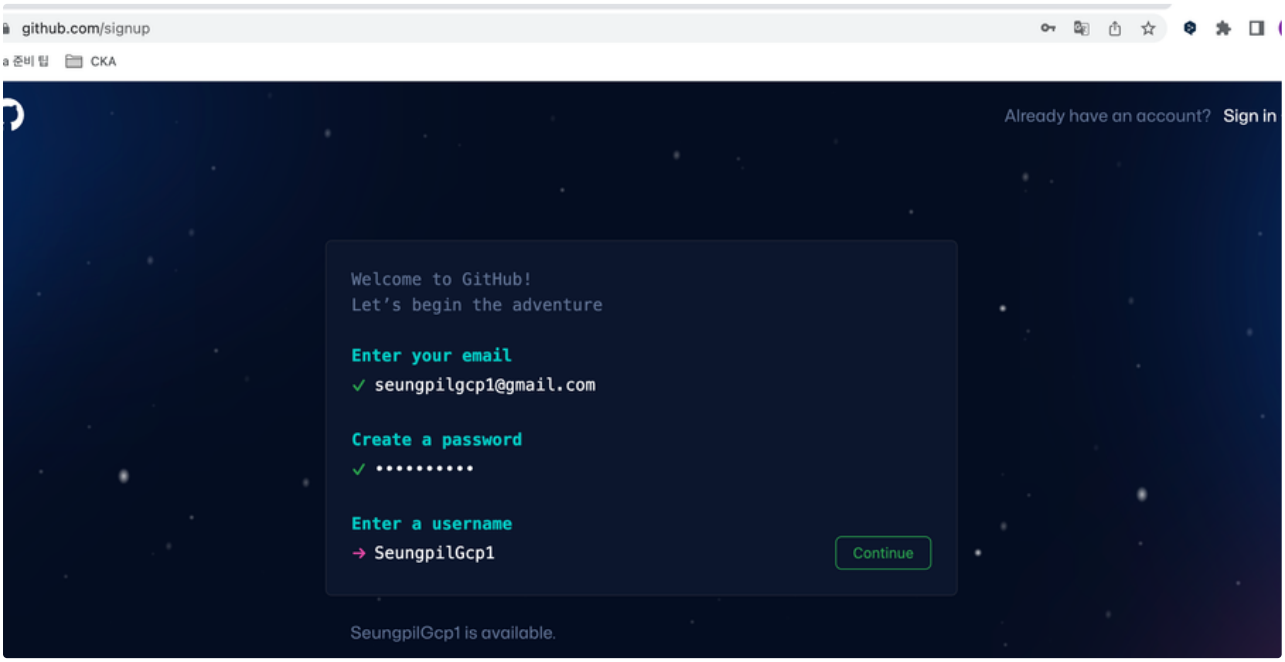
2. Git 기초 및 명령어 I

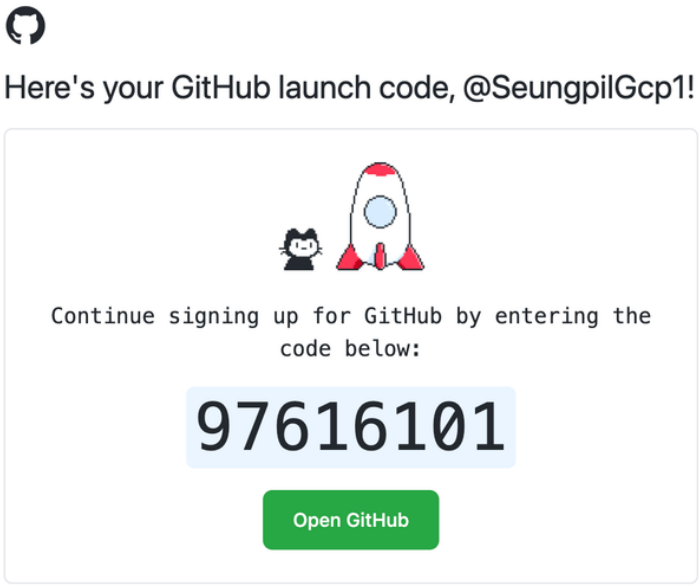
1. Github 가입

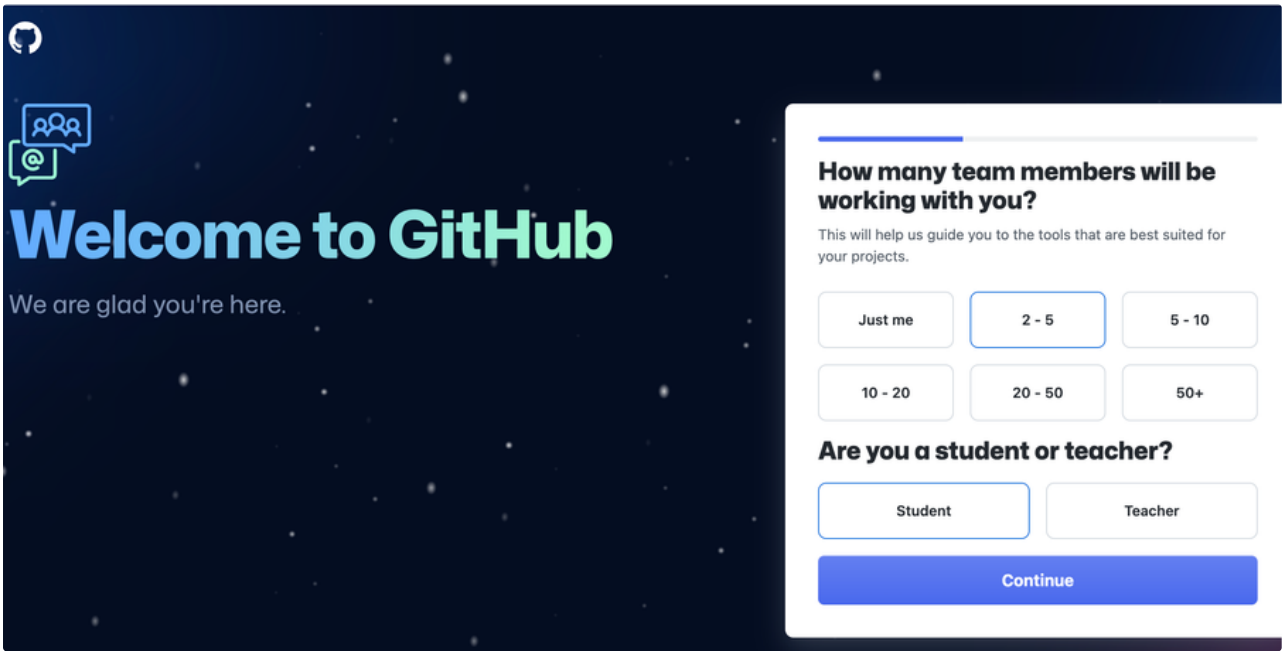
강의 시작 전에 Github 가입 을 완료 해 주세요.

[가입시 이메일 인증이 있으므로 정확한 이메일을 기입 해 주세요.]

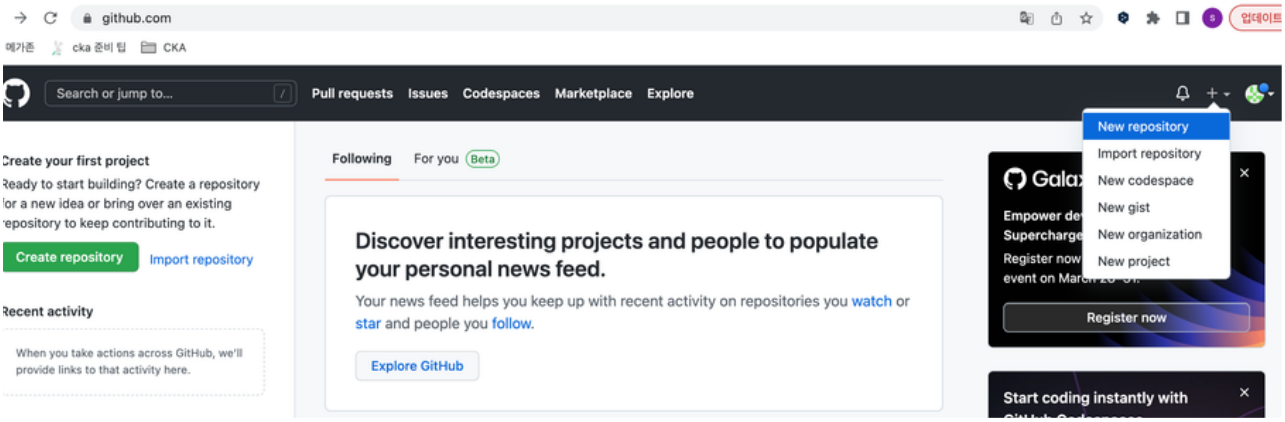


[가입시 이메일로 인증코드 기입]



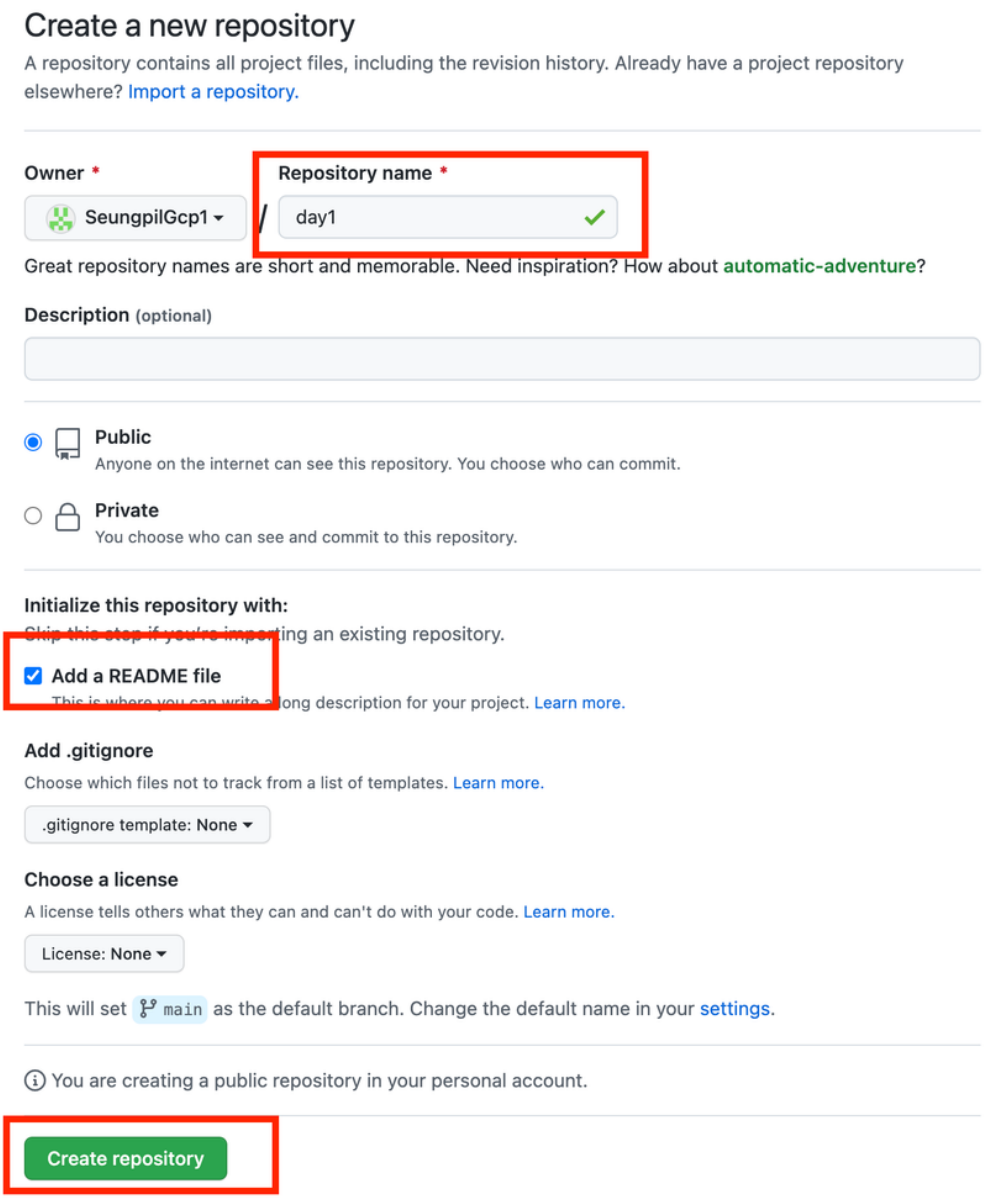


[가입 후 우측 상단 + 클릭 후 New repository]

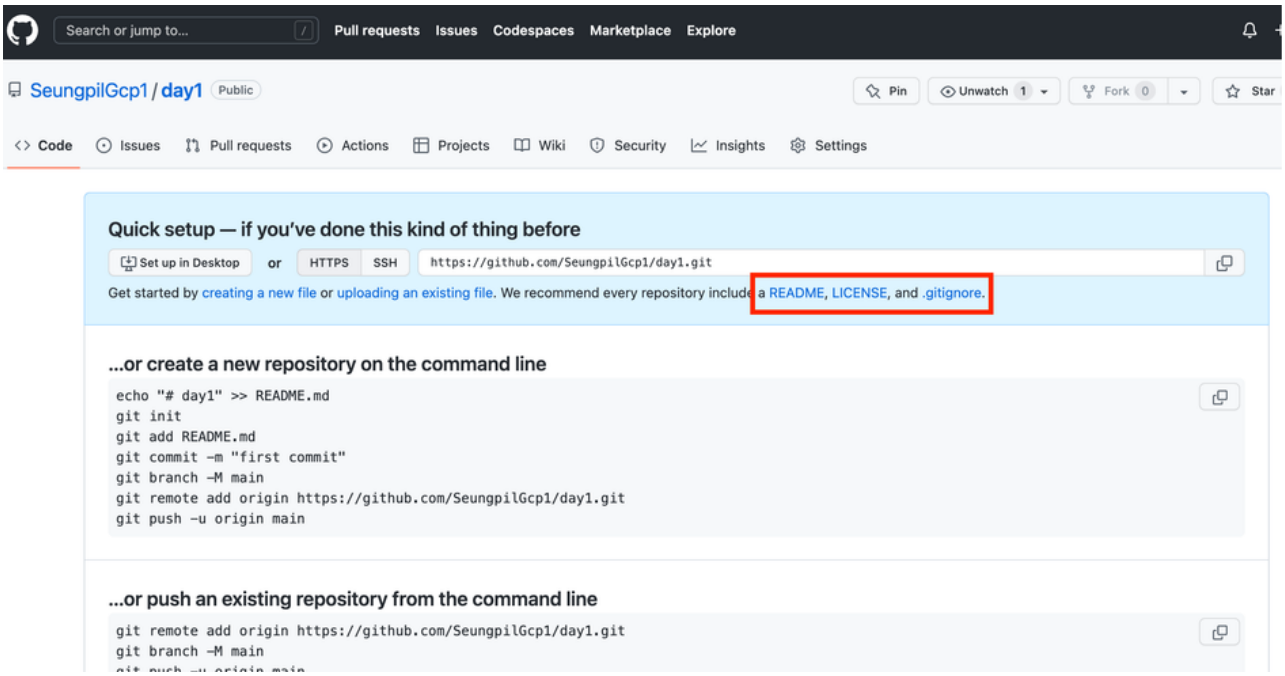


[신규 레파지토리 생성]

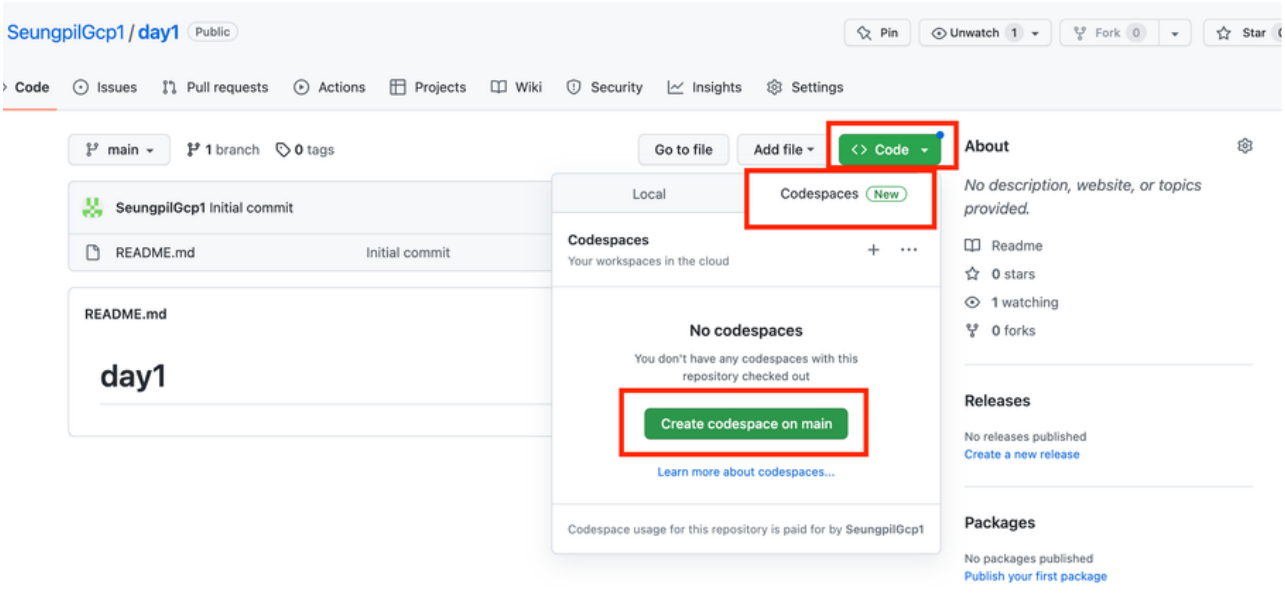
- 레파지토리 이름 기입
- Add a README 체크박스



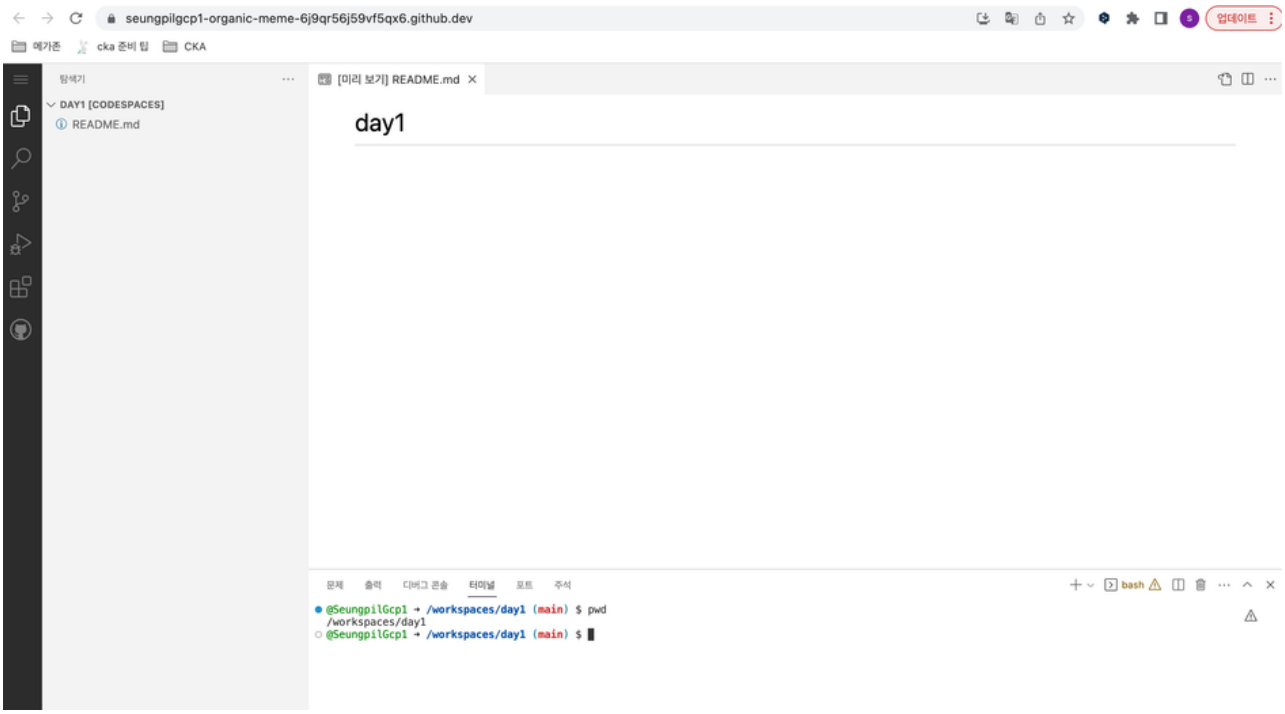
- 만약 Add a README 체크 박스 하지 않았을 시에는 아무런 파일도 생성되지 않은 레파지토리 이기 때문에 초기 브랜치가 없습니다. 이 상태에서는 codespace 를 실행할 수 없으니, 그럴 경우 아래 영역을 클릭하셔서 파일을 추가 해 주세요.



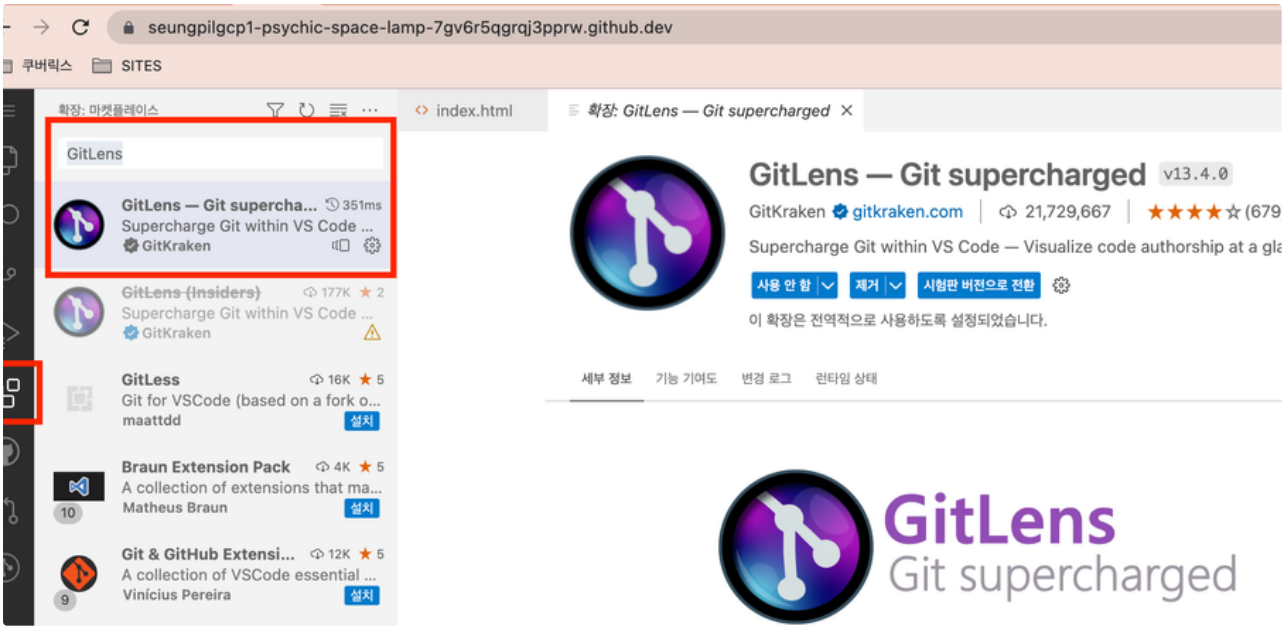
- 레파지토리 생성 후에 아래 그림을 따라 Create codespace on main 클릭



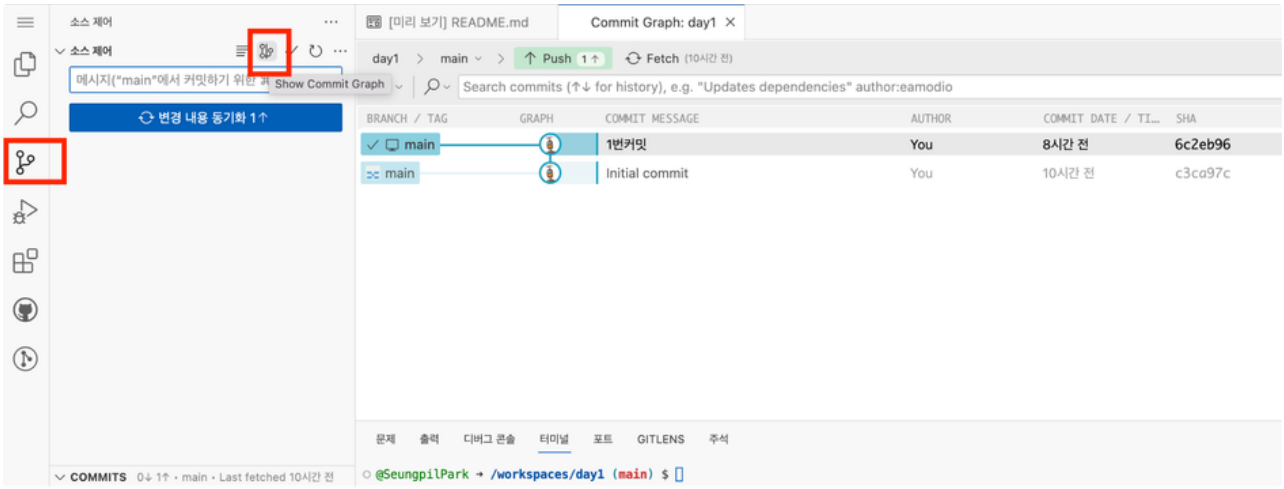
- 잠시 기다린 후 아래 처럼 웹 터미널이 열리면 실습 환경 구성이 완료되었습니다.



- 확장 프로그램에서 GitLens 까지 설치 해 주세요.



- GitLens 설치 후에는 소스제어 메뉴를 연 상태에서 Git 트리 모양 아이콘을 클릭하면 커밋 내역을 비쥬얼라이징 하게 확인 할 수 있습니다.



---

2. Git 저장소 만들기

기존 디렉토리 Git 저장소로 만들기

디렉토리 생성 및 이동

```
$ mkdir /workspaces/new-project
$ cd /workspaces/new-project
```

디렉토리를 git 저장소로 만들기

```
$ git init
```

이 명령은 `.git` 이라는 하위 디렉토리를 만든다. `.git` 디렉토리에는 저장소에 필요한 뼈대 파일(Skeleton)이 들어 있다. 이 명령만으로는 아직 프로젝트의 어떤 파일도 관리하지 않는다. (`.git` 디렉토리가 막 만들어진 직후에 정확히 어떤 파일이 있는지에 대한 내용은 Git의 내부에서 다룬다)

저장소에 파일을 추가하고 커밋

Git 이 파일을 관리하게 하려면 저장소에 파일을 추가하고 커밋해야 한다. `git add` 명령으로 파일을 추가하고 `git commit` 명령으로 커밋한다

```
$ vim LICENSE
$ git add LICENSE
$ git commit -m 'initial project version'
```

명령어 몇 개로 Git 저장소를 만들고 파일 버전 관리를 시작했다.

기존 저장소를 Clone 하기

다른 프로젝트에 참여 하려거나(Contribute) Git 저장소를 복사하고 싶을 때 `git clone` 명령을 사용한다.

- 이미 Subversion 같은 VCS 에 익숙한 사용자에게는 "checkout" 이 아니라 "clone" 이라는 점
- Git이 Subversion과 다른 가장 큰 차이점은 서버에 있는 거의 모든 데이터를 복사한다.

기존 저장소를 로컬에 `clone` 하기

`git clone` 을 실행하면 프로젝트 히스토리를 전부 받아온다. 실제로 서버의 디스크가 망가져도 클라이언트 저장소 중에서 아무거나 하나 가져다가 복구하면 된다 (서버에만 적용했던 설정은 복구하지 못하지만 모든 데이터는 복구된다)

`git clone <url>` 명령으로 저장소를 Clone 한다. `libgit2` 라이브러리 소스코드를 Clone 하려면 아래과 같이 실행한다.

```
$ cd /workspaces
$ git clone https://github.com/libgit2/libgit2
```

이 명령은 libgit2 라는 디렉토리를 만들고 그 안에 `.git` 디렉토리를 만든다. 그리고 저장소의 데이터를 모두 가져와서 자동으로 가장 최신 버전을 Checkout 해 놓는다. `libgit2` 디렉토리로 이동하면 Checkout 으로 생성한 파일을 볼 수 있고 당장 하고자 하는 일을 시작할 수 있다.

다른 디렉토리 이름으로 `clone` 하기

아래과 같은 명령을 사용하여 저장소를 Clone 하면 `libgit2` 이 아니라 다른 디렉토리 이름으로 Clone 할 수 있다.

```
$ cd /workspaces
$ git clone https://github.com/libgit2/libgit2 mylibgit
```

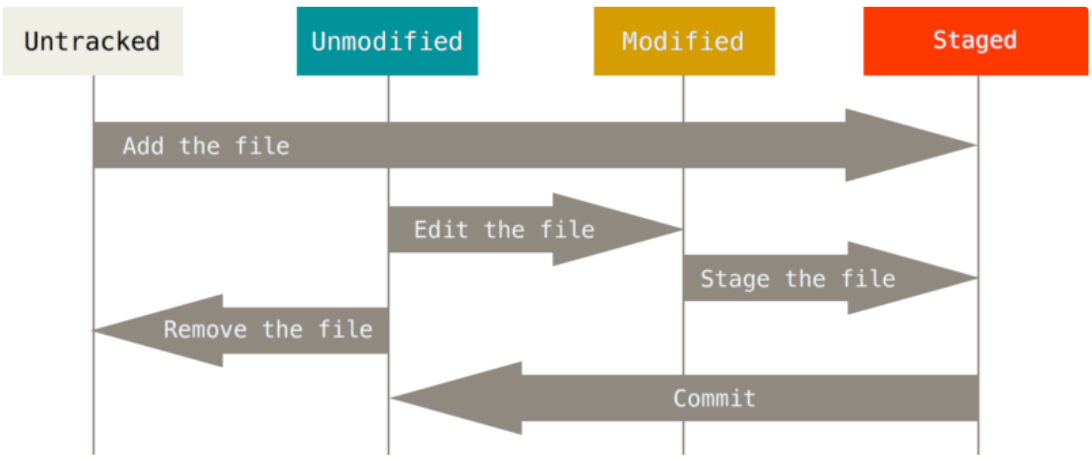
디렉토리 이름이 `mylibgit` 이라는 것만 빼면 이 명령의 결과와 앞선 명령의 결과는 같다.

---

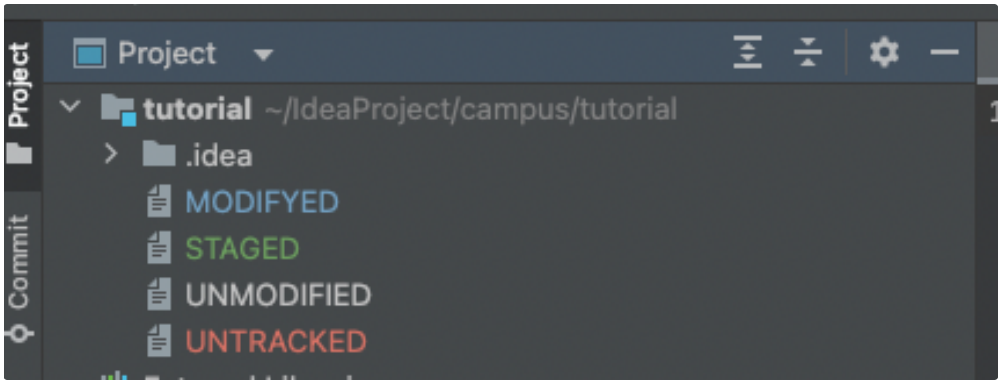
3. Git 으로 파일 형상관리 하기

Git 파일상태 알아보기

- Tracked 파일
  - Git 이 알고 있는 파일.
  - Tracked 파일은 또 Unmodified(수정하지 않음)와 Modified(수정함) 그리고 Staged(커밋으로 저장소에 기록할) 상태 중 하나이다.
  - Modified 상태는 파일을 변경했지만 아직 데이터베이스에 커밋하지 않았음을 의미합니다.
  - Staged 상태는 수정된 파일을 현재 버전에서 다음 커밋 스냅샷으로 이동하도록 표시했음을 의미합니다.
  - Committed 상태는 데이터가 로컬 데이터베이스에 안전하게 저장되었음을 의미합니다.
- 나머지 파일은 모두 Untracked 파일이다.
  - Untracked 파일은 워킹 디렉토리에 있는 파일 중 스냅샷에도 Staging Area에도 포함되지 않은 파일.



[IDEA 툴 등에서는 Git 파일상태에 따라 파일이 보여지는 색이 다르다.]



처음 저장소를 Clone 하면 모든 파일은 Tracked이면서 Unmodified 상태이다. Clone 을 하였다는 것은 이미 git 이 알고있는 파일(Tracked) 을 내려받았다는 것이고, 그 이후 아무것도 수정하지 않았기(Unmodified) 때문에 그렇다.

마지막 커밋 이후 아직 아무것도 수정하지 않은 상태에서 어떤 파일을 수정하면 Git은 그 파일을 **Modified** 상태로 인식한다. 실제로 커밋을 하기 위해서는 이 수정한 파일을 Staged 상태로 만들고, Staged 상태의 파일을 커밋한다. 이런 라이프사이클을 계속 반복한다.

Untracked 파일 생성 하고 상태 확인 하기

git status 명령을 사용하여 파일의 상태를 확인할 수 있다. 아직 아무 파일도 수정되지 않았음을 확인 가능하다.

```
$ cd /workspaces/day1
$ git status
On branch main
Your branch is up-to-date with 'origin/main'.
nothing to commit, working directory clean
```

프로젝트에 README 파일을 만들어보자. README 파일은 새로 만든 파일이기 때문에 git status 를 실행하면 'Untracked files'에 들어 있다.

```
$ echo 'My Project' > README
$ git status
On branch main
Your branch is up to date with 'origin/main'.

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

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

`README` 파일은 “Untracked files” 부분에 속해 있는데 이것은 `README` 파일이 Untracked 상태라는 것을 말한다.

Git은 Untracked 파일을 아직 스냅샷(커밋) 에 넣어지지 않은 파일이라고 본다. 파일이 Tracked 상태가 되기 전까지는 Git은 그 파일을 커밋 하지 않는다. `README` 파일을 add 하여 직접 Tracked 상태로 만들어 보자.

파일을 새로 추적(Tracked) 하기

`git add` 명령으로 파일을 새로 추적(Tracked) 할 수 있다. 아래 명령을 실행하면 Git은 `README` 파일을 추적한다.

```
$ git add README
```

`git status` 명령을 다시 실행하면 `README` 파일이 Tracked 상태이면서 커밋에 추가될 Staged 상태라는 것을 확인할 수 있다.

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

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   README
```

“Changes to be committed” 에 들어 있는 파일은 Staged 상태라는 것을 의미한다.

- `git add`
  - git 을 커밋하면, `git add` 를 실행한 시점의 파일이 커밋되어 저장소 히스토리에 남는다.
  - 이 명령을 통해 디렉토리에 있는 파일을 추적하고 관리하도록 한다.
  - 파일 또는 디렉토리의 경로를 아규먼트로 받는다.
  - 디렉토리면 아래에 있는 모든 파일들까지 재귀적으로 추가한다.

Modified 상태의 파일을 Stage 하기

이미 Tracked 상태인 파일을 수정하는 법을 알아보자. `CONTRIBUTING.md` 라는 파일을 수정하고 나서 `git status` 명령을 다시 실행하면 결과는 아래와 같다.

```
# CONTRIBUTING.md 추가하고 Tracked 상태 만들기
$ echo 'My Project' > CONTRIBUTING.md
$ git add CONTRIBUTING.md

# CONTRIBUTING.md 변경 하고 상태 확인
$ echo 'Changed!!' > CONTRIBUTING.md

$ git status
On branch main
Your branch is up to date with 'origin/main'.

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   CONTRIBUTING.md
        new file:   README

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   CONTRIBUTING.md
```

`CONTRIBUTING.md` 가 Staged 상태이면서 *동시에* Unstaged 상태로 나온다.

`git add` 명령을 실행하면 Git은 파일을 바로 Staged 상태로 만든다. 지금 이 시점에서 커밋을 하면 `git commit` 명령을 실행하는 시점의 버전이 커밋되는 것이 아니라 마지막으로 `git add` 명령을 실행했을 때의 버전이 커밋된다.

그러니까 `git add` 명령을 실행한 후에 또 파일을 수정하면 `git add` 명령을 다시 실행해서 최신 버전을 Staged 상태로 만들어야 한다.

Modified 된 `CONTRIBUTING.md` 파일을 다시 Staged 상태로 만들려면 `git add` 명령을 실행해야 한다. `git add` 명령은 파일을 새로 추적할 때도 사용하고 수정한 파일을 Staged 상태로 만들 때도 사용한다. Merge 할 때 충돌난 상태의 파일을 Resolve 상태로 만들때도 사용한다.

add 의 의미는 프로젝트에 파일을 추가한다기 보다는 다음 커밋에 추가한다고 받아들이는게 좋다. `git add` 명령을 실행하여 `CONTRIBUTING.md` 파일을 Staged 상태로 만들고 `git status` 명령으로 결과를 확인해보자.

```
$ git add CONTRIBUTING.md
$ git status
On branch main
Your branch is up to date with 'origin/main'.

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   CONTRIBUTING.md
        new file:   README
```

두 파일 모두 Staged 상태이므로 다음 커밋에 포함된다.

파일 상태 short 보기

`git status` 명령으로 확인할 수 있는 내용이 좀 많아 보일 수 있다.

`git status -s` 또는 `git status --short` 처럼 옵션을 주면 현재 변경한 상태를 간소화 하여 보여준다.

```
# FILE1, FILE2 추가
$ echo 'new' > FILE1
$ echo 'new' > FILE2

# FILE2 는 Staged 상태로 변경
$ git add FILE2

# FILE2 를 다시 수정
$ echo 'changed again' > FILE2

# 상태 조회
$ git status -s
A  CONTRIBUTING.md
A  README
AM FILE2
?? FILE1
```

- 아직 추적하지 않는 새 파일 : `??` 표기
- Tracked 파일
  - 왼쪽: Staging Area 에서의 상태. `A` 표시가 붙었다.
  - 오른쪽: Working Tree 에서의 상태. 위 명령의 결과 에서 `FILE2` 는 수정하였으므로 `M` 표시가 추가로 붙었다.

Staged (commit 예정) 와 워킹디렉터리 사이의 변경 내용을 보기

모든 변경사항을 commit 하여 Staging Area 를 초기화 한다.

```
$ git add .
$ git commit -m 'cleanup'
```

단순히 파일이 변경됐다는 사실이 아니라 어떤 내용이 변경됐는지 살펴보려면 `git diff` 명령을 사용 한다.

더 자세하게 볼 때는 `git diff` 명령을 사용하는데 어떤 라인을 추가했고 삭제했는지가 궁금할 때 사용한다.

`STAGED_FILE` 파일을 수정해서 Staged 상태로 만들고 `MODIFIED_FILE` 파일은 그냥 수정만 해둔다. 이 상태에서 `git status` 명령을 실행하면 아래와 같은 메시지를 볼 수 있다.

```
$ echo 'Init' > STAGED_FILE
$ echo 'Init' > MODIFIED_FILE
$ git add .

$ echo 'Changed!!' > MODIFIED_FILE
$ git status
On branch main
Your branch is ahead of 'origin/main' by 2 commits.
  (use "git push" to publish your local commits)

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   MODIFIED_FILE
        new file:   STAGED_FILE

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   MODIFIED_FILE
```

`git diff` 명령을 실행하면 수정했지만 아직 staged 상태가 아닌 파일(`MODIFIED_FILE`) 을 비교해 볼 수 있다.

이 명령은 워킹 디렉토리에 있는 것과 Staging Area에 있는 것을 비교한다. 그래서 수정하고 아직 Stage 하지 않은 것을 보여준다.

```
$ git diff
diff --git a/MODIFIED_FILE b/MODIFIED_FILE
index 972ea8f..b150970 100644
--- a/MODIFIED_FILE
+++ b/MODIFIED_FILE
@@ -1 +1 @@
-Init
+Changed!!
```

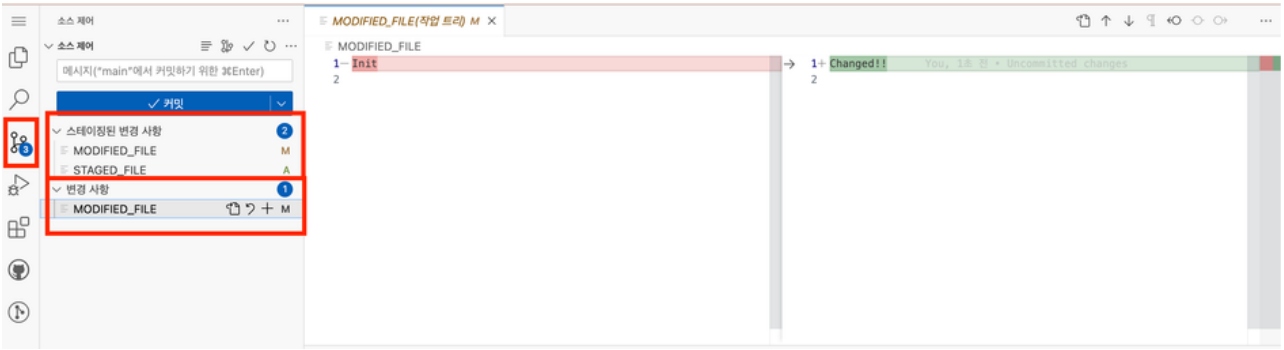
Staged (commit 예정) 의 변경 내용을 보기

만약 커밋하려고 Staging Area에 넣은 파일의 변경 부분을 보고 싶으면 `git diff --staged` 옵션을 사용한다. 이 명령은 저장소에 커밋한 것과 Staging Area에 있는 것을 비교한다.

```
$ git diff --staged
diff --git a/MODIFIED_FILE b/MODIFIED_FILE
new file mode 100644
index 0000000..972ea8f
--- /dev/null
+++ b/MODIFIED_FILE
@@ -0,0 +1 @@
+Init
diff --git a/STAGED_FILE b/STAGED_FILE
new file mode 100644
index 0000000..972ea8f
--- /dev/null
+++ b/STAGED_FILE
@@ -0,0 +1 @@
+Init
```

살펴본 2가지의 변경 내역 보기는 CodeSpace 상의 화면에서도 확인이 가능하다.

또한 IDEA 도구의 git 플러그인은 대다수 이러한 파일 변경 내역 보기 UI 를 제공하며 개발 시에 매우 유용하다.



변경사항 커밋하기

Unstaged 상태의 파일은 커밋되지 않는다.
커밋하기 전에 `git status` 명령으로 모든 것이 Staged 상태인지 확인한 후에 `git commit` 을 실행하여 커밋한다.

```
$ git status
$ git add MODIFIED_FILE
$ git commit
```

Git 설정에 지정된 편집기가 실행되고, 아래와 같은 텍스트가 자동으로 포함된다. 편집기는 아래와 같은 내용을 표시한다.

```

# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
#
# On branch main
# Your branch is ahead of 'origin/main' by 2 commits.
#   (use "git push" to publish your local commits)
#
# Changes to be committed:
#   new file:   MODIFIED_FILE
#   new file:   STAGED_FILE
#
```

- 자동으로 생성되는 커밋 메시지의 첫 라인은 비어 있고 둘째 라인부터 `git status` 명령의 결과가 채워진다.
- 커밋한 내용을 쉽게 기억할 수 있도록 이 메시지를 포함할 수도 있고 메시지를 전부 지우고 새로 작성할 수 있다
- 정확히 뭘 수정했는지도 보여줄 수 있는데, `git commit` 에 -v 옵션을 추가하면 편집기에 diff 메시지도 추가된다.
- 내용을 저장하고 편집기를 종료하면 Git은 입력된 내용(#로 시작하는 내용을 제외한)으로 새 커밋을 하나 완성한다.

메시지를 인라인으로 첨부할 수도 있다. `commit` 명령을 실행할 때 아래와 같이 `-m` 옵션을 사용한다.

```
$ git commit -m "Story 182: Fix benchmarks for speed"
[main b37981e] Story 182: Fix benchmarks for speed
 2 files changed, 2 insertions(+)
 create mode 100644 MODIFIED_FILE
 create mode 100644 STAGED_FILE
```

`commit` 명령은 몇 가지 정보를 출력하는데 위 예제는 (`main`) 브랜치에 커밋했고 체크섬은 (`b37981e`)이라고 알려준다. 그리고 수정한 파일이 몇 개이고 삭제됐거나 추가된 라인이 몇 라인인지 알려준다.

Git 은 Staging Area 에 속한 스냅샷을 커밋한다. 수정은 했지만, 아직 Staging Area 에 넣지 않은 것은 다음에 커밋할 수 있다. 커밋할 때마다 프로젝트의 스냅샷을 기록하기 때문에 나중에 스냅샷끼리 비교하거나 예전 스냅샷으로 되돌릴 수 있다.

파일 삭제하기

Git 에서 파일을 제거하려면 `git rm` 명령으로 Tracked 상태의 파일을 Staging Area 에서 삭제한 후에 커밋해야 한다. 이 명령은 워킹 디렉토리에 있는 파일도 삭제하기 때문에 실제로 파일도 지워진다.

Git 명령을 사용하지 않고 단순히 워킹 디렉터리에서 파일을 삭제하고 `git status` 명령으로 상태를 확인하면 Git은 현재 “Changes not staged for commit” (즉, *Unstaged* 상태)라고 표시해준다.

```
$ echo 'text' > PROJECTS.md
$ git add PROJECTS.md
$ git commit -m 'add PROJECTS.md'
$ rm PROJECTS.md
$ git status
On branch main
Your branch is ahead of 'origin/main' by 3 commits.
  (use "git push" to publish your local commits)

Changes not staged for commit:
  (use "git add/rm <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        deleted:    PROJECTS.md

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

그리고 `git rm` 명령을 실행하면 삭제한 파일은 Staged 상태가 된다.

```
$ git rm PROJECTS.md
rm 'PROJECTS.md'
$ git status
On branch main
Your branch is ahead of 'origin/main' by 3 commits.
  (use "git push" to publish your local commits)

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        deleted:    PROJECTS.md
```

커밋하면 파일은 삭제되고 Git은 이 파일을 더는 추적하지 않는다.

파일 삭제하기 옵션들

이미 파일을 수정했거나 Staging Area에(역주 - Git Index라고도 부른다) 추가했다면 `-f` 옵션을 주어 강제로 삭제해야 한다. 이 점은 실수로 데이터를 삭제하지 못하도록 하는 안전장치다. 커밋 하지 않고 수정한 데이터는 Git 으로 복구할 수 없기 때문이다.

```
$ git rm -f README
```

또 Staging Area에서만 제거하고 워킹 디렉토리에 있는 파일은 지우지 않고 남겨둘 수 있다. 다시 말해서 하드디스크에 있는 파일은 그대로 두고 Git만 추적하지 않게 한다. 이것은 `.gitignore` 파일에 추가하는 것을 빼먹었거나 대용량 로그 파일이나 컴파일된 파일인 `.a` 파일 같은 것을 실수로 추가했을 때 쓴다. `--cached` 옵션을 사용하여 명령을 실행한다.

```
$ git rm --cached README
```

여러 개의 파일이나 디렉토리를 한꺼번에 삭제할 수도 있다. 아래와 같이 `git rm` 명령에 file-glob 패턴을 사용한다.

```
$ git rm log/\*.log
```

`*` 앞에 `\` 을 사용한 것을 기억하자. 파일명 확장 기능은 쉘에만 있는 것이 아니라 Git 자체에도 있기 때문에 필요하다. 이 명령은 `log/` 디렉토리에 있는 `.log` 파일을 모두 삭제한다. 아래의 예제처럼 할 수도 있다.

```
$ git rm \*~
```

이 명령은 `~` 로 끝나는 파일을 모두 삭제한다.

파일 이름 변경하기

아래와 같이 파일 이름을 변경할 수 있다.

```
$ git mv file_from file_to
```

잘 동작한다. 이 명령을 실행하고 Git의 상태를 확인해보면 Git은 이름이 바뀐 사실을 알고 있다.

```
$ git mv README.md README
$ git status
On branch main
Your branch is ahead of 'origin/main' by 4 commits.
  (use "git push" to publish your local commits)

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        renamed:    README.md -> README
```

사실 `git mv` 명령은 아래 명령어를 수행한 것과 완전 똑같다.

```
$ mv README.md README
$ git rm README.md
$ git add README
```

`git mv` 명령은 일종의 단축 명령어이다. 이 명령으로 파일 이름을 바꿔도 되고 `mv` 명령으로 파일 이름을 직접 바꿔도 된다. 단지 `git mv` 명령은 편리하게 명령을 세 번 실행해 주는 것이다.

---

4. 커밋 히스토리 조회

커밋 히스토리 기본 조회

`git log` 명령을 실행하면 아래와 같이 출력된다.

```
$ git log
commit b37981e04b715b646e00643b9fdd306178779fd9 (HEAD -> main)
Author: Seungpil Park <darkgodarkgo@gmail.com>
Date:   Mon May 29 04:23:13 2023 +0000

    Story 182: Fix benchmarks for speed

commit ef0c7f1e3327b8a47ed03bc86cf9d98b4913f48f
```

```
Author: Seungpil Park <darkgodarkgo@gmail.com>
Date:   Mon May 29 04:08:53 2023 +0000

    cleanup

commit c3ca97cbb451aae3754d4c5fbead9b083f7d1f66 (origin/main, origin/HEAD)
Author: Seungpil Park <darkgodarkgo@gmail.com>
Date:   Sun May 28 15:42:03 2023 +0900

    Initial commit
```

특별한 아규먼트 없이 `git log` 명령을 실행하면 저장소의 커밋 히스토리를 시간순으로 보여준다. 즉, 가장 최근의 커밋이 가장 먼저 나온다. 그리고 이어서 각 커밋의 SHA-1 체크섬, 저자 이름, 저자 이메일, 커밋한 날짜, 커밋 메시지를 보여준다.

커밋 히스토리 옵션

각 커밋의 diff 결과를 함께 보면서, 최근 두개의 결과만 보여주기.

```
$ git log -p -2
```

커밋 히스토리 통계 보기

```
$ git log --stat
```

각 커밋 로그를 한줄로 요약하여 보기

```
$ git log --pretty=oneline
```

그래프 형태로 한줄로 요약하여 보기

```
$ git log --pretty=oneline --graph
```

포맷을 지정하여 원하는 항목 보기

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

	표 2. `git log` 주요 옵션
옵션	설명
`-p`	각 커밋에 적용된 패치를 보여준다.
`--stat`	각 커밋에서 수정된 파일의 통계정보를 보여준다.
`--shortstat`	`--stat` 명령의 결과 중에서 수정한 파일, 추가된 라인, 삭제된 라인만 보여준다.
`--name-only`	커밋 정보중에서 수정된 파일의 목록만 보여준다.
`--name-status`	수정된 파일의 목록을 보여줄 뿐만 아니라 파일을 추가한 것인지, 수정한 것인지, 삭제한 것인지도 보여준다.
`--abbrev-commit`	40자 짜리 SHA-1 체크섬을 전부 보여주는 것이 아니라 처음 몇 자만 보여준다.

`--relative-date`	정확한 시간을 보여주는 것이 아니라 “2 weeks ago” 처럼 상대적인 형식으로 보여준다.
`--graph`	브랜치와 머지 히스토리 정보까지 아스키 그래프로 보여준다.
`--pretty`	지정한 형식으로 보여준다. 이 옵션에는 oneline, short, full, fuller, format이 있다. format은 원하는 형식으로 출력하고자 할 때 사용한다.
`--oneline`	`--pretty=oneline --abbrev-commit` 두 옵션을 함께 사용한 것과 같다.

커밋 히스토리 포맷 옵션

`git log --pretty=format` 에 쓸 몇가지 유용한 옵션` 포맷에서 사용하는 유용한 옵션.

	표 1. `git log --pretty=format` 에 쓸 몇가지 유용한 옵션`
옵션	설명
`%H`	커밋 해시
`%h`	짧은 길이 커밋 해시
`%T`	트리 해시
`%t`	짧은 길이 트리 해시
`%P`	부모 해시
`%p`	짧은 길이 부모 해시
`%an`	저자 이름
`%ae`	저자 메일
`%ad`	저자 시각 (형식은 –-date=옵션 참고)
`%ar`	저자 상대적 시각
`%cn`	커미터 이름
`%ce`	커미터 메일
`%cd`	커미터 시각
`%cr`	커미터 상대적 시각
`%s`	요약

커밋 히스토리 그래프 보기 예제

`oneline` 옵션과 `format` 옵션은 `--graph` 옵션과 함께 사용할 때 더 효과적이다. 이 명령은 브랜치와 머지 히스토리를 보여주는 아스키 그래프를 출력한다. SpringBoot 오픈소스 프로젝트를 clone 받아 살펴보자.

```
$ git clone https://github.com/spring-projects/spring-boot
$ cd spring-boot
```

```
$ git log --pretty=format:"%h %s" --graph
* b91f814e42 Fix incomplete assertions
*   7828fbfdab Merge branch '3.0.x'
|\
| *   d4ef5dc151 Merge branch '2.7.x' into 3.0.x
| |\
| | * 17a4d303f8 Switch Java 20 CI to Bellsoft Liberica
* | | e8e29ecfb8 Merge branch '3.0.x'
|\| |
| * | 1933e9a28e Merge branch '2.7.x' into 3.0.x
| |\|
| | * 04e7e70ebd Remove JDK 19 CI
* | | 718e2ff276 Merge branch '3.0.x'
|\| |
| * | 1da4271f81 Merge branch '2.7.x' into 3.0.x
| |\|
| | * e79ac4b24e Upgrade CI images to ubuntu:jammy-20230308
* | | 54a623f4c9 Merge branch '3.0.x'
|\| |
| * | 042c8e94a9 Merge branch '2.7.x' into 3.0.x
| |\|
| | * 0d82729a29 Upgrade to Gradle Enterprise Gradle plugin 3.12.5
* | |   fd9b8fe020 Merge branch 'gh-34658'
|\ \ \
| * | | 95f45eab1f Create service connections from Testcontainers-managed containers
| * | | 8ec266bea4 Add infrastructure for pluggable connection details factories
|/ / /
* | |   8e4b8a869e Merge branch 'gh-34657'
|\ \ \
| * | | 8721c0e64f Add ConnectionDetail support to Zipkin auto-configuration
```

위 다음 장에서 살펴볼 브랜치나 Merge 결과의 히스토리를 조회할 때 사용된다.