GIT

- Как удалить историю GIT
- Как откатить последний коммит
- Полное слияние ветки test в main с удалением test
- Процесс разработки и интеграции изменений в основную ветку
- Переключение на другую задачу, когда текущая еще не закончена
- Squash & Merge в Git или сохранение истории коммитов без промежуточных записей
- Варианты размещения конфигурационных файлов GIT
- Project Access Token (PAT)

Как удалить историю GIT

1. **Создайте новый начальный коммит:**

- Переключитесь на новую ветку без истории:
  `bash git checkout --orphan new-branch`
- Добавьте все файлы в новый коммит:
  `bash git add -A`
- Создайте начальный коммит:
  `bash git commit -m "Начальный коммит"`

---

2. **Удалите старую ветку `main`:**

- Удалите текущую ветку `main`:
  `bash git branch -D main`

---

3. **Переименуйте новую ветку в `main`:**

- Переименуйте новую ветку (`new-branch`) в `main`:
  `bash git branch -m main`

---

4. **Принудительно отправьте изменения в удаленный репозиторий:**

- Принудительно замените историю в удаленном репозитории:
  `bash git push -f origin main`

---

5. **Очистите локальный репозиторий (опционально):**

Если вы хотите очистить локальный репозиторий от старой истории, выполните:

```
git reflog expire --expire=now --all
git gc --prune=now --aggressive
```

---

6. **Обновите локальный репозиторий (если необходимо):**

Если у вас есть другие клоны репозитория, обновите их:

```
git fetch --all
git reset --hard origin/main
```

Важно:

- **Это действие необратимо**: после удаления истории коммитов восстановить их будет невозможно.
- **Уведомите коллег**: если вы работаете в команде, убедитесь, что все участники знают о предстоящих изменениях, так как это может повлиять на их локальные репозитории.

Теперь ваш удаленный репозиторий будет содержать только один коммит с текущим состоянием проекта.

Как откатить последний коммит

Инструкция для отката последнего коммита, в зависимости от ситуации.

---

1. Откат последнего коммита без публикации

Если коммит ещё **не был отправлен** в удалённый репозиторий:

- Удалить коммит, сохранив изменения в индексе (стадии):

```
git reset --soft HEAD~1
```

- Удалить коммит и убрать изменения из индекса, оставив их в рабочей папке:

```
git reset --mixed HEAD~1
```

- Удалить коммит и изменения полностью (⚠ безвозвратно):

```
git reset --hard HEAD~1
```

---

2. Откат опубликованного коммита (без изменения истории)

Если коммит уже **отправлен в удалённый репозиторий**:

- Создать новый коммит, который отменяет изменения предыдущего:

```
git revert HEAD
```

После этого отправить изменения в репозиторий:

```
git push
```

---

3. Жёсткий откат опубликованного коммита (с изменением истории)

Если нужно полностью удалить коммит (и изменения), и ты готов "переписать" историю репозитория:

- Удалить последний коммит и изменения:

  ```
  git reset --hard HEAD~1
  ```

- Принудительно отправить изменения:

  ```
  git push --force
  ```

⚠ **Предупреждение:** Этот способ может привести к проблемам для других разработчиков, так как история репозитория будет изменена.

Вывод

Выбирай способ в зависимости от ситуации: для локальных коммитов подойдут `reset`, для опубликованных — безопасный `revert` или жёсткий `reset --hard`. Если есть сомнения, уточни детали своей задачи!

Полное слияние ветки test в main с удалением test

□□ Задача

У вас есть две ветки: `main` и `test`. Вам нужно:

- **Полностью заменить содержимое `main` на текущее состояние `test`.**
- **Удалить ветку `test` после слияния.**

□ Пошаговое решение

1□ **Переключаемся в `main`**

```
git checkout main
```

2□ **Обновляем `main` перед слиянием** (если работаем с удалённым репозиторием)

```
git pull origin main
```

3□ **Полностью заменяем `main` на `test`**

```
git reset --hard test
```

4⃣ Принудительно отправляем изменения в удалённый репозиторий

```
git push --force origin main
```

⚠️ **Внимание!** Использование `--force` полностью заменит удалённую ветку `main` на состояние `test`. Будьте уверены, что ничего важного не потеряете!

5⃣ Удаляем ветку `test` локально

```
git branch -D test
```

6⃣ Удаляем ветку `test` на удалённом сервере

```
git push origin --delete test
```

🔍 Объяснение команд

Команда	Описание
`git checkout main`	Переключаемся в ветку `main`.
`git pull origin main`	Загружаем актуальные изменения `main`.
`git reset --hard test`	Полностью заменяем содержимое `main` на `test`.
`git push --force origin main`	Принудительно обновляем `main` на удалённом сервере.
`git branch -D test`	Удаляем локальную ветку `test`.
`git push origin --delete test`	Удаляем ветку `test` в удалённом репозитории.

□□ Итог

- Ветка **`main` станет идентичной `test`**.
- Ветка **`test` будет удалена локально и на сервере**.

⚠ Важно!

- **Если `main` используется другими разработчиками**, предупредите их перед выполнением `git push --force origin main`, так как это **перезапишет всю историю**.
- **Более безопасный способ**, если `main` нельзя перезаписывать принудительно:

```
git checkout main
git merge --strategy=ours test
git push origin main
```

Процесс разработки и интеграции изменений в основную ветку

Процесс разработки и интеграции изменений в основную ветку (`main` или `master`) в GIT обычно включает несколько этапов. Описание ниже основано на стандартном Git Flow, но может адаптироваться под конкретные требования.

---

1. Планирование и создание задачи

Прежде чем приступить к разработке, необходимо:

- Определить задачу (например, новую фичу, исправление бага, рефакторинг).
- Завести тикет в системе управления задачами (Jira, YouTrack, GitLab Issues и т.д.).
- Определить, в какой ветке будет вестись работа.

---

2. Создание новой ветки

Разработчик создает новую ветку на основе последней актуальной версии `main`:

```
git checkout main
git pull origin main
git checkout -b feature/new-feature
```

Где `feature/new-feature` — имя ветки, соответствующее задаче.

Если работа ведется с багфиксом, может использоваться ветка `bugfix/bug-id`.

---

3. Разработка и коммиты

Разработчик пишет код, делает изменения и фиксирует их:

```
git add .
git commit -m "Добавлена новая функциональность"
```

Рекомендуется делать осмысленные коммиты и использовать семантические сообщения (`fix:`, `feat:`, `refactor:`, `docs:`, `test:`).

---

4. Локальное тестирование

Перед отправкой изменений важно:

- Запустить юнит-тесты.
- Протестировать код в рабочем окружении (локальном, dev).
- Убедиться, что не нарушена работа других модулей.

---

5. Отправка изменений в удаленный репозиторий

После завершения работы над задачей:

```
git push origin feature/new-feature
```

Создается **Merge Request (Pull Request)** в GitLab/GitHub.

---

6. Код-ревью

- Коллеги проверяют код, дают замечания.
- Разработчик вносит правки, дополняет тестами (если нужно).
- Повторяет `git push` до окончательного одобрения.

7. Интеграция в основную ветку

После успешного ревью:

- Ветка `feature/new-feature` вливается в `develop` (или сразу в `main`, если процесс без `develop`).
- Используется `merge` или `rebase` (в зависимости от стратегии проекта):

```
git checkout main
git pull origin main
git merge feature/new-feature
git push origin main
```

- В некоторых случаях может использоваться **Squash & Merge** для объединения коммитов.

---

8. Автоматизация через CI/CD

После слияния в `main` могут автоматически запускаться:

- Автоматические тесты.
- Деплой на тестовый или production сервер (в зависимости от настроек CI/CD).

---

9. Удаление ненужных веток

После успешного слияния:

```
git branch -d feature/new-feature
git push origin --delete feature/new-feature
```

Это помогает поддерживать чистоту репозитория.

---

10. Деплой и релиз

Если проект использует версионирование:

- Генерируется новая версия (`git tag v1.2.3`).
- Запускается CI/CD пайплайн, который деплоит код.

Итог

Этот процесс помогает обеспечить стабильность разработки, контроль качества и чистоту репозитория. В реальных проектах могут быть добавлены дополнительные этапы, например, работа с `hotfix`-ветками, релизными ветками (`release/`), более строгий контроль тестирования и деплоя.

Переключение на другую задачу, когда текущая еще не закончена

Если текущая задача еще не завершена, но требуется переключиться на другую, важно сохранить прогресс, чтобы потом без проблем вернуться к работе. В GIT это можно сделать несколькими способами.

---

1. Использование `git stash` (Временное сохранение изменений)

Если изменения не готовы для коммита, можно их временно отложить:

```
git stash
```

Теперь рабочая директория станет чистой, и можно переключаться на другую ветку:

```
git checkout feature/another-task
```

Когда будет возможность вернуться к предыдущей задаче, можно восстановить изменения:

```
git checkout feature/new-feature
git stash pop  # Вернет изменения и удалит их из stash
```

Если требуется сохранить несколько отложенных изменений, можно посмотреть список stash:

```
git stash list
```

А затем применить нужный:

```
git stash apply stash@{0}
```

---

2. Коммит незавершенных изменений в черновую ветку

Если изменения значительные, но не готовы для основного репозитория, можно создать временную ветку:

```
git checkout -b wip/feature-new-feature
git add .
git commit -m "WIP: временное сохранение"
git push origin wip/feature-new-feature
```

После этого можно переключиться на другую задачу:

```
git checkout feature/another-task
```

Когда будет возможность вернуться, просто переключиться обратно:

```
git checkout feature/new-feature
git merge wip/feature-new-feature
```

---

3. Коммит с пометкой `WIP` (Work In Progress)

Если возможно закоммитить частичные изменения, можно сделать коммит с пометкой `WIP`:

```
git add .
git commit -m "WIP: начало работы над фичей"
```

Затем переключиться на другую ветку:

```
git checkout feature/another-task
```

Когда будет возможность вернуться, переключиться обратно и продолжить работу.

Какой способ выбрать?

- **`git stash`** — если изменения локальные и их не нужно пушить.
- **Черновая ветка (`wip/feature-new-feature`)** — если требуется сохранить изменения на сервере.
- **Коммит `WIP`** — если можно сделать осмысленный коммит.

Обычно в командах `git stash` подходит для быстрого переключения, а `WIP`-коммиты или черновые ветки — для более долгосрочных перерывов.

Squash & Merge в Git или сохранение истории коммитов без промежуточных записей

Squash & Merge — это способ слияния изменений, при котором несколько коммитов объединяются (squash — "сжимать") в один перед вливанием в основную ветку. Этот метод позволяет сохранить чистую историю коммитов без лишних промежуточных записей.

Как работает Squash & Merge?

Допустим, в ветке `feature/new-feature` есть несколько коммитов:

```
commit a1b2c3d - Fix bug in query handler
commit d4e5f6g - Add logging
commit h7i8j9k - Implement new feature
commit l0m1n2o - Fix typo in previous commit
```

При обычном `merge` они просто добавятся в основную ветку (`main`).

Но если использовать **Squash & Merge**, все эти коммиты объединяются в один:

```
commit p9q8r7s - Implement new feature and fixes
```

Таким образом, история остается чистой.

Когда использовать Squash & Merge?

Когда нужно объединить "мусорные" коммиты (например, тестовые исправления, исправления опечаток).
Когда код писался в несколько итераций, но результат — одно логическое изменение.
При работе в feature-ветках, чтобы в `main` попадали только готовые коммиты.
Не подходит для веток, где важна детальная история коммитов (например, в `develop` или `release`).

Как выполнить Squash & Merge?

1. Вручную через `git rebase -i`

Переключаемся на нужную ветку:

```
git checkout feature/new-feature
```

Запускаем интерактивный rebase:

```
git rebase -i main
```

Git покажет список коммитов, например:

```
pick a1b2c3d Fix bug in query handler
pick d4e5f6g Add logging
pick h7i8j9k Implement new feature
pick l0m1n2o Fix typo in previous commit
```

Заменяем `pick` на `squash (s)` у всех коммитов, кроме первого:

```
pick a1b2c3d Fix bug in query handler
squash d4e5f6g Add logging
squash h7i8j9k Implement new feature
squash l0m1n2o Fix typo in previous commit
```

После сохранения Git предложит изменить сообщение коммита, оставляем итоговое.

Затем пушим изменения (если уже был push в удаленный репозиторий, потребуется `--force`):

```
git push origin feature/new-feature --force
```

---

2. Через GitHub/GitLab UI

При слиянии Pull Request (Merge Request) можно выбрать опцию **Squash & Merge**:

- В **GitHub**: `Merge pull request -> Squash & Merge`
- В **GitLab**: `Merge Request -> Squash commits`

Вывод

Squash & Merge удобен для поддержания чистоты истории коммитов. Важно применять его осознанно: если детальная история не нужна, squash поможет убрать ненужные коммиты и оставить только финальные изменения.

Варианты размещения конфигурационных файлов GIT

□□ Где Git ищет конфиги (Windows

Git использует три уровня конфигурации, **в жёстко заданном порядке**:

Уровень	Команда	Файл по умолчанию
System	`git config --system`	`C:\Program Files\Git\etc\gitconfig`
Global	`git config --global`	`%USERPROFILE%\.gitconfig`
Local	`git config --local` (по умолчанию)	`<путь_к_проекту>\.git\config`

□□ Как Git находи`C:\Program Files\Git\etc\gitconfig`

Git встроенно знает путь к своему системному конфигу:

- Этот путь **жёстко прошит в бинарный Git for Windows**
- Он **не переопределяется**
- Используется **для установки глобальных параметров по умолчанию**, например:

  ```
  [credential] helper = store
  ```

  или

  ```
  [core] autocrlf = true
  ```

□□ Можно ли его изменить?

□Нет — стандартный путь `etc/gitconfig` не переопределяется в переменных окружения.

Однако ты можешь:

2. Открыть его вручную:

```
notepad "C:\Program Files\Git\etc\gitconfig"
```

3. Удалить или закомментировать ненужные строки:

```
[credential] helper = store ; ← закомментировать
```

> ⚠ Требуются права администратора.

□ Альтернатива: переопределить в глобальной/локальной конфигурации

Если ты не хочешь трогать системный файл, просто **переопредели**:

```
git config --global credential.helper ""
```

или

```
git config --local credential.helper "store --file=.git/.git-credentials"
```

Git будет использовать **наиболее приоритетный из найденных** (local > global > system).

Project Access Token (PAT)

□□ Кто может видеть токен?

- **Сразу после создания** токен можно скопировать **один раз**.
- **Все пользователи с правами Maintainer/Owner** проекта **могут видеть, что токен существует**, но **не видят его значение**.
- **Ты мог видеть его под другим пользователем**, только если:
  - ты смотришь в список активных токенов,
  - но **не можешь использовать его**, если не скопировал ранее его значение.

□□ Для кого работает Project Acce Token?

Токен **работает от имени "виртуального пользователя" проекта**:

- В GitLab в списке пользователей он отображается как:
  `Project access token (имя_токена)`
  и имеет отдельный ID

□**Он не зависит от логина, под которым ты залогинен в GitLab UI или Git**.

□□ Как узнать, к какому пользователю "привязан" Project Access Token?

> □ Он **не привязан к реальному пользователю**. Он:

- создаётся **в рамках проекта**
- действует **в рамках прав, которые ты задал при создании** (например, `api`, `read_repository`, `write_repository`)
- фигурирует как "сервисный пользователь" проекта

□□ Кто может использовать этот токен?

Любой, у кого есть:

- **Сам токен (строка)**,
- **И URL проекта**, к которому он относится.

□□ Поэтому **сам токен нужно хранить безопасно**, как обычный пароль.

□ Как использовать:

```
# Формат для .git-credentials: https://<token_name>:<access_token>@gitlab.example.com
```

Пример:

```
https://ci-bot:glpat-abc123@gitlab.example.com/group/project.git
```

□□ Вывод:

Вопрос	Ответ
Кто видит токен?	Только в момент создания — и владельцы с правами Maintainer в проекте
Кому принадлежит?	Проекту, не пользователю
Кто может использовать?	Любой, у кого есть значение токена
Где отображается?	В настройках проекта → Access Tokens
Можно ли узнать, кто создал?	Нет, GitLab не отображает автора токена