Лекция 8

GIT

Система контроля версий *(от англ.* ***Version Control System, VCS)*** позволяет сохранять историю изменений в файлах и возвращаться к предыдущим состояниям. Это необходимо для восстановления предыдущих версий, понимания, кто и когда внес определенные изменения, и управления параллельными изменениями от разных разработчиков.

Git — это распределенная система контроля версий, которая позволяет разработчикам отслеживать изменения в файлах и координировать работу над общими проектами. Git популярен в разработке программного обеспечения из-за его мощных возможностей ветвления и слияния, а также из-за его эффективности и надежности.

Распределенность означает, что каждый разработчик работает с локальной копией репозитория, которая включает всю историю проекта. Это позволяет работать с кодом и вносить изменения без постоянного доступа к сети и обеспечивает высокий уровень отказоустойчивости.

Репозиторий, коммит и ветка

Репозиторий — это хранилище данных, содержащее все файлы проекта и их историю изменений. Репозитории могут существовать локально на компьютере разработчика или удаленно, например, на таких сервисах, как GitHub, GitLab или Bitbucket.

Коммит — это запись в истории изменений репозитория, который содержит снимок всех файлов проекта на момент коммита. Каждый коммит имеет уникальный идентификатор (хэш), автора, дату и сообщение, описывающее суть изменений.

Ветка — это параллельная версия репозитория. Ветки позволяют вам работать над отдельными функциями вашего проекта, не влияя на основную версию. Закончив работу над новой фичей, вы можете объединить эту ветку с основной версией проекта.

В репозитории всегда есть по крайней мере одна ветка, даже если вы сами ее не создавали. Обычно ее называют веткой **main** (или **master**).

О системе контроля версий подробнее

https://git-scm.com/book/ru/v2/%D0%92%D0%B2%D0%B5%D0%B4%D0%B5%D0%BD%D0%B8%D0%B5-%D0%9E-%D1%81%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B5-%D0%BA%D0%BE%D0%BD%D1%82%D1%80%D0%BE%D0%BB%D1%8F-%D0%B2%D0%B5%D1%80%D1%81%D0%B8%D0%B9

Практикум: https://githowto.com/ru

Что такое VCS ввиде коимкса https://habr.com/ru/articles/552872/

Процесс работы в git

1. **Клонирование репозитория**: Создается локальная копия удаленного репозитория, включая всю историю и ветки проекта. Это делается с помощью команды `git clone`.
2. **Создание новой ветки**: Перед внесением изменений рекомендуется создать новую ветку командой `git branch <branch_name>`. Это позволяет разработчикам работать изолированно от главной линии разработки (обычно ветка `master` или `main`).
3. **Переключение между ветками**: Для начала работы в новой ветке используется команда `git checkout <branch_name>`. Это изменяет рабочую копию файлов на те, что соответствуют состоянию выбранной ветки.
4. **Добавление изменений**: После внесения изменений в файлы они добавляются в индекс с помощью команды `git add`. Это подготавливает измененные файлы к коммиту.
5. **Совершение коммита**: Подготовленные изменения фиксируются в истории репозитория с помощью команды `git commit`. Каждый коммит требует сообщения, описывающего сделанные изменения.
6. **Отправка изменений**: Чтобы поделиться изменениями с другими, коммиты из локальной ветки отправляются в удаленный репозиторий командой `git push`.

7. **Слияние веток**: После завершения работы в ветке её изменения можно объединить с главной веткой (например, `main`). Это обычно делается через запрос на слияние (pull request) в веб-интерфейсе сервиса (например, GitHub) или командой `git merge` в командной строке.
8. **Решение конфликтов**: Если одни и те же части кода были изменены в разных ветках, Git может не суметь автоматически слить их. В этом случае разработчику потребуется вручную разрешить конфликты и выполнить коммит этих изменений.
9. **Обновление локального репозитория**: Регулярно рекомендуется обновлять локальный репозиторий, чтобы он соответствовал удаленному. Для этого используется команда `git pull`.

Дополнительная информация

1. Инсталлятор GIT https://git-scm.com/downloads
2. Регистрация в GitHub https://github.com/
3. Авторизация в GitHub https://docs.github.com/en/authentication/connecting-to-github-with-ssh

Практическая часть

Справочник команд

1. git config

- Команда `git config` используется для настройки параметров Git на уровне конкретного репозитория, пользователя или всей системы. Эти настройки контролируют поведение Git, включая такие вещи, как имя пользователя, адрес электронной почты, форматирование вывода, цвет командной строки и многое другое.

Настройка имени пользователя и электронной почты:

- Одна из первых задач, которую обычно выполняют с помощью `git config`, — это настройка имени пользователя и адреса электронной почты. Это важно, поскольку каждый Git коммит использует эту информацию, и она включена в коммиты, которые вы отправляете.

Команды:

- **Установка имени пользователя:**
  - `git config --global user.name "Your Name"`
  - Пример: `git config --global user.name "John Doe"`
  - Эта команда устанавливает имя, которое будет прикреплено к вашим коммитам и тегам.
- **Установка адреса электронной почты:**
  - `git config --global user.email "your_email@example.com"`
  - Пример: `git config --global user.email "johndoe@example.com"`
  - Этот адрес электронной почты будет использоваться в информации о коммите.

Корректная обработка окончаний строк:

- Для пользователей Unix/Mac:

  `git config --global core.autocrlf input`

  `git config --global core.safecrlf warn`
- Для пользователей Windows:

  `git config --global core.autocrlf true`

  `git config --global core.safecrlf warn`

Имя ветки по умолчанию

Мы будем использовать main в качестве имени ветки по умолчанию. Чтобы установить его, выполните следующую команду:

```
git config --global init.defaultBranch main
```

2. git init

- Используется для инициализации нового Git репозитория в существующем каталоге. Это первый шаг в создании нового проекта под управлением Git.
- Команда: `git init`
- После выполнения, в текущем каталоге создается подкаталог `.git`, содержащий все необходимые файлы репозитория — скелет нового Git репозитория.

3. git clone

- Служит для клонирования репозитория с удаленного репозитория. Эта команда создает полную копию всех данных репозитория, включая всю историю и ветки.
- Команда: `git clone [url]`
- Пример: `git clone https://github.com/example/repo.git`
- Выполнение этой команды создаст каталог `repo` с подкаталогом `.git`, копирует все данные репозитория и установит текущую рабочую копию в последнюю версию.

4. git status

- Показывает состояние изменений как индексированных, так и неиндексированных. Это помогает понять, какие изменения были сделаны и какие из них будут включены в следующий коммит.
- Команда: `git status`
- Вызов этой команды отобразит список измененных файлов, а также файлов, которые готовы к коммиту (после использования `git add`).

5. git add

- Добавляет изменения в файлах в индекс (подготовка к коммиту). Можно добавлять отдельные файлы или все изменения сразу.

- Команда: `git add [file]` или `git add .` для добавления всех изменений.
- Пример: `git add index.html` добавит изменения только для файла `index.html`.

6. git commit

- Коммитит подготовленные изменения и сохраняет их в истории репозитория. Каждый коммит требует сообщения, которое описывает совершенные изменения.
- Команда: `git commit -m "commit message"`
- Пример: `git commit -m "Add feature X"`
- Это создает новый снимок текущего состояния проекта, который можно потом восстановить или на который можно ссылаться.
- Статусы файлов в Git:
    - **Untracked**: Файлы новые, не отслеживаемые Git'ом. Появляются в репозитории, но не добавлены в индекс с помощью `git add`.
    - **Modified**: Файлы изменены в рабочем каталоге, но еще не добавлены в индекс.
    - **Staged**: Измененные файлы добавлены в индекс и готовы к коммиту.
    - **Committed**: Изменения в файлах зафиксированы в истории репозитория.

7. git push

- Отправляет локальные коммиты на удаленный репозиторий. Это позволяет поделиться своей работой с другими.
- Команда: `git push [alias] [branch]`
- Пример: `git push origin main`
- Эта команда отправит изменения из локальной ветки `main` в удаленный репозиторий, связанный с алиасом `origin`.

8. git fetch

- Команда `git fetch` скачивает коммиты, файлы и ссылки из удаленного репозитория в ваш локальный репозиторий.
- Команда: `git fetch [remote-name]`
- Пример: `git fetch origin`
- `git fetch` обновляет локальную информацию о состоянии удаленного репозитория, но не влияет на вашу рабочую директорию.

9. git pull

- Команда `git pull` используется для получения и слияния любых изменений с удаленного репозитория в текущую ветку в локальной рабочей директории.
- Команда: `git pull [remote-name] [branch-name]`
- Пример: `git pull origin master`
- `git pull` является комбинацией команд `git fetch` и `git merge`, обновляя текущую локальную ветку до последней версии на удаленном сервере.

10. git show

- Команда `git show` отображает информацию о любом git объекте, например коммите или теге, вместе с соответствующими изменениями.
- Команда: `git show [commit-hash]`
- Пример: `git show 1a410efbd13591db07496601ebc7a059dd55cfe9`
- По умолчанию `git show` отображает информацию о последнем коммите, если не указан конкретный объект.

11. git log

- Команда `git log` используется для просмотра истории коммитов в текущей ветке.

- Команда: `git log`
- Дополнительные опции могут быть использованы для настройки вывода, например, `--oneline` для сокращенного представления или `--graph` для визуального представления ветвления.

12. git blame

- Команда `git blame` показывает, кто и когда внес изменения в каждую строку файла.
- Команда: `git blame [file-name]`
- Пример: `git blame README.md`
- Это полезно для отслеживания истории изменений и выяснения авторства кода.

13. git diff

- Команда `git diff` используется для сравнения изменений между коммитами, ветками, файлами и т.д.
- Команда: `git diff [branch1]..[branch2]`
- Пример: `git diff master..feature`
- Это позволяет увидеть различия в коде между разными состояниями репозитория.

14. git reset

- Команда `git reset` используется для отмены изменений. Она может изменять индекс (staging area), рабочий каталог и историю коммитов в зависимости от переданных параметров.
- Примеры:
    - `git reset` без параметров сбросит индекс, но оставит рабочий каталог нетронутым.
    - `git reset --soft [commit]` отменит все коммиты после указанного, но оставит изменения в индексе.

- `git reset --hard [commit]` полностью уберет все изменения после указанного коммита, включая индекс и рабочий каталог.

15. git checkout

- Команда `git checkout` используется для переключения между ветками или восстановления рабочих файлов.
- Примеры:
  - `git checkout [branch-name]` переключит HEAD на указанную ветку.
  - `git checkout [commit-hash]` переключит HEAD на указанный коммит.
  - `git checkout -- [file-name]` отменит все изменения в указанном файле, возвращая его к последнему коммиту.

16. git stash

- Команда `git stash` используется для временного сохранения изменений, которые вы не готовы коммитить. Это позволяет очистить рабочий каталог, не теряя изменений.
- Примеры:
  - `git stash` сохранит ваши изменения и очистит рабочий каталог.
  - `git stash list` покажет список всех сохраненных изменений.
  - `git stash apply` восстановит последние сохраненные изменения обратно в рабочий каталог.
  - `git stash drop` удалит последние сохраненные изменения из списка stash.

17. git branch

- Команда `git branch` используется для управления ветками в Git.

Создание ветки:

- Чтобы создать новую ветку:
    - Команда: `git branch [branch-name]`
    - Пример: `git branch feature_x`

Переключение веток:

- Чтобы переключиться на другую ветку:
    - Команда: `git checkout [branch-name]`
    - Пример: `git checkout feature_x`

Удаление ветки:

- Чтобы удалить ветку:
    - Команда: `git branch -d [branch-name]` (безопасное удаление, удаляет только смерженные ветки)
    - Команда: `git branch -D [branch-name]` (принудительное удаление, используется для удаления несмерженных веток)
    - Пример: `git branch -d feature_x`

Переименование ветки:

- Чтобы переименовать ветку:
    - Команда: `git branch -m [old-branch-name] [new-branch-name]`
    - Пример: `git branch -m feature_x feature_y`

18. git merge

- Команда `git merge` используется для слияния изменений из одной ветки в другую.

- Перед слиянием необходимо переключиться на ветку, в которую вы хотите слить изменения.
- Команда: `git merge [source-branch]`
- Пример: переключиться на ветку `master` командой `git checkout master`, затем слить в нее ветку `feature_x` командой `git merge feature_x`.
- При слиянии могут возникнуть конфликты, которые нужно разрешать вручную.

Последовательность действий для создания удаленной ветки:

1. **Создание новой локальной ветки (если это необходимо):**
    - Команда: `git branch [branch-name]`
    - Пример: `git branch new-feature`
2. **Переключение на новую ветку:**
    - Команда: `git checkout new-feature`
3. **Добавление и коммит изменений (если это необходимо):**
    - Команда: `git add .`
    - Команда: `git commit -m "Initial commit on new-feature branch"`
4. **Первый push и установка upstream (связанной) ветки:**
    - Команда: `git push -u origin new-feature`
    - Эта команда отправит изменения в ветке `new-feature` в удаленный репозиторий (предполагая, что удаленный репозиторий называется `origin`).
    - Опция `-u` устанавливает `new-feature` как upstream ветку для `origin/new-feature`, что позволяет в дальнейшем использовать `git push` и `git pull` без указания имени ветки.

Что такое Pull Request?

- **Pull Request** (PR) — это функциональность платформ для совместной разработки, таких как GitHub, GitLab, Bitbucket, которая позволяет разработчикам сообщать о готовности внесенных изменений и запросить их слияние в основную ветку проекта.
- PR используется для обсуждения изменений с другими участниками проекта, проведения код-ревью и запуска автоматизированных тестов перед слиянием изменений.
- Процесс создания PR обычно включает следующие шаги:
    1. Разработчик делает форк (копию) репозитория, вносит изменения и публикует их в своем форке.
    2. Разработчик создает PR на оригинальном репозитории, указывая ветку с изменениями и целевую ветку для слияния.
    3. Другие разработчики проводят ревью изменений, оставляют комментарии и предложения.
    4. После одобрения, изменения могут быть слиты в основную ветку.

Задание № 1 Базовые команды git

Шаг 1: Зарегистрируйте учетную запись на GitHub.
1. Перейдите на [главную страницу GitHub](https://github.com/).
2. Нажмите "Зарегистрироваться" в правом верхнем углу.
3. Введите свои данные:
   - Имя пользователя: Выберите уникальное имя пользователя.
   - Адрес электронной почты: Укажите действительный адрес электронной почты.
   - Пароль: Создайте надежный пароль.
4. Нажмите "Создать учетную запись".
5. Подтвердите свой адрес электронной почты, перейдя по ссылке, отправленной на ваш электронный адрес.

Шаг 2: Установите Git **(если установлен пропускаем этот шаг)**
GitHub работает с Git, системой контроля версий. Вам необходимо установить Git на свой компьютер.

1. Загрузите Git с официального сайта:
   - Для Windows: https://git-scm.com/download/win
   - Для macOS: https://git-scm.com/download/mac

- Для Linux: Установите через менеджер пакетов (например, "sudo apt-get install git" в Ubuntu).
2. Следуйте инструкциям по установке для вашей операционной системы.

Шаг 3: Настройте Git
1. Откройте терминал в папке вашего проекта.
2. Укажите свое имя пользователя в Git:
   git config --global user.name "Ваше имя"
3. Укажите свой адрес электронной почты:
   git config --global user.email "Ваша почта"
4. Укажите имя ветки по умолчанию
   git config --global init.defaultBranch main
5. Установите корректную обработку окончаний строк
   git config --global core.autocrlf true
   git config --global core.safecrlf warn

Шаг 4: Создайте репозиторий
Репозиторий (или "репо") - это место, где хранятся файлы вашего проекта.

Вариант 1: Создайте репозиторий на GitHub
1. Войдите в свою учетную запись на GitHub.
2. Нажмите кнопку "+" в правом верхнем углу панели мониторинга.
3. Выберите "Создать репозиторий".
4. Заполните данные:
   - Название репозитория: укажите название вашего репозитория.
   - Описание: Добавьте краткое описание (необязательно).
   - Выберите Частный.
   - Установите галочку напротив “**Add a README file”**
5. Нажмите Создать репозиторий.

Вариант 2: Создайте локальный репозиторий и загрузите его на GitHub
1. Создайте папку для своего проекта на своем компьютере. (Опционально)
2. Откройте терминал или командную строку и перейдите в эту папку.
3. Инициализируйте репозиторий Git:
   git init
4. Привяжите свой локальный репозиторий к GitHub:
   git remote add origin https://github.com/yourusername/yourrepositoryname.git
5. Разместите свои файлы на GitHub:
   git add .
   git commit -m "Начальная фиксация"
   git push -u origin master

Шаг 5: Клонируем репозиторий
Если вы хотите поработать над существующим проектом, вы можете клонировать его на свой компьютер.

1. Перейдите в репозиторий GitHub, который вы хотите клонировать.
2. Нажмите кнопку "Код" и скопируйте URL-адрес.
3. Откройте терминал или командную строку.
4. Перейдите в папку, в которую вы хотите клонировать репозиторий.
5. Выполните следующую команду:
   git clone https://github.com/username/repositoryname.git
6. Репозиторий будет загружен на ваш компьютер.

Шаг 6: Работа с репозиторием
Вот как добавлять, фиксировать и отправлять изменения в репозиторий.

1. Откройте файл README.md уже в локальном репозитории и добавьте в него строку "Мое первое изменение"
2. Проверьте состояние изменений
   git status
3. Добавьте файлы в репозиторий:
   git add . # Добавляет все файлы в текущий каталог
4. Проверьте состояние изменений
   git status
5. Зафиксируйте изменения:
   git commit -m "Мое первое изменение"
6. Проверьте состояние изменений
   git status
7. Внесите изменения в GitHub:
   git push

Задание № 2 Отслеживание изменений в git

Подготовка к выполнению упражнений
Создайте тестовый репозиторий:
- Создайте новый репозиторий на GitHub (например, "git-practice").
- Создайте две папки с вашим ИМЕНЕМ и ФАМИЛИЕЙ (будем имитировать работу двух разработчиков над одним проектом)
- Клонируйте репозиторий в обе папки на свой локальный компьютер:
  git clone https://github.com/yourusername/git-practice.git
- Перейдите в каталог **git-practice**, находящийся в папке с вашим ИМЕНЕМ :
  cd ./имя/git-practice
- В папке с вашим именем укажите имя пользователя в Git:
  git config user.name "Ваше имя"
- Укажите адрес электронной почты:
  git config user.email "имя@ex.com"
- Создайте файл (например, `example.txt`) и добавьте немного контента:
  echo "Line 1: Initial content" > example.txt
  git add example.txt
  git commit -m "Initial commit: Created example.txt"

git push

- Перейдите в каталог **git-practice**, находящийся в папке с вашей ФАМИЛИЕЙ:

cd ../фамилия/git-practice

- В папке с вашей фамилией укажите имя пользователя в Git:

git config user.name "Ваше фамилия"

- Укажите адрес электронной почты:

git config user.email "фамилия@ex.com"

Упражнение 1: Попрактикуйтесь с "**git fetch**" и "**git pull**".
Цель: Понять разницу между "**git fetch**" и "**git pull**".

1. Располагаясь в каталоге **git-practice** из папки с фамилией используйте `git fetch`:

- Запустите следующую команду, чтобы получить последние изменения из удаленного репозитория:

git fetch

- Обратите внимание, что изменения еще не внесены в вашу локальную ветку. Убедитесь в этом, проверив содержимое каталога:

ls -l

2. Используйте "**git pull**":

- Теперь запустите:

git pull

- Это позволит извлечь изменения и объединить их в вашей локальной ветке.
- Проверьте каталог:

ls -l

3. Основные выводы:

- `**git fetch**` обновляет вашу локальную копию удаленного репозитория, но не изменяет ваши рабочие файлы.
- "**git pull**" обновляет ваш локальный репозиторий и рабочие файлы, извлекая и объединяя изменения.

Упражнение 2: Попрактикуйтесь с "**git show**"
Цель: Используйте "git show" для просмотра подробной информации о фиксации.

1. Создайте новую фиксацию:

- Отредактируйте `example.txt" и добавьте новую строку:

echo "Строка 2: Вторая строка" >> example.txt
git add .
git commit -m "Добавлена вторая строка"
git push

2. Используйте `**git show**`:

- Запустите следующую команду, чтобы просмотреть подробную информацию о самой последней фиксации:

git show

- Здесь будет отображен хэш коммита, автор, дата и изменения, внесенные в коммит.

3. Просмотрите конкретный коммит.:
  - Найдите хэш предыдущего коммита (например, используя "git log").
  - Запустите:
    git show хэш_вашего_коммита
  - Это отобразит подробную информацию о заданном коммите.

4. Вывод:
  - "**git show**" - это удобный способ просмотра подробной информации о точке изменения (коммите), включая внесенные изменения.

Упражнение 3: Попрактикуйтесь с "**git log**".
Цель: Используйте "git log" для изучения истории коммитов.

1. Сделайте несколько коммитов:
  - Отредактируйте `**example.txt"** и внесите несколько изменений, фиксируя после каждого изменения:

echo "Строка 3: Третья строка" >> example.txt
git add .
git commit -m "Добавлена третья строка"
git push

echo "Строка 4: Четвертая строка" >> example.txt
git add .
git commit -m "Добавлена четвертая строка"
git push

2. Просмотр истории коммитов:
  - Запустите следующую команду, чтобы просмотреть историю фиксации:
git log
  - На экране отобразится список фиксаций, включая хэш фиксации, автора, дату и сообщение.

3. Настройте выходные данные:
  - Используйте параметр "--oneline" для компактного просмотра:
    git log --oneline
  - Используйте опцию `--graph`, чтобы визуализировать историю коммитов в виде графика:
    git log --graph

4. Просмотр конкретных коммитов:
  - Используйте `git log` для ряда коммитов:
    git log HEAD~2..HEAD
  - Здесь будут показаны два последних коммита.

5. Вывод:

- "**git log**" - это мощный инструмент для изучения истории коммитов и понимания эволюции вашего проекта.

Упражнение 4: Попрактикуйтесь с `**git blame**`
Цель: Используйте "git blame", чтобы определить, кто внес изменения в файл.

1. Имитируйте совместную работу:
- Создайте несколько коммитов от имени разных пользователей.

2. Используйте `**git blame**`:
- Запустите следующую команду, чтобы увидеть, кто внес изменения в каждую строку `example.txt`:
git blame example.txt
- В каждой строке будет отображен хэш фиксации, автор и дата.

3. Просмотрите подробную информацию об авторах изменений.:
- Используйте опцию `-c`, чтобы отобразить сообщение о фиксации:
git blame -c example.txt

4. Вывод:
- `git blame" помогает определить, кто внес изменения в определенные строки в файле, что полезно для понимания и помогает в отладке.

Упражнение 5: Попрактикуйтесь с "git diff".
Цель: Используйте `**git diff**` для сравнения изменений между файлами или коммитами.

1. Внесите незафиксированные изменения:
- Отредактируйте `example.txt" и добавьте новую строку:
echo "Строка 5: Пятая строка" >> example.txt

2. Используйте "git diff":
- Запустите следующую команду, чтобы увидеть внесенные вами изменения:
git diff
- Это отобразит различия между вашим рабочим каталогом и промежуточной областью.

3. Внесите изменения и зафиксируйте их:
- Поэтапно внести изменения:
git add .
- Зафиксировать изменения:
git commit -m "Добавлена пятая строка"

4. Сравните коммиты:
- Используйте "git diff" для сравнения двух коммитов. Например:
git diff HEAD~3 HEAD
git diff fc3fda8 3026298 (можно указать хэши коммитов)

- Это покажет различия между последней фиксацией и 4 по счету состоянием репозитория.

5. Вывод:
- `git diff` необходим для проверки изменений перед фиксацией или объединением ветвей.

Задание № 3 Отмена, перезапуск и сохранение изменений, переключение и управление ветками в git.

Упражнение 1: Использование **git reset** для отмены изменений

Цель: Научиться использовать git reset для отмены или перезапуска изменений в вашем репозитории.

Установка:

1. Создайте тестовый репозиторий:
- Откройте терминал и создайте новый каталог для вашего тестового репозитория.
- Инициализируйте Git-репозиторий:

```
mkdir git-test
cd git-test
git init
```

- Создаем файл и фиксируем его:

```
echo "Hello, Git!" > hello.txt
git add hello.txt
git commit -m "Initial commit"
```

2. Внесите изменения:
- Редактировать `hello.txt`:

```
echo "Hello, Git! This is a test." >> hello.txt
```

Упражнение:

1. Просмотрите статус:
- Проверьте статус вашего репозитория, чтобы увидеть изменения:

```
git status
```

Вы должны увидеть, что "hello.txt` было изменено.

2. Сбросьте изменения с помощью `**git reset --hard**`:
- Сбросьте репозиторий до последней фиксации, отменив все изменения:

```
git reset --hard
```

- Проверьте статус еще раз:

```
git status
```

Изменения в "hello.txt` должны быть отменены.

3. Измените файл еще раз:
- Внесите еще одно изменение в `hello.txt`:
echo "Another line." >> hello.txt

4. Внесите изменения и выполните сброс с помощью `git reset --mixed` (по умолчанию):
- Поэтапно внесите изменения:
git add hello.txt
- Сбросить репозиторий до последней фиксации, отменив все изменения, но сохранив их в рабочем каталоге.:
git reset
- Проверьте статус:
git status
Изменения в файле по-прежнему присутствует, статус файла модифицирован, но не проиндексирован.

5. Сбросьте конкретную фиксацию с помощью "git reset --soft":
- Сначала зафиксируйте изменения:
git add hello.txt
git commit -m "Second commit"
- Теперь возвращаемся к первой фиксации, сохраняя изменения поэтапными:
git reset --soft HEAD~1
- Проверьте статус:
git status
Изменения должны быть поэтапными, но не обязательными.

Вывод:

- `git reset --hard`: отменяет все изменения и выполняет сброс до указанной фиксации.
- `git reset --mixed` (по умолчанию): отменяет изменения, но сохраняет их в рабочем каталоге.
- `git reset --soft`: отменяет изменения, но сохраняет их поэтапными, что полезно для внесения изменений в коммиты.

Упражнение 2: Использование **git checkout** для переключения ветвей.

Цель: Научиться использовать "git checkout" для переключения между ветвями и управления различными направлениями разработки.

Установка:

1. Создайте тестовый репозиторий:
- Если вы еще этого не сделали, создайте репозиторий "git-test" из предыдущего упражнения.

2. Создайте новую ветку:
- Создайте новую ветку с именем **"feature-branch"** и переключитесь на нее:

git checkout -b feature-branch

- Проверьте текущую ветку:

git branch

Вы должны увидеть `**feature-branch**` как активную ветвь.

3. Внесите изменения в новую ветвь:
   - Отредактируйте `hello.txt`:

echo "Feature branch change." >> hello.txt

- Зафиксировать изменения:

git add hello.txt

git commit -m "Change from feature branch"

Упражнение:

1. Переключитесь обратно на главную ветку:
   - Проверьте "главную" ветку:

git checkout main

- Проверьте текущую ветку:

git branch

Теперь вы должны быть в ветке "main".

2. Проверьте содержимое файла:
   - Проверьте содержимое файла `hello.txt`:

cat hello.txt

Изменения, внесенные в `**feature-branch**`, не должны присутствовать.

3. Переключитесь обратно на `**feature-branch**`:

git checkout feature-branch

- Проверьте текущую ветку и содержимое `hello.txt`:

git branch

cat hello.txt

Вы должны увидеть изменения, внесенные в `**feature-branch**`.

4. Объедините `**feature-branch**` ветку с основной:
   - Переключитесь обратно на `основную` ветку:

git checkout main

- Объединить `**feature-branch**` с `**main**`:

git merge feature-branch

- Убедитесь, что изменения из `**feature-branch**` теперь внесены в "**main**":

cat hello.txt

Вывод:

- **git checkout -b branch-name**: создает новую ветку и переключается на нее.
- **git checkout branch-name**: переключается на существующую ветку.

- Используйте ветки для управления различными направлениями разработки и легкого переключения контекстов.

Упражнение 3: Использование **git stash** для сохранения и отмены изменений

Цель: Научиться использовать git stash для временного сохранения и отмены изменений.

Установка:

1. Создайте тестовый репозиторий:
   - Если вы еще этого не сделали, создайте репозиторий "git-test" из предыдущих упражнений.

2. Внесите некоторые изменения:
   - Отредактируйте `hello.txt`:
     echo "Temporary change." >> hello.txt

Упражнение:

1. Сохраните свои изменения:
   - Сохраняйте ваши изменения, не фиксируя их:
     git stash
   - Проверить статус:
     git status
     Статус “Модифицирован” больше не должно быть.

2. Просмотрите свой тайник:
   - Перечислите сохраненные изменения:
     git stash list
     Вы должны увидеть запись типа `stash@{0}: WIP on main.

3. Повторно примените свой Stash:
   - Примените сохраненные изменения обратно к вашему рабочему каталогу:
     git stash apply
   - Проверьте статус:
     git status
     Изменения должны быть возвращены в ваш рабочий каталог.

4. Зафиксируйте изменения:
   - Этапировать и зафиксировать изменения:
     git add hello.txt
     git commit -m "Stashed changes”

5. Очистите хранилище:
   - Удалите запись в хранилище после того, как вы применили и зафиксировали изменения:
     git stash drop stash@{0}

- Убедитесь, что хранилище пусто:
    git stash list

Основные выводы:

- `**git stash**`: сохраняет ваши изменения, не фиксируя их.
- "**git stash list**": выводит список всех сохраненных изменений.
- "**git stash apply**": применяет сохраненные изменения к вашему рабочему каталогу.
- `**git stash drop**`: удаляет запись о тайнике.

Упражнение 4. Объединение **git reset, git checkout и git stash** в рабочем процессе.

Цель: Попрактиковаться в совместном использовании "git reset", "git checkout" и "git stash" в типичном рабочем процессе.

Установка:

1. Начните с хранилища тестов:
   - Убедитесь, что у вас есть хранилище "git-test` из предыдущих упражнений.

2. Создайте новую ветку:
   - Создайте новую ветку с именем "feature-2" и переключитесь на нее:
     git checkout -b feature-2

3. Внесите некоторые изменения:
   - Редактировать `hello.txt`:
     echo "Feature 2 change." >> hello.txt

Упражнение:

1. Сохраните свои изменения:
   - Перед фиксацией сохраните свои изменения:
     git stash
   - Убедитесь, что изменения сохранены:
     git stash list

2. Переключитесь на основную ветку:
   - Проверьте ветку "main":
     git checkout main
   - Проверьте текущую ветку:
     git branch

3. Сбросьте основную ветвь:
   - Верните основную ветвь в исходное состояние:
     git reset --hard HEAD~1

- Проверьте содержимое файла `hello.txt`:
cat hello.txt
Файл должен вернуться к более раннему состоянию.

4. Повторно примените свой Stash к ветви Feature:
- Переключитесь обратно на ветвь "feature-2".:
git checkout feature-2
- Применить сохраненные изменения:
git stash apply
- Проверьте изменения:
git status
cat hello.txt

5. Зафиксируйте свои изменения:
- Подготовьте и зафиксируйте изменения в ветке `feature-2`:
git add hello.txt
git commit -m "Feature 2 changes"

6. Объедините функциональную ветку с основной:
- Переключитесь на ветку "главная`:
git checkout main
- Объединить ветвь "feature-2" с веткой `main`:
git merge feature-2
- Убедитесь, что изменения уже внесены в ветку "main":
cat hello.txt

Вывод:

- Используйте `**git stash**` для сохранения изменений при переключении контекстов.
- Используйте `**git checkout**` для управления различными ветвями и направлениями разработки.
- Используйте "**git reset**", чтобы отменить изменения и управлять историей репозитория.

Конфликты при слиянии

Чтобы разрешить конфликты слияния в Git при объединении изменений из разных ветвей, выполните следующие действия:

1. Определите конфликт:
- Когда вы пытаетесь объединить две ветви и Git обнаруживает конфликты, он уведомляет вас сообщением, указывающим на наличие конфликтующих изменений в одном или нескольких файлах.

2. Откройте конфликтующий файл:
- Найдите файлы с конфликтами. Git помечает эти файлы маркерами конфликтов (`<<<<<<<`, `=======`, и `>>>>>>>`).

3. Проверьте конфликт:
  - Откройте конфликтующий файл в текстовом редакторе.
  - Ищите маркеры конфликта. Раздел между "<<<<<<< HEAD" и "======" представляет ваши изменения (текущую ветку), а раздел между "======" и ">>>>>>" представляет входящие изменения из другой ветки.

4. Устраните конфликт:
  - Решите, какие изменения сохранить:
    - Вариант 1: Сохраните свои изменения (из текущей ветки).
    - Вариант 2: Сохраните входящие изменения (из другой ветки).
    - Вариант 3: Объедините оба набора изменений вручную.
  - Отредактируйте файл, чтобы отразить желаемый результат, удалив маркеры конфликтов.

5. Подготовьте разрешенный файл.:
  - Как только конфликт будет разрешен, добавьте файл в промежуточную область, чтобы указать, что конфликт устранен:
    git add example.txt

6. Зафиксируйте разрешение:
  - Зафиксируйте изменения для завершения слияния:
    git commit -m " Разрешил конфликт слияния в example.txt"

7. Проверьте разрешение:
  - После фиксации убедитесь, что файл содержит правильное объединенное содержимое.
  - Используйте "git log" или "git status", чтобы убедиться, что слияние прошло успешно и конфликтов не осталось.

8. Используйте инструменты слияния (необязательно):
  - Если конфликт сложный, рассмотрите возможность использования инструмента слияния Git для графического интерфейса, который поможет устранить различия:
    git mergetool

9. При необходимости отмените изменения:
  - Если вы допустите ошибку во время разрешения конфликта, вы можете вернуть файл в предыдущее состояние, используя:
    git checkout -- example.txt

Git Flow

Git Flow - популярная модель ветвления для Git, которая помогает командам управлять сложными рабочими процессами разработки, особенно для проектов с множеством функций и запланированными выпусками. Вот структурированный обзор Git Flow:

Ключевые компоненты Git Flow

1. Основная ветвь (master):
   - Назначение: Это ветка, готовая к работе, в которой находится действующее приложение.
   - Использование: В эту ветку не вносятся прямые изменения. Все изменения объединяются в основную после тщательного тестирования.

2. Ветка разработки (develop):
   - Назначение: Эта ветвь служит точкой интеграции для всех ветвей функций.
   - Использование: Функциональные ветви объединяются в develop для тестирования, прежде чем они будут признаны готовыми к работе.

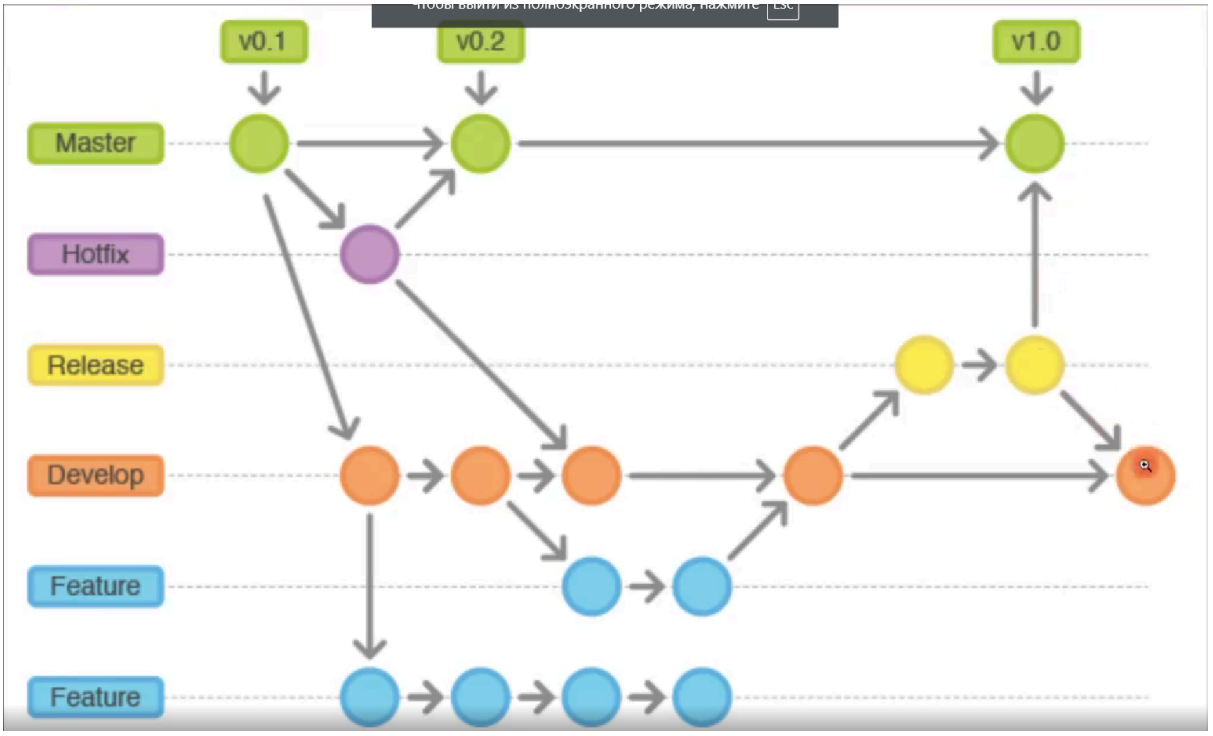
3. Функциональные ветви:
   - Назначение: Эти ветви используются для разработки новых функций.
   - Использование: Функциональная ветвь создается из ветви develop. Как только функция будет завершена, она будет объединена обратно в develop.

4. Ветки выпуска:
   - Назначение: Эти ветки используются для подготовки к новому выпуску.
   - Использование: Ветка выпуска создается из ветки разработки. Здесь вносятся окончательные корректировки и исправления ошибок перед объединением в основную.

5. Ветки исправлений:
   - Назначение: Эти ветки используются для решения срочных проблем в производственном коде.
   - Использование: Ветка исправления создается из основной ветки. После устранения проблемы исправление объединяется обратно в основную и разрабатываемую.

Модель ветвления Git Flow : https://bitworks.software/2019-03-12-gitflow-workflow.html

Еще один пример ветвления Git Flow : https://habr.com/ru/articles/106912/



Этапы рабочего процесса в Git Flow

1. Запуск новой функции:
   - Создайте ветвь функции из ветви разработки.
   - Работайте над функцией, регулярно внося изменения.

2. Завершите работу над функцией:
   - Объедините ветвь функции обратно в ветвь разработки.
   - Разрешите все конфликты, возникающие во время слияния.

3. Подготовка пресс-релиза:
   - Создайте ветку выпуска из ветки разработки.
   - Внесите все окончательные изменения и исправления ошибок в ветку выпуска.
   - Объедините ветку выпуска с основной и пометьте ее тегом версии выпуска.

4. Обработка исправлений:
   - Создайте ветку исправлений из основной ветки для решения неотложных проблем.
   - Примените исправление и объедините ветку исправлений обратно в основную и продолжайте разработку.

Преимущества Git Flow

- Структурированный рабочий процесс: обеспечивает четкую структуру, которая помогает командам эффективно сотрудничать.

- Разделение задач: различает текущую разработку (development), новые функции (feature branches) и релизы (release branches).
- Стабильность: Гарантирует, что основная ветвь остается стабильной и готовой к работе.

Инструменты и интеграция

- Инструменты Git Flow: Существуют такие инструменты, как "git-flow", которые можно установить для автоматизации и упрощения рабочего процесса.
- Интеграция с платформами: Git Flow можно интегрировать с такими платформами, как GitHub или GitLab, для более эффективного управления слияниями, выпусками и совместной работой.

Пример рабочего процесса

1. Создайте ветвь функциональности:
   git checkout develop
   git checkout -b feature/new-feature

2. Поработайте над функцией:
   git add .
   git commit -m "Implement new feature"

3. Объедините функцию с разработкой:
   git checkout develop
   git merge feature/new-feature

4. Создайте ветку выпуска:
   git checkout -b release/v1.0 develop

5. Завершите работу над релизом:
   git add .
   git commit -m "Prepare for release v1.0"

6. Объединить выпуск в Main и пометить тегом:
   git checkout main
   git merge release/v1.0
   git tag -a v1.0 -m "Version 1.0 release"

7. Обработайте исправление:
   git checkout -b hotfix/fix-critical-issue main

git add .
git commit -m "Fix critical issue"
git checkout main
git merge hotfix/fix-critical-issue
git checkout develop
git merge hotfix/fix-critical-issue

Задание № 4 Git Flow для разработки и тестирования приложения на C#

1. Установка Git
Перед началом работы с Git Flow убедитесь, что Git установлен на вашем компьютере. Если Git еще не установлен, скачайте и установите его с официального сайта: https://git-scm.com/.

2. Инициализация репозитория
Откройте командную строку или терминал и перейдите в папку вашего проекта калькулятора на C#. Выполните команду для инициализации репозитория Git:

```bash
**git init**
```

3. Создание основной структуры веток
Git Flow предполагает использование двух основных веток: `master` и `develop`.

- **master**: содержит стабильную версию кода, готовую для релиза.
- **develop**: используется для текущей разработки и тестирования.

Создайте ветку `develop`:

```bash
**git checkout -b develop**
```

4. Работа над новой функцией
Допустим, вы хотите добавить новую функцию в ваш калькулятор, например, вычисление площади круга.

- Создайте ветку `feature` из `develop`:

```bash
**git checkout develop**
**git checkout -b feature/circle-area**
```

```

- В этой ветке вносите изменения в код, добавляйте новую функциональность.
- Сохраняйте изменения с осмысленными сообщениями:

```bash
**git add .**
**git commit -m "Added circle area calculation feature"**
```

5. Слияние ветки `feature` в `develop`
После завершения работы над функцией слейте ветку `feature/circle-area` обратно в `develop`:

```bash
**git checkout develop**
**git merge feature/circle-area**
```

6. Создание релиза
Когда вы готовы выпустить новую версию приложения, создайте ветку `release` из `develop`:

```bash
**git checkout -b release**
```

В этой ветке вносите окончательные настройки, обновляйте номер версии и т.д.

Слейте ветку `release` в `master`:

```bash
**git checkout master**
**git merge release**
```

И добавьте тег для новой версии:

```bash
**git tag -a v1.1.0 -m "Version 1.1.0 release"**
```

Затем слейте изменения обратно в `develop`:

```bash
**git checkout develop**
**git merge release/v1.1.0**
```
```

7. Исправление багов (Hotfix)
Если в стабильной версии (`master`) обнаружена ошибка, создайте ветку `hotfix`:

```bash
**git checkout -b hotfix/bugfix-001**
```

Исправьте баг, committing изменения:

```bash
**git add .**
**git commit -m "Fixed critical bug in circle area calculation"**
```

Слейте изменения в `master` и `develop`:

```bash
**git checkout master**
**git merge hotfix/bugfix-001**
**git checkout develop**
**git merge hotfix/bugfix-001**
```

8. Работа с удаленным репозиторием
Если вы хотите работать с удаленным репозиторием (например, на GitHub), выполните следующие шаги:

- Создайте пустой репозиторий на GitHub.
- Свяжите ваш локальный репозиторий с удаленным:

```bash
**git remote add origin https://github.com/yourusername/yourrepo.git**
```

- Отправьте изменения в удаленный репозиторий:

```bash
**git push -u origin master**
**git push origin develop**
```

9. Тестирование
Интегрируйте тестирование в свой процесс. Для этого можно использовать фреймворки тестирования, такие как MSTest, NUnit или xUnit для C#. Создайте тесты для каждой новой функции.

Пример теста для функции вычисления площади круга:

```csharp
[Test]
public void CircleAreaTest()
{
    // Arrange
    double radius = 5;
    double expectedArea = Math.PI * radius * radius;

    // Act
    double actualArea = Calculator.CircleArea(radius);

    // Assert
    Assert.AreEqual(expectedArea, actualArea, 0.001);
}
```

10. Создаем команды из 3-5 человек, клонируем репозиторий одного из участников, каждый участник добавляет новый метод для класса калькулятор на выбор. Каждый участник тестирует свой метод, после все добавленные новые методы должны оказаться в новом релизе программы.