

```
# 무슨무슨Project
### 🎉 **협업 고수 caramelpopcorn의 git 모음zip**: **Git 사용법** (협업)
---

## 컴퓨터에서 초기 git 설정
```bash
git config --global user.email "you@example.com"
git config --global user.name "Your Name"
```

## **Git 기본 명령어** (초기 설정 및 커밋)
### 1. **저장소 만들기**
```bash
git init
```
* Git 저장소를 초기화하여 버전 관리를 시작합니다.

```
2. **파일 추가하기**
```bash
git add .
```
* 프로젝트 폴더 내 **모든 파일**을 Git에 추가합니다. (`.`은 모든 파일을 의미)

```
### 3. **상태 확인**
```bash
git status
```
* 현재 파일 상태와 **추적된 파일**을 확인합니다. **초록색**으로 표시된 파일은 커밋 준비가 완료된 파일입니다.

```
4. **커밋하기**
```bash
git commit -m "프로젝트 생성"
```
* 변경 사항을 **커밋**하고, "프로젝트 생성"과 같은 메시지를 작성하여 이 커밋에 대한 설명을 추가합니다.

```
### 5. **원격 저장소 연결하기 (origin)**
```bash
git remote add origin https://github.com/너깃허브닉네임/너깃허브리포지터리.git
```
* 원격 저장소(`origin`)와 로컬 저장소를 연결합니다. GitHub에서 제공한 URL을 입력하세요.

---
```

```
### 6. **원격 저장소 확인하기**  
````bash  
git remote -v
````  
* 원격 저장소 주소가 제대로 연결되었는지 확인합니다.  
---  
### 7. **원격 저장소로 푸시하기**  
````bash  
git push origin master
````  
* 로컬 저장소에서 작업한 내용을 원격 저장소의 `master` 브랜치로 **푸시**합니다.  
---  
### 8. **푸시가 안될 때 해결 방법**  
````bash  
git config --system --unset-all credential.helper
````  
* 인증 관련 문제 발생 시 자격 증명 캐시를 초기화하여 문제를 해결할 수 있습니다.  
---  
---  
## **조원이 할 차례: 파일 받아오기**  
### 1. **작업 공간에 파일 만들기**  
* **JavaSQLProject** 폴더를 작업 공간 내에 생성합니다.  
---  
### 2. **저장소 초기화하기**  
````bash  
git init
````  
* 작업 공간에 **Git 저장소를 초기화**합니다.  
---  
### 3. **원격 저장소 연결 및 파일 받아오기**  
````bash  
git remote add origin https://github.com/너깃허브닉네임/너깃허브리포지터리.git
git pull origin master
````  
* 원격 저장소를 추가하고, **`master` 브랜치**에서 **파일을 받아옵니다**.  
---  
### 4. **상태 확인**  
````bash  
ls
````  
* 파일들이 정상적으로 내려왔는지, **작업 폴더 내 파일 목록**을 확인합니다.
```

```
## **최초 작업 없이 받기**
### 1. **파일 받아오기**
```
bash
git pull origin master
```

* **최초 작업** 없이, 이미 저장소가 설정된 경우 이 명령어로 **파일을 받기만** 하면 됩니다.
```

```
## **정리**
이 단계들을 통해 **로컬 저장소 초기화**, **파일 추가 및 커밋**, **원격 저장소와 연결** 후 **푸시**까지 쉽게 할 수 있습니다. 또한, **다른 사람의 작업을 받을 때**는 `git pull` 명령어를 통해 최신 상태를 가져옵니다.
```

```
## 협업 시 주의 사항**
**AI가 수정한 버전을 새 브랜치에서 관리**하는 건 아주 좋은 판단이에요. 팀원들과 기존 코드도 보존하면서 안전하게 비교/리뷰할 수 있습니다. 아래는 깃 명령어 단계별로 정리한 것입니다:
```

```
## 1. 현재 변경사항 커밋 또는 스테시
우선 수정한 코드가 있다면 저장해야 합니다.
```
bash
git add .
git commit -m "AI 리팩토링: 통계 출력 모듈 분리 및 가독성 개선"
```

> 또는 커밋하고 싶지 않다면:
```
bash
git stash
```

---
```

```
## 2. 새로운 브랜치 생성하고 이동
예를 들어 브랜치 이름을 `ai-refactor`로 만들고 싶다면:
```
bash
git checkout -b ai-refactor
```

> 이 명령은 `ai-refactor`라는 **새 브랜치를 만들고 그 브랜치로 이동**합니다.
```

```
## 3. 원격 브랜치로 푸시
브랜치를 GitHub에 업로드하려면:
```
bash
git push -u origin ai-refactor
```
```

```

> `-u`는 이 브랜치와 원격 브랜치를 연결해 다음부터는 `git push`만 써도 됩니다.

---

## 💡 (선택) `stash`를 했던 경우라면 복구

```bash

```
git stash apply
```

```

---

## 🎯 결과

이제 GitHub에 `ai-refactor` 브랜치가 생기고, 그 안에 AI 리팩토링 코드가 반영됩니다. 기존 `master` 또는 `main` 브랜치와는 안전하게 분리돼 있으므로 `\*\/\*PR(Pull Request)\\*\\*`로 비교하거나 리뷰하기도 편해요.

---

필요하면 GitHub에서 \*\*PR 만드는 방법\*\*이나 `\*\/\*브랜치` 관리 전략(예: feature 브랜치 모델)\\*\\*도 안내해드릴게요.

- \* \*\*커밋 메시지\*\*는 팀원들이 이해할 수 있도록 간결하고 명확하게 작성해야 합니다.
- \* 푸시 전에 `git pull`을 통해 충돌을 방지하고 항상 최신 버전의 파일을 받도록 합니다.

---

좋아, `ai-refactor` 브랜치로 전환해서 `pull`까지 하려면 아래 순서대로 터미널에서 입력하면 돼:

---

### ⓘ 1. 현재 브랜치 확인

```bash

```
git branch
```

```

```bash

```
git branch -a
```

```

현재 어떤 브랜치에 있는지 확인할 수 있어.

---

### ⓘ 2. 로컬에 `ai-refactor` 브랜치가 있는지 확인

```bash

```
git branch --list ai-refactor
```

```

- \* \*\*있다면\*\* 다음 4번으로 바로 가고
- \* \*\*없다면\*\* → 3번으로 이동해서 브랜치를 가져와야 해

---

### ⓘ 3. 리모트에서 `ai-refactor` 브랜치 가져오기 (로컬에 없을 때)

```bash

```
git fetch origin ai-refactor
```

```
git checkout -b ai-refactor origin/ai-refactor
```

```
```
```
#### 4. 브랜치 전환 (이미 로컬에 있을 때)
```bash
git checkout ai-refactor
```
```
5. 해당 브랜치에서 최신 코드 가져오기
```bash
git pull origin ai-refactor
```
```
#### 전체 명령 예시 (브랜치 없을 때부터)
```bash
git fetch origin ai-refactor
git checkout -b ai-refactor origin/ai-refactor
git pull origin ai-refactor
```
```
필요하면 다시 `master`로 돌아가려면:
```bash
git checkout master
```
```
## **Git Reset: 커밋 되돌리기**
### **git reset의 3가지 옵션**
Git reset은 커밋을 되돌리는 강력한 명령어입니다. 옵션에 따라 작업 영역과 스테이징 영역에 미치는 영향이 다릅니다.
```
1. **git reset --soft [커밋 아이디]**
```bash
git reset --soft HEAD~1
```
* **커밋만** 되돌리고, **스테이징 영역과 작업 디렉토리는 그대로** 유지
* 변경된 파일들이 **스테이징 상태**로 남아있음 (git add된 상태)
* **가장 안전한 옵션** - 작업 내용을 잃어버리지 않음
* **언제 사용?** 커밋 메시지를 다시 작성하고 싶을 때
```
### 2. **git reset --mixed [커밋 아이디]** (기본값)
```bash
git reset --mixed HEAD~1
```

```
또는
git reset HEAD~1
```
* **커밋과 스테이징을 되돌리지만**, **작업 디렉토리는 유지***
* 변경된 파일들이 **언스테이징 상태**로 남아있음 (git add 전 상태)
* 파일 내용은 그대로 있지만 다시 `git add`를 해야 함
**언제 사용?** 커밋을 취소하고 파일을 다시 선별해서 커밋하고 싶을 때
```
3. **git reset --hard [커밋 아이디]** △□
```bash
git reset --hard HEAD~1
```
* **커밋, 스테이징, 작업 디렉토리 모두** 되돌림
* **모든 변경사항이 완전히 사라짐** - 복구 불가능!
* 해당 커밋 이후의 모든 작업이 삭제됨
△□주의! 작업한 내용을 완전히 잃어버리므로 신중하게 사용
```
### **커밋 아이디 확인하기**
```bash
git log --oneline
```
* 커밋 히스토리를 한 줄씩 간단하게 확인
* 각 커밋의 **해시값(아이디)**과 **커밋 메시지** 표시
```
상대적 위치로 되돌리기
```bash
git reset --soft HEAD~1    # 바로 이전 커밋으로
git reset --mixed HEAD~2   # 2개 이전 커밋으로
git reset --hard HEAD~3    # 3개 이전 커밋으로
```
* `HEAD~1`: 현재에서 1개 이전 커밋
* `HEAD~2`: 현재에서 2개 이전 커밋
```
### **특정 커밋으로 되돌리기**
```bash
git reset --soft a1b2c3d # 특정 커밋 해시로
```
```
Reset 후 원격 저장소에 강제 푸시 △□
```bash
git push -f origin master
```

△경고! 다른 사람과 협업 중일 때는 절대 사용하지 마세요. 팀원들의 작업에 문제가 생길 수 있습니다.

Git Tag: 버전 관리와 중요한 시점 표시

Git 태그는 특정 커밋에 **버전 번호**나 **중요한 이정표**를 표시하는 기능입니다. 주로 릴리즈 버전을 관리할 때 사용합니다.

1. 태그 생성하기

현재 커밋에 태그 달기

```bash

```
git tag v1.0
git tag version-1.0
```

```

특정 커밋에 태그 달기

```bash

```
git tag version-1 egss5
git tag v2.0 a1b2c3d
```

```

* `egss5`, `a1b2c3d`는 커밋 해시 (아이디)

* `git log --oneline`으로 커밋 해시를 확인할 수 있습니다

주석이 있는 태그 생성

```bash

```
git tag -a v1.0 -m "첫 번째 릴리즈 버전"
git tag -a version-2.0 egss5 -m "주요 기능 업데이트"
```

```

* `-a`: annotated tag (추천)

* `-m`: 태그에 대한 메시지 추가

2. 태그 정보 확인하기

모든 태그 목록 보기

```bash

```
git tag
```

```

특정 패턴의 태그 검색

```bash

```
git tag -l "v1.*"
git tag -l "version-*"
```

```

태그 상세 정보 보기

```
```bash
git show version-1
git show v1.0
```
* 태그가 가리키는 커밋의 상세 정보를 표시
* 커밋 메시지, 변경사항, 작성자 등 정보 포함
---

#### **3. 태그를 원격 저장소에 푸시하기**
#### **특정 태그 푸시**

```bash
git push origin version-1
git push origin v1.0
```
#### **모든 태그 한번에 푸시**

```bash
git push origin --tags
```
---

#### **4. 태그 삭제하기**
#### **로컬 태그 삭제**

```bash
git tag -d version-1
git tag -d v1.0
```
#### **원격 태그 삭제**

```bash
git push origin :refs/tags/version-1
또는
git push origin --delete version-1
```
---

#### **5. 태그로 체크아웃하기**
#### **특정 태그 시점으로 이동**

```bash
git checkout version-1
git checkout v1.0
```
* **주의**: "detached HEAD" 상태가 됩니다
* 이 상태에서 작업하려면 새 브랜치를 만드는 것이 좋습니다
#### **태그를 기준으로 새 브랜치 만들기**

```bash
```

```
git checkout -b hotfix-v1.0 v1.0
```
```
```
### **6. 실무 태그 사용 예시**
```bash
개발 완료 후 버전 태그 생성
git add .
git commit -m "v1.0 기능 완료"
git tag -a v1.0 -m "첫 번째 정식 릴리즈"
원격 저장소에 코드와 태그 함께 푸시
git push origin master
git push origin v1.0
나중에 해당 버전으로 돌아가기
git checkout v1.0
git checkout -b hotfix-v1.0 v1.0 # 수정이 필요한 경우
```
```
```
### **7. 태그 vs 브랜치 차이점**
**구분**	**태그 (Tag)**	**브랜치 (Branch)**
**목적**	특정 시점 표시 (읽기 전용)	지속적인 개발 작업
**이동성**	고정됨 (이동하지 않음)	새 커밋 시 자동 이동
**사용 예시**	v1.0, v2.0 (릴리즈)	feature, develop, master
```
```
## **정리**
태그는 **"이 시점이 중요해!"** 라고 표시하는 북마크 같은 개념입니다. 프로젝트의 **버전 관리**나 **중요한 마일스톤** 표시에 매우 유용하니 적극 활용해보세요!
좋습니다 ✅ 깔끔하게 보기 좋게 정리된 **Git 브랜치 & 머지 가이드**를 만들어드릴게요.
```
```
# 🔍 Git 브랜치 & 머지 가이드
## 1. 브랜치 생성 & 이동
```bash
현재 커밋을 기준으로 새 브랜치 생성
git branch [브랜치이름]
생성된 브랜치로 이동
git checkout [브랜치이름]
브랜치를 만들고 동시에 이동
git checkout -b [브랜치이름]
```
```
2. 브랜치 삭제
```

```
```bash
# 병합된 브랜치만 삭제 (안전)
git branch -d [브랜치이름]
# 강제로 삭제 (병합되지 않아도 삭제됨)
git branch -D [브랜치이름]
```

3. 브랜치 병합
```bash
# 현재 위치한 브랜치에 develop-chat 브랜치 내용을 병합
git merge develop-chat
```

4. 충돌 발생 시 처리
```bash
# 충돌 난 파일을 직접 수정 후
git add .
# 병합 완료 커밋
git commit
```

5. 병합 취소
```bash
# 아직 커밋하지 않은 병합을 취소
git merge --abort
```

6. 실무에서 자주 쓰는 워크플로우 예시
```bash
# 1. 작업용 feature 브랜치 생성
git checkout -b feature/login
# 2. 작업 진행 후 커밋
git add .
git commit -m "feat: 로그인 기능 구현"
# 3. develop 브랜치로 이동
git checkout develop
# 4. feature 브랜치 병합
git merge feature/login
# 5. 필요 없어진 브랜치 삭제
git branch -d feature/login
```
```

---

☞ 이렇게 하면

- \* \*\*작업은 feature 브랜치에서 안전하게\*\*
- \* \*\*최종 코드는 develop/master에 병합\*\*
- \* \*\*병합 충돌 발생 시 수정 → add → commit or abort\*\*