

Git/GitHub 필수

[버전 관리](#)

[Git 훑어보기](#)

[Git을 사용해보자](#)

[Git 저장소 만들기](#)

[Git 파일 수정하기](#)

[내 코드를 올리기 전에 확인을 받아볼까?](#)

[branch?](#)

[Pull Request 생성 방법](#)

[PR 리뷰](#)

[Conflict?](#)

[Git은 어떻게 관리하지?](#)

[Git-flow](#)

[GitHub-flow](#)

버전 관리

Git에 대해 공부하기 전에, 먼저 Git의 역할인 “**버전 관리**”에 대해 알아보자.

버전 관리란 파일들이 변할 때마다 이를 기록해 두었다가 후에 특정 시점의 파일들을 다시 가져올 수 있도록 하는 것이고, 이를 쉽게 하기 위한 도구들을 **버전 관리 시스템(VCS, Version Control System)**이라 부른다. 우리가 배울 Git도 대표적인 버전 관리 시스템 중 하나이다.

버전 관리 시스템도 크게 아래 3가지 종류로 나눌 수 있으며, Git은 이 중 **분산 버전 관리 시스템**에 속한다.

- **로컬 버전 관리 시스템**

별도의 서버 없이 로컬 환경에서 버전을 관리할 수 있는 시스템.

- **중앙집중식 버전 관리 시스템**

버전을 관리하는 별도의 서버가 하나 있으며, 클라이언트는 이 서버로부터 파일을 받아서 사용하는 방식의 시스템. 로컬 버전 관리 시스템에 비해 많은 장점이 있지만, 서버에 문제가 발생할 경우 심각한 경우에는 모든 히스토리가 날아갈 위험까지도 있다는 단점이 있다.

- **분산 버전 관리 시스템**

버전을 관리하는 별도의 서버 외에도 각 클라이언트의 로컬 환경에서도 버전을 관리하는 방식의 시스템이다. 클라이언트는 서버에 저장되어있는 히스토리를 전부 복제할 수 있으며, 이로 인해 서버에 문제가 발생하여도 로컬에 존재하는 복제물로 다시 복구할 수 있다.

Git 훑어보기

Git은 관리하는 파일을 **Modified, Staged, Committed** 이 세 가지 상태로 관리한다.

- **Modified**
수정한 파일을 아직 로컬 데이터베이스에 저장하지 않았음을 의미한다.
- **Staged**
현재 수정한 파일을 곧 저장할 것이라고 표시한 상태를 의미한다.
- **Committed**
데이터가 로컬 데이터베이스에 안전하게 저장되었음을 의미한다.

이 상태들은 Git 프로젝트의 세 가지 단계와 연결되어 있다.

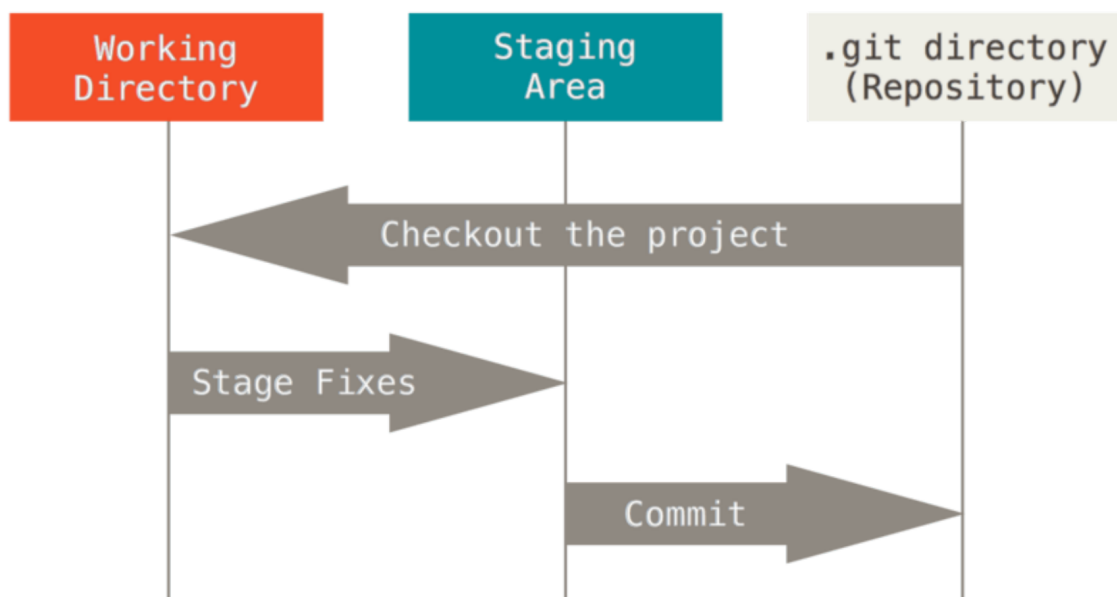


그림 1. Working Directory, Staging Area, Repository

- **Working Directory**
로컬에서 실제로 작업하는 공간이다. 수정된 파일들 중 기록할 파일들을 선택하여 **Staging Area**로 보낸다.
- **Staging Area**

버전을 기록할 파일에 대한 정보를 저장한다. 스냅샷을 만들 때, 이 안에 있는 파일들에 대해 스냅샷을 만든다.

- `.git` 폴더

프로젝트의 메타데이터와 객체 데이터베이스가 저장되는 곳이다.

실제로 Git을 사용해보며 이를 이해해보자.

참고로 Git이 설치되어 있지 않다면 설치하고 오자. 설치 방법까지 알려주지는 않는다.

Git을 사용해보자

Git 저장소 만들기

Git 저장소를 만드는 방법은 두 가지가 있다.

- 버전 관리를 하지 않는 로컬 프로젝트를 Git 저장소로 만드는 방법
(이 뒤로 실습에 사용할 저장소이므로, 하는 걸 추천한다)

Git으로 관리하고 싶은 프로젝트 폴더로 이동한 후 아래 명령을 실행한다.

```
$ git init
```

이 명령은 `.git` 폴더를 생성하며, 처음에는 버전 관리에 필요한 기본적인 것밖에 없어 아직 아무것도 관리하지 않는다.

- 기존 Git 저장소를 clone하는 방법

`clone` 은 서버에 있는 프로젝트의 히스토리를 거의 복사해오며, 명령어는 아래와 같다.

```
$ git clone {clone할 레포지토리 주소}
```

예시로 Learn-Vim 레포지토리를 클론해보자.

우리가 클론하고 싶은 레포지토리 주소는 아래와 같다.

```
https://github.com/gcc-mirror/gcc.git
```

터미널에 아래 명령을 실행하면 해당 레포지토리가 클론된다.

```
$ git clone https://github.com/gcc-mirror/gcc.git
```

Git 파일 수정하기

이제 막 Git 저장소를 생성하였다. 그러니 Git은 아직 관리할 파일이 아무것도 없다.

관리할 파일이 없는 지 어떻게 알 수 있는가? 그건 `status` 명령어로 확인할 수 있다. 아직은 어떠한 커밋도 없기 때문에 아래와 같은 결과가 나올 것이다.

```
$ git status

On branch main

No commits yet

nothing to commit (create/copy files and use "git add" to track)
```

README.md 파일을 만들고 다시 `status` 를 보자.

```
$ echo 'Hello, Git!' > README.md
$ git status

On branch main

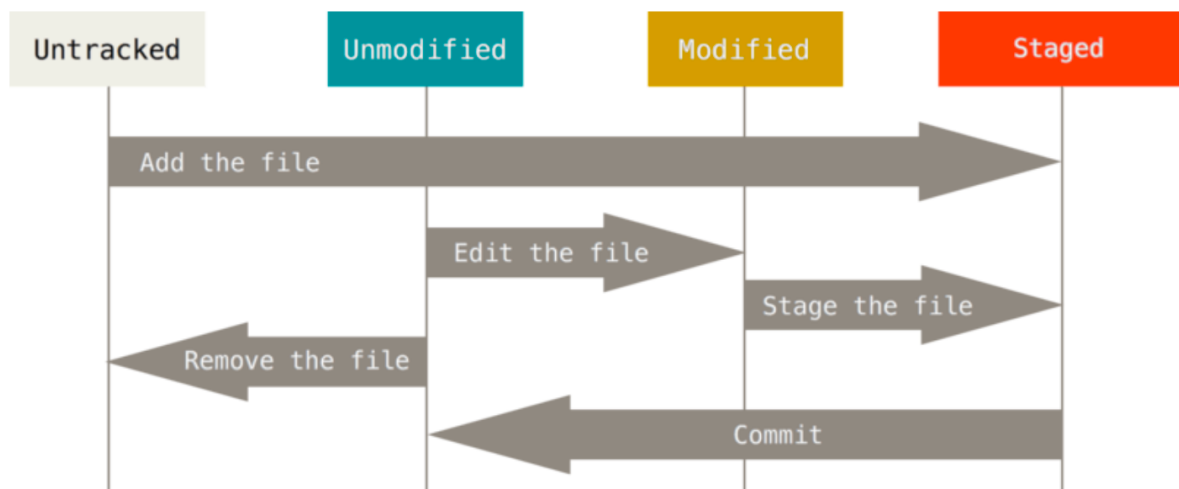
No commits yet

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

nothing added to commit but untracked files present (use "git add" to track)
```

우리가 생성한 **README.md**가 **Untracked files** 아래에 나오기 시작했다. 이게 무슨 소리일까?

Git은 아직 기록하지 않은 파일들을 관리할 때 크게 2가지(**Untracked**, **Tracked**)로 나뉘며, **Tracked**는 또 다시 세 가지(**Unmodified**, **Modified**, **Staged**)로 나뉜다. 그림으로 보면 아래와 같다.



- **Untracked**

스냅샷에도, Staging Area에도 존재하지 않는 파일들을 의미한다. 그렇기에 Git은 이 파일들의 기록을 추적할 수 없기 때문에 **Untracked**라는 이름이 붙었다.

- **Unmodified**

기록된 이후로 파일이 변경되지 않은 상태이다. 변경되지 않았기 때문에 기록할 것이 없으며, 신경쓰지 않아도 된다.

- **Modified**

Unmodified 파일들의 일부가 수정되었지만, 아직 기록할 준비가 되지 않은 상태이다.

- **Staged**

Staging Area에 올라가 생성된, 또는 수정된 파일들을 기록할 준비가 되었다는 뜻이다.

`commit` 명령어를 통해 기록하면, **Staged** 상태의 파일들이 기록된다.

이제 `add` 명령어를 통해 **README.md** 파일을 Staging Area에 올려보자.

```
$ git add README.md
$ git status
```

```
On branch main
```

```
No commits yet
```

```
Changes to be committed:
```

```
(use "git rm --cached <file>..." to unstage)
```

```
new file:   README.md
```

우리는 이제 README 파일을 기록할 준비가 되었다. 기록해보자.

```
$ git commit -m "int: initial commit"
```

```
[main (root-commit) c2089c8] int: initial commit
```

```
1 file changed, 1 insertion(+)
```

```
create mode 100644 README.md
```

```
$ git status
```

```
git status
```

```
On branch main
```

```
nothing to commit, working tree clean
```

기록을 완료하였기에, 이제 다시 Git은 추적할 파일을 잃어버렸다. 실업자가 된 Git을 위해 README.md 파일을 수정해보자.

```
$ echo 'Hello, Modified README!' > README.md
```

```
$ git status
```

```
On branch main
```

```
Changes not staged for commit:
```

```
(use "git add <file>..." to update what will be committed)
```

```
(use "git restore <file>..." to discard changes in working tree)
```

```
modified:   README.md
```

```
no changes added to commit (use "git add" and/or "git commit")
```

파일이 변경되었으며 아직 Staging Area에 올리지 않아 **Modified** 상태로 변경된 것을 볼 수 있다.

위에서처럼 `add`, `commit` 명령어로 기록할 수 있으므로 생략하겠다.

우리는 지금까지 로컬 컴퓨터에 기록했다. 이제 이 기록들을 다른 사람들이 볼 수 있도록 깃헙 레포지토리에 저장해보자. 레포지토리 생성부터 주소 가져오는 것까지는 생략하겠다.

필자는 `git-playground` 라는 레포지토리를 만들어 사용하였다.

```
$ git remote add origin https://github.com/mobius29/git-playg
$ git remote -v

origin  https://github.com/mobius29/git-playground.git (fetch
origin  https://github.com/mobius29/git-playground.git (push)
```

이 레포지토리 주소는 이제 `origin` 이라는 이름으로 저장되었다. 우리가 기록한 것을 저장해 보자.

```
$ git push origin main

Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 8 threads
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 296 bytes | 296.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
To https://github.com/mobius29/git-playground.git
723a502..2f7a7eb  main -> main
```

이제 깃헙 레포지토리로 접속하면 무사히 저장된 코드를 볼 수 있다.

내 코드를 올리기 전에 확인을 받아볼까?

우리는 Git과 GitHub을 **버전 관리**로도 사용하지만, 협업을 위해 많이 사용한다. 그리고 그 협업 과정 중에 가장 많이 쓰이는 기능은 **Pull Request**라고 봐도 무방할 것 같다.

Pull Request는 한 브랜치(**source branch**)에서 다른 브랜치(**target branch**)로 병합하기 전, 한 번 확인 절차를 밟는 것이다. 그 사이에 공동 작업자는 변경 사항을 검토하고 이에 대해 논의할 수 있다. 뿐만 아니라 Pull Request 과정 중에 테스트 등을 실행하여, 코드의 안정성 또한 추가해줄 수 있다.

그러면 이제 이를 어떻게 사용하는 지 알아보자. 그 전에 위에서 Pull Request에 대해 설명할 때, "브랜치"라는 단어가 등장했다. 이 또한 처음보는 용어인데, 이것부터 공부하자.

branch?

브랜치는 **독립적인 작업 흐름을 위한 분리된 작업 공간**으로 설명이 가능하다. 이 또한 협업을 위해 주로 사용되며, 어떤 느낌인지는 사용해보면서 깨달아보도록 하자.

먼저 브랜치를 생성해보자. 직관적이게도 `branch` 라는 명령을 사용하면 되며, 테스트용으로 **test**라는 이름의 브랜치를 생성해보자.

```
$ git branch test
```

어떠한 출력도 나오지 않아 브랜치가 생성이 제대로 되었는지 알 수가 없다. 현재 로컬에 존재하는 브랜치 목록을 나열해보자. 이 또한 `branch` 명령을 사용하면 되지만, 브랜치 생성 때와는 다르게 뒤에 브랜치 명을 쓰지 않는다.

```
$ git branch
```

```
* main
test
```

기존의 **main** 브랜치와 함께 **test** 브랜치가 생성된 것을 볼 수 있다. **main** 옆의 별(*)은 현재 브랜치를 나타내는 표시이며, 아직 현재 **main** 브랜치에 있음을 알 수 있다. **test** 브랜치로 이동하기 위해 `checkout` 또는 `switch` 를 사용한다.

```
$ git checkout test
# git switch test
```

```
Switched to branch 'test'
```



```
$ git branch
```

```
main  
* test
```

test 브랜치로 무사히 이동했다. 참고로 브랜치를 생성함과 동시에 해당 브랜치로 이동하는 방법도 있다.

```
# git branch test | git checkout(switch) test  
$ git checkout -b test  
$ git switch -c test
```

이제 **test.txt** 파일을 생성하는 하나의 커밋을 생성해보자.

```
$ echo 'This is a test file' > test.txt  
$ cat test.txt
```

```
This is a test file
```

```
$ git add test.txt  
$ git commit -m "test.txt"
```

```
[test ea0b891] test.txt  
1 file changed, 1 insertion(+)  
create mode 100644 test.txt
```

이제 다시 **main** 브랜치로 이동 후, **test.txt** 파일을 확인해보자.

```
$ git checkout main
```

```
git checkout main  
Switched to branch 'main'  
Your branch is up to date with 'origin/main'.
```

```
$ cat test.txt
```

```
cat test.txt
cat: test.txt: No such file or directory
```

분명 위에서 **test.txt** 파일을 생성하고, 커밋까지 완료했다. 근데 **main** 브랜치에서 이 파일을 찾을 수 없다고 한다. 어떻게 이런 일이 발생한 것일까?

이는 브랜치에 대해 설명할 때, 가장 처음에 설명했던 **분리된 작업공간**으로 설명 가능하다. 우리는 **test** 브랜치를 만듦으로써 그 순간의 **main** 브랜치와 분리된 공간을 생성하였고, 그렇기에 **test** 브랜치의 수정사항이 **main** 브랜치로는 반영이 되지 않은 것이다. 이 두 브랜치 간의 동기화 작업을 **merge**라 하며, 동기화 전 확인을 받는 과정이 **Pull Request** 이다.

Pull Request 생성 방법

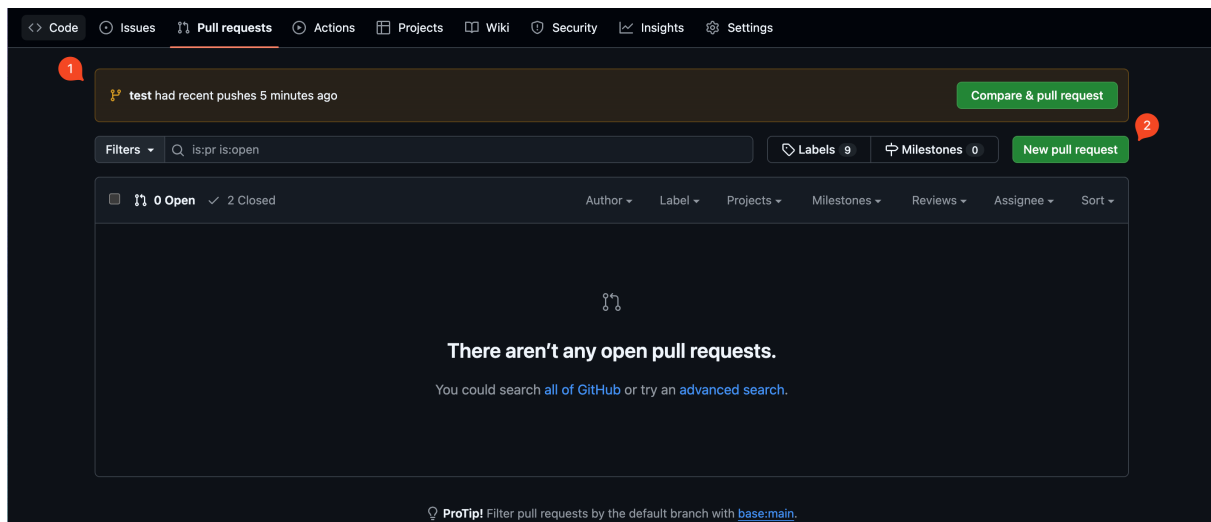
우리는 위에서 **test** 브랜치에 커밋까지만 했으니, 이제 GitHub에 올려보자.

```
$ git push origin test

git push origin test
Enumerating objects: 4, done.
Counting objects: 100% (4/4), done.
Delta compression using up to 8 threads
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 314 bytes | 314.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
remote:
remote: Create a pull request for 'test' on GitHub by visiting
remote:      https://github.com/mobius29/git-playground/pull/
remote:
To https://github.com/mobius29/git-playground.git
 * [new branch]      test -> test
```

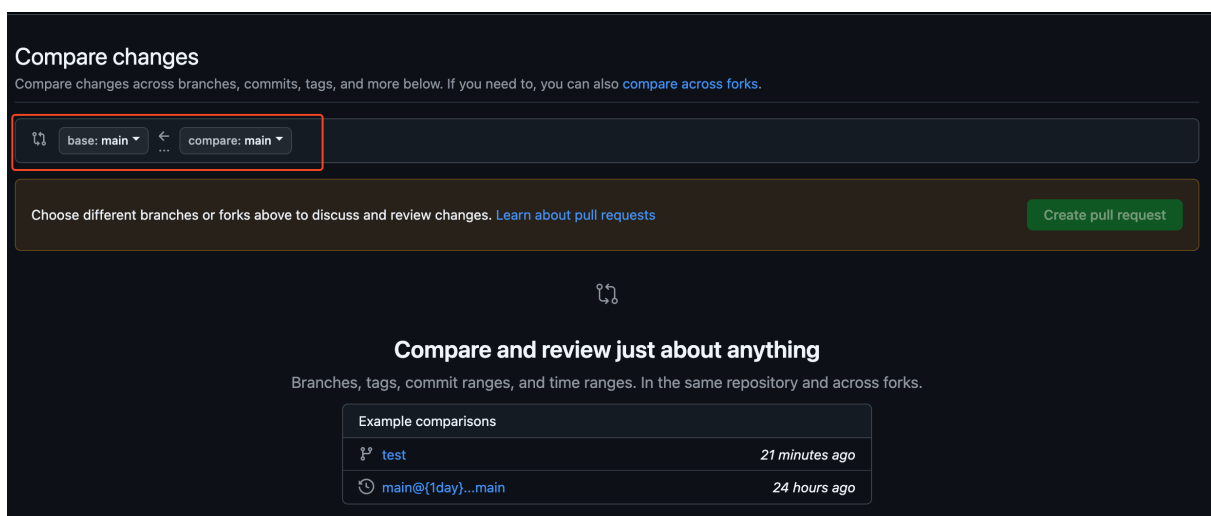
이제 GitHub의 해당 레포지토리에 접속하여 Pull Request를 생성해보자.

해당 레포지토리에 들어가보면, 네이게이션 바에 Pull Request가 보인다.이를 클릭하면 아래와 같은 화면이 나온다.

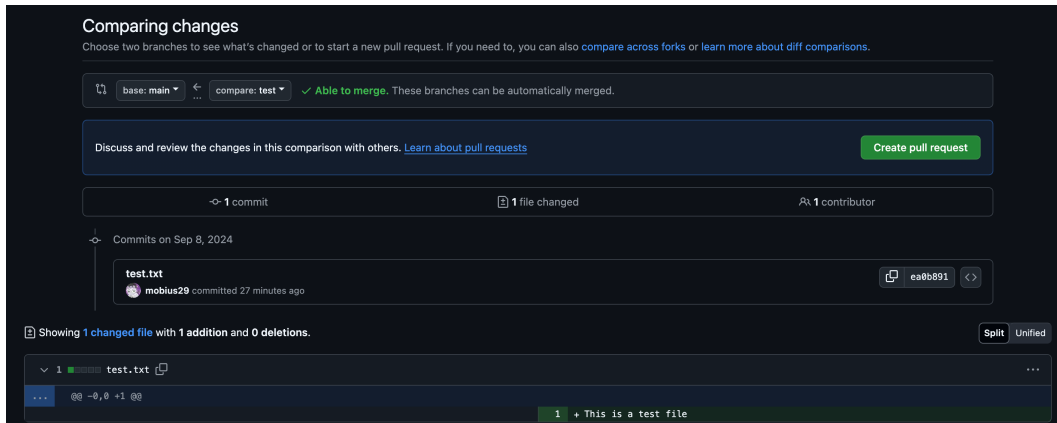


1. 업데이트된 지 얼마 안 된 브랜치들에 대해서 나오는 것으로 Pull Request 생성할 때 설정해야할 하나를 건너뛰어준다. 중요한 기능은 아니다.
2. 우리가 지금 당장 볼 버튼이다. 클릭해보자.

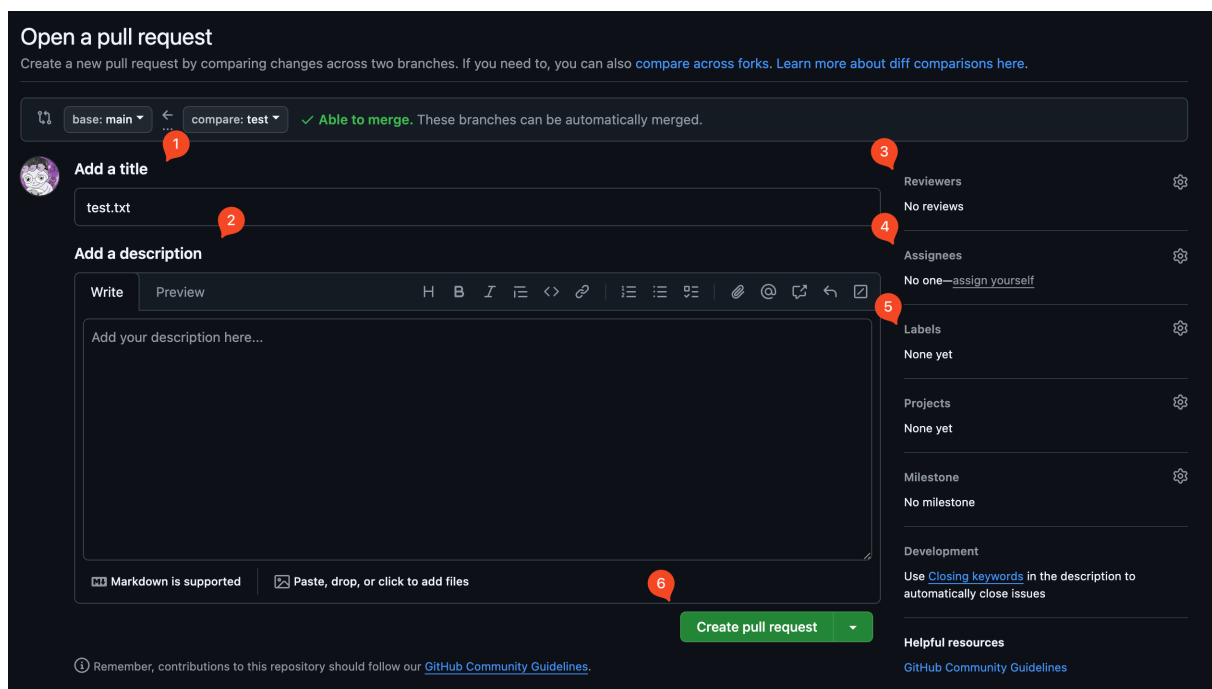
New pull request 버튼을 클릭하면 다음과 같은 화면을 볼 수 있다. 아직 **Create pull request** 버튼이 활성화되어 있지 않다. 이유를 알기 위해 빨간색 네모를 친 부분을 보자.



base는 병합의 목적지를, **compare**는 병합의 시작점을 나타낸다. 즉, compare에서의 커밋들을 base에 똑같이 적용시키는 것이다. 지금은 둘 다 main으로 설정되어 있으며, 이 둘은 당연하게도 바뀐 점이 없으니 Pull Request를 생성할 수 없는 것이다. 우리는 test 브랜치의 변화를 main 브랜치에 적용하고 싶으니 compare를 **test**로 변경시키자.



test 브랜치에서 변경한 내역이 아래에 나오며, 버튼이 활성화되었다. 클릭해보자.



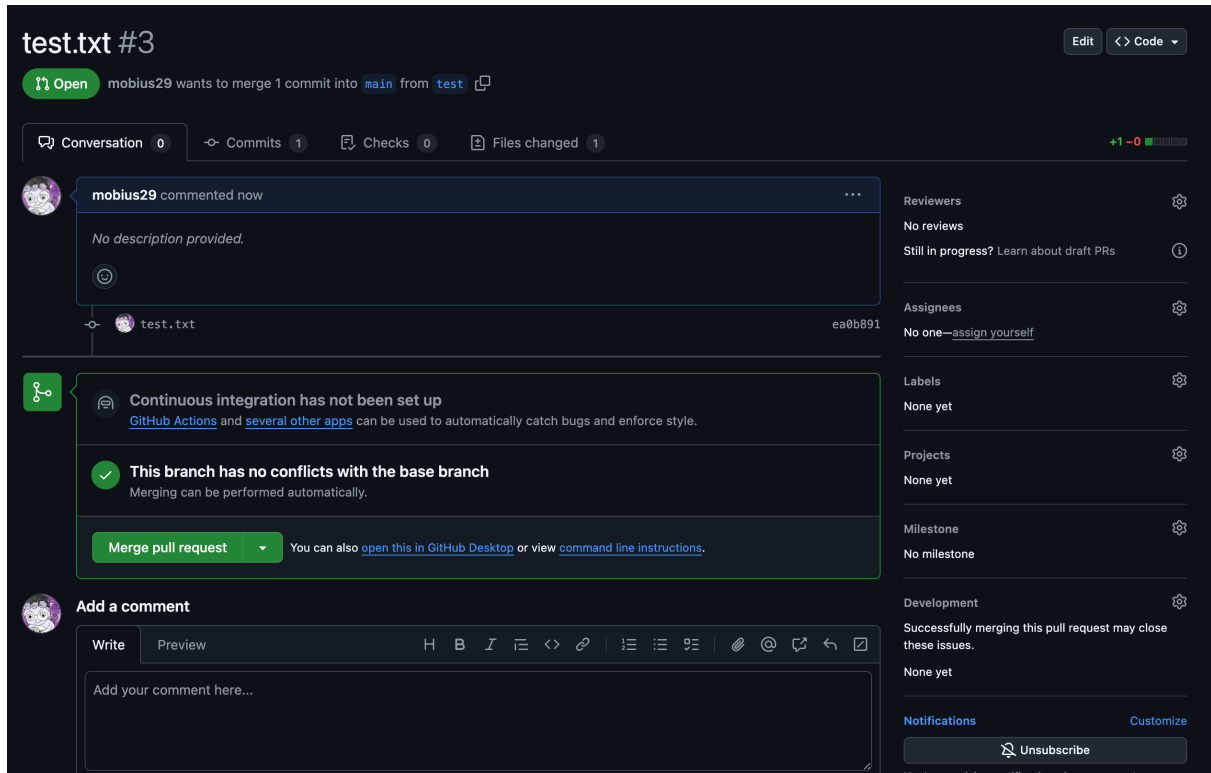
1. Pull Request의 제목을 지정할 수 있다.
2. 부가 설명을 추가할 수 있다.
3. 이 PR을 리뷰해줄 사람을 지정할 수 있다. 이를 지정하고 생성 시, 그 사람들에게 알림이 간다.
4. 이 PR의 담당자를 지정할 수 있다. 보통 브랜치를 생성한 사람이 PR을 생성하기에 자기 자신을 임명하는 편이다.

5. 이 PR이 어떤 동작을 하는 지 등의 레이블을 붙이는 곳이다.

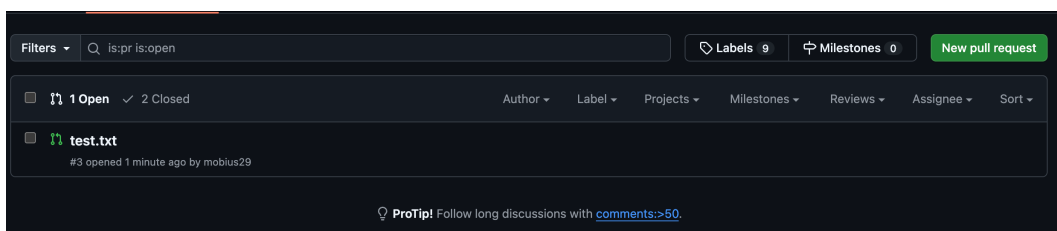
기본적으로 **bug**, **documentation** 등이 존재하고, 원하면 프로젝트 별로 커스텀할 수 있다.

6. Pull Request를 생성한다.

Create pull request를 클릭하여 생성해보자.



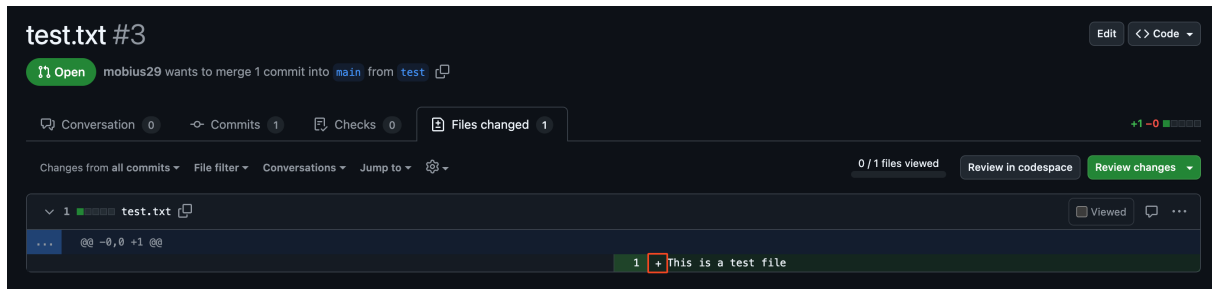
이제 PR이 생성되었다. 이제 Pull Request 목록에서 생성된 PR을 볼 수 있다.



PR 리뷰

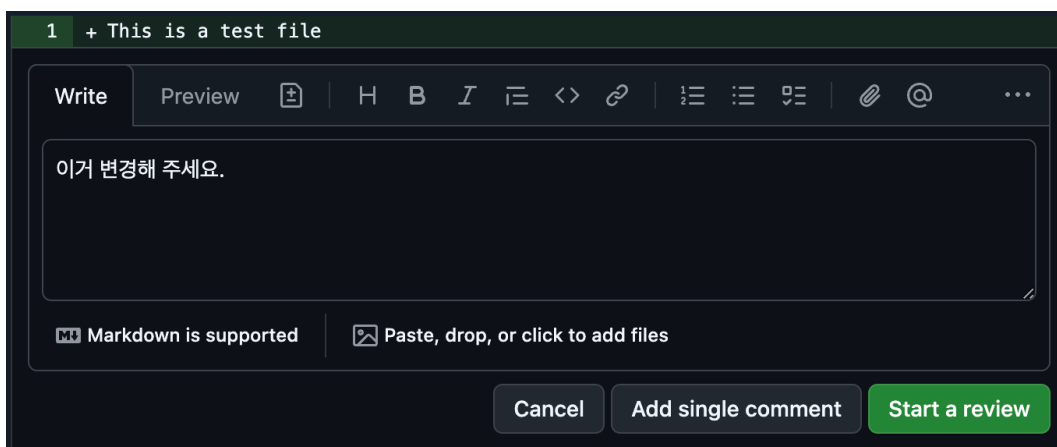
우리는 PR을 생성하기도 하지만, 다른 사람의 PR을 리뷰해주기도 해야 한다. 그렇다면 리뷰하는 방법도 당연히 알아야 한다.

다시 우리가 작성한 PR을 들어가, **File changed**를 클릭해보자. 그러면 아래와 같은 페이지로 이동한다. 이 페이지에서는 이 PR에서의 **source** 브랜치와 **target** 브랜치와의 차이를 보여준다.

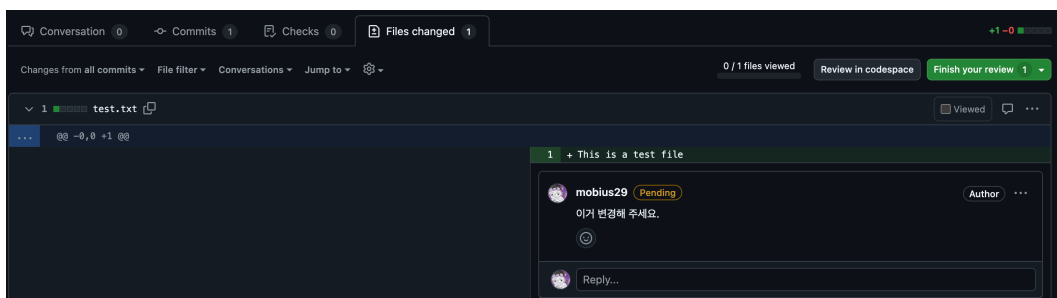


빨간색으로 네모친 부분을 호버하면 클릭할 수 있는 파란색 버튼으로 바뀐다. 클릭하면 아래와 같이 나오고, 코멘트를 작성할 수 있다.

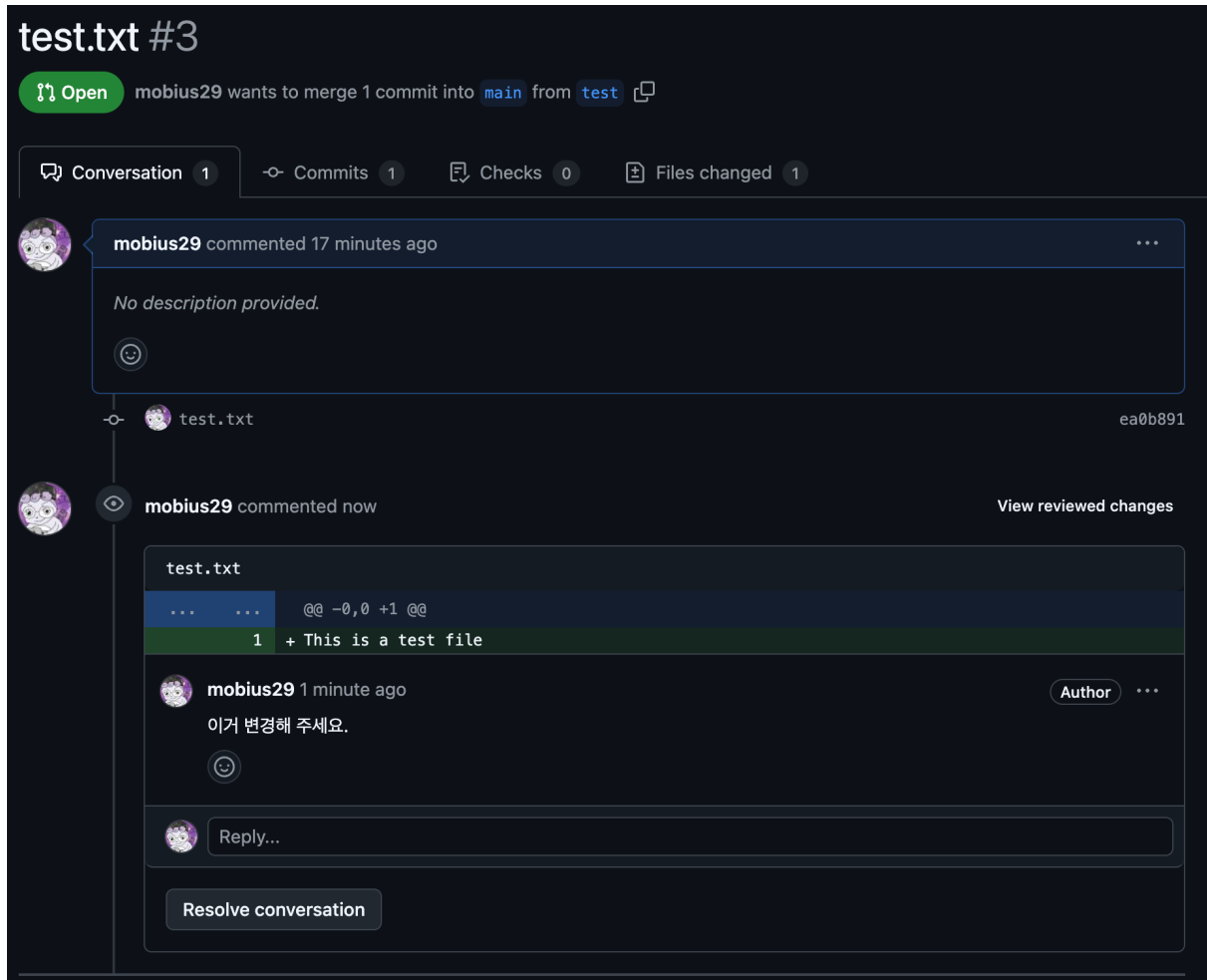
필자는 **이거 변경해 주세요.**라고 입력한 상태이다. 이는 경우에 따라 쓰고싶은 대로 쓰면 된다.



내용을 적으면 **Start a review** 버튼을 누르면 내용이 기록되는데, 아직 Pending 상태인 걸 볼 수 있다.



오른쪽 위, **Finish your review** 버튼 클릭 후, **Submit review** 버튼을 누르면 그제서야 리뷰를 등록할 수 있다. 그러면 이제 Conversation에서 리뷰 코멘트를 확인할 수 있다.



정말로 간략하게 리뷰 방식을 설명하였으며, 리뷰 기능들은 정말 다양하다. 이는 차차 알아가보면 좋겠다.

그리고 아래 **Merge pull request, Confirm merge**를 클릭하여 머지를 완료하자.

다시 CLI로 돌아와 **main** 브랜치를 보면 여전히 반영이 안되어있다. 그 이유는 우리는 병합을 GitHub, 즉 원격 레포지토리에서만 진행했기 때문이다. 그래서 우리는 원격 레포지토리에 있는 정보를 로컬 레포지토리로 가져올 필요가 있다. 이 때 쓰는 명령어는 **pull** 이다.

```
$ git checkout main

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

$ git pull origin main

remote: Enumerating objects: 1, done.
remote: Counting objects: 100% (1/1), done.
remote: Total 1 (delta 0), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (1/1), 907 bytes | 907.00 KiB/s, done
From https://github.com/mobius29/git-playground
* branch                main          -> FETCH_HEAD
   2f7a7eb..067f04d    main          -> origin/main
Updating 2f7a7eb..067f04d
Fast-forward
 test.txt | 1 +
 1 file changed, 1 insertion(+)
 create mode 100644 test.txt

$ cat test.txt
This is a test file
```

무사히 **test.txt** 파일이 존재하는 것을 확인할 수 있다.

Conflict?

우리는 branch를 통해 사람마다 독립적으로 작업을 할 수 있게 되었다.

그러면 다음과 같은 의문이 들 수 있다.

여러 사람이 같은 부분을 수정하면 어떡하지?
누구의 것을 선택해야 하지? 둘 다 필요하다면?

애초에 같은 부분을 수정했는지 어떻게 알지?

한 번 직접 테스트를 해보자. 브랜치를 2개 생성하여 같은 부분을 수정할 것이다.

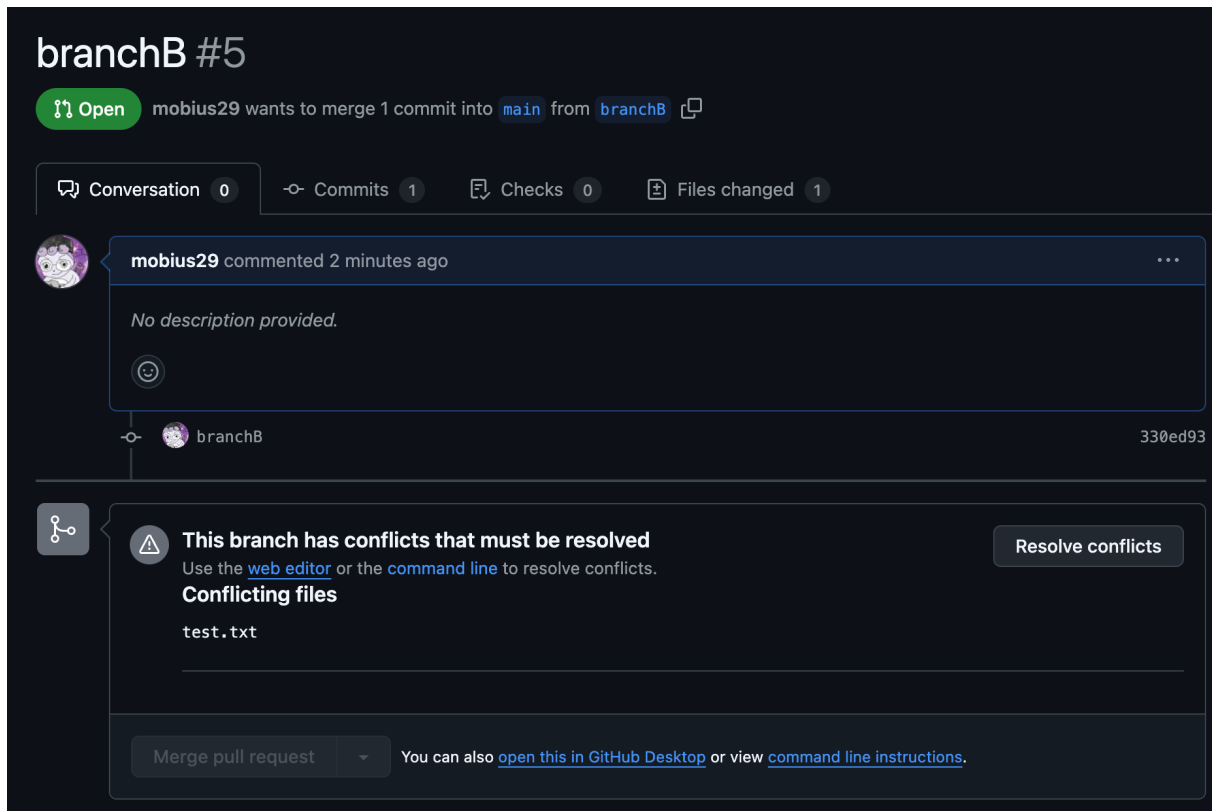
main 브랜치에서 **branchA** 와 **branchB**를 만들어 각각 **test.txt** 파일을 수정해보자.

```
# branchA
$ git checkout main
$ git checkout -b branchA
$ echo 'This is a branchA file' > test.txt
$ git add test.txt
$ git commit -m "branchA"
$ git push origin branchA

# branchB
$ git checkout main
$ git checkout -b branchB
$ echo 'This is a branchB file' > test.txt
$ git add test.txt
$ git commit -m "branchB"
$ git push origin branchB
```

이제 branchA와 branchB를 각각 main으로 병합하기 위해 PR을 생성해보자.

두 개가 모두 무사히 생성되며, **Merge pull request** 버튼도 둘 다 잘 활성화되어 있다. 그러면 이제 branchA를 먼저 병합하고, branchB의 PR을 확인해보자.



갑자기 병합 버튼이 비활성화로 변경되었다. 그리고 에러로 다음과 같은 메시지가 떠있다.


This branch has conflicts that must be resolved

같은 파일이 수정되어, 즉 충돌이 발생하여 어떤 수정사항을 반영해야할 지 Git이 알지 못해 우리보고 수정하라고 경고하는 것이다. 그러니 이제 해결해보자.

옆에

Resolve conflicts 버튼을 통해 웹에서도 해결을 할 수 있지만, 개인적으로 추천하지 않는다. 충돌되는 파일이 많거나 특별한 경우에는 저 버튼조차 쓸 수 없기 때문이다.

그러면 어떻게 해결해야 하는가? 친절히도 아래에 **GitHub Desktop**을 사용하거나 **Command line instructions**를 사용할 수 있다고 한다. 우리는 커맨드 라인 명령어로 해결할 것이며, 저 파란 부분을 클릭해보자.

 **This branch has conflicts that must be resolved** Resolve conflicts

Use the [web editor](#) or the [command line](#) to resolve conflicts.

Conflicting files

test.txt

Merge pull request

You can also [open this in GitHub Desktop](#) or view [command line instructions](#).

Checkout via command line

If the conflicts on this branch are too complex to resolve in the web editor, you can check it out via command line to resolve the conflicts.

HTTPSSSHPatch

<https://github.com/mobius29/git-playground.git>

Step 1: Clone the repository or update your local repository with the latest changes.

`git pull origin main`

Step 2: Switch to the head branch of the pull request.

`git checkout branchB`

Step 3: Merge the base branch into the head branch.

`git merge main`

Step 4: Fix the conflicts and commit the result.

See [Resolving a merge conflict using the command line](#) for step-by-step instructions on resolving merge conflicts.

Step 5: Push the changes.

`git push -u origin branchB`

정말 친절하게도 아래에 충돌해결하는 명령어마저 다 설명해주고 있다. 내가 이 설명을 쓸 필요가 있는지도 의문이다.

각설하고, 각 단계별로 진행해보자. 먼저 main 브랜치 최신화 후, branchB로의 이동은 따로 설명하지 않겠다.

```
$ git checkout main
$ git pull origin main
$ git checkout branchB
```

이제 충돌을 해결하기 위해 먼저 최신화된 main 브랜치를 branchB에 반영한다.

```
git merge main
Auto-merging test.txt
```

```
CONFLICT (content): Merge conflict in test.txt
Automatic merge failed; fix conflicts and then commit the res
```

자동 병합이 실패하였으며, **test.txt**에서 충돌이 발생했다고 알려주고 있다. 그러면 충돌난 부분을 확인하기 위해 test.txt 파일을 열어보자.

```
<<<<<<< HEAD
This is a branchB file
=====
This is a branchA file
>>>>>> main
```

분명 우리는 한 줄씩만 작성하였는데, 뭔가 이상한 것들이 더 붙어있다. <<~ HEAD 부터 ===== 까지는 현재 브랜치인 **branchB**의 변경사항이고, ===== 부터 >> ~ main 까지는 우리가 병합을 시도한 **main** 브랜치의 변경사항을 의미한다. 충돌해결은 저 꺾쇠들을 전부 없애는 것이다.

그래서 우리는 branchA와 branchB를 전부 적용하기 위해 다음과 같이 남겼다.

```
This is a branchB file
This is a branchA file
```

그리고 다시 커밋과 푸시를 진행하자.

```
$ git add test.txt
$ git commit -m "branchB"
$ git push origin branchB
```

다시 한 번 PR을 확인해보자. 무사히 병합 버튼이 활성화된 것을 볼 수 있다. 병합하는 것으로 마무리를 하자.

Git은 어떻게 관리하지?

우리는 지금까지 Git/GitHub으로 협업하는 아주 기초적인 방법들을 배웠다. 말그대로 아주 **기초적인** 방법들이며, 다른 방법들 또한 많다. 그리고 각기 다른 방법들을 사용하는 사람들끼리 모여 각자의 방식으로 협업하면, Git을 쓰지 않는 편이 낫다고 생각할 수도 있다.

그래서 사람들끼리 모여 Git 사용 방법을 통일하기로 하였다. 그리고 이는 프로젝트 by 프로젝트이므로 매 프로젝트마다 달라질 수 있으니 하나만 맹신하지 않기로 미리 약속하길 바란다.

그럼에도 가장 많이 차용되는 방법들이 있다. Git-flow와 GitHub-flow이다.

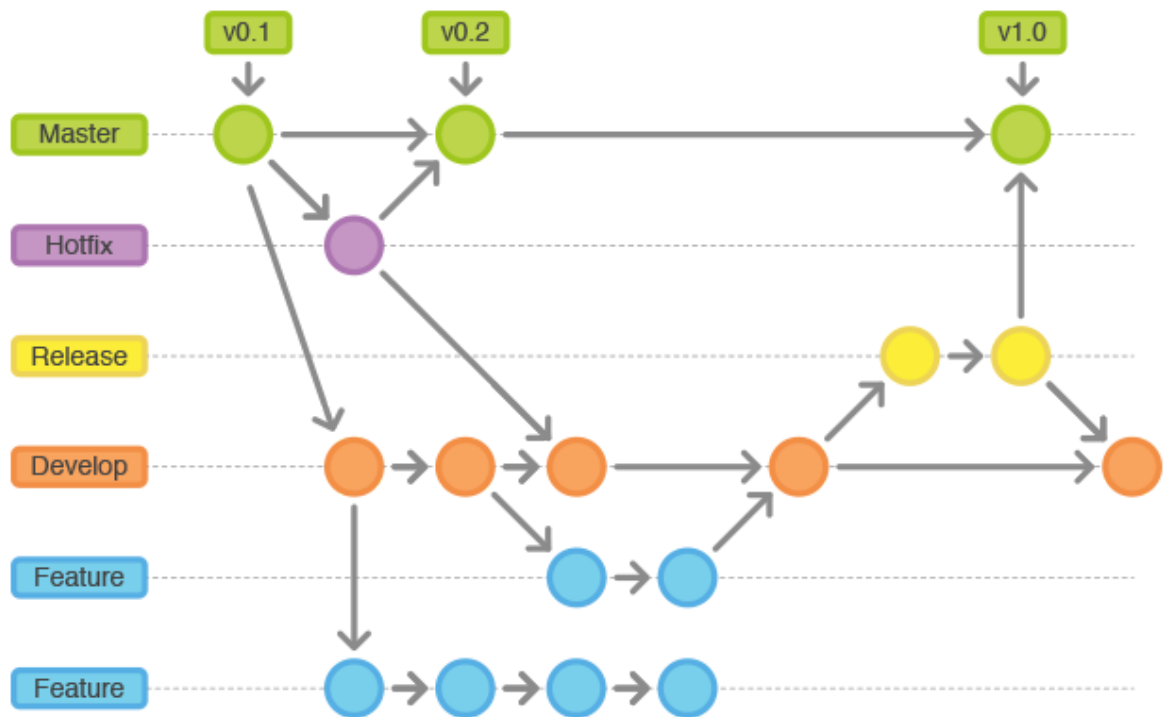
GitLab-flow도 있긴 한데 필자가 주로 써본 방식은 아니며, GitLab의 기능을 효과적으로 사용할 수 있는 거라 GitHub을 사용할 우리와는 안맞다고 생각하여 생략하였다.

그러므로 Git-flow와 GitHub-flow, 두 개를 비교해보자.

Git-flow

Git-flow는 크게 3개의 역할(**master**, **develop**, **support**)로 나눌 수 있다. 이 중 **master**와 **develop**은 역할 그 자체로 브랜치가 되지만, **support**는 이 역할 내에서 또다시 3개의 역할(**feature**, **hotfix**, **release**)로 나뉘어 총 5개의 브랜치(**master**, **develop**, **feature**, **hotfix**, **release**)가 생긴다.

각 브랜치의 역할은 프로젝트가 진행되는 과정을 따라가면서 설명할 것이다. [그림 3-1]도 함께 보도록 하자.



1. 모든 프로젝트는 **master** 브랜치에서 시작하며, **master** 브랜치는 실제로 배포되는 브랜치이다.
2. **master** 브랜치에서 **develop** 브랜치를 생성한다. 개발자들은 **develop** 브랜치를 기준으로 **feature** 브랜치를 생성한다.
3. 기능을 작업하기 위해 개발자가 **develop** 브랜치에서 **feature** 브랜치를 생성한다. **feature** 브랜치는 해당 브랜치에서 개발할 기능에 따라 이름을 달리할 수 있으며, 해당 기능은 해당 브랜치 외의 다른 **feature** 브랜치에서 작업하지 않는 것이 원칙이다.
4. 기능 작업이 완료되어 **feature** 브랜치를 **develop** 브랜치로 머지시킨다.
5. 배포에 필요한 모든 기능 작업이 완료되면 **develop** 브랜치에서 **release** 브랜치를 생성한다. 이 브랜치에서는 테스트를 통해 버그를 고치며, 이 브랜치의 존재 덕에 아직 배포하지 않을 기능은 **feature** 브랜치에서 계속 작업할 수 있다.
6. 테스트까지 마무리가 되면 배포할 준비가 끝난다. **release** 브랜치를 **master** 브랜치로 머지시키며, **master** 브랜치에 버전을 붙인다. 동시에 **release** 브랜치와 **develop** 브랜치의 동기화를 위해 **develop**으로도 머지시킨다.
7. 배포된 버전에서 치명적인 문제가 발생하여 긴급 수정이 필요할 경우, **master** 브랜치에서 **hotfix** 브랜치를 생성한다. **hotfix** 브랜치에서 문제를 수정 후 바로 **master**에 머지시켜 배포한다. 그리고 **develop**에도 머지시켜 수정사항을 동기화한다.

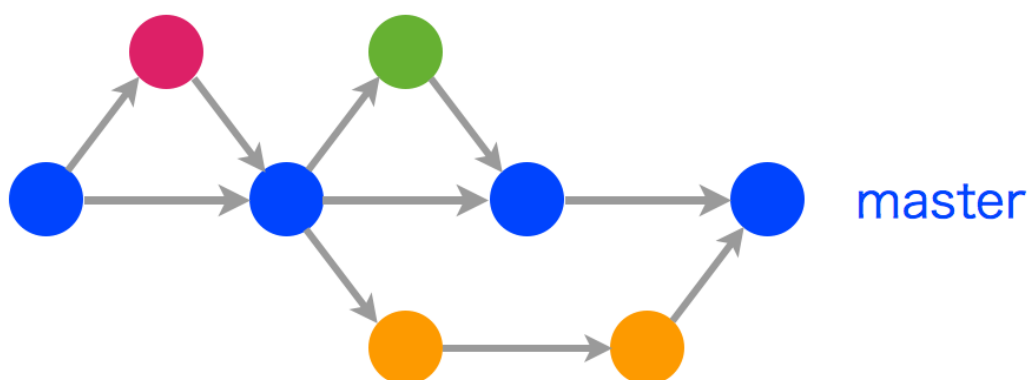
이 전략은 앱이나 게임 등 버전이 확실하며, 배포가 자주 일어나지 않는 상황에서 유용한 전략이다. 하지만 웹처럼 배포가 자주 일어나며, 이전 버전을 사용할 일이 없는 경우에 Git-flow 전략은 적합하지 않으며 오히려 과한 작업일 수 있다. 그래서 이를 해결하기 위한 전략이 GitHub-flow이다.

GitHub-flow

GitHub-flow는 Git-flow와 다르게 브랜치의 종류가 **master** 브랜치와 **feature** 브랜치로 단 2개 뿐이다. 그래서 브랜치 관리는 Git-flow에 비해 확실히 편하지만, 그만큼 우리가 생각해야 할 것들이 많아진다.

GitHub-flow 역시 동작하는 과정을 통해 브랜치도 함께 설명하도록 하겠다.

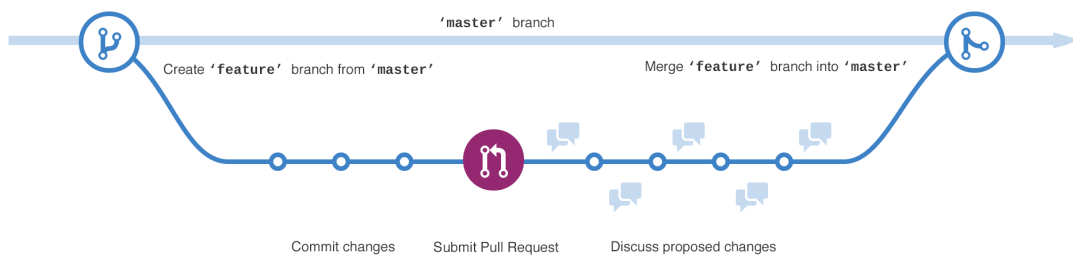
GitHub flow



1. **master** 브랜치는 배포 브랜치이다. 이 브랜치는 항상 **stable(배포 가능)**해야 하며, 그러기에 엄격하게 관리해야 한다.
2. 작업을 위해 **feature** 브랜치를 생성한다. GitHub-flow는 **release**, **hotfix** 등으로 나누지 않기 때문에 브랜치명을 자세히 적어두어야 한다.

- a. 작업하면서 원격 저장소로 수시로 푸시해준다. 팀원들이 작업 진행 상황을 알기가 쉽다.
3. 작업이 완료되면 **pull request**를 생성한다. 코드 리뷰 및 테스트를 통해 코드를 정리하고, 버그 등을 방지한다.
4. 작업이 완료되면 **master**로 머지 후 **즉시** 배포한다. 이 과정에서 **배포 자동화**를 사용하는 것이 매우 편하다.

이 중에 제일 까다로운 점은 1번이다. 항상 배포가 가능해야하며, 테스트를 위한 브랜치가 따로 존재하지 않기 때문에 머지하는 과정에서 최대한 버그가 일어나지 않도록 코드 리뷰 및 테스트를 꼼꼼하게 진행해야 한다. [그림 3-1]에서 하나의 브랜치만 따로 보면 아래 [그림 3-2]와 같다.



(개인적으로?) GitHub-flow는 이들이 준비되지 않으면 안쓰는게 낫다고 생각한다.

- 테스트 주도 개발(Test-Driven Development; TDD)
테스트 전용 브랜치가 없기 때문에 실제로 QA를 하기가 힘들다. 그래서 개발 과정에 테스트를 추가하여 버그를 제거해야 한다.
- 배포 자동화: 브랜치가 머지될 때마다 배포가 되어야 하며, 심지어 **master** 브랜치는 항상 배포가 가능해야 하기 때문에 배포 전에 빌드가 성공적으로 되는 지도 확인해야 한다. 이 과정을 배포 자동화를 통해 해결하면 GitHub-flow의 질이 높아지기 때문에 사용해야 한다.

주로 버전으로 관리되지 않는 **웹 개발**에서 주로 사용되는 브랜치 전략이라고 생각된다. 하지만 테스트 전용 브랜치가 없다는 단점으로 인해 정말로 까다롭게 사용해야 한다. 테스트 전용 브랜치를 만드는 방법도 추가할 수 있겠지만, 이 경우에는 당연하게도 브랜치 관리가 힘들어진다. 그러니 본인의 상황에 맞게 알맞은 전략과 알맞게 커스텀할 수 있도록 하자.