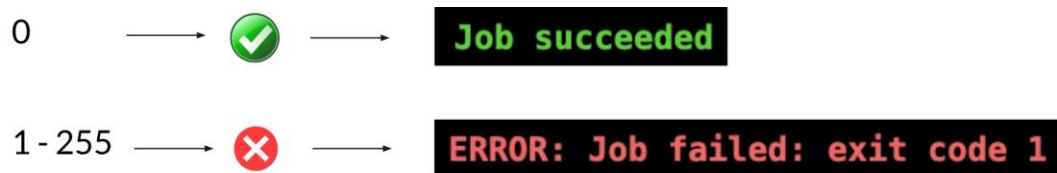


Agregar el Stage de pruebas

Esta vez vamos a revisar como añadir un test a nuestro pipeline con el fin de probar lo que se ha construido previamente para asegurarnos de que lo que tenemos como resultado sea lo que realmente queremos.

La primera pregunta que nos puede surgir es ¿qué hace que los Jobs de GitLab CI fallen? ¿por qué un pipeline de repente falla y no continúa con los siguientes pasos?

Bueno, cada vez que se ejecuta un comando, éste devolverá algo, y por convención devolverá cero en caso de éxito o devolverá un número entero en el rango de 1 a 255 en caso de error.



Así que cada vez que GitLab CI obtiene un estado que no es 0, significa que algo falló. La ejecución de un comando puede devolver un código de error si no se ejecutó correctamente, si le faltan parámetros, o si cierta aserción no se cumple.

Ahora, idealmente, debemos construir nuestra aplicación junto con sus pruebas unitarias, en este caso ejecutamos todo lo necesario para asegurarnos que nuestra aplicación cumple con todo lo que necesitamos.

En este sencillo ejemplo, vamos a ver las pruebas de forma muy básica para demostrar cómo se trabaja con este Stage y lo que podemos hacer.

Vamos ahora a la carpeta “public” de nuestro proyecto, y vamos a ejecutar el comando “grep” para buscar un string específico dentro de un archivo. Busquemos el texto “Gatsby” dentro del archivo index.html:

```
~/workspace/static-website/public ➤ master ➤ grep "Gatsby" index.html
```

El resultado de ejecutar este comando es que nos trae todo el contenido del archivo que contiene este string:

```

7pvUEjx28V00WUg9jIVwAAAABJRUS5ErkJggg==" alt="" style="position:absolute;top:0;left:0;
width:100%;height:100%;object-fit:cover;object-position:center;opacity:1;transition-delay:0.5s"/><noscript><picture></picture></noscript></div></div><a href="/page-2/">Go to page 2</a></main><footer>© <!-- -->2019<!-- -->, Built with<!-- --> <a href="https://www.gatsbyjs.org">Gatsby</a></footer></div></div><script id="gatsby-script-loader">*<![CDATA[*/*window.page={"componentChunkName":"component---src-pages-index-ic","isopName":"index","path":"/"},"window.dataPath="/140/path-

```

Este es un simple comando de ejemplo que podemos usar dentro de GitLab para hacer los test de nuestro pipeline. Así que sigamos adelante y agreguemos este Job adicional.

Agreguemos en nuestro archivo de pipeline “test artifact”:

```

1  build:website:
2    image: node
3    script:
4      - npm install
5      - npm install -g gatsby-cli
6      - gatsby build
7    artifacts:
8      paths:
9      - ./public
10
11
12  test:artifact:
13    script:
14      - grep "Gatsby" ./public/index.html

```

Ahora definamos los Stages que utilizaremos:

```

1  stages:
2    - build
3    - test
4
5  build:website:
6    image: node
7    script:
8      - npm install
9      - npm install -g gatsby-cli
10     - gatsby build
11  artifacts:
12    paths:
13    - ./public

```

Ahora especificamos qué Jobs pertenecen a qué Stages:

```

5 build website:
6   · stage: build
7   · image: node
8   · script:
9     · - npm install
10    · - npm install -g gatsby-cli
11    · - gatsby build
12   · artifacts:
13     · paths:
14       · - ./public

```

```

16 test artifact:
17   · stage: test
18   · script:
19     · - grep "Gatsby" ./public/index.html

```







Es una buena idea probar lo que estamos haciendo. Así que vamos a añadir algo que se va a ejecutar bien y algo que falle, solo para tener aserciones, pruebas que si funcionen y otras que simplemente nos den error:

```

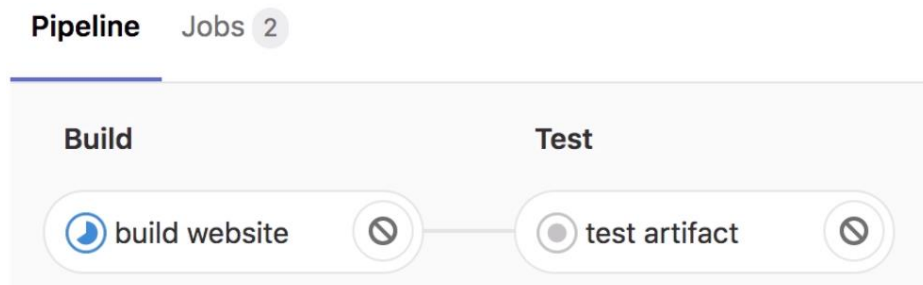
16 test artifact:
17   · stage: test
18   · script:
19     · - grep "Gatsby" ./public/index.html
20     · - grep "XXXXXXXX" ./public/index.html

```

Veamos ahora como luce nuestro pipeline:

Status	Pipeline	Commit	Stages
 running	#56030709 by  latest	 master -> 16c38da7  Added test stage	 

Podemos ver que ahora nuestro pipeline tiene dos Stages, de los cuales sabemos que uno fallará:



Ahora entremos al segundo Stage:

```
Running with gitlab-runner 11.9.0-rc2 (227934c0)
  on docker-auto-scale 72989761
Using Docker executor with image ruby:2.5 ...
Pulling docker image ruby:2.5 ...
```

Podemos ver que, como no hemos especificado una imagen, el runner utiliza por default la imagen que contiene Ruby.

A continuación, podemos ver la salida de nuestro test. Una de las pruebas ha sido exitosa ya que el primer comando se ejecutó y retornó un cero, pero el segundo comando falló, retornando un 1:

```
position:center;opacity:1;transition-delay:0.5s"/><noscript><picture></picture></noscript></div></div><a href="/page-2/">Go to page 2</a></main>
<footer><!-- -->2019<!-- -->, Built with<!-- --> <a href="https://www.gatsbyjs.org">Gatsby</a>
</footer></div></div></div><script id="gatsby-script-loader">*<![CDATA[*/window.page=
{"componentChunkName":"component---src-pages-index-
js","jsonName":"index","path":"/"};window.dataPath="140/path---index-6a9-
0SUcWyAf8ecbYDsMhQkEfPzV8";/*]]>*</script><script id="gatsby-chunk-mapping">*<![
CDATA[*/window.__chunkMapping={"app":["/app-fcee1bc0b98561142ff0.js"],"component---src-pages-
404-js":["/component---src-pages-404-js-8e1643e05587f89c70d6.js"],"component---src-pages-index-
js":["/component---src-pages-index-js-a3a1582b188d80057d48.js"],"component---src-pages-page-2-
js":["/component---src-pages-page-2-js-0479e3779ff85cb5431c.js"],"pages-manifest":["/pages-
manifest-5c619b10fb87f94c2d85.js"]};/*]]>*</script><script src="/component---src-pages-index-js-
a3a1582b188d80057d48.js" async=""></script><script src="/1-f2577b7d9227c6ab20bd.js" async="">
</script><script src="/styles-0375cdcc38b87565858c.js" async=""></script><script src="/app-
fcee1bc0b98561142ff0.js" async=""></script><script src="/webpack-runtime-aaec63d11721b79af7a1.js"
async=""></script></body></html>
$ grep "XXXXXXXX" ./public/index.html
ERROR: Job failed: exit code 1
```

Y de esta forma hemos conseguido añadir un simple Stage y un test para probar lo que estamos haciendo, al menos a un nivel muy básico.