



0,00

Рейтинг

Softmart

Компания



nem 7 сентября 2016 в 10:35

Введение в GitLab CI

Автор оригинала: Ivan Nemytchenko

Блог компании Softmart, Тестирование IT-систем, Git, Тестирование веб-сервисов, Системы сборки

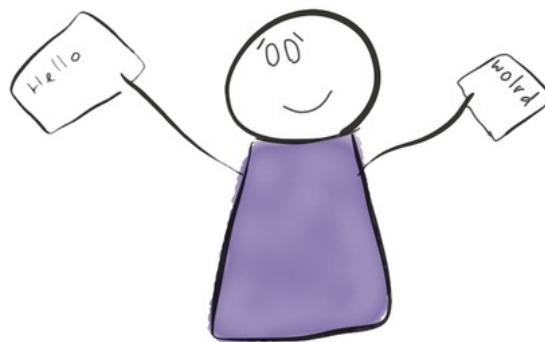
Перевод

Tutorial

Публикую перевод моей статьи из блога ГитЛаба про то как начать использовать CI. Остальных постов можно найти в блоге компании Softmart.

Представим на секунду, что вы не знаете ничего о концепции непрерывной интеграции (Сон нужна. Или вы всё это забыли. В любом случае, начнем с основ.

Представьте, что вы работаете над проектом, в котором вся кодовая база состоит из двух текстовых файлов. Важно, чтобы при конкатенации этих файлов в результате всегда получалась фраза "Hello world". Вся команда лишается месячной зарплаты. Да, все настолько серьезно.



Реклама

Один ответственный разработчик написал небольшой скрипт, который нужно запускать перед каждой отправкой кода заказчиком. Скрипт нетривиален:

```
cat file1.txt file2.txt | grep -q "Hello world"
```

Проблема в том, что в команде десять разработчиков, а человеческий фактор еще никто не отменял.

Неделю назад один новичок забыл запустить скрипт перед отправкой кода, в результате чего трое заказчиков получили поломанные сборки. Хотелось бы в дальнейшем избежать подобного, так что вы решаете положить конец этой проблеме раз и навсегда. К счастью, ваш код уже находится на GitLab, а вы помните про встроенную CI-систему. К тому же, на конференции вы слышали, что CI используется для тестирования...

Запуск первого теста в CI

После пары минут, потраченных на поиск и чтение документации, оказывается, что все что нужно сделать — это добавить две строки кода в файл `.gitlab-ci.yml`:

```
test:
  script: cat file1.txt file2.txt | grep -q 'Hello world'
```

Добавляем, коммитим — и ура! Сборка успешна!

✓ passed Build #2346110 for commit f8a26206 from master by @inem about an hour ago

```
gitlab-ci-multi-runner 1.3.2 (0323456)
Using Docker executor with image ruby:2.1 ...
Pulling docker image ruby:2.1 ...
Running on runner-8a2f473d-project-1398078-concurrent-0 via runner-8a2f473d-machine-1468420047-2374f4cc-digital-ocean-4gb...
Cloning repository...
Cloning into '/builds/inem/ci'...
Checking out f8a26206 as master...
$ cat file1.txt file2.txt | grep -q "Hello world"

Build succeeded
```

Поменяем во втором файле "world" на "Africa" и посмотрим, что получится:

✗ failed Build #2346623 for commit b978b9f6 from master by @inem about an hour ago

```
gitlab-ci-multi-runner 1.3.2 (0323456)
Using Docker executor with image ruby:2.1 ...
Pulling docker image ruby:2.1 ...
Running on runner-30dcea4b-project-1398078-concurrent-0 via runner-30dcea4b-machine-1468421193-15f1e5c5-digital-ocean-4gb...
Cloning repository...
Cloning into '/builds/inem/ci'...
Checking out b978b9f6 as master...
$ cat file1.txt file2.txt | grep -q "Hello world"

ERROR: Build failed: exit code 1
```

Сборка неудачна, как и ожидалось.

Итак, у нас теперь есть автоматизированные тесты. GitLab CI будет запускать наш тестовый скрипт при каждом пуше нового кода в репозиторий.

Возможность загрузки результатов сборки

Следующим бизнес-требованием является архивация кода перед отправкой заказчиком. Почему бы не автоматизировать и его?

Все, что для этого нужно сделать — определить еще одну задачу для CI. Назовем ее "package":

```
test:
  script: cat file1.txt file2.txt | grep -q 'Hello world'

package:
  script: cat file1.txt file2.txt | gzip > package.gz
```

В результате появляется вторая вкладка

✓ test ✓ package

```
gitlab-ci-multi-runner 1.3.2 (0323456)
Using Docker executor with image ruby:2.1 ...
Pulling docker image ruby:2.1 ...
Running on runner-8a2f473d-project-1447361-concurrent-0 via runner-8a2f473d-machine-1469549805-b2f018ac-digital-ocean-4gb...
Cloning repository...
Cloning into '/builds/inem/ci2'...
Checking out be833f45 as master...
$ cat file1.txt file2.txt | grep -q 'Hello world'

Build succeeded
```

Однако мы забыли уточнить, что новый файл является артефактом сборки, что позволит его скачивать. Это легко поправить, добавив раздел `artifacts`:

```
test:
  script: cat file1.txt file2.txt | grep -q 'Hello world'

package:
  script: cat file1.txt file2.txt | gzip > packaged.gz
  artifacts:
    paths:
      - packaged.gz
```

Проверяем... Все на месте:

The screenshot shows a GitLab CI build page for commit 0e99231d. The build status is 'passed'. The build details show a duration of 49 seconds and a runner ID of #21099. The build artifacts section shows a 'Download' button. The build log shows the following steps:

```
gitlab-ci-multi-runner 1.3.2 (0323456)
Using Docker executor with image ruby:2.1 ...
Pulling docker image ruby:2.1 ...
Running on runner-30dcea4b-project-1447361-concurrent-0 via runner-30dcea4b-machine-1469550348-1f5e7833-digital-ocean-4gb...
Cloning repository...
Cloning into '/builds/inem/ci2'...
Checking out 0e99231d as master...
$ cat file1.txt file2.txt | gzip > packaged.gz
Uploading artifacts...
packaged.gz: found 1 matching files
Uploading artifacts to coordinator... ok      id=2630934  responseStatus=201 Created token=7PE1aQwt

Build succeeded
```

Отлично! Однако, осталась одна проблема: задачи выполняются параллельно, а нам не нужно архивировать наше приложение в случаях, когда тест не пройден.

Последовательное выполнение задач

Задача `'package'` должна выполняться только при успешном прохождении тестов. Определим порядок выполнения задач путем введения стадий (`stages`):

```
stages:
  - test
  - package

test:
  stage: test
  script: cat file1.txt file2.txt | grep -q 'Hello world'

package:
  stage: package
  script: cat file1.txt file2.txt | gzip > packaged.gz
  artifacts:
    paths:
      - packaged.gz
```

Должно сработать.

Также не стоит забывать о том, что компиляция (которой в нашем случае является конкатенация файлов) занимает время, поэтому не стоит проводить ее дважды. Введем отдельную стадию для компиляции:

```
stages:
  - compile
  - test
  - package

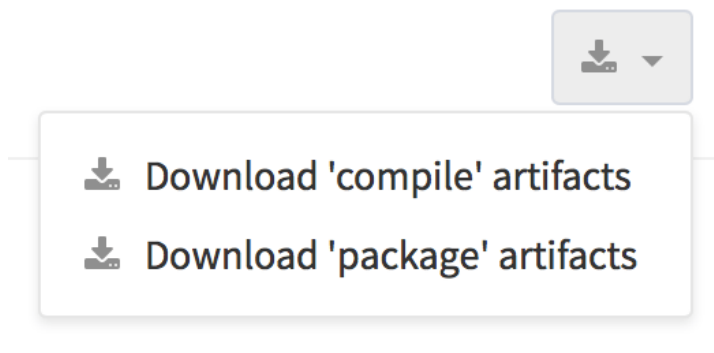
compile:
  stage: compile
```

```
script: cat file1.txt file2.txt > compiled.txt
artifacts:
  paths:
    - compiled.txt

test:
  stage: test
  script: cat compiled.txt | grep -q 'Hello world'

package:
  stage: package
  script: cat compiled.txt | gzip > packaged.gz
artifacts:
  paths:
    - packaged.gz
```

Посмотрим на получившиеся артефакты:



Скачивание файла "compile" нам ни к чему, поэтому ограничим длительность жизни временных артефактов 20 минутами:

```
compile:
  stage: compile
  script: cat file1.txt file2.txt > compiled.txt
artifacts:
  paths:
    - compiled.txt
  expire_in: 20 minutes
```

Итоговая функциональность конфига впечатляет:

- Есть три последовательных стадии: компиляция, тестирование и архивация приложения.
- Результат стадии компиляции передается на последующие стадии, то есть приложение компилируется только однажды (что ускоряет рабочий процесс).
- Архивированная версия приложения хранится в артефактах сборки для дальнейшего использования.

Какие образы Docker лучше использовать

Прогресс налицо. Однако, несмотря на наши усилия, сборка до сих пор проходит медленно. Взглянем на логи:

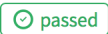






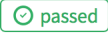






```
gitlab-ci-multi-runner 1.3.2 (0323456)
Using Docker executor with image ruby:2.1 ...
Pulling docker image ruby:2.1 ...
Running on runner-30dcea4b-project-1398078-concurrent-0 via runner-30dcea4b-machine-1469178408-7ba044d5-digital-ocean-4gb...
Cloning repository...
Cloning into '/builds/inem/ci'...
Checking out bdc26c45 as master...
$ cat file1.txt file2.txt > compiled.txt
Uploading artifacts...
compiled.txt: found 1 matching files
Uploading artifacts to coordinator... ok      id=2545753 responseStatus=201 Created token=39RXWAPw

Build succeeded
```

Что, простите? Ruby 2.1?

Зачем тут вообще Ruby? А затем, что GitLab.com использует образы Docker для запуска сборок, а по умолчанию для этого используется образ `ruby:2.1`. Само собой, в этом образе содержится множество пакетов, которые нам ни к чему. Спросив помощи у гугла, узнаем, что существует образ `alpine`, который представляет собой практически «голый» образ Linux.

Для того, чтобы использовать этот образ, добавим `image: alpine` в `.gitlab-ci.yml`. Благодаря этому время сборки сокращается почти на три минуты:

Status	Commit	Compile	Test	Package	
	#3795973  master → 628c2511 latest Update .gitlab-ci.yml				 00:35  about a minute ago
	#3792115  master → 4b5604cc Update .gitlab-ci.yml				 03:27  about 14 hours ago

А вообще, в свободном доступе находится довольно много разных образов, так что можно без проблем подобрать один для нашего стека. Главное — помнить о том, что лучше подходят образы, не содержащие дополнительной функциональности — такой подход минимизирует время скачивания.

Работа со сложными сценариями

Теперь представим, что у нас появился новый заказчик, который хочет, чтобы вместо `.gz` архива наше приложение поставлялось в виде образа `.iso`. Поскольку весь процесс сборки реализован через CI, все, что нам нужно сделать — добавить еще одну задачу. Образы ISO создаются с помощью команды `mkisofs`. В итоге конфигурационный файл должен выглядеть следующим образом:

```
image: alpine

stages:
  - compile
  - test
  - package

# ... задания "compile" и "test" в данном примере пропущены ради краткости

pack-gz:
  stage: package
  script: cat compiled.txt | gzip > packaged.gz
  artifacts:
    paths:
      - packaged.gz

pack-iso:
  stage: package
  script:
    - mkisofs -o ./packaged.iso ./compiled.txt
  artifacts:
```

```
paths:
- packaged.iso
```

Обратите внимание на то, что названия задач не обязательно должны быть одинаковыми. Более того, в таком случае параллельное выполнение задач на одной стадии было бы невозможным. Так что относитесь к одинаковым названиям задач и стадий как к совпадению.

А тем временем сборка не удалась:

🟢 compile 🟢 test 🟢 pack-gz ❌ pack-iso

```
gitlab-ci-multi-runner 1.3.2 (0323456)
Using Docker executor with image alpine ...
Pulling docker image alpine ...
Running on runner-30dcea4b-project-1447361-concurrent-0 via runner-30dcea4b-machine-1469599207-54a20bb2-digital-ocean-4gb...
Cloning repository...
Cloning into '/builds/inem/ci2'...
Checking out 7d771ca2 as master...
Downloading artifacts for compile (2642195)...
Downloading artifacts from coordinator... ok      id=2642195 responseStatus=200 OK token=7PE1aQwt
$ mkisofs -o ./packaged.iso ./compiled.txt
/bin/sh: eval: line 40: mkisofs: not found

ERROR: Build failed: exit code 127
```

Проблема в том, что команда `mkisofs` не входит в состав образа `alpine`, так что нужно установить ее отдельно.

Установка дополнительного ПО

На сайте [Alpine Linux](#) указано, что `mkisofs` входит в состав пакетов `xorriso` и `cdrkit`. Для установки пакета нужно выполнить следующие команды:

```
echo "ipv6" >> /etc/modules # включить поддержку сети
apk update                  # обновить список пакетов
apk add xorriso              # установить пакет
```

Все это — тоже валидные команды CI. Полный список команд в разделе `script` должен выглядеть следующим образом:

```
script:
- echo "ipv6" >> /etc/modules
- apk update
- apk add xorriso
- mkisofs -o ./packaged.iso ./compiled.txt
```

С другой стороны, семантически более корректно выполнять команды, ответственные за установку пакетов до раздела `script`, а именно в разделе `before_script`. При размещении этого раздела в верхнем уровне файла конфигурации, его команды будут выполнены раньше всех задач. Однако в нашем случае достаточно выполнить `before_script` раньше одной определенной задачи.

Итоговая версия `.gitlab-ci.yml`:

```
image: alpine

stages:
- compile
- test
- package

compile:
stage: compile
script: cat file1.txt file2.txt > compiled.txt
artifacts:
paths:
- compiled.txt
```

```

    expire_in: 20 minutes

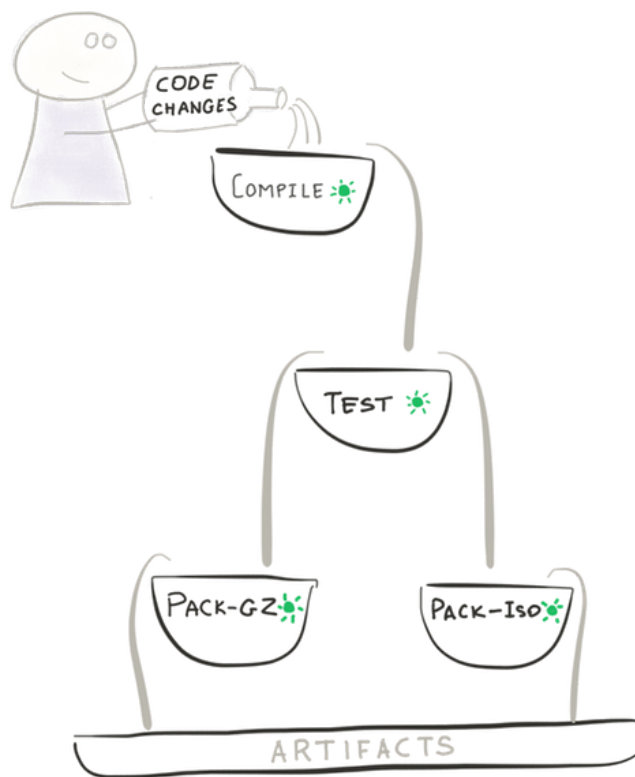
test:
  stage: test
  script: cat compiled.txt | grep -q 'Hello world'

pack-gz:
  stage: package
  script: cat compiled.txt | gzip > packaged.gz
  artifacts:
    paths:
      - packaged.gz

pack-iso:
  stage: package
  before_script:
    - echo "ipv6" >> /etc/modules
    - apk update
    - apk add xorriso
  script:
    - mkisofs -o ./packaged.iso ./compiled.txt
  artifacts:
    paths:
      - packaged.iso

```

А ведь мы только что создали конвейер! У нас есть три последовательные стадии, при этом задачи pack-gz и pack-iso стадии package выполняются параллельно:



Подводя итоги

В этой статье приведены далеко не все возможности GitLab CI, однако пока что остановимся на этом. Надеемся вам понравился этот небольшой рассказ. Приведенные в нем примеры были намеренно тривиальными — это было сделано для того, чтобы наглядно показать принципы работы CI не отвлекаясь на незнакомые технологии. Давайте подытожим изученное:

1. Для того, чтобы передать выполнение определенной работы в GitLab CI, нужно определить одну или более задач в `.gitlab-ci.yml`.
2. Задачам должны быть присвоены названия, советуем делать их осмысленными, чтобы потом самим не запутаться.
3. В каждой задаче содержится набор правил и инструкций для GitLab CI, определяющийся ключевыми словами.

- Задачи могут выполняться последовательно, параллельно, либо вы можете задать свой собственный порядок выполнения, создав конвейер.
- Существует возможность передавать файлы между заданиями и сохранять их как артефакты сборки для последующего скачивания через интерфейс.

В последнем разделе этой статьи приведен более формализованный список терминов и ключевых слов, использованных в данном примере, а также ссылки на подробные описания функциональности GitLab CI.

Описания ключевых слов и ссылки на документацию

Ключевое слово/термин	Описание
.gitlab-ci.yml	Конфигурационный файл, в котором содержатся все определения сборки проекта
script	Определяет исполняемый shell-скрипт
before_script	Определяет команды, которые выполняются перед всеми заданиями
image	Определяет используемый Docker-образ
stage	Определяет стадию конвейера (test по умолчанию)
artifacts	Определяет список артефактов сборки
artifacts:expire_in	Используется для удаления загруженных артефактов по истечению определенного промежутка времени
pipeline	Конвейер — набор сборок, которые выполняются стадиями

Также обратите внимание на другие примеры работы с GitLab CI:

- Migrating from Jenkins to GitLab CI
- Decreasing build time from 8 minutes 33 seconds to just 10 seconds

(Автор перевода — @sgnl_05)

Теги: git, gitlab, gitlab-ci, gitlab ci, continuous integration, ci

↑

+43

↓

🔖

370

👁

111k

💬

17



Softmart

0,00

Компания



49,0

75,2

54

10

Карма

Рейтинг

Подписчики

Подписки

Иван Немытченко @nem

Smartprogrammer.ru

Поделиться публикацией

ПОХОЖИЕ ПУБЛИКАЦИИ

3 января 2017 в 10:37

Вышел GitLab 8.15

↑ +27 👁 14k 📖 66 💬 13

20 сентября 2016 в 14:22

GitLab CI: Учимся деплоить

↑ +26 👁 71,4k 📖 245 💬 3

2 сентября 2016 в 16:55

GitLab Container Registry

↑ +24 👁 26,9k 📖 93 💬 26


Реклама

Комментарии 17

 **ivanych** 7 сентября 2016 в 12:30 🗑 📖 ↑ +1 ↓

```
compile:
stage: compile
script: cat file1.txt file2.txt > compiled.txt
artifacts:
paths:
— compiled.txt
```

А для чего указывать артефакт в задаче compile? Файл compiled.txt понятно, он нам будет нужен в следующих задачах. А зачем его описывать как артефакт?

 **nem** 7 сентября 2016 в 12:58 🗑 📖 📄 🔄 ↑ +1 ↓

artifacts используется и для передачи файлов между stages, и для попадания в downloadable artifacts. Возможно это поменяется, но пока так.

 **ivanych** 7 сентября 2016 в 13:05 🗑 📖 📄 🔄 ↑ 0 ↓


А что значит «для передачи файлов между stages»?

В следующих стадиях/задачах используется имя файла compiled.txt. Я так понимаю, это тот самый файл compiled.txt, который создался в задаче compile. Вот просто создался он в задаче compile и лежит себе, а в следующих задачах мы к нему обращаемся.

Без указания артефакта мы не сможем обратиться к нему в следующей задаче? Он будет удален по окончании задачи compile? А указание артефакта это типа «поставить галочку, что этот файл удалять не надо»?

 **abogoyavlensky** 7 сентября 2016 в 13:34 🗑 📖 📄 🔄 ↑ 0 ↓

Разные задачи могут выполняться на разных раннерах, поэтому нужно передавать артефакты.

 **ivanych** 7 сентября 2016 в 13:38 🗑 📖 📄 🔄 ↑ 0 ↓

А вот в обсуждаемой статье — используется один раннер? В одном раннере сработает обращение к файлу без указания артефакта?






 **DAiMor** 7 сентября 2016 в 22:42 🗑 📖 📄 🔄 ↑ 0 ↓

В статье не указано ничего о том что только один раннер, каждая задача выполняется независимо, есть раннер которые его выполняет. раннеры бывают разные, можно использовать те что бесплатно предоставлены gitlab там их сейчас два, физически они в DigitalOcean
раннеры можно и свои запустить, linux, macos или windows, раннер этот может и не использовать докер вообще, а только локально выполнять команды
поэтому и нужна возможность перетягивать файлы между задачами, но при этом весь репозиторий клонируется на всех задачах

 **past** 7 сентября 2016 в 12:43  

 0 

Подскажите, можно ли иметь единый .gitlab-ci.yml для всех веток?

 **ivanych** 7 сентября 2016 в 13:19    




 0 

Так это же просто файл в репозитории. Он одинаковый во всех ветках. Как он может быть разным? Если только Вы специально измените его в конкретной ветке, но зачем это делать, если Вам как-раз надо, чтобы он был одинаковым.

 **past** 7 сентября 2016 в 14:30    

 0 

На стадии его написания многократно приходится синхронизировать его между ветками.

 **ivanych** 7 сентября 2016 в 14:50    

 0 






Ну да, как и любой другой файл. Вы в любой случае все ветки синхронизируете с мастером, заодно и новый файл прилетит в ветку.

 **past** 7 сентября 2016 в 14:32  

 +1 

И еще вопрос, можно как-то динамически формировать имя артефакта?

У меня после сборки получается файл `package-${version}-${release}.el7.centos.${arch}.rpm` при чем переменные вычисляются в процессе сборки.

 **DAiMor** 7 сентября 2016 в 22:33    

 0 

Можно самому упаковать папку и задать имя на основе переменных, есть ряд переменных которые gitlab сам формирует если переменные вычисляются, то они могут попасть в переменные окружения и использоваться для формирования имени
нужно учитывать что блок `scripts`, это команды операционной системы и выбранного шелла, да ведь раннер может быть и на винде и команды могут быть powershell

 **utoplenick** 7 сентября 2016 в 16:51  

 0 

Было бы неплохо рассказать и о раннерах, раз уж зашла речь про gitlab-ci, потому как в данном примере совсем непонятно зачем вообще нужен докер чтобы грепнуть файл? Понятно что пример простой, а статья — переводю

 **hippoage** 8 сентября 2016 в 11:51  

 +1 

Функционала меньше, чем в Jenkins (это нормально, нужно учитывать):

- нет постоянных ссылок на скачивание последних артефактов (latest, а не номер билда; удобно для скриптов бутстрапа, например)
- вроде бы в последних версиях ручной запуск появился, но нет параметризованного запуска (может и не нужно, можно тот же список серверов забить в файл отдельными ручными работами, но нужно учитывать при проектировании)
- нет работ не привязанных к ветке/репозиторию (из-за этого придется для некоторых вещей сохранить Jenkins)
- нотификации (типа интеграции со slackом) идут вне файла настройки ci

Текущие минусы:

- долго промучился, но команды сборки докера в докере не заработали (ни `docker in docker`, который не предназначен для систем CI и постоянно в документации Gitlab CI упоминается; ни пробрасывание сокета докера), использую отдельный shell runner для этого.
- кеширует только после успешного завершения работы (например, пока настраиваешь работу каждый раз качает плагины maven)
- медленно (пофайлово?) сохраняет кеш
- нет способа через UI сбросить кеш и посмотреть какие кеши есть. все-таки кеши время от времени приходится сбрасывать
- по ощущениям собирает медленней Jenkins, относительно долго (секунд 15) запускается любая докер-работа с одной командой `echo`
- нет готового рецепта для Java/Maven, что-то настроил, но еще не все

В целом, удобно, что интегрировано, но еще сыро.

 fshp 8 сентября 2016 в 16:46



 0 

Можно ли указать в качестве зависимости другой проект, что бы он тоже подтянулся для сборки?
И можно ли шарить артефакты между проектами?

 ALexhha 8 сентября 2016 в 17:22



 0 

Такой вопрос, столкнулся с казалось бы тривиальной задачей в Jenkins — необходимо запускать job при попадании нового тега (с определенным именем, например `build_[0-9]{6,10}`). Казалось бы все должно быть очень просто, а на деле оказалось что нет. Например у нас есть следующая история

```
550313e -> 22ce31f -> e31e663 -> e4c4bf6 (head)
tag2      tag3      tag4      tag1
```

Так вот в Jenkins сборка будет запущена только для head, для остальных 3х тегов она будет игнорироваться. Можно ли в gitlab ci обеспечить сборку в любом "направлении", чтобы при командах

```
$ git tag build_000001 e4c4bf6
$ git tag build_000002 550313e
$ git tag build_000003 22ce31f
$ git tag build_000004 e31e663
```

я получил в итоге 4 билда?

 ivanych 12 сентября 2016 в 15:18



 0 

Ничего из описанного не работает:(Вообще не запускается, появляется значок «pending» и всё.

Я так понимаю, нужен некий раннер. Какой минимум нужно сделать, чтобы запустить хотя бы самый простейший тест?

Только полноправные пользователи могут оставлять комментарии. Войдите, пожалуйста.

САМОЕ ЧИТАЕМОЕ

Сутки

Неделя

Месяц

Как американцы живого хорька в коллайдер засунули

 +63  33,4k  36  37

Что айтишнику не стоит делать в 2020?

 +33  29,2k  92  58

В Windows 10 версии 2004 можно отслеживать температуру видеокарты, а новые драйверы будут помечать как обновления

 +15  21,7k  2  37

Еще один способ высокотехнологичного мошенничества

 +73  35,8k  79  91

Прионы — страх и ужас будущего

Ваш аккаунт	Разделы	Информация	Услуги
Войти	Публикации	Правила	Реклама
Регистрация	Новости	Помощь	Тарифы
	Хабы	Документация	Контент
	Компании	Соглашение	Семинары
	Пользователи	Конфиденциальность	Мегaproекты
	Песочница		

Если нашли опечатку в посте, выделите ее и нажмите Ctrl+Enter, чтобы сообщить автору.

