CI/CD на GitHub Actions и GitLab CI для самых маленьких

Часть 1: Основы CI/CD – что это и зачем нужно; обзор GitHub Actions и GitLab CI

> Когда я только начинал свой путь в сторону DevOps и автоматизации, мне не хватало материалов, которые не просто показывали бы, как настроить CI/CD, но объясняли бы — *зачем* это нужно, *как* это работает, *почему* это стало стандартом. Слишком часто всё сводилось к "вот скопируй `.yml` и всё будет". Этот цикл статей — попытка дать вам то, чего тогда не хватило мне самому.

> Данная статья только начало пути становления, вот небольшая карта:
>
> **Часть 1: Основы CI/CD – что это и зачем нужно; GitHub Actions и GitLab CI**
>
> **^ Вы сейчас здесь ^**
>
> **Часть 2: Настройка GitHub Actions и GitLab CI – первый workflow и деплой**
>
> **Часть3: CI/CD – ветки, условия, секреты и окружения**

Введение

В последние годы вы, вероятно, слышали слова "DevOps", "CI", "CD", возможно, даже "GitHub Actions" или "GitLab CI". Но что это всё значит на практике? Нужно ли быть сеньором DevOps-инженером, чтобы с этим разобраться? К счастью — нет. Эта серия статей написана простым языком для тех, кто только начинает знакомство с автоматизацией разработки.

Сегодня мы разберём базовые понятия: что такое CI/CD, как оно работает, какие инструменты для этого есть — и чем отличаются GitHub Actions и GitLab CI. Мы не будем

углубляться в абстракции или перегружать терминологией. Вместо этого — объясню на примерах, аналогиях и легкостью.

CI/CD — это не какая-то магия или модный корпоративный тренд. Это ваш личный **робот-помощник**, который берёт на себя скучные задачи: запускает тесты, собирает проект, выкладывает его на сервер. Представьте себе автоматизированный **конвейер**, куда вы кладёте свежий код — а на выходе получаете готовое к работе приложение. Звучит круто? Тогда поехали.

Что такое CI и CD

Аббревиатуры CI и CD — одни из самых часто упоминаемых в мире DevOps. Давайте разберёмся с каждой по порядку:

CI — Continuous Integration (непрерывная интеграция)

Идея простая: когда несколько разработчиков работают над проектом, они постоянно вносят изменения в код. Раньше это приводило к хаосу — слияние изменений, баги, несостыковки. С CI при **каждом коммите** изменения автоматически собираются, тестируются и проверяются. Это позволяет быстрее находить ошибки и быть уверенным, что "ничего не сломалось".

Непрерывная интеграция — это практика:

- частого слияния изменений в основной репозиторий (хотя бы раз в день),
- автоматического запуска тестов и проверок при каждом изменении,
- своевременного выявления ошибок до того, как они попадут на продакшн.

Цель — **не допустить "интеграционного ада"**, когда всё работает по отдельности, но ничего не работает вместе.

Пример: вы изменили один файл и закоммитили. Система тут же запускает сборку и юнит-тесты. Если что-то пошло не так — вы узнаете об этом сразу, а не через неделю, когда баг всплывёт у пользователя.

CD — Continuous Delivery / Deployment (непрерывная доставка / развёртывание)

CD идёт дальше. Если CI гарантирует, что ваш код работает корректно после изменений, то CD гарантирует, что **его можно развернуть**. Это может быть:

- **Delivery** — автоматическая подготовка к выпуску, но с ручным одобрением;
- **Deployment** — полностью автоматическое развертывание на сервер после прохождения всех тестов.

Проще говоря: если CI проверяет, что ваш код **не сломан**, то CD делает так, чтобы этот исправный код оказался **на сервере или в продакшене**.

Pipeline (пайплайн)

Пайплайн — это цепочка шагов, которая запускается при изменении кода. Обычно она состоит из:

1. Сборки проекта.
2. Запуска тестов.
3. Развёртывания (если всё прошло успешно).

Аналогия: вы кладёте ингредиенты на вход, конвейер готовит, проверяет и подаёт готовое блюдо. И всё это без участия человека. CI/CD — это не гаджет, а **умный робот**, который каждый день помогает команде экономить часы времени.

Зачем нужна CI/CD

Если коротко — чтобы **разработка шла быстрее, качественнее и спокойнее**.

Преимущества:

- **Раннее обнаружение ошибок.** Лучше поймать баг сразу, чем искать его в продакшене.
- **Быстрая обратная связь.** Коммит → проверка → результат.
- **Автоматизация.** Не нужно запускать тесты вручную — всё происходит само.
- **Готовность к релизу.** Вы всегда уверены, что проект можно выкатить.

Без CI/CD:

- Каждый разработчик вручную запускает сборку и тесты.
- Часто забывают что-то проверить.
- Ошибки всплывают поздно, когда их уже сложно исправить.

С CI/CD:

- Ошибка ловится за минуты.
- Никто не забывает протестировать.
- Релизы стабильнее и выходят быстрее.

CI/CD — это не просто удобство, это **стратегическое преимущество** команды. И главное — настроить это можно с минимальными усилиями, даже если вы работаете один.

GitHub Actions — обзор

Если вы уже пользуетесь GitHub, то хорошая новость — система CI/CD уже встроена прямо туда. Называется она **GitHub Actions**.

Основные термины:

- **Workflow** — сценарий, написанный в YAML-файле. Он описывает, что и когда должно происходить.
- **Job** — задача внутри workflow. Например, "проверить тесты".
- **Step** — шаг внутри job. Например, "установить зависимости", "запустить npm test".
- **Action** — готовая команда или скрипт. Например, `actions/checkout` — чтобы получить код репозитория.

Где находятся конфигурации?

В папке `.github/workflows/` вашего репозитория. Вы можете создать столько файлов, сколько нужно. Например, один файл для тестов, другой — для деплоя.

Пример:

```
name: Simple CI

on:
  push:
    branches: [main]

jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - run: echo "Hello, GitHub Actions!"
```

> В данной статье не будем углубляться в код и рассматривать его подробно, об этом поговорим чуть позже, в следующей части.

При каждом пуше в main ветку GitHub запустит пайплайн, и вы увидите результат во вкладке **Actions**.

GitHub также предоставляет Marketplace с тысячами готовых actions: для деплоя, линтеров, публикации на Docker Hub и многого другого.

GitLab CI — обзор

Если вы используете GitLab, то CI/CD тоже встроено — под названием **GitLab CI/CD**. Работает оно немного иначе, но идея та же.

Файл конфигурации:

Один файл `.gitlab-ci.yml` в корне репозитория.

Основные термины:

- **Pipeline** — весь процесс CI/CD.
- **Stage (этап)** — логическая часть пайплайна: build, test, deploy.
- **Job** — конкретная задача в рамках этапа.
- **Script** — список команд, которые выполняются в job-е.

Пример:

```
stages:
  - test

hello_job:
  stage: test
  image: node:16
  script:
    - echo "Hello, GitLab CI!"
```

После коммита GitLab запускает пайплайн. Всё можно отслеживать через интерфейс GitLab — вкладка **CI/CD → Pipelines**.

GitLab CI хорошо интегрирован со своими фичами: issues, merge requests, environments. Также можно использовать как **общие раннеры GitLab**, так и настраивать свои.

Сравнение GitHub Actions и GitLab CI

Параметр	**GitHub Actions**	**GitLab CI**
Файл конфигурации	`.github/workflows/*.yml`	`.gitlab-ci.yml`
Формат YAML	`jobs`, `runs-on`, `steps`	`stages`, `script`, `image`
Триггеры	`on: push/pull_request/...`	`only`, `except`, `rules`
Среда выполнения	Linux, Windows, Mac runners	Чаще Docker-образы (например, `node:14`)

Интерфейс	Вкладка **Actions** в репозитории	Вкладка **CI/CD > Pipelines**
Тарифы	Бесплатно для публичных и приватных репозиториев	Бесплатно до лимита минут

Оба инструмента отлично подходят для автоматизации. Выбор между ними зависит скорее от платформы, на которой вы работаете, и ваших предпочтений.

Примеры использования

CI/CD можно использовать для самых разных проектов:

- **Статический сайт (HTML/CSS/JS):** при пуше автоматически деплоится на GitHub Pages или GitLab Pages.
- **Node.js приложение:** ставим зависимости, запускаем тесты, выкладываем.
- **Python-скрипт:** устанавливаем requirements, запускаем `pytest`, деплоим на сервер.
- **Фронтенд (React, Vue) и бэкенд (Express, Flask):** автоматизируем и сборку, и тесты, и релиз.

Неважно, какой у вас стек — если проект использует git, его можно обернуть в CI/CD.

Терминология

- **CI (Continuous Integration)** — автоматическая сборка и тестирование при каждом коммите.
- **CD (Continuous Delivery/Deployment)** — автоматическая подготовка или публикация релиза.
- **Pipeline** — цепочка шагов: сборка, тесты, деплой.
- **Workflow** — сценарий в GitHub Actions.
- **Job** — отдельная задача: сборка, тесты и т.д.
- **Step** — шаг внутри job-а.
- **Runner** — сервер/контейнер, исполняющий задачи.
- **Stage** — этап в GitLab CI.
- **YAML** — текстовый формат, в котором описывается pipeline.

Итоги

Мы познакомились с основами CI/CD — что это, зачем нужно, и как это устроено. Узнали про два популярных инструмента:

- **GitHub Actions** — CI/CD, встроенное прямо в GitHub.
- **GitLab CI** — мощная система автоматизации в GitLab.

Они позволяют автоматизировать почти любую задачу: от проверки синтаксиса до деплоя приложения.

В следующей статье мы пойдём дальше: **создадим первый workflow и pipeline своими руками**, увидим их в действии и задеплоим сайт на GitHub Pages и GitLab Pages.

Часть 2: Настройка GitHub Actions и GitLab CI – первый workflow и деплой

> В первой статье мы разобрались с основами CI/CD: что это такое, зачем нужно и какие инструменты существуют. Теперь пришло время перейти от теории к практике – создадим наши первые рабочие CI/CD-конвейеры на GitHub Actions и GitLab CI.

> Вот небольшая карта для навигации между частями:
>
> **Часть 1: Основы CI/CD – что это и зачем нужно; GitHub Actions и GitLab CI**
>
> **Часть 2: Настройка GitHub Actions и GitLab CI – первый workflow и деплой**
>
> **^ Вы сейчас здесь ^**
>
> **Часть3: CI/CD – ветки, условия, секреты и окружения**

Введение

Помните, как в первой статье мы говорили о CI/CD как о вашем личном роботе-помощнике? Сегодня мы этого робота соберём и запрограммируем. Мы настроим репозитории на GitHub и GitLab, напишем первые CI/CD-скрипты и проверим их работу.

Не переживайте, если вы никогда раньше не сталкивались с DevOps-инструментами. Все шаги будут описаны настолько подробно, что справится даже новичок. Нам не потребуется глубокое образование или многолетний опыт – только желание автоматизировать рутину и немного терпения.

Даже с нуля вы справитесь, по порядку распишем каждое действие. Это как собирать конструктор по инструкции – сначала может показаться сложным, но когда всё встанет на свои места, вы удивитесь, насколько это просто.

Наш первый workflow будет таким же простым, как программа "Hello, World" – но это уже настоящий шаг вперёд. Вы же помните свою первую программу? Такая простая, но какую гордость вы испытали, когда она заработала. Сегодня вы испытаете то же самое, только с автоматизацией.

Итак, приступим.

Создание репозитория

Прежде чем настраивать CI/CD, нам нужно место, где будет храниться наш код и конфигурации. Таким местом станет репозиторий на GitHub или GitLab. Если у вас уже есть аккаунт и репозиторий – отлично! Если нет – давайте создадим.

На GitHub:

1. Войдите в свой аккаунт на GitHub.
2. В правом верхнем углу нажмите на "+" и выберите "New repository".
3. Дайте репозиторию понятное имя, например, "my-first-cicd".
4. Поставьте галочку "Initialize this repository with a README" – это создаст начальный файл README.md.
5. По желанию можете добавить LICENSE (лицензию) и .gitignore.
6. Нажмите "Create repository".

На GitLab:

1. Войдите в свой аккаунт на GitLab.
2. Нажмите на кнопку "New project" (или "+" в верхнем меню).
3. Выберите "Create blank project".
4. Укажите имя проекта, например, "my-first-cicd".
5. Поставьте галочку "Initialize repository with a README".
6. Нажмите "Create project".

Что такое репозиторий? Это не просто хранилище кода – это ваша мастерская, где будут храниться не только исходники, но и инструкции для нашего "робота" (CI/CD-конфигурации). Представьте репозиторий как рабочий стол мастера: здесь лежат и материалы, и чертежи, и инструменты.

По умолчанию в репозитории создаётся ветка `main` (или `master` в более старых репозиториях). Большинство примеров в этой статье будет привязано именно к этой ветке. Вы можете использовать её по умолчанию для наших экспериментов.

Совет: Если вы предпочитаете работать в cli, можно создать репозиторий вот таким способом:

```
# Создаём папку и инициализируем Git
mkdir my-first-cicd
cd my-first-cicd
git init
echo "# My First CI/CD Project" &gt; README.md
git add README.md
git commit -m "Initial commit"

# Связываем с удалённым репозиторием (замените URL на свой)
git remote add origin https://github.com/username/my-first-cicd.git
git push -u origin main
```

Отлично! Теперь у нас есть место, где будет жить наш код и CI/CD-конфигурации. Переходим к самому интересному – настройке автоматизации.

Первый workflow на GitHub Actions

GitHub Actions – это встроенный в GitHub инструмент для автоматизации. Он позволяет запускать различные задачи при определённых событиях в репозитории. Настройка происходит через YAML-файлы, которые хранятся в специальной папке.

Давайте создадим наш первый workflow:

1. В вашем репозитории создайте папку `.github/workflows/`. Для этого можно использовать веб-интерфейс GitHub:
   - Нажмите "Add file" > "Create new file"
   - В поле имени введите `.github/workflows/ci.yml` (GitHub автоматически создаст папки)
2. В этот файл добавьте следующий код:

```
name: CI for project

on:
  push:
    branches: [ main ]
  pull_request:
    branches: [ main ]

jobs:
```

```
  hello-world:
    runs-on: ubuntu-latest

    steps:
    - name: Checkout code
      uses: actions/checkout@v3

    - name: Say Hello
      run: echo "Hello, GitHub Actions! This is my first workflow!"

    - name: Show date
      run: date
```

3. Нажмите "Commit new file" внизу страницы.

Что мы только что сделали? Давайте разберём этот файл по частям:

- `name: CI for project` – это название нашего workflow, которое будет отображаться в интерфейсе GitHub.
- `on: push: branches: [ main ]` – здесь мы указываем, когда запускать workflow. В данном случае – при каждом push в ветку `main` или при создании pull request в эту ветку.
- `jobs:` – раздел, где описываются задачи, которые нужно выполнить.
- `hello-world:` – название нашего job-а (можно придумать любое).
- `runs-on: ubuntu-latest` – указывает, на какой операционной системе запускать задачу. GitHub предоставляет виртуальные машины с разными ОС.
- `steps:` – последовательность шагов, которые нужно выполнить в рамках job-а.
- `uses: actions/checkout@v3` – этот шаг клонирует ваш репозиторий в виртуальную машину, чтобы последующие шаги могли работать с вашим кодом.
- `run: echo "Hello, GitHub Actions!"` – простая команда, которая выводит текст в консоль.

После того как вы закоммитите этот файл, GitHub автоматически обнаружит его и запустит workflow. Чтобы увидеть результаты:

1. Перейдите на вкладку "Actions" в вашем репозитории.
2. Вы увидите запущенный workflow с названием "CI for project".
3. Кликните на него, чтобы увидеть детали выполнения.
4. Нажмите на job "hello-world", чтобы увидеть вывод каждого шага.

Поздравляю! Вы только что создали и запустили свой первый CI/CD-workflow. Он пока очень простой, но это уже настоящая автоматизация – код запускается автоматически при изменении репозитория.

Важно: YAML очень чувствителен к отступам. Если workflow не запускается или выдаёт ошибку, первым делом проверьте, правильно ли расставлены пробелы в начале строк. В YAML используются именно **пробелы**, а не табуляции.

Первый pipeline на GitLab CI

GitLab CI работает похожим образом, но имеет некоторые отличия в синтаксисе и организации. Давайте создадим наш первый pipeline на GitLab:

1. В вашем GitLab-репозитории создайте файл `.gitlab-ci.yml` в корне проекта:
   - Нажмите "+" > "New file"
   - Назовите файл `.gitlab-ci.yml`
2. Добавьте в файл следующий код:

```
stages:
  - test
  - deploy

hello_job:
  stage: test
  image: alpine:latest
  script:
    - echo "Hello, GitLab CI! This is my first pipeline!"
    - date

# Пока закомментируем этап деплоя, мы вернёмся к нему позже
#deploy_job:
#  stage: deploy
#  script:
#    - echo "This is where we would deploy our application"
#  only:
#    - main
```

3. Нажмите "Commit changes".

Разберём, что мы написали:

- `stages:` – здесь мы определяем этапы выполнения pipeline. Они будут выполняться последовательно: сначала все job-ы из этапа `test`, затем из `deploy`.
- `hello_job:` – название нашего job-а.
- `stage: test` – указывает, к какому этапу относится этот job.
- `image: alpine:latest` – Docker-образ, в котором будет выполняться job. Alpine – это лёгкий Linux-дистрибутив.

- `script:` – команды, которые нужно выполнить в рамках job-а.

После коммита GitLab автоматически обнаружит файл `.gitlab-ci.yml` и запустит pipeline. Чтобы увидеть результаты:

1. Перейдите в раздел "CI/CD" > "Pipelines" в вашем проекте.
2. Вы увидите запущенный pipeline.
3. Кликните на него, чтобы увидеть детали выполнения.
4. Нажмите на job "hello_job", чтобы увидеть вывод каждой команды.

Отлично! Теперь у вас есть работающие CI/CD-конфигурации и на GitHub, и на GitLab. Давайте сделаем что-то более полезное – настроим автоматический деплой простого сайта.

Пример: деплой статического сайта

Статические сайты – идеальный первый проект для CI/CD, потому что их легко развернуть и они не требуют сложной инфраструктуры. И GitHub, и GitLab предоставляют бесплатный хостинг для статических сайтов через GitHub Pages и GitLab Pages соответственно.

Давайте создадим простой HTML-файл и настроим его автоматический деплой при каждом изменении.

Подготовка проекта

1. В корне вашего репозитория создайте файл `index.html`:

```
<!DOCTYPE html>
<html>
<head>
    <title>Мой первый CI/CD проект</title>
    <style>
        body {
            font-family: Arial, sans-serif;
            max-width: 800px;
            margin: 0 auto;
            padding: 20px;
            line-height: 1.6;
        }
        h1 {
            color: #2c3e50;
        }
        .container {
            border: 1px solid #ddd;
```

```
            padding: 20px;
            border-radius: 5px;
            background-color: #f9f9f9;
        }
    </style>
</head>
<body>
    <h1>Привет, CI/CD!</h1>
    <div class="container">
        <p>Эта страница была автоматически развёрнута с помощью CI/CD.</p>
        <p>Время последнего обновления: <span id="timestamp"></span></p>
    </div>

    <script>
        document.getElementById('timestamp').textContent = new Date().toLocaleString();
    </script>
</body>
</html>
```

Деплой на GitHub Pages

Обновите ваш файл `.github/workflows/ci.yml`:

```
name: CI/CD for Static Website

on:
  push:
    branches: [ main ]

jobs:
  deploy:
    runs-on: ubuntu-latest

    steps:
    - name: Checkout code
      uses: actions/checkout@v3

    - name: Deploy to GitHub Pages
      uses: peaceiris/actions-gh-pages@v3
      with:
```

```
        github_token: ${{ secrets.GITHUB_TOKEN }}
        publish_dir: ./
```

Этот workflow использует готовый action `peaceiris/actions-gh-pages`, который автоматизирует деплой на GitHub Pages. `secrets.GITHUB_TOKEN` – это токен, который GitHub автоматически создаёт для каждого workflow, так что вам не нужно настраивать его вручную.

После успешного выполнения workflow:

1. Перейдите в настройки вашего репозитория (Settings).
2. Прокрутите вниз до раздела "GitHub Pages".
3. Убедитесь, что в качестве источника выбрана ветка `gh-pages`.
4. GitHub предоставит URL, по которому доступен ваш сайт (обычно это `https://username.github.io/repository-name/`).

Деплой на GitLab Pages

Обновите ваш файл `.gitlab-ci.yml`:

```
stages:
  - deploy

pages:
  stage: deploy
  script:
    - mkdir -p public
    - cp index.html public/
  artifacts:
    paths:
      - public
  only:
    - main
```

Этот pipeline создаёт папку `public` (это специальное имя для GitLab Pages) и копирует туда наш HTML-файл. Затем он сохраняет эту папку как артефакт, который GitLab автоматически публикует.

После успешного выполнения pipeline:

1. Перейдите в "Settings" > "Pages" вашего проекта.
2. GitLab предоставит URL, по которому доступен ваш сайт (обычно это `https://username.gitlab.io/repository-name/`).

Поздравляю! Теперь у вас есть настоящий CI/CD-конвейер, который автоматически публикует ваш сайт при каждом изменении кода. Это уже не просто "Hello, World!" – это полноценный процесс доставки.

Пример: тестовый скрипт

CI – это не только о деплое, но и о проверке качества кода. Давайте добавим простой тест, который будет проверяться автоматически при каждом изменении.

Для JavaScript-проекта

1. Создайте файл `app.js`:

```
function sum(a, b) {
    return a + b;
}

module.exports = { sum };
```

2. Создайте файл `test.js`:

```
const { sum } = require('./app');

if (sum(2, 3) !== 5) {
    console.error('Test failed: 2 + 3 should equal 5');
    process.exit(1);
}

if (sum(-1, 1) !== 0) {
    console.error('Test failed: -1 + 1 should equal 0');
    process.exit(1);
}

console.log('All tests passed!');
```

3. Создайте файл `package.json`:

```
{
  "name": "my-first-cicd",
  "version": "1.0.0",
  "scripts": {
```

```
    "test": "node test.js"
  }
}
```

Для Python-проекта

1. Создайте файл `app.py`:

```
def sum(a, b):
    return a + b
```

2. Создайте файл `test.py`:

```
from app import sum

def test_sum():
    assert sum(2, 3) == 5, "2 + 3 should equal 5"
    assert sum(-1, 1) == 0, "-1 + 1 should equal 0"
    print("All tests passed!")

if __name__ == "__main__":
    test_sum()
```

Обновление GitHub Actions workflow

Обновите ваш файл `.github/workflows/ci.yml` для JavaScript:

```
name: CI/CD for Project

on:
  push:
    branches: [ main ]
  pull_request:
    branches: [ main ]

jobs:
  test:
    runs-on: ubuntu-latest

    steps:
    - name: Checkout code
```

```
      uses: actions/checkout@v3

    - name: Setup Node.js
      uses: actions/setup-node@v3
      with:
        node-version: '16'

    - name: Run tests
      run: npm test

  deploy:
    needs: test
    runs-on: ubuntu-latest
    if: github.ref == 'refs/heads/main'

    steps:
    - name: Checkout code
      uses: actions/checkout@v3

    - name: Deploy to GitHub Pages
      uses: peaceiris/actions-gh-pages@v3
      with:
        github_token: ${{ secrets.GITHUB_TOKEN }}
        publish_dir: ./
```

Или для Python:

```
name: CI/CD for Project

on:
  push:
    branches: [ main ]
  pull_request:
    branches: [ main ]

jobs:
  test:
    runs-on: ubuntu-latest

    steps:
```

```
    - name: Checkout code
      uses: actions/checkout@v3

    - name: Setup Python
      uses: actions/setup-python@v4
      with:
        python-version: '3.10'

    - name: Run tests
      run: python test.py

  deploy:
    needs: test
    runs-on: ubuntu-latest
    if: github.ref == 'refs/heads/main'

    steps:
    - name: Checkout code
      uses: actions/checkout@v3

    - name: Deploy to GitHub Pages
      uses: peaceiris/actions-gh-pages@v3
      with:
        github_token: ${{ secrets.GITHUB_TOKEN }}
        publish_dir: ./
```

Обновление GitLab CI pipeline

Обновите ваш файл `.gitlab-ci.yml` для JavaScript:

```
stages:
  - test
  - deploy

test:
  stage: test
  image: node:16
  script:
    - npm test
```

```
pages:
  stage: deploy
  script:
    - mkdir -p public
    - cp index.html public/
  artifacts:
    paths:
      - public
  only:
    - main
  needs:
    - test
```

Или для Python:

```
stages:
  - test
  - deploy

test:
  stage: test
  image: python:3.10
  script:
    - python test.py

pages:
  stage: deploy
  script:
    - mkdir -p public
    - cp index.html public/
  artifacts:
    paths:
      - public
  only:
    - main
  needs:
    - test
```

Теперь наш CI/CD-процесс стал ещё более полноценным:

1. При каждом изменении кода автоматически запускаются тесты.

2. Если тесты проходят успешно, код автоматически публикуется.
3. Если тесты не проходят, деплой не выполняется – это защищает нас от публикации сломанного кода.

Это базовый, но уже вполне рабочий CI/CD-конвейер. В реальных проектах тесты будут более сложными, но принцип остаётся тем же: автоматическая проверка → автоматический деплой.

Полезные советы и выводы

Теперь, когда у вас есть работающие CI/CD-конфигурации, вот несколько полезных советов:

1. Проверяйте статус

После каждого пуша убедитесь, что ваш workflow или pipeline действительно запускается:

- На GitHub: вкладка "Actions"
- На GitLab: "CI/CD" > "Pipelines"

Зелёный статус означает, что всё прошло успешно. Красный – что-то пошло не так.

2. Читайте логи

Если что-то не работает, логи – ваш лучший друг. Они покажут, на каком именно шаге произошла ошибка и что пошло не так. Кликните на job, чтобы увидеть подробный вывод каждой команды.

3. Быстро правьте

Одно из преимуществ CI/CD – быстрая обратная связь. Если вы видите ошибку, исправьте её и снова отправьте изменения. Через несколько минут вы узнаете, помогло ли исправление.

4. Экспериментируйте

Не бойтесь экспериментировать! Специально сломайте тест, чтобы увидеть, как система отреагирует. Добавьте новые шаги в workflow. Чем больше вы экспериментируете, тем лучше понимаете, как всё работает.

5. Используйте готовые actions и templates

И GitHub, и GitLab предлагают множество готовых actions и templates для типичных задач. Не изобретайте велосипед – ищите готовые решения в маркетплейсе GitHub Actions или в документации GitLab CI.

6. Постепенно усложняйте

Начните с простого, как мы сделали в этой статье, и постепенно добавляйте новые возможности: больше тестов, статический анализ кода, автоматическую сборку более сложных приложений.

Поздравляю, вы только что построили конвейер! Ну почти, но уже уверенно движетесь в правильном направлении. Теперь у вас есть автоматизированный процесс, который проверяет и публикует ваш код без ручного вмешательства. Это уже настоящий DevOps!

Терминология

В мире CI/CD используется множество специфических терминов. Вот краткий словарик, который поможет вам лучше понимать документацию и обсуждения:

- **Workflow** – сценарий автоматизации, описанный в YAML-файле на GitHub Actions.
- **Job** – отдельная задача в workflow или pipeline, которая выполняется на одном runner-е.
- **Runner** – виртуальная машина или контейнер, на котором исполняются задачи.
- **Step** – отдельный шаг внутри job-а, например, команда или action.
- **YAML** – формат файла, используемый для описания workflow и pipeline. Отступы в нём критически важны!
- **Script** – раздел с командами в GitLab CI-файле.
- **Action** – готовое действие на GitHub, которое можно использовать в своих workflow, например, `actions/checkout`.
- **Stage** – этап pipeline в GitLab, например, build, test, deploy.
- **Artifact** – файл или набор файлов, созданный во время выполнения job-а и сохранённый для использования в других job-ах или для скачивания.
- **Environment** – среда, в которую выполняется деплой, например, staging или production.

Не переживайте, если сразу не запомните все термины – они станут понятнее по мере практики.

Итоги

В этой статье мы:

1. Создали репозитории на GitHub и GitLab.
2. Настроили первые workflow и pipeline.
3. Автоматизировали деплой статического сайта.
4. Добавили автоматические тесты.
5. Построили полноценный CI/CD-конвейер, который проверяет и публикует код при каждом изменении.

Теперь при каждом пуше в ветку `main` автоматически выполняются тесты, и если они проходят успешно, сайт обновляется. Команде больше не нужно вручную запускать

проверку – теперь ошибки видны сразу, а деплой происходит автоматически.

Это только начало вашего путешествия в мир CI/CD. В следующей статье мы рассмотрим более сложные сценарии: работу с разными ветками, условное выполнение шагов, использование секретов для безопасного хранения чувствительных данных и лучшие практики настройки CI/CD для реальных проектов.

А пока – экспериментируйте с тем, что вы узнали сегодня. Добавьте больше тестов, попробуйте другие actions, настройте деплой для своего проекта. Практика – лучший способ закрепить знания.

Часть 3: CI/CD – ветки, условия, секреты и окружения

Введение

Если вы дошли до этой части в серии, то вы уже знакомы с базовыми принципами CI/CD и даже настроили свой первый простой пайплайн. Поздравляю! Вы сделали важный шаг в мир автоматизации разработки. Но как и в программировании, где после "Hello World" начинается настоящее обучение, в CI/CD после базовой настройки открывается целый мир возможностей.

> Если вы попали сюда случайно, или не видели первых двух частей, то рекомендую ознакомиться:
>
> **Часть 1: Основы CI/CD – что это и зачем нужно; GitHub Actions и GitLab CI**
>
> **Часть 2: Настройка GitHub Actions и GitLab CI – первый workflow и деплой**
>
> **Часть3: CI/CD – ветки, условия, секреты и окружения**
>
> **^ Вы сейчас здесь ^**

В этой статье мы поговорим о том, как сделать ваш CI/CD по-настоящему гибким и мощным. Мы рассмотрим:

- **Ветвление и условия** — как заставить пайплайн вести себя по-разному в зависимости от ветки или других условий
- **Секреты и переменные окружения** — как безопасно хранить и использовать чувствительные данные

- **Окружения и этапы** — как организовать многоступенчатый процесс с разными средами выполнения
- **Отладку и разбор ошибок** — как понять, что пошло не так, и быстро это исправить

Эти темы критически важны для любого реального проекта. Согласитесь, в продакшн вы хотите деплоить только проверенный код из основной ветки, а не каждый экспериментальный коммит. Для доступа к серверам вам нужны пароли и токены, которые нельзя хранить в открытом виде. А для полноценного процесса вам нужны разные окружения — разработка, тестирование, продакшн.

Не волнуйтесь, если некоторые концепции кажутся сложными. Мы разберём всё пошагово, с примерами и объяснениями. К концу статьи вы будете уверенно настраивать продвинутые CI/CD-конфигурации и сможете автоматизировать практически любой процесс разработки.

Итак, пристегните ремни — мы отправляемся в путешествие по миру продвинутого CI/CD, где ваш код сам себя проверяет, тестирует и деплоит именно так, как вам нужно. И помните: мы превращаем наш "Hello World" workflow во взрослую программу, которая умеет принимать решения, хранить секреты и работать в разных окружениях.

Ветвление и условия в workflow

Концепция ветвления в Git и CI/CD

Если вы когда-нибудь работали с Git, то знаете, что ветки — это одна из его самых мощных возможностей. Они позволяют разработчикам работать параллельно над разными задачами, не мешая друг другу. Типичный проект может иметь десятки активных веток одновременно:

- `main` (или `master`) — стабильная версия, готовая к использованию
- `develop` — ветка разработки, куда сливаются все новые функции
- `feature/*` — ветки для отдельных функций
- `hotfix/*` — срочные исправления критических ошибок
- `release/*` — подготовка к выпуску новой версии

Но что это значит для CI/CD? А то, что не все ветки равны! Представьте, что у вас есть магазин одежды. Вы же не будете выставлять на продажу каждый незавершённый эскиз от дизайнера? Точно так же не стоит деплоить в продакшн каждый коммит из каждой ветки.

Вот где на сцену выходит **условное выполнение** в CI/CD. Оно позволяет настроить разное поведение пайплайна в зависимости от ветки или других условий:

- Для веток `feature/*` — только сборка и базовые тесты
- Для ветки `develop` — полное тестирование и деплой на тестовый сервер
- Для ветки `main` — полное тестирование и деплой на продакшн

Это как если бы у вас был умный конвейер на фабрике, который автоматически определяет, что делать с каждым изделием в зависимости от его маркировки.

Реализация в GitHub Actions

В GitHub Actions условное выполнение можно настроить на двух уровнях:

1. **На уровне workflow** — когда запускать весь пайплайн
2. **На уровне job или step** — какие задачи выполнять внутри запущенного пайплайна

Фильтрация на уровне workflow

Чтобы запускать workflow только для определённых веток, используйте секцию `on`:

```
name: CI Pipeline

on:
  push:
    branches: [ main, develop, 'feature/**' ]
  pull_request:
    branches: [ main, develop ]
```

Этот код говорит: "Запускай пайплайн при пуше в ветки main, develop или любую ветку, начинающуюся с feature/, а также при создании PR в main или develop".

Обратите внимание на синтаксис `'feature/**'` — это шаблон (pattern), который соответствует всем веткам, начинающимся с `feature/`. Звёздочки здесь работают как подстановочные знаки.

Условия внутри workflow

Для более гибкого контроля используйте условие `if` на уровне job или step:

```
jobs:
  test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - name: Run tests
        run: npm test

  deploy-to-staging:
    runs-on: ubuntu-latest
```

```
    needs: test
    if: github.ref == 'refs/heads/develop'
    steps:
      - uses: actions/checkout@v3
      - name: Deploy to staging
        run: ./deploy.sh staging

  deploy-to-production:
    runs-on: ubuntu-latest
    needs: test
    if: github.ref == 'refs/heads/main'
    steps:
      - uses: actions/checkout@v3
      - name: Deploy to production
        run: ./deploy.sh production
```

В этом примере тесты запускаются для всех веток, но деплой на staging происходит только для ветки `develop`, а деплой на production — только для ветки `main`.

Выражение `github.ref == 'refs/heads/main'` проверяет, является ли текущая ветка main. GitHub Actions предоставляет множество контекстных переменных, которые можно использовать в условиях:

- `github.ref` — полное имя ссылки (например, `refs/heads/main` или `refs/tags/v1.0.0`)
- `github.event_name` — тип события (например, `push`, `pull_request`)
- `github.actor` — пользователь, инициировавший событие
- и многие другие

Вы можете комбинировать условия с помощью операторов `&&` (И) и `||` (ИЛИ):

```
if: github.ref == 'refs/heads/main' && github.event_name == 'push'
```

Это условие будет истинным только для пушей в ветку main.

Реализация в GitLab CI

GitLab CI предлагает похожие, но немного отличающиеся механизмы для условного выполнения.

Простой способ: only/except

Традиционный способ фильтрации в GitLab CI — использование ключевых слов `only` и `except`:

```
stages:
  - test
  - deploy

test:
  stage: test
  script:
    - npm test

deploy-staging:
  stage: deploy
  script:
    - ./deploy.sh staging
  only:
    - develop

deploy-production:
  stage: deploy
  script:
    - ./deploy.sh production
  only:
    - main
```

Здесь `only: - develop` означает "выполнять только для ветки develop". Аналогично, `except` указывает, для каких веток задачу выполнять НЕ нужно.

Продвинутый способ: rules

Более новый и гибкий подход — использование `rules`:

```
deploy-staging:
  stage: deploy
  script:
    - ./deploy.sh staging
  rules:
    - if: $CI_COMMIT_BRANCH == "develop"
      when: on_success
    - when: never
```

Секция `rules` содержит список правил, которые проверяются по порядку. Первое совпавшее правило определяет, будет ли выполняться задача. В этом примере задача выполнится, если

ветка — `develop` и предыдущие шаги успешны. В противном случае сработает правило `when: never`, и задача будет пропущена.

GitLab CI предоставляет множество предопределённых переменных:

- `$CI_COMMIT_BRANCH` — имя ветки
- `$CI_PIPELINE_SOURCE` — источник запуска пайплайна (например, `push`, `merge_request_event`)
- `$CI_COMMIT_TAG` — имя тега (если есть)
- и многие другие

Вы можете создавать сложные условия, комбинируя эти переменные:

```
rules:
  - if: $CI_COMMIT_BRANCH == "main" && $CI_PIPELINE_SOURCE == "push"
    when: on_success
  - if: $CI_COMMIT_BRANCH =~ /^feature\/.*/
    when: manual
  - when: never
```

Это правило говорит: "Выполнять автоматически для пушей в main, требовать ручного запуска для веток `feature/*`, и пропускать для всех остальных случаев".

Практический пример полного workflow

Давайте рассмотрим более полный пример, который демонстрирует мощь условного выполнения. Представим, что у нас есть веб-приложение, и мы хотим настроить следующий процесс:

1. Для всех веток — сборка и базовые тесты
2. Для веток `feature/*` — дополнительно линтинг и проверка типов
3. Для ветки `develop` — всё вышеперечисленное + e2e-тесты + деплой на тестовый сервер
4. Для ветки `main` — всё вышеперечисленное + деплой на продакшн (с ручным подтверждением)

GitHub Actions

```
name: CI/CD Pipeline

on:
  push:
    branches: [ main, develop, 'feature/**' ]
  pull_request:
```

```
    branches: [ main, develop ]

jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - name: Set up Node.js
        uses: actions/setup-node@v3
        with:
          node-version: '16'
      - name: Install dependencies
        run: npm ci
      - name: Build
        run: npm run build
      - name: Run basic tests
        run: npm test

  lint-and-typecheck:
    runs-on: ubuntu-latest
    if: startsWith(github.ref, 'refs/heads/feature/')
    steps:
      - uses: actions/checkout@v3
      - name: Set up Node.js
        uses: actions/setup-node@v3
        with:
          node-version: '16'
      - name: Install dependencies
        run: npm ci
      - name: Lint
        run: npm run lint
      - name: Type check
        run: npm run typecheck

  e2e-tests:
    runs-on: ubuntu-latest
    needs: [build]
    if: github.ref == 'refs/heads/develop' || github.ref == 'refs/heads/main'
    steps:
      - uses: actions/checkout@v3
```

```
      - name: Set up Node.js
        uses: actions/setup-node@v3
        with:
          node-version: '16'
      - name: Install dependencies
        run: npm ci
      - name: Run E2E tests
        run: npm run test:e2e

  deploy-staging:
    runs-on: ubuntu-latest
    needs: [build, e2e-tests]
    if: github.ref == 'refs/heads/develop'
    steps:
      - uses: actions/checkout@v3
      - name: Deploy to staging
        run: |
          echo "Deploying to staging environment..."
          # Здесь команды для деплоя на тестовый сервер

  deploy-production:
    runs-on: ubuntu-latest
    needs: [build, e2e-tests]
    if: github.ref == 'refs/heads/main'
    environment:
      name: production
      url: https://example.com
    steps:
      - uses: actions/checkout@v3
      - name: Deploy to production
        run: |
          echo "Deploying to production environment..."
          # Здесь команды для деплоя на продакшн
```

GitLab CI

```
stages:
  - build
  - test
  - deploy
```

```
build:
  stage: build
  script:
    - npm ci
    - npm run build
  artifacts:
    paths:
      - dist/

basic-tests:
  stage: test
  script:
    - npm test

lint-and-typecheck:
  stage: test
  script:
    - npm run lint
    - npm run typecheck
  rules:
    - if: $CI_COMMIT_BRANCH =~ /^feature\/.*/
      when: on_success
    - when: never

e2e-tests:
  stage: test
  script:
    - npm run test:e2e
  rules:
    - if: $CI_COMMIT_BRANCH == "develop" || $CI_COMMIT_BRANCH == "main"
      when: on_success
    - when: never

deploy-staging:
  stage: deploy
  script:
    - echo "Deploying to staging environment..."
    # Здесь команды для деплоя на тестовый сервер
  environment:
```

```
    name: staging
    url: https://staging.example.com
  rules:
    - if: $CI_COMMIT_BRANCH == "develop"
      when: on_success
    - when: never

deploy-production:
  stage: deploy
  script:
    - echo "Deploying to production environment..."
    # Здесь команды для деплоя на продакшн
  environment:
    name: production
    url: https://example.com
  rules:
    - if: $CI_COMMIT_BRANCH == "main"
      when: manual
    - when: never
```

Обратите внимание, как в GitLab CI мы используем `when: manual` для деплоя на продакшн, что требует ручного подтверждения.

Визуализация процесса принятия решений

Чтобы лучше понять, как работает условное выполнение, представьте его как блок-схему:

1. Событие (пуш, PR) → Запуск workflow
2. Проверка ветки:
   - Если `feature/*` → Сборка + Базовые тесты + Линтинг
   - Если `develop` → Сборка + Все тесты + Деплой на тестовый сервер
   - Если `main` → Сборка + Все тесты + Деплой на продакшн (с подтверждением)

Такой подход позволяет автоматизировать процесс разработки, при этом обеспечивая необходимый уровень контроля и безопасности.

Веток в Git-репозитории может быть больше, чем веток на новогодней ёлке — CI/CD помогает упорядочить их и обеспечить правильную обработку каждой из них, позволяет точно определить, какие ветки и изменения можно запускать, а какие — должны пройти ручную проверку.

Секреты и переменные окружения

Безопасность в CI/CD

Представьте, что вы строите дом. Вы же не оставите ключи от входной двери под ковриком, где их может найти любой прохожий? Точно так же в мире CI/CD есть данные, которые нельзя хранить в открытом виде: пароли от серверов, токены доступа к API, приватные ключи для подписи кода и многое другое.

Проблема в том, что конфигурационные файлы CI/CD (`.github/workflows/*.yml` или `.gitlab-ci.yml`) хранятся в репозитории, который может быть публичным или к которому имеет доступ множество людей. Если вы напрямую запишете пароль в такой файл, он станет доступен всем, кто имеет доступ к репозиторию. Это всё равно что написать свой PIN-код на банковской карте.

Вот типичные виды чувствительных данных, которые требуют защиты:

- **Пароли** — для доступа к серверам, базам данных и т.д.
- **API-токены** — для взаимодействия с внешними сервисами
- **SSH-ключи** — для безопасного подключения к серверам
- **Сертификаты** — для подписи кода или установки защищённых соединений
- **Учётные данные облачных провайдеров** — для управления ресурсами в AWS, Azure, GCP и т.д.

К счастью, и GitHub Actions, и GitLab CI предоставляют механизмы для безопасного хранения и использования таких данных — **секреты** и **переменные окружения**.

Управление секретами в GitHub Actions

GitHub Actions позволяет хранить секреты на трёх уровнях:

1. **Уровень репозитория** — секреты доступны только в конкретном репозитории
2. **Уровень организации** — секреты доступны во всех репозиториях организации
3. **Уровень окружения** — секреты доступны только в определённом окружении (например, production)

Добавление секрета на уровне репозитория

1. Перейдите в свой репозиторий на GitHub
2. Нажмите на вкладку "Settings"
3. В боковом меню выберите "Secrets and variables" → "Actions"
4. Нажмите "New repository secret"
5. Введите имя секрета (например, `DATABASE_PASSWORD`) и его значение
6. Нажмите "Add secret"

Использование секретов в workflow

После добавления секрета вы можете использовать его в своём workflow через синтаксис `${{ secrets.SECRET_NAME }}`:

```
jobs:
  deploy:
    runs-on: ubuntu-latest
    steps:
      - name: Deploy to server
        run: |
          sshpass -p ${{ secrets.SERVER_PASSWORD }} ssh user@example.com "cd /var/www && git pull"
```

Важно понимать, что GitHub автоматически маскирует секреты в логах. Если секрет случайно попадёт в вывод команды, GitHub заменит его звёздочками. Однако это не работает, если вы намеренно выводите секрет или его часть, например, через `echo ${{ secrets.API_TOKEN }}`.

Секреты на уровне организации

Если у вас есть организация на GitHub и несколько репозиториев используют одни и те же секреты, удобнее настроить их на уровне организации:

1. Перейдите на страницу вашей организации
2. Нажмите на вкладку "Settings"
3. В боковом меню выберите "Secrets and variables" → "Actions"
4. Нажмите "New organization secret"
5. Введите имя и значение секрета
6. Выберите, к каким репозиториям применить этот секрет
7. Нажмите "Add secret"

Секреты на уровне окружения

Для особо чувствительных данных, таких как ключи от продакшн-серверов, лучше использовать секреты на уровне окружения:

1. В репозитории перейдите в "Settings" → "Environments"
2. Создайте новое окружение или выберите существующее
3. Нажмите "Add secret" и добавьте секрет
4. В workflow укажите окружение для job-а:

```
jobs:
  deploy-production:
    runs-on: ubuntu-latest
    environment: production
```

```
    steps:
      - name: Deploy to production
        run: ./deploy.sh ${{ secrets.PRODUCTION_API_KEY }}
```

Преимущество этого подхода в том, что вы можете настроить дополнительные защитные меры для окружения, например, требовать ручное одобрение перед выполнением.

Управление секретами в GitLab CI

GitLab CI использует термин "переменные" (variables) вместо "секреты", но концепция та же. Переменные могут быть защищёнными (protected) и/или маскированными (masked).

- **Защищённые переменные** доступны только в защищённых ветках и тегах
- **Маскированные переменные** автоматически скрываются в логах

Добавление переменных на уровне проекта

1. Перейдите в свой проект на GitLab
2. Нажмите на "Settings" → "CI/CD"
3. Разверните секцию "Variables"
4. Нажмите "Add variable"
5. Введите ключ (например, `DATABASE_PASSWORD`) и значение
6. При необходимости отметьте "Protect variable" и/или "Mask variable"
7. Нажмите "Add variable"

Использование переменных в pipeline

После добавления переменной вы можете использовать её в своём `.gitlab-ci.yml`:

```
deploy:
  stage: deploy
  script:
    - sshpass -p $DATABASE_PASSWORD ssh user@example.com "cd /var/www && git pull"
```

Обратите внимание, что в GitLab CI переменные используются напрямую через `$VARIABLE_NAME`, без дополнительного синтаксиса.

Переменные на уровне группы

Если у вас есть группа проектов в GitLab, вы можете настроить переменные на уровне группы:

1. Перейдите на страницу вашей группы
2. Нажмите на "Settings" → "CI/CD"
3. Разверните секцию "Variables"

4. Добавьте переменные так же, как на уровне проекта

Переменные группы будут доступны во всех проектах этой группы, но их можно переопределить на уровне проекта.

Типы переменных в GitLab CI

GitLab CI поддерживает два типа переменных:

- **Переменные среды (environment variables)** — доступны как переменные окружения в скриптах
- **Файловые переменные (file variables)** — сохраняются как файлы в директории сборки

Файловые переменные особенно полезны для хранения сертификатов, ключей SSH и других данных, которые обычно хранятся в файлах:

```
deploy:
  stage: deploy
  script:
    - ssh -i $SSH_PRIVATE_KEY_FILE user@example.com "cd /var/www && git pull"
```

Практические примеры использования секретов

Давайте рассмотрим несколько реальных сценариев использования секретов в CI/CD.

Пример 1: Деплой на сервер через SSH

GitHub Actions:

```
jobs:
  deploy:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3

      - name: Install SSH key
        uses: shimataro/ssh-key-action@v2
        with:
          key: ${{ secrets.SSH_PRIVATE_KEY }}
          known_hosts: ${{ secrets.KNOWN_HOSTS }}

      - name: Deploy to server
```

```
        run: |
          ssh user@example.com "cd /var/www && git pull && npm ci && npm run build"
```

GitLab CI:

```
deploy:
  stage: deploy
  before_script:
    - 'which ssh-agent || ( apt-get update -y && apt-get install openssh-client -y )'
    - eval $(ssh-agent -s)
    - echo "$SSH_PRIVATE_KEY" | tr -d '\r' | ssh-add -
    - mkdir -p ~/.ssh
    - chmod 700 ~/.ssh
    - echo "$KNOWN_HOSTS" > ~/.ssh/known_hosts
    - chmod 644 ~/.ssh/known_hosts
  script:
    - ssh user@example.com "cd /var/www && git pull && npm ci && npm run build"
```

Пример 2: Публикация пакета в npm

GitHub Actions:

```
jobs:
  publish:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - uses: actions/setup-node@v3
        with:
          node-version: '16'
          registry-url: 'https://registry.npmjs.org'
      - run: npm ci
      - run: npm publish
        env:
          NODE_AUTH_TOKEN: ${{ secrets.NPM_TOKEN }}
```

GitLab CI:

```
publish:
  stage: deploy
  script:
```

```
    - echo "//registry.npmjs.org/:_authToken=${NPM_TOKEN}" > .npmrc
    - npm ci
    - npm publish
```

Пример 3: Деплой в облачный сервис (AWS)

GitHub Actions:

```
jobs:
  deploy:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3

      - name: Configure AWS credentials
        uses: aws-actions/configure-aws-credentials@v1
        with:
          aws-access-key-id: ${{ secrets.AWS_ACCESS_KEY_ID }}
          aws-secret-access-key: ${{ secrets.AWS_SECRET_ACCESS_KEY }}
          aws-region: us-east-1

      - name: Deploy to S3
        run: aws s3 sync ./dist s3://my-bucket/
```

GitLab CI:

```
deploy:
  stage: deploy
  image: amazon/aws-cli
  script:
    - aws configure set aws_access_key_id $AWS_ACCESS_KEY_ID
    - aws configure set aws_secret_access_key $AWS_SECRET_ACCESS_KEY
    - aws configure set region us-east-1
    - aws s3 sync ./dist s3://my-bucket/
```

Лучшие практики работы с секретами

1. **Минимизируйте доступ** — предоставляйте минимально необходимые права. Например, для деплоя на сервер используйте отдельного пользователя с ограниченными правами.

2. **Регулярно обновляйте секреты** — периодически меняйте пароли и токены, особенно если в команде были кадровые изменения.
3. **Используйте временные токены** — по возможности используйте токены с ограниченным сроком действия.
4. **Проверяйте логи** — убедитесь, что секреты не попадают в логи. Даже с автоматической маскировкой могут быть утечки.
5. **Не хардкодьте секреты** — никогда не записывайте секреты напрямую в код или конфигурационные файлы.
6. **Используйте разные секреты для разных окружений** — не используйте один и тот же токен для тестового и продакшн-окружений.
7. **Ограничьте видимость секретов** — используйте секреты на уровне окружений или с ограничением доступа к определённым веткам.

Секреты в CI/CD требуют такой же осторожности, как пароли или банковские данные — используйте механизмы защиты и избегайте попадания в открытый код. Правильная работа с секретами — это не просто вопрос удобства, это вопрос безопасности вашего проекта и данных ваших пользователей.

Окружения и этапы (стейджи)

Концепция окружений в разработке

Представьте, что вы шеф-повар, разрабатывающий новое блюдо. Вы же не будете сразу подавать его посетителям ресторана, верно? Сначала вы экспериментируете на кухне, затем даёте попробовать коллегам, и только после всех корректировок и одобрений блюдо попадает в меню.

В разработке программного обеспечения работает тот же принцип. **Окружения** (environments) — это различные среды, через которые проходит ваш код на пути от разработчика к конечному пользователю:

- **Development (разработка)** — здесь разработчики экспериментируют и создают новые функции
- **Testing/QA (тестирование)** — здесь тестировщики проверяют функциональность и ищут ошибки
- **Staging (предпродакшн)** — максимально приближенная к продакшну среда для финальных проверок
- **Production (продакшн)** — реальная среда, с которой взаимодействуют пользователи

Каждое окружение имеет свои особенности:

- Разные серверы или облачные ресурсы
- Разные базы данных (часто с тестовыми данными в непродакшн-окружениях)
- Разные уровни доступа и безопасности

- Разные конфигурации (например, более подробное логирование в dev)

Этапы (stages) — это последовательные шаги в процессе CI/CD, которые выполняются в определённом порядке:

1. **Build (сборка)** — компиляция кода, сборка артефактов
2. **Test (тестирование)** — запуск автоматических тестов
3. **Deploy (развёртывание)** — выкладка кода на серверы

Этапы и окружения тесно связаны: на этапе деплоя код может быть развёрнут в разные окружения в зависимости от условий.

Зачем это нужно? Представьте, что вы обнаружили критическую ошибку в продакшне. Что лучше:

1. Исправить её напрямую на продакшн-сервере, рискуя внести новые проблемы?
2. Исправить в коде, проверить на тестовом окружении, и только потом выкатить на продакшн?

Очевидно, второй вариант безопаснее. Правильно настроенные окружения и этапы в CI/CD помогают минимизировать риски и обеспечивают плавный процесс доставки кода от разработчика к пользователю.

Настройка окружений в GitHub Actions

GitHub Actions предлагает встроенную поддержку окружений, которая позволяет не только организовать деплои, но и добавить защитные правила.

Создание окружений

Окружения создаются в настройках репозитория:

1. Перейдите в репозиторий на GitHub
2. Нажмите на вкладку "Settings"
3. В боковом меню выберите "Environments"
4. Нажмите "New environment"
5. Введите имя окружения (например, "production")
6. Настройте защитные правила (при необходимости)
7. Нажмите "Configure environment"

Защитные правила для окружений

GitHub позволяет настроить несколько типов защиты:

- **Required reviewers (обязательные проверяющие)** — люди, которые должны одобрить деплой
- **Wait timer (таймер ожидания)** — задержка перед деплоем (например, 10 минут)

- **Deployment branches (ветки для деплоя)** — ограничение веток, из которых можно деплоить
- **Environment secrets (секреты окружения)** — секреты, доступные только в этом окружении

Использование окружений в workflow

После создания окружений вы можете использовать их в своём workflow:

```
name: Deploy

on:
  push:
    branches: [ main ]

jobs:
  deploy-to-staging:
    runs-on: ubuntu-latest
    environment:
      name: staging
      url: https://staging.example.com
    steps:
      - uses: actions/checkout@v3
      - name: Deploy to Staging
        run: ./deploy.sh staging

  deploy-to-production:
    needs: deploy-to-staging
    runs-on: ubuntu-latest
    environment:
      name: production
      url: https://example.com
    steps:
      - uses: actions/checkout@v3
      - name: Deploy to Production
        run: ./deploy.sh production
```

В этом примере:

- Деплой сначала идёт на staging, а затем на production (благодаря `needs: deploy-to-staging`)

- Для каждого окружения указан URL, который будет отображаться в интерфейсе GitHub
- Если для production настроены защитные правила (например, обязательное одобрение), workflow будет ждать этого одобрения перед деплоем

Переменные окружения

Помимо секретов, вы можете использовать обычные переменные окружения:

```
jobs:
  deploy:
    runs-on: ubuntu-latest
    environment: production
    env:
      NODE_ENV: production
      API_URL: https://api.example.com
    steps:
      - uses: actions/checkout@v3
      - name: Deploy
        run: |
          echo "Deploying with NODE_ENV=$NODE_ENV"
          ./deploy.sh
```

Переменные `env` доступны всем шагам в job-е как обычные переменные окружения.

Настройка окружений в GitLab CI

GitLab CI имеет мощную поддержку как этапов (stages), так и окружений (environments).

Определение этапов

Этапы в GitLab CI определяются в начале файла `.gitlab-ci.yml`:

```
stages:
  - build
  - test
  - deploy
```

Это задаёт порядок выполнения: сначала все job-ы этапа build, затем test, и наконец deploy.

Определение окружений

Окружения определяются внутри job-ов:

```
deploy-staging:
  stage: deploy
  script:
    - ./deploy.sh staging
  environment:
    name: staging
    url: https://staging.example.com
```

Ключ `environment` указывает, что этот job связан с окружением staging и предоставляет URL для доступа к нему.

Динамические окружения

Одна из мощных возможностей GitLab CI — динамические окружения, которые создаются автоматически, например, для каждой ветки или merge request:

```
deploy-review:
  stage: deploy
  script:
    - ./deploy.sh review/$CI_COMMIT_REF_SLUG
  environment:
    name: review/$CI_COMMIT_REF_SLUG
    url: https://$CI_COMMIT_REF_SLUG.review.example.com
    on_stop: stop-review
  only:
    - branches
  except:
    - main

stop-review:
  stage: deploy
  script:
    - ./cleanup.sh review/$CI_COMMIT_REF_SLUG
  environment:
    name: review/$CI_COMMIT_REF_SLUG
    action: stop
  when: manual
  only:
    - branches
  except:
    - main
```

В этом примере:

- Для каждой ветки создаётся отдельное окружение `review/имя-ветки`
- URL формируется динамически на основе имени ветки
- Добавлен job `stop-review` для удаления окружения, когда оно больше не нужно

Защита окружений

GitLab позволяет защитить окружения, ограничив, кто может деплоить в них:

1. Перейдите в проект на GitLab
2. Нажмите на "Settings" → "CI/CD"
3. Разверните секцию "Protected environments"
4. Нажмите "Protect an environment"
5. Выберите окружение и укажите, кто может деплоить в него

Ручное одобрение

Для критических окружений, таких как production, часто требуется ручное одобрение:

```
deploy-production:
  stage: deploy
  script:
    - ./deploy.sh production
  environment:
    name: production
    url: https://example.com
  when: manual
  only:
    - main
```

Параметр `when: manual` означает, что job не будет запущен автоматически — требуется ручной запуск через интерфейс GitLab.

Практический пример многоэтапного pipeline

Давайте рассмотрим более полный пример, который демонстрирует использование этапов и окружений в реальном проекте.

GitHub Actions

```
name: CI/CD Pipeline


on:
```

```yaml
  push:
    branches: [ main, develop ]
  pull_request:
    branches: [ main, develop ]

jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - name: Set up Node.js
        uses: actions/setup-node@v3
        with:
          node-version: '16'
      - name: Install dependencies
        run: npm ci
      - name: Build
        run: npm run build
      - name: Upload build artifacts
        uses: actions/upload-artifact@v3
        with:
          name: build-files
          path: dist/

  test:
    needs: build
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - name: Set up Node.js
        uses: actions/setup-node@v3
        with:
          node-version: '16'
      - name: Install dependencies
        run: npm ci
      - name: Download build artifacts
        uses: actions/download-artifact@v3
        with:
          name: build-files
          path: dist/
```

```yaml
    - name: Run tests
      run: npm test

  deploy-dev:
    if: github.ref == 'refs/heads/develop'
    needs: test
    runs-on: ubuntu-latest
    environment:
      name: development
      url: https://dev.example.com
    steps:
      - uses: actions/checkout@v3
      - name: Download build artifacts
        uses: actions/download-artifact@v3
        with:
          name: build-files
          path: dist/
      - name: Deploy to development
        run: |
          echo "Deploying to development environment..."
          # Здесь команды для деплоя

  deploy-staging:
    if: github.ref == 'refs/heads/main'
    needs: test
    runs-on: ubuntu-latest
    environment:
      name: staging
      url: https://staging.example.com
    steps:
      - uses: actions/checkout@v3
      - name: Download build artifacts
        uses: actions/download-artifact@v3
        with:
          name: build-files
          path: dist/
      - name: Deploy to staging
        run: |
          echo "Deploying to staging environment..."
          # Здесь команды для деплоя
```

```
  deploy-production:
    if: github.ref == 'refs/heads/main'
    needs: deploy-staging
    runs-on: ubuntu-latest
    environment:
      name: production
      url: https://example.com
    steps:
      - uses: actions/checkout@v3
      - name: Download build artifacts
        uses: actions/download-artifact@v3
        with:
          name: build-files
          path: dist/
      - name: Deploy to production
        run: |
          echo "Deploying to production environment..."
          # Здесь команды для деплоя
```

GitLab CI

```
stages:
  - build
  - test
  - deploy-dev
  - deploy-staging
  - deploy-production

variables:
  NODE_ENV: production

build:
  stage: build
  image: node:16
  script:
    - npm ci
    - npm run build
  artifacts:
    paths:
```

```
      - dist/

test:
  stage: test
  image: node:16
  script:
    - npm ci
    - npm test
  dependencies:
    - build

deploy-dev:
  stage: deploy-dev
  script:
    - echo "Deploying to development environment..."
    # Здесь команды для деплоя
  environment:
    name: development
    url: https://dev.example.com
  only:
    - develop

deploy-staging:
  stage: deploy-staging
  script:
    - echo "Deploying to staging environment..."
    # Здесь команды для деплоя
  environment:
    name: staging
    url: https://staging.example.com
  only:
    - main

deploy-production:
  stage: deploy-production
  script:
    - echo "Deploying to production environment..."
    # Здесь команды для деплоя
  environment:
    name: production
```

```
    url: https://example.com
  when: manual
  only:
    - main
```

Визуализация процесса и зависимостей

Чтобы лучше понять, как работают этапы и окружения, представьте их как конвейер на фабрике:

1. **Сборка (Build)** — сырьё превращается в полуфабрикат
2. **Тестирование (Test)** — полуфабрикат проходит контроль качества
3. **Деплой в Dev** — первичная проверка в "лабораторных условиях"
4. **Деплой в Staging** — проверка в условиях, приближенных к реальным
5. **Деплой в Production** — выпуск готового продукта для потребителей

Каждый этап зависит от успешного завершения предыдущего. Если на любом этапе возникает проблема, конвейер останавливается, предотвращая попадание дефектного продукта к пользователю.

Обработка ошибок и восстановление

Что делать, если на каком-то этапе возникла ошибка? В CI/CD есть несколько стратегий:

1. **Автоматический откат (rollback)** — возврат к предыдущей стабильной версии
2. **Ручное вмешательство** — остановка пайплайна и ожидание действий от разработчика
3. **Условное продолжение** — игнорирование некритичных ошибок (например, в тестах производительности)

В GitHub Actions можно использовать условия для обработки ошибок:

```
deploy-with-rollback:
  runs-on: ubuntu-latest
  steps:
    - uses: actions/checkout@v3
    - name: Deploy
      id: deploy
      continue-on-error: true
      run: ./deploy.sh
    - name: Rollback if deploy failed
      if: steps.deploy.outcome == 'failure'
      run: ./rollback.sh
```

В GitLab CI можно использовать `when: on_failure`:

```
deploy:
  stage: deploy
  script:
    - ./deploy.sh

rollback:
  stage: deploy
  script:
    - ./rollback.sh
  when: on_failure
```

Окружения и этапы — это не просто технические термины, это философия разработки, которая помогает доставлять качественный продукт пользователям с минимальными рисками. Правильно настроенный процесс CI/CD с множеством окружений похож на систему шлюзов на реке: он позволяет контролировать поток изменений и предотвращать "наводнения" в виде критических ошибок в продакшне.

Отладка и разбор ошибок

Общие принципы отладки CI/CD

Даже самые тщательно настроенные CI/CD-пайплайны иногда дают сбои. Это нормально — в конце концов, мы имеем дело со сложными системами, где множество компонентов должны работать вместе. Важно уметь быстро находить и устранять проблемы.

Отладка CI/CD во многом похожа на отладку обычного кода, но имеет свои особенности. Вот основные принципы:

1. **Изоляция проблемы** — определите, на каком именно этапе и в каком job-е возникает ошибка
2. **Анализ логов** — внимательно изучите вывод команд, которые завершились с ошибкой
3. **Воспроизведение локально** — попробуйте выполнить те же команды на своей машине
4. **Пошаговое упрощение** — временно отключите сложные части конфигурации, чтобы локализовать проблему
5. **Инкрементальные изменения** — вносите и тестируйте изменения постепенно

Типичные причины ошибок в CI/CD:

- **Синтаксические ошибки** в YAML-файлах (неправильные отступы, пропущенные кавычки)
- **Отсутствующие зависимости** (пакеты, библиотеки, инструменты)
- **Проблемы с правами доступа** (к репозиториям, серверам, API)
- **Несовместимость версий** (например, код требует Node.js 16, а в CI используется Node.js 14)
- **Проблемы с сетью** (недоступность внешних сервисов, таймауты)
- **Различия между окружениями** (код работает локально, но не в CI)

Отладка в GitHub Actions

GitHub Actions предоставляет удобный интерфейс для просмотра логов и отладки workflow.

Где искать логи

1. Перейдите на вкладку "Actions" в вашем репозитории
2. Выберите конкретный запуск workflow (обычно последний)
3. В левой части экрана вы увидите список job-ов
4. Нажмите на job, чтобы увидеть его шаги
5. Нажмите на шаг, чтобы развернуть его лог

GitHub Actions группирует вывод по шагам, что упрощает поиск проблемы. Если шаг завершился с ошибкой, он будет отмечен красным крестиком.

Расширенное логирование

Для более подробной отладки можно включить расширенное логирование:

```
jobs:
  debug-job:
    runs-on: ubuntu-latest
    steps:
      - name: Enable debug logging
        run: echo "ACTIONS_RUNNER_DEBUG=true" >> $GITHUB_ENV
      - name: Your step
        run: your-command
```

Также можно включить отладку на уровне всего workflow, добавив секрет `ACTIONS_RUNNER_DEBUG` со значением `true` в настройках репозитория.

Использование debug-действий

Существуют специальные actions для отладки:

```yaml
jobs:
  debug:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - name: Dump GitHub context
        env:
          GITHUB_CONTEXT: ${{ toJson(github) }}
        run: echo "$GITHUB_CONTEXT"
      - name: Dump job context
        env:
          JOB_CONTEXT: ${{ toJson(job) }}
        run: echo "$JOB_CONTEXT"
      - name: Dump steps context
        env:
          STEPS_CONTEXT: ${{ toJson(steps) }}
        run: echo "$STEPS_CONTEXT"
      - name: Dump runner context
        env:
          RUNNER_CONTEXT: ${{ toJson(runner) }}
        run: echo "$RUNNER_CONTEXT"
```

Этот job выведет в лог все доступные контексты, что поможет понять, какие переменные доступны и какие значения они имеют.

Локальное тестирование с act

Инструмент act позволяет запускать GitHub Actions локально:

```bash
# Установка на macOS
brew install act

# Установка на Linux
curl https://raw.githubusercontent.com/nektos/act/master/install.sh | sudo bash

# Запуск всех workflow
act

# Запуск конкретного workflow
act -W .github/workflows/specific-workflow.yml
```

```
# Запуск с конкретным событием
act push
```

Это особенно полезно для быстрой итерации при разработке workflow, без необходимости коммитить и пушить изменения.

Отладка в GitLab CI

GitLab CI также предоставляет инструменты для отладки pipeline.

Где искать логи

1. Перейдите в раздел "CI/CD" → "Pipelines" вашего проекта
2. Выберите конкретный pipeline
3. Нажмите на job, который вас интересует
4. Вы увидите вкладку "Job log" с полным выводом команд

GitLab показывает вывод в реальном времени, что удобно для отслеживания длительных операций.

Использование CI_DEBUG_TRACE

Для более подробного логирования можно включить переменную `CI_DEBUG_TRACE`:

```
job-name:
  stage: test
  variables:
    CI_DEBUG_TRACE: "true"
  script:
    - your-command
```

Это выведет в лог все выполняемые команды и их вывод, включая команды, которые обычно скрыты.

Локальное тестирование с gitlab-runner

GitLab Runner можно установить локально и использовать для тестирования:

```
# Установка на macOS
brew install gitlab-runner

# Установка на Ubuntu
curl -L https://packages.gitlab.com/install/repositories/runner/gitlab-runner/script.deb.sh |
```

```
sudo bash
sudo apt-get install gitlab-runner

# Запуск job-а локально
gitlab-runner exec docker job-name
```

Это позволяет отлаживать конфигурацию без коммитов в репозиторий.

Отладка с использованием services

Если ваш pipeline использует сервисы (например, базу данных), вы можете отладить их взаимодействие:

```
job:
  services:
    - name: postgres:13
      alias: db
  variables:
    POSTGRES_PASSWORD: password
  script:
    - apt-get update && apt-get install -y postgresql-client
    - psql -h db -U postgres -c "SELECT 1"
    - echo "Database connection successful"
```

В случае проблем с подключением к сервису, вы можете добавить дополнительные команды для проверки сетевого соединения:

```
script:
  - apt-get update && apt-get install -y postgresql-client iputils-ping net-tools
  - ping -c 3 db
  - netstat -tuln
  - psql -h db -U postgres -c "SELECT 1"
```

Типичные проблемы и их решения

1. Ошибки синтаксиса YAML

Проблема: Workflow не запускается или завершается с ошибкой из-за неправильного синтаксиса YAML.

Решение:

- Используйте онлайн-валидаторы YAML (например, YAML Lint)
- Установите расширение для вашего редактора, которое подсвечивает синтаксис YAML
- Обратите внимание на отступы (в YAML они имеют значение)
- Проверьте кавычки вокруг строк, содержащих специальные символы

Пример ошибки:

```
# Неправильно
job:
  steps:
  - name: Step with error
    run: echo "Hello
    World"

# Правильно
job:
  steps:
  - name: Step without error
    run: echo "Hello World"
```

2. Проблемы с переменными и секретами

Проблема: Скрипты не могут получить доступ к переменным или секретам.

Решение:

- Проверьте, правильно ли настроены секреты в настройках репозитория/проекта
- Убедитесь, что вы используете правильный синтаксис для доступа к секретам
- Добавьте отладочный вывод (но не выводите сами секреты!)

Пример отладки:

```
# GitHub Actions
steps:
  - name: Debug API key
    run: |
      if [ -n "${{ secrets.API_KEY }}" ]; then
        echo "API key is set"
      else
        echo "API key is NOT set"
      fi
```

```
# GitLab CI
script:
  - |
    if [ -n "$API_KEY" ]; then
      echo "API key is set"
    else
      echo "API key is NOT set"
    fi
```

3. Проблемы с зависимостями

Проблема: Сборка или тесты падают из-за отсутствующих зависимостей.

Решение:

- Убедитесь, что все зависимости явно указаны
- Зафиксируйте версии зависимостей (например, `npm ci` вместо `npm install`)
- Используйте кеширование для ускорения установки зависимостей

Пример с кешированием:

```
# GitHub Actions
steps:
  - uses: actions/checkout@v3
  - uses: actions/setup-node@v3
    with:
      node-version: '16'
      cache: 'npm'
  - run: npm ci
  - run: npm test

# GitLab CI
job:
  cache:
    key: ${CI_COMMIT_REF_SLUG}
    paths:
      - node_modules/
  script:
    - npm ci
    - npm test
```

4. Проблемы с сетевым доступом

Проблема: CI не может подключиться к внешним сервисам.

Решение:

- Проверьте доступность сервисов из CI (добавьте команды ping, curl и т.д.)
- Убедитесь, что у вас есть необходимые учётные данные
- Рассмотрите возможность использования моков или стабов для тестов

Пример проверки:

```
steps:
  - name: Check connectivity
    run: |
      curl -Is https://api.example.com | head -1
      ping -c 3 api.example.com
```

5. Временные ограничения и таймауты

Проблема: Job превышает максимальное время выполнения.

Решение:

- Оптимизируйте длительные операции
- Разделите большие job-ы на несколько меньших
- Увеличьте таймаут, если это возможно

Пример настройки таймаута:

```
# GitHub Actions
jobs:
  long-job:
    timeout-minutes: 120
    runs-on: ubuntu-latest
    steps:
      - name: Long running task
        run: ./long-task.sh

# GitLab CI
long-job:
  script:
    - ./long-task.sh
```

```
  timeout: 2h
```

Лучшие практики отладки

1. **Добавляйте информативные сообщения** — используйте `echo` или `printf` для вывода промежуточных результатов
2. **Проверяйте окружение** — выводите переменные окружения с помощью `env` или `printenv`
3. **Разделяйте сложные команды** — вместо одной длинной команды используйте несколько простых с промежуточными проверками
4. **Используйте флаги подробного вывода** — многие инструменты имеют флаги вроде `-v` или `--verbose`
5. **Сохраняйте артефакты** — файлы логов, отчёты и другие результаты могут помочь в отладке
6. **Тестируйте изменения в отдельной ветке** — не меняйте основную конфигурацию, пока не убедитесь, что всё работает

Отладка CI/CD может быть сложной, но с правильным подходом вы сможете быстро находить и устранять проблемы. Помните: даже когда CI падает — всегда есть лог, позволяющий найти проблему и быстро её исправить. Это как расследование детектива — у вас есть все улики, нужно только правильно их интерпретировать.

Хорошие практики настройки CI/CD

Структурирование pipeline

Хорошо структурированный CI/CD pipeline похож на хорошо написанный код — он должен быть понятным, поддерживаемым и эффективным. Вот несколько принципов, которые помогут вам создать качественные конфигурации:

Модульность и переиспользование кода

Как и в программировании, в CI/CD стоит избегать дублирования кода. Повторяющиеся блоки лучше выносить в отдельные компоненты.

В GitHub Actions для этого используются:

Composite actions — собственные переиспользуемые действия:

```
# .github/actions/setup-project/action.yml
name: 'Setup Project'
description: 'Sets up Node.js and installs dependencies'
runs:
  using: "composite"
```

```
  steps:
    - uses: actions/setup-node@v3
      with:
        node-version: '16'
        cache: 'npm'
    - run: npm ci
      shell: bash
```

Использование:

```
steps:
  - uses: actions/checkout@v3
  - uses: ./.github/actions/setup-project
  - run: npm test
```

Reusable workflows — переиспользуемые workflow:

```
# .github/workflows/reusable-test.yml
name: Reusable test workflow
on:
  workflow_call:
    inputs:
      node-version:
        required: false
        default: '16'
        type: string

jobs:
  test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - uses: actions/setup-node@v3
        with:
          node-version: ${{ inputs.node-version }}
      - run: npm ci
      - run: npm test
```

Использование:

```
jobs:
  call-test-workflow:
    uses: ./.github/workflows/reusable-test.yml
    with:
      node-version: '18'
```

В GitLab CI для переиспользования используются:

Includes — включение внешних файлов:

```
include:
  - local: 'ci/build.yml'
  - local: 'ci/test.yml'
  - project: 'my-group/my-project'
    file: '/templates/deploy.yml'
```

Extends — наследование конфигурации:

```
# ci/templates.yml
.base-job:
  image: node:16
  before_script:
    - npm ci

# .gitlab-ci.yml
include:
  - local: 'ci/templates.yml'

test:
  extends: .base-job
  script:
    - npm test
```

Anchors — якоря для повторяющихся блоков:

```
.setup: &setup
  - npm ci
  - npm run build

test:
  script:
```

```
    - *setup
    - npm test

lint:
  script:
    - *setup
    - npm run lint
```

Именование job-ов и шагов

Хорошие имена делают конфигурацию более понятной и упрощают отладку:

```
# Плохо
job1:
  steps:
    - run: npm ci
    - run: npm test

# Хорошо
install-and-test:
  steps:
    - name: Install dependencies
      run: npm ci
    - name: Run unit tests
      run: npm test
```

Придерживайтесь единого стиля именования и используйте имена, которые отражают суть операции.

Документирование конфигурации

Комментарии в YAML-файлах помогают объяснить неочевидные моменты:

```
jobs:
  deploy:
    # Этот job деплоит только на staging, для production используйте deploy-prod
    environment: staging
    steps:
      # Используем --force, потому что иногда возникают конфликты с существующими файлами
      - name: Deploy to server
        run: rsync --force -avz ./dist/ user@server:/var/www/
```

Также полезно создать файл `CONTRIBUTING.md` с описанием CI/CD-процесса для новых участников проекта.

Версионирование используемых actions/images

Всегда указывайте конкретные версии используемых actions и Docker-образов:

```
# Плохо - может сломаться при обновлении action
- uses: actions/checkout@master

# Хорошо - фиксированная версия
- uses: actions/checkout@v3
```

```
# Плохо - может сломаться при обновлении образа
image: node

# Хорошо - фиксированная версия
image: node:16.14.2
```

Это делает ваши сборки более предсказуемыми и защищает от неожиданных изменений в зависимостях.

Оптимизация производительности

Быстрый CI/CD — это не просто удобство, это экономия времени и ресурсов. Вот несколько способов ускорить ваши пайплайны:

Кеширование зависимостей и артефактов

В GitHub Actions:

```
jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3

      - uses: actions/setup-node@v3
        with:
          node-version: '16'
          cache: 'npm'
```

```
      - name: Install dependencies
        run: npm ci

      - name: Build
        run: npm run build

      - name: Cache build output
        uses: actions/cache@v3
        with:
          path: dist
          key: ${{ runner.os }}-build-${{ github.sha }}
```

В GitLab CI:

```
build:
  stage: build
  script:
    - npm ci
    - npm run build
  cache:
    key: ${CI_COMMIT_REF_SLUG}
    paths:
      - node_modules/
  artifacts:
    paths:
      - dist/
```

Параллельное выполнение задач

Независимые задачи можно выполнять параллельно:

В GitHub Actions:

```
jobs:
  lint:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - run: npm ci
      - run: npm run lint
```

```
  test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - run: npm ci
      - run: npm test

  # Этот job запустится только после успешного завершения lint и test
  build:
    needs: [lint, test]
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - run: npm ci
      - run: npm run build
```

В GitLab CI:

```
stages:
  - validate
  - build

lint:
  stage: validate
  script:
    - npm ci
    - npm run lint

test:
  stage: validate
  script:
    - npm ci
    - npm test

build:
  stage: build
  script:
    - npm ci
    - npm run build
```

Условное выполнение для экономии ресурсов

Не все задачи нужно выполнять при каждом коммите:

```
jobs:
  e2e-tests:
    runs-on: ubuntu-latest
    # Запускаем e2e-тесты только для PR в main или при пуше в main
    if: github.event_name == 'pull_request' && github.base_ref == 'main' || github.ref ==
'refs/heads/main'
    steps:
      - uses: actions/checkout@v3
      - run: npm ci
      - run: npm run test:e2e
```

Матрицы для тестирования в разных средах

Матрицы позволяют запускать один и тот же job с разными параметрами:

В GitHub Actions:

```
jobs:
  test:
    runs-on: ${{ matrix.os }}
    strategy:
      matrix:
        os: [ubuntu-latest, windows-latest, macos-latest]
        node-version: [14, 16, 18]
    steps:
      - uses: actions/checkout@v3
      - uses: actions/setup-node@v3
        with:
          node-version: ${{ matrix.node-version }}
      - run: npm ci
      - run: npm test
```

В GitLab CI:

```
test:
  parallel:
    matrix:
```

```
      - NODE_VERSION: [14, 16, 18]
        OS: [linux, windows]
  script:
    - npm ci
    - npm test
```

Интеграция с инструментами разработки

CI/CD становится ещё полезнее, когда интегрируется с другими инструментами разработки.

Бейджи статуса в README

Бейджи показывают текущий статус вашего проекта:

GitHub Actions:

```
![CI Status](https://github.com/username/repo/actions/workflows/ci.yml/badge.svg)
```

GitLab CI:

```
[![pipeline
status](https://gitlab.com/username/repo/badges/main/pipeline.svg)](https://gitlab.com/usernam
e/repo/-/commits/main)
```

Интеграция с системами отслеживания задач

CI/CD может автоматически обновлять статус задач:

GitHub Actions с Jira:

```
jobs:
  update-jira:
    runs-on: ubuntu-latest
    steps:
      - name: Jira Login
        uses: atlassian/gajira-login@v3
        env:
          JIRA_BASE_URL: ${{ secrets.JIRA_BASE_URL }}
          JIRA_API_TOKEN: ${{ secrets.JIRA_API_TOKEN }}
          JIRA_USER_EMAIL: ${{ secrets.JIRA_USER_EMAIL }}

      - name: Update Jira issue
```

```
        uses: atlassian/gajira-transition@v3
        with:
          issue: ${{ github.event.pull_request.title | regex_match('^([A-Z]+-[0-9]+)') }}
          transition: "In Review"
```

Автоматические комментарии в PR/MR

CI/CD может оставлять комментарии с результатами проверок:

GitHub Actions:

```
jobs:
  comment:
    runs-on: ubuntu-latest
    steps:
      - name: Comment PR
        uses: actions/github-script@v6
        with:
          github-token: ${{ secrets.GITHUB_TOKEN }}
          script: |
            github.rest.issues.createComment({
              issue_number: context.issue.number,
              owner: context.repo.owner,
              repo: context.repo.repo,
              body: '□ Tests passed successfully!'
            })
```

Уведомления в мессенджеры

CI/CD может отправлять уведомления о статусе сборки:

GitHub Actions с Slack:

```
jobs:
  notify:
    runs-on: ubuntu-latest
    steps:
      - name: Slack Notification
        uses: rtCamp/action-slack-notify@v2
        env:
          SLACK_WEBHOOK: ${{ secrets.SLACK_WEBHOOK }}
```

```
          SLACK_CHANNEL: ci-notifications
          SLACK_TITLE: Build Result
          SLACK_MESSAGE: 'Build ${{ job.status }}'
          SLACK_COLOR: ${{ job.status == 'success' && 'good' || 'danger' }}
```

Масштабирование и поддержка

С ростом проекта растёт и сложность CI/CD. Вот как сделать его более масштабируемым:

Шаблоны для повторяющихся конфигураций

Создавайте шаблоны для типовых задач:

GitHub Actions:

```
# .github/workflow-templates/node-test.yml
name: Node.js Test
on:
  push:
    branches: [ main ]
  pull_request:
    branches: [ main ]
jobs:
  test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - uses: actions/setup-node@v3
        with:
          node-version: '16'
      - run: npm ci
      - run: npm test
```

GitLab CI:

```
# gitlab-ci-templates/node-test.yml
.node-test:
  image: node:16
  script:
    - npm ci
    - npm test
```

Самодокументируемые workflow

Используйте говорящие имена и комментарии:

```
name: Deploy to Production

# Этот workflow деплоит приложение на продакшн-сервер
# Запускается только при создании тега с префиксом 'v'
# Требует ручного одобрения от команды DevOps

on:
  push:
    tags:
      - 'v*'

jobs:
  # Проверяем, что все тесты проходят
  test:
    runs-on: ubuntu-latest
    steps:
      # ...

  # Собираем приложение для продакшна
  build:
    needs: test
    runs-on: ubuntu-latest
    steps:
      # ...

  # Деплоим на продакшн (требуется одобрение)
  deploy:
    needs: build
    environment:
      name: production
      url: https://example.com
    runs-on: ubuntu-latest
    steps:
      # ...
```

Мониторинг производительности CI/CD

Отслеживайте время выполнения пайплайнов и оптимизируйте узкие места:

GitHub Actions:

```
jobs:
  benchmark:
    runs-on: ubuntu-latest
    steps:
      - name: Start timer
        run: echo "start_time=$(date +%s)" >> $GITHUB_ENV

      - name: Run your task
        run: npm run build

      - name: Calculate duration
        run: |
          end_time=$(date +%s)
          duration=$((end_time - ${{ env.start_time }}))
          echo "Task took $duration seconds"
```

GitLab CI предоставляет встроенную метрику времени выполнения для каждого job-а.

Стратегии обновления конфигураций

При обновлении CI/CD-конфигураций:

1. Создавайте отдельную ветку для изменений
2. Тестируйте изменения на этой ветке
3. Используйте PR/MR для обсуждения изменений с командой
4. После слияния внимательно следите за первыми запусками

Хорошо настроенный CI/CD — это инвестиция, которая окупается экономией времени и повышением качества продукта. Как говорил Авраам Линкольн: "Дайте мне шесть часов на то, чтобы срубить дерево, и я потрачу первые четыре на заточку топора". Время, потраченное на настройку CI/CD, — это и есть заточка вашего инструмента разработки.

Расширенные возможности и интеграции

Матрицы для тестирования

Представьте, что вы разрабатываете библиотеку, которая должна работать в разных окружениях: на разных операционных системах, с разными версиями языка программирования или с разными версиями зависимостей. Проверять каждую комбинацию вручную было бы крайне утомительно. Здесь на помощь приходят **матрицы** — мощный инструмент CI/CD для параллельного тестирования в разных условиях.

Концепция матриц в CI/CD

Матрица — это способ определить несколько вариаций одного и того же job-а с разными параметрами. Вместо того чтобы писать отдельный job для каждой комбинации параметров, вы определяете одну матрицу, и система CI/CD автоматически создаёт и запускает все необходимые комбинации.

Типичные параметры для матриц:

- Версии языка программирования (Node.js 14, 16, 18)
- Операционные системы (Linux, Windows, macOS)
- Версии браузеров (Chrome, Firefox, Safari)
- Версии зависимостей (React 16, 17, 18)

Настройка матриц в GitHub Actions

В GitHub Actions матрицы определяются с помощью ключа `strategy.matrix`:

```
jobs:
  test:
    runs-on: ${{ matrix.os }}
    strategy:
      matrix:
        os: [ubuntu-latest, windows-latest, macos-latest]
        node-version: [14, 16, 18]
        # Можно добавить исключения
        exclude:
          # Не тестируем Node.js 14 на macOS
          - os: macos-latest
            node-version: 14
        # Можно добавить дополнительные комбинации
        include:
          # Дополнительно тестируем Node.js 12 только на Ubuntu
          - os: ubuntu-latest
            node-version: 12
    steps:
      - uses: actions/checkout@v3
```

```
      - name: Use Node.js ${{ matrix.node-version }}
        uses: actions/setup-node@v3
        with:
          node-version: ${{ matrix.node-version }}
      - run: npm ci
      - run: npm test
```

В этом примере будет создано 8 job-ов:

- Ubuntu + Node.js 14
- Ubuntu + Node.js 16
- Ubuntu + Node.js 18
- Ubuntu + Node.js 12 (из include)
- Windows + Node.js 14
- Windows + Node.js 16
- Windows + Node.js 18
- macOS + Node.js 16
- macOS + Node.js 18

Обратите внимание, что комбинация macOS + Node.js 14 исключена с помощью `exclude`.

Настройка матриц в GitLab CI

В GitLab CI матрицы реализуются с помощью ключа `parallel.matrix`:

```
test:
  parallel:
    matrix:
      - OS: [ubuntu, windows]
        NODE_VERSION: [14, 16, 18]
  script:
    - echo "Testing on $OS with Node.js $NODE_VERSION"
    - npm ci
    - npm test
```

Этот пример создаст 6 job-ов для всех комбинаций OS и NODE_VERSION.

Оптимизация матричных сборок

Матрицы могут быстро увеличить количество job-ов, что приведёт к увеличению времени сборки и потреблению ресурсов. Вот несколько советов по оптимизации:

1. **Используйте исключения** для пропуска ненужных комбинаций

2. **Добавьте условия для раннего завершения** — например, если тесты падают на одной ОС, нет смысла продолжать на других
3. **Кешируйте зависимости** для ускорения установки
4. **Группируйте связанные параметры** — например, тестируйте только LTS-версии Node.js на всех ОС, а промежуточные версии — только на одной ОС

Работа с артефактами

В процессе CI/CD часто создаются файлы, которые нужно сохранить для последующего использования: скомпилированный код, отчёты о тестировании, документация и т.д. Такие файлы называются **артефактами**.

Что такое артефакты и зачем они нужны

Артефакты — это файлы, созданные во время выполнения pipeline, которые нужно сохранить после завершения job-а. Они могут быть использованы:

- Для передачи результатов сборки между job-ами
- Для скачивания и анализа (например, отчёты о покрытии кода)
- Для деплоя (например, скомпилированные файлы)
- Для архивации (например, логи или снимки экрана при падении тестов)

Сохранение и передача артефактов между job-ами

В GitHub Actions:

```
jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - name: Build
        run: npm run build
      - name: Upload build artifacts
        uses: actions/upload-artifact@v3
        with:
          name: build-files
          path: dist/
          retention-days: 7  # Хранить 7 дней

  deploy:
    needs: build
    runs-on: ubuntu-latest
```

```
    steps:
      - name: Download build artifacts
        uses: actions/download-artifact@v3
        with:
          name: build-files
          path: dist/
      - name: Deploy
        run: ./deploy.sh
```

В GitLab CI:

```
build:
  stage: build
  script:
    - npm run build
  artifacts:
    paths:
      - dist/
    expire_in: 1 week  # Хранить 1 неделю

deploy:
  stage: deploy
  script:
    - ./deploy.sh
  dependencies:
    - build  # Автоматически скачивает артефакты из job-а build
```

Публикация артефактов для скачивания

Артефакты можно скачать через веб-интерфейс GitHub или GitLab после завершения workflow/pipeline. Это удобно для анализа результатов или для ручного деплоя.

В GitHub Actions артефакты доступны на странице конкретного запуска workflow.

В GitLab CI артефакты доступны на странице job-а или pipeline.

Управление жизненным циклом артефактов

Артефакты занимают место в хранилище, поэтому важно управлять их жизненным циклом:

1. **Устанавливайте срок хранения** — не храните артефакты дольше, чем нужно
2. **Ограничивайте размер** — сохраняйте только необходимые файлы
3. **Используйте шаблоны исключения** — например, не сохраняйте node_modules

```
# GitHub Actions
- uses: actions/upload-artifact@v3
  with:
    name: build-files
    path: |
      dist/
      !dist/**/*.map  # Исключаем source maps
    retention-days: 3

# GitLab CI
artifacts:
  paths:
    - dist/
  exclude:
    - dist/**/*.map  # Исключаем source maps
  expire_in: 3 days
```

Интеграция с облачными платформами

Современные приложения часто развёртываются в облачных платформах, таких как AWS, Azure или Google Cloud. CI/CD может автоматизировать этот процесс.

Деплой в AWS, Azure, GCP

Деплой в AWS с GitHub Actions:

```
jobs:
  deploy:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3

      - name: Configure AWS credentials
        uses: aws-actions/configure-aws-credentials@v1
        with:
          aws-access-key-id: ${{ secrets.AWS_ACCESS_KEY_ID }}
          aws-secret-access-key: ${{ secrets.AWS_SECRET_ACCESS_KEY }}
          aws-region: us-east-1

      - name: Deploy to S3
        run: aws s3 sync ./dist s3://my-bucket/ --delete
```

```
      - name: Invalidate CloudFront cache
        run: aws cloudfront create-invalidation --distribution-id ${{
secrets.CF_DISTRIBUTION_ID }} --paths "/*"
```

Деплой в Azure с GitLab CI:

```
deploy:
  stage: deploy
  image: mcr.microsoft.com/azure-cli
  script:
    - az login --service-principal -u $AZURE_SP_CLIENT_ID -p $AZURE_SP_CLIENT_SECRET --tenant
$AZURE_TENANT_ID
    - az webapp deployment source config-zip --resource-group myResourceGroup --name myWebApp
--src dist.zip
```

Использование облачных CLI и SDK

Большинство облачных провайдеров предоставляют CLI-инструменты и SDK, которые можно использовать в CI/CD:

- AWS: aws-cli, AWS SDK
- Azure: azure-cli, Azure SDK
- Google Cloud: gcloud, Google Cloud SDK

Эти инструменты можно использовать напрямую в скриптах или через специальные actions/images.

Управление инфраструктурой через CI/CD

CI/CD может не только деплоить приложения, но и управлять инфраструктурой с помощью инструментов Infrastructure as Code (IaC):

Terraform с GitHub Actions:

```
jobs:
  terraform:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3

      - name: Setup Terraform
        uses: hashicorp/setup-terraform@v2
```

```
      - name: Terraform Init
        run: terraform init

      - name: Terraform Plan
        run: terraform plan -out=tfplan

      - name: Terraform Apply
        if: github.ref == 'refs/heads/main'
        run: terraform apply -auto-approve tfplan
```

Безопасная работа с облачными учетными данными

При работе с облачными платформами критически важно безопасно хранить учётные данные:

1. **Используйте секреты** для хранения ключей и токенов
2. **Ограничивайте права** — предоставляйте минимально необходимые права
3. **Используйте временные учётные данные** — например, AWS STS для получения временных токенов
4. **Настройте ротацию ключей** — регулярно обновляйте ключи доступа

Контейнеризация и CI/CD

Контейнеры стали стандартом для упаковки и развёртывания приложений. CI/CD может автоматизировать работу с контейнерами.

Сборка и публикация Docker-образов

GitHub Actions:

```
jobs:
  build-and-push:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3

      - name: Set up Docker Buildx
        uses: docker/setup-buildx-action@v2

      - name: Login to DockerHub
```

```
        uses: docker/login-action@v2
        with:
          username: ${{ secrets.DOCKERHUB_USERNAME }}
          password: ${{ secrets.DOCKERHUB_TOKEN }}

      - name: Build and push
        uses: docker/build-push-action@v3
        with:
          context: .
          push: true
          tags: username/app:latest,username/app:${{ github.sha }}
          cache-from: type=registry,ref=username/app:buildcache
          cache-to: type=registry,ref=username/app:buildcache,mode=max
```

GitLab CI:

```
build:
  image: docker:20.10.16
  services:
    - docker:20.10.16-dind
  variables:
    DOCKER_TLS_CERTDIR: "/certs"
  script:
    - docker login -u $CI_REGISTRY_USER -p $CI_REGISTRY_PASSWORD $CI_REGISTRY
    - docker build -t $CI_REGISTRY_IMAGE:$CI_COMMIT_SHA .
    - docker push $CI_REGISTRY_IMAGE:$CI_COMMIT_SHA
    - docker tag $CI_REGISTRY_IMAGE:$CI_COMMIT_SHA $CI_REGISTRY_IMAGE:latest
    - docker push $CI_REGISTRY_IMAGE:latest
```

Тестирование в контейнерах

Контейнеры обеспечивают изолированную и воспроизводимую среду для тестирования:

```
# GitHub Actions
jobs:
  test:
    runs-on: ubuntu-latest
    container:
      image: node:16
    services:
      postgres:
```

```
        image: postgres:13
        env:
          POSTGRES_PASSWORD: postgres
        options: >-
          --health-cmd pg_isready
          --health-interval 10s
          --health-timeout 5s
          --health-retries 5
    steps:
      - uses: actions/checkout@v3
      - run: npm ci
      - run: npm test
```

```
# GitLab CI
test:
  image: node:16
  services:
    - postgres:13
  variables:
    POSTGRES_PASSWORD: postgres
  script:
    - npm ci
    - npm test
```

Интеграция с реестрами контейнеров

CI/CD может автоматически публиковать образы в реестры контейнеров:

- Docker Hub
- GitHub Container Registry
- GitLab Container Registry
- AWS ECR
- Google Container Registry
- Azure Container Registry

Оркестрация контейнеров через CI/CD

CI/CD может автоматизировать развёртывание контейнеров в оркестраторы, такие как Kubernetes:

GitHub Actions с Kubernetes:

```
jobs:
  deploy:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3

      - name: Set up kubeconfig
        uses: azure/k8s-set-context@v3
        with:
          kubeconfig: ${{ secrets.KUBE_CONFIG }}

      - name: Deploy to Kubernetes
        run: |
          kubectl apply -f k8s/deployment.yaml
          kubectl rollout status deployment/my-app
```

GitLab CI с Kubernetes:

```
deploy:
  image: bitnami/kubectl:latest
  script:
    - kubectl config set-cluster k8s --server="$KUBE_URL" --insecure-skip-tls-verify=true
    - kubectl config set-credentials admin --token="$KUBE_TOKEN"
    - kubectl config set-context default --cluster=k8s --user=admin
    - kubectl config use-context default
    - kubectl apply -f k8s/deployment.yaml
    - kubectl rollout status deployment/my-app
```

Расширенные возможности CI/CD позволяют автоматизировать практически любой аспект разработки, тестирования и развёртывания приложений. Это как швейцарский нож для DevOps-инженера — множество инструментов для решения разных задач в одном месте.

Терминология и глоссарий

Для удобства навигации по миру CI/CD, давайте разберёмся с основными терминами, которые вы будете встречать при настройке и использовании этих инструментов. Хорошее понимание терминологии поможет вам быстрее освоиться и эффективнее общаться с коллегами.

Branch (ветка)

Определение: Параллельная версия кода в репозитории Git, которая позволяет разработчикам работать независимо друг от друга.

Практическое применение: В CI/CD ветки часто определяют, какие действия должны быть выполнены. Например, пуш в ветку `main` может запускать полный цикл тестирования и деплоя, а пуш в ветку `feature/*` — только базовые тесты.

Пример использования:

```
on:
  push:
    branches: [ main, develop ]
```

Tag (метка)

Определение: Специальная ссылка в Git, указывающая на конкретный коммит, обычно используется для маркировки релизов.

Практическое применение: Теги часто используются для запуска процессов деплоя или публикации релизов.

Пример использования:

```
on:
  push:
    tags:
      - 'v*'  # Любой тег, начинающийся с 'v'
```

Environment (окружение)

Определение: Среда, в которой выполняется код — например, разработка, тестирование, продакшн.

Практическое применение: Разные окружения могут иметь разные конфигурации, переменные и требования безопасности.

Пример использования:

```
deploy:
  environment:
    name: production
    url: https://example.com
```

Secret (секрет)

Определение: Защищённая переменная, содержащая чувствительные данные, такие как пароли, токены API или ключи SSH.

Практическое применение: Секреты используются для безопасного хранения и использования чувствительной информации в CI/CD-процессах.

Пример использования:

```
steps:
  - name: Deploy
    env:
      API_TOKEN: ${{ secrets.API_TOKEN }}
```

Artifact (артефакт)

Определение: Файл или набор файлов, созданных во время выполнения CI/CD-процесса, которые нужно сохранить для последующего использования.

Практическое применение: Артефакты используются для передачи результатов между job-ами или для скачивания и анализа.

Пример использования:

```
- name: Upload artifact
  uses: actions/upload-artifact@v3
  with:
    name: build-files
    path: dist/
```

Cache (кеш)

Определение: Временное хранилище данных, которые могут быть повторно использованы между запусками CI/CD для ускорения процесса.

Практическое применение: Кеширование часто используется для зависимостей (node_modules, vendor и т.д.), чтобы не устанавливать их заново при каждом запуске.

Пример использования:

```
- name: Cache dependencies
  uses: actions/cache@v3
  with:
    path: ~/.npm
```

```
    key: ${{ runner.os }}-node-${{ hashFiles('**/package-lock.json') }}
```

Matrix (матрица)

Определение: Стратегия в CI/CD, которая позволяет запускать один и тот же job с разными параметрами (например, на разных ОС или с разными версиями языка).

Практическое применение: Матрицы используются для тестирования кода в различных окружениях без необходимости дублировать конфигурацию.

Пример использования:

```
strategy:
  matrix:
    os: [ubuntu-latest, windows-latest]
    node-version: [14, 16, 18]
```

Manual job (ручной запуск)

Определение: Job, который не запускается автоматически, а требует ручного запуска через интерфейс CI/CD-системы.

Практическое применение: Ручные job-ы часто используются для деплоя в продакшн или для других критических операций, требующих человеческого контроля.

Пример использования:

```
# GitHub Actions
jobs:
  deploy:
    environment:
      name: production
    # Требует ручного одобрения через интерфейс GitHub

# GitLab CI
deploy:
  when: manual
```

Runner (раннер)

Определение: Сервер или агент, который выполняет задачи CI/CD.

Практическое применение: Раннеры могут быть общими (предоставляемыми GitHub/GitLab) или самостоятельно размещёнными (self-hosted).

Пример использования:

```
jobs:
  build:
    runs-on: ubuntu-latest  # Использование общего раннера GitHub
```

Workflow (рабочий процесс)

Определение: В GitHub Actions — настраиваемый автоматизированный процесс, который выполняет одну или несколько задач.

Практическое применение: Workflow определяется в файле YAML и может быть запущен различными событиями.

Пример использования:

```
name: CI
on: [push, pull_request]
jobs:
  # ...
```

Pipeline (пайплайн)

Определение: В GitLab CI — набор job-ов, сгруппированных по стадиям, которые выполняются последовательно или параллельно.

Практическое применение: Pipeline определяется в файле `.gitlab-ci.yml` и запускается при определённых событиях.

Пример использования:

```
stages:
  - build
  - test
  - deploy
```

Job (задача)

Определение: Набор шагов, которые выполняются на одном раннере.

Практическое применение: Job-ы могут выполняться параллельно или последовательно, в зависимости от настроек.

Пример использования:

```
jobs:
  build:
    # ...
  test:
    # ...
```

Step (шаг)

Определение: В GitHub Actions — отдельная команда или action внутри job-а.

Практическое применение: Шаги выполняются последовательно и могут использовать результаты предыдущих шагов.

Пример использования:

```
steps:
  - name: Checkout code
    uses: actions/checkout@v3
  - name: Build
    run: npm run build
```

Action (действие)

Определение: В GitHub Actions — повторно используемый блок кода, который выполняет определённую задачу.

Практическое применение: Actions могут быть официальными (от GitHub), сторонними или собственными.

Пример использования:

```
- uses: actions/setup-node@v3
  with:
    node-version: '16'
```

Stage (стадия)

Определение: В GitLab CI — группа job-ов, которые выполняются параллельно.

Практическое применение: Стадии выполняются последовательно, что позволяет организовать pipeline в логические этапы.

Пример использования:

```
stages:
  - build
  - test
  - deploy

build-job:
  stage: build
  # ...
```

Trigger (триггер)

Определение: Событие, которое запускает workflow или pipeline.

Практическое применение: Триггерами могут быть пуши, pull request-ы, создание тегов, расписание и т.д.

Пример использования:

```
on:
  push:
    branches: [ main ]
  pull_request:
    branches: [ main ]
  schedule:
    - cron: '0 0 * * *'  # Ежедневно в полночь
```

Self-hosted runner (самостоятельно размещённый раннер)

Определение: Раннер, который вы устанавливаете и управляете самостоятельно, а не используете предоставляемые GitHub/GitLab.

Практическое применение: Self-hosted runner-ы полезны, когда вам нужен доступ к специфическому оборудованию или внутренним ресурсам.

Пример использования:

```
jobs:
  build:
    runs-on: self-hosted
```

Dependency (зависимость)

Определение: В контексте CI/CD — job, который должен быть успешно выполнен перед запуском другого job-а.

Практическое применение: Зависимости позволяют создавать последовательности job-ов.

Пример использования:

```
# GitHub Actions
jobs:
  test:
    # ...
  deploy:
    needs: test
    # ...

# GitLab CI
deploy:
  dependencies:
    - test
```

Понимание этих терминов поможет вам лучше ориентироваться в документации и обсуждениях, связанных с CI/CD. Не стесняйтесь возвращаться к этому глоссарию, когда встречаете незнакомый термин — со временем вся эта терминология станет для вас второй натурой.

Заключение

Вот мы и подошли к концу нашего путешествия по миру продвинутого CI/CD. Мы начали с простых конфигураций, а теперь умеем настраивать сложные пайплайны с ветвлением, секретами, окружениями и множеством других возможностей. Давайте подведём итоги того, что мы узнали.

Что мы изучили

Мы научились:

- **Настраивать условное выполнение** в зависимости от ветки или других параметров — теперь ваш CI/CD может принимать умные решения о том, что и когда запускать
- **Безопасно работать с секретами** — больше никаких паролей в открытом виде в репозитории
- **Управлять разными окружениями** — от разработки до продакшна, с разными уровнями защиты и автоматизации
- **Отлаживать и решать проблемы** в CI/CD — теперь вы знаете, где искать логи и как анализировать ошибки
- **Применять лучшие практики** для структурирования и оптимизации пайплайнов
- **Использовать продвинутые возможности** — матрицы, артефакты, интеграции с облачными платформами и контейнеризацию

Всё это делает ваш CI/CD не просто набором скриптов, а полноценным инструментом, который помогает доставлять качественный код пользователям быстро и безопасно.

Практические рекомендации по внедрению

Если вы только начинаете внедрять CI/CD или хотите улучшить существующие процессы, вот несколько рекомендаций:

1. **Начинайте с малого** — не пытайтесь сразу настроить идеальный пайплайн со всеми возможностями. Начните с базовых тестов и постепенно добавляйте новые функции.
2. **Автоматизируйте рутину в первую очередь** — определите, какие задачи отнимают больше всего времени у команды, и автоматизируйте именно их.
3. **Инвестируйте в тесты** — CI/CD без хороших тестов — это быстрый способ доставки багов пользователям. Чем лучше ваши тесты, тем больше пользы от автоматизации.
4. **Документируйте процессы** — убедитесь, что все члены команды понимают, как работает ваш CI/CD и что делать в случае проблем.
5. **Регулярно пересматривайте конфигурацию** — CI/CD не высечен в камне. По мере роста проекта и команды пересматривайте и улучшайте процессы.

Перспективы развития навыков CI/CD

Мир CI/CD постоянно развивается, и есть множество направлений для дальнейшего изучения:

- **GitOps** — подход к управлению инфраструктурой и приложениями через Git
- **Непрерывный мониторинг** — интеграция мониторинга и оповещений в процесс CI/CD
- **Канареечные релизы** — постепенное развёртывание новых версий для ограниченной группы пользователей
- **Автоматизированное тестирование безопасности** — интеграция инструментов безопасности в пайплайн

- **Машинное обучение в CI/CD** — использование ML для оптимизации процессов и предсказания проблем

Призыв к действию

Теперь, когда у вас есть знания и инструменты, самое время применить их на практике. Начните с небольших изменений в своих существующих workflow или создайте новый с нуля. Экспериментируйте, не бойтесь ошибок — CI/CD как раз и нужен для того, чтобы ловить их до того, как они попадут к пользователям.

Помните, что хороший CI/CD — это не цель, а путь. Он должен постоянно эволюционировать вместе с вашим проектом и командой. То, что работает сегодня, может потребовать изменений завтра, и это нормально.

И напоследок: поздравляем, вы прокачали свой CI/CD до продвинутого уровня! Теперь ваш код сам себя проверяет и разворачивает — а вы можете расслабиться и заняться более интересными задачами. Ну, почти расслабиться — роботы пока не захватили мир, и кто-то должен писать сам код □□

Удачи в ваших CI/CD-приключениях!