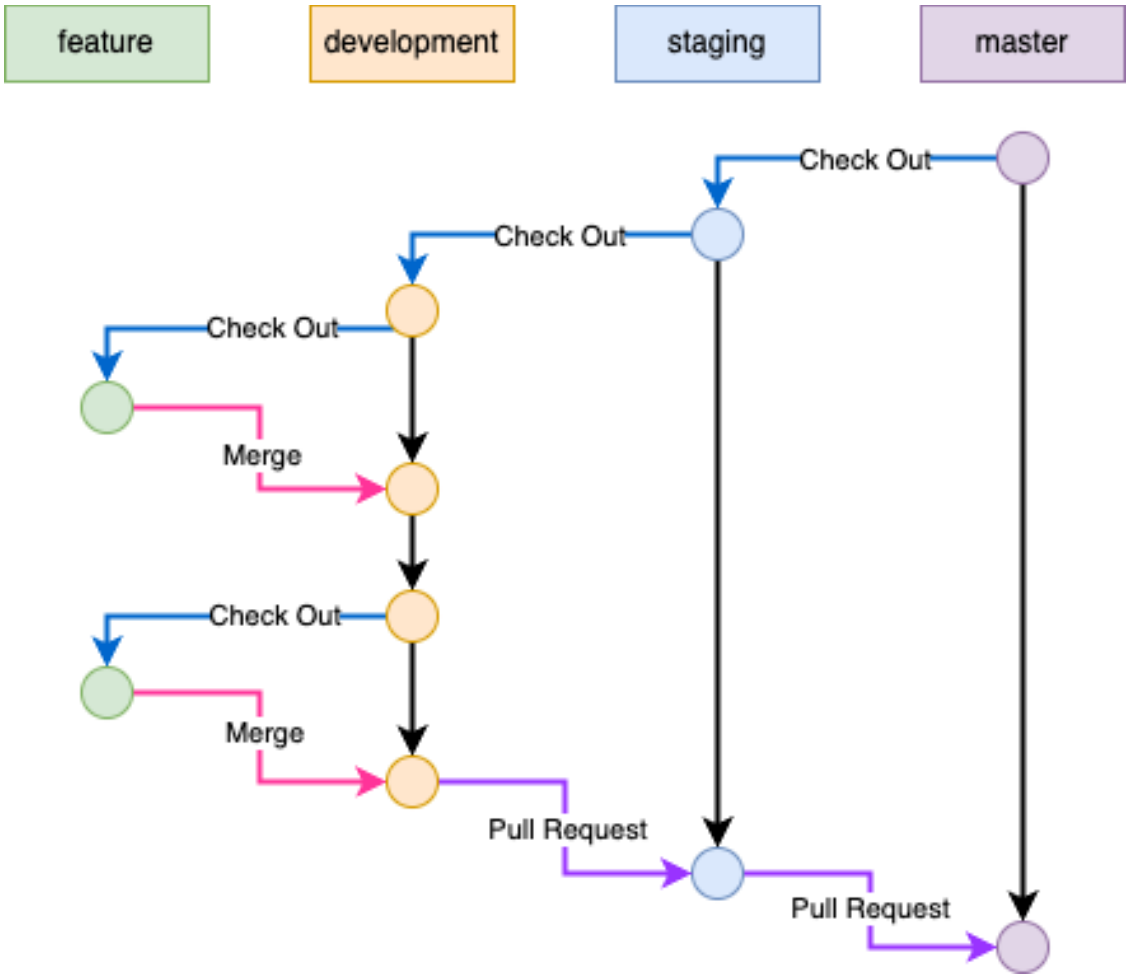
GIT Branch 전략 변경

1. 개요

현재 기획 된 git brach 전략을 개선하고자 새로운 git branch 전략을 제안합니다. 기존의 git branch 전략에서 문제로 제기 되었던 개발이 완료되지 않은 기능이 Master 브랜치를 통해 Production 환경에 배포 될 수 있는 시나리오가 존재 하였는대 이 부분을 보완하였습니다. 그리고 각 환경을 위해 development , staging, master 브랜치가 존재하였지만 불필요하다고 판단 되어 development , master 브랜치만을 운영하고도 development, staging, production 환경에 배포할 수 있는 방안에 대해 기술 하였습니다.

2. 기존 Git Branch 전략의 이슈 사항



2.1. 미완의 기능이 Production 환경에 배포 가능성

현재 Git Branch 전략은 feature -> development -> staging -> master 브랜치로 순차적으로 merge 되는 방식을 차용하고 있습니다. 이 Git Branch 전략을 운용하면서 대응하기 어려운 시나리오가 있다는 것을 확인하게 되었습니다

Production 환경에 배포하기 전 Staging 환경에서 테스트 하기 위해 Development 브랜치의 데이터를 Staging 브랜치로 Merge를 해야 합니다. 이 과정에서 모든 기능이 staging 브랜치로 Merge 되기를 원하지 않는 경우를 확인하게 되었습니다. 즉, development 브랜치에 Merge 된 데이터들은 feature 브랜치에서 개발 완료 된 버전들이 아닌 현재 개발중인 기능의 테스트를 Development 환경에서 테스트 하기위해 Megre 된 데이터로 확인하였습니다.

미 완성의 기능이 Production 환경에 배포 되었을 때 사용자들에게 버그를 그대로 제공할 수 있기에 이번 Git 정책 변경 시 이 부분을 보완할 방안을 생각하기로 하였습니다

2.2. 단계가 많은 브랜치

현재 Git Branch 전략에는 각 환경에 배포하기 위해 환경별 브랜치를 생성하여 운용하고 있습니다. 단순하게 보면 환경명과 브랜치명이 동일하기에 해당 브랜치가 어느 환경을 위한것인지 쉽게 확인이 가능합니다

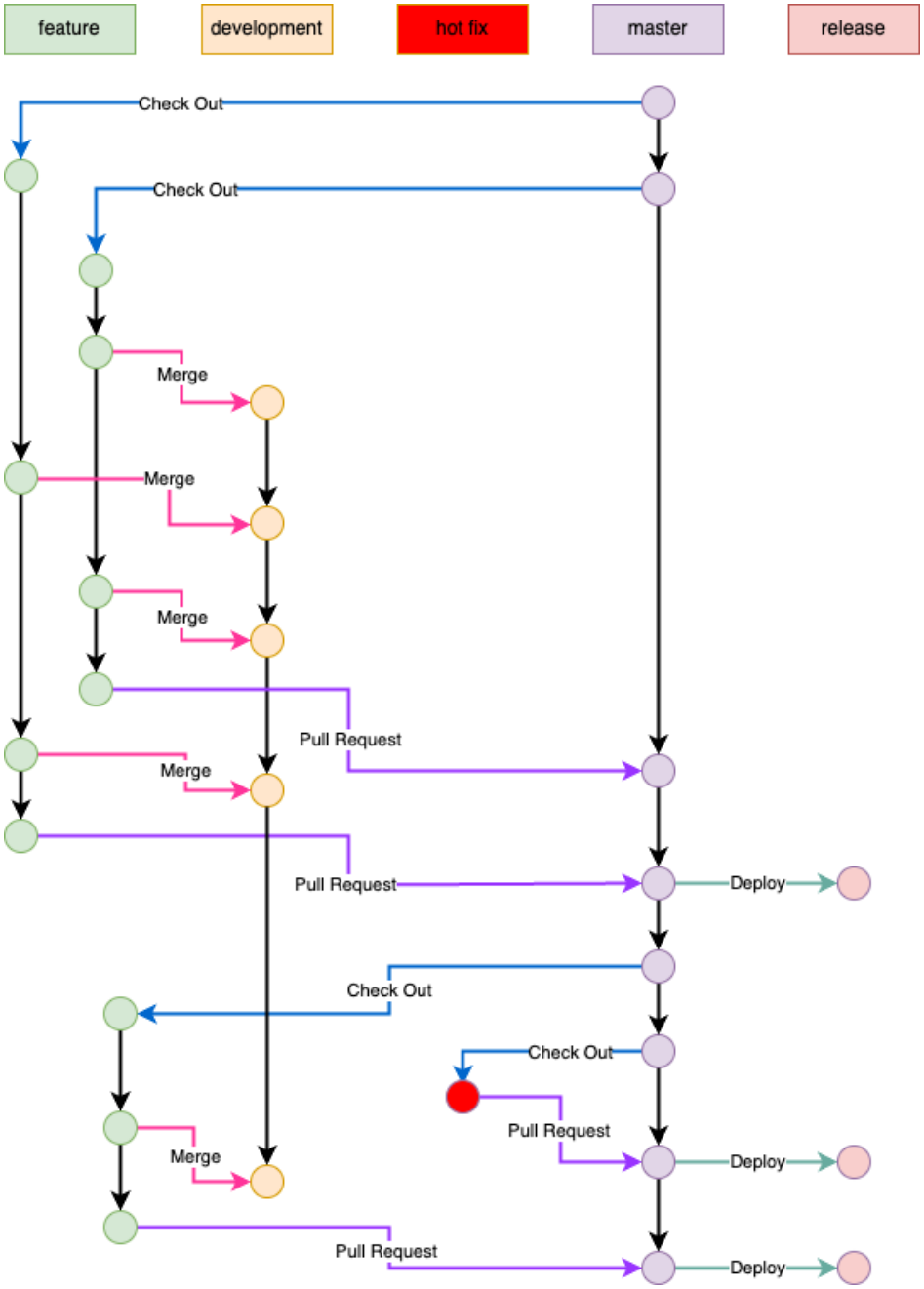
다만 이렇게 되면 불필요한 PR과 Check Out 단계 발생하게 된다는 것을 확인하였습니다. PR은 단순히 Staging, Production 환경에 배포하기 위해 과정이라 판단하였기에 현재 생성 된 대표 브랜치들을 줄이면서 Development, Staging, Production 환경에 배포할 수 있는 방법을 찾아야 한다고 생각했습니다

2.3. Feature 브랜치의 삭제 정책

현재 Git Branch 전략에서는 개발자들에게 별도의 Feature 브랜치 삭제 시기에 대해 별도로 고지를 하지 않았으며 삭제 시기는 개발자들이 자유롭게 본인의 판단하에 삭제를 하게끔 가이드 하였습니다.

이로 인해 삭제가 되지 않는 Feature 브랜치들이 우후죽순 발생하게 되었으며 기능 개발이 완료되었음에도 삭제가 되지 않아 Feature 브랜치들을 관리하지 못하는 현상이 확인되어 이를 개선해야 할 방안을 찾아야 한다고 판단하였습니다

3. 개선 된 Git Branch 전략

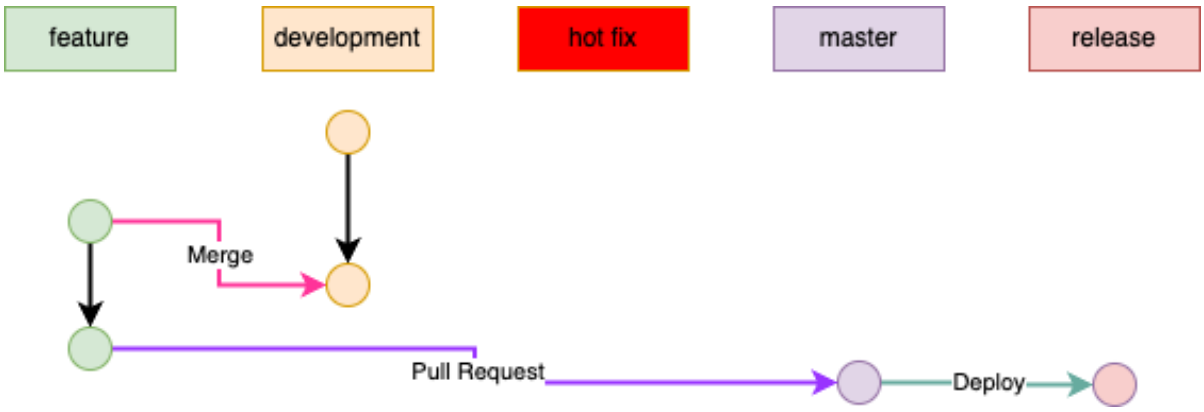


기존 Git Branch 전략에서 제기되었던 문제점들을 보완하기 위해 위와 같은 Git 흐름 구조를 그리게 되었습니다. 환경 별 배포하기 위해 존재하였던 Development, Staging, Master 브랜치에서 Development, Master 브랜치만 유지하는 것으로 변경하였습니다.

그리고 개발이 완료 된 기능만 Master 브랜치에 병합되어 Staging, Production 환경에 배포할 수 있게 하였습니다

추가적으로 Feature 브랜치를 삭제 할 시기를 정의하여 브랜치를 좀 더 용이하게 관리 할 수 있도록 하였습니다

3.1. 브랜치 항목 개선



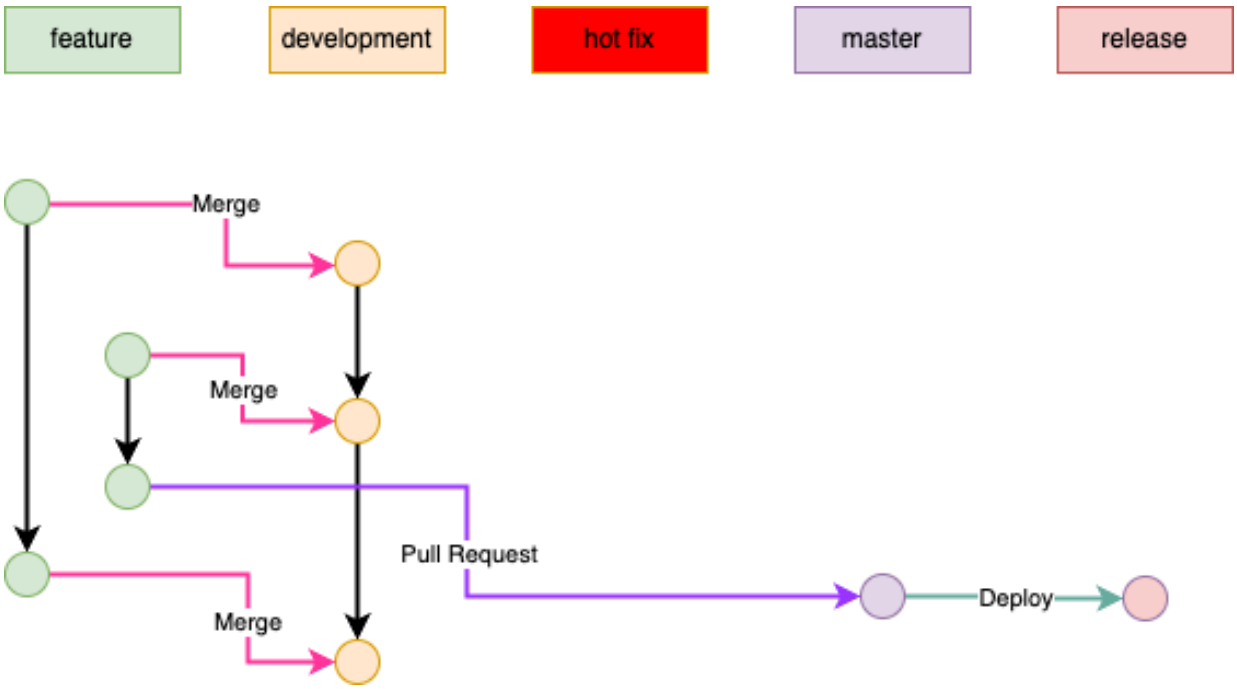
이번 버전의 Git Branch 전략에서 브랜치 운용은 feature, development, master, hot fix 이 네가지의 종류로 운용하고자 합니다.

기존 전략 대비 가장 큰 차이점은 staging 브랜치를 제거하였다는 점입니다. 이전 전략에서는 배포 할 대상의 환경과 브랜치간 1:1로 유지하기 위해 development, staging, master 브랜치가 존재하였고 각각 development, staging, production 환경에 배포하기 위한 브랜치 들이었습니다.

각 환경과 브랜치간 1:1로 매핑하였다는 것에 의의가 있지만 staging, production 환경에 배포해야 할 경우 PR (Pull Request) 이 발생 (development -> staging, staging -> master) 하게 됩니다. PR은 다른 브랜치로 코드를 Merge 하기 위한 작업으로 이 작업은 상위 개발자나 관리자의 승인이 있어야 합니다. staging 환경에 어플리케이션을 배포할 때는 PR이 한번 발생하지만 production 환경에 어플리케이션을 배포하기 위해서는 PR이 총 두 번 발생하게 되는 대 동일한 코드로 동일한 작업을 두 번 진행해야 하는 불 필요한 프로세스가 있다는 것을 확인하게 되었습니다

불 필요한 PR을 두 번해야 하는 프로세스를 개선하기 위해 staging 브랜치를 제거하기로 하였습니다. staging 브랜치를 제거함에 따라 개발자들은 feature 브랜치를 통해 개발 완료 된 기능은 master 브랜치로 한 번의 Pull Request 작업만 진행하면 됩니다. PR 작업이 마무리 되면 staging 환경에 어플리케이션 배포가 이루어지고 테스트까지 마무리가 되면 프로젝트 관리자는 ecode github repository의 Actions 페이지에서 Production 환경 배포 스크립트를 실행합니다

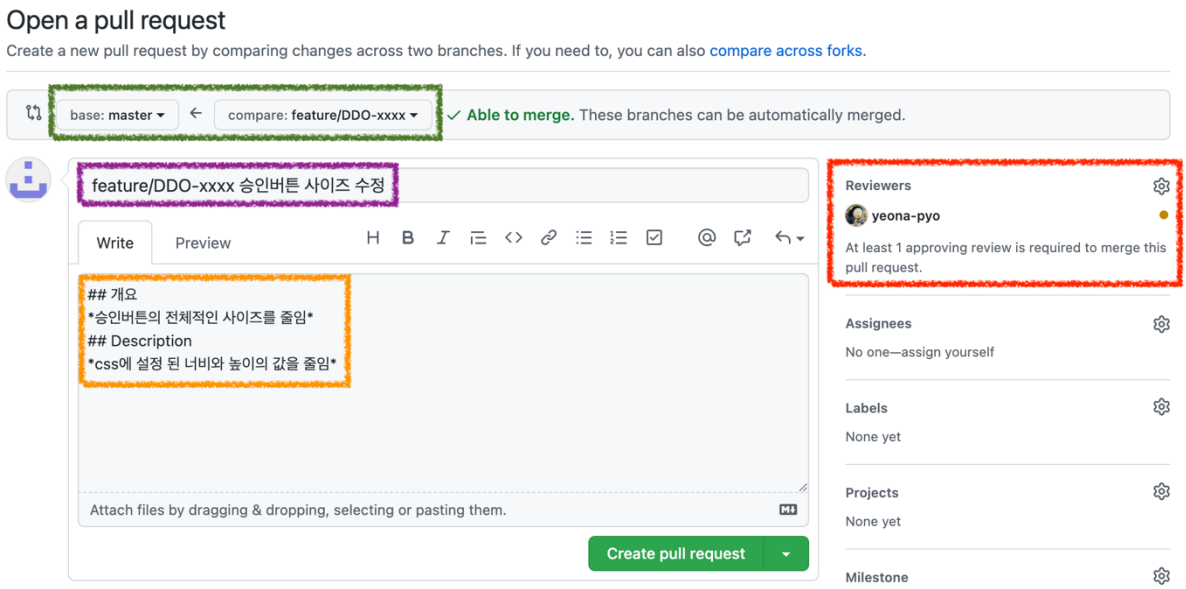
3.2. 개발이 완료 된 Feature들만 배포



기존 Git Branch 전략에서는 구조적인 문제로 개발이 완료 되지 않은 기능이 Production에 배포 될 시나리오가 존재하였습니다. 이 문제를 보완하기 위해 개발이 완료 된 Feature 들만 배포가 되면서 테스트도 진행 할 수 있는 방안을 찾아보았습니다.

Feature 브랜치에서 개발 중인 기능은 수시로 development 브랜치와 merge하여 development 환경에서 테스트를 진행합니다. 이 후 development 브랜치의 데이터가 다른 브랜치와 병합하거나 PR 작업을 진행하는 것이 아닌 개발자는 feature 브랜치에서 바로 master 브랜치에 PR을 보내야 합니다.

PR 보내는 방법은 아래의 내용을 참고하여 관리자에게 PR을 보내면 됩니다



초록색 네모

- 어느 브랜치에서 어느 브랜치로 PR을 할 것인지 정의 합니다
- base 란에는 master 브랜치로 설정합니다
- compare 란에는 개발자가 개발한 feature 브랜치로 설정합니다

보라색 네모

- PR 타이틀 명입니다
- Feature/[JIRA 번호] [JIRA 타이틀명] 으로 PR 타이틀 명으로 적어주세요

주황색 네모

- PR 내용을 적어주시면 됩니다
- 개요 란에는 작업한 내용을 간략하게 적어주시면 됩니다
- Description 란에는 관리자가 작업 내용을 알기 쉽게 기능 개발을 위해 어떤 작업을 하였는지 적어주시면 됩니다

빨간색 네모

- 리뷰어를 지정해 줍니다
- 리뷰어는 Project Manager로 지정해 줍니다

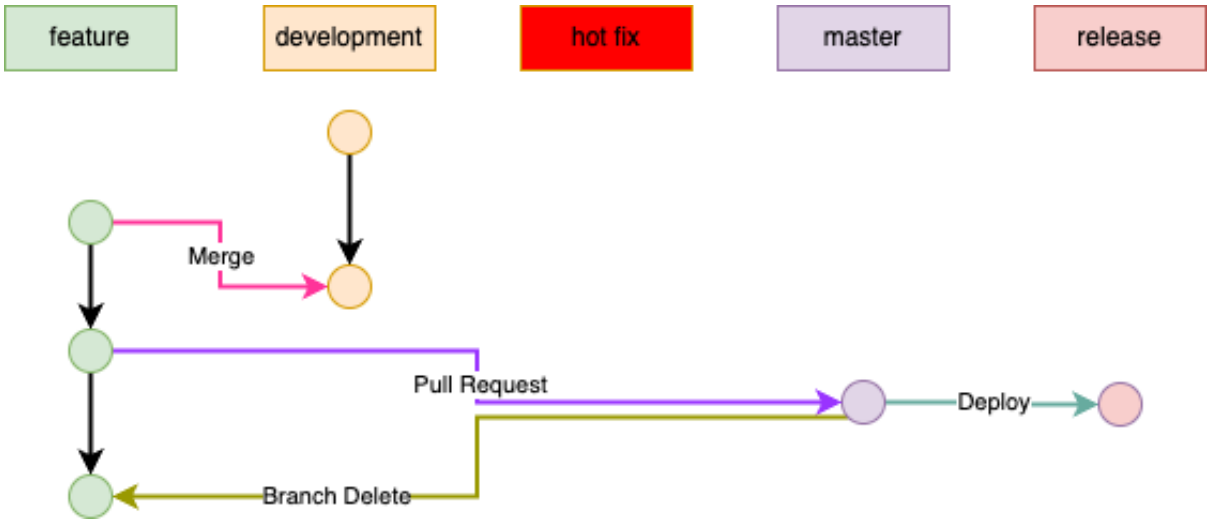
PR에 대한 승인은 관리자의 참여하에 개발자는 관리자에게 개발 한 기능에 대한 설명 및 구현 된 부분을 애기합니다. 관리자는 개발자가 애기한 내용을 기반으로 PR 승인 여부를 결정합니다.

Master 브랜치로 PR 된 기능은 Staging 환경에 바로 배포되어 Production 환경에

배포 되기 전 테스트를 진행합니다.

Staging 환경에서 테스트가 마무리 되면 release 단계로 Prodction 환경에 배포를 진행하며 수정해야 할 사항이 있으면 master 브랜치에서 feature 브랜치를 생성하여 수정 작업을 진행합니다

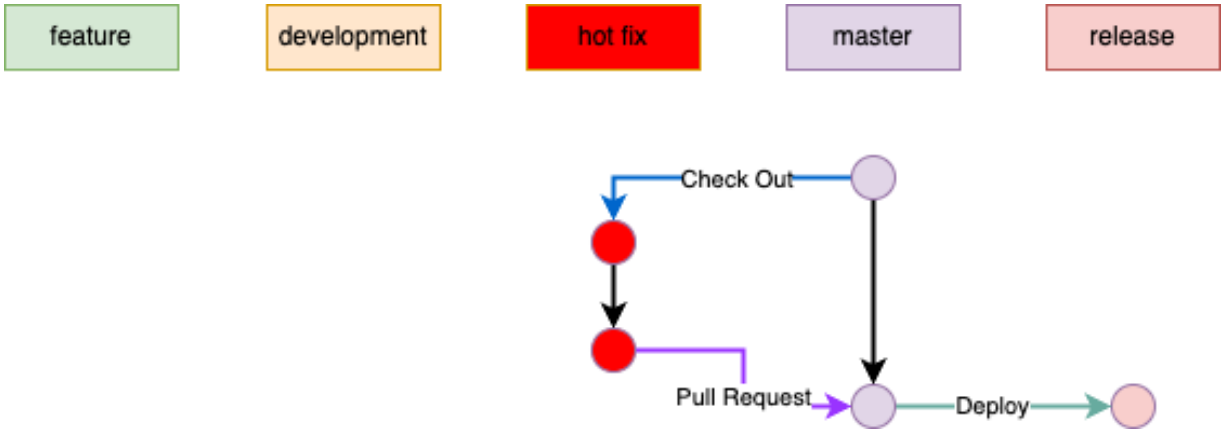
3.3. Feature 브랜치의 수명 주기



기존 Branch 전략에서는 Feature 브랜치에 대해 언제 삭제 할 것인지에 대한 가이드가 없어서 현재 사용되지 않지만 삭제되지 않아 계속 방치중이거나 별도의 Feature 브랜치 생성 없이 계속 동일한 Feature 브랜치에서 기능 개발을 하고 있는 것을 확인하였습니다.

이에 이번 Branch 전략에서는 Feature 브랜치에 대한 삭제 시기를 가이드 하고자 합니다. Feature 브랜치의 삭제는 Feature 브랜치에서 Master 브랜치로 요청 한 PR이 완료하면 이 때 Feature 브랜치를 삭제해야 합니다. Feature 브랜치의 삭제는 관리자가 진행을 해주시면 됩니다.

3.4. Hotfix 브랜치의 주기

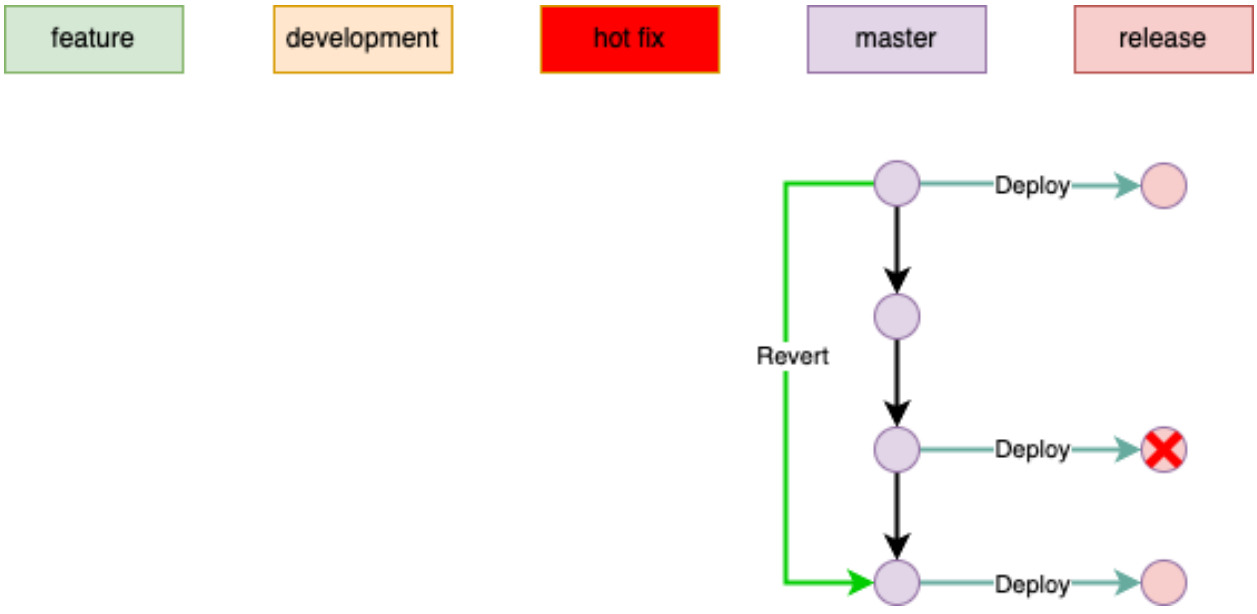


Hotfix 브랜치에 대한 방안도 기존 Branch 전략에서는 명시되지 않았던 부분으로 Hotfix 브랜치 방안도 이번 Brach 전략에 포함 시켰습니다.

Hotfix 브랜치는 Production 환경에 배포 된 특정 기능을 긴급하게 수정 후 배포하기 위한 목적이기에 Master 브랜치에서 hotfix 브랜치를 생성하고 기능 수정 작업을 진행하면 됩니다.

완성 된 기능은 master 브랜치로 PR을 요청 하여 요청이 완료 되면 staging 환경에 배포하고 테스트가 완료 되면 production 환경에 배포 작업을 진행하면 됩니다

3.5. Master Branch 복원 방안



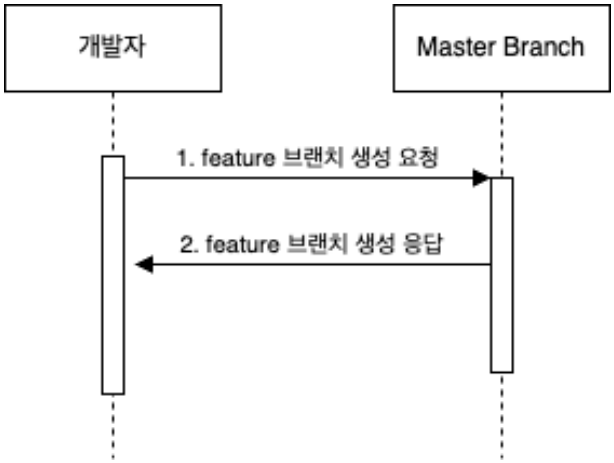
Production에 배포 된 버전에 문제가 확인 되었을 때 hot fix로 수정 버전을 진행하는 것이 아닌

아예 이전 버전으로의 복원 방안에 대한 것도 생각해 보았습니다
복원 해야한다는 결정이 나오면 복원 시점은 최신의 기능이 Master 브랜치로 PR 된 시점이 아닌 Production 환경에 배포 된 버전 기준으로 이전 버전으로 복원을 진행합니다.

4. 프로세스 별 흐름

Git Branch 전략과 관련 된 프로세스들은 아래와 같습니다.

4.1. Feature 브랜치 생성 프로세스



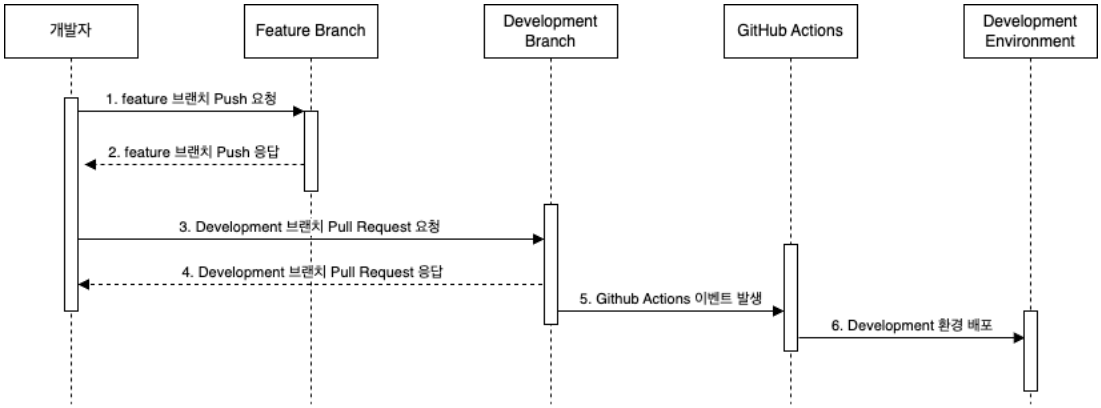
1. feature 브랜치 생성 요청

- **git branch feature/DDO-xxxx master** 명령어를 이용하여 master를 부모 브랜치로 하여 feature 브랜치를 자식 브랜치로 생성 한다.

2. Feature 브랜치 생성 응답

- **git checkout feature/DDO-xxxx** 명령어로 생성한 브랜치로 전환 한다
- **git branch** 명령어를 이용하여 내가 생성한 feature 브랜치로 전환되였는지 확인한다

4.2. Development 환경 배포 프로세스



1. feature 브랜치 Push 요청

- **git add .**
  git commit -m "commit 내용"
  git push origin feature/DDO-xxxx
  위의 명령어를 이용하여 feature 브랜치에 개발한 내용을 push 한다

2. feature 브랜치 push 응답

- **git push origin feature/DDO-xxxx** 명령어 실행 후 발생하는 응답에 대해 확인 한다.

3. Development 브랜치 Pull Request 요청

- 개발자는 git repository 페이지에서 Pull Request 를 생성하여 feature/DDO-xxxx 브랜치의 데이터를 Development 브랜치와 Merge 작업을 진행한다

4. Development 브랜치 Pull Request 응답

- Development 브랜치와 Merge 작업이 정상적으로 완료 되면 다음 단계로 넘어가게 된다

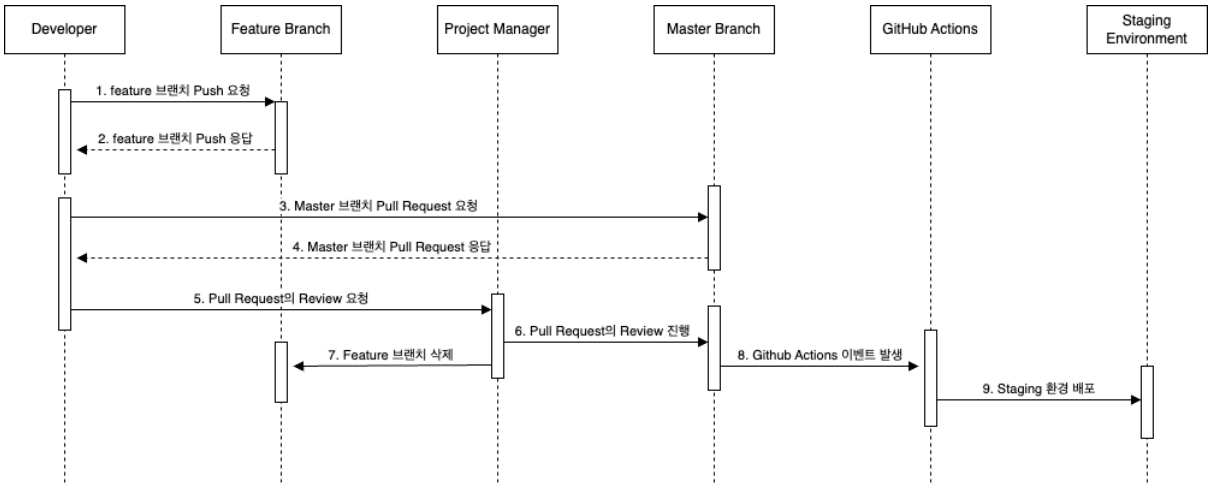
5. Github Actions 이벤트 발생

- Development 환경에 배포 하는 스크립트가 자동적으로 실행 되어 어플리케이션을 빌드하고 배포 작업이 진행 된다

6. Development 환경 배포

- Github Actions 의 workflow가 정상적으로 완료 되면 개발자는Development 환경에 배포 된 결과를 확인한다

4.3. Staging 환경 배포 프로세스



1. feature 브랜치 Push 요청
   - **git add .**
     git commit -m "commit 내용"
     git push origin feature/DDO-xxxx
     위의 명령어를 이용하여 feature 브랜치에 개발한 내용을 push 한다

2. feature 브랜치 push 응답
   - **git push origin feature/DDO-xxxx** 명령어 실행 후 발생하는 응답에 대해 확인 한다.

3. Master 브랜치 Pull Request 요청
   - 개발자는 git repository 페이지에서 Pull Request 를 생성하고 리뷰어로 Project Manager를 지정한다
   - 개발자는 feature/DDO-xxxx 브랜치의 데이터를 Master 브랜치와 Merge 작업을 생성한다.

4. Master 브랜치 Pull Request 응답
   - Development 브랜치와 Merge 작업이 정상적으로 완료 되면 다음 단계로 넘어가게 된다

5. Pull Request 의 Review 요청
   - 개발자는 Project Manager (PM)에게 리뷰 요청을 진행합니다

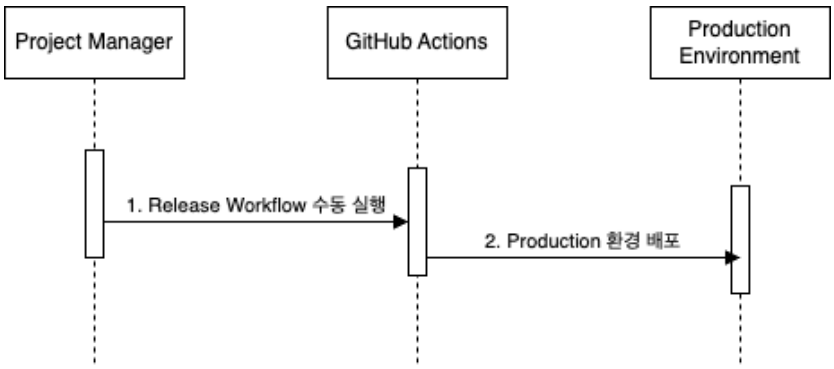
6. Pull Request 의 Review 진행
   - Project Manager (PM)은 개발자가 개발한 기능에 대한 리뷰를 진행합니다
   - 리뷰를 완료하였다면 PM은 리뷰에 대한 내용을 작성하고 승인 작업을 진행합니다

7. Feature 브랜치 삭제
   - Review 진행을 완료하였다면 Project Manager(PM)은 개발자가 생성한 Feature 브랜치를 삭제 합니다.
   - Feature 브랜치는 개발자가 요청 한 Pull Request 페이지에서 삭제합니다

8. Github Actions 이벤트 발생
   - Staging 환경에 배포 하는 스크립트가 자동적으로 실행 되어 어플리케이션을 빌드하고 배포 작업이 진행 된다

9. Staging 환경 배포
   - Github Actions 의 workflow가 정상적으로 완료 되면 개발자는 Staging 환경에 배포 된 결과를 확인한다

4.4. Production 배포 프로세스

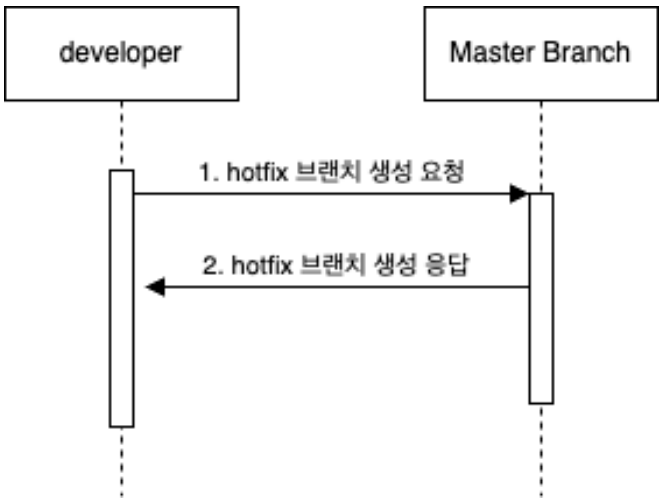


1. Release Workflow 수동 실행
   - Project Manager (PM) 은 git repository 의 Actions 페이지에서 'Release' workflow를 수동으로 실행 해준다

2. **Production 환경 배포**
   - Github Actions 의 workflow가 정상적으로 완료 되면 개발자는 Production

환경에 배포 된 결과를 확인한다

4.5. Hotfix 브랜치 생성 프로세스



1. hotfix 브랜치 생성 요청
   - **git branch hotfix/DDO-xxxx master** 명령어를 이용하여 master를 부모 브랜치로 하여 hotfix 브랜치를 자식 브랜치로 생성 한다.
2. hotfix 브랜치 생성 응답
   - **git checkout hotfix/DDO-xxxx** 명령어로 생성한 브랜치로 전환 한다
   - **git branch** 명령어를 이용하여 내가 생성한 hotfix 브랜치로 전환되였는지 확인한다

5. 향후 개선해야 할 사항

1. 자동화로 현 배포 프로세스 보완
   - Feature 브랜치에 Push 행위가 발생 했을때 바로 Development 환경에 배포 될 수 있게 배포 프로세스 변경 필요
2. Review 알람 기능 보완
   - 현재는 개발자가 개발 한 사항을 Project Manager에게 Review 요청을 하게 되면 Knox 메일로 알람을 수신 받게 되어 있다. RBS을 띄우고 Knox 포탈을 활성화 하였다면 Review 요청 온 것에 대해 확인이 가능하지만 항시 확인이 가능한 환경이 아니다

- 이에 슬랙이나 Gmail Chat을 이용하여 개발자가 리뷰 요청한 것에 대해 알람을 받을 수 있도록 구성이 필요하다