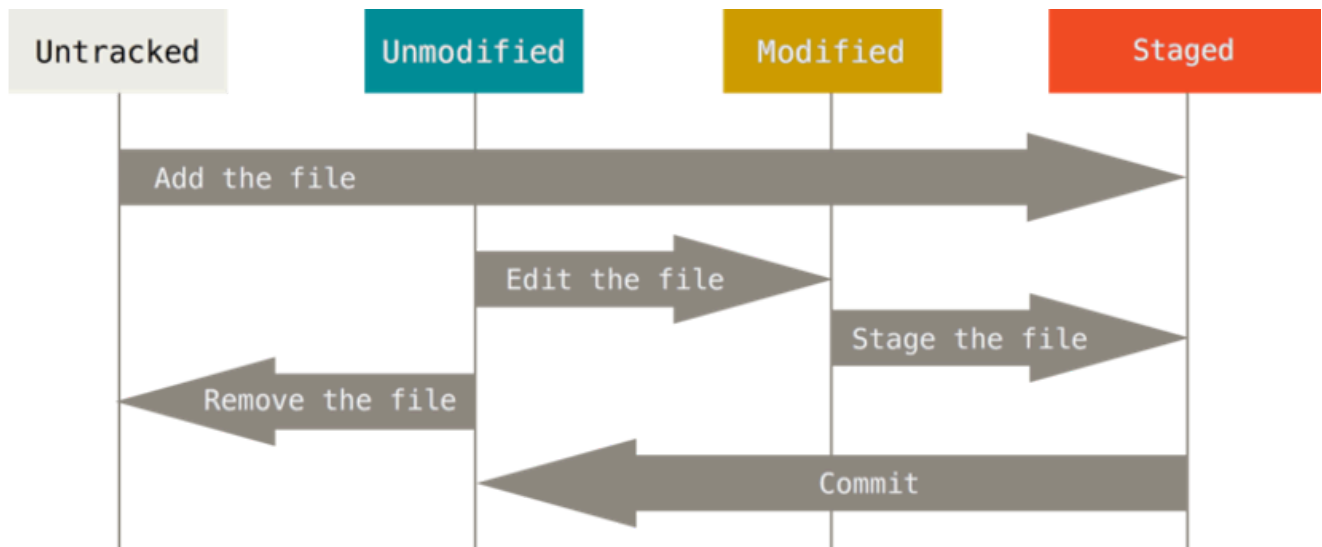


# GIT 튜토리얼

## 1. Git 초기화 (Initialize)

Git을 사용하려면 먼저 저장소(Repository)를 초기화해야 합니다.

### 작업흐름



- Git 의 파일 상태
  - Tacked : Git 이 관리하는 파일
    - UnModified : 수정되지 않는 상태
    - Modified : 수정된 상태
    - Staged: 커밋할 준비가 되어 있는 상태
  - UnTracked : Git이 관리하지 않는 파일

### 명령어:

```
git init          # Git 저장소 초기화
git status        # 현재 저장소 확인
```

### 예제:

```
cd ~/ # home 디렉토리 이동
mkdir tutorial # /home/mhb8436/tutorial
cd tutorial
mkdir my_project # 프로젝트 폴더 생성 # /home/mhb8436/tutorial/my_project
cd my_project # 폴더 이동
echo "# First README.md" >> README.md
cat README.md
```

```
git init          # Git 저장소 초기화
git status        # 저장소 확인
```

이제 `my_project` 폴더가 Git 저장소로 설정되었습니다.

## 2. Git 설정 (Configuration)

Git을 사용하기 전에 사용자 등 기본 정보를 설정해야 합니다.

### Git 설정 파일 종류

- 시스템 전체 설정(--system) : `/etc/gitconfig` 파일
- 사용자 단위 설정(--global) : `~/.gitconfig` 파일
- 개별 저장소 설정(--local) : `.git/conig` 파일
- 설정 우선 순위 : `local > global > system`

### 명령어

```
# 커밋 기록에 남을 사용자 이름과 이메일 설정
# 한번 만 설정하면 모든 커밋에 자동적용
git config --global user.name "사용자이름"
git config --global user.email "사용자이메일"
```

### 실습:

```
git config --list # config 내용 확인
git config --global user.name "홍길동" # 사용자 이름설정
git config --global user.email "hong@example.com" # 사용자 이메일 설정
git config --global core.precomposeunicode true #한글이 깨질경우
git config --global core.quotepath false # 한글이 깨질 경우

git config --list # config 내용 확인
git status # 현재 저장소 상태 확인
```

### 1. `core.precomposeunicode true`

🔪 macOS에서 한글 파일명이 깨질 경우 설정

#### ◆ 문제의 원인

- macOS에서는 파일명을 저장할 때 정준형(NFD, Normalization Form Decomposed) 방식으로 유니코드를 저장합니다.
  - 예: 가 (U+AC00) → ᄀ + ㅏ (U+1100 + U+1161)

- 반면, Linux 및 Windows는 일반적으로 정규형(NFC, Normalization Form Composed) 방식을 사용합니다.
  - 예: 가 (U+AC00) 그대로 저장

Git은 기본적으로 Linux에서 개발되었기 때문에, macOS에서 한글 파일명을 저장하고 다시 가져올 때 깨질 수 있습니다.

### ◆ 해결 방법

```
git config --global core.precomposeunicode true
```

- macOS에서 Git이 파일명을 NFC(정규형) 방식으로 처리하도록 강제하여 한글 파일명이 깨지지 않도록 합니다.

## 2. core.quotepath false

🔗 Git 출력에서 한글 파일명을 원래 그대로 보이게 설정

### ◆ 문제의 원인

- Git은 기본적으로 파일명을 ASCII 문자로만 표시하려고 합니다.
- 한글이나 다른 유니코드 문자가 포함된 파일명을 표시할 때, Git은 이 파일명을 이스케이프된 유니코드 형식( \uxxxx )으로 변환합니다.

### ◆ 예제 (설정 전)

```
$ git ls-files
"\354\225\260\353\205\225\354\236\220.txt"
```

- 위와 같이 한글 파일명이 유니코드 이스케이프 형식으로 변환되어 보입니다.

### ◆ 해결 방법

```
git config --global core.quotepath false
```

- Git이 한글 파일명을 유니코드 이스케이프 없이 원래 문자 그대로 출력하도록 합니다.

### ◆ 예제 (설정 후)

```
$ git ls-files
한글파일.txt
```

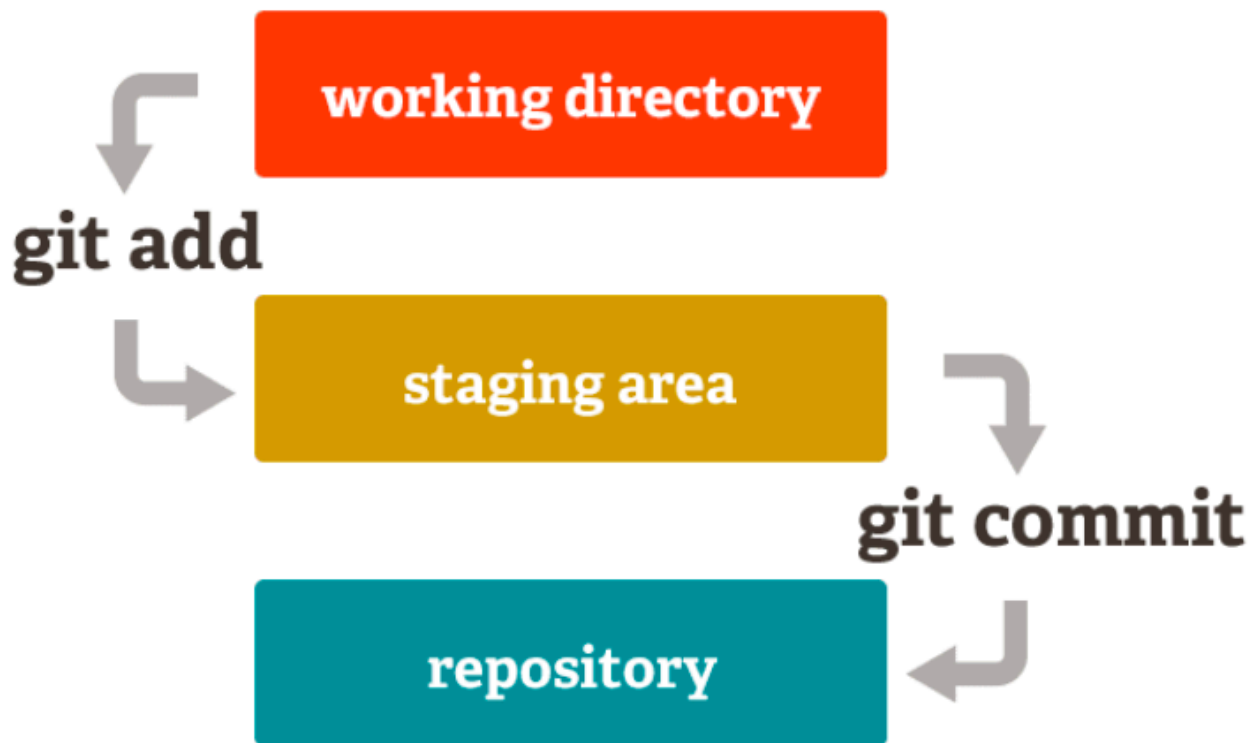
## ✅ 결론

- ✓ `core.precomposeunicode true` → macOS에서 한글 파일명이 깨지지 않도록 설정
- ✓ `core.quotePath false` → Git 명령어 출력에서 한글 파일명을 원래 문자 그대로 표시

이 두 설정을 적용하면 macOS에서 한글 파일명을 다룰 때 발생하는 불편함을 해결할 수 있습니다! 😊

## 3. 파일 추가 (Add)

Git은 변경 사항을 추적하기 위해 파일을 추가해야 합니다.



## 파일 추가 명령어

```
git add 파일명 # 특정 파일만 추가
git add A # 파일 A 추가
git add . # 모든 파일에 추가
```

실습 :

```
echo "Hello, Git!" > hello.txt # 파일 생성
git add hello.txt # 파일을 스테이징 영역에 추가
git status
git add . # 모든 파일 추가
git status
```

---

## 4. 변경 사항 저장 (Commit)

Commit을 하면 변경 사항이 로컬 저장소에 기록됩니다.

### 명령어

```
git commit
git commit -m "commit message" # 메시지 인라인 추가
git commit -a # git add 하지 않고 tracked 상태의 파일을 자동추가
```

### 실습:

```
git commit -m "첫 번째 커밋"
git status
git log
```

### 커밋 로그 확인:

```
git log # 상세한 로그 확인
git log --oneline # 간단하게 한줄로 로그 보기
git log --oneline --graph --all --decorate # 그래프로 확인
git config --global alias.tree "log --oneline --graph --all --decorate"
git tree
```

---

## 5. 되돌리기 및 커밋 간 이동

작업을 되돌리는 방법에는 여러 가지가 있습니다.

### 명령어 :

```
git commit --amend -m "새로운 메시지" # 마지막 커밋 메시지 수정
git checkout -- hello.txt # 변경사항 취소
git reset --soft <커밋 해시> # 커밋취소 + 변경사항 유지
git reset --hard <커밋 해시> # 커밋취소 + 변경사항 삭제
```

```
git revert <커밋 해시> # 특정 커밋 취소
git checkout <커밋 해시> # 특정 커밋 이동
```

## 실습 :

```
git commit --amend -m "새로운 메시지 " # 새로운 메시지 추가
git log --oneline # 변경된 커밋 확인

# git reset --soft 커밋취소 + 변경사항 유지
echo "World" >> world.txt # 신규 파일 추가
git add world.txt
git commit -m "두번째 커밋"
git log
git reset --soft HEAD~1 # 가장 최근 커밋취소 + 변경사항 유지

git add .
git commit -m "세번째 커밋"
git log --oneline

echo "login" >> login.txt
git add login.txt
git commit -m "로그인 커밋"
git log --oneline

# git reset --soft <특정커밋>
git reset --soft ff60afa # 특정 커밋으로 복구
git log --oneline # 확인

git add .
git commit -m "특정커밋테스트완료"
git log --oneline # 확인

# git checkout <커밋>
git checkout ff60afa # 특정 커밋으로 이동
git checkout - # 최신 커밋으로 이동

# git rest --hard 커밋 완전 삭제 + 변경사항 삭제
git reset --hard HEAD~1 # 최근 커밋 완전 삭제 + 변경사항 삭제

# git revert
echo "a" >> a.txt
git add .
git commit -m "a"
echo "b" >> b.txt
git add .
git commit -m "b"
git log --oneline
```

```
git revert 7cb74e7 # a 커밋 번호
```

## 6. 삭제 (Delete)

### 명령어

```
rm <파일명> # 물리적인 파일 삭제  
git rm <파일명> # 물리적인 파일 삭제, 삭제 이력 추가
```

### 실습

```
echo "Hello World" > file.txt  
git add file.txt  
git commit -m "Add file.txt"  
  
git log --oneline # 확인  
  
# 파일 삭제 + 스테이징 영역에 반영  
git rm file.txt  
# 상태 확인  
git status  
git commit -m "Remove file.txt with git rm"  
  
echo "Hello World" > file2.txt  
git add file2.txt # 또는 git rm file.txt  
git commit -m "Remove file2.txt with rm"  
  
rm file2.txt #파일 삭제  
# 수동 스테이징  
git add file2.txt # 또는 git rm file.txt  
git commit -m "Remove file.txt with rm"
```

### git rm vs rm 비교표

구분	git rm	rm
기능	파일 삭제 + 스테이징 자동 반영	파일 삭제만 수행
Git 추적	삭제 이력을 즉시 기록	수동으로 git add 필요
사용 시점	Git으로 관리되는 파일 삭제 시	파일 시스템에서만 삭제할 때
복구 가능성	git reset HEAD file.txt → git checkout file.txt	git checkout file.txt 로 복구 가능

## 7. .gitignore 사용하기

.gitignore 파일을 사용하면 특정 파일이 Git에 추가되지 않도록 설정할 수 있습니다.

실습 :

```
echo "*.log" > .gitignore # 모든 .log 파일을 무시
echo "node_modules/" >> .gitignore # node_modules 폴더 무시
echo "secret-config.env" >> .gitignore # 특정 파일 무시
```

.gitignore 적용 여부 확인:

```
echo "debug.log" >> debug.log
git add debug.log
git commit -m "add debug.log for test"

git add .gitignore
git commit -m "add .gitignore"
git log --oneline
echo "log.txt" >> t.log
git add t.log
```

.gitignore 에 추가된 파일이 Untracked files 에서 사라지는 것을 확인할 수 있습니다.

실수로 추가된 파일을 제거하고 .gitignore 적용하기:

```
git rm --cached <파일명>
```

실습 :

```
git rm --cached debug.log
git ls-files
ls
```

이제 debug.log 는 Git에서 제거되지만, 로컬 파일은 삭제되지 않습니다.

## 8. 브랜치 (Branch) 사용하기

브랜치는 독립적으로 개발 할 수 있도록 코드의 복사본(스냅샷)을 가리키는 포인터

명령어 :

```
git branch <브랜치명> # 브랜치 생성
git checkout <브랜치명> # 브랜치로 이동
```



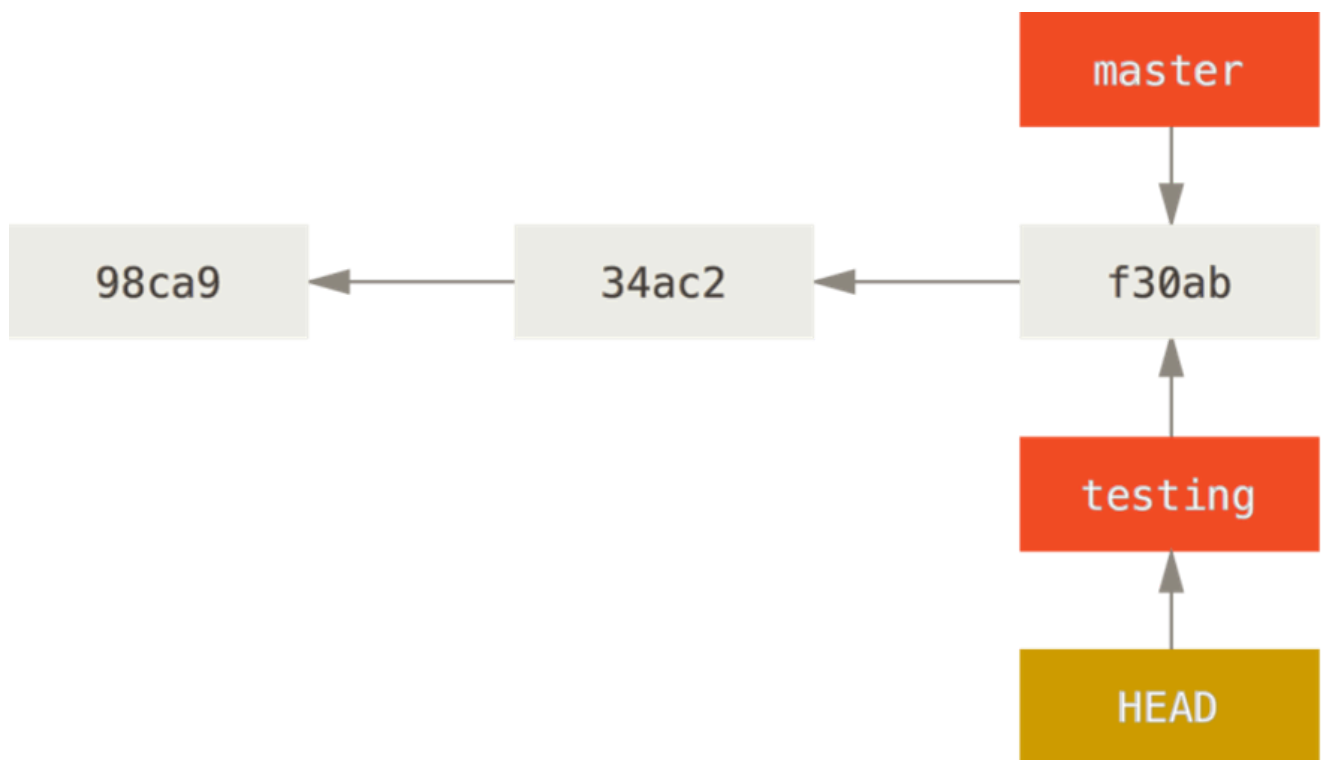
```
git checkout main # 브랜치 main 으로 변경
git merge <브랜치명> # 브랜치 병합
git branch -d <브랜치명> # 브랜치 삭제
```

## 실습:

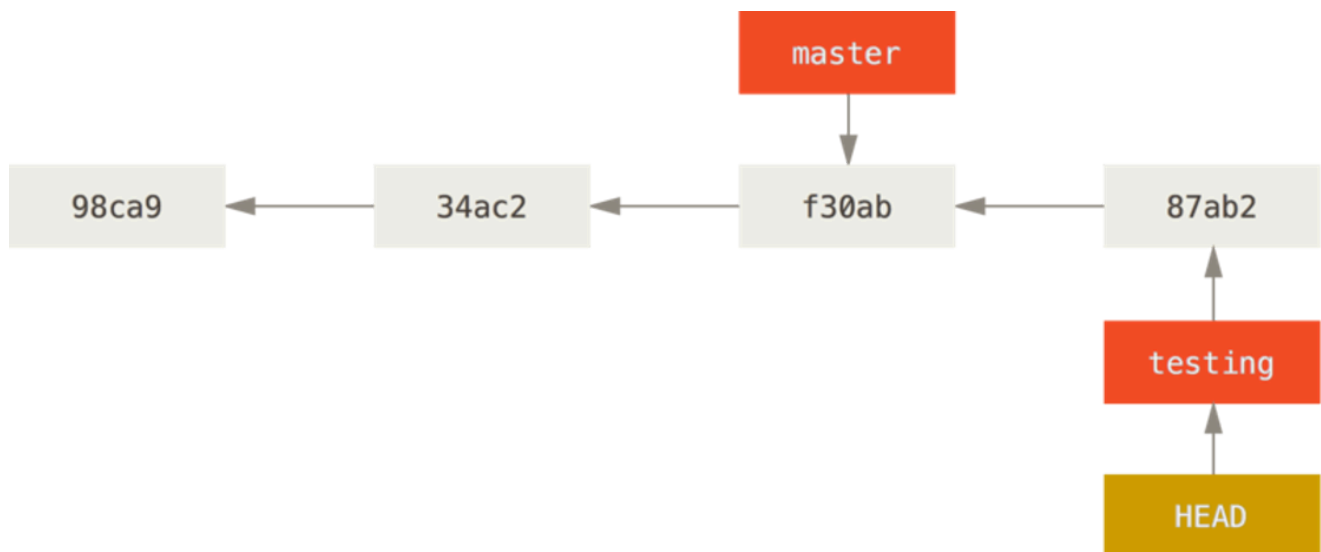
```
cd ~/tutorial
mkdir branch && cd branch
git init
echo "a" > a.txt
git add a.txt && git commit -a -m "a"
echo "b" > b.txt
git add b.txt && git commit -a -m "b"
git log --oneline
git branch testing
git log --oneline
```



```
git switch testing
```



```
echo "testing" >> test.txt
git add . && git commit -a -m "made a change"
git log --oneline
```

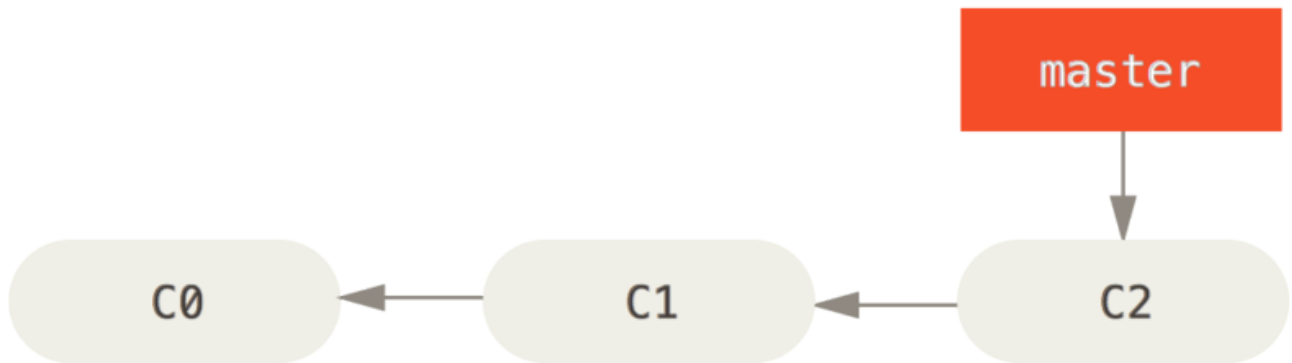


```
git switch main
echo "main" >> main.txt
git commit -a -m "made main change"
git log --oneline --decorate --graph --all
```

## 9. 브랜치와 Merge

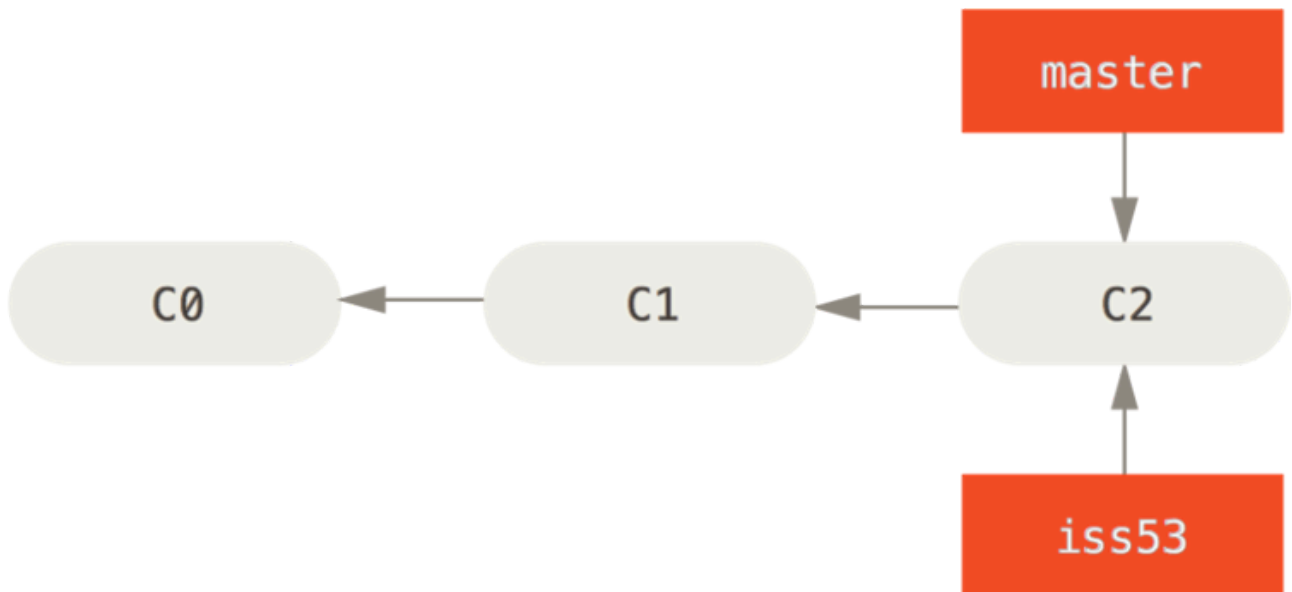
1. 지금 하고 있는 프로젝트의 main 브랜치에 최종 커밋했다고 가정

```
echo "<html/>" >> index.html
git add . && git commit -m "add index.html"
git log --oneline --graph --all --decorate
```



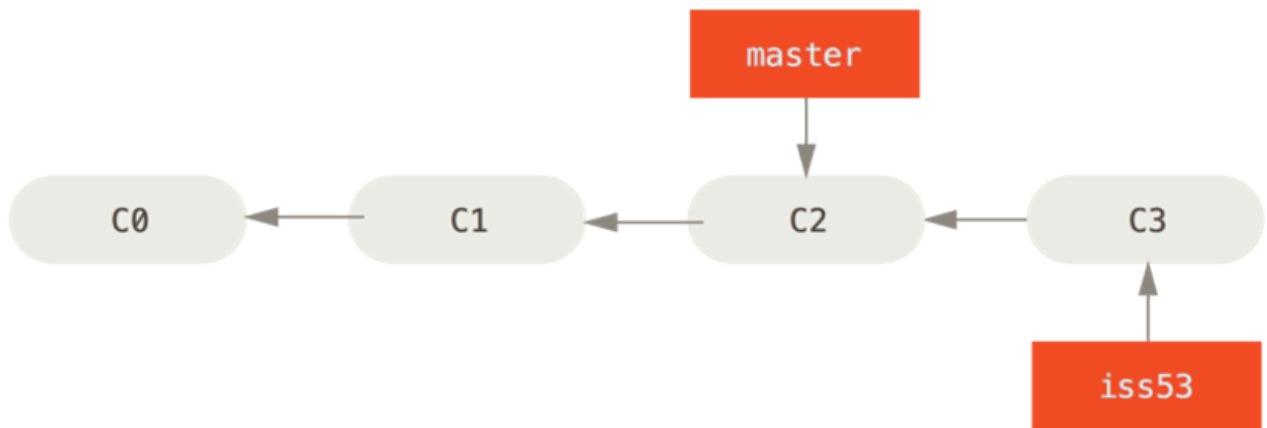
2. 53번 이슈가 발생해서 신규 브랜치 생성

```
git checkout -b iss53
# git branch iss53
# git checkout iss53
```



3. iss53에서 파일 수정

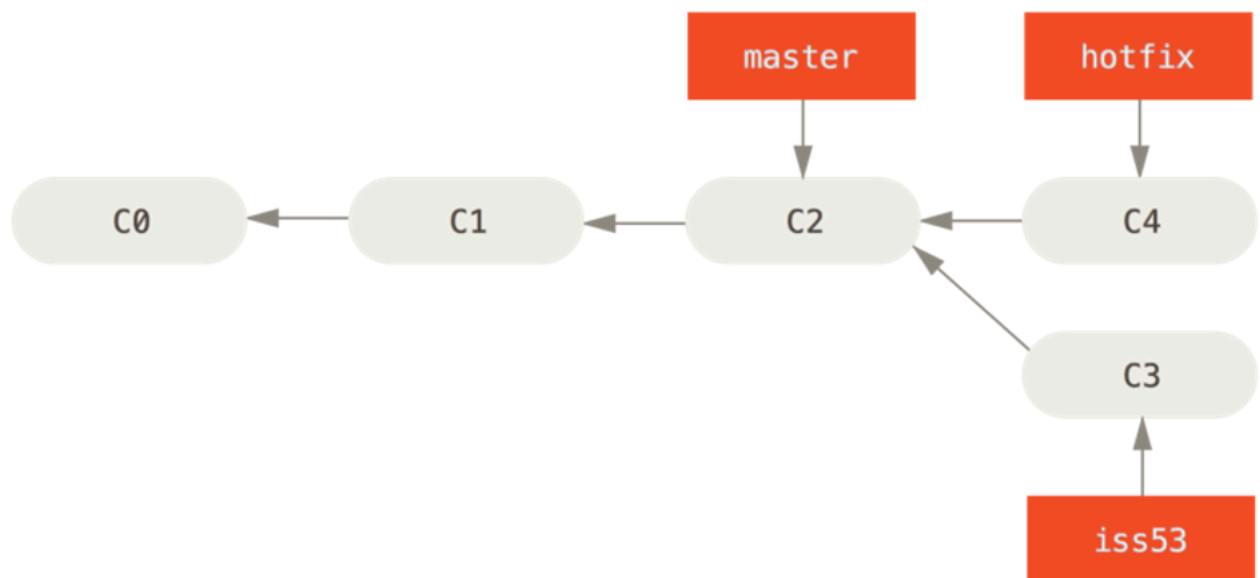
```
echo "<footer/>" >> index.html
git commit -a -m 'added html issue 53'
git log --oneline --graph --all --decorate
```



4. 갑자기 운영 중인 사이트에 문제가 생겨 긴급 수정 필요하다고 가정

```

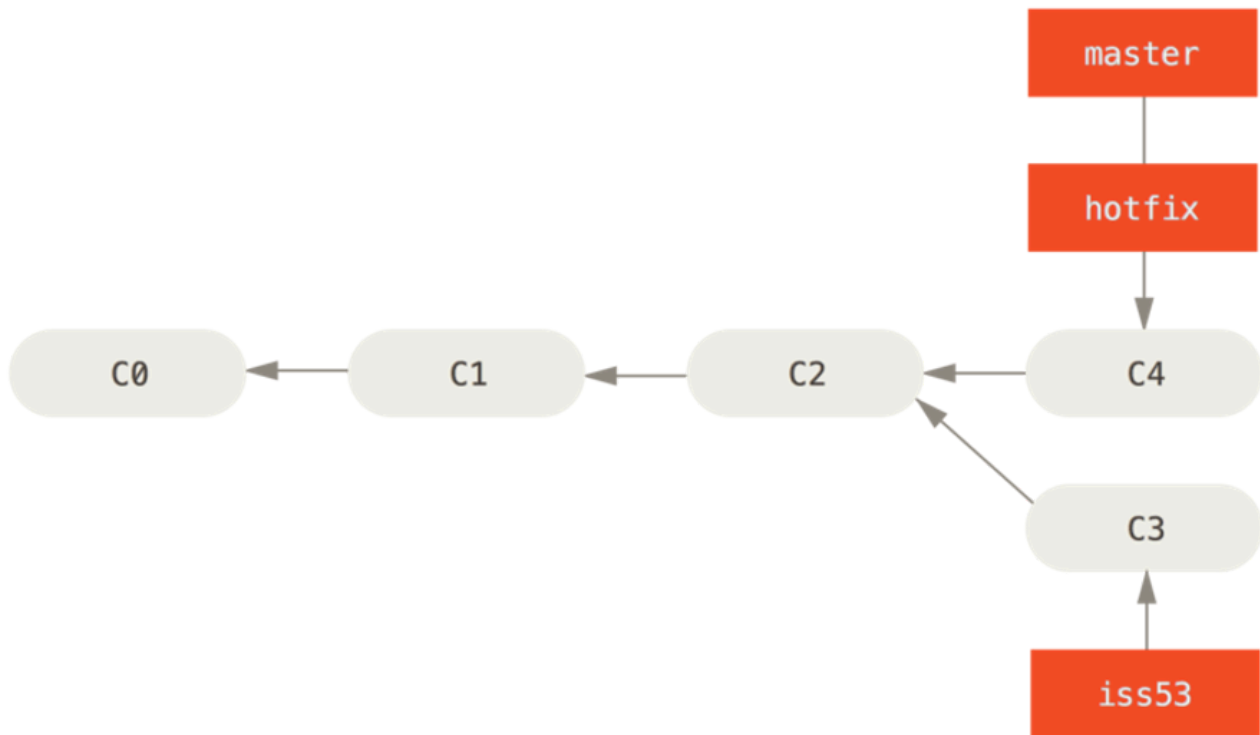
git checkout -b hotfix
echo "<script></script>" >> index.html
git commit -a -m "fixed script"
  
```



5. main 브랜치에 hotfix 적용 후 merge

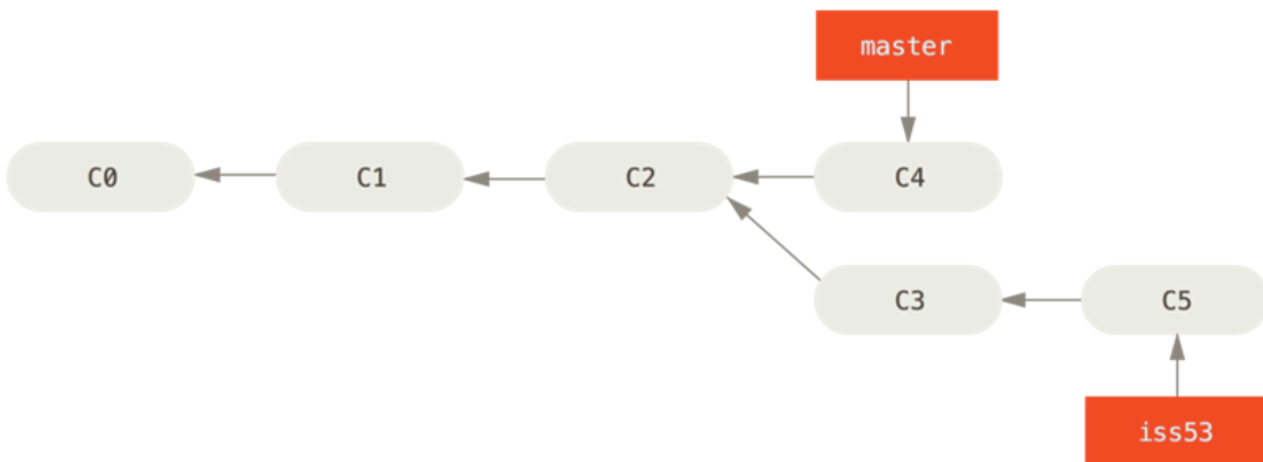
```

git checkout main
git merge hotfix
git log --oneline --graph --all --decorate
git brach -d hotfix
  
```



6. 이슈 53을 처리하던 환경으로 이동

```
git checkout iss53
echo "<a>email</a>" >> index.html
git commit -a -m 'add email to index.html '
```

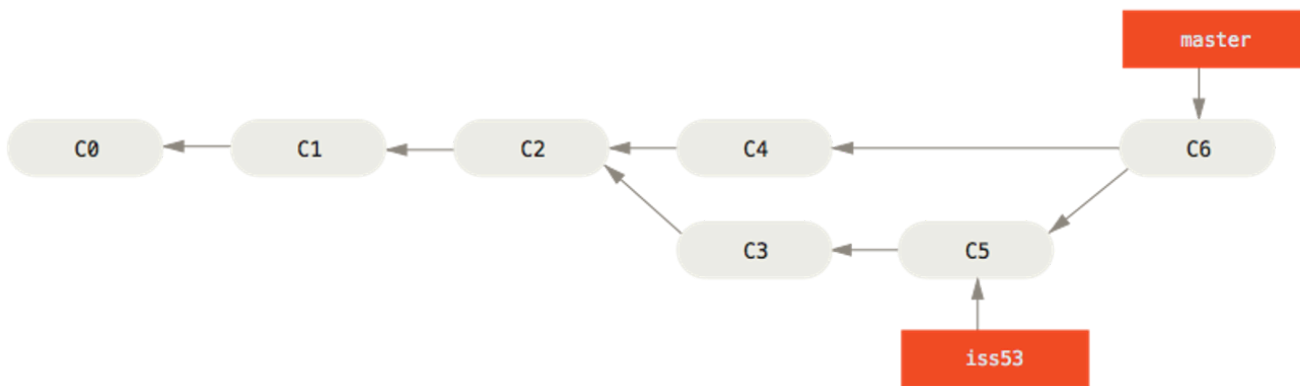


7. 이슈 53을 main 브랜치에 병합

```
git checkout main
git merge iss53
cat index.html
```

```
</html/>
</footer/>
<<<<<<< HEAD
<script/>
=====
<a>email</a>
>>>>>>> iss53
```

```
git add . && git commit -a -m "merge complete"
git log --oneline --graph --all --decorate
git branch -d iss53
```



## 10. Merge 실습

### 1. 저장소 초기화 및 기본 설정

```
# 새 디렉토리 생성 및 이동
cd ~/tutorial
mkdir merge-practice
cd merge-practice

# Git 저장소 초기화
git init

# 기본 브랜치 이름 설정
git config --local init.defaultBranch main
```

## 2. main 브랜치에 커밋 생성

```
# 첫 번째 커밋 (기반이 될 커밋)
echo "# Merge 연습 프로젝트" > README.md
git add README.md
git commit -m "Init: 프로젝트 초기화"

# 두 번째 커밋 (기능 A 기반 작업)
echo "기능 A: 기본 구조" >> feature.txt
git add feature.txt
git commit -m "Feat: 기능 A 시작"
```

## 3. feature 브랜치 생성 및 작업

```
# feature/login 브랜치 생성 및 이동
git checkout -b feature/login

# 기능 개발 커밋 1
echo "로그인 버튼 추가" >> login.txt
git add login.txt
git commit -m "Feat: 로그인 버튼 구현"

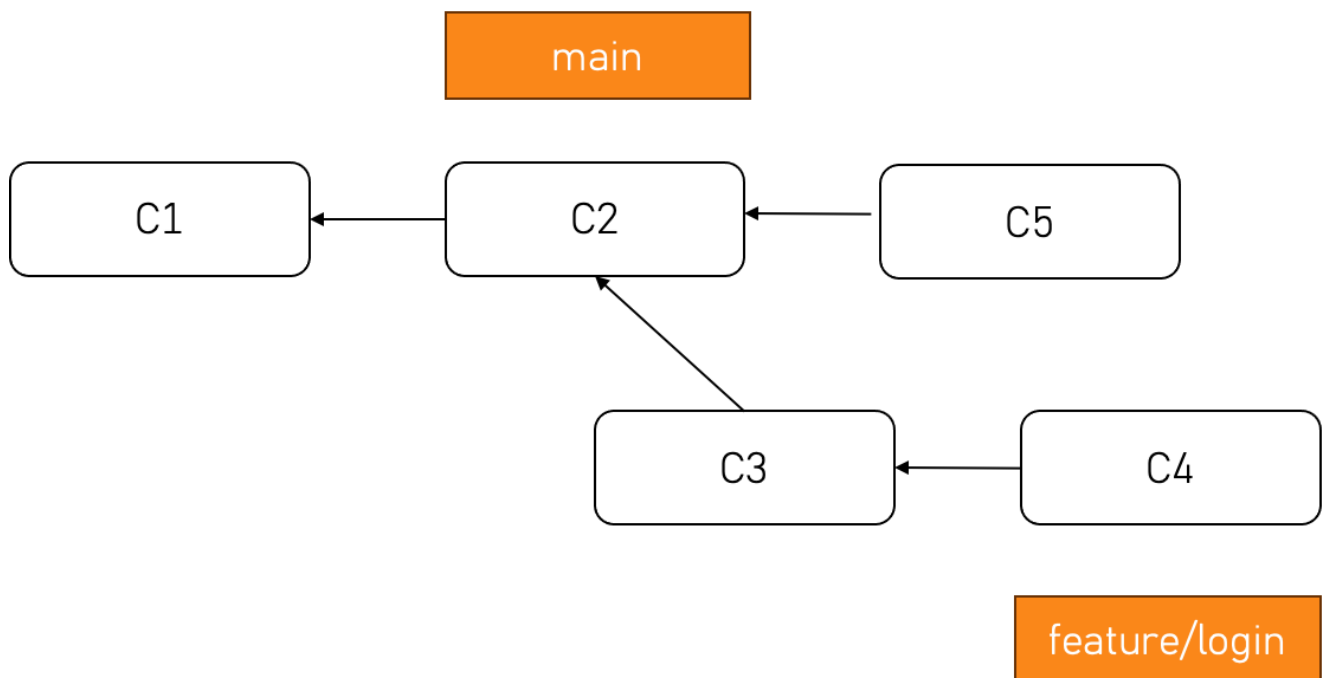
# 기능 개발 커밋 2
echo "소셜 로그인 기능 추가" >> login.txt
git add login.txt
git commit -m "Feat: 소셜 로그인 지원"
```

## 4. main 브랜치에서 추가 작업 (충돌 유발)

```
# main 브랜치로 돌아가기
git checkout main

# main에서 새로운 커밋 생성 (login.txt 수정으로 충돌 유발)
echo "메인 페이지 헤더" >> login.txt
git add login.txt
git commit -m "Design: 메인 페이지 헤더 추가"
```

커밋 모양



## 5. feature 브랜치에서 Merge 실행

```
# main 브랜치로 이동
git checkout main

# feature/login 브랜치 병합
git merge feature/login
```

### ⚠ 충돌 발생 메시지 예시:

```
Auto-merging login.txt
CONFLICT (add/add): Merge conflict in login.txt
```

## 6. 충돌 해결 및 Merge 완료

```
# 충돌 파일 편집 (login.txt)
# <<<<<< HEAD (main의 내용)
# ===== (feature/login의 내용)
# >>>>>> feature/login

# 최종 파일 모습 예시:
메인 페이지 헤더
로그인 버튼 추가
소셜 로그인 기능 추가

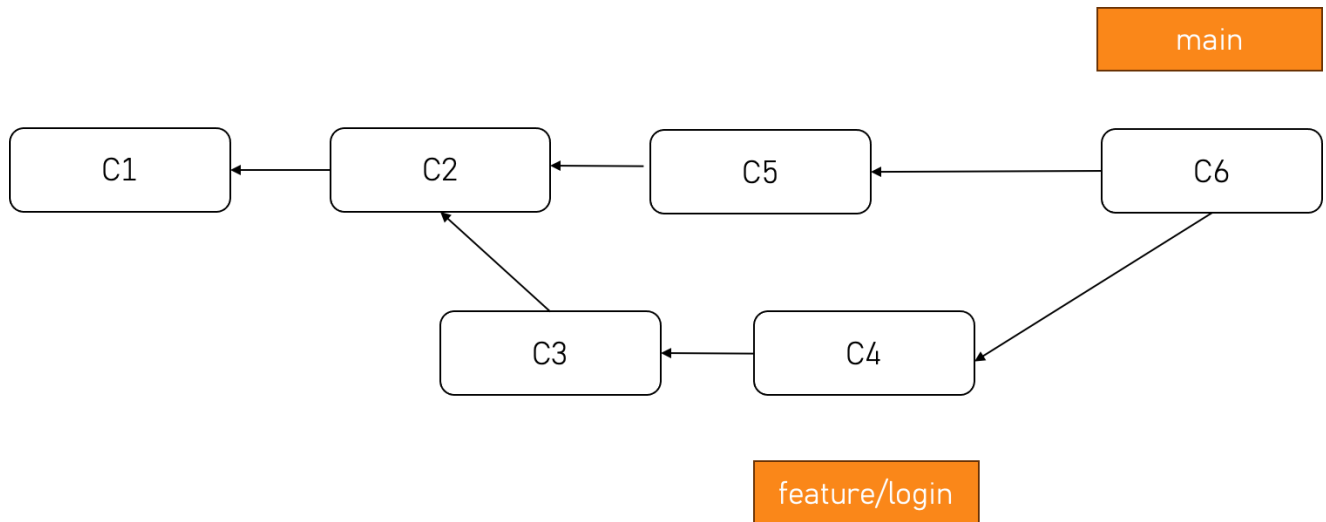
# 충돌 해결 후 스테이징
git add login.txt
```



```
# Merge 커밋 생성
```

```
git commit -m "Merge branch 'feature/login' into main"
```

## 커밋모양



## ✅ 7. 최종 결과 확인

```
# 히스토리 확인 (병합 커밋 생성됨)
```

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

```
# 예상 출력:
```

```
* 8f7d1e6 (HEAD -> main) Merge branch 'feature/login' into main
```

```
| \
```

```
| * 4a3b2c1 (feature/login) Feat: 소셜 로그인 지원
```

```
| * d9e8f7d Feat: 로그인 버튼 구현
```

```
* | 1a2b3c4 Design: 메인 페이지 헤더 추가
```

```
| /
```

```
* 5e6f7a8 Feat: 기능 A 시작
```

```
* 2b3c4d5 Init: 프로젝트 초기화
```

## 🧩 Merge 동작 원리

1. **3-way Merge**: 공통 조상 커밋 + 두 브랜치의 최신 커밋 비교
2. **병합 커밋 생성**: 두 브랜치의 병합 사실을 기록하는 새 커밋 생성
3. **브랜치 구조 유지**: 각 브랜치의 개발 히스토리가 그대로 보존됨

## 💡 Rebase vs Merge 비교

	Rebase	Merge
히스토리	선형 구조	분기된 구조
커밋 해시	변경됨	유지됨
사용 시기	개인 브랜치 정리	공유 브랜치 병합
장점	깔끔한 히스토리	충돌 해결이 한 번만 필요
단점	공유 브랜치 사용 시 위험	히스토리가 복잡해짐

---

## 11. Rebase 실습

1. **Base 변경**: feature/login의 시작점을 main의 최신 커밋으로 이동
2. 커밋 재적용: feature의 커밋을 새로운 base에 차례대로 적용
3. 충돌 처리: main과 feature가 동일 파일 수정 시 직접 해결 필요

---

### 1. 저장소 초기화 및 기본 설정

```
# 새 디렉토리 생성 및 이동
cd ~/
mkdir rebase-practice
cd rebase-practice

# Git 저장소 초기화
git init

# 기본 브랜치 이름을 main으로 설정 (필요시)
git config --local init.defaultBranch main
```

---

### 2. main 브랜치에 커밋 생성

```
# 첫 번째 커밋 (기반이 될 커밋)
echo "# Rebase 연습 프로젝트" > README.md
git add README.md
git commit -m "Init: 프로젝트 초기화"

# 두 번째 커밋 (기능 A 기반 작업)
echo "기능 A: 기본 구조" >> feature.txt
```

```
git add feature.txt
git commit -m "Feat: 기능 A 시작"
```

---

### 3. feature 브랜치 생성 및 작업

```
# feature/login 브랜치 생성 및 이동
git checkout -b feature/login

# 기능 개발 커밋 1
echo "로그인 버튼 추가" >> login.txt
git add login.txt
git commit -m "Feat: 로그인 버튼 구현"

# 기능 개발 커밋 2
echo "소셜 로그인 기능 추가" >> login.txt
git add login.txt
git commit -m "Feat: 소셜 로그인 지원"
```

---

### 4. main 브랜치에서 추가 작업 (충돌 유발)

```
# main 브랜치로 돌아가기
git checkout main

# main에서 새로운 커밋 생성 (login.txt 수정으로 충돌 유발)
echo "메인 페이지 헤더" >> login.txt
git add login.txt
git commit -m "Design: 메인 페이지 헤더 추가"
```

---

### 5. feature 브랜치에서 Rebase 실행

```
# feature/login 브랜치로 이동
git checkout feature/login

# main 브랜치를 기준으로 Rebase 시작
git rebase main
```

 충돌 발생 메시지 예시:

Auto-merging login.txt

CONFLICT (add/add): Merge conflict in login.txt

---

## 🔧 6. 충돌 해결 및 Rebase 계속

```
# 충돌 파일 편집 (login.txt)
# <<<<<<< HEAD (main의 내용)
# ===== (feature/login의 내용)
# >>>>>>> Feat: 소셜 로그인 지원
```

# 최종 파일 모습 예시:

메인 페이지 헤더

로그인 버튼 추가

소셜 로그인 기능 추가

# 충돌 해결 후 스테이징

```
git add login.txt
```

# Rebase 계속 진행

```
git rebase --continue
```

---

## ✅ 7. 최종 결과 확인

# 히스토리 확인 (선형으로 정렬됨)

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

# 예상 출력:

```
* 8f7d1e6 (HEAD -> feature/login) Feat: 소셜 로그인 지원
* 4a3b2c1 Feat: 로그인 버튼 구현
* d9e8f7d (main) Design: 메인 페이지 헤더 추가
* 1a2b3c4 Feat: 기능 A 시작
* 5e6f7a8 Init: 프로젝트 초기화
```

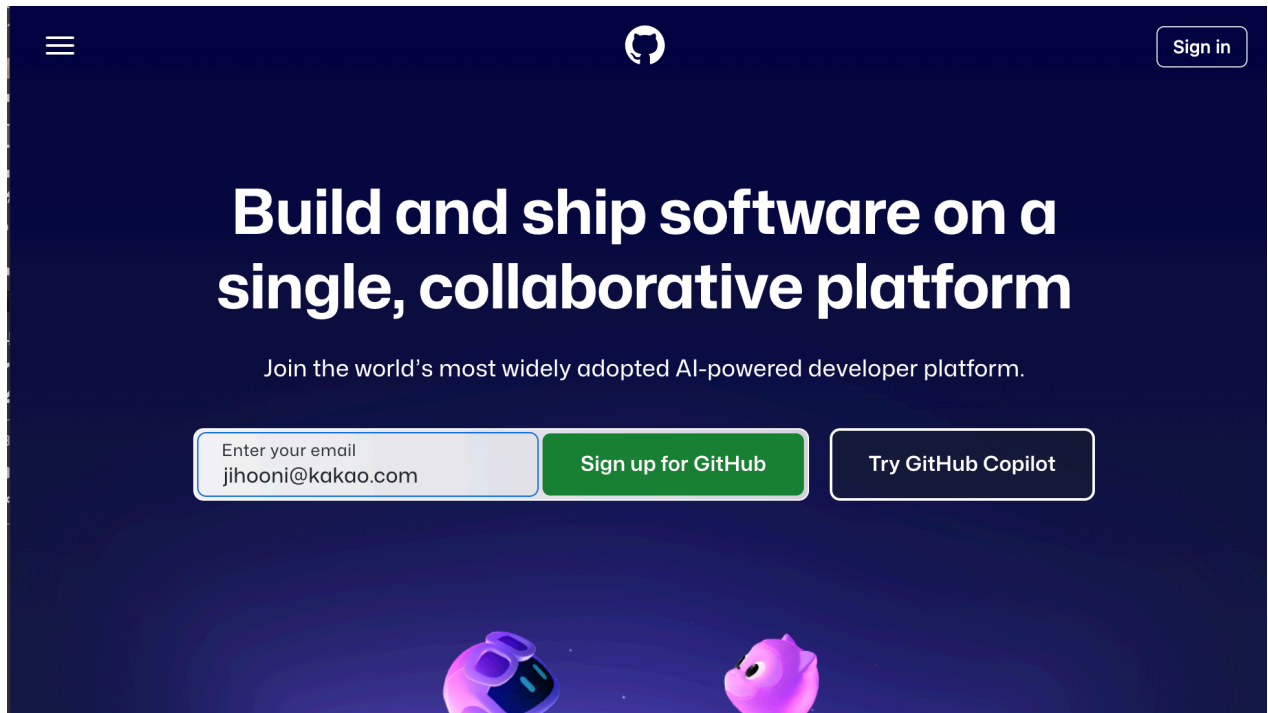
---

## 12. Remote 실습

github 계정이 없으신 분만 해당

### 1. GitHub 접속 및 클론

## 1. 깃헙 사이트 접속




## 2. 이메일, 이름, 비밀번호를 넣고 회원가입

### Create your free account

Explore GitHub's core features for individuals and organizations.

See what's included ▾



### Sign up to GitHub

Email\*

Password\*

Password should be at least 15 characters OR at least 8 characters including a number and a lowercase letter.

Username\*

Username may only contain alphanumeric characters or single hyphens, and cannot begin or end with a hyphen.

[Continue >](#)

By creating an account, you agree to the [Terms of Service](#). For more information about GitHub's privacy practices, see the [GitHub Privacy Statement](#). We'll occasionally send you account-related emails.


### 3. 시각퍼즐 또는 오디오퍼즐을 풀고 회원가입완료

Already have an account? [Sign in](#) →

## Create your free account

Explore GitHub's core features for individuals and organizations.

[See what's included](#) ▾



### Verify your account

다음 중 **고양이** 소리는 무엇인가요?



답변을 숫자로 입력한 다음 Enter 키를 누르거나 아래의 '완료' 버튼을 누르세요.










[▶ 재생](#)

[제출하십시오](#)

[시각 퍼즐](#) [재시작](#)

### 4. 로그인 및 홈화면

 **Dashboard**


 | 

#### Create your first project

Ready to start building? Create a repository for a new idea or bring over an existing repository to keep contributing to it.



[Create repository](#)



[Import repository](#)



[🔗 Branch sync patterns](#) [<> Learn JS closures](#)

## Home

 **Filter** 

 Learn with a tutorial project 

[Introduction to GitHub](#)  
Get started using GitHub in less than an hour.

[GitHub Pages](#)  
Create a site or blog from your GitHub repositories with GitHub Pages.

[Code with Copilot](#)  
Develop with AI-powered code suggestions using

[Hello GitHub Actions](#)  
Create a GitHub Action and use it in a workflow.


## 5. 레포지토리 생성

### Create a new repository

A repository contains all project files, including the revision history. Already have a project repository elsewhere? [Import a repository.](#)

Required fields are marked with an asterisk (\*).

Owner \*

 jihoonlee781

Repository name \*

test

✔ test is available.

Great repository names are short and memorable. Need inspiration? How about [effective-invention](#) ?

Description (optional)

test



Public

Anyone on the internet can see this repository. You choose who can commit.



Private

You choose who can see and commit to this repository.

Initialize this repository with:



Add a README file

This is where you can write a long description for your project. [Learn more about READMEs.](#)

Add .gitignore

## 6. test 레포지토리 이동



test

Public



Pin



Unwatch

1



Fork

0



Star

0

main



Go to file



<> Code

About



jihoonlee781

README.md

README

test

test

Local

Codespaces

Clone

HTTPS

SSH

GitHub CLI

<https://github.com/jihoonlee781/test.git>

Clone using the web URL.

Open with GitHub Desktop

Download ZIP

test

Readme

Activity

0 stars

1 watching

0 forks

Releases

No releases published

[Create a new release](#)

Packages

No packages published

[Publish your first package](#)

## 7. 주소를 복사 후에 클론

```
mkdir mygit
cd mygit
```

```
git clone https://github.com/jihoonlee781/test.git # 원격저장소 클론
cd test
```

## 8. 신규 파일 생성 및 커밋과 푸쉬

```
echo "<html><h1>Welcome My git </h1></html>" >> index.html
git add .
git commit -a -m "first commit"
git push
```

## 9. 푸쉬할 때 에러가 나는 경우

```
echo "url=https://github.com" | git credential reject # 기존 인증정보삭제
```

Settings / Developer Settings

GitHub Apps

OAuth Apps

Personal access tokens ^

Fine-grained tokens (Preview)

Tokens (classic)

### Personal access tokens (classic)

Generate new token ▼

Tokens you have generated that can be used to access the [GitHub API](#).

**mykey** — admin:repo\_hook, repo, workflow Last used within the last week **Delete**

⚠ This token has no expiration date.

Personal access tokens (classic) function like ordinary OAuth access tokens. They can be used instead of a password for Git over HTTPS, or can be used to [authenticate to the API over Basic Authentication](#).

1. github developer 사이트의 Settings > Developer Settings 이동
2. Generate new Token --> classic 선택
3. scope 설정
  1. repo
  2. workflow
  3. admin:repo\_hook
4. 토큰 복사 후 안전한 곳에 저장

## 10. 다시 푸쉬

```
$ git push
Username for 'https://github.com': jihoon.lee@dongobi.com
Password for 'https://jihoon.lee@dongobi.com@github.com':
Enumerating objects: 4, done.
```




```
Counting objects: 100% (4/4), done.
Delta compression using up to 12 threads
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 271 bytes | 271.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
To https://github.com/jihoonlee781/test.git
34b6fa3..57ad026  main -> main
```


## 로컬에 remote 서버 추가


### 1. 새로운 로컬 git 작업 디렉토리 생성


```
mkdir test2
cd test2
git init
echo "# test2" >> README.md
git add .
git commit -m "first commit "
```


### 2. 새로운 레포지토리 생성 (이름: test2)


 **test2** Public

 Pin

 Unwatch **1**

 Fork **0**


 Star **0**



#### Set up GitHub Copilot

Use GitHub's AI pair programmer to autocomplete suggestions as you code.

Get started with GitHub Copilot




#### Add collaborators to this repository

Search for people using their GitHub username or email address.

Invite collaborators

### Quick setup — if you've done this kind of thing before

 Set up in Desktop

 or 

HTTPS

SSH

https://github.com/jihoonlee781/test2.git

Get started by [creating a new file](#) or [uploading an existing file](#). We recommend every repository include a [README](#), [LICENSE](#), and [.gitignore](#).

### ...or create a new repository on the command line

```
echo "# test2" >> README.md
git init
git add README.md
git commit -m "first commit"
git branch -M main
git remote add origin https://github.com/jihoonlee781/test2.git
git push -u origin main
```

### 3. 로컬 git 디렉토리에 remote 서버 추가후에 커밋

```
git branch -M main
git remote add origin https://github.com/jihoonlee781/test2.git
git push -u origin main

Enumerating objects: 3, done.
Counting objects: 100% (3/3), done.
Writing objects: 100% (3/3), 196 bytes | 196.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
To https://github.com/jihoonlee781/test2.git
 * [new branch]      main -> main
branch 'main' set up to track 'origin/main'.
```

## 업스트림 브랜치란?

Git에서 브랜치를 원격과 연결하면, 해당 원격 브랜치를 업스트림 브랜치(upstream branch) 라고 합니다.

업스트림 브랜치를 설정하면 이후 `git push` 와 `git pull` 을 간단하게 실행할 수 있습니다.

## 업스트림 브랜치 설정 예시

```
git push --set-upstream origin main
```

위 명령어를 실행하면, 로컬 `main` 브랜치가 `origin/main` 브랜치와 연결(업스트림 설정) 됩니다.

이후부터는 브랜치를 명시하지 않고도 `git push` 또는 `git pull` 을 실행할 수 있습니다.

```
git push # 원격 main 브랜치로 자동 푸시됨
git pull # 원격 main 브랜치에서 자동으로 가져옴
```

---

## 업스트림 브랜치 확인 방법

현재 브랜치의 업스트림 정보를 확인하려면 다음 명령어를 사용합니다.

```
git branch -vv
```

출력 예시:

```
* main 123abc [origin/main] 작업 중
  develop 456def [origin/develop] 새 기능 추가
```

여기서 `[origin/main]` 이 현재 `main` 브랜치의 업스트림을 의미합니다.

---

# 업스트림 브랜치 변경 방법

기존 업스트림을 다른 원격 브랜치로 변경하고 싶다면 다음과 같이 실행합니다.

```
git branch --set-upstream-to=origin/develop
```

이제 `git push` 와 `git pull` 을 실행하면 `origin/develop` 을 기본적으로 사용합니다.

---

## 업스트림과 다운스트림(Downstream)

- 업스트림(Upstream): 로컬 브랜치가 추적하는 원격 브랜치 ( `origin/main` , `origin/develop` 등 )
- 다운스트림(Downstream): 업스트림 브랜치의 변경 사항을 로컬 브랜치로 가져오는 것 ( `git pull` )

즉, 업스트림은 기준(branch), 다운스트림은 그 기준을 따르는 브랜치라고 이해하면 됩니다.

---

## 정리

개념	설명
업스트림(Upstream) 브랜치	로컬 브랜치가 연결된 원격 브랜치 ( <code>origin/main</code> 등 )
업스트림 설정 명령어	<code>git push --set-upstream origin main</code>
업스트림 확인 명령어	<code>git branch -vv</code>
업스트림 변경 명령어	<code>git branch --set-upstream-to=origin/develop</code>
다운스트림(Downstream)	업스트림 브랜치의 변경 사항을 로컬로 가져오는 것 ( <code>git pull</code> )

## 다른 사람과 Git 브랜치를 생성하여 원격 작업

목표:

1. `main` 브랜치에서 새 `feature-branch` 브랜치를 생성
  2. 원격 저장소( `origin` )로 푸시
  3. 다른 사람이 해당 브랜치를 가져와 작업 후 다시 원격 저장소에 푸시
-

# 1. A 개발자가 브랜치 생성 및 원격 푸시

A 개발자(Alice)가 새로운 기능을 개발하기 위해 feature-branch 를 생성합니다.

```
# 기존 프로젝트 가져오기
git clone https://github.com/example/repo.git
cd repo

# 새 브랜치 생성
git checkout -b feature-branch

# 코드 작업 후 커밋
echo "print('Hello, Git!')" > app.py
git add app.py
git commit -m "Add app.py with Hello Git"

# 원격 저장소에 브랜치 푸시
git push -u origin feature-branch
```

✅ -u (--set-upstream) 옵션을 사용하면 이후 git push 만으로 푸시 가능

---

## 2. B 개발자가 원격 브랜치를 가져와 작업

B 개발자(Bob)가 feature-branch 를 원격 저장소에서 가져와 작업합니다.

```
# 기존 저장소가 없는 경우, 클론 후 이동
git clone https://github.com/example/repo.git
cd repo

# 최신 원격 브랜치 목록 가져오기
git fetch origin

# 원격 브랜치를 로컬로 체크아웃
git checkout -b feature-branch origin/feature-branch
```

✅ git fetch 를 하면 원격 브랜치 목록을 업데이트함

✅ git checkout -b 를 사용하면 feature-branch 를 로컬에서 작업 가능

---

## 3. B 개발자가 코드 수정 후 다시 원격 푸시

B 개발자가 파일을 수정하고 원격 저장소로 다시 푸시합니다.

```
# 코드 수정 후 커밋
echo "print('Feature updated by Bob')" >> app.py
git add app.py
git commit -m "Update app.py with feature by Bob"

# 원격 저장소에 푸시
git push origin feature-branch
```

---

## 4. A 개발자가 B의 작업을 가져옴

A 개발자는 `git pull` 을 실행하여 B의 변경 사항을 자신의 로컬 브랜치에 반영합니다.

```
git checkout feature-branch
git pull origin feature-branch
```

✅ `git pull` 을 실행하면 **B** 개발자의 변경 사항을 가져와 자동 병합

---

## 5. 작업 완료 후 `main` 브랜치에 병합

작업이 완료되면 `feature-branch` 를 `main` 브랜치에 병합하고 원격 저장소에 반영합니다.

```
git checkout main
git pull origin main # 최신 상태로 업데이트
git merge feature-branch # feature-branch 병합
git push origin main # 원격 저장소 업데이트
```

✅ `git merge` 를 실행하여 `feature-branch` 의 변경 사항을 `main` 에 병합

✅ `git push origin main` 을 실행하면 원격 저장소가 업데이트됨

---

## 6. 사용이 끝난 `feature-branch` 삭제

브랜치 작업이 끝나면 로컬과 원격 브랜치를 삭제할 수 있습니다.

### A, B 개발자가 로컬에서 브랜치 삭제

```
git branch -d feature-branch
```

### 원격 브랜치 삭제

```
git push origin --delete feature-branch
```

- ✓ `git branch -d` → 로컬 브랜치 삭제
- ✓ `git push origin --delete` → 원격 브랜치 삭제

---

## 7. 전체 과정 요약

단계	A 개발자(Alice)	B 개발자(Bob)
1	<code>feature-branch</code> 생성 및 푸시	
2		<code>git fetch + git checkout feature-branch</code>
3		코드 수정 후 <code>git commit + git push</code>
4	<code>git pull</code> 로 B의 변경 사항 반영	
5	<code>main</code> 브랜치로 병합 ( <code>git merge feature-branch</code> )	
6	<code>feature-branch</code> 삭제	<code>feature-branch</code> 삭제