

# Система контроля версий Git

## 1 Введение в системы контроля версий

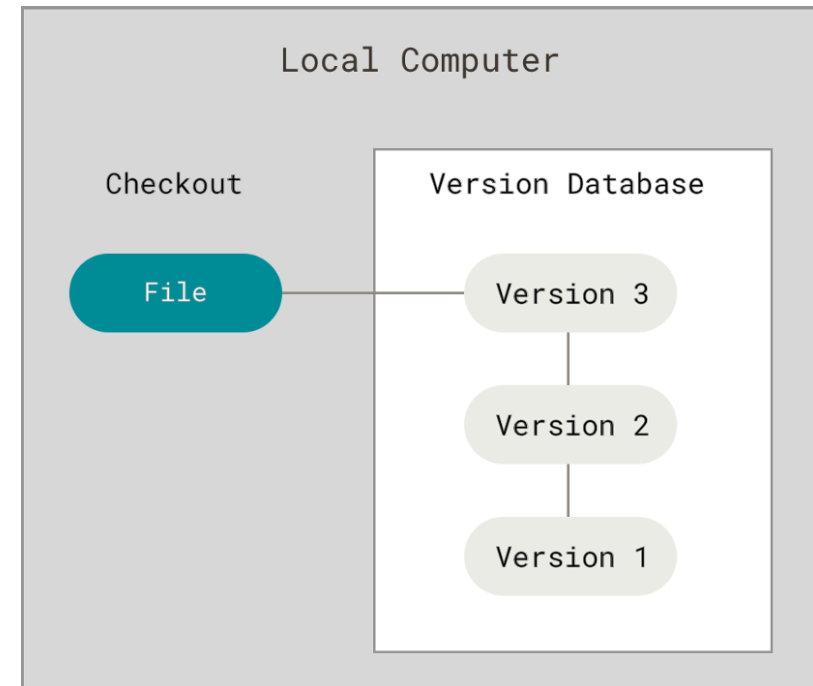
Представьте себе команду разработчиков, которая одновременно работает над большим проектом: одни исправляют баги, другие добавляют новые функции, третьи обновляют документацию. Без надёжной системы учёта изменений быстро возникает хаос: чьи правки самые свежие, как объединить правки разных участников, и как вернуться к более ранней, «рабочей» версии кода, если что-то пошло не так? Именно для решения этих задач и существуют **системы контроля версий (Version Control Systems, VCS)**. Системы контроля версий позволяют:

- сохранять историю каждого изменения в проекте;
- работать над одним и тем же кодом нескольким людям параллельно, минимизируя риски потери работы и конфликтов;
- быстро откатиться к любой предыдущей версии проекта, если новая реализация оказалась неправильной или сломала что-то важное;
- анализировать, кто и когда внёс то или иное изменение.

Типы систем контроля версий:

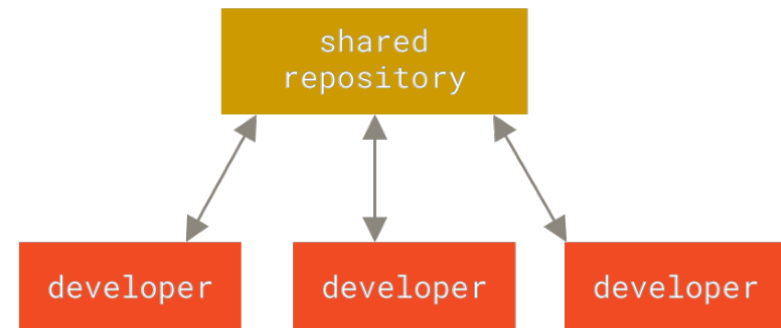
### 1. Локальные

Всё хранится на компьютере разработчика — удобно, но чревато проблемами при совместной работе, нужно вручную копировать файлы коллегам.



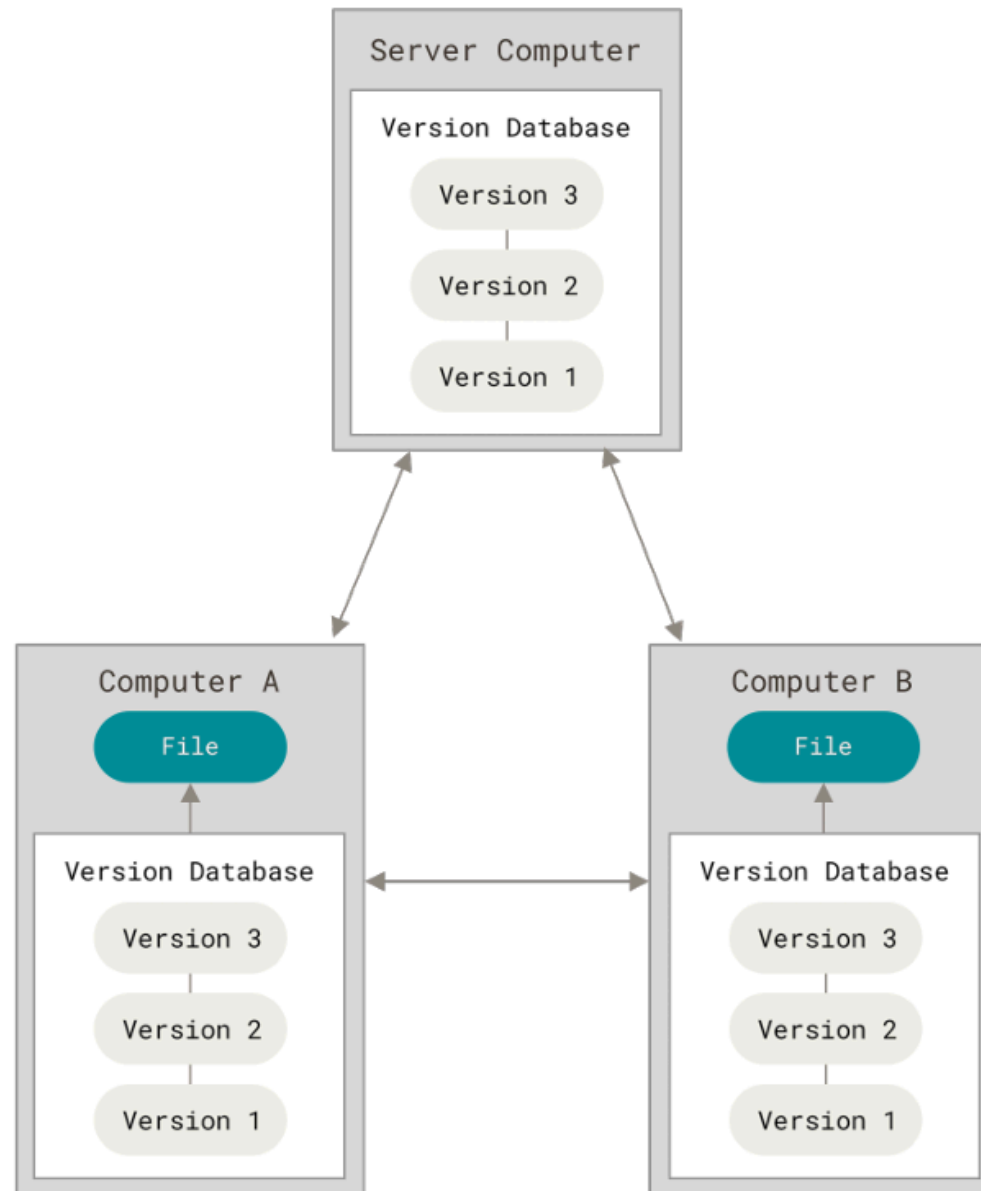
### 2. Централизованные

Есть единый сервер-репозиторий, к которому все подключаются, история изменений хранится централизованно. Главный плюс — не возникает конфликтов версий, минус — при недоступности сервера вся работа останавливается.



### 3. Распределённые

Каждый разработчик имеет полную копию всего репозитория, включая всю историю. Они могут вести разработку параллельно и независимо друг от друга в режиме офлайн, периодически проводя синхронизацию.



Основные термины:

- **Репозиторий** — хранилище проекта под управлением системы контроля версий. Содержит файлы, историю изменений и служебную информацию.
- **Коммит (commit)** — зафиксированное изменение в истории проекта. Каждый коммит содержит снимок состояния файлов и сообщение, описывающее внесённые правки.
- **Ветка (branch)** — независимая линия разработки.
- **Слияние (merge)** — объединение изменений из одной ветки в другую.

**Почему именно Git?**

**Git разработан в 2005 году Линусом Торвальдсом** — создателем ядра Linux. Первоначально инструмент создавался для управления исходниками ядра Linux, где ежедневно вносятся тысячи изменений сотнями участников по всему миру. **Git является распределенной системой контроля версий.** На данный момент умение пользоваться Git является необходимым навыком для любого программиста. Ключевые преимущества Git:

- **Скорость и эффективность.** Операции коммита, ветвления и слияния работают мгновенно, даже в очень больших по размеру репозиториях.
- **Гибкая модель ветвления.** Ветки в Git «лёгкие» и дешёвые по ресурсам, что стимулирует активное использование веток для каждой новой фичи или исправления.
- **Полная локальная копия.** Каждый разработчик имеет свой «клон» репозитория со всей историей, что защищает проект от проблем с центральным сервером.
- **Широкая экосистема.** Над Git построены сервисы (GitHub, GitLab, Bitbucket), графические клиенты, плагины для IDE.
- **Прозрачная история.** Мощные инструменты визуализации, анализа и поиска изменений позволяют легко понимать, кто и зачем внёс те или иные правки.



---

## 2 Основы Git

### 2.1 Установка Git

Скачать Git можно с официального сайта [Git - Downloads](#). После установки нужно будет провести первоначальную настройку. Первое, что вам следует сделать после установки Git — указать ваше имя и адрес электронной почты. Это важно, потому что каждый коммит в Git содержит эту информацию, и она включена в коммиты, передаваемые вами, и не может быть далее изменена. Команды нужно вводить в командной строке.

```
git config --global user.name "Имя Фамилия"  
git config --global user.email myemail@example.com
```

## 2.2 Создание репозитория

Репозиторий Git можно создать одним из двух способов:

1. Вы можете взять локальный каталог (папку), и превратить его в репозиторий Git
2. Вы можете клонировать существующий репозиторий Git из любого места.

Для создания репозитория необходимо открыть командную строку и перейти в каталог, в котором вы собираетесь его создать. В Windows это можно сделать с помощью команды `cd`.

```
cd C:/Users/user/my_project
```

Затем вводите команду для инициализации репозитория:

```
git init
```

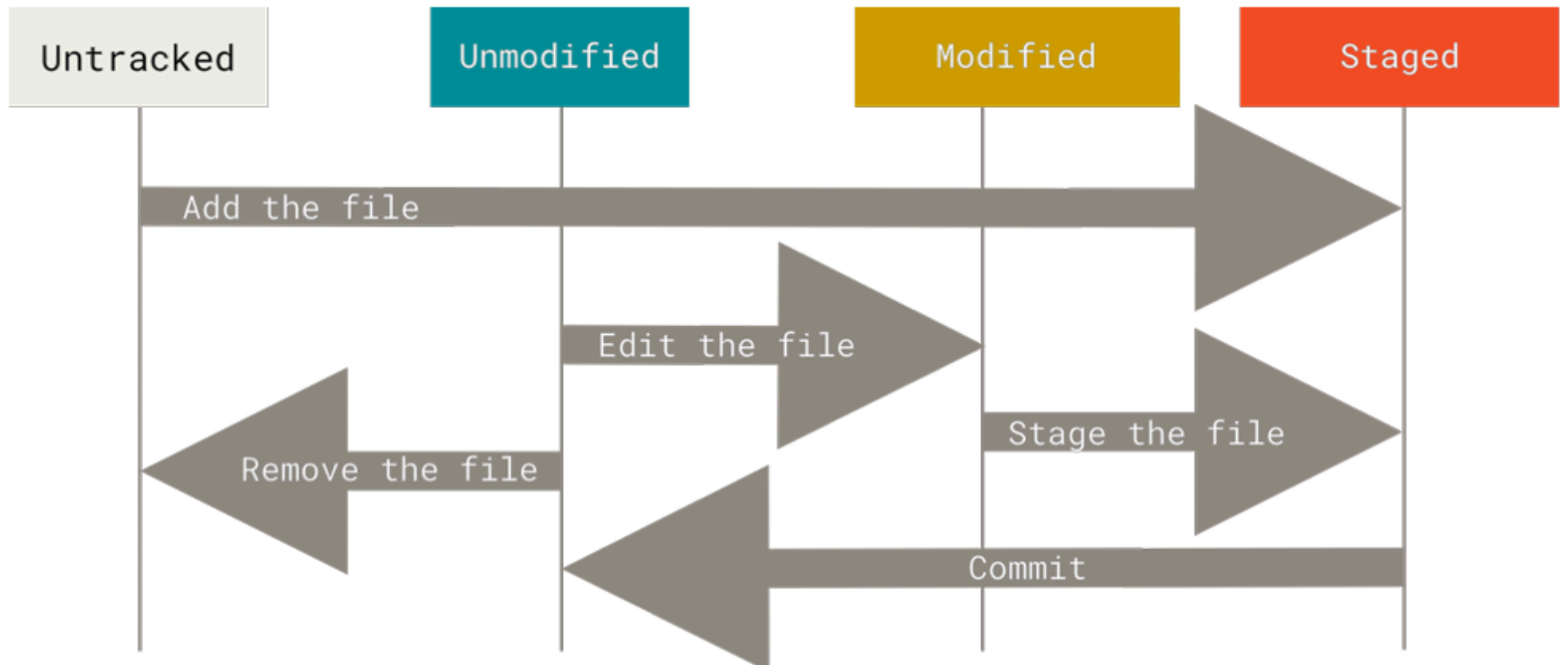
Эта команда создаёт в текущем каталоге новый подкаталог с именем `.git`, содержащий все необходимые файлы репозитория — структуру Git репозитория.

## 2.3 Запись изменений в репозиторий

Каждый файл в рабочем каталоге может находиться в одном из двух состояний: под версионным контролем (отслеживаемые) и нет (неотслеживаемые).

**Отслеживаемые файлы** — это те файлы, которые были в последнем снимке состояния проекта, они могут быть **неизменёнными (unmodified)**, **изменёнными (modified)** или **подготовленными к коммиту (staged)**. Если кратко, то отслеживаемые файлы — это те файлы, о которых знает Git.

**Неотслеживаемые файлы** — это всё остальное, любые файлы в вашем рабочем каталоге, которые не входили в ваш последний снимок состояния и не подготовлены к коммиту. Когда вы впервые создаете репозиторий все файлы будут неотслеживаемыми.



Для того, чтобы понять, какой статус у каждого файла в репозитории нужно использовать команду:

```
git status
```

Для того, чтобы добавить неотслеживаемые файлы в индекс (staged). Нужно использовать команду:

```
git add <file1> <file2> ...
```

Либо

```
git add .
```

Чтобы добавить сразу все неотслеживаемые файлы.

После этой команды можно будет зафиксировать изменения с помощью коммита:

```
git commit -m "Короткое, ёмкое сообщение о сути изменений"
```

Вкратце процесс создания коммита выглядит следующим образом:

1. Вносим изменения в каталоге (добавляем, удаляем, изменяем файлы).
2. Добавляем в индекс `git add`
3. Фиксируем с помощью коммита `git commit`

Для отмены последнего коммита нужно использовать команду:

```
git commit --amend
```

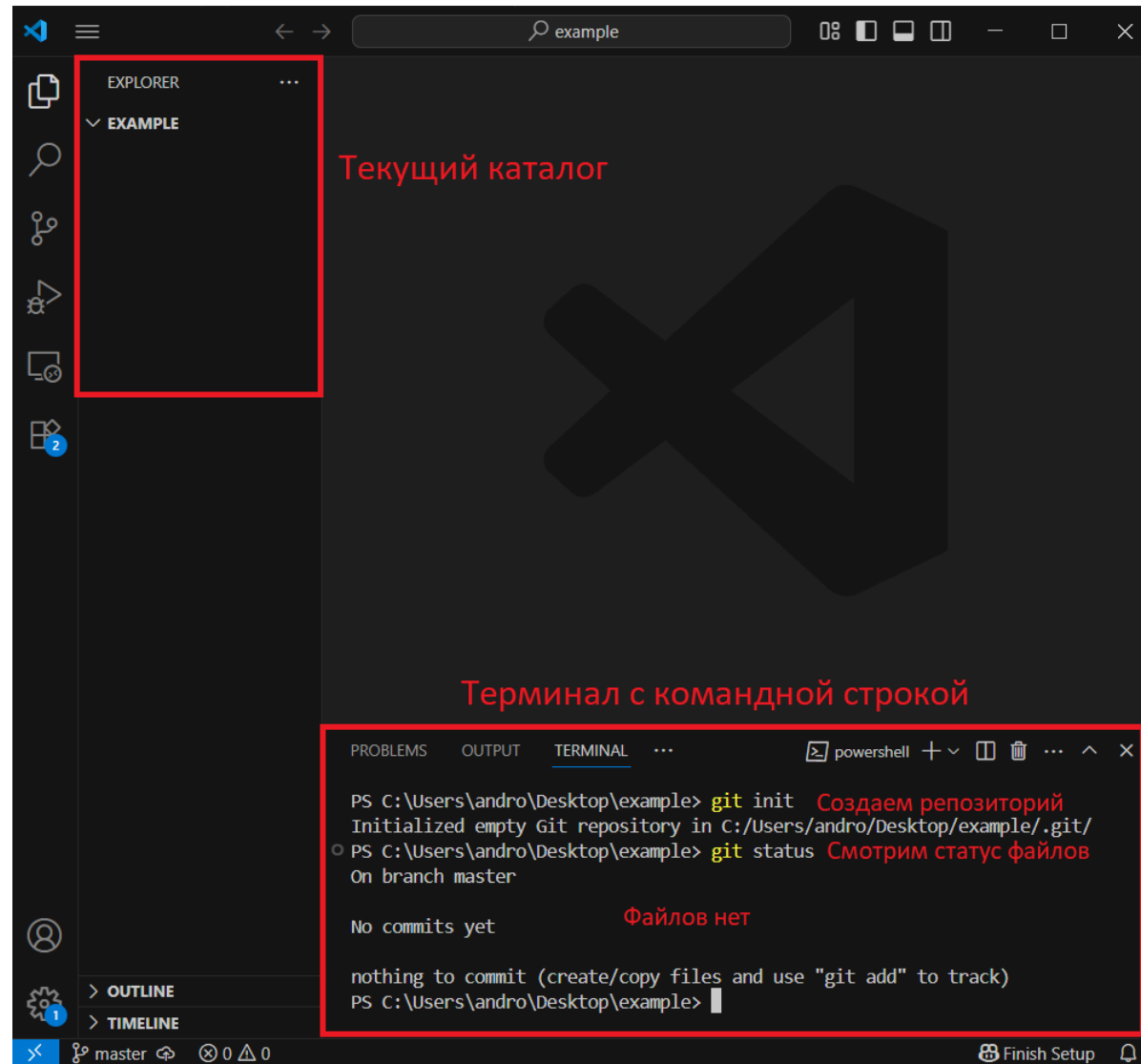
К другим полезным командам можно отнести `git diff`, который позволяет увидеть изменения в файлах относительно последнего коммита и `git log` который позволяет увидеть историю коммитов.

Для указания файлов и папок, которые Git не должен отслеживать и включать в коммиты используется файл `.gitignore`. Это особенно полезно для исключения временных файлов, логов, конфигураций среды, скомпилированных бинарников и других данных, не относящихся к исходному коду проекта. Git читает `.gitignore` построчно: каждая строка описывает шаблон имени (например, `*.log` или `node_modules/`). Эти файлы остаются в вашей рабочей директории, но Git будет их игнорировать при добавлении (`git add`) и не включать в коммиты, если они не были добавлены ранее.



## 2.4 Практический пример

Пример работы с Git с помощью VS Code



Visual Studio Code interface showing a project named 'example' with three untracked files: 'text1.txt', 'text2.txt', and 'text3.txt'.

**EXPLORER**

- ✓ EXAMPLE
  - text1.txt U
  - text2.txt U
  - text3.txt U

**EDITOR**

text3.txt

1

**TERMINAL**

```
PS C:\Users\andro\Desktop\example> git status
On branch master

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)
    text1.txt
    text2.txt
    text3.txt

nothing added to commit but untracked files present (use "git add" to track)
PS C:\Users\andro\Desktop\example>
```

**Annotations:**

- Добавили несколько файлов (Added several files) - points to the Explorer view.
- Есть 3 неотслеживаемых файла (There are 3 untracked files) - points to the Untracked files section in the terminal output.

Visual Studio Code interface showing a terminal window with Git commands and their output. The Explorer sidebar on the left shows a folder named "EXAM..." containing three files: "text1.txt", "text2.txt", and "text3.txt". The terminal window at the bottom shows the execution of "git add .", "git status", and "git commit -m \"first commit\"".

Terminal Output:

```
PS C:\Users\andro\Desktop\example> git add .
PS C:\Users\andro\Desktop\example> git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   text1.txt
        new file:   text2.txt
        new file:   text3.txt

PS C:\Users\andro\Desktop\example> git commit -m "first commit"
[master (root-commit) 9879c5a] first commit
3 files changed, 1 insertion(+)
 create mode 100644 text1.txt
 create mode 100644 text2.txt
 create mode 100644 text3.txt

PS C:\Users\andro\Desktop\example> git status
On branch master
nothing to commit, working tree clean

PS C:\Users\andro\Desktop\example>
```

Annotations in the terminal output:

- nothing added to commit but untracked files present (use "git add" to track)
- PS C:\Users\andro\Desktop\example> **git add .** Добавили все файлы в индекс
- PS C:\Users\andro\Desktop\example> **git status**
- On branch master
- No commits yet
- Changes to be committed:
- (use "git rm --cached <file>..." to unstage)
- new file: text1.txt
- new file: text2.txt
- new file: text3.txt
- Теперь они все отслеживаются
- PS C:\Users\andro\Desktop\example> **git commit -m "first commit"** Сделали КОММИТ
- [master (root-commit) 9879c5a] first commit
- 3 files changed, 1 insertion(+)
- create mode 100644 text1.txt
- create mode 100644 text2.txt
- create mode 100644 text3.txt
- PS C:\Users\andro\Desktop\example> **git status** Теперь файлы стали unmodified
- On branch master
- nothing to commit, working tree clean
- PS C:\Users\andro\Desktop\example>

Status bar at the bottom: master 0 0 Ln 1, Col 2 Spaces: 4 UTF-8 CRLF {} Plain Text Finish Setup Prettier

```
PS C:\Users\andro\Desktop\example> git diff
diff --git a/text2.txt b/text2.txt
index f04fd8d..f541c22 100644
--- a/text2.txt
+++ b/text2.txt
@@ -1,1 @@
-qwertyuiop
\ No newline at end of file
+qwertyuiopfgfgfgfgfgff
\ No newline at end of file
```

```
PS C:\Users\andro\Desktop\example> git log
commit bdfa440e9a11ca65a24a65314a5f2b74054d580e (HEAD -> master)
Author: OrlovAndrei <androrvit@gmail.com>
Date: Sat Jun 28 18:29:00 2025 +0300

    second commit

commit 9879c5a24feb81095c4a9c675b159403337af484
Author: OrlovAndrei <androrvit@gmail.com>
Date: Sat Jun 28 18:16:36 2025 +0300

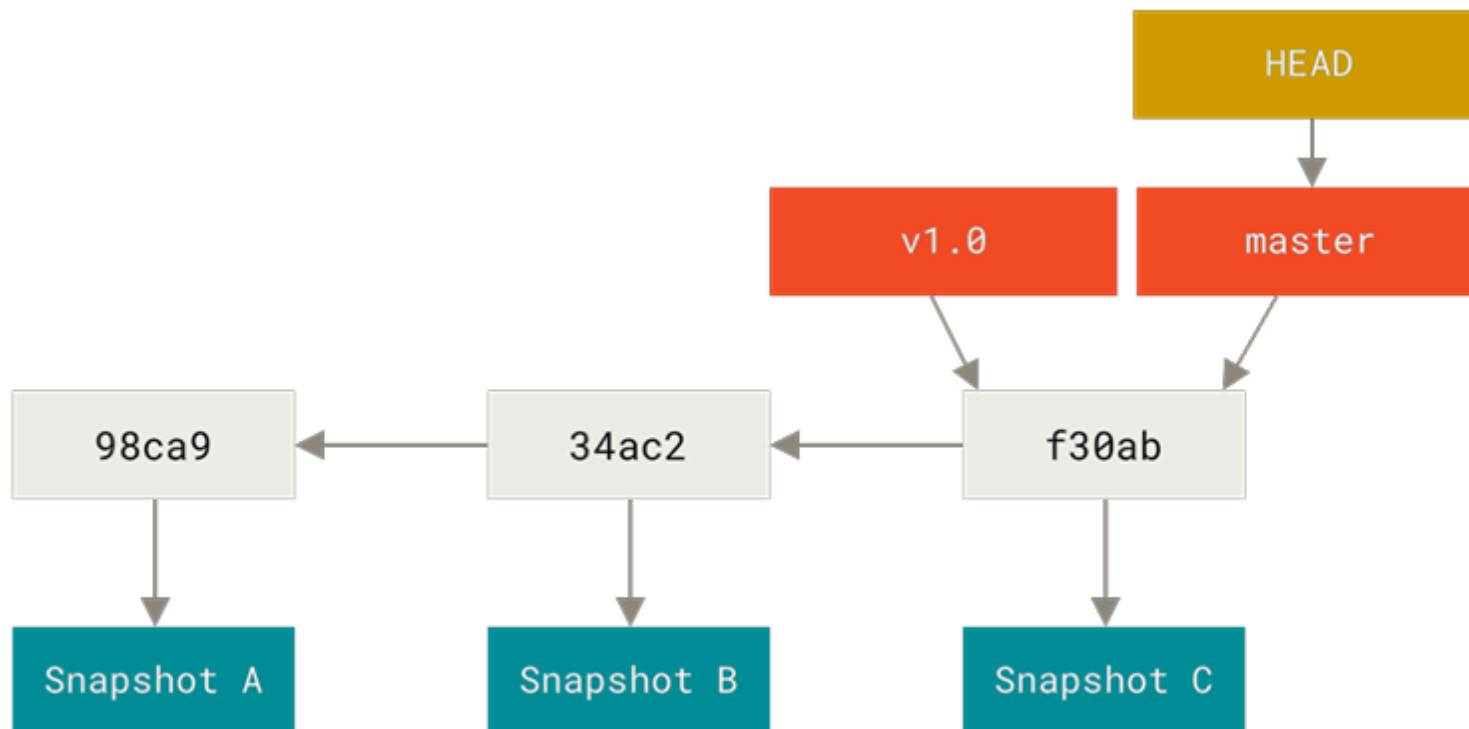
    first commit
```

### 3 Ветки

Ветки позволяют создавать изолированные копии основной линии разработки, где вы продолжаете работу независимо от неё. Для точного понимания механизма ветвлений, необходимо вернуться назад и изучить то, как Git хранит данные.

Когда вы делаете коммит, Git сохраняет его в виде объекта, у которого есть указатель на коммит или коммиты непосредственно предшествующие данному. Ветка в Git — это простой перемещаемый указатель на один из таких коммитов. По умолчанию, имя основной ветки в Git — `master`. Как только вы начнёте создавать коммиты, ветка `master` будет всегда указывать на последний коммит. Каждый раз при создании коммита указатель ветки `master` будет передвигаться на следующий коммит автоматически.

Рассмотрим пример, у нас есть три последовательных коммита, которые ссылаются на предыдущий. И две ветки `v1.0` и `master`, которые указывают на один и тот же последний коммит (ветки одинаковые). Так же есть указатель `HEAD`, который указывает на ту ветку, на которой мы сейчас находимся.

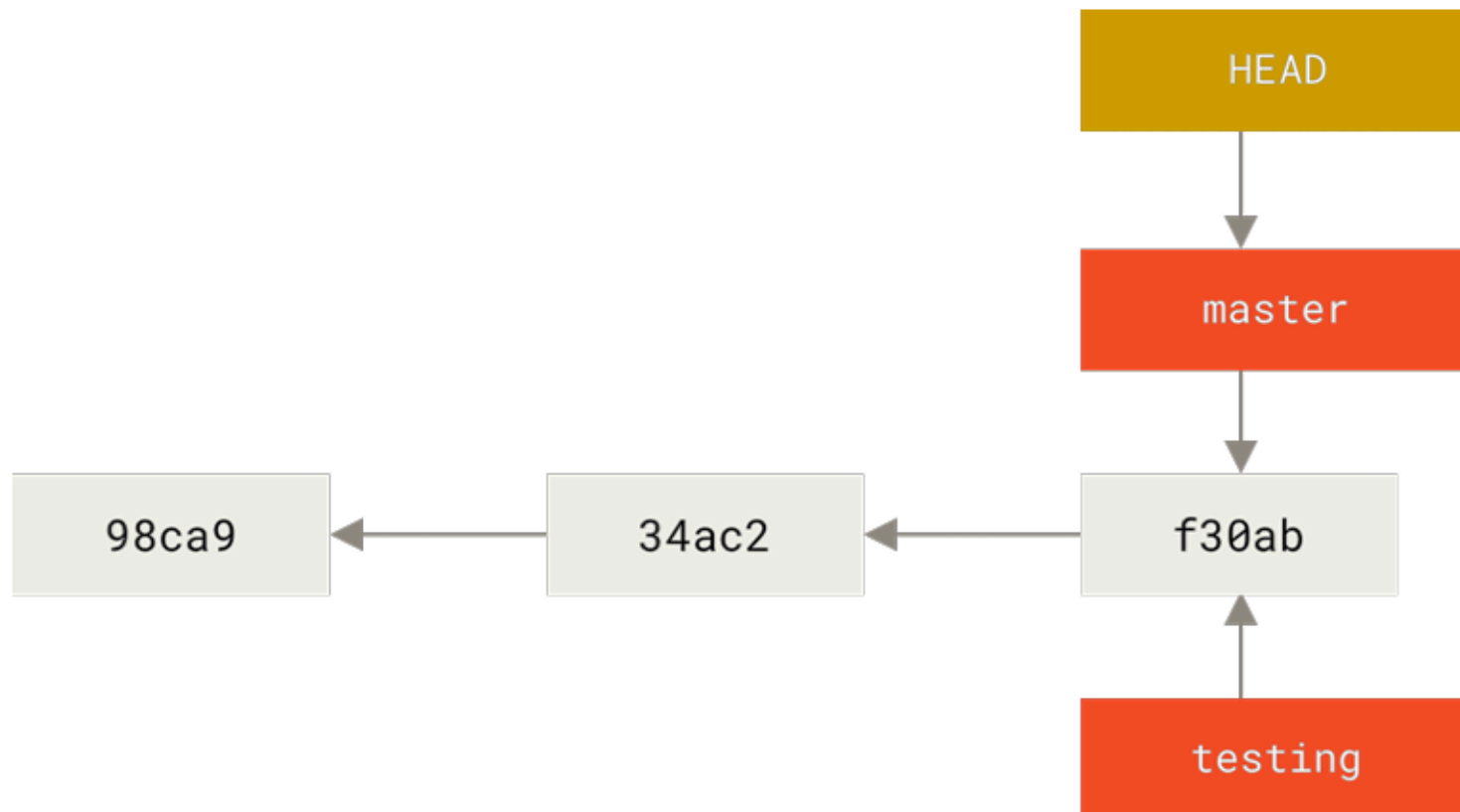


## 3.1 Создание ветки

Для создания новой ветки нужно ввести команду:

```
git branch <название ветки>
```

Это создаст новый указатель, на тот коммит, на котором вы находитесь сейчас.



Но при этом указатель HEAD не перенесется автоматически и вы останетесь на той же ветке.

Для того, чтобы переключиться на новую ветку, нужно ввести команду:

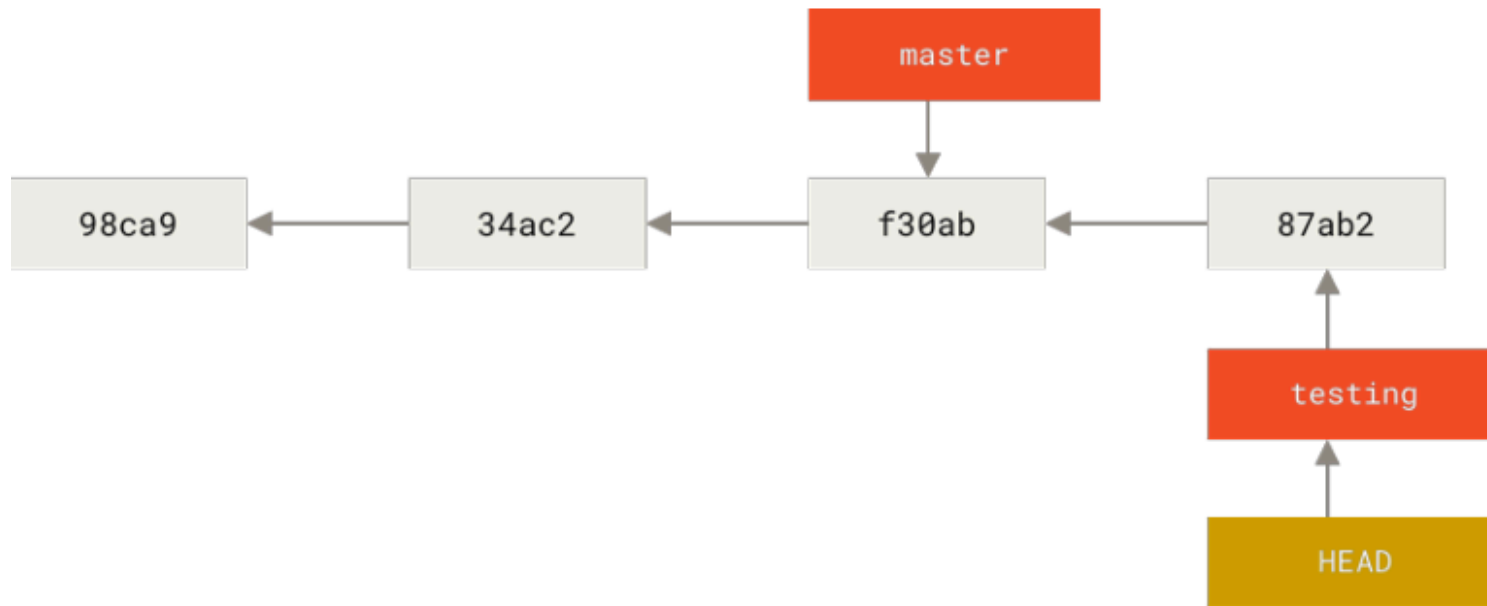
```
git checkout <название ветки>
```

Так же можно написать такую команду для того, чтобы сразу создать и переключиться на новую ветку:

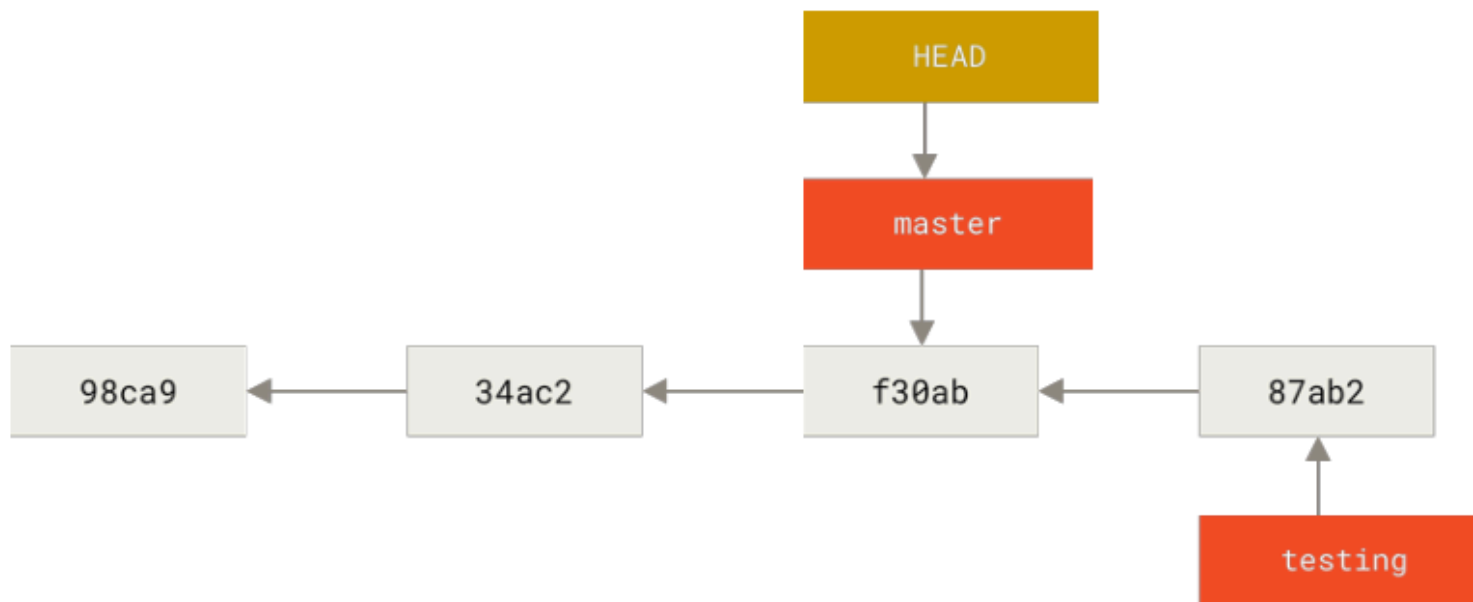
```
git checkout -b <название ветки>
```



На данный момент ветки идентичные. Но теперь, если находясь в новой ветке, мы добавим коммит, то на него перенесется указатель новой ветки, а указатель `master` останется на месте. Ветки разойдутся, но история коммитов останется линейной.

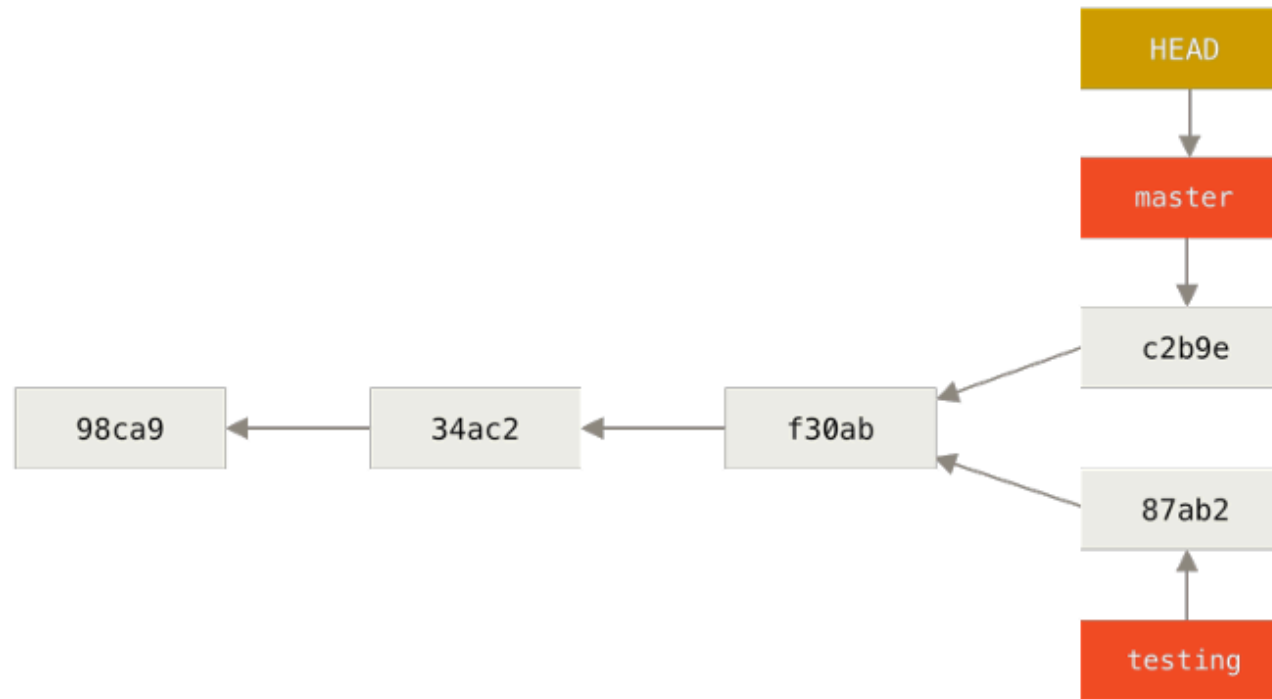


Теперь, если мы обратно вернемся на ветку `master`.



И сделаем в ней изменения и коммит, то произойдет разветвление истории.





Для того чтобы удалить ветку, нужно использовать команду:

```
git branch -d <название ветки>
```

Для визуализации всех веток и точек слияния, используйте команду:

```
git log --oneline --graph --all
```

Если нужно изменить название ветки, то можно воспользоваться командой:

```
git branch --move <старое название> <новое название>
```

## 3.2 Слияние веток

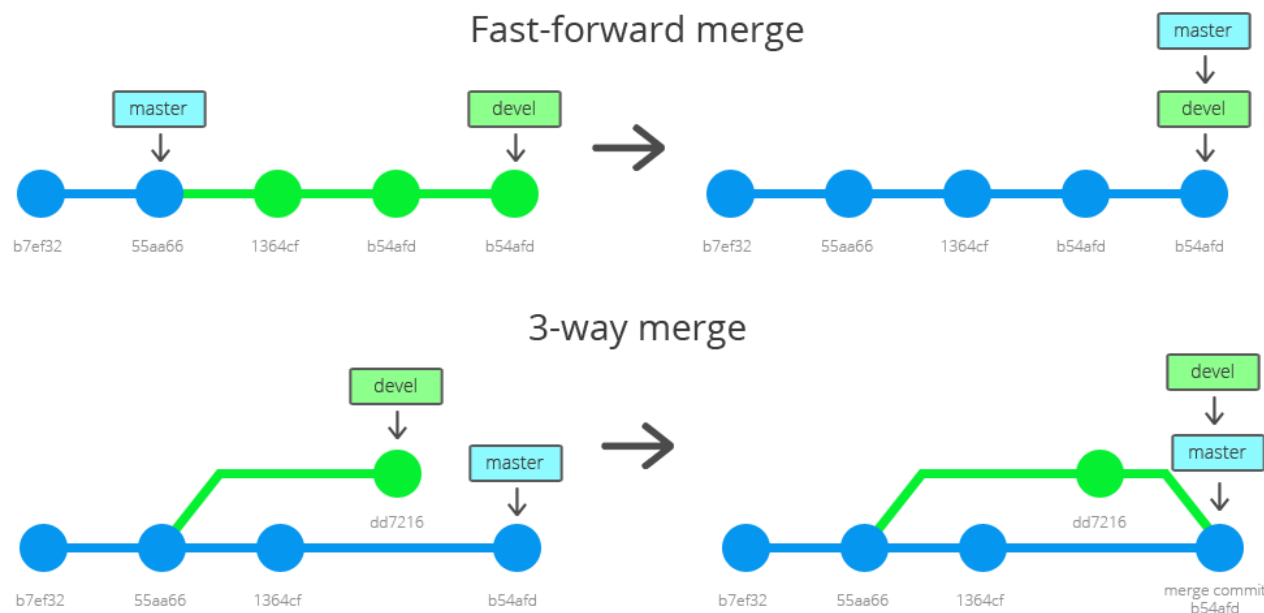
Слияния веток нужны для того, чтобы перенести изменения из одной ветки в другую. Для этого используется команда:

```
git merge <название ветки>
```

Вы должны находиться в той ветке, в которую вам нужно перенести изменения, а в команде указать название ветки, из которой вы хотите эти изменения перенести.

Слияние будет происходить по-разному, в зависимости от того, как расположены ветки относительно друг друга.

- Если указатели находятся на одной линии, то указатель одной ветки просто переносится на место указателя другой, такое слияние называется **fast-forward**.
- Если указатели находятся на разных ветках, то создается новый коммит слияния, который создается на основе последних коммитов из двух веток. Это называется **трехсторонним слиянием**.



При трехстороннем слиянии могут возникнуть **конфликты слияния**. Они возникают тогда, когда один и тот же файл в разных ветках был изменен по-разному и Git не может их автоматически объединить. Git помечает конфликтные участки в файлах специальными маркерами ( <<<<<< , ===== , >>>>>> ), например:

```
<<<<<< master
изменения которые были внесены в ветке master
=====
изменения которые были внесены в ветке hotfix
>>>>>> hotfix
```

Для того, чтобы разрешить конфликт, нужно

- Открыть файл
- Найти маркеры
- Вручную выбрать или объединить нужные части кода
- Добавить в индекс и закоммитить

### 3.3 Практический пример

```
git checkout -b testing
git commit -am "test change"
git log --oneline --graph --all
git checkout master
git merge testing
git log --oneline --graph --all
```

Создали новую ветку и переключились на нее

Добавили в нее коммит

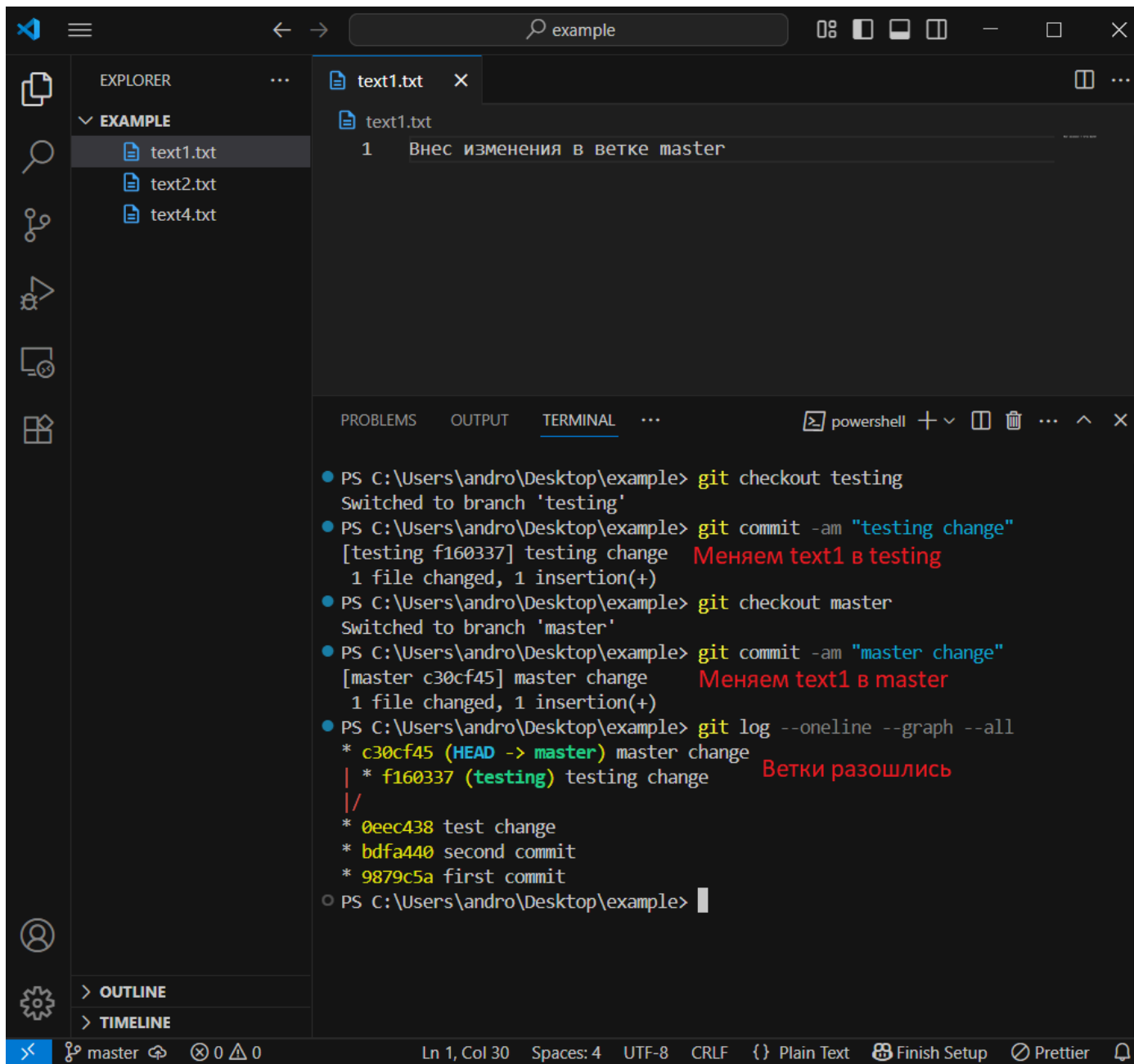
Ветки разошлись

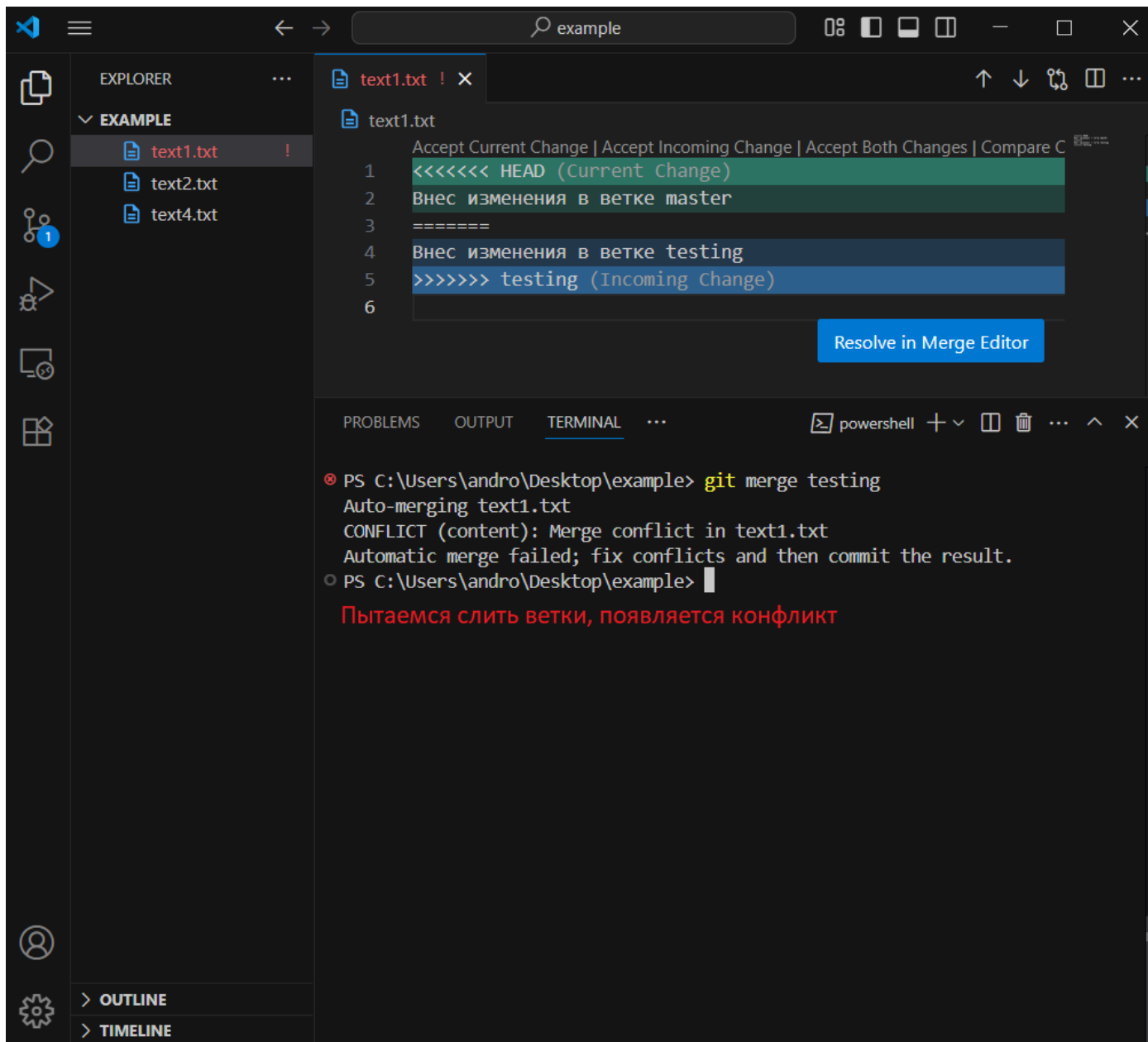
Вернулись на master

Слили с testing

Находятся на одной линии, поэтому просто перенесли указатель

Теперь обе ветки на одном и том же коммите





VS Code interface showing a file explorer, editor, and terminal.

**EXPLORER**

- EXAMPLE
  - text1.txt
  - text2.txt
  - text4.txt

**text1.txt**

```
1 Внес изменения в ветке master
2 Внес изменения в ветке testing
3 Убираем все лишние
```

**TERMINAL**

```
PS C:\Users\andro\Desktop\example> git merge testing
Auto-merging text1.txt
CONFLICT (content): Merge conflict in text1.txt
Automatic merge failed; fix conflicts and then commit the result.
PS C:\Users\andro\Desktop\example> git commit -am "resolve conflict"
[master 52e093b] resolve conflict
PS C:\Users\andro\Desktop\example> git commit -am "resolve conflict"
[master 6450317] resolve conflict
1 file changed, 3 deletions(-)
PS C:\Users\andro\Desktop\example> git log --oneline --graph --all
* 6450317 (HEAD -> master) resolve conflict
* 52e093b resolve conflict
|\
| * f160337 (testing) testing change
| * c30cf45 master change
|/
* 0eec438 test change
* bdfa440 second commit
* 9879c5a first commit
PS C:\Users\andro\Desktop\example>
```

**STATUS BAR**

master 0 0 Ln 3, Col 1 Spaces: 4 UTF-8 CRLF {} Plain Text Finish Setup Prettier

## 4 Работа с удалёнными репозиториями

Мы рассмотрели работу с локальным репозиторием, то есть находящимся у вас на компьютере. Следующим шагом становится взаимодействие с удалёнными хранилищами — серверами, где хранится центральная (или коллективная) копия вашего проекта. Это позволяет нескольким разработчикам совместно работать над кодом.

Самой популярной платформой для размещения репозитория является [GitHub](#). После того, как вы там зарегистрируетесь, у вас будет возможность создать репозиторий. Ему можно дать название и сделать публичным или приватным.

### 4.1 Подключение к удаленному репозиторию

Чтобы связать локальный проект с удалённым, выполните команду:

```
git remote add origin <URL-репозитория>
```

Где `origin` это псевдоним для удаленного репозитория. Для того, чтобы убедиться, что репозитории связаны, выполните команду:

```
git remote -v
```

Если вы наоборот, хотите создать локальный репозиторий на основе удаленного, то введите команду, которая скопирует его на ваш компьютер:

```
git clone <URL-репозитория>
```

GitHub рекомендует называть основную ветку `main`, а не `master`.




**Create a new repository**

A repository contains all project files, including the revision history. Already have a project repository elsewhere? [Import a repository.](#)

*Required fields are marked with an asterisk (\*).*

**Owner \*** **Repository name \***


 OrlovAndrei / Test


✔ Test is available.

Great repository names are short and memorable. Need inspiration? How about **redesigned-bassoon** ?

**Description** (optional)

test repo

☒  **Public**  
Anyone on the internet can see this repository. You choose who can commit.

☐  **Private**  
You choose who can see and commit to this repository.

**Initialize this repository with:**

☐ Add a README file  
This is where you can write a long description for your project. [Learn more about READMEs.](#)

**Add .gitignore**

.gitignore template: **None**

Choose which files not to track from a list of templates. [Learn more about ignoring files.](#)

**Choose a license**

License: **None**

A license tells others what they can and can't do with your code. [Learn more about licenses.](#)

Основными командами являются:

- **git push** — отправляет ваши локальные коммиты в удалённый репозиторий. При первом пуше в ветку `main` (или `master`) обычно используют: `git push -u origin master` дальнейшем `git push` можно вызывать без параметров, также GitHub попросит вас авторизоваться.
- **git fetch** — загружает новые коммиты из удалённого репозитория, но не сливает их с вашей текущей веткой. Это безопасный способ проверить, что изменилось на сервере: `git fetch origin`
- **git pull** — сочетает `fetch` и `merge`: сразу подтягивает коммиты и объединяет их с вашей веткой: `git pull`

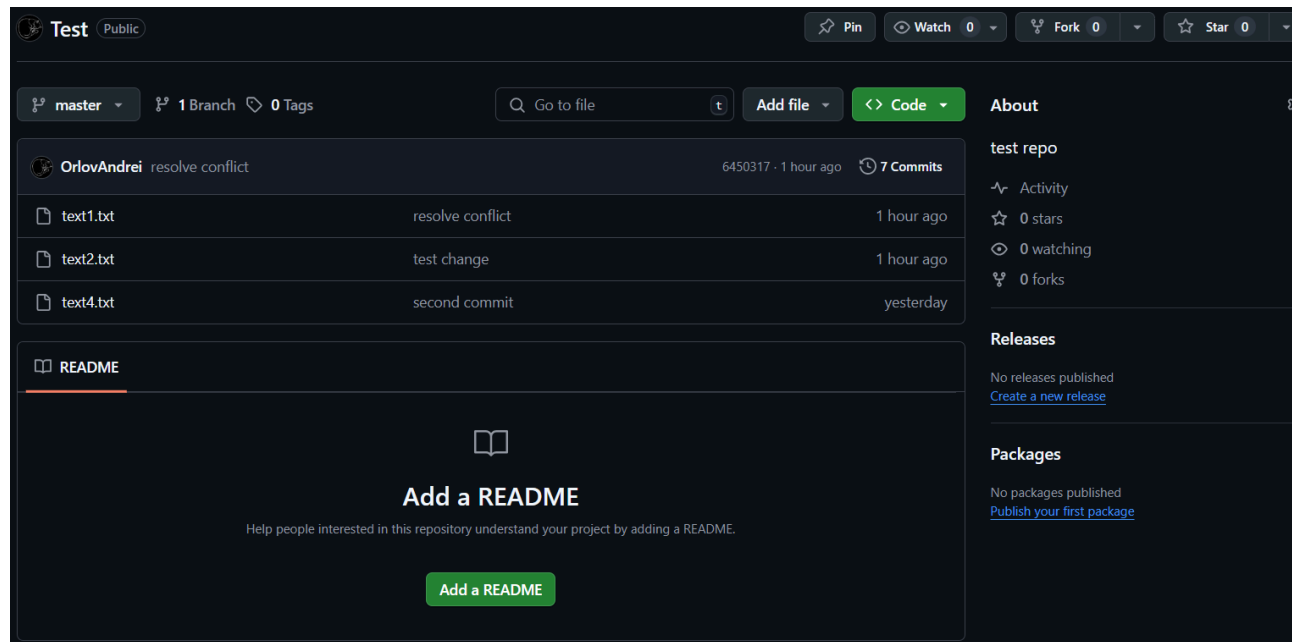
При подключении к удалённому репозиторию, в репозитории появятся **ветки слежения**, которые служат для связи с удалёнными ветками. Имена веток слежения имеют вид `<remote>/<branch>`.

The screenshot shows the Visual Studio Code interface with the following components:

- EXPLORER:** A folder named "EXAMPLE" containing three files: "text1.txt", "text2.txt", and "text4.txt".
- text1.txt:** A text file with three lines of Russian text:

```
1 Внес изменения в ветке master
2 Внес изменения в ветке testing
3
```
- TERMINAL:** A PowerShell terminal window showing the execution of several git commands with Russian annotations:
  - `git remote add origin https://github.com/OrlovAndrei/Test.git` followed by the annotation "Добавили удаленный репозиторий".
  - `git remote -v` followed by the annotation "Он добавился".
  - `git push -u origin master` followed by the annotation "Отправляем все коммиты на удаленный".
  - `git log --oneline --graph --all` followed by a log output with Russian annotations:
    - "Появилась ветка слежения" (Tracking branch appeared) pointing to the HEAD commit.
    - "testing change" (testing change) pointing to the testing branch commit.
    - "master change" (master change) pointing to the master branch commit.
    - "test change" (test change) pointing to the first commit.
    - "second commit" (second commit) pointing to the second commit.
    - "first commit" (first commit) pointing to the first commit.

The status bar at the bottom indicates the current branch is "master", with 0 changes and 0 conflicts. The file encoding is UTF-8, and the line ending is CRLF.



## 4.2 README

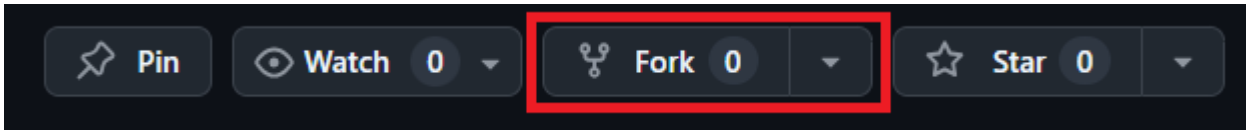
Помимо кода, каждый проект нуждается в понятной и структурированной документации. Самый распространённый способ оформить описание проекта — создать файл `README.md` в корне репозитория. Он автоматически отображается на главной странице проекта в GitHub.

Формат `.md` — это **Markdown**, лёгкий язык разметки, который позволяет удобно оформлять текст. Изучить синтаксис Markdown можно [здесь](#).

## 4.3 Форки и Pull/Merge-запросы

В больших проектах (особенно с открытым исходным кодом) часто используется подход «fork & pull request»:

1. **Fork** — вы создаёте свою копию чужого репозитория на платформе (например, GitHub), чтобы иметь право пушить в неё.
2. **Clone** — клонируете ваш форк локально.
3. **Создаёте ветку** для новой функциональности или исправления:
4. **Делаете коммиты** и пушите ветку в ваш форк:
5. **Открываете Pull Request** (или Merge Request) в исходный репозиторий, где другие участники смогут просмотреть изменения, обсудить их и принять.



Этот подход позволяет защитить главный репозиторий от прямых, возможно неблагонадёжных коммитов и обеспечивает централизованное обсуждение каждого изменения.

---

## Ссылки

1. Скачать Git — [Git - Downloads](#)
  2. Книга по Git, рекомендую прочитать разделы: Введение, Основы Git, Ветвление Git, Распределенный Git, GitHub — [Git](#)
  3. [GitHub](#)
  4. Синтаксис Markdown — [Basic writing and formatting syntax - GitHub Docs](#)
  5. Наглядное приложение для изучения Git — [Learn Git Branching](#)
-

# Практическое задание

## Что нужно сделать:

1. **Создайте новый локальный репозиторий.**
2. **Создайте два текстовых файла, добавьте их в репозиторий и сделайте первый коммит.**
3. **Добавьте файл `.gitignore`, в который включите шаблон для игнорирования определённых файлов.**
4. **Измените один из отслеживаемых файлов и сделайте второй коммит.**
5. **Создайте новую ветку для изменений.**  
Переключитесь в неё, внесите изменения в другой файл и сделайте коммит.
6. **Вернитесь в основную ветку и выполните слияние с новой веткой.**  
Слияние должно быть fast-forward (без конфликтов).
7. **Сделайте ещё одно изменение в основной ветке и зафиксируйте его коммитом.**
8. **Перейдите обратно в новую ветку и измените тот же файл, что и в основной ветке.**  
Выполните коммит, а затем попробуйте слить ветки. Должен произойти конфликт.  
Разрешите его вручную и сделайте коммит.
9. **Просмотрите историю коммитов с отображением ветвлений.**
10. **Зарегистрируйтесь на GitHub (если ещё не зарегистрированы) и создайте удалённый репозиторий.**  
Свяжите его с локальным и отправьте в него все коммиты.
11. **На GitHub создайте файл `README.md` с кратким описанием проекта.**
12. **В локальном репозитории получите изменения с GitHub с помощью команды `pull`, чтобы скачать файл `README.md`.**

## Дополнительно (обязательно):

- **Каждый шаг задания необходимо заскриншотить.**

Скриншоты должны подтверждать выполнение всех действий — содержание файлов, команды в терминале, изменения на GitHub и т.д.

- **Оформите все скриншоты в одном PDF-файле.**

Упорядочьте их по шагам, добавьте подписи (по желанию — краткие комментарии).

- **Добавьте получившийся PDF-отчёт в ваш GitHub-репозиторий.**

Файл можно назвать, например, `report.pdf` и поместить в корень проекта.