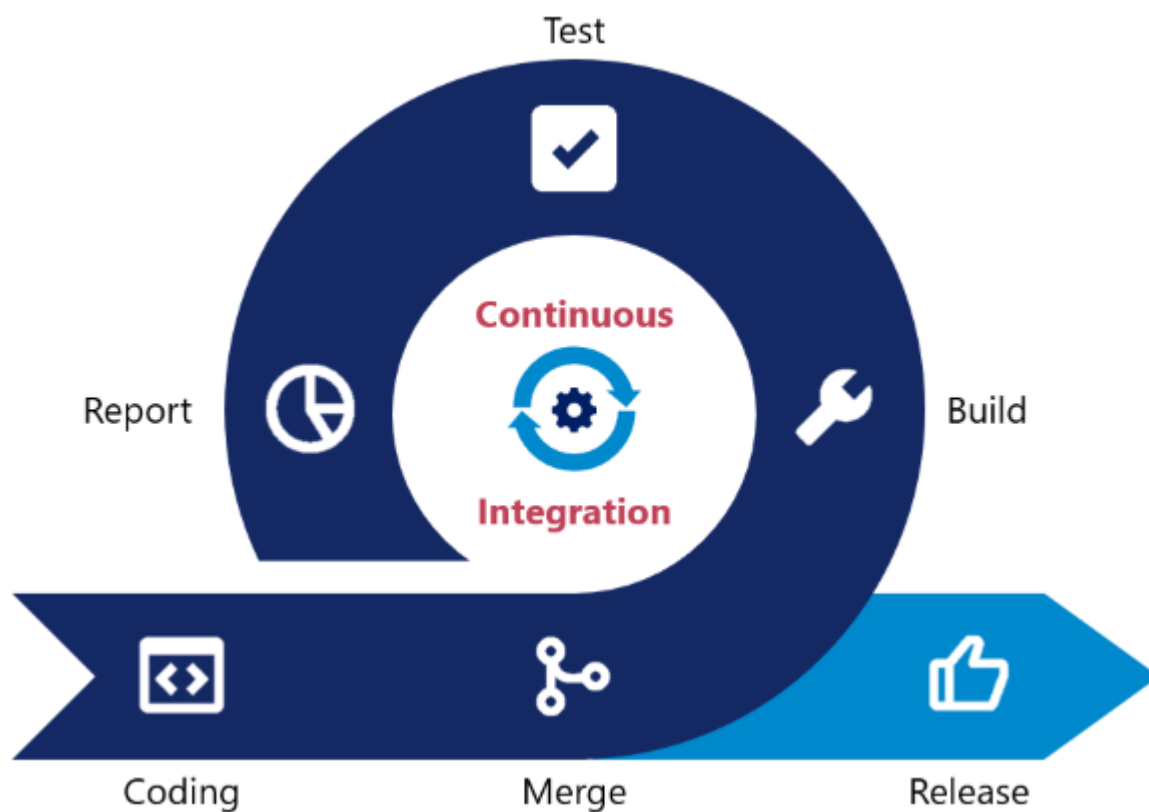


CI/CD

# Continuous Integration

---





## На этом уроке

1. Более подробно рассмотрим CI.
2. Научимся делать merge request'ы.
3. Узнаем, что такое job artifacts и как их добавить в GitLab Registry.
4. Узнаем, для чего нужны теги.
5. Разберёмся, как использовать cache для уменьшения времени выполнения job.

## Оглавление

[Merge requests](#)

[Job artifacts](#)

[artifacts:paths](#)

[artifacts:exclude](#)

[artifacts:name](#)

[artifacts:expire\\_in](#)

## [Gitlab Registry](#)

[Авторизация в GitLab Registry](#)

[Пример GitLab CI/CD с размещением docker image в Container Registry](#)

[Удаление docker images](#)

[Примеры регулярных выражений](#)

## [Tags](#)

## [Cache](#)

[Хранение cache по умолчанию](#)

[Сохранение и извлечение cache](#)

[Cache и artifacts](#)

[Cache](#)

[Artifacts](#)

[Практическое задание](#)

[Глоссарий](#)

[Дополнительные материалы](#)

[Используемые источники](#)

Как мы уже говорили на предыдущем занятии, Continuous Integration (CI) — это непрерывная интеграция. На сегодняшнем уроке мы более подробно рассмотрим основные инструменты CI.

## Merge requests

На предыдущем занятии мы производили все изменения прямо в master-е репозитория. Это не совсем правильно, потому что в master-ветке должен быть протестированный и рабочий код, который можно раскатить в production.

Обычно вся разработка ведётся в отдельной ветке репозитория. Это позволяет разработчикам независимо друг от друга изменять и тестировать код, не ломая «эталонный» master.

Создать ветку можно двумя способами:

1. Из консоли, предварительно клонировав нужный репозиторий:

```
git checkout -b название-ветки
```

2. Через веб-интерфейс самого GitLab: Repository → Branches → New branch.

После того как были протестированы все изменения в созданной ветке, создаётся запрос на merge в master (Merge request), чтобы протестированный код попал в master-ветку и в дальнейшем в production.

По дефолту GitLab запускает pipeline при любом push в ветку: как в master-ветку, так и в свою. Но это может быть изменено.

Может возникнуть необходимость запускать job'ы только в ветке разработчика, которая в дальнейшем будет мержиться в master. Это так называемая a branch that is associated with a merge request. Например, так запускают тесты разрабатываемой фичи. В таком случае можно использовать pipelines для merge request'ов.

Это можно сделать двумя способами:

- с помощью **rules**;
- с помощью **only** или **except**.

Подробнее об этих способах можно прочитать в [GitLab Docs](#). Здесь же мы разберём самые простые из них — only и except.

Чтобы обозначить, что job должна выполняться только на commit'ах в ветке, необходимо в каждой job указать:

```
only: - merge_requests
```

Обратите внимание, что в примере ниже `only: - merge_requests` указан только в job'e test.

Для build и deploy он не указан, значит, build и deploy не будут запускаться при изменении в ветке.

```
build:
  stage: build
  script: ./build
  only:
    - master

test:
  stage: test
  script: ./test
  only:
    - merge_requests

deploy:
  stage: deploy
  script: ./deploy
  only:
    - master
```

**Внимание!** Конструкция такого вида не будет работать как задумано:

```
test:
  only: [merge_requests]
  except: [/^docs-/]
```

Чтобы job test работала на всех изменениях в ветке, за исключением веток, имя которых начинается на docs, нужно использовать конструкцию:

```
test:
  only: [merge_requests]
  except:
    variables:
      - $CI_COMMIT_REF_NAME =~ /^docs-/
```

## Job artifacts

Job artifacts (артефакты job) — это список файлов и директорий, созданных в результате работы job.

Артефакты, созданные на GitLab Runner'е на GitLab, доступны для скачивания и просмотра.



По умолчанию артефакты аплоадируются, только когда job успешно отработала, но также можно настроить, чтобы артефакты аплоадились всегда (always) или когда job сфейлилась (fails). Для этого используйте параметр `artifacts:when` parameter.

## artifacts:paths

Путь, по которому будут браться артефакты. Указывается относительно директории проекта (\$CI\_PROJECT\_DIR):

```
artifacts:
```

```
paths:
  - binaries/
  - .config
```

В указанном примере в качестве артефактов будут директория `binaries` и файл `.config`, лежащие в корне репозитория.

## artifacts:exclude

Exclude позволяет указывать файлы, которые НЕ будут включены в артефакты.

Так же, как и в `artifacts:paths`, в `artifacts:exclude` указывается путь к файлу относительно корня репозитория.

```
artifacts:
  paths:
    - binaries/
  exclude:
    - binaries/**/*.o
```

## artifacts:name

С помощью `artifacts:name` можно задавать имя для архива артефактов (имя архива при скачивании):

```
job:
  artifacts:
    name: "$CI_JOB_NAME"
    paths:
      - binaries/
```

## artifacts:expire\_in

Также можно задавать время хранения артефактов с помощью `expire_in`:

```
pdf:
  script: xelatex mycv.tex
  artifacts:
    paths:
      - mycv.pdf
    expire_in: 6 days
```

В таком случае в job artifacts вы увидите, что артефакт будет удалён через 6 дней.



Вы можете оставить этот артефакт, нажав на Кеер. Тогда конкретно этот артефакт не будет удалён.

Это основные и часто используемые параметры артефактов. Об остальных можно почитать в [GitLab Docs](#).

## Gitlab Registry

Registry — это хранилище. В GitLab есть возможность хранить докер-образы (docker image) и собранные пакеты в Container Registry и Package Registry соответственно.

О Package Registry можно почитать самостоятельно в [GitLab Docs](#), а мы рассмотрим Container Registry.

Чтобы собрать и отправить docker image, нужно сначала написать dockerfile. Возьмём пример с предыдущего урока и упакуем его в dockerfile:

```
FROM busybox:latest
RUN mkdir build &&\
  cd build &&\
  touch house.txt &&\
  echo "walls" >> house.txt \
  echo "roof" >> house.txt \
  echo "floor" >> house.txt \
  echo "doors" >> house.txt \
  echo "window" >> house.txt
```

Дальше необходимо добавить в gitlab\_ci.yml шаг со сборкой докер-образа из dockerfile.

```

variables:
  BUILD_IMAGE: $CI_REGISTRY_IMAGE:${CI_COMMIT_REF_SLUG}

stages:
  - build
  - test
  - pages

docker build the house:
  # Official docker image.
  image: docker:latest
  stage: build
  services:
    - docker:dind
  script:
    - docker login -u "$CI_REGISTRY_USER" -p "$CI_REGISTRY_PASSWORD"
    $CI_REGISTRY
    - docker build -t $BUILD_IMAGE .
    - docker push $BUILD_IMAGE

```

## Авторизация в GitLab Registry

Чтобы разместить docker image в registry, необходимо авторизоваться. Это можно сделать несколькими способами:

1. Использовать специальную переменную `CI_REGISTRY_USER`. Также автоматически создаётся пароль для этого юзера, который хранится в переменной `CI_REGISTRY_PASSWORD`. Это даёт возможность автоматически собирать (build) и деплоить (deploy) docker images, а также даёт доступ на чтение и запись в Registry. Переменная валидна только для одной job. Пример:

```
docker login -u $CI_REGISTRY_USER -p $CI_REGISTRY_PASSWORD $CI_REGISTRY
```

2. Использовать [GitLab Deploy Token](#). Создать GitLab deploy token можно в Settings → Repository → Deploy Token.

Плюс такого метода в том, что эту переменную можно использовать в репозитории многократно. Пример docker login с использованием GitLab deploy token:

```
docker login -u $CI_DEPLOY_USER -p $CI_DEPLOY_PASSWORD $CI_REGISTRY
```



3. Использовать персональный токен (personal access token). Как создать персональный токен, можно почитать в [GitLab Docs](#).

Пример docker login с использованием personal access token:

```
docker login -u <username> -p <access_token> $CI_REGISTRY
```

## Пример GitLab CI/CD с размещением docker image в Container Registry

```
build:
  image: docker:19.03.12
  stage: build
  services:
    - docker:19.03.12-dind
  variables:
    IMAGE_TAG: $CI_REGISTRY_IMAGE:$CI_COMMIT_REF_SLUG
  script:
    - docker login -u $CI_REGISTRY_USER -p $CI_REGISTRY_PASSWORD $CI_REGISTRY
    - docker build -t $IMAGE_TAG .
    - docker push $IMAGE_TAG
```

Здесь `$CI_REGISTRY_IMAGE` — адрес репозитория проекта.

`$CI_COMMIT_REF_NAME` — имя ветки или тег. Поскольку ветка может содержать слеш (», например feature/my-feature), более безопасно использовать переменную `$CI_COMMIT_REF_SLUG` для image tag, так как Image tags не может содержать прямых слешей.

## Удаление docker images

Удалять образы также можно несколькими способами:

1. В самом Container Registry. Packages & Registries → Container Registry, нажать красную иконку с мусорным баком (Trash).
2. С помощью GitLab CI/CD. В примере ниже два stage: build и clean. Первый собирает image, второй удаляет его. В примере удаляется image, который подходит под переменную `$CI_PROJECT_PATH:$CI_COMMIT_REF_SLUG`.

```
stages:
  - build
```

```

- clean


build_image:
  image: docker:19.03.12
  stage: build
  services:
    - docker:19.03.12-dind
  variables:
    IMAGE_TAG: $CI_REGISTRY_IMAGE:$CI_COMMIT_REF_SLUG
  script:
    - docker login -u $CI_REGISTRY_USER -p $CI_REGISTRY_PASSWORD
      $CI_REGISTRY
    - docker build -t $IMAGE_TAG .
    - docker push $IMAGE_TAG
  only:
    - branches
  except:
    - master

delete_image:
  image: docker:19.03.12
  stage: clean
  services:
    - docker:19.03.12-dind
  variables:
    IMAGE_TAG: $CI_PROJECT_PATH:$CI_COMMIT_REF_SLUG
    REG_SHA256:
ade837fc5224acd8c34732bf54a94f579b47851cc6a7fd5899a98386b782e228
    REG_VERSION: 0.16.1
  before_script:
    - apk add --no-cache curl
    - curl --fail --show-error --location
      "https://github.com/genuinetools/reg/releases/download/v$REG_VERSION/reg-l
      inux-amd64" --output /usr/local/bin/reg
    - echo "$REG_SHA256 /usr/local/bin/reg" | sha256sum -c -
    - chmod a+x /usr/local/bin/reg
  script:
    - /usr/local/bin/reg rm -d --auth-url $CI_REGISTRY -u
      $CI_REGISTRY_USER -p $CI_REGISTRY_PASSWORD $IMAGE_TAG
  only:
    - branches
  except:
    - master

```

3. Удалять image, используя Cleanup Policy. Settings → CI/CD → Cleanup policy for tags.

Tag expiration policy

Cleanup policy:  **Enabled** - Tags matching the patterns defined below will be scheduled for deletion

Expiration interval:

90 days until tags are automatically removed

Expiration schedule:

Every day

Number of tags to retain:

10 tags per image name

Tags with names matching this regex pattern will *expire*:

\*

\*

Wildcards such as `.*-test` or `dev-.*` are supported. To select all tags, use `.*`

Tags with names matching this regex pattern will *be preserved*:

Wildcards such as `.*-master` or `release-.*` are supported

Cancel

Set cleanup policy

Поле	Описание
Cleanup policy	Включить или выключить настройку по удалению
Expiration interval	Время жизни tag'a
Expiration schedule	Как часто запускать чистку
Number of tags to retain	Сколько tag'ов <b>всегда</b> хранить для каждого image
Tags with names matching this regex pattern expire	Регулярное выражение, которое описывает, какие tag'и должны быть удалены
Tags with names matching this regex pattern are preserved	Регулярное выражение, которое описывает, какие tag'и должны быть сохранены

## Примеры регулярных выражений

Все теги (tags):

```
.*
```

Tag'и, начинающиеся на v:

```
v.+
```

Tag'и, включающие в себя master:

```
master
```

Tag'и, которые начинаются на v и включают в себя слова master и release:

```
(?:v.+|master|release)
```

## Tags

Под tags имеются в виду git'овые теги, которые позволяют пометать/выделять commit'ы.

Теги могут быть полезными при маркировании релизов, версий кода. Например, протестированный merge request обычно помечается тегом, и этот commit будет с тегом, допустим, version\_0.1. Следующий тег будет version\_0.2. Это удобно тем, что мы можем оперировать простыми и понятными значениями — тегами, а не хешами commit'ов. Плюс такая последовательность будет понятна человеку, и в ней проще ориентироваться.

Подробнее о создании тегов можно почитать на [сайте git-scm.com](https://git-scm.com).

## Cache

Обычно самый долгий этап пайплайна — build, потому что на этом этапе происходит установка различных зависимостей, необходимых для выполнения job.

Это происходит потому, что каждая job начинает свою работу с чистым окружением (environment), в отличие от «традиционных» CI, таких как, например, Jenkins, где зависимости собираются один раз и хранятся в отдельной директории.

Когда на GitLab Runner'е запускается job, происходит следующее:

1. Скачивается и запускается docker-образ.
2. Клонировается репозиторий.
3. Устанавливаются все необходимые зависимости.
4. Выполняются шаги, указанные в описании job.
5. Сохраняется результат (при необходимости).

Использование cache может ускорить выполнение job.

Cache указывает, какие файлы стоит сохранить, чтобы в дальнейшем их можно было повторно использовать.

Чтобы job внутри одной ветки всегда использовали один и тот же cache, необходимо использовать переменную в `cache key` `${CI_COMMIT_REF_SLUG}` и файлы, которые необходимо кешировать в `cache:path`:

```
cache:
  key: ${CI_COMMIT_REF_SLUG}
  paths:
  public/
```

Cache можно указывать как внутри конкретной job, так и вне job. В последнем случае cache будет глобальным.

Это может быть необходимо, если несколько job должны использовать один и тот же кеш. Например, job build и job test. Первая собирает build со всеми зависимостями, вторая тестирует build, предварительно также устанавливая зависимости.

Всё, что указано в `cache:path` в `gitlab_ci.yml`, архивируется в архив `cache.zip` и хранится на GitLab.

## Хранение cache по умолчанию

GitLab Runner executor	Дефолтный путь
Shell	Локально на GitLab Runner в домашней директории пользователя: <code>/home/gitlab-runner/cache/&lt;user&gt;/&lt;project&gt;/&lt;cache-key&gt;/cache.zip</code>

Docker	Локально в Docker volumes: /var/lib/docker/volumes/<volume-id>/_data/<user>/<project>/<cache-key>/cache.zip
--------	--

## Сохранение и извлечение cache

Разберём на примере, как происходит сохранение и извлечение cache:

```

stages:
  - build
  - test

before_script:
  - echo "Hello"

job A:
  stage: build
  script:
    - mkdir vendor/
    - echo "build" > vendor/hello.txt
  cache:
    key: build-cache
    paths:
      - vendor/
  after_script:
    - echo "World"

job B:
  stage: test
  script:
    - cat vendor/hello.txt
  cache:
    key: build-cache

```

Что происходит в примере:

1. Запускается pipeline.
2. Запускается job A.
3. Выполняется before\_script.
4. Выполняется script.
5. Выполняется after\_script.
6. Запускается cache, и директория vendor/ архивируется в cache.zip. Архив сохраняется по дефолтному пути с именем cache: key.
7. Запускается job B.

8. Извлекается cache (если существует).
9. Выполняется `before_script`.
10. Выполняется `script`.
11. Завершается pipeline.

При использовании cache стоит учитывать две вещи:

1. Если какая-то другая job в процессе работы сохранит cache с таким ZIP-именем, то cache перезапишется. Таким образом, если в двух разных job будут указаны разные path, но одинаковые cache key, cache перезапишется.
2. При извлечении cache из `cache.zip`, всё, что находится внутри архива, извлекается в рабочую директорию job. Обычно это спулленный репозиторий. И GitLab Runner не следит за тем, что архив job'ы А перезапишется данными из архива job'ы Б.

## Cache и artifacts

На первый взгляд cache и artifact кажутся очень похожими сущностями. Посмотрим, в чём они похожи и в чём разница.

### Cache

1. Не используется, если явно не указать в job или глобально с помощью `cache:`.
2. Доступен для всех job, если используется глобально.
3. Хранится на GitLab Runner.

### Artifacts

1. Не используется, если явно не определён в job с помощью `artifacts:`.
2. Доступен в пределах указанной job, не может быть глобально доступным.
3. Хранится в GitLab.
4. Для artifact можно задать время жизни. По умолчанию — 30 дней.

## Практическое задание

См. задание к лекции

## Глоссарий

**Merge request** — запрос за merge изменений из своей ветки в master-ветку.

**Job artifacts** (артефакты job'ы) — это список файлов и директорий, созданных в результате работы job.

**Registry** — это хранилище. В основном служит для хранения docker images, пакетов, архивов целых приложений.

## Дополнительные материалы

1. [Официальная документация по GitLab CI.](#)
2. [Тегирование в git.](#)

## Используемые источники

1. Бетси Бейер, Крис Джоунс, Дженнифер Петофф, Нейл Ричард Мёрфи «Site Reliability Engineering. Надёжность и безотказность, как в Google», 2018.
2. [Официальная документация по GitLab CI.](#)
3. [Pipelines для merge request'ов.](#)
4. [Job Artifacts.](#)
5. [Артефакты.](#)
6. [GitLab Deploy Token.](#)
7. [GitLab Container Registry.](#)
8. [Cleanup policy.](#)
9. [Примеры регулярных выражений для cleanup policy.](#)