

Pipelines

Para entender qué son los pipelines, hagamos una analogía con una cadena de montaje de coches: Con el fin de ensamblar un coche, es necesario seguir una secuencia de pasos, por ejemplo, primero necesitas un chasis, luego instalas sobre el motor, y una vez que tienes el motor ensamblado entonces puedes añadir las ruedas y así sucesivamente. La cadena de ensamblaje de coches es pues, una serie de pasos que hay que dar en un orden determinado. Usted no puede comenzar añadiendo las ruedas hasta que tenga el chasis y el motor. Además, los pasos están conectados, podemos decir que el resultado del paso anterior es la entrada del siguiente paso.

Algunos pasos por supuesto pueden hacerse en paralelo. Por ejemplo, la instalación de las ruedas. Usted puede trabajar en las cuatro ruedas al mismo tiempo sin afectar a la salida.

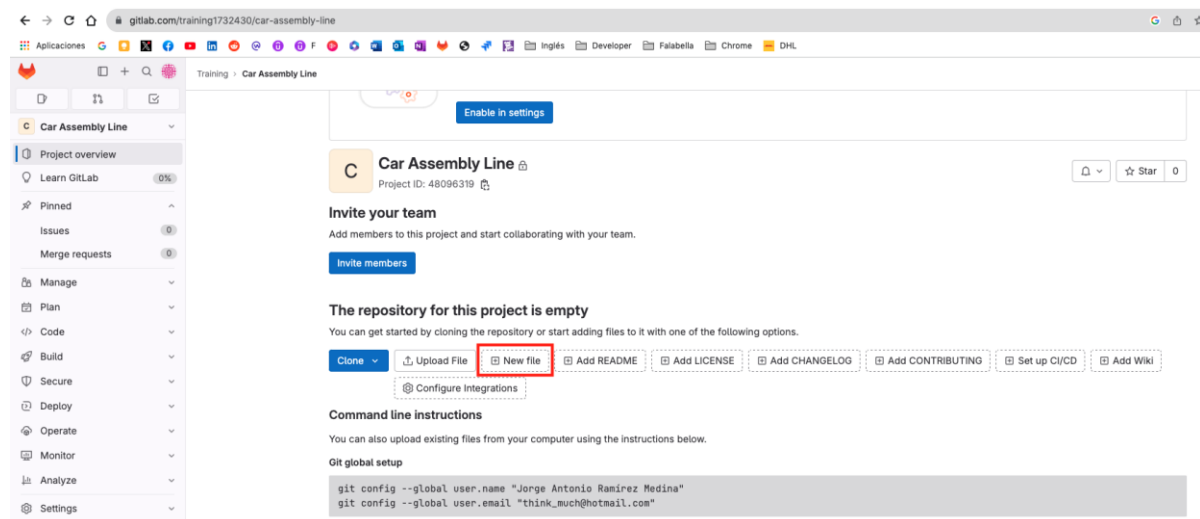
Ahora, se requieren muchos más pasos para construir el producto final, no solo el montaje. La producción de un coche es importante, pero antes de que el coche pueda salir de la fábrica, primero tiene que ser testado para asegurarse de que el consumidor final realmente recibe un coche que funciona bien.

El objetivo final de una línea de ensamblaje es que el producto salga de la fábrica.

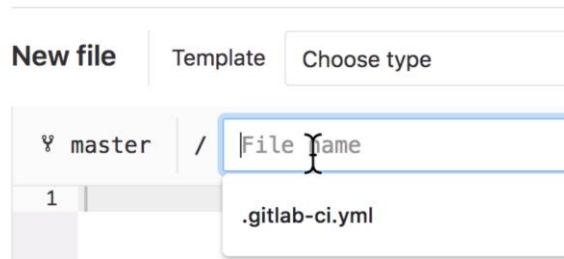
Podemos notar que construir software es, en algunos aspectos, bastante parecido a construir un coche. Esto es exactamente en lo que GitLab nos está ayudando, sacar el producto de la fábrica, sea cual sea el producto.

Ahora vamos a construir algo similar en GitLab CI, pero solo construiremos un Pipeline.

Cuando creamos un nuevo proyecto en GitLab, obtenemos un proyecto vacío, por esta razón, para crear un pipeline en GitLab CI, necesitamos crear un archivo que defina ese pipeline, y eso es bastante fácil usando la interfaz de GitLab. Solo tenemos que crear un nuevo archivo:



Este archivo debe tener un nombre específico, ese nombre es *.gitlab-ci.yml*:



YAML es un formato especial para los archivos que definen esos pipelines en GitLab CI. Así que probablemente vea este formato generalmente para especificar cualquier configuración. En GitLab es la única forma de especificar un pipeline.

Para que GitLab haga algo, necesitamos especificar un *Job*. Un Job, como su nombre lo indica, es una tarea que GitLab debe hacer por nosotros. Así que, digamos que el primer Job es construir el coche:

Vamos a escribir “build the car”:

```
.gitlab-ci.yml x
.gitlab-ci.yml
1 build the car:
```

Después, en una nueva línea, vamos a agregar dos espacios en blanco y escribimos la instrucción “script”:

```
.gitlab-ci.yml x
.gitlab-ci.yml
1 build the car:
2 script:
```

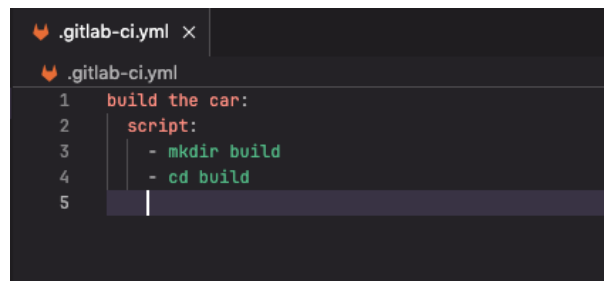
A partir de aquí, podemos comenzar a escribir comandos usando un guion por cada línea:

Ahora, lo que estamos tratando de hacer es crear un archivo y añadir algo de texto. Así que imaginemos que ese archivo es nuestro coche y luego vamos a añadir un chasis, un motor, unas ruedas y poner todo dentro de ese archivo.

Así que vamos a crear una carpeta llamada “build” escribiendo el comando shell “mkdir build”:

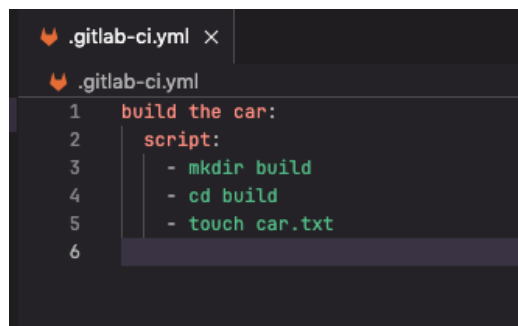
```
.gitlab-ci.yml x
.gitlab-ci.yml
1 build the car:
2 script:
3   - mkdir build
4
```

Luego, vamos a entrar en esa carpeta con el comando “cd build”:



```
.gitlab-ci.yml x
.gitlab-ci.yml
1  build the car:
2    script:
3      - mkdir build
4      - cd build
5
```

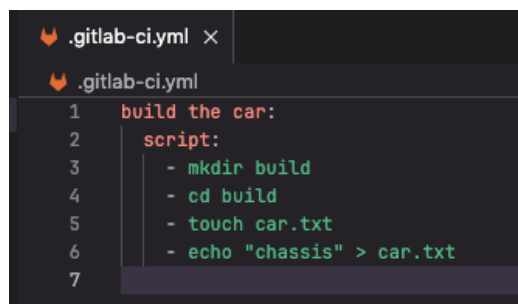
Ahora vamos a crear un archivo llamado car.txt y utilizamos el comando “touch” para crear ese archivo:



```
.gitlab-ci.yml x
.gitlab-ci.yml
1  build the car:
2    script:
3      - mkdir build
4      - cd build
5      - touch car.txt
6
```

Lo que hace el comando “touch” es cambiar el timestamp de creación o actualización de un archivo, y en caso de que no exista, simplemente crea uno nuevo.

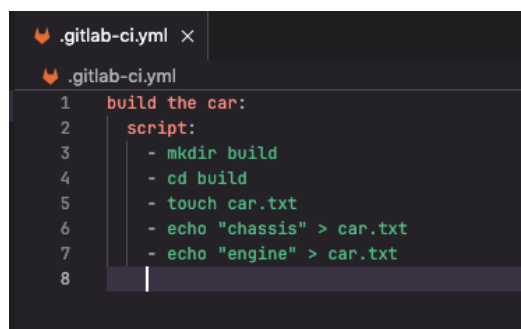
Ahora lo que haremos será añadir algo de información al archivo. Primero vamos a añadir al archivo la palabra “chassis”:



```
.gitlab-ci.yml x
.gitlab-ci.yml
1  build the car:
2    script:
3      - mkdir build
4      - cd build
5      - touch car.txt
6      - echo "chassis" > car.txt
7
```

Con el operador > vamos a guardar la salida del comando echo en el archivo car.txt.

Agregamos también un “engine”:



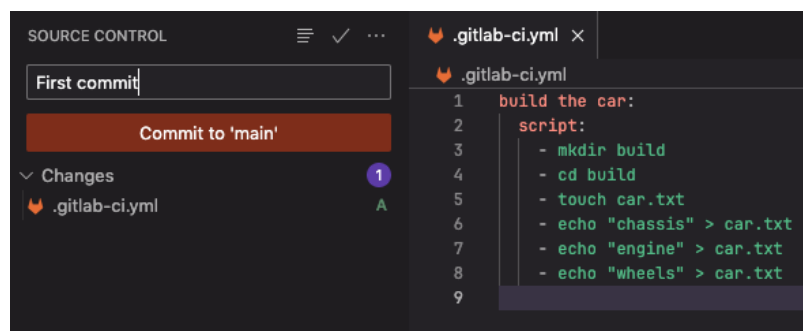
```
.gitlab-ci.yml x
.gitlab-ci.yml
1  build the car:
2    script:
3      - mkdir build
4      - cd build
5      - touch car.txt
6      - echo "chassis" > car.txt
7      - echo "engine" > car.txt
8
```

Y las ruedas:

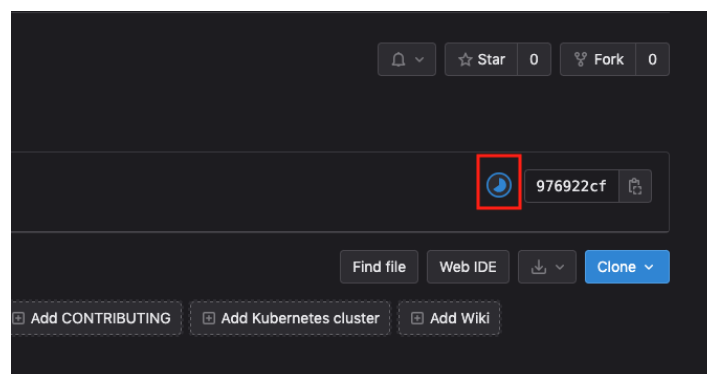
```
.gitlab-ci.yml x
.gitlab-ci.yml
1  build the car:
2    script:
3      - mkdir build
4      - cd build
5      - touch car.txt
6      - echo "chassis" > car.txt
7      - echo "engine" > car.txt
8      - echo "wheels" > car.txt
9
```

Estos son los pasos necesarios para crear nuestro coche.

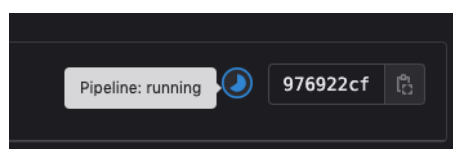
Ahora vamos a guardar nuestra configuración del pipeline mediante un commit de este archivo:



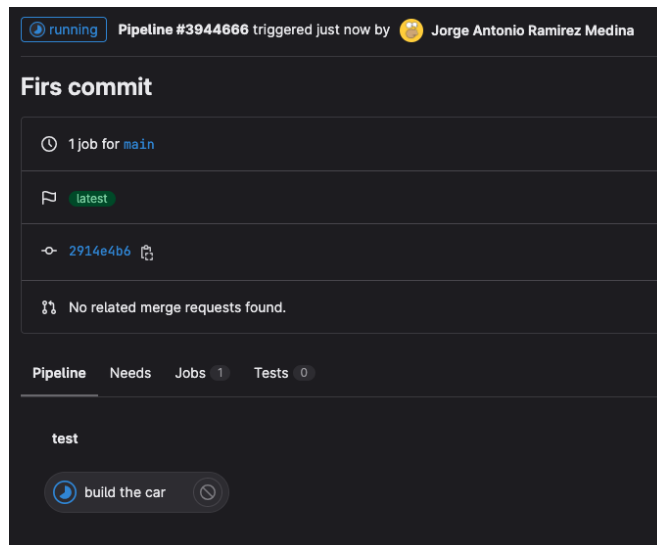
Tan pronto como hemos guardado los cambios del archivo con el commit, GitLab ha detectado nuestro nuevo pipeline y verá un indicador que muestra el progreso de la ejecución de nuestro pipeline:



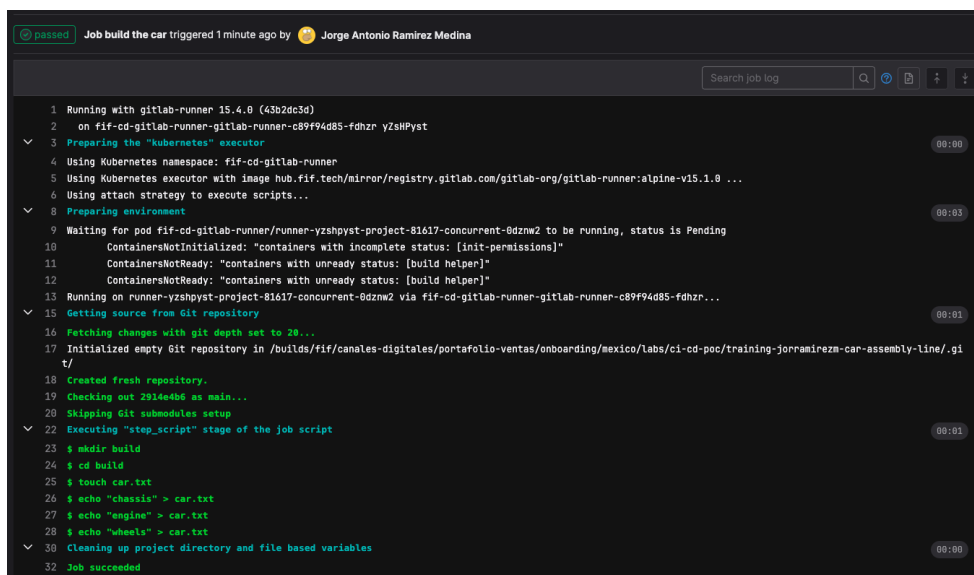
Si queremos ver lo que está haciendo, solo debemos hacer click en el círculo de progreso:



Ahora podemos ver nuestro pipeline, el cual solo contiene un Job:



Hagamos click en el Job:



GitLab ejecutará los scripts que hemos escrito previamente y mostrará sus salidas. La herramienta que ejecuta los scripts se llama GitLab Runner, y se parece mucho a nuestra terminal del computador.

Podemos ver exactamente los pasos que hemos dado:

1. Hemos creado una carpeta build:

```
Skipping Git submodules setup
$ mkdir build
```

2. Después creamos un archivo llamado car.txt:

```
Skipping Git submodules setup
$ mkdir build
$ cd build
$ touch car.txt
```

3. Después, añadimos unos strings a ese archivo:

```

Skipping Git submodules setup
$ mkdir build
$ cd build
$ touch car.txt
$ echo "chassis" > car.txt
$ echo "engine" > car.txt
$ echo "wheels" > car.txt

```

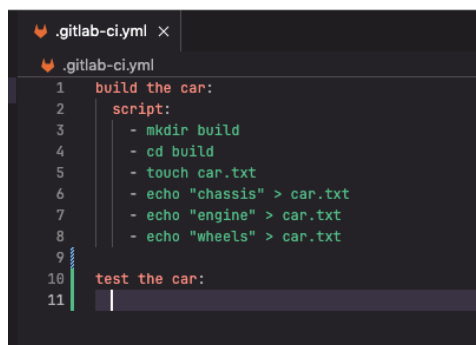
4. Y finalmente tenemos “Job succeeded” porque ninguno de nuestros scripts tuvo ningún problema:

```

23 $ mkdir build
24 $ cd build
25 $ touch car.txt
26 $ echo "chassis" > car.txt
27 $ echo "engine" > car.txt
28 $ echo "wheels" > car.txt
30 Cleaning up project directory and file based variables
32 Job succeeded

```

Ahora, nos gustaría añadir otro paso en el que podamos testear todo. Así que volvamos a editar nuestro archivo YAML y agreguemos el siguiente paso “test the car”:

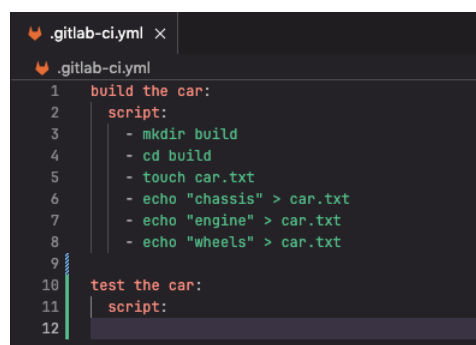


```

.gitlab-ci.yml x
.gitlab-ci.yml
1  build the car:
2    script:
3      - mkdir build
4      - cd build
5      - touch car.txt
6      - echo "chassis" > car.txt
7      - echo "engine" > car.txt
8      - echo "wheels" > car.txt
9
10 test the car:
11

```

Añadimos un nuevo script:



```

.gitlab-ci.yml x
.gitlab-ci.yml
1  build the car:
2    script:
3      - mkdir build
4      - cd build
5      - touch car.txt
6      - echo "chassis" > car.txt
7      - echo "engine" > car.txt
8      - echo "wheels" > car.txt
9
10 test the car:
11    script:
12

```

El primer paso es testear si el archivo car.txt fue creado. Una forma de verificar esto es usando el comando test:

```

.gitlab-ci.yml x
.gitlab-ci.yml
1  build the car:
2      script:
3          - mkdir build
4          - cd build
5          - touch car.txt
6          - echo "chassis" > car.txt
7          - echo "engine" > car.txt
8          - echo "wheels" > car.txt
9
10 test the car:
11     script:
12         - test -f build/car.txt
13

```

Si el comando test falla nos devolverá un código de estado e informará a GitLab de que algo no va bien en nuestro script y entonces GitLab marcará todo el pipeline con un fallo.

Añadimos ahora algunos tests adicionales para comprobar si el contenido del archivo es el que esperamos. Con los test que agregaremos queremos preguntar ¿tenemos un chassis? ¿tenemos un motor? ¿tenemos ruedas?

Primero entramos a la carpeta build:

```

10  test the car:
11      script:
12          - test -f build/car.txt
13          - cd build

```

Ahora utilizaremos el comando “grep” para buscar un string específico dentro del archivo, así que primero hacemos grep al texto “chassis”:

```

10  test the car:
11      script:
12          - test -f build/car.txt
13          - cd build
14          - grep "chassis" car.txt
15

```

Hacemos también un grep al texto “engine” y “wheels”:

```

10  test the car:
11      script:
12          - test -f build/car.txt
13          - cd build
14          - grep "chassis" car.txt
15          - grep "engine" car.txt
16          - grep "wheels" car.txt
17

```

Ahora bien, no hemos especificado realmente en qué orden deben ejecutarse estos Jobs. GitLab por default intentará ejecutarlos en paralelo, y esto es algo que no nos interesa en este momento. Así que, para especificar el orden en que los diferentes Jobs deben ser ejecutados, necesitamos definir “Stages”.

La instrucción para definir los “Stages” es bastante intuitiva. Debemos usar la instrucción “stages”:

```
.gitlab-ci.yml
1  stages:
2
3
4  build the car:
5    script:
6      - mkdir build
7      - cd build
8      - touch car.txt
9      - echo "chassis" > car.txt
10     - echo "engine" > car.txt
11     - echo "wheels" > car.txt
12
```

Tendremos dos Stages, build y test:

```
.gitlab-ci.yml
1  stages:
2    - build
3    - test
4
5  build the car:
6    script:
7      - mkdir build
8      - cd build
9      - touch car.txt
10     - echo "chassis" > car.txt
11     - echo "engine" > car.txt
12     - echo "wheels" > car.txt
13
```

El orden en que se declaran los Stages es importante, ya que define el orden de ejecución.

Ahora debemos especificar cuál de los Jobs que tenemos pertenecen a qué Stage. Añadimos al Stage “build” al Job “build the car”:

```
1  stages:
2    - build
3    - test
4
5  build the car:
6    stage: build
7    script:
8      - mkdir build
9      - cd build
10     - touch car.txt
11     - echo "chassis" > car.txt
12     - echo "engine" > car.txt
13     - echo "wheels" > car.txt
14
```

Y añadimos el Stage “test” al Job “test the car”:

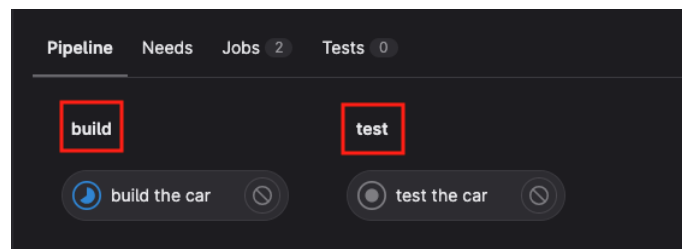

```

.gitlab-ci.yml x
.gitlab-ci.yml
1  stages:
2    - build
3    - test
4
5  build the car:
6    stage: build
7    script:
8      - mkdir build
9      - cd build
10     - touch car.txt
11     - echo "chassis" > car.txt
12     - echo "engine" > car.txt
13     - echo "wheels" > car.txt
14
15  test the car:
16    stage: test
17    script:
18      - test -f build/car.txt
19      - cd build
20      - grep "chassis" car.txt
21      - grep "engine" car.txt
22      - grep "wheels" car.txt
23

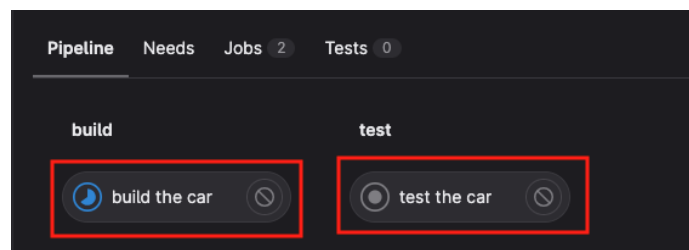
```

Ahora guardemos los cambios y veamos cómo trabaja nuestro pipeline:

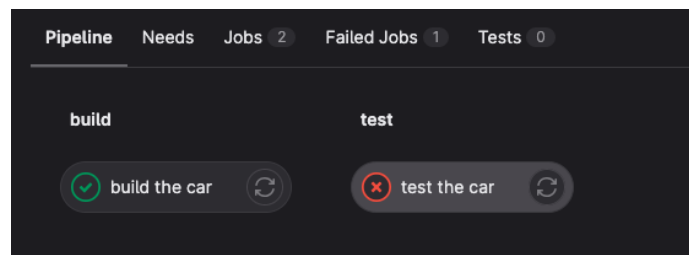
Ahora, desde el principio, verá que nuestro pipeline se ve un poco diferente de lo que era antes. Ahora nos muestra dos Stages: “build” y “test”:



Perteneciente al Stage “build” tenemos el Card “build the car”, y perteneciente al Stage “test” tenemos el card “test the car”:



Después de unos instantes, podemos ver que el Job “test the car” ha fallado:



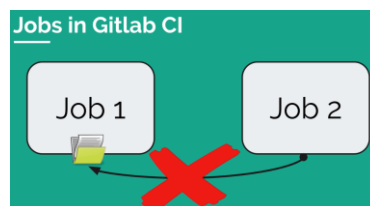
Hagamos click en el Job “test the car” para entrar a los logs del Job:

```
failed Job test the car triggered 6 minutes ago by Jorge Antonio Ramirez Medina

1 Running with gitlab-runner 15.4.0 (43b2dc3d)
2 on fif-cd-gitlab-runner-gitlab-runner-c89f94d85-fdhzr yZsHPyst
3 Preparing the "kubernetes" executor 00:00
4 Using Kubernetes namespace: fif-cd-gitlab-runner
5 Using Kubernetes executor with image hub.fif.tech/mirror/registry.gitlab.com/gitlab-org/gitlab-runner:alpine-v15.1.0 ...
6 Using attach strategy to execute scripts...
7
8 Preparing environment 00:04
9 Waiting for pod fif-cd-gitlab-runner/runner-yzshpyst-project-81617-concurrent-0pxs2b to be running, status is Pending
10 Running on runner-yzshpyst-project-81617-concurrent-0pxs2b via fif-cd-gitlab-runner-gitlab-runner-c89f94d85-fdhzr...
11
12 Getting source from Git repository 00:01
13 Fetching changes with git depth set to 20...
14 Initialized empty Git repository in /builds/fif/canales-digitales/portafolio-ventas/onboarding/mexico/labs/ci-cd-poc/training-jorramirez-m-car-assembly-line/.git/
15 Created fresh repository.
16 Checking out c828b3a2 as main...
17 Skipping Git submodules setup
18
19 Executing "step_script" stage of the job script 00:00
20 $ test -f build/car.txt
21
22 Cleaning up project directory and file based variables 00:01
23
24 ERROR: Job failed: command terminated with exit code 1
```

Parece que el primer test falló, en el que comprobamos que se ha creado el archivo. El *Status Code* debe ser 0 para que GitLab Runner marque un script como exitoso. El test ha respondido con el Status Code 1, lo que significa que el Job ha fallado.

Este error es correcto, ya que los Jobs que estamos ejecutando son independientes entre si, no intercambian ningún dato si no se les especifica que lo hagan.



Así que el Job anterior, el Job “build the car” dejó un archivo creado, peor no hemos especificado que debería pasar con ese archivo. Entonces, después de que finaliza el Job, se elimina todo el entorno donde se ejecutó ese Job, incluido el archivo generado. Entonces, cuando el segundo Job se ejecuta, busca el archivo car.txt, pero éste ya no existe.

Necesitamos indicar a nuestros Jobs cuál es el resultado que deseamos guardar, cual es nuestro “artifact”. Con estos Jobs estamos construyendo “algo” y esto es algo que queremos conservar, no queremos deshacernos de el después de finalizado el proceso.

Así que modifiquemos nuestro archivo de configuración una vez más.

En nuestro primer Job, definiremos nuestro “artifact” utilizando precisamente la instrucción “artifact”:

```

.gitlab-ci.yml x
.gitlab-ci.yml
1  stages:
2    - build
3    - test
4
5  build the car:
6    stage: build
7    script:
8      - mkdir build
9      - cd build
10     - touch car.txt
11     - echo "chassis" > car.txt
12     - echo "engine" > car.txt
13     - echo "wheels" > car.txt
14   artifacts:
15

```

La instrucción artifact necesita una ruta, por lo que debemos indicar en donde se encuentra exactamente el artifact:

```

.gitlab-ci.yml x
.gitlab-ci.yml
1  stages:
2    - build
3    - test
4
5  build the car:
6    stage: build
7    script:
8      - mkdir build
9      - cd build
10     - touch car.txt
11     - echo "chassis" > car.txt
12     - echo "engine" > car.txt
13     - echo "wheels" > car.txt
14   artifacts:
15     paths:
16       -
17

```

En la instrucción “path” podemos especificar un archivo o una carpeta completa. Así que vamos a especificar la carpeta “build/”:

```

.gitlab-ci.yml x
.gitlab-ci.yml
1  stages:
2    - build
3    - test
4
5  build the car:
6    stage: build
7    script:
8      - mkdir build
9      - cd build
10     - touch car.txt
11     - echo "chassis" > car.txt
12     - echo "engine" > car.txt
13     - echo "wheels" > car.txt
14   artifacts:
15     paths:
16       - build/
17

```

Ahor probemos una vez más nuestro pipeline.

Verás que cada vez que hacemos un cambio en nuestro repositorio, el pipeline se ejecuta. La ejecución del pipeline se realiza no solo cuando editemos el archivo de configuración .gitlab-ci.yml, si no cada vez que hagamos un cambio dentro de cualquier parte de nuestro repositorio, ya sea que agreguemos o editemos cualquier otro archivo, el pipeline se ejecutará para garantizar que el cambio realizado funcione correctamente.

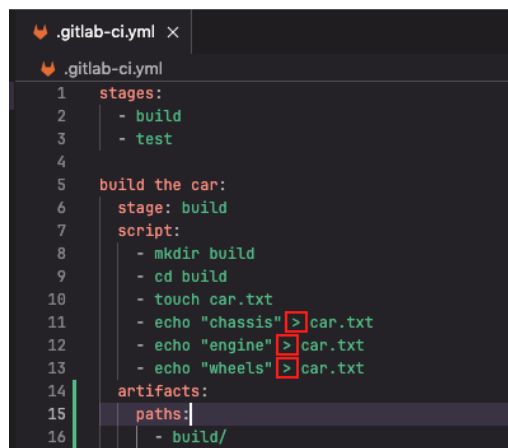
Vemos que el Job “test the car” ha fallado nuevamente. Vayamos a los logs para ver qué sucedió:

```
24 Created fresh repository.
25 Checking out 735a6b87 as main...
26 Skipping Git submodules setup
27
28 Downloading artifacts
29 Downloading artifacts for build the car (19429894)...
30 Downloading artifacts from coordinator... ok      host=gitlab.falabella.tech id=19429894 responseStatus=200 OK token=64_hraAt
31
32 Executing "step_script" stage of the job script
33 $ test -f build/car.txt
34 $ cd build
35 $ grep "chassis" car.txt
36
37 ERROR: Job failed: command terminated with exit code 1
```

Podemos que el primer (verificar la existencia del archivo car.txt) paso parece estar funcionando bien, el artifact generado por el primer Job está disponible para el segundo Job. Pero, por alguna razón, el comando grep no está encontrando el texto "chassis" dentro del archivo. El motivo de este error es que hemos agregado un error desde el principio, solo para demostrar lo importante que es escribir los tests.

Volvamos ahora a nuestro archivo de configuración.

El problema está en el comando *echo*. En lugar de agregar cada línea de texto, lo que hicimos fue reemplazar el contenido completo del archivo por una nueva línea de texto usando el símbolo >:



```
.gitlab-ci.yml x
.gitlab-ci.yml
1  stages:
2    - build
3    - test
4
5  build the car:
6    stage: build
7    script:
8      - mkdir build
9      - cd build
10     - touch car.txt
11     - echo "chassis" > car.txt
12     - echo "engine" > car.txt
13     - echo "wheels" > car.txt
14
15  artifacts:
16    paths:
17      - build/
```

La forma correcta de añadir texto al archivo sin reemplazar el existente es usando el símbolo >>:

```

.gitlab-ci.yml X
.gitlab-ci.yml
1  stages:
2    - build
3    - test
4
5  build the car:
6    stage: build
7    script:
8      - mkdir build
9      - cd build
10     - touch car.txt
11     - echo "chassis" >> car.txt
12     - echo "engine" >> car.txt
13     - echo "wheels" >> car.txt
14  artifacts:
15    paths:
16      - build/
17

```

En cualquier momento que necesitemos debuguear, siéntete libre de añadir comandos como “ls” para listar el contenido de la carpeta:

```

18  test the car:
19    stage: test
20    script:
21      - ls
22      - test -f build/car.txt
23      - cd build
24      - grep "chassis" car.txt
25      - grep "engine" car.txt
26      - grep "wheelss" car.txt
27

```

O puedes hacer algo como “cat” y especificar el archivo car.txt:

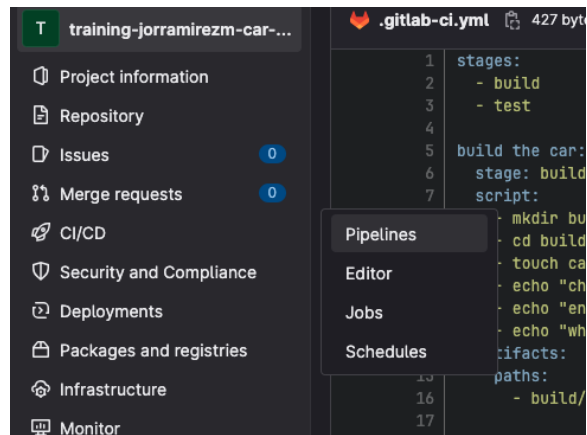
```

17
18  test the car:
19    stage: test
20    script:
21      - ls
22      - test -f build/car.txt
23      - cd build
24      - cat car.txt
25      - grep "chassis" car.txt
26      - grep "engine" car.txt
27      - grep "wheelss" car.txt
28

```

Así que puede añadir cualquier cosa que sea necesaria para debuguear el problema, tal cual lo haria en una terminal normal.

Cada vez que queramos mirar lo que ha pasado anteriormente, podemos ir a donde dice CI / CD para ver los pipelines:



Puede ver todos los pipelines anteriores que hemos ejecutado. Hasta el momento tenemos 6:

Alt

6

Finished

Branches

Tags

Clear runner caches

CI lint

Run pipeline

Filter pipelines

Q

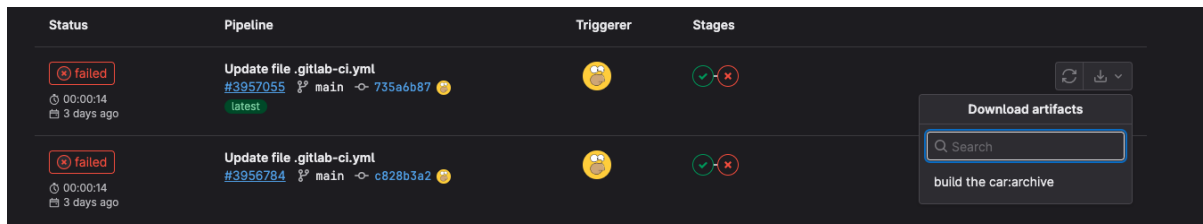
Show Pipeline ID

Status	Pipeline	Triggerer	Stages	
<div><div>failed</div><div>00:00:14</div><div>2 days ago</div></div>	<div>Update file .gitlab-ci.yml</div> <div>#3957055</div> <div>main</div> <div>735a6b87</div> <div>latest</div>	<div></div>	<div><div>✓</div><div>✗</div></div>	<div><div>↺</div><div>⬇</div></div>
<div><div>failed</div><div>00:00:14</div><div>3 days ago</div></div>	<div>Update file .gitlab-ci.yml</div> <div>#3956784</div> <div>main</div> <div>c828b3a2</div>	<div></div>	<div><div>✓</div><div>✗</div></div>	<div><div>↺</div><div>⬇</div></div>
<div><div>failed</div><div>00:00:15</div><div>3 days ago</div></div>	<div>Update file .gitlab-ci.yml</div> <div>#3956782</div> <div>main</div> <div>19d4b5f8</div>	<div></div>	<div><div>✓</div><div>✗</div></div>	<div><div>↺</div><div>⬇</div></div>
<div><div>failed</div><div>3 days ago</div></div>	<div>Update file .gitlab-ci.yml</div> <div>#3956739</div> <div>main</div> <div>9c42a194</div> <div>yml invalid</div> <div>error</div>	<div></div>		<div><div>⬇</div></div>
<div><div>passed</div><div>00:00:07</div><div>4 days ago</div></div>	<div>Firs commit</div> <div>#3944666</div> <div>main</div> <div>2914e4b6</div>	<div></div>	<div><div>✓</div></div>	<div><div>⬇</div></div>
<div><div>passed</div><div>00:00:07</div><div>4 days ago</div></div>	<div>Update file .gitlab-ci.yml</div> <div>#3944665</div> <div>main</div> <div>976922cf</div>	<div></div>	<div><div>✓</div></div>	<div><div>⬇</div></div>

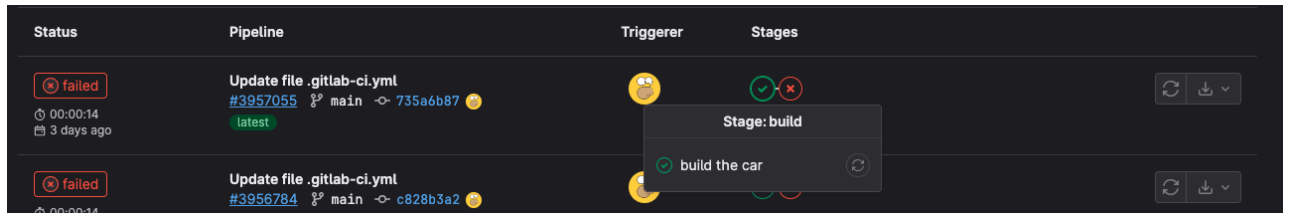
Aquí está el pipeline donde hemos añadido los artifact y los Jobs que han creado esos artifacts, los cuales están disponibles para su descarga:

Status	Pipeline	Triggerer	Stages				
<div>failed</div> <div>00:00:14</div> <div>3 days ago</div>	<div>Update file .gitlab-ci.yml</div> <div>#3957055</div> <div>main</div> <div>735a6b87</div> <div>latest</div>	<div>Triggerer</div>	<div>Stages</div>	<div>Refresh</div>	<div>Download</div>		
<div>failed</div> <div>00:00:14</div> <div>3 days ago</div>	<div>Update file .gitlab-ci.yml</div> <div>#3956784</div> <div>main</div> <div>c828b3a2</div>	<div>Triggerer</div>	<div>Stages</div>	<div>Refresh</div>	<div>Download</div>		
<div>failed</div> <div>00:00:15</div> <div>3 days ago</div>	<div>Update file .gitlab-ci.yml</div> <div>#3956782</div> <div>main</div> <div>19d4b5f8</div>	<div>Triggerer</div>	<div>Stages</div>	<div>Refresh</div>	<div>Download</div>		

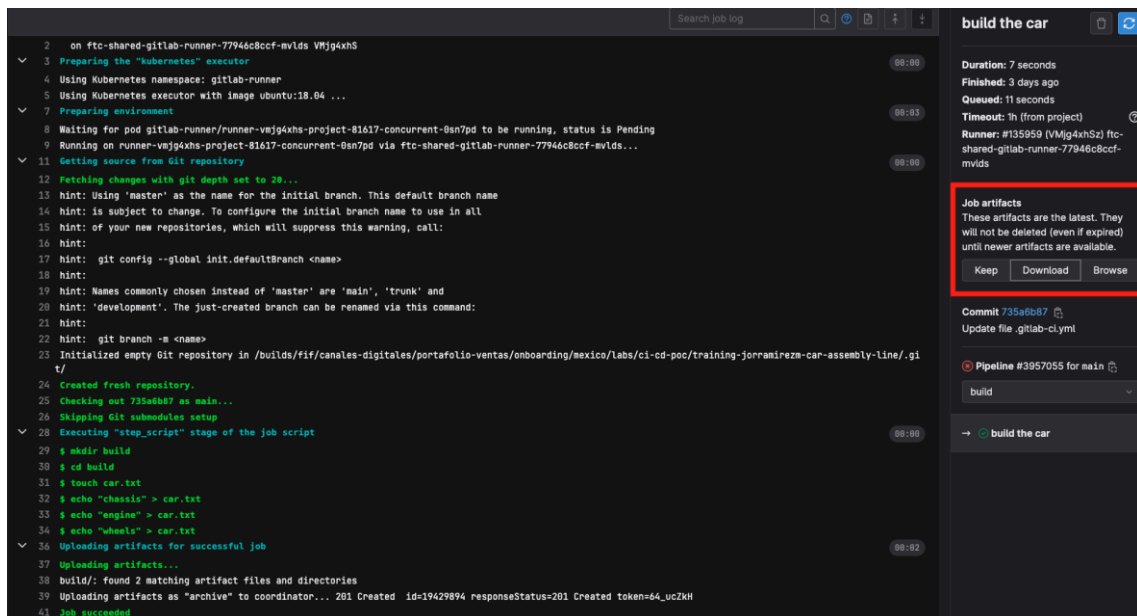
Puede inspeccionar los archivos que se han generado de manera sencilla:



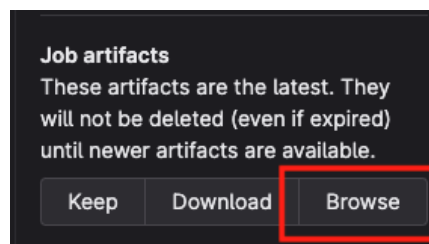
Ingresa al primer Job:



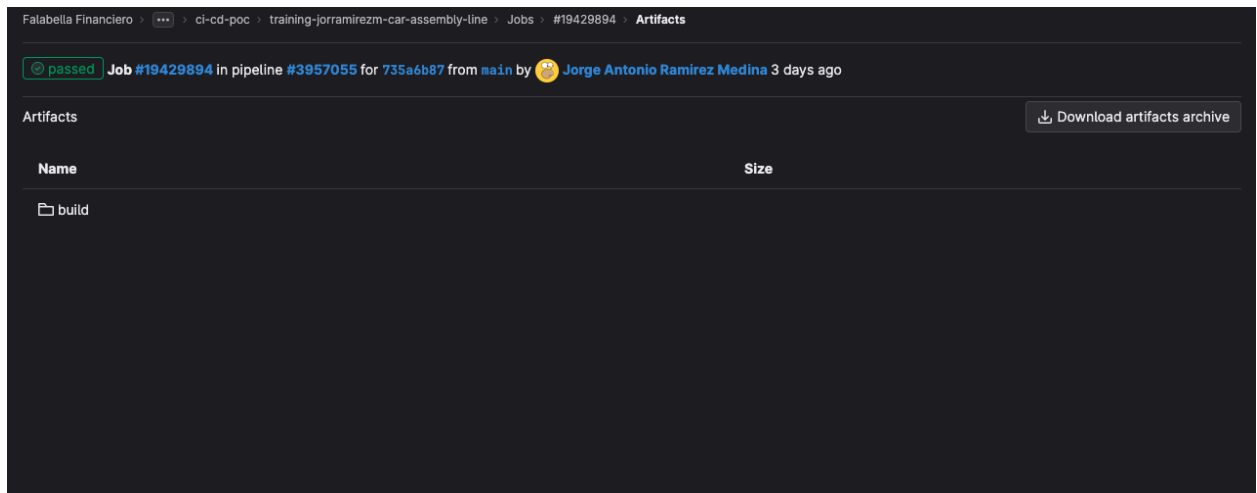
Justo a la derecha podemos ver los “Job Artifacts”:



Podemos navegar por los artifacts dando click en el botón “Browser”:



Recordemos que, en uno de nuestros Jobs, creamos una carpeta llamada build:



Podemos navegar hacia adentro de la carpeta y encontramos nuestro archivo car.txt, el cual podemos descargar:

