ozon{tech

Обзор GitLab CI/CD

План занятия

Это занятие начинает новую большую тему — CI/CD. До этого мы писали только автотесты, но пока они не запускаются автоматически, можно считать, что их нет. Только регулярные запуски дают настоящий результат: воспроизводимость, быструю обратную связь, доверие к результатам. Поэтому тесты необходимо встраивать в процесс выпуска релиза чем раньше, тем лучше. Об этом и поговорим на занятиях шестой недели.

CI/CD

CI/CD — процесс того, как вновь написанный код попадает в продакшн. Наверняка вы уже встречали эту аббревиатуру. Повторим, что она означает.

В свежем, майском, материале на эту тему от RedHat для иллюстрации базовых принципов CI/CD используется такая картинка:

Первое, на что стоит обратить внимание: по умолчанию все этапы автоматические. Как изменения в коде, так и деплой в production не предполагают ручного вмешательства. Автоматизация обеспечивает повторяемость процесса.

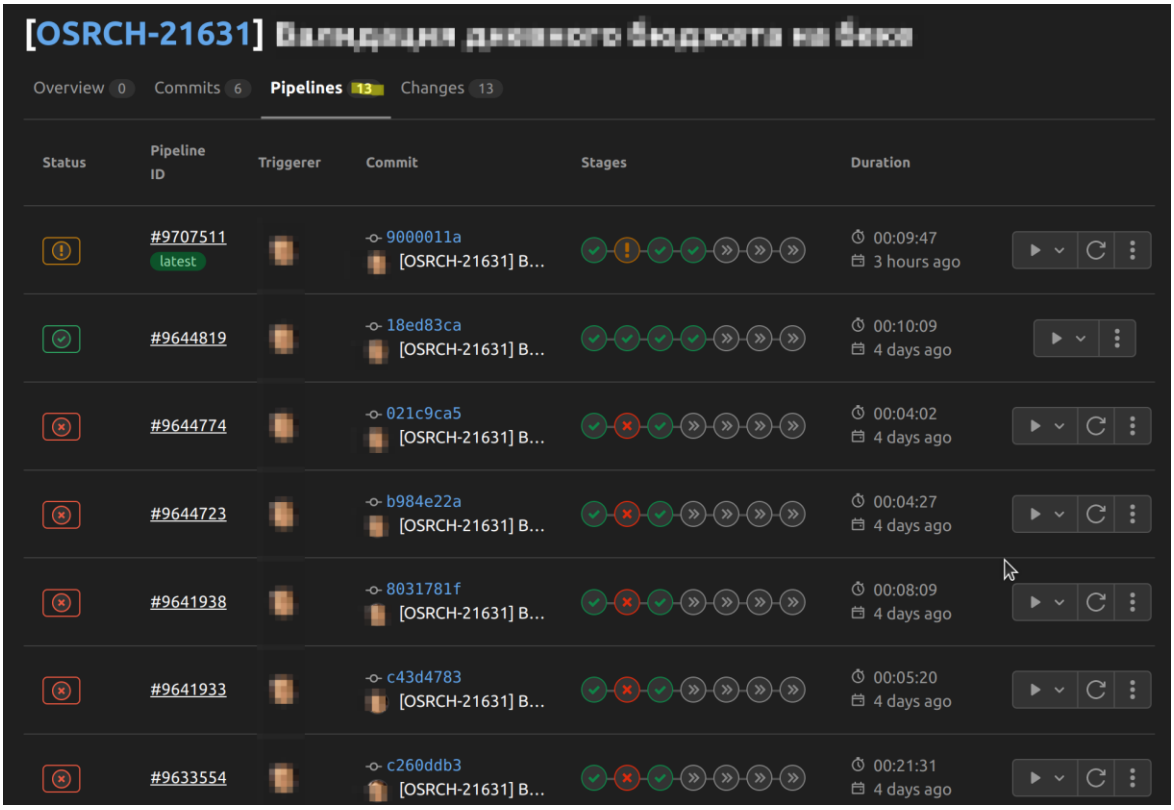
Второе: CD — не только continuous delivery, но и continuous deployment. В чем отличие? Доставка (delivery) предполагает, что сборка, готовая к развертыванию в production, создается автоматически, развертывание (deployment) дальше уже работает с этой сборкой. В теме про docker под сборкой понимался docker-образ. Обратите внимание: код микросервиса может собираться несколько раз, в ходе CI/CD могут создаваться разные образы, но не все эти артефакты попадут в продакшн.

Еще очевиднее разница между continuous deployment и continuous delivery станет после знакомства с позицией Мартина Фаулера (автора книг и статей про микросервисы и не только: https://martinfowler.com/). Он говорит, что continuous delivery подразумевает **возможность** часто развертываться в продакшн, при этом ее необязательно использовать, если она не востребована бизнесом.

Какие проблемы существовали до подхода CI и были им решены? Разработчик, кроме написания кода, должен заботиться о том, может ли новый код быть влит (merge) в репозиторий, быть собран и пройдут ли на нем тесты. В подходе CI он отправляет свой код в проект несколько раз в день, и почти сразу может проверить результаты интеграции кода. Чем выше частота отправки, тем меньше проблем с интеграцией изменения. Так, CI — не только автоматизация, но и образец для того,

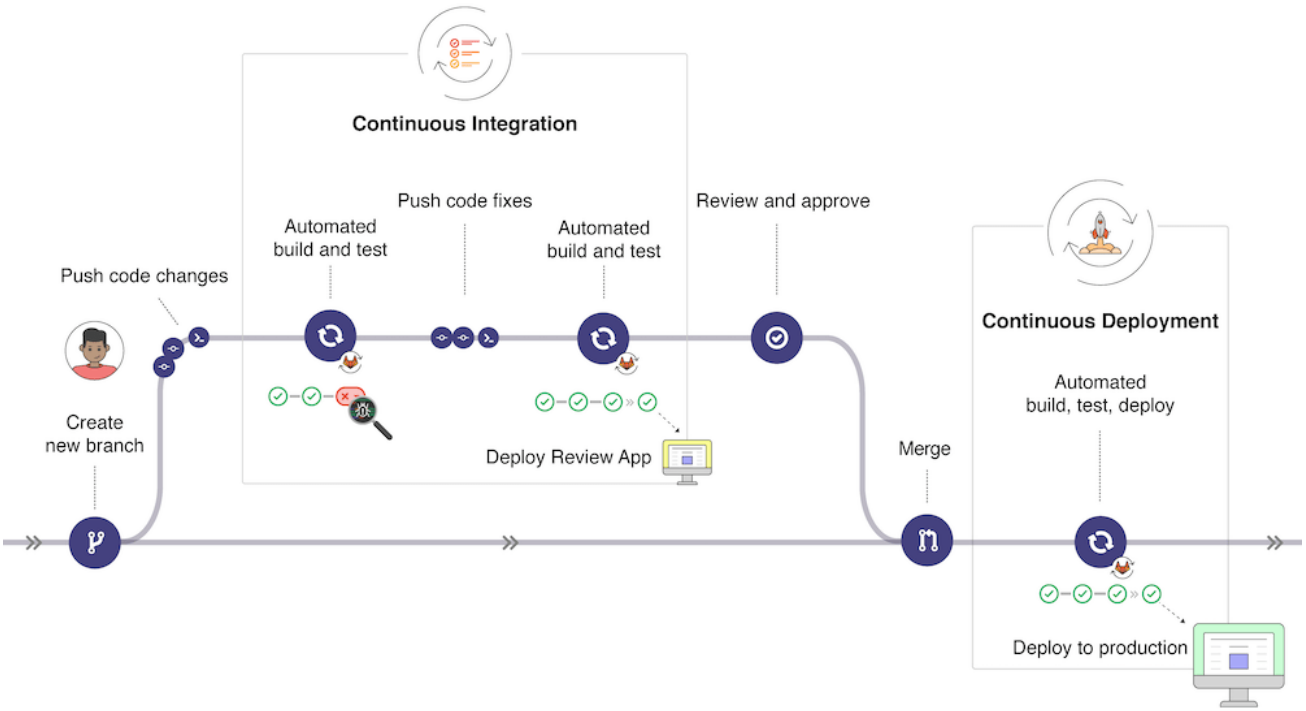
кто пишет код. Тесты — часть CI (и, возможно CD тоже), с ними контроль качества (Quality Control, QC) встроен в процесс работы с кодом.

Вот как это может выглядеть в пайплайне:



Тут мы видим больше 10 пайплайнов, проверяющих в CI изменения.

В Ozon мы используем GitLab. Так как дальше речь пойдет о нем, дадим ссылку на описанный в GitLab процесс: https://docs.gitlab.com/ee/ci/introduction/#gitlab-cicd-workflow.



Эта картинка — более детализированное представление первой. На ней видно, что между CI и CD все-таки может быть фаза ручного вмешательства — код-ревью.

Итак, еще раз повторим важные свойства CI/CD:

- Повторяемость.
- Автоматизация процесса.
- Все это код (и код — это все, что нужно сделать вручную).
- Качество встроено в процесс.
- Частая обратная связь, короткие итерации.

GitLab

В Ozon выбран GitLab в качестве единственного инструмента CI/CD. Разрабатывает его одноименная компания GitLab Inc.

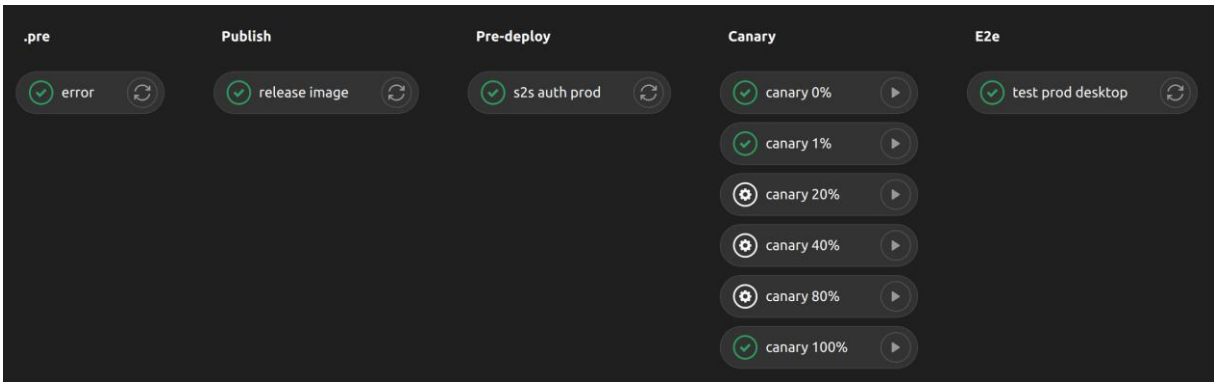
CI/CD — только часть платформы GitLab, инструментария достаточно, чтобы все, что касается разработки, было в GitLab. Возможности GitLab шире возможностей, например, не менее популярного GitHub. Подробно об их сравнении, например, тут: https://about.gitlab.com/devops-tools/github-vs-gitlab/. А сравнения с другими тут: https://about.gitlab.com/handbook/marketing/strategic-marketing/competitive/cicd/#competitor-scope---continuous-integration.

Термины GitLab

Чтобы двигаться дальше, нам понадобятся следующие термины:

- Job (задача) — обособленная по цели задача сборки (например, прогон линтера или сборка контейнера).
- Stage (этапы) — при работе с задачами важна последовательность выполнения, в-основном для этого служат stages (также ключевые слова rules, needs, dependencies, но о них позже)
- Pipeline (пайплайн) — процесс CI/CD может состоять из одной или нескольких (в Ozon — трех) пайплайн, собранных из jobs, объединенных в stages.
- Runner (раннер) — хост, на котором выполняется job. Может быть контейнером или физической машиной с нужными настройками, выбор runner управляется тегом.

Давайте посмотрим на пайплайн выкатки в продакшн уже готового сервиса (как раз CD):



Canary 0%, test prod desktop — это названия задач, .pre, Publish, Pre-deploy, Canary — названия этапов.

Как запускать: .gitlab-ci.yml

Описание jobs и взаимодействия между ними содержатся в yml-файлах. По умолчанию они лежат в корне проекта, не рекомендуется менять их расположение.

Внутри файла можете найти YAML-конфиг, который знаком нам по темам про docker-compose, но в GitLab синтаксис часто сложнее, поэтому кое-что вспомним.

- Для форматирования файлы используются пробелы.
- В строках мы видим пары key: value.
- Синтаксис case sensitive.
- YAML — суперсет JSON, вы можете конвертировать файлы из одного формата в другой. Но в YAML можно использовать комментарии, которые могут начинаться в любом месте с #.
- Кроме простых типов данных (строки, int, т. п.), используются словари и списки.
- Можно ссылаться на другие части YAML, используя anchors.

В отличие от Jenkins, например, .gitlab-ci.yml — декларативное описание того, как должны собираться и проходить пайплайны. Это, скорее, конфигурация, чем код, поэтому смело работайте с ним как с конфигурацией.

include

Общий для всех go-сервисов, которые попадают в production. Это требование платформы Озон. Так, мы получаем неснижаемую планку проверок (например, использования нужных версий библиотек, отсутствия токенов в файлах и т. п.), возможность централизации этих проверок, а владельцы сервиса — поддержку, если что-то пойдет не так.

Так что .gitlab-ci.yml go-сервиса в Ozon будет начинаться с include.

```
include:
  - project: pub/ci
    ref: 0.0.5
    file: .go.gitlab-ci.yml
```

Include — это механизм, который позволяет переиспользовать jobs (и другие сущности) из одного проекта в других. Сам файл из примера будет лежать в нашем же GitLab по пути: pub/ci/.go.gitlab-ci.yml.

Include нужен, чтобы также встраивать тесты из стороннего репозитория. Более того, описание подключаемых файлов и сами тесты тоже могут находиться в разных

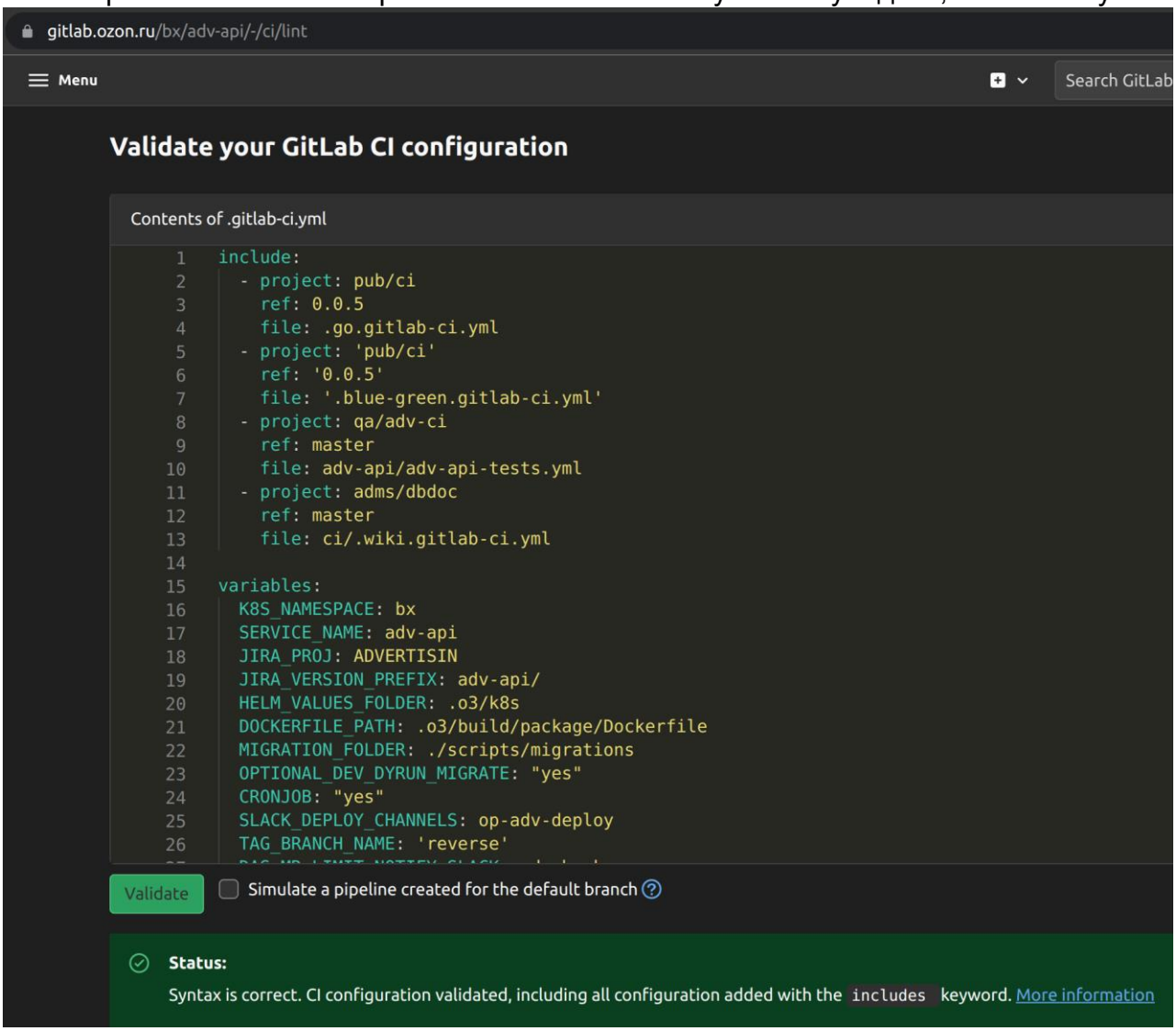
репозиториях:

```
include:
  - project: qa/adv-ci
    ref: master
    file: adv-api/adv-api-tests.yml
```

adv-api-tests.yml 983 Bytes

```
include:
  - project: qa/adv-ui
    ref: master
    file: pipeline/wdio.yml
  - project: qa/adv-public-api
    ref: master
    file: pipeline/.base.gitlab-ci.yml
  - project: qa/adv-moderation-e2e
    ref: master
    file: .common.gitlab-ci.yml
```

И тут не избежать отладки: раз итоговые пайплайны проекта собираются из нескольких файлов, в них могут пересекаться имена jobs, переменные и т. п. Для этого есть CI Lint, в который можно скопировать описание из yml и увидеть, что получится.



script и все-все-все

Внутри job обычно находится shell-команда, часто используют make-alias. При этом там может быть и сложный скрипт или даже код на Python/Golang. В Ozon наши SRE управляют образом-раннером по умолчанию, он содержит специфичные для Ozon утилиты. GitLab использует unix exit code, чтобы понять, успешно ли выполняется job. На примере job, выполняющей go-тесты, мы видим использование почти всех keywords GitLab.

gitlab.ozon.ru/pub/ci/-/blob/0.0.5/templates/go/.go.gitlab-ci.yml

Menu

```yaml
.test:
  extends: .go
  tags: [tests, k8s]
  variables:
    GO_GENERATE: "no"
    GO_TEST_ARGS: "-covermode count"
    GO_COVER_EXCLUDE: ""
  before_script:
    - git config --global credential.helper store && echo "https://gitlab-ci-token:${CI_JOB_TOK
    - test "${GO_GENERATE}" = "yes" && go generate -x ./... || true
  script:
    - ulimit -n 65535
    # https://github.com/jstemmer/go-junit-report/issues/71
    - set -o pipefail; gotest -v ./... $GO_TEST_ARGS -coverprofile=cover.out.tmp 2>&1 | tee /de
  after_script:
    - grep -vE "${GO_COVER_EXCLUDE}" cover.out.tmp > cover.out || cp cover.out.tmp cover.out
    - go tool cover -func cover.out
  cache:
    key: "go-tests-cache"
    policy: pull
    paths:
      - .cache/go
      - .cache/gocache
  artifacts:
    when: always
    paths:
      - test.out
    expire_in: 1 day
    reports:
      junit: junit.xml
  coverage: '/^total:\s+\(statements\)\s+\d+.\d+%$/'
  allow_failure: false
  rules:
    - if: '$DEPLOY_TAG || $CI_COMMIT_TAG || $CI_PIPELINE_SOURCE == "merge_request_event"'
      when: never
    - if: '$CI_COMMIT_BRANCH && $CI_COMMIT_REF_NAME =~ /^(hotfix\/.+)$/'
      allow_failure: true
    - when: always
  timeout: 15 minutes
```

Поясним некоторые:

- Job по умолчанию блокирует выполнение пайплайна. Если job завершится неуспешно, остальной pipeline не запустится. Это поведение можно изменить, используя allow_failure.
- На результат выполнения job влияют только команды из before_script и script.
- После сборки job может оставить artifact (сборка — например, в Ozon в artifact можно найти тестовые сборки для мобильных устройств). Если указать, что артефакт является не просто файл, а отчет о тестировании (например junit), результаты тестов из него будут видны и в GitLab.
- Job может не создаваться для ветки, правила создания его указаны в rules.

Тут мы видим, что название job начинается с точки: .test. Это так называемый [hidden-job,](#) он не будет создаваться и запускаться в пайплайне. Но его описание такой hidden-job можно переиспользовать для создания других jobs на его основе:

```yaml
canary 1%:
  extends: .canary
  variables:
    CANARY_WEIGHT: "1"
```

Переменные

В GitLab легко настраивать jobs и даже пайплайны целых сервисов, используя переменные. Они могут быть объявлены как в самом job (см. выше), так и на уровне файла .yml.

При выполнении они становятся переменными окружения.

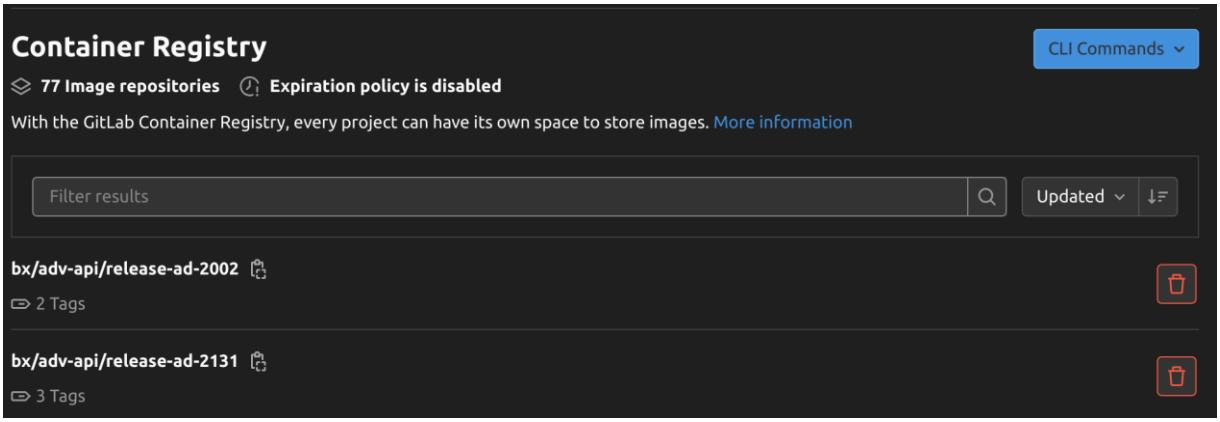
Переменные можно объявить и в настройках репозитория.

GitLab также инициализирует и использует свои переменные, доступны они и нашим скриптам.

Очень полезна для отладки переменная CI_DEBUG_TRACE, она выводит в лог выполнения и переменные окружения и те команды, которые сейчас выполняются. Вместе с CI Lint и командой set -x в скриптах отлаживать и настраивать новые jobs не так сложно, как может показаться.

Registry

Кроме артефактов выполнения, собираются образы микросервисов. Они хранятся в GitLab registry, их можно посмотреть, зайдя на соответствующую вкладку в репозитории проекта:



Когда запускать: триггеры

Описаний в файлах yml достаточно, чтобы декларативно описать, что запускать. Теперь разберемся, что может триггерить запуск.

Первый и самый частый вариант запуска — git push нового кода, это следует из определения CI.

Часто пайплайн нужно запустить из UI GitLab. Например, нужно обновить описание job в yml. Обратите внимание: что пайплайн целиком рендерится в момент триггера, изменения в yml не подхватятся, если перезапустить одну из job уже созданного пайплайна.

В тестировании мы используем расписание, но регулярные процедуры в продакшн используют другой механизм — Kubernetes Cronjob. Запускаемые по расписанию тесты проверяют интеграционные сценарии на сайте или находят проблемы на stage до запуска тестов в пайплайне.

Последний, четвертый вариант запуска — GitLab API. Например, в ходе выполнения пайплайна скрипт понял, что мастер обновился. Тогда он создает новый пайплайн и отменяет задачи старого, используя API.

Заключение

Мы узнали, чем отличается CI от CD и какие проблемы решает каждый из них. GitLab предлагает зрелое функциональное решение, которое активно используется в Ozon. На занятии мы познакомились с основными терминами GitLab CI и их использованием в синтаксисе .yml. В следующий раз рассмотрим, как применять эти знания, а именно как встроить линтеры для наших тестов в пайплайн. Кроме того, узнаем больше о практике CD в нашей компании.