توی این دوره قراره با یکی از مهم‌ترین ابزارهای دنیای برنامه‌نویسی آشنا بشی؛ ابزاری که تقریباً همه‌ی توسعه‌دهنده‌ها، از فریلنسرها گرفته تا شرکت‌های بزرگ مثل گوگل و مایکروسافت ازش استفاده می‌کنن

Git چیه و چرا انقدر مهمه؟

تصور کن روی یه پروژه نرم‌افزاری کار می‌کنی. در طول زمان، بارها فایل‌هاتو تغییر می‌دی — یه قابلیت جدید اضافه می‌کنی، باگ‌ها رو اصلاح می‌کنی، یا بخش‌هایی از کدت رو بازنویسی می‌کنی. حالا فرض کن بخوای بدونی دقیقاً چه تغییراتی انجام دادی، اون تغییرات چه زمانی بوده، توسط چه کسی انجام شده یا حتی بخوای به نسخه‌ای قبلی از کدت برگردی.

اینجاست که مفهوم کنترل نسخه اهمیت پیدا می‌کنه.

کنترل نسخه به سیستمی گفته می‌شه که تمام تغییرات ایجاد شده در فایل‌ها رو به‌صورت ساخت‌یافته و قابل پیگیری ذخیره می‌کنه. این سیستم بهت اجازه می‌ده تاریخچه‌ی توسعه‌ی پروژه رو مرحله به مرحله دنبال کنی، بین نسخه‌ها جابه‌جا بشی، یا حتی خطاها رو به‌راحتی اصلاح کنی.

اما Git فقط یه سیستم کنترل نسخه‌ی ساده نیست — یه سیستم کنترل نسخه‌ی توزیع‌شده‌ست.

برخلاف سیستم‌های قدیمی که فقط یه نسخه اصلی روی یه سرور مرکزی نگه‌داری می‌شد، در Git هر نفر که روی پروژه کار می‌کنه، یه نسخه‌ی کامل و مستقل از کل تاریخچه‌ی پروژه رو روی سیستم خودش داره. این ساختار مزایای زیادی داره:

* حتی اگه سرور اصلی خراب بشه یا در دسترس نباشه، پروژه از بین نمی‌ره، چون هر هم‌تیمی یه نسخه‌ی کامل از اون رو داره.
* می‌تونی بدون نیاز به اینترنت یا اتصال به سرور، آفلاین کار کنی و تغییراتت رو بعداً همگام‌سازی کنی.
* همکاری تیمی خیلی ساده‌تر و ایمن‌تر انجام می‌شه، چون هیچ وابستگی مطلقی به یک نقطه مرکزی وجود نداره.

این فلسفه‌ی توزیع‌شده بودن، دلیل اصلی قدرتمند بودن Git در پروژه‌های حرفه‌ای و تیمی‌ست.

Git چطوری به‌وجود اومد؟

سال ۲۰۰۵، لینوس توروالدز، خالق سیستم‌عامل لینوکس، بعد از اختلافاتی که با نرم‌افزار BitKeeper (یه سیستم کنترل نسخه متمرکز) پیدا کرد، تصمیم گرفت خودش یه سیستم کنترل نسخه جدید بسازه—یه چیزی که هم سریع باشه، هم امن، هم قابل اعتماد، و هم مناسب پروژه‌های متن‌باز.

نتیجه؟  
Git رو فقط در ۵ روز نوشت!

از اون موقع تا امروز، Git به یکی از مهم‌ترین ابزارهای توسعه نرم‌افزار در دنیا تبدیل شده و تقریباً همه جا استفاده می‌شه.

Git دقیقاً چیکار می‌کنه؟

Git این امکان رو بهت می‌ده که:

* کدت رو مرحله‌به‌مرحله ثبت کنی (commit)
* بفهمی هر تغییر، چه زمانی، توسط چه کسی انجام شده
* شاخه‌سازی کنی (branch) و به‌طور هم‌زمان روی چند مسیر توسعه کار کنی
* شاخه‌هات رو ترکیب کنی (merge)
* تغییرات اشتباه رو برگردونی (revert)
* و در نهایت با ابزارهایی مثل GitHub، GitLab یا Bitbucket، پروژه‌ت رو با دیگران به اشتراک بذاری و به‌صورت تیمی توسعه بدی
* . نصب Git
* اگه Git رو هنوز نصب نکردی، می‌تونی از سایت رسمی [git-scm.com](https://git-scm.com) دانلودش کنی. نسخه متناسب با سیستم‌عاملت رو انتخاب کن (ویندوز، مک یا لینوکس) و طبق مراحل پیش‌فرض نصبش کن.
* بعد از نصب، با این دستور می‌تونیم مطمئن بشیم Git درست نصب شده:

git –version

Porcelain و Plumbing در Git

توی Git، دستورها به دو دسته تقسیم می‌شن:

* Porcelain (سطح بالا): دستورهایی که معمولاً توسعه‌دهنده‌ها باهاشون کار می‌کنن. مثل:
  + git status
  + git add
  + git commit
  + git branch
  + git push  
    اینا همون ابزارهایی هستن که روزمره استفاده می‌کنی. ما توی این دوره تمرکزمون روی همین‌هاست.
* Plumbing (سطح پایین): دستورهای تخصصی‌تر و فنی‌تر که برای توسعه ابزارها یا تحلیل‌های دقیق‌تر استفاده می‌شن. مثل:
  + git hash-object
  + git cat-file
  + git commit-tree

البته خیلی از توسعه‌دهنده‌ها، حتی حرفه‌ای‌ها، هیچ‌وقت مستقیم با plumbing کار نمی‌کنن. اما شناخت ساختار درونی Git باعث می‌شه درک بهتری از رفتارش داشته باشی.

تنظیمات اولیه Git (Config)

حالا وقتشه Git رو برای خودمون شخصی‌سازی کنیم.

Git اطلاعات نویسنده هر تغییر (commit) رو ذخیره می‌کنه. پس باید یه بار نام و ایمیل خودمون رو ثبت کنیم:

git config --global user.name "نام شما"

git config --global user.email "ایمیل شما"

این اطلاعات در فایلی به نام .gitconfig ذخیره می‌شه که معمولاً در پوشه Home شما قرار داره.

اگه خواستی بررسیش کنی، می‌تونی بنویسی:

git config --list

اگه اطلاعات name و email رو نشون داد، یعنی درست ثبت شدن.

تنظیم نام شاخه پیش‌فرض

به‌صورت پیش‌فرض، بعضی از نسخه‌های جدید Git شاخه اصلی رو main در نظر می‌گیرن. اما ما برای یکپارچگی توی این دوره، شاخه پیش‌فرض رو روی master تنظیم می‌کنیم:

git config --global init.defaultBranch master

بعداً در مورد تفاوت‌های main و master صحبت می‌کنیم، اما الان فقط می‌خوایم مطمئن شیم همه توی دوره روی یه پایه‌ی مشترک هستن.

چرا این تنظیمات مهمه؟

شاید الان این بخش‌ها یه مقدار خشک یا گنگ به‌نظر برسن، مخصوصاً اگه اولین باره داری Git کار می‌کنی. ولی این پیکربندی اولیه خیلی مهمه. چون:

جلوی اشتباهات بعدی رو می‌گیره

commit‌هات رو به اسم خودت ثبت می‌کنه

و پروژه‌هات رو استانداردتر و مرتب‌تر نگه می‌داره

نگران نباش اگه الان همه‌چیز برات واضح نیست. کم‌کم که با Git کار کنی، این چیزا برات جا می‌افته.

فعلاً همین که Git نصبه، اطلاعاتت ثبت شده و شاخه‌ی پیش‌فرض تنظیم شده، یعنی آماده‌ای که وارد دنیای واقعی Git بشی

خب، حالا که Git رو نصب و پیکربندی کردیم، وقتشه بریم سراغ ساخت اولین پروژه.

۱. ساخت مخزن (Repository)

اولین قدم توی هر پروژه با Git اینه که یه مخزن (Repository) بسازیم.

مخزن Git یا به‌اختصار repo، در واقع نماینده‌ی یک پروژه‌ی مشخصه.

معمولاً برای هر پروژه‌ی جداگانه، یک مخزن جداگانه هم داریم.

(مگر اینکه بحث Fork یا پروژه‌های منشعب مطرح باشه.)

از نظر ساختاری، repo فقط یه پوشه معمولیه که فایل‌ها و فولدرهای پروژه داخلش قرار می‌گیرن. تنها تفاوتش با بقیه پوشه‌ها اینه که یه فولدر مخفی به اسم .git توش وجود داره.

و اینجاست که نکته مهم رو باید بدونی:

تمام اطلاعات Git توی همین پوشه .git ذخیره می‌شه!

شاخه‌ها، تاریخچه، commitها، تگ‌ها، همه‌چی... دقیقاً همینجاست.

اگه برات سواله Git چطوری می‌دونه چه فایل‌هایی هست یا کدوم شاخه فعاله، جوابش توی این فولدره.

۲. ساخت پروژه نمونه – Webflix

توی این دوره، ما با یه پروژه فرضی به اسم Webflix کار می‌کنیم؛ یه پلتفرم خیالی برای پخش ویدیو به‌صورت سلف‌ هاست.

برای شروع، بریم یه فولدر بسازیم:

mkdir webflix

cd webflix

الان توی یه فولدر خالی هستیم. هیچ فایلی اینجا نیست.

حالا Git رو توی این پوشه راه‌اندازی می‌کنیم:

git init

این دستور باعث می‌شه Git یه مخزن جدید بسازه و پوشه .git رو به این دایرکتوری اضافه کنه. اگه با ls -la چک کنی، این پوشه رو می‌بینی.

۳. آشنایی با وضعیت فایل‌ها

توی Git، فایل‌ها همیشه توی یکی از این سه حالت قرار دارن:

Untracked – فایلی که Git هنوز هیچ اطلاعی ازش نداره.

Staged – فایلی که آماده‌ی commit شده، یعنی توی staging area قرار گرفته.

Committed – فایلی که تغییراتش ذخیره شده و توی تاریخچه Git ثبت شده.

برای اینکه بفهمیم یه فایل توی چه وضعیه، از دستور زیر استفاده می‌کنیم:

git status

۴. ایجاد فایل محتوا

حالا بیایم یه فایل ساده داخل پروژه ایجاد کنیم.

میتونی این فایل رو با هر ویرایشگری که دوست داری بسازی — مثلاً Notepad یا VS Code — فقط کافیه یه فایل جدید به اسم contents.md یا contents.txt درست کنی و داخلش یه لیست ساده بنویسی:

# Contents

- Home

- Library

- Search

- Settings

بعد از اینکه فایل رو ذخیره کردی، دوباره دستور زیر رو اجرا کن:

git status

می‌بینی که فایل به صورت Untracked نمایش داده می‌شه. یعنی Git هنوز اون رو نمی‌شناسه یا دنبال نمی‌کنه.

اگه فایل untracked رو پاک کنی، Git نمی‌تونه هیچ نسخه‌ای ازش برات بازیابی کنه. خیلی‌ها این اشتباه رو انجام می‌دن و بدون اینکه فایل رو به Git معرفی کنن (یعنی add بزنن)، حذفش می‌کنن و همه‌چی از دست می‌ره.

۵. اضافه کردن فایل به staging

برای اینکه Git شروع به پیگیری فایل بکنه، باید اونو به staging اضافه کنیم:

git add contents.md

الان بازم git status رو اجرا کن. باید ببینی که فایل از حالت untracked به staged تغییر کرده.

نکته مهم:

git add . فقط فایل‌ها و پوشه‌های داخل همون دایرکتوری فعلی و زیرمجموعه‌هاش رو اضافه می‌کنه. ممکنه یه‌سری فایل تو دایرکتوری‌های دیگه باقی بمونن. برای اضافه کردن همه‌چیز بهتره از git add -A استفاده کنی. فعلاً همینو بدون؛ بعداً کامل توضیح می‌دم.

۶. انجام اولین commit

commit کردن یعنی ثبت یک نسخه از تغییراتت در تاریخچه‌ی پروژه.

مثل اینه که یه عکس (snapshot) از وضعیت فعلی فایل‌هات بگیری و بگی:  
«این تغییرات رو در این لحظه ثبت کن، با این توضیح.»

بعد از مرحله staging، وقتشه کامیت کنیم :

git commit -m "add contents"

فلگ -m (مخفف message) بهت اجازه می‌ده مستقیم داخل ترمینال پیام commit رو بنویسی،

معمولاً باید برای هر commit یک پیام توضیحی بنویسی که بگه این تغییرات مربوط به چی بودن.

اینجا Git یه نسخه کامل از وضعیت پروژه‌تو در این لحظه ذخیره می‌کنه.

یه نکته خیلی مهم:

برخلاف تصور بعضیا، Git فقط تفاوت‌ها (diff) رو ذخیره نمی‌کنه، بلکه برای هر commit، یک تصویر کامل از فایل‌های پروژه رو ثبت می‌کنه.

یعنی هر commit یک نسخه مستقل از پروژه‌ست.

برای بررسی تاریخچه commitها می‌تونی از این دستور استفاده کنی:

git log --oneline

اینجا می‌تونی ببینی چه کسی چه تغییری، کی انجام داده. این همون چیزیه که Git رو تبدیل به یه سیستم کنترل نسخه قدرتمند می‌کنه.

الان شما اولین commit‌ت رو ثبت کردی و وارد دنیای واقعی Git شدی.

🔍 تا اینجا چه کردیم؟

✅ یه مخزن جدید ساختیم

✅ با وضعیت فایل‌ها آشنا شدیم

✅ یه فایل جدید اضافه کردیم

✅ فایل رو به staging بردیم

✅ commit زدیم

✅ تاریخچه رو با git log دیدیم

حالا وقتشه وارد یکی از مهم‌ترین و جذاب‌ترین بخش‌های Git بشیم:

🧠 Git پشت‌صحنه commit رو چطور ذخیره می‌کنه؟

وقتی یک commit جدید ثبت می‌کنی، Git به اون یک شناسه اختصاص می‌ده که بهش می‌گیم commit hash — یک رشته‌ی طولانی که با الگوریتم SHA-1 تولید می‌شه (مثلاً چیزی شبیه به 7805172...).

اما نکته جالب اینجاست:

این هش همیشه تکرارپذیره؛ یعنی اگر موارد زیر دقیقاً یکسان باشن، هش هم کاملاً یکسان درمیاد:

محتوای دقیق فایل‌ها

تاریخ و زمان commit

نام و ایمیل نویسنده

پیام commit

و commit parent (اشاره‌گر به commit قبلی)

❓ چرا دو نفر با محتوای مشابه، هش متفاوت دارن؟

حتی اگر دو برنامه‌نویس دقیقاً همون فایل‌ها رو commit کنن، باز هم ممکنه هش commitها متفاوت باشه. دلیلش ساده‌ست:

هش فقط به محتوای فایل‌ها بستگی نداره؛ بلکه به اطلاعات جانبی مثل نام و ایمیل نویسنده، زمان commit، پیام commit و parent هم وابسته‌ست.

📁 در پوشه .git چه خبره؟

همه‌چیز در Git داخل دایرکتوری .git ذخیره می‌شه — دقیق‌تر بگیم داخل .git/objects/.

در این مسیر:

پوشه‌هایی با نام‌های دوحرفی ساخته می‌شن (برای جلوگیری از شلوغی فایل‌سیستم)

و داخل اون‌ها فایل‌هایی قرار داره که هرکدوم نماینده‌ی یک شیء (object) از Git هستن: مثل commit، tree یا blob

🤯 می‌شه مستقیم این فایل‌ها رو بخونیم؟

نه دقیقاً. اگر اون فایل‌ها رو با cat یا یک ادیتور باز کنی، فقط با یه مشت داده‌ی ناخوانای فشرده‌شده مواجه می‌شی. چون Git همه داده‌ها رو با فشرده‌سازی zlib ذخیره می‌کنه.

اما نگران نباش!

Git یه دستور جادویی داره:

git cat-file -p <hash>

این دستور، محتوای هر شیء Git رو—خواه commit، tree یا blob—به‌شکل قابل خوندن نشون می‌ده.

📂 Tree و Blob چی هستن؟

در Git:

Blob نمایانگر محتوای یک فایله (بدون نام یا مسیر)

Tree نمایانگر یک دایرکتوریه؛ شامل مجموعه‌ای از فایل‌ها، پوشه‌ها و permissionها

🧩 آیا Git فقط تغییرات (diff) رو ذخیره می‌کنه؟

نه! برخلاف تصور خیلی‌ها، Git برای هر commit یک snapshot کامل از کل پروژه در اون لحظه ذخیره می‌کنه.

اما این کارو خیلی هوشمندانه انجام می‌ده:

فایل‌های بدون تغییر رو مجدداً ذخیره نمی‌کنه

فقط به blobهای قبلی ارجاع می‌ده

داده‌های تکراری رو حذف می‌کنه (deduplication)

و همه‌چیز رو فشرده نگه می‌داره

برای همینه که Git هم سریع عمل می‌کنه، هم مصرف حافظه‌ی کمی داره.

🧪 تمرین عملی: بررسی commit با cat-file

بیاید خودمون همه این ساختارها رو ببینیم:

گرفتن هش آخرین commit:

git log --oneline

نمایش commit:

git cat-file -p <commit-hash>

→ این دستور یک tree به شما نشون می‌ده.

نمایش tree:

git cat-file -p <tree-hash>

→ حالا لیستی از فایل‌ها و blobها رو می‌بینید.

نمایش محتوای یک فایل:

git cat-file -p <blob-hash>

→ محتوای دقیق فایل رو می‌بینید!

🎬 بخش جدید – شاخه‌ها در Git (Branching)

حالا نوبت یکی از پرکاربردترین قابلیت‌های Git رسیده: شاخه‌ها (branches).

حالا نوبت یکی از پرکاربردترین قابلیت‌های Git رسیده: شاخه‌ها (branches).

🌱 Branch در Git یعنی چی؟

Branch بهت این امکان رو می‌ده که بتونی به‌صورت موازی روی ویژگی‌ها یا ایده‌های جدید کار کنی، بدون اینکه روی نسخه اصلی پروژه تأثیر بذاری.

مثال: فرض کن داری روی یه پروژه وب کار می‌کنی و می‌خوای رنگ‌بندی سایت رو عوض کنی. به‌جای اینکه مستقیم روی شاخه اصلی (مثلاً master یا main) تغییر بدی، می‌تونی یه شاخه جدا بسازی به اسم color-scheme و فقط اونجا تغییرات رو انجام بدی.

اگه نتیجه رو دوست داشتی، تغییرات رو با شاخه اصلی ادغام می‌کنی (merge).  
اگه دوست نداشتی؟ شاخه رو پاک می‌کنی و هیچ تغییری روی پروژه اصلی اعمال نمی‌شه.

👨‍💻 بررسی شاخه فعلی

برای اینکه بدونی الان روی کدوم شاخه هستی:

git branch

علامت \* نشون می‌ده که در حال حاضر روی کدوم شاخه هستی.

✏️

🌿 ساخت شاخه جدید

دو روش برای ساختن شاخه جدید داریم:

فقط ساخت شاخه (بدون سوییچ):

git branch new-branch

ساخت شاخه و سوییچ هم‌زمان (روش پیشنهادی):

git switch -c new-branch

یا به روش کلاسیک‌تر:

git checkout -b new-branch

🔁 سوییچ بین شاخه‌ها

برای جابه‌جایی بین شاخه‌ها از دستور switch استفاده می‌کنیم:

git switch main

در گذشته از git checkout برای این کار استفاده می‌شد، ولی چون checkout دستور شلوغ و گیج‌کننده‌ایه، بهتره برای سوییچ از switch استفاده کنیم.

تغییر نام یک شاخه

برای تغییر نام یک شاخه مثلاً از master به main:

git branch -m master main

💡 اضافه‌کردن commit در یک شاخه‌ی جدید

فرض کن توی شاخه‌ی main سه commit داری: A → B → C

حالا یه شاخه‌ی جدید می‌سازی به اسم ad-classics و روی اون commit جدیدی به اسم D اضافه می‌کنی:

git switch -c ad-classics

touch classics.csv

# و محتوا رو بنویس

git add classics.csv

git commit -m "D: add classics"

الان ad-classics شامل commitهای A, B, C و D هست، ولی main فقط A, B, C رو داره.

🧭 بررسی تاریخچه با log

برای دیدن تاریخچه ساده و خلاصه:

git log --oneline

برای دیدن اینکه هر شاخه کجاست و کجا diverge کرده:

git log --oneline --decorate --graph

پرچم‌ها:

--oneline: نمایش خلاصه‌ی هر commit

--decorate: نشون دادن نام شاخه‌ها کنار commitها

--graph: نمایش ساختار شاخه‌ها به‌صورت گرافیکی

شاخه‌سازی (Branching) توی Git

شاخه‌سازی توی Git یکی از عملیات‌های اصلیه که دائم باهاش سر و کار داریم. ولی خب یه جایی باید کاری که روی یه شاخه انجام دادی رو دوباره بیاری توی شاخه‌ی اصلی. اینجاست که مبحث ادغام (merge) وارد می‌شه.

تقریباً همه‌ی برنامه‌نویسا توی دنیای واقعی، این مراحل رو طی می‌کنن:

اول مخزنتو با آخرین تغییرات به‌روز می‌کنی. معمولاً با git fetch یا git pull (که جلوتر بیشتر توضیحش می‌دیم).

یه شاخه جدید از شاخه‌ی اصلی (مثلاً main) می‌سازی و اسمش رو می‌ذاری چیزی که نشون بده روی چی کار می‌کنی (مثلاً feature/login).

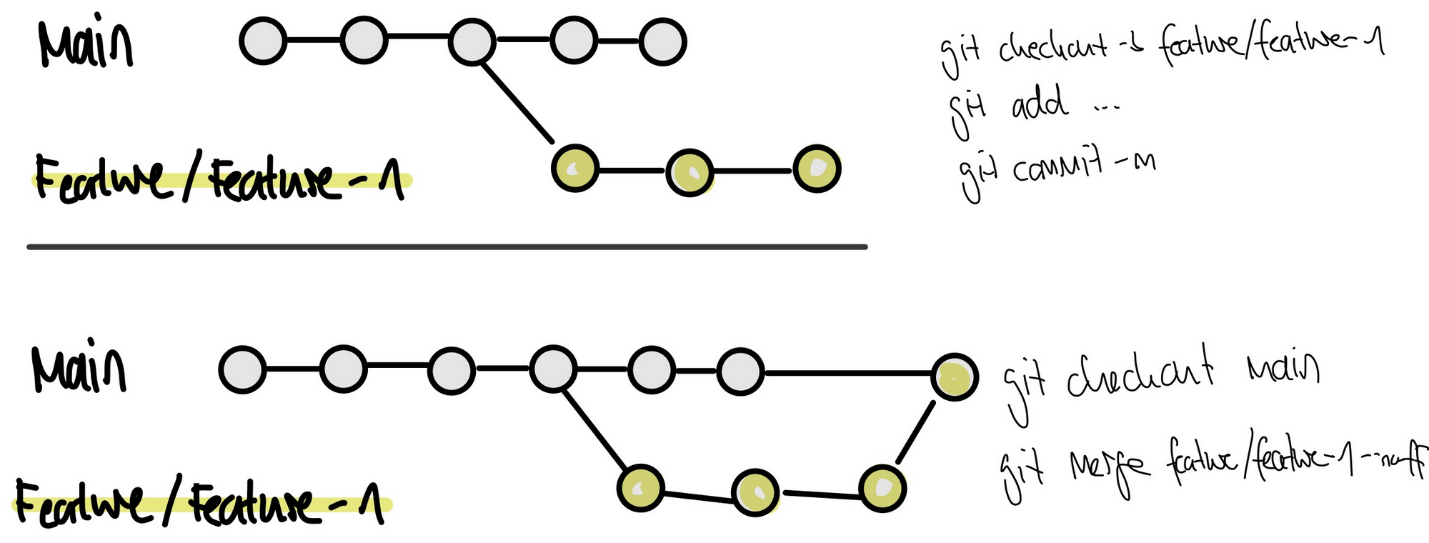
یه باگ رو درست می‌کنی، یه قابلیت جدید اضافه می‌کنی یا داکیومنت‌ها رو آپدیت می‌کنی و توی این شاخه چند تا commit می‌زنی.

آخرش، تغییراتت رو با merge می‌ریزی توی شاخه‌ی اصلی.

چرا اصلاً چندتا شاخه داشته باشیم؟

چون باهاش می‌تونی تغییراتو با خیال راحت انجام بدی بدون اینکه روی کار بقیه یا نسخه اصلی پروژه تأثیر بذاری. ولی وقتی کارت تموم شد و همه‌چی اوکی بود، وقتشه که این تغییرات برن توی نسخه نهایی.

حالا یه نگاهی بنداز به این نمودار قشنگ! 👇



تمرین: ساخت یک merge commit

خب اول برگردیم به شاخه‌ی main:

git switch main

(نکته: دیگه از git checkout استفاده نکن! از switch و restore استفاده کن، چون خوندنشون راحت‌تره و دقیق‌ترن.)

بعدش فایلی به اسم contents.md داریم که خالیه. بیایم توش چند خط اضافه کنیم. بعد:

git add contents.md

git commit -m "e: updated contents with a new line"

حالا توی تاریخچه‌ی ما، چند تا commit داریم: A, B, C, D، و الان هم E.

برای دیدنش از این دستور استفاده کن:

git log --oneline --graph --decorate --parents

وای ببین چقدر قشنگه! حالا معلوم می‌شه کی کجا انشعاب گرفته و برگشته. مثلاً شاخه‌ی add-classics از یه نقطه جدا شده و الان می‌خوایم برش گردونیم به main.

چی می‌شه وقتی merge می‌کنی؟

وقتی دوتا شاخه مسیر جداگانه‌ای داشتن، Git باید یه commit خاص بسازه که اون دو مسیر رو یکی کنه. این همون چیزیه که بهش می‌گیم merge commit.

ویژگی merge commit:

دو تا والد (parent) داره، چون داره دو مسیر متفاوت رو به هم می‌رسونه.

برای ساختنش، Git اول دنبال یه merge base می‌گرده؛ یعنی نزدیک‌ترین اجداد مشترک دوتا شاخه. مثلاً اگه شاخه‌ها از commit A منشعب شدن، همون می‌شه merge base.

بعد تغییرات شاخه‌ی main و شاخه‌ی جانبی رو می‌ریزه توی یه commit جدید (که مثلاً اسمش می‌شه F).

اجرای merge واقعی

git merge add-classics

الان یه merge commit ساخته می‌شه. با دستور git log --graph --oneline --decorate --parents می‌تونی ببینی:

merge commit دو parent داره

شاخه‌ها چطور جدا و دوباره یکی شدن

خیلی از کاربران Git اینو نمی‌دونن که merge commit چجوری کار می‌کنه، اما اینجا کاملاً شفافه!

⏩ Fast-forward Merge یعنی چی؟

وقتی دو شاخه داری—مثلاً main و update-titles—و می‌خوای اون‌ها رو با هم ترکیب (merge) کنی، Git اول بررسی می‌کنه که آیا می‌تونه این کار رو بدون نیاز به ساخت merge commit انجام بده یا نه.

اگه شاخه‌ی جانبی (مثل update-titles) تمام commitهای شاخه‌ی اصلی (main) رو دقیقاً شامل بشه و هیچ شاخه‌ای از هم جدا نشده باشن، Git فقط اشاره‌گر (pointer) شاخه‌ی اصلی رو جلو می‌بره تا برسه به جایی که شاخه‌ی جانبی هست.

به این حالت می‌گیم:

✅ Fast-forward merge (

یعنی انگار هیچ شاخه‌ای جدا نشده بوده، و همه commitها به‌صورت خطی ثبت شدن.

🧪 مثال عملی

فرض کن روی main بودی و بعد یه شاخه جدید به اسم update-titles ساختی و فقط روی همون شاخه کار کردی:

git switch main

git merge update-titles

اگه در این مدت، کسی روی main تغییری نداده باشه، Git می‌فهمه که هیچ تعارض یا دوشاخه‌گی‌ای وجود نداره.

پس:

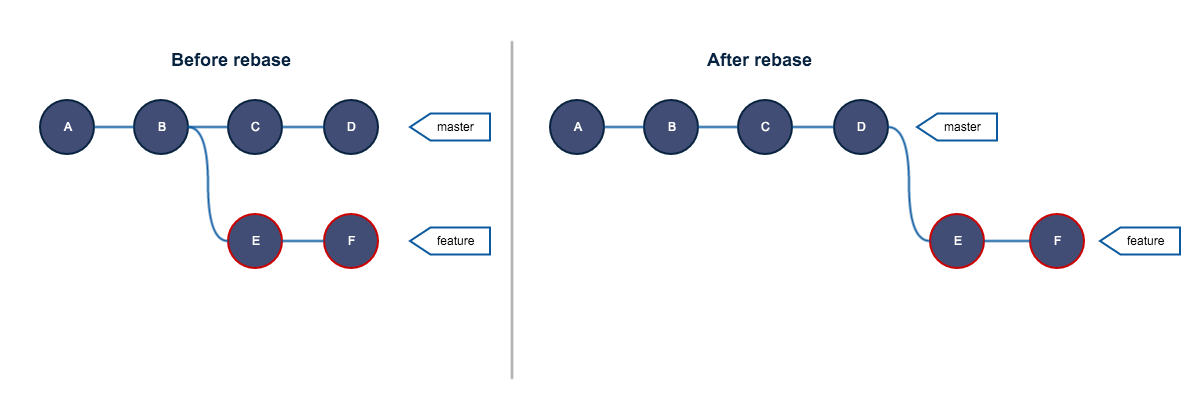
نه نیازی به ایجاد merge commit هست

نه Git مجبور می‌شه تاریخچه‌ی شاخه‌ها رو با هم ترکیب کنه

فقط کافیه pointer شاخه‌ی main رو ببره به‌جلو، جایی که update-titles قرار داره.

🔄 Rebase در Git — دقیق، ساده و با مثال

✅ Rebase یعنی چی؟



در Git، وقتی روی یک شاخه (branch) کار می‌کنی و در همین مدت شاخه‌ی اصلی (مثلاً main) هم به‌روزرسانی شده، برای هماهنگ کردن تغییرات خودت با آخرین وضعیت پروژه می‌تونی از rebase استفاده کنی.

به زبان ساده:

Rebase یعنی "برداشتن" تغییرات شاخه‌ی خودت، "چسبوندن" اون‌ها به آخرین نقطه‌ی شاخه‌ی اصلی، طوری که انگار این تغییرات از ابتدا روی آخرین نسخه‌ی پروژه اعمال شده‌ن.

🎯 هدف اصلی Rebase

هدف اصلی Rebase اینه که:

تاریخچه‌ی commit‌هات یک‌دست و خطی (linear) باشه

کار با git log و بررسی تغییرات آسون‌تر شه

بدون تولید merge commit اضافی، تغییراتتو به شاخه‌ی اصلی وصل کنی

در پروژه‌های متن‌باز یا تیمی، pull request‌هات تمیزتر و قابل فهم‌تر باشه

📌 چه‌طور کار می‌کنه؟ با یه مثال:

فرض کن این تاریخچه رو داریم:

Main branch:

A — B — C

Feature branch (شخصی):

A — D — E

الان main تا C پیش رفته ولی شاخه‌ی تو (feature) هنوز روی A بوده. حالا اگه بزنی:

git rebase main

Git کاری می‌کنه که تاریخچه‌ی تو بشه:

A — B — C — D′ — E′

یعنی D و E رو «دوباره پخش می‌کنه» (replay) بعد از C.

البته چون پایه‌ی commit‌ها تغییر کرده، SHA اون‌ها هم عوض می‌شه (به همین خاطر D و E تبدیل به D′ و E′ شدن).

🧠 چرا Rebase محبوبه؟

چند دلیل مهم:

تاریخچه‌ی تمیز: همه چیز توی یه خطه، بدون شاخه‌های زیاد و merge commit‌های گیج‌کننده

کد قابل فهم‌تر: وقتی افراد زیادی با پروژه کار می‌کنن، بررسی تغییرات راحت‌تر می‌شه

برای review عالیه: وقتی می‌خوای Pull Request بدی، همه‌ی تغییراتت مرتب پشت‌سر هم دیده می‌شن

امکان squash راحت: می‌تونی چند commit رو تبدیل به یک commit تمیز کنی (با interactive rebase)

⚠️ ولی چرا می‌تونه خطرناک باشه؟

هر ابزار قدرتمندی می‌تونه اگر درست استفاده نشه، مشکل‌ساز بشه. Rebase هم همینطوره.

دلایل حساس بودن rebase:

تاریخچه رو تغییر می‌ده

وقتی rebase می‌کنی، Git در واقع commit‌های جدیدی می‌سازه با SHA جدید.

اگه اینو روی یه شاخه‌ی عمومی (مثل main) انجام بدی، بقیه‌ی اعضای تیم به مشکل می‌خورن چون Git فکر می‌کنه تاریخچه عوض شده و نمی‌دونه چطور merge کنه.

نباید روی شاخه‌های public استفاده بشه

هرگز روی شاخه‌هایی که به اشتراک گذاشتی و دیگران ازش استفاده می‌کنن، rebase نکن.

چون باعث conflict، ارور و حتی از دست رفتن داده‌ می‌شه (مخصوصاً اگه force push بزنی).

استفاده نادرست = دردسر بزرگ

مثلا اگه وسط کار rebase رو اشتباه متوقف کنی یا conflict‌هارو اشتباه حل کنی، ممکنه کدت خراب شه یا تاریخچه به‌هم بریزه.

🧪 دستور کار rebase:

فرض کنیم روی شاخه‌ی feature هستی:

git switch feature

git rebase main

Git:

آخرین وضعیت main رو می‌گیره

commit‌های شاخه‌ی feature رو یکی‌یکی برمی‌داره

اون‌ها رو مثل queue پشت main قرار می‌ده

و شاخه‌ی feature رو به اون نقطه منتقل می‌کنه

🟢 Fast-forward merge یعنی چی؟

فرض کن تو شاخه‌ی main هستی. ازش یه شاخه‌ی جدید به نام feature ساختی، و فقط تو همون feature کار کردی. main هیچ تغییری نکرده.

حالا می‌خوای تغییرات feature رو به main برگردونی. چون main عقب‌تره و هیچ کاری نکرده، Git به راحتی فقط اشاره‌گر main رو می‌بره جلو تا برسه به آخرین commit در feature.

یعنی:

main ----> (پیش از merge)

\

feature ---- A ---- B

بعد از fast-forward merge:

main ---- A ---- B (اشاره‌گر main فقط جلو رفته)

📌 هیچ merge commit خاصی ساخته نمی‌شه، تاریخچه خطیه، انگار همه چی همیشه روی main بوده.

🔁 Rebase یعنی چی؟

حالا فرض کن توی main و feature هر دو تغییراتی اتفاق افتاده. حالا می‌خوای تغییرات feature رو به main ببری، اما به‌جای merge، می‌خوای تاریخچه‌ات تمیز و خطی بمونه.

در rebase، Git میاد commitهای feature رو برمی‌داره، می‌ذاره سر main، انگار که feature از همون‌جا ساخته شده بوده.

main ---- M1 ---- M2

\

feature ---- F1 ---- F2

بعد از:

git switch feature

git rebase main

نتیجه:

main ---- M1 ---- M2 ---- F1' ---- F2'

📌 F1' و F2' همون تغییرات قبلی‌ان، ولی با hash جدید چون زمان، parent و ترتیب عوض شده.

تما! در این بخش از آموزش، قراره درباره‌ی یکی از مهم‌ترین و البته حساس‌ترین ویژگی‌های Git صحبت کنیم:

✅ Undo کردن تغییرات یا همون برگردوندن اشتباهات.

بیاین مرحله‌به‌مرحله با زبان ساده و لحن خودمونی ببینیم Git چطوری به دادمون می‌رسه.

🧭 چرا Undo مهمه؟

یکی از بزرگ‌ترین مزایای Git اینه که تقریباً هیچ کاری غیرقابل بازگشت نیست.

اگه یه فایل رو اشتباهی پاک کردی، یا یه commit بد ساختی، یا حتی یه rebase خراب اجرا کردی... Git راه برگشت داره.

اما باید بدونی کِی، چی رو و با چه دستوری باید برگردونی.

✨ داستان کوچیک: اشتباه انسانی

فرض کن یه دولوپر به‌نام webflix می‌خواست اسم فیلم موردعلاقه‌ش رو به فایل titles.txt اضافه کنه.

ولی به‌جای اضافه کردن، کل فایل رو پاک کرد و فقط یه خط جدید نوشت:

The Internship

برای شبیه‌سازی این اشتباه توی پروژه‌مون:

echo "The Internship" > titles.txt

git status

git add titles.txt

git commit -m "J: interns am I right?"

خب حالا می‌خوایم این تغییر رو برگردونیم.

🔁 git reset — غول بازگشت!

دستور git reset یکی از ابزارهای اصلی برای undo کردنه. ولی چند مدل مختلف داره:

🟡 --soft

فقط commit‌ها رو برمی‌گردونه

تغییراتت رو نگه می‌داره و توی staging می‌ذاره

مناسب برای وقتی که می‌خوای پیام commit رو عوض کنی یا کامیتتو بهتر بنویسی

🔴 --hard

همه‌چیزو پاک می‌کنه: commit + stage + unstaged

مثل زدن دکمه "بازگشت به تنظیمات کارخانه" 😬

خطرناکه، چون ممکنه کدت برای همیشه از بین بره (مخصوصاً اگه هنوز commit نشده باشه)

🧪 اجرای reset نرم (soft)

می‌خوایم برگردیم به قبل از کامیت J:

git reset --soft HEAD~1

یا اگه می‌خوای به یه commit خاص بری (مثلاً I):

git log --oneline

git reset --soft <commit-hash-of-I>

بعد از این دستور:

کامیت J از تاریخچه حذف می‌شه

تغییراتی که توش بود، هنوز توی staging هستن

می‌تونی راحت‌تر تغییرش بدی یا دوباره commit کنی

🔎 بررسی وضعیت بعد از soft reset:

git status # ببین تغییرات توی staging هست یا نه

git diff --staged # محتوای staged فعلی رو نشون می‌ده

git log --oneline # ببین الان روی کدوم کامیت هستی

💥 اجرای reset سخت (hard)

اگه بخوای کدت رو کامل برگردونی و همه‌چیزو پاک کنی:

git reset --hard HEAD~1

یا:

git reset --hard <commit-hash>

این کار:

شاخه‌تو می‌بره به اون commit

فایل‌ها رو توی disk هم به اون وضعیت برمی‌گردونه

تمام تغییرات ذخیره‌نشده‌ت از بین می‌رن

❗ اگه فایل یا تغییری هنوز commit نشده باشه و hard reset بزنی، دیگه راه برگشتش خیلی سخته.

🔍 فرق بین Unstaged و Untracked چیه؟

Unstaged: فایلی که تغییر دادی ولی هنوز git add نکردی

Untracked: فایلی که جدید ساختی و اصلاً Git نمی‌شناسدش (هنوز git add نشده)

reset --hard فقط روی Unstaged اثر داره. Untracked فایل‌ها باقی می‌مونن.

🧨 خطر واقعی: از بین رفتن داده

فرض کن یه فایل رو تازه ساختی ولی هنوز add نکردی، اگه بزنی:

git reset --hard

اون فایل برای همیشه پاک می‌شه.

راه نجات نداری، چون Git هیچ‌وقت اون فایل رو نشناخته که بخواد ذخیره‌ش کنه.

🔁 Reset به commit خاص

اگه بخوای شاخه‌تو ببری عقب به یه commit مشخص:

git reset --hard <commit-hash>

این کار هم شاخه رو می‌بره عقب، هم فایل‌هاتو پاک می‌کنه و به اون لحظه برمی‌گردونه.

✅ ولی آیا واقعاً همه‌چیز نابود می‌شه؟

نه کاملاً. Git یه ابزار نجات داره به اسم:

📦 git reflog

اگه اشتباهی reset کردی، با git reflog می‌تونی ببینی Git شاخه‌تو قبلاً کجاها برده، و برگردی.

git reflog

git checkout <previous-hash>

یه‌جور تاریخچه‌ی حرکات Branch هست. خیلی مواقع می‌تونه نجاتت بده.

🌐 Remote چیه و چرا اهمیت داره؟

فرض کن داری روی یه پروژه‌ی برنامه‌نویسی کار می‌کنی. تو لپ‌تاپ خودت یه‌سری تغییر اعمال می‌کنی، یکی دیگه از اعضای تیم هم روی سیستم خودش داره تغییراتی می‌ده. حالا چطور این تغییرات باید با هم به‌اشتراک گذاشته بشه؟ اینجا دقیقاً جاییه که Remote وارد می‌شه.

✅ Remote یعنی چی؟

در ساده‌ترین تعریف:

Remote Repository به یک نسخه‌ی خارجی از مخزن (Repository) گفته می‌شه که معمولاً روی یک سرور یا فضای آنلاین قرار داره و Git می‌تونه با اون ارتباط بگیره.

یعنی در واقع، Remote نسخه‌ای از پروژه‌ت هست که روی یه جای دیگه ذخیره شده—مثلاً روی GitHub، GitLab

🧠 چرا Git Remote مهمه؟

Git به‌صورت پیش‌فرض یه ابزار local هست؛ یعنی همه‌چیز فقط روی سیستم خودت انجام می‌شه. اما وقتی بخوای به صورت تیمی (collaborative) کار کنی، یا پروژه‌تو روی یه سرور مثل GitHub منتشر کنی، یا حتی فقط یه backup از کدت داشته باشی، وجود یه Remote Repository ضروریه.

در واقع، بدون Remote، Git یه ابزار شخصیه.  
ولی با Remote، Git تبدیل می‌شه به یه ابزار حرفه‌ای برای توسعه‌ی تیمی، همکاری، انتشار و مدیریت کد.

💡 GitHub چیه و چه فرقی با Git داره؟

یکی از رایج‌ترین اشتباه‌ها بین برنامه‌نویس‌های تازه‌کار اینه که Git و GitHub رو یکی بدونن. در حالی که این دوتا ابزار، دو چیز کاملاً متفاوت هستن.

Git یک نرم‌افزار کنترل نسخه (Version Control System)‌ هست که بهت اجازه می‌ده تغییرات کدت رو مرحله‌به‌مرحله ثبت، پیگیری و در صورت نیاز برگردونی. Git روی سیستم خودت نصب می‌شه و تمام عملیات‌هاش—از ساخت شاخه تا commit و reset—همه‌چیز به‌صورت آفلاین انجام می‌شن. یعنی حتی اگه اینترنت هم نداشته باشی، باز می‌تونی با Git کار کنی.

اما GitHub یک پلتفرم مبتنی‌بر اینترنت یا به‌عبارتی یک سرویس ابریه که بهت اجازه می‌ده مخازن Git‌ات رو روی فضای آنلاین ذخیره و به اشتراک بذاری. GitHub در واقع یک سایت بزرگه که میزبان میلیون‌ها مخزن Git هست. خیلی از توسعه‌دهنده‌ها ازش برای همکاری تیمی، مشارکت در پروژه‌های متن‌باز (open-source)، و انتشار عمومی کدهاشون استفاده می‌کنن.

پس:

* Git موتور اصلیه، همون ابزاریه که تغییرات رو مدیریت می‌کنه.
* GitHub صرفاً یه مکان برای نگهداری و تعامل آنلاین با مخازن Git ـه.

GitHub فقط یک نوع Remote برای Git هست — یعنی یه نسخه‌ی اینترنتی از مخزن شما. یه مخزنه که روی سروره و بقیه هم می‌تونن بهش دسترسی داشته باشن.

🧰 دستورات مهم برای کار با Git Remote (ریموت‌ها)

وقتی بخوای پروژه‌ت رو با یه مخزن دیگه (مثلاً روی GitHub) هماهنگ کنی، Git یه‌سری ابزار در اختیارت می‌ذاره که بهت اجازه می‌دن اون ریموت رو بشناسی، تغییراتش رو بگیری یا براش بفرستی. بیاین این دستورات مهم رو با توضیح کامل بررسی کنیم:

🔗 git remote add origin <url>

این دستور برای اضافه کردن یک ریموت جدید به پروژه‌ته.  
وقتی آدرس یه مخزن (مثلاً GitHub) رو به پروژه‌ت معرفی می‌کنی، Git می‌تونه باهاش ارتباط بگیره.

* origin اسم قراردادیه که معمولاً برای ریموت اصلی استفاده می‌شه.
* <url> می‌تونه یه آدرس HTTPS یا SSH باشه، مثل:  
  https://github.com/username/myproject.git

این دستور فقط اتصال رو می‌سازه، هنوز هیچ داده‌ای رد و بدل نشده.

📥 git fetch origin

این دستور می‌گه:  
«برو آخرین تغییرات مخزن origin رو بگیر، ولی وارد پروژه‌م نکن! فقط دانلود کن.»

* با این کار، Git فقط تاریخچه‌ی commit‌ها و شاخه‌ها رو می‌گیره.
* هیچ تغییری توی فایل‌های لوکال تو ایجاد نمی‌شه.
* برای مقایسه‌ی تغییرات یا دیدن وضعیت ریموت مفیده.

🔄 git pull origin main

اینجا دیگه کار جدی‌تر می‌شه:  
این دستور یعنی:  
«برو تغییرات شاخه‌ی main روی origin رو بگیر و بیار توی شاخه‌ی فعلی من ادغام کن.»

* انگار fetch + merge با هم زده باشی.
* اگه شاخه‌ی تو عقب‌تر از ریموته، commitهای جدید وارد می‌شن.
* ممکنه در صورت تضاد، conflict پیش بیاد.

🚀 git push origin main

وقتی می‌خوای تغییراتت رو بفرستی روی ریموت، این دستور به‌کارت میاد.

* شاخه‌ی main از سیستم تو، روی origin آپلود می‌شه.
* این همون کاریه که وقتی پروژه‌ت رو با دنیا share می‌کنی استفاده می‌کنی.

👁️‍🗨️ git remote -v

با این دستور می‌تونی لیست همه‌ی ریموت‌هایی که به پروژه‌ت اضافه کردی رو ببینی.

* -v مخفف verbose هست و آدرس دقیق اون‌ها رو هم نشون می‌ده.
* خروجی معمولاً چیزی شبیه اینه:

🎯 مفهوم لوکال و ریموت در Git

حواست باشه:

Git یه سیستم کنترل نسخه توزیع‌شده (Distributed Version Control System) هست. یعنی هیچ مخزن مرکزی‌ای وجود نداره که "رسمی‌ترین" یا "اصل‌ترین" باشه.

هر رپوزیتوری (repo) یه نسخه‌ی کامل و مستقل از پروژه‌ست. تو می‌تونی هر تعداد رپوی مستقل داشته باشی و بینشون سینک (sync) انجام بدی.

حالا معمولاً برای راحتی کار، تیم‌ها یه رپو رو به عنوان مرجع اصلی (مثلاً GitHub) در نظر می‌گیرن، و بقیه اعضا فقط از اون pull و بهش push می‌کنن.

git

🔗 اتصال پروژه به GitHub

حالا وقتشه که پروژه‌مون (مثلاً webflix) رو به GitHub وصل کنیم.

قدم‌ها:

وارد GitHub بشو و یه رپوی جدید بساز به اسم webflix.

→ توی صفحه اصلی، روی دکمه New کلیک کن، اسم webflix رو وارد کن، و روی Create Repository بزن.

حالا باید این رپو رو به پروژه‌ی لوکال‌مون وصل کنیم:

git remote add origin https://github.com/<username>/webflix.git

با این دستور مطمئن شو که اتصال برقرار شده:

git ls-remote

حالا اگه خواستی، می‌تونی تغییراتت رو Push کنی:

git push origin main

✅ ساختن Branch و Push کردن به GitHub

مثلاً می‌خوای یه شاخه جدید به اسم foo بسازی و بفرستی به GitHub:

git switch -c foo

git push origin foo

اگه بری داخل GitHub می‌بینی Branch جدید foo به لیست اضافه شده.

🧪 تست Pull Request

حالا می‌خوایم تست کنیم که چطور میشه از GitHub برای commit استفاده کرد و بعد تغییرات رو توی پروژه‌ی لوکال خودمون بگیریم (pull کنیم):

وارد GitHub شو، مثلاً فایل classics.txt رو باز کن، یه خط جدید بهش اضافه کن و Commit کن (با پیام مثلا J: update classic list).

بعد از اینکه از طریق UI تغییرات رو دادی، برگرد به ترمینال:

git pull origin main

اگه همه‌چی درست باشه، تغییرات به پروژه‌ی لوکال تو اضافه می‌شن.

⚠️ Diverging Branches (شاخه‌های واگرا)

اگه خطایی مثل diverging branches گرفتی، یعنی شاخه‌ات روی سیستم با شاخه‌ی روی GitHub تفاوت داره.

برای حلش:

git config pull.rebase false

git pull origin main

این‌طوری Git به‌جای Rebase، از Merge استفاده می‌کنه.

🔀 ساختن Pull Request (در GitHub)

اگه داری روی یه ویژگی جدید کار می‌کنی (مثلاً اضافه کردن لیست فیلم‌های کلاسیک)، می‌تونی:

یه Branch بسازی:

git switch -c add-classics

فایلت رو ویرایش و Commit کن:

git add classics.txt

git commit -m "Add Alfred Hitchcock classics"

git push origin add-classics

برو داخل GitHub، یه Pull Request جدید باز کن تا تغییراتت به شاخه‌ی اصلی (main) اضافه بشه.

🔁 Fork یعنی چی و چرا باید انجامش بدیم؟

Fork در دنیای GitHub یعنی:  
یه نسخه‌ی کامل و مستقل از یک مخزن (Repository) روی حساب کاربری خودت بسازی، طوری‌که بتونی تغییرات دلخواهت رو روش انجام بدی بدون اینکه به نسخه‌ی اصلی آسیبی بزنی.

درواقع، Fork نوعی کپی‌برداری پیشرفته‌ست که مخصوص همکاری در پروژه‌های متن‌بازه.

وقتی یه پروژه‌ی public رو در GitHub می‌بینی و می‌خوای:

* چیزی بهش اضافه کنی،
* یه باگ رو درست کنی،
* یا حتی فقط توی نسخه‌ی خودت باهاش تمرین کنی،

بهترین و رایج‌ترین راه، اینه که اون پروژه رو Fork کنی.

برخلاف clone که فقط نسخه‌ای از پروژه رو برای خوندن یا تست‌کردن بهت می‌ده، Fork یه مخزن جداگانه در حساب GitHub خودت ایجاد می‌کنه — با تمام قابلیت‌ها و دسترسی‌ها.

🧩 اگه فقط clone کنی، درواقع داری یه کپی محلی از پروژه رو می‌گیری که فقط برای مشاهده یا تست کاربرد داره — بدون اینکه دسترسی برای تغییر در مخزن اصلی داشته باشی.

مگر اینکه یکی از اعضای رسمی پروژه (Contributor) باشی

خب، حالا که فهمیدیم Fork چیه و چرا لازمه، وقتشه ازش استفاده کنیم.

اولین قدم برای همکاری با پروژه‌ی MegaCorp اینه که یه Fork از مخزن اصلی توی حساب GitHub خودمون بسازیم.

یعنی چی؟  
یعنی یه کپی کامل از اون پروژه داخل حساب ما ساخته می‌شه، که می‌تونیم هر کاری خواستیم روش انجام بدیم — بدون اینکه به پروژه‌ی اصلی صدمه بزنیم.

✅ می‌ریم به صفحه‌ی پروژه‌ی MegaCorp توی GitHub و روی دکمه‌ی Fork کلیک می‌کنیم.

نکته مهم:  
Fork کردن یک عملیات Git نیست.  
یعنی دستور git fork وجود نداره.

Fork یکی از قابلیت‌های سرویس‌هایی مثل GitHub، GitLab یا هست — نه خود Git.  
این سرویس‌ها با استفاده از Git، یه کپی جدید از پروژه می‌سازن و اون رو به‌عنوان "Fork" به پروژه‌ی اصلی وصل می‌کنن.

🔗 وقتی می‌خوای یه مخزن خارجی رو اضافه کنی، چه اسمی براش انتخاب می‌کنی؟

اول از همه باید بدونیم origin توی گیت به چه معنا هست

وقتی یه پروژه‌ی Git رو clone می‌کنی یا یه remote repo رو بهش اضافه می‌کنی، Git به طور پیش‌فرض اون remote رو با اسم origin می‌شناسه.

🔹 origin = مخزنی که تو معمولاً باهاش کار می‌کنی، push و pull انجام می‌دی.  
مثلاً اگه یه پروژه رو از GitHub خودت clone کنی، origin همون مخزن توی GitHubه.

📌 اسم origin فقط یه قرارداد رایجه. تو می‌تونی هر اسمی که بخوای بذاری، ولی عرف همینه.

⬆️ UPSTREAM چیه؟

حالا فرض کن اون پروژه‌ای که fork کردی، همچنان در حال توسعه‌ست و تغییراتی روش داره اعمال می‌شه.  
تو هم می‌خوای این تغییرات رو توی نسخه‌ی خودت داشته باشی.

🔹 اینجاست که یه remote دیگه اضافه می‌کنی به اسم upstream.  
🔸 upstream = مخزن اصلی‌ای که ازش Fork کردی.  
معمولاً فقط ازش pull می‌کنی تا پروژه‌ت رو به‌روز نگه داری.

🔄 مراحل معمول همکاری با پروژه‌های متن‌باز از طریق Fork

اگه بخوای توی یه پروژه‌ی عمومی مثل MegaCorp مشارکت کنی، مراحل کلیش اینه:

1. پروژه‌ی اصلی رو Fork می‌کنی (توی حساب GitHub خودت)
2. اون Fork رو به سیستم خودت Clone می‌کنی
3. یه Branch جدید می‌سازی (برای ویژگی یا باگ خاص)
4. تغییراتت رو انجام می‌دی و Commit می‌کنی
5. تغییرات رو به GitHub Push می‌کنی
6. یه Pull Request باز می‌کنی تا تیم پروژه تغییراتت رو بررسی و (اگه لازم بود) Merge کنه

 برو به صفحه‌ی ریپوزیتوری در GitHub.

 GitHub معمولاً خودش پیشنهاد ساخت PR می‌ده:  
**"Compare & pull request"** → روش کلیک کن.

یا از طریق تب **"Pull requests"** → دکمه‌ی **"New pull request"**.

 مبدا و مقصد رو بررسی کن:

* **base** = شاخه‌ی اصلی پروژه (مثلاً main)
* **compare** = شاخه‌ی خودت (مثلاً my-feature)

 عنوان و توضیح مناسب برای PR بنویس:

* عنوان باید واضح و توصیفی باشه.
* در توضیح می‌تونی بنویسی چه کاری انجام شده، چرا انجام شده و چه اثری داره.

 دکمه‌ی **Create Pull Request** رو بزن.

❓ Clone یا Fork؟ کدوم‌ رو انتخاب کنیم؟

* اگه فقط می‌خوای کد یه پروژه رو بخونی یا بررسی کنی و یا اینکه جزو اعضای رسمی اون پروژه هستی ، clone کافیه.
* ولی اگه می‌خوای تغییراتی اعمال کنی و به پروژه‌ی اصلی پیشنهاد بدی، باید fork کنی.

🔁 Git Reflog چیه و چرا نجات‌دهنده‌ست؟

خیلی وقت‌ها پیش میاد که توی گیت یه کار اشتباه انجام می‌دی — مثلاً:

یه شاخه رو پاک می‌کنی،

با git reset برمی‌گردی به یه commit قبلی،

یا با commit --amend چیزی رو تغییر می‌دی و نسخه‌ی قبلی از بین می‌ره.

تو این شرایط، معمولاً با دستور git log دیگه اون commit‌ یا شاخه‌ی قبلی رو نمی‌بینی، و ممکنه فکر کنی برای همیشه از دست رفتن... ولی نترس! چون اینجا git reflog به دادت می‌رسه.

🧠 Reflog دقیقاً چیه؟

reflog مخفف reference log هست و به زبان ساده یعنی:

گیت تمام حرکت‌هایی که HEAD انجام داده رو یادش می‌مونه — یعنی هر دفعه که شاخه عوض کردی، reset کردی، commit زدی یا حتی delete branch انجام دادی، گیت همه‌ی این‌ها رو توی یه تاریخچه‌ی مخفی ثبت می‌کنه.

این یعنی حتی اگه شاخه‌ای رو حذف کرده باشی، با کمک reflog هنوز می‌تونی به کامیت‌های اون شاخه دسترسی پیدا کنی.

🔍 چطوری ازش استفاده کنیم؟

فقط کافیه توی ترمینال بزنی:

git reflog

و گیت یه لیست از تمام تغییر موقعیت‌های HEAD رو بهت نشون می‌ده؛ مثلاً:

e7f22ad HEAD@{0}: reset: moving to e7f22ad

d03f5a8 HEAD@{1}: commit: add slander file

4acb91e HEAD@{2}: checkout: moving from main to slander

...

هر خط یعنی یه حرکت – از یه شاخه به شاخه‌ی دیگه، از یه commit به یه commit دیگه، یا reset و غیره.

🛟 چطور از Reflog برای بازیابی استفاده کنیم؟

فرض کن یه فایل مهم رو توی یه commit نوشتی، ولی بعدش اشتباهی اون شاخه رو پاک کردی. با git log چیزی نمی‌بینی. اما:

با git reflog دنبال کامیتی که می‌خوای بگرد.

SHA اون کامیت (مثلاً d03f5a8) رو بردار.

با یکی از روش‌های زیر بازیابی کن:

روش ۱: ساختن شاخه از اون commit

git checkout -b recovery d03f5a8

یا

روش ۲: Cherry-pick کردن اون commit توی شاخه فعلی

git cherry-pick d03f5a8

یا

روش ۳: فقط گرفتن محتوای فایل با plumbing (پیشرفته)

git cat-file -p d03f5a8

🎯 چه موقع‌هایی reflog به‌دردت می‌خوره؟

وقتی کامیت مهمی رو حذف کردی

وقتی شاخه‌ای رو پاک کردی و می‌خوای برش گردونی

وقتی با rebase خراب‌کاری کردی و همه چی بهم ریخته!

وقتی نمی‌دونی از آخرین بار چی کار کردی و می‌خوای ردّش رو پیدا کنی

🧱 Squash یعنی چی؟ و چرا ممکنه بهش نیاز داشته باشیم؟

توی تیم‌های برنامه‌نویسی، سلیقه‌ها و استانداردهای مختلفی برای ثبت commit وجود داره.

بعضی تیم‌ها دوست دارن هر Pull Request فقط یه commit داشته باشه. بعضیا ترجیح می‌دن تاریخچه‌ی دقیق‌تری از commitهای جزئی ببینن.

🔧 اما حتی اگر تیم شما روی «یک commit» حساس نباشه، باز هم squash کردن می‌تونه مفید باشه:

تاریخچه‌ی پروژه تمیز و قابل فهم می‌مونه.

وقتی یه ویژگی در چندین commit پراکنده شده، squash می‌تونه همه رو یکی کنه و جمع‌وجور نگه داره.

🎯 Squash دقیقاً چیه؟

Squash یعنی گرفتن تغییرات چند commit و فشرده کردنشون توی یه commit واحد.

مثلاً اگه سه تا commit A، B و C داشته باشی، با squash می‌تونی همه‌شونو توی یه commit جدید مثل X بریزی.

انگار می‌گی: «این سه‌تا در اصل یک کار بودن و حالا توی تاریخچه، فقط با یک اسم ثبت بشن.»

🛠 چطور squash کنیم؟

Squashing معمولاً با دستور git rebase -i انجام می‌شه:

فرض کن می‌خوای ۳ commit آخر رو یکی کنی.

بنویس:

git rebase -i HEAD~3

Git یه ویرایشگر برات باز می‌کنه که لیست commitها رو نشون می‌ده:

pick 123abc Commit A

pick 456def Commit B

pick 789ghi Commit C

کافیه اولی رو بذاری pick، بقیه رو squash یا s کنی:

pick 123abc Commit A

squash 456def Commit B

squash 789ghi Commit C

بعد یه مرحله می‌رسی که می‌تونی متن commit نهایی رو بنویسی یا ویرایش کنی.

به همین راحتی ولی باید حواستون باشه که

Squash کردن باعث تغییر تاریخچه می‌شه.

اگه این تغییر رو روی یک شاخه‌ی عمومی مثل main انجام بدی که قبلاً روی GitHub یا جای دیگه Push شده، ممکنه دردسر درست شه، چون commitهای قدیمی حذف و با یه commit جدید جایگزین می‌شن.

اگه می‌خوای همچین کاری رو روی شاخه‌ی اصلی انجام بدی، باید git push --force بزنی — که خطرناکه و فقط باید با آگاهی کامل انجامش بدی.

🧠 چه زمانی squash کنیم؟

وقتی توی شاخه‌ی feature کار می‌کنی و قبل از Merge کردن به main می‌خوای همه‌ی commit‌هات یکپارچه باشه.

✅ خلاصه‌اش:

squash یعنی ترکیب چند commit به یک commit.

برای مرتب‌سازی تاریخچه و کاهش شلوغی به درد می‌خوره.

با git rebase -i HEAD~n انجام می‌شه.

برای شاخه‌های عمومی (مثل main) خطرناک‌تره و نباید بدون دلیل force push انجام بدی.

وقتی توی پروژه‌ات داری روی یه فیچر کار می‌کنی، ممکنه وسط کار مجبور بشی بری سراغ یه تسک فوری دیگه (مثلاً رفع یک باگ). اما هنوز تغییراتت رو کامل نکردی و نمی‌خوای اون‌ها رو commit کنی یا از دست بدی.

🔧 اینجا دستور git stash به کارت میاد.

**چی کار می‌کنه؟**

۱. هر تغییری که دادی — چه اونایی که فقط save کردی (unstaged)، چه اونایی که stage کردی — برداشته می‌شن و به‌صورت موقت در یک «محل پنهان» ذخیره می‌شن.  
۲. بعد از اون، Git فایل‌های پروژه‌ات رو به آخرین commit برمی‌گردونه، یعنی همون حالتی که قبل از شروع تغییرات بودن.

بعدش هم به‌راحتی با یه دستور stash pop برمی‌گردی سراغ همون جایی که بودی، انگار هیچ اتفاقی نیفتاده.

بدون نیاز به commit موقت، بدون clone اضافی، بدون دردسر.

اگر بخوای حرفه‌ای با Git کار کنی، یاد گرفتن stash جزو اون مهارت‌های کوچیکی‌ـه که تفاوت بزرگی ایجاد می‌کنه

**🧠 فرقش با commit چیه؟**

stash مثل commit نیست که وارد تاریخچه پروژه بشه.  
یه مکان موقتیه — مثل clipboard سیستم‌ت.  
و دقیقاً هم به همین خاطر **نباید از stash برای نگهداری طولانی‌مدت استفاده کنی.**

.

**🧠 تعریف Stash**

git stash ابزاری در Git برای ذخیره‌ی موقتی تغییرات فعلیه — هم تغییرات **unstaged** و هم **staged** — بدون نیاز به commit.

Git این تغییرات رو توی یک ساختار پشته‌ای (stack) ذخیره می‌کنه و پروژه‌ی شما رو به آخرین commit بازمی‌گردونه؛ یعنی انگار هیچ تغییری ندادید.

**📋 چرا stash مهمه؟**

بجای commit موقت یا گرفتن clone جداگانه، می‌تونی با stash تغییرات نیمه‌کاره‌ت رو کنار بذاری، روی مشکل فوری تمرکز کنی، و بعداً دوباره ادامه بدی.

**🎯 دستورات اصلی stash**

**🔹 git stash**

ذخیره تغییرات فعلی (هم staged، هم unstaged).

**🔹 git stash list**

دیدن لیست stashها — مثل یه استک (stack) هست، یعنی آخرین چیزی که ذخیره کردی، اولی هست که بازیابی می‌شه (LIFO).

**🔹 git stash pop**

آخرین stash رو از لیست بیرون می‌کشه و دوباره به پروژه‌ت اضافه می‌کنه.  
**هم ذخیره رو اعمال می‌کنه، هم حذف می‌کنه.**

**🔹 git stash apply**

فقط اعمال می‌کنه، ولی توی لیست stash باقی می‌مونه.

**🔹 git stash drop stash@{1}**

یه stash خاص رو از لیست پاک می‌کنه.

**🔹 git stash save "توضیح"**

برای اضافه کردن پیام به stash (اختیاری ولی برای فهم بهتر خوبه).

🎯 Git Revert دقیقاً چیکار می‌کنه؟

✅ **یک commit خاص رو "معکوس" می‌کنه و نتیجهٔ این معکوس کردن رو به‌صورت یک commit جدید، به تاریخچه پروژه اضافه می‌کنه.**

**🧪 یعنی چی "برعکس" می‌کنه؟**

فرض کن یه commit داشتی که توش محتوای یک فایل رو تغییر دادی یا یه فایل حذف کردی.

وقتی git revert <commit-hash> رو اجرا می‌کنی:

* Git میاد تغییرات اون commit رو **برعکس** می‌کنه (یعنی اگر چیزی اضافه شده بود، حذف می‌کنه؛ اگر چیزی پاک شده بود، برمی‌گردونه).
* بعد **یه commit جدید** ایجاد می‌کنه که این برعکس‌سازی توشه.

در نتیجه:

تاریخچه پروژه دست‌نخورده باقی می‌مونه ✅

همه می‌دونن چه چیزی انجام شده و کی برگشت خورده 🧠

برای شاخه‌های عمومی (مثل main) کاملاً امنه 🔐

🛠 چطور revert کنیم؟

اول باید hash اون commit که می‌خوای برگردونی رو پیدا کنی:

git log --oneline

بعد با دستور زیر اون commit رو revert می‌کنی:

git revert <commit-hash>

Git بهت اجازه می‌ده که پیام commit جدید رو ویرایش کنی.

بهتره توضیح بدی چرا revert کردی (مثلاً «رفع خطای مستندات مارکتینگ»).

🧠 چه زمانی از reset استفاده کنیم؟

وقتی روی شاخه‌ی شخصی یا feature branch کار می‌کنی

وقتی می‌خوای تغییرات اشتباهی رو پاک کنی و از اول انجام بدی

وقتی قصد داری قبل از ارسال pull request، تاریخچه رو تمیز و مرتب کنی

🧠 و چه زمانی revert مناسب‌تره؟

وقتی commit اشتباهی وارد شاخه‌ی عمومی شده (مثل main یا release)

وقتی نمی‌خوای تاریخچه رو به‌هم بزنی ولی باید اثر یه commit رو برداری

وقتی تیم‌های دیگه هم دارن روی همون شاخه کار می‌کنن و نمی‌خوای مشکل ایجاد کنی

📝 نتیجه‌گیری

git revert ابزار برای خنثی‌سازی تغییرات بدون از بین بردن تاریخچه.

برخلاف reset، برای پروژه‌هایی که چند نفر روی یه شاخه کار می‌کنن کاملاً ایمن‌ه.

اما این ابزار برای مواقع اضطراری یا مدیریت تغییرات ناخواسته بسیار کاربردیه.

🍒 Git Cherry-Pick چیه و چه زمانی استفاده می‌شه؟

git cherry-pick یک دستور در Git هست که بهت اجازه می‌ده **فقط یک یا چند commit خاص از یک**

**شاخه‌ی دیگر رو به شاخه‌ی فعلی‌ت بیاری** — بدون اینکه کل تاریخچه یا ساختار اون شاخه رو وارد کنی.

درواقع، وقتی cherry-pick می‌کنی، Git محتوای commit مورد نظر رو گرفته و **به‌صورت یک commit**

**جدید** توی شاخه‌ی فعلی ذخیره می‌کنه.

این commit جدید، محتوای همون تغییرات رو داره، ولی انگار از ابتدا در شاخه فعلی ایجاد شده.

.فرض کن تو شاخه‌ bug fixیه باگ رو درست کردی، ولی نمی‌خوای همه‌ی تغییرات اون شاخه برن روی سایت.

با cherry-pick فقط همون commit باگ‌فیکس رو میاری تو شاخه‌ی main و سریع منتشرش می‌کنی.

🛠 مراحل انجام cherry-pick

1. ✅ مطمئن شو شاخه‌ت تمیزه

یعنی هیچ فایل unstaged یا تغییر ذخیره‌نشده‌ای نداشته باشی:

git status

2. 🔍 شناسایی commit مورد نظر

با استفاده از git log کامیتی که می‌خوای انتخاب کن و hash اون رو بردار:

git log --oneline

مثلاً خروجی:

3e25abc Fix: typo in partners name

a15c9d2 Add partners section to homepage

تو می‌خوای فقط کامیت اول (3e25abc) رو cherry-pick کنی.

3. 🍒 اجرای cherry-pick

فرض کن الان روی شاخه release هستی و می‌خوای کامیت 3e25abc رو از main بیاری:

git cherry-pick 3e25abc

Git اون کامیت رو برمی‌داره و به عنوان یه کامیت جدید (با همون تغییرات) توی release اعمال می‌کنه.

💡 نکات مهم درباره cherry-pick

cherry-pick تاریخچه کامل شاخه رو نمیاره، فقط همون کامیت خاص رو منتقل می‌کنه.

این کار مخصوصاً وقتی مفیده که نمی‌خوای ویژگی‌های ناقص شاخه توسعه وارد شاخه انتشار بشن.

ممکنه در بعضی موارد با conflict مواجه بشی؛ باید به‌صورت دستی حلش کنی.

git bisect بهت کمک می‌کنه **خیلی سریع اون commitی رو پیدا کنی که باعث به‌وجود اومدن یه باگ شده.**

این کار رو با استفاده از تکنیکی به اسم **جستجوی دودویی (Binary Search)** انجام می‌ده.

**🧠 سناریو فرضی:**

فرض کن یه branch داری که کلی commit روش هست. آخرین commit‌ت باگ داره، ولی می‌دونی که مثلاً صد commit قبل‌تر، اون باگ وجود نداشته. حالا می‌خوای بفهمی که **از کی این باگ وارد شده؟**

خب یه راه اینه که دونه‌به‌دونه همه‌ی commitها رو بررسی کنی تا باگ‌دار رو پیدا کنی…  
اما این کار خیلی خسته‌کننده‌ست، مخصوصاً اگه هر بررسی چند دقیقه زمان ببره. ممکنه چند ساعت یا حتی چند روز طول بکشه.

ولی چون commitها بر اساس زمان مرتب شدن، می‌تونیم باهاشون مثل یه **آرایه‌ی مرتب** رفتار کنیم و با الگوریتم جستجوی دودویی (Binary Search) بررسی‌شون کنیم.

**🧪 Git Bisect چطور کار می‌کنه؟**

بیاید فرض کنیم:

* یه وضعیت "خراب" داریم که توش باگ وجود داره،
* یه وضعیت "سالم" هم داریم که توش باگ نبوده،
* و کلی commit بین این دوتاست (تعدادش رو X در نظر بگیر).

حالا اگه وسط این دو نقطه رو بررسی کنیم و معلوم بشه اون commit وسطی **سالمه**، می‌فهمیم که نصف اول commitها مشکلی ندارن. یعنی فضای جستجو رو نصف کردیم، و فقط X/2 + 1 commit باقی می‌مونه که باید بررسی کنیم.

اگه این روند رو تکرار کنیم، هی فضا رو کوچیک‌تر و کوچیک‌تر می‌کنیم تا در نهایت فقط **یه commit باقی بمونه** — همون commitی که باگ رو وارد کرده.

مثلاً اگه X برابر با 1000 باشه، با حدود **۱۰ بار بررسی** می‌تونی باگ‌دار رو پیدا کنی! 🔍

**🛠 استفاده از Git Bisect خیلی ساده‌ست:**

1. مشخص می‌کنی کدوم commit **خراب**ه (مثلاً HEAD)
2. مشخص می‌کنی کدوم commit **سالم**ه (مثلاً یک نسخه‌ی قدیمی‌تر)
3. دستور git bisect start رو می‌زنی، و گیت ازت می‌پرسه:  
   «این commit خوبه یا بده؟»

اگه بگی **خوبه**، گیت می‌فهمه که تمام commitهای قبلش خوب بودن.  
اگه بگی **بده**، می‌فهمه که مشکل از اینجا به بعده.

این فرایند اونقدر تکرار می‌شه تا به commit مشکل‌دار برسی.

**🔧 رفع باگ بعد از پیدا شدن**

بعد از اینکه commit مشکل‌دار رو پیدا کردی، دو تا راه داری:

1. **Revert (برگردوندن commit)**: بیشتر توی تیم‌های بزرگ استفاده می‌شه.
2. **Fix forward (رفع و ادامه دادن)**: یعنی یه commit جدید می‌نویسی که مشکل رو حل می‌کنه. این روش توی تیم‌های کوچیک بیشتر رایجه.

در حالت ایده‌آل، همیشه بهتره که **fail forward** انجام بدی — یعنی بدون پاک‌کردن تاریخچه، یه commit جدید بنویسی که کد رو درست کنه. این کار شفاف‌تر، حرفه‌ای‌تر و امن‌تره.

🏷️ Git Tag چیست؟

Tag (برچسب) در Git یک اشاره‌گر ثابت و غیرقابل‌تغییر به یک commit خاص است.

tag به یک commit خاص قفل میشه و حرکت نمی‌کنه.

📌 چرا از Tag استفاده می‌کنیم؟

تگ‌ها در Git برای نشانه‌گذاری نقاط مهم در تاریخچه پروژه به کار می‌رن. معمولاً زمانی که بخوای نسخه‌ای پایدار از پروژه رو منتشر کنی، یا بخوای وضعیت خاصی از کد رو ذخیره و بعدها بهش مراجعه کنی، از تگ استفاده می‌کنی. این تگ‌ها مثل برچسب‌هایی هستن که به یک commit خاص چسبیده می‌شن تا مثلاً بگن «اینجا نسخه ۱.۰.۰ منتشر شد» یا «در این نقطه به یک milestone مهم رسیدیم». به این شکل می‌تونی به‌راحتی نسخه‌های مهم دولوپ پروژه رو دنبال کنی

مثال واقعی:

تو مخزن React در GitHub اگه بخش "Tags" رو ببینی، می‌بینی که نسخه‌هایی مثل v18.2.0 یا v17.0.1 به صورت tag مشخص شدن — نشونه‌ای از نسخه‌های منتشرشده.

🛠️ چطور تگ بسازیم؟

git tag -a v1.0.0 -m "نسخه اول پایدار"

دیدن همه تگ‌ها:

git tag

اتصال تگ به یک commit خاص (مثلاً قبلی):

git tag -a v0.9.0 <commit-hash> -m "نسخه آزمایشی"

🧭 نکات مهم:

با git checkout <tag> وارد وضعیت Detached HEAD می‌شی. یعنی نمی‌تونی مستقیماً تغییراتی روی اون تگ انجام بدی.

می‌تونی چند تگ روی یک commit داشته باشی.

تگ‌ها همون commit هستن، پس هرجا که می‌تونی از hash استفاده کنی، می‌تونی اسم تگ رو هم استفاده کنی.

تگ‌ها معمولاً طبق اصول **Semantic Versioning (SemVer)** نام‌گذاری می‌شن:.

اصول Semantic Versioning (یا به‌اختصار SemVer) یک استاندارد برای نام‌گذاری نسخه‌های نرم‌افزاره که کمک می‌کنه بفهمیم یک نسخه جدید دقیقاً چه نوع تغییری ایجاد کرده. این نسخه‌ها به‌صورت سه عدد نوشته می‌شن:

MAJOR.MINOR.PATCH

مثال: 2.4.1

هرکدوم از این عددها معنی خاصی دارن:

1. 🔴 MAJOR (نسخه اصلی)

عدد اول — وقتی تغییرات ناسازگار با نسخه‌های قبلی اتفاق می‌افته.

📌 یعنی API یا رفتار برنامه جوری عوض شده که پروژه‌های وابسته به نسخه قبلی ممکنه بشکنن.

مثال:

1.4.3 → 2.0.0

حذف یک تابع قدیمی

تغییر نام یا پارامترهای توابع

2. 🟡 MINOR (نسخه فرعی)

عدد دوم — وقتی قابلیت‌های جدید اضافه می‌کنی که با نسخه‌های قبلی سازگار هستن.

📌 یعنی تغییرات جدید باعث خرابی کد قبلی نمی‌شن.

مثال:

2.1.0 → 2.2.0

اضافه شدن یک endpoint جدید به API

اضافه شدن یک کامپوننت در UI

3. 🟢 PATCH (وصله یا اصلاح)

عدد سوم — وقتی باگ‌فیکس یا اصلاحات جزئی انجام می‌دی که هیچ تغییری در عملکرد API ایجاد نمی‌کنه.

📌 هدف فقط پایداری و رفع مشکل بدون اضافه کردن ویژگی جدیده.

مثال:

2.3.1 → 2.3.2

رفع مشکل لود شدن تصاویر

اصلاح غلط املایی در متن‌های رابط کاربری

✨ جمع‌بندی:

Tag ابزاری ساده ولی قدرتمند برای علامت‌گذاری نسخه‌هاست.

وقتی می‌خوای یک نسخه پایدار، منتشرشده، یا خاص از پروژه‌ت رو مشخص کنی، تگ‌ها بهترین راه هستن.