

CICD

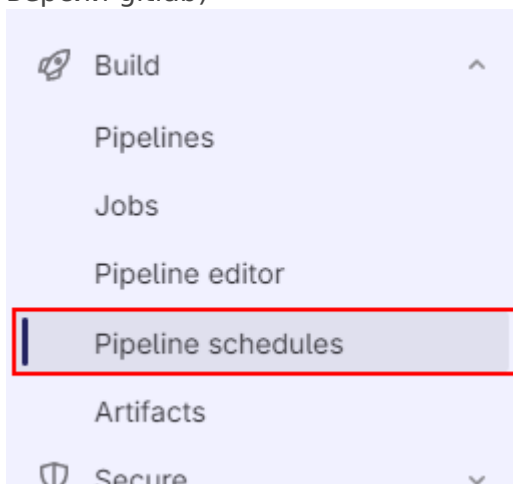
Информация по непрерывной интеграции (continuous integration) и непрерывному развертыванию (continuous delivery или continuous deployment) программного обеспечения в процессе разработки

- [Как реализовать запуск задач в .gitlab-ci.yml с различной периодичностью выполнения](#)
- [Основные шаблоны для работы с файлами и каталогами GitLab CI/CD](#)
- [Рекомендуемый порядок расположения блоков внутри job в .gitlab-ci.yml?](#)

Как реализовать запуск задач в .gitlab-ci.yml с различной периодичностью выполнения

1. Создать 2 планировщика (sheduler)

1. Меню: **Build - Pipeline Schedules** или **CI/CD - Schedules** (в зависимости от версии gitlab)



2. Нажать на кнопку "New schedule"



3. Указать в разделе переменных переменную для одного расписания

Edit Pipeline Schedule

Description

Every 60 min

Interval Pattern

- ☐ Every day (at 2:00am)
- ☐ Every week (Thursday at 2:00am)
- ☐ Every month (Day 5 at 2:00am)
- ☒ Custom ([Learn more.](#))

* / 60 9-07 * * 1-5

Cron Timezone

[UTC+3] Moscow

Target branch or tag

main

Variables

Variable

interval

long

X

Variable

Input variable key

Input variable

value

Hide value

Activated

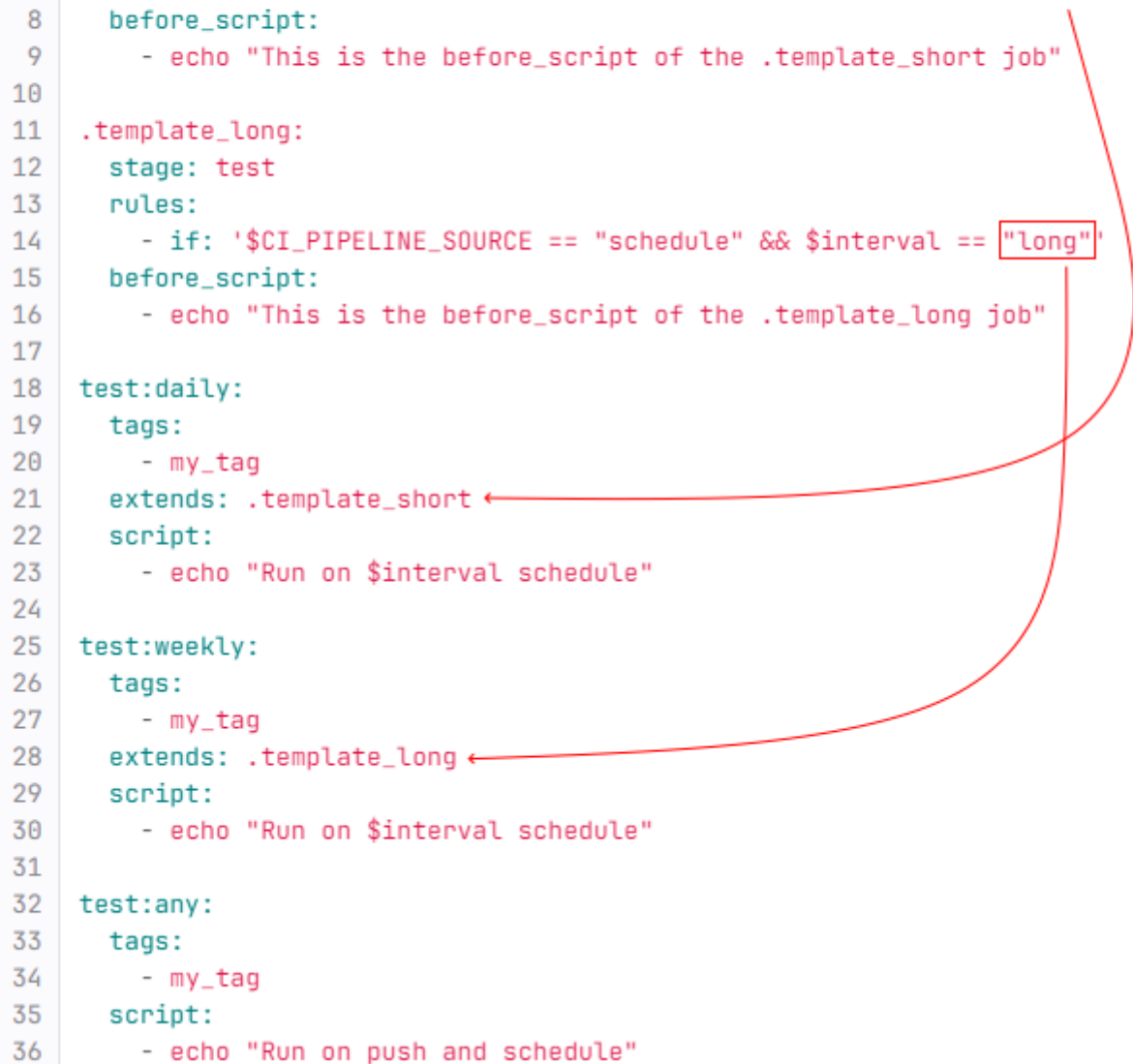
☒ Active

Save pipeline schedule

Cancel

4. Аналогично добавить второй планировщик и указать для него свою переменную
2. В файле .gitlab-ci.yml добавить правила выполнения для каждой переменной

```
1 stages:
2   - test
3
4 .template_short:
5   stage: test
6   rules:
7     - if: '$CI_PIPELINE_SOURCE == "schedule" && $interval == "short"'
8   before_script:
9     - echo "This is the before_script of the .template_short job"
10
11 .template_long:
12   stage: test
13   rules:
14     - if: '$CI_PIPELINE_SOURCE == "schedule" && $interval == "long"'
15   before_script:
16     - echo "This is the before_script of the .template_long job"
17
18 test:daily:
19   tags:
20     - my_tag
21   extends: .template_short
22   script:
23     - echo "Run on $interval schedule"
24
25 test:weekly:
26   tags:
27     - my_tag
28   extends: .template_long
29   script:
30     - echo "Run on $interval schedule"
31
32 test:any:
33   tags:
34     - my_tag
35   script:
36     - echo "Run on push and schedule"
```



3. Пример файла .gitlab-ci.yml

```
stages:
  - test

.template_short:
  stage: test
  rules:
```

```
- if: '$CI_PIPELINE_SOURCE == "schedule" && $interval == "short"'
```

```
before_script:
```

```
- echo "This is the before_script of the .template_short job"
```

```
.template_long:
```

```
stage: test
```

```
rules:
```

```
- if: '$CI_PIPELINE_SOURCE == "schedule" && $interval == "long"'
```

```
before_script:
```

```
- echo "This is the before_script of the .template_long job"
```

```
test:daily:
```

```
tags:
```

```
- my_tag
```

```
extends: .template_short
```

```
script:
```

```
- echo "Run on $interval schedule"
```

```
test:weekly:
```

```
tags:
```

```
- my_tag
```

```
extends: .template_long
```

```
script:
```

```
- echo "Run on $interval schedule"
```

```
test:any:
```

```
tags:
```

```
- my_tag
```

```
script:
```

```
- echo "Run on push and schedule"
```

Основные шаблоны для работы с файлами и каталогами GitLab CI/CD

GitLab CI/CD позволяет отслеживать изменения файлов и каталогов с помощью **glob patterns**. Они используются в `changes`, `artifacts`, `cache` и других разделах `.gitlab-ci.yml`. Эта статья разберет основные шаблоны и их поведение.

Основные glob patterns

Шаблон	Описание
<code>path/*</code>	Отслеживает только файлы и каталоги первого уровня в <code>path</code> , без вложенных файлов .
<code>path/**/*</code>	Отслеживает все файлы и папки внутри <code>path</code> , включая вложенные файлы на всех уровнях .
<code>path/*/*</code>	Отслеживает файлы и папки только второго уровня внутри <code>path</code> .
<code>path/*/**</code>	Отслеживает файлы на первом уровне + все вложенные файлы во втором уровне и глубже.
<code>path/**</code>	Аналог <code>path/**/*</code> , отслеживает всё, включая подпапки и файлы.
<code>path/**/file.txt</code>	Отслеживает конкретный файл , независимо от его глубины.

Вывод

- Используйте `/**/*`, если хотите **отслеживать все файлы и папки**.
- Используйте `/*/*`, если хотите **только второй уровень вложенности**.
- Используйте `path/**/file.txt`, если хотите **отслеживать конкретные файлы на всех уровнях**.

Рекомендуемый порядок расположения блоков внутри job в .gitlab-ci.yml?

В GitLab CI/CD **есть рекомендованный порядок**, который улучшает **читаемость и поддержку pipeline**.

□□ Стандартный порядок блоков внутри job

```
job_name:
  extends: .some_template # 1□ Наследование (если есть)
  stage: build            # 2□ Определение этапа
  tags:                   # 3□ Теги для раннера (если нужны)
    - docker
  needs:                  # 4□ Зависимости от других job'ов
    - previous_job
  variables:              # 5□ Локальные переменные для job
    SOME_VAR: "value"
  before_script:          # 6□ Подготовка окружения перед `script`
    - echo "Setting up environment..."
  script:                 # 7□ Основной код
    - echo "Executing job..."
  after_script:           # 8□ Завершающие действия (если нужны)
    - echo "Cleanup after job"
  artifacts:              # 9□ Артефакты (если нужны)
    paths:
      - my_output/
  expire_in: 1 hour
```



```
rules:           # [] Условия выполнения job
- if: '$CI_COMMIT_BRANCH == "main"'

allow_failure: false  # 1[] Позволять ли падение job

retry: 2           # 1[] Число повторных попыток

timeout: 30m       # 1[] Лимит времени выполнения
```

Разбор структуры

Секция	Описание
<code>extends:</code>	Наследование от шаблонного <code>job</code> (если есть).
<code>stage:</code>	Назначение <code>job</code> к определенному этапу.
<code>tags:</code>	Указывает, на каких раннерах будет выполняться <code>job</code> .
<code>needs:</code>	Указывает зависимости от других <code>job</code> (если надо запустить <code>job</code> только после выполнения других).
<code>variables:</code>	Определяет переменные, видимые только внутри <code>job</code> .
<code>before_script:</code>	Выполняется один раз перед <code>script:</code> (установка окружения, логирование, авторизация и т. д.).
<code>script:</code>	Основной код <code>job</code> , выполняется после <code>before_script:</code> .
<code>after_script:</code>	Выполняется после <code>script:</code> , даже если <code>script:</code> упал. Подходит для логов, очистки файлов.
<code>artifacts:</code>	Определяет файлы, которые сохраняются после завершения <code>job</code> .
<code>rules:</code>	Условия выполнения <code>job</code> (например, только в <code>main</code> , только при изменениях в <code>src/</code>).
<code>allow_failure:</code>	Разрешает <code>job</code> завершаться с ошибкой (<code>true</code> → ошибки не остановят pipeline).
<code>retry:</code>	Число повторных попыток при падении <code>job</code> .
<code>timeout:</code>	Лимит времени выполнения <code>job</code> (например, <code>30m</code>).

Пример 1: Минимальная конфигурация `job`

Если у вас простой `job`, можно оставить только **ключевые секции**:

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

Минимально необходимый `job`: `stage:` + `script:`.

Пример 2: Расширенная конфигурация `job`

Если `job` сложный, используем **все ключевые блоки**:

```
deploy_job:
  extends: .deploy_template
  stage: deploy
  tags:
    - production-runner
  needs:
    - build_job
  variables:
    DEPLOY_ENV: "production"
  before_script:
    - echo "Подготовка к деплою..."
  script:
    - echo "Выполняем деплой..."
  after_script:
    - echo "Очистка после деплоя..."
  artifacts:
    paths:
      - deploy_logs/
    expire_in: 1 day
  rules:
    - if: '$CI_COMMIT_BRANCH == "main"'
  allow_failure: false
```

retry: 2

timeout: 20m

□ Этот `job` должен выполняться ТОЛЬКО в `main`.

□ Если `job` упадет, он попытается перезапуститься дважды (`retry: 2`).

□ `before_script:` подготавливает окружение, а `after_script:` выполняет очистку.

□□ Итог

□□ Оптимальный порядок блоков `job:`

- 1□ `extends:` (если используется)
- 2□ `stage:`
- 3□ `tags:` (если есть)
- 4□ `needs:` (если `job` зависит от других `job`'ов)
- 5□ `variables:` (локальные переменные)
- 6□ `before_script:` (подготовка окружения)
- 7□ `script:` (основная логика `job`)
- 8□ `after_script:` (очистка, логи)
- 9□ `artifacts:` (если нужны)
- `rules:` (условия выполнения)
- 1□1 `allow_failure:` (можно ли игнорировать ошибки)
- 1□2 `retry:` (количество повторных попыток)
- 1□3 `timeout:` (ограничение времени выполнения)