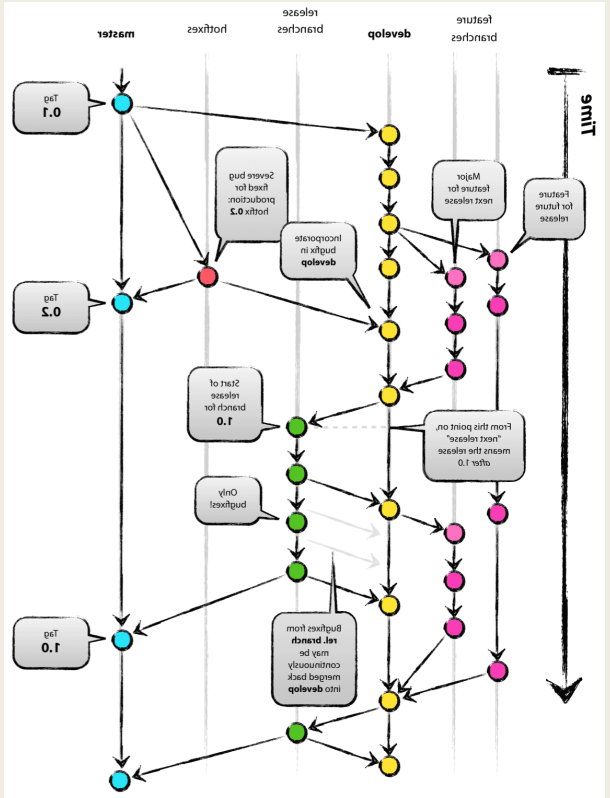
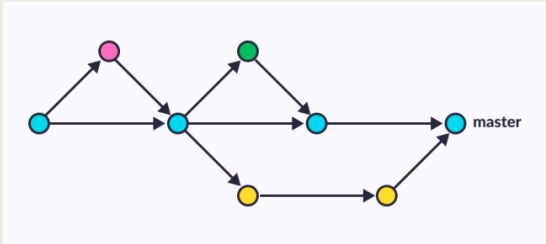
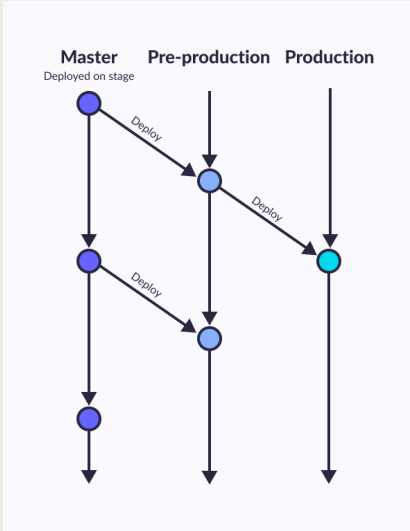






Git과 Git Branch 전략

목차

- Git의 필요성과 방식
- Git Branch 전략의 필요성
- Git Flow
- GitHub Flow & Trunk-Based
- GitLab Flow
- 어떤 전략을 취해야 할까?
- 자주 쓰는 명령어

Git의 필요성과 방식

- 일시적인 서버 장애가 있어도 개발을 계속할 수 있다. 로컬 저장소를 이용하면 되기 때문.
- 속도가 빠르다. 각각의 개발자들이 모두 분산처리 서버의 주인이 되는 셈이므로 서버가 직접 해야 될 일들이 많이 줄어든다.
- Git의 가장 큰 장점 중 하나는 브랜칭 기능입니다. 중앙 집중식 버전 제어 시스템과는 달리 Git 브랜치는 비용이 저렴하고 병합하기 쉽습니다.
- 작업되는 모든 내역(Commit내역)들은 모두 별도의 영역에서 관리되어 안전하게 프로젝트를 운영할 수 있습니다.

Git의 필요성과 방식

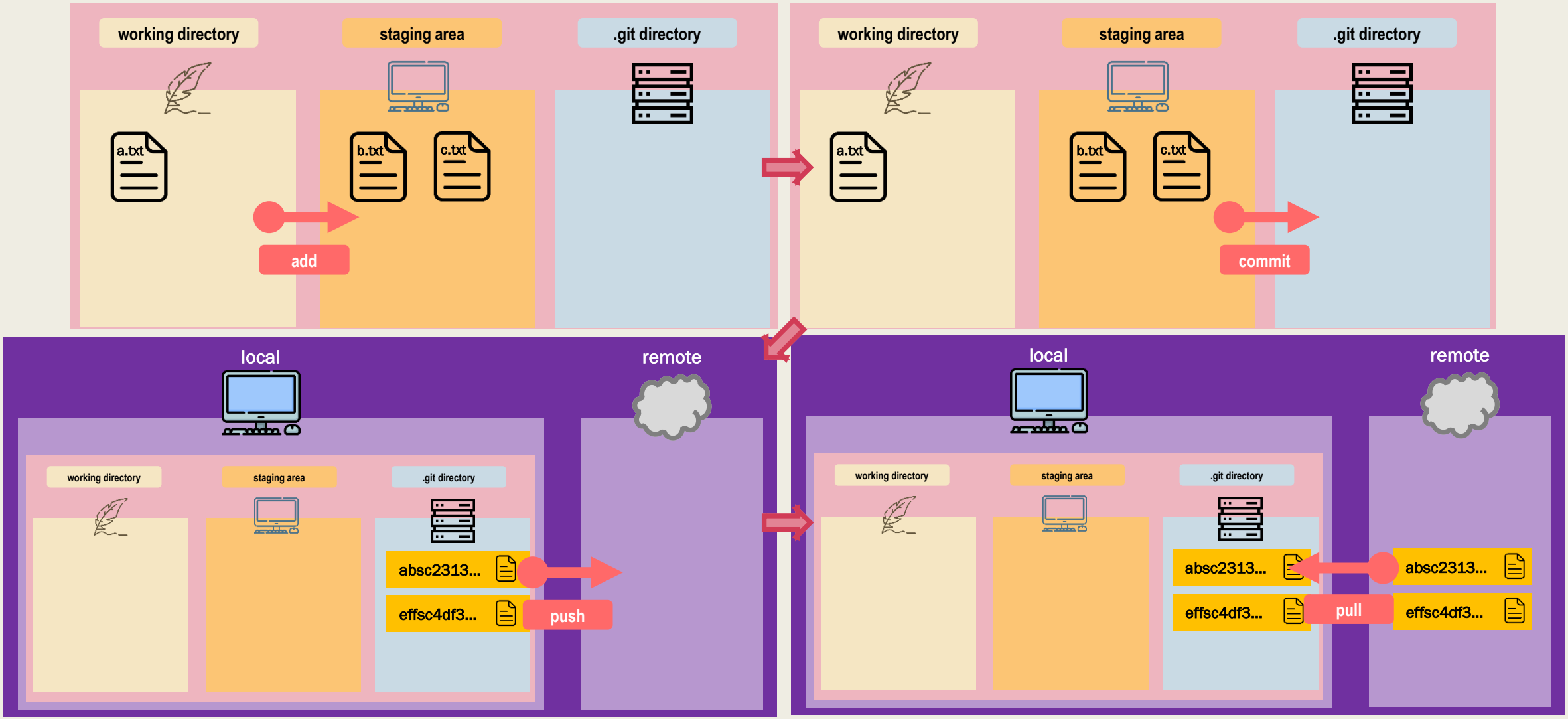
VCS (Version Controll System) 은 **델타 방식**과 **스냅샷 방식**으로 버전을 관리한다.

델타 방식은 SVN 등에서 사용하는 방식으로 수정사항이 발생한 버전만 따로 관리가 된다. 즉, 변경점들만 저장이 되기 때문에 누적되어서 관리가 된다.

스냅샷 방식은 **Git** 등에서 사용하는 방식으로 새로운 버전이 만들어질 때, 해당 버전의 최종 파일로 관리가 되고 있다.

델타 방식은 프로젝트의 규모가 커질 수록 변경 사항을 계산해야 되니까 느려지는 반면, 스냅샷 방식은 현재 버전만 사용하면 되므로 편리하다. 또한 Git은 분산 버전 관리를 하고 있어서 원격 저장소에 의존적이지 않게 작업할 수 있다. (다른 사람의 브랜치를 가져와 동기화하여 사용하는 등)

Git의 필요성과 방식

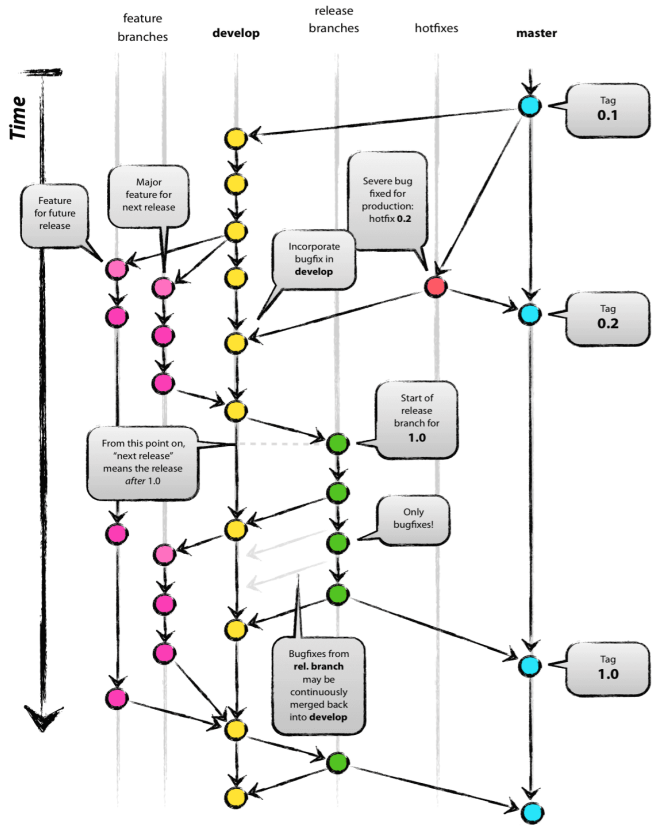

working directory
staging area
.git directory
a.txt
b.txt
c.txt
add
working directory
staging area
.git directory
a.txt
b.txt
c.txt
commit
local
remote
working directory
staging area
.git directory
absc2313...
effsc4df3...
push
local
remote
working directory
staging area
.git directory
absc2313...
effsc4df3...
pull
absc2313...
effsc4df3...

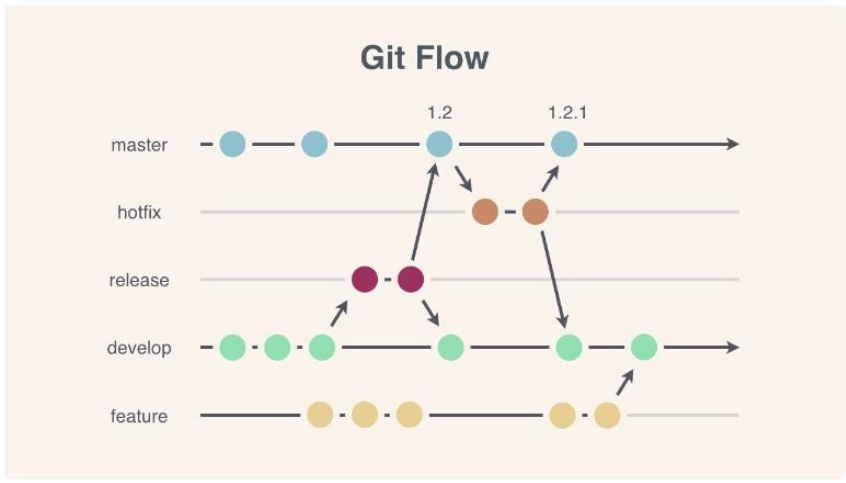
Git Branch 전략의 필요성

- 오직 메인 브랜치에서 수 많은 개발자들이 협업한다면, 내가 **작업중인 파일을 누군가 건드릴 수 있게** 된다. 여러 기능을 개발하면서 남겨진 커밋 **히스토리가 메인 브랜치에 뒤죽박죽 섞이게** 될 것이다.

- Git 브랜치 전략은 **여러 개발자가** 프로젝트의 **Git 브랜치를 효과적으로 관리하기 위한 워크플로우**이다.
- **브랜치 생성에 규칙을 만들어서 협업을 유연하게 하는 방법론**
- 브랜치 기능을 사용하면 **여러 기능을 여러 사람이 병렬적으로 개발**할 수 있게 된다.

Git Flow





•**master** : 라이브 서버에 제품으로 출시되는 브랜치.
•**develop** : 다음 출시 버전을 대비하여 개발하는 브랜치.
•**feature** : 추가 기능 개발 브랜치. **develop** 브랜치에 들어간다.
•**release** : 다음 버전 출시를 준비하는 브랜치. **develop** 브랜치를 **release** 브랜치로 옮긴 후 **QA**, 테스트를 진행하고 **master** 브랜치로 합친다.
•**hotfix** : **master** 브랜치에서 발생한 버그를 수정하는 브랜치.

- 주기적으로 배포하는 서비스에 적합하다.
- 버전 관리가 필요한 서비스에 적합하다.

Git Flow

Vincent Driessen 작성
, 2010년 1월 5일 화요일

반성문 (2020년 3월 5일)

이 모델은 2010년에 고안되었습니다. 이제 10년이 훨씬 넘었고 Git 자체가 등장한 지 얼마 되지 않았습니다. 그 10년 동안 git-flow(이 기사에 제시된 분기 모델)는 사람들이 일종의 표준처럼 취급하기 시작할 정도로 많은 소프트웨어 팀에서 엄청난 인기를 얻었습니다. .

그 10년 동안 Git 자체는 폭풍으로 세상을 휩쓸었고 Git으로 개발되고 있는 가장 인기 있는 소프트웨어 유형은 적어도 내 필터 버블에서는 웹 앱으로 더 많이 이동하고 있습니다. 웹 앱은 일반적으로 롤백되지 않고 지속적으로 제공되며 야생에서 실행되는 여러 버전의 소프트웨어를 지원할 필요가 없습니다.

이것은 내가 10년 전에 블로그 게시물을 작성할 때 염두에 두었던 소프트웨어 클래스가 아닙니다. 팀에서 소프트웨어를 지속적으로 제공하는 경우 팀에 git-flow를 적용하는 대신 훨씬 간단한 워크플로(예: GitHub 흐름) 를 채택하는 것이 좋습니다 .

그러나 명시적으로 버전이 지정된 소프트웨어를 빌드하거나 실제 여러 버전의 소프트웨어를 지원해야 하는 경우 git-flow는 여전히 팀에 적합할 수 있습니다. 지난 10년. 그 경우에는 계속 읽어 주십시오.

결론적으로 만병통치약은 존재하지 않는다는 점을 항상 기억하십시오. 자신의 상황을 고려하십시오. 미워하지 마십시오. 스스로 결정하십시오.

이처럼 Git-flow는 소프트웨어 버전 관리가 필요한 앱이나 솔루션, 혹은 public API에 적합한 워크플로우입니다. 웹 애플리케이션에서 Git-flow는 고려할 전략이 아닙니다.

개발자가 요구하는 상황보다 훨씬 더 복잡한 프로세스라고 지적합니다. 워크플로우를 단순화하면 개발자들이 빨리 습득할 수 있죠. 단순할수록 오버헤드가 발생하지 않아 **브랜치 전략을 이해하지 못하여 생기는 문제**가 사라진다고 말합니다.

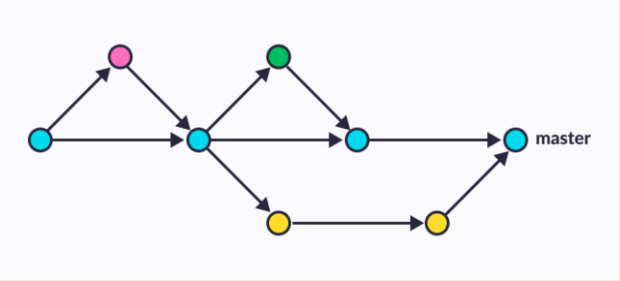
즉, Git Flow는 명시적으로 버전관리가 필요한 이를 테면, 스마트폰 어플리케이션, 오픈소스 라이브러리/프레임워크 등의 프로젝트에 적합하다. 유명한 글인 우아한형제들 기술 블로그에 **우린 Git-flow를 사용하고 있어요** 글을 작성한 팀도 안드로이드 앱 개발팀이다.

웹 어플리케이션은 특성상 사용자는 항상 최신의 단일 버전만을 사용하게된다. 여러 버전을 병렬적으로 지원할 필요가 없는 것이다. 또한 웹 어플리케이션은 하루에 몇번이고 릴리즈될 수 있다. 이런 특성상 **웹 어플리케이션 개발에 Git Flow는 다소 적합하지 않다.**

- 1개월 이상의 긴 호흡으로 개발하여 주기적으로 배포, QA 및 테스트, hotfix 등 수행할 수 있는
  여력이 있는 팀이라면 **git-flow**가 적합하다

- 수시로 릴리즈 되어야 할 필요가 있는 서비스를 지속적으로 테스트하고 배포하는 팀이라면
  github-flow 와 같은 간단한 work-flow가 적합하다

GitHub Flow & Trunk-Based



GitHub-flow는 Git-flow가 가진 복잡성을 전부 제거합니다. 유지 브랜치는 `master` 하나를 두는 방식으로 고수합니다. `master` 를 제외한 브랜치는 개발자 재량에 맡기니 복잡한 정책이 필요치 않습니다.

GitHub-flow 정책

- `master` 는 언제든지 배포가 가능하다.
- 새로운 프로젝트는 `master` 를 기반으로 별도 브랜치를 생성하여 작업을 진행한다.
- 브랜치는 로컬에 commit하고, 정기적으로 원격 브랜치에 push한다.
- 피드백이나 도움이 필요하거나, 코드 병합할 준비가 되었다면 pull request를 만든다.
- 다른 사람이 변경된 코드를 검토한 뒤 승인하면 `master` 에 병합한다.
- 병합된 `master` 는 즉시 배포할 수 있으며, 배포 해야만 한다.

이와 같은 정책을 가질 수 있는 이유는 **GitHub-flow가 상시 배포 모델**이기 때문입니다. Scott Chacon은 정기 배포가 가진 이점도 분명히 존재한다고 이야기하며, 다음과 같이 첨언합니다.

> **"만약 장기간 프로젝트가 존재하고, 핫픽스 등 유지보수를 위한 작업을 수행해야 되는 팀은 Git-flow가 타당하다. 상시 배포하는 팀은 간단한 GitHub-flow가 적합하다."**

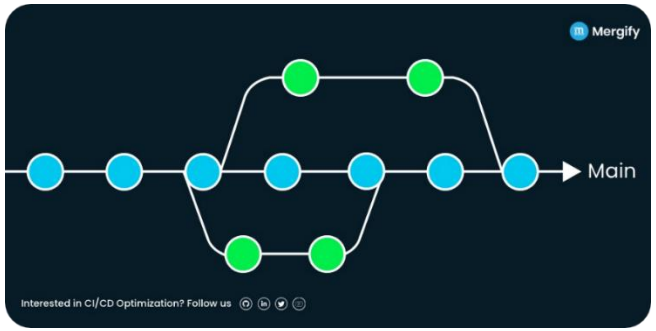
GitHub-flow를 공개한 그의 의견처럼 이 워크플로우는 상시 배포가 가능한 프로세스에서 유효합니다.

앞서 소개한 Git Flow는 대부분의 케이스에서 매우 복잡하다. 기계적으로 규칙을 따르기만 하면 큰 문제 없다고 하지만, 결국 복잡하기 때문에 많은 사람들이 실수하고 헤매게된다. Github Flow는 Git Flow와 다르게 굉장히 **간단한 구조**이다.

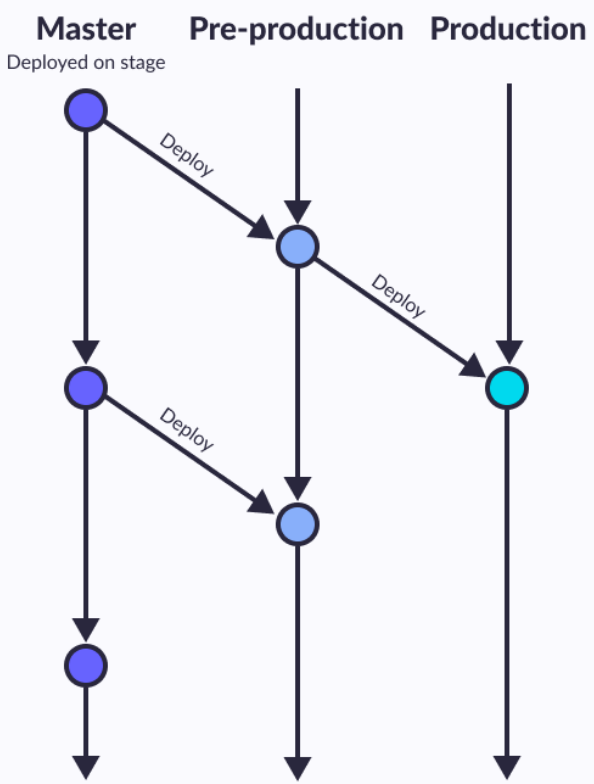
Github Flow는 이름 그대로 Github 환경에서 사용하기 적합한 브랜치 전략이기도하다. 또한 **자동화**를 적극 활용하기도 한다. 한번 Github Flow에 대해 알아보자.

GitHub Flow & Trunk-Based

1. Main만 장기유지 Branch로 사용
2. 다른 Branch들은 짧게 유지
3. 기능은 플래그를 사용해 Off 상태로 병합해 항상 최신 코드 유지
4. CI/CD를 적극적으로 활용



GitLab Flow



GitLab은 GitHub-flow가 지속 배포 모범 사례라는 점에 동의합니다. 다만 배포, 환경, 릴리즈 및 소스 통합 등 다양한 이슈에 대한 궁금증을 GitHub가 답하지 않았다고 지적합니다. GitLab은 GitHub-flow를 기반으로 상황에 따라 워크플로우를 활용하는 방법에 대하여 **추가적인 가이드**를 제공합니다.

1. 배포 기간이 정해진 경우 발생하는 통합 이슈 해결책
2. 서버 환경에 따른 브랜치 구성 방법
3. 릴리즈 버전이 `master` 와 분리되어 운영되어야 하는 상황 타개법

이 외에도 rebase를 활용한 commit 압축하기, merge commit 줄이는 방법, commit 메시지 작성 방법, 기능 분기에 대한 정책 등 다양한 방면에 대한 해답을 제시합니다. 그중에서도 GitLab이 내세운 건 이슈 기반 트래킹입니다.

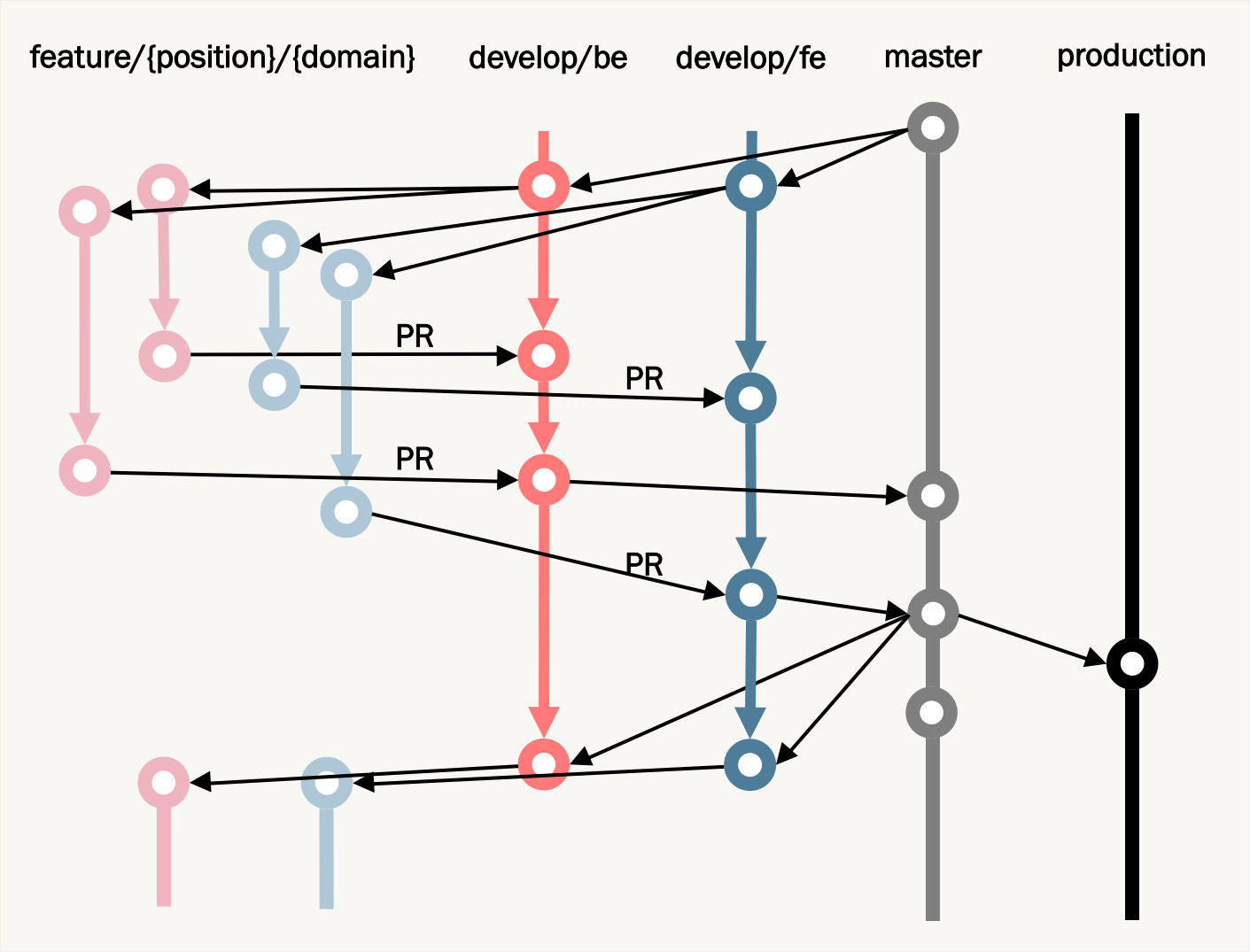
'코드를 변경하는 목적은 분명하므로 이슈로 생성하여 관리하자.'는 이념 아래로 코드 변경 사유를 투명하게 공개하자는 원칙입니다. GitLab은 동일한 repository를 사용하는 개발자들끼리 작업 내역을 알아야 된다고 말합니다. 또한 이슈를 기반으로 관리하면 브랜치 생명 주기가 보다 명확하게 파악되어 유지보수가 용이하죠.

어떤 전략을 취해야 할까?

- Repository를 구분하지 못하고, 하나의 Repository에 FE/BE/Infra 코드를 넣어야 한다.
  - BE/FE/Infra Position별로 Branch를 관리할 필요가 있다.
  - 실제 서비스를 배포하기 전에, BE와 FE를 통합하여 테스트해볼 필요가 있다.
- 웹 서비스를 개발하기에 기능이 추가되기만 할 뿐, 버전관리가 필요하지 않다.

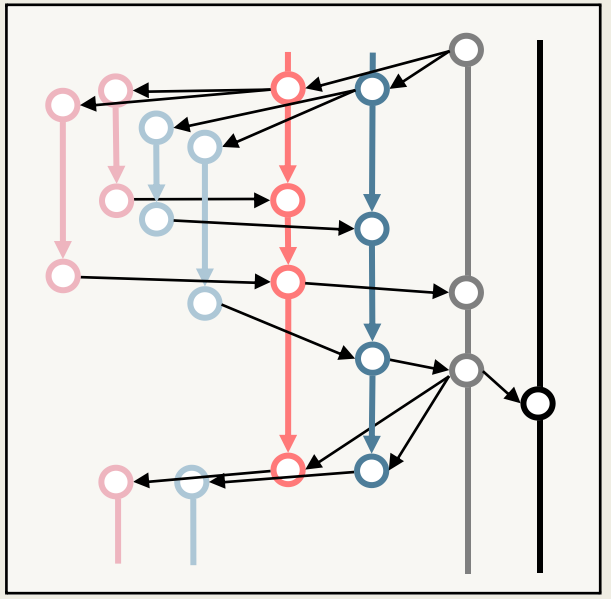
- **기능 개발 : feature**
- **position별 통합 : develop**
- **전체 코드 통합 및 테스트 : master**
- **배포 : production**

어떤 전략을 취해야 할까?



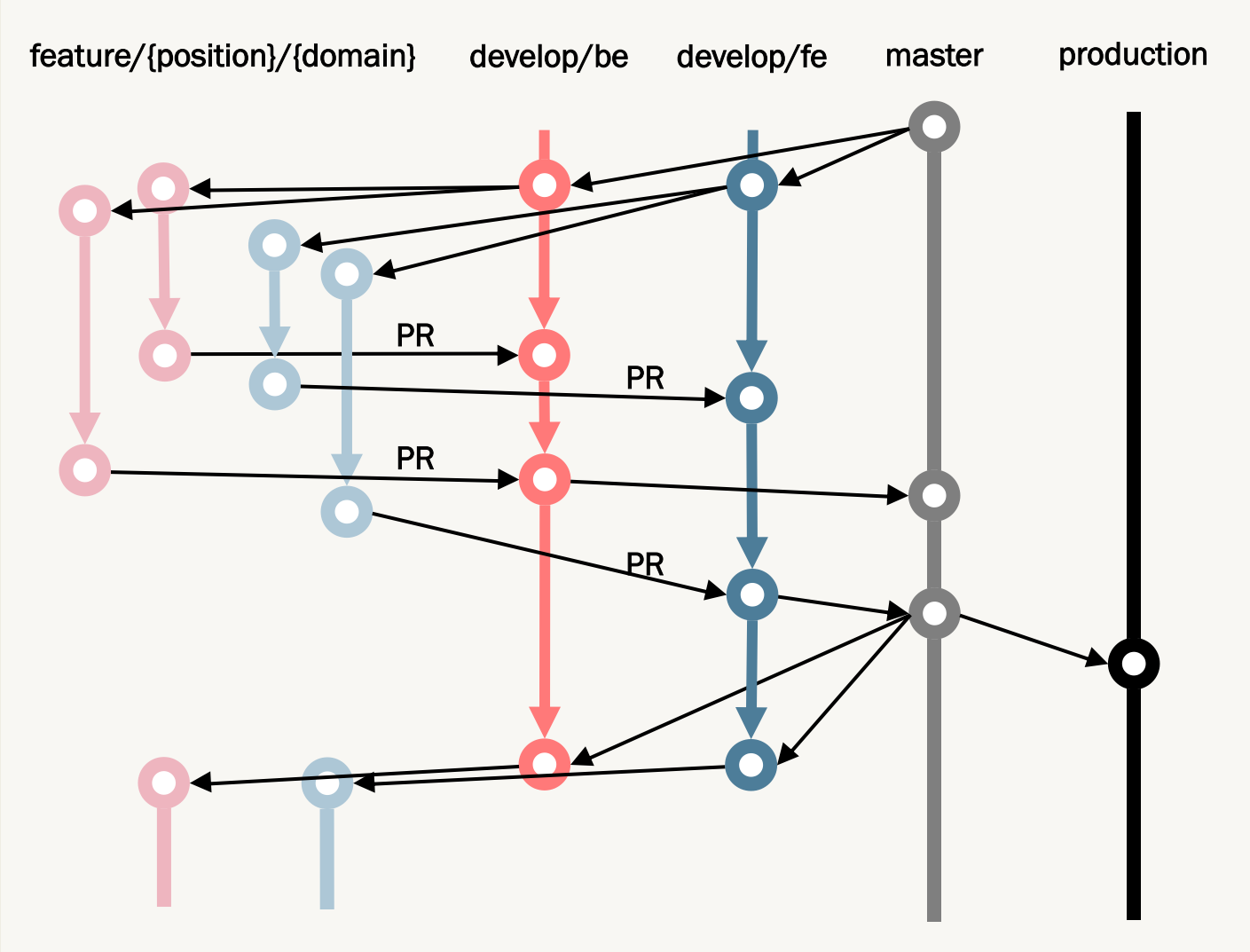
- **production**
  - ✓ 오류 없는 코드를 merge하여 배포하기 위한 브랜치
- **master**
  - ✓ develop/fe, be 브랜치를 통합하고, 테스트하는 용도의 브랜치
- **develop/{position}**
  - ✓ feature에서 개발된 기능을 통합하는 브랜치
  - ✓ develop에서 전체 개발 상황을 파악할 수 있다
  - ✓ 이렇게 통합되었을 때, 테스트를 하고 master로 merge한다
- **feature/{position}/{domain}**
  - ✓ 각각의 기능을 개발하기 위한 브랜치
  - ✓ 같은 브랜치에서 작업하더라도, 서로 작업하는 파일을 미리 구분하여 겹치지 않아야 한다. 그래야 충돌이 일어나지 않는다
- **PR**
  - ✓ feature에서 개발을 완료할 때마다 develop으로 PR을 보내어, PR 주기를 짧게 한다
  - ✓ develop이라는 하나의 브랜치에서 코드리뷰가 용이하다

어떤 전략을 취해야 할까?



- **production**
  - ✓ master에서 테스트 후, 오류 없는 코드만 merge하여 배포하기 위한 브랜치. 이 브랜치는 하나만 있다.
  - ✓ master > **production** 으로만 merge 받는다.
  - ✓ **production**에서는 코드를 수정하지 않을 것이기에 다른 브랜치로 pull 할 필요가 없다.
  - ✓ **develop과 feature 브랜치와 소통하지 않는다 (정합성이 깨질 수 있다)**
- master (master > **production 주기는 5일 목표)**
  - ✓ FE와 BE 브랜치를 통합하여 테스트하는 용도의 브랜치다. 이 브랜치는 하나만 있다.
  - ✓ 이 브랜치로 checkout 하여 웹 서비스 테스트를 진행하고, 다른 브랜치에서는 웹 서비스 테스트를 하지 않는다.
  - ✓ **develop** > master 로만 merge 받고, master > **develop** 으로만 pull 한다.
  - ✓ feature **브랜치와 소통하지 않는다 (정합성이 깨질 수 있다)**
- **develop/{position} (develop > master 주기는 3일 목표)**
  - ✓ feature에서 개발이 완료된 기능을 통합하기 위한 브랜치 (처음에 다른 파일들과의 연동을 확인하기 위해 merge 하는 것은 허용)
  - ✓ BE에서 개발된 기능과 FE에서 개발된 기능을 통합하여 테스트가 필요할 때 master로 merge한다.
  - ✓ 새로운 position이 생길 때만 "develop/{position}" 꼴로 새로 만든다.
  - ✓ 기능이 개발될 때마다 PR을 받고 merge 되기에, 전체적인 개발 진행 상황을 파악할 수 있다. 그래서 position별로 코드리뷰를 할 수 있다. 대부분의 코드리뷰는 **develop** 브랜치에서만 진행되고, 이외의 브랜치에서는 코드리뷰가 이루어지지 않는다. feature에 대한 코드리뷰가 완료되고 코드에 이상이 없다면 **develop** 으로의 merge를 승인한다.
  - ✓ **production 브랜치와 소통하지 않는다, FE와 BE가 직접 소통하지 않는다. (정합성이 깨질 수 있다)**
- feature/{position}/{domain} (feature > **develop 주기는 2일 목표)**
  - ✓ 각각의 기능을 개발하기 위한 브랜치. 필요에 따라 **누구든 새로 만들 수 있다.**
  - ✓ 여러 명이 동일한 브랜치에서 작업하더라도, 작업 파일을 미리 구분하여 겹치지 않도록 한다. 그래야 충돌이 일어나지 않는다 (설계를 잘하면 된다.)
  - ✓ 기능 개발이 완료되면 개발한 position에 맞는 **develop** 브랜치로 PR을 보내고 팀원들과 코드리뷰를 한다.
  - ✓ 수정할 일이 있다면 브랜치를 유지하지만, 더 이상 수정할 일이 없다면 merge 후 브랜치를 삭제한다.
- **(PR)**
  - ✓ feature에서 개발을 완료할 때마다 develop으로 PR을 보내어, PR 주기를 짧게 한다.
  - ✓ **develop**이라는 하나의 브랜치에서 코드리뷰가 용이하다.

어떤 전략을 취해야 할까? – 브랜치 이름 패턴



- **production**
- **master**

develop/{position}
- develop/be
- develop/fe

feature/{position}/{domain}
- feature/be/login
- feature/be/admin
- feature/be/cure
- feature/be/community
- feature/be/coaching
- feature/fe/login
- feature/fe/admin
- feature/fe/cure
- feature/fe/coaching
- feature/fe/community

어떤 전략을 취해야 할까? – commit 컨벤션

타입

타입은 태그와 제목으로 구성되고, 태그는 영어로 쓰되 첫 문자는 대문자로 합니다.

"태그: 제목"의 형태이며, : 뒤에만 space가 있음에 유의합니다.

태그 이름	설명
Feat	새로운 기능을 추가할 경우
Fix	버그를 고친 경우
Design	CSS 등 사용자 UI 디자인 변경
!BREAKING CHANGE	커다란 API 변경의 경우
!HOTFIX	급하게 치명적인 버그를 고쳐야하는 경우
Style	코드 포맷 변경, 세미 콜론 누락, 코드 수정이 없는 경우
Refactor	프로덕션 코드 리팩토링
Comment	필요한 주석 추가 및 변경
Docs	문서를 수정한 경우
Test	테스트 추가, 테스트 리팩토링(프로덕션 코드 변경 X)
Chore	빌드 태스트 업데이트, 패키지 매니저를 설정하는 경우(프로덕션 코드 변경 X)
Rename	파일 혹은 폴더명을 수정하거나 옮기는 작업만인 경우
Remove	파일을 삭제하는 작업만 수행한 경우

기능

Feat, Fix, Design, !BREAKING CHANGE 태그가 기능 태그의 종류입니다.

Feat: 새로운 기능을 추가할 경우
Fix: 버그를 고친 경우
Design: CSS 등 사용자 UI 디자인 변경
!BREAKING CHANGE: 커다란 API 변경의 경우 (ex API의 arguments, return 값의 변경, DB 테이블 변경, 급하게 치명적인 버그를 고쳐야 하는 경우)

개선

Style, Refactor, Comment 태그가 개선 태그의 종류입니다.

Style: 코드 포맷 변경, 세미 콜론 누락, 코드 수정이 없는 경우
Refactor: 프로덕션 코드 리팩토링, 새로운 기능이나 버그 수정없이 현재 구현을 개선한 경우
Comment: 필요한 주석 추가 및 변경

Style의 경우 오타 수정, 탭 사이즈 변경, 변수명 변경 등에 해당하고, Refactor의 경우 코드를 리팩토링 하는 경우에 적용할 수 있습니다.

어떤 전략을 취해야 할까? – commit 컨벤션

그 외

Docs, Test, Chore, Rename, Remove 태그가 그 외 태그의 종류입니다.

> Docs: 문서를 수정한 경우
> Test: 테스트 추가, 테스트 리팩토링 (프로덕션 코드 변경 없음)
> Chore: 빌드 태스크 업데이트, 패키지 매니저 설정할 경우 (프로덕션 코드 변경 없음)
> Rename: 파일 혹은 폴더명을 수정하는 경우
> Remove: 사용하지 않는 파일 혹은 폴더를 삭제하는 경우

Docs의 경우 README.md 수정 등에 해당하고, Test는 test 폴더 내부의 변경이 일어난 경우에만 해당합니다. Chore의 경우 package.json의 변경이나 dotenv의 요소 변경 등, 모듈의 변경에 해당됩니다.

"태그: 제목"의 형태

```
Feat: Setup project
Update: update README.md
Merge branch 'be' into 'develop'
Fix: Rename 포팅매뉴얼
Design: 디자인 수정
Test: add Dockerfile, nginx.conf, docker-compose.yml, .env
Update: update README.md
Refactor: docker copmse redis 비밀번호 설정 추가
!HOTFIX: nginx 포트 번호 수정
```

제목은 어떻게 적는가?

제목은 코드 변경 사항에 대한 짧은 요약을 나타냅니다. 제목은 다음의 규칙을 지킵니다.

> 1. 제목의 처음은 동사 원형으로 시작합니다.
> 2. 총 글자 수는 50자 이내로 작성합니다.
> 3. 마지막에 특수문자는 삽입하지 않습니다. 예) 마침표(.), 느낌표(!), 물음표(?)
> 4. 제목은 **개조식 구문**으로 작성합니다.

만약 영어로 작성하는 경우 다음의 규칙을 따릅니다.

> 1. 첫 글자는 **대문자**로 작성합니다.
> 2. "Fix", "Add", "Change"의 명령어로 시작합니다.

한글로 제목을 작성하는 경우 다음의 규칙을 따릅니다.

> 1. "고침", "추가", "변경"의 명령어로 시작합니다.

> 예시)
> Feat: "추가 get data api 함수"

어떤 전략을 취해야 할까? – 브랜치 삭제 이유

[git] 깃 브랜치를 닫았으면 삭제하는것이 좋다

브랜치 삭제 명령어 :

git **branch** -d [branch-name] # 로컬 브랜치 삭제

git **push** --delete [remote] [branch-name] # 원격 브랜치 삭제

4) 그래서 Branch를 왜 삭제해야 해요?

앞서 말씀드렸듯이 branch는 다른 작업환경에 방해받지 않고, 독립적으로 작업할 수 있습니다. 즉, 각각의 작업을 분기할 수 있다는 것인데요. 이를 위해 이전 commit을 point 합니다.

하지만 작업한 내용이 **이미 master 브랜치에 merge 됐고,** head가 이를 가리키고 있다면, **과거에 작업한 branch를 남겨야 하는 이유가 있을까요?**

최신의 결과물은 master 브랜치를 참고하면 되고, 과거의 내역이 궁금하다면 이전 commit으로 돌아가서 보면 됩니다. branch라는 작업 환경 자체를 유지해야 할 필요는 없어 보이는데요.

현재 근무하는 회사에서는 이를 계속해서 쌓고 있었고, 삭제를 제안하고 적용했습니다.

어떤 전략을 취해야 할까? – 브랜치 삭제 이유

나동호

안녕하세요. 늦게나마 답변을 드립니다...

Q1. QA 중 버그를 수정할 때 티켓마다 브랜치를 생성하고 삭제하는 과정은 많이 귀찮지 않나요?
A1. Github에서는 'Merge pull request' 버튼 클릭 후 'Delete branch' 버튼 클릭으로 쉽게 코드 병합과 브랜치 삭제를 할 수 있습니다.

Q2. 왜 티켓마다 브랜치를 생성/삭제 해야하는가?
A2. 저희는 티켓마다 브랜치를 생성한 후 하나의 작업만을 수행합니다. 이 행위는 몇 가지 장점이 있습니다.
- 첫째, 하나의 티켓에 하나의 작업만 진행하기 때문에 코드의 변경이 무분별하게 일어나지 않습니다.
- 둘째, 리뷰어들이 코드 리뷰를 할 때 의도를 파악하기 쉽습니다. (그 만큼 빠른 리뷰를 받을 수 있습니다)
- 셋째, 불가피하게 작업 전환을 해야할 때 쉽게 작업을 전환할 수 있고, 시간이 어느정도 지난 후 다시 원래 작업으로 되돌아가도 작업을 이어가기가 좀 더 수월합니다.
- 셋째, 티켓과 작업을 연결되어 어떤 버전에 작업이 포함됐는지 파악하기 쉽고, 그 당시 히스토리를 추적하는데 도움이 됩니다.

좋아요 · 답글 달기 · 4년

나동호

저희는 더 이상 사용하지 않는 브랜치는 제거하고 있습니다.
release 브랜치도 릴리즈를 하면 더 이상 사용하지 않기 때문이 제거하고 있습니다.

좋아요 · 답글 달기 · 3년

자주 쓰는 명령어

🏳 git **config** --global user.name "[name]" # 사용자 이름 설정

🏳 git **config** --global user.email "[email]" # 사용자 Email 설정

git **config** --global color.ui auto # 출력되는 command line를 읽기 쉽도록 자동으로 색깔 설정

git **config** –list # 사용자 설정 정보 확인

git **config** --global user.name

git **config** --global user.email

cd [my_project path] # 프로젝트 directory로 이동

echo "# my_project" >> README.md # README 파일 생성

🏳 git **init** # git 저장소로 등록

🏳 **rm -rf .git** # 로컬리포지토리와 연결을 끊어준다. (git의 관리에서 삭제한다.)

🏳 git **clone** [url] # 기존에 사용 중인 저장소를 clone 해서 가져옴

git **add** README.md # 관리할 file을 Staging Area 추가

🏳 git **add .** # 현재 directory 의 모든 파일을 Staging Area 로 이동

🏳 git **status** # git의 현재 상태 확인한다. Staging Area, UnStage Area 에 있는 파일 및 Untracked 상태의 파일을 확인 할 수 있음

🏳 git **commit** -m "[커밋 상세 메시지]" # Staging Area의 파일들 commit 하기

git **commit** --amend # -amend 옵션으로 최근 커밋을 재작성할 수 있음

🏳 git **restore** --staged [파일명] # add 취소. 즉, staging상태에서 unstaging 상태로 변경한다는 의미.

🏳 git **push** (-u) [remote] [branch] # 저장소에 commit 반영하기. -u 옵션은 upstream repository를 설정해준다. 즉 한번 설정한 후로는 git push, git pull 만 간단히 쓸 수 있음

🏳 git **pull** [remote] [branch] # 저장소에서 commit 가지고 오기

자주 쓰는 명령어

🏳 git **diff** # stage 되지 않은 변경 비교

git **diff** --staged # stage 되어 있으나 아직 commit 되지 않은 변경 비교

git **diff** [commit-id] [commit-id] # 두 commit 사이의 차이점을 볼 수 있다

git **revert** [commit-id] # 커밋해시코드 시점으로 되돌아간다. (이전 커밋 유지)

git **reset** # Staged 상태의 파일을 Unstage 로 변경

git **reset** --[hard/mixed/soft] [commit] # staging area가 초기화되고 working tree를 특정 커밋시점으로 덮어쓰기 (이전 커밋 삭제)

(mixed, soft는 변경 이력 즉 커밋은 삭제되지만, 변경된 code의 내용은 남아있게 된다.)

git **reset** --hard HEAD^ # 바로 이전 Head 시점으로 초기화. working directory, 최근 commit 이력이 삭제된다.

git **reset** HEAD~3 # 앞의 3개 다 없앤다는 뜻

git **rm** [file] # Staging Area나 Working directory에 있는 파일을 삭제

git **mv** [existing-path] [new-path] # 파일/폴더의 이름을 변경

git **branch** # branch 목록 조회, *가 현재 branch

git **branch** -r # 원격 branch 목록 조회

git **branch** -a # 전체 branch 목록 조회

🏳 git **branch** [branch-name] # branch 생성

git **branch** -m [old_version] [new_version] # 브랜치 이름 변경

git **branch** -d [branch-name] # 로컬 브랜치 삭제

git **push** --delete [remote] [branch-name] # 원격 브랜치 삭제

🏳 git **checkout** [branch-name] # Branch를 변경하고 해당 파일을 Working directory로 복사

git **merge** [branch] # 특정 branch를 현재 checkout 된 branch에 merge

git **merge** origin/main # remote origin의 main branch merge

git **merge** main # local 저장소의 main branch merge

자주 쓰는 명령어

🏳 git **log** # 커밋 히스토리를 시간 순으로 조회

git **log** -[number] # 최근 number개만 노출

git **log** --oneline # oneline 으로 히스토리 노출

git **log** --graph # graph 로 출력

git **log** --oneline --graph # 한 줄로 그래프 형태로 commit 히스토리 보기

git **reflog** # 헤드가 가리키던 커밋 기록을 모두 보여줌. HEAD@{숫자}꼴로 나오는데 숫자가 작을수록 최근에 헤드가 가리켰던 커밋이라는 뜻

git **fetch** [remote] # 로컬 저장소에 있는 것을 뺀 remote 저장소의 모든 것을 가져옴

git **fetch** --prune # --prune 옵션으로 remote 저장소에 지워진 브랜치를 local 반영하여 local의 불필요한 branch를 삭제

🏳 git **remote** # 현재 프로젝트에 등록된 리모트 저장소를 확인

git **remote** –v # 단축이름과 URL을 함께 조회

🏳 git **remote** add [remote] [url] # git url 에 remote(저장소) 이름으로 등록

git **remote** add origin https://github.com/{username}/{repository}.git

git **remote** remove origin # 원격리포지토리의 연결을 끊어준다.

git **stash** # modified, staged 변경 내용 임시 저장

git **stash** list # stashed 조회

git **stash** pop # stash 내역 working directory로 추가

git **stash** drop # stash file 제거

git **rm** [file-name] # 로컬, 원격 저장소 모두 파일 삭제

git **rm** --cached [file-name] # 원격 저장소에서만 파일 삭제 (git로 관리가 불필요한 파일인데 .gitignore에 추가하는 것을 빼먹고 이미 commit하는 경우)

출처

git flow 전략

- https://inpa.tistory.com/entry/GIT-%E2%9A%A1%EF%B8%8F-github-flow-git-flow-%F0%9F%93%88-%EB%B8%8C%EB%9E%9C%EC%B9%98-%EC%A0%84%EB%9E%B5#
- https://hyeon9mak.github.io/git-branch-strategy/
- https://sungjk.github.io/2023/02/20/branch-strategy.html
- https://techblog.woowahan.com/2553/
- https://nvie.com/posts/a-successful-git-branching-model/
- https://tech.mfort.co.kr/blog/2022-08-05-trunk-based-development/

출처

웹 개발에 적합한 git flow

- https://hudi.blog/git-branch-strategy/
- https://blog.hwahae.co.kr/all/tech/9507

github action testing

- https://velog.io/@adam2/GitHub-Actions-%EB%A1%9C-%ED%92%80%EB%A6%AC%ED%80%98%EC%8A%A4%ED%8A%B8-test-%EA%B2%80%EC%A6%9D%ED%95%98%EA%B8%B0

출처

브랜치 삭제 이유

- https://yeon-kr.tistory.com/200

- https://velog.io/@koyk0408/git-%EA%B9%83-%EB%B8%8C%EB%9E%9C%EC%B9%98%EB%A5%BC-%EB%8B%AB%EC%95%98%EC%9C%BC%EB%A9%B4-%EC%82%AD%EC%A0%9C%ED%95%98%EB%8A%94%EA%B2%83%EC%9D%B4-%EC%A2%8B%EB%8B%A4

브랜치 이름 패턴

- https://junjunrecord.tistory.com/131

gitlab flow

- https://brownbears.tistory.com/605

- https://docs.gitlab.com/ee/topics/gitlab_flow.html

출처

git 충돌이 발생하는 경우

- https://www.atlassian.com/ko/git/tutorials/using-branches/merge-conflicts

git commit 컨벤션

- https://overcome-the-limits.tistory.com/entry/%ED%98%91%EC%97%85-%ED%98%91%EC%97%85%EC%9D%84-%EC%9C%84%ED%95%9C-%EA%B8%B0%EB%B3%B8%EC%A0%81%EC%9D%B8-git-%EC%BB%A4%EB%B0%8B%EC%BB%A8%EB%B2%A4%EC%85%98-%EC%84%A4%EC%A0%95%ED%95%98%EA%B8%B0

자주 쓰는 명령어

- https://wecandev.tistory.com/152