1. **Введение в Git**
   * Что такое Git и история его развития
   * Основные преимущества и использование Git в современной разработке
2. **Основы работы с Git**
   * Установка и настройка Git
   * Создание и клонирование репозиториев
3. **Основы работы с репозиториями**
   * Основы системы контроля версий
   * Работа с локальными и удалёнными репозиториями
4. **Коммиты и ветвление в Git**
   * Создание коммитов: добавление и фиксация изменений
   * Введение в ветвление и мерджи
5. **Работа с ветками**
   * Создание, переключение и слияние веток
   * Решение конфликтов при слиянии
6. **Теги и логирование в Git**
   * Понятие тегов и их использование
   * Просмотр истории изменений: команды log и diff
7. **Работа с удаленными репозиториями**
   * Подключение к удаленным репозиториям
   * Основы fetch, push и pull
8. **Эффективная работа с командами Git**
   * Основы сташинга и ребейзинга
   * Часто используемые команды и их параметры
9. **Введение в Git Flow и другие модели ветвления**
   * Обзор Git Flow, GitHub Flow и GitLab Flow
   * Выбор подходящей модели работы для проекта
10. **Работа с GUI для Git и интеграция с IDE**
    * Обзор популярных GUI клиентов Git
    * Интеграция Git в различные среды разработки
11. **Продвинутые техники работы с Git**
    * Интерактивный ребейз и cherry-pick
    * Работа с большими репозиториями и оптимизация производительности
12. **Лучшие практики и общие рекомендации**
    * Обзор лучших практик работы с Git
    * Советы по устранению распространенных проблем и эффективной работе

# Введение в Git

## Что такое Git и история его развития

**Определение и история:**

* **Git** — это распределенная система контроля версий, созданная Линусом Торвальдсом в 2005 году.
* Разработана для управления разработкой ядра Linux, Git быстро стала популярной среди разработчиков за её эффективность и гибкость.

**Ключевые вехи:**

* **2005**: Создание Git Линусом Торвальдсом после разногласий с BitKeeper.
* **2008**: Основание GitHub, платформы для хостинга репозиториев Git, которая значительно увеличила популярность Git.

**Основные преимущества и использование Git в современной разработке**

**Эффективность:**

* **Быстрый доступ к истории изменений:** Git обеспечивает мгновенный доступ к истории проекта, что ускоряет процесс разработки.
* **Локальная работа:** Все операции выполняются локально, что повышает скорость и позволяет работать вне сети.

**Безопасность:**

* **Надёжность данных:** Каждый коммит в Git имеет уникальный SHA-1 хеш, обеспечивая целостность истории версий.
* **Отказоустойчивость:** Распределенная структура делает Git устойчивым к сбоям сервера.

**Гибкость:**

* **Ветвление и слияние:** Git облегчает параллельную работу над разными функциями и легко справляется с слиянием изменений.
* **Масштабируемость:** Git хорошо масштабируется от маленьких до очень больших проектов.

## Примеры использования Git

**1. Создание нового репозитория:**

git init

Эта команда инициализирует новый Git репозиторий в текущем каталоге, создавая подкаталог .git с необходимыми файлами репозитория.

**2. Клонирование существующего репозитория:**

git clone https://github.com/username/repository.git

Эта команда создаёт локальную копию репозитория, находящегося по указанному URL.

**3. Просмотр изменений:**

git status

Показывает статус файлов в рабочем каталоге и индексе: какие изменены, какие готовы к коммиту.

**4. Добавление изменений в индекс:**

git add <filename>

Добавляет указанный файл или файлы к следующему коммиту.

**5. Совершение коммита:**

git commit -m "Ваше сообщение о коммите"

Создаёт новый коммит с внесенными изменениями и прикрепляет к нему сообщение.

# Основы работы с Git

## Установка и настройка Git

**Установка Git:**

1. **Windows:**
   * Скачайте установщик с [git-scm.com](https://git-scm.com/).
   * Запустите установщик и следуйте инструкциям. Рекомендуется использовать настройки по умолчанию.
   * После установки проверьте работоспособность, открыв Git Bash или командную строку и введя git --version.
2. **MacOS:**
   * Установите Git через Homebrew: brew install git.
   * Или скачайте установщик с [git-scm.com](https://git-scm.com/).
3. **Linux:**
   * Установите Git через менеджер пакетов. Например, для Ubuntu/Debian: sudo apt-get install git.

**Настройка Git:**

* Установите имя пользователя и email, которые будут использоваться в ваших коммитах.
* git config --global user.name "Ваше Имя"
* git config --global user.email "ваш.email@example.com"
* Проверьте настройки:
* git config --list

## Создание и клонирование репозиториев

**Создание нового репозитория:**

1. Перейдите в каталог, где будет находиться ваш проект.
2. Инициализируйте репозиторий:
3. git init

Эта команда создаст подкаталог .git, который содержит все необходимые файлы репозитория.

## **Клонирование существующего репозитория:**

1. Найдите URL репозитория. Например, на GitHub это может быть что-то вроде https://github.com/username/repository.git.
2. Клонируйте репозиторий:
3. git clone https://github.com/username/repository.git

Это создаст копию репозитория в вашей локальной системе.

## Полезные советы и тонкости

1. **Игнорирование файлов:**
   * Создайте файл .gitignore в корне вашего репозитория.
   * Добавьте в него шаблоны файлов, которые вы хотите игнорировать (например, \*.log, node\_modules/).
2. **Просмотр изменений:**
   * Используйте git status, чтобы увидеть, какие файлы изменены или готовы к коммиту.
   * git diff покажет конкретные изменения в файлах.
3. **Коммиты:**
   * Делайте частые коммиты с четкими и понятными сообщениями.
   * Используйте git commit -am "Сообщение коммита" для добавления изменений всех отслеживаемых файлов и совершения коммита одной командой.
4. **Ветвление:**
   * Не бойтесь использовать ветки для разработки новых функций.
   * git branch <имя\_ветки> для создания новой ветки.
   * git checkout <имя\_ветки> для переключения на другую ветку.
5. **Обновление и слияние:**
   * Регулярно обновляйте вашу локальную копию с помощью git pull.
   * Для слияния изменений из одной ветки в другую используйте git merge <имя\_ветки>.
6. **Работа с удаленными репозиториями:**
   * git push отправит ваши коммиты на удаленный сервер.
   * git fetch получит последние изменения, но не сольет их с вашей рабочей копией.
7. **Резервное копирование настроек:**
   * Резервное копирование файла .gitconfig поможет вам быстро настроить Git на новой машине.

# Основы работы с репозиториями в Git

#### Основы системы контроля версий

**Типы систем контроля версий:**

* **Локальные системы контроля версий:** Простейшие системы, сохраняющие изменения в локальном хранилище.
* **Централизованные системы контроля версий (CVCS):** Единый сервер содержит все версии файлов, а клиенты загружают нужную версию оттуда.
* **Распределённые системы контроля версий (DVCS), например Git:** Каждый участник хранит полную копию репозитория, что обеспечивает высокую отказоустойчивость и гибкость.

#### Работа с удалёнными репозиториями

**Работа с удалёнными репозиториями:**

* **Клонирование удалённого репозитория:**
* git clone <url>

Создает локальную копию удаленного репозитория.

* **Синхронизация с удалённым репозиторием:**
  + **Получение изменений:**
  + git pull

git pull получает изменения из удаленного репозитория и объединяет их с текущей веткой.

* + **Отправка изменений:**
  + git push

git push отправляет коммиты из локальной ветки в удаленный репозиторий.

**3. Ветвление и слияние:**

* **Создание ветки:**
* git branch <имя\_ветки>

Создает новую ветку.

* **Переключение между ветками:**
* git checkout <имя\_ветки>

Переключает на указанную ветку.

* **Слияние веток:**
* git merge <имя\_ветки>

Объединяет изменения из указанной ветки с текущей веткой.

#### Полезные советы и хитрости

1. **Использование .gitignore:**
   * Создайте файл .gitignore в корне вашего проекта, чтобы игнорировать временные файлы, логи, зависимости.
2. **Часто используемые команды:**
   * git status для просмотра состояния репозитория.
   * git log для просмотра истории коммитов.
3. **Хранение учетных данных:**
   * Используйте git config credential.helper store для сохранения имени пользователя и пароля.
4. **Использование тегов:**
   * git tag для маркировки важных моментов, например, релизов.
5. **Работа с ветками:**
   * Регулярно обновляйте ветки, выполняя git pull, чтобы избежать конфликтов при слиянии.
6. **Использование Git в командной строке:**
   * Изучение командной строки улучшает понимание и гибкость работы с Git.

# Коммиты и ветвление в Git

#### Создание коммитов: добавление и фиксация изменений

**1. Основы коммитов:**

* **Что такое коммит?** Коммит в Git — это запись в истории изменений вашего проекта. Каждый коммит содержит информацию об изменениях, авторе изменений, дате и уникальный идентификатор (SHA-1 хеш).
* **Важность коммитов:** Коммиты позволяют вам возвращаться к предыдущим состояниям проекта, отслеживать изменения и разбираться в истории разработки.

**2. Процесс создания коммита:**

* **Добавление изменений:**
* git add <файл\_или\_папка>

git add добавляет изменения в файлах в область подготовленных файлов (staging area), подготавливая их к коммиту.

* **Создание коммита:**
* git commit -m "Ваше сообщение коммита"

git commit сохраняет подготовленные изменения в репозитории с указанным сообщением коммита.

**3. Хорошие практики для коммитов:**

* **Четкие сообщения коммитов:** Каждое сообщение должно четко описывать внесенные изменения.
* **Атомарность коммитов:** Стремитесь к тому, чтобы каждый коммит представлял собой одно логическое изменение.
* **Частые коммиты:** Регулярное коммитирование помогает отслеживать прогресс и облегчает откат изменений при необходимости.

**4. Просмотр истории коммитов:**

* git log показывает историю коммитов.
* git log --oneline для компактного представления истории.

#### Введение в ветвление и мерджи

**1. Ветвление в Git:**

* **Что такое ветка?** Ветка в Git — это отдельная линия разработки. Основная ветка в Git называется master или main.
* **Создание новой ветки:**
* git branch <имя\_ветки>

Эта команда создает новую ветку.

* **Переключение между ветками:**
* git checkout <имя\_ветки>

git checkout переключает вас на указанную ветку.

**2. Слияние веток (Merge):**

* **Основы слияния:** Слияние веток — это процесс объединения изменений из одной ветки в другую.
* **Выполнение слияния:**
* git merge <имя\_ветки>

Эта команда объединяет изменения из <имя\_ветки> с текущей веткой.

**3. Разрешение конфликтов слияния:**

* В случае конфликтов Git помечает их в файлах. Вам нужно вручную разрешить эти конфликты и затем завершить процесс слияния.

**4. Полезные советы и хитрости:**

* **Использование git checkout -b <имя\_ветки>:** Эта команда создает новую ветку и сразу переключает вас на нее.
* **Применение git merge --no-ff <имя\_ветки>:** Опция --no-ff создает новый коммит слияния даже если это "fast-forward merge", что полезно для сохранения истории изменений.
* **Визуализация ветвей:** git log --graph показывает дерево ветвей в графическом представлении.

# Работа с ветками в Git

#### Создание, переключение и слияние веток

**1. Создание веток:**

* Ветки в Git позволяют разработчикам работать параллельно над разными функциями или версиями проекта.
* **Создание ветки:**
* git branch <имя\_ветки>

Эта команда создаёт новую ветку, но не переключает на неё автоматически.

**2. Переключение между ветками:**

* **Переключение на ветку:**
* git checkout <имя\_ветки>

git checkout переключает текущую рабочую ветку.

* **Совет:** Используйте git checkout -b <имя\_ветки> для создания новой ветки и немедленного переключения на неё.

**3. Слияние веток:**

* **Слияние (Merge):**
* git merge <имя\_ветки>

Эта команда объединяет изменения из <имя\_ветки> с текущей веткой. Git пытается автоматически слить изменения.

* **Fast-forward Merge vs. No-Fast-Forward Merge:**
  + **Fast-forward:** Если нет отдельных коммитов в текущей ветке с момента отделения, Git выполнит fast-forward merge.
  + **No-fast-forward:** Если в текущей ветке есть коммиты, Git создаст новый коммит слияния.

#### Решение конфликтов при слиянии

**1. Понимание конфликтов:**

* Конфликты возникают, когда изменения в одних и тех же частях файла не могут быть автоматически объединены Git.
* Git помечает эти области и останавливает процесс слияния, требуя ручного разрешения.

**2. Разрешение конфликтов:**

* Конфликты отмечены в файле с помощью специальных маркеров <<<<<<<, =======, >>>>>>>.
* Разработчик должен решить, какие изменения должны остаться, изменить файл соответственно и удалить маркеры Git.

**3. После разрешения конфликтов:**

* После редактирования файла, сохраните его и выполните:
* git add <файл>
* Затем завершите слияние:
* git commit

Git автоматически создаст коммит слияния с подходящим сообщением.

#### Полезные советы и хитрости

1. **Визуализация ветвления:**

Используйте git log --graph --all --oneline для визуального представления истории ветвей.

1. **Удаление веток:**

После слияния ветки часто можно удалить: git branch -d <имя\_ветки> Используйте -d для безопасного удаления (ветка должна быть полностью слита) или -D для принудительного.

1. **Использование rebase вместо merge:**

git rebase может использоваться для переписывания истории, делая её более чистой, но следует быть осторожным, так как это может запутать историю коммитов.

1. **Работа с удалёнными ветками:**

Для синхронизации с удалённой веткой используйте: bash git push origin <имя\_ветки>  или для получения изменений: bash git pull origin <имя\_ветки>

1. **Сохранение изменений перед переключением веток:**

Если нужно переключиться на другую ветку, но не хочется терять текущие изменения, используйте git stash для временного сохранения.

# Теги и логирование в Git

#### Понятие тегов и их использование

**1. Введение в теги:**

* **Что такое теги в Git?** Теги в Git используются для маркировки определенных значимых точек в истории репозитория, обычно для обозначения версий релизов (например, v1.0, v2.0).
* **Виды тегов:** Есть два типа тегов в Git: легковесные (lightweight) и аннотированные (annotated). Аннотированные теги содержат дополнительную информацию: имя создателя тега, email, дату создания и сообщение.

**2. Создание тегов:**

* **Аннотированный тег:**
* git tag -a v1.0 -m "Сообщение к тегу"

Здесь -a v1.0 создает аннотированный тег с именем v1.0, а -m добавляет к нему сообщение.

* **Легковесный тег:**
* git tag v1.0-light

Легковесные теги создаются без дополнительных опций и используются для временной маркировки.

**3. Просмотр тегов:**

* **Список тегов:**
* git tag

Эта команда отображает список всех тегов в репозитории.

* **Подробная информация о теге:**
* git show v1.0

Показывает информацию о теге v1.0, включая соответствующий коммит.

**4. Публикация тегов:**

* Теги не отправляются на сервер автоматически при использовании git push. Для отправки тегов используйте:
* git push origin <имя\_тега>
* Чтобы отправить все теги, используйте:
* git push origin --tags

#### Просмотр истории изменений: команды log и diff

**1. Git log:**

* **Основы:**
* git log

Эта команда показывает историю коммитов в текущей ветке, включая автора, дату и сообщение коммита.

* **Команды для улучшения читаемости:**
  + git log --oneline показывает каждый коммит в одной строке.
  + git log --graph отображает ветки в виде ASCII-графа.
  + git log --since="2 weeks ago" показывает коммиты за последние две недели.

**2. Git diff:**

* **Основы:**
* git diff

Показывает разницу в файлах между рабочим каталогом и индексом (тем, что готово к коммиту).

* **Сравнение коммитов:**
* git diff <commit1> <commit2>

Показывает различия между двумя коммитами.

* **Сравнение веток:**
* git diff <branch1>..<branch2>

Показывает различия между двумя ветками.

**3. Полезные советы и хитрости:**

* **Сохранение вывода log в файл:**
* git log > history.txt

Это сохранит историю коммитов в файл history.txt.

* **Использование псевдонимов (aliases):** Д

ля удобства можно создать псевдонимы для часто используемых команд с помощью git config.

* **Поиск в истории:** git log -S"search\_term" поможет найти коммиты, где встречается определенный термин.

# Работа с удаленными репозиториями в Git

#### Подключение к удаленным репозиториям

**1. Основы удаленных репозиториев:**

* **Что такое удаленный репозиторий?** Удаленный репозиторий — это версия вашего проекта, которая хранится в интернете или на другом сетевом месте. Это позволяет многим пользователям работать с одним проектом из любого места.
* **Популярные сервисы:** Наиболее известные хосты для Git репозиториев включают GitHub, GitLab, Bitbucket.

**2. Добавление удаленного репозитория:**

* **Клонирование существующего репозитория:**
* git clone <url>

Эта команда создает локальную копию удаленного репозитория.

* **Добавление удаленного репозитория к существующему локальному:**
* git remote add origin <url>

Здесь origin — стандартное имя для удаленного репозитория, но вы можете использовать любое другое.

**3. Просмотр удаленных репозиториев:**

* **Список удаленных репозиториев:**
* git remote -v

Показывает список всех удаленных репозиториев, связанных с локальным.

#### Основы fetch, push и pull

**1. Fetch:**

* **Использование fetch:**
* git fetch origin

git fetch загружает изменения с удаленного репозитория, но не объединяет их с вашей локальной веткой.

* **Просмотр изменений:**
* git log HEAD..origin/master

Это покажет, что изменилось с момента последнего fetch.

**2. Pull:**

* **Использование pull:**
* git pull origin master

git pull это по сути комбинация команд fetch и merge, она получает изменения с удаленного сервера и сразу объединяет их с текущей локальной веткой.

**3. Push:**

* **Использование push:**
* git push origin master

git push отправляет ваши локальные коммиты на удаленный репозиторий. Здесь master - это ветка, которую вы отправляете.

#### Полезные советы и хитрости

1. **Работа с несколькими удаленными репозиториями:**
   * Вы можете иметь несколько удаленных репозиториев для одного локального, например, для различных серверов или для резервных копий.
2. **Отслеживание веток:**
   * **Установка отслеживаемой ветки:**
   * git branch --set-upstream-to=origin/<branch>

Это настроит локальную ветку для отслеживания удаленной ветки, что упрощает команды pull и push.

1. **Использование тегов с удаленными репозиториями:**
   * Не забывайте отправлять теги на удаленный сервер после их создания.
2. **Избежание проблем с push:**
   * Если git push не работает, возможно, вам нужно сначала выполнить git pull, чтобы объединить любые удаленные изменения.
3. \*\*Удаление удал

енных веток:\*\*

* Чтобы удалить ветку на удаленном сервере, используйте:

git push origin --delete <branch>

# Эффективная работа с командами Git

#### Основы сташинга и ребейзинга

**1. Сташинг (Stashing):**

* **Что такое сташ?** Git stash позволяет временно сохранить изменения, которые вы еще не готовы коммитить. Это полезно, когда нужно быстро переключиться на другую задачу.
* **Создание сташа:**
* git stash

Это сохраняет ваши текущие изменения в стэке сташей и возвращает рабочую директорию в состояние последнего коммита.

* **Применение сташа:**
* git stash pop

Это применяет изменения из последнего сташа и удаляет его из стека.

**2. Ребейзинг (Rebasing):**

* **Что такое ребейзинг?** Ребейзинг — это процесс перемещения или «переписывания» последовательности коммитов на новую базу.
* **Использование ребейза:**
* git rebase <base>

Например, git rebase master перемещает текущую ветку на вершину ветки master.

* **Интерактивный ребейз:**
* git rebase -i HEAD~3

Интерактивный ребейз позволяет изменять коммиты в указанном диапазоне (здесь последние 3 коммита).

Полезные советы и хитрости

1. **Алиасы команд:** Для повышения эффективности можно настроить алиасы (короткие команды) для часто используемых команд Git. Например, **git config --global alias.co checkout**.
2. **Отмена сташа без применения изменений:**
   * Используйте **git stash drop** для удаления последнего сташа без применения его изменений.
3. **Выборочный ребейз:**
   * Интерактивный ребейз позволяет объединять, изменять и удалять коммиты для чистой истории.
4. **Извлечение определенных изменений из сташа:**
   * **git stash show -p stash@{0} | git apply** для применения конкретных изменений из определенного сташа.
5. **Использование .gitconfig:**
   * Настройка вашего файла **.gitconfig** для настройки алиасов, установки глобальных настроек и т.д. повышает эффективность работы.

# Введение в Git Flow и другие модели ветвления

#### Обзор Git Flow, GitHub Flow и GitLab Flow

**1. Git Flow:**

* **Описание:** Git Flow — это модель ветвления, предложенная Винсентом Дриессеном. Она предназначена для проектов с регулярными релизами и состоит из двух главных веток (master и develop) и трех типов вспомогательных веток (фича, релиз, хотфикс).
* **Компоненты:**
  + **Master:** Основная ветка, содержит готовый код релизов.
  + **Develop:** Ветка разработки, содержит последние изменения для следующего релиза.
  + **Feature branches:** Ветки для разработки новых функций.
  + **Release branches:** Ветки для подготовки релизов, корректировки ошибок и других задач.
  + **Hotfix branches:** Ветки для быстрого исправления ошибок в продакшене.

**2. GitHub Flow:**

* **Описание:** GitHub Flow — это более простая модель, предложенная GitHub. Она предполагает непрерывную доставку и подходит для проектов с частыми релизами.
* **Процесс:**
  + Создание ветки от master для новой работы.
  + Регулярные коммиты в эту ветку.
  + Открытие Pull Request для обсуждения изменений.
  + Слияние ветки обратно в master после ревью.

**3. GitLab Flow:**

* **Описание:** GitLab Flow сочетает элементы Git Flow и GitHub Flow, добавляя концепцию окружения (например, production, staging).
* **Ключевые аспекты:**
  + Использование ветки master для стабильной версии продукта.
  + Ветки предварительных релизов для стадии тестирования.
  + Ветки функций для новых разработок и изменений.

#### Выбор подходящей модели работы для проекта

**1. Факторы для выбора:**

* **Размер и сложность проекта:** Для больших проектов с регулярными релизами Git Flow может быть предпочтительнее.
* **Частота релизов:** Для проектов с частыми или непрерывными релизами GitHub Flow или GitLab Flow могут быть более уместны.
* **Командная работа и ревью:** Если акцент делается на код-ревью и коллаборации, GitHub Flow предлагает удобный процесс с Pull Requests.

**2. Примеры использования:**

* **Стартапы и маленькие команды:** Вероятно, оценят простоту и гибкость GitHub Flow.
* **Крупные проекты с четким расписанием релизов:** Скорее всего, предпочтут структурированный подход Git Flow.
* **Компании, требующие строгого контроля над процессом разработки и релизами:** Могут найти GitLab Flow более подходящим для своих нужд.

# Использование Git как инструмента CI/CD с примером GitHub Actions

**Введение в CI/CD и Git**

**Непрерывная интеграция (CI)** и **непрерывная доставка/развертывание (CD)** являются ключевыми практиками в современной разработке программного обеспечения. CI/CD обеспечивает автоматизацию процессов тестирования и развертывания, улучшая скорость и качество разработки. **Git**, популярная система контроля версий, играет важную роль в этом процессе, обеспечивая эффективное управление исходным кодом.

**Git в контексте CI/CD**

В контексте CI/CD, Git используется для отслеживания каждого изменения кода. Каждый коммит в репозитории Git может автоматически инициировать процесс CI, который включает в себя запуск тестов, проверку кода на соответствие стандартам и другие проверки качества. После успешного прохождения этих этапов CI, начинается процесс CD, включающий в себя подготовку и развертывание кода в производственной среде.

**GitHub Actions как инструмент CI/CD**

**GitHub Actions** — это функциональность, встроенная в GitHub, предоставляющая мощные возможности для автоматизации CI/CD. Actions позволяют автоматизировать рабочие процессы, основанные на различных событиях в репозитории Git, таких как пуш (push), пул реквест (pull request) или тег.

**Особенности GitHub Actions:**

1. **Гибкость**: GitHub Actions позволяет настроить рабочие процессы, которые соответствуют уникальным требованиям проекта.
2. **Интеграция с GitHub**: Так как Actions интегрированы непосредственно с GitHub, управление рабочими процессами становится удобным и централизованным.
3. **Расширяемость**: Можно использовать предварительно созданные действия из Marketplace или создать собственные.

**Пример использования:**

При коммите в ветку **master**, GitHub Actions может автоматически запускать тесты, строить билд и, при успешном прохождении всех проверок, развертывать приложение на сервере.

Вот пример GitHub Action для репозитория с сервером на Go, который выполняет тестирование, сборку бинарного файла, сохранение его на Linux сервер через SSH и запуск:

name: Go Server CI/CD

on:

push:

branches:

- master # Запускать этот workflow при пушах в ветку master

jobs:

build-and-deploy:

name: Build and Deploy Go Server

runs-on: ubuntu-latest # Выбор среды выполнения

steps:

- name: Checkout Code

uses: actions/checkout@v3 # Получение исходного кода из репозитория

- name: Set up Go

uses: actions/setup-go@v3

with:

go-version: 1.18 # Указание версии Go

- name: Test

run: go test ./... # Запуск тестов

- name: Build

run: |

CGO\_ENABLED=0 GOOS=linux go build -a -installsuffix cgo -o myserver . # Сборка сервера

env:

GO111MODULE: on

- name: Copy binary to Server

uses: appleboy/scp-action@master

with:

host: ${{ secrets.SERVER\_HOST }} # Использование секретов GitHub для хранения конфиденциальных данных

username: ${{ secrets.SERVER\_USER }}

password: ${{ secrets.SERVER\_PASSWORD }}

port: ${{ secrets.SERVER\_PORT }}

source: "./myserver" # Путь к бинарному файлу

target: "/path/on/server" # Путь на сервере

- name: Restart server

uses: appleboy/ssh-action@master

with:

host: ${{ secrets.SERVER\_HOST }}

username: ${{ secrets.SERVER\_USER }}

password: ${{ secrets.SERVER\_PASSWORD }}

port: ${{ secrets.SERVER\_PORT }}

script: |

systemctl stop myserver.service # Остановка старого процесса сервера

mv /path/on/server/myserver /usr/local/bin/myserver # Перемещение нового бинарного файла

systemctl start myserver.service # Запуск нового процесса сервера

Этот пример предполагает, что у вас на сервере Linux уже настроен systemd service для вашего сервера Go (myserver.service). Также важно сохранить данные для доступа к серверу (например, SERVER\_HOST, SERVER\_USER, SERVER\_PASSWORD, SERVER\_PORT) в секретах репозитория на GitHub, чтобы обеспечить безопасность этих данных.

Этот GitHub Action автоматизирует весь процесс CI/CD для Go сервера, начиная от тестирования и сборки и заканчивая развертыванием и запуском на удаленном сервере.

# Что почитать

<https://git-scm.com/book/ru/v2>

Git для новичков (часть 1) <https://habr.com/ru/articles/541258/>

Gitflow workflow <https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow>

Introducing GitFlow <https://datasift.github.io/gitflow/IntroducingGitFlow.html>

Удачная модель ветвления для Git <https://habr.com/ru/articles/106912/>