ГЛЕБ ЗАЙЦЕВ

Git

на практике

ВЕТКИ · КОНФЛИКТЫ · ВОССТАНОВЛЕНИЕ

12 **лабораторных работ**

в локальном репозитории

Команды, объяснения
и проверяемый результат

Глеб Зайцев

Git на практике. 12 лабораторных работ в локальном репозитории

«Автор»
2026

Зайцев Г.

Git на практике. 12 лабораторных работ в локальном репозитории / Г. Зайцев — «Автор», 2026

Выучить названия команд Git недостаточно: важно понимать, какие данные изменятся после их выполнения. В этом практикуме — 12 самостоятельных лабораторных работ на маленьких текстовых проектах. Вы сравните рабочие файлы с индексом, потренируете ветвление и слияние, разберёте конфликт, отмените ошибочный коммит и смоделируете работу двух участников без подключения к серверу. Для каждой работы даны команды, проверяемый результат, объяснение и задания с ответами. Нужны установленный Git и терминал; опыт программирования и аккаунт GitHub не требуются. Это учебная песочница, а не инструкция по аварийному восстановлению рабочих репозиториев.

Содержание

Глеб Зайцев
Git на практике. 12 лабораторных работ в локальном репозитории

Что вы получите

У новичка Git часто превращается в набор заклинаний: add, commit, push. Пока всё работает, этого хватает. Но однажды в коммит попадает лишний файл, слияние останавливается или изменения будто исчезают при переключении ветки. В этот момент полезнее не ещё одна запомненная команда, а привычка сначала установить состояние проекта.

Эта книга устроена как тренажёр. У каждой работы отдельная папка, короткая история и наблюдаемый результат. Вы не разворачиваете приложение, не подключаете чужой сервер и не отправляете учебные коммиты в интернет. Вместо сотен строк кода используются небольшие текстовые файлы. Так легче отличить действие Git от поведения программы.

Практикум адресован человеку, который умеет открыть терминал и хочет уверенно выполнять базовые действия с репозиторием. Знание языка программирования не требуется. Здесь нет настройки CI, администрирования Git-сервера, сложного rebase и восстановления повреждённой базы объектов. Для этих задач нужны другие материалы и отдельная практика.

Работы независимы по данным, но объяснения идут от простого к сложному. Лучше выполнить их по порядку. После команды не торопитесь к следующей: прочитайте контрольный вопрос, предскажите ответ и лишь затем проверьте себя. Если результат отличается, не подгоняйте его случайными командами. Сначала посмотрите текущую папку, ветку и status.

Три состояния, которые нельзя смешивать

Рабочая папка — файлы, которые вы сейчас видите и редактируете. Индекс — подготовленный снимок следующего коммита. Коммит — сохранённый снимок с метаданными и связью с предыдущей историей. Команда add копирует текущее содержимое выбранного пути в индекс. Она не подписывается на будущие изменения этого файла.

HEAD обычно указывает через имя текущей ветки на последний коммит. Название main не обладает специальной магией: это выбранное нами имя ветки. Ветка — изменяемый указатель на коммит, а не отдельная физическая папка с копиями всех файлов. При переключении Git обновляет рабочие файлы в соответствии с выбранной историей, если этому не мешают незавершённые изменения.

git diff без дополнительных параметров сравнивает рабочие файлы с индексом. git diff --cached сравнивает индекс с HEAD. Поэтому пустой обычный diff ещё не означает, что следующий коммит пуст. И наоборот: пустой cached diff не означает, что на диске нет изменений. Первая работа специально разводит эти две ситуации.

Новые неотслеживаемые файлы обычно не попадают в diff. Их видно через git status. Короткий status содержит два столбца состояния: первый относится к индексу, второй — к рабочей папке. Например, «M » и « M» означают разные вещи. Пробел в этих обозначениях важен; полный status объясняет их словами.

Среда и границы безопасности

Установите Git из официального источника git-scm.com/install и проверьте командой git --version. В практикуме используется синтаксис Git 2.28 и новее. Реальные проверки этой редакции проведены на macOS с Apple Git 2.39.5. Команды оболочки рассчитаны на Bash или совместимую оболочку; в Windows для них нужен Git Bash, а не командная строка cmd или PowerShell. Эти варианты Windows и Linux отдельно не тестировались.

В командах используются mkdir, printf, cat, cp, test и mktemp. Символ > в учебных примерах записывает содержимое файла заново, а >> добавляет в конец. Выполняйте эти строки только внутри созданных ниже учебных папок. В настоящем проекте такая запись может уничтожить нужное содержимое ещё до того, как Git получит управление.

Никогда не начинайте работу из папки своего приложения, домашней папки или чужого репозитория. Переменная GIT_LAB будет содержать путь к новой временной песочнице. Все имена и адрес student@example.invalid в примерах вымышлены. Настройки автора записываются только в учебный репозиторий с --local; глобальная конфигурация не меняется.

Каждую работу выполняйте один раз в новой папке. Если mkdir сообщает, что папка уже существует, остановитесь: не продолжайте команды поверх прежнего опыта. Можно открыть новый терминал и создать новую песочницу целиком. Учебные файлы во временном каталоге могут быть очищены системой; важные собственные данные туда не переносите.

При неожиданной ошибке остановитесь на этой строке. Не копируйте в терминал сразу всю главу. Исключения явно обозначены только в работах 6 и 11: там мы намеренно получим отказ Git, изучим его и продолжим по объяснённой процедуре. Книга не требует reset --hard, clean -fd, принудительной отправки, удаления репозиториев или ввода паролей.

Как читать команды и проверять результат

Каждая отдельная строка примера — одна команда. Оформление шрифта зависит от читалки. Если при копировании перед командой появились пробелы, в том числе неразрывные, удалите только этот начальный отступ; пробелы внутри команды сохраняйте. Перенос по ширине экрана может быть только визуальным: не добавляйте в команду лишний Enter. Копируйте прямые кавычки и два обычных дефиса в параметрах. Символ приглашения терминала вроде $ перед строкой не нужен. Вывод Git зависит от языка и версии, поэтому мы проверяем прежде всего содержимое файлов и структуру истории, а не точное оформление служебных сообщений.

Короткие идентификаторы коммитов у вас будут другими: на них влияют содержимое, время и метаданные. Не подставляйте случайный хеш из чужого скриншота. В книге используются HEAD, имена веток и специально созданные метки. Пустой вывод status --short в контрольной точке означает чистые индекс и рабочую папку относительно отслеживаемых файлов; проигнорированные файлы при этом могут существовать.

Текст подготовлен с помощью ИИ; команды и заявленные контрольные состояния проверены автоматическим запуском на синтетических данных. Это не гарантия безопасности для любого существующего проекта. Ограничения среды и действия с потерей незакоммиченных изменений отмечены отдельно. Если учебный пример работает, следующий разумный шаг — небольшой тест на копии своего проекта, а не применение ко всей рабочей истории.

Подготовка песочницы

В новом терминале выполните этот блок один раз. Напечатанный путь сохраните. После перезапуска терминала переменная не сохранится автоматически: проще создать новую песочницу. Следующие работы создают внутри неё свои каталоги.

```
GIT_LAB=$(mktemp -d "${TMPDIR:-/tmp}/git-lab.XXXXXX")
export GIT_LAB
mkdir "$GIT_LAB/no-hooks"
printf '%s\n' "$GIT_LAB"
```

Перед каждой работой проверьте, что GIT_LAB не пустая и указывает на напечатанную папку. Начальные настройки повторяются намеренно: каждая работа самостоятельна. Любая неожиданная ошибка — повод остановиться, а не продолжить запись файлов в неизвестной папке.

Работа 1. Коммит сохраняет индекс, а не всё с диска

Ситуация: вы подготовили файл к коммиту, затем ещё раз его отредактировали. Какую версию сохранит commit? Проверим это на плане небольшой инструкции. Важно не просто увидеть две версии, а понять, в какой момент снимок попал в индекс.

Начальный блок создаёт отдельный репозиторий. Параметры подписи отключаются только здесь, чтобы учебный коммит не запрашивал настроенный в вашей системе ключ. Пустой локальный каталог hooks не даёт запускаться пользовательским хукам в этом упражнении. В рабочих проектах эти настройки без согласования не меняйте.

```
cd "$GIT_LAB"
mkdir lab-01
cd lab-01
git init -b main
git config --local user.name "Учебный автор"
git config --local user.email "student@example.invalid"
git config --local commit.gpgsign false
git config --local tag.gpgsign false
git config --local core.autocrlf false
git config --local core.hooksPath "$GIT_LAB/no-hooks"
printf 'topic: Git\nstatus: draft\n' > plan.txt
git add plan.txt
printf 'topic: Git\nstatus: reviewed\n' > plan.txt
git diff --cached
git diff
git commit -m "Save draft plan"
git show HEAD:plan.txt
cat plan.txt
git status --short
```

После первого блока в HEAD находится status: draft, а в рабочем файле — status: reviewed. Короткий status показывает изменение plan.txt во втором столбце. Теперь осознанно подготовим новую версию и сохраним второй коммит.

```
cd "$GIT_LAB/lab-01"
git add plan.txt
git diff --cached
git commit -m "Review plan"
git status --short
git log --oneline
```

Контроль результата

В HEAD и на диске теперь status: reviewed. В истории два коммита; status --short не выводит строк.

Первый add видел версию draft. Более поздняя запись reviewed изменила только рабочий файл. Commit не проигнорировал вашу правку по ошибке: он выполнил контракт и сохранил подготовленный снимок. Для включения последней версии нужен ещё один add.

В реальной работе перед коммитом полезно просмотреть cached diff, даже если вы только что смотрели обычный diff в редакторе. Это разные сравнения. Такая привычка помогает не отправить устаревшую или лишнюю часть изменения.

Если после второго коммита status остался непустым, сравните имя файла и убедитесь, что редактор не сохранил его повторно уже после add. Не решайте проблему широким git add .: сначала установите, какие именно пути изменены.

Проверьте понимание

Если после первого add изменить файл ещё три раза, сколько этих версий попадёт в первый коммит?

Ответ и объяснение

Ни одна из трёх новых: первый коммит сохранит снимок, существовавший на момент add. Git не ведёт журнал каждого сохранения редактора.

Работа 2. Убрать лишний файл из будущего коммита

Ситуация: две заметки изменены и подготовлены вместе, но закончена только первая. Требуется сохранить первую, не теряя работу над второй. Здесь особенно важно различать отмену подготовки и уничтожение содержимого рабочего файла.

Создадим исходный коммит, затем изменим оба файла. Перед исправлением ошибки посмотрим список подготовленных путей. В команде restore явно укажем --staged: целью будет индекс, а не рабочая копия.

```
cd "$GIT_LAB"
mkdir lab-02
cd lab-02
git init -b main
git config --local user.name "Учебный автор"
git config --local user.email "student@example.invalid"
git config --local commit.gpgsign false
git config --local tag.gpgsign false
git config --local core.autocrlf false
git config --local core.hooksPath "$GIT_LAB/no-hooks"
printf 'A: draft\n' > a.txt
printf 'B: draft\n' > b.txt
git add a.txt b.txt
git commit -m "Initial entries"
printf 'A: ready\n' > a.txt
printf 'B: in progress\n' > b.txt
git add a.txt b.txt
git diff --cached --name-only
git restore --staged -- b.txt
```

Конец ознакомительного фрагмента.