

Альтернативные способы CI/CD

GitHub Actions и GitLab CI/CD




Работа с GitHub Actions







- Мы уже разбирали программу Jenkins - систему автоматизации тестирования и развертывания приложений, которая является отличным DevOps инструментом. Однако ее нельзя назвать единственным средством для решения проблем - существует технологии, которые могут предоставить схожий функционал. Одной из таких технологий является платформа GitHub Actions, которая встроена в популярный сайт GitHub, используемый для хранения Git репозиториях.
- GitHub Actions позволяет настроить CI/CD процессы внутри самого репозитория - мы просто добавляем в репозиторий описание того, что надо сделать с нашим кодом при определенном событии (например, при загрузке данных - **push** - в определенную ветку репозитория), и уже сам GitHub, а не внешняя система, обнаружит эти инструкции и попытается выполнить их. Безусловно, это очень удобно, но надо помнить о материальной стороне вопроса - GitHub Actions бесплатен только для использования в публичных репозиториях или в том случае, если мы используем внешние компьютеры для исполнении задач (такая опция существует). Если же у нас закрытый репозиторий, то нам бесплатно доступны только те задачи, общее время работы которых не превышает 2000 минут. Более подробную информацию можно узнать по следующей ссылке: <https://docs.github.com/en/billing/managing-billing-for-github-actions/about-billing-for-github-actions>









Начало работы с GitHub Actions

- Для того, чтобы написать инструкцию GitHub Actions, внутри нашего репозитория нам необходимо создать директорию **.github**, а уже внутри нее еще одну директорию **workflows** (т.е. полный путь должен быть таким - **.github/workflows**).
- Внутри созданной директории **.github/workflows** мы должны создавать файлы в **yml** формате (название файла не имеет никакого значения - обязателен только формат **yml**), где и будут описываться те действия, которые мы хотим осуществить в рамках нашего **CI/CD** процесса.
- Также можно создавать эти файлы на самом сайте GitHub. Для этого нам следует, перейти в нужный нам репозиторий и нажать пункт меню **Actions**. Затем нужно заполнить этой файл самостоятельно после нажатии кнопки **"set up a workflow yourself"** или выбрать какой-либо предложенный вариант под ваш конкретный случай (будут приведены множество уже готовых сценариев тестирования и развертывания под разные технологии и платформы - Java, Nodejs, Python, Docker и многое другое). Какой бы вариант мы не выбрали, затем заполняем открывшийся файл инструкциями (о них мы поговорим уже очень скоро) и нажимает вверху справа зеленую кнопку **"Commit changes ..."**.

Начало работы с GitHub Actions (возможности создания CI/CD инструкции)

 itat-mysql / blog 

Q Type  to search     

 Code  Issues  Pull requests  Actions  Projects  Security  Insights  Settings

Get started with GitHub Actions


Build, test, and deploy your code. Make code reviews, branch management, and issue triaging work the way you want. Select a workflow to get started.

[Skip this and set up a workflow yourself](#) →

Q Search workflows


Suggested for this repository

Docker image
By GitHub Actions




Build a Docker image to deploy, run, or push to a registry.

Configure


Dockerfile 

Deno
By GitHub Actions




Test your Deno project

Configure


JavaScript 

Django
By GitHub Actions




Build and Test a Django Project

Configure


Python 

Jekyll using Docker image
By GitHub Actions




Package a Jekyll site using the jekyll/builder Docker image.

Configure


HTML 

Grunt
By GitHub Actions




Build a NodeJS project with npm and grunt.

Configure


JavaScript 

Gulp
By GitHub Actions



Build a NodeJS project with npm and gulp.

Configure

JavaScript 

Создание простейшего файла с CI/CD инструкциями (глобальные параметры)

- Как уже было сказано, наш файл с инструкциями должен быть написан в формате **yaml**. В самом начале нам необходимо указать свойство **name** и в качестве его значения подставить имя нашей инструкции.
- Затем нужно указать одно или несколько событий, после которых и будут запущены наши задачи. Это осуществляется при помощи свойства **on**. Полный список разрешенных значений для этого свойства можно найти здесь: <https://docs.github.com/en/actions/using-workflows/events-that-trigger-workflows>
Здесь же приведем самые полезные из возможных значений:
 - **workflow_dispatch** - задания надо будет запускать вручную
 - **push** - запуск произойдет после обновления репозитория, свойство можно также дополнить списком веток (**branches**), в которых должно произойти обновление, например: [**"main", "development"**]
 - **pull_request** - запуск произойдет после некоего действия, связанного с **pull request**. Также можно указывать ветки (**branches**) - [**"main", "development"**].
 - **pull_request_review** - запуск произойдет после оценки изменений в **pull request**. Поддерживается указание списка веток (**branches**) и вид события связанного с оценкой (**types**). Например, если оценка была выдана, для свойства **types** можно указать следующее значение-список: [**"submitted"**]

Создание простейшего файла с CI/CD инструкциями (задачи)

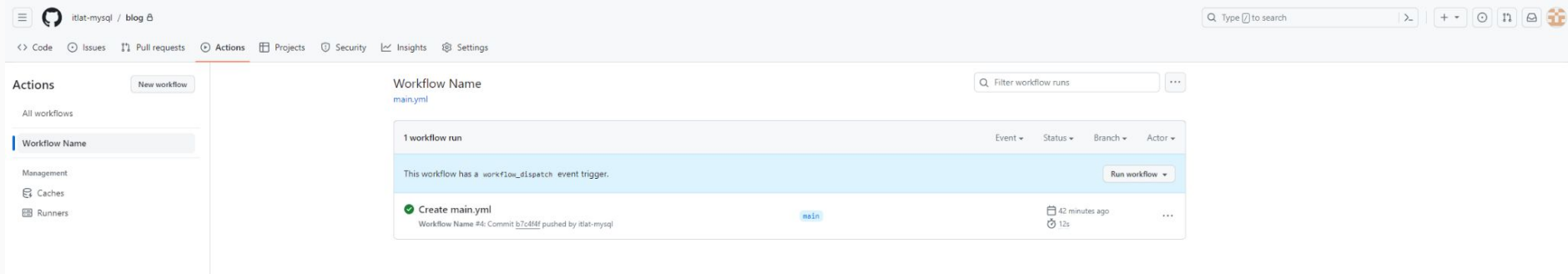
- Далее мы должны перейти к описанию задач. Это осуществляется при помощи свойства **jobs**. Задач может быть несколько - и у каждой из них должен быть уникальный идентификатор (уникальная строка). Внутри задачи обязательно должно быть свойство **runs-on**, внутри которого надо вписать операционную систему, которая будет использована при запуске шагов задачи, например, **ubuntu-22.04**. Полный список с названиями можно посмотреть здесь: <https://docs.github.com/en/actions/using-workflows/workflow-syntax-for-github-actions#choosing-github-hosted-runners>
- Внутри задачи следует описать каждый ее шаг (свойство **steps**), который может состоять из имени задачи (свойство **name**) в который надо вписать наименование шага, и, например, некую директиву командной строки (**touch file.txt, npm run prod, pip install -r requirements.txt**) - это осуществляется при помощи свойства **run**. Шагов может быть несколько и каждый может обладать собственной логикой.
- Важный момент - данные, которые были созданы или изменены, передаются только от шага к шагу в рамках одной задачи. По умолчанию между разными задачами данные не передаются!

Создание простейшего файла с CI/CD инструкциями (пример)

```
name: Workflow Name
on:
  workflow_dispatch:
  push:
    branches: [ "main", "development" ]
  pull_request_review:
    branches: [ "main", "development" ]
    types: [ "submitted" ]
jobs:
  work_with_file:
    runs-on: ubuntu-22.04
    steps:
      - name: Create File
        run: touch example.txt
      - name: Write into the File
        run: echo "SOME CONTENT" > example.txt
      - name: Read from the File
        run: cat example.txt
  work_with_directory:
    runs-on: ubuntu-20.04
    steps:
      - name: Create Directory
        run: mkdir some-dir
      - name: Show Directory Contents
        run: ls -la
      - name: Remove Directory
        run: rm -rf some-dir
```


Запуск файла с инструкциями

- Когда мы сделаем коммит и загрузим наши изменения на GitHub (с помощью команды **push** или с помощью интерфейса самого GitHub), наши GitHub Actions инструкции вступят в силу. Если в файле написано, что задачи должны выполняться после команды **push**, то запуск инструкций произойдет незамедлительно.
- Также можно опять перейти в раздел **Action** нашего репозитория. Когда у нас появляются GitHub Actions инструкции, то там несколько меняется интерфейс - в этом случае мы можем посмотреть результаты предыдущих выполнений наших инструкций, а также сами запустить наши задания. Для этого надо выбрать соответствующую задачу и нажать кнопку **Run workflow** вверху слева:



Использование **actions** (введение)

- Если мы продолжим создавать GitHub Actions файлы с командами, которые знаем на данный момент, то обнаружим, что нам предоставлены пустые операционные системы, которые следует настраивать и устанавливать на них необходимые для работы программы (единственным исключением является Docker, который обычно уже установлен). Более того, по умолчанию внутри систем отсутствуют файлы нашего репозитория. Чтобы решить эту проблему, следует использовать т.н. **actions** - наборы действий, которые посвящены определенной теме. Например, предустановка **python** и всех его инструментов, инсталляция **nodejs** и **npm**, а также так необходимая подгрузка файлов из нашего репозитория.
- Список **actions** доступен по ссылке <https://github.com/marketplace?type=actions> - лучше всего выбирать те, которые помечены синим значком "**Creator verified by GitHub**". Существуют как платные, так и бесплатные версии **actions**, однако все стандартные опции доступны без всякой платы. Также надо упомянуть, что каждый **action** имеет свои собственные настройки и особенности (хотя общие настройки есть у всех **actions**). Далее мы рассмотрим самые популярные версии таких наборов действий.

Использование **actions** (конкретные подвиды)

- Самый простой и необходимый **action** - подгрузка данных из репозитория в нашу рабочую директорию. Данное действие производится при помощи свойства **uses** (должно располагаться внутри какого-нибудь из шагов - **steps**), значением которого должна стать следующая конструкция - **actions/checkout@v4**. Этот **action** имеет множество настроек, которые можно посмотреть в документации - в качестве примера приведем настройку **ref**, с помощью которой можно указать ветку или конкретный коммит, загружаемый в рабочую директорию (без применения настройки загружается последний коммит ветки по умолчанию).
- Также к числу популярных **actions** можно причислить **actions/setup-python@v5**. Как можно понять из названия, данный набор действий связан с установкой и настройкой языка Python. Сюда входит установка самого языка Python и менеджера зависимостей PIP. **actions/setup-python@v5** включает в себе несколько настроек, самой полезной из которых является установка версии языка Python. Применить эту настройку можно с помощью свойства **python-version**, которому в качестве значения нужно подставить нужную нам версию языка программирования.

Использование **actions** (пример с конкретными подвидами)

```
name: Python Workflow
on:
  workflow_dispatch:
jobs:
  python_example_job:
    runs-on: ubuntu-22.04
    steps:
      - uses: actions/checkout@v4
      - uses: actions/setup-python@v5
        with:
          python-version: '3.12'
      - name: Enable Virtual Environment
        run: |
          python -m venv .venv
          source ./venv/bin/activate
      - name: Install Dependencies
        run: pip install -r requirements.txt
      - name: Check System State
        run: python manage.py check
```

Использование переменных окружения и CI конвейер

- Часто при работе с приложением нам может потребоваться использование глобальных переменных окружения. Самый удобный вариант разместить их в свойстве **env** на самом высоком уровне в рамках нашего файла GitHub Actions. Сами переменные объявляются в виде объектных свойств верховного свойства **env**, например, **VARIABLE_NAME: VARIABLE_VALUE**. В свою очередь обращаться к ним можно с помощью следующей конструкции: **\${{ env.VARIABLE_NAME }}**.
- Применим наши знания - как о переменных, так и обо всех предыдущих темах, чтобы организовать первый CI конвейер в рамках GitHub Actions. Как мы помним, в качестве примера используется Django приложение, которое должно быть развернуто в Docker контейнерах. Для его развертывания объявим переменные окружения, которые применим для создания файла **.env** (несколько надуманный пример, но для практики подойдет). Затем запустим наши Docker контейнеры (посредством команды **docker compose**), а в самом конце запустим тесты и убедимся, что приложение работает правильно.

Создание CI конвейера (пример)

```
name: Django CI Workflow
on:
  workflow_dispatch:
env:
  DJANGO_SECRET_KEY: sf8d84098trfewfc0s8w048e7fs98d67gfsd786df8s56df8675df
  DJANGO_DEBUG: False
  DB_ENGINE: django.db.backends.mysql
  DB_NAME: django
  DB_USER: root
  DB_PASSWORD: secret
  DB_HOST: db
  DB_PORT: 3306
jobs:
  django_ci_job:
    runs-on: ubuntu-22.04
    steps:
      - uses: actions/checkout@v4
      - name: Fill .env File
        run: |
          echo "DJANGO_SECRET_KEY=${{env.DJANGO_SECRET_KEY}}" >> .env
          echo "DJANGO_DEBUG=${{env.DJANGO_DEBUG}}" >> .env
          echo "DB_ENGINE=${{env.DB_ENGINE}}" >> .env
          echo "DB_NAME=${{env.DB_NAME}}" >> .env
          echo "DB_USER=${{env.DB_USER}}" >> .env
          echo "DB_PASSWORD=${{env.DB_PASSWORD}}" >> .env
          echo "DB_HOST=${{env.DB_HOST}}" >> .env
          echo "DB_PORT=${{env.DB_PORT}}" >> .env
          echo "MYSQL_ROOT_PASSWORD=${{env.DB_PASSWORD}}" >> .env
          echo "MYSQL_DATABASE=${{env.DB_NAME}}" >> .env
      - name: Initialize Containers
        run: docker compose up --build -d
      - name: Wait for Containers Initialization
        run: sleep 30
      - name: Test Application
        run: docker compose exec -T web python manage.py test
```

Использование секретных данных (введение)

- Иногда при развертывании приложений или при загрузке приложений на DockerHub в виде готовых образов нам могут потребоваться секретные данные - пароли, логины и прочие коды доступа. Хранить их в открытом виде в файлах внутри директории **.github/workflows** является порочной практикой, т.к. они могут попасть в чужие руки и быть использованы для взлома ваших программ - особенно актуальной эта проблема становится в том случае, если ваш репозиторий является открытым. Однако эта проблема решаема - для этого надо использовать механизм секретов GitHub Actions - мы заранее создаем специальные переменные, значения которых можно увидеть только в момент создания. Затем мы применяем эти переменные в нужных местах.
- Для создания переменных следует перейти в настройки репозитория ("**Settings**"), выбрать раздел "**Secrets and variables**", выбрать подраздел "**Actions**" и нажать на зеленую кнопку "**New repository secret**". В открывшемся окне нам надо задать имя переменной (например, **SECRET_KEY**) и ее значение, а затем нажать на кнопку "**Add secret**". Внутри наших инструкций мы можем обращаться к секретам следующим образом: `${{ secrets.SECRET_KEY }}`.

Использование секретных данных (пример)

- Предположим, что нам нужно создать новый Docker образ из репозитория и загрузить этот образ на DockerHub. Для этого создадим секретные переменные **DOCKERHUB_USERNAME** (логин для DockerHub) и **DOCKERHUB_TOKEN** (код доступа для DockerHub). Затем применим их в нашей инструкции для подключения к DockerHub (также будем использовать готовый набор действий **docker/login-action@v3**). Интересный момент - в отчете исполнения задачи значения переменных отображаться не будут, чтобы мы ни делали - вместо них будут подставлены звездочки или они будут отсутствовать вообще:

```
name: DockerHub Upload Workflow
on:
  workflow_dispatch:
jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - name: Login to Docker Hub
        uses: docker/login-action@v3
        with:
          username: ${ secrets.DOCKERHUB_USERNAME }
          password: ${ secrets.DOCKERHUB_TOKEN }
      - name: Create Image
        run: docker build . -t our_dockerhub_repo/our_image:1.0 -f docker/web/Dockerfile
      - name: Upload Image
        run: docker push our_dockerhub_repo/our_image:1.0
```


Создание CD конвейера (введение)

- Теперь пришло время внутри наших GitHub Actions инструкций создать CD (Continuous Delivery) часть. При желании можно также добавить CI часть, в рамках которой можно протестировать наше приложение, однако для простоты оставим только CD. Предположим, что у нас есть удаленный сервер, где уже есть SSH и Docker, внутри которого работают контейнеры с частями нашего приложения, а также есть доступ к нашему GitHub. Очевидно, что нам нужно подключиться к удаленному серверу с помощью SSH, остановить работающие контейнеры, скачать обновления из репозитория и запустить контейнеры вновь.
- Для подключения по SSH применим готовый набор решений (**action**) под названием **appleboy/ssh-action@v1.0.1**. Специально для него создадим ряд секретных переменных - **HOST** (IP адрес или доменное имя удаленного сервера), **USERNAME** (имя пользователя), **KEY** (закрытый ключ, парный с открытым на удаленном сервере), **PORT** (порт, на котором работает SSH сервер) и **FINGERPRINT** (подпись удаленного сервера).

Создание CD конвейера (пример)

```
name: Django CD
on:
  push:
    branches: [ "main" ]
jobs:
  delivery:
    runs-on: ubuntu-latest
    steps:
      - name: Apply Changes
        uses: appleboy/ssh-action@v1.0.1
        with:
          host: ${ secrets.HOST }
          username: ${ secrets.USERNAME }
          key: ${ secrets.KEY }
          port: ${ secrets.PORT }
          fingerprint: ${ secrets.FINGERPRINT }
          script: |
            cd /app
            docker compose down
            git pull
            docker compose up -d --build
            docker image prune --force
```

Просмотр глобального контекста (введение)

- При работе с репозиторием нам может понадобиться некая “метаинформация” - например, тип события, который послужил инициатором запуска задания, название репозитория, владелец репозитория, код последнего коммита и т.д. Подобные данные аккумулирует в себе специальный держатель глобального контекста - объект **github**.
- Если мы захотим посмотреть внутренности объекта **github**, то мы должны применить специальную функцию **toJSON**, т.к. в “сыром” виде вложенные свойства объекта не отображаются. Это следует делать примерно так: **echo "\${toJSON(github)}"**.
- Вложенные свойства объекта **github** можно получить через точку, например, тип события-инициализатора - **\${github.event_name}**, название репозитория - **\${github.repository}**, имя владельца репозитория - **\${github.repository_owner}**, код последнего коммита - **\${github.sha}**.

Просмотр глобального контекста (пример)

```
name: Context Learning Workflow
on:
  workflow_dispatch:
jobs:
  build:
    runs-on: 'ubuntu-22.04'
    steps:
      - name: Display Global Context Object
        run: echo "${{ toJSON(github) }}"
      - name: Display Global Context Properties
        run: |
          echo "${{ github.event_name }}"
          echo "${{ github.repository }}"
          echo "${{ github.repository_owner }}"
          echo "${{ github.sha }}"
```

Регистрация собственных обработчиков (введение)

- Как уже было сказано, бесплатно и неограниченно (до определенного предела) мы можем применять обработчики наших заданий только при использовании открытых репозиторий - закрытые репозитории имеют жесткие лимиты. Однако эти лимиты можно убрать, если мы применим собственные обработчики - т.е. предоставим ресурсы своего компьютера для выполнения задач. Для этого надо перейти в настройки репозитория ("**Settings**"), выбрать пункт **Actions**, а внутри него пункт **Runners**. В открывшемся окне нажимаем большую зеленую кнопку **New self-hosted runner** и переходим на страницу, где подробно описаны возможности установки - с выбором типа операционной системы и архитектуры процессора.
- Когда мы хотим использовать собственный обработчик внутри задания, мы обращаемся к свойству **runs-on**. Все наши обработчики имеют название **self-hosted**, но если мы хотим выбрать обработчик с конкретными параметрами, то следует подставить эти параметры после названия:

```
runs-on: [self-hosted, linux, x64]
```

Регистрация собственных обработчиков (запуск обработчика на Linux)

```
# Создаем специальную директорию
mkdir actions-runner && cd actions-runner

# скачиваем архив с инсталлятором
curl -o actions-runner-linux-x64-2.311.0.tar.gz -L
https://github.com/actions/runner/releases/download/v2.311.0/actions-runner-linux-x64-2.311.0.tar.gz

# проверяем, что архив скачался без ошибок
echo "29fc8cf2dab4c195bb147384e7e2c94cfd4d4022c793b346a6175435265aa278
actions-runner-linux-x64-2.311.0.tar.gz" | shasum -a 256 -c

# распаковываем архив
tar xzf ./actions-runner-linux-x64-2.311.0.tar.gz

# настраиваем обработчик (с помощью своих собственных уникальных ключей)
./config.sh --url https://github.com/itlat-mysql/blog --token AJKHASFYAYSFYSHHCASHC

# запускаем обработчик
./run.sh
```

Сохранение артефактов после выполнения задачи (введение)

- Может так случиться, что нам потребуется не проверка работоспособности кода и даже не его размещение на каком-либо сервере, а получение результатов сборки нашего приложения в каком-либо упакованном виде, например, в виде архива.
- В процессе выполнения задания нам нужно создать сам архив. Для этого используется набор готовых решений (**action**) под названием **actions/upload-artifact@v4**. У этого свойства есть несколько настроек, но две из них основные - **path** (путь к директории или файлу, которые нужно упаковать) и **name** (идентификатор архива - для возможности обратиться к нему позже).

Сохранение артефактов после выполнения задачи (пример)

```
name: Django Artifact Workflow
on:
  workflow_dispatch:
jobs:
  build:
    runs-on: 'ubuntu-22.04'
    steps:
      - uses: actions/checkout@v4
      - uses: actions/setup-python@v5
        with:
          python-version: '3.12'
      - name: Enable Virtual Environment
        run: |
          python -m venv .venv
          source ./venv/bin/activate
      - name: Install Dependencies
        run: pip install -r requirements.txt
      - name: Compile Static Data
        run: python manage.py collectstatic --no-input --clear
      - name: Prepare Artifact
        uses: actions/upload-artifact@v4
        with:
          path: static
          name: django-archive
```


Использование матриц (**matrix**) (введение)

- Наше приложение может быть использовано не только в рамках какой-то одной операционной системы или даже версии языка программирования. Безусловно, мы можем написать несколько различных инструкций, однако все их различие будет состоять в том, что изменятся уже упомянутые версии операционных систем и языков. Чтобы избежать такого повторения, нам предлагается использовать концепцию т.н. матриц. Мы объявляем набор определенных переменных, в процессе исполнения GitHub Actions выделит различные комбинации значений этих переменных и запустит столько задач, сколько у нас есть комбинаций. Притом в рамках каждой задачи будут доступна уникальная комбинация переменных матрицы.
- Матрицу следует определять внутри какого-либо задания (**jobs**) - сначала применяется свойство **strategy**, внутри него **matrix**, а уже внутри **matrix** определяются свойства-переменные. Для получения доступа к переменным матрицы внутри задания, следует применять следующую конструкцию: `${{ matrix.some_variable }}`.

Использование матриц (matrix) (пример)

```
name: Django Matrix Workflow
on:
  workflow_dispatch:
jobs:
  build:
    strategy:
      matrix:
        python: [3.11, 3.12]
        os: [ubuntu-latest, windows-latest]
    runs-on: ${ matrix.os }
    steps:
      - uses: actions/checkout@v4
      - uses: actions/setup-python@v5
        with:
          python-version: ${ matrix.python }
      - name: Enable Windows Virtual Environment
        if: matrix.os == 'windows-latest'
        run: |
          python -m venv .venv
          .\.venv\Scripts\activate
      - name: Enable Linux Virtual Environment
        if: matrix.os == 'ubuntu-latest'
        run: |
          python -m venv .venv
          source ./.venv/bin/activate
      - name: Install Dependencies
        run: pip install -r requirements.txt
      - name: Check System State
        run: python manage.py check
```

Отправка сообщений в чрезвычайной ситуации (введение)

- Рано или поздно мы столкнемся с тем, что наше задание будет выполнено с ошибкой. Если мы запускали его вручную, мы сразу увидим это и предпримем соответствующие меры. А что если задание будет запущено в автоматическом режиме? В данном случае мы должны перехватить ошибку и отправить нашему человеку сообщение - по электронной почте, Telegram, Slack и т.д. GitHub Actions позволяет это сделать - нам надо всего лишь создать задание (внутри **jobs**), которое будет обрабатывать только при определенном условии (свойство **if**) и условие это будет заключаться в том, что у нас произошла ошибка - **`${{ failure() }}`**.
- Чтобы отправить сообщение по электронной почте, воспользуемся готовым решением **dawidd6/action-send-mail@v3**. Для правильного функционирования нужны следующие параметры: **server_address** - адрес нашего SMTP сервера (можно использовать **mailtrap**), **server_port** - порт SMTP сервера, **username** - логин SMTP сервера, **password** - пароль SMTP сервера, **subject** - тема сообщения, **body** - текст сообщения, **to** - адрес получателя сообщения, **from** - адрес отправителя сообщения. Также создадим две секретных переменных - **EMAIL_USERNAME** (логин SMTP сервера) и **EMAIL_PASSWORD** (пароль SMTP сервера).

Отправка сообщений в чрезвычайной ситуации (пример)

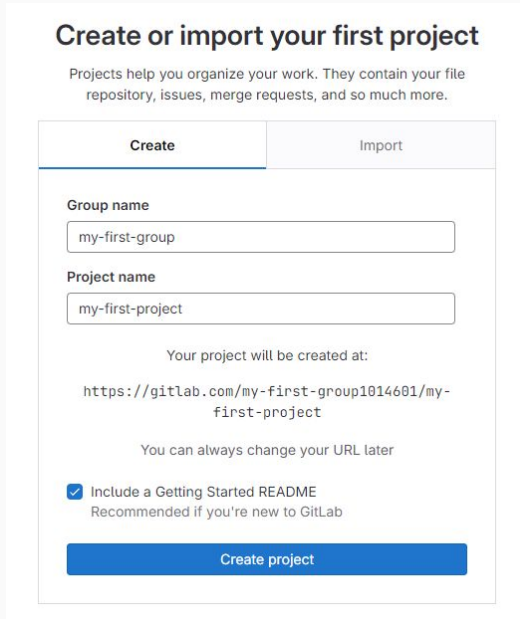
```
name: Email on Error Workflow
on:
  workflow_dispatch:
jobs:
  error_simulation:
    runs-on: 'ubuntu-22.04'
    steps:
      - name: Error
        run: exit 1
  email_sending:
    runs-on: 'ubuntu-22.04'
    needs: error_simulation
    if: ${{ failure() }}
    steps:
      - name: Send Email
        uses: dawidd6/action-send-mail@v3
        with:
          server_address: sandbox.smtp.mailtrap.io
          server_port: 2525
          username: ${{ secrets.EMAIL_USERNAME }}
          password: ${{ secrets.EMAIL_PASSWORD }}
          subject: An unexpected error occurred!
          body: ${{ github.workflow }} of ${{ github.repository }} has failed
          to: receiver@email.com
          from: sender@email.com
```

Использование GitLab CI/CD

- Сервис GitLab (<https://gitlab.com>) в некотором смысле является альтернативой сайту GitHub. Внутри него также можно хранить свои Git репозитории, вести разработку совместно с другими людьми, пользоваться CI/CD средствами и т.д.
- GitLab появился на несколько лет позже, чем GitHub, и его основной целью было вовсе не размещение у себя репозитория, а предоставление максимально удобного способа для применения DevOps технологий и коммерческой разработки с обеспечением разнородной политики доступа к исходному коду.
- Со временем GitHub и GitLab переняли друг у друга часть функционала, что сделало их очень похожими. Однако по-прежнему там, где GitHub полагается на некие внешние средства (**actions** и т.д.) GitLab имеет встроенные решения. Хотя надо признать, что слишком много встроенных решений привели к тому, что GitLab имеет перегруженный интерфейс, который иногда сложно понять.
- В числе прочего GitLab имеет возможность создавать встроенную в проект документацию, мониторить безопасность кода, переписываться с другими участниками разработки, а также развернуть на своем сервере собственную версию GitLab, которая распространяется по свободной лицензии и может быть настроена каким угодно образом.

Создание первого проекта/репозитория

- Сразу же после регистрации нам будет предложено создать наш первый проект, внутри которого и будет находиться репозиторий, а также все, что с ним связано. Кроме того, для создания проекта нам потребуется группа - она необходима для того, чтобы потом добавлять в группу новых участников, настраивать права доступа для участников, добавлять в группу новые проекты и т.д. Условимся, что названием группы будет **my-first-group**, а названием проекта - **my-first-project**:



The screenshot shows the 'Create or import your first project' page in GitLab. It features a 'Create' tab and an 'Import' tab. The 'Create' tab is active, showing input fields for 'Group name' (my-first-group) and 'Project name' (my-first-project). Below these fields, it displays the project URL: https://gitlab.com/my-first-group1014601/my-first-project. There is a checkbox for 'Include a Getting Started README' which is checked. At the bottom, there is a blue 'Create project' button.

Create or import your first project

Projects help you organize your work. They contain your file repository, issues, merge requests, and so much more.

Create Import

Group name
my-first-group

Project name
my-first-project

Your project will be created at:

<https://gitlab.com/my-first-group1014601/my-first-project>

You can always change your URL later

☒ Include a Getting Started README
Recommended if you're new to GitLab

Create project

Навигация по проекту

- После того, как проект был создан, мы можем в него перейти. Если мы нажмем на иконку собачки в верхнем левом углу, то увидим список всех доступных нам проектов/репозиториев. Нажав на название проекта, мы окажемся внутри него - слева будет располагаться меню с настройками, а по центру - сам репозиторий:

The screenshot displays the GitLab web interface for a project named "my-first-project". The interface is divided into several sections:

- Left Sidebar:** Contains the GitLab logo, navigation icons, a search bar, and a list of project items. The "Project" section is expanded, showing "my-first-project" as the selected item. Below it are links for "Learn GitLab", "Pinned", "Manage", "Plan", "Code", "Build", "Secure", "Deploy", "Operate", "Monitor", "Analyze", and "Settings".
- Header:** Shows the breadcrumb "my-first-group / my-first-project" and the project name "my-first-project" with a "Free" badge. It also includes buttons for "main", "my-first-project /", and a "+" icon. On the right, there are buttons for "History", "Find file", "Edit", and "Code".
- Commit Information:** Displays the "Initial commit" by Alexander Kovalenko, authored 4 minutes ago, with the commit hash "a485e779".
- File List:** A table showing the project's files and their last commit information.

Name	Last commit	Last update
README.md	Initial commit	4 minutes ago
- Project Information:** A sidebar on the right containing project statistics and links for adding various files and configurations.
 - 1 Commit
 - 1 Branch
 - 0 Tags
 - 3 KiB Project Storage
 - Links: README, Add LICENSE, Add CHANGELOG, Add CONTRIBUTING, Enable Auto DevOps, Add Kubernetes cluster, Set up CI/CD, Add Wiki, Configure Integrations.

Получение данных проекта

- Если мы желаем получить данные проекта - в качестве архива, HTTPS ссылки или SSH ссылки, мы должны внутри репозитория нажать большую синюю кнопку **"Code"** - внутри как раз будут доступны ссылки для HTTP и SSH клонирования, загрузка данных в архивах разных форматов, а также интеграция для различных IDE:

The screenshot displays the GitLab web interface for a project named 'my-first-project'. The interface is divided into several sections:

- Left Sidebar:** Contains navigation links for 'Project', 'Learn GitLab', 'Pinned', 'Manage', 'Plan', 'Code', 'Build', 'Secure', 'Deploy', 'Operate', 'Monitor', 'Analyze', and 'Settings'.
- Header:** Shows the project name 'my-first-project' and a 'Free' badge.
- Main Content Area:** Displays the 'Initial commit' by Alexander Kovalenko, a table of files (README.md), and a 'Getting started' section.
- Right Sidebar:** Contains 'Project information' including commit count, branch count, tags, and project storage.

The 'Code' button is highlighted, and a dropdown menu is open, showing the following options:

- Clone with SSH:** `git@gitlab.com:my-first-group10`
- Clone with HTTPS:** `https://gitlab.com/my-first-gro`
- Open in your IDE:** Visual Studio Code (SSH), Visual Studio Code (HTTPS), IntelliJ IDEA (SSH), IntelliJ IDEA (HTTPS)
- Download source code:** zip, tar.gz, tar.bz2, tar

Основы совместной работы - доступ в рамках группы

- Единоличная разработка крупного проекта в наше время практически невозможна - необходимо подключать коллег. В GitLab это можно сделать на двух уровнях - во-первых на уровне группы (участники группы будут иметь права на все проекты, которые входят в группу) и, во-вторых, на уровне проекта - права участников будут распространяться только на конкретный проект.
- Чтобы подключить коллег на уровне группы, надо нажать на иконку собачки в левом верхнем углу, в меню слева выбрать пункт **Groups**, затем выбрать в списке групп необходимую группу, после этого в левом меню нужно навести мышь на пункт **Manage** и выбрать в открывшемся списке пункт **Members**. Появится список участников группы и вверху справа большая синяя кнопка **"Invite members"**. Нажимаем на нее - в открывшемся окне мы можем найти конкретного пользователя, присвоить ему роль (о ролях поговорим позже), а также выбрать период его участия в группе. Затем нажимаем кнопку **"Invite"**.

Основы совместной работы - доступ в рамках группы (пример)

Invite members

×

You're inviting members to the **my-first-group** group.

Username, name or email address


Select members or type email addresses

Select a role

Guest ▾

[Read more](#) about role permissions

Access expiration date (optional)

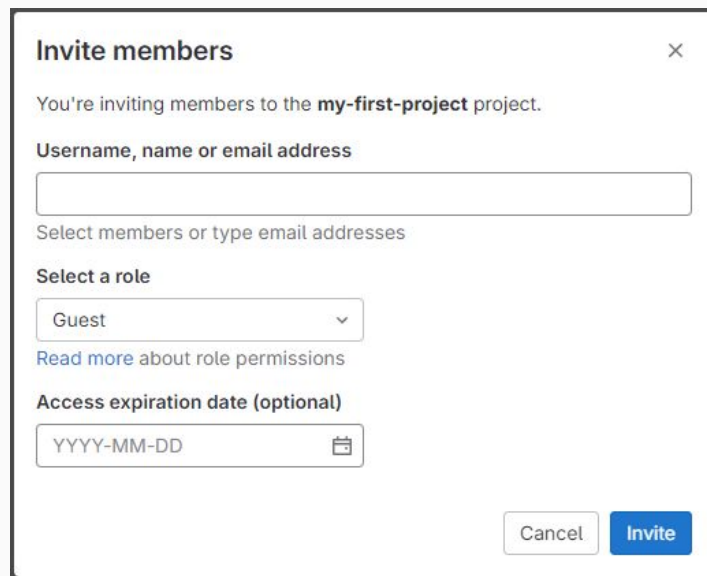
YYYY-MM-DD 

Cancel

Invite

Основы совместной работы - доступ в рамках проекта

- Доступ в рамках проекта предоставляется схожим образом. Нажимаем на иконку собачки в верхнем левом углу, затем выбираем в списке проектов проект, после этого в левом меню наводим мышь на пункт **Manage** и выбираем в открывшемся списке пункт **Members**. Появится список участников проекта и вверху сверху синяя кнопка **"Invite members"**. Нажимаем на нее - в открывшемся окне мы можем найти конкретного пользователя, присвоить ему роль, а также выбрать период его участия в проекте. Затем нажимаем кнопку **"Invite"**:



The screenshot shows a modal dialog titled "Invite members" with a close button (X) in the top right corner. The dialog contains the following elements:

- A message: "You're inviting members to the **my-first-project** project."
- A text input field labeled "Username, name or email address".
- A label "Select members or type email addresses" below the input field.
- A section titled "Select a role" containing a dropdown menu currently set to "Guest".
- A link "[Read more](#) about role permissions" below the role dropdown.
- A section titled "Access expiration date (optional)" containing a date input field with the placeholder "YYYY-MM-DD" and a calendar icon.
- At the bottom right, there are two buttons: "Cancel" and "Invite".

Особенности ролей

- При подключении работников в рамках проекта и группы мы уже упоминали роли. Всего их 5 видов - **Guest**, **Reporter**, **Developer**, **Maintainer** и **Owner**:
 - **Guest** имеет только права на чтение. Могут просматривать код, задачи, и другие ресурсы, но не могут вносить изменения.
 - **Reporter** обладает правами на чтение и создание задач (**issues**) и запросов на слияние (**merge requests**). Может комментировать задачи и запросы на слияние, но не может вносить изменения в код.
 - **Developer** имеет права на чтение и запись. Может создавать, изменять и удалять код, а также открывать и закрывать задачи и запросы на слияние.
 - **Maintainer** обладает всеми правами, которые есть у предыдущих типов, плюс может добавлять или удалять пользователей, управлять правами доступа, а также создавать новые ветки.
 - **Owner** имеет полный контроль над проектом и всеми его ресурсами. Может управлять членами команды, включая их правами доступа.

Настройка прав ролей на ветки

- Дополнительно мы можем настроить права для ролей Developer и Maintainer. Для подтверждения этого тезиса переходим в наш проект **my-first-project**, в левом меню находим пункт **Settings** и нажимаем подпункт **Repository**. В открывшемся окне отрываем (**Expand**) блок **“Protected Branches”**. Внутри мы можем создавать защищенные ветки (**Add Protected Branches**), внутри которых можно указать те роли, которые могут заниматься слиянием в эти ветку (**merge**) и сохранением кода (**push**) в эту ветку:

Protected branches

Collapse

Keep stable branches secure and force developers to use merge requests. [What are protected branches?](#)

⚠ Giving merge rights to a protected branch also gives elevated permissions for certain CI/CD features. [What are the security implications?](#)

Protected branches 1

Add protected branch

By default, protected branches restrict who can modify the branch. [Learn more.](#)

Branch	Allowed to merge	Allowed to push and merge	Allowed to force push ?
main default	No one	No one	<input checked="" type="checkbox"/> Unprotect
<div>Roles<ul style="list-style-type: none">Developers + MaintainersMaintainers✓ No one</div>			

Protected tags

Limit access to creating and updating tags. [What are protected tags?](#)

Expand

Удаленный доступ - токены доступа (введение)

- Очевидно, что мы не будем все время менять и добавлять файлы в репозитории с помощью интерфейса самого GitLab - рано или поздно нам понадобится удаленный доступ. Для получения этого доступа есть много возможностей, но две являются основными - создания токена доступа и аутентификация по ключу.
- Для создания токена доступа мы должны перейти в наш профиль и в левом меню выбрать пункт **Access Tokens**. Затем вверху справа мы должны нажать на кнопку **Add new token**. В открывшемся окне мы должны указать название токена, срок его годности и те права, которые получит пользователь токена. Прав много, но основных два - **read_repository** (позволяет просматривать репозитории) и **write_repository** (позволяет просматривать репозитории и вносить в них изменения). В конце мы должны нажать кнопку **Create personal access token**.
- Сразу после создания токена у нас будет возможность его скопировать - в дальнейшем мы не сможем увидеть токен в явном виде. Использовать токен можно следующим образом:

```
git clone https://<username>:<token>@gitlab.com/<group>/<project>.git
```

```
// в этом случае мы должны будем ввести токен вручную  
git clone https://<username>@gitlab.com/<group>/<project>.git
```

Удаленный доступ - токены доступа (пример)

Personal Access Tokens

You can generate a personal access token for each application you use that needs access to the GitLab API. You can also use personal access tokens to authenticate against Git over HTTP. They are the only accepted password when you have Two-Factor Authentication (2FA) enabled.

Active personal access tokens 0

Add a personal access token

Token name

For example, the application using the token or the purpose of the token.

Expiration date

Select scopes

Scopes set the permission levels granted to the token. [Learn more.](#)

- ☐ **api**
Grants complete read/write access to the API, including all groups and projects, the container registry, the dependency proxy, and the package registry.
- ☐ **read_api**
Grants read access to the API, including all groups and projects, the container registry, and the package registry.
- ☐ **read_user**
Grants read-only access to the authenticated user's profile through the /user API endpoint, which includes username, public email, and full name. Also grants access to read-only API endpoints under /users.
- ☐ **create_runner**
Grants create access to the runners.
- ☐ **k8s_proxy**
Grants permission to perform Kubernetes API calls using the agent for Kubernetes.
- ☒ **read_repository**
Grants read-only access to repositories on private projects using Git-over-HTTP or the Repository Files API.
- ☒ **write_repository**
Grants read-write access to repositories on private projects using Git-over-HTTP (not using the API).
- ☐ **read_registry**
Grants read-only access to container registry images on private projects.
- ☐ **write_registry**
Grants write access to container registry images on private projects.
- ☐ **ai_features**
Grants access to GitLab Duo related API endpoints.

Create personal access token

Cancel

Удаленный доступ - публичный ключ (введение)


- Также мы можем удаленно подключиться к проекту с помощью пары закрытый ключ - публичный ключ. Если на нашем компьютере еще нет созданных ключей, то мы можем создать их при помощи команды **ssh-keygen**. После этого файл с закрытым ключом размещаем в директории **.ssh** нашего пользователя, а содержимое публичного ключа копируем и переходим на сайт GitLab.
- Переходим в наш профиль, в левом меню выбираем пункт меню **SSH Key** и нажимаем кнопку **Add new key**. Внутри в поле **Key** копируем наш публичный ключ, в поле **Title** вставляем пишем название нашего ключа, затем выбираем тип (тип с наиболее полным доступом - **Authentication & Signing**). Кроме того, можно указать срок годности нашего ключа, но это необязательное условие.
- Использовать публичный ключ вручную нам не надо - просто следует применять следующую ссылку на проект:

```
git@gitlab.com:<group>/<project>.git
```

Удаленный доступ - публичный ключ (пример)

SSH Keys

SSH keys allow you to establish a secure connection between your computer and GitLab. SSH fingerprints verify that the client is connecting to the correct host. Check the [current instance configuration](#).

Your SSH keys  0

Add an SSH key

Add an SSH key for secure access to GitLab. [Learn more](#).

Key

Begins with 'ssh-rsa', 'ecdsa-sha2-nistp256', 'ecdsa-sha2-nistp384', 'ecdsa-sha2-nistp521', 'ssh-ed25519', 'sk-ecdsa-sha2-nistp256@openssh.com', or 'sk-ssh-ed25519@openssh.com'.

Title

Key titles are publicly visible.

Usage type

Expiration date










Optional but recommended. If set, key becomes invalid on the specified date.

Создание запроса на слияние - “merge request” (введение)














- Предположим, что мы начали реализацию нашего проекта - как известно, разработка не ведется в основной ветке (более того, она обычно защищена от изменений), поэтому мы создаем ветку **development**. Сохраним в эту ветку наш проект с Django и загрузим файлы (**push**) в наш репозиторий на GitLab.
- Теперь нам необходимо создать запрос на слияние нашей ветки **development** с основной веткой. Для этого надо перейти в наш проект и выбрать в меню слева пункт **Merge requests**. После этого вверху слева мы должны нажать на кнопку **New merge request**. В результате мы сможем выбрать ветку, из которой надо забрать изменения, и ветку, куда надо влить изменения. Затем мы можем выбрать ответственного за выполнение этого запроса и некоторые другие опции.
- После того, как запрос создан, участники проекта с соответствующими правами смогут просмотреть запрос, познакомиться с теми изменениями, которые он в себе несет, оставить комментарии, а затем, если все в порядке, подтвердить слияние данных в основную ветку.

Создание запроса на слияние - “merge request” (пример)





Project

-  my-first-project
-  Learn GitLab 17%
-  Pinned >
-  Manage >
-  Plan >
-  Code >
-  Build >
-  Secure >
-  Deploy >
-  Operate >
-  Monitor >
-  Analyze >
-  Settings >


my-first-group / my-first-project / Merge requests / New


New merge request

Source branch

my-first-group1014601/my-first-project ▾

development ▾

**added project files**
Alex Kova authored Jan 02, 2024


9c81c5e5 


Compare branches and continue

Target branch

my-first-group1014601/my-first-project ▾

main ▾

**Initial commit**
Alexander Kovalenko authored Jan 02, 2024

a485e779 

Основы создания GitLab CI/CD конвейера (введение)

- Если мы хотим внедрить CI/CD подход в наш GitLab репозиторий, то в корне репозитория надо создать файл **.gitlab-ci.yml**, который и будет содержать все необходимые команды. Как можно понять из названия, файл должен содержать инструкции в формате YAML.
- Каждая задача внутри контейнера должна иметь свое уникальное имя. По умолчанию все задачи файла исполняются внутри Docker контейнера **ruby:3.1**. Однако мы можем переписать это поведение при помощи свойства **image** (должно располагаться внутри конкретной задачи), где следует указать имя нашего образа, например, **python:3.12-slim**.
- Сами команды могут быть указаны внутри свойства **script** - каждую отдельную команду следует оформлять в виде элемента YAML массива. Файлы репозитория подгружать не нужно, они уже находятся внутри контейнера в момент запуска команд задачи.
- По умолчанию команды запускаются при каждом изменении (**push**) внутри репозитория, но позже это поведение можно настроить под наши нужды.

Основы создания GitLab CI/CD конвейера (пример)

```
simple-job:
  script:
    - ls -la

another-image-job:
  image: python:3.12-slim
  script:
    - python --version
```

Определение параметров по умолчанию и этапов (введение)

- Очень часто нам потребуется некоторое поведение, которое будет необходимо сразу для нескольких заданий. Для этого существует глобальное свойство **default**, которому в виде массива можно прописать некоторые действия. Например, общий для всех Docker образ (свойство **image**), либо специальное свойство **before_script**, внутри которого можно в виде массива написать те команды, которые будут выполняться до конкретных шагов каждой задачи (например установка всех зависимостей Django проекта).
- Кроме поведения по умолчанию, также глобально можно указать т.н. этапы (**stages**). Каждой задаче можно указать принадлежность к тому или иному этапу. Данное действие не влечет за собой каких-либо особых последствий - во-первых, это логически группирует задачи, и, во-вторых, задания будут выполняться в том порядке, в котором прописаны их этапы.

Определение параметров по умолчанию и этапов (пример)

```
default:
  image: python:3.12-bookworm
  before_script:
    - python -m venv .venv
    - source ./venv/bin/activate
    - pip install -r requirements.txt

stages:
  - check
  - build

compile-project:
  stage: build
  script:
    - python manage.py collectstatic --no-input --clear
    - ls -la static

check-project-health:
  stage: check
  script:
    - python manage.py check
```


Использование переменных (введение)

- Как и в случае GitHub Actions, внутри конвейера GitLab нам могут пригодиться переменные, которые могут быть использованы в различных задачах. Безусловно, мы можем прописывать значения каждый раз вручную, но если потом придется что-то поменять, то будет гораздо удобнее изменить значение в одном месте, чем искать повторяющиеся значения в нескольких инструкциях.
- Переменные могут быть двух видов - для каждого задания, а также глобальные, которые будут доступны сразу всем заданиям. И в первом и во втором случае переменные объявляются при помощи свойства **variables**, только глобальные прописываются на самом верхнем уровне вложенности, а переменные для заданий - непосредственно внутри заданий.

Использование переменных (пример)

```
default:
  image: python:3.12-bookworm
  before_script:
    - python -m venv .venv
    - source ./venv/bin/activate
    - pip install -r requirements.txt

variables:
  DJANGO_SECRET_KEY: my-secret-key
  DB_NAME: test
  DB_ENGINE: django.db.backends.sqlite3

check-project:
  script:
    - cp .env.example .env
    - sed -i "s/DJANGO_SECRET_KEY=/DJANGO_SECRET_KEY=${DJANGO_SECRET_KEY}/" .env
    - sed -i "s/DB_NAME=/DB_NAME=${DB_NAME}/" .env
    - sed -i "s/DB_ENGINE=/DB_ENGINE=${DB_ENGINE}/" .env
    - python manage.py check

test-project:
  script:
    - cp .env.example .env
    - sed -i "s/DJANGO_SECRET_KEY=/DJANGO_SECRET_KEY=${DJANGO_SECRET_KEY}/" .env
    - sed -i "s/DB_NAME=/DB_NAME=${DB_NAME}/" .env
    - sed -i "s/DB_ENGINE=/DB_ENGINE=${DB_ENGINE}/" .env
    - python manage.py test
```

Использование секретных переменных (введение)

- Если мы используем переменные для неких технических целей, то их значения можно прописывать так, чтобы они были видны при исполнении и в самом **.gitlab-ci.yml** файле. Однако мы можем применять переменные, которые служат реквизитами для аутентификации в неких внешних сервисах - очевидно, что их нельзя использовать в открытом виде.
- Для создания переменных нам необходимо перейти в наш проект, в левом меню выбрать пункт **Settings** и нажать на подпункт **CI/CD**. Внутри открывшегося окна мы должны найти блок **Variables** и нажать на кнопку **Add Variable**. Теперь мы можем создавать переменные - уберем опцию, что переменная используется только в защищенных ветках (**Protect variable**) и включим опцию, что переменная должна быть замаскированной (**Mask variable**). Затем создадим переменную **DOCKER_USERNAME**, которая будет содержать имя нашего DockerHub пользователя. Затем схожим образом создадим переменную **DOCKER_TOKEN**, которая включает в себя токен доступа для DockerHub профиля.

Использование служб (введение)

- Далее нужно использовать скрытые переменные, чтобы аутентифицироваться на DockerHub, создать новый образ и загрузить его на DockerHub. Для этого нам нужен специальный образ, который поддерживает команды для Docker. Такой образ существует, он называется **docker**. Однако у него внутри не хватает Docker демона, который позволяет функционировать всей инфраструктуре Docker в качестве единого целого.
- Чтобы решить нашу проблему, следует воспользоваться свойством GitLab под названием **services**. Внутри него надо прописывать те Docker образы, контейнеры которых будут запускаться параллельно с основным контейнером. Важный момент - программы внутри этих контейнеров доступны по сети, и GitLab присваивает им имена, которые соответствуют именам образов. Нам нужен контейнер, который также называется **docker**, но имеет специальный тэг **dind** (Docker in Docker) - внутри него как раз и доступен Docker демон.
- Подробнее об этой не очень простой для понимания теме можно прочитать по следующей ссылке: <https://docs.gitlab.com/ee/ci/services/>

Использование секретных переменных и служб (пример)

```
build-new-image:
  image: docker:24.0
  services:
    - docker:24.0-dind
  script:
    - docker login -u $DOCKER_USERNAME -p $DOCKER_TOKEN
    - docker build . -t our_dockerhub_repo/our_image:1.0 -f docker/web/Dockerfile
    - docker push our_dockerhub_repo/our_image:1.0
```

Работа с артефактами (введение)

- Как и GitHub, Gitlab позволяет скачать результаты деятельности наших задач (артефакты) после того, как эти задачи были завершены. Для этого необходимо воспользоваться свойством **artifacts**, которое должно располагаться внутри задания. Этому свойству необходимо указать массив с путями (**paths**), которые будут использованы для формирования артефакта-архива.
- Также мы можем указать дополнительные параметры. Во-первых, имя артефакта в самой базовой конфигурации будет сгенерировано автоматически. Если мы хотим подставить свое собственное имя, нам следует применить свойство **name**. Во-вторых, мы можем указать срок годности нашего артефакта. По умолчанию, все архивы будут нам доступны в течение 30 дней. Если мы хотим изменить поведение по умолчанию, то следует применить свойство **expire_in** - можно подставить ему значение, равное, например, **1 day**.

Работа с артефактами (пример)

```
create-artifact:
  image: python:3.12-bookworm
  script:
    - python -m venv .venv
    - source ./venv/bin/activate
    - pip install -r requirements.txt
    - python manage.py collectstatic --no-input --clear
  artifacts:
    name: "my-archive"
    paths:
      - static/
```

Использование встроенных переменных (введение)

- Мы уже умеем использовать переменные, созданные нами. Также существуют встроенные переменные, полный список которых представлен по следующей ссылке - https://docs.gitlab.com/ee/ci/variables/predefined_variables.html. Здесь приведем самые полезные их варианты:
 - **\$CI_COMMIT_SHA** - хэш коммита, для которого было запущено задание
 - **\$CI_COMMIT_BRANCH** - имя ветки, для которой было запущено задание
 - **\$CI_MERGE_REQUEST_TARGET_BRANCH_NAME** - запрос на слияние был для этой ветки
 - **\$CI_PIPELINE_SOURCE** - вид события, которое запустило задание
 - **\$CI_COMMIT_TITLE** - комментарий к коммиту
 - **\$CI_COMMIT_TIMESTAMP** - временная метка коммита

Использование встроенных переменных (пример)

```
show-variables:  
  script:  
    - echo $CI_COMMIT_SHA  
    - echo $CI_COMMIT_BRANCH  
    - echo $CI_MERGE_REQUEST_TARGET_BRANCH_NAME  
    - echo $CI_PIPELINE_SOURCE  
    - echo $CI_COMMIT_TITLE  
    - echo $CI_COMMIT_TIMESTAMP
```

Настройка событий для запуска конвейера (введение)

- По умолчанию конвейер, описанный в файле **.gitlab-ci.yml**, запускается при обновлении любой ветки репозитория - т.е. в том случае, если произошел **push**. Однако у нас есть возможность настроить это более гибким способом - использовать разные события и ветки. Все события перечислены по ссылке: https://docs.gitlab.com/ee/ci/jobs/job_control.html#common-if-clauses-for-rules
- Для того, чтобы настроить поведения для всех задач, используется свойство **workflow**, внутри которого следует расположить свойство **rules**. Внутри **rules** надо прописать массив внутренних свойств, главным из которых является **if** - как раз внутри **if** и следует прописывать условия, при которых будет запускаться конвейер. Также важным внутренним свойством является **when** - в нем можно уточнить, как будет запускаться задание: **always** (всегда), **never** (никогда), **manual** (только вручную).
- Условия для запуска можно устанавливать и для отдельных заданий, а не для всего конвейера. Для этого надо прописать свойство **rules** внутри самого задания. Вся внутренняя структура будет такая же, как и у глобальных условий.

Настройка событий для запуска конвейера (пример)

```
workflow:
  rules:
    - if: $CI_PIPELINE_SOURCE == "push" && $CI_COMMIT_BRANCH == 'main'

main-job:
  script:
    - echo "Hello from main job!"
```

```
workflow:
  rules:
    - if: $CI_COMMIT_BRANCH == "main"
      when: never
    - if: $CI_PIPELINE_SOURCE == "push"

non-main-branch-job:
  script:
    - echo "Hello from non-main branch job!"
```

```
manual-job:
  script:
    - echo "Hello from manual job!"
  rules:
    - if: $CI_PIPELINE_SOURCE == "push"
      when: manual

automatic-job:
  script:
    - echo "Hello from automatic job!"
```

Запуск тестов в контексте использования нескольких контейнеров (пример)

```
run-docker-compose-tests:
  image: docker:24.0
  services:
    - docker:24.0-dind
  script:
    - |
      echo "DJANGO_SECRET_KEY=sf8d84098trfewfc0s8w0" >> .env
      echo "DJANGO_DEBUG=False" >> .env
      echo "DB_ENGINE=django.db.backends.mysql" >> .env
      echo "DB_NAME=django" >> .env
      echo "DB_USER=root" >> .env
      echo "DB_PASSWORD=secret" >> .env
      echo "DB_HOST=db" >> .env
      echo "DB_PORT=3306" >> .env
      echo "MYSQL_ROOT_PASSWORD=secret" >> .env
      echo "MYSQL_DATABASE=django" >> .env
    - docker compose up --build -d
    - sleep 30
    - docker compose exec -T web python manage.py test
```