GitLab - CI

Índice de contenidos

1. Entorno

Este tutorial está escrito usando el siguiente entorno:

- Lenovo Thinkpad L15
- Intel Core i7 64GB
- Ubuntu 20.04

2. Introducción

Desde el año 2014 venimos hablando de GitLab, únicamente como alternativa para tener repositorios privados de Git de forma gratuita, pero es espectacular el cambio que ha dado en estos últimos 4 años, convirtiéndose en una herramienta que prácticamente es autosuficiente para cubrir con todo el ciclo de integración y despliegue continuo, incluso en su versión community que puedes disfrutar si lo instalas on-premise, es decir, en tu propio servidor.

La otra opción para disfrutar de GitLab es utilizar su plataforma cloud GitLab.com, pero para sacarle el máximo partido te aconsejo que lo hagas on-premise.

Esto te va a ofrecer de forma gratuita una herramienta que te va a permitir issue tracking al estilo Kanban, gestión de usuarios y grupos, dashboards con distintas métricas, wiki por proyecto, snippets de código al estilo Gist de GitHub, almacenamiento de variables secretas (muy útil para no subir a los repositorios las passwords que vamos utilizando), code review a través de merge request como las pull request de GitHub y, entre otras muchas cosas, un sistema completo de integración y despliegue continuo con gestión de entornos (integración, qa, uat, producción, …) y la posibilidad de integrar con Kubernetes. También cuenta con un registro privado para Docker, aunque para este tipo de repositorios yo siempre recomiendo Nexus 3.

En este tutorial nos vamos a centrar en cómo empezar a aprovechar la parte de CI de GitLab, vamos a ver cómo configurar los runners y las partes más importantes de un pipeline y la gestión de variables secretas.

3. Creación de un repositorio y configuración de runners

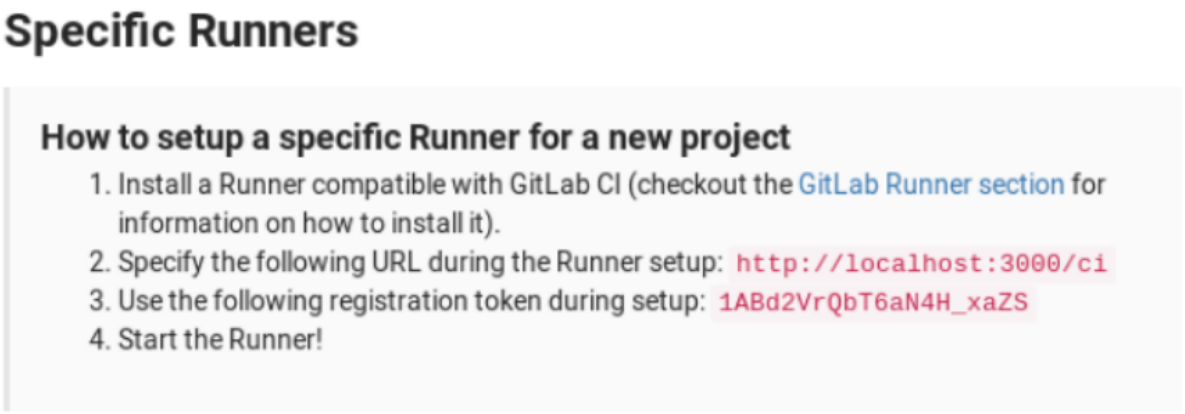
Partimos de que ya tenemos una instancia on-premise de GitLab, recalco lo de on-premise porque la versión gratuita del cloud no te va a dejar configurar runners y te tienes que «pegar» con los otros usuarios gratuitos para hacer uso de los runners compartidos, lo que hace que los procesos de pipeline sean más lentos.

Las distintas opciones de instalación las tenéis en la documentación oficial, que está bastante completa y actualizada. Teniendo en cuenta tener también una instancia de docker y docker-compose instalada en la misma máquina.

Selecciona la que mejor se ajuste, personalmente selecciono la opción de Ubuntu Server 16.04 en un contenedor LXD con privilegios para poder crear contenedores de docker y hacer uso de docker-compose.

Lo primero que tenemos que hacer es acceder con una cuenta que tenga los suficientes permisos para crear un nuevo repositorio. Para ello una vez dentro vamos a «New Project» y rellenamos la información del repositorio.

Una vez el respositorio está creado, lo seleccionamos y vamos a «Settings» –> «CI/CD» –> «Runner Settings» y en la sección «Specific Runners» vamos a encontrar la URL y el token necesarios para registrar los nuevos runners asociados al repositorio que son necesarios para poder ejecutar las distintas tareas que definamos en el pipeline.



Una vez que tenemos localizada esta información. Vamos a acceder al terminal de la máquina donde esté corriendo GitLab para instalar el gestor de runners. En Ubuntu esto es tan sencillo como ejecutar:

```
Shell
```

```
1   $> wget
    https://gitlab-runner-downloads.s3.amazonaws.com/master/deb/gitlab-runner_amd6
2   4.deb

    $> sudo dpkg -i gitlab-runner_amd64.deb
```

Inicialmente para cada repositorio vamos a tener un runner de tipo docker y otro runner de tipo shell.

Ahora para registrar nuestro primer runner de tipo docker vamos a ejecutar:

```
Shell
```

```
1   $> sudo gitlab-runner register
```

Este comando nos va a lanzar una serie de preguntas:

- **Please enter the gitlab-ci coordinator URL (e.g. https://gitlab.com)**: contestamos con la URL de la imagen

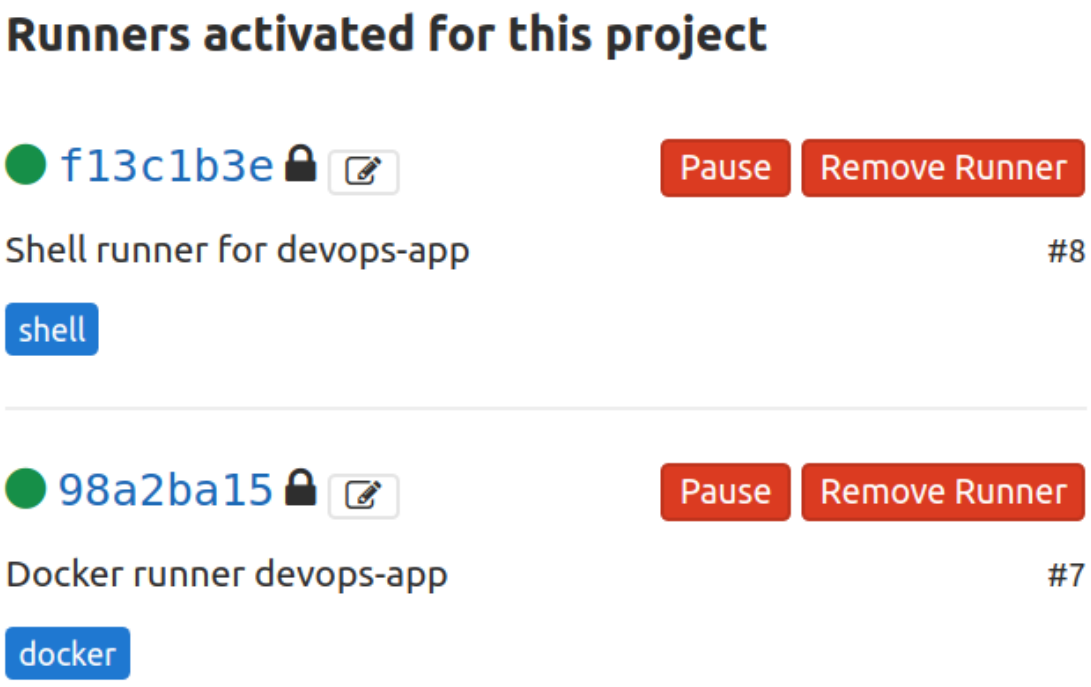
- **Please enter the gitlab-ci token for this runner**: contestamos con el token de la imagen
- **Please enter the gitlab-ci description for this runner**: le damos una descripción al runner
- **Please enter the gitlab-ci tags for this runner (comma separated)**: le asociamos un tag, el cuál se va a utilizar para determinar el runner a utilizar en los jobs. Por ejemplo, docker; pero podría ser cualquier otro descriptivo.
- **Whether to run untagged jobs [true/false]**: con true permitimos que corra jobs que no tengan tags. Mejor false.
- **Whether to lock Runner to current project [true/false]**: con true bloqueamos este runner para este proyecto. Mejor true.
- **Please enter the executor: ssh, docker+machine, docker-ssh+machine, kubernetes, docker, parallels, virtualbox, docker-ssh, shell**: seleccionamos el tipo de runner, ahora mismo ponemos docker.
- **Please enter the Docker image (eg. ruby:2.1)**: nos insta a que pongamos una imagen de docker por defecto. Vamos a poner alpine:latest

Este runner nos va a permitir definir imágenes de docker que serán las encargadas de ejecutar los distintos comandos en su interior. Esto tiene la ventaja de que no necesitamos instalar en la máquina ningún runtime de Java o Maven o NodeJS, solo tenemos que utilizar las imágenes apropiadas a cada tarea.

Para la creación del runner de tipo shell, volvemos a ejecutar el comando register pero está vez le indicamos un tag y descripción distintos y le decimos que va a ser de tipo shell.

Este runner nos va a dar la flexibilidad de poder ejecutar cualquier tipo de tarea que pudiéramos ejecutar en la shell de la máquina, por ejemplo, ejecutar docker o docker-compose, o comandos como wget, etc… de todos modos debemos minimizar su uso e intentar hacerlo todo con runners de docker.

Si todo es correcto, al refrescar la página, vemos que aparece una lista con los runners instalados.



Dentro del fichero /etc/gitlab-runner/config.toml podemos definir otros parámetros de configuración más avanzados para cada uno de los runners,

aquí se encuentran todos los posibles: **https://gitlab.com/gitlab-org/gitlab-runner/blob/master/docs/configuration/advanced-configuration.md**

Uno de los más habituales es indicar que se le pase un determinado host a la imagen de docker del runner con el fin de que tenga visibilidad de otra máquina, donde por ejemplo, estemos corriendo Nexus o Sonarqube, para ello editamos este fichero y dentro de la sección [runners.docker] añadimos extra_hosts = [«nombre-host:ip-host”]. Además es útil añadir para todos los runners que su tamaño de log sea más grande con la propiedad output_limit = valor-deseado(2000000000)

Para que los cambios tengan efecto tenemos que reiniciar el servicio.

```
Shell

1   $> sudo systemctl restart gitlab-runner
```

4. Configuración de un pipeline

En GitLab el pipeline se configura en el fichero «.gitlab-ci.yml» que tiene que estar en la raíz del repositorio.

GitLab en cuanto detecta la presencia de este fichero comienza la ejecución del pipeline definido.

En la primera parte del fichero definimos los distintos stages o fases que componen nuestro pipeline. Por ejemplo:

YAML

```yaml
stages:
  - init
  - build
  - prepare-run
  - run-master
  - run-develop
  - run-releases
```

A continuación tenemos que definir los jobs que se van a ejecutar en cada stage del pipeline, para asociar un job a su stage utilizamos la propiedad «stage» teniendo en cuenta que si dos jobs tienen la misma propiedad «stage» se van a ejecutar en paralelo siempre que el número de runners lo permita. En este caso definimos dos jobs: el primero que se va a ejecutar en un runner que tenga tag docker; y el segundo que se va a ejecutar en un runner que tenga tag shell.

YAML

```yaml
version:
  stage: init
  image: alpine:latest
  script:
    - whoami
  tags:
    - docker

build:
  stage: build
  script:
    - docker-compose -f docker/test.yml up -d
    - mvn clean deploy
```

```
14      - docker-compose -f docker/test.yml down
15    tags:
16      - shell
```

Dentro del job definimos tareas a ejecutar. Estas tareas se definen bajo la propiedad «script» como vemos en el ejemplo de arriba. El job de tipo docker ejecutará el comando dentro del contenedor que venga definido en la propiedad «image», teniendo en cuenta que GitLab va a copiar por nosotros todo el contenido del repositorio en la raíz del contenedor. En el segundo ejemplo, las tareas las ejecutamos en la shell de la máquina, es por ello, que en este caso tiene que tener una instancia de docker-compose y de maven para que el proceso se pueda llevar a cabo.

Puede ser que solo nos interese ejecutar un job definido cuando estemos en una determinada rama, para como en el ejemplo, ejecutar el despliegue de una release o de producción en función de si estamos en una rama de release o estamos en la rama master. Para eso utiilizamos la propiedad «only».

YAML

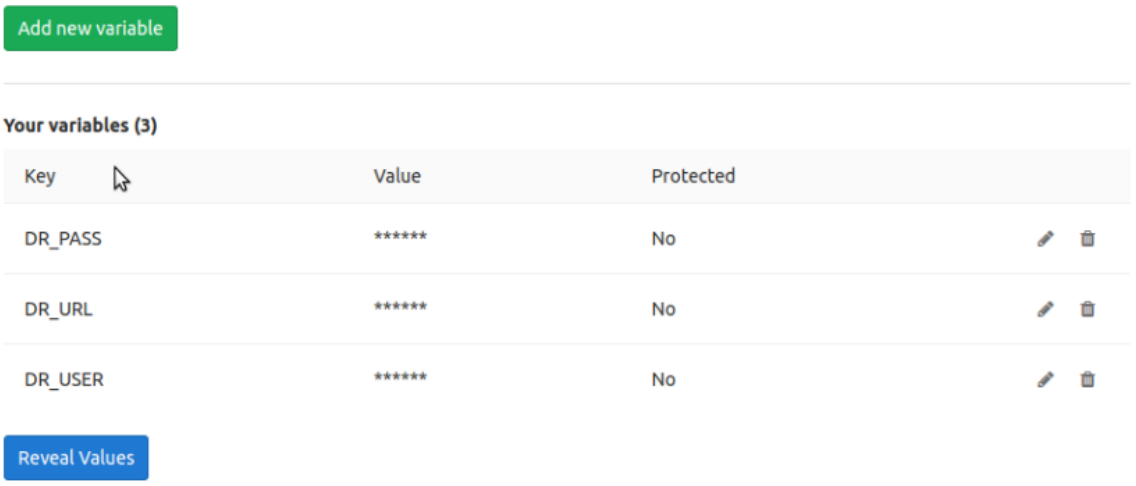
```
run-master:
  stage: run-master
  script:
    - docker-compose -f docker/master.yml down
    - docker-compose -f docker/master.yml up -d
  only:
    - master
  tags:
    - shell

run-releases:
  stage: run-releases
  script:
    - docker-compose -f docker/releases.yml down
    - docker-compose -f docker/releases.yml up -d
  only:
    - /^release.*$/
  tags:
    - shell
```

Almacenamiento y uso de secretos

Una de las cosas que tenemos que tener en cuenta a la hora de crear el pipeline, es que este fichero tiene que ser versionado, y por tanto, si en algún script hacemos uso de passwords éstas van a quedar públicas.

GitLab cuenta con una gestión de variables secretas, por proyecto, a las que podemos acceder a través de «Settings» — «CI / CD» — «Secret Variables»



Y simplemente podemos hacer uso de estas claves en nuestros jobs de esta forma:

YAML

```yaml
prepare-run:
  stage: prepare-run
  script:
    - docker login -u $DR_USER -p $DR_PASS $DR_URL
  tags:
    - shell
```

Para permitir el uso de imágenes almacenadas en un Docker privado, no basta con hacer el login sino que tenemos que definir una variable secreta especial que se tiene que llamar exactamente DOCKER_AUTH_REGISTRY y debe contener exactamente el contenido del fichero ~/.docker/config.json que puede tener el siguiente aspecto:

JavaScript

```
{
        "auths": {
                "url_registry:port": {
                        "auth": "YWRtaW46YWRtaW4xMjM="
                }
        }
}
```

Pasar ficheros entre jobs de distintos stages

Es posible que nos encontremos con que un job genera ciertos ficheros que otro job necesita y que no se encuentran subidos al control de versiones, dado que se pueden generar.

Como los jobs se ejecutan de forma independiente, incluso podrían ejecutarse en distintos contenedores, GitLab ofrece la forma de compartir esta información a través de la etiqueta “artifacts” al que le podemos asociar una lista de paths que queremos que se conserven para otros jobs o indicar que queremos almacenar todos los ficheros que estén en estado untracked. Por ejemplo, este es el caso de una compilación de un proyecto maven con un runner de tipo de docker, donde queremos que la carpeta «target» quede disponible para otros jobs, por ejemplo, el de pasar Sonarqube.

```
YAML
```

```yaml
build:
  stage: build
  image: maven:3-jdk-8-alpine
  services:
    - name: mysql:5.7
      alias: devops-db-test
  variables:
    MYSQL_ROOT_PASSWORD: $MYSQL_ROOT_PASSWORD
    MYSQL_DATABASE: 'devops'
    MYSQL_USER: 'devops'
    MYSQL_PASSWORD: $MYSQL_PASSWORD
    DATABASE_HOST: 'devops-db-test'
    DATABASE_PORT: '3306'
    DATABASE_NAME: 'devops'
    DATABASE_USER: 'devops'
    DATABASE_PASS: $DATABASE_PASS
    NEXUS_USER: $NEXUS_USER
    NEXUS_PASS: $NEXUS_PASS
    MAVEN_OPTS: -Dmaven.repo.local=/cache/maven.repository
  script:
    - mvn clean deploy --settings .m2/settings.xml
  artifacts:
    paths:
      - target
  tags:
    - docker
```

Y en los jobs que necesiten esta información hacemos uso de la propiedad «dependencies»

```
YAML
```

```
sonar:
  image: local.nexus.com:8083/devops/sonar-scanner:3.0.3.778-linux
  stage: qa
  variables:
    SONAR_URL: http://local.sonarqube.com:9000
    SONAR_PROJECT_KEY: $CI_PROJECT_NAME
    SONAR_PROJECT_NAME: $CI_PROJECT_NAME
  script:
    - APP_VERSION="$(grep -v '\[' version.log)"
    - /usr/bin/sonar-scanner-run.sh $APP_VERSION
  dependencies:
    - calculate-version
    - build
  tags:
    - docker
```

Creando entornos de ejecución

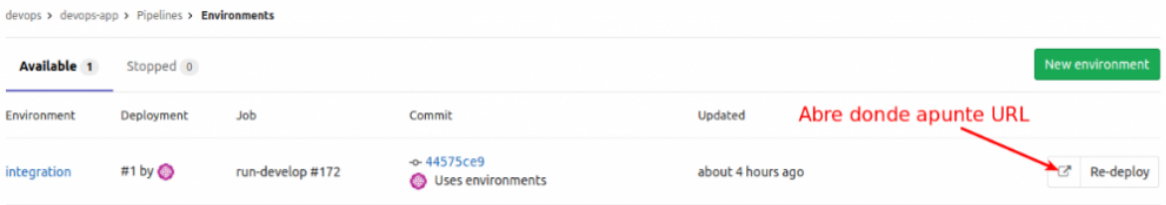
Los entornos nos permiten definir distintos estados de ejecución de la aplicación, por ejemplo, integración, qa, pre-producción, producción… para ello en los jobs de tipo deploy usamos la etiqueta «environment» que admite el nombre del entorno («name») y una URL donde se publica el resultado («url»).

En este ejemplo, vemos como gracias a docker-compose dejamos corriendo los contenedores necesarios para poder levantar la aplicación en el estado de integración, automáticamente cuando se suben cambios a la rama develop.

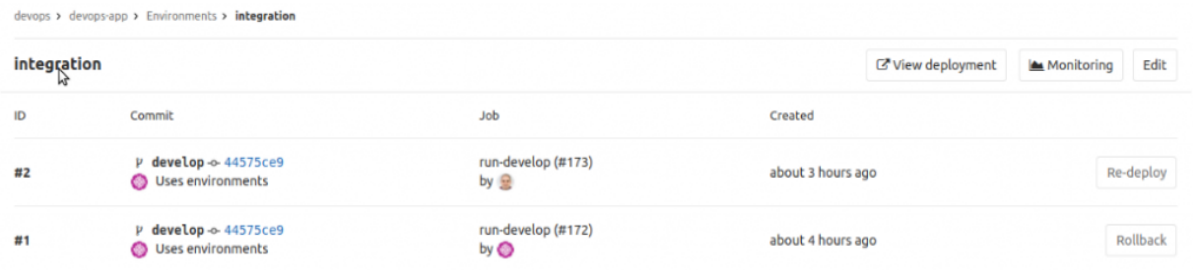
```
YAML
```

```
1   run-develop:
2     stage: run-develop
3     script:
4       - docker-compose -f docker/develop.yml down
5       - docker-compose -f docker/develop.yml up -d
6     only:
7       - develop
8     tags:
9       - shell
10    environment:
11      name: integration
12      url: http://devops-ci:8070/version
```

Si el entorno todavía no existe lo creará automáticamente, dejándolo reflejado en la sección «CI / CD » — «Environments»



También podemos hacer un re-despliegue pulsando en “Re-deploy”. Cada vez que hagamos un despliegue en el entorno se irá acumulando, el listado lo podemos ver pinchando en el enlace del nombre de la versión.



Lo que permite hacer rollback a cualquier otro despliegue anterior. El botón de “View deployment” siempre muestra el resultado actual.

Además si queremos que algún job solo se ejecute de forma manual, por ejemplo, el despliegue a producción, para hacer entrega continua en vez de despliegue continuo, podemos utilizar la etiqueta «when» con valor manual, como en este ejemplo:

```
YAML
```

```yaml
run-master:
  stage: run-master
  script:
    - docker-compose -f docker/master.yml down
    - docker-compose -f docker/master.yml up -d
  only:
    - master
  tags:
    - shell
  environment:
    name: produccion
    url: http://devops-ci:8090/version
  when: manual
```

Esto hace que dentro del pipeline, se muestra un play para arrancar estos jobs marcados como manual. Otros valores de «when» puede ser:

- **on_failure**: sólo se ejecuta el job cuando el último job antes de éste falla.
- **on_success**: sólo se ejecuta el job cuando todos los de antes han acabado con éxito. Es el que se pone por defecto.

- **always**: se ejecuta independientemente del resultado de los anteriores.

También podemos ignorar el resultado fallido de un job con la etiqueta «allow_failure» a true.

Validación de la sintaxis

Es interesante, antes de hacer la subida al repositorio, validar que la sintaxis del fichero es correcta, de lo contrario romperá la build indicando el error en el fichero. Para hacer esto vamos a la URL: http://url-gitlab/ci/lint, pegamos el contenido del fichero y pulsamos en «Validate».

5. Conclusiones

Como ves no es nada complicado hacer uso de GitLab para definir el pipeline de nuestros proyectos y exprimir más esta pieza de la arquitectura que nos va a permitir ir simplificando la arquitectura, eliminando la necesidad de Jenkins.