GITHUB 사용 설명서

GIT 설치 및 최초 설정
GIT을 명령줄(CMD)에서 사용하기 위해
https://desktop.github.com/download/에서 Git을 다운로드하여 설치한다.

다운로드 후 CMD에서 git --version을 입력 후 버전 정보가 나오면 성공
(절대 CMD를 관리자 권한으로 열지 말 것.)

#본인의 영문 이름으로 설정
git config --global user.name "Your Name"

GITHUB에 가입한 이메일로 설정
git config --global user.email "your.email@example.com"

프로젝트 시작하기 (두 가지 시나리오)
GITHUB 웹사이트 접속 후 회원 가입을 하고, 오른쪽 상단 +를 눌러 새로운 repository를 생성할 수 있다.

프로젝트를 시작하는 방법은 크게 두 가지이다.

1.깃허브에 이미 있는 프로젝트를 내 컴퓨터로 가져올 때 (Clone)

각자의 GITHUB에서 가져올 프로젝트 페이지로 이동 후 초록색 Code 버튼을 누르고 HTTPS 주소를 복사합니다.

CMD에서 프로젝트를 저장하고 싶은 폴더로 cd [경로]를 사용하여 이동한다.
ex) cd C:₩MyProjects

git clone [복사한 깃허브 주소]를 실행하면 내 컴퓨터의 새 폴더로 복사된다.)
ex) git clone https://github.com/user/project.git

!!주의!!

git clone으로 깃허브(GitHub) 등에서 받아온 프로젝트 폴더 안에서는 절대 git init을 다시 실행하면 안 된다. git clone은 이미 .git 저장소 본체까지 통째로 복사해 오는 명령어이기 때문에, init(초기화)이 이미 완료된 상태이다.

2.내 컴퓨터에서 새 프로젝트를 만들어 깃허브에 올릴 때 (Init)
(GITHUB 웹사이트에서 비어있는 새 repository를 생성한다.)

CMD에서 프로젝트를 저장할 폴더로 cd [경로]를 사용하여 이동한다.

ex) cd C:₩MyProjects₩project

현재 폴더를 Git 저장소로 초기화 (최초로 한 번만 실행)
git init

GITHUB 웹사이트 주소와 내 로컬 폴더를 연결 (origin은 깃허브 주소의 별명이다.)
git remote add origin [복사한 깃허브 주소]

후에 잘 연결됐는지 확인을 위해 git remote -v를 사용하여 연결을 확인한다.

일상적인 작업 흐름 (cd, Add, Commit, Push, Pull)

두 시나리오(Clone 또는 Init) 중 하나로 저장소 설정이 완료되었다면, 이후의 작업 흐름은 동일하다.

변경 내용 업로드 (cd -> Add -> Commit -> Push)

(프로젝트 이동) → 스테이지에 파일 업로드 → 커밋 → 푸쉬

#프로젝트 이동 cd [경로]
ex) cd C:₩MyProjects

#변경된 모든 파일을 선택
git add .

스테이지에 올린 파일들을 확정하고, 어떤 변경인지 메세지를 남긴다.

#변경 내용 저장
git commit -m “변경 내용“

push는 스테이지에 있는 파일들을 커밋 내용과 함께 GITHUB에 업로드한다.
본인 branch에서 맞는 이름을 확인하고 네개 중 하나를 사용하면 된다.
보통은 main으로 설정되어 있다.

#업로드
git push origin main
git push origin master
git push -u origin main
git push -u origin master

Q.git push -u origin main/master 가 뭐가 달라요?

A.-u 옵션은 -set-upstream의 줄임말이며, "현재 로컬 브랜치(main)를 원격 저장소(origin)의 main 브랜치에 연결(추적)하여 다음부터 git push나 git pull만 입력해도 되게끔 설정하는 것"이다.

간혹가다 push전에 pull을 하라는 경우가 발생한다.
둘 중 하나를 branch이름에 맞게 사용하면 된다.

#저장소 내용 불러오기
git pull origin main
git pull origin master

주의: 충돌(Conflict)이 날 수 있다!

git pull을 했는데, 만약 팀원이 수정한 파일/부분과 내가 수정한 파일/부분이 겹치면 **충돌(Conflict)**이 발생할 수 있다.
이것은 오류가 아니라, Git이 "이 부분은 네가 직접 보고 어떤 코드를 남길지 결정해 줘!"라고 알려주는 **정상적인 과정이**다.
충돌을 해결한 뒤 커밋하면 된다.

이 모든 것을 한뒤에 GITHUB에서 F5 즉 새로고침을 하여 커밋 내용과 파일들을 확인한다.

< > Code를 클릭 한 뒤 Codespaces에서 파일들을 수정 가능하다.

기타 오류 해결과 다른 명령어

로컬 커밋 취소하기 (Push 전)
방금 한 커밋 자체를 아예 없던 일로 하고 싶을 때 쓴다.
git commit --amend

[Soft] 커밋만 취소 (파일 변경 내용은 남김)
git reset --soft HEAD~1

[Hard] 커밋과 파일 변경 내용을 전부 취소 (!!주의: 복구 안 됨!!)
git reset --hard HEAD~1

HEAD~1은 마지막 1개 커밋을 의미한다. HEAD~2는 2개를 취소한다.

[Revert] 특정 커밋을 되돌리는 새 커밋을 생성
git revert [취소하고 싶은 커밋 해시]

예: git revert a1b2c3d

파일 상태 되돌리기
[restore --staged] 스테이징된 파일을 다시 내린다.
git restore --staged [파일 이름]

예: git restore --staged config.js

작업 중인 파일 변경사항 버리기
[restore] 수정한 내용을 버리고 마지막 커밋 상태로 돌린다 (!!주의: 복구 안 됨!!)

git restore [파일 이름]

branch 삭제 복구
1. 내 모든 활동 기록을 확인
git reflog

2. 삭제된 브랜치의 마지막 커밋 해시(예: a1b2c3d)를 찾는다.
(로그에 'checkout: moving from [삭제된 브랜치] to...' 기록이 있음)

3. 해당 커밋 해시로 새 브랜치를 생성
git checkout -b [새 브랜치 이름] a1b2c3d

원격 저장소 문제 해결
현재 연결된 원격 주소 확인
git remote -v

원격 주소 변경 (이름은 보통 origin)
git remote set-url origin [새로운 깃허브 주소]

push가 거부될 때(강제 푸쉬)
reset 등으로 로컬 히스토리를 바꾼 뒤 push하면 "rejected" 오류가 난다. 이땐 강제로 밀어 올려야 한다.

!!절대 쓰지 마시오!!
git push --force(팀원의 커밋을 덮어쓸 수 있어 매우 위험하다.
궁금하면 직접 해봐라.)
작성자는 귀찮아서 이 명령어를 사용했다가 갈아 엎었다.

[force-with-lease] 더 안전한 강제 푸시
git push --force-with-lease
--force-with-lease는 혹시 그 사이에 팀원이 다른 커밋을 올렸다면, 내 push를 거부하여 사고를 막아준다.

git commit -am “커밋 내용”
git commit -am은 이미 Git이 추적 중인 파일들을 수정하거나 삭제했을 때, 그 변경 사항들을 한 번에 스테이징(add)하고 커밋(commit)까지 하려는 단축키(shortcut) 같은 명령어이다.
a와 m 두 가지 옵션이 합쳐진 것이다.

-a (--all): Git이 이미 알고 있는(Tracked) 파일 중에서 **수정(Modified)**되거나 **삭제(Deleted)**된 모든 파일을 자동으로 스테이징(git add)한다.

-m (--message): 커밋 메시지를 코드 에디터 없이 바로 인라인으로 작성한다.

git commit -am의 -a 옵션은 새로 생성된 파일(Untracked files)은 절대 추가(add)하지 않는다.

git rebase
git rebase는 **이미 커밋한 내역(히스토리)을 깔끔하게 재정렬**할 때 쓰는 **강력하고 위험한** 명령어이다. 브랜치의 시작점(Base)을 다시(Re) 정렬(배치)한다는 뜻이다.

Q.어떨 때 쓰나요?
A. 내 브랜치를 최신 버전으로 업데이트할 때 (가장 흔한 용도)
main 브랜치에서 내 feature 브랜치를 만들었는데, 그사이에 다른 팀원이 main을 업데이트했을 때 쓴다.
[Merge]: main의 변경 사항을 가져와 합치면, "main과 feature를 합쳤다"는 **'Merge 커밋'이 남아서** 히스토리가 지저분해진다.
[Rebase]: feature 브랜치의 커밋들을 통째로 들어다가, **최신 main 브랜치 끝에** 다시 차곡차곡 쌓는다.

결과적으로 rebase를 쓰면, 불필요한 Merge 커밋 없이 히스토리가 한 줄로 깔끔하게 정리된다. 마치 처음부터 최신 main에서 작업을 시작한 것처럼 보인다.

```
# 1. feature 브랜치로 이동
git checkout feature

# 2. main 브랜치를 기준으로 재배치(rebase) 실행
git rebase main
```

!!절대 주의!! (Rebase의 황금률)
rebase는 기존 커밋을 지우고 **새로운 커밋으로 복제**하는 방식이라 히스토리를 **'다시 쓴다.'**
절대로 main, master처럼 여러 사람이 함께 쓰는 공용 브랜치에서는 rebase를 사용하면 안 된다!
rebase는 **나 혼자 쓰는 로컬 브랜치**를 main에 합치기 전에 깔끔하게 정리할 용도로만 사용해야 한다. 이미 깃허브에 올린 커밋을 rebase하면 팀원 전체의 저장소가 꼬이게 된다.

"다 망한 것 같아요" (최후의 보루)
git reflog
이 명령어는 **당신이 Git에서 했던 거의 모든 행동(커밋, 브랜치 이동, reset 등)을 기록**해 둔다.
실수로 reset --hard를 하거나 브랜치를 날려도, git reflog를 입력하면 과거의 커밋 해시(hash) 값을 찾을 수 있다. 그 해시 값만 있으면 git checkout이나 git reset으로 대부분 복구할 수 있다.