Основные команды GIT

О чем статья

В этой статье я расскажу о том, как работает ***Git*** , какие основные команды нужно знать, как пользоваться, и как они помогают решать повседневные задачи. Даже если ты только начинаешь свой путь в программировании — здесь всё будет объяснено просто, без сложных терминов и с примерами из жизни.

Что такое GIT

Git — это распределённая система контроля версий, разработанная для отслеживания изменений в исходном коде во время разработки программного обеспечения. Она позволяет нескольким разработчикам работать над одним проектом одновременно, управлять изменениями, создавать различные ветки для экспериментов и слияний, а также легко откатывать изменения при необходимости.

Предустановка

Необходимые инструменты для работы:

1. `git`. Скачать `git` можно по ссылке https://git-scm.com/downloads.
2. Текстовый редактор. Можете использовать любой. В моем случае `VS Code`. Ссылка: https://code.visualstudio.com/download.
3. Зарегистрировать аккаунт на Github и создать новый репозиторий на странице https://github.com/new. Вы можете оставить все по умолчанию, указав только название репозитория. Я же назову его ***`Guide`***. Затем выйдет страница, которая также описывает как начать пользоваться `Git`-ом.
4. Два каталога (папки) для наглядной работы некоторых `git`-команд (имитации командной работы). Я расположил их на рабочем столе и назвал `root` — будем работать в основном в ней, и `addon` — имитация командной работы.

Содержание

Первые шаги, первый push

1. `git init` — Инициализация репозитория
2. `git config` — Настройки параметров конфигурации Git.
3. `git status` — Состояние рабочего каталога и индекса.
4. `git add` — Добавление изменений в индекс.
5. `git reset` — Отмена изменений в репозитории.
6. `git commit` — Сохранение изменений в локальном репозитории Git.
7. `git log` — Просмотр истории коммитов
8. `git push` — Отправка коммитов в удаленный репозиторий

Вторые шаги

9. `git branch` — Управление ветками
10. `git switch` — Переключение между ветками
11. `git clone` — Создание копии удаленного репозитория
12. `git stash` — Временное хранилище
13. `git config alias` — Создание псевдонимов
14. `git checkout` — Работа с ветками, восстановление файлов и переключение на конкретные коммиты
15. `git merge` — Слияние изменений веток
16. `git fetch` — Загрузка обновлений из удаленного репозитория
17. `git pull` — Извлечение и слияние изменений
18. `Работа с текстовым редактором`
19. `git rebase` — Ашалеть, что это за ЗВЕРЬ?
20. `git diff` — Просмотр различий между файлами

Менее юзаемые команды

21. `git difftool` — Просмотр различий и редактирование файлов
22. `git remote` — Работа с удалёнными репозиториями
23. `git tag` — Теги
24. `git restore` — Восстановления файлов из индекса или коммитов
25. `git cherry-pick` — Применение коммитов из одной ветки на другую
26. `git revert` — Откат изменений

Основные команды

Обзор

Вывод

Первые шаги, первый push

1. Инициализация репозитория

```
git init
```

Команда используется для создания нового репозитория Git. Она инициализирует пустой репозиторий в указанной директории, создавая все необходимые файлы и директории для работы с Git.

---

2. Настройки параметров конфигурации Git.

```
git config
```

Команда, используемая для настройки параметров конфигурации Git. Эти параметры могут быть настроены для текущего пользователя, всего репозитория или глобально (флаг `--global`) для всех репозиториев на компьютере.

При помощи следующей команды, Вы укажете имя (ТОЛЬКО для текущего репозитория), которое будет отображаться на отправленных Вами коммитами:

```
git config user.name "Ivan"
```

Если Вы хотите указать имя также и для будущих проектов, то используйте флаг `--global`:

```
git config --global user.name "Ivan"
```

Тоже самое можно проделать и с почтой. Она также будет отображаться в дальнейших коммитах:

```
git config user.email "your@gmail.com"
    или глобально
git config --global user.email "your@gmail.com"
```

Если Вы пропустите этот пункт, не указав `user.email` и `user.name`, то на этапе использования команды `git commit` Вас попросят их ввести.

---

3. Состояние рабочего каталога и индекса

```
git status
```

Команда используется для отображения состояния рабочего каталога и индекса (staging area). Она показывает изменения, которые были добавлены в индекс и готовы для коммита, а также изменения, которые еще не были добавлены. Это помогает разработчикам отслеживать текущие изменения в проекте.

Добавим в текущий каталог файл `index.html`:

```
echo "<p>Hello Git!</p>" > index.html
```

Теперь можно использовать команду `git status`, после которой Вы увидите следующее:

```
git status
On branch master


No commits yet


Untracked files:
  (use "git add <file>..." to include in what will be committed)
        index.html


nothing added to commit but untracked files present (use "git add" to track)
```

Мы можем увидеть:

1. Мы находимся на ветке `master` (On branch master).
2. У нас до сих пор нет коммитов (No commits yet).
3. У нас имеются неотслеживаемые файлы (Untracked files).
4. Ничего не добавлено в commit, но имеются неотслеживаемые файлы (nothing added to commit but untracked files present).

Флаг `-u` отслеживает только `Untracked files`. Используется, когда необходимо увидеть только неотслеживаемые файлы и каталоги.

---

4. Добавление изменений в индекс

```
git add
```

Команда используется для добавления изменений в рабочем каталоге (working directory) в индекс (staging area), подготавливая их для следующего коммита. Это первый шаг в

процессе сохранения изменений в Git.

Чтобы добавить файлы в индекс, мы должны указать какие именно файлы мы хотим добавить. Давайте добавим `index.html`:

```
git add index.html
```

Теперь можем прописать команду `git status`:

```
...
Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   index.html
...
```

Как мы можем видеть, файл index.html добавился в индекс.

Основные команды:

1. `git add index.html index.css` — добавить сразу несколько файлов;
2. `git add . (-A)` — добавить все файлы

---

5. Отмена изменений в репозитории

```
git reset
```

Команда используется для отмены изменений в репозитории. Она может изменять как индекс (staging area), так и рабочий каталог. `git reset` имеет несколько режимов, которые определяют степень отката изменений: `--soft`, `--mixed` и `--hard`.

1. `--soft` — оставляет изменения в рабочем каталоге и в индексе. Перемещает указатель текущей ветки к указанному коммиту.
2. `--mixed` (по умолчанию) — Оставляет изменения в рабочем каталоге, но удаляет их из индекса.
3. `--hard` — удаляет изменения из рабочего каталога и индекса, возвращая состояние файлов к указанному коммиту.
   Будьте аккуратны, когда используете флаг `--hard` !!!
4. `--soft HEAD~1` — откатить коммит, оставить изменения в индексе.
5. `git reset --keep HEAD~1`: откатить, сохранив локальные изменения.

Если Вы хотите удалить какой либо файл (допустим `index.js`) из индекса пропишите:

```
git reset index.js
```

После чего он снова станет неотслеживаемым.

Если хотите удалить все файлы из индекса, просто пропишите `git reset`.

---

6. Сохранение изменений в локальном репозитории Git

```
git commit
```

Команда используется для сохранения изменений в локальном репозитории Git. Когда вы выполняете `git commit`, Git сохраняет текущие изменения, добавленные в индекс (staging area), в виде коммита с уникальным идентификатором (SHA-1 хэшем), который содержит информацию о всех изменениях, авторе, времени и сообщении коммита.

Создадим `commit`:

```
git commit -m "Initial commit"
```

Предположим, что мы забыли добавить index.css. Для начала создадим его, затем добавим его в индекс, используя команды:

```
echo "* { margin: 0; }" > index.css
git add index.css
git commit --amend --no-edit
```

Основные команды:

1. `git commit --amend --no-edit` — изменить коммит без изменения его сообщения.
2. `git commit -a -m "Обновил все изменённые файлы"` — добавление всех изменённых файлов (не добавляет новые файлы) и создание коммита.
3. `git commit --author="Ivan <ivan@example.com>" -m "Добавил новые функции"` — создание коммита с указанием автора.
4. `git commit --allow-empty -m "Пустой коммит"` — создание пустого коммита (без изменений).

---

7. Просмотр истории коммитов

```
git log
```

Команда используется для просмотра истории коммитов в репозитории Git. Эта команда предоставляет информацию о каждом коммите, включая SHA-1 хэш коммита, автора, дату и сообщение коммита. С помощью различных опций можно фильтровать и форматировать вывод для более удобного анализа истории проекта.

Если мы введем данную команду, то увидим следующее:

```
git log


commit c136c5185822bf95b6636851cf6e67a8f732cfeb (HEAD -> master, origin/master)
Author: Ivan <ivan@gmail.com>
Date:   Mon Jul 22 16:30:24 2024 +0700

    Initial commit
```

`c136c5185822bf95b6636851cf6e67a8f732cfeb` является хэшом.

Основные команды:

1. `git log --oneline` — компактный вывод
2. `git log -p` — просмотр коммитов с определенными изменениями
3. `git log -n` — количество выводимых коммитов
4. `git log --graph` — вывод изменений в виде дерева
5. `git log --stat` — вывод статистики изменений
6. `git log --author="Ivan"` — фильтрация коммитов по автору
7. `git log --since="2024-22-07"` (с) и `--until="2024-22-08"` (до) — фильтрация коммитов по дате
8. `git log index.html"` — показывает коммиты, затрагивающие конкретный файл
9. `git log --name-only"` — показывает только имена изменённых файлов

---

8. Отправка коммитов в удаленный репозиторий

```
git push
```

Команда используется для отправки коммитов из локального репозитория в удаленный репозиторий. Эта команда обновляет удаленные рефы вместе с объектами, необходимыми для полного копирования истории изменений.

Если мы введем команду:

```
git push
```

То получим:

```
fatal: No configured push destination.
Either specify the URL from the command-line or configure a remote repository using
```

```
    git remote add <name> <url>

and then push using the remote name

    git push <name>
```

Вот здесь нам как раз-таки и понадобится та самая предустановка, которую мы провели в самом начале.

Вы должны будете скопировать строку с

`https://github.com/<Ваш аккаунт>/<Ваш репозиторий>.git` и добавить удаленный репозиторий:

```
git remote add origin https://github.com/<Ваш аккаунт>/<Ваш репозиторий>.git
```

> Вместо `origin` можете дать свое название, но по умолчанию обычно используется `origin`.

Если мы снова попробуем ввести команду `git push`, то получим:

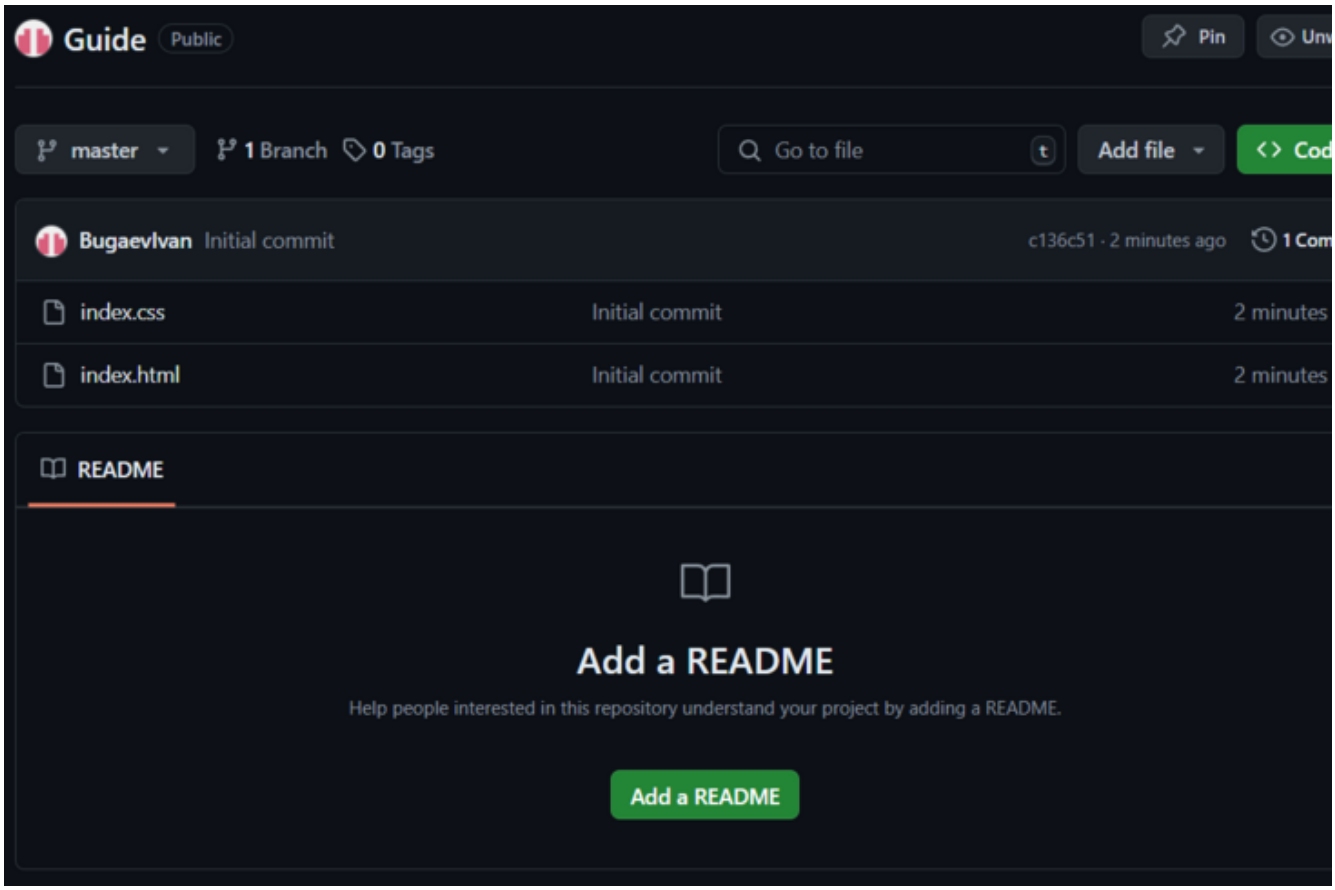
```
fatal: The current branch master has no upstream branch.
To push the current branch and set the remote as upstream, use

    git push --set-upstream origin master

To have this happen automatically for branches without a tracking
upstream, see 'push.autoSetupRemote' in 'git help config'.
```

Сообщение, которое вы получили, означает, что ваша текущая ветка `master` не связана с удаленной веткой в вашем удаленном репозитории (origin). Upstream ветка — это удаленная ветка, с которой ваша локальная ветка синхронизируется при выполнении команд `git push` и `git pull`.

Если вы хотите, чтобы Git автоматически устанавливал upstream ветку при первом пуше новой локальной ветки, вы можете настроить параметр push.autoSetupRemote. Для этого выполните команду:

```
git config --global push.autoSetupRemote true
```

Теперь перейдя по ссылке `https://github.com/<Ваш аккаунт>/<Ваш репозиторий>` Вы увидите следующее:



Поздравляю с Вашим первым push-ем!!!□□□□□

Основные команды:

1. `git push -force` (`-f`) — принудительно перезаписывает историю на удалённом репозитории. **Осторожно!** Может сломать работу других разработчиков.
2. `git push --force-with-lease` — то же, что `--force`, но проверяет, не изменилась ли удалённая ветка за время твоей работы. Более безопасно
3. `git push --set-upstream <удаленный репозиторий> <удаленная ветка>` (`-u`): устанавливает связь между локальной и удалённой веткой. Полезно при первом пушинге новой ветки.

Вторые шаги

---

9. Управление ветками

```
git branch
```

Команда используется для управления ветками в репозитории Git. С её помощью можно создавать, удалять и перечислять ветки. Ветки в Git позволяют вам изолировать работу на различных функциях, исправлениях или версиях проекта.

Так как мы имеем только одну ветку, то при выполнении данной команды увидим:

```
* master
```

Создадим новую ветку `develop`:

```
git branch develop
```

Основные команды:

1. `git branch --delete` (`-d`) — удаляет указанную ветку (только если она объединена с другой)
2. `git branch -D` — принудительно удаляет ветку, даже если она не была объединена
3. `git branch --move` (`-m`) — переименовывает ветку
4. `git branch --all` (`-a`) — показывает все ветки: локальные и удалённые
5. `git branch --remotes` (`-r`) — показывает только удалённые ветки
6. `git branch <name> <commit>` — создаёт ветку на указанном коммите или теге

---

10. Переключение между ветками

```
git switch
```

Команда `git switch` используется для переключения между ветками.

Переключимся на ветку, которую мы только что создали:

```
git switch develop
```

Мы также могли создать ветку при помощи `git switch` и сразу переключиться на нее при помощи флага `-c`:

```
git switch -c develop
```

Добавим файл `index.js`:

```
echo "console.log('Hello')" > index.js
```

Добавим файлы в индекс, сделаем `commit` и `push` данной ветки:

```
git add .
git commit -m "First commit on develop"
git push
```

Теперь перейдя в удаленный репозиторий на сайт, мы увидим, что у нас появилась новая ветка.

Основные команды:

1. `git switch --create <имя>` (`-c`) — создаёт новую ветку и переключается на неё
2. `git switch --track` (`-t`) — устанавливает связь с отслеживаемой удалённой веткой

---

11. Создание копии удаленного репозитория

```
git clone
```

Команда используется для создания копии (клонирования) удаленного репозитория на ваш локальный компьютер. Это один из наиболее распространенных способов начать работу с существующим проектом в Git. При клонировании репозитория Git создает локальную копию всех данных и истории коммитов из удаленного репозитория.

Клонируем наш репозиторий в каталог `addon`, который мы создали на этапе предустановки в 4 пункте. Советую открыть новое окно VS Code и уже в нем клонировать репозиторий.

```
git clone `https://github.com/<Ваш аккаунт>/<Ваш репозиторий>.git` .
```

Если мы выполним команду `git branch -a`, то увидим:

```
* master
  remotes/origin/HEAD -> origin/master
  remotes/origin/develop
  remotes/origin/master
```

Нам нужно клонировать ветку:

```
git switch --track origin/develop
    или
git checkout -b develop origin/develop
    или можно было сразу
git clone -b develop https://github.com/<Ваш аккаунт>/<Ваш репозиторий>.git` .
```

---

12. Временное хранилище

```
git stash
```

Команда используется для временного сохранения (stash) изменений в рабочем каталоге, которые еще не готовы для коммита. Это полезно, когда нужно переключиться на другую ветку или работу, но текущие изменения не должны быть потеряны. С помощью `git stash` изменения сохраняются в специальное место, откуда их можно восстановить позже.

> ***Отступление***
>
> Представим ситуацию:
>
> Вы офисный работник, выполняете 3-ий день подряд Task-у с Azure (TFS) `#111 Добавление печати пропуска` вместе со своими коллегами и сидите на локальной ветке `feature/111_add_print`, которой соответствует удаленная `origin/feature/111_add_print`. Вы уже перелопатили половину кода, что-то в текущий момент перестало работать.
>
> Как вдруг вбегает начальник и говорит: "Мы теряем ***бабки***!!! Бросай свои дела и быстро исправляй ошибки! Кто умудрился в `console.log()` выводить пароль?".
>
> У Вас код перестал работать, а он со своими проблемами. Не пошлешь же начальника, хотя скорее всего очень хочется. Что ж, придется исправлять. Для этого нужно создать новую ветку `hotfix/change_clg`, переключиться на нее и убрать эту лишнюю строку.
>
> Давайте попробуем это сделать.

1. Для начала создадим удаленную ветку `origin/feature/111_add_print`. Для этого переключимся на новую ветку `feature/111_add_print`, добавим немного кода в `index.js` и в `index.css`, сделаем коммит и запушим:

```
git switch -c feature/111_add_print


echo "function Print(){}" >> index.js
echo "p { text-align: center; }" >> index.css


git add .
git commit -m "Add print function"
git push
```

2. "Перелопатим" код (добавим изменения в файл `index.js`):

```
echo "const y = true;" >> index.js
echo "console.log(y)" >> index.js
```

3. Так как нам нужно исправлять код рабочий — наследоваться от ветки `develop`, а не от `feature/111_add_print`, то нам для начала необходимо переключиться на develop и от него создать ветку `hotfix/change_clg`:

   ```
   git switch develop


   error: Your local changes to the following files would be overwritten by checkout:
           index.js
   Please commit your changes or stash them before you switch branches.
   Aborting
   ```

   Но увидим ошибку, которая говорит о том, что мы сделали локальные изменения и их нужно либо запушить, либо отправить в `stash`, прежде чем мы сможем переключиться.
4. Для этого используем команду:

   ```
   git stash
   ```

   Если мы пропишем `git status`, то увидим, что наши изменения пропали. На деле они находятся в `stash`.
   Теперь мы сможем переключиться на `develop`.

Основные команды:

1. `git stash` — отправка изменений в стэш;
2. `git stash push -m "Временные изменения"` — Отправка изменений в стэш с указанием сообщения;
3. `git stash list` — список сохранённых стэшей;
4. `git stash apply` — применение последнего стэш;
5. `git stash pop` — применение и удаление последнего стэша;
6. `git stash apply stash@{n}` — применение конкретного стэша;
7. `git stash drop stash@{n}` — если 0, то удаление всех стэшей, другое число — удаление определенного стэша;
8. `git stash clear` — очистка всех стэшей.

---

13. Создание псевдонимов

```
git config alias
```

Данная подкоманда используется для создания псевдонимов команд, которые позволяют заменить одним словом целую строку. Псевдонимы упрощают и ускоряют выполнение часто используемых команд.

Если Вы заметили, то мы часто используем команду `git commit -m ""`. Мы можем упростить ее написание создав для нее алиас:

```
git config --global alias.ctm 'commit -m'
```

Теперь вместо `commit -m` мы будем использовать `ctm`.

Создадим еще парочку алиасов:

```
git config --global alias.st status
git config --global alias.lg "log --oneline --graph --decorate --all"
```

Получить список созданных алиасов можно командой:

```
git config --get-regexp alias
```

Удалить ненужный алиас (допустим `ctm`):

```
git config --global --unset alias.ctm
```

---

14. Старший брат команды git switch

```
git checkout
```

Команда используется для различных целей, таких как переключение между ветками, создание новых веток, восстановление файлов и переключение на конкретные коммиты.

Переключимся на новую ветку, унаследовавшись от последнего коммита в `develop`:

```
git log develop --oneline -1
126dadc (HEAD -> develop, origin/develop) First commit on branch develop
```

Во 2-ой строке получили хэш. Теперь создаем ветку на основе этого хэша:

```
git checkout -b hotfix/change_clg 126dadc
```

Изменим файл `index.js` на:

```
console.log("Вы успешно вошли")
```

Сделаем commit и push при помощи созданного алиаса:

```
git add .
git ctm "Убрал `pass` в console.log()"
```

```
git push
```

Основные команды:

1. `git checkout <branch-name>` — создание новой ветки
2. `git checkout -b <branch-name>` — создание новой ветки и переключение на нее
3. `git checkout <commit-hash>` — переключение на конкретный коммит
4. `git checkout <branch-name> <commit-hash>` — создание новой ветки от определенного коммита
5. `git checkout -- <file-path>` — восстановление файла до состояния последнего коммита
6. `git checkout -- .` — восстановление всех файлов до состояния последнего коммита

---

15. Слияние изменений веток

```
git merge
```

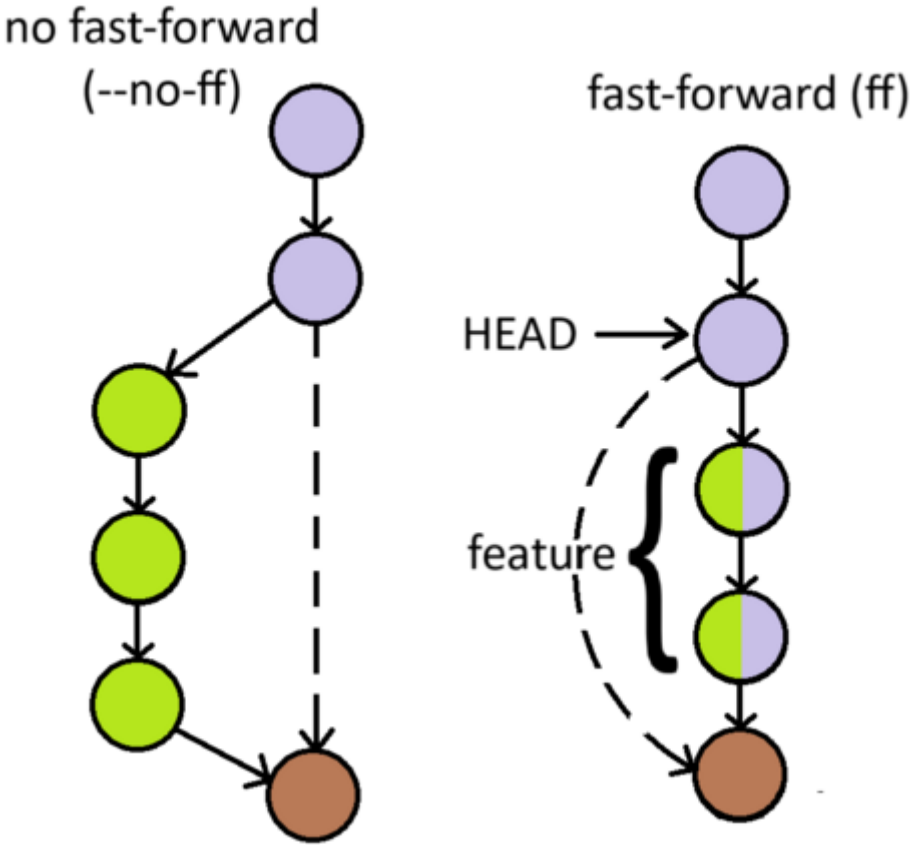
Команда используется для слияния изменений из одной ветки в другую. Эта команда объединяет истории коммитов двух веток, создавая новый коммит (по умолчанию), который включает изменения из обеих веток.

Вольем наши изменения из ветки `hotfix/change_clg` в `develop`:

```
git switch develop
git merge --no-ff hotfix/change_clg -m "Слияние изменений из hotfix/change_clg в develop"
git push
```

Очень важно писать `--no-ff`, чтобы сохранить структуру разработки.

Наглядный пример работы:



Теперь на ветке `develop` в удаленном и локальном репозитории файлы будут обновлены.

Основные команды:

1. `git merge <from-branch> --no-ff` — создаёт новый коммит слияния, даже если возможно быстрое слияние (fast-forward). Полезно для сохранения истории слияния.
2. `git merge <from-branch> --no-commit` — объединяет изменения, но не делает коммит автоматически.
3. `git merge <from-branch> --squash` — объединяет все коммиты из сливаемой ветки в один коммит в текущей ветке. Полезно для объединения истории коммитов перед слиянием.
4. `git merge --continue` — продолжает слияние после разрешения конфликтов
5. `git merge --abort` — прерывает начатое слияние при возникновении конфликтов

---

16. Загрузка обновлений из удаленного репозитория

```
git fetch
```

Команда используется для загрузки обновлений из удаленного репозитория в ваш локальный репозиторий. В отличие от `git pull`, `git fetch` не сливает изменения автоматически в вашу текущую ветку. Вместо этого она загружает все обновления, которые затем можно просмотреть и слить вручную.

Переключимся обратно на каталог `root`.

Загрузим обновления из удаленного репозитория командой `git fetch`.

В консоли видим:

```
...
  652b6ff..b585d2a  develop               -> origin/develop
* [new branch]      feature/111_add_print -> origin/feature/111_add_print
* [new branch]      hotfix/change_clg    -> origin/hotfix/change_clg
```

Здесь видим краткую информацию, что `develop` обновился и появились две новые ветки

Пишем `git log -1 --oneline` и видим, что коммит не соответствует тому, что должно быть в удаленном репозитории.

Пишем `git log origin/develop -1 --oneline` и видим последний коммит.

Если запускать с флагом `--prune`, то он уберет локальные ссылки на удалённые ветки, которых уже нет на сервере.

---

17. Извлечение и слияние изменений

```
git pull
```

Команда используется для извлечения и слияния изменений из удалённого репозитория в вашу текущую ветку. Эта команда объединяет две операции: `git fetch` и `git merge`. Сначала она извлекает изменения из удалённого репозитория (аналогично `git fetch`), а затем автоматически сливает их с текущей веткой (аналогично `git merge`).

Выполним команду `git pull` и наша локальная ветка `develop` будет соответствовать удаленной ветке `origin/develop`.

Переключимся на ветку `feature/111_add_print` и вытянем изменения, которые мы сделали в `develop`:

```
git switch -t origin/feature/111_add_print
git merge develop
```

Как мы можем увидеть, у нас конфликты:

```
warning: Cannot merge binary files: index.js (HEAD vs. develop)
Auto-merging index.js
CONFLICT (content): Merge conflict in index.js
Automatic merge failed; fix conflicts and then commit the result.
```

Сам же файл:

```
<<<<<<< HEAD
console.log("Вы успешно вошли")
=======
console.log(pass)
function Print(){}
>>>>>>> 2c0279a (Add print function)
```

Вы можете отредактировать файл вручную, удалив строки `<<<<<<< HEAD`, `=======`, `>>>>>>> 742b087 (Add print function)` и внести нужные изменения, сохраняя только те строки, которые вы хотите оставить.

Или Вам предложат открыть `Merge editor` (возможно автоматически откроется), чтобы решить данный конфликт. В нем `current` будет ветка `develop`, `incoming` — `feature/111_add_print`, а `result` — результат слияния. Можно выбрать оба изменения, либо одно, нажав на соответствующие галочки, либо написать что-то свое и нажать на `Complete Merge`. Я добавлю оба изменения:

```
console.log("Вы успешно вошли")
function Print(){}
```

Если мы пропишем `git status`, то увидим:

```
git status
On branch feature/111_add_print
Your branch is up to date with 'origin/feature/111_add_print'.

All conflicts fixed but you are still merging.
  (use "git commit" to conclude merge)

Changes to be committed:
        modified:   index.js
```

Как видно мы в состояния `merging` (you are still merging), нужно завершить `merge`, просто пишем:

```
git merge --continue
```

---

18. Работа с текстовым редактором vim

У вас откроется редактор, в котором Вы можете поменять сообщение коммита. Строки начинающиеся с `#` считаются комментариями. По умолчанию `merge` создает `commit`: `Merge branch 'develop' into feature/111_add_print`. Чтобы изменить его, нужно перейти в интерактивный режим — нажмите `i`, выйти из режима без сохранения `:q!`. Чтобы сохранить изменения необходимо нажать клавишу `Esc` затем ввести `:wq!` (Посимвольно означает: Перейти в командный режим + Записать + Выйти + Принудительно выполнить).

Оставлю название `commit` по умолчанию. Осталось только запушить эти изменения.

```
git push
```

Переключимся на `addon`. Пропишем `git switch feature/111_add_print`, `git pull`. У нас подтянутся изменения — строка с `console.log`.

Наконец-то возвратимся к изменениям, которые сохраняли в стэш. Применим их:

```
git stash pop
```

Опять конфликты, но мы на опыте, решаем их.

Добавим изменения в индекс, закоммитим и запушим:

```
git add .
git ctm "Make print"
git push
```

Вольем наши изменения в ветку `develop` и запушим:

```
git switch develop
git merge --no-ff feature/111_add_print -m "Слияние изменений из feature/111_add_print в
develop"
git push
```

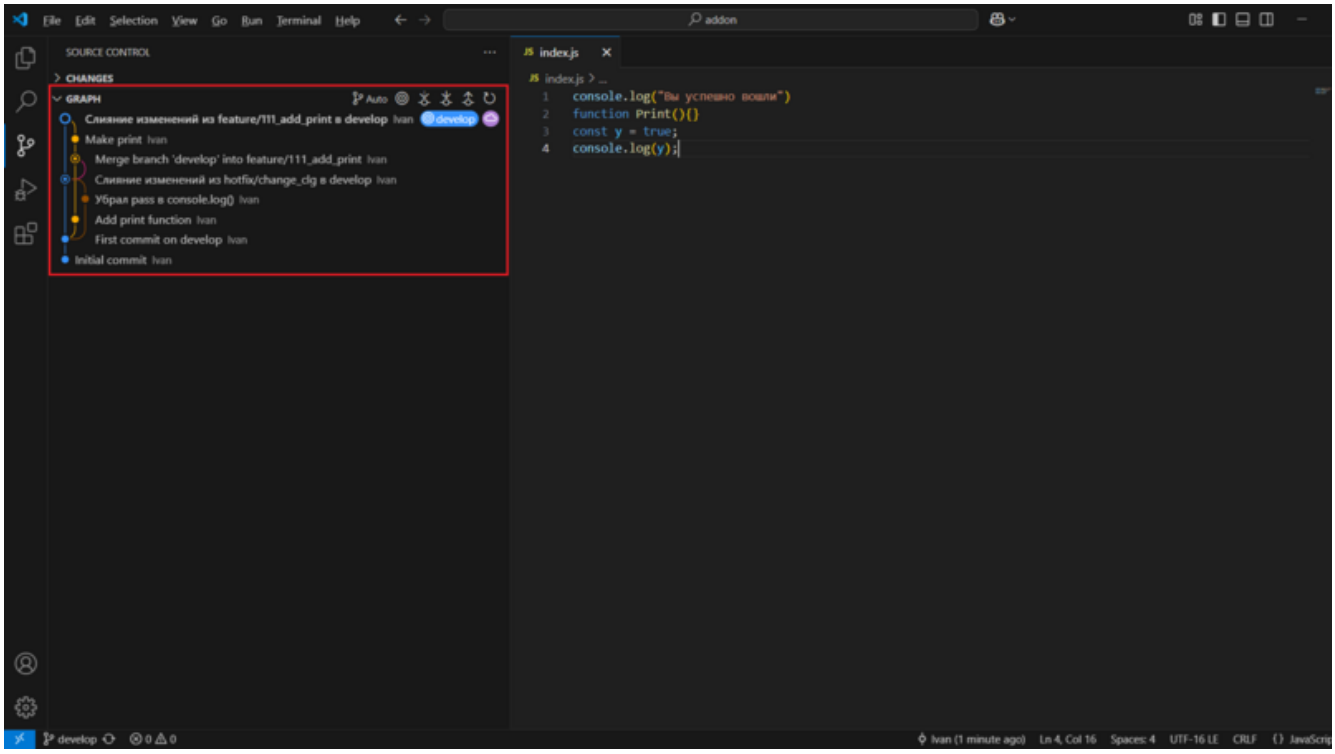
---

19. Ашалеть, что это за ЗВЕРЬ?

```
git rebase
```

Команда используется для интеграции изменений из одной ветки в другую. Она позволяет перенести или "переписать" серию коммитов из одной ветки в другую, сохраняя линейную историю. Это полезно для упрощения истории проекта и устранения ненужных коммитов слияния.

В Visual Studio Code (VS Code) можно наглядно увидеть графическую визуализацию истории коммитов на ветке `develop`:



Или при помощи этих команд:

```
git log --oneline
    или
git log --oneline --graph --decorate --all
```

Возьмем последние два коммита:

```
git rebase -i HEAD~2
```

Выйдет такая страница:

```
pick abac516 Add print function
pick 5201b7f Убрал pass в console.log()
pick 8699ff2 Make print

# Rebase 1e6c9c2..aeba1fd onto 1e6c9c2 (2 commands)
#
# Commands:
# p, pick <commit> = use commit
# r, reword <commit> = use commit, but edit the commit message
# e, edit <commit> = use commit, but stop for amending
```

```
# s, squash <commit> = use commit, but meld into previous commit
# f, fixup [-C | -c] <commit> = like "squash" but keep only the previous
#                    commit's log message, unless -C is used, in which case
#                    keep only this commit's message; -c is same as -C but
#                    opens the editor
# x, exec <command> = run command (the rest of the line) using shell
# b, break = stop here (continue rebase later with 'git rebase --continue')
# d, drop <commit> = remove commit
...
```

Основные команды:

1. `p, pick <commit>` — оставить коммит как есть
2. `r, reword <commit>` — оставить изменения, но изменить сообщение коммита
3. `e, edit <commit>` — остановиться на этом коммите для правок (можно добавить/изменить файлы)
4. `s, squash <commit>` — объединить с предыдущим коммитом (сохраняя сообщение текущего)
5. `f, fixup [-C | -c] <commit>` — объединить с предыдущим, но игнорировать его сообщение
6. `d, drop <commit>` — удалить коммит (опасно — приведёт к потере изменений)

> Если ты хочешь изменить историю, но не удалить изменения, то никогда не используй drop или fixup без понимания последствий.

Как мы можем увидеть у нас не попали `merge` (слияния веток). Merge `commit`-ы — это специальные коммиты, которые Git обрабатывает особо при `rebase`.

Попробуем переименовать один из коммитов. О том как пользоваться `vim` найдете в Работа с текстовым редактором *vim*:

```
pick abac516 Add print function
r 5201b7f Убрал pass в console.log()
pick 8699ff2 Make print
...
```

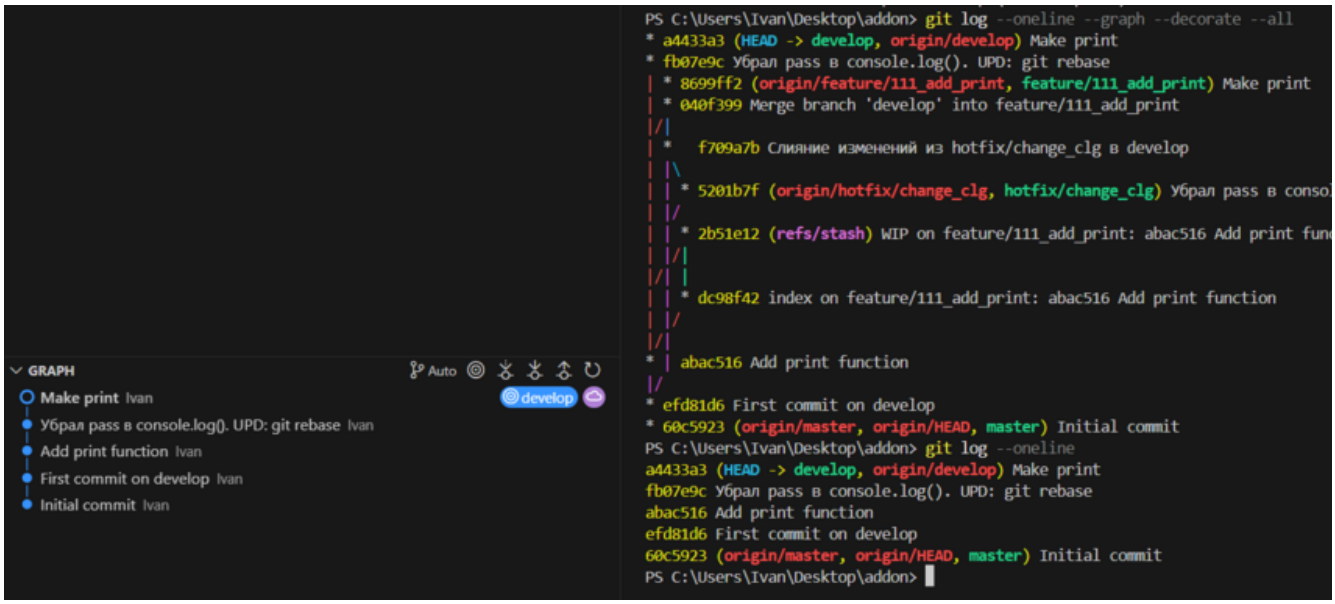
У нас появится опять конфликт, решаем, пишем `git rebase --continue`, и далее меняем сообщение в редакторе у `commit` на:

```
Убрал pass в console.log(). UPD: git rebase
```

Делаем отправку:

```
git push --force-with-lease
```

`--force-with-lease` безопаснее, чем `--force`: он проверит, не поменялась ли удалённая ветка за время твоей работы.



Как мы видим многие коммиты потерлись. Это происходит потому, что цель `rebase` — сделать историю линейной, а не сохранять информацию о слияниях.

`git rebase` лучше всего использовать, когда ты работаешь один над своей веткой, не публичной и тебе нужно изменить коммит, его сообщение, объединить несколько в один. Но как показывает практика, чтобы применять эту команду — нужно больше работы с ней.

Конечно, можно вернуться к предыдущему состоянию, но оно уже будет немного видоизменено:

```
git reset --hard origin/develop
git merge feature/111_add_print
git push
```

---

20. Просмотр различий между файлами

```
git diff
```

Команда используется для отображения различий между состояниями файлов в вашем репозитории Git. Она позволяет сравнивать изменения, которые произошли в файлах между различными состояниями, такими как рабочий каталог, индекс (staging area) и коммиты.

Давайте добавим различия, что увидеть работу данной команды. Добавим в файл `index.html` строку:

```
echo "<span>What is diff!</span>" >> index.html
```

И в файл `index.css`:

```
echo "span { color: red; }" >> index.css
```

При выполнении команды `git diff` получим:

```
diff --git a/index.css b/index.css
index 6dd919a..9e7b7bf 100644
Binary files a/index.css and b/index.css differ
diff --git a/index.html b/index.html
index db400e4..8ea75d9 100644
Binary files a/index.html and b/index.html differ
```

Если у Вас что-то подобное значит у Вас файл сохранен в неверном формате.

Если напишем команду `git diff --text`, то увидим строки, которые нельзя прочитать. Все дело в кодировке и русские символы будут видны как кракозябры. Исправить это можно, сохранив файл с определенной кодировкой.

> В VS CODE меняется так: нажмите `Ctrl+Shift+P`, введите `Change File Encoding`, затем выберите `Save with Encoding` и наконец-то `UTF-8`.

После изменения вызов `git diff` покажет читаемую информацию:

```
diff --git a/index.css b/index.css
index b36d0b9..99d90d6 100644
--- a/index.css
+++ b/index.css
@@ -1,2 +1,3 @@
 * { margin: 0; }
 p { text-align: center; }
+span { color: red; }
diff --git a/index.html b/index.html
index 410a29a..acd0531 100644
--- a/index.html
+++ b/index.html
@@ -1 +1,2 @@
 <p>Hello Git!</p>
+<span>What is diff!</span>
```

Если измененные файлы добавлены в индекс, то точно такую же информацию можно увидеть при выполнении команды:

```
git status -v
```

При сравнении с веткой `master` вдобавок получим `index.js`:

```
git diff master


...
+<p>I made my first push!</p>
\ No newline at end of file
diff --git a/index.js b/index.js
new file mode 100644
index 0000000..43d2c75
--- /dev/null
+++ b/index.js
@@ -0,0 +1 @@
+console.log(pass)
```

Основные команды:

1. `git diff index.html` — вывести изменения только для определенного файла.
2. `git diff --staged` — просмотр изменений, добавленных в индекс
3. `git diff <commit1> <commit2>` — сравнение изменений между двумя коммитами по их хэшам
4. `git diff <branch1> <branch2>` — сравнение изменений между ветками
5. `git diff --name-only` — вывести только имена файлов, в которых произошли изменения.
6. `git diff -- *.css` — сравнивает все файлы с расширением `.css` .

Менее юзаемые команды

Далее будут описаны команды, менее употребляемые при написании кода, но тоже важные.

---

21. Просмотр различий и редактирование файлов

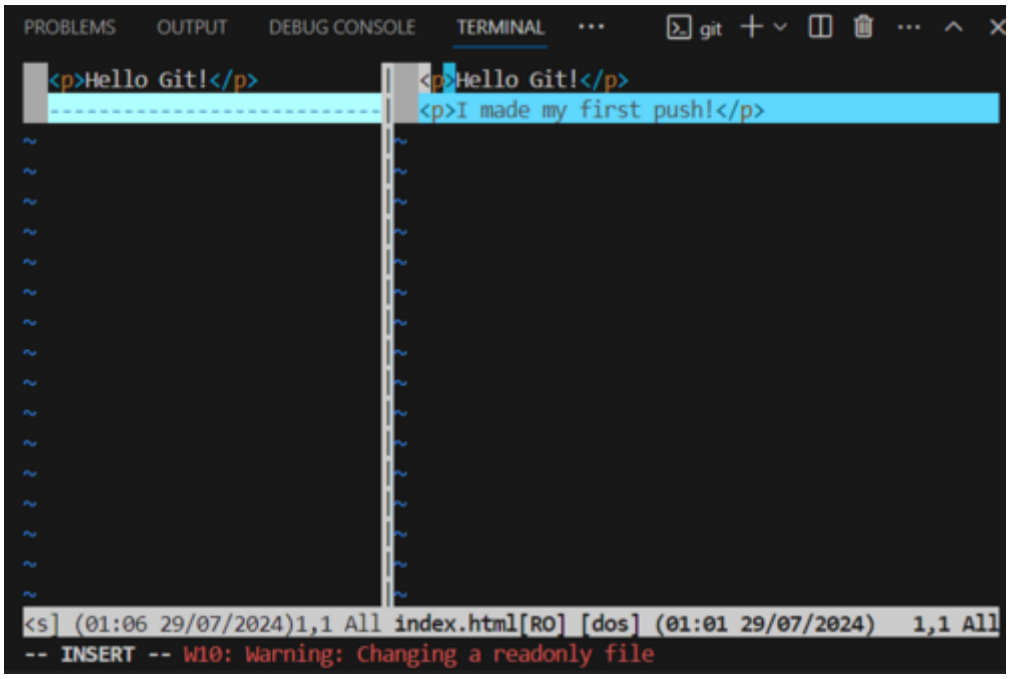
```
git difftool
```

Команда позволяет использовать внешние дифф-утилиты для просмотра различий между файлами, а также менять код внутри файла.
Команда `git difftool` является дополнением команды `git diff`, но главное отличие — она умеет также и редактировать файлы.

Теперь используем команду.

```
This message is displayed because 'diff.tool' is not configured.
See 'git difftool --tool-help' or 'git help config' for more details.
'git difftool' will now attempt to use one of the following tools:
kompare emerge vimdiff nvimdiff

Viewing (1/2): 'index.css'
Launch 'vimdiff' [Y/n]?
```

Вводим `Y` и `Enter`, попадаем в редактор ***vim*** для редактирования файла.



Как работать в данном редакторе описано ранее в [Работа с текстовым редактором `vim`]:

При помощи данной команды мы также можем сравнивать ветки, коммиты и отдельные файлы.

Основные команды:

`git difftool --tool-help`: Показывает список доступных инструментов. Так как я работаю в VS Code, то в списке доступных покажется `vscode`. Вы можете установить его как дефолтный инструмент командой:

```
git config --global diff.tool vscode
```

И тогда будет открываться не в консоли, а во вкладке VS Code. Закрытие вкладки — выход из difftool.

---

22. Работа с удалёнными репозиториями

```
git remote
```

Команда `git remote` используется для управления удалёнными репозиториями в Git.

Основные команды:

1. `git remote -v` — выводит список имён удалённых репозиториев с URL-ом.
2. `git remote add <name> <url>` — добавление удалённого репозитория
3. `git remote remove <name>` — удаление удалённого репозитория
4. `git remote rename <old-name> <new-name>` — переименование удалённого репозитория

5. `git remote set-url <name> <new-url>` — изменение URL удалённого репозитория

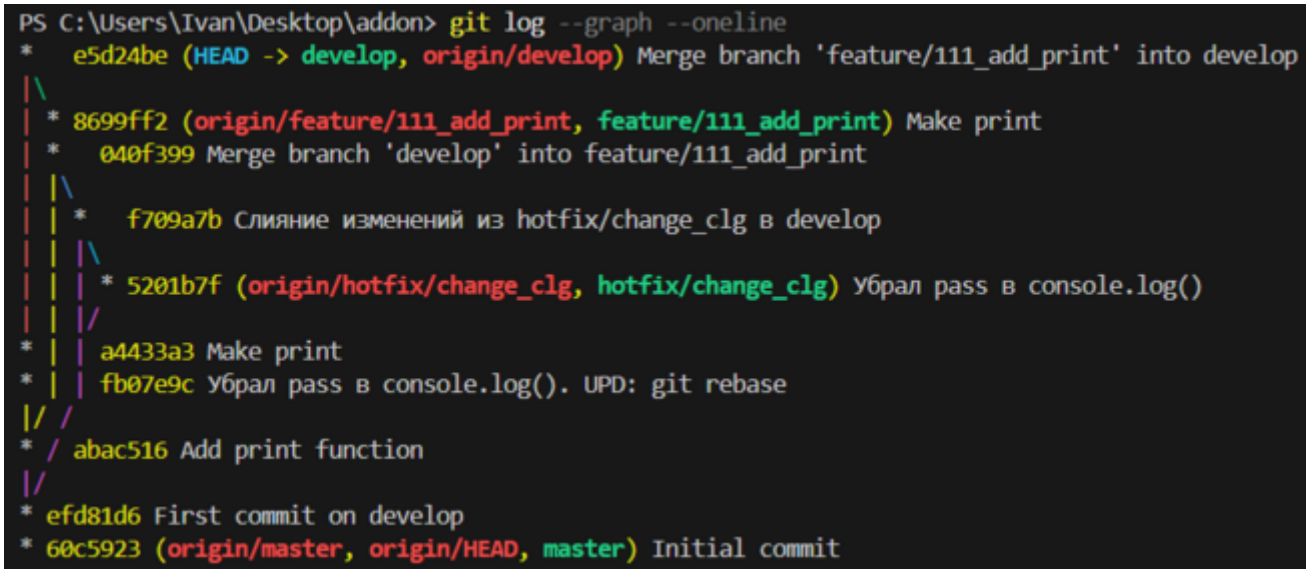
---

23. Теги

```
git tag
```

Команда `git tag` используется для создания, просмотра и управления тегами в Git. Теги — это метки, которые обычно используются для обозначения стабильных версий проекта (например, `v1.0.0`, `v2.1.3` и т.д.).

Выведем все коммиты:



Добавим для коммита `fb07e9c Убрал pass в console.log(). UPD: git rebase` тег:

```
git tag -a v1.0.0 fb07e9c -m "Рабочая версия v1.0.0"
```

Команда `git tag -n` выводит информацию:

```
v1.0.0          Рабочая версия v1.0.0
```

Команда `git show v1.0.0` выводит информацию:

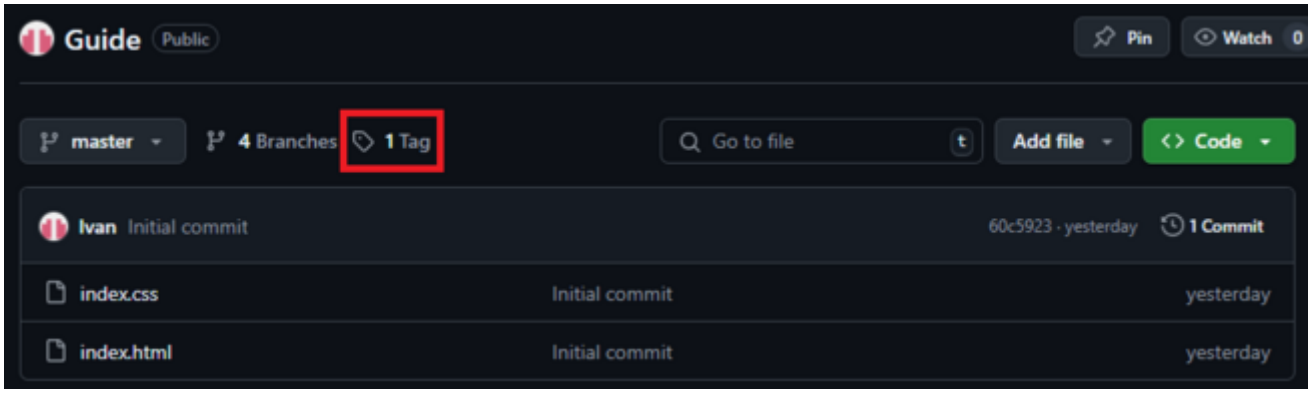
```
tag v1.0.0
Tagger: Ivan <ivan@gmail.com>
Date:   Fri Jun 13 20:20:22 2025 +0300

Рабочая версия v1.0.0

commit fb07e9cdd9c778bb52bb4b3b08f9fc37f512dba4 (tag: v1.0.0)
```

```
Author: Ivan <ivan@gmail.com>
Date:   Thu Jun 12 23:37:23 2025 +0300

    Убрал pass в console.log(). UPD: git rebase

diff --git a/index.js b/index.js
index 2db8952..913f3b6 100644
Binary files a/index.js and b/index.js differ
```

Отправим в репозиторий тег:

```
git push origin v1.0.0
```

Теперь перейдя в удаленный репозиторий, мы увидим наш тег:



Основные команды:

1. `git tag` — выводит список всех тегов
2. `git tag -a <tagname> -m "message"` — создаёт аннотированный тег с сообщением
3. `git tag <tagname> <commit>` — создаёт тег на определённом коммите
4. `git show <tagname>` — показывает информацию о теге: автор, дата, сообщение, изменения
5. `git tag -n` — показывает список тегов с кратким сообщением (если есть)
6. `git push origin <tagname>` — отправляет указанный тег на удалённый репозиторий
7. `git tag -d <tagname>` — удаляет тег из локального репозитория
8. `git push origin --delete <tagname>` — удаляет тег из удалённого репозитория

---

24. Восстановления файлов из индекса или коммитов

```
git restore
```

Команда `git restore` предназначена для восстановления файлов из индекса или коммитов, заменяя собой некоторые функции `git checkout` и `git reset`. Она делает работу с откатом

изменений более понятной и явной.

Основные команды:

1. `git restore <file>` — отменяет локальные изменения. Никак не влияет, если файл в индексе
2. `git restore --staged <file>` — убирает файл из индекса (не трогает рабочую директорию)
3. `git restore --staged .` — убирает все файлы из индекса (не трогает рабочую директорию)
4. `git restore -- .` — похоже на команду `git checkout -- .`. Восстанавливает все файлы в текущем каталоге и подкаталогах

---

25. Применение коммитов из одной ветки на другую

```
git cherry-pick
```

Команда `git cherry-pick` позволяет применить один или несколько конкретных коммитов из одной ветки на другую. Это удобно, если ты хочешь перенести только определённые изменения, а не всю ветку целиком.

Переключимся на ветку `master` для наглядной демонстрации, получим коммиты `develop` и выполним:

```
git log develop --oneline --reverse
60c5923 (HEAD -> master, origin/master, origin/HEAD) Initial commit
efd81d6 First commit on develop
abac516 Add print function
5201b7f (origin/hotfix/change_clg, hotfix/change_clg) Убрал pass в console.log()
f709a7b Слияние изменений из hotfix/change_clg в develop
040f399 Merge branch 'develop' into feature/111_add_print
8699ff2 (origin/feature/111_add_print, feature/111_add_print) Make print
fb07e9c (tag: v1.0.0) Убрал pass в console.log(). UPD: git rebase
a4433a3 Make print
e5d24be (origin/develop, develop) Merge branch 'feature/111_add_print' into develop
```

Возьмем допустим коммит `8699ff2`:

```
git cherry-pick --edit 8699ff2
```

Откроется редактор, пропишем `Cherry-pick from feature/111_add_print` и запушим `git push`. Как видим, наша ветка стянула состояние ветки `feature/111_add_print`, имеющееся на данном коммите.

Если необходимо откатить коммит и убрать его из истории, то просто делаем следующие шаги:

```
git log --oneline
3dd9bb9 (HEAD -> master, origin/master, origin/HEAD) Cherry-pick from develop
60c5923 Initial commit


git reset --hard 60c5923
HEAD is now at 60c5923 Initial commit


git push -f
```

Основные команды:

1. `git cherry-pick <commit>` — применяет указанный коммит на текущей ветке
2. `git cherry-pick <commit1>..<commit2>` — применяет все коммиты между `commit1` и `commit2` (включая `commit2`, исключая `commit1`)
3. `git cherry-pick <commit1>^..<commit2>` — применяет все коммиты от `commit1` до `commit2` включительно
4. `git cherry-pick --edit (-e)` — позволяет редактировать сообщение коммита перед применением
5. `git cherry-pick --no-commit (--n)` — применяет изменения в индекс, но не создаёт коммит
6. `git cherry-pick --continue` — продолжает процесс после разрешения конфликтов
7. `git cherry-pick --abort` — отменяет cherry-pick и возвращает всё к исходному состоянию
8. `git cherry-pick --quit` — выходит из процесса cherry-pick без изменений

---

26. Откат изменений

```
git revert
```

Команда `git revert` используется для отката изменений, создавая новый коммит, который "отменяет" изменения, внесённые ранее указанным коммитом. Это безопасный способ отката, потому что он не переписывает историю (в отличие от `git reset`).

Добавим какие-нибудь данные `echo "<span>git revert</p>" > index.html` и запушим. Мы можем откатить последний коммит через HEAD или через его хэш:

```
git log --oneline
f5ecd54 (HEAD -> master, origin/master, origin/HEAD) Git revert test
60c5923 Initial commit
```

```
git revert HEAD
    или
git revert f5ecd54
```

В редакторе можем поменять сообщение коммита у `revert` и можем пушить. Сам же файл `index.html` откатился к предыдущей версии.

Основные команды:

1. `git revert <commit>` — откатывает указанный коммит, создавая новый коммит с обратными изменениями
2. `git revert --edit (-e)` — позволяет редактировать сообщение коммита перед сохранением
3. `git revert --no-edit` — применяет revert без изменения сообщения
4. `git revert --no-commit (-n)` — применяет изменения в рабочую область, но не создаёт коммит
5. `git revert --continue` — продолжает процесс после разрешения конфликтов
6. `git revert --abort` — прерывает процесс revert и возвращает всё к исходному состоянию
7. `git revert --quit` — выходит из режима revert без изменений
8. `git revert --mainline <n>` — используется при откате merge-коммита; указывает, какую родительскую ветку считать основной (например, 1 или 2)

Основные команды

1. Клонирование репозитория — `git clone https://github.com/<Ваш аккаунт>/<Ваш репозиторий>.git .`
2. Список имён удалённых репозиториев с URL-ом — `git remote -v`
3. Добавление файлов в индекс — `git add .`
4. Удаление файлов из индекса — `git reset`
5. Создание коммита — `git commit -m "Сообщение коммита"`
6. Добавить забытые файлы в предыдущий коммит без изменения сообщения:

   ```
   git add .
   git commit --amend --no-edit
   ```

7. Отправить изменения в удаленный репозиторий — `git push`
8. Создание ветки и переключение на нее — `git switch -c <новая_ветка>`
9. Просмотр локальных и удалённых веток — `git branch -a`
10. Принудительное удаление ветки — `git branch -D <ветка>`
11. Переименовать ветку — `git branch --move (-m) <старая_ветка> <новая_ветка>`
12. Состояние рабочего каталога и индекса — `git status`
13. Просмотр истории коммитов в удобном виде — `git log --graph --oneline`

14. Восстановление всех файлов, не вошедших в индекс, до состояния последнего коммита — `git checkout -- .`
15. Убирает все файлы из индекса — `git restore --staged .`. `.` - все файлы, можно заменить на название определенного файла.
16. Слияние изменений веток — `git merge --no-ff <ветка, которую хотим слить в текущую> -m "<Сообщение>"`
17. Извлечение и слияние изменений — `git pull`
18. Удаление файла из индекса (staging area) и из remote репозитория при следующем push, но сохранение в рабочей директории — `git rm --cached <имя-файла>`. `-r` флаг действует для папок.
19. Если необходимо откатить коммит и убрать его из истории, то просто делаем следующие шаги:

```
git log --oneline
3dd9bb9 (HEAD -> master, origin/master, origin/HEAD) Cherry-pick from develop
60c5923 Initial commit


git reset --hard 60c5923
HEAD is now at 60c5923 Initial commit


git push -f
```

Вывод

Вот и всё! Мы разобрали основные команды Git, которые помогут тебе эффективно управлять проектами и работать с кодом. Надеюсь, эта статья помогла тебе разобраться в базовых принципах работы с Git.

Revision #7
Created 2026-02-05 06:39:10 UTC by Эд Кукса
Updated 2026-02-05 07:53:27 UTC by Эд Кукса