부록 3. Git & Github (feat. Sourcetree)

💡 개발 공부를 하다 보면 git과 github 라는 단어를 거의 100%의 확률로 듣게 됩니다. 과연 이 두 가지가 하는 역할이 무엇이길래 이렇게 끊임없이 나타나는 걸까요?
처음에는 git과 github를 혼동하는 경우가 많습니다. 간단히 말하면 git은 내 로컬 저장소의 소스 코드를 형상관리하기 위한 것이고 github 는 로컬이 아닌 원격환경에 소스코드를 저장할 수 있도록 원격 저장소를 제공하는 서비스 입니다. 저희가 학습했던 Docker Hub 가 github와 유사한 역할을 하는 것이라고 볼 수 있습니다. 자세한 내용은 하단 설명에서 이미지와 함께 다루겠습니다.

💡 **Git vs SVN(Subversion)**
git과 svn은 버전 별로 소스를 관리해주는 **형상관리 시스템(Software Configuration Management)**입니다. 두 시스템은 유사하면서도 확연히 다른 점이 있습니다.
svn은 중앙저장소 방식으로 운용됩니다. 여러 개발자가 작성한 소스코드를 쉽게 한 곳에 모을 수 있기 때문에 편리하고 아직도 사용하는 곳이 많습니다. 하지만 commit한 소스가 중앙저장소에 즉시 반영되어 다른 개발자의 작업에 영향을 미칠 수 있다는 치명적인 단점이 있습니다. Conflict를 방지하기 어려운 것이죠. 게다가 중앙서버에서 모든 소스를 관리하기 때문에 서버에 이상이 생기면 소스 코드 관리에도 중대한 이슈가 발생하게 됩니다.
svn과 같은 중앙집중형 형상관리 시스템로는 소스 관리가 어렵다고 느낀 리눅스 창시자 리누스 토발즈(Linus Torvalds)는 git이라는 분산형 형상관리 시스템을 고안하게 됩니다. 개발자 각자가 독립된 저장소에서 commit을 하다가 반영이 필요한 때에 이를 서버에 전송할 수 있으며, 소스 관리자가 개발자의 작업 내용을 직접 가져올 수도 있습니다. 작성한 변경사항을 commit 한다고 해서 서버에 바로 반영되는 것이 아니기 때문에 서버의 소스와 개발자의 소스가 서로 달라 발생하는 이슈를 방지할 수 있습니다.

- Git
- Github

- Sourcetree

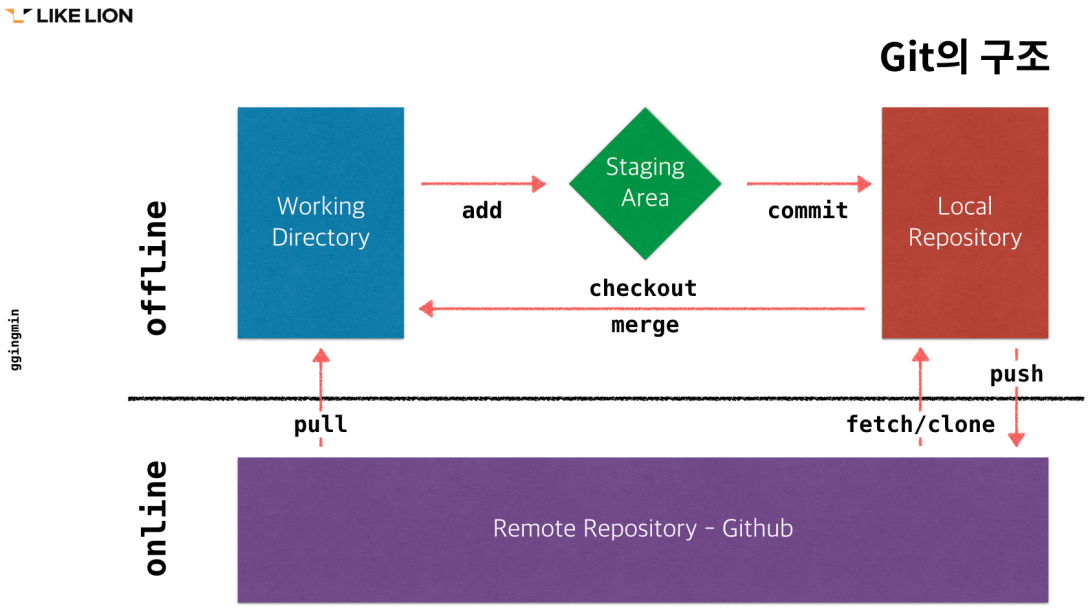
0. git의 구조

git에서 관리하는 영역은 크게 3가지가 있습니다.

- 현재 작업중인 **Working Directory**
- commit 할 파일의 예비 저장소, 혹은 추적 대상 파일의 공간인 **Staging Area**
- 각 유저의 컴퓨터에서 관리되고 있는 **로컬 저장소**

여기서 우리가 직접 눈으로 확인할 수 있는 저장 공간은 Working Directory이며, 현재 프로젝트가 담긴 폴더의 디렉토리라고 생각해도 무방합니다.

많은 사람이 혼란스러워 하는 부분은 바로 **로컬 저장소** 이죠. 내 컴퓨터에 분명이 있는 저장 공간이지만 실제로는 눈에 보이지 않기 때문입니다. 이 로컬 저장소는 우리가 Working Directory로 부터 commit 한 내용들이 스냅샷으로 저장되는 곳입니다. 즉, commit이 된 순간의 파일과 그 내용을 로컬 저장소에서 가지고 있기 때문에 언제든지 commit 했던 지점으로 돌아가는 것이 가능합니다.

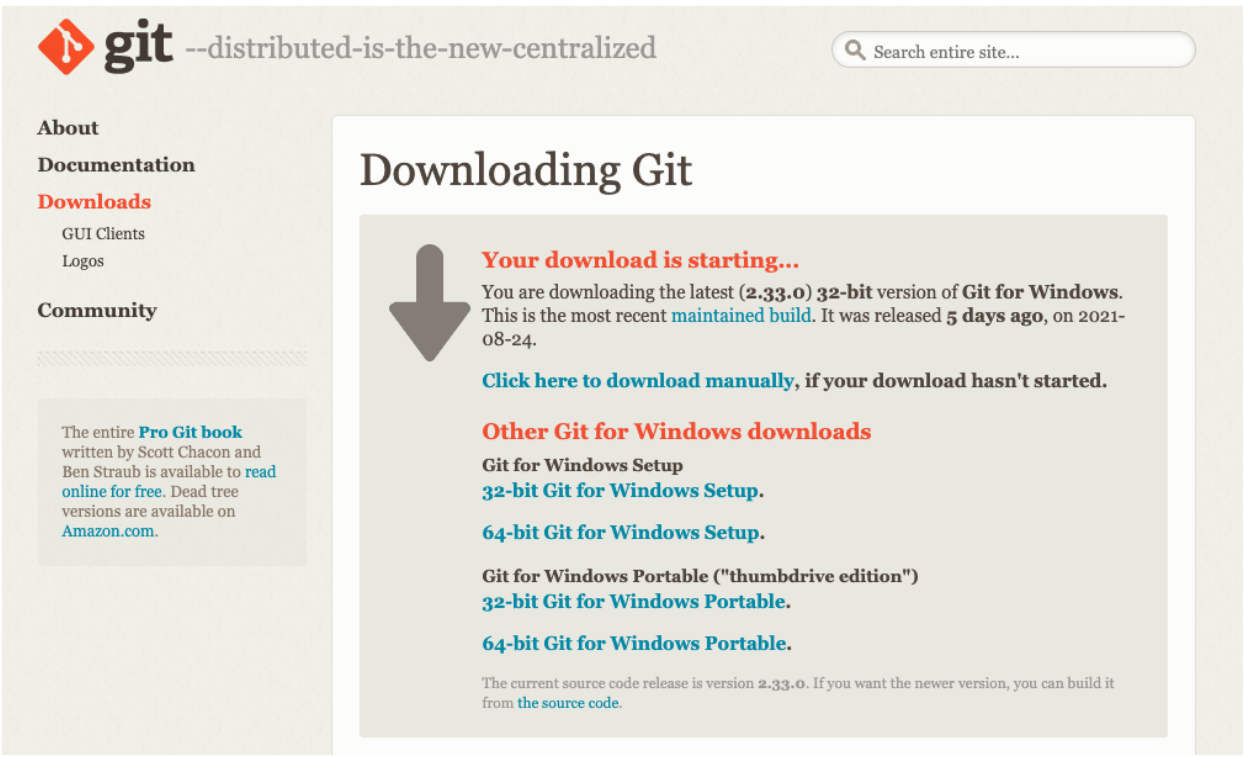


1. git 설치

git은 운영체제별로 설치하는 방법이 상이합니다. 각 운영체제에 맞는 방법을 참고 해주세요. 참고로 2.28 버전부터 기본 체크아웃 브랜치명이 `master` 에서 `main` 으로 변경되었기 때문에 설치 후 반드시 버전 확인 부탁드립니다.

1.1 Windows

1) 윈도우 환경에서는 공식 홈페이지에서 제공하는 인스톨러 파일을 통해 손쉽게 설치할 수 있습니다.



1.2 Mac

1) mac에서는 패키지 관리자인 Homebrew를 이용해서 설치를 진행합니다. 다음의 명령을 실행하여 Homebrew를 설치해줍니다.

```
/bin/bash -c "$(curl -fsSL
https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"
```

2) Homebrew 설치가 완료되면 다음의 명령어를 실행하여 git을 설치합니다.

```
brew install git
```

1.3 Ubuntu Linux

1) Ubuntu에서는 기본 패키지 매니저인 apt를 통해 설치합니다. 다음의 명령을 순차적으로 실행합니다.

```
sudo add-apt-repository ppa:git-core/ppa -y sudo apt-get update -y sudo apt-
get install git -y
```

2. 사용자 등록

git은 이름과 이메일을 등록한 후에 정상적으로 사용이 가능합니다. 하기 명령어를 실행합니다.

```
git config --global user.name "KYEONGMIN CHO" git config --global
user.email"ggingmin@likelion.net"
```

3. 로컬 저장소 초기화

내가 작업하고 있는 프로젝트 디렉토리를 git이 관리하는 영역으로 만들어주기 위해서는 초기화를 해주어야 합니다. 출력되는 메세지는 운영체제 환경에 따라 영문일 수도 있습니다.

```
git init
```

```
→  test git init
/Users/ggingmin/test/.git/ 안의 빈 깃 저장소를 다시 초기화했습니다
```

2) 출력된 경로를 확인하면 `.git` 이라는 폴더가 생성된 것을 확인할 수 있습니다. 해당 저장소와 관련된 메타 정보들이 저장되는 곳입니다.

```
➜  test git:(main) cd /Users/ggingmin/test/.git/
➜  .git git:(main) ls -al
total 24
drwxr-xr-x   9 ggingmin  staff  288  8 29 10:59 .
drwxr-xr-x   3 ggingmin  staff   96  8 29 10:59 ..
-rw-r--r--   1 ggingmin  staff   21  8 29 10:59 HEAD
-rw-r--r--   1 ggingmin  staff  137  8 29 10:59 config
-rw-r--r--   1 ggingmin  staff   73  8 29 10:59 description
drwxr-xr-x  15 ggingmin  staff  480  8 29 10:59 hooks
drwxr-xr-x   3 ggingmin  staff   96  8 29 10:59 info
drwxr-xr-x   4 ggingmin  staff  128  8 29 10:59 objects
drwxr-xr-x   4 ggingmin  staff  128  8 29 10:59 refs
```

4. `.gitignore` 파일을 통한 commit 대상 제외

- `.gitignore` 파일은 말 그대로 git 으로 하여금 특정 파일을 무시하여 commit에서 제외시키는 역할을 합니다. 보통 저장소의 최상위 디렉토리에 위치하고 있습니다.
- `.gitignore` 에서 관리하는 파일은 보통 OS상에서 자동으로 생성하는 파일이나, 보안상 공개되어서는 안되는 파일, 공동 작업에 있어서 불필요한 것들로 이루어져 있습니다. ssh key, IAM pem 등은 절대 commit 되지 않도록 주의해야 합니다.
- 통합 개발 환경에 따라 각 프레임워크에서 꼭 제외되어야 하는 파일들을 자동으로 설정해주기도 합니다.
- 아래 예시는 Spring Boot 프로젝트에서 제외하고자 하는 파일을 작성한 파일입니다.

```
HELP.md .gradle build/ !gradle/wrapper/gradle-wrapper.jar
!**/src/main/**/build/ !**/src/test/**/build/ ### STS ### .apt_generated
.classpath .factorypath .project .settings .springBeans .sts4-cache bin/
!**/src/main/**/bin/ !**/src/test/**/bin/ ### IntelliJ IDEA ### .idea *.iws
*.iml *.ipr out/ !**/src/main/**/out/ !**/src/test/**/out/ ### NetBeans ###
/nbproject/private/ /nbbuild/ /dist/ /nbdist/ /.nb-gradle/ ### VS Code ###
.vscode/
```

5. 저장소 상태 확인

1) 위에서 초기화 했던 영역에서 다음의 명령을 실행하면 아무 파일도 존재하지 않기 때문에 `커밋할 사항이 없음` 이라는 메세지가 출력됩니다.

```
git status
```

```
➜  test git:(main) ✗ git status
현재 브랜치 main

아직 커밋이 없습니다

커밋할 사항 없음 (파일을 만들거나 복사하고 "git add"를 사용하면 추적합니다)
```

2) 동일한 디렉토리 내에 `index.html` 파일을 하나 만들어 주고 다시 명령을 실행합니다. 새로운 파일이 `추적하지 않는 파일` 이라고 메세지가 뜹니다. 여기서 **추적하지 않는다**는 것은 Staging Area에 아직 소스코드가 올라가지 않았다는 뜻입니다.

```
➜  test git:(main) git status
현재 브랜치 main

아직 커밋이 없습니다

추적하지 않는 파일:
  (커밋할 사항에 포함하려면 "git add <파일>..."을 사용하십시오)
        index.html

커밋할 사항을 추가하지 않았지만 추적하지 않는 파일이 있습니다 (추적하려면 "git
add"를 사용하십시오)
```

6. 파일 추적하기

작성한 소스를 추적하기 위한 명령어를 실행해보겠습니다. 이어서 `git status` 명령어를 실행하면 색깔이 바뀌어 있고 **커밋할 변경 사항**으로 설정된 것을 확인할 수 있습니다.

```
git add <file name> // 특정 파일을 추적하고 싶을때 git add -A // 프로젝트 전체를 추적하고 싶을때
```

```
➜  test git:(main) ✗ git add -A
➜  test git:(main) ✗ git status
현재 브랜치 main

아직 커밋이 없습니다

커밋할 변경 사항:
  (스테이지 해제하려면 "git rm --cached <파일>..."을 사용하십시오)
        새 파일:       index.html
```

7. 추적 해제하기

추적하고 있는 파일 중 commit 대상에서 제외가 필요한 것은 다음의 명령어로 해제가 가능합니다. 추적이 해제가 되면 `git add` 명령어를 실행하기 이전으로 되돌아 갑니다.

```
git rm --cached <file name>
```

```
➜  test git:(main) ✗ git rm --cached index.html
rm 'index.html'
➜  test git:(main) ✗ git status
현재 브랜치 main

아직 커밋이 없습니다

추적하지 않는 파일:
  (커밋할 사항에 포함하려면 "git add <파일>..."을 사용하십시오)
        index.html

커밋할 사항을 추가하지 않았지만 추적하지 않는 파일이 있습니다 (추적하려면 "git
add"를 사용하십시오)
```

8. 로컬 저장소에 소스 반영하기

추적했던 파일을 로컬 저장소에 반영하기 위한, git에서 가장 중요한 명령어를 실행해보겠습니다

```
git commit -m "<message>" // <message> 에 반드시 commit 메세지를 적어주어야 합니다. // -m 옵션을 사용하면 commit 명령과 동시에 메세지를 입력할 수 있습니다. git commit -a -m "<message>" // -a 옵션을 추가하면 이미 추적 중인 파일에서 변동 사항이 있을때 굳이 다시 add를 거치지 않고 commit을 할 수 있습니다. // 즉, add를 생략하게 해주는 옵션입니다. // 주의할 점은 이미 Staging Area에 올라간 파일에 대해서 생략이 가능한 것이지, 한번도 add가 되지 않았던 파일에 대해서는 해당사항이 없습니다.
```

```
➜  test git:(main) ✗ git add -A
➜  test git:(main) ✗ git commit -m "commit"
[main (최상위-커밋) 561173a] commit
 1 file changed, 13 insertions(+)
 create mode 100644 index.html
```

9. 작업 로그 확인하기

작성한 소스를 누가, 언제 commit 했는지 확인이 필요한 경우 로그를 조회할 수 있습니다. 로그 창에서 빠져나오기 위해서는 `:` 을 누르고 `q` 를 누르면 됩니다.

`--oneline` 옵션을 적용하면 로그를 건 당 한 줄로 표현할 수 있습니다.

```
git log
```

```
commit 561173a35984a120f7168b6895a438a8462fee0f (HEAD -> main)
Author: KYEONGMIN CHO <[redacted]>
Date:   Sun Aug 29 11:31:13 2021 +0900

    commit
(END)
```

```
git log --oneline
```

```
561173a (HEAD -> main) commit
(END)
```

10. 소스 변동 사항 확인

최종 commit 버전의 소스와 작업이 이루어 지고 있는 순간의 소스 비교해야 할 때가 있습니다. 만약 commit 이후 소스가 변경 되었다면 다음과 같은 내역을 확인할 수 있습니다. 이렇게 변동사항이 확인되면 새로 commit 을 하거나 commit 된 버전에 맞게 소스를 수정해야 합니다.

```
git diff
```

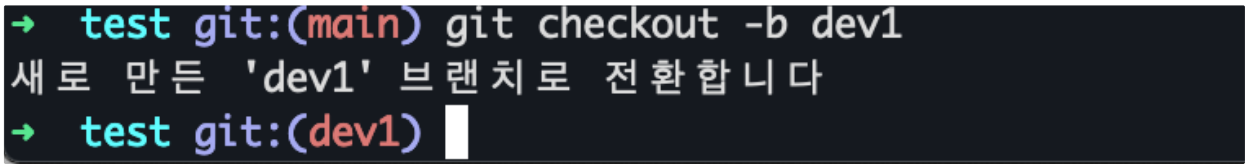
```
diff --git a/index.html b/index.html
index eb320ce..25aeb3b 100644
--- a/index.html
+++ b/index.html
@@ -7,7 +7,7 @@
 </head>

 <body>
-    <h1>Hello, Git!</h1>
+    <h1>Hello, Git!!</h1>
 </body>

 </html>
(END)
```

11. 브랜치 설정하기

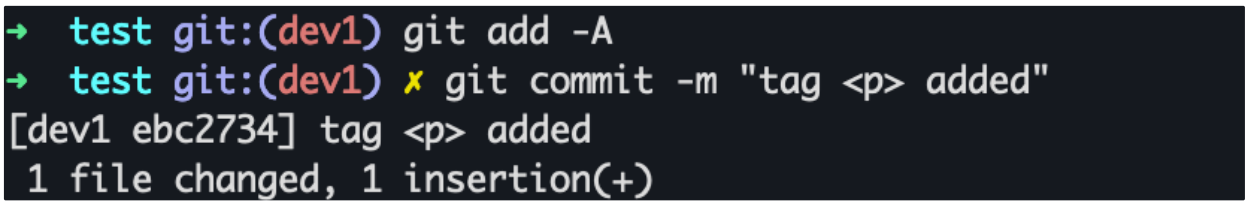
A, B, C 라는 기능 3가지를 구현한다고 생각해봅시다. 만약 `main` 브랜치에서 모든 진행을 하게 된다면 협업은 물론이거니와 1인 작업이어도 효율적으로 작업을 하기가 힘듭니다. 소스 commit도 어지럽게 엉키고 버그가 발생해도 찾기가 굉장히 어렵게 되죠. 브랜치라는 기능은 여기서 빛을 발합니다. A라는 기능을 위한 브랜치, B를 위한 브랜치 등을 따로 설정합니다. 이들 브랜치에서는 병렬적으로 작업이 가능하며, 나중에 소스 작성이 완료된 이후 `main` 브랜치로 병합하게 됩니다.

1) `dev1` 라는 브랜치를 새로 생성함과 동시에 그곳으로 브랜치를 이동하는 명령을 실행해보겠습니다.

```
git checkout -b dev1
```



2) 변경된 브랜치 상태에서 `index.html` 파일을 일부 변경한 후에 추적 및 commit 을 수행합니다.



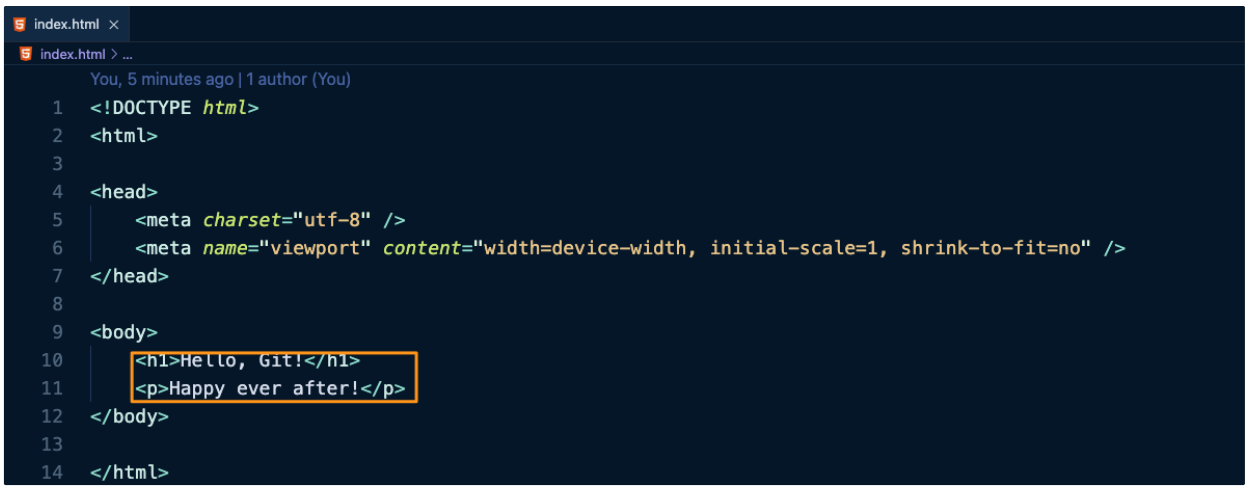
3) 다시 `main` 브랜치로 돌아간 후 소스에 어떤 변경이 있는지 확인합니다. `dev1` 브랜치에서 작성했던 소스가 사려졌다면 정상적으로 `main` 브랜치로 이동된 것입니다.

```
git checkout main
```

12. 브랜치 병합하기

브랜치 별로 작업이 완료되면 소스를 하나로 병합해주어야 합니다. `main` 브랜치로 돌아간 상태에서 `dev1` 의 작업내용을 합치기 위해 다음의 명령을 실행합니다. `dev1` 에서 작성했던 소스가 나오면 정상입니다.

```
git merge dev1
```



13. 원격 저장소 설정

여기까지 로컬 저장소에서 소스를 다루어보았습니다. 이렇게 작업이 완료된 소스를 github의 원격 저장소에 올려보겠습니다. 먼저 github 저장소 주소를 등록해야 합니다.

사전에 github 저장소를 새로 만들어주고 명령을 실행해줍니다. 저는 test라는 이름의 저장소를 만들었습니다.

```
git remote add origin https://github.com/ggingmin/test.git // git remote add [원격 저장소명] [원격 저장소 주소]
```

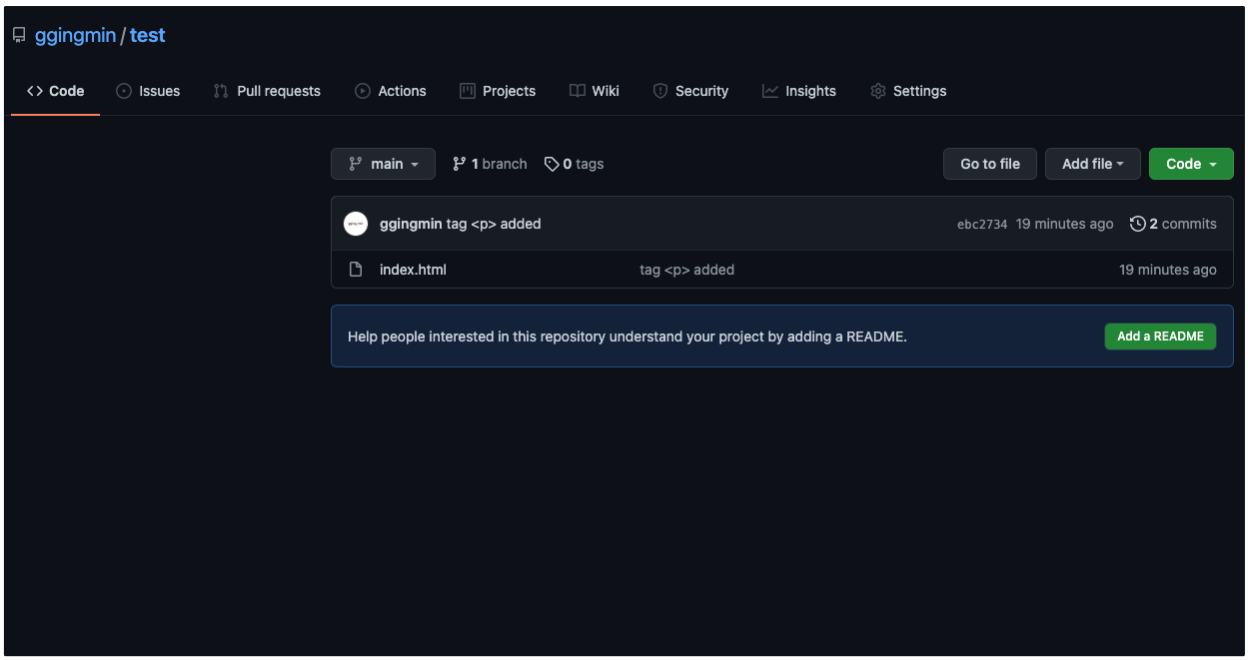
```
➜  test git:(main) git remote add origin https://github.com/ggingmin/test.git
➜  test git:(main) git remote
origin
```

14. 원격 저장소에 프로젝트 업로드

원격 저장소 설정이 완료 되었다면 이제 업로드를 진행해보겠습니다.

```
git push -u origin main // git push -u [원격 저장소 명] [브랜치명] // -u 옵션을 사용하면 추후 동일한 원격 저장소와 브랜치에 push 할 때 git push 만 실행할 수 있습니다.
```

```
➜  test git:(main) git push -u origin main
오브젝트 나열하는 중: 6, 완료.
오브젝트 개수 세는 중: 100% (6/6), 완료.
Delta compression using up to 8 threads
오브젝트 압축하는 중: 100% (4/4), 완료.
오브젝트 쓰는 중: 100% (6/6), 632 bytes | 632.00 KiB/s, 완료.
Total 6 (delta 1), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (1/1), done.
To https://github.com/ggingmin/test.git
 * [new branch]      main -> main
'main' 브랜치가 리모트의 'main' 브랜치를 ('origin'에서) 따라가도록 설정되었습니다.
```



15. 원격 저장소 프로젝트를 로컬로 가져오기

로컬에 원격 저장소 프로젝트를 가져오는 방법은 두 가지가 있고 많은 사람들이 헷갈려합니다. 두 명령의 차이점을 확인해보겠습니다.

- `git fetch`
  원격 저장소의 내용을 로컬 저장소로 받아오기는 합니다. 하지만 로컬 저장소는 내 컴퓨터에 저장이 되어 있을 뿐이지 우리가 우리가 실제로 작업하고 있는 Working Directory가 아닙니다. 즉, `fetch` 만으로는 직접 작업을 하거나 수정을 할 수가 없습니다.
- `git pull`
  원격 저장소의 소스를 받아올 뿐만 아니라 Working Directory 병합까지 수행합니다. 순서대로 정리 하면 원격 저장소에서 받은 내용이 로컬 저장소에, 로컬 저장소의 내용이 다시 Working Directory 까지 도달하는 것입니다.

상기 명령어를 사용하기 위해서는 현재 위치한 경로가 git 영역으로 초기화된 상태여야 합니다.

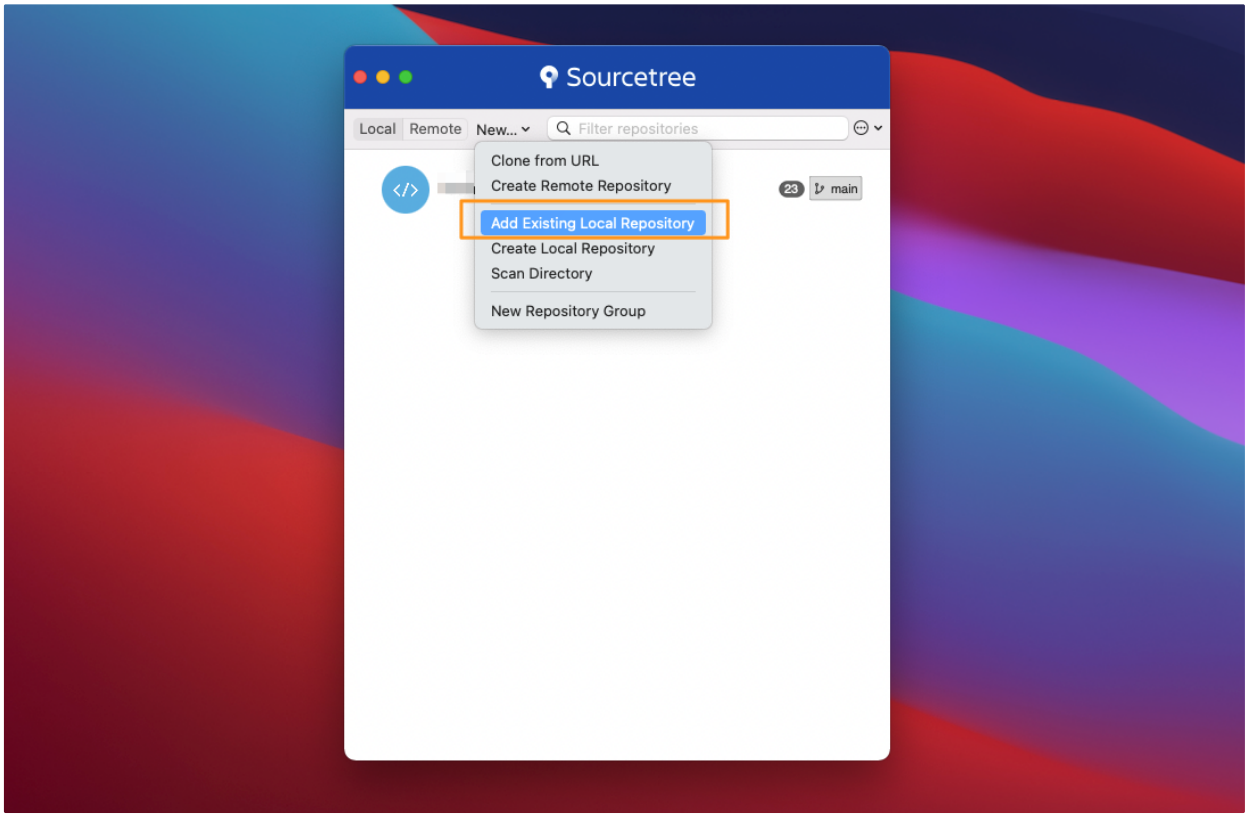
```
➜  project git:(main) git fetch https://github.com/ggingmin/test.git
remote: Enumerating objects: 6, done.
remote: Counting objects: 100% (6/6), done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 6 (delta 1), reused 6 (delta 1), pack-reused 0
오브젝트 묶음 푸는 중: 100% (6/6), 612 bytes | 102.00 KiB/s, 완료.
https://github.com/ggingmin/test URL에서
 * branch            HEAD       -> FETCH_HEAD
➜  project git:(main) ll
➜  project git:(main) ls -al
total 0
drwxr-xr-x   3 ggingmin  staff    96  8 29 12:13 .
drwxr-xr-x+ 61 ggingmin  staff  1952  8 29 12:14 ..
drwxr-xr-x  10 ggingmin  staff   320  8 29 12:13 .git
```

```
➜  project git:(main) git pull https://github.com/ggingmin/test.git
https://github.com/ggingmin/test URL에서
 * branch            HEAD       -> FETCH_HEAD
➜  project git:(main) ls -al
total 8
drwxr-xr-x   4 ggingmin  staff   128  8 29 12:14 .
drwxr-xr-x+ 61 ggingmin  staff  1952  8 29 12:14 ..
drwxr-xr-x  12 ggingmin  staff   384  8 29 12:14 .git
-rw-r--r--   1 ggingmin  staff   240  8 29 12:14 index.html
```

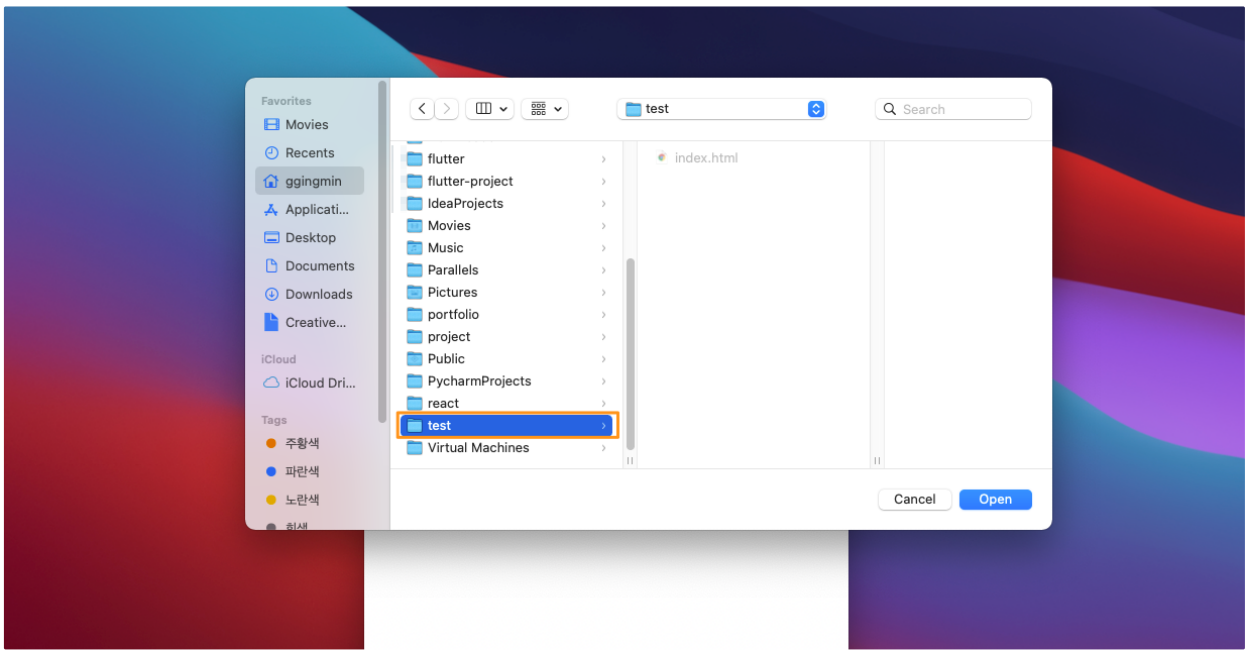
16. Sourcetree 로 git 관리하기

Sourcetree는 협업관리 툴 Jira로 널리 알려진 Atlassian 社의 git GUI 툴이며, 좀 더 보기 편하게 저장소를 관리할 수 있습니다. Windows와 Mac 을 모두 지원합니다.

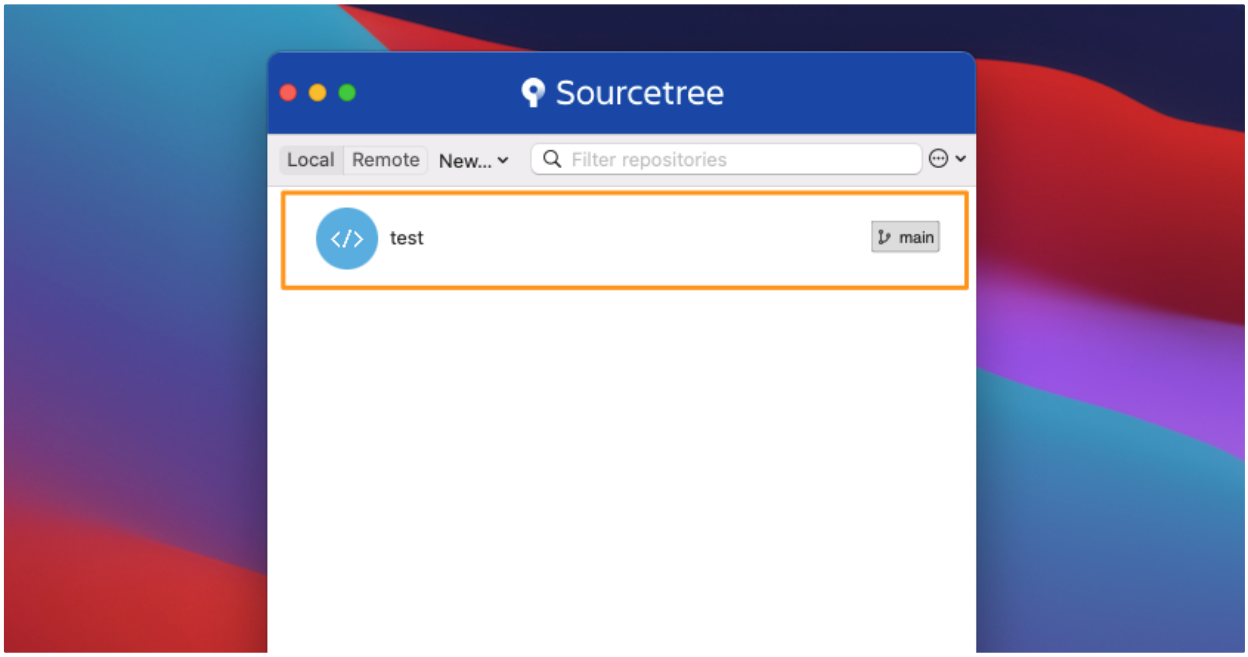
1) 메인 화면에서 **Add Existing Local Repository** 를 클릭합니다.



2) 작업 중인 디렉토리를 선택합니다. 이 디렉토리는 git 초기화가 되어있어야 합니다.



3) 정상적으로 추가가 되었습니다.



4) 해당 프로젝트를 클릭하면 다음과 같이 CLI 명령을 통해 진행했던 것들을 대부분 이용할 수 있는 대시보드가 뜹니다.