Git

수업 과제를 제출했었을 때, 다들 어떻게 파일을 관리하셨었나요?

```
제출파일.hwp
제출파일-최종.hwp
제출파일-최최종.hwp
제출파일-최최최최종 진짜 마지막.hwp
...
```

위와 같은 상황이 발생하였을때, 마주하게 되는 문제점들이 있습니다.

1. 팀원과 동시에 작업하면 매번 업데이트 되는데 매번 또는 시간 단위로 업데이트 되는 파일을 팀원과 어떻게 공유할 것인가
2. 팀원 중 누군가 잘 못 작업하여 특정부분을 날리거나 업데이트 하지 못할 경우를 대비하여 백업은 언제 할 것인가
3. 팀원이 어떤 부분을 제작하였고, 또 다른 부분을 만들어야하는가
4. 문제가 발생했을때 어디서 발생했는가

일반적으로는 회사나 팀에서는 같은 LAN이나 FTP 서버 , 클라우드 서비스 등을 이용하여 팀원과 문서를 공유하면서 버전관리를 할 것입니다.

하지만 멀리 떨어지거나 보안, 비용적인 측면으로 비효율이 발생.

이를 해결하기 위해 버전관리 시스템이 등장하였습니다.

버전 관리란?

버전관리 시스템은 파일변화를 시간에 따라 기록했다가 나중에 특정시점의 버전을 다시 꺼내올 수 있는 시스템

- 각 파일을 이전 상태로 되돌릴 수 있음

- 프로젝트를 통째로 이전 상태로 되돌릴 수 있음
- 시간에 따라 수정 내용을 비교해 볼 수 있음
- 누가 문제를 일으켰는지도 추적할 수 있음
- 누가 언제 만들어 낸 이슈인지도 알수있음
- 파일을 잃어버리거나 잘못 고쳤을 때도 쉽게 복구할 수 있음

우리가 레포트를 제출한다고 가정했을때, 처음에 저장했을때 '제출파일.hwp'라고 저장을 했다가 수정을 하면서 '제출파일-최종'로 저장하고 또 수정을 거치면서 '제출파일-최최종'...로 수정을 하게 되는 경험이 있었을 것입니다. 여기서 이 파일들을 **복사, 백업, 저장** 등을 한것이고, 이러한 것을 **버전 관리**라고 합니다.

Git

프로그램 등의 소스 코드 관리를 위한 **분산 버전 관리 시스템**

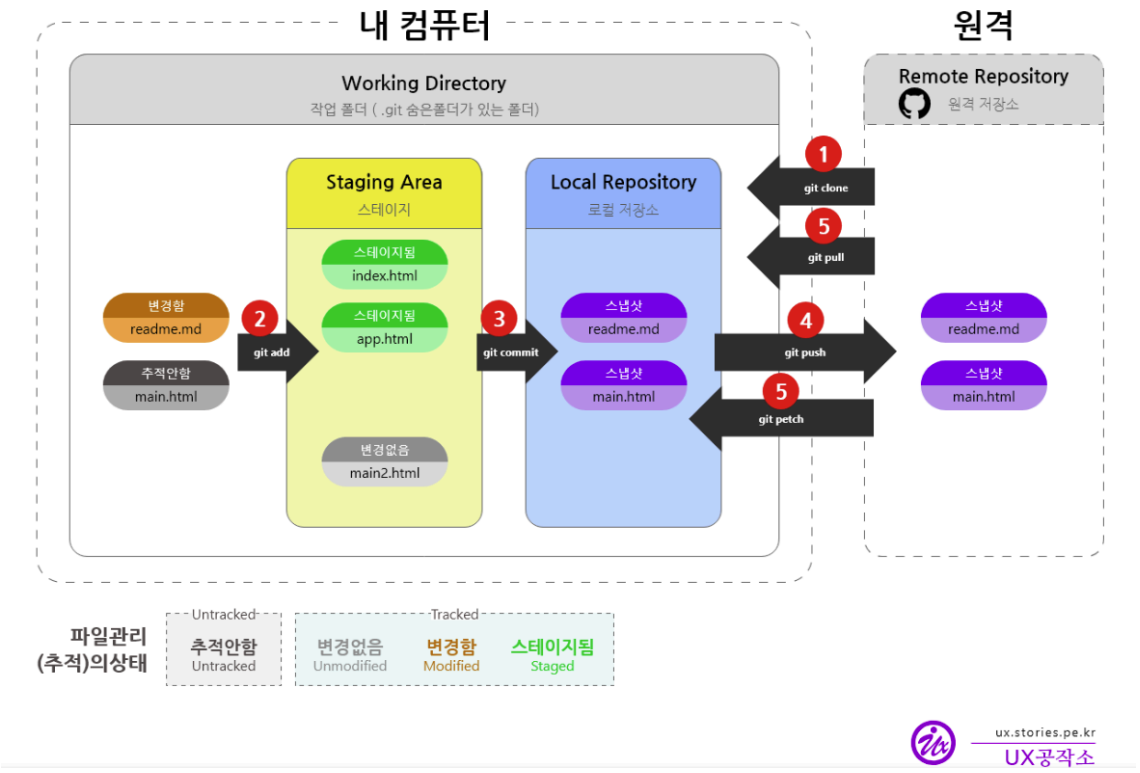
여기서 분산은 여러 명, 다양한 사람들에 의해 프로젝트를 관리한다는 의미에서의 분산

GIT 장점

소스 코드를 주고 받을 필요 없이, 같은 파일을 여러 명이 동시에 작업하는 병렬 개발이 가능.

- 즉 브랜치를 통해 개발한 뒤, 본 프로그램에 합치는 방식(Merge)으로 개발 진행가능
- 분산 버전관리 방식 이기에 인터넷이 연결되지 않는 곳에서도 개발을 진행할 수 있으며, 중앙 저장소가 날아가도 다시 원상복귀 가능 (로컬 저장소가 따로 있기에)
- 프로젝트를 GIT을 통해 관리하면 체계적인 개발이 가능해지고, 프로그램이나 패치를 배포하는 과정도 간단(Pull을 통한 업데이트, Patch을 통한 파일 배포)

Git 동작원리



Repository

스테이지에서 대기하고 있던 파일들을 버전으로 만들어 저장하는 곳

Repo는 두 가지 저장소가 존재.

원격 저장소(Github)와 로컬 저장소 두 종류의 저장소를 제공함.

원격 저장소(Remote Repository)

파일이 원격 저장소 전용 서버에서 관리되며 여러 사람이 함께 공유하기 위한 저장소

로컬 저장소(Local Repository)

내 PC에 저장되는 개인 전용 저장소

해당 저장소를 만드는 방법은 두 가지가 존재.

아예 Git Init 명령어를 통해 새로 만들거나,

이미 만들어져 있는 원격 저장소를 로컬 저장소로 복사해 올 수 있음.

Working Directory(Working Tree)

저장소를 어느 한 시점을 바라보는 작업자의 현재 시점.

파일 수정, 저장 등의 작업을 하는 디렉토리

SnapShot

특정 시점에서 파일, 폴더 또는 워크스페이스의 상태를 의미.

스냅샷을 통해 특정 시점에 어떤 파일에 어떤 내용이 기록되어 있었는지, 폴더 구조는 어떠했는지, 어떤 파일이 존재했는지 등 저장소의 모든정보를 확인할 수 있다.

Git에서는 새로운 버전을 기록하기 위한 명령인 커밋(commit)을 실행하면 스냅샷이 저장된다

1. 개발자는 작업 디렉토리에 있는 파일을 수정
2. 수정된 파일을 모아 정리하여 만든 Snapshot을 Staging 디렉토리에 추가하고 저장
3. GIT 디렉토리에 저장 (Staging 디렉토리에 저장된 파일을 앞으로 영구 불변의 상태를 유지하는 Snapshot 으로서 git 디렉토리에 저장하는 것)

Checkout

이전 버전 작업을 불러오는 것.

(브랜치를 갈아타는 것)

Staging Area

저장소에 커밋하기 전 커밋을 준비하는 위치.

Commit

현재 변경된 작업 상태를 점검을 마치면 확정하고 저장소에 저장하는 작업.

Branch

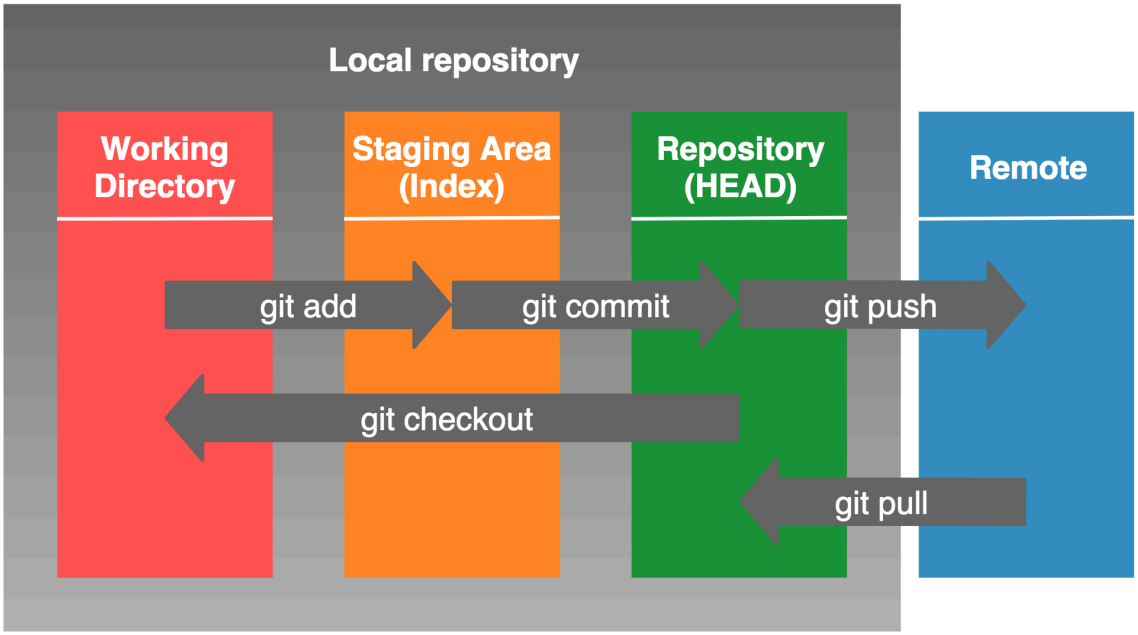
가지 또는 분기점.

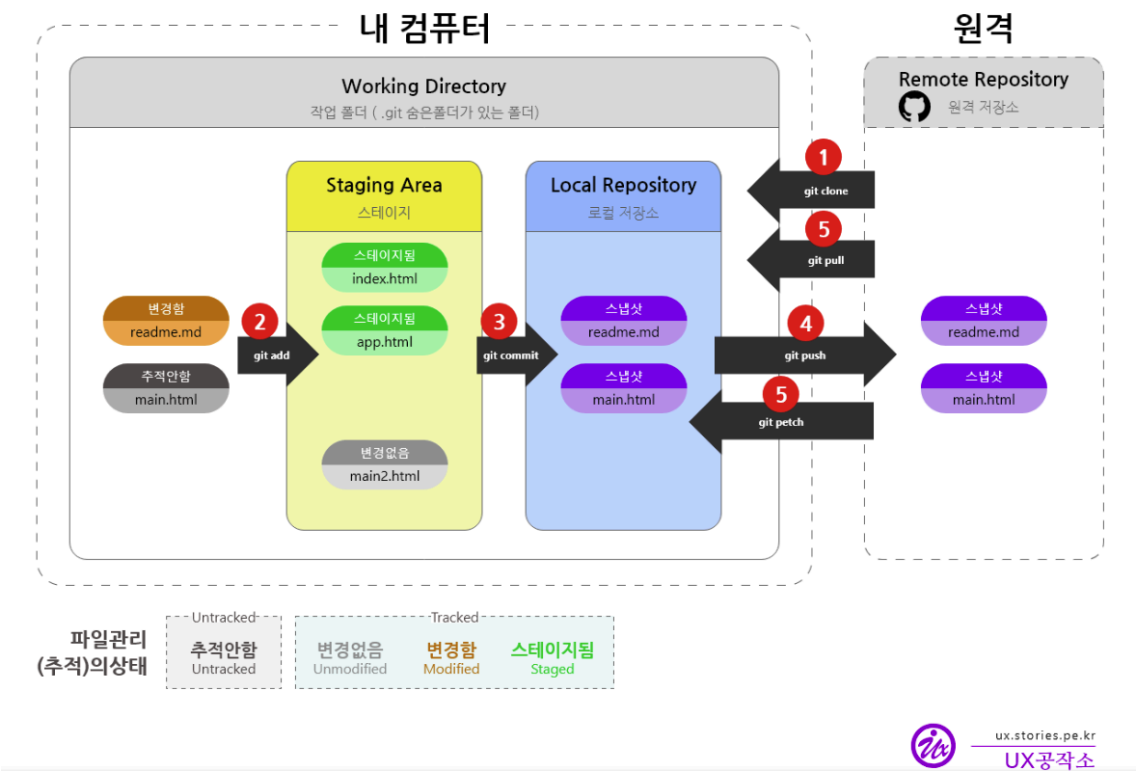
HEAD

현재 작업중인 Branch

Merge

다른 Branch의 내용을 현재 Branch로 가져와 합치는 작업을 의미


Local repository
Working Directory
Staging Area (Index)
Repository (HEAD)
Remote
git add
git commit
git push
git checkout
git pull


내 컴퓨터
원격
Working Directory
작업 폴더 (.git 숨은폴더가 있는 폴더)
Staging Area
스테이지
Local Repository
로컬 저장소
Remote Repository
원격 저장소
변경함
readme.md
추적안함
main.html
git add
스테이지됨
index.html
스테이지됨
app.html
변경없음
main2.html
git commit
스냅샷
readme.md
스냅샷
main.html
git push
git clone
git pull
git petch
스냅샷
readme.md
스냅샷
main.html
파일관리
(추적)의상태
Untracked
추적안함
Untracked
Tracked
변경없음
Unmodified
변경함
Modified
스테이지됨
Staged
ux.stories.pe.kr
UX공작소

Git 기초 명령어

- `git help` : **도움말** 기능(가장 많이 사용하는 21개의 깃 명령어 출력).

  사용법이 궁금한 명령어에 대해 'git help [궁금한 명령어]'를 타이핑시, 해당 깃 명령어의 설정과 사용에 대한 도움말 출력.

- `git init` : 깃 저장소를 **초기화**.

  저장소나 디렉토리 안에서 이 명령을 실행하기 전까지는 그냥 일반 폴더이다. **이 명령어를 입력한 후에야 추가적인 깃 명령어 입력 가능**.

- `git status` : 저장소 **상태 체크.**

  어떤 파일이 저장소 안에 있는지, 커밋이 필요한 변경사항이 있는지, 현재 저장소의 어떤 브랜치에서 작업하고 있는지 등의 상태정보 출력.

- `git branch` : **새로운 브랜치 생성.**

  여러 협업자와 작업할 시, 이 명령어로 새로운 브랜치를 만들고, 자신만의 변경사항과 파일 추가 등의 커밋 타임라인을 생성, 완성 후 협업자의 branch 혹은 main과 merge.

- `git add` : '**staging 영역'에 변경내용 추가**.

  다음 commit명령 전까지 변경분을 staging 영역에 보관하여 변동내역을 저장

명령어	작용
git add [업로드 하고싶은 파일 혹은 디렉토리 경로]	해당 파일 혹은 디렉토리 변경 내용 staging area 등록
git add .	현재 디렉토리 모든 변경내용 staging area 등록(상위X)
git add -A	작업 디렉토리 모든 변경내용 staging area 등록
git add -p	터미널에서 staging area로 넘길 파일 선택 가능

- `git commit` : **staging area에 있는 변경 내용 묶음 및 정의. (**깃의 가장 중요한 명령어)

  cf) git commit -m [커밋 메시지] : staging area에 있는 내용은 "커밋 메시지"를 반영한 수정본 파일의 묶음임.

- `git log` : 커밋 내역 확인
- `git push` : 로컬 컴퓨터에서 서버로 변경사항을 “push”
- `git pull` : 서버 저장소로부터 최신 버전을 "pull"
  (서버 저장소의 데이터를 가져와, 현재 브랜치와 merge)

  cf) 작업 도중 기존 작업 내용은 유지하면서, 최신 코드로 업테이트 할 때 사용.

- `git clone` : 서버 저장소의 데이터를 로컬 컴퓨터로 복사.
  (서버 저장소의 데이터를 그대로 가져옴. 작업중이던 내역 있을시 덮어쓰기 됨)

  cf) 프로젝트에 처음 투입 될 때 사용

- `git checkout` : **작업하기 원하는 브랜치로 이동**.

  cf1) git checkout Yana : Yana 브렌치로 이동.
  cf2) git checkout -b 야나 : '야나' 라는 브랜지를 생성 후 야나 브렌치로 이동(생성과 이동 동시에)

- `git merge` : **개별 branch에서 마친 작업을 master branch로 병합.**

GITHUB

GIT을 이용하여 자신의 지역 저장소를 만들고 파일, 코드 등을 관리하는 작업을 하였다면

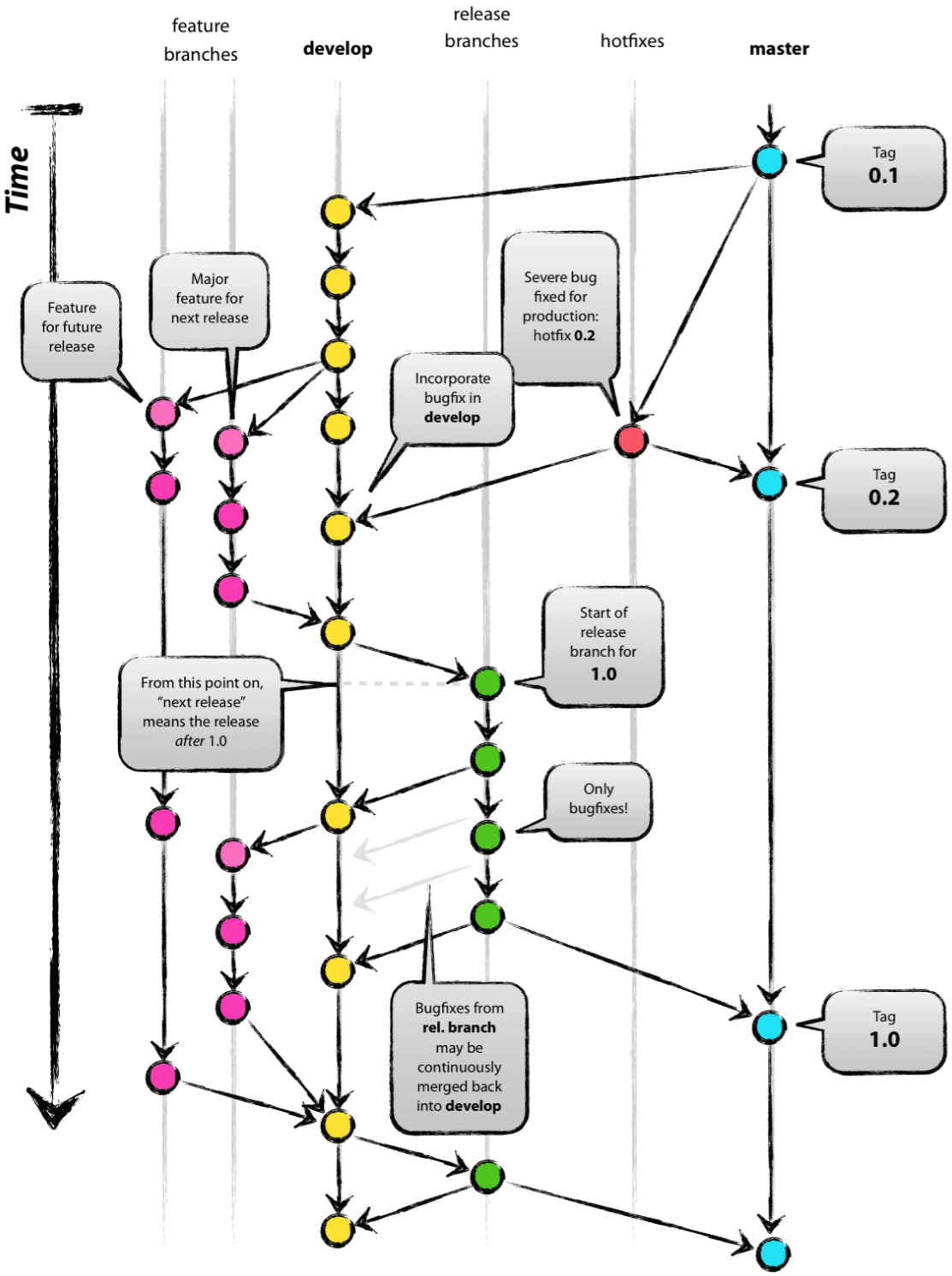
GITHUB는 내가 로컬에서 git으로 관리한 자료를 다른 사람과 공유하거나 백업해 둘 수 있는 웹사이트.

내가 작업한 자료 뿐만 아니라 다른 사람이 Github에 올린 자료를 복제해 올 수 있고 어떤 커밋을 했는지 어떤 소스코드를 사용하고 있는지 확인하고 그 자료에 새롭게 반영할 수도 있음.

GIT - local 내에서 소스코드 버전관리

GITHUB - local에서 관리한 소스코드를 업로드하고 공유할 수 있는 공간

GIT Flow



1. feature

feature 브랜치는 기능의 구현을 담당

예를 들어, `feature/login`은 login 기능을 구현하는 브랜치임을 알 수 있다. feature 브랜치는 develop 브랜치에서 생성되며, develop 브랜치로 머지. 머지된 후에는 해당 브랜치가 삭제

2. develop

develop 브랜치는 말 그대로 개발을 진행하는 브랜치

하나의 feature 브랜치가 머지될 때마다 develop 브랜치에 해당 기능이 더해지며 살을 붙여간다. develop 브랜치는 배포할 수준의 기능을 갖추면 release 브랜치로 머지

3. release

release 브랜치는 개발된 내용을 배포하기 위해 준비하는 브랜치

release 브랜치에서 충분한 테스트를 통해 버그를 검사하고 수정해 배포할 준비가 완전히 되었다고 판단되면 **master로 머지해 배포**. release 브랜치는 develop 브랜치에서 생성되며 버그 수정 내용을 develop 브랜치에도 반영하고, 최종적으로 master 브랜치에 머지

4. hotfix

hotfix 브랜치는 배포된 소스에서 버그가 발생하면 생성되는 브랜치

release 브랜치를 거쳐 한차례 버그 검사를 했지만 예상치 못하게 배포 후에 발견된 버그들에 대해서 수정

5. master

master 브랜치는 최종적으로 배포되는 가장 중심의 브랜치

develop 브랜치에서는 개발이 진행되는 와중에도 이전 release 브랜치 내용이 master에 있어 배포되어 있음

이 외에도 github flow, git-lab flow 등등 각 프로젝트마다 flow는 다를 수 있음.

git 브랜치 전략에 대해서

우리는 개발을 진행하면서 우리가 구현한 소스코드를 git이라는 버전 관리 시스템을 통해 관리한다. git을 사용하지 않았더라면 협업을 진행하면서 메일이나 USB로 소스코드를 주고받아야 했을 것이다. git을 사용함으로써 우리는 시시각각 코드를 전송할 수 있다. 게다가 일일이 병합

https://tecoble.techcourse.co.kr/post/2021-07-15-git-branch/

[GIT] ⚡ git 개념 & 원리 (그림으로 알기쉽게 비유 😃)

Git 이란 무엇인가? Git이란 분산형 버전 관리 시스템(Version Control System) 의 한 종류이며, 빠른 수행 속도에 중점을 둔다. 우리가 레포트를 제출한다고 가정했을때, 처음에 저장했을때 'report.txt'라고 저장을 했다가 수정을 하면서 'report_final.txt'로 저장하고 또 수정을

https://inpa.tistory.com/entry/GIT-⚡-개념-원리-쉽게이해

▼ Git branch 적용

Branch를 생성하기 전 Issue를 먼저 작성한다.

Issue 작성 후 생성되는 번호와 Issue의 간략한 설명 등을 조합하여 Branch의 이름을 결정한다.

- `main` : 개발이 완료된 산출물이 저장될 공간
- `develop` : feature 브랜치에서 구현된 기능들이 merge될 브랜치
- `release` : 릴리즈를 준비하는 브랜치, 릴리즈 직전 QA 기간에 사용한다
- `feature` : 기능을 개발하는 브랜치, 이슈별/작업별로 브랜치를 생성하여 기능을 개발한다
- `hotfix` : 정말 급하게, 데모데이 직전에 에러가 난 경우 사용하는 브랜치
- `bug` : 버그를 수정하는 브랜치

커밋 메시지 컨벤션 Commit Message Convention

1. **Commit Message Structure**

- 기본적인 커밋 메시지 구조 (각 파트는 빈줄로 구분한다.)

```
제목 (Type: Subject)
(한줄 띄어 분리)
본문 (Body)
(한줄 띄어 분리)
꼬리말 (Footer)
```

2. **Commit Type**

- `Feat` : 새로운 기능 구현
- `Docs` : 문서 수정 (README나 WIKI 등의 문서 개정)
- `Add` : 새로운 파일/라이브러리 추가 or Feat 이외의 부수적인 코드 추가.
- `Fix` : 버그, 오류 수정
- `Style` : 코드 포맷팅, 코드 변경이 없는 경우
- `Del` : 쓸모없는 라인/코드 삭제
- `MOVE` : 프로젝트 내 파일이나 코드의 이동
- `Remove` : 파일을 삭제하는 작업만 수행한 경우
- `Refactor` : 코드 리펙토링
- `Merge` : 브랜치를 합칠때 사용
- `Rename` : 파일 이름 변경
- `!HOTFIX` : 급하게 치명적인 버그를 고쳐야하는 경우
- `test` : 테스트 코드, 리펙토링 테스트 코드 추가
- `chore` : 빌드 업무 수정, 패키지 매니저 수정
- `Design` : UI/Storyboard 수정한 경우

3. **Subject**

- 제목은 **50글자** 이내
- ⭐첫 글자는 대문자로 작성한다.
- 마침표 및 특수기호는 사용하지 않는다.
- 간결하고 요점적 작성.

  ex)
  Fixed → Fix
  Added → Add
  Modified → Modify

4. **Body**

- 어떤 작업을 했는지에 대해 기술
- 최대한 상세히 작성 (명확할수록 좋음)
- 어떻게 변경했는지보다 무엇을, 왜 변경했는지 작성

5. Example

```
Feat: 회원 가입 기능 구현

- SMS, 이메일 중복확인 API 개발
```

```
Docs: README 수정

- 프로젝트 소개 추가
- 기능 설명
- 사진 추가
```

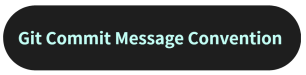
```
Add: 뷰 컨트롤러/스위프트 파일 추가, main 부분 코드 수정

- main 랜덤 숫자 추출 코드 수정
- 뷰 컨트롤러 파일 추가(파일명: ~~~.swift)
```

Git Flow 적용

1. feature Branch를 생성한다.
2. Add - Commit - Push - Pull Request 의 과정을 거친다.
3. Pull Request가 작성되면 작성자 이외의 다른 팀원이 Code Review를 한다.
4. Code Review가 완료되면 Pull Request 작성자가 develop Branch로 merge 한다.

▼ 참조

[Git] Commit Message Convention
Git을 협업에 알맞게, 커뮤니케이션에 유용하게, 깔끔한 가독성을 가지도록 사용하기 위해서 좋은 커밋 세미지를 사용하는 것이 중요하다. 그러기 위해서 커밋 컨벤션을 정리하였다.
https://velog.io/@archivvonjang/Git-Commit-Message-Convention



프로젝트에서 github/gitlab 관리 방법

프로젝트를 진행하다보면 여러 사람이 작업을 진행하기에 각자 알아서 커밋하고 푸시하는 방식은 규모가 커질수록 한계에 부딪힌다.

그렇기에 다음 세 가지를 따라야 하는데

1. 브랜치 전략을 먼저 수립. (git-flow, github-flow, gitlab-flow 등)
2. 브랜치 전략에 따라 브랜치는 PR 기반으로만 병합해야 함.
3. PR 리뷰를 통해 코드 품질과 맥락을 함께 검증

---

1. 브랜치 관리 전략 수립이 먼저

브랜치 전략이 없으면, PR 규칙도 리뷰 기준도 모두 흐려짐..

기본 브랜치 구조 예시 (gitflow 기준)

- `main`
  - 실제 배포 가능한 코드만 존재
- `develop`
  - 다음 배포를 위한 통합 브랜치
- `feature/*`
  - 기능 단위 개발 브랜치
- `bug/*`
  - 버그 수정 전용 브랜치
- `hotfix/*`
  - 배포 직후 긴급 수정용 브랜치

핵심 규칙

- 직접 `main`, `develop`에 커밋 금지

- 모든 변경은 feature/bug/hotfix 브랜치 → PR(git Pull Request) → merge 흐름만 허용
- 브랜치 이름에는 작업 의도와 범위가 드러나야 함

```
feature/login-32(이슈번호)
bug/null-pointer-exception-128
1``hotfix/urgent-auth-fail
```

---

2. 작업은 항상 Issue → Branch → PR 흐름으로

Issue를 먼저 작성한다 (매우 중요함미다)

모든 작업은 Issue에서 시작함.

- 왜 이 작업이 필요한가
- 무엇을 변경할 것인가
- 완료 조건은 무엇인가

이를 통해 **작업의 목적과 범위를 팀 전체가 공유**한다.

이슈는 팀원들이 정해둔 이슈 템플릿에 맞춰 이슈를 작성하면 됨.

만약 Jira 또는 Notion을 사용할 경우는 Jira와 Notion을 github을 연동하고 Jira나 Notion에 상세히 작성하고, github에는 일부만 작성.

단, 종종 면접관이나 타 개발자들이 github 주소에 들어오는 경우가 많기에 어느정도 꼼꼼하게 작성하는게 좋음.

Issue 기반 Branch 생성

- Issue 번호 + 작업 요약으로 브랜치 생성
- 작업 단위가 명확해지고, 추적 가능성이 확보된다

```
feature/user-profile-32
```

---

3. PR(git Pull Request)이란?

PR은 **다른 팀원이 이해하고 검증할 수 있도록 맥락을 제공하는 문서**

PR 작성 시 반드시 포함할 것

- 작업 배경 (왜 필요한가)
- 주요 변경 사항
- 영향 범위 (API, DB, 성능, 보안 등)
- 테스트 여부 및 방법

PR 운영 규칙 예시

- PR은 **작업자 본인이 직접 merge하지 않는다**

  ⇒ 하지만 최소 1명의 리뷰어가 있다면 허락 하에 승인해도 될듯
- 최소 1명 이상 리뷰 승인 필요
- 리뷰 코멘트 반영 후 merge

4. PR 리뷰 문화: 코드 + 의도 + 맥락

좋은 PR 리뷰란?

리뷰에서 확인해야 할 것

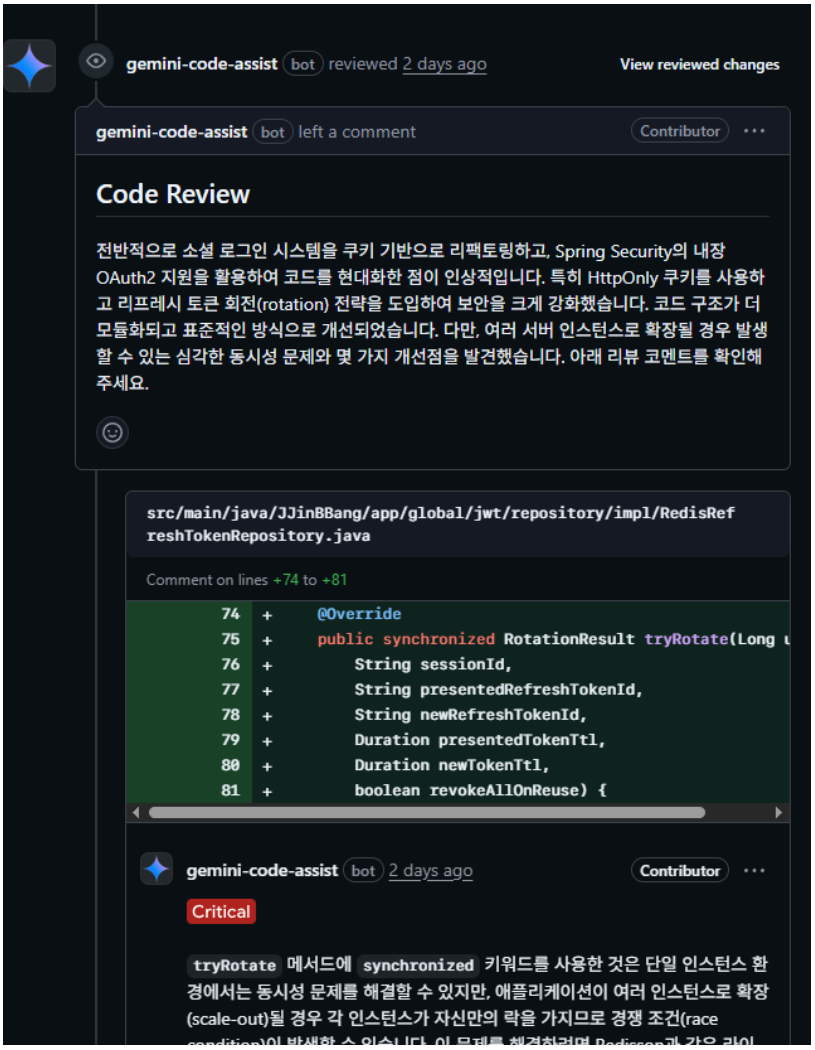
- 이 코드가 **요구사항을 정확히 충족하는지**
- 기존 코드 흐름과 **일관성이 있는지**
- 성능, 예외 처리, 확장성 관점에서 문제는 없는지
- 더 단순하거나 안전한 대안은 없는지

리뷰는 평가가 아니라 팀 전체의 코드 이해도를 높이는 과정

요새는 AI PR리뷰가 정말 좋음

하지만 아직은 받아들이는 부분도 있고 흘려 넘겨야하는 부분도 있어서

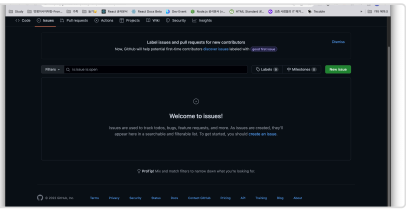
부분적으로 적용하면 좋을듯.



Github issues

"본 문서는 생활코딩의 github issues 강의를 정리한 글입니다."업무와 협업을 위한 게시판을 작성하는 곳일종의 레포의 게시판"새로운 이슈들을 만들고(Open Issue) 작업이 끝나면 이슈를 닫는다.(Closing Issus)"이 개념이 가장 기본이 되는 개념

https://velog.io/@leewooseong/Github-issues



Github PR

[Git 삽질기록] 'PR을 올리다'? Pull Request에 대해서

서론 이전 글에서 언급했듯이 과제를 Fork하는 방법에 대해 고민하다가, 어찌어찌 Fork를 하고 작업을 시작했다. BUT, git과의 전쟁은 여기서 끝이 아니었다. 나는 또 이런 말을 듣게 된다. "작업한 내용은 PR 올려주시면 좋을 것 같아요." ? 그게 뭔데... 그거 어떻게 하는 건데... PR...

https://holika.tistory.com/entry/Git-삽질기록-PR을-올리다-Pull-Request에-대해서