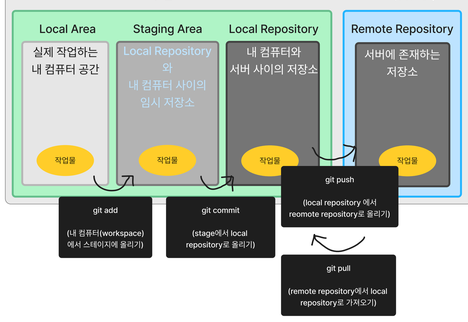
Git 기본 1

Git 기본 1



Git 기본 2

Git 기본 2

add : workspace 의 변경내용을 Staging Area 로 올리기.

stash : Local Area 에서 수정한 파일들을 임시 저장소 (stage 와는 다른 임시저장소) 에 저장.
아직 작업이 마무리되지 않은 상태에서 잠시 브랜치를 변경하거나 다른 작업을 해야할 때 사용

commit : 현재 상황(snapshot)을 hash값 함께 기록을 남기는 것.
commit 은 영구저장되며, roll-back 하려면 revert 와 reset 의 두가지 방법이 존재함.

1. reset : 해당 commit 까지 실행 취소
해당 commit 뿐 아니라 해당 commit 이후 commit 하지 않은 작업들까지 실행취소된다.
따라서 되돌려야 하는 commit 이 local 에만 존재하는 경우, 재랑껏 사용하면 되고,
remote 에서는 reset 명령을 사용하면 commit 기록이 변경되기 때문에 사용하지
않는 것을 권장.
2. revert : 해당 commit에 있는 변경사항을 commit하기 전으로 되돌린다.
그 후 되돌린 상태를 자동으로 commit 한다.

branch : '현재 commit' 을 기준으로 분기점을 만들어 작업하는 것.

merge : 'Local Repository' 에서 다른 branch 를 현재 branch 로 합치는 작업.
합치는 기준은 각 branch 의 최근 commit 이 기점.

push : 'Local Repository' 의 내용을 'Remote Repository' 로 올리는 것.

pull : 'Remote Repository' 의 branch를 'Local Repository' 의 branch로 가져오는 작업

Git 기본 3

Git 기본 3

add : workspace 의 변경내용을 Staging Area 로 올리기.

stash : Local Area 에서 수정한 파일들을 임시 저장소 (stage 와는 다른 임시저장소) 에 저장.
아직 작업이 마무리되지 않은 상태에서 잠시 브랜치를 변경하거나 다른 작업을 해야할 때 사용

commit : 현재 상황(snapshot)을 hash값 함께 기록을 남기는 것.
commit 은 영구저장되며, roll-back 하려면 revert 와 reset 의 두가지 방법이 존재함.

1. reset : 해당 commit 까지 실행 취소
해당 commit 뿐 아니라 해당 commit 이후 commit 하지 않은 작업들까지 실행취소된다.
따라서 되돌려야 하는 commit 이 local 에만 존재하는 경우, 재랑껏 사용하면 되고,
remote 에서는 reset 명령을 사용하면 commit 기록이 변경되기 때문에 사용하지
않는 것을 권장.
2. revert : 해당 commit에 있는 변경사항을 commit하기 전으로 되돌린다.
그 후 되돌린 상태를 자동으로 commit 한다.

branch : '현재 commit' 을 기준으로 분기점을 만들어 작업하는 것.

merge : 'Local Repository' 에서 다른 branch 를 현재 branch 로 합치는 작업.
합치는 기준은 각 branch 의 최근 commit 이 기점.

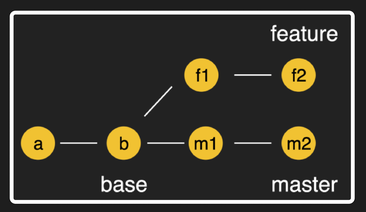
push : 'Local Repository' 의 내용을 'Remote Repository' 로 올리는 것.

pull : 'Remote Repository' 의 branch를 'Local Repository' 의 branch로 가져오는 작업

Git 기본 4

Git 기본 4

기본적으로 merge 는 3-way-merge 의 방법을 기준으로 진행됨.



1. 기본적으로 base 파일의 내용을 참고한다.
2. 한 branch 만 바뀐 경우 바뀐 쪽으로 변경된다.
3. 둘 다 바뀌지 않았을 경우 base 의 내용을 따른다.
4. 이름이 같은 파일의 내용이 다를 경우 conflict가 일어날 수 있으며, 이는 직접 수정해야한다.

merge 한 이후 의 상황은 크게 2가지로 나뉜다.
1. 서로 다른 파일을 수정했을 때
= 자동으로 merge 가 완료된다. (충돌 없음)
2. 서로 같은 파일을 수정했을 때
2-1. 수정한 부분이 다를 때
= 자동으로 merge 가 완료된다. (충돌 없음)
2-2. 수정한 부분이 같을 때
= conflict 가 일어난다. 직접 수정해줘야 한다.
2-3. 병합할 수 없는 변경사항 (ex) 두 변경사항이 충돌이 일어날 때, 코드라인이 다를 때 등)
= conflict 가 일어난다. 직접 수정해줘야 한다.

Git 기본 5

Git 기본 5

fetch : 'Remote Repository' 의 최신 commit 을 임시 branch 로 가져오기.

pull : fetch + merge 의 기능을 가지고 있음.

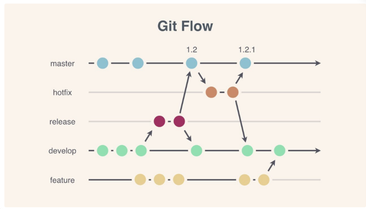
⇒ merge 만 한 경우 Local Repository 에서만 작업한 것이기 때문에 Remote Repository로 push 를 따로 해주어야 한다.

merge 없이 본인 branch에서만 계속 개발하면 안된다.
⇒최종적으로 merge 할 때 변경사항이 많아서 conflict 수정이 엄청 많아진다.
⇒지속적으로 변경사항을 반영하고 충돌을 제거하면서 최신화된 branch 로 만들어가야 바람직

pull 은 'Remote Repository' 가 'Local Repository' 보다 최신일 때만 내려받는 걸 권장함.
⇒ Develop 에서는 fetch를 통해 오류가 있는지 확인한 후 merge 를 진행 하는 게 안전함.
⇒ Develop 이 가장 최신이 되고, 각 branch 에서 origin/Develop 을 pull 해서 사용.

Branch 전략 1_ Git Flow

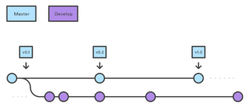
Git-flow 브랜치 구조



- **master** : 라이브 서버에 제품으로 **출시**되는 브랜치.
- **develop** : 다음 출시 버전을 대비하여 **개발**하는 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 수정**하는 브랜치.

메인 브랜치

메인 브랜치는 **master 브랜치**와 **develop 브랜치** 두 종류를 말한다.



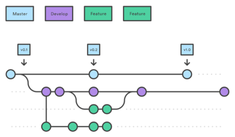
master 브랜치는 **배포 가능한 상태**만을 관리하는 브랜치를 말하며,

develop브랜치는 다음에 **배포할 것을 개발**하는 브랜치이다.

즉 develop 브랜치는 통한 브랜치의 역할을 하며, 평소에는 이 브랜치를 기반으로 개발을 진행한다.

보조 브랜치

보조 브랜치는 **피처 브랜치(feature branch)** 또는 **토픽 브랜치(topic branch)를** 말한다.



- **가지가 뻗어나오는 곳** : develop
- **뻗어나갔던 가지가 다시 합쳐지는 곳** : develop
- **이름 설정** : master, develop, release-*, hotfix-* 를 제외하기만 하면 자유롭게 이름 설정이 가능하다.
- **새로운 기능을 추가**할 때 주로 사용하는 가지이다.

master 브랜치에서 develop 브랜치를 만들었고,

develop 브랜치에서 다시 **feature 브랜치를 나눠 작업**을 하고 있는 것을 그림을 통해 알 수 있다.

develop 브랜치에는 기존에 잘 작동하는 개발코드가 담겨있으며,

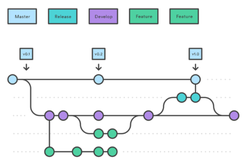
보조 브랜치는 새로 변경될 개발코드를 분리하고 각각 보존하는 역할을 한다.

즉, 보조 브랜치는 **기능을 다 완성할 때까지 유지하고, 다 완성되면 develop 브랜치로 merge 하고 결과가 좋지 못하면 버리는 방향**을 취한다.

보조 브랜치는 보통 개발자 저장소에만 있는 브랜치고, origin에는 push하지 않는다.

릴리즈 브랜치(release branch)

릴리즈 브랜치는 **배포를 위한 최종적인 버그 수정** 등의 개발을 **수행**하는 브랜치를 말한다.



- **가지가 뻗어나오는 곳** : develop
- **뻗어나갔던 가지가 다시 합쳐지는 곳** : develop, master
- **이름 설정** : release-*
- 새로운 **제품을 배포**하고자 할 때 사용하는 가지이다.

develop 브랜치에 버전에 포함되는 기능이 merge 되었다면 QA를 위해 develop 브랜치에서부터 release 브랜치를 생성한다.

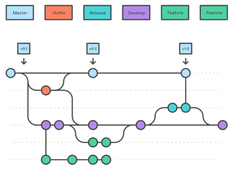
배포 가능한 상태가 되면 master 브랜치로 병합시키고, **출시된 master 브랜치에 버전 태그(ex, v1.0, v0.2)를 추가**한다.

release 브랜치에서 기능을 점검하며 발견한 버그 수정 사항은 develop 브랜치에도 적용해줘야 한다.

그러므로 배포 완료 후 develop 브랜치에 대해서도 merge 작업을 수행해야 한다.

핫픽스 브랜치(hotfix branch)

핫픽스 브랜치는 **배포한 버전에서 긴급하게 수정**할 필요가 있을 때 master 브랜치에서 분리하는 브랜치를 말한다.



- **가지가 뻗어나오는 곳** : master
- **뻗어나갔던 가지가 다시 합쳐지는 곳** : develop, master
- **이름 설정** : hotfix-*
- 제품에서 버그가 발생했을 경우에는 처리를 위해 이 가지로 해당 정보들을 모아준다. 버그에 대한 수정이 완료된 후에는 develop, master에 공장 반영해주며 tag를 통해 관련 정보를 기록해둔다.

버그를 잡는 사람이 알하는 동안에도 다른 사람들은 develop 브랜치에서 하던 일을 계속할 수 있다.

이 때 만든 hotfix 브랜치에서의 변경 사항은 develop 브랜치에도 merge 하여 문제가 되는 부분을 처리해줘야 한다.

release 가지가 생성되어 관리되고 있는 상태라면 해당 가지에 hotfix정보를 병합시켜 다음번 배포 시 반영이 정상적으로 이루어질 수 있도록 해준다.

Hotfix는 보통 다급하게 버그를 고치기 위해 생성되는 가지이기 때문에 버그를 해결하면 보통 제거하는 일회성 가지다.

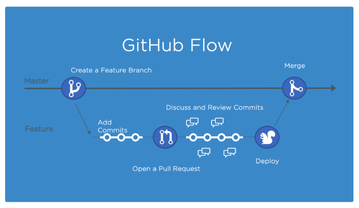
Git flow 흐름

- 앞에서 적었던 기본 구조 5개 중 가장 많이 사용되는 가지는 master와 develop가 되며 정상적인 프로젝트를 진행하기 위해서는 둘 모두를 운용해야 한다.
- 나머지 feature, release, hotfix branch는 사용하지 않는다면 지우더라도 오류가 발생하지 않기 때문에 깔끔한 프로젝트 진행을 원한다면 지워뒀다가 해당 가지를 활용해야 할 상황이 왔을 때 만들어줘도 괜찮다.
- 대부분의 작업은 develop에서 취합한다 생각하면 되며 테스트를 통해 정말 확실하게 더 이상 변동사항이 없다 싶을 때 master로의 병합을 진행하게 된다.
- master가 아닌 가지들은 master의 변동사항을 꾸준히 주시해야 한다.

출처 : 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

Branch 전략 2_GitHub Flow

GITHUB-FLOW 전략



- Git flow가 좋은 방식이지만 GitHub에 적용하기에는 **복잡**하다는 Scott Chacon의 판단에 따라 만들어진 새로운 것 관리 방식이다.
- **자동화 개념이 들어가 있다**라는 큰 특징이 존재하여 만일 자동화가 적용되어 있지 않은 곳에서만 수동으로 진행해야 한다.
- Git flow에 비해 흐름이 **단순해짐에 따라 그 규칙도 단순**해졌다.
- 기본적으로 master branch에 대한 규칙은 정확하게 정립되어 있다면 나머지 가지들에 대해서는 특별한 관여를 하지 않으며 pull request기능을 사용하도록 권장한다.

GitHub-Flow 특징

- release branch가 명확하게 구분되지 않은 시스템에서의 사용이 유용하다.
- GitHub 자체의 서비스 특성상 배포의 개념이 없는 시스템으로 되어있기 때문에 이 flow가 유용하다.
- 웹 서비스들에 배포의 개념이 없어지고 있는 추세이기 때문에 앞으로도 Git flow에 비해 사용하기에 더 수월할 것이다.
- hotfix와 가장 작은 기능을 구분하지 않는다.
모든 구분사항들도 결국 개발자가 전부 수정하는 일들 중 하나이기 때문이다.
이 대신 구분하는 것은 우선 순위가 어떤 것이 더 높은지에 대한 것이다.

Github-Flow 흐름

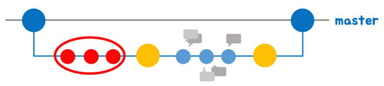
1. 브랜치 생성

Github-flow 전략은 기능 개발, 버그 픽스 등 어떤 이유로든 **새로운 브랜치를 생성**하는 것으로 시작된다.

단, 이때 체계적인 분류 없이 브랜치 하나에 의존하게 되기 때문에 **브랜치 이름을 통해 의도를 명확하게** 드러내는 것이 매우 중요하다.

- master 브랜치는 항상 최신 상태이, stable 상태로 product에 배포되는 브랜치다. 이 브랜치에 대해서는 엄격한 rule과 함께 사용한다
- **새로운 브랜치는 항상 master 브랜치에서 만든다**
- Git-flow와는 다르게 feature 브랜치나 develop 브랜치가 존재하지 않는다.
- 그렇지만, 새로운 기능을 추가하거나 버그를 해결하기 위한 **브랜치 이름은 자세하게** 어떤 일을 하고 있는지에 대해서 작성해주도록 하자

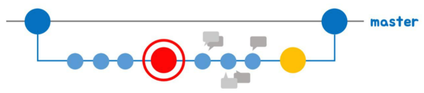
2. 개발 & 커밋 & 푸쉬



개발을 진행하면서 커밋을 남긴다.
이때도 브랜치와 같이 커밋 메세지에 의존해야 하기 때문에, 커밋 메세지를 최대한 상세하게 적어주는 것이 중요하다.

- **커밋메시지를 명확하게 작성하자**
- 원격지 브랜치로 **수시로 push** 하자
- Git-flow와 상반되는 방식
- 항상 원격지에 자신이 하고 있는 일들을 올려 다른 사람들도 확인할 수 있도록 해준다
- 이는 하드웨어에 문제가 발생해 작업하던 부분이 없어지더라도, 원격지에 있는 소스를 받아서 작업할 수 있도록 해준다

3. PR(Pull Request) 생성



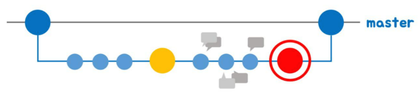
피드백이나 도움이 필요할 때, 그리고 merge 준비가 완료되었을 때는 pull request를 생성한다

- pull request는 **코드 리뷰**를 도와주는 시스템
- 이것을 이용해 자신의 코드를 공유하고, 리뷰받는다.
- merge 준비가 완료되었다면 master 브랜치로 반영을 요구한다.

4. 리뷰 & 토의

Pull-Request가 master 브랜치 쪽에 합쳐진다면 공장 라이브 서버에 배포되는 것과 다름 없으므로, 상세한 리뷰와 토의가 이루어져야 한다.

5. 테스트



리뷰와 토의가 끝났다면 해당 내용을 라이브 서버(혹은 테스트 환경)에 배포해본다.

배포시 문제가 발생한다면 공장 master 브랜치의 내용을 다시 배포하여 초기화 시킨다.

6. 최종 Merge

라이브 서버(혹은 테스트 환경)에 배포했음에도 문제가 발견되지 않았다면 **그대로 master 브랜치에 푸시를 하고, 즉시 배포**를 진행한다.

대부분의 Github-flow 에선 master 브랜치를 최신 브랜치라고 가정하기 때문에 배포 자동화 도구를 이용해서 Merge 즉시 배포를 시킨다.

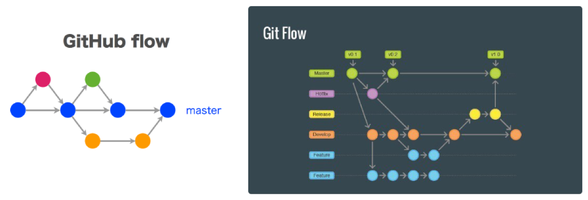
master로 merge되고 push 되었을 때는, 즉시 배포되어야한다

- GitHub-flow의 핵심
- master로 merge가 일어나면 자동으로 배포가 되도록 설정해놓는다. (CI / CD)

출처 : 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

Branch 전략 3. 비교

github flow vs git flow

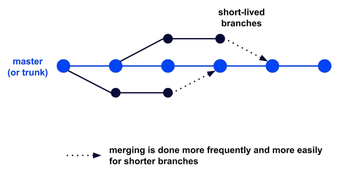


- 1개월 이상의 긴 호흡으로 개발하여 주기적으로 배포, QA 및 테스트, hotfix 등 수행할 수 있는 여력이 있는 팀이라면 **git-flow**가 적합하다
- 수시로 릴리즈 되어야 할 필요가 있는 서비스를 지속적으로 테스트하고 배포하는 팀이라면 **github-flow** 와 같은 간단한 work-flow가 적합하다

출처 : 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

Branch 전략 4. TBD Branch 전략

Trunk-based development



💡 트렁크 기반 개발(TBD: Trunk-Based Development) 브랜치의 흐름

1. main branch
- main 브랜치에서 시작합니다. 이 브랜치는 개발 중인 최신 코드를 포함하고 있습니다.

2. main branch
- 기능 개발을 위해 새로운 브랜치를 생성하지 않고, main 브랜치에서 직접 작업을 진행합니다.

3. main branch
- 기능 개발이 완료되면 개발한 기능을 브랜치의 '로컬 저장소'를 '원격 저장소'로 병합합니다.
- 이를 통해 새로운 기능이 main 브랜치에 통합됩니다.

4. main branch
- 테스트 및 리뷰를 거친 후 안정화된 코드를 프로덕션 환경에 배포합니다.

💡 트렁크 기반 개발(Trunk-Based Development)

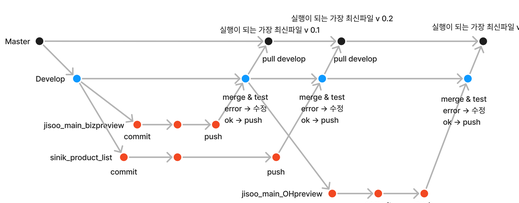
- 작은 규모의 프로젝트나 소규모 개발팀에서 주로 사용되며 릴리즈 주기가 짧은 프로젝트에 적합한 전략입니다.

- 단일 브랜치를 사용하여 개발을 진행하며 모든 개발자가 동일한 브랜치(트렁크)에서 작업을 합니다.
- 개발자는 자신이 작업한 변경사항을 주기적으로 커밋하고 다른 개발자의 변경사항과 통합합니다. 이로써 지속적인 통합과 릴리즈를 강조하며 브랜치 관리에 따른 복잡성을 줄이는 장점이 있습니다.

- 그러나 브랜치 충돌이 발생할 수 있고 변경 사항이 모든 개발자에게 바로 적용되기 때문에 신중하게 사용해야 합니다.
- 해당 전략은 'Hammant와 Dave Farley'가 제안한 방식으로 다른 Git workflow 모델입니다.

출처 : https://adjh54.tistory.com/364

Branch 전략 5. 그래서 우리는..?



안정성을 위해 git-flow branch 전략의 develop 부분을 가져오고 최종적인 구현 결과물이 필요한 프로젝트이고, 중간 또는 이후의 주기적인 배포가 필요 없기 때문에 release, hotfix 등의 과정은 생략합니다.

기능 구현을 위한 각 branch 는 Develop branch 를 기준으로 생성해서 작업 진행하고, Commit & Push 는 가능한 자주 하는 것이 좋다고 생각되지만 너무 잦지 않은 본인의 기준대로 작업해주시면 될 것 같습니다.

하필과 집에서 왔다갔다하며 작업하는 환경 특성상 오류가 있다 하더라도 Remote Repository 의 본인 Branch 에 push 하는 것은 가능 합니다.

이를 통해 다른 사람들도 각자의 작업진행을 알 수 있음.

하지만 기능 구현이 끝난 후의 Push 상태 에서는 관리자에게 ' Pull 요청 ' 해주시길 바랍니다.

ex) write 기능을 위한 writeView.jsp 생성, 기본적인 html 틀 완성 = commit (push 는 자유)
writeController.java 내부 코드 완성 → writeView 출력 가능 = commit (push 는 자유)
writeService.java 내부 코드 50% 완성 → Controller로 전달아짐 = (commit 자유, push 자유)
writeService.java 내부 코드 완성 → Controller로 반아짐 → writeView 에 출력 안됨 = (commit 자유, push 자유)

writeService.java 완성 → Controller 기능 작동 확인 → writeView 후력 됨 = commit, push, ' pull 요청 '

아직 GitHub.com 을 이용하는 것이 불편할 것이라 생각되서 Pull 요청은 요청 시점의 commit message 를 아래와 같이 작성하고, 단톡에 branch명 pull 요청한다고 말해주시면 좋을 것 같습니다.

Commit Message : ✅ [branch 이름] [pull request] [상세내용]
ex) ✅ jisoo_main_bizpreview pull request complete bizPreview
(gitmoji는 :white_check_mark: 입니다.)

Pull 요청 이후 관리자는 Develop branch 에서 요청한 branch 를 merge & test 하며, error 가 발생할 경우 해당 branch 개발자와 상의하여 오류를 해결하고, 오류 내용 및 결과를 모두에게 전달하는 방법으로 진행하겠습니다.
충돌없이 merge가 성공하거나 오류를 해결한 후 push를 한 이후 관리자는 Master branch 에서 Develop 의 최신 내용을 pull 해서 최신화 시키고, 해당 개발자에게 최신화가 완료됨을 알리겠습니다.

(Master Branch 와 Develop Branch 는 관리자만 수정 가능하게 옵션 걸어주세요)

이 작업이 끝난 후 개발자는 다시 Develop 에서 작업할 branch 를 새로 생성한 후 그 branch 에서 작업, commit, push, pull 요청 과정을 반복 진행해주시면 됩니다.

- 관리자 : (초기작업) ' Master ' Branch 에서 ' Develop ' Branch 생성
  → Master Branch 와 Develop Branch 는 관리자만 수정 가능하게 옵션 수정
  → (각 개발자 작업), 이 때 관리자는 현재 작업중인 branch 목록 숙지
  → Pull 요청 시 요청한 ' Remote Repository ' 의 branch 를 'Local Repository' 와 최신화 (Local [Branch명] 에서 Remote [Branch명] 을 Pull)
  → Local Repository 의 ' Develop Repository ' 와 해당 [Branch] 를 merge
  → 충돌 시 오류 수정, 충돌없이 merge 성공 시 ' Develop ' 을 push
  → Local 의 Master 에서 Remote Develop 을 Pull, 다시 Local Master 의 내용을 Remote Master 로 Push
  → 요청한 개발자에게 Develop 최신화 완료 message 전달.
- 개발자 : ' Develop ' Branch 를 기준으로 본인이 구현할 기능 이름으로 Branch 생성 후 작업
  → 작업 후 Commit, Push, Pull 요청
  → 관리자의 작업이 끝남
  → 본인의 'Local Repository' 의 ' Develop ' Branch 를 Remote Repository 의 ' Develop ' Branch 와 동기화 (Local Develop 에서 Remote Develop 을 Pull)
  → 다시 ' Develop ' Branch 를 기준으로 본인이 구현할 기능 이름으로 Branch 생성 후 작업

바로 Local 의 Develop 과 Remote 의 Branch 를 Pull 해도 별차상 상관은 없으나 만약을 대비해서 1단계 더 작업하는 것이 좋다고 판단했습니다.

Branch 전략 6. 주의사항

#주의사항

Branch 이름은 대소문자를 구분하지 않기때문에 단어 사이에 ' _ ' (밑줄) 로 구분해주세요

Pull 요청 외의 Commit Message Convention 은 기존의 규칙을 따르겠습니다.

Add 및 Commit 작업 시 본인이 작업한 파일들만 Staging 되어있는지, 본인이 작업하지 않은 파일들이 Modified 되어 있지 않은지 확인 후 실행해주세요. 혹시 작업하지 않은 파일들이 떠 있다면 해당 내용 파악 후 원상태로 복구시키는 게 Merge 시 Conflict 를 줄일 수 있습니다.

Pull 작업까지 끝난 Local Branch 는 삭제해도 상관없으나 삭제하더라도 만약의 상황을 위해 항상 Local file 로 BackUp 을 권장합니다.

Pull 작업이 끝난 Remote Branch 는 일단 보관할 생각입니다. 차후 모든 작업이 끝난 후 또는 해당 Branch의 유지이유가 없을 때에는 삭제할 예정입니다.

Pull 요청 후에는 수정사항이 있어서 Pull 완료 연락을 받기 전까지 재 Commit, Push 를 하거나 새 브랜치로 작업 하는 등의 행동은 지양 해주세요. 급하게 작업할 사항은 Local WorkSpace에서만 작업해주세요.

관리자는 Pull 작업 시 신중하고 천천히 작업하세도 됩니다.
Pull 작업 시 작업자가 구현한 기능이 완전히 작동하는지 확인해주세요.

바뀐 Branch Flow 때문에 이전처럼 매일 오전 오후시간의 Merge 는 없는 것으로 하겠습니다...만
가능하면 Pull 요청은 관리자도 같이 자리에 있을 때 요청해주세요.

+ 진행하다 생기는 사소한 오류들은 보통 내가 기억하지 못하는 클릭 이나 space bar, backspace 하나, WorkSpace 나 Server clean, Chrome에 쌓여 있는 쿠키, 잊어버린 Rename, 잊어버린 경로 변경, 오타 등의 이유로 일어날 텐데, 어라? 어째다 해결되었다? 하고 넘어가기 보다는 조금 더 확실히 해결한 후 잘 기억하고 숙지 또는 commit 에 기록 하는 것도 다른 코드와의 충돌을 예방하는 데에 좋을 것 같습니다.

Branch 전략 6. 예상 질문..?

Q. 다른 사람의 작업물이 필요한데, 제 Branch 에는 그 코드가 아직 반영이 안되어 있어요, 그러면 어떻게 하죠?
A. 그렇다면 Branch를 생성하는 시점에 필요한 그 사람의 code 가 없었을 거예요, 그렇다면 Branch 를 생성하는 시점에 해당 code가 필요한 기능이 아닌 다른 기능을 먼저 구현하고, 해당 개발자의 code 가 반영이 된 이후 Branch 를 생성해서 작업해주시면 될 것 같습니다.

+ 그렇다면 충돌하지 않는 2개 혹은 여러개의 기능을 구현하는 Branch 를 미리 파서 작업을 동시에 진행할 수도 있을 것 같네요 (굳이 그렇게 할 필요는 없겠지만요..? ㅎㅎ;;)

Q. 해야 하는 내용이 이해가 안되요.
A. 제가 다시 그림으로 그려드리겠습니닝, WorkSpace, Staging Area, Local Repository, Remote Repository 에 대한 개념이 잡히게 되면 조금 더 이해하기 쉬우실 거예요..