

**MANUAL DE CI/CD DE WATSON ASSISTANT**

TECNOlOGÍA

**HISTORIAL DE REVISIONES**

|  |  |  |  |
| --- | --- | --- | --- |
| **Revisión** | **Fecha** | **Responsable** | **Comentarios** |
| V1 | 22/08/2022 | Diego Guerrón | Creación de documento |

**CONTENIDO**

[1. OBJETIVO 2](#_Toc108521410)

[2. REQUISITOS PREVIOS. 2](#_Toc108521411)

[3. Análisis de despliegue de la Solución 2](#_Toc108521412)

[4. Configuración de entornos de integración 3](#_Toc108521413)

[4.1. IBM Cloud 3](#_Toc108521414)

[4.1.1. Creación de Skills ambiente productivo 3](#_Toc108521415)

[4.1.2. Obtención de Skills details 4](#_Toc108521416)

[4.2. Creación repositorio GitHub 5](#_Toc108521417)

[4.2.1. Edición de config.properties 6](#_Toc108521418)

[4.2.2. Creación del job en Jenkins 6](#_Toc108521419)

[4.3. Integración Jenkins con Github 7](#_Toc108521420)

[4.4. Crear usuarios en Jenkins y Administración de permisos y Roles. 9](#_Toc108521421)

[4.5. Complemento Role Strategy Plugin 10](#_Toc108521422)

[4.6. ScriptApproval 13](#_Toc108521423)

[5. Flujo de ejecución de la solución (pipeline) 14](#_Toc108521424)

**MANUAL DE CI/CD PARA CANAL DIGITAL**

## INTRODUCCIÓN

La rápida evolución del desarrollo de software, aunque ha sido una constante desde su nacimiento, ha sufrido un cambio exponencial en los últimos años. No estamos hablando sólo de cambios a nivel técnico con la aparición y evolución de nuevos lenguajes, framework y tecnologías, sino también a nivel de metodologías de desarrollo con nuevos paradigmas, procesos y prácticas. Todo ello siempre encaminado y con objetivo de generar un mejor software, más robusto ante el error y más seguro ante posibles ataques, y que pueda desarrollarse y adaptarse al cambio en el menor tiempo posible.

Los, conceptos como Agile, CI/CD o DevOps, cuyo objetivo es facilitar, mejorar y transformar los procesos “clásicos” de construcción y despliegue/entrega de aplicaciones, están a la orden del día dentro del panorama actual del desarrollo como estándares de facto dentro de la comunidad y de la industria del software. Muy ligado a ello, una tecnología, ya conocida pero que había pasado un tanto desapercibida hasta no hace tanto, como la de contenedores de software, ha aprovechado el momento para crecer, hacerse fuerte e integrarse completamente dentro del ecosistema como un elemento común más.

## OBJETIVO

La construcción de una plataforma estándar de ALM (Application Lifecycle Management) que permita poner en práctica la metodología de integración continua, con las máximas prestaciones de integración e interoperabilidad. Para ello, se pretende cubrir e integrar los métodos más comunes asociados a los entornos y procesos de integración continua mediante software que cubran las siguientes necesidades:

* Gestión y control de versiones de código fuente, elementos de configuración y desarrollo de software colaborativo.
* Centralización y almacenamiento de paquetes software.
* Revisión y evaluación de calidad de software.
* Orquestación y automatización de tareas para el despliegue de software.

Al final, el objetivo que se persigue es la automatización, integración y simplificación de los elementos y procesos más comunes involucrados en el desarrollo de software, se plantea que la solución sea capaz de que ante cambios del código fuente por parte del desarrollador, el proceso por el que debe pasar el software hasta su despliegue sea lo más transparente y automático posible para este.

Aunque ya se verá en detalle en el capítulo dedicado a la implementación técnica de la solución propuesta, planteamos dos escenarios u objetivos que puedan poner en práctica el entorno y ver que es una solución válida para ello:

• Generación, centralización y almacenamiento de un paquete o dependencia software.

• Generación, centralización, almacenamiento y despliegue de una aplicación web.

## REQUISITOS PREVIOS.

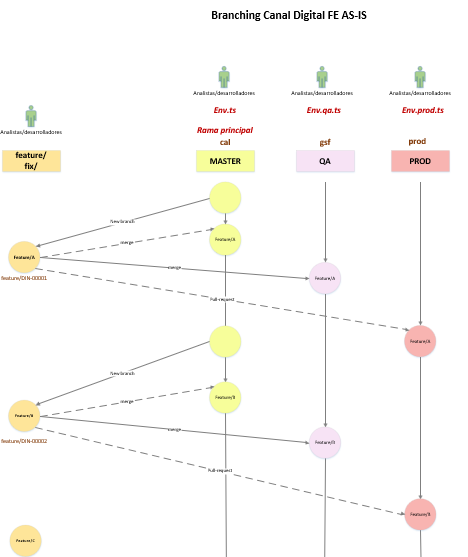
En resumen, para habilitar la integración continua mediante “Gitlab CI/CD” necesitaremos:

* Fichero. **gitlab-ci.yml** con las características necesarias
* **Gitlab** runner, instalado y configurado

## Análisis de despliegue de la Solución

## Situación actual del proceso de despliegue backend

1. Analista se crea una rama feature a partir de la rama master (rama principal).
2. Analista realiza los desarrollos requeridos en esta nueva rama feature.
3. Una vez terminados los desarrollos, el Analista es el encargado de realizar los merge requeridos en las ramas superiores: intergracion, integración\_qa, master e intergracion\_prod, según sea necesario de acuerdo al ciclo de vida de desarrollo.



**Consideraciones**

1. Analistas tienen acceso de escritura a las ramas (Github) de los diferentes ambientes.
2. No existe controles o aprobadores para realizar los merge en las diferentes ramas.

## Situación actual del proceso de despliegue FrontEnd

1. Analista se crea una rama feature a partir de la rama master (rama principal).
2. Analista realiza los desarrollos requeridos en esta nueva rama feature.
3. Una vez terminados los desarrollos, el Analista es el encargado de realizar los merge requeridos en las ramas superiores: master, qa y prod, según sea necesario de acuerdo al ciclo de vida de desarrollo.

**Consideraciones**

1. Analistas tienen acceso de escritura a las ramas (Github) de los diferentes ambientes.
2. Los nombres de las ramas (Github) no coinciden con el ambiente al que corresponden. Ej: QA 🡪 GSF
3. Existen archivos de configuración por ambiente.



## Esquema de Branching Propuesta BackEnd y FrontEnd

1. Analista se crea una rama ***feature*** a partir de la rama ***main*** (rama principal).
2. Analista realiza los desarrollos requeridos en esta nueva rama ***feature.***
3. Una vez terminados los desarrollos, el Analista es el encargado de realizar un pull request a la rama ***main***.
4. Líder Técnico o Arquitecto realiza la aprobación del pull request en la rama ***main***.
5. Control de Software será el encargado de realizar los merge a las ramas superiores: calidad, gsf. Para la rama de prod el encargado será Control Tecnológico.



**Puntos de mejora**

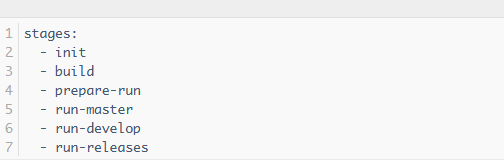
1. Estandarizar los nombres de las ramas en GitHub, para que tengan correspondencia con los ambientes de despliegue.
2. Estandarizar nombres de las ramas de FrontEnd y BackEnd en GitHub.
3. Evitar que los analistas/desarrolladores tengas permisos de escritura sobre las ramas de ambientes no productivos y de producción.
4. Incluir a Control de Software como aprobador para realizar los despliegues en ambientes no productivos.
5. Separar los archivos de configuración del código fuente, con el propósito de evitar las dependencias de las configuraciones requeridas por cada ambiente en sus respectivos despliegues.
6. A nivel del BackEnd, se requiere modificar las url de llamadas a microservicios, con el objetivo de obtener una url base parametrizable y que debe ser parte de las configuraciones indicadas en el punto anterior.
7. Validar la rama master contra la rama de producción para verificar otros posibles cambios de configuraciones integradas en el código.

## Configuración de un pipeline

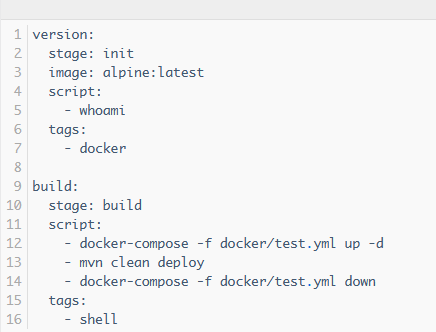
En GitLab el pipeline se configura en el fichero «.gitlab-ci.yml» que tiene que estar en la raíz del repositorio.

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:



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.

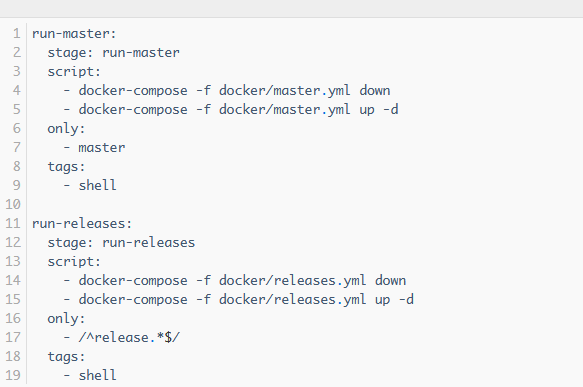


Dentro del job definimos tareas a ejecutar. Estas tareas se definen bajo la propiedad «script» como vemos en el ejemplo de arriba. El job de tipo docker ejecutará el comando dentro del contenedor que venga definido en la propiedad «image», teniendo en cuenta que GitLab va a copiar por nosotros todo el 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.

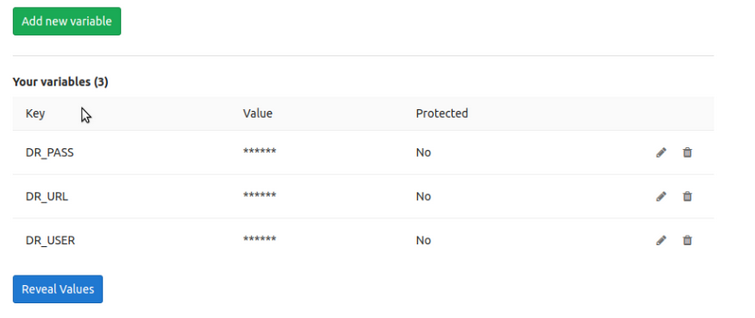
Para eso utiilizamos la propiedad «only».



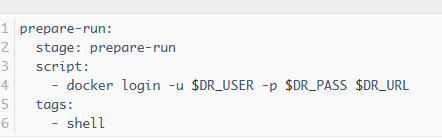
## Variables

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»



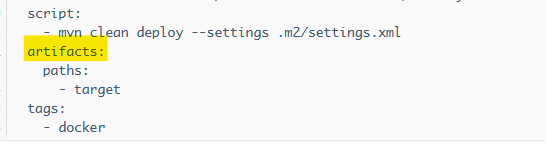
Y simplemente podemos hacer uso de estas claves en nuestros Jobs de esta forma:



## Artifacts

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.



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



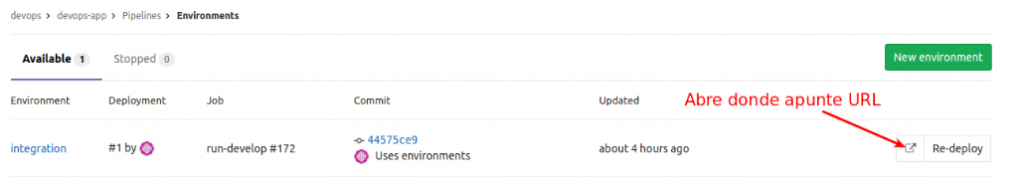
## Environments

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»).

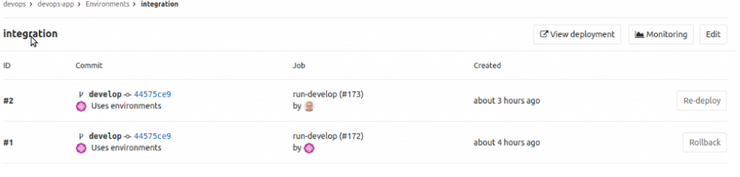
En este ejemplo, vemos como gracias a docker-compose dejamos corriendo los contenedores necesarios para poder levantar la aplicación en el estado de integración, automáticamente cuando se suben cambios a la rama develop.



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

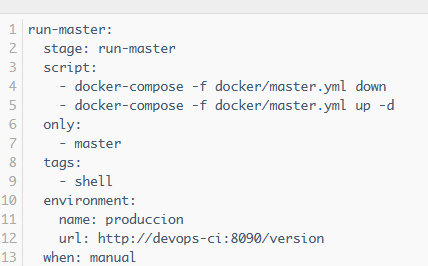


También podemos hacer un re-despliegue pulsando en “Re-deploy”. Cada vez que hagamos un despliegue en el entorno se irá acumulando, el listado lo podemos ver pinchando en el enlace del nombre de la versión.



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:



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

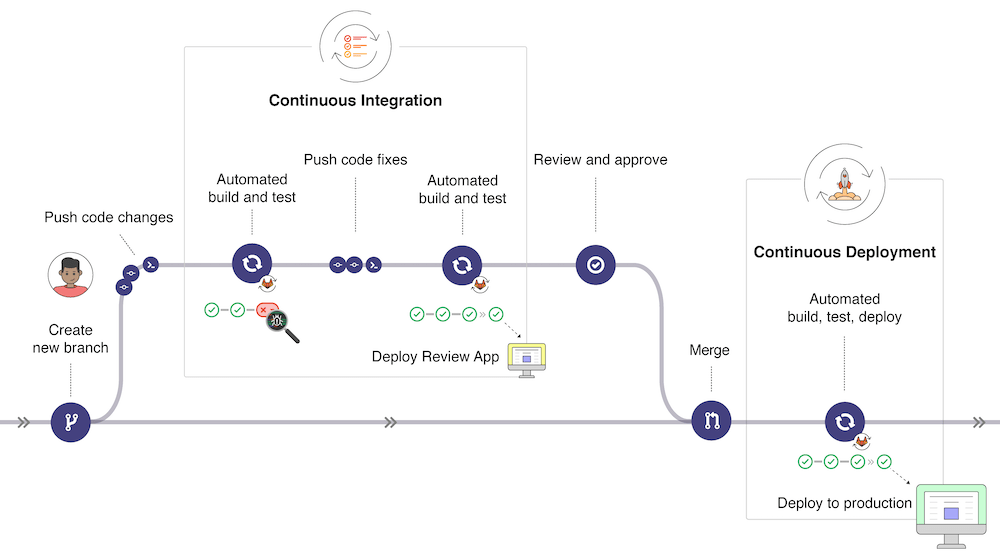
* on\_failure: sólo se ejecuta el job cuando el último job antes de éste falla.
* on\_success: sólo se ejecuta el job cuando todos los de antes han acabado con éxito. Es el que se pone por defecto.
* always: se ejecuta independientemente del resultado de los anteriores.

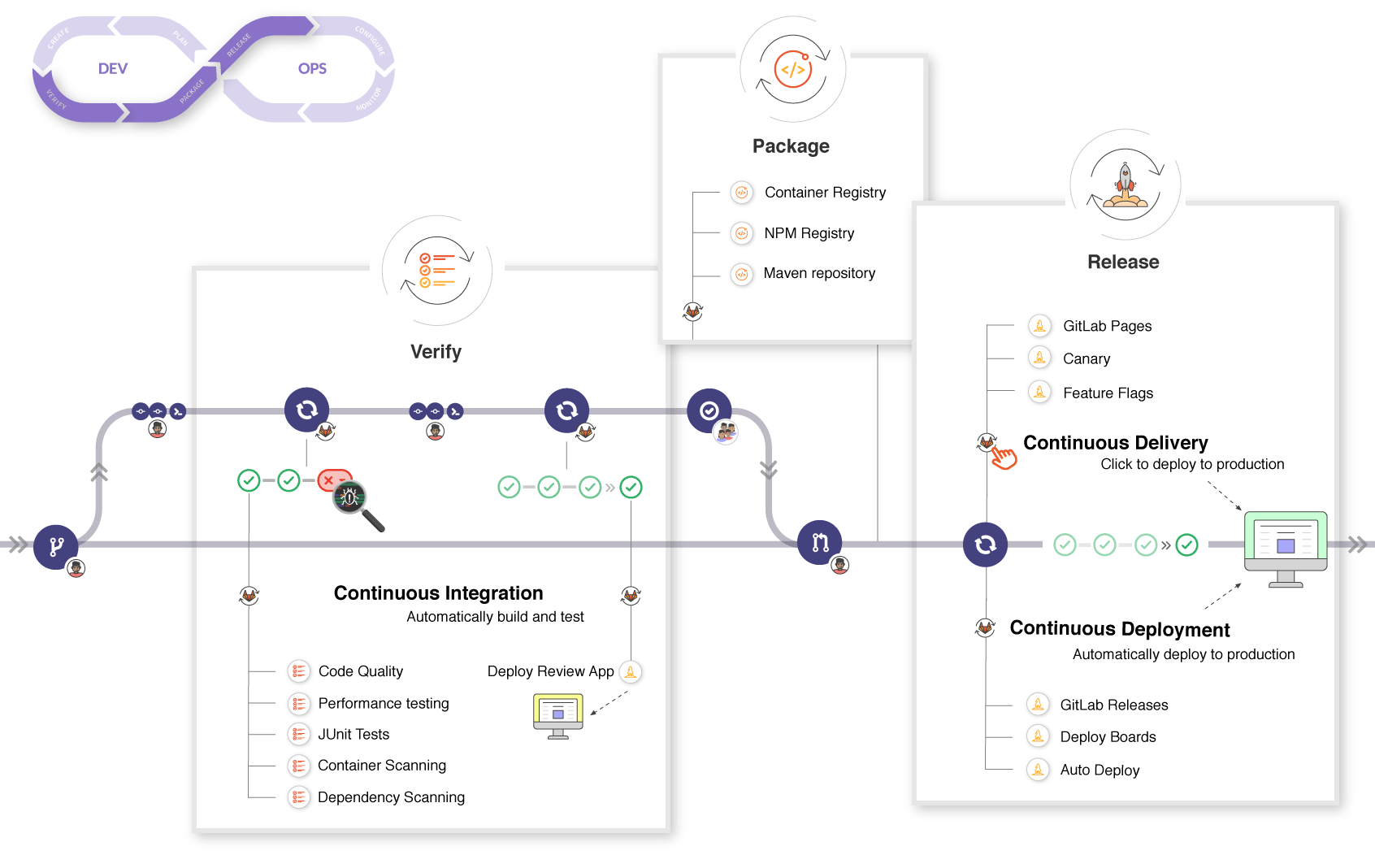
También podemos ignorar el resultado fallido de un job con la etiqueta «allow\_failure» a true.

## Workflow

Use la palabra clave workflow para controlar cuándo se crean los pipelines, se evalúa antes que los Jobs. Por ejemplo, si un trabajo está configurado para ejecutarse con etiquetas, pero el flujo de trabajo impide los pipelines con etiquetas, el trabajo nunca se ejecuta.

ifCláusulas comunes paraworkflow:rule





## Validación de la sintaxis

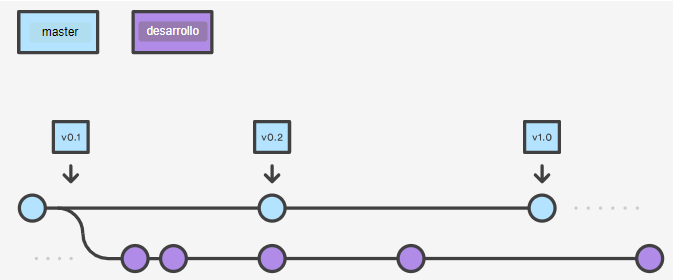
Es interesante, antes de hacer la subida al repositorio, validar que la sintaxis del fichero es correcta, de lo contrario romperá la build indicando el error en el fichero. Para hacer esto vamos a la URL: http://url-gitlab/ci/lint, pegamos el contenido del fichero y pulsamos en «Validate».

## Estructura del proyecto

## Migración de proyecto desde Github

## Creación Branchs

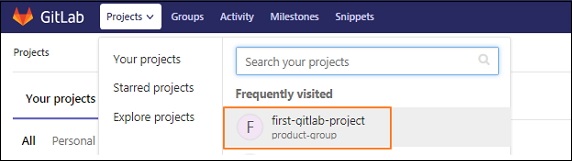
El esquema de branch que se va a manejar es:



En lugar de una única rama master, este flujo de trabajo utiliza dos ramas para registrar el historial del proyecto. La rama master o principal almacena el historial de publicación oficial y la rama develop o de desarrollo sirve como rama de integración para las funciones. Asimismo, conviene etiquetar todas las confirmaciones de la rama master con un número de versión.

Branch es una línea independiente y parte del proceso de desarrollo. La creación de un branch implica los siguientes pasos.

1. Inicie sesión en su cuenta de GitLab y vaya a su proyecto en la sección Proyectos.



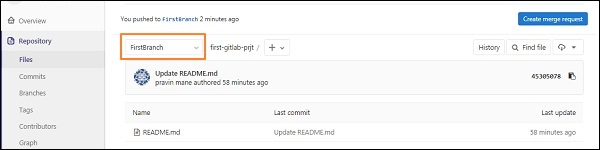
1. Para crear una sucursal, haga clic en la opción branch en la sección Repositorio y haga clic en el botón New branch.



1. En la pantalla New branch, ingrese el nombre Del branch (ya sea desarrollo o master según sea el caso siendo el master el principal ), haga clic en el botón Create branch.

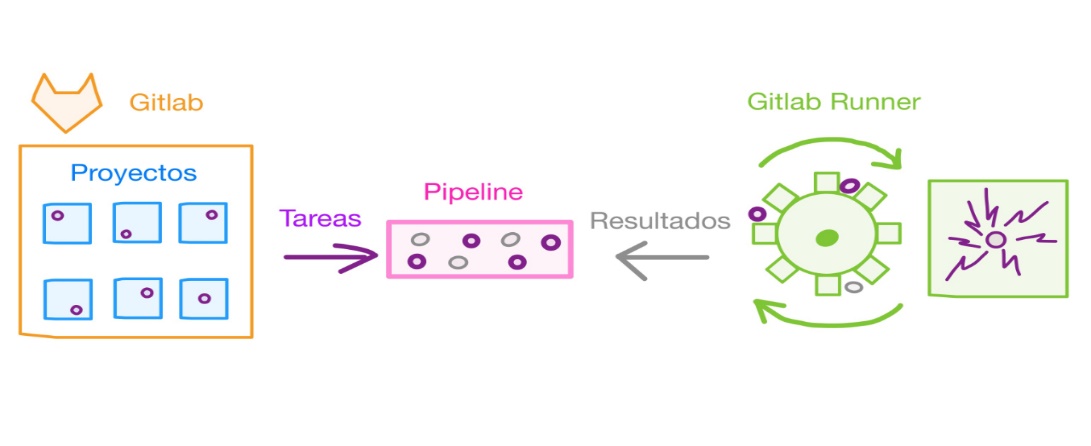


1. Después de crear la rama, obtendrá una pantalla a continuación junto con la rama creada

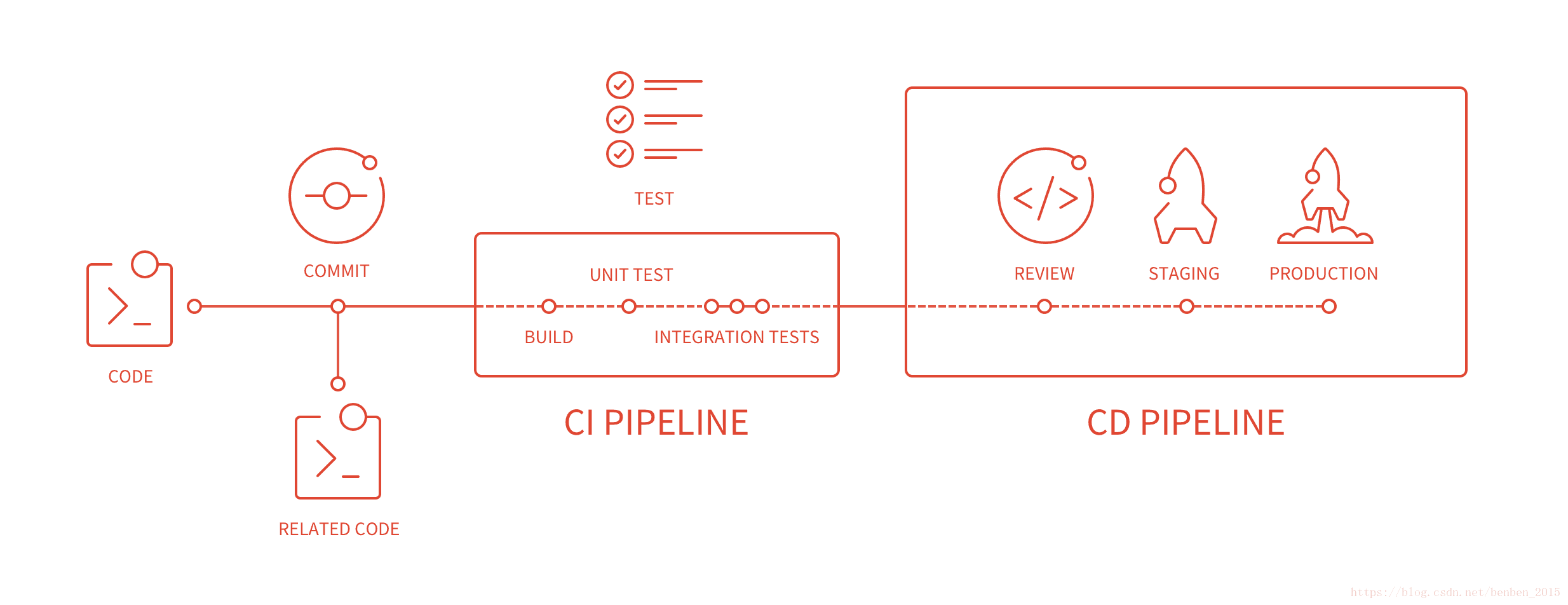


## Templates

## GitLab - CI/CD



El servicio GitLab CI (Integración continua) es una parte de GitLab que crea y prueba el software cada vez que el desarrollador inserta el código en la aplicación. GitLab CD (Implementación continua) es un servicio de software que coloca los cambios de cada código en la producción, lo que da como resultado la implementación diaria de la producción.



Los siguientes puntos describen el uso de GitLab CI/CD:

* Es fácil de aprender, usar y escalable.
* Es un sistema más rápido que se puede utilizar para la implementación y el desarrollo de código.
* Puede ejecutar los trabajos más rápido configurando su propio corredor (es una aplicación que procesa las compilaciones) con todas las dependencias que están preinstaladas.
* Las soluciones de CI de GitLab son económicas y seguras, y son muy flexibles en costos tanto como la máquina utilizada para ejecutarlas.
* Permite que los miembros del equipo del proyecto integren su trabajo diariamente, de modo que los errores de integración puedan identificarse fácilmente mediante una compilación automatizada.

## Glosario

ALM: application lifecycle management

API: application programming interface

CD: continuous deployment/delivery

CI: continuous integration

CLI: commandline interface

DevOps: development & operations

IC: integración continua

YAML: yaml ain't markup language