2. 기본 Git 명령어

저장소 생성 및 클론

✅ 1. 저장소(Repository)란?

> Git에서 저장소(Repository)란 **프로젝트의 전체 변경 이력(history)** 을 저장하는 공간이다.
> `.git` 디렉토리가 생성되면 그 폴더는 Git 저장소가 된다.

- `git init`, `git clone`

✅ 2. 저장소 생성: `git init`

🔹 사용법

```
git init
```

- 현재 디렉토리를 Git 저장소로 초기화함
- `.git`이라는 숨김 디렉토리가 생김 → 여기에 모든 Git 정보가 저장됨
- 이후부터는 `git add`, `git commit`, `git log` 등이 사용 가능해짐

🔹 예시

```
mkdir my-project
cd my-project
git init
```

```
Initialized empty Git repository in /Users/you/my-project/.git/
```

🔹 확인

```
ls -a
```

```
.git/    ← 이 폴더가 Git 저장소임을 뜻함
```

✅ 3. 저장소 복제: `git clone`

🔹 사용법

```
git clone <원격 저장소 주소>
```

- 원격 저장소(Remote repository, 예: GitHub)로부터 **전체 프로젝트와 히스토리를 복사**함
- 클론 후 자동으로 해당 디렉토리로 이동 가능

🔹 예시 (GitHub)

```
git clone https://github.com/torvalds/linux.git
```

- `linux/` 디렉토리가 생기고, 그 안에 전체 파일 + `.git` 히스토리 포함
- 자동으로 `origin`이라는 이름으로 원격 저장소가 등록됨

```
cd linux
git remote -v
```

```
origin  https://github.com/torvalds/linux.git (fetch)
origin  https://github.com/torvalds/linux.git (push)
```

✅ 4. `git init` vs `git clone` 비교

항목	`git init`	`git clone`
용도	**새로운 로컬 저장소 생성**	**기존 원격 저장소 복제**
초기 상태	빈 저장소 + `.git`	전체 파일 + 커밋 이력 포함
원격 설정	없음	자동으로 `origin` 등록
작업 흐름	로컬부터 시작 → 원격 추가	원격에서 복제 → 작업 시작

✅ 5. 원격 저장소를 수동으로 추가하려면?

`git init`으로 만든 저장소에 GitHub와 같은 원격 저장소를 수동으로 연결할 수도 있다:

```
git remote add origin https://github.com/yourname/project.git
```

- 이 명령을 써야 `git push`, `git pull` 등이 가능해짐

✅ 6. 초기 커밋 순서 정리

```
# 1. 로컬 디렉토리 생성
mkdir project && cd project

# 2. 저장소 초기화
git init

# 3. 파일 생성
echo "# README" > README.md

# 4. Git에 파일 추가
git add README.md
```

```
12
13  # 5. 커밋
14  git commit -m "Initial commit"
15
16  # 6. GitHub 원격 저장소 연결
17  git remote add origin https://github.com/you/project.git
18
19  # 7. 푸시
20  git push -u origin main
```

> `-u` 옵션은 이후 `git push`만으로 자동 추적 가능하게 함

✅ 7. 추가 꿀팁

상황	명령어
특정 브랜치만 클론	`git clone -b 브랜치명 --single-branch <URL>`
GitHub SSH 방식	`git clone git@github.com:user/repo.git`
저장소 + 특정 폴더만 클론	sparse checkout (고급 주제)

✅ 마무리 요약

명령어	설명
`git init`	로컬 Git 저장소 초기화
`git clone`	원격 저장소 복제
`git remote add origin <URL>`	수동 원격 저장소 연결
`git push -u origin main`	첫 푸시 및 브랜치 추적 설정

스테이징과 커밋

> Git은 단순히 `저장`하는 도구가 아니다.
> **"어떤 변경을, 언제, 어떻게 저장할지" 선택하고 기록하는 도구**다.

✅ 1. Git의 작업 영역 구조

Git은 다음의 **3단계 구조**로 이루어져 있다:

```
1  [Working Directory] → [Staging Area] → [Local Repository]
```

영역	설명
Working Directory	실제 파일이 있는 로컬 폴더. 편집은 여기서 이루어짐

영역	설명
Staging Area (Index)	커밋할 변경 사항을 **임시로 보관**하는 공간
Repository (.git)	커밋이 저장되는 진짜 Git 저장소 (이력 추적됨)

- `git add`, `git commit`

✅ 2. `git add` – 변경 사항을 스테이징에 올리기

```
git add <파일명>
```

📌 의미

- 변경된 파일을 **커밋 대상 후보 목록(Staging Area)** 에 추가
- 아직 커밋되지는 않았음

🧪 예시

```
echo "hello" > a.txt
git add a.txt
```

> 이제 a.txt는 "커밋 대기 중" 상태가 됨

🔹 전체 파일 스테이징

```
git add .
# 또는
git add -A
```

🔹 일부 수정만 스테이징

```
git add -p   # patch 모드로 줄 단위 선택
```

✅ 3. `git commit` – 스테이징된 변경사항을 저장소에 기록

```
git commit -m "의미 있는 메시지"
```

📌 의미

- 스테이징된 모든 변경 사항을 하나의 **스냅샷(commit)** 으로 기록
- SHA-1 해시값으로 식별되는 **불변의 커밋 객체** 생성

🧪 예시

```
git commit -m "add hello message"
```

🔹 다단계 예시

```
echo "print('hello')" > main.py
git add main.py
git commit -m "Add main.py with hello message"
```

✅ 4. 변경 상태 흐름도

```
# 파일 수정
Working Directory
    ↓ git add
Staging Area (Index)
    ↓ git commit
Local Repository (.git)
```

이 흐름을 통해 Git은 **명확한 커밋 단위 관리**가 가능해.

✅ 5. 상태 확인

🔹 변경 감지 여부 확인

```
git status
```

결과 예:

```
Changes not staged for commit:
  modified:   hello.txt

Untracked files:
  newfile.py
```

🔹 스테이징된 파일 확인

```
git diff --cached
```

> 커밋될 예정인 변경 내용 미리보기

✅ 6. 커밋 메시지 작성 팁

좋은 커밋 메시지는 협업에 큰 도움이 된다.

🔹 기본 형식

```
<타입>: <무엇을 변경했는지>
```

타입	설명 예시
feat	새로운 기능
fix	버그 수정
refactor	코드 리팩토링
docs	문서 변경
test	테스트 코드 추가/수정
chore	빌드 설정 등 기타

🔹 예시

```
git commit -m "feat: 로그인 폼 유효성 검사 추가"
```

✅ 7. 실수했을 때

🔹 스테이징 취소

```
git reset <파일명>
```

🔹 마지막 커밋 메시지 수정

```
git commit --amend
```

🔹 커밋 취소하고 다시 작업

```
git reset --soft HEAD^
```

✅ 8. 실무 예시 전체 흐름

```
# 1. 파일 수정
nano app.js

# 2. 스테이징
git add app.js

# 3. 커밋
git commit -m "fix: remove redundant console.log in app.js"
```

✅ 마무리 요약

명령어	설명
`git add`	파일을 스테이징 영역에 추가
`git commit`	스테이징된 파일을 저장소에 영구 기록
`git status`	현재 변경 상태 확인
`git diff` / `--cached`	워킹 디렉토리 ↔ 스테이징 상태 비교

- # 커밋 메시지 컨벤션 (예: Conventional Commits)

> 협업에서 커밋 메시지를 일관되게 관리하기 위한 **표준화된 작성 규칙**

✅ 1. 왜 커밋 메시지 컨벤션이 중요한가?

이유	설명
👀 이력 가독성 향상	팀원들이 변경 이력을 빠르게 파악 가능
🧪 자동화 도구 연계	릴리즈 로그 생성, 린트 검사, 배포 자동화에 활용
🚫 코드 리뷰 기준 강화	커밋 메시지로 작업 목적과 범위를 명확히 함
🚀 changelog 자동 생성	semantic-release 같은 도구와 연계 가능

✅ 2. Conventional Commits란?

> **Conventional Commits**는 커밋 메시지에 **명확한 구조와 의미를 부여**하는 표준이다.
> → https://www.conventionalcommits.org/

기본 형식:

```
<type>[optional scope]: <description>

[optional body]

[optional footer]
```

✅ 3. 기본 type 종류

타입	설명
feat	새로운 기능 추가
fix	버그 수정
docs	문서 수정 (코드 변경 없음)
style	코드 스타일 변경 (세미콜론, 들여쓰기 등)
refactor	리팩토링 (기능 변경 없이 코드 구조 개선)
test	테스트 코드 추가/수정
chore	빌드 설정, 패키지 매니저 등 기타
perf	성능 개선
ci	CI 설정 및 관련 코드 변경
build	빌드 시스템 혹은 외부 의존성 변경
revert	이전 커밋 되돌리기

✅ 4. 예시 메시지

```
feat: 로그인 폼에 비밀번호 확인 필드 추가

fix: 로그인 시 서버 오류 코드 처리 로직 수정

docs: README에 배포 방법 추가

refactor: BoardList 컴포넌트를 함수형으로 변경

style: 들여쓰기 오류 수정

test: Auth 모듈 단위 테스트 추가

chore: ESLint 설정 수정
```

✅ 5. scope (선택)

> 변경 대상이 특정 모듈/컴포넌트인 경우 괄호로 범위를 명시할 수 있다

```
feat(login): remember-me 기능 추가

fix(api): 응답값에서 null 처리 추가

refactor(user): 유저 정보 저장 로직 개선
```

✅ 6. Breaking Change 표현

```
feat!: API 응답 포맷 변경 (기존과 호환되지 않음)
```

또는

```
feat(account): 비밀번호 암호화 알고리즘 변경

BREAKING CHANGE: 기존 SHA1 → bcrypt로 변경됨. 사용자 재등록 필요.
```

✅ 7. 커밋 메시지 작성 실전 팁

요소	좋은 예	나쁜 예
명확한 type	`fix: 로그인 오류 수정`	`commit`
구체적 description	`feat: 결제 완료 후 이메일 발송`	`작업함`
영어 vs 한글	일관되게만 하면 됨 (대부분 영어 권장)	섞이면 안 좋음
시제	일반적으로 **명령형** 사용 (`Add`, `Fix`)	과거형 (`Added`, `Fixed`) 지양

✅ 8. 자동화와 연계

도구	설명
semantic-release	커밋 메시지 기반으로 버전 태깅 & 배포 자동화
commitlint	커밋 메시지 형식을 검사 (CI 연동 가능)
husky	Git hook을 활용해 커밋 전에 메시지 검사
standard-version	changelog 자동 생성 + 버전업

```
# 예: commitlint + husky 설치
npm install --save-dev @commitlint/{config-conventional,cli} husky

# 설정
echo "module.exports = { extends: ['@commitlint/config-conventional'] };" > commitlint.config.js
npx husky install
npx husky add .husky/commit-msg "npx --no-install commitlint --edit $1"
```

✅ 마무리 요약

구성요소	예시	의미
type	`feat`, `fix`, `docs`, `test` 등	변경 목적 분류
scope	`feat(auth):`	변경 대상 범위
description	`로그인 로직 수정`	구체적 설명
body	선택	상세 내용 기술
footer	선택	Breaking changes, 이슈 참조 등

예시 커밋 로그:

```
feat(login): remember-me 기능 추가

- 쿠키 저장 방식으로 remember-me 구현
- 백엔드와 연동 완료

Closes #124
```

상태 확인

> Git은 항상 "**어떤 파일이, 어떤 상태인지**"를 명확히 구분하고 추적한다.
> 그래서 Git을 잘 쓰려면 **변경 상태를 확인하는 명령어**에 익숙해져야 한다.

- `git status`, `git diff`

✅ 1. `git status` – 현재 작업 상태 확인

```
git status
```

🔎 설명

- 워킹 디렉토리와 스테이징 영역의 **변경 상태를 보여주는 명령어**
- 무엇이 변경됐고, 커밋 준비가 되었는지를 표시함

💉 예시 출력

```
On branch main
Your branch is up to date with 'origin/main'.

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    modified:   index.html

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
    modified:   app.js

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

📘 해석

항목	의미
Changes to be committed	`git add`로 스테이징된 파일 – 커밋할 준비 완료
Changes not staged	수정했지만 `add`하지 않은 파일
Untracked files	Git이 아직 모르는 새 파일 (추적 시작하려면 `git add`)

✅ 2. `git diff` – 변경된 내용 확인

🔹 워킹 디렉토리 vs 스테이징 영역 비교

```
git diff
```

- 현재 파일에서 **수정은 했지만 add 안한 내용**을 보여줌
- 아직 커밋에 포함되지 않음

🔹 스테이징 vs 마지막 커밋 비교

```
git diff --cached
# 또는
git diff --staged
```

- `git add`로 스테이징된 내용이 **어떻게 변경되었는지** 보여줌

✅ 3. `git log` – 커밋 이력 확인

```
git log
```

🔹 기본 출력 예시

```
commit 87c02a52b5878df8dfc4...
Author: o jeongseok <oh@example.com>
Date:   Fri May 31 14:30:01 2025 +0900

    feat: 로그인 페이지 스타일 개선
```

🔹 한 줄 요약

```
git log --oneline
```

```
87c02a5 feat: 로그인 페이지 스타일 개선
65ab3de fix: 로그인 오류 메시지 표시
```

🔹 브랜치 그래프 형태

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

> 실무에서 **브랜치 구조를 시각적으로 파악**할 때 매우 유용해

✅ 4. `git show` – 특정 커밋 상세 보기

```
git show <커밋 해시>
```

- 해당 커밋의 **diff 내용**, 작성자, 메시지 등을 보여줌

✅ 5. `git blame` – 누가 언제 어떤 줄을 수정했는지 추적

```
git blame <파일명>
```

🔹 예시

```
^7b9f83 (ohjeongseok 2025-05-29 14:05:01 +0900  1) import React from 'react';
a1f308b (minjae      2025-05-30 11:03:20 +0900  2) const Login = () => {
```

- 줄 단위로 **최종 수정한 사람**, **날짜**, **커밋 ID**를 확인할 수 있음
- 디버깅, 책임 추적 시 매우 유용

✅ 6. `git reflog` – 브랜치, HEAD 이동 히스토리 확인

```
git reflog
```

- **reset, rebase, checkout 등 HEAD 이동 기록**까지 추적 가능
- 실수로 커밋 날려도 여기서 되살릴 수 있어

```
e1c2a1b HEAD@{0}: reset: moving to HEAD^
c4fa891 HEAD@{1}: commit: fix: 로그인 에러 해결
```

✅ 7. 전체 상태 점검 순서 예시

```
git status          # 어떤 파일이 바뀌었는지?
git diff            # 내용이 어떻게 바뀌었는지?
git add file.js     # 커밋할 파일만 선택
git diff --cached   # 커밋될 변경 내용 보기
git commit -m "feat: 기능 추가"
git log             # 커밋 이력 보기
```

✅ 마무리 요약표

명령어	설명
`git status`	현재 변경 상태 확인
`git diff`	수정했지만 add 안 한 변경 보기
`git diff --cached`	add 한 변경 내용 보기
`git log`	커밋 이력 보기
`git show`	특정 커밋 상세 보기
`git blame`	누가 어느 줄을 수정했는지 보기
`git reflog`	HEAD 이동 히스토리 복구용

로그 확인

> Git의 **commit history를 조회하고 분석**하는 가장 기본이자 강력한 명령어

✅ 1. 기본 사용법

```
git log
```

🔹 출력 예시:

```
commit a1f308b998ad90cb0f55bcdd47f3f...
Author: o jeongseok <oh@example.com>
Date:   Fri May 31 18:20:51 2025 +0900

    feat: 로그인 API 연동
```

✅ 2. 로그 출력 형식 옵션들

🔸 `--oneline`

```
git log --oneline
```

- 커밋 해시 + 커밋 메시지를 한 줄로 출력
- 브랜치 병합 여부 등 빠르게 파악 가능

```
87c02a5 feat: 로그인 스타일 수정
65ab3de fix: 로그인 오류 해결
```

🔸 `--graph` + `--oneline` + `--all`

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

- **브랜치 구조 시각화** 및 커밋 트리 보기
- `--decorate`: 브랜치 이름, HEAD, 태그 등을 커밋 옆에 표시

예시:

```
* 87c02a5 (HEAD -> main, origin/main) feat: 로그인 스타일 수정
|  * 223a7f9 (feature/login) feat: 로그인 상태 저장
|/
* 65ab3de fix: 로그인 오류 해결
```

🔸 `--pretty=format:` – 커스텀 출력 형식

```
git log --pretty=format:"%h - %an, %ar : %s"
```

- `%h` : 짧은 커밋 해시
- `%an` : 작성자 이름
- `%ar` : 상대 시간
- `%s` : 커밋 메시지

결과 예시:

```
87c02a5 - o jeongseok, 3 hours ago : feat: 로그인 스타일 수정
```

✅ 3. 필터링 옵션

🔹 파일 기준 로그

```
git log <파일명>
```

예:

```
git log src/LoginForm.js
```

- 해당 파일에 대한 변경 히스토리만 출력

🔹 커밋 개수 제한

```
git log -n 5
```

- 최근 5개 커밋만 보기

🔹 날짜 범위로 보기

```
git log --since="2 days ago"
git log --after="2025-05-01" --before="2025-05-10"
```

🔹 특정 작성자만 보기

```
git log --author="o jeongseok"
```

🔹 메시지 키워드 검색

```
git log --grep="로그인"
```

✅ 4. 머지 커밋 제외/포함

```
# 머지 커밋 제외
git log --no-merges

# 머지 커밋만 보기
git log --merges
```

✅ 5. `HEAD^`, `HEAD~` 등으로 특정 커밋 타겟팅

```
git show HEAD
git show HEAD^      # 바로 이전 커밋
git show HEAD~2     # 현재에서 두 단계 전
```

✅ 6. 태그/브랜치 이름과 함께 보기

```
git log v1.0.0       # 해당 태그부터의 히스토리
git log develop..main  # develop과 main 사이 차이 비교
```

✅ 7. 기타 유용한 팁

명령어	설명
`git show <커밋 해시>`	해당 커밋의 상세 내용 보기 (diff 포함)
`git log -p`	각 커밋마다 diff까지 같이 출력
`git log --stat`	각 커밋에서 변경된 파일과 줄 수 통계
`gitk`	GUI 기반 로그 뷰어 (설치 필요)
`tig`	터미널 기반 Git log 뷰어 (강력 추천)

✅ 8. 실무 예시

브랜치 구조 보기

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

특정 파일의 변경 이력 보기

```
git log -p README.md
```

최근 1주일 이내의 feat 커밋만 보기

```
git log --since="1 week ago" --grep="^feat"
```

✅ 마무리 요약표

명령어	설명
`git log`	기본 로그 보기
`--oneline`	간결하게 한 줄씩 보기
`--graph`	브랜치 구조 시각화
`--pretty=format:`	출력 형식 커스터마이징
`--author`, `--grep`	작성자, 메시지 필터링
`--since`, `--until`	시간 기반 필터링
`-p`, `--stat`	diff 또는 통계 포함 보기

- `git log`, `git log --graph`, `git show`

✅ 1. `git log` – 커밋 히스토리 기본 조회

🔹 명령어

```
git log
```

🔹 결과 예시

```
commit a1f308b998ad90cb0f55bcdd47f3f...
Author: o jeongseok <oh@example.com>
Date:   Fri May 31 20:14:13 2025 +0900

    feat: 로그인 폼에 remember-me 추가
```

🔹 옵션 요약

옵션	설명
`--oneline`	커밋 해시와 메시지를 한 줄로 요약
`-n <숫자>`	최근 N개 커밋만 출력
`--author="이름"`	특정 작성자 커밋만
`--since="2 weeks ago"`	날짜 기준 필터
`--grep="문자열"`	메시지 내 문자열 검색
`-p`	각 커밋의 diff도 함께 보기
`--stat`	파일 변경 요약 보기

✅ 2. `git log --graph` – 브랜치 구조 시각화

🔹 명령어

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

🔹 설명

옵션	의미
`--graph`	커밋 간 관계(병합, 분기 등)를 **ASCII 그래프**로 시각화
`--oneline`	간결하게 한 줄 출력
`--decorate`	HEAD, 브랜치, 태그 등 참조를 표시
`--all`	현재 브랜치 외의 브랜치까지 모두 표시

🔹 예시 출력

```
* 87c02a5 (HEAD -> main) feat: 로그인 스타일 수정
|  * 223a7f9 (feature/login) feat: 로그인 상태 저장
|/
* 65ab3de fix: 로그인 오류 해결
* 34c2a71 init: 프로젝트 생성
```

> 이 구조를 보면 **브랜치가 언제 갈라졌고, 어디서 머지되었는지**가 한눈에 보임
> 대규모 협업이나 릴리즈 시 브랜치 동선 확인에 아주 유용함

✅ 3. `git show` – 특정 커밋 상세 조회

🔹 명령어

```
git show <커밋 해시>
```

> 커밋 해시 없이 쓰면 **HEAD (가장 최근 커밋)** 을 보여줌

```
git show
```

🔹 출력 내용

- 커밋 메타 정보 (해시, 작성자, 날짜)
- 커밋 메시지
- 변경된 파일 목록
- diff 내용

🔹 예시

```
commit 87c02a567...
Author: o jeongseok <oh@example.com>
Date:   Fri May 31 18:52:14 2025 +0900

    fix: 로그인 상태 저장 로직 수정

diff --git a/src/login.js b/src/login.js
index e69de29..d95f3ad 100644
--- a/src/login.js
+++ b/src/login.js
@@ -0,0 +1,5 @@
+function login() {
+  localStorage.setItem("token", "abc123");
+}
```

✅ 실무에서 세 명령어의 사용 시점

상황	사용 명령어	목적
전체 히스토리 확인	`git log`	작업 순서 추적
브랜치 흐름 파악	`git log --graph --oneline --all`	브랜치 병합 구조 분석
특정 커밋 자세히 보기	`git show <해시>`	해당 커밋의 diff, 메시지 확인
최근 커밋만 간단히 보기	`git log -n 5 --oneline`	요약된 최근 기록 확인

✅ 실무 예시 흐름

```
git log --oneline              # 최근 작업 요약 보기
git log --graph --all          # 브랜치 구조 이해
git show 65ab3de               # 특정 커밋에서 무슨 변경이 있었는지
```

✅ 마무리 요약

명령어	기능	비고
`git log`	커밋 히스토리 상세 조회	날짜/작성자/메시지/해시 포함
`git log --graph`	커밋 트리 시각화	브랜치 병합 구조 분석
`git show`	커밋 1건 상세 diff 보기	메시지, 변경사항, 작성자 모두 포함