Неделя 7. Лекция 4. Git и GitHub: как команда работает с кодом

Часть 1 из 2.

После лекции: вы понимаете, как Git хранит историю, что такое ветки, слияние и конфликты, и умеете вести проект так, как это делают в командах: задача → ветка → Pull Request → ревью → merge → релиз, и `main` всегда рабочий. **Инструменты:** Git, GitHub, Visual Studio 2026 (окна Git Changes и Git Repository), PowerShell 7.

1. Зачем контроль версий

Сейчас сервис задач лежит у вас в папке `C:\dev\demo-api`. Как вернуться к вчерашней версии, если сегодня всё сломалось? Как двоим одновременно менять код и не затереть друг друга? Как через полгода понять, кто и зачем поменял строку с проверкой имени проекта?

Git хранит всю историю изменений, позволяет вести параллельные линии работы и объединять их. **GitHub** — сервис, где лежит общая копия репозитория и где команда обсуждает изменения: задачи, Pull Request, ревью, релизы.

Для DevOps Git — ещё и **точка запуска автоматизации и источник правды**. Каждый push запускает тесты и деплой (тема 5), а на вопрос «что сейчас работает в проде» отвечают коммитом и тегом, а не словами «вроде последняя версия». Когда в теме 8 сервис сломается после релиза, расследование начнётся именно с истории в Git.

2. Как Git хранит историю

- **Коммит** — снимок всех файлов проекта в момент фиксации, плюс автор, дата, сообщение и ссылка на родительский коммит. У коммита уникальный хэш: `a1b2c3d`.
- **Ветка** — подвижный указатель на коммит. Новый коммит в ветке передвигает указатель вперёд.
- **`HEAD`** — где вы сейчас: какая ветка или коммит выгружены в рабочую папку.

```
gitGraph
  commit id: "init"
  commit id: "Dockerfile"
  branch feat/12-stats
  commit id: "stats: repository"
  commit id: "stats: tests"
  checkout main
  merge feat/12-stats id: "PR #13"
```

История — цепочка снимков. Любой снимок можно достать, сравнить с другим или отменить.

Что внутри. Папка `.git` — это база объектов трёх видов. **Blob** хранит содержимое одного файла, без имени. **Tree** — содержимое папки: список имён, и у каждого ссылка на blob или вложенный tree. **Commit** ссылается на корневой tree (снимок всего проекта), на родителя и содержит автора и сообщение:

```
PS> git cat-file -p HEAD
tree 7bb75930cc63fee4229c00f080f04e2b7c37c5c1
parent fed7c896de2c6d922db5e144d7db6b09b8844450
author Anna <anna@example.com> 1789792689 +0500
committer Anna <anna@example.com> 1789792689 +0500

feat: переименование проекта (#9)
```

Хэш — отпечаток содержимого. Имя любого объекта — хэш от его содержимого (по умолчанию SHA-1, 40 шестнадцатеричных символов, обычно показывают первые 7). Одинаковые файлы дают одинаковый blob, поэтому неизменённый `Program.cs` в сотне коммитов хранится один раз. А коммит, который «весит» как полный снимок, на деле добавляет в базу только изменённые файлы.

Почему история неизменяема. Хэш коммита зависит от tree и от хэша родителя, а тот — от своего родителя. Поменяйте одну букву в старом коммите — изменится его хэш, а за ним хэши всех последующих. Поэтому Git не «правит» историю: `amend`, `rebase`, `reset` создают **новые** коммиты рядом со старыми. Подменить прошлое незаметно нельзя: у коллег и на GitHub хэши останутся прежними, и расхождение сразу видно.

Ветка и `HEAD` — просто указатели. Ветка — файл с одним хэшем внутри (`.git/refs/heads/main`). Поэтому

создать ветку стоит одну запись в файл, а не копию проекта. `HEAD` обычно указывает на ветку (`ref: refs/heads/main`), и коммит двигает вперёд именно её. Если переключиться на конкретный коммит или тег, `HEAD` указывает прямо на хэш — это состояние *detached HEAD*: смотреть можно, а новые коммиты не попадут ни в одну ветку.

> **Демонстрация для преподавателя.** В репозитории сервиса:

```
"Сервис задач" | git hash-object --stdin   # у всех в аудитории один и тот же хэш
git cat-file -p HEAD                        # коммит: tree, parent, автор, сообщение
git cat-file -p "HEAD^{tree}"               # корневая папка: blob и tree с именами
Get-Content .git\HEAD                       # ref: refs/heads/main
Get-Content .git\refs\heads\main            # ветка — это один хэш
```

3. Три области: рабочая папка, индекс, репозиторий

```
рабочая папка  --git add-->  индекс (staging)  --git commit-->  репозиторий (история)
```

Команда	Что делает
`git status`	Что изменено, что в индексе, что не отслеживается
`git diff` / `git diff --staged`	Изменения в рабочей папке / в индексе построчно
`git add <файл>` / `git add -A`	Положить изменения в индекс — «вот это пойдёт в коммит»
`git commit -m "..."`	Зафиксировать индекс как новый коммит
`git restore <файл>`	Отменить незакоммиченные правки файла
`git log --oneline --graph`	История

Индекс — черновик следующего коммита. Он позволяет собрать коммит из части изменений: поправили репозиторий и заодно опечатку в README — закоммитили отдельно. Коммит, который делает одну вещь, легко ревьюить, откатить и найти потом через `bisect` (п. 11).

`.gitignore` — что Git не отслеживает. У нас это стандартный шаблон `dotnet new gitignore` плюс `.env`: `bin/`, `obj/`, `.vs/`, `*.user`, `*.pfx`. Логика простая: в репозитории лежит то, из чего проект **собирается**, а не то, что получается при сборке или живёт только на вашей машине.

- **Бинарники (`bin/`, `obj/`)** пересобираются из кода, различаются от машины к машине и дают конфликты, которые невозможно разрешить построчно. Каждая их версия навсегда остаётся в истории и раздувает клон.
- **Настройки IDE (`.vs/`, `*.user`)** — личные, у соседа они другие.
- **Секреты (`.env`, сертификаты `*.pfx`)** — пароль, попавший в коммит, остаётся в истории навсегда (п. 10). Вместо `.env` в репозиторий кладут `.env.example` с пустыми значениями.

`.gitignore` действует только на **неотслеживаемые** файлы. Если `.env` уже закоммичен, добавление в `.gitignore` ничего не изменит — сначала `git rm --cached .env`, а пароль всё равно придётся сменить.

4. Удалённый репозиторий

У каждого разработчика полная копия репозитория со всей историей. Общая копия — на GitHub, она называется **`origin`**. Git распределённый: коммитить, смотреть историю и создавать ветки можно без сети, а синхронизация — отдельный явный шаг.

Команда	Что делает
`git clone <url>`	Скачать репозиторий целиком
`git fetch`	Узнать, что нового в `origin`, не меняя свою работу
`git pull`	`fetch` + влить изменения в текущую ветку
`git push`	Отправить свои коммиты в `origin`

После `fetch` у вас появляется ветка `origin/main` — снимок того, где был `main` на GitHub в момент последней синхронизации. Ваш локальный `main` и `origin/main` — разные указатели, и `git status` пишет, насколько они разошлись: *Your branch is behind 'origin/main' by 2 commits*. Поэтому в лабе каждую новую ветку начинают с `git switch main; git pull`: иначе вы строите работу на устаревшем коде и получаете конфликты, которых могло не быть.

5. Ветки, слияние и конфликты

Ветка — изолированная линия работы: `git switch -c feat/3-rename-project`. Пока вы работаете в ветке, `main` не меняется.

Слияние (merge) объединяет ветки. Есть два случая:

- **Fast-forward.** Если `main` не ушёл вперёд с момента создания ветки, объединять нечего: Git просто передвигает указатель `main` на последний коммит ветки. Новых коммитов не появляется.
- **Коммит слияния.** Если в `main` за это время влили чужой PR, у истории две линии. Git создаёт коммит с **двумя родителями**, который соединяет их. На графе это «ромб», как `PR #13` на схеме в п. 2.

Squash merge — режим GitHub, который мы используем: все коммиты ветки сжимаются в один новый коммит в `main` с заголовком PR и номером, например `feat: переименование проекта (#9)`. В `main` — одна запись на задачу, а промежуточные «поправил по ревью» остаются в PR. Цена: коммиты ветки в `main` не попадают, поэтому после squash ветку удаляют и новую работу начинают от свежего `main`.

Rebase переносит ваши коммиты на вершину свежего `main`, как будто вы начали работу позже. Технически он создаёт **новые** коммиты с новыми хэшами (п. 2), а старые бросает. История получается линейной, но **общие ветки ребейзить нельзя**: у коллег останутся старые коммиты, и при следующем `pull` Git увидит две версии одной работы. Правило: ребейзить можно то, что ещё не отправляли или что точно никто не взял к себе. После rebase уже отправленной ветки нужен `git push --force-with-lease`, а на `main` force-push запрещён ruleset'ом (п. 8).

Как Git находит конфликт. Слияние трёхстороннее: Git берёт **общего предка** двух веток (merge base) и сравнивает с ним обе стороны. Строку изменили только в одной ветке — берётся эта версия, без вопросов. Изменили разные места одного файла — Git совмещает автоматически. Конфликт возникает, только когда обе ветки по-разному изменили **одно и то же место** относительно предка. Git не решает за вас и помечает место:

```
<<<<<<< HEAD
# Сервис задач
=======
# Task Service API
>>>>>>> origin/main
```

Между `<<<<<<<` и `=======` — ваша сторона (`HEAD`, текущая ветка), между `=======` и `>>>>>>>` — то, что вливаете. Вы оставляете правильный вариант (или объединяете оба), удаляете маркеры, делаете `git add` и `git commit`. Visual Studio показывает для этого редактор слияния с тремя панелями.

Полезно видеть и версию предка: с настройкой `merge.conflictStyle zdiff3` Git добавляет её между `|||||||` и `=======`. Тогда видно, кто что поменял: здесь исходный заголовок был `# DemoApi`, и изменили его оба.

```
<<<<<<< HEAD
# Сервис задач
||||||| fed7c89
# DemoApi
=======
# Task Service API
>>>>>>> origin/main
```

Конфликт — не ошибка Git, а вопрос к человеку. Разрешать его нужно, понимая чужое изменение: в коде «оставить своё» часто означает молча удалить чужую функцию. После разрешения соберите проект и прогоните тесты — Git проверяет строки, а не смысл.

> **Демонстрация для преподавателя.** Конфликт и отмена слияния:
>
> ```
> git config --global merge.conflictStyle zdiff3   # показывать версию предка
> git switch -c demo/title        # правим 1-ю строку README, git commit -am "docs: title A"
> git switch main                 # ту же строку иначе, git commit -am "docs: title B"
> git merge demo/title            # CONFLICT (content): Merge conflict in README.md
> git merge --abort               # вернуть всё как было до слияния
> ```