otus

Использование services, rules, artifacts, cache

CI/CD на основе GitLab

Проверить, идет ли запись

Меня хорошо видно & слышно?

Николай Осипов

MLOps Engineer

- 5+ лет в области **Data Science**
- **MLOps**, DS инфраструктура, табличные данные
- Выступаю наставником и преподавателем на курсах по **ML**
- Преподаватель курса **CI/CD на основе GitLab**

@NickOsipov

github.com/NickOsipov

Правила вебинара

Активно
участвуем

Off-topic обсуждаем
в учебной группе #**OTUS CICD-GitLab-2025-09**

Задаем вопрос
в чат

Вопросы вижу в чате,
могу ответить не сразу

Карта курса

СТАРТ

CI/CD - системы, подходы и workflow

Gitlab CI

Безопасность

Проектная работа

ФИНИШ

Маршрут вебинара

- Знакомство
- Services
- Artifacts и Cache
- Rules
- Needs
- Практика

Цели вебинара

К концу занятия вы сможете

1. Научиться разбираться в базовой функциональности GitLab CI, включая пайплайны и их основные компоненты.

2. Узнать, как эффективно использовать такие функции, как services, artifacts, cache, и rules для оптимизации пайплайнов.

3. Научиться использовать needs для настройки сложных пайплайнов с зависимостями и параллельным выполнением задач.

Смысл

Это нужно уметь чтобы:

1. Понимать пайплайны и их компоненты для автоматизации разработки

2. Делать пайплайны быстрыми

3. Уметь работать с зависимостями и параллельным выполнением

Кто-то сейчас уже использует функции services, artifacts, cache, rules, needs?

Services

Services – это

контейнеры, которые запускаются параллельно с основной задачей (job) в GitLab CI.

Services

Зачем нужны Services?

- Тестировать взаимодействие приложения с базой данных.
- Подключать кэш-системы для тестов производительности.
- Проверять интеграцию с внешними сервисами (например, REST API).

Как работают Services?

Ограничения:

- Performance.
- Network.

Services

Конфигурация Services

```
job:
  image: alpine:3.12
  services: # помимо основного контейнера, будут запущены 3 дополнительных контейнера
    - php:7
    - node:latest
    - golang:1.10
  script:
    - apk add --no-cache php7 nodejs go# устанавливает PHP, Node.js и Go в основном контейнере
    - php -v # проверяем версию PHP
    - node -v # проверяем версию Node.js
    - go version # проверяем версию Go
```

Services

Пример 2.

Предположим, что у вас есть приложение, которое использует базу данных PostgreSQL.

```
test-job:
  image: python:3.9 # используется Docker-образ с Python для выполнения команды.
  services: # запускается контейнер с PostgreSQL версии 13 с псевдонимом db.
    - name: postgres:13
      alias: db
  variables: # определяются переменные окружения для подключения к базе данных (POSTGRES_DB, 
POSTGRES_USER, POSTGRES_PASSWORD, POSTGRES_HOST).
    POSTGRES_DB: test_db
    POSTGRES_USER: user
    POSTGRES_PASSWORD: 12345
    POSTGRES_HOST: db
  before_script: # устанавливается PostgreSQL клиент для взаимодействия с базой данных.
    - apt-get update -y && apt-get install -y postgresql-client
  script: # выполняется команда для создания таблицы в базе данных через psql
    - export PGPASSWORD=$POSTGRES_PASSWORD# Устанавливаем переменную окружения для пароля
    - psql -h db -U $POSTGRES_USER -d $POSTGRES_DB -c "CREATE TABLE test (id SERIAL PRIMARY 
KEY);"
```

Services

Настройка нескольких сервисов
GitLab CI позволяет использовать несколько сервисов в одной джобе. Например, можно подключить как базу данных, так и Redis

```
job_with_multiple_services: # джоба взаимодействует как с db, так и с Redis.
  image: ruby:3.1 # используем образ с необходимыми зависимостями
  services:
    - name: postgres:13
      alias: db
    - name: redis:latest
      alias: cache
  variables:
    POSTGRES_DB: test_db
    POSTGRES_USER: user
    POSTGRES_PASSWORD: 12345 # задаем пароль, если он требуется
  before_script:
    - apt-get update -y && apt-get install -y postgresql-client redis-tools#устанавливаем клиента
  script:
    - export PGPASSWORD=$POSTGRES_PASSWORD# устанавливаем переменную окружения для пароля
    - psql -h db -U $POSTGRES_USER -d $POSTGRES_DB -c "SELECT 1"# проверка соединения с PostgreSQL
    - redis-cli -h cache ping # проверка соединения с Redis
```

Services

Дополнительные примеры:

```
services:
  - name: postgres:13
    alias: db
    command: ["postgres", "-N", "200"] # Увеличение числа подключений. Определяет команду, с
которой запускается сервис. Это полезно для настройки сервисов, если стандартная команда
Docker-образа не подходит.


services:
  - name: custom-db-image
    entrypoint: ["/custom-entrypoint.sh"] # позволяет переопределить стандартную точку входа
(entrypoint) контейнера.
```

Вопросы

если есть вопросы

если вопросов нет

Artifacts и Cache

Artifacts – это

файлы, которые создаются в одной джобе и сохраняются для последующего использования в других джобах или для скачивания после завершения пайплайна.

Artifacts

Как работают Artifacts?

- В последующих джобах того же пайплайна.
- Для анализа результатов после завершения пайплайна.
- Для передачи артефактов в другие стадии (stages) пайплайна, как промежуточные результаты.

Artifacts

Пример конфигурации Artifacts.
Makefile определяет целр build, которая компилирует файл source.c и сохраняет результат в папке binaries/.

```
build:
  stage: build
  script:
    - make build # с учетом того, что в корневой директории вашего проекта есть файл
Makefile, и что в нем определена цель build
  artifacts: # секция для настройки артефактов
    paths: # указывает, какие файлы или директории нужно сохранить (в данном случае
директория binaries/)
      - binaries/
    expire_in: 1 week # задаёт время хранения артефактов на сервере GitLab. В данном примере
артефакты будут храниться неделю. По умолчанию артефакты хранятся неограниченное время.
```

Artifacts

Использование артефактов в следующих джобах.
Джобы, которые идут после джобы с артефактами, могут использовать эти артефакты.
Например, джоба, отвечающая за тестирование, может использовать скомпилированные бинарные файлы из предыдущей джобы.

```
test:
  stage: test
  script:
    - ./binaries/my_app --test
  dependencies: # указывает, что эта джоба зависит от результатов джобы build.
    - build # Артефакты из джобы build будут доступны в текущей джобе.
```

Artifacts

Пример: сохраняет содержимое директории logs/ как артефакты

```
artifacts:
  paths:
    - logs/
      when: always # Всегда сохраняем логи
# when: — определяет, при каком результате джобы сохраняются артефакты.
# when Может принимать значения:
  # on_success (по умолчанию) — артефакты сохраняются, если джоба завершилась успешно.
  # on_failure — артефакты сохраняются только при провале джобы.
  # always — артефакты сохраняются всегда.


artifacts:
  reports: # используется для создания отчётов, таких как результаты тестов, покрытия кода.
    junit: report.xml # Результаты тестов в формате JUnit
```

Cache – это

механизм кэширования файлов между джобами или разными пайплайнами для ускорения их выполнения.

Cache

Как работает Cache?

Кэш помогает избежать повторной загрузки одинаковых данных в каждой джобе или пайплайне.

Например, если в проекте есть большой набор зависимостей, вы можете сохранить их в кэше, чтобы не скачивать их заново при каждом запуске пайплайна.

Cache

Пример использования Cache.

```yaml
build:
  stage: build
  script:
    - npm install # наличие файла package.json с необходимыми зависимостями и скриптами
    - npm run build
  cache: # секция для настройки кэша.
    paths: # указывает, какие файлы или директории нужно закэшировать
      - node_modules/
```

В следующий раз, когда джоба будет выполняться, GitLab попытается восстановить кэшированные файлы из предыдущих джоб.

Cache

Разделение кэша по ключам (Keys)
Можно использовать разные кэши для различных задач, определив ключ кэша:

```
cache:
  key: "$CI_COMMIT_REF_SLUG" # создание отдельного кэша для каждой ветки. Задаёт уникальный ключ для кэша. В примере выше кэш будет уникален для каждой ветки благодаря переменной $CI_COMMIT_REF_SLUG.
  paths:
    - dependencies/
```

Пример, где кэширует все файлы, не отслеживаемые Git.

```
cache:
  untracked: true
```

Cache

Работа с policy

```
cache:
  paths:
    - node_modules/
  policy: pull # использовать кэш, но не обновлять

# policy: — определяет политику использования кэша. Может принимать значения:
  # pull-push (по умолчанию) — кэш используется и обновляется после выполнения джобы.
  # pull — кэш только используется, но не обновляется.
  # push — кэш только обновляется, но не используется.
```

Artifacts и Cache

Допустим, вы разрабатываете приложение на Python с использованием виртуального окружения и хотите закэшировать зависимости, но сохранить артефакты сборки.

```
stages:
  - install
  - build
  - test

install_dependencies: # создаем виртуальное окружение Python, и оно кэшируется для последующего использования.
  image: python:3.11
  stage: install
  script:
    - python -m venv venv
    - source venv/bin/activate
    - pip install -r requirements.txt # при наличии файла в проекте
  cache:
  paths:
    - venv/

build: # происходит сборка приложения, артефакты сборки сохраняются для других джоб в папке dist/.
  stage: build
  script:
    - source venv/bin/activate
    - python setup.py sdist bdist_wheel # Добавлено для создания дистрибутива
  artifacts:
    paths:
      - dist/
    expire_in: 1 week

test: # запускаются тесты, используя артефакты сборки, а результаты тестов сохраняются в формате JUnit для дальнейшего анализа.
  stage: test
  script:
    - source venv/bin/activate
    - pytest
  dependencies:
    - build
  artifacts:
    reports:
      junit: report.xml
```

Cache vs Artifacts

Cache

- Ускоряет сборку между запусками
- Временное хранение (node_modules, .gradle, pip cache)
- Не гарантирует сохранность данных
- Автоматически очищается при нехватке места
- Используется для оптимизации производительности

Artifacts

- Передает результаты между джобами и стейджами
- Хранит конечные продукты сборки (бинарники, отчеты, логи)
- Гарантированное сохранение на заданный период
- Доступны для скачивания из веб-интерфейса
- Используется для передачи данных в пайплайне

Ключевое различие: Cache для скорости, Artifacts для сохранения результатов

Вопросы

если есть вопросы

если вопросов нет

Rules

Rules – это

инструмент в GitLab CI, который позволяет контролировать выполнение джобов (jobs) в пайплайне на основе различных условий.

Rules

Зачем нужны Rules?

С помощью rules можно:

- Выполнять джобы только на определённых ветках или тегах.
- Запускать джобы в зависимости от изменения файлов.
- Контролировать выполнение на основе переменных окружения или статуса других джоб.

Как работают Rules?

Rules

Пример использования rules.

```
job_name:
  script:
    - echo "This is a conditional job"
  rules: # секция для указания правил.
    - if: '$CI_COMMIT_BRANCH == "main"'# условие, на основе которого принимается решение,
   выполняется джоба или нет. В первом правиле указано, что джоба выполняется всегда, если
   текущая ветка main.
      when: always
    - if: '$CI_COMMIT_TAG'
      when: manual


# when: — указывает, как джоба должна быть выполнена:
# always — джоба всегда выполняется, если условие истинно.
# manual — джоба может быть выполнена вручную, если условие истинно.
# never — джоба не будет выполнена при любом условии.

# $CI_COMMIT_TAG: Это встроенная переменная окружения в GitLab CI, которая содержит имя тега,
   если текущий коммит связан с тегом. Если текущий коммит не имеет тега, эта переменная будет
   пустой. Если вы создадите тег (например, v1.0.0), эта джоба запустится и выведет сообщение
   "Creating a release". Если тег не будет создан, джоба release не будет выполнена.
```

Частые сценарии использования rules

1. Запуск джобы только в определённой ветке

Вы можете настроить выполнение джобы только для ветки main или другой специфической ветки.

```
deploy:
  script:
    - echo "Deploying to production"
  rules:
    - if: '$CI_COMMIT_BRANCH == "main"'
```

Частые сценарии использования rules

2. Выполнение джобы для определённого файл

Можно настроить правила так, чтобы джоба выполнялась, если был изменён определённый файл или папка:

```
test:
  script:
    - echo "Running tests"
  rules:
    - changes:
    - src/* # джоба будет запущена, если изменились файлы в папке src/.
```

Частые сценарии использования rules

3. Запуск на основе тегов.

Иногда нужно запускать джобы только при создании тега (например, для релизов).
Это можно сделать с помощью правила

```
release:
  script:
    - echo "Creating a release"
  rules:
    - if: '$CI_COMMIT_TAG’ # джоба запускается только при наличии тега
```

Частые сценарии использования rules

4. Комбинированные правила с использованием логических операторов

Вы можете комбинировать условия с помощью логических операторов, таких как && (и) и || (или):

```
complex_rule_job:
  script:
    - echo "Running complex job"
  rules:
    - if: '$CI_COMMIT_BRANCH == "main" && $CI_PIPELINE_SOURCE == "push"' # джоба выполнится, только если текущая ветка main и пайплайн был запущен в результате push-коммита
```

Частые сценарии использования rules

5. Управление джобами вручную.
Иногда может потребоваться запускать джобы вручную. Это можно сделать с помощью директивы when: manual.

```
manual_test:
  script:
    - echo "This job is executed manually"
  rules:
    - if: '$CI_COMMIT_BRANCH == "main"'# проверяет условие на основе переменных GitLab CI (например, ветка, тег, пользовательские переменные). Условие может быть любой логической проверкой.
      when: manual # Джоба будет доступна для ручного запуска в интерфейсе GitLab, если выполняется условие ветки main.
```

Rules

```
rules:
  - changes: # проверяет, были ли изменены определённые файлы. Это полезно для выполнения
тестов или сборки только при изменениях в определённых частях проекта.
    - Dockerfile

rules:
  - exists: # проверяет существование файлов перед выполнением джобы.
    - .env

rules:
  - if: '$CI_COMMIT_BRANCH == "main"'
    when: manual

# when: — управляет тем, когда должна выполняться джоба:
# on_success — джоба выполняется только при успешном завершении предыдущих джоб (по умолчанию).
# manual — джоба доступна для ручного запуска.
# always — джоба всегда выполняется.
# never — джоба никогда не выполняется.
```

Различие между Rules и Only/Except.

Only и Except - это старые директивы, которые использовались для контроля выполнения джобов на основе условий.

```
# Пример с использованием only:

job:
  script:
    - echo "Running job"
  only:
    - main



# Пример с использованием rules для того же сценария:

job:
  script:
    - echo "Running job"
  rules:
    - if: '$CI_COMMIT_BRANCH == "main"'
```

Пример сложного пайплайна с Rules

Допустим, у вас есть проект с несколькими ветками, и вы хотите настроить пайплайн, который выполняет разные джобы в зависимости от того, какая ветка используется и какие файлы были изменены.

```
stages:
  - build
  - test
  - deploy

build_job: # выполняется, если текущая ветка main или если были изменения в папке src/.
  stage: build
  script:
    - echo "Building project"
  rules:
    - if: '$CI_COMMIT_BRANCH == "main"'
      changes:
        - src/*

test_job: # выполняется для веток main и develop.
  stage: test
  script:
    - echo "Running tests"
  rules:
    - if: '$CI_COMMIT_BRANCH == "main"'
    - if: '$CI_COMMIT_BRANCH == "develop"'

deploy_job: # можно запустить вручную, но только для ветки main.
  stage: deploy
  script:
    - echo "Deploying project"
  rules:
    - if: '$CI_COMMIT_BRANCH == "main"'
      when: manual
```

Вопросы

если есть вопросы

если вопросов нет

Needs

Needs

Зачем нужны Needs?
В GitLab CI, когда вы определяете stages, все джобы внутри одной стадии выполняются параллельно, а следующая стадия начинается только тогда, когда все джобы предыдущей стадии завершены.

Как работает Needs?
Needs указывает, что конкретная джоба зависит от выполнения другой джобы.

Основные особенности Needs:

- Оптимизация параллельного выполнения.
- Явное указание зависимостей.
- Ускорение пайплайнов.

Needs

В стандартной настройке пайплайна test_job началась бы только после завершения всех джоб стадии build. Однако, используя needs, вы указываете, что test_job может начаться сразу после завершения build_job, игнорируя другие джобы на стадии build.
Аналогично, deploy_job зависит от test_job, и поэтому начнётся после её завершения.

```
stages:
  - build
  - test
  - deploy

build_job:
  stage: build
  script:
    - echo "Building the project"

test_job:
  stage: test
  script:
    - echo "Running tests"
  needs:
    - job: build_job

deploy_job:
  stage: deploy
  script:
    - echo "Deploying the project"
  needs:
    - job: test_job
```

Needs

Оптимизация с помощью Needs. В этом примере test_job_1 и test_job_2 начнутся после завершения build_job, и не нужно ждать завершения других джоб на стадии build. А джоба deploy_job выполнится только после завершения обоих тестов.

```
stages:
  - build
  - test
  - deploy

build_job:
  stage: build
  script:
    - echo "Building project..."

test_job_1:
  stage: test
  script:
    - echo "Running tests 1"
  needs:
    - job: build_job

test_job_2:
  stage: test
  script:
    - echo "Running tests 2"
  needs:
    - job: build_job

deploy_job:
  stage: deploy
  script:
    - echo "Deploying project..."
  needs:
    - job: test_job_1
    - job: test_job_2
```

Update .gitlab-ci.yml

Running **Kirill Senchik** created pipeline for commit `00143a88` just now

For `main`

latest 4 jobs In progress, queued for 1 seconds

Pipeline Jobs 4 Tests 0

Group jobs by Stage Job dependencies

Needs

```
stages:
  - build
  - test
  - deploy

build:
  stage: build
  script:
    - echo "Building the project..."

unit_tests:  # после завершения стадии build, обе джобы unit_tests и integration_tests начнут выполняться параллельно.
  stage: test
  script:
    - echo "Running unit tests"
  needs:
    - job: build

integration_tests:
  stage: test
  script:
    - echo "Running integration tests"
  needs:
    - job: build

deploy_staging:  # deploy_staging начнётся только после успешного выполнения всех тестов.
  stage: deploy
  script:
    - echo "Deploying to staging"
  needs:
    - job: unit_tests
    - job: integration_tests
```

Практика

Ключевые тезисы

1. **Эффективность CI/CD:** GitLab CI позволяет автоматизировать процессы сборки, тестирования и развертывания приложений, что значительно ускоряет разработку и повышает качество программного обеспечения.
2. **Использование внешних сервисов:** Параллельный запуск сервисов в контейнерах (например, базы данных) упрощает настройку тестового окружения, позволяя быстро и удобно тестировать интеграцию приложения с внешними компонентами.
3. **Оптимизация пайплайнов:** Применение артефактов, кэширования и динамических правил позволяет улучшить производительность пайплайнов, минимизируя время выполнения и эффективно управляя зависимостями между задачами.

Вопросы

если есть вопросы

если вопросов нет

Домашнее задание

Напишите gitlab-ci к проекту. Ссылка на проект:
https://github.com/jellyfin/jellyfin-web

Рекомендации: Чтобы добиться большей скорости и меньшего веса конечного docker image не используя в Dockerfile мультистейдж сборку (используйте cache и artifacts) image сложите в registry gitlab.

Подведём итоги занятия

Теперь вы сможете:

Отправьте в чат номера достигнутых целей

1. Научиться разбираться в базовой функциональности GitLab CI, включая пайплайны и их основные компоненты.

2. Узнать, как эффективно использовать такие функции, как services, artifacts, cache, и rules для оптимизации пайплайнов.

3. Научиться использовать компоненты и needs для настройки сложных пайплайнов с зависимостями и параллельным выполнением задач.

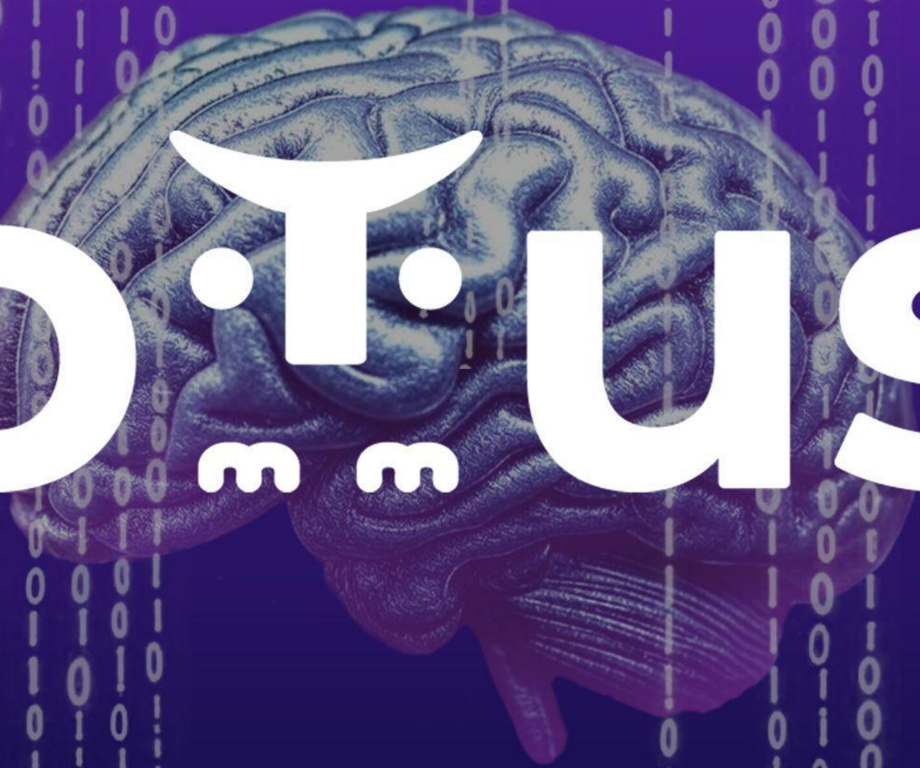
Отправьте в чат эмодзи, который отражает ваше настроение после занятия

Продолжите высказывание: теперь я умею/знаю...

Что планируете использовать в работе?

Заполните, пожалуйста, опрос о занятии

Мы читаем все ваши сообщения
и берем их в работу ✏️❤️


OTUS

ТЫ ЕЩЕ НЕ В КЛУБЕ?

ПРИСОЕДИНЯЙСЯ И НАЧИНАЙ ЭКОНОМИТЬ!

ЭТОТ КОНТЕНТ КУПЛЕН НА САЙТЕ SKLADCHIK.ORG

ЧТО ТАКОЕ КЛУБ «СКЛАДЧИК»?

Платформа

Это платформа, где каждый день тысячи людей собираются вместе, чтобы общими усилиями находить, приобретать и изучать курсы интересные именно им.

Сообщество

Это сообщество из 500 000 людей, открывших для себя выгодный способ быть в тренде самых актуальных и получать ценные знания по минимальной цене.

Библиотека

Это крупнейшая библиотека инфопродуктов в которой можно найти практически любой курс или тренинг продававшийся за последние 10 лет, а также уникальные авторские инфопродукты, которые не получиться найти больше нигде.