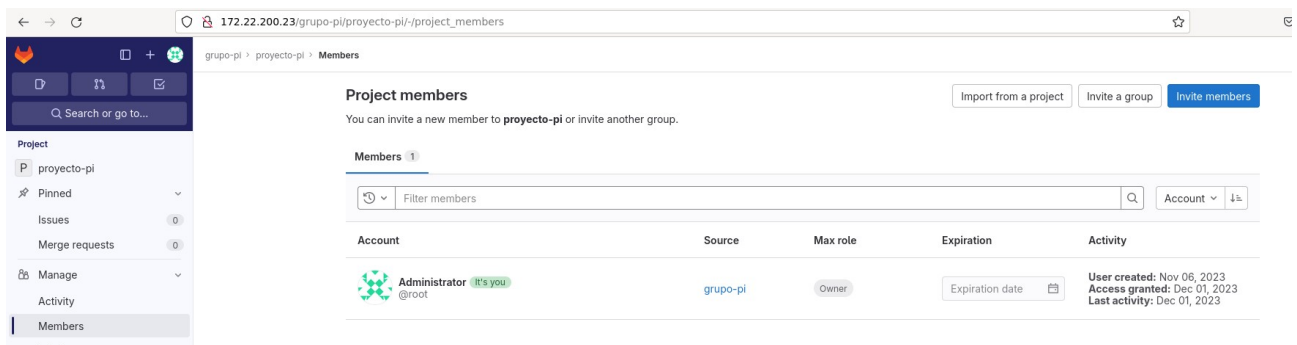
Le he dado nivel de acceso Administrador ya que este será el usuario que utilice a partir de ahora. Cuando esté listo le damos a Create user y se creará el usuario.

Ahora podemos acceder con este usuario y crear un nuevo grupo y proyecto o añadirlo al proyecto anteriormente creado desde el área de administración. En mi caso, haré esta última opción.

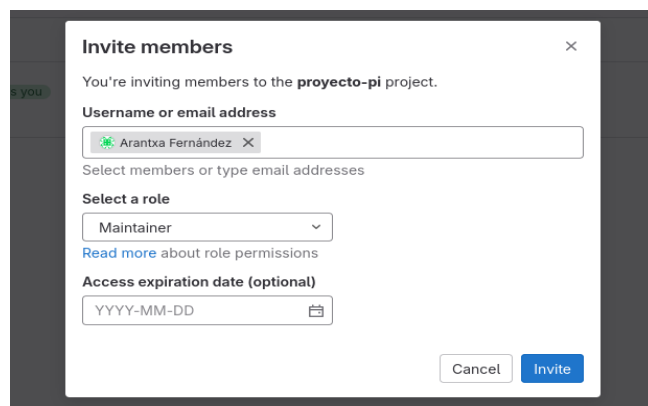
Para agregar el usuario arantxa-pi al proyecto proyecto-pi vamos a Projects y seleccionamos grupo-pi / proyecto-pi. Aquí veremos la siguiente información.

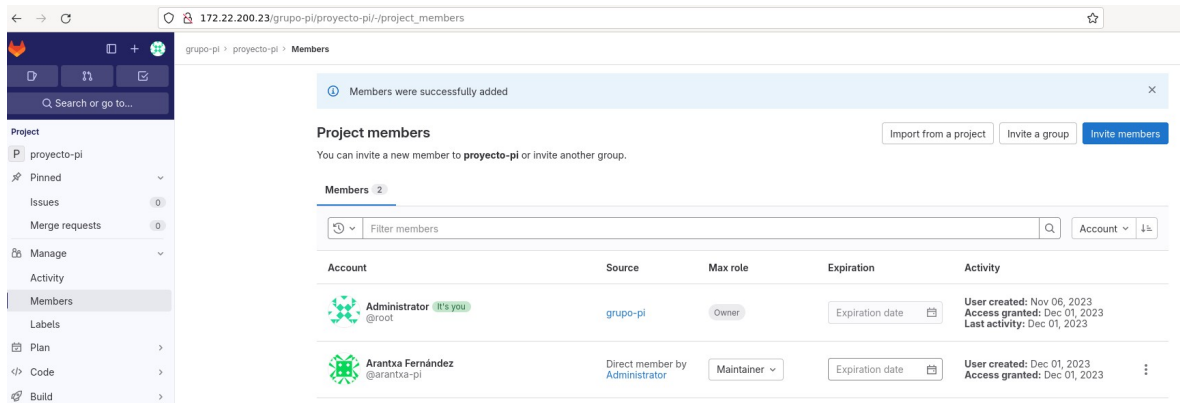


En Project members hacemos click en Manage access.

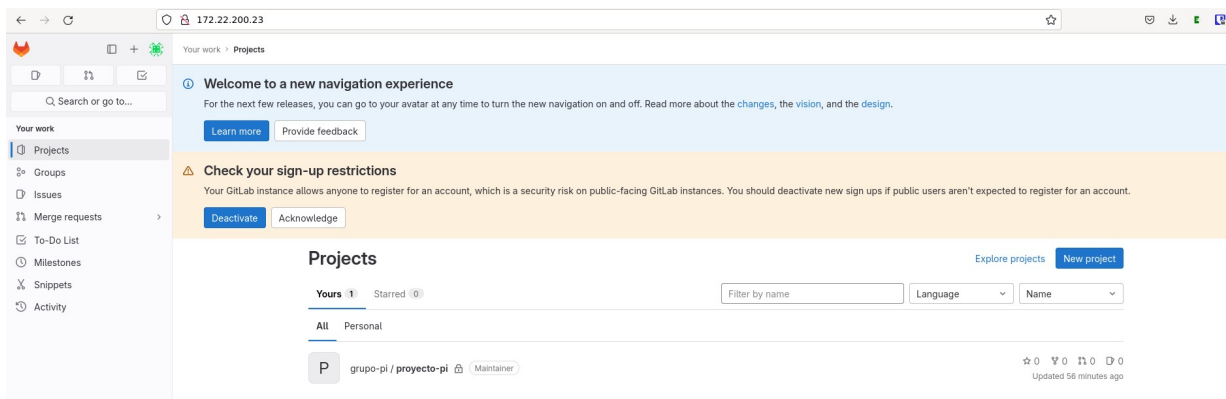


Y seleccionamos Invite members. En la ventana que aparece buscamos el usuario que queremos añadir y le damos un rol. En este caso le he dado el rol Maintainer (en el punto 4.1.5 hablo más en detalle de los roles y sus permisos).





Si accedo ahora con el usuario arantxa-pi puedo ver el proyecto al que me han dado acceso.



### 4.1.5 Roles y permisos

Cuando se agrega un usuario a un proyecto o grupo, se le asigna un rol. El rol determina qué acciones pueden realizar en GitLab. Si agregamos un usuario tanto al grupo de un proyecto como al proyecto en sí, se utiliza el rol superior. El rol Owner proporciona todos los permisos, pero está disponible solo:

- Para propietarios de grupos y proyectos.
- Para administradores.

Los roles disponibles son:

- **Guest:** este rol aplica solo a proyectos privados e internos. Es el rol con menos privilegios y tiene una vista muy limitada. Además, los usuarios invitados pueden simplemente ver los problemas personales que ellos mismos crearon o a los que están asignados.



- **Reporter:** tienen acceso de vista completa y de solo lectura. El usuario Reporter también puede comentar proyectos, en commits, crear hitos (milestones)... Tanto el rol Guest como el Reporter no pueden ejecutar trabajos de CI.
- **Developer:** es el primer nivel donde el nivel de permiso obtiene acceso de escritura, como confirmar y administrar ramas y etiquetas. Además, el usuario Developer puede:
  - Ejecutar trabajos de CI.
  - Crear nuevas ramas
  - Crear nuevas etiquetas
  - Cambiar el nombre de una rama
  - Enviar merge request
  - Aprobar merge request (depende de la configuración del proyecto)

Un desarrollador no puede gestionar proyectos, por ejemplo, darle acceso a alguien. Además, no puede eliminar ni crear proyectos.

- **Maintainer:** puede realizar la mayoría de acciones en un proyecto específico. Además, el rol maintainer puede realizar todo lo que hace el rol developer, más las acciones de la siguiente lista:
  - Asignar nuevos miembros
  - Administrar GitLab Pages
  - Administrar clústeres
  - Aprobación de merge request (depende de la configuración del proyecto)
  - Cambiar nombre del proyecto
- **Owner:** tienen el más alto nivel de acceso y control en todas las áreas del proyecto o grupo en GitLab. Cuando creas un nuevo proyecto en GitLab, tu usuario se configura como propietario automáticamente. Significa que tiene acceso completo a todas las funciones de ese proyecto. Puede realizar cualquier acción, modificar configuraciones y tienen la capacidad de agregar o eliminar miembros. Tienen el mayor nivel de permisos y responsabilidades.

Por otro lado, los usuarios que son administradores tienen acceso completo a todo el servidor GitLab. El rol de administrador es un rol especial que no se administra a nivel de proyecto o grupo.

Los **administradores**, entre otras cosas, pueden:

- Clonar cualquier proyecto

- Crear/renombrar/eliminar cualquier proyecto
- Administrar cuentas de usuarios y grupos
- Administrar GitLab Runners
- Administrar cualquier configuración en GitLab

## 4.2 Harbor

### 4.2.1 Instalación

Docker y docker-compose son necesarios para la instalación de Harbor, por lo que ese será el primer paso.

```
sudo apt install docker.io docker-compose
```

Voy al último release de Harbor, la versión 2.9.1, y me descargo el instalador online y su correspondiente fichero \*.asc (el fichero \*.asc es una clave OpenPGP).

```
wget https://github.com/goharbor/harbor/releases/download/v2.9.1/harbor-online-installer-v2.9.1.tgz
```

```
wget https://github.com/goharbor/harbor/releases/download/v2.9.1/harbor-online-installer-v2.9.1.tgz.asc
```

Realizar los siguientes pasos para verificar que el paquete descargado sea genuino.

- Instalar OpenPGP:

```
sudo apt install gpa gnupg2
```

- Obtener la clave pública del fichero \*.asc:

```
gpg --keyserver hkps://keyserver.ubuntu.com --receive-keys 644FF454C0B4115C
```

- Debería verse el mensaje: public key "Harbor-sign (The key for signing Harbor build) <jiangd@vmware.com>" imported
- Verificar la autenticidad del paquete:

```
gpg -v --keyserver hkps://keyserver.ubuntu.com --verify harbor-online-installer-version.tgz.asc
```

Se verá el siguiente mensaje de confirmación de que la firma del paquete coincide con la del archivo de la clave \*.asc:

```

arantxa@harbor:~$ gpg -v --keyserver hkps://keyserver.ubuntu.com --verify harbor-online-installer-v2.9.1.tgz.asc
gpg: assuming signed data in 'harbor-online-installer-v2.9.1.tgz'
gpg: Signature made Wed Nov  1 07:06:25 2023 UTC
gpg:                using RSA key 7722D168DAEC457806C96FF9644FF454C0B4115C
gpg: using pgp trust model
gpg: Good signature from "Harbor-sign (The key for signing Harbor build) <jiangd@vmware.com>" [unknown]
gpg: WARNING: This key is not certified with a trusted signature!
gpg:                There is no indication that the signature belongs to the owner.
Primary key fingerprint: 7722 D168 DAEC 4578 06C9  6FF9 644F F454 C0B4 115C
gpg: binary signature, digest algorithm SHA512, key algorithm rsa4096

```

A continuación, extraemos el paquete descargado anteriormente.

```
tar xzvf harbor-online-installer-v2.9.1.tgz
```

Accedemos al directorio creado y establecemos los parámetros de configuración en el fichero harbor.yml, que tendrán efecto cuando se corra el script install.sh para instalar o reconfigurar Harbor.

```
cd harbor
cp harbor.yml.tpl harbor.yml
nano harbor.yml
```

Cambio los siguientes valores (todavía no he configurado el DNS y el certificado https así que comento esas líneas):

```

hostname: 172.22.201.183
#hostname: harbor.arantxa.gonzalonazareno.org
# http related config
http:
  # port for http, default is 80. If https enabled, this port will redirect to
  https port
  port: 80

# https related config
#https:
  # https port for harbor, default is 443
  # port: 443
  # The path of cert and key files for nginx
  # certificate: /your/certificate/path
  # private_key: /your/private/key/path
...
harbor_admin_password: h4rb0r-pr0y3ct0
...
trivy:
  ignore_unfixed: false
  skip_update: false
  offline_scan: true
...

```

Una vez configurado pasamos a instalar Harbor. Hay varias formas de hacerlo.

- Instalando solo Harbor (sin Notary, Trivy o Chart Repository Service)

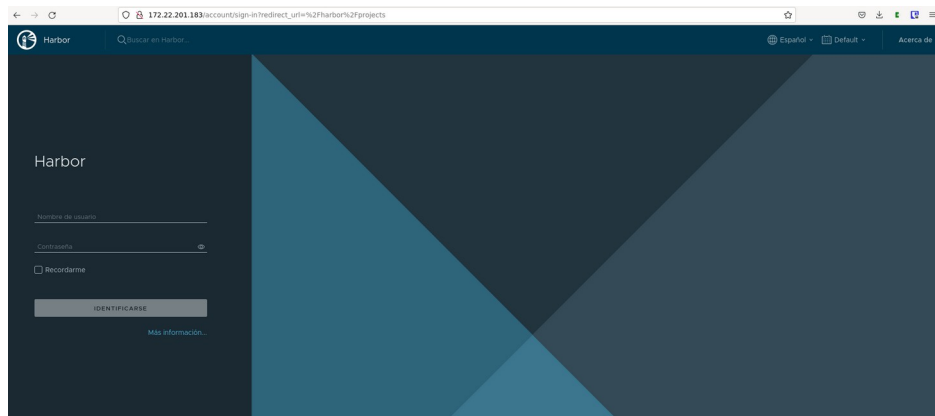
- Harbor con Notary
- Harbor con Trivy
- Harbor con Chart Repository Service
- Harbor con dos o tres de las anteriores.

Estas herramientas son escáneres de vulnerabilidades. En mi caso he decidido hacer la instalación usando [Trivy](#).

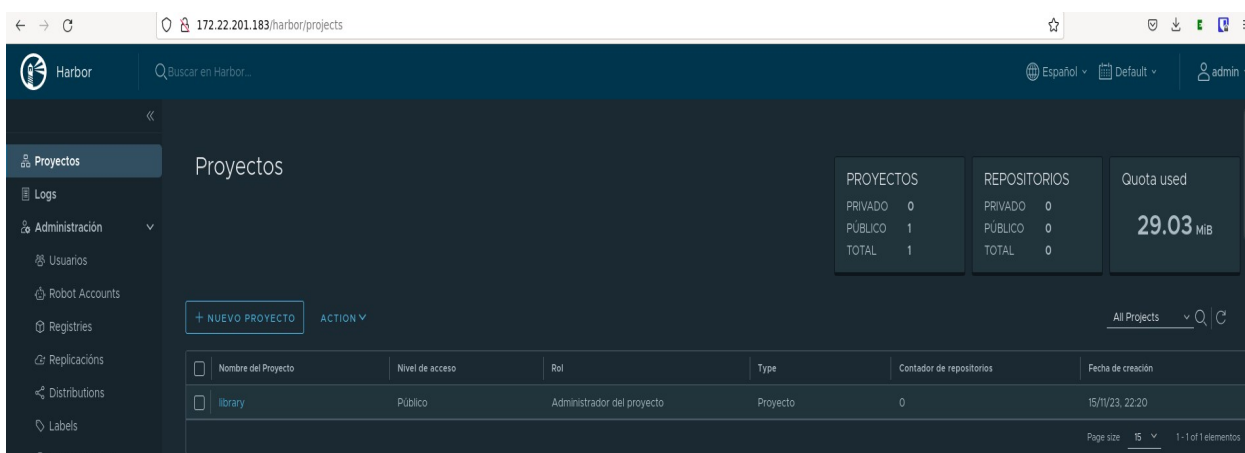
```
sudo ./install.sh --with-trivy
```

## 4.2.2 Primer acceso

Accedemos desde el navegador.



El usuario es admin y la contraseña la especificamos en el fichero de configuración (h4rb0r-pr0y3ct0).



Voy a realizar una prueba para comprobar que puedo subir imágenes al registro de Harbor. Para ello, voy a descargar una imagen oficial.

```
sudo docker pull hello-world
```

Le pongo un tag a la imagen.

```
sudo docker tag hello-world 172.22.201.183/library/hello-world
```

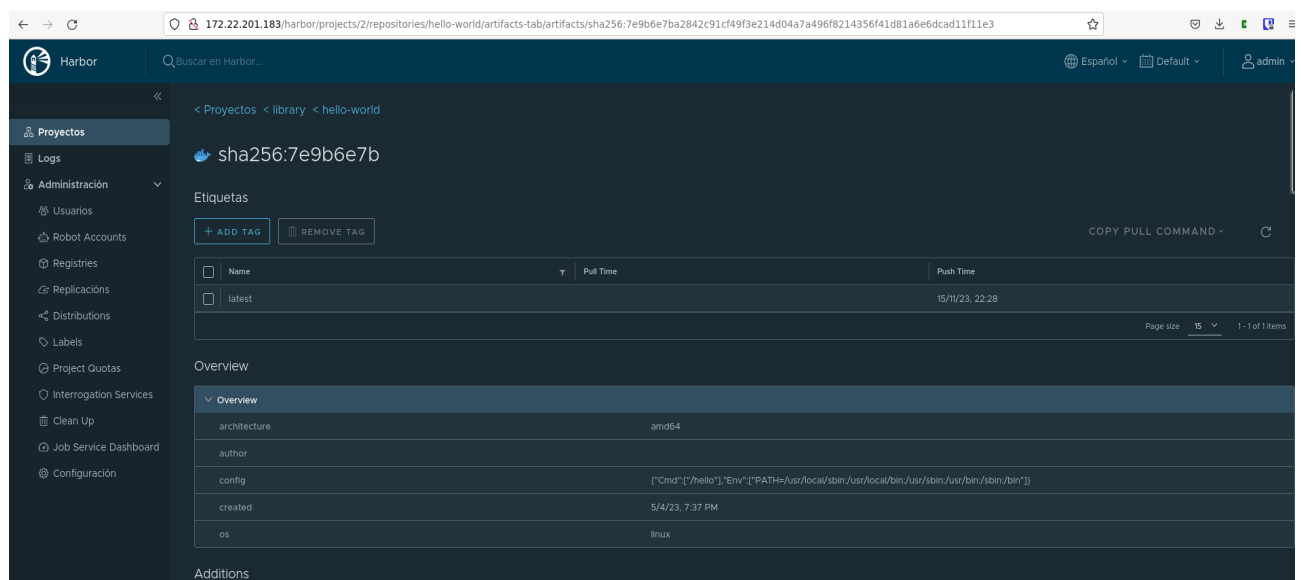
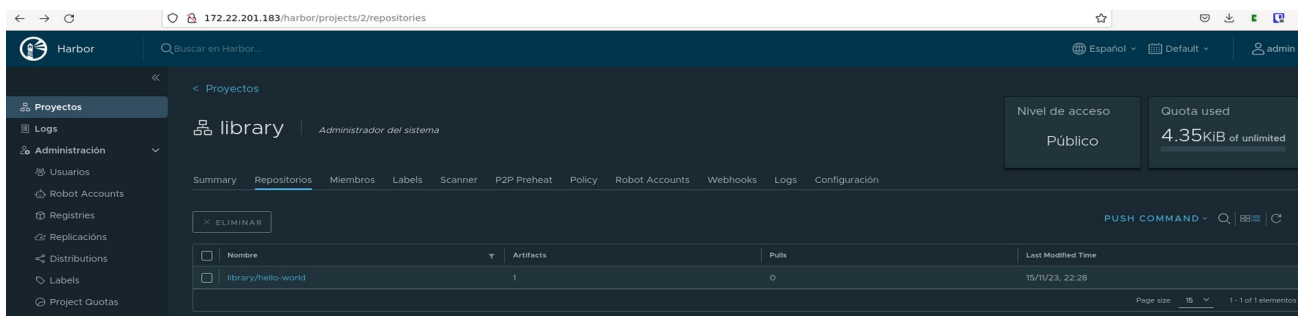
Hago login en mi Harbor.

```
sudo docker login 172.22.201.183
```

Y subo la imagen.

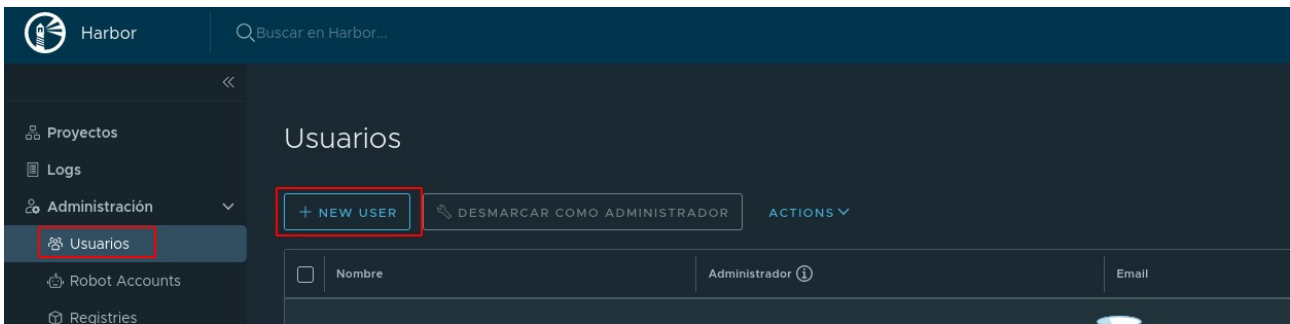
```
sudo docker push 172.22.201.183/library/hello-world
```

Compruebo que se ha subido.



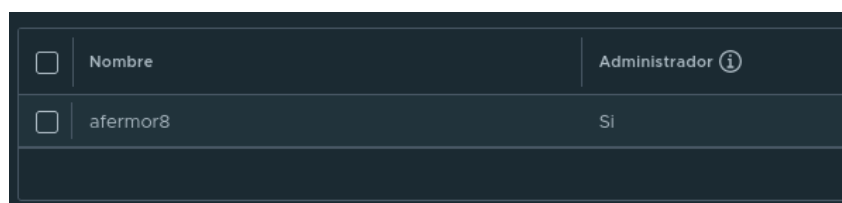
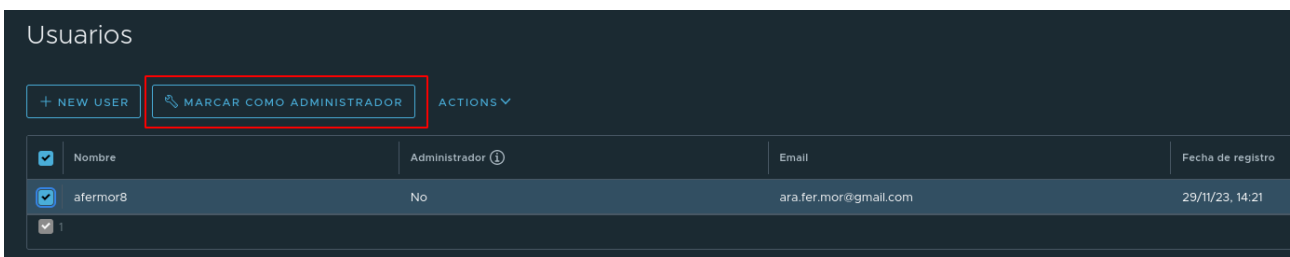
### 4.2.3 Creación de un usuario

En Harbor, en la columna de la izquierda, vamos a Administración > Usuarios > New User.



Rellenamos los campos y hacemos click en Ok.

Seleccionamos el usuario creado y lo marcamos como administrador.



## 4.2.4 Acceso desde un servidor externo

Para poder acceder desde un servidor externo, en este caso desde el servidor Gitlab, habrá que realizar los siguientes pasos:

### En el servidor Harbor:

1. Modificamos el demonio de docker para que permita realizar la conexión http.

```
sudo nano /etc/docker/daemon.json
{
  "insecure-registries" : ["0.0.0.0/0"]
}
```

2. Reiniciar el servicio de docker.

```
sudo systemctl restart docker
```

3. Reiniciamos Harbor.

```
cd harbor/
sudo docker-compose down -v
sudo docker-compose up -d
```

4. Podemos comprobar que Harbor sigue funcionando y seguimos teniendo acceso desde el navegador.

### En el servidor Gitlab:

1. Instalamos docker.

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

2. Modificamos el demonio del cliente docker para que permita realizar la conexión http.

```
sudo nano /etc/docker/daemon.json
{
  "insecure-registries" : ["0.0.0.0/0"]
}
```

3. Reiniciar el servicio de docker.

```
sudo systemctl restart docker
```

4. Y ahora sí podremos hacer el login. He hecho el login con el usuario creado anteriormente 'afermor8'.

```
sudo docker login http://172.22.201.183
```

```
arantxa@gitlab:~$ sudo docker login http://172.22.201.183
Username: afermor8
Password:
WARNING! Your password will be stored unencrypted in /root/.docker/config.json.
Configure a credential helper to remove this warning. See
https://docs.docker.com/engine/reference/commandline/login/#credentials-store

Login Succeeded
```

## 5 Gitlab CI/CD

### 5.1 Conceptos Básicos

GitLab CI/CD automatiza el proceso de integración, prueba, entrega y despliegue del código. Los desarrolladores definen pipelines que describen cómo se deben realizar estas acciones. Esto asegura la calidad del código y facilita la entrega continua de nuevas versiones del software.

A continuación, se definen algunos conceptos básicos a tener en cuenta para la utilización de Gitlab CI/CD.

- **Pipeline:** es el conjunto de pasos automáticos que definen la integración, prueba y despliegue del código fuente. Un pipeline en GitLab CI/CD consiste en jobs y stages que se ejecutan secuencialmente.
- **Stage:** una fase dentro de un pipeline que agrupa uno o varios jobs relacionados. Los stages permiten organizar y estructurar el flujo de trabajo del pipeline.
- **Job:** es una tarea o unidad de trabajo dentro de un pipeline. Cada job realiza una acción específica, como compilar el código, ejecutar pruebas o desplegar la aplicación.
- **Variables de entorno:** valores configurables que se pueden declarar y usar en el fichero `.gitlab-ci.yml` para parametrizar la ejecución de los jobs.
- **Variables definidas en la interfaz de usuario:** las variables confidenciales, como tokens o contraseñas, deben almacenarse en la configuración de la interfaz de usuario, no en el archivo `.gitlab-ci.yml`
- **Variables predefinidas:** variables predefinidas en Gitlab que podemos utilizar en cualquier momento en el pipeline.
- **Runners:** son agentes que ejecutan los jobs del pipeline.
- **Fichero `.gitlab-ci.yml`:** es un archivo de configuración escrito en yml que define la estructura y los pasos del pipeline. Este fichero se encuentra en la raíz del repositorio y especifica cómo se deben ejecutar los jobs y las tareas a realizar.
- **Trigger:** son eventos que inician la ejecución del pipeline. Puede ser un push al repositorio, la creación de una solicitud de fusión (Merge Request), o un disparador manual.
- **Artifacts (artefactos):** archivos generados por un job que se pueden utilizar en otros jobs del mismo pipeline.



- **Registro de Pipelines:** registro que muestra el historial de todos los pipelines ejecutados en un proyecto. Proporciona detalles sobre el estado de cada ejecución, los jobs y las duraciones.

Un ejemplo básico de un fichero `.gitlab-ci.yml` es el siguiente:

```
stages:
  - build-job
  - test

build-job:
  stage: build
  script:
    - echo "Hello, $GITLAB_USER_LOGIN !"

test-job1:
  stage: test
  script:
    - echo "This job tests something"

test-job2:
  stage: test
  script:
    - echo "This job tests something, but takes more time than test-job1."
    - echo "It runs the sleep command for 20 seconds after the echo command"
    - echo "which simulates a test that runs 20 seconds longer than test-job1"
    - sleep 20
```

Este ejemplo muestra dos etapas: build y test. La etapa build tiene un job llamado build-job, y la etapa test tiene dos jobs, test-job1 y test-job2. Los comentarios enumerados en los comandos echo se muestran en la interfaz de usuario cuando vemos la ejecución de los jobs. El valor de la variable predefinida `$GITLAB_USER_LOGIN` se completa cuando se ejecuta los trabajos. En este caso la variable predefinida mostrará el nombre del usuario que ha lanzado el pipeline.

## 5.2 Ventajas de Gitlab CI/CD

Existen herramientas CI/CD populares externas, como Jenkins, que tienen la misma función. ¿Cuál serían las ventajas de Gitlab CI/CD frente a estas herramientas? A continuación, se destacan algunas ventajas específicas de GitLab CI/CD en comparación con Jenkins:

- Integración nativa y agilidad: GitLab CI/CD está integrado directamente en la plataforma GitLab, lo que significa que no es necesario configurar o mantener servidores adicionales. Todo está centralizado en la misma interfaz, facilitando la entrega continua de nuevas

funcionalidades y correcciones de errores, lo que simplifica la gestión del ciclo de vida del desarrollo.

- Control: Gitlab CI/CD permite un mayor control sobre el proceso de desarrollo, garantizando la calidad y la estabilidad del software.
- Configuración como código y simplicidad en la configuración: en Gitlab CI/CD se utilizan archivos YAML (.gitlab-ci.yml) para definir los pipelines como código. La interfaz de usuario y la configuración basada en YAML simplifican la creación y gestión de pipelines. La mayoría de las configuraciones se pueden realizar a través de la interfaz web.
- Automatización de pruebas y despliegues.

### 5.3 Integraciones de GitLab

GitLab se destaca por su capacidad para integrarse con una amplia variedad de herramientas y servicios, lo que permite a los equipos construir flujos de trabajo personalizados y más eficientes. A continuación, se presenta un listado de algunas herramientas y servicios que GitLab puede integrar:

- Control de versiones: aunque Gitlab se basa en Git, también puede integrarse con otros sistemas de control de versiones, como Subversion y Mercurial.
- Herramientas de CI/CD: Jenkins, Travis CI, CircleCI
- Registros de contenedores: Harbor
- Herramientas de gestión de proyectos: Jira, Redmine, Trello, Bugzilla, Asana
- Comunicación y colaboración: Slack, Mattermost
- Monitorización: Grafana, Prometheus
- Automatización y orquestación: Ansible, Puppet, Chef
- Pruebas y calidad de código: Selenium, SonarQube
- Gestión de configuración: Terraform
- Infraestructura como Servicio (IaaS): AWS, Azure, Google Cloud

### 5.4 Instalación de Gitlab Runner

Existen diferentes formas de instalar runners en GitLab. En mi caso lo haré usando el paquete de instalación oficial ofrecido por GitLab.

Agrego el repositorio:

```
curl -L https://packages.gitlab.com/install/repositories/runner/gitlab-runner/
script.deb.sh | sudo bash
```

Debian tiene en sus repositorios oficiales un paquete gitlab-runner, pero suele estar desactualizado en comparación con los del repositorio GitLab. Para que el paquete que vamos a instalar de GitLab tenga prioridad sobre el proveniente de los repositorios de Debian vamos a pinnear mediante un fichero de configuración.

```
sudo nano /etc/apt/preferences.d/pin-gitlab-runner.pref
```

```
Explanation: Prefer GitLab provided packages over the Debian native ones
Package: gitlab-runner
Pin: origin packages.gitlab.com
Pin-Priority: 1001
```

Y ahora instalamos gitlab-runner.

```
sudo apt-get install gitlab-runner
```

Para registrar el nuevo runner en Gitlab usamos el siguiente comando y nos hará una serie de preguntas.

```
sudo gitlab-runner register
```

```
arantxa@gitlab:~$ sudo gitlab-runner register
[sudo] password for arantxa:
Sorry, try again.
[sudo] password for arantxa:
Runtime platform arch=amd64 os=linux pid=1013647 revision=f5da3c5a ve
rsion=16.6.1
Running in system-mode.

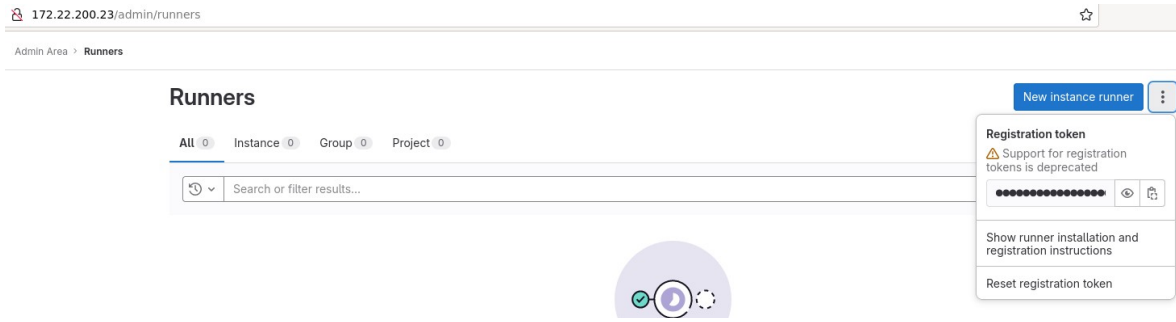
Enter the GitLab instance URL (for example, https://gitlab.com/):
http://172.22.200.23
Enter the registration token:
8xZnRpgRGL22g2SMRMW
Enter a description for the runner:
[gitlab]: Shared runner PI
Enter tags for the runner (comma-separated):

Enter optional maintenance note for the runner:

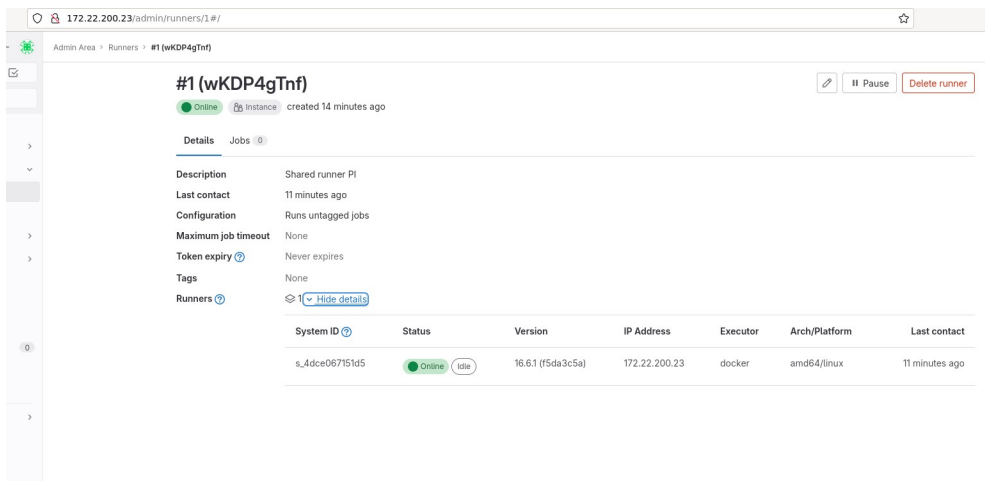
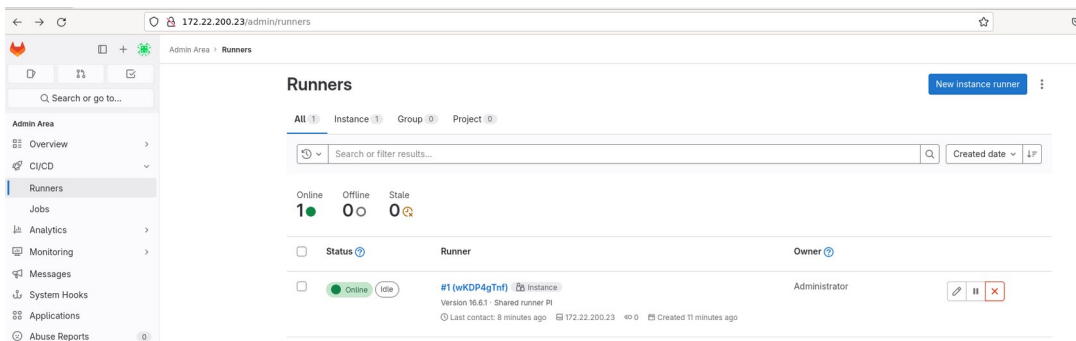
WARNING: Support for registration tokens and runner parameters in the 'register' command has been deprec
ated in GitLab Runner 15.6 and will be replaced with support for authentication tokens. For more informa
tion, see https://docs.gitlab.com/ee/ci/runners/new_creation_workflow
Registering runner... succeeded runner=8xZnRpgR
Enter an executor: docker, parallels, docker-autoscaler, virtualbox, docker+machine, instance, kubernet
e, custom, docker-windows, shell, ssh:
docker
Enter the default Docker image (for example, ruby:2.7):
debian:bullseye
Runner registered successfully. Feel free to start it, but if it's running already the config should be
automatically reloaded!

Configuration (with the authentication token) was saved in "/etc/gitlab-runner/config.toml"
arantxa@gitlab:~$
```

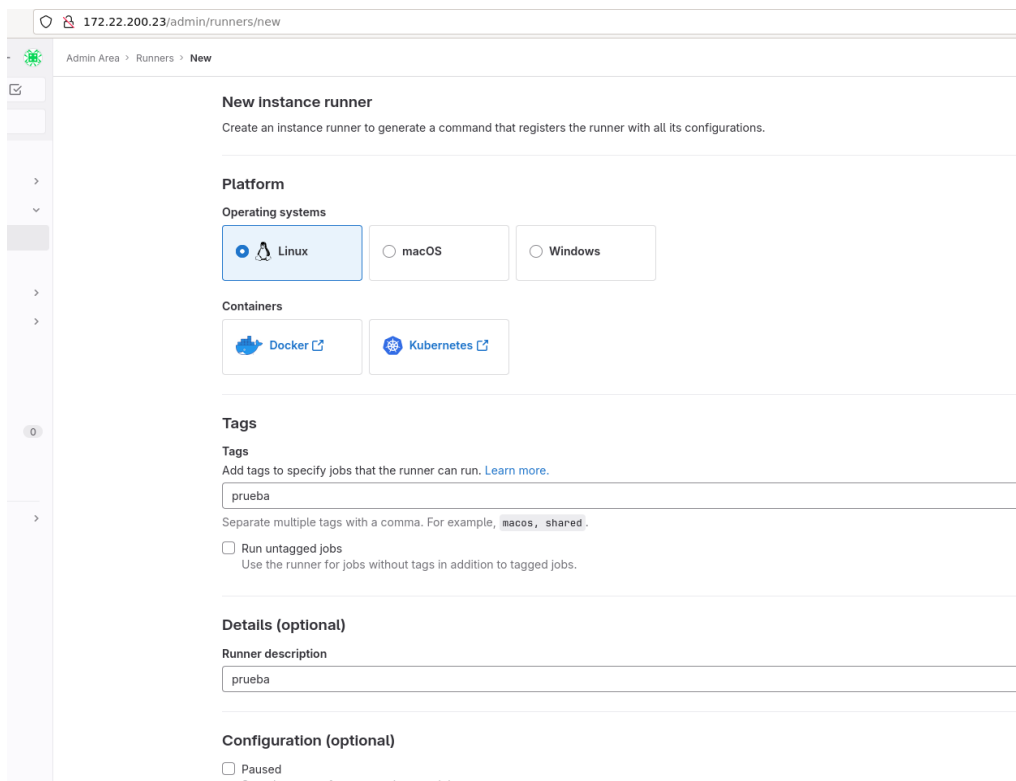
Primero nos pedirá la URL. El Token de registro se consigue en Gitlab desde el área de administración en el apartado Runners (<http://172.22.200.23/admin/runners>). Ahí hacemos click en los tres puntos que están al lado de donde pone New instance y copiamos el token.



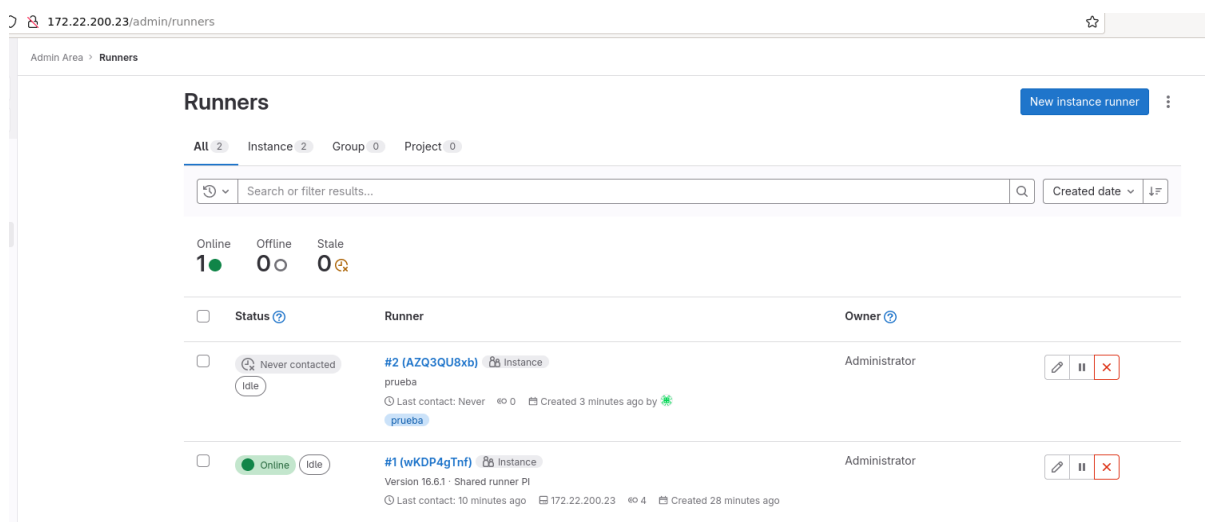
Podemos poner una descripción al runner si queremos, que en mi caso ha sido “Shared runner PI”. No le he dado tags (para que corra todos los trabajos sin etiqueta) ni nota de mantenimiento. El ejecutor será docker con la imagen debian:bullseye. Una vez añadido podemos actualizar la página de runners que habíamos abierto y podremos ver el runner creado.



Los runners también se pueden crear desde esa misma página haciendo click sobre New instance runner.



Una vez se crea aparecerán los pasos a seguir para que se registre el runner. Si no lo registramos aparecerá como no registrado.



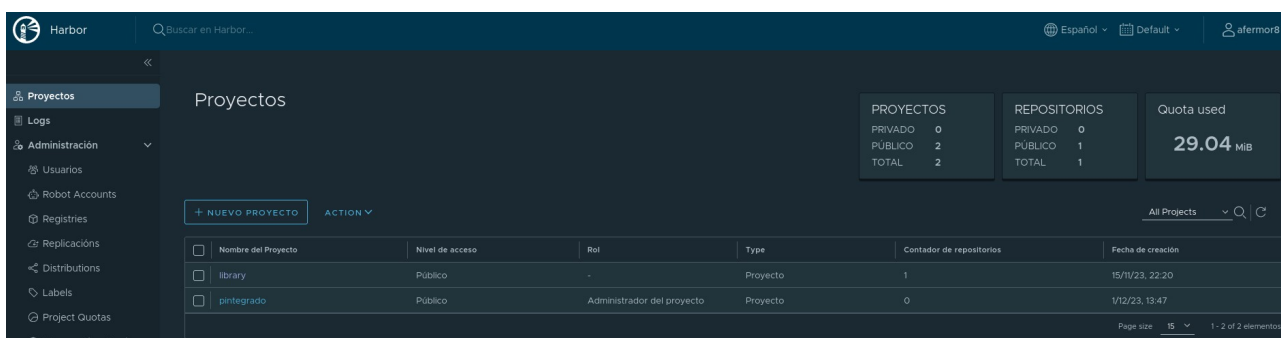
## 6 Integración de Harbor

### 6.1 Requisitos previos

Antes de realizar los pasos de la integración debemos tener en cuenta que hay que cumplir unos requisitos. En la instancia de Harbor hay que asegurarse de que:

- El proyecto a integrar ha sido creado.
- El usuario autenticado tiene permiso para extraer, enviar y editar imágenes en el proyecto Harbor.

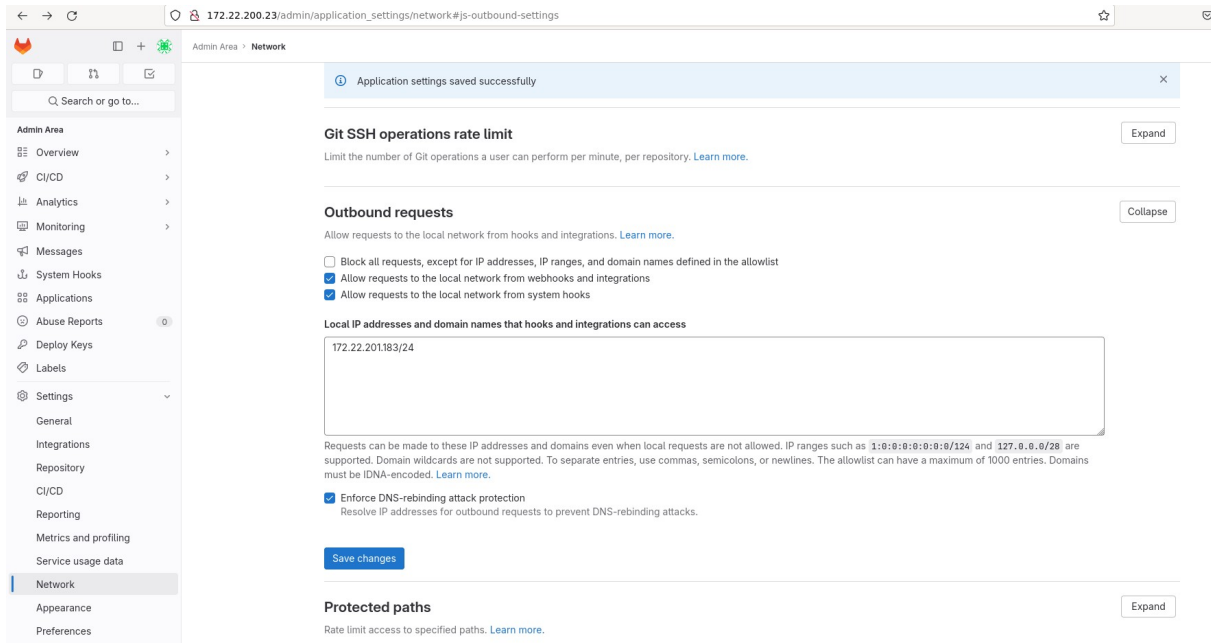
En mi caso usaré el usuario admin y he creado un proyecto llamado pintegrado al que le he añadido dicho usuario.



### 6.2 Integración en GitLab

GitLab admite la integración de proyectos Harbor a nivel de grupo o proyecto, pero como administrador de una instancia GitLab se puede también añadir unos parámetros de configuración para una integración en concreto (Harbor en este caso) y que todos los proyectos hereden y usen esta configuración por defecto. Esto activa la integración de Harbor para todos los proyectos que no estén usando una configuración propia. Esta configuración por defecto se puede actualizar siempre que se quiera, cambiando la configuración de todos los proyectos que la usaban y de los que todavía no la tenían activa.

Para proceder a la integración se necesitará el siguiente paso previo. Vamos al área de administración y seleccionamos Settings > Network > Outbound requests. Al expandir esa opción de configuración debemos marcar la casilla donde pone “Allow requests to the local network from webhooks and integrations”. Además he añadido la IP del servidor Harbor.



A continuación, activamos la integración siguiendo estos pasos:

1. Vamos al área de administración.
2. Seleccionamos Settings > Integrations > Harbor.
3. Dejamos activa la casilla Enable integration y proporcionamos la información de configuración del servidor:
  - URL de Harbor: la URL base de la instancia de Harbor que se vincula a este proyecto de GitLab. En este caso <http://172.22.201.183>
  - Nombre del proyecto Harbor: el nombre del proyecto en la instancia de Harbor. En este caso pintegrado
  - Nombre de usuario: el nombre de usuario en la instancia de Harbor. En este caso admin
  - Contraseña: la contraseña del usuario.
4. Seleccionamos Save changes.

**H Harbor**

Settings Projects using custom settings

After the Harbor integration is activated, global variables `$HARBOR_USERNAME`, `$HARBOR_HOST`, `$HARBOR_OCI`, `$HARBOR_PASSWORD`, `$HARBOR_URL` and `$HARBOR_PROJECT` will be created for CI/CD use.

**Enable integration**  
 Active

**Harbor URL**  
http://172.22.201.183  
Base URL of the Harbor instance.

**Harbor project name**  
pintegrado  
The name of the project in Harbor.

**Harbor username**  
admin

**Enter new Harbor password**  
●●●●●●●●●●  
Leave blank to use your current password.

Save changes Cancel Reset

Como vemos en el mensaje superior, tras activar la integración de Harbor:

- Las variables globales `$HARBOR_USERNAME`, `$HARBOR_HOST`, `$HARBOR_OCI`, `$HARBOR_PASSWORD`, `$HARBOR_URL` y `$HARBOR_PROJECT` se crean para el uso de CI/CD. Se crean con esos nombres y se podrán usar directamente en el pipeline.
- Hay que tener en cuenta que la configuración de integración a nivel de proyecto anula la configuración de integración a nivel de grupo.

Si vamos al proyecto que nos interesa, que en mi caso es proyecto-pi, y accedemos a Settings > Integrations > Harbor vemos que éste ha obtenido la configuración por defecto de la integración de Harbor que hicimos a nivel de administrador.

grupo-pi > proyecto-pi > Integration Settings > Harbor

Search page

**H Harbor** Active

Default settings are inherited from the instance level. [Learn more](#) Use default settings

After the Harbor integration is activated, global variables `$HARBOR_USERNAME`, `$HARBOR_HOST`, `$HARBOR_OCI`, `$HARBOR_PASSWORD`, `$HARBOR_URL` and `$HARBOR_PROJECT` will be created for CI/CD use.

**Enable integration**  
 Active

**Harbor URL**  
http://172.22.201.183  
Base URL of the Harbor instance.

**Harbor project name**  
pintegrado  
The name of the project in Harbor.

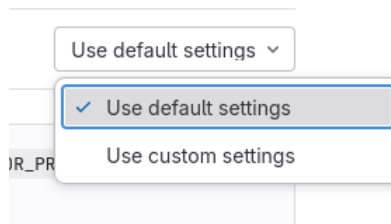
**Harbor username**  
admin

**Enter new Harbor password**  
Leave blank to use your current password.

Save changes Cancel



Si quisiéramos podríamos customizar la configuración en la pestaña de arriba seleccionando la opción "Use custom settings".



### 6.3 Ejemplo de pipeline usando las variables de Harbor

Voy a crear un fichero `.gitlab-ci.yml` sencillo en el proyecto `proyecto-pi` de GitLab que creé anteriormente.

Para empezar me he clonado el repositorio para trabajar desde la terminal.

```
mkdir git && cd git
git clone http://172.22.200.23/grupo-pi/proyecto-pi.git
```

\*Pedirá el usuario y contraseña.

En nuestro repositorio habrá que crear el fichero `.gitlab-ci.yml` en la raíz del proyecto.

```
nano .gitlab-ci.yml
```

En este fichero voy a añadir el siguiente contenido básico (si no pusiéramos el apartado `stages` al principio los jobs se ejecutarían pero sin ningún orden concreto):

```
stages:
  - test
  - build
  - deploy

test-job1:
  stage: test
  script:
    - echo "Hola, $GITLAB_USER_LOGIN ! Vamos a probar que las variables creadas
    en la integración de Harbor funcionan correctamente"
    - echo "Usuario:" $HARBOR_USERNAME
    - echo "Host:" $HARBOR_HOST
    - echo "OCI:" $HARBOR_OCI
    - echo "Password:" $HARBOR_PASSWORD
    - echo "URL:" $HARBOR_URL
    - echo "Proyecto de Harbor:" $HARBOR_PROJECT

build-job:
  stage: build
  variables:
    DOCKER_IMAGE_NAME: nombre-imagen:tag
```

```

before_script:
  - echo "Aquí haríamos login"
script:
  - echo "Construcción de la imagen"
  - echo "Push de la imagen"

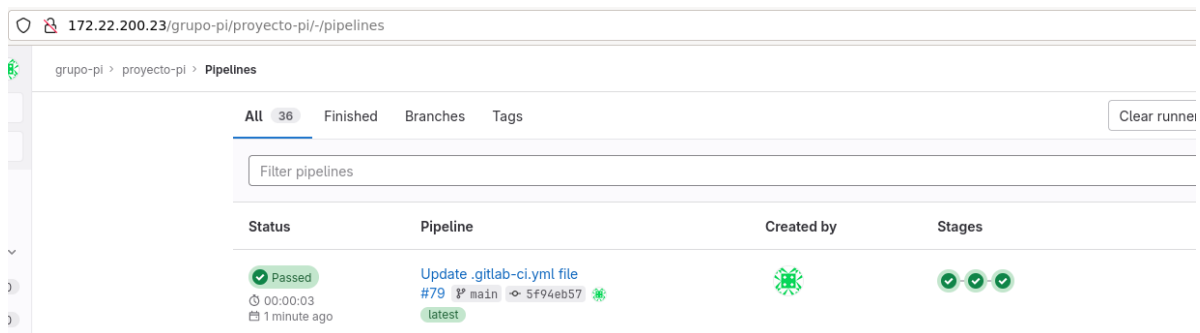
deploy-prod:
  stage: deploy
  script:
    - echo "This job deploys something from the $CI_COMMIT_BRANCH branch."
  environment: production
    
```

Para poder hacer el commit primero voy a identificarme con el usuario arantxa-pi.

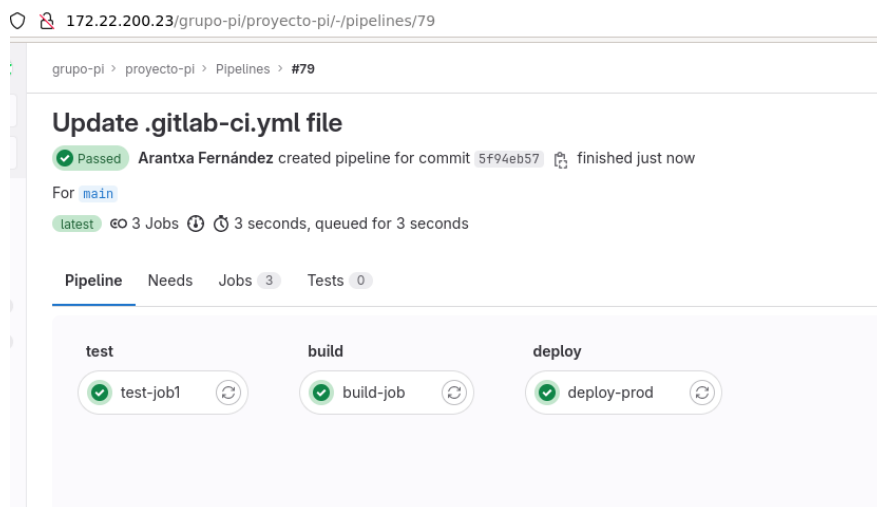
```
git config --global user.name "arantxa-pi"
```

Hago commit y push de los cambios.

Al crear el fichero .gitlab-ci.yml y hacer push del commit se lanzará el pipeline automáticamente, por lo que si vamos al proyecto en GitLab al apartado Build > Pipelines, podremos verlo.



Si hacemos click en Passed vemos más información de los jobs del pipeline.



**test-job1**

Passed Started 3 minutes ago by Arantxa Fernández

Search job log

```

1 Running with gitlab-runner 16.6.1 (f5da3c5a)
2   on Runner shell tUnDpUga, system ID: s_4dce067151d5
3   ✓ Preparing the "shell" executor
4     Using Shell (bash) executor...
5   ✓ Preparing environment
6     Running on gitlab...
7   ✓ Getting source from Git repository
8     Fetching changes with git depth set to 20...
9     Reinitialized existing Git repository in /home/gitlab-runner/builds/tUnDpUga/0/grupo-pi/proyecto-pi/.git/
10    Checking out 3bece5d3 as detached HEAD (ref is main)...
11    Skipping Git submodules setup
12  ✓ Executing "step_script" stage of the job script
13    $ echo "Hola, $GITLAB_USER_LOGIN ! Vamos a probar que las variables creadas en la integración de Harbor funcionan correctamente"
14    Hola, arantxa-pi ! Vamos a probar que las variables creadas en la integración de Harbor funcionan correctamente
15    $ echo "Usuario:" $HARBOR_USERNAME
16    Usuario: admin
17    $ echo "Host:" $HARBOR_HOST
18    Host: 172.22.201.183
19    $ echo "OCI:" $HARBOR_OCI
20    OCI: oci://172.22.201.183
21    $ echo "Password:" $HARBOR_PASSWORD
22    Password: [MASKED]
23    $ echo "URL:" $HARBOR_URL
24    URL: http://172.22.201.183
25    $ echo "Proyecto de Harbor:" $HARBOR_PROJECT
26    Proyecto de Harbor: pintegrado
27    Job succeeded

```

Vemos que los valores de las variables son los que habíamos configurado en la integración de Harbor y que la contraseña no aparece en texto plano.

## 6.4 Uso de comandos docker en el pipeline. Ejecutores del runner

Para activar el uso de los comandos docker en los jobs de GitLab CI/CD se puede usar:

- [Docker in Docker \(dind\)](#): el runner registrado utiliza el ejecutor Docker o el ejecutor Kubernetes. El ejecutor utiliza una imagen de contenedor de Docker, proporcionada por Docker, para ejecutar sus trabajos de CI/CD. La imagen de Docker incluye todas las herramientas de Docker y puede ejecutar el script del trabajo en el contexto de la imagen en modo privilegiado. Se debe utilizar Docker-in-Docker con TLS habilitado, que es compatible con los ejecutores compartidos de gitlab.com.
- [Enlazando el socket de Docker](#): se puede enlazar y montar (bind-mount) /var/run/docker.sock en el contenedor. Esto es incompatible con dind
- [Usando el ejecutor de shell](#): se puede configurar el runner para usar el ejecutor shell

En mi caso, utilizaré la última opción, el ejecutor de la shell. En esta configuración el gitlab-runner ejecuta los comandos Docker pero necesita permiso para hacerlo.<sup>2</sup>

Registro el runner con el ejecutor shell.

```
sudo gitlab-runner register -n \
  --url "http://172.22.200.23" \
  --registration-token 8xZnRpgRGL22g2SMRMW \
  --executor shell \
  --description "Runner shell"
```

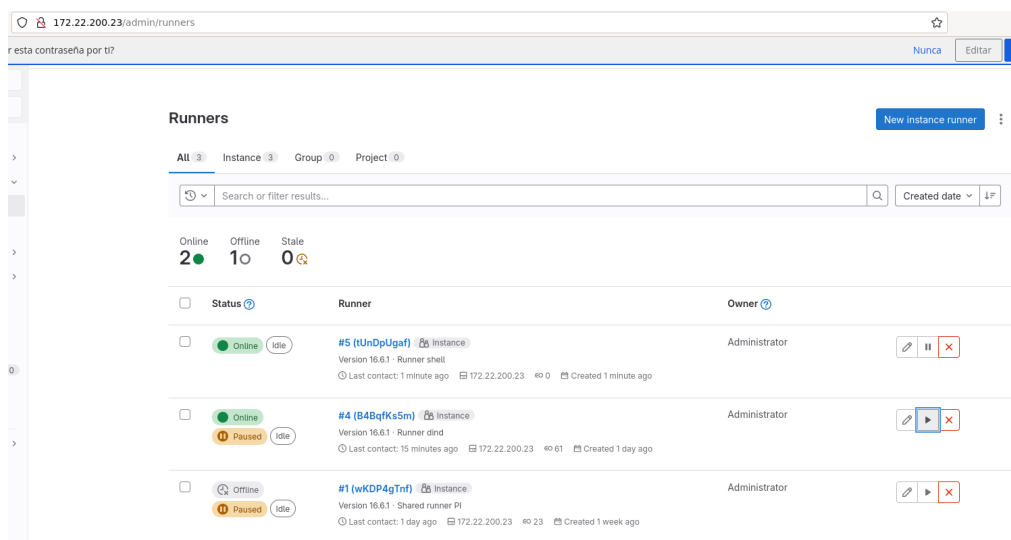
Añado el usuario gitlab-runner al grupo docker.

```
sudo usermod -aG docker gitlab-runner
```

Puedo verificar que gitlab-runner tiene acceso a docker con el siguiente comando.

```
sudo -u gitlab-runner -H docker info
```

Algo obvio pero a tener en cuenta es que se debe tener instalado Docker en el servidor donde se encuentra el runner para que funcionen los comandos.



Al ejecutar el comando de registro del runner se crea una entrada `/etc/gitlab-runner/config.toml`.

Pero, como no es el primer runner que se registra, solo pondré aquí la información relevante.

```
concurrent = 1
check_interval = 0
shutdown_timeout = 0
```

<sup>2</sup> Intenté hacerlo primero usando el ejecutor Docker in Docker, pero daba muchos errores al lanzar el pipeline. Algunos conseguí solventarlos, pero hubo un error concreto que no encontré solución. Adjunto en [Anexos](#) los pasos realizados y el error que no me dejó avanzar por esta vía.

```
[session_server]
  session_timeout = 1800
...
[[runners]]
  name = "Runner shell"
  url = "http://172.22.200.23"
  id = 5
  token = "tUnDpUgafsamWQANxaWo"
  token_obtained_at = 2023-12-12T15:37:45Z
  token_expires_at = 0001-01-01T00:00:00Z
  executor = "shell"
[runners.cache]
  MaxUploadedArchiveSize = 0
```

Ya solo habría que utilizar en el fichero `.gitlab-ci.yml` los comandos docker. Podemos comprobar que funciona usando `docker info`.

```
build_image:
  stage: build
  script
    - docker info
```

# 7 Demostración con aplicación Python

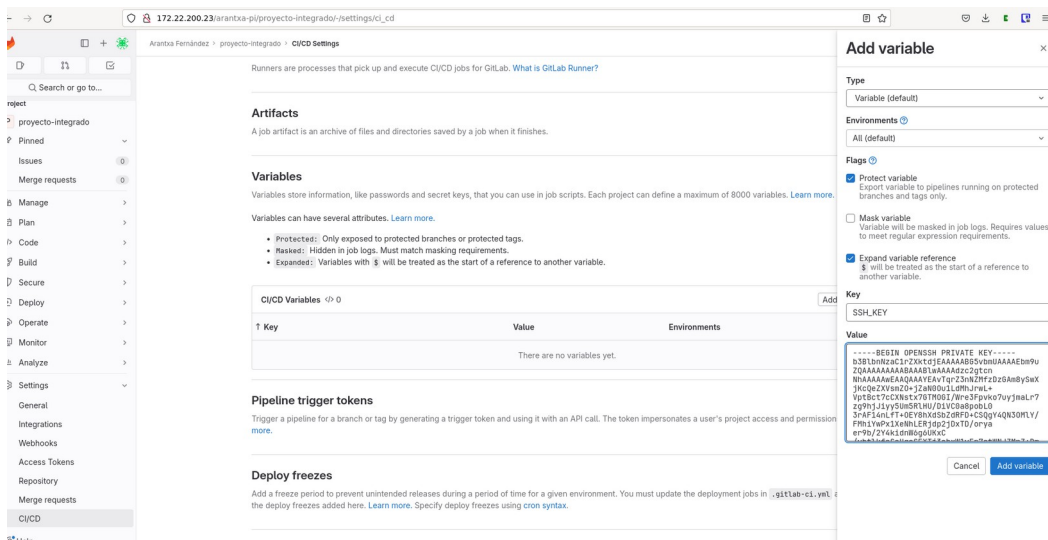
## 7.1 Pasos previos

Antes de empezar con la demo hay que tener en cuenta que para hacer el deploy he creado un nuevo servidor en Openstack llamado deployserver con la IP 172.22.201.134. He instalado Docker, he añadido el insecure-registries al fichero /etc/docker/daemon.json y he reiniciado el servicio docker. En el servidor gitlab he creado un par de claves con ssh-keygen. He copiado la clave pública a authorized\_keys del servidor deployserver.

Accedo desde el servidor gitlab usando el siguiente comando para comprobar que puedo acceder sin que me pida la contraseña del usuario:

```
ssh -i ~/.ssh/id_rsa root@172.22.201.134
```

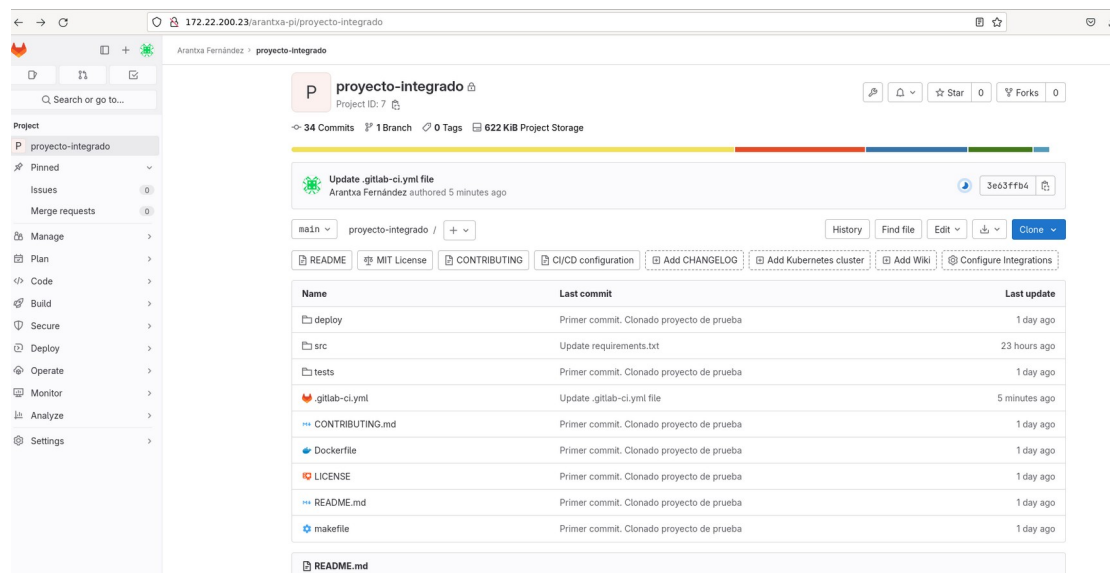
He creado un nuevo proyecto llamado proyecto-integrado en el que voy a crear una variable en GitLab que contenga la clave privada que se creó anteriormente. La variable no la crearé en el pipeline, ya que escribir la clave en texto plano puede suponer problemas de seguridad. La variable a nivel de proyecto se crea accediendo al proyecto en cuestión, que en mi caso se llama proyecto-integrado. Ahí vamos a Settings > CI/CD > Variables > Add variable. La guardamos como fichero.



Key	Value	Environments	Actions
SSH_KEY	*****	All (default)	[Edit] [Delete]

En proyecto-integrado he copiado un proyecto clonado de [otro repositorio de Github](#), el cual es una aplicación web Python Flask sencilla que te da información del sistema y lo monitoriza.

El enlace a mi proyecto en mi servidor GitLab [es este](#), pero como se debe tener acceso específico, he copiado toda la información a [otro repositorio de Github](#) público.



## 7.2 Fichero .gitlab-ci.yml

### Variables:

En el fichero .gitlab-ci.yml he definido una variable llamada IMAGE\_TAG con un valor específico, que es la combinación del nombre del proyecto (\$CI\_PROJECT\_NAME) y el ID del pipeline (\$CI\_PIPELINE\_ID). Esta variable se utilizará para etiquetar la imagen Docker construida.

### Stages y jobs:

Voy a añadir tres stages (test, build y deploy) y crearé tres jobs (run\_tests, build\_image y dev\_deploy). Cada job pertenecerá a una etapa, aunque podrían haber más jobs por cada etapa.

Los jobs realizan las siguientes tareas:

- run\_tests: pertenece a la etapa test y utiliza la imagen python:3.9-slim-buster. Antes de ejecutar el script, hace update e instala la herramienta make. Ejecuta el comando make test para realizar pruebas.
- build\_image: pertenece a la etapa build. Antes de ejecutar el script, hace login al registro Harbor utilizando las variables que se crearon al integrar Harbor en GitLab

(\$HARBOR\_URL, \$HARBOR\_USERNAME y \$HARBOR\_PASSWORD). En el script, construye una imagen Docker de la aplicación Python que tengo en el repositorio, la etiqueta, y luego la sube a mi registro Harbor usando el host, el nombre del proyecto y la imagen etiquetada (\$HARBOR\_HOST/\$HARBOR\_PROJECT/\$IMAGE\_TAG). Quedaría algo así: 172.22.201.183/pintegrado/proyecto-integrado:70

- dev\_deploy: pertenece a la etapa deploy. Antes de ejecutar el script, ajusta los permisos de la clave SSH (\$SSH\_KEY). Esto se hace para que tenga permisos de solo lectura, ya que si tiene más permisos saltará un error por esta razón. En el script, se realiza un despliegue al entorno de desarrollo, que es mi servidor deployserver (172.22.201.134). Se conecta mediante ssh y lanza los comandos necesarios para realizar las siguientes tareas:
  - Realiza la autenticación en mi registro Harbor.
  - Detiene y elimina contenedores Docker existentes.
  - Ejecuta un nuevo contenedor Docker a partir de la imagen que se subió al registro Harbor en el job anterior.

El fichero .gitlab-ci.yml quedaría de la siguiente forma:

```
variables:
  IMAGE_TAG: $CI_PROJECT_NAME:$CI_PIPELINE_ID

stages:
  - test
  - build
  - deploy

run_tests:
  stage: test
  image: python:3.9-slim-buster
  before_script:
#   - echo "Hello, $GITLAB_USER_LOGIN !"
  - apt-get update && apt-get install make
  script:
  - make test

build_image:
  stage: build
  before_script:
#   - docker info
#   - docker version
  - echo -n $HARBOR_PASSWORD | docker login -u $HARBOR_USERNAME --password-stdin $HARBOR_URL
#   - docker login -u $REGISTRY_USER -p $REGISTRY_PASS
  after_script:
  - docker logout $HARBOR_URL
  script:
  - docker build -t $IMAGE_TAG .
  - docker tag $IMAGE_TAG $HARBOR_HOST/$HARBOR_PROJECT/$IMAGE_TAG
```



```

- docker push $HARBOR_HOST/$HARBOR_PROJECT/$IMAGE_TAG
dev_deploy:
  stage: deploy
  before_script:
    - chmod 400 $SSH_KEY
  script:
    - ssh -o StrictHostKeyChecking=no -i $SSH_KEY root@172.22.201.134 "
      echo -n $HARBOR_PASSWORD | docker login -u $HARBOR_USERNAME --password-
stdin $HARBOR_URL &&
      docker ps -aq | xargs docker stop | xargs docker rm &&
      docker run -d -p 5000:5000 $HARBOR_HOST/$HARBOR_PROJECT/$IMAGE_TAG"

```

La primera vez que se corra el pipeline se tendrá que hacer sin los comandos:

```
docker ps -aq | xargs docker stop | xargs docker rm
```

Estos comandos se han agregado posteriormente para que pare el contenedor y lo borre antes de volverlo a crear con la nueva imagen. Si se ponen estos comandos la primera vez que se lanza el pipeline dará error, ya que no habrá nada que parar ni borrar.

## 7.3 Resultados obtenidos

Cada vez que se realice un commit se lanzará el pipeline automáticamente y se ejecutarán sus jobs. Por ejemplo haciendo un commit directamente desde Pipeline editor.

The screenshot shows the GitLab Pipeline Editor interface. The left sidebar contains navigation options like Project, Pinned, Issues, Merge requests, Manage, Plan, Code, Build, Pipelines, Jobs, Pipeline editor (selected), Pipeline schedules, Artifacts, Secure, Deploy, Operate, Monitor, Analyze, and Settings. The main area displays the pipeline configuration for the 'main' branch. The configuration includes variables for IMAGE\_TAG, stages for test, build, and deploy, and jobs for run\_tests, build\_image, and after\_script. The after\_script job contains the Docker commands to stop and remove containers before running the new image.

```

1 variables:
2   IMAGE_TAG: $CI_PROJECT_NAME:$CI_PIPELINE_ID
3
4 stages:
5   - test
6   - build
7   - deploy
8
9 run_tests:
10  stage: test
11  image: python:3.9-slim-buster
12  before_script:
13    # - echo "Hello, $GITLAB_USER_LOGIN !"
14    | - apt-get update && apt-get install make
15  script:
16    | - make test
17
18 build_image:
19  stage: build
20  before_script:
21    # - docker info
22    # - docker version
23    | - echo -n $HARBOR_PASSWORD | docker login -u $HARBOR_USERNAME --password-stdin $HARBOR_URL
24    # - docker login -u $REGISTRY_USER -p $REGISTRY_PASS
25  after_script:
26    | - docker logout $HARBOR_URL

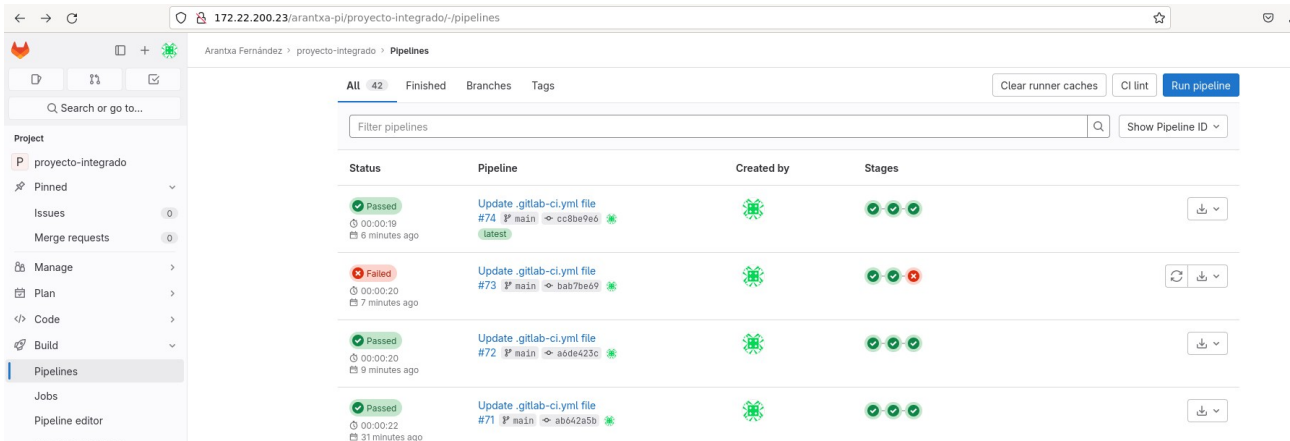
```

Commit message: Update .gitlab-ci.yml file

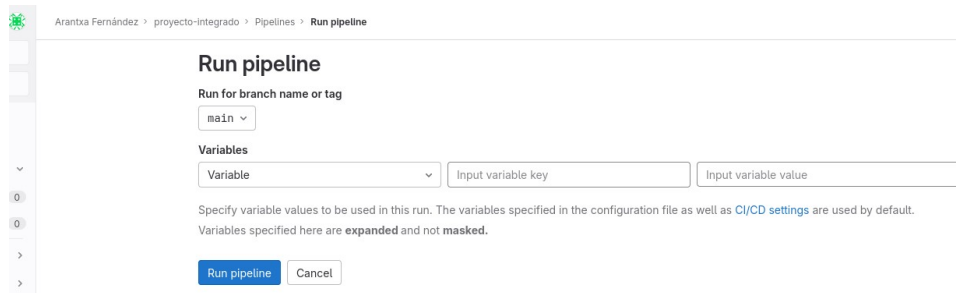
Branch: main

Buttons: Commit changes, Reset

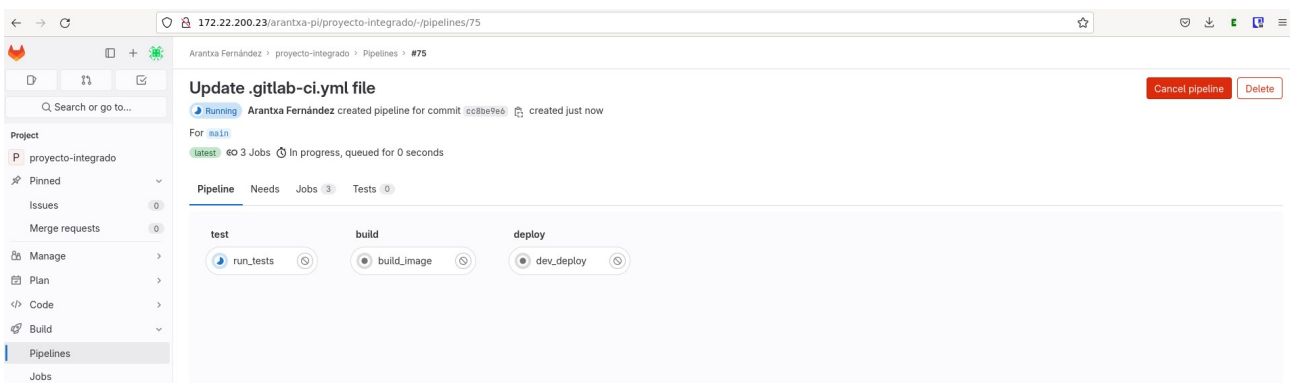
Si no tenemos que hacer commit también podemos lanzar el pipeline desde Build > Pipelines > Run pipeline.



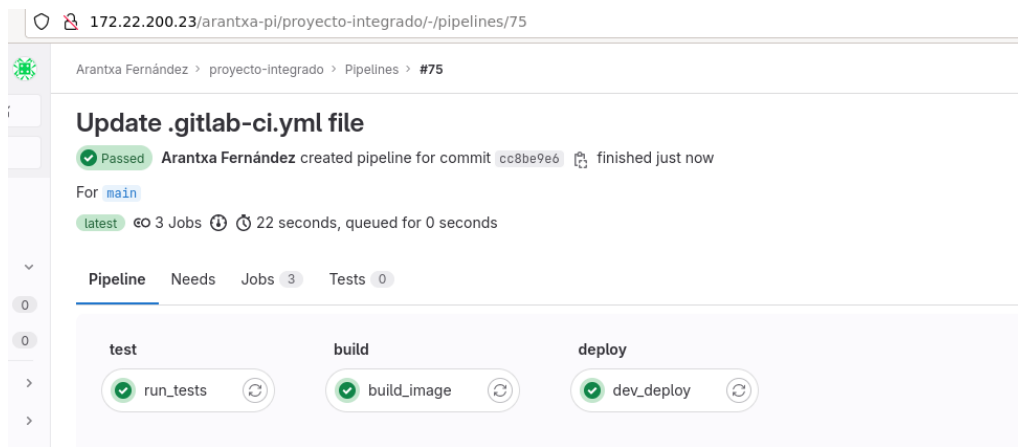
En la captura anterior vemos que el último pipeline lanzado tiene el ID 74. En la nueva ventana hacemos click en Run pipeline para que lance uno nuevo.



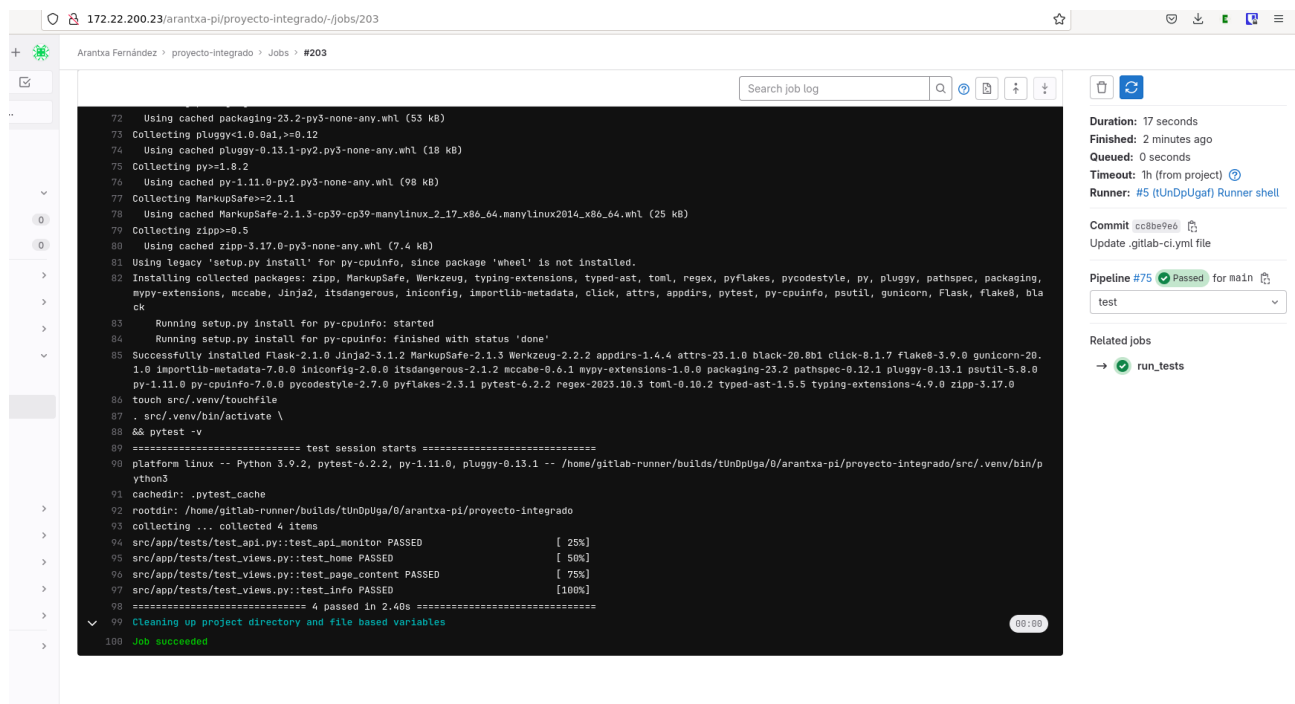
Comenzará a ejecutarse el pipeline directamente.



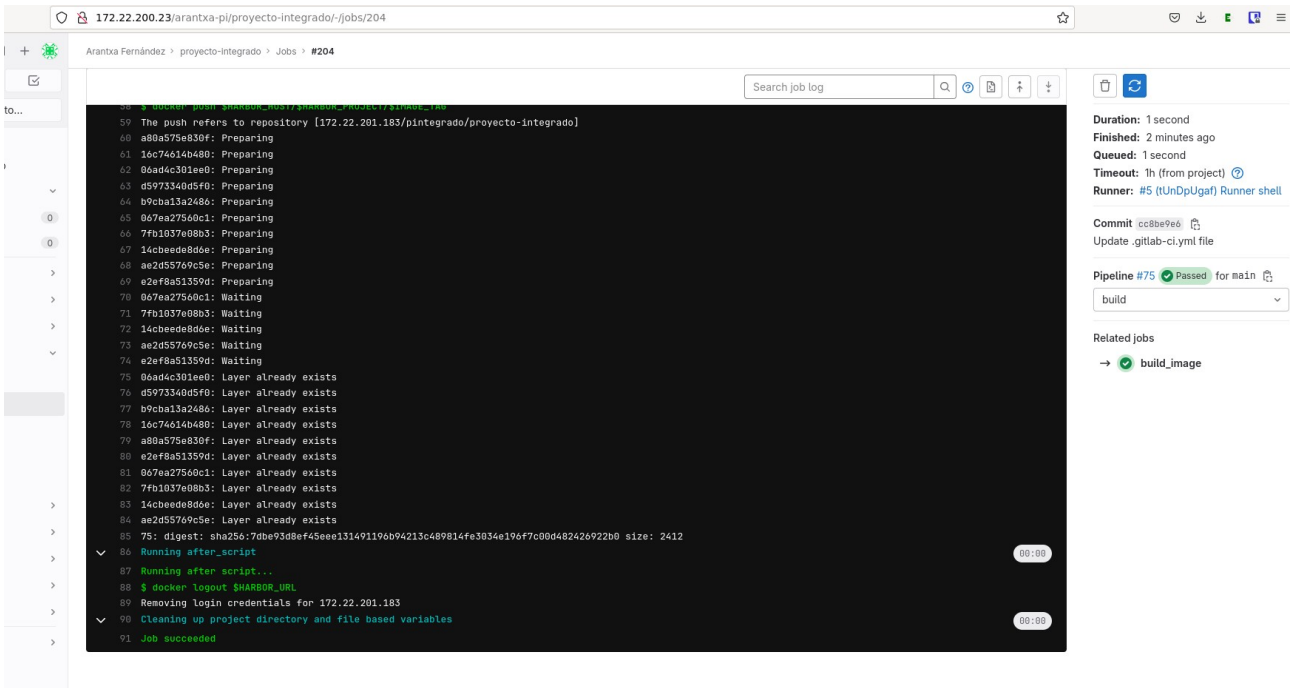
Cuando termine comprobamos que el pipeline 75 se ha ejecutado sin problemas.



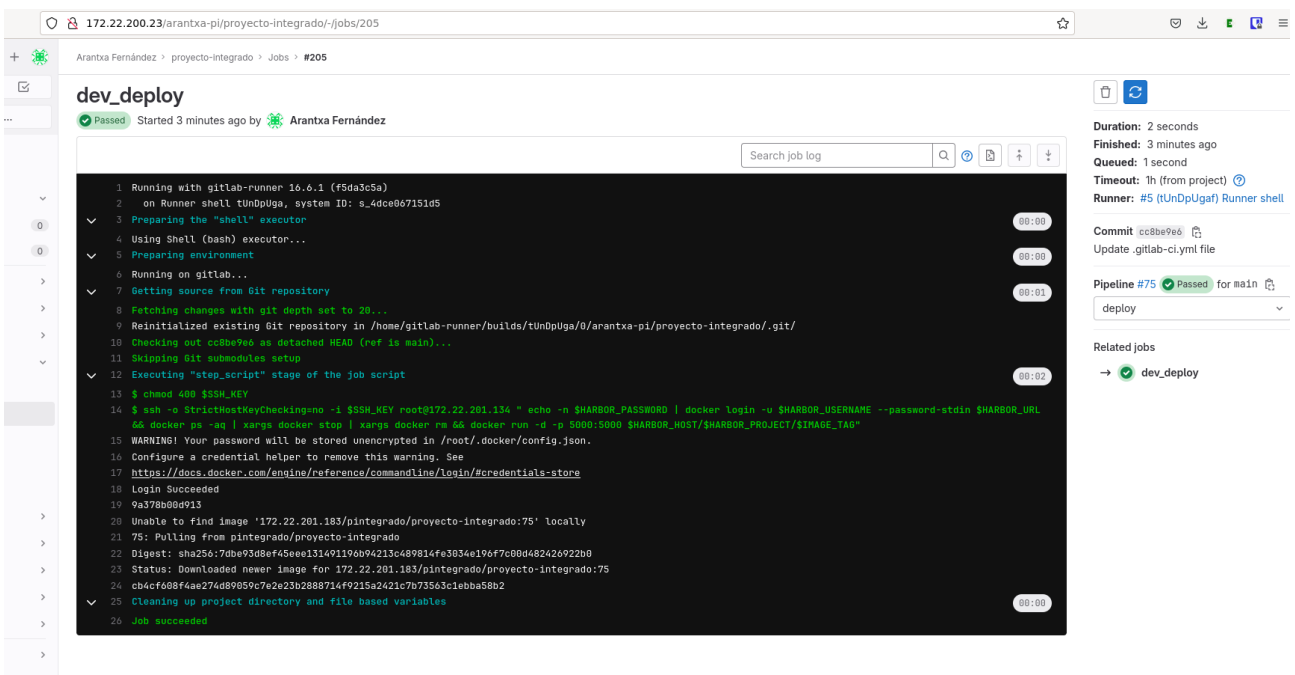
Vemos el job run\_tests:



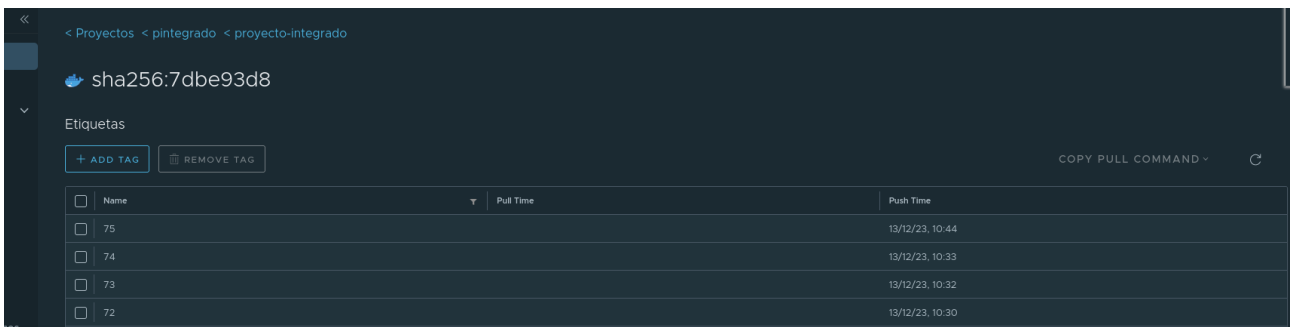
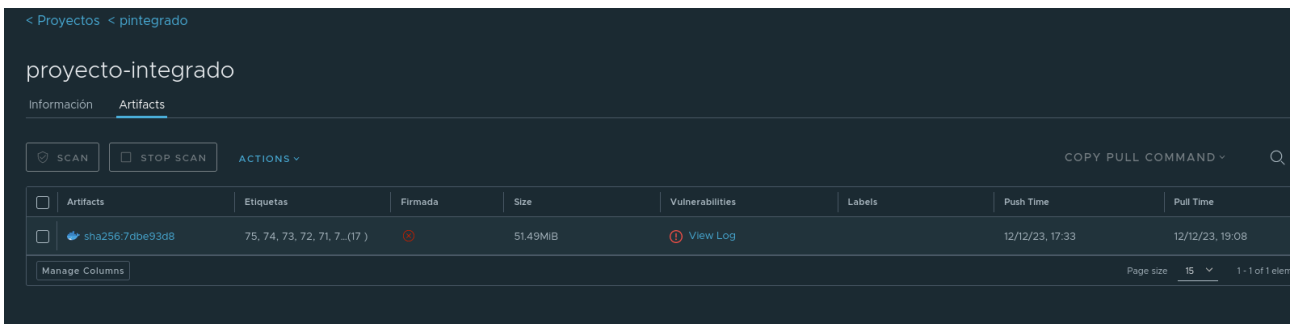
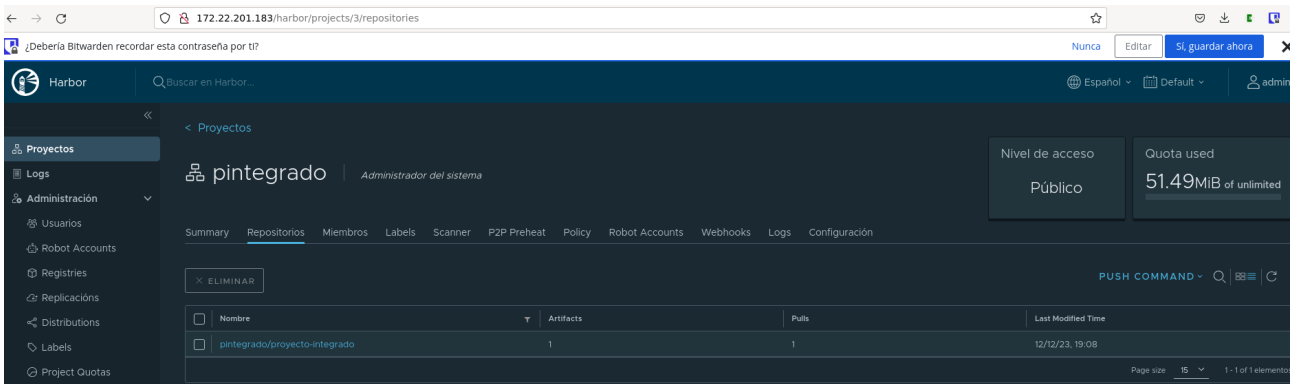
Volvemos atrás y vemos el job build\_image:



Volvemos atrás y vemos el último job dev\_deploy:



Como vemos, no han habido errores aparentes. Para comprobar que las tareas se han ejecutado correctamente accedido al servidor Harbor para ver la imagen Docker subida con el TAG 75, que era el ID del pipeline.



La subida de la imagen ha sido exitosa.

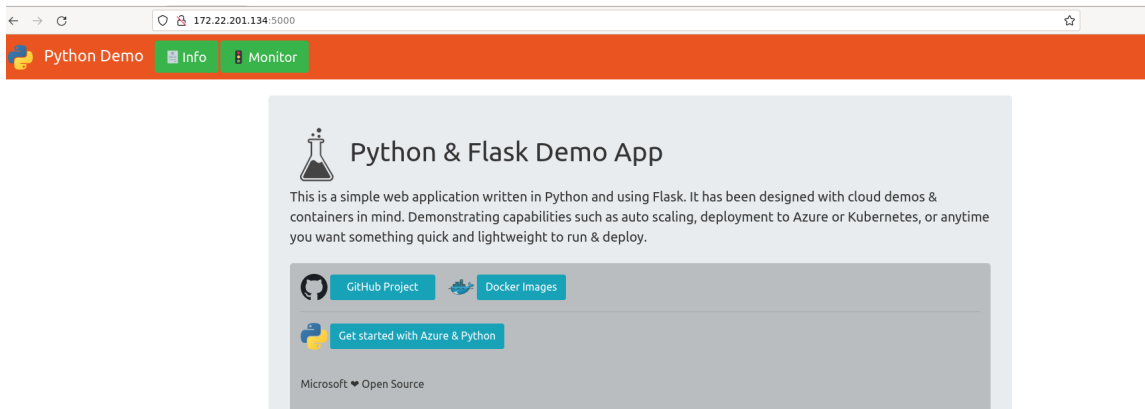
Por último, comprobamos que el despliegue en mi servidor deployserver se ha realizado correctamente. Accedo al servidor y compruebo que hay una imagen docker corriendo.

```

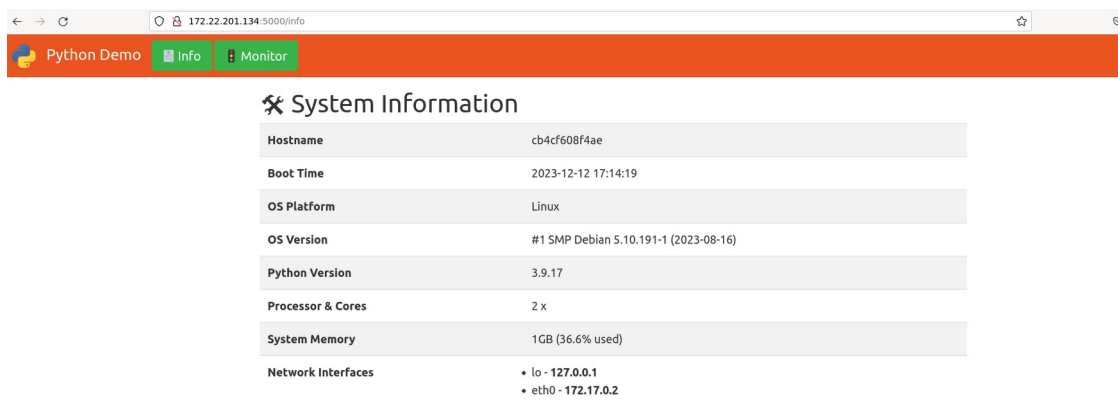
arantxa@deployserver:~$ sudo docker ps
[sudo] password for arantxa:
CONTAINER ID   IMAGE                                COMMAND                                CREATED
STATUS        PORTS                                NAMES                                UP 8 minutes ago
cb4cf608f4ae   172.22.201.183/pintegrado/proyecto-integrado:75   "gunicorn -b 0.0.0.0..."           8 minutes ago
Up 8 minutes   0.0.0.0:5000->5000/tcp                vigorous_nash
arantxa@deployserver:~$
    
```

El ID del contenedor docker es: cb4cf608f4ae

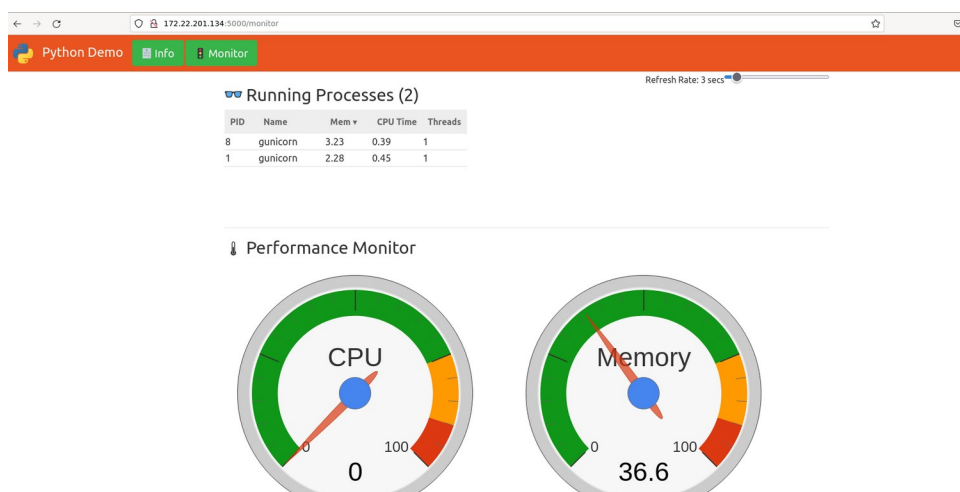
Accedo en el navegador a la IP 172.22.201.134 en el puerto 5000.



Si accedo a Info podemos ver que el hostname de la máquina corresponde con el nombre del contenedor docker (cb4cf608f4ae).



Y si accedo a Monitor vemos información de los procesos corriendo e información de la CPU, memoria, etc.



Tras realizar las comprobaciones podemos decir que se han cumplido los objetivos que se explicaban en el punto 1.2 de este proyecto ([Resultados que se esperan obtener](#)).

Los hitos conseguidos son:

- Harbor se ha integrado de manera exitosa en GitLab.
- Se lanza el pipeline sin problemas y se completa con éxito todas las tareas del pipeline.
- Se han ejecutado tests de manera exitosa en la aplicación.
- Se ha construido una imagen Docker y se ha subido al registro Harbor correctamente.
- Se ha desplegado correctamente la aplicación en el entorno de desarrollo, descargando y corriendo la imagen Docker subida anteriormente al registro Harbor. La aplicación funciona correctamente.

## **8 Conclusiones**

La incorporación exitosa de Harbor en el flujo de CI/CD de GitLab no solo ha potenciado la eficiencia, seguridad y confiabilidad en el ciclo de desarrollo de software, sino que también ha abordado de manera destacada dos problemáticas fundamentales en el ámbito de la informática.

En primer lugar, la adopción de Harbor como registro de imágenes Docker ha resuelto de manera efectiva la gestión de imágenes. Al proporcionar un repositorio centralizado, se ha logrado una optimización significativa en la administración y distribución de imágenes Docker. Este enfoque centralizado garantiza la consistencia y disponibilidad de las versiones de la aplicación, abordando así uno de los problemas cruciales en el desarrollo de software moderno.

En segundo lugar, la implementación de GitLab CI/CD ha abordado la problemática de la gestión de código. La automatización eficiente de todo el ciclo de vida del proyecto, desde la ejecución de pruebas hasta la implementación en entornos de desarrollo, no solo ha reducido errores humanos y mejorado la consistencia, sino que también ha facilitado la gestión y control de las imágenes creadas, acelerando el tiempo de entrega. La combinación de GitLab CI/CD y Harbor ha proporcionado un mecanismo seguro y preciso para controlar quién puede acceder y modificar los artefactos de construcción, elevando así el nivel de seguridad en el proceso.

El uso de imágenes Docker almacenadas en Harbor facilita el despliegue rápido y reproducible en entornos de desarrollo. La capacidad de versionar imágenes permite mantener un historial claro de las implementaciones y facilita la reversión en caso de problemas.

En resumen, la integración exitosa de Harbor en GitLab CI/CD no solo ha mejorado la eficiencia operativa y la seguridad, sino que también ha abordado de manera efectiva dos grandes desafíos en el ámbito de la informática: la gestión de código y la gestión de imágenes Docker. Estos resultados positivos respaldan la elección de esta configuración para futuros proyectos y subrayan la importancia de la integración continua en el desarrollo de software moderno.

Cabe mencionar que este proyecto es una base sólida para construir un entorno de desarrollo eficiente y robusto. Sin embargo, existen oportunidades infinitas para mejorarlo y llevarlo a un nivel superior. Por ejemplo, se podría considerar desplegar Harbor o Gitlab en un clúster Kubernetes



utilizando Helm. Esta medida no solo garantizaría la alta disponibilidad, sino que también simplificaría la administración y escalabilidad del sistema, facilitando futuras expansiones y alineándose con prácticas de GitOps.

Otras propuestas de mejora son las siguientes:

- Se podrían explorar otras integraciones, como Maven o Artifactory, lo que podría mejorar significativamente la gestión de dependencias y artefactos en el proceso de desarrollo. Esto proporcionaría una mayor flexibilidad y compatibilidad con diversas tecnologías.
- Implementar notificaciones y avisos a través de correo electrónico, Slack o Jira sería fundamental para mantener a los equipos informados sobre el estado de los pipelines y despliegues. Esta mejora fomentaría una comunicación más eficiente y permitiría una respuesta más rápida ante posibles problemas.
- La inclusión de pruebas unitarias en el proceso de CI/CD es otra área de mejora a considerar. Esto fortalecería la calidad del código y aumentaría la robustez de la aplicación, contribuyendo a la detección temprana de posibles problemas.
- Asimismo se podrían realizar pruebas usando escáneres de vulnerabilidades en las imágenes que se suben a Harbor, aumentando así la seguridad y fiabilidad de lo que se sube.
- En Harbor se podrían aplicar políticas de retención, es decir, que se almacenen imágenes concretas y se borren las que no sean necesarias (por problemas de seguridad, por ser repetidas, etc.).
- Mover las bases de datos de Harbor y GitLab (PostgreSQL) a servidores externos mejorarían la seguridad y simplificarían el mantenimiento, reduciendo la carga en los servidores principales.
- Es crucial considerar la implementación de certificados para HTTPS (que no se ha hecho en este proyecto), proporcionando una capa adicional de seguridad y protegiendo la integridad de los datos transferidos entre los sistemas.
- La configuración del DNS es algo esencial (tampoco se ha realizado en este proyecto) para garantizar una resolución de nombres de dominio correcta y mejorar la accesibilidad del sistema. Esto contribuiría a una experiencia de usuario más fluida y consistente.

Estas propuestas no solo buscan optimizar el presente, sino que también preparan el terreno para futuras mejoras y expansiones continuas en el flujo de trabajo de CI/CD, asegurando un entorno de desarrollo ágil, seguro y altamente eficiente.

## 9 Bibliografía y enlaces de interés

Web Oficial de Harbor:

<https://goharbor.io/docs/1.10/install-config/>

Web Oficial de Gitlab:

[https://docs.gitlab.com/ee/topics/build\\_your\\_application.html](https://docs.gitlab.com/ee/topics/build_your_application.html)

Use Self-hosted Gitlab to build and deploy images to Harbor:

<https://number1.co.za/use-gitlab-to-build-and-deploy-images-to-harbor/>

Cómo instalar el Registro de Imágenes Docker de Harbor en Ubuntu 22.04:

<https://howtoforge.es/como-instalar-el-registro-de-imagenes-docker-de-harbor-en-ubuntu-22-04/>

CI/CD Gitlab with Harbor Registry (StackOverflow):

<https://stackoverflow.com/questions/66326659/ci-cd-gitlab-with-harbor-registry>

Configurar GitLab Runner Desde 0:

<https://tehuel.blog/posts/configurar-gitlab-runner-desde-0/>

Debian gitlab and gitlab-runner installation tutorial:

<https://gist.github.com/e-cite/b6854548145fdb0e2e3dbce7fbb4f253>

GitLab Tutorial For Beginners | Build and Push Docker Image to GitLab Container Registry (Youtube):

<https://www.youtube.com/watch?v=AR29V1wWjjk>

GitLab CI CD Tutorial for Beginners [Crash Course] (Youtube):

<https://www.youtube.com/watch?v=qP8kir2GUgo&t=1587s>

Connecting Gitlab with Harbor for automated token issuing:

<https://kvaps.medium.com/connecting-gitlab-with-harbor-for-automated-token-issuing-6446f58269a7>

GitLab Roles – How to define Permissions:

<https://www.bitslovers.com/gitlab-roles/>

Bringing DevOps on Premises - Gitlab and Harbor (Youtube):

<https://www.youtube.com/watch?v=A6hzn9tUoK4>

## 10 Anexos

### 10.1 Problemas encontrados con Docker in Docker

Registro de un runner con ejecutor docker in docker:

```
sudo gitlab-runner register -n \
  --url "http://172.22.200.23/" \
  --registration-token xxxREGISTRATION_TOKENxxx \
  --executor docker \
  --description "Runner dind" \
  --docker-image "docker:24.0.7" \
  --docker-privileged \
  --docker-volumes "/certs/client"
```

Este comando registra un nuevo ejecutor para usar la imagen docker:24.0.7 (se debe especificar una versión, no es recomendable usar latest). Para iniciar los contenedores de compilación y servicio, utiliza el modo privilegiado. Para utilizar Docker-in-Docker, siempre se debe usar privileged = true en los contenedores Docker.

Este comando, además, monta /certs/client para el servicio y el contenedor de compilación, que es necesario para que el cliente Docker use los certificados en ese directorio.

En el fichero .gitlab-ci.yml se debe especificar los siguiente:

```
default:
  image: docker:24.0.7
  services:
  - docker:24.0.7-dind
```

Tras el registro correcto del runner, el fichero .gitlab-ci.yml quedaría como se muestra a continuación. He dejado comentado algunos de los problemas que tuve y lo que intenté.

```
stage: build
image: docker:24.0.7
services:
  - name: docker:24.0.7-dind
#   alias: docker
#   command: ["--insecure-registry=0.0.0.0/0"]
variables:
  DOCKER_TLS_CERTDIR: "/certs"
# Las siguientes dos variables solo se utilizan si no usamos TLS
# DOCKER_HOST: tcp://docker:2375
# DOCKER_DRIVER: overlay2
```

```

before_script:
#   Al principio me daba error por no tener configurado el DNS para que
#   resolviera la url de harbor
#   Intenté hacer la resolución directa añadiendo al fichero /etc/hosts pero al
#   haber tantas capas de abstracción seguía dando fallo
#   Hasta que no cambié el hostname de harbor.arantxa.gonzalonazareno.org a su ip
#   y dejó de dar ese error
#   - echo "172.22.201.183 harbor.arantxa.gonzalonazareno.org" | tee -a
#   /etc/hosts
#   - cat /etc/hosts
#   - ping -c 1 8.8.8.8
#   - ping -c 1 $HARBOR_HOST
#
#   También intenté meter el insecure-registries de la siguiente forma, como se
#   explica en la web oficial de Gitlab
#   Pero no funcionaba, no sé la razón. No funcionó hasta que puse el comando
#   directamente en services
#   - mkdir -p ~/.docker
#   - echo "{\"insecure-registries\": [\"0.0.0.0/0\"]}" > ~/.docker/config.json
#   - cat ~/.docker/config.json
#   - docker info
#   - docker version
#
#   Después de los cambios anteriores ya me dejó hacer login
#   - echo -n $HARBOR_PASSWORD | docker login -u $HARBOR_USERNAME --password-stdin
#   $HARBOR_URL
#   - docker login -u $HARBOR_USERNAME -p $HARBOR_PASSWORD $HARBOR_URL

after_script:
- docker logout $HARBOR_URL

script:
# - echo "Hello, $GITLAB_USER_LOGIN !"
- docker build -t $IMAGE_TAG .
- docker push $HARBOR_URL/$HARBOR_PROJECT/$IMAGE_TAG

stage: deploy
  script:
    - echo "This job deploys something from the $CI_COMMIT_BRANCH branch."

```

El error que no conseguí solventar es el siguiente:

```

ERROR:         failed          to          do          request:          Head
"https://registry-1.docker.io/v2/library/python/manifests/3":    net/http:    TLS
handshake timeout

```

grupo-pl > proyecto-pl > Jobs > #141

🗑️ 🔍 🔄

```

19 Using docker image sha256:aa93deb4ad1bd89a24411a139b6867bd175cfc9e0e8e87bfc69911f99943d for docker:24.0.7 with digest docker@sha256:c98e58d3078478fc59bda
   af802340fd25c1db628756d7b7f74e180c56bba9589 ...
20 $ echo -n $HARBOR_PASSWORD | docker login -u $HARBOR_USERNAME --password-stdin $HARBOR_URL
21 WARNING! Your password will be stored unencrypted in /root/.docker/config.json.
22 Configure a credential helper to remove this warning. See
23 https://docs.docker.com/engine/reference/commandline/login/#credentials-store
24 Login Succeeded
25 $ docker build -t $IMAGE_TAG .
26 #0 building with "default" instance using docker driver
27 #1 [internal] load .dockerignore
28 #1 transferring context:
29 #1 transferring context: 28 done
30 #1 DONE 0.2s
31 #2 [internal] load build definition from Dockerfile
32 #2 transferring dockerfile: 702B done
33 #2 DONE 0.2s
34 #3 [internal] load metadata for docker.io/library/python:3
35 #3 ERROR: failed to do request: Head "https://registry-1.docker.io/v2/library/python/manifests/3": net/http: TLS handshake timeout
36 -----
37 > [internal] load metadata for docker.io/library/python:3:
38 -----
39 WARNING: current commit information was not captured by the build: git was not found in the system: exec: "git": executable file not found in $PATH
40 Dockerfile:1
41 -----
42 1 | >>> FROM python:3
43 2 | WORKDIR /usr/src/app
44 3 | MAINTAINER Arantxa Fernández Morat0 'ara.fer.morat@gmail.com'
45 -----
46 ERROR: failed to solve: python:3: failed to do request: Head "https://registry-1.docker.io/v2/library/python/manifests/3": net/http: TLS handshake timeout
47 Running after_script
48 Running after script...
49 $ docker logout $HARBOR_URL
50 Removing login credentials for 172.22.201.183
51 ERROR: Job failed: exit code 1

```

Duration: 27 seconds

Finished: 2 days ago

Queued: 1 second

Timeout: 1h (from project)

Runner: #4 (B4BqfKs5m) Runner dind

Commit e70181a9

Update .gitlab-ci.yml file

Pipeline #52 Failed for main

build

Related jobs

→ build-job

Intenté hacer pull de la imagen Python directamente, pero seguía apareciendo el mismo error.

He leído que podía deberse a problemas de velocidad en las conexiones de red (MTU), problemas de certificados o del proxy usado. En esta pregunta de Openstack se habla de todo esto: <https://serverfault.com/questions/908141/docker-pull-tls-handshake-timeout>

Como estaba dedicando mucho tiempo a resolver el problema, sin tener éxito, decidí hacerlo con un runner con ejecutor shell, el cual es más inseguro, pero más simple de usar.

Webs oficiales de Gitlab interesantes de revisar si se quiere investigar más sobre el tema:

[https://docs.gitlab.com/ee/ci/docker/authenticate\\_registry.html](https://docs.gitlab.com/ee/ci/docker/authenticate_registry.html)

<https://docs.gitlab.com/runner/executors/docker.html>

[https://docs.gitlab.com/ee/ci/docker/using\\_docker\\_build.html#use-docker-in-docker](https://docs.gitlab.com/ee/ci/docker/using_docker_build.html#use-docker-in-docker)