Git & Gerrit 교육과정

이론과 실습

1.1 Git 의 배경

이론

교육 시작 전에...

이번 교육내용은 Perforce등 기존 Tool에 익숙한 개발 리더들을 대상으로
Git을 왜 도입해야하는 지에 대한 공감대를 만들고
자연스럽게 Git이 퍼져나가도록 하기 위해
만들어진 과정입니다.

손에 익지 않은 도구는 쓸모없는 도구입니다.
Git과 Gerrit이 아무리 개념이 좋더라도 손에 익지 않음 무용지물이니,
이번 교육을 통해 재미있게 개념도 공유하고 손에 익혀 보아요.

들어가며...

전세계를 주도하는 무선 사업부 개발자...

지금까지 Perforce를 능숙하게 다루던능력자들이라면

Git이 뭐야? 뭐긴 하겠지만... 난 깃알못!!!

Git을 도입하겠다는 이야기를 듣는다면 이런 표정 아닐까요?

하지만 GIT을 우습게 보다간

이렇게 열받고, 눈커지고, 소리지르게 될 일이 있을 지 모릅니다.

그래서 기존에 비해 Git이 어떻게 달라지는지 같이 보도록 할께요.

참고 사이트

볼만한 사이트 들 알려드릴께요.

튜토리얼

- 본 교육 교재 및 강의 관련 자료: http://wiki.twoseed.co.kr
- GIT 을 시작하기 위한 간편 안내서 어렵지 않아요 : http://rogerdudler.github.io/git-guide/index.ko.html
- Git-SCM : https://git-scm.com/book/ko/v1
- Atlassian Git Tutorial : https://www.atlassian.com/git

참고 자료 & 인터뷰

- 버전관리를들어본적없는사람들을위한 DVCS-GIT : https://www.slideshare.net/ibare/dvcs-git
- SVN 능력자를 위한 개념 가이드 엄청난 간략 비교 : https://www.slideshare.net/einsub/svn-git-17386752
- Subversion Vs. Git 엄청난 간략 비교 : https://www.slideshare.net/ienvyou/subversion-vs-git-42605130
- Rebase는 언제 어떻게 해야 할까? : http://dogfeet.github.io/articles/2012/git-merge-rebase.html
- 완전 초보를 위한 깃허브 : https://nolboo.github.io/blog/2013/10/06/github-for-beginner/
- 리누스 토발즈 인터뷰 : http://techholic.co.kr/archives/31767

Git != Perforce,SVN

GIT을 처음 들어보면 이런 생각들이 들 수 있습니다.

- ✓ 요즘 오픈 소스에서 GIT이 뜨는 이유는?
- ✓ GitHub는 또 뭐냐? GIT이랑 뭐가 다른가?
- ✓ 자료를 읽다보니, inline command가 많은 데, GiT쓸려면 이거 다 알아야 하나?
- ✓ Branch, Tag ???
- ✓ Merge, Rebase 이건 또 뭐냐...
- ✓ Push, Pull, Pull Request ???
- ✓ ...

- 또 P4와 비슷한 것같지만 막상 해보면 많이 달라서,
  직접 겪어보면 아리송한 게 꽤 많다는 거죠.
- 일단 GIT이 뭐길래 깃깃 거리는 지... 지금까지 잘 쓰던 도구를 왜 버리고
  Git으로 가야한다는 건 지 같이 둘러보겠습니다.

짧게 보는 Git의 역사

"우리네 삶의 삼라만상처럼 Git 또한 창조적 파괴와 활활 타오르는 갈등 속에서 시작됐다."

VCS가 없던 다크 에이지

VCS가 없던 다크 에이지 시절 한 번 보시죠.

- 파일 이름에 날짜, 버전(v1,v2..)를 붙여서 복사 • tar등으로 압축하여 통재로 복사
- 변경된 부분에 대해서 소스코드 비교
- 실수로 인한 코드 유실
- \rm -rf
- 사장님은 그져 백업이 중요하다고만 하셨어~

Git - Let there be light

Let there be light

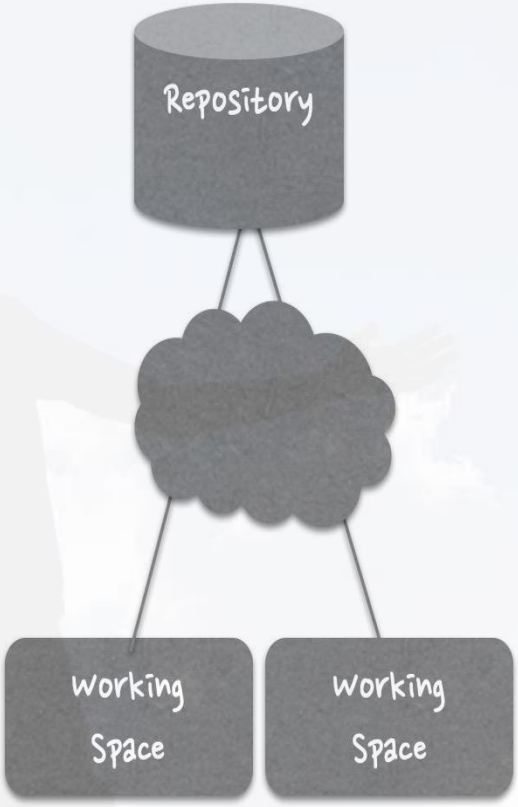
Git 10 years

- Linux 커널은 굉장히 규모가 큰 오픈소스 프로젝트다.
- Linux 커널의 삶 대부분은(1991–2002) Patch와 단순 압축 파일로만 관리했다.
- 2002년에 드디어 Linux 커널은 BitKeeper라고 불리는 상용 DVCS를 사용하기 시작했다.
- 2005년에 커뮤니티가 만드는 Linux 커널과 이익을 추구하는 회사가 개발한 BitKeeper의 관계는 틀어지고 BitKeeper와 Linux와의 관계는 안드로메다로...
- 이 사건은 Linux 개발 커뮤니티(특히 Linux 창시자 Linus Torvalds)가 자체 도구를 만드는 계기가 됐다.
- GitHub(2008년경)로 인해서 폭발적으로 성장함. 그 뒤 GitLab, BitBucket등 다양한 철학을 가진
  → **리눅스 창시자 리누스 토발즈 형이 만든 분산 버전 관리 시스템 (DVCS)**
- git 10 YEARS - https://www.atlassian.com/git/articles/10-years-of-git

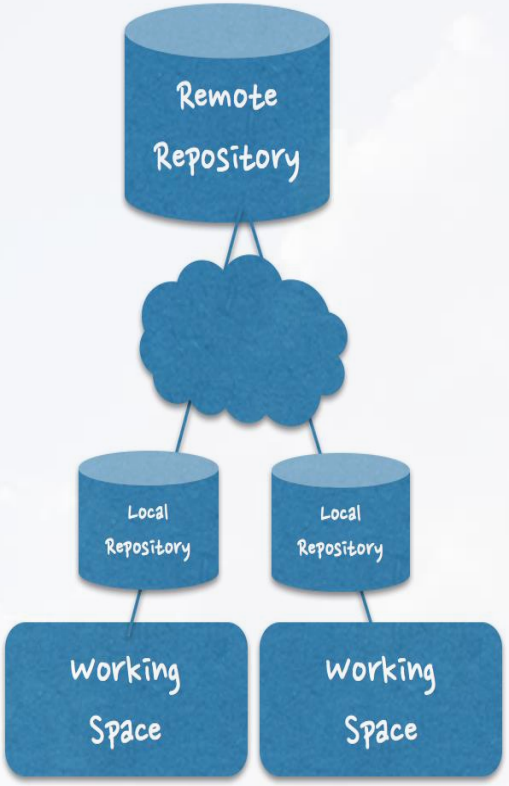
As is Git

Git의 현재

- ✓ Git은 2005년 탄생하고 나서 아직도 초기 철학를 그대로 유지하고 있다.
- ✓ 그러면서도 사용하기 쉽게 진화하고 성숙했다.
- ✓ Git은 미친 듯이 빨라서 대형 프로젝트에 사용하기도 좋다.
- ✓ Git은 동시다발적인 브랜치에도 끄떡없는 슈퍼 울트라 브랜칭 시스템이다

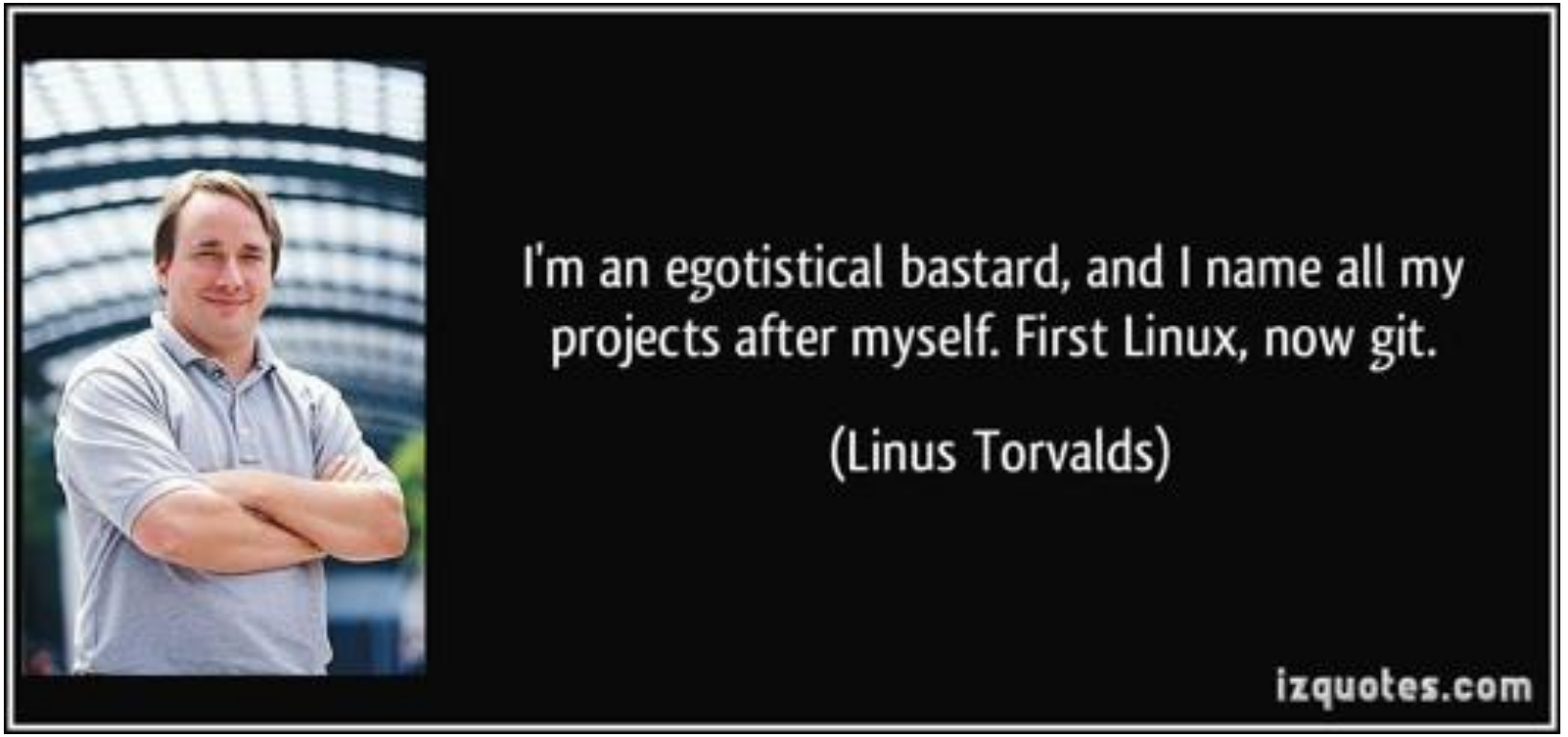


중앙집중형



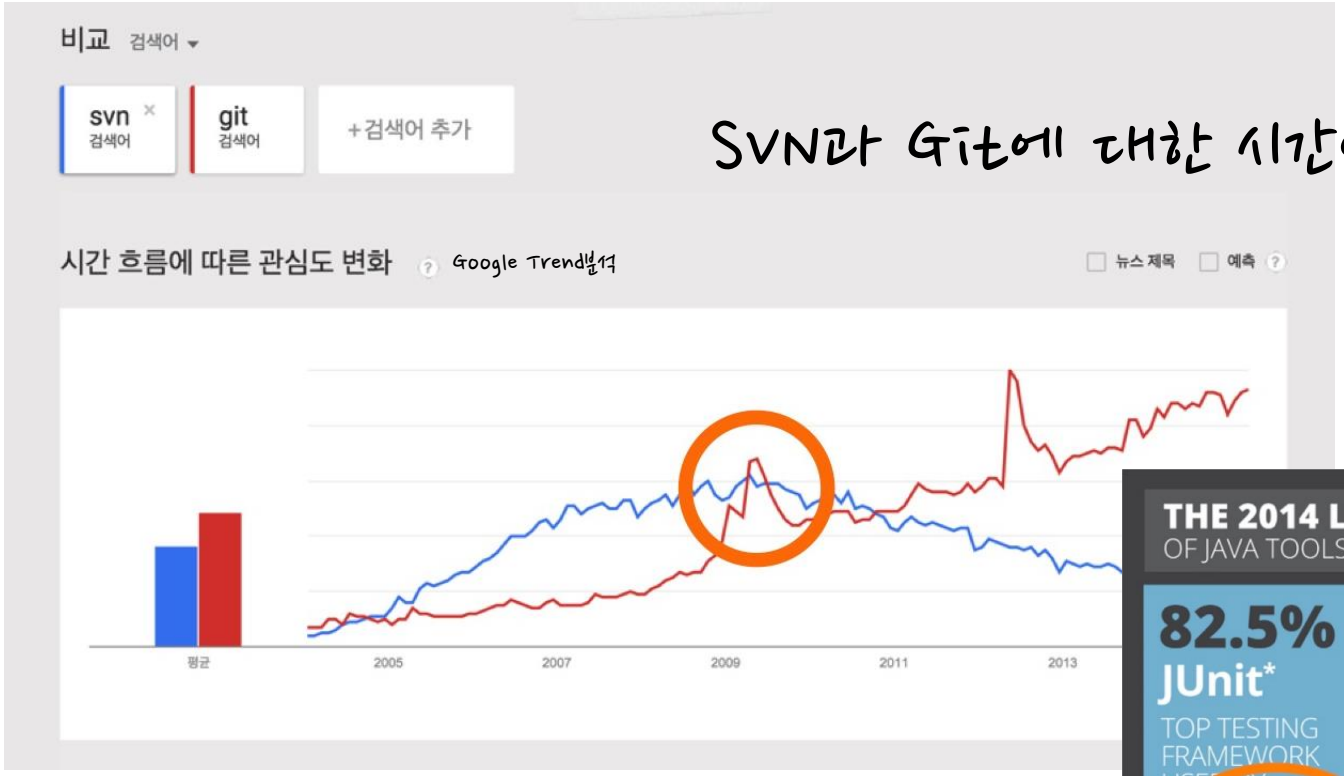
분산형

Global Information Tracker?

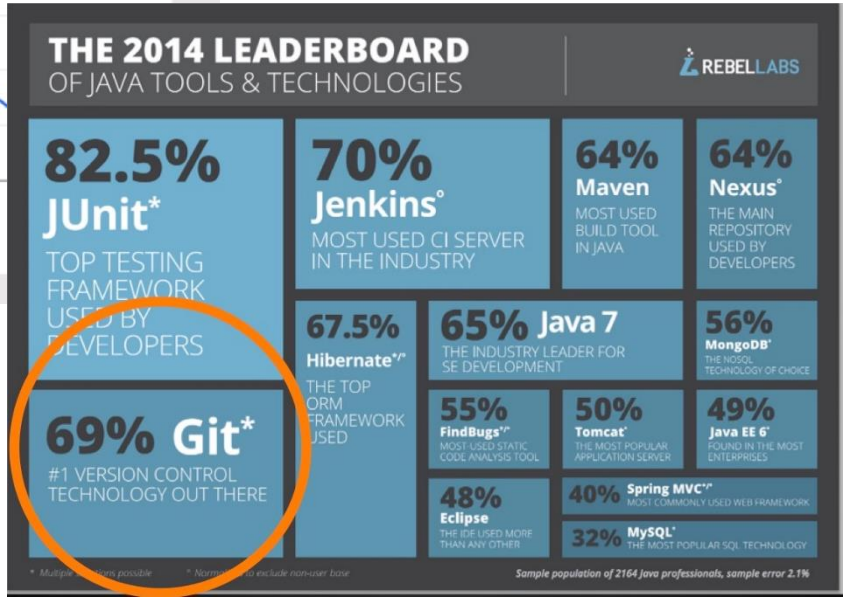


- **리누스 토발즈 인터뷰**(http://techholic.co.kr/archives/31767) **도 한 번 보셔요**.
- random three-letter combination that is pronounceable, and not actually used by any common UNIX command.
- "global information tracker": you're in a good mood, and it actually works for you. Angels sing, and a light suddenly fills the room.
- "g*dd*mn idiotic truckload of sh*t": when it breaks

Why Git?



SVN과 Git에 대한 시간에 따른 관심도 변화



Leaderboard of Java Tools

So many books, info... but...

주위에서 흔히 볼 수 있는 git 가이드들은 무척 친절합니다. 하지만, 다른 도구 숙련자들에게는 오히려 혼란스럽습니다.

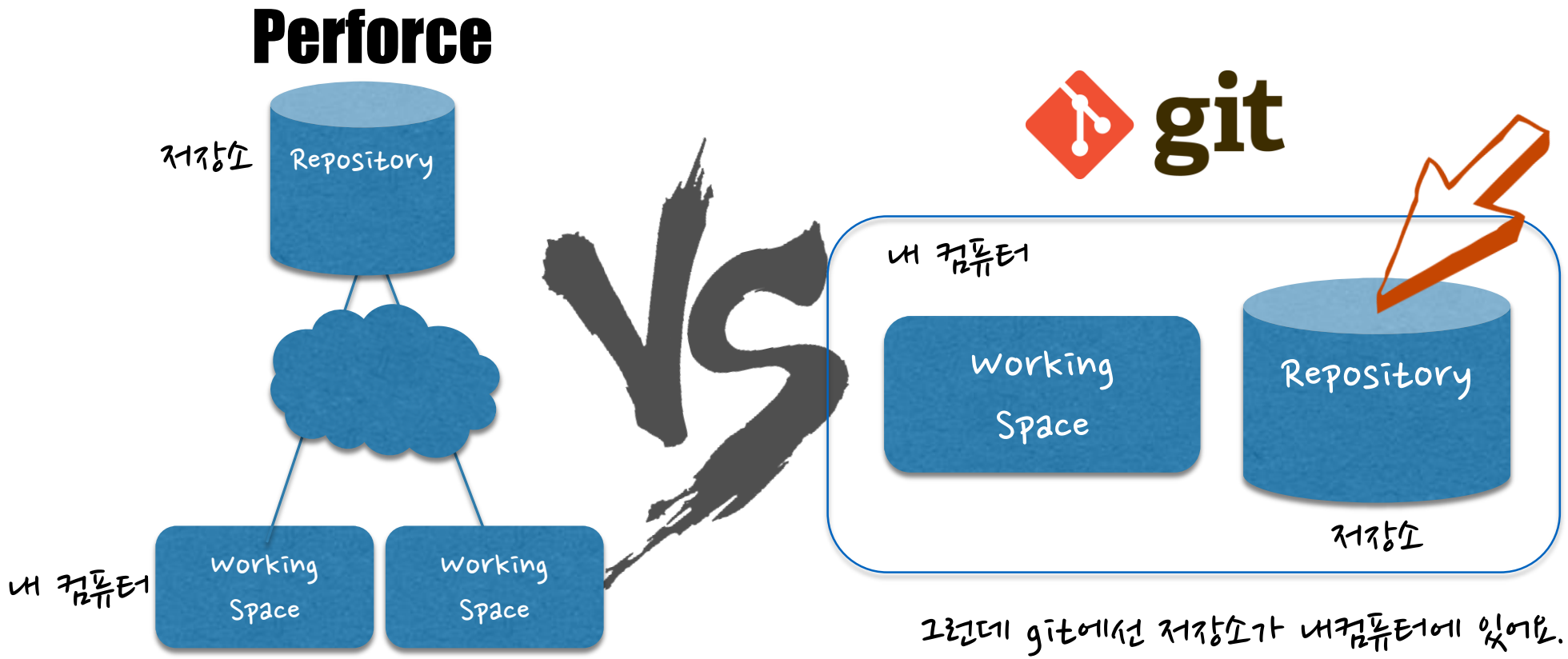
왜냐면, 다 아는 것같은 데 조금 다르고... 지향하는 철학이 다르다보니
why란 부분에 대해 깊은(?) 공감을 하지 않으면 왜 이런 닭질을 하느냐라는 것처럼 보이거든요.
그래서 Git의 철학과 다른 도구를 계속 비교해서 알려드리겠습니다.

1.2 Git 의 구조

이론

Perforce Vs. git

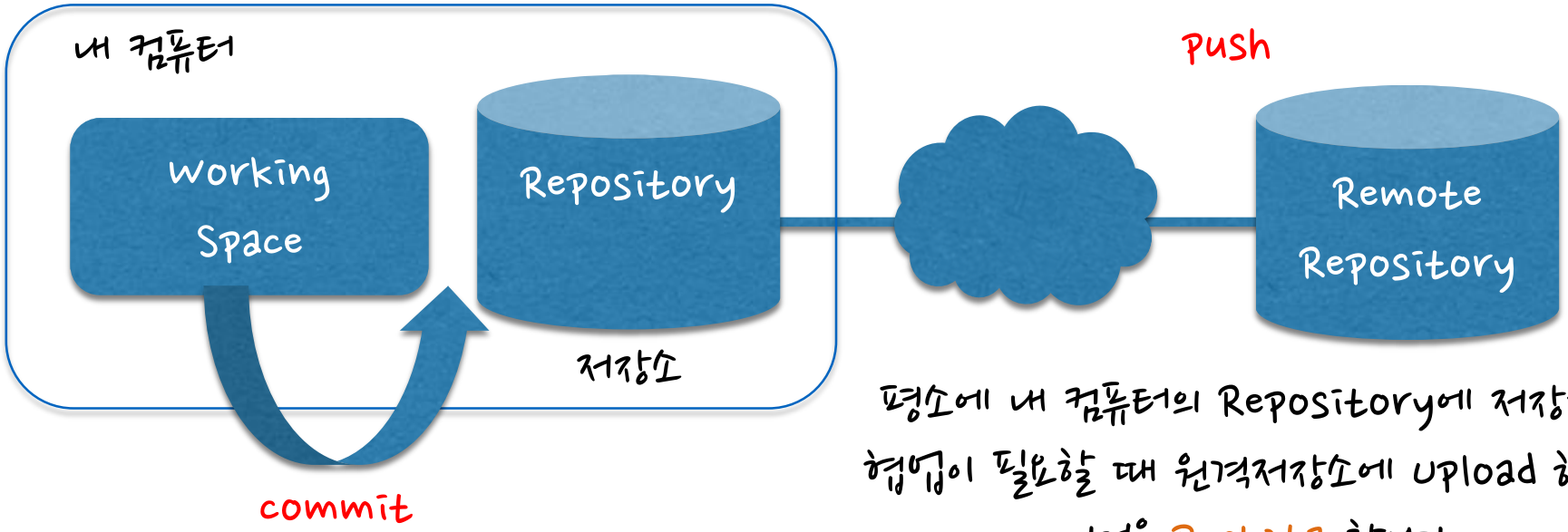
지금부터 Preforce를 기준으로 git을 설명드리겠습니다.



그런데 git에선 저장소가 내컴퓨터에 있어요.

그럼 혼자서만 일하나요? 팀이 같이 일해야하는 데...

안될리가 없죠. 저장소를 외부에도 만들면 됩니다.
이 것을 원격 저장소 Remote Repository라합니다.

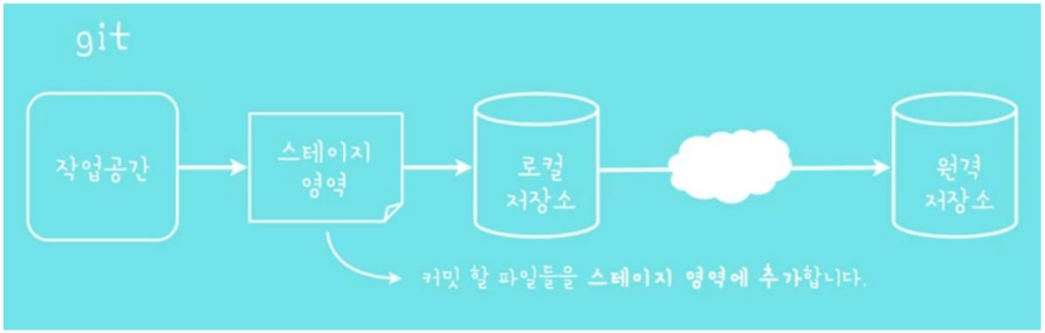


평소에 내 컴퓨터의 Repository에 저장해두고
협업이 필요할 때 원격저장소에 upload 하는 데
이것을 Push라고 합니다.

Staging

DB서버에 바로 올릴 경우, 생길 수 있는 여러 위험 때문에 일단 올려두고 DBA 승인 하에 deploy가 됩니다.
git에선 repository에 commit하기 전에 후보가 된다는 뜻에서 staging이란 개념이 생겼습니다.
이 곳을 스테이지 영역이라고 하고, 인덱스(index)라고도 부릅니다.

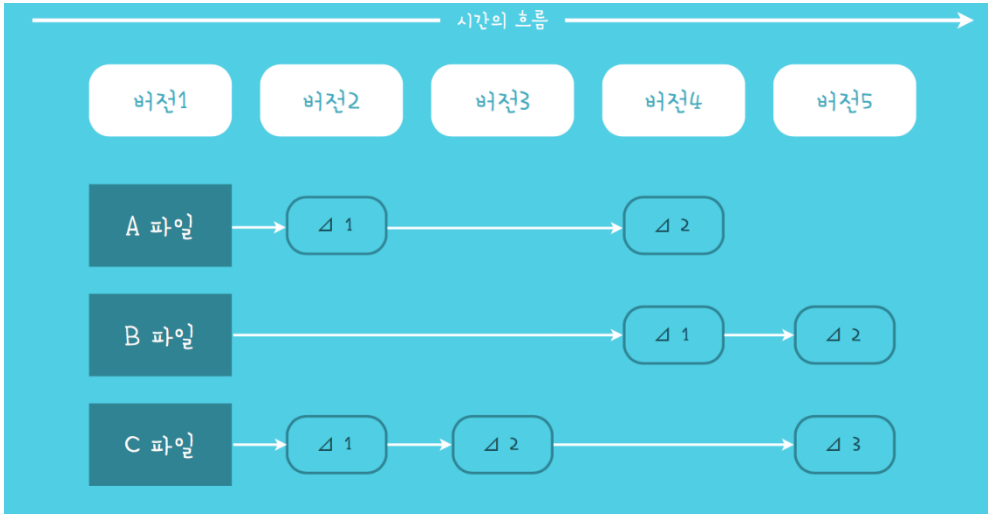
아까 개념으로 보면 개발자의 작업 공간과 로컬 저장소 사이에 이렇게 위치합니다.

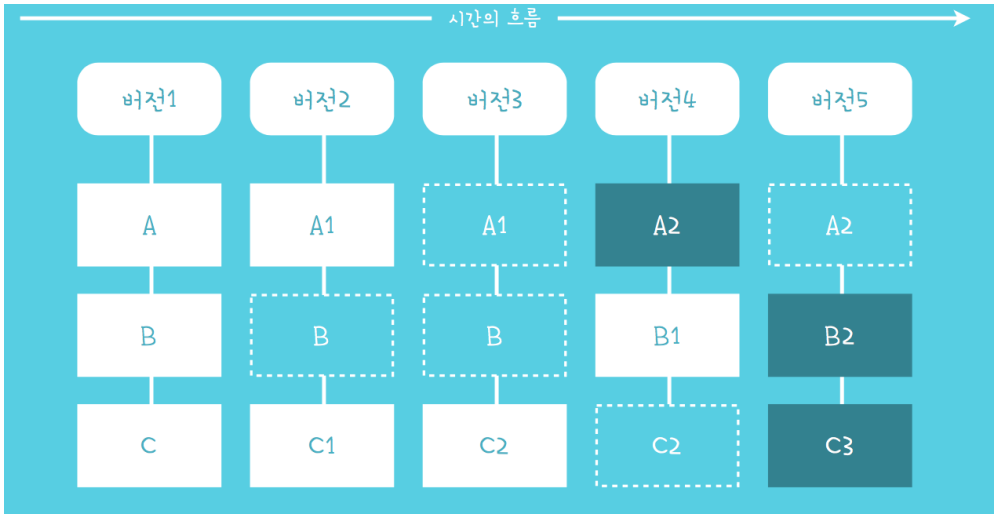


Tracked File - Repository **에 등록하기 위해서 수정/추가/삭제 등 작업을 관리 하고 있는 파일**.
Untracked File - **스냅샷에 존재하지 않으므로 등록되지 않 아 삭제되면 복구할 수 없다**.
(add**를 통하여** Tracked File**로 전환**)

Snapshot

Perforce는 Meta 파일과 변경 내역을 조합하여 최신 버전의 파일을 만들고 전달합니다.
반면 git으로 최신버전 파일들을 가져올 때는 오른쪽과 같이 최신의 스냅샷으로 만들 수 있습니다.

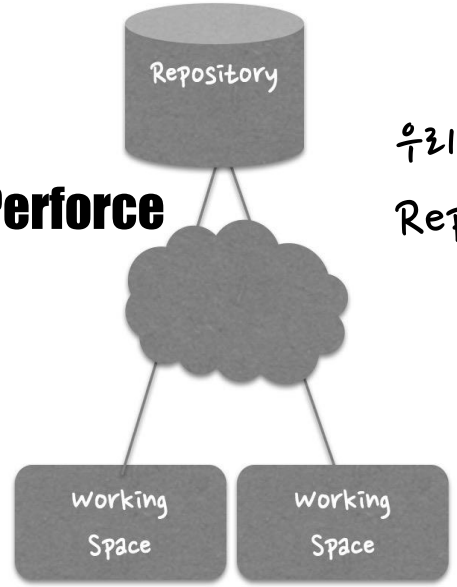




git은 복잡한 연산 과정 없이, 마지막에 해당하는 스냅샷들만으로 최신버전을
아주 빠르게 조합할 수 있습니다. 심지어 Local Repository에서 일어나기 때문에 네트웍으로 인한 Lack도
없습니다. (4X ~ 300X 까지 faster) 체감하기에도 엄청 빠릅니다.
이런 개념은 git의 철학인 **비선형적인 개발 (Branch)**을 위해 만들어 졌습니다.

https://upload.wikimedia.org/wikipedia/commons/1/1b/Linux_Distribution_Timeline.svg

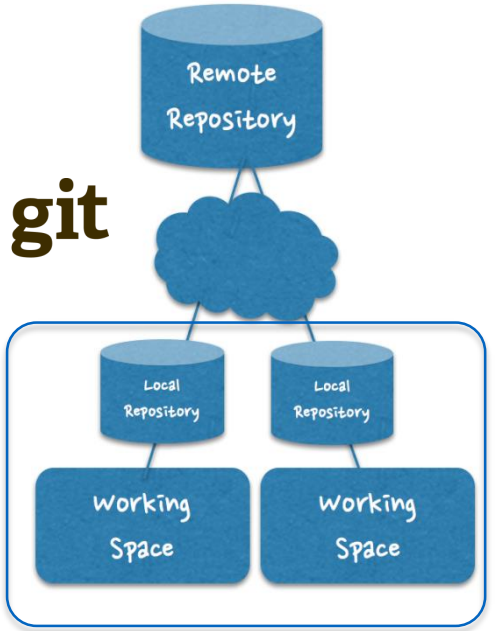
분산형 (Distributed) VCS (버전 콘트롤 시스템)



우리가 일했던 Perforce가 이렇게 commit을 할 때마다
Repository에 저장되는 구조였다면

git

Git은 local repository에 commit해두고
Remote Reposotory에 push하는 구조입니다.
이런 구조를 D-VCS라고합니다.



DVCS?

DVCS라고 붙여 읽으면 잘 안와닿는 데, 분산형 VCS라고 읽으면 훨씬 개념이 쉽게 와닿습니다. 분산형 ? VCS ? → 버전 콘트롤 하는 서버가 여러개겠구나. 맞습니다. 로컬 저장소, 원격 저장소 모두가 버전 콘트롤을 수행합니다.당연히 git은네트워크가 되지 않아도우선 내 컴퓨터에서 버전관리를 하고 네트워크 통신이 될 때 서버와 통신해도 됩니다.

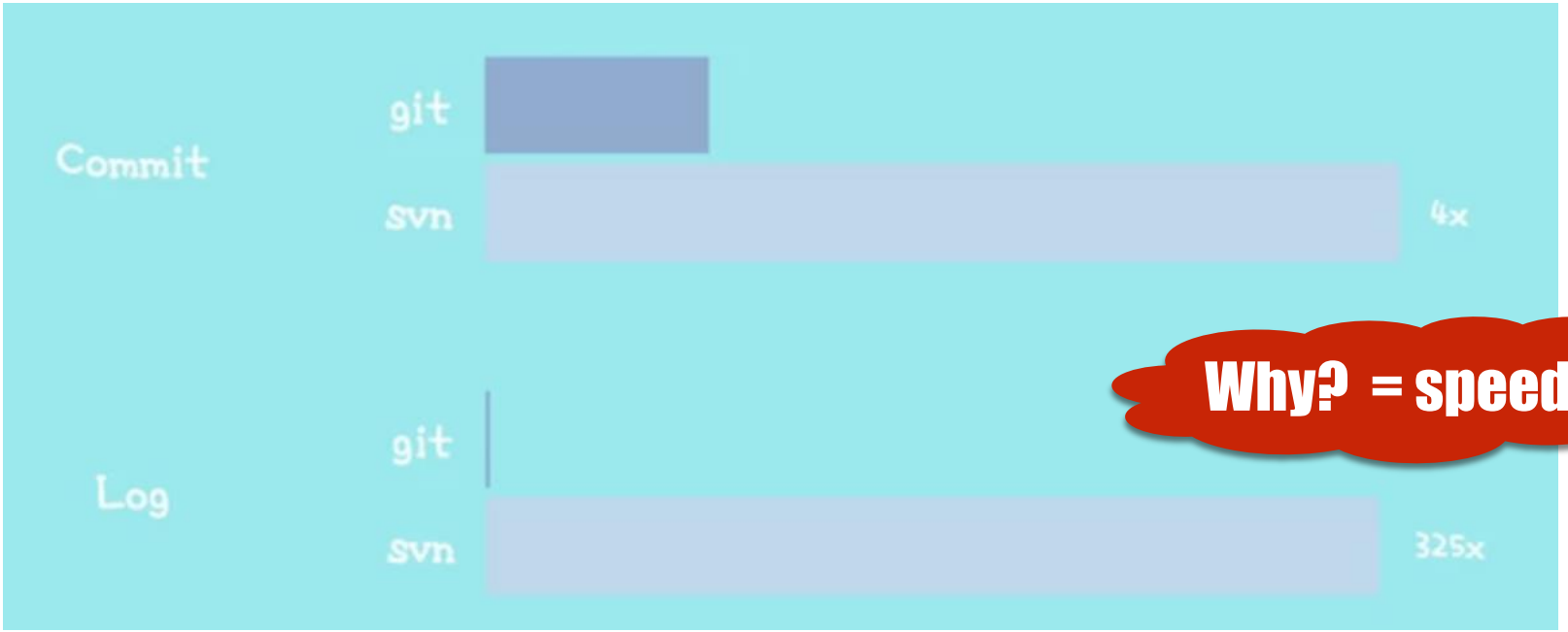
Git, Mecurial, Bazaar, Darcs 같은 DVCS에서의 클라이언트 는 단순히 파일의 마지막 스냅샷을 Checkout 하지 않습니다. 그냥 저장소를 전부 복제합니다.그 결과, 서버에 문제가 생기면 이 복제물로 다시 작업을 시작할 수 있습니다. 심지어, 클라이언트는 아무거나 골라도 서버 를 복원할 수 있습니다. 모든 Checkout은 모든 데이터를 가진 진정한 백업이라고 볼 수 있고 원격저장소 장애나 인터넷연결이되 지 않아도 버전 관리 가능합니다.

이젠 인터넷이 안되는 휴양지에 가서도 DVCS를 이용하면 충분히 개발할 수 있고, 나중에 인터넷 되는 곳에 가서 서버에 Push해주면 됩니다.

Why D-VCS?

이렇게 로컬 저장소가 따로 있어서 어떤 점이 좋은 지 3가지 특징을 알려드릴게요.

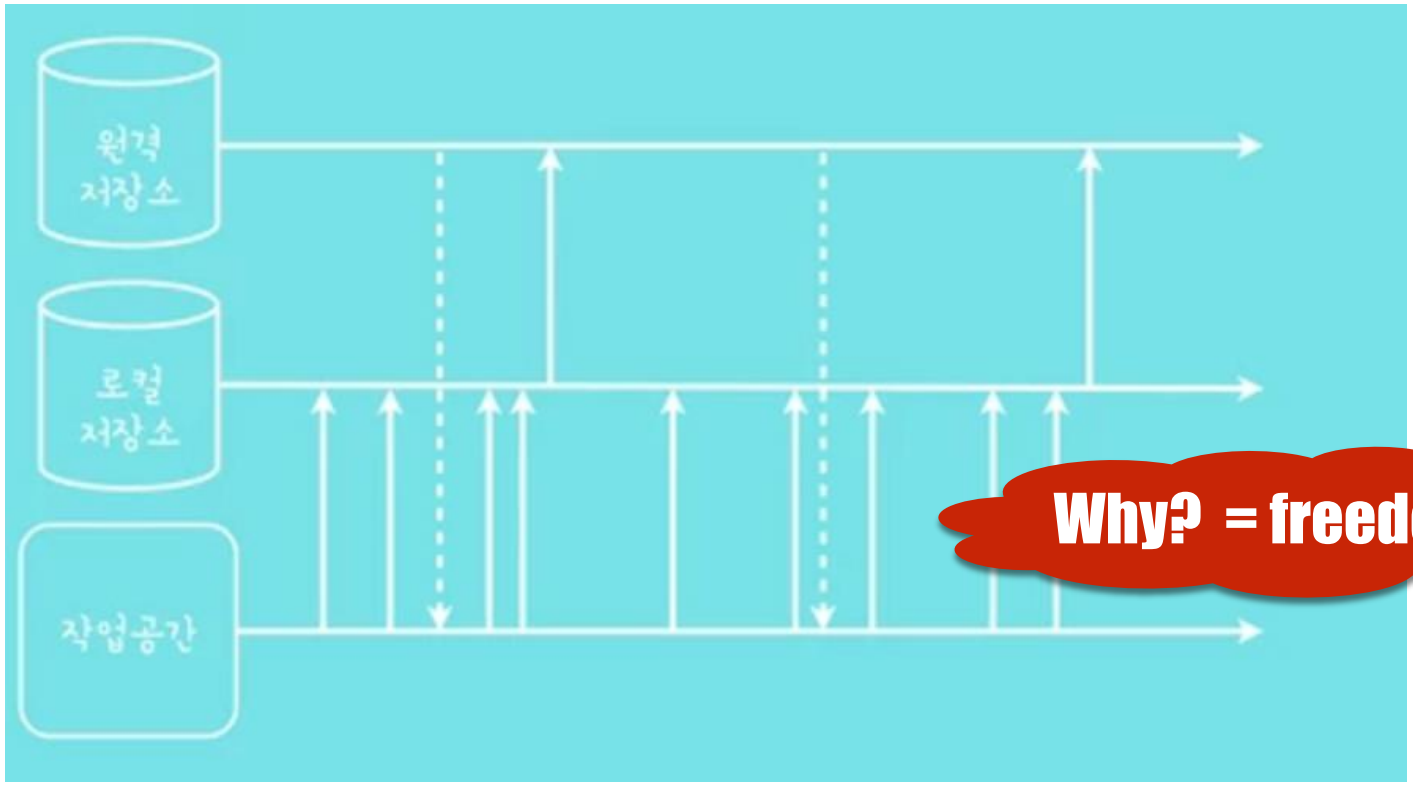
1. 일단 commit 할 때 엄청 빠릅니다.



아무리 기가빗 인터넷이 있고, 고용량 서버가 있어도.나만 바라보고 고속 SSD를 갖춘 로컬 컴퓨터를 따를 순 없겠죠?
git 공식 홈피에 따르면 commit은 4배, 로그처리는 325배 빠르다고 하네요.

Why D-VCS?

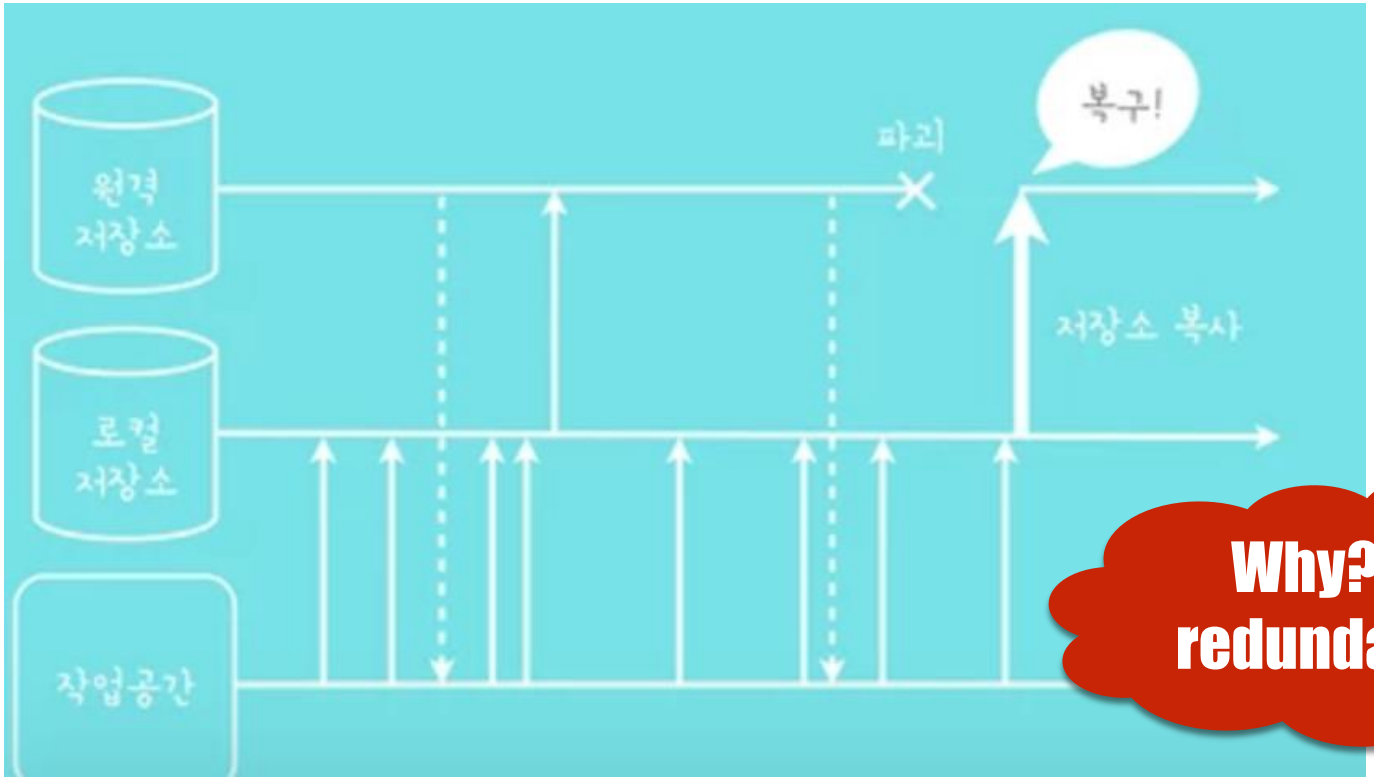
2. commit할 때 그 까이꺼 하고 부담없이 할수가 있습니다.흠흠... 뭐라 하든 내 컴터인데요 뭐...



언제든 나혼자 쓰는 소스서버, conflict **나는 것 걱정할 필요도 없고 최신 스냅샷은 언제나 내 컴터에**...

Why D-VCS?

3. 네트워크 문제가 있거나, 서버가 교체 중이라도 계속 버전 관리가 가능합니다.
(SVN때 처럼 서버 문제로 개발자 전원이 멘붕되는 사태는 없겠죠)



교체 중에 심지어, 서버가 교체 중에 날라가 버려도 복원이 가능합니다.

GIT (DVCS) 의 구조 - DAG

IMPORTANT

"Directed Acyclic Graph"

SHA-1 Hash

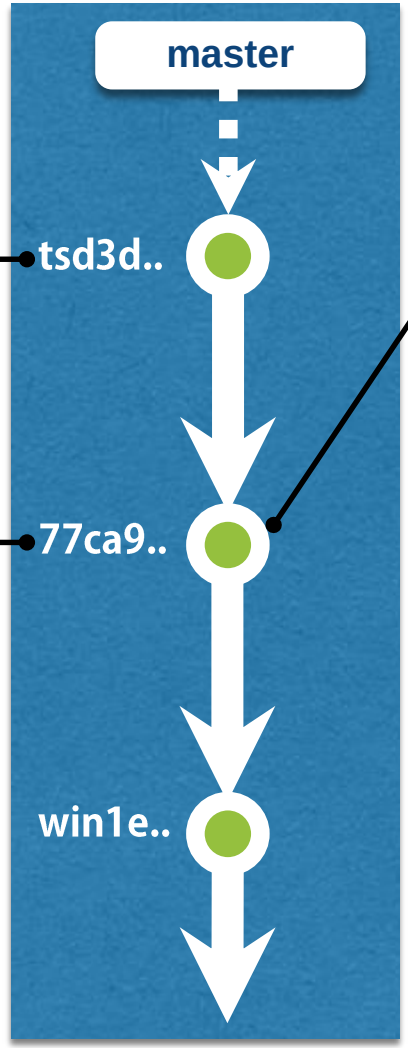
$ cat .git/refs/heads/master

tsd3dd41d0cf3d1b6aa2d8ad0549ab2fcb1575d1

Object

$ git cat-file -p 77ca9

Parent	tsd3d..
Tree	sndg1..
Committer	Js.park<jsp@…> 1424..
Auyhor	Js.park<jsp@…> 1424..
	this **is my commit message.**



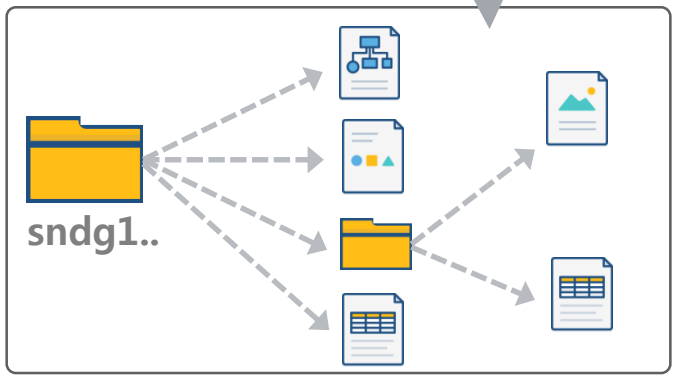
Object (Tree Detail)

$ git cat-file -p sndg1

sndg1..

100644	blob	ace23..	.gitignore
100644	blob	dbdbd..	.gitattributes
040000	tree	a0bc3..	Assets
040000	tree	33d33..	ProjectSettings
100755	blob	b1de7..	build.sh
100755	blob	7011e..	README.md

Filemode Type SHA-1

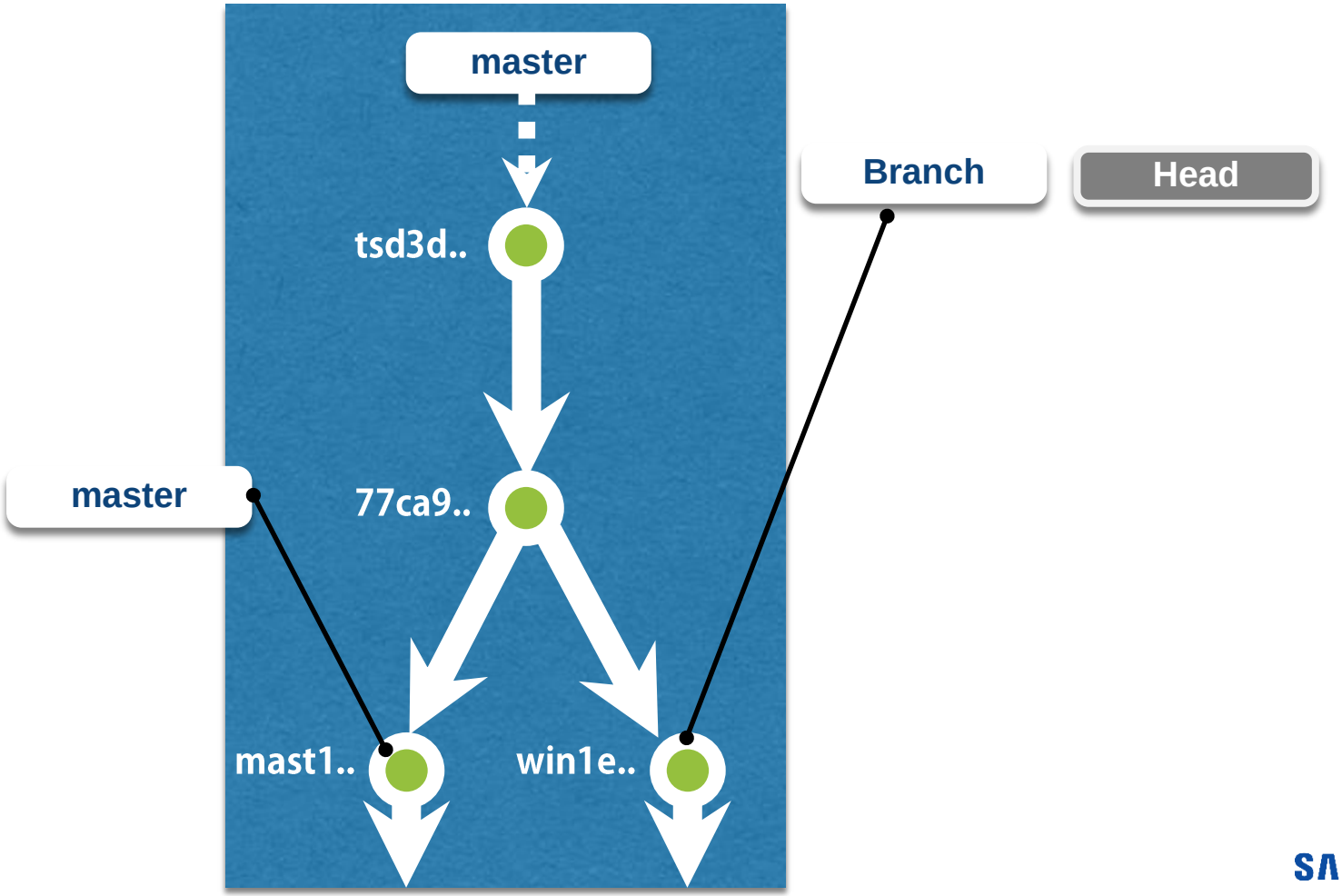


TwoSeed

SAMSUNG

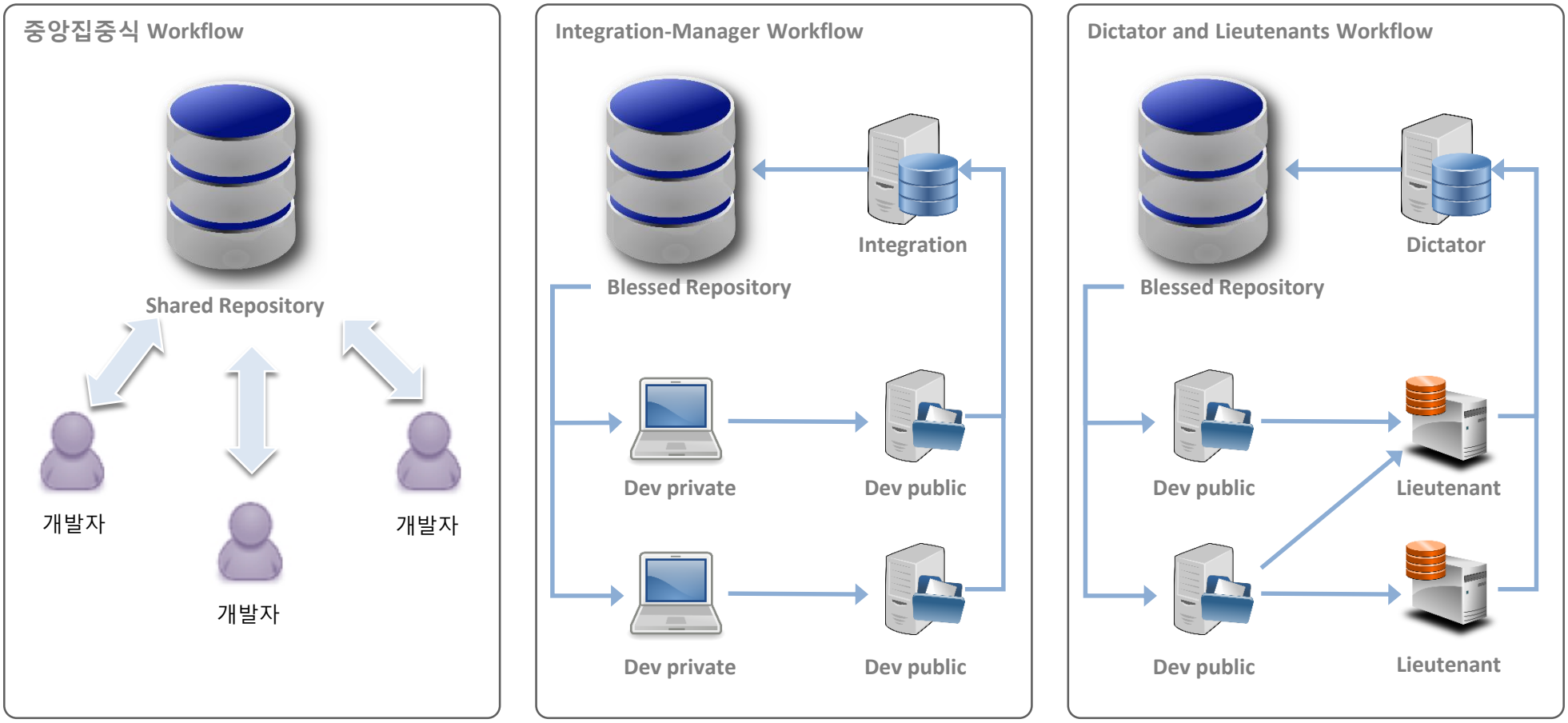
GIT (DVCS) 의 Branch 구조

모든 commit은 Snapshot으로 저장되며,
Branch와 Master의 분할 작업 역시 각 Snapshot에 대해 포인터로 연결됩니다.



GIT 의 workflow 전략

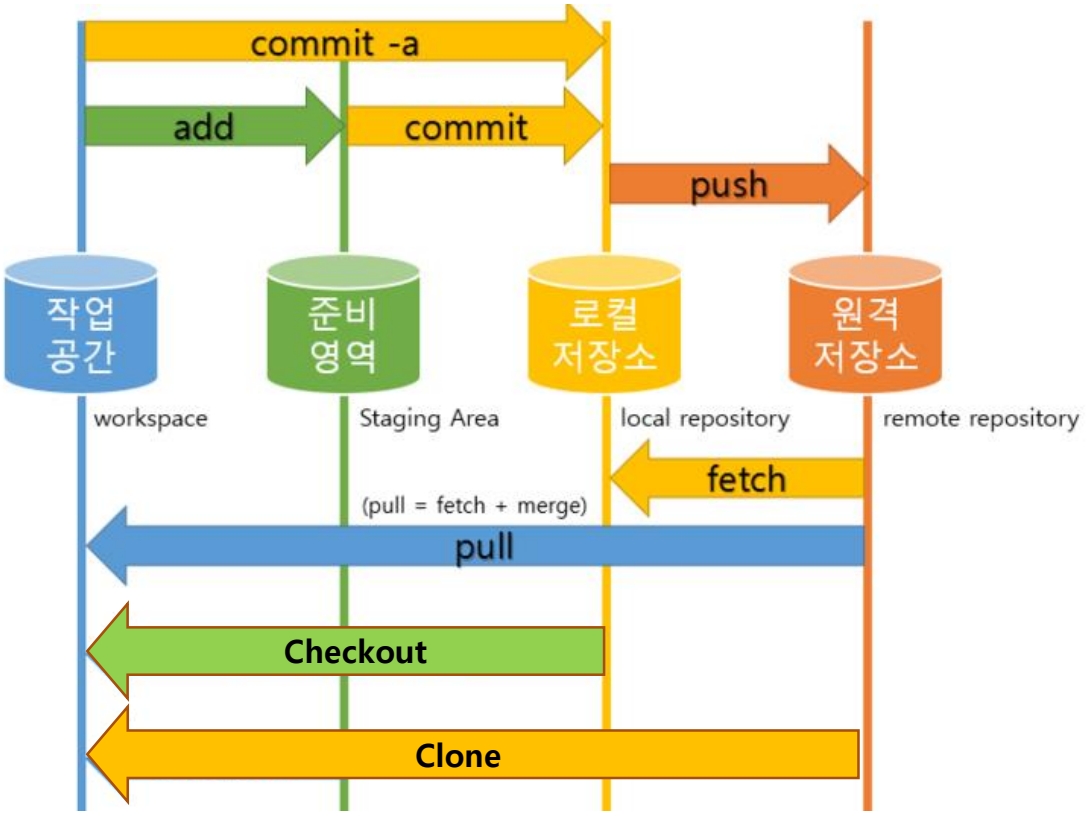
분산 환경 하에서의 대표적인 3가지 workflow 기준 모델을 제시합니다.



Git Commands & Wrap up

IMPORTANT

앞으로 다룰 명령어를 한장의 그림으로 정리해보았습니다.



- Git는 분산 소스 관리 시스템 이다.
- Git은 빈 디렉토리는 추적하지 않는다.
  (최소 한 번 커밋해야 추적시작)
- 관리하지 않을 파일은 .gitignore 파일에 추가한다.

- **HEAD는** 현재 브랜치의 가장 최신 커밋을 의미한다.
- 기본 원격 저장소를 **origin** 라고 부른다.
- 기본 브랜치를 **master** 라고 부른다.

1.3 Git 의 사용
실습

1.3-1. Git 기본 설정

- Git을 사용하기 위하여 설치 및 기본 사용 환경을 구성

Chapter Target

Step #1

Git 설치

- OS 환경에 알맞은 설치
- 설치 가능한 제품
- 설치 진행

Step #2

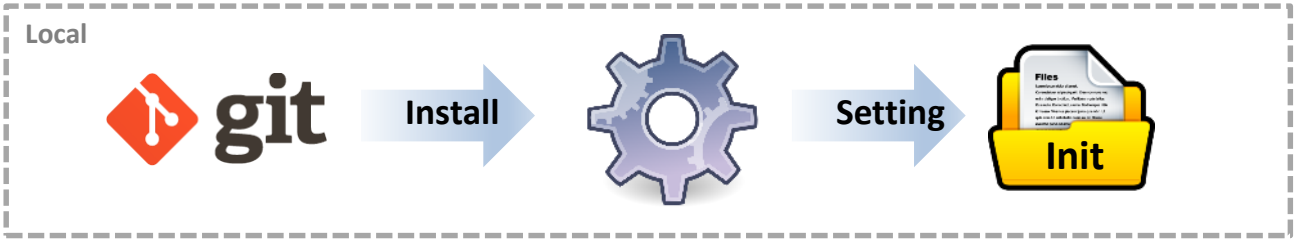
사용자 정보 및 환경 설정

- 사용자 기본 정보 등록
- 명령어를 통한
  부가 환경 설정 (Optional)

Step #3

폴더 생성 및 초기화

- 작업 공간 생성
- 초기화 명령어 사용
- 작업 준비 완료 및 참고사항



1.3-1. Git 기본 설정

1. 설치하기

✓ **Linux 설치: Linux에서 Git을 설치 시에는 각 OS의 배포판에서 사용하는 Package 관리 Tool을 사용한다.**

Fedora 계열 (RedHat, CentOS)

$ **sudo yum install git-all**

Debian 계열 (Unubtu)

$ **sudo apt-get install git-all**

✓ **Windows에서 설치하기: 필요한 Installer를 다운받아 설치 진행 (취향에 맞게 선택 활용)**

Windows용 Git 다운로드: http://git-scm.com/download/win

Totoise Git: http://code.google.com/p/tortoisegit/ (msysgit의 추가 설치 필요: http://msysgit.github.io/)

Atlassian SourceTree: https://ko.atlassian.com/software/sourcetree

GitHub: http://windows.github.com

GitSwarm: https://www.perforce.com/ko/git

GitLab: https://about.gitlab.co₩m/downloads/

1.3-1. Git 기본 설정

2. 초기설정

✓ **Linux 에서 설정 (Console 기반)**

I. **/etc/gitconfig** : 시스템의 모든 사용자와 모든 저장소에 적용되는 설정. git config --system 옵션으로 이 파일을 수정

II. **~/.gitconfig, ~/.config/git/config** : 특정 사용자에게만 적용되는 설정. Git config –global 옵션으로 이 파일을 수정.

III. **.git/config**: 이 파일은 Git 디렉토리에 있고 특정 저장소(혹은 현재 작업 중인 프로젝트)에만 적용.

※ 각 설정은 역순으로 우선시 된다. (`.git/config`의 설정이 `/etc/gitconfig`보다 우선 적용.)

a. Git 설정내역은 사용자 홈 폴더의 .gitconfig 파일에 저장되어 있으며 직접 편집 또는 Config 명령어를 활용하여 사용자 계정 정보를 등록 한다. (필수)

$ git config –global user.name " <사용자명> "
$ git config –global user.email " <메일 주소>“

b. 출력 메시지의 색상 변경을 원하는 경우 (선택)

$ git config –global color.ui auto

c. 기본 Editor 변경 (선택)

$ git config --global core.editor vim

d. 명령어를 단축 키로 사용하기 위해 설정 (ex, Commit을 “cm“으로 대체) (선택)

$ git config –global alias.cm commit

e. 설정 확인

$ git config --list

1.3-1. Git 기본 설정

2. 초기설정

✓ **Windows / MAC 에서 설정 (GUI 기반)**

설치한 제품의 설정 항목에서 사용자 계정 정보 및 기타 설정을 진행하며,
기본 설정은 Linux의 내용과 동일

1.3-1. Git 기본 설정

3. 새 저장소(로컬 작업 공간) 설정

1) 작업 Directory 생성

2) 생성 된 작업 Dir로 이동하여 Git 초기화 명령을 통해 저장 공간 지정

   - $ mkdir workdir
   - $ cd workdir
   - **$ git init**
   - Initialized empty Git repository in /Users/yourname/Desktop/ workdir/.git/

3) 모든 작업을 완료 시 Git을 통해 사용 할 기본 준비가 완료 되며 생성한 지정 Dir에서 파일을 추가/변경 등의 작업을 통해 Version 관리를 할 수 있는 기본 환경이 구성

4) 참고사항
   - Git은 DVCS 이므로 기존 다른 도구와 달리 개인 별 설정이 우선 함에 유의
   - **개인이 저장소를 생성하거나 원격의 저장소에서 파일을 받기 위해서는 반드시 폴더 지정 및 초기화가 필수**

4. Git 파일 구조

1) HEAD
   - **현재 Branch의 가장 마지막 Commit 파일**

2) Index (Stage)
   - Index는 워킹 디렉토리에서 마지막으로 Checkout 한 브랜치의 파일 목록과 파일 내용을 기록하며, Commit 시 반영 할 정보를 관리

3) Working Directory
   - 실제 작업을 진행하는 Dir

1.3-1. Git 기본 설정

* 시작 시 참고 할 내용

1) Git 은 빈 디렉터리는 추적하지 않음
2) 형상관리를 하지 않을 파일은 .gitignore 파일에 추가 (Android studio 및 Eclipse 등의 개발도구 사용 시 주로 사용)
3) **HEAD는 현재 브랜치의 가장 최신커밋을 의미 (다른 버전관리 도구의 Tip revision과 유사)**
4) **Origin은 기본 원격 저장소를 의미**
5) **모든 파일 추가 및 변경 작업 후에는 반드시 add 명령을 사용하여 Staged 상태로 전환 필요**
6) Git은 변경 이력을 Reverse delta 방식의 Change set으로 관리하는 기존 도구와 달리 Snapshot 방식으로 변경을 저장 (Staging Area에 있는 데이터의 스냅샷에 대한 포인터, 저자나 커밋 메시지 같은 메타데이터, 이전 커밋에 대한 포인터 등을포함하는 커밋 개체(커밋 Object)를 저장)

1.3-2. Git 기본 명령어 사용

Git을 사용하는 가장 기본적인 명령어에 대한 이해 및 활용 (add, Commit, push)

Chapter Target

Step #1

관리 File의 추가 (ADD)

- Source 및 File을 Version 관리 하기 위해 작업 공간에 추가
- 추가 된 파일을 식별하여 등록
- Status 명령을 활용하여 상태 확인

Step #2

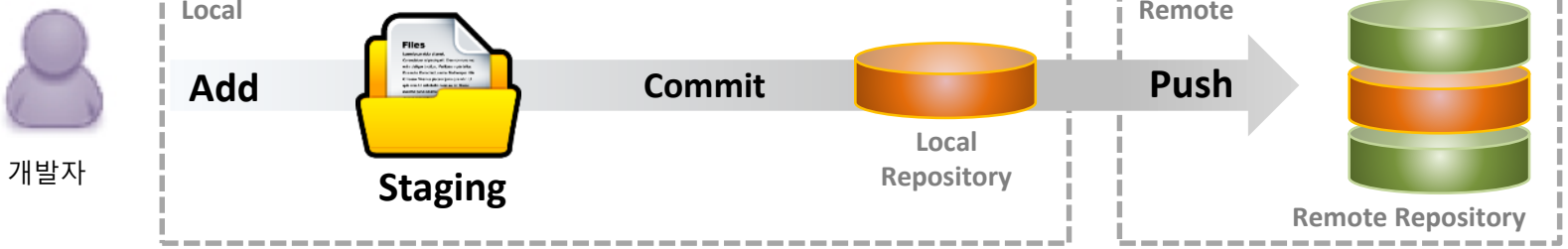
추가 된 File을 로컬 저장소에 등록하여 이력관리 (Commit)

- 등록 된 파일의 저장소 반영
- 로컬 저장소에 반영 된 파일의 버전 관리

Step #3

로컬 저장소에 등록 된 File을 원격 저장소에 공유

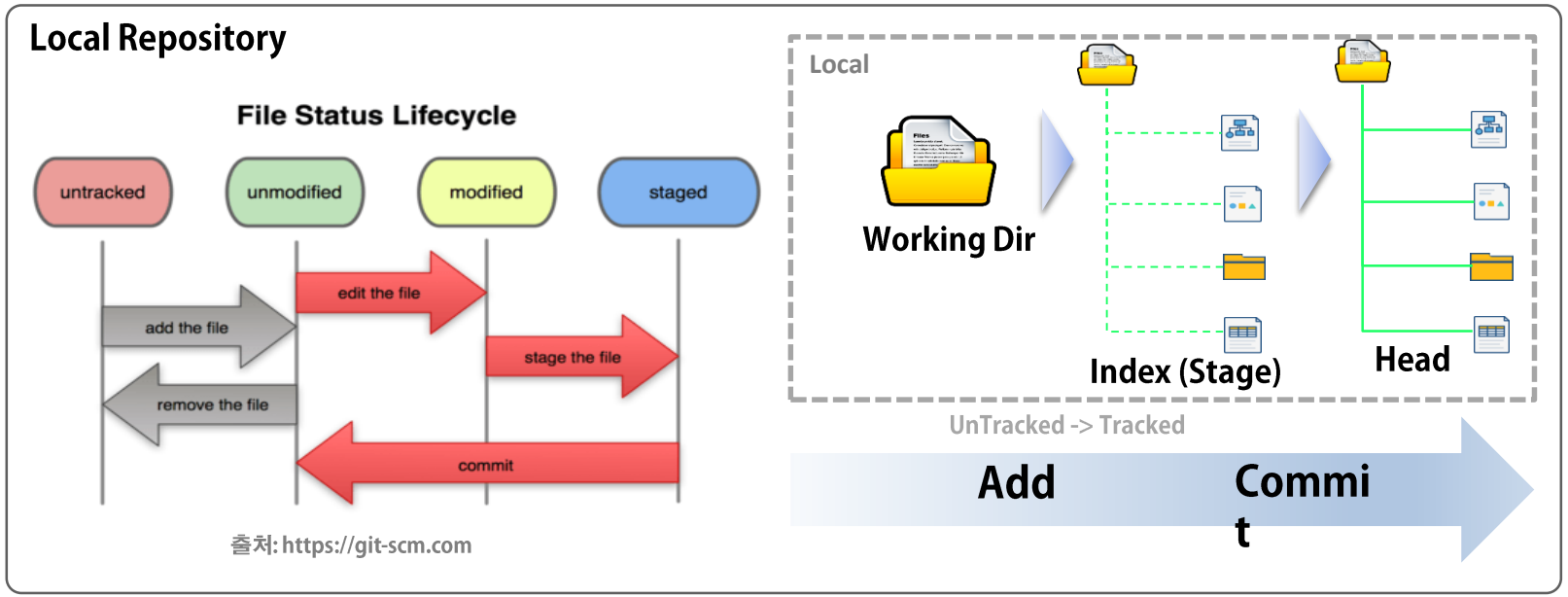
- 원격의 저장소에 반영
- 로컬에서 반영 된 내용을 원격의 저장소에 공유/반영



1.3-2. Git 기본 명령어 사용

1. 형상 식별 (관리 할 File 식별하기)

1) Add **명령어**: File **또는** Folder**를** Staging **영역에 등록하여** Commit**을 위한 사전 식별을 진행**
2) **버전관리 등록** Process



3) **명령어 활용**: add **명령을 통해 파일/폴더를** Stage (Index) **상태로 등록**

명령어 활용:

- **$ git add <파일 이름>**
  추가 하고자 하는 파일 (Untracked)을 Stage (Index)에 있는 Tracked상태로 전환, (파일 명에 *사용 가능)

1.3-2. Git 기본 명령어 사용

2. 형상 등록 (로컬 저장소에 파일/폴더 등록하기)

◆ **Commit 명령어: Stage (Index)에 있는 Tracked 상태의 파일/폴더를 로컬 저장소에 등록, 버전 관리**

명령어 활용:

- **$ git commit -m “Comments"**
  Stage (Index)에 있는 Tracked 상태의 파일/폴더를 로컬 저장소에 Commit 하며.
  -m 은 Commit 메시지를 등록하는 옵션으로, 여러 줄의 메시지를 쓸 경우 -m 을 여러 개 사용 가능

- **$ git commit -a –m ”Comments”**
  Tracked 파일 중 수정 중인 파일의 Stage (Index) 등록과 동시에 상태의 파일/폴더를 로컬 저장소에 Commit .
  단, Untracked 상태의 File을 자동으로 Tracked 상태로 변환 해 주지는 않으므로 사용에 유의

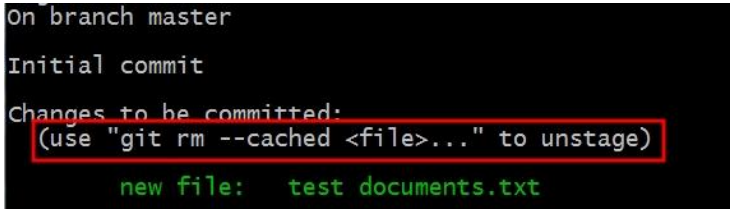
- **$ git commit -v -m “Comments"**
  -m을 사용하지 않을 때 -v옵션을 사용하면 편집기에 Commit 하려는 변경사항의 다른 점을 보여주며,
  특정파일만 Commit 하려면 마지막에 파일명을 추가하여 Commit 진행

1.3-2. Git 기본 명령어 사용

3. 등록작업 취소하기

1) Add 작업 취소하기 : Stage 상태의 파일 등록을 해제하기 (Unstaging)

작업 순서:
1. 현재 상황에 따라, Unstaging 명령어가 다름 : git status를 통해 현재 상태를 확인
2. git status 출력 메시지 중 “Changes to be committed: ..“ 아래 표시되는 명령을 통해 Unstaging (add 작업 취소)



2) Commit 작업 수정하기 (amend): Commit 메시지 변경이나 수정이 미처 덜 반영 되어 다시 반복 등록하고자 할 때

명령어 활용(amend):
- **$ git commit --amend** :전체 Commit 된 내용을 다시 Commit 반영, 이 때 변경 내용이 없다면 기존 내용으로 그대로 유지
- 만약 Commit에서 누락 된 파일이 있을 경우 아래와 같이 명령을 수행하여 해당 파일을 추가하고 재 Commit

  $ git commit -m 'initial commit'
  $ git add <forgotten file>
  $ git commit --amend

3) 수정 중인 파일(Modified)을 최종 Commit 시점의 원래 파일로 되돌리기

명령어 활용(Check out):
- **$ git checkout <Hash 또는 Branch 등>**
  기존 Commit 된 파일 중 현재 추가 수정 중인 파일을 Commit 시점으로 되돌리는 명령

1.3-2. Git 기본 명령어 사용

4. Commit 이력 조회하기 (Log조회 및 Diff)

◆ **Log 명령어: Commit 이력을 조회하고, Commit 된 이력 간의 비교를 할 수 있음**

명령어 활용(Git log): 기본 Commit log 확인

- **$ git log**
  저장소의 Commit 히스토리를 시간순으로 보여주며, 최신의 내용을 우선으로 표시
- **$ git log -p -2**
  최신 Commit 2개에 대해서 비교하여 표시하는 명령으로 직접 diff를 실행한 것과 같은 결과를 출력
- ※ Log의 다양한 Option에 대해서는 “1.3-8. Git 추가 기능 활용”에서 설명

1.3-2. Git 기본 명령어 사용

5. 로컬에서 등록한 변경 내용을 원격 저장소에 저장하기

1) Push 명령어: 로컬 저장소에 관리되는 버전을 원격 저장소에 전달, 저장하기

작업 순서:

1. Git remote 명령을 통해 원격 Repository 지정
2. 해당 원격 Repository에 대해 Push 명령을 사용하여 로컬 저장소의 내용 보내기

명령어 활용(Git remote):

- **$ git remote -v**
  원격 저장소를 확인
- **$ git remote add [저장소 명] [저장소 URL]**
  원격 저장소를 추가, 삭제는 rm 인자 사용
- ※ push 나 pull 실행 단계에서 원격 저장소명을 생략하면, 자동으로 origin 이라는 이름으로 생성

명령어 활용(Git push):

- **$ git push [저장소 명] master**
  [저장소 명] (origin) 원격 저장소에 master 브랜치에 추가된 버전(스냅샷)들을 Upload.

Remote repository 만들기:

- **$ git init --bare [저장소 명].git**
  로컬 저장소가 아닌 원격 저장소를 생성하는 명령으로 Work directory가 포함되지 않는 순수 Repository 생성.

1.3-2. Git 기본 명령어 사용

5. 실습하기

- **Scenario**

1. File을 Workdir에 생성
2. 생성 된 파일을 git의 stage영역으로 등록 (add)
3. Stage 영역의 파일을 로컬 저장소에 등록 (Commit)
4. 저장 된 Commit에 대한 이력 조회 (Log)
5. 원격 저장소 조회 및 추가 (Remote)
6. 원격 저장소에 Commit 이력 Push하기 (Push)

1.3-3. Git 저장소 공유

원격 저장소의 Source를 공유하기 (Clone, Pull, fetch)

Chapter Target

Step #1

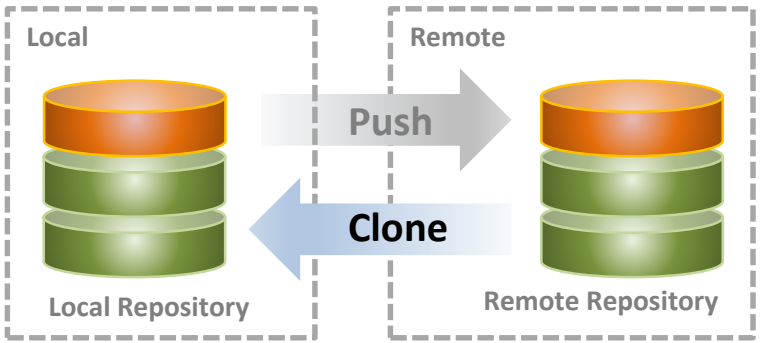
원격 저장소의 프로젝트를 로컬 저장소로 복제 하기 (Clone)

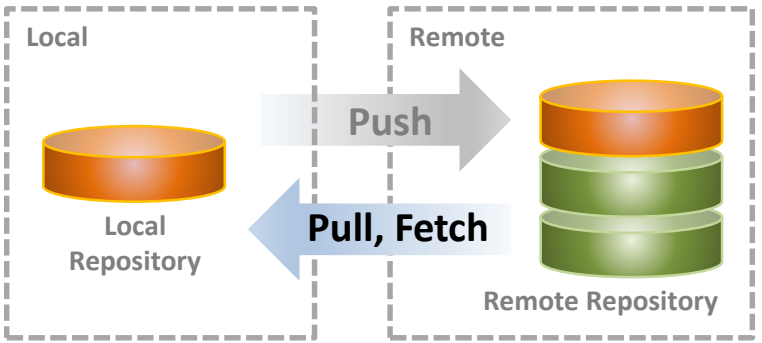
- 원격 저장소의 프로젝트를 로컬에 복제 (내려받기)
- git remote add + git pull 과 거의 유사
- 원본 (origin) 확인 / 추적 / 동기화

Step #2

원격 저장소의 변경 내용을 로컬로 내려 받기 (Pull, fetch)

- 원격 저장소의 내용 중 작업 할 최신 항목을 내려 받기
- 원격 저장소에 있는 commit을 fetch하고, 로컬과 merge
- 원격서버 (remote) 추가 / 확인 / 추적 / 동기화





1.3-3. Git 저장소 공유

1. 원격 저장소의 파일을 로컬 저장소로 내려 받기

1) Clone 명령어: 원격 저장소의 프로젝트를 복제하여 로컬에 저장소 생성

명령어 활용(Git Clone):

- **$ git clone [저장소 URL] [저장될 폴더]**
  원격 저장소에 있는 프로젝트를 로컬로 내려받기 (가장 마지막의 폴더는 지정하지 않으면 현재 위치를 기준)
- **$ git clone –depth [숫자] [저장소 URL]**
  공용 프로젝트의 경우 버전 히스토리가 많으므로 최신 Commit 일부만 내려받고자 할 때 사용

※ Clone을 통해 원격의 내용을 갖고 오는 경우 자동으로 원격의 원본은 Origin 이라는 Remote 식별자를 갖는다.

2) Fetch 명령어: 로컬에 존재하지 않는 원격 저장소의 변경 내용을 확인하여 내려받는 명령으로, 로컬과 병합은 하지 않음

명령어 활용(Git Fetch):

- **$ git fetch [저장소 URL]**
  원격 저장소에 있는 프로젝트의 최신 변경 내용을 로컬로 내려받기 (병합은 하지 않음)

3) Pull 명령어: 로컬에 존재하지 않는 원격 저장소의 변경 내용을 확인하여 내려받은 뒤 로컬과 병합

명령어 활용(Git Pull):

- **$ git pull [저장소 URL]**
  원격 저장소에 있는 프로젝트의 최신 변경 내용을 로컬로 내려받은 뒤 로컬의 브랜치와 자동 병합

1.3-3. Git 저장소 공유

5. 실습하기

- **Scenario**

1. 생성된 작업 dir에서 원격 저장소의 프로젝트를 git clone으로 내려 받기 (Clone)
2. 내려 받은 파일 중 필요한 파일을 수정하기
3. git add 명령을 통해 수정한 파일을 Stage 등록 (Add)
4. git commit 명령을 통해 stage 상태의 파일을 로컬 저장소에 등록 (Commit)
5. 로컬 저장소에 저장 된 변경 이력을 원격 저장소로 보내기 전에 git fetch를 통해 원격의 최신 변경 확인 (Fetch)
6. git push 명령을 통해 로컬에서 작업한 이력을 원격으로 보내기 (Push)

1.3-4. Git Branch

Branch의 생성, 전환, 통합

Chapter Target

Step #1

Branch의 생성
(Branch, Checkout)

- 작업을 위해 Branch 생성하기
- 생성 된 Branch에서 작업 진행

Step #2

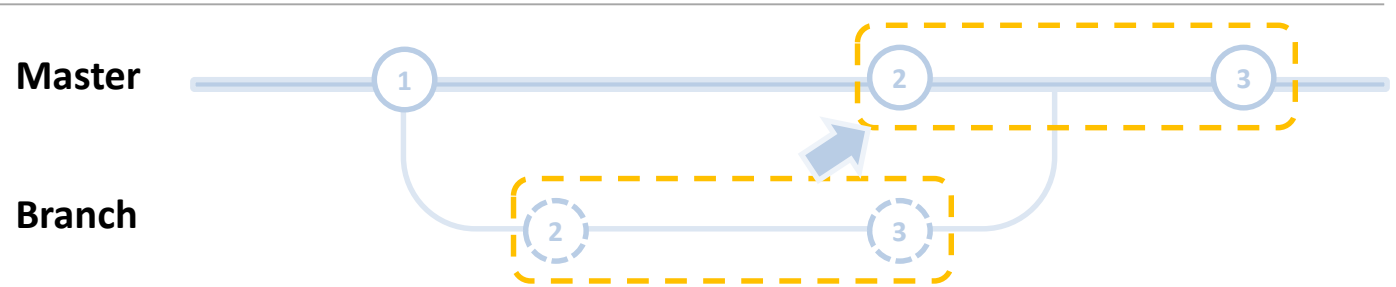
수정하고 반영하기
(Commit)

- 진행 중인 작업의 수정을 반영

Step #3

Branch를 Master에 통합하기
(Merge)

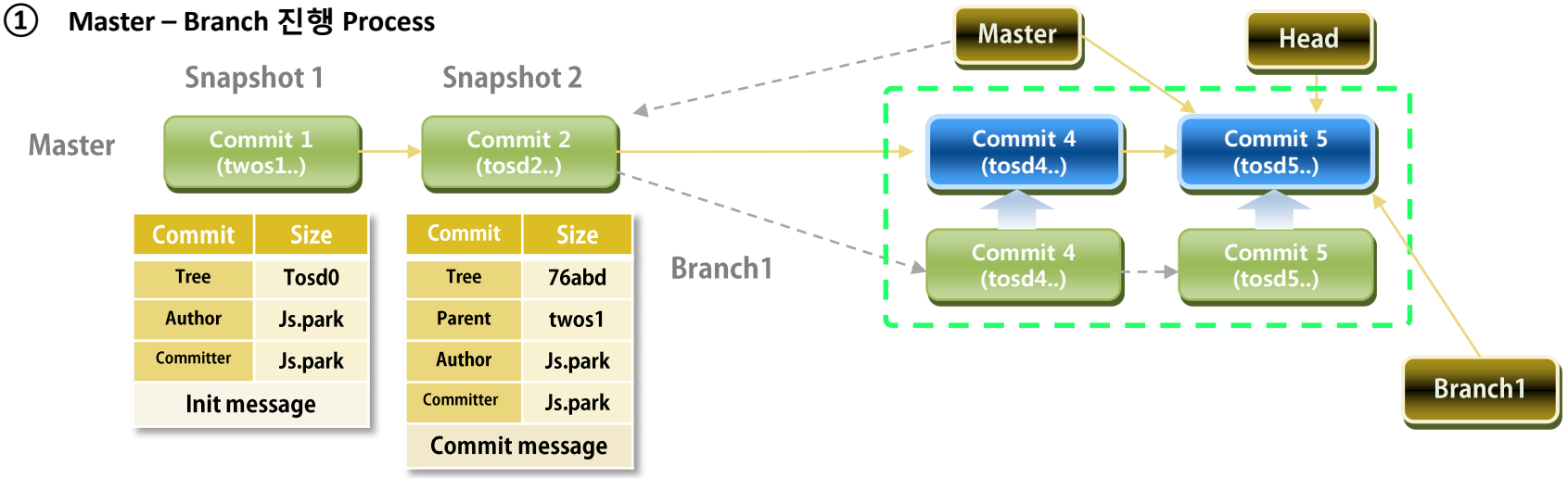
- Fast forward 방식으로 Master 에 최신 버전으로 Merge
- 작업이 완료 된 Branch 삭제



1.3-4. Git Branch

1. Branch 생성

① **Master – Branch 진행 Process**



② **Branch 명령어: Master에서 작업 Branch를 생성하여 Main과 분리 된 별도의 개별 작업 Branch를 생성**

명령어 활용(Git Branch):

- **$ git branch [Branch 명]**
  Master에서 별개의 작업 Branch 생성. 단, Branch를 생성해도 작업 Head는 여전히 현재 Branch(Master)에 존재.
- **$ git checkout [Branch 명]**
  Branch를 작업하기 위해 Head를 Branch로 이동하는 명령, 이 작업을 수행해야 Branch 작업의 진행이 가능
  -b option 을 추가 시 Branch의 생성과 Checkout을 한 번에 진행 할 수 있음

※ Branch를 생성 후 Check out을 하지 않고 파일의 변경 시 현재 Master Branch에 변경이 반영 됨에 유의

1.3-4. Git Branch

1. Branch 병합

① **Merge 명령어: Branch와 Master의 병합**

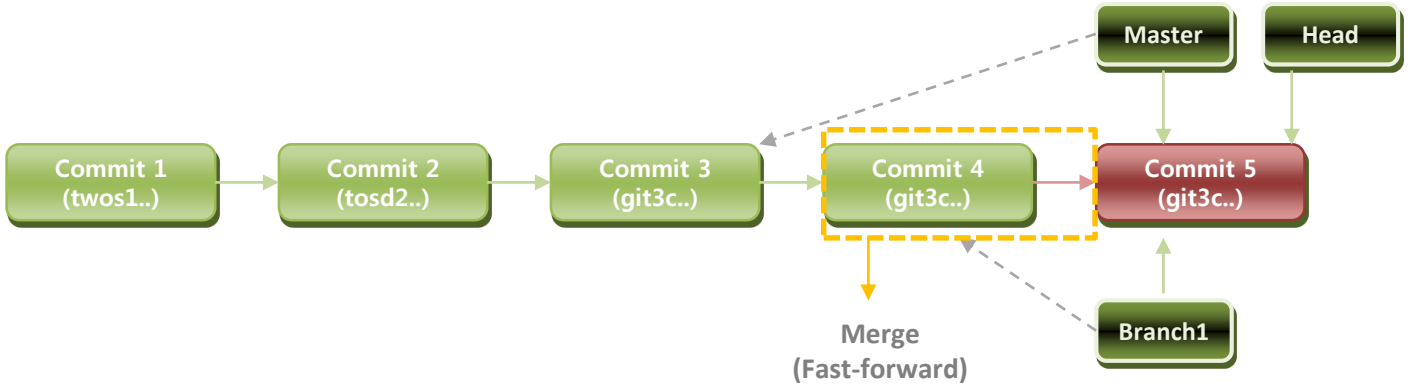
명령어 활용(Git Branch):

- **$ git merge [Branch 명]**
  현재 Branch에서 병합 할 Branch 명에 대해 병합 시 사용

- Branch를 병합하는 과정 중 가장 기본적인 Fast-Forward 방식의 병합은 아래와 같이 진행
  $ git checkout [Branch 명1] -> Merge 작업을 진행 할 Branch
  $ git merge [Branch 명2] -> 현재 Branch에서 Merge 하고자 하는 대상 Branch

※ Branch를 로컬 저장소에서 생성하고 진행하는 예제로서 나머지 응용은 다음 장에서 상세 설명

Process



1.3-4. Git Branch

5. 실습하기

- **Scenario**

1. Branch1 을 생성 후 Head를 새로운 Branch로 이동 ($ git checkout -b branch1)
2. 새로운 Branch1에서 파일 변경 후 Commit ($ git commit -a -m 'added file to branch1’)
3. Branch1을 Master branch에 Fast-forward 방식으로 Merge
    1. $ git checkout master
    2. $ git merge branch1

1.3-5. Git 변경 이력 통합 및 충돌 해결

3Way Merge 활용 시 Conflict, Merge, Rebase 기능 활용

Chapter Target

Step #1

Branch의 생성 (Branch, Checkout)

- 작업을 위해 Branch 생성하기
- 생성 된 Branch에서 작업 진행

Step #2

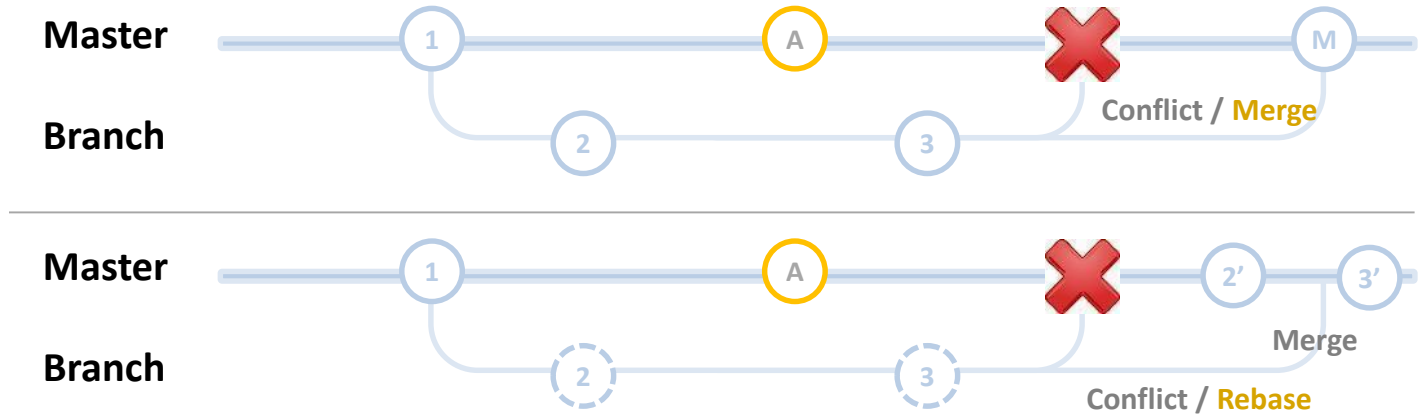
Branch를 Master에 통합 시 Conflict 발생 확인 (Merge)

- 진행 중인 작업의 수정을 반영하고자 할 때 충돌 상황 확인 (Reject, Conflict)
- 충돌 발생 시 확인

Step #3

Branch를 Master에 통합하기 (Rebase, Merge)

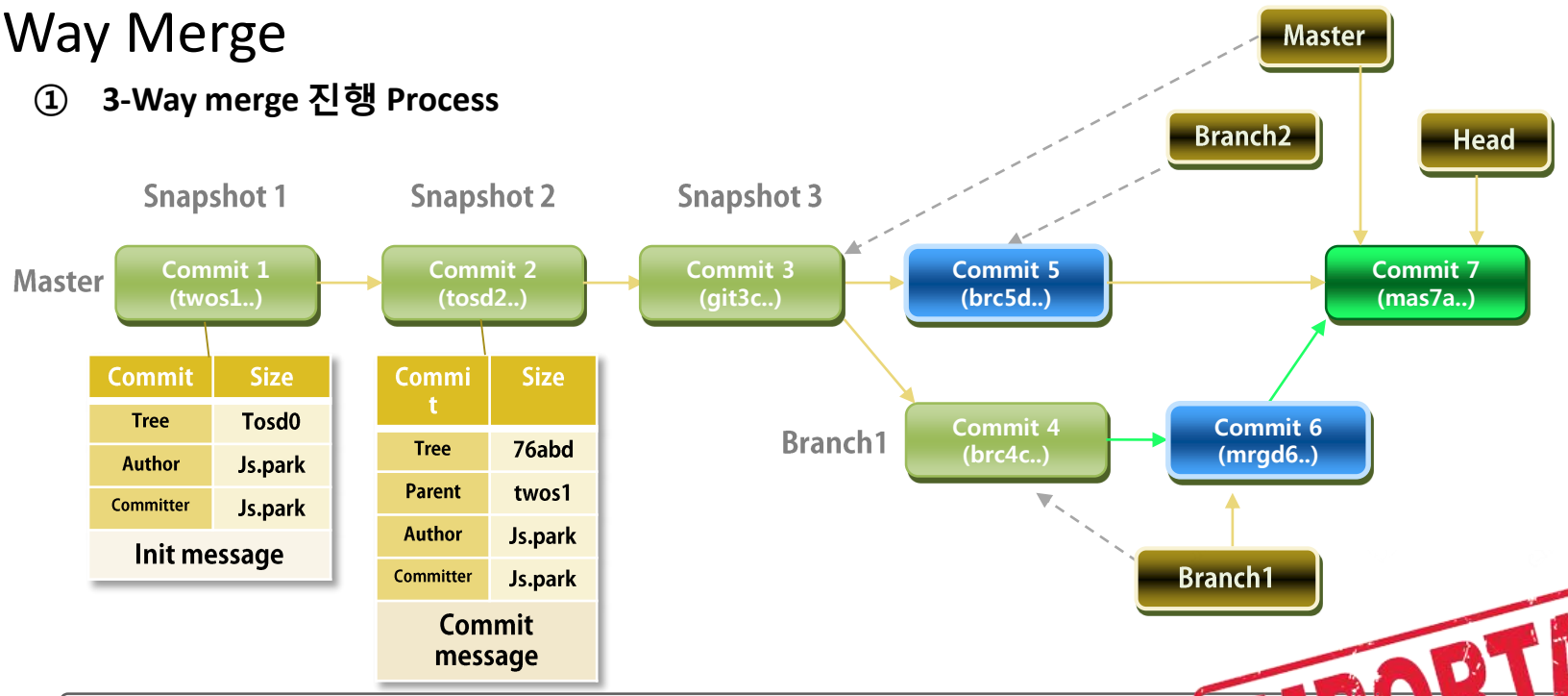
- 전체 Branch 이력을 Master 에 반영하기 (Merge)
- 최종 Branch 이력을 Master 에 반영하기 (Rebase)



1.3-5. Git 변경 이력 통합 및 충돌 해결

3-Way Merge

① **3-Way merge 진행 Process**



IMPORTANT

Scenario

1) Master branch 에서 Branch1을 생성하여 수정 진행
2) 다시 Master branch 에서 Branch2를 생성하고 변경 반영
3) Branch2를 Master Branch와 Merge 후 Branch2 삭제 (Fast-forward)
4) Branch1을 Master Branch와 Merge 시 3-way merge 진행 및 Conflict 해결 (Master branch + Branch1 + Branch2)

1.3-5. Git 변경 이력 통합 및 충돌 해결

상세 진행 순서:

1. Branch1 을 생성 후 Head를 새로운 Branch로 이동 **($ git checkout -b branch1)**
2. 새로운 Branch1에서 파일 변경 후 Commit **($ git commit -a -m 'added file to branch1)**
3. 새로운 Branch를 만들기 위해 다시 Master branch로 Head 이동하여 작업 환경 전환 **($ git checkout master)**
4. Master branch에서 또 다른 Branch2를 생성 후 Head를 Branch2로 이동 **($ git checkout -b branch2)**
5. 새로운 Branch2에서 파일 변경 후 Commit **($ git commit -a -m 'added file to branch2')**
6. Branch2를 Master branch에 Fast-forward 방식으로 Merge
    1. **$ git checkout master**
    2. **$ git merge branch2**
7. Master에 병합한 Branch2는 삭제 **($ git branch –d branch2)**
8. 다시 Branch1으로 Head를 이동하여 작업 환경 전환 **($ git checkout branch1)**
9. Branch1에서 추가 파일 변경 후 Commit **($ git commit -a -m 'More added file to branch1')**
10. Branch1을 Master branc에 Merge. 이 때, 자동으로 원래의 Master, Branch2의 변경,Branch1의 변경 Commit 3개에 대한 3-Way merge가 진행되며 Fast-forward로는 진행되지 않으며 새로운 Merge commit을 Master에 생성
    1. **$ git checkout master**
    2. **$ git merge branch1**

※ 10번을 진행 시 동일 파일에 대한 변경이 중복 될 경우 Conflict 가 발생되며 조치는 다음 장을 참조

1.3-5. Git 변경 이력 통합 및 충돌 해결

3-Way Merge

② 3-Way merge 진행 시 발생한 Conflict 조치

조치 순서:

① Merge 진행 중 Conflict 발생 시에는 Conflict 메시지와 함께 작업 진행이 되지 않음

$ git merge branch1
Auto-merging index.html
CONFLICT (content): Merge conflict in [파일명]
Automatic merge failed; fix conflicts and then commit the result.

② Merge가 진행이 되지 않는 상황을 Status 명령으로 확인 **($ git status)**

③ 2번을 실행 시 Unmerged로 표시되는 파일을 수동으로 조치 **($ git mergetool 또는 사용자 지정 도구)**

④ Merge도구를 사용하지 않을 경우 조치한 파일을 다시 git add를 통해 변경 등록 **($ git add [파일명])**

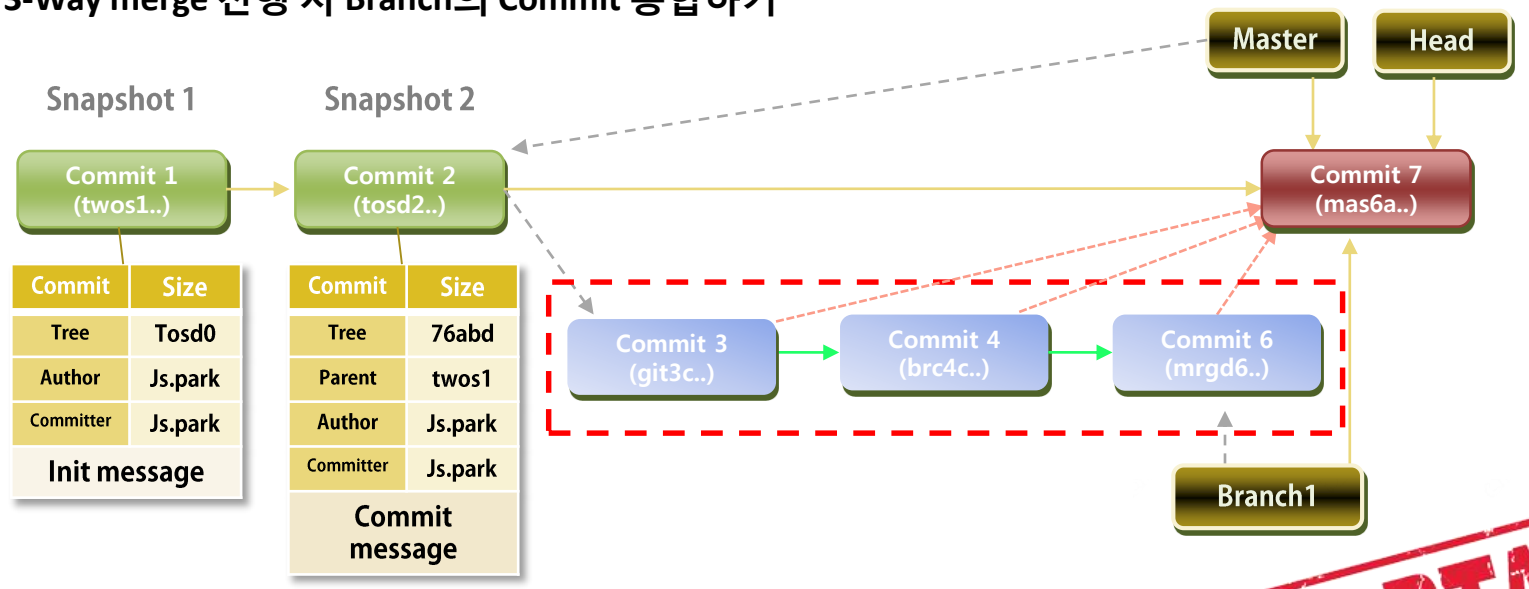
⑤ Git status 명령을 통해 변경 상태를 확인 한 후, Commit

I. **$ git status**
II. **$ git commit** -> 단, 이 때에는 Commit 에 표시되는 System 메시지가 약간 다름

1.3-5. Git 변경 이력 통합 및 충돌 해결

3-Way Merge에서 Commit 합치기

① **3-Way merge 진행 시 Branch의 Commit 통합하기**



IMPORTANT

Scenario

1) Master branch 에서 Branch1을 생성하여 수정 진행
2) Master branch 에서 Branch1의 Commit을 하나로 묶어서 하나의 Commit으로 병합 (**merge --squash** option)
3) Branch1 을 Master Branch와 Merge 후 Branch1 삭제

1.3-5. Git 변경 이력 통합 및 충돌 해결

상세 진행 순서:

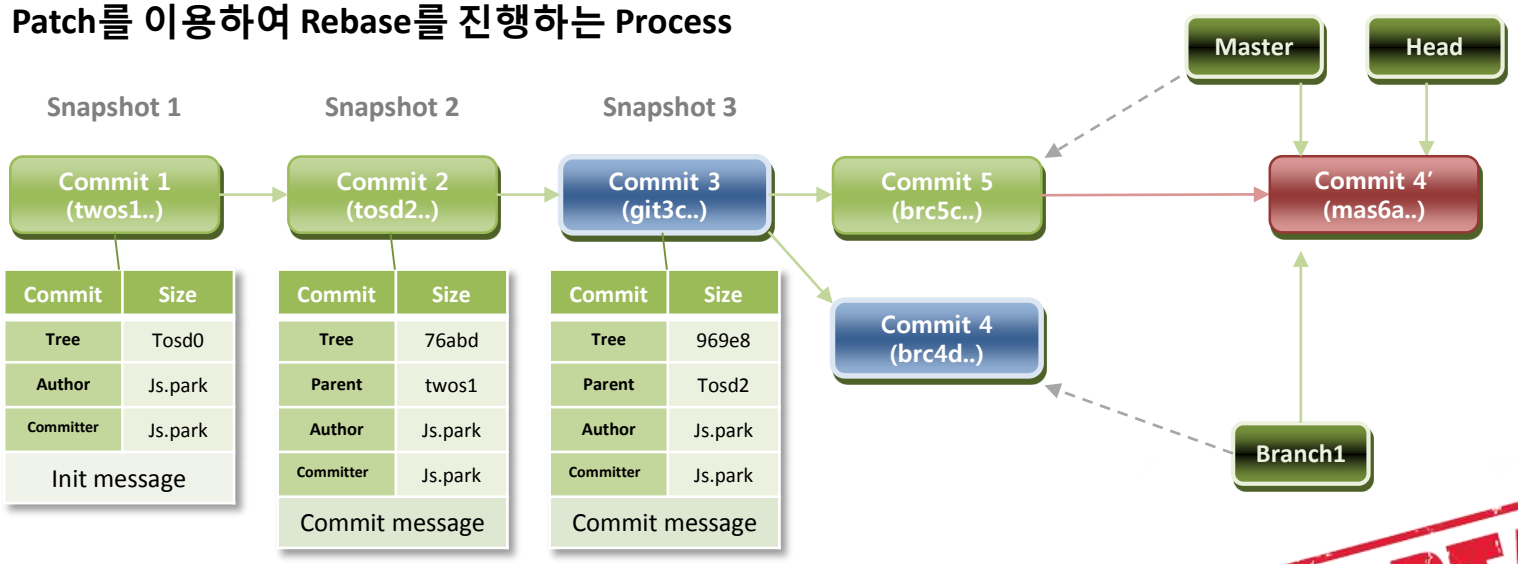
1. Branch1 을 생성 후 Head를 새로운 Branch로 이동 **($ git checkout -b branch1)**
2. 새로운 Branch1에서 파일 변경 후 Commit **($ git commit -a -m ‘Changed file to branch1)**
3. Branch1에서 추가 파일 변경 후 Commit **($ git commit –a –m 'Changed file to branch1’)**
4. Branch1을 Master branch에 Merge, 단 Squash option 추가하여 Branch의 Commit을 하나로 통합
    1. **$ git checkout master**
    2. **$ git merge --squash branch1**

※4번을 진행 시 동일 파일에 대한 변경이 중복 될 경우 Conflict 가 발생되며 조치는 merge와 동일

1.3-5. Git 변경 이력 통합 및 충돌 해결

Rebase

③ **Rebase (Merge와 비슷하지만 Commit 이력을 깔끔하게 사용 시 활용)**
Patch를 이용하여 Rebase를 진행하는 Process



Scenario

1) Master branch 에서 Branch1을 생성하여 수정 진행
2) 다시 Master branch 에서 파일을 수정하고 변경 반영
3) Branch1의 변경사항을 Patch화 하여 Master Branch에 반영 (Rebase)
4) 두 Branch가 나뉘기 전인 Commit3 으로 이동 후 현재까지 Checkout한 Branch가 가리키는 Commit까지 diff를 차례로 만들어 어딘가에 임시로 저장해 놓은 다음, Rebase할 브랜치(Branch1)가 통합 될 대상(Master branch)을 가리키는 Commit을 향하여 임시 저장해 놓았던 변경사항을 차례대로 적용한 후 새로운 Commit을 생성

IMPORTANT

1.3-5. Git 변경 이력 통합 및 충돌 해결

상세 진행 순서:

1. Branch1 을 생성 후 Head를 새로운 Branch로 이동 **($ git checkout -b branch1)**
2. 새로운 Branch1에서 파일 변경 후 Commit **($ git commit -a -m 'added file to branch1')**
3. 다시 Master branch로 Head 이동하여 작업 환경 전환 **($ git checkout master)**
4. Master branch에서 파일 변경 후 Commit **($ git commit -a -m 'added file to master')**
5. Branch1으로 작업 환경 전환 후 Master branch 에 Rebase (Branch1의 내용을 Patch화 하여 Master로 반영)
    a. **$ git checkout branch1**
    b. **$ git rebase master**

참고사항:

1. Rebase의 결과물은 Merge와 거의 동일
2. Rebase는 보통 Remote Branch에 Commit을 깔끔하게 적용하고 싶을 때 사용
3. Master branch의 이력은 선형으로 존재하므로 통합 작업 시 편의성 제공
4. Rebase는 하나의 프로젝트에서 Branch를 생성하고 또 거기서 Sub-Branch를 생성 했을 경우, Sub-Branch를 Master에 바로 통합하는 경우에도 유용하게 사용 할 수 있음
5. **모든 이력을 관리하여야 할 경우 사용을 자제하여야 함**
6. **이미 공개 저장소에 Push한 커밋을 Rebase하지 말 것!!**

1.3-5. Git 변경 이력 통합 및 충돌 해결

실습하기

- **Scenario**

1. 동시에 두 개의 Branch를 생성하고 작업 후 3-Way merge 수행하기
2. Conflict 발생 상황을 만든 후 조치하기
3. Squash option을 활용하여 Merge하기
4. Rebase를 활용하여 Merge와의 차이 비교해보기

1.3-6. Remote Branch

원격 저장소를 기반으로 Branch 활용

Chapter Target

Step #1

Remote Branch의 생성 (Branch, Checkout)

- 작업을 위해 Branch 생성하기
- 생성 된 Branch에서 작업 진행

Step #2

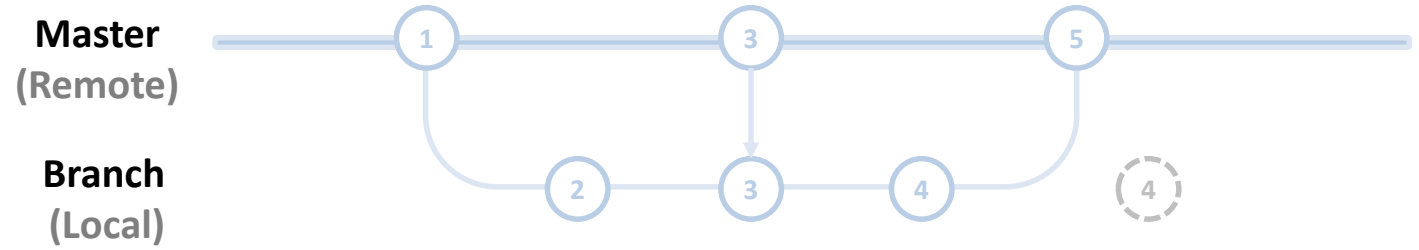
Remote의 정보를 갱신하여 로컬에 반영(Fetch)

- 원격에서 Update 된 내용을 로컬에 반영하기 (Fetch)
- 원격 Branch 추적하기 (--track)

Step #3

Branch를 Master에 통합하기 (Rebase, Merge)

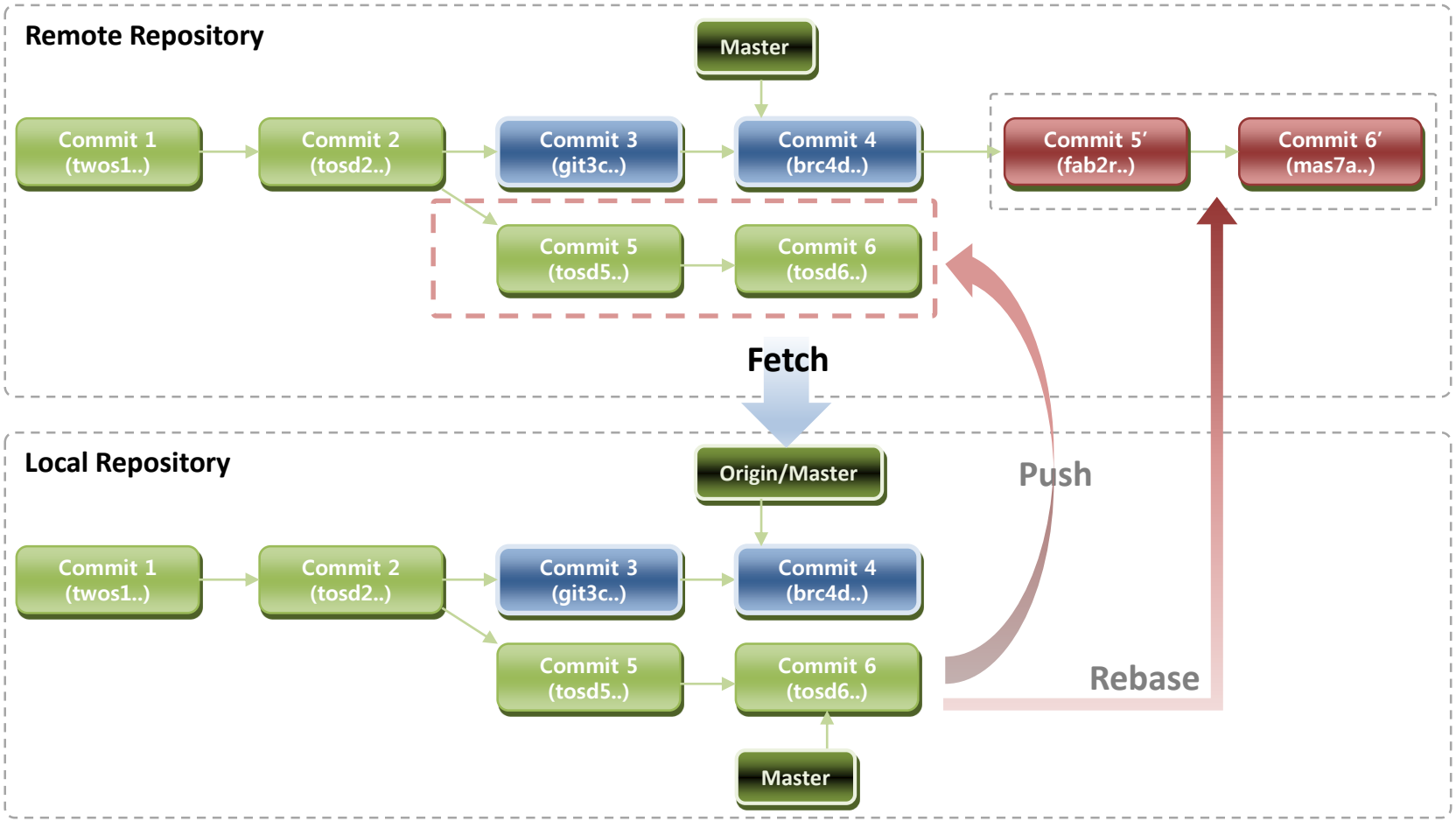
- 전체 Branch 이력을 Master 에 반영하기 (Push)
- 작업 후 로컬 Branch 삭제하기



1.3-6. Remote Branch

Remote Branch 생성 및 병합

① **진행 Process**



1.3-6. Remote Branch

Scenario

1. 원격 Repository에서 Clone으로 로컬에 프로젝트 복제
2. 로컬에서 수정 및 Commit 진행
3. Remote의 Master에 Commit이 별도로 진행되어 로컬과 동기화 (Fetch)
4. 로컬에서 작업 된 내용을 Remote에 Push
5. Rebase를 통해 원격 저장소에 새로운 Commit 등록하기

명령어 활용:

1. Remote branch의 이름은 (remote)/(branch) 형식으로 구성
2. **$ git remote show origin** : 원격의 저장소 원본 정보 보기
3. **$ git checkout -b <branch> origin/<branch> :** branch 생성과 동시에 checkout 하고 remote와 동기화 하기
4. **$ git push origin <branch> 또는 git push <remote> <branch> :** 로컬 branch 를 원격 브랜치로 push 하기
5. **$ git push origin <branch>:master :** 로컬 branch 를 remote의 master branch 에 push 하기
6. **$ git branch -D <branch> :** 로컬 branch 삭제하기
7. **$ git push origin :<branch> 또는 git push <remote> :<branch> :** 원격 branch 삭제하기
8. **$ git branch --set-upstream-to=origin/<branch> <branch> :** 원격 branch 와 로컬 branch tracking 정보 설정

1.3-6. Remote Branch

상세 진행 순서:

1. 원격 저장소의 프로젝트를 Clone하여 로컬 저장소에 생성
   - 원격의 Branch를 통해 새로운 로컬 Branch를 만들고 작업 시작 시 **($ git checkout -b branch1 origin/ branch1)**
2. 로컬 저장소의 내용 변경 후 Commit
3. 원격 저장소에서 변경 된 내용을 로컬에 갱신 **($ git fetch origin)**
4. 로컬의 변경 내역을 원격 저장소에 반영 **($ git push origin branch1)**
   - 로컬의 Branch 명을 원격 저장소에 다른 이름으로 반영을 하고자 할 경우 **($ git push origin branch1 :local1)**
5. 원격 저장소에 변경 분 반영 후 로컬의 Branch 삭제하기 **($ git push origin :branch1)**

실습하기

1. 원격의 저장소에서 Remote branch 생성하기
2. 로컬에 생성 된 Remote branch를 작업 중 원격 저장소의 내용 동기화 하기
3. 원격 저장소에 변경 내역을 Upload하기
4. Rebase를 활용하여 원격의 저장소 Update하기

1.3-7. Git Tag

Tag의 용도 및 추가, 삭제

Chapter Target

Step #1

Tag의 종류
(Lightweight, Annotated tag)

- Tag의 종류 및 용도
- 각 Tag 타입의 내용

Step #2

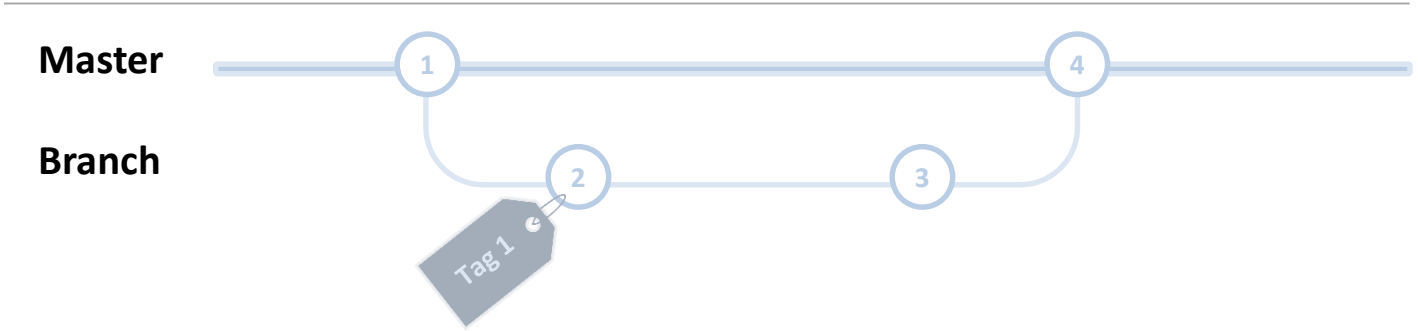
Tag 생성

- 특정 버전에 Tag를 생성
- 생성 된 Tag 정보 확인

Step #3

Tag 삭제
(Reset)

- Tag 정보 삭제



1.3-7. Git Tag

1. Tag 활용

① Tag 종류 및 사용

명령어 활용:

- Tag는 Annotated Tag 와 Lightweight Tag로 구성
  I. Annotation Tag: Tag 생성자 이름, 이메일과 Tag 를 만든 날짜, Tag 메시지를 저장하며 GPG서명 추가 가능
  II. Lightweight Tag: 단순한 특정 Commit에 대한 식별자로 별도의 정보를 저장하지 않음
- **$ git tag :** Tag 정보 보기
- **$ git tag -l '[tag 명]' :** 특정 태그 이름 검색하여 선택적 정보 보기
- **$ git show :** Tag정보와 Commit 정보를 모두 보고자 할 경우
- **$ git tag [tag 명] :** Lightweight tag 생성
- **$ git tag -a [tag 명] -m '[Tag 메시지]' 또는 git tag -s [tag 명] -m '[Tag 메시지]'**
  Annotation Tag 생성 (-s option의 경우 개인 GPG Key를 서명하고자 할 때 사용하며 **git tag -v [tag 명]** 으로 확인)
- **$ git tag -a [tag 명] -m '[Tag 메시지]' [예전 Commit의 Checksum]**
  이전 Commit 내용에 대하여 Tag 부여 시 (Checksum은 6자리 이상만 사용하면 됨)
- **$ git push origin [tag 명] :** Push 할 경우 Commit만 반영이 되므로 Tag는 별도의 Push가 필요 함
- **$ git tag –d [tag 명] :** Tag 삭제
- **$ git push origin :[tag 명] :** 원격 저장소의 Tag 삭제

1.3-7. Git Tag

실습하기

- **Scenario**

1. Tag 생성하기
2. Tag 종류 별 활용
3. Tag 이력 조회하기
4. Tag를 원격 저장소에 공유하기
5. Tag 삭제하기

1.3-8. Git 추가 기능 활용

Stash, Cherry-pick, Commit 내용 수정/삭제/통합 등 부가 기능 활용

Chapter Target

Step #1

작업의 임시저장 (Stash, Cleaning)

- 작업 중의 파일을 임시 저장
- 저장 된 임시 항목 삭제

Step #2

작업의 이력 조회 (git log)

- Commit 이력 조회
- Log의 상세 내용

Step #3

삭제파일만 되돌리기

- 작업 진행 중 삭제 된 파일만 복구하기

Step #4

Commit 이력 수정/삭제 (reset)

- Commit 이력 수정
- Commit 이력 삭제

Step #5

Bundle

- 로컬의 작업을 하나의 파일로 묶어 내보내기

Step #6

Cherry-Pick

- 다른 Branch의 작업 내용 중 특정 Commit을 현재 Branch로 갖고 오기

Step #7

특정 구문 찾기 (Grep)

- Repository 내의 파일 특정 구문 검색

1.3-8. Git 추가 기능 활용

1. 작업의 임시 저장

- **Stash를 활용한 작업의 임시 저장**

명령어 활용:

- 용도: 작업 도중 변경한 내용을 임시로 저장 후 변경 전 내용을 기반으로 다른 작업을 진행하고자 할 때 활용
- **$ git stash**
  작업 중인 변경 내용을 임시 저장소에 기록, (임시 저장 후 Workdir에서 작업 중이던 내용은 초기화 됨)
- **$ git stash list**
  임시 저장 된 내용 보기
- **$ git stash apply**
  임시 저장 된 내용을 복구하고자 하는 경우 (특정 Stash를 복구하고자 할 경우 이름을 추가)
- **$ git stash apply --index**
  임시 저장 된 내용의 복구 시 Stage 상태였던 파일의 상태를 Stage 상태인 채로 복구하고자 할 때
- **$ git stash drop**
  stash를 삭제
- **$ git stash show -p stash@{0} | git apply –R**
  Stash 적용 후 다시 원복시키고자 할 때
- **$ git stash branch**
  Stash 적용 후 많은 변경을 진행 했을 때 다시 Stash 적용 시 새로운 Branch를 만들어서 Stash저장 당시의 Checkout 된 내용과 임시 저장내용을 Merge하고 성공 시 Stash 파일을 삭제

1.3-8. Git 추가 기능 활용

2. 작업의 상세 이력 조회

- **Log 명령의 Option을 통한 상세 이력 조회**

명령어 활용:

- 용도: log를 활용하여 상세 이력 조회 및 Commit 이력 비교
- **$ git log -U1 --word-diff :** 단어 단위로 Commit 내역의 비교
- **$ git log –stat** : Commit history의 수정 파일 통계 정보
- **$ git log --name-status :** 수정된 파일의 목록 및 추가/수정/삭제 여부 확인
- **$ git log apply --since=2.weeks :** 현재부터 2 주 전 까지 한정된 정보 확인
- **$ git log –all :** 현재 Branch 이외의 다른 Branch의 Commit 이력을 포함한 전체 이력 보기

3. 삭제 된 파일만 되돌리기

- **작업 진행 중 변경 내용 중에서 삭제 된 파일만 되돌리기**

명령어 활용:

- 용도: Workdir에서 작업 진행 중인 내용 중 삭제 된 내용을 되살리기
- **$ git ls-files -d | xargs git checkout --**
  현재 작업 dir에서 삭제 된 내용만 다시 원복

1.3-8. Git 추가 기능 활용

4. Commit 이력 수정/삭제

- **Commit 된 내용의 수정 및 통합**

명령어 활용:

- **$ git commit --amend**
  최종 Commit의 내용을 현재 Stage 내용으로 저장 및 Commit 메시지 변경
- **$ git rebase -i HEAD~3**
  최근 Commit으로 부터 3번 째 까지의 Commit 내용을 병합/수정 (대화형으로 진행)
  수정 진행 시 대화형 창에서 아래 내용과 같이 변경을 진행
  - 병합: pick으로 표기 된 내용 중 합쳐질 내용을 pick에서 squash 로 변환하고 대상은 pick으로 저장
  - 순서: 항목의 순서를 재배치
  - 수정: pick이라는 단어를 edit으로 변경
- **$ git filter-branch --tree-filter 'rm -f <파일 명>' HEAD**
  프로젝트 전체에서 Commit 된 특정 파일을 삭제, --all 옵션 추가 시 모든 branch에 명령이 적용

- **Commit 이력의 삭제 (Local)**

명령어 활용:

- **$ git reset --hard HEAD^**
  가장 최신의 Commit을 삭제
  HEAD^ 는 가장 최신의 Commit을 의미하며, 최신으로 부터 2개 이상의 Commit을 지우고자 하는 경우 HEAD~2 와 같이 ~(숫자) 로 표시

1.3-8. Git 추가 기능 활용

4. Commit 이력 수정/삭제

- **Commit 된 내용을 통합하여 하나의 Commit 만들기**

명령어 활용:

- **Commit을 지운 뒤 하나의 Commit으로 통합하기 (reset --hard 활용)**
  - ① 가장 최종 Commit의 Workdir을 다른 Dir에 복사 (.git 폴더 제외)
  - ② $ git reset --hard HEAD~(지우려는 Commit 숫자) 명령을 실행하여 Commit 삭제
    - 예) 4개의 Commit을 지우려는 경우
    - **$ git reset --hard HEAD~4**
  - ③ 다른 위치에 복사 한 작업 File을 다시 갖고와 Work dir에 덮어 쓴 다음 git add 후 Commit

- **Commit을 지운 뒤 하나의 Commit으로 통합하기 (reset –soft 활용)**
  - ① $ git reset –soft HEAD~(지우려는 Commit 숫자) 명령을 실행하여 Commit 삭제
    - 예) 4개의 Commit을 지우려는 경우
    - **$ git reset –soft HEAD~4**
  - ② 1번을 통해 Commit을 삭제 한 다음 Commit
    - Soft option을 사용 할 경우, Commit은 삭제하지만 그 간의 변경 내용은 Staging에 보관

1.3-8. Git 추가 기능 활용

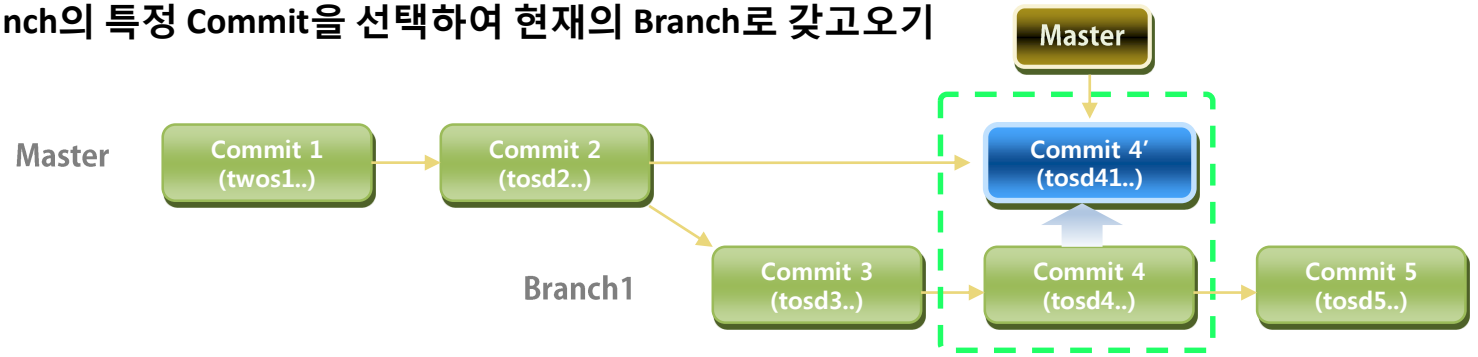
5. Bundle

- **로컬 작업 이력을 파일로 묶어서 내보내기**

  - **$ git bundle create [내보낼 파일 명] HEAD master**
    현재 저장소의 master를 모두 Export
    추가 인자 값 지정을 통해 저장소의 범위 지정 및 특정 커밋 위주의 Export가 가능하며 Clone으로 다시 Import (ex. $ git clone [Import 할 파일 명] [대상dir])

6. Cherry-pick

① **Branch의 특정 Commit을 선택하여 현재의 Branch로 갖고오기**



상세 진행 순서:

① Branch1 을 생성 후 지속적인 변경/등록 수행

② Branch1의 특정 Commit을 Master에 반영

I. **$ git checkout master**
II. **$ git cherry-pick tosd4** -> Cherry-pick 을 진행 할 대상 Commit의 키 값
III. **$ git add [파일 명]** -> 위의 Cherry-pick 수행 시 충돌 발생하는 경우에 수정 후 등록
IV. **$ git commit**

1.3-8. Git 추가 기능 활용

7. 특정 구문 검색

- **특정 구문 검색을 통해 특정 파일 또는 Commit history 찾기**

명령어 활용:

- **$ git grep –I [검색어]**
  검색어가 포함 된 파일 리스트를 조회, 검색어에 정규식의 사용이 가능.
- **Git grep에 대한 설명은 https://git-scm.com/docs/git-grep 참조**
- **$ git log--grep [검색조건]**
  Commit 메시지에 해당 검색 조건이 있는 Commit을 조회

실습하기

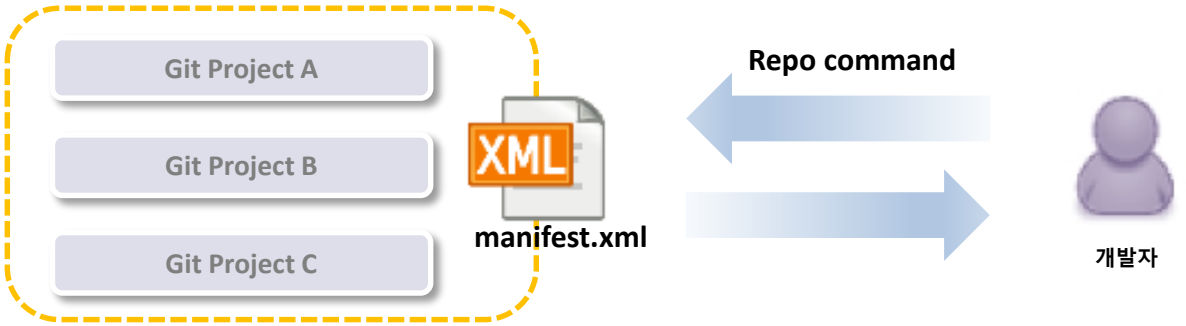
1. Stash를 활용하여 작업의 임시 저장 및 복원하기
2. Log 명령을 활용하여 이력 조회 및 Commit 비교하기
3. Working dir에서 작업 중인 파일 삭제 후 복원하기
4. Commit 이력 수정하고, Commit을 병합하기
5. Bundle을 통해 Export하고 Clone으로 Import
6. Cherry-pick을 통해 특정 Commit을 특정 Branch에 갖고오기
7. 특정 파일 검색하기

1.3-9. repo를 이용한 Multi project 관리

Repo를 활용한 멀티 Project 관리 기능

Chapter Target

Step #1	Step #2	Step #3
Git project 생성	**manifest git 생성**	**Repo 사용**
• 각각의 개별 프로젝트 생성	• 생성 된 Git 프로젝트를 기반으로 Manifest.xml 생성 • 여러 프로젝트를 묶어 하나의 프로젝트로 관리	• Repo 명령어 활용 • Repo를 이용한 여러 프로젝트 동시 활용



1.3-9. repo를 이용한 Multi project 관리

1. 여러 Git project의 통합 관리

- **Repo를 통한 Multi git project 관리**

[개요]

- **Repo**: Repository 관리 Tool로 여러 Git Repository를 묶어서 관리 할 수 있는 Python 기반 Tool
- **용도**: 여러 Git project를 묶어서 하나의 묶음으로 관리하며 내려받기 등 작업을 한 번에 진행
- **Repo 설치하기** (Ubuntu OS 기준):

① $ curl https://storage.googleapis.com/git-repo-downloads/repo > ~/bin/repo
② $ chmod a+x ~/bin/repo
③ $ vi ~/.bashrc
-> 추가: PATH=$PATH:~/bin/
④ $ source ~/.bashrc

[설정]

① **Git project 설정:** 생성된 git proejct 를 --bare 형태로 복사하고 공개 git 으로 설정
- git clone --bare <my_project> <my_project>.git

① **manifest.xml 에 Project 정보 추가:** 기본적으로 관리자에 의해 배포되며, default.xml 또는 manifest.xml 등의 이름으로 관리 되고, 사용자는 해당 파일 내에 추가 관리 할 Project를 등록 할 수 있다.

- 관리하는 프로젝트를 추가 시 <manifest> TAG 내에 아래의 내용을 해당 XML 파일에 추가

I. <default revision="master 또는 [Branch 명]" remote="origin" />
II. <project path="[로컬에 저장 되는 경로]" name="[저장소 명]" />

※ Gerrit의 url정보도 Manifest에 저장

1.3-9. repo를 이용한 Multi project 관리

2. Repo 명령 활용

- **Repo 명령어 활용**

명령어 활용: (기본적으로 git명령과 거의 유사)

- **$ repo init –u [원격 URL]**
  소스를 갖고 올 원격 URL의 선언 및 현재 DIR 사용 초기화, 특정 Branch를 갖고오기 위해서 –b option 추가 가능
  하며 –m option을 사용 시 특정 manifest file을 지정 가능
- **$ repo sync**
  init을 통해 선언 된 원격 저장소에서 소스를 내려받기, -j optio을 사용하여 Multi thread 작업이 가능
  최초 사용 시 git clone과 동일하며, 그 이후 부터는 remote update 및 rebase origin/Branch 를 실행하는 것과 동일
- **$ repo start [banch 명]**
  지정 Branch에서 작업 시작
- **$ repo upload [project 명 또는 리스트]**
  로컬의 변경 내용을 원격 저장소로 update, Review 도구에 업로드 여부에 대해 대화 창을 띄움
- **$ repo forall -c git reset --hard [특정 Tag명]**
  특정 Tag 배포 시 해당 Tag를 내려받아 반영 할 때
- **$ repo status [project 명 또는 리스트]**
  staging area의 변경 사항과 최종 Commit의 변경 사항을 비교 표시

1.3-8. Git 추가 기능 활용

실습하기

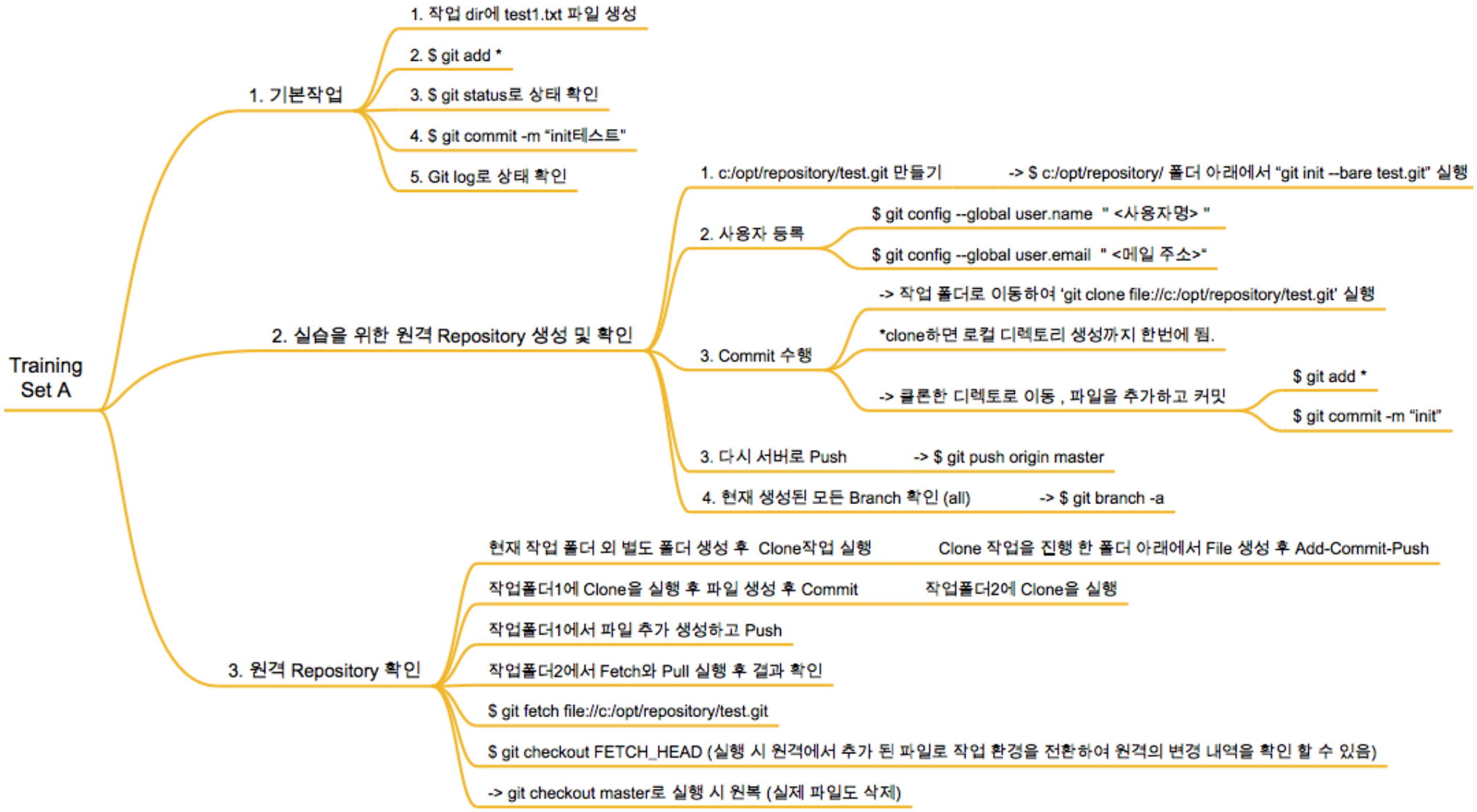
- **Scenario**

1. Repo 설치하기
2. Repo를 통해 원격 저장소의 프로젝트 내려받기
3. Manifest file에 자신의 git 프로젝트 추가하기
4. 작업 후 변경 분 Commit 하기
5. 변경 상태 확인 후 원격 저장소에 변경 분 반영하기

1.4 git Training

실습

Training Scripts A (Basics)



Training Scripts B (git cmds)

Staging, Commit, Push

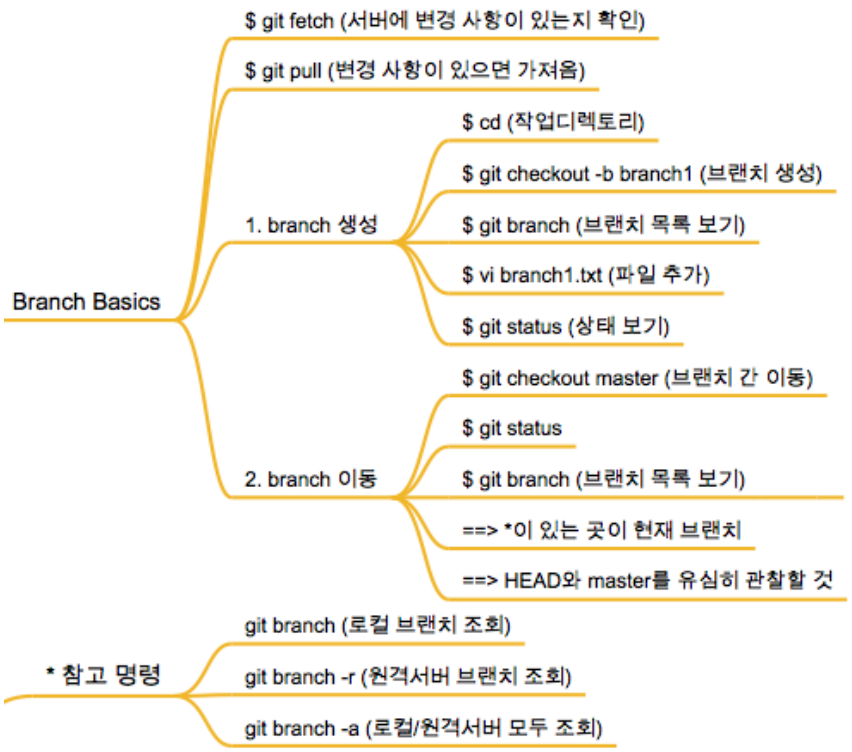
- 생성
  - $ vi(notepad) test1.txt (파일 생성)
  - 또는 touch (파일명) 이후, 탐색기와 노트패드로 작성
  - $ git status (파일 변경 메시지 확인)
- 스테이징
  - $ git add test1.txt (Commit 대상으로 추가)
  - $ git status
- 커밋
  - $ git commit
    - (커밋 및 커밋메시지 추가)
    - 또는 git commit -m "메시지"
    - 또는 git commit -a -m "메시지" (git add와 commit을 한꺼번에)
  - $ git status (커밋 후 상태 변화 확인)
- 확인
  - $ git log
  - $ gitk --all
- 푸시
  - $ git push origin master (로컬 Repo의 내용을 서버 Repo로 전송)
  - * 파일 변경 후 Commit + Push는 각자 테스트
- 참고
  - $ git remote -v
  - $ git branch
  - $ git branch -r

Clone, Pull, Fetch

- 1. 작업폴더 2 생성 및 클론
  - $ cd ~/git 현재 쓰고 있는 작업 디렉토리
  - $ git clone "생성한 (가장) 원격 리파지토리 file://c:/
  - (공동 작업을 테스트 하려고)
- 2. 작업폴더 2에 새로운 파일 추가
  - $ vi test2.txt
  - $ git add test2.txt
  - $ git commit -m "2nd developer"
  - $ git status
- 3. 작업폴더 2에서 서버로 Push
  - $ git push origin master
  - $ git status
- 4. 작업폴더 1에서 Fetch & Pull
  - $ cd ~/작업폴더 1
  - $ git fetch
  - $ git status
  - $ git pull

Training Scripts C (Branch)

Branch 만들고, 이동하고



병합 (충돌 없는 경우)

Training Scripts D (Conflict / Branch)

Conflict내기 위한 준비

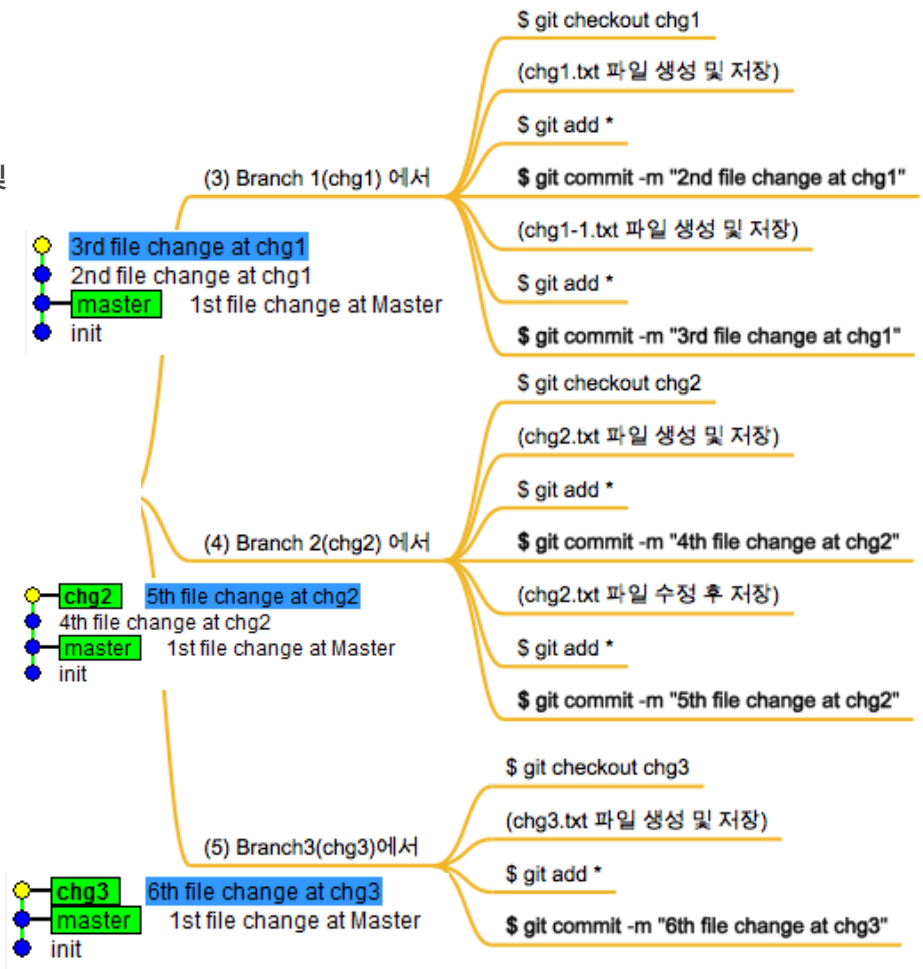
Conflict를 해소하고 Merge수행

Merge Vs. Rebase (1)

진행방안

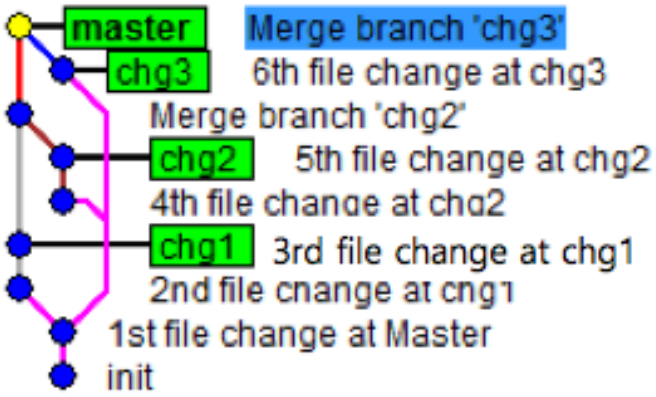
- Master에서 Branch 3개를 각각 생성(chg1~3)한 다음
- Master에서 파일을 변경하고, chg1부터 3까지 각각의 Branch에 고유 파일 생성 및 등록을 통해 변경 진행
- 단, 파일 변경 시 Branch 명과 동일 또는 유사하게 생성하고
- Commit 메시지를 "변경순서, 현재 브랜치명" 으로 입력하여 Rebase 및 Merge의 차이점을 확인 할 수 있도록

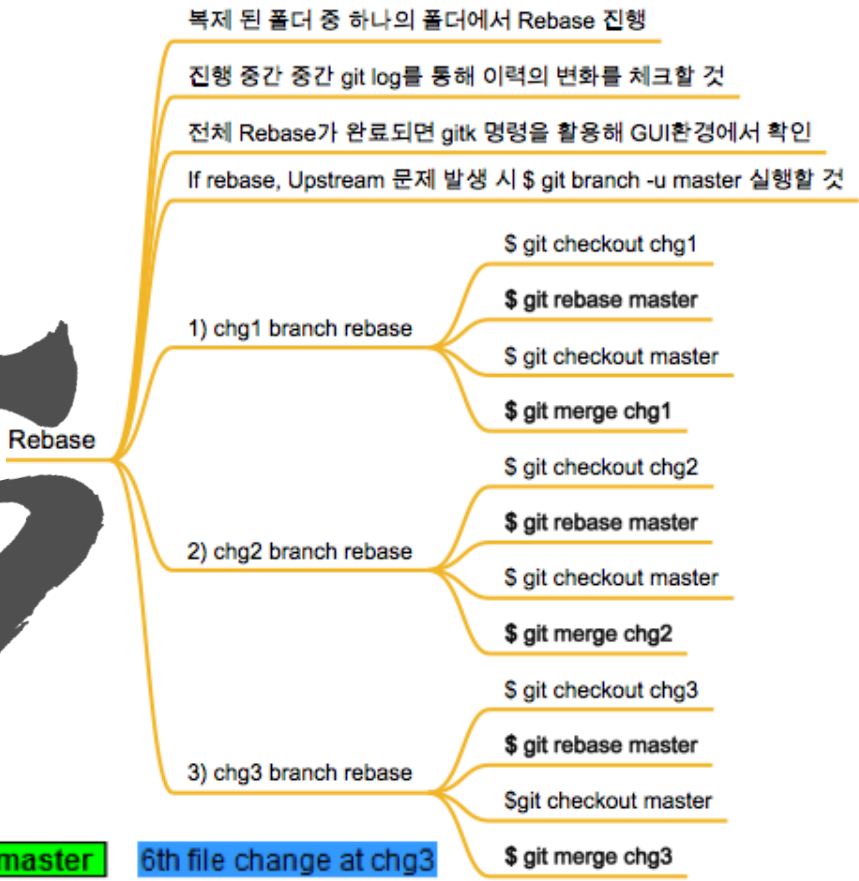
- Rebase 또는 Merge를 진행 전 해당 폴더 자체를 복제하여 Merge 먼저,
- 그 다음 Rebase를 진행 한 후, 각각의 폴더에서 Git log 및 Gitk 명령을 통해 GUI 상태에서 변경 내역을 확인
- 이 내용은 로컬에서만 진행하여 그 변화를 보다 쉽게 확인 할 수 있도록 진행

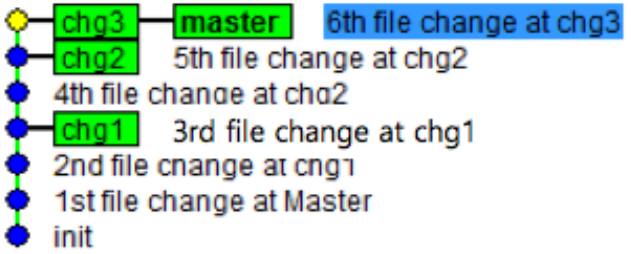


- 여기까지 작업 후 해당 작업 폴더를 복제하여 2개의 폴더로 분리하고 각 폴더의 이름을 Rebase , Merge로 분리

Merge Vs. Rebase (2)



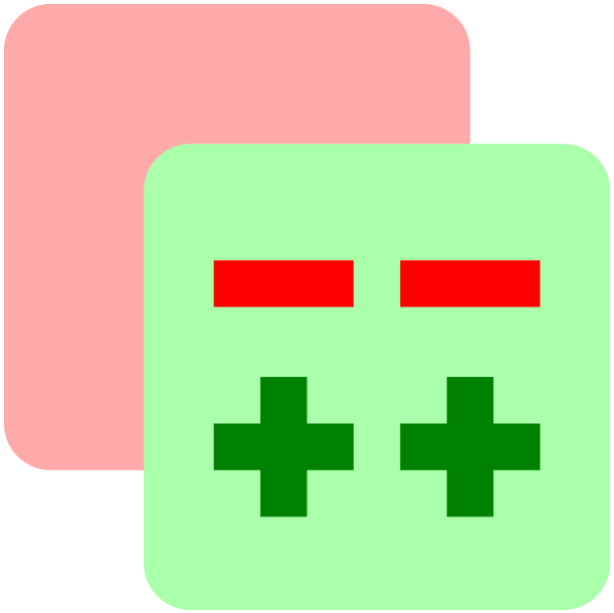




2.1 Gerrit ?

이론

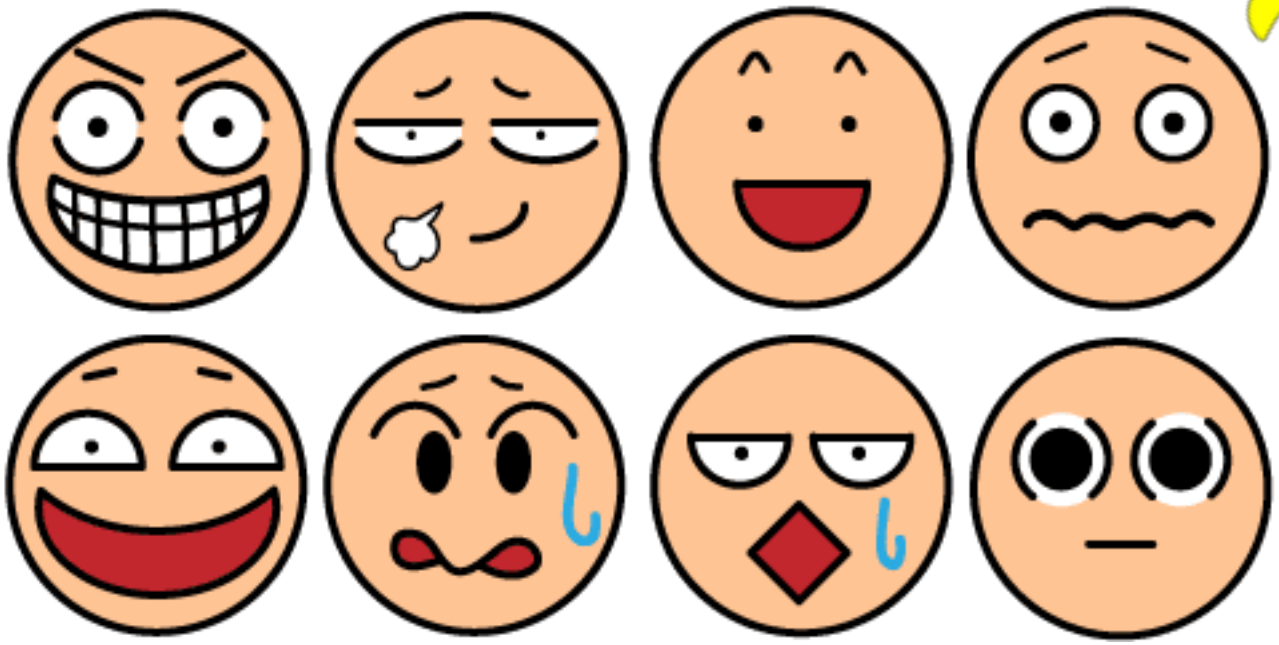
Gerrit?



지금까지 다소 생소할 수도 있는 git에 적응하시느라 고생많으셨습니다.

D-VCS도 생소 한데, 이제 코드 리뷰란 것까지 살펴보겠습니다.

Code Review?



흠... 여러분이 어떤 표정일까요? Gerrit의 아이콘

DIFFY 쿵후 리뷰 배꾸기가 보여주는 장난기처럼 코드 리뷰도 장난기 있게 하는 게 중요합니다.

Gerrit by Wikipedia

원저자	구글
최근 버전	2.13.5 / 2017년 1월 6일, 110일 경과
운영 체제	자바 플랫폼, 엔터프라이즈 에디션
플랫폼	자바, 서브릿, GWT
언어	영어
종류	코드 리뷰
라이선스	아파치 라이센스 v2
웹사이트	www.gerritcodereview.com

1. **게릿(Gerrit)**은 무료 웹 팀 코드 협업 도구이다. 소프트웨어 개발자가 팀에서 웹 브라우저를 사용해 소스 코드의 다른 사람의 수정 사항을 검토하거나 변경 사항을 승인 또는 거부할 수 있다. 분산 버전 관리 시스템인 Git과 밀접하게 통합된다.

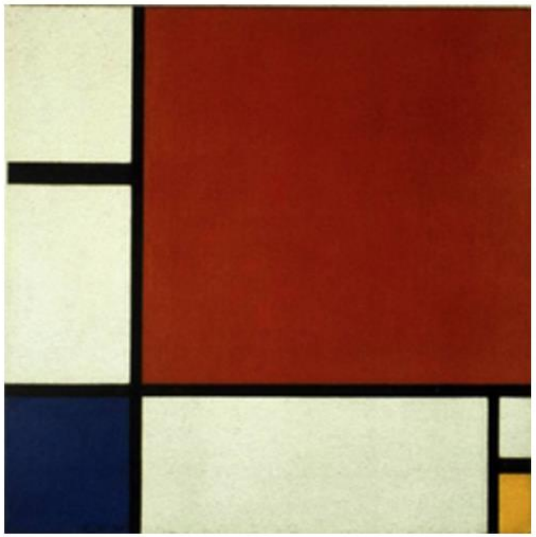
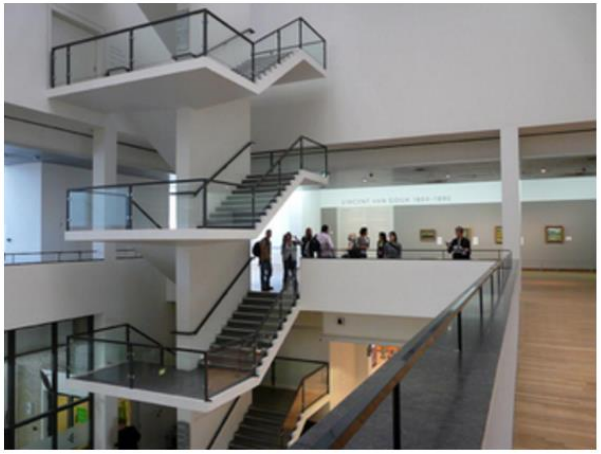
2. 게릿은 또 다른 코드 리뷰 툴인 Rietveld의 포크이다. "게릿"은 **Rietveld**라는 명칭의 유래가 된 네덜란드 개발자 Gerrit Rietveld(게릿, 리트벨드 1888-1964)의 이름이다.[1]

3. **포크**(fork) 또는 **소프트웨어 개발 포크**(project fork) : 개발자들이 하나의 소프트웨어 소스 코드를 통째로 복사하여 독립적인 새로운 소프트웨어를 개발하는 것을 말한다.)

4. 소프트웨어 리뷰 툴인 Rietveld 용 패치 묶음에서 시작하여 포크되었고 ACL 패치가 Rietveld로 합쳐지지 않으면서 저자인 귀도 반 로섬(Guido van Rossum)에 의해 별도의 프로젝트로 진화했다.[3]

5. 게릿은 안드로이드 프로젝트의 개발을 위해 션 피어스(Shawn Pearce, JGit의 설립자)에 의해 구글에서 개발되었다.[2]

Gerrit History

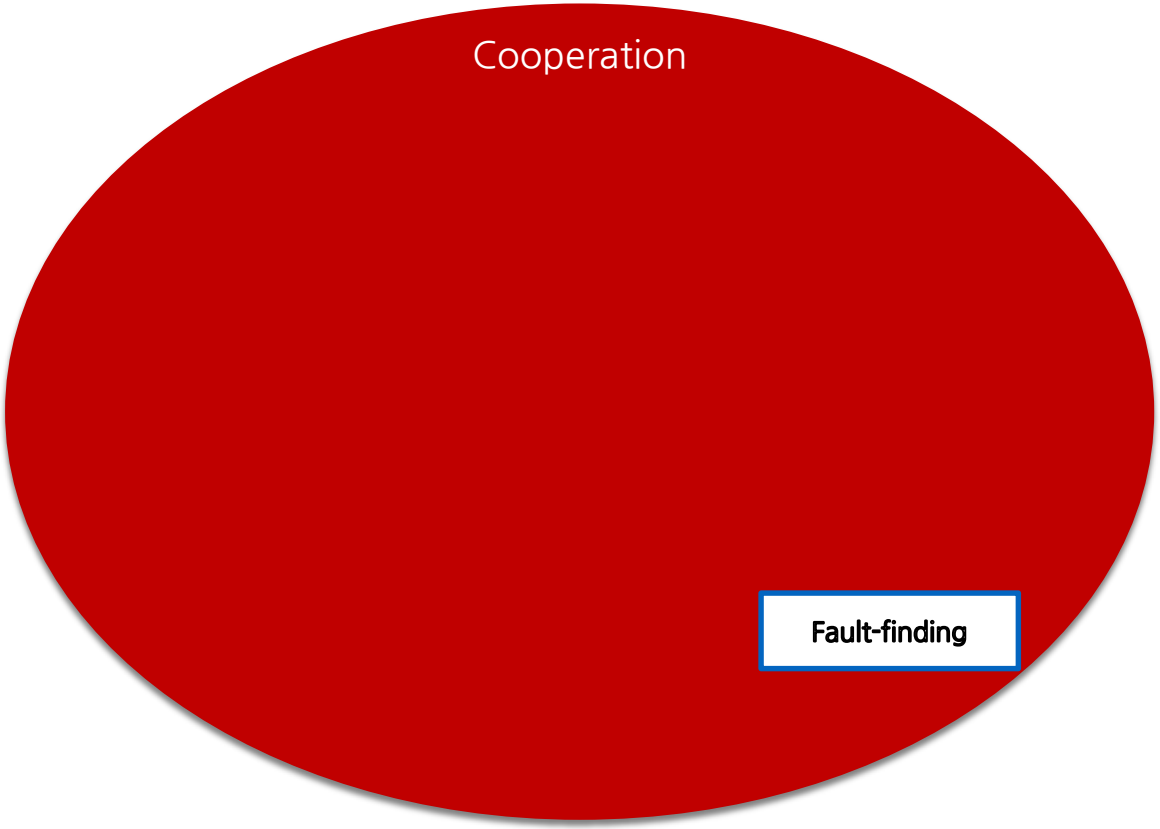
Guido Van Rossum	Mondrian : 코드 리뷰 시스템	RietVeld
• Creator of Python • 한때 구글러(2005 - 2012) • 지금은 Dropbox에 drop되어 있음 • 	Guido 할아부지의 첫번째 미션  • Written by Guido van Rossum • Written with Python • Announced in 2006 • Integrated with Perforce • Hosted and Used atGoogle Internal	 • Written by Guido van Rossum • Written with Python • Announced in 2008 • Integrated with Subversion • Host on Google Ap Engine • Used by Chrome Project

- Gerrit (2008 - 2012) 쉽게 말하는 결론 :
  귀도 할아버지가는 구글 들어가서 몬드리안이란 코드 리뷰 시스템 개발하고 있었고, 크롬 개발하느라고 리트벨드로 진화
  이를 션 피어스(Sean O. Pearce) 가 리트벨드 포크로 개발한 것

Code Review is...

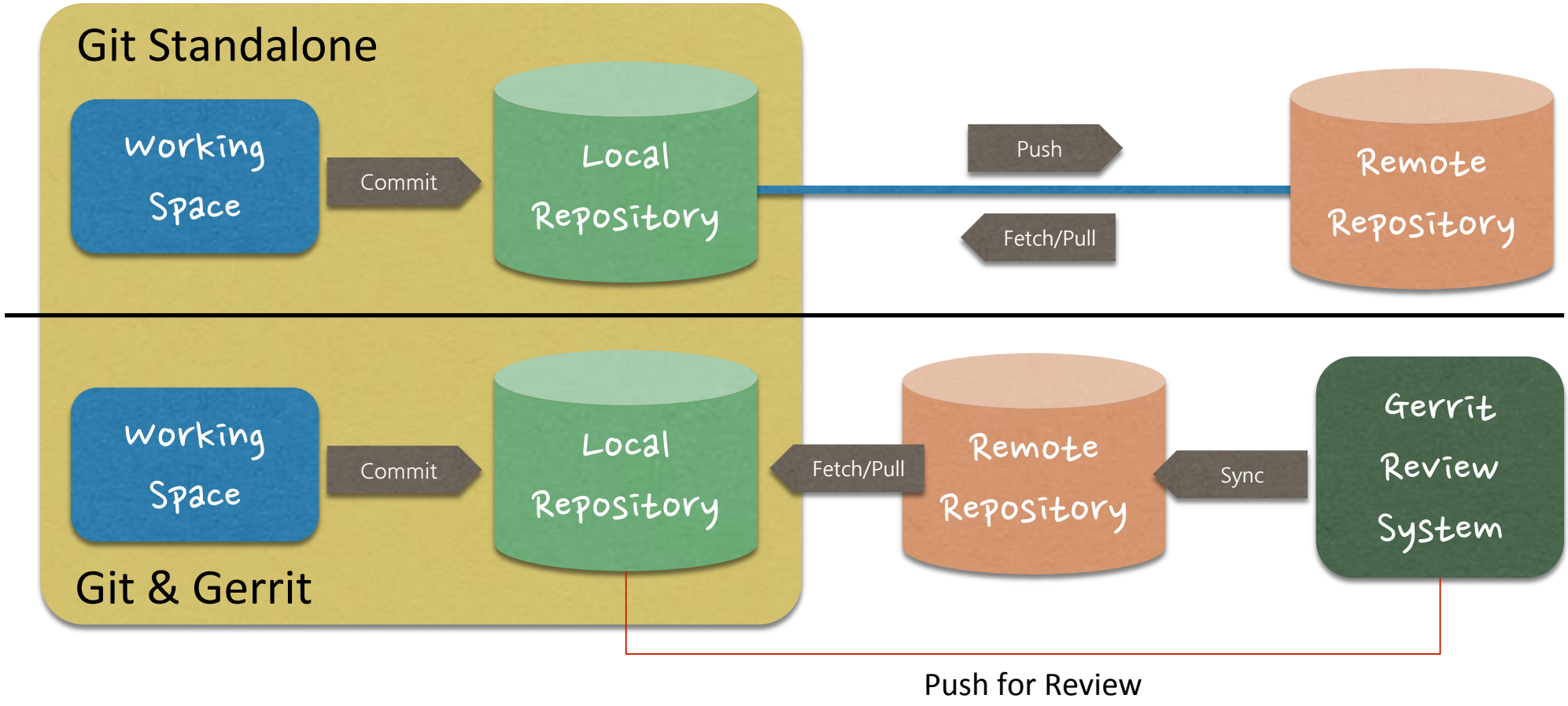
Wikipedia : It is intended to find and fix MISTACKES overlooked

Guido van Rossum says : " Goal is COOPERATION, not fault-finding "



Git Standalone Vs. Git & Gerrit

Gerrit이 git과 같이 구성되면 구조는 아래와 같이 달라집니다.



Gerrit : Overview

리뷰 중인 코드 / 리뷰할 코드 / 리뷰 완료된 코드 목록

- ✓ 웹 기반의 리뷰 UI / 팀의 작업 방식 선택 가능 / 자동화하기 쉬움, | 커뮤니티 / 문서화

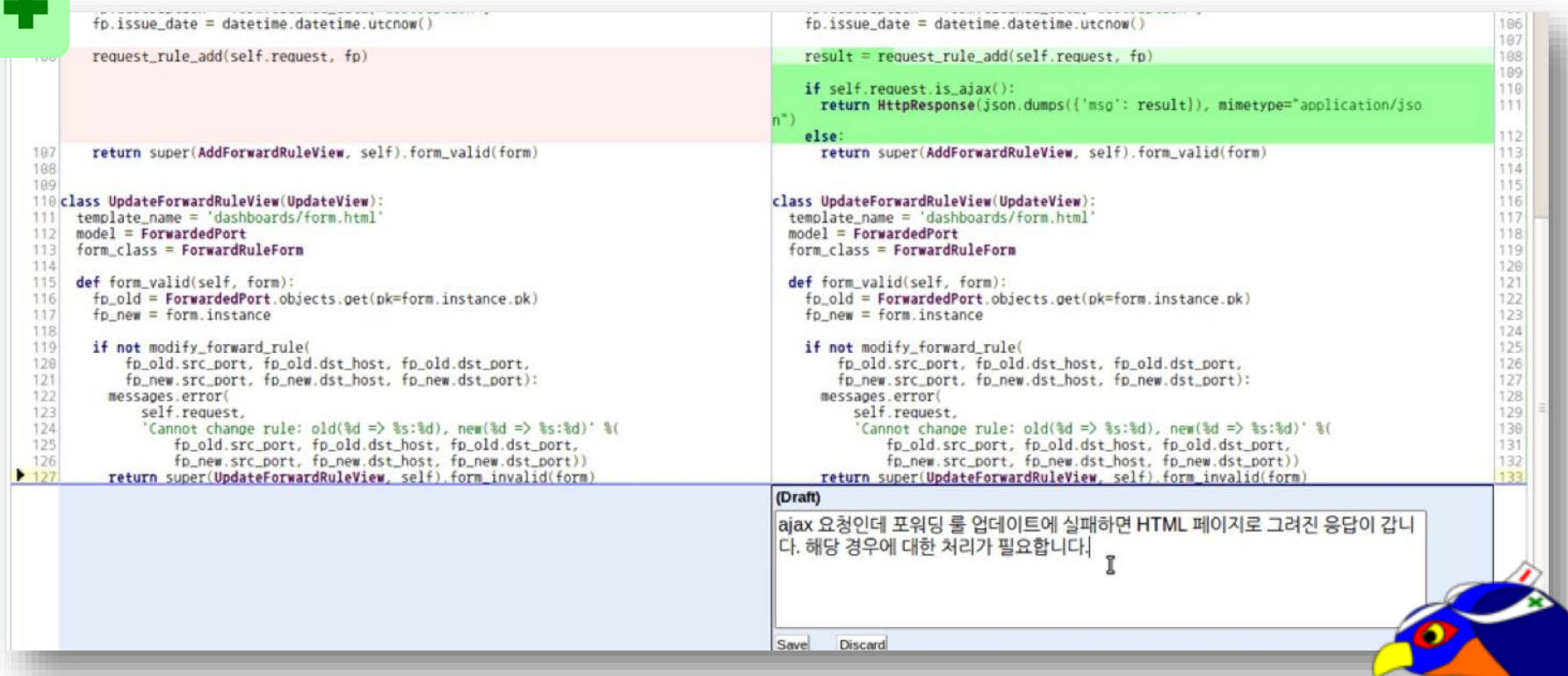
Gerrit

TwoSeed

SAMSUNG

Gerrit : diff.view

리뷰할 코드의 diff를 보면서 의견 제시



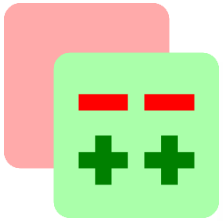
✓ 리뷰할 때는 문제점을 찾는다기 보다, 같이 코딩한다는 마음으로... Pair Coding !!!

TwoSeed

SAMSUNG

Gerrit : review-vote

Repository에 넣을 지 (+2) / 다른 사람의 의견을 더 들을 지 (+1,0) / 추가 작업할 지 (-1,-2)



Change Id23e24f3 - Patch Set 3: Publish Comments

Change-Id:	Id23e24f36760eae2f24422342dee4ac0c2647036
Owner	Miyu Park
Project	argus
Branch	master
Topic	
Uploaded	2013-01-21 18:29
Updated	2013-01-21 18:43
Status	Review in Progress

Commit Message — Permalink

gd-dashboard 전체 UI 수정

테이블 태그의 스타일 및 class 수정
각 페이지의 form 기능의 modal화
Deploy 부분의 게임 패키지 파일 업로드 ajax 수정
파일업로드를 위한 jquery.form.js 추가

Issue:600

Change-Id: Id23e24f36760eae2f24422342dee4ac0c2647036

Reviewer	Verified	Code-Review
Riab Nk	✓	
Jinuk Kim		
Jioh L. Jung		
Suwon Jang		
Esun Kim		

- Need Code-Review

Name or Email or Group — Add Reviewer

Code Review:

- +2 Looks good to me, approved
- +1 Looks good to me, but someone else must approve
- 0 No score
- -1 I would prefer that you didn't submit this
- -2 Do not submit

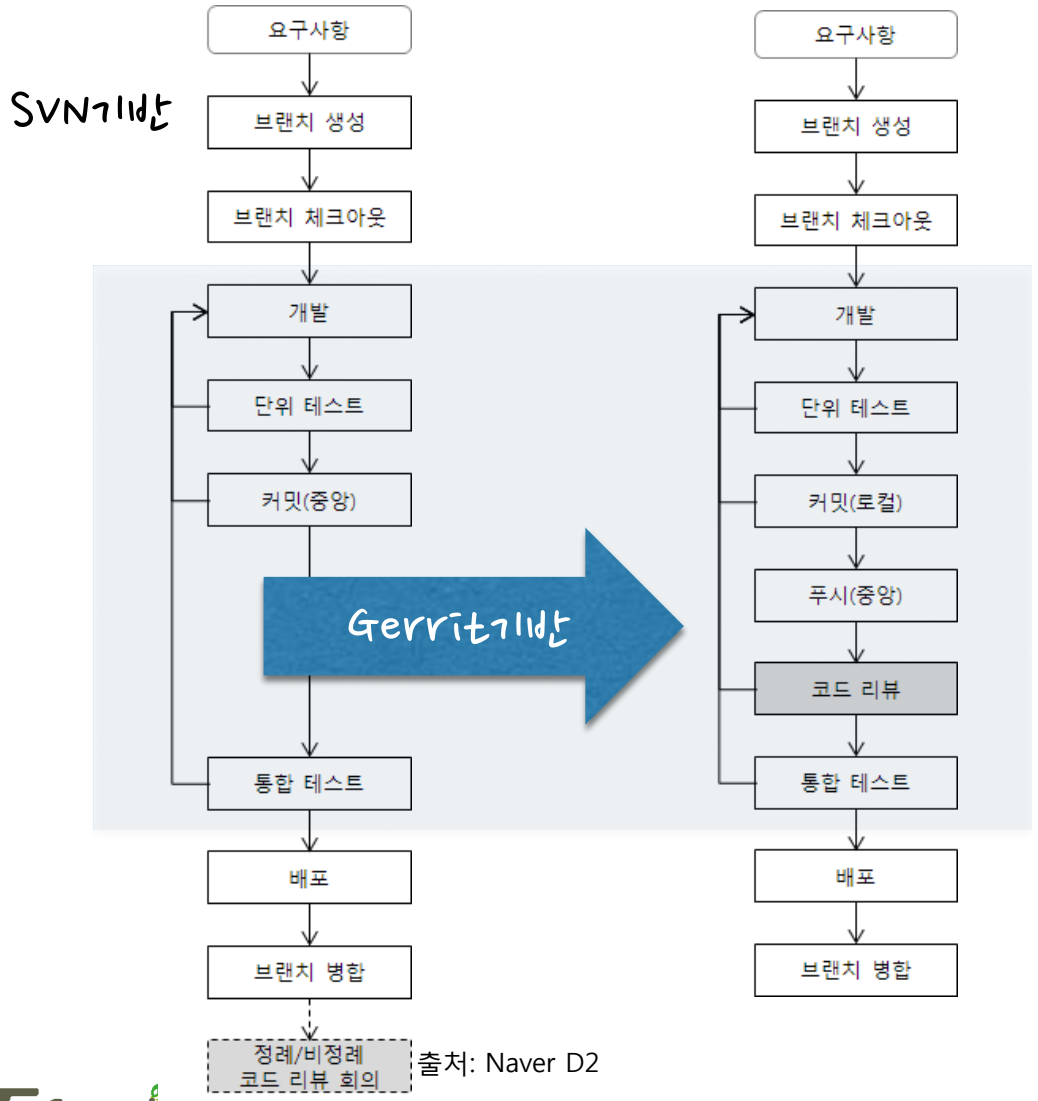
Cover Message:

ajax 요청 / 일반 요청이 섞였을 때 처리가 실패하는 부분에서

2.2 Gerrit 의 구조

이론

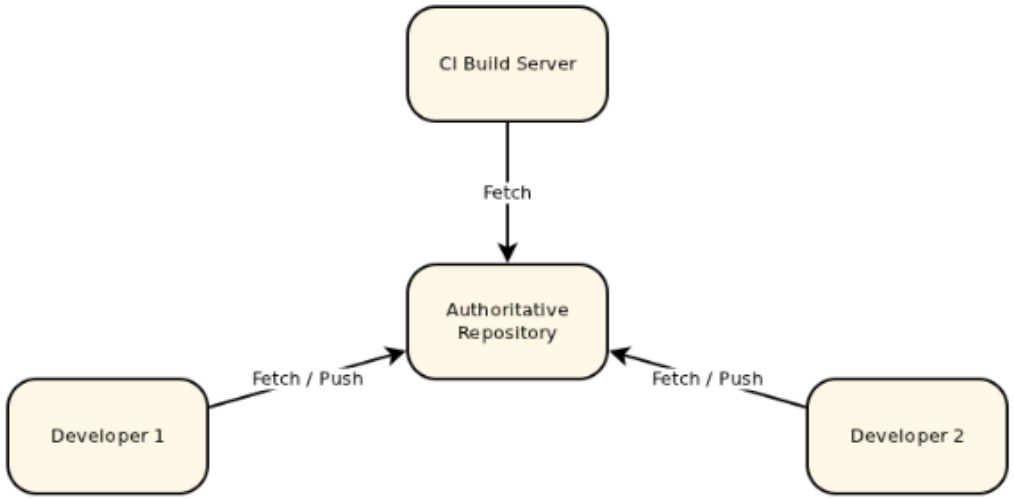
Gerrit은 소스 Merge 과정에 Code Review 포함



출처: Naver D2

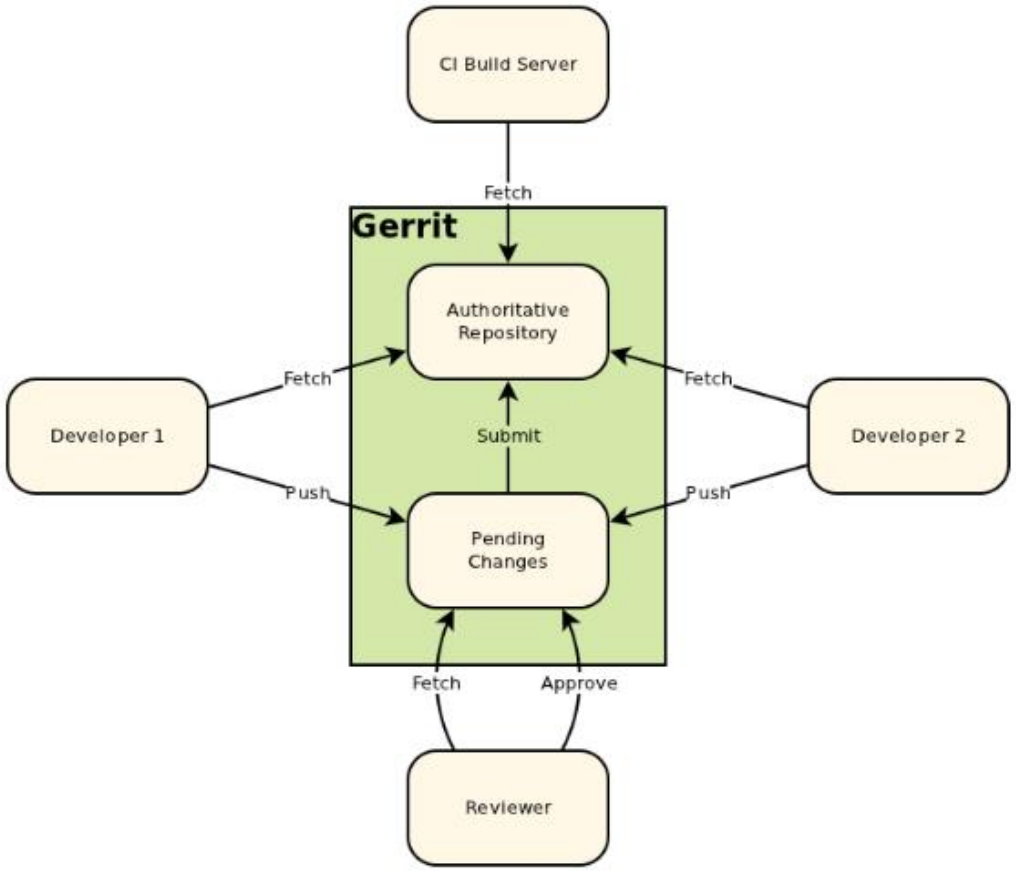
- ✓ SVN은 병합 후 코드 리뷰를 별도로 수행
- ✓ Gerrit은 소스 Merge(Push) 과정에 code review를 자연스럽게 포함
- ✓ git 프로세스와 통합
- ✓ Jenkins 연동을 통해 품질측정 결과를 Review 시에 활용 가능
- ✓ IDE 도구와 연동

접근 제어가 없는 기존 Git Repository 구조



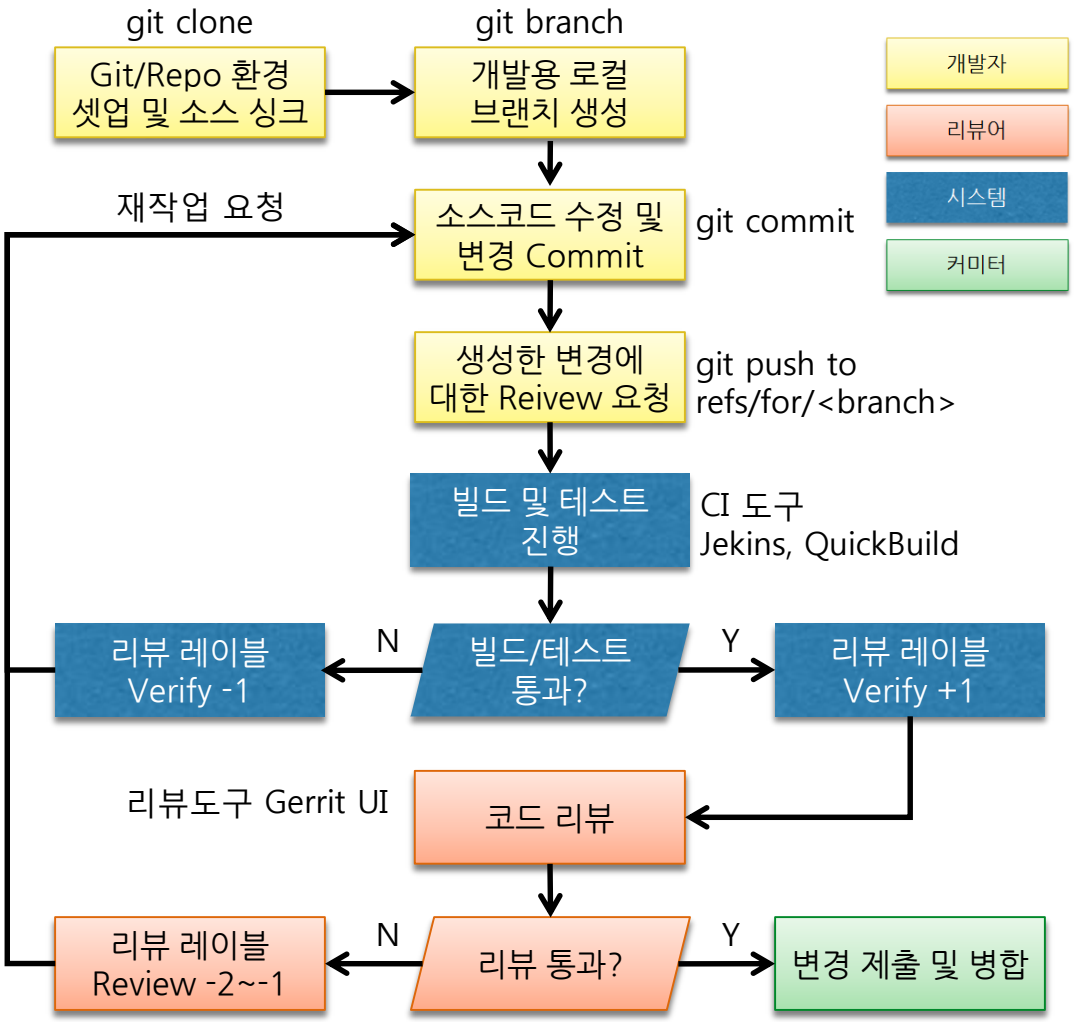
- ✓ 개별 작업이 가능한 분산버전관리 시스템
- ✓ 공동 개발 과정에서 일관성을 위해 중앙저장소(Authoritative Repository)를 둠
- ✓ 개발자는 각각 개별 로컬 Git 저장소(Developer1, Developer2) 사용
- ✓ CI 빌드서버 역시 로컬 저장소 사용
- ✓ 이 과정에 접근 제어를 하지 않기 때문에 누구나 소스를 받거나 저장할 수 있음

접근 제어와 Review 과정을 추가한 Gerrit Repository



- ✓ 중앙저장소를 기능 별로 분리하여 접근 제어 관리
- ✓ Review를 위한 임시저장소(Pending Changes)
- ✓ 최종 소스가 저장되는 중앙저장소 (Authoritative Repository)
- ✓ 개발자는 최신 소스를 가져올 때(Read)만 중앙저장소 사용
- ✓ 변경 사항 저장은 임시저장소에
- ✓ 리뷰어는 임시저장소에서 변경 사항 조회하여 Review 수행

Gerrit 기반 Code Review 워크플로우 예시



- ✓ 개발 시작 (Gerrit 프로젝트 생성)
- ✓ 개발자 로컬저장소로 소스 싱크
- ✓ 개발용 브랜치 생성

- ✓ 코드 작성 후 리뷰 요청(리뷰 브랜치로 Push)

- ✓ CI 도구로 자동 코드 검증
- ✓ 리뷰어 Code Review

- ✓ 변경 사항 적용, 중앙저장소 (Authoritative Repository)에 Merge 및 Conflict Resolution

Git에서 커밋 ID 관리

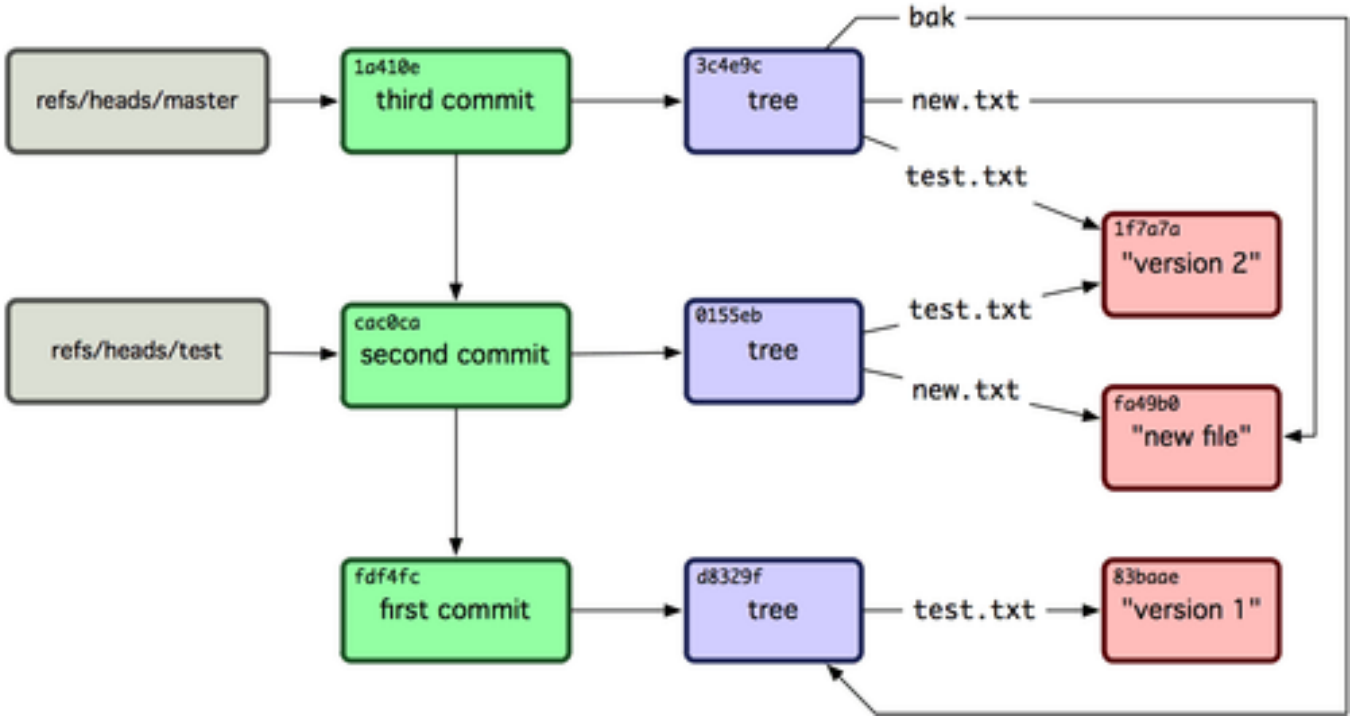
✓ Git에서 내부적으로 특정 커밋을 가리키는 커밋 ID는 SHA-a Hashing 값

✓ 사람이 보거나 이해하기에 불편

```
author     admin <admin>
           Tue, 28 Oct 2014 14:59:41 +0000 (16:59 +0200)
committer  admin <admin>
           Tue, 28 Oct 2014 14:59:41 +0000 (16:59 +0200)
commit     1ed15a62cbb570c1579c9acc3a491d2275a9cae4
tree       2c2df8b8aa331723ab687a0f400f6267a4f0afa2       tree | snapshot
```

특정 커밋에 대한 알기 쉬운 포인터 Reference

- ✓ SHA-1 Hashing으로 된 **커밋 ID에 의미있는 이름을 부여한 것**
- ✓ 각 브랜치, 태그 등의 Referencer가 .git/refs 디렉토리에 파일로 저장
- ✓ 기본으로 사용되는 master 브랜치는 /refs/heads/master 파일에 저장
- ✓ 참고로 Gerrit에서 Review를 받으려면 **"/refs/for/브랜치"로 Push**



Gerrit에서는 Review를 위해 Change-Id를 사용

```
commit aeef0e3d6e4a1db31c3ca5ec3fcddc4dc6c2c7fd
Author: mytory <mytory@gmail.com>
Date:   Mon Feb 24 04:31:05 2014 +0900

    test.

    Change-Id: Iaea0f4d225beed878b3d17d2ca421c93d22f7e96
```

✓각 Review를 구분하기 위한 식별자

✓커밋 메시지의 마지막에 추가해야 함

✓커밋을 수정하거나 cherry-pick, rebase 한 경우에도 변경을 방지하기 위함

✓일반적으로 Prefix로 "I"를 붙임

출처:https://git-scm.com/book/ko/v1/Git%EC%9D%98-%EB%82%B4%EB%B6%80-Git-%EB%A0%88%ED%8D%BC%EB%9F%B0%EC%8A%A4

Commit Hook을 이용한 Change-Id 자동 생성

```
scp -p -P 29418 mytory@gerrit.domain.com:hooks/commit-msg .git/hooks/
chmod u+x .git/hooks/commit-msg
```

✓커밋 메시지 작성 시, 매번 change-id를 생성하기는 매우 불편

✓Git의 commit Hook 스크립트를 이용하여 자동 생성 기능 지원

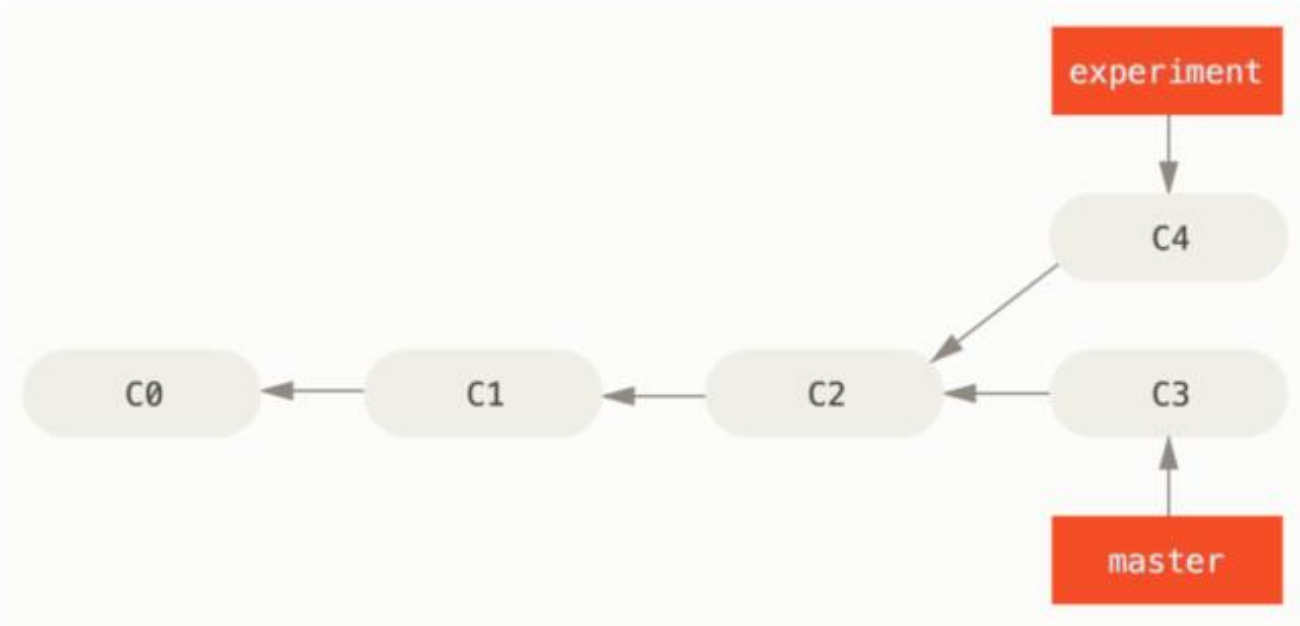
출처:https://git-scm.com/book/ko/v1/Git%EC%9D%98-%EB%82%B4%EB%B6%80-Git-%EB%A0%88%ED%8D%BC%EB%9F%B0%EC%8A%A4

Git에서 Commit Hook의 활용

```
MacBook-Pro-2:hooks jerryj$ ls -l
total 104
-rwxr-xr-x  1 jerryj  staff   478  4 27 10:02 applypatch-msg.sample
-rwxr-xr-x  1 jerryj  staff  4691  4 27 10:18 commit-msg
-rwxr-xr-x  1 jerryj  staff   896  4 27 10:02 commit-msg.sample
-rwxr-xr-x  1 jerryj  staff   189  4 27 10:02 post-update.sample
-rwxr-xr-x  1 jerryj  staff   424  4 27 10:02 pre-applypatch.sample
-rwxr-xr-x  1 jerryj  staff  1642  4 27 10:02 pre-commit.sample
-rwxr-xr-x  1 jerryj  staff  1348  4 27 10:02 pre-push.sample
-rwxr-xr-x  1 jerryj  staff  4951  4 27 10:02 pre-rebase.sample
-rw-r--r--  1 jerryj  staff   544  4 27 10:02 pre-receive.sample
-rwxr-xr-x  1 jerryj  staff  1239  4 27 10:02 prepare-commit-msg.sample
-rwxr-xr-x  1 jerryj  staff  3610  4 27 10:02 update.sample
MacBook-Pro-2:hooks jerryj$
```

- ✓ 특정 Hook 이벤트(Git 명령)가 실행될 때 자동으로 실행되는 스크립트
- ✓ .git/hooks에 저장되는 실행 가능한 모든 스크립트 (예: Shell, Perl, Python)
- ✓ 샘플 스크립트들이 기본적으로 들어 있으며 “.sample” 확장자만 제거하면 동작
- ✓ 로컬저장소에서 동작하는 Client Hook과 서버 중앙저장소에서 동작하는 Server Hook이 있음 (자세한 내용은 Git 매뉴얼 참고)

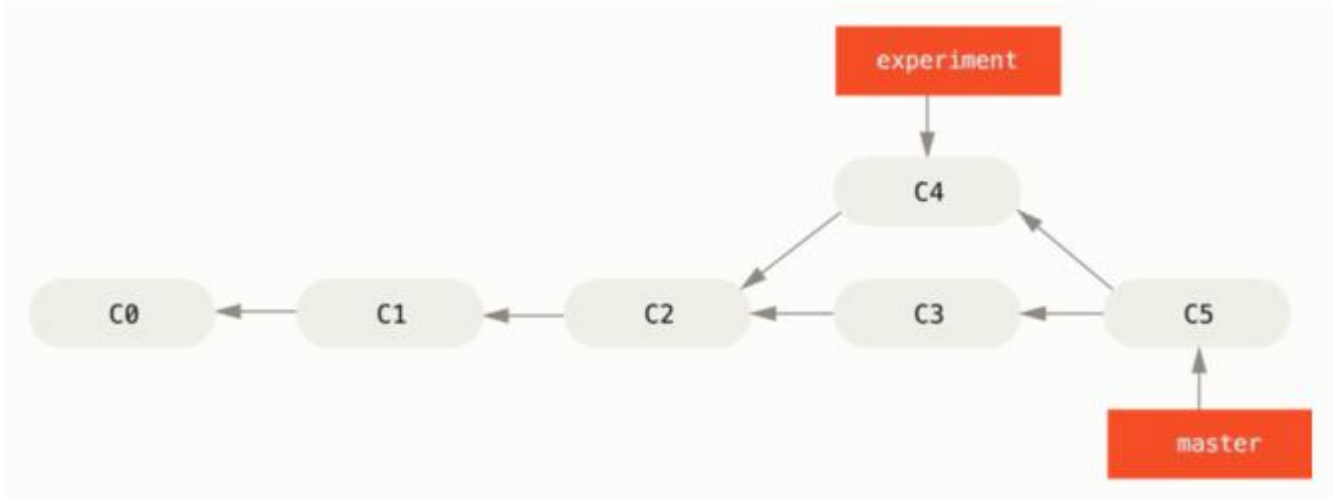
Git 브랜치 합치기



✓공통 조상인 C2를 기준으로 두 개의 브랜치 experiment(C3), master(C4)

✓Git에서 브랜치를 합하는 방법은 두 가지: Merge, Rebase

Git Merge를 이용한 브랜치 합치기

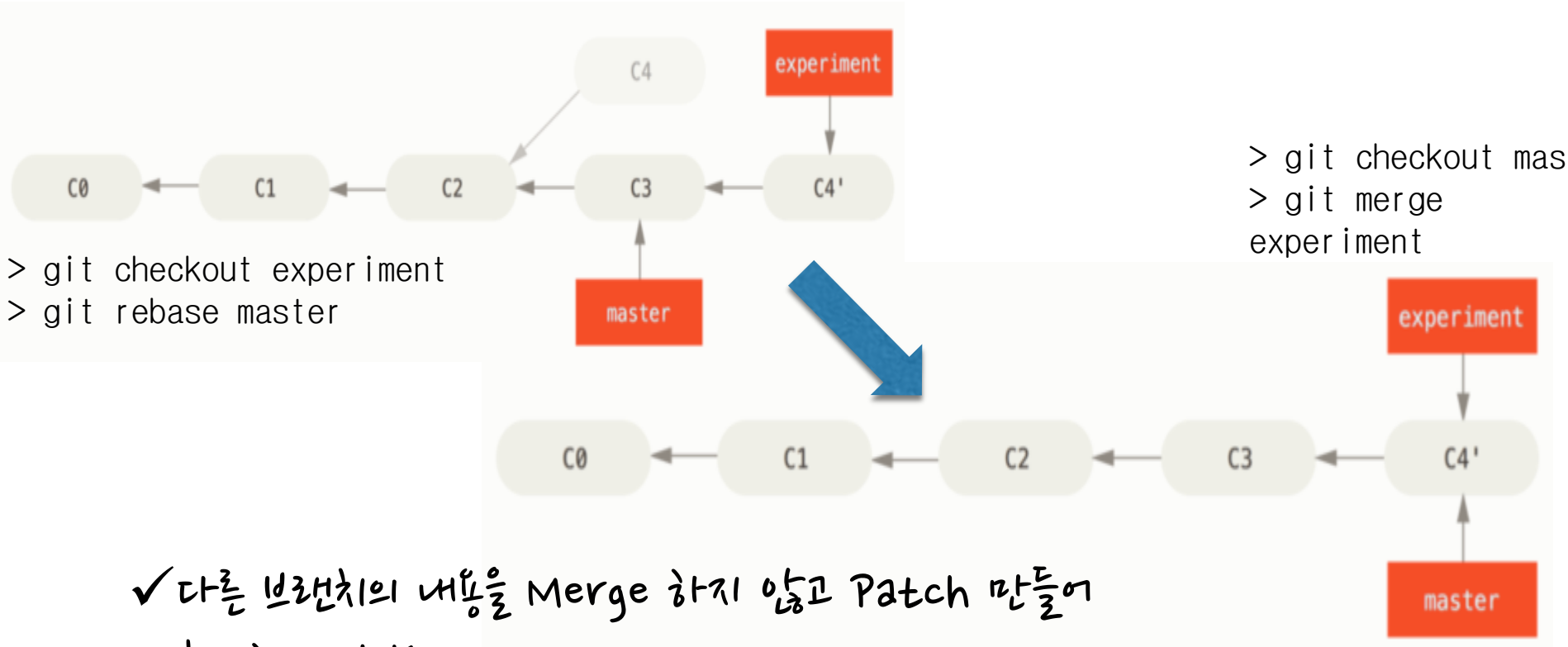


✓Git Merge를 이용한 일반적인 브랜치 합치기

✓다음 git merge 명령을 통해 C3, C4의 변경 내용을 합하여 master(C5) 브랜치 생성

```
> git checkout master
> git merge experiment
```

Git Rebase를 이용한 브랜치 합치기

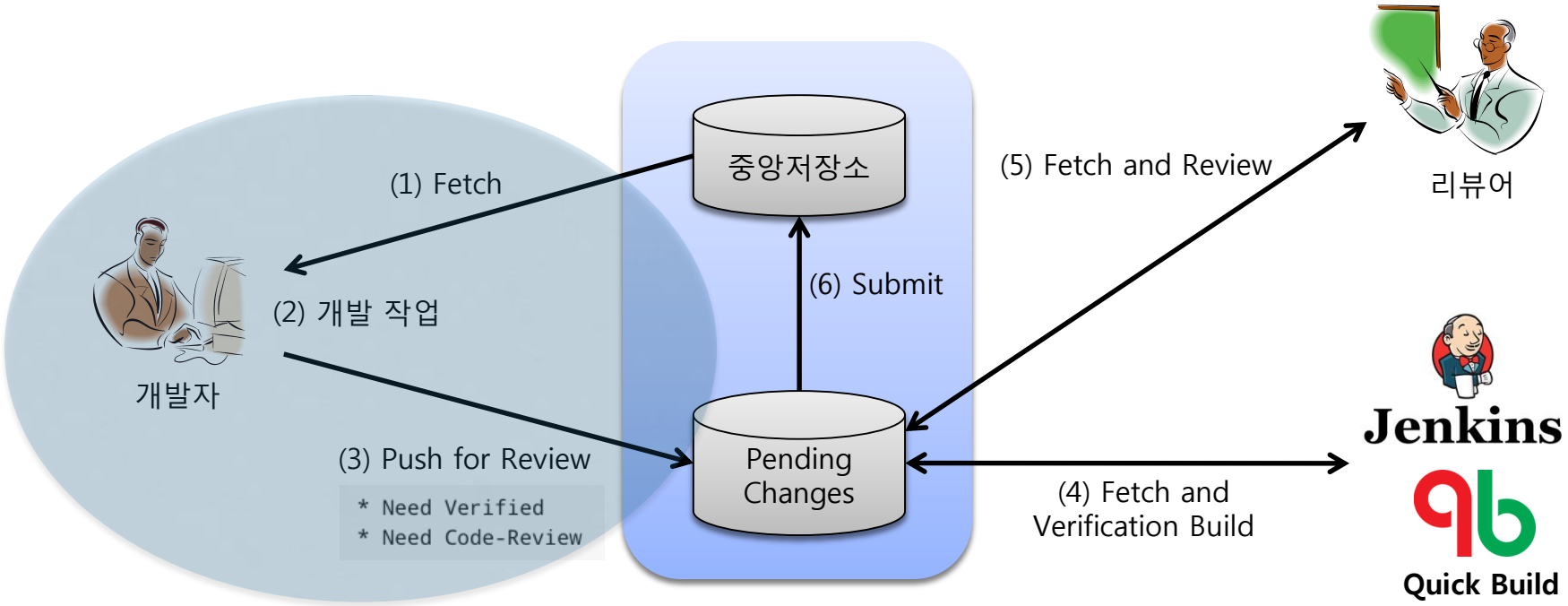


✓다른 브랜치의 내용을 Merge 하지 않고 Patch 만들어 추가하는 방법

✓히스토리가 트리 구조가 아닌 선형. 단, 최종 소스는 Merge와 동일

✓서버에 Push(다른 사람과 공유)한 소스에 Rebase하면 안 됨

Review 요청 및 처리 과정

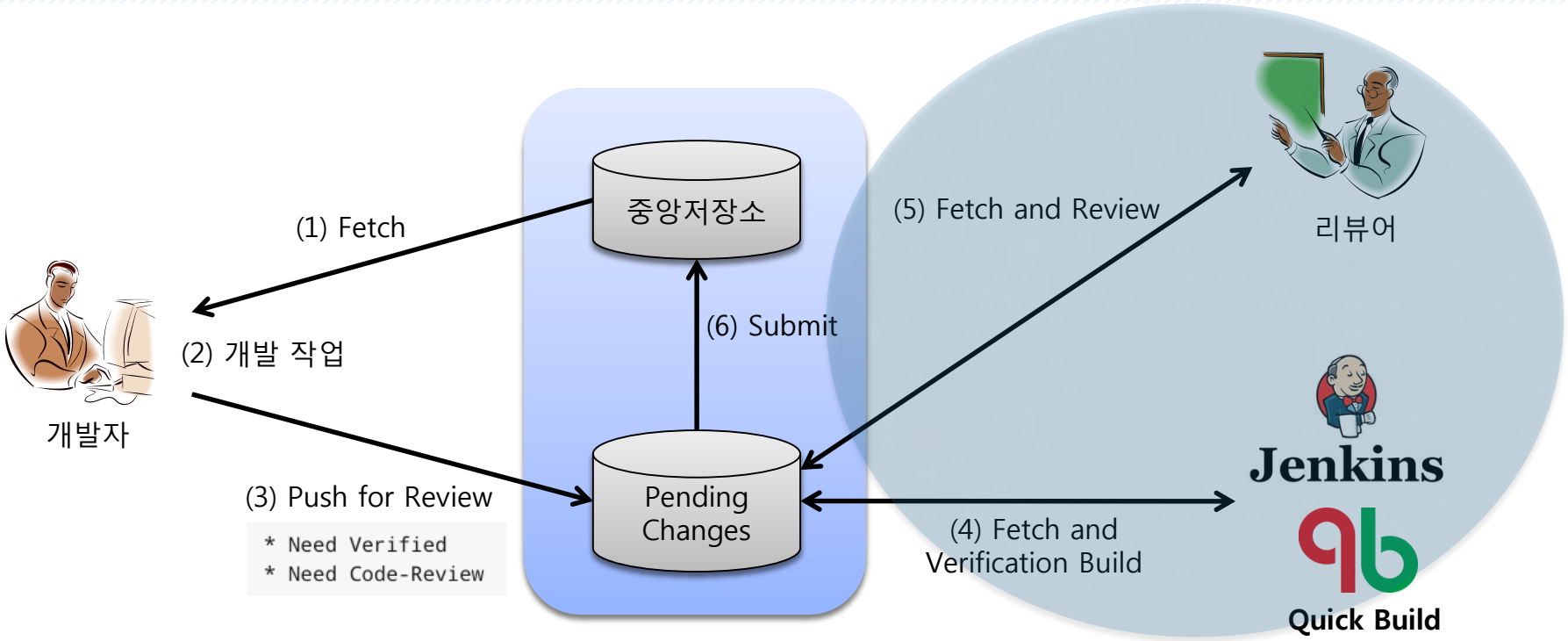


✓ 개발자는 중앙저장소에서 가져온 소스를 기준으로 개발 작업

✓ 변경된 소스에 대한 Review 요청을 위해 임시저장소로 Push

✓ Push 과정에 verify, Review 요청을 선택

Verify와 Review

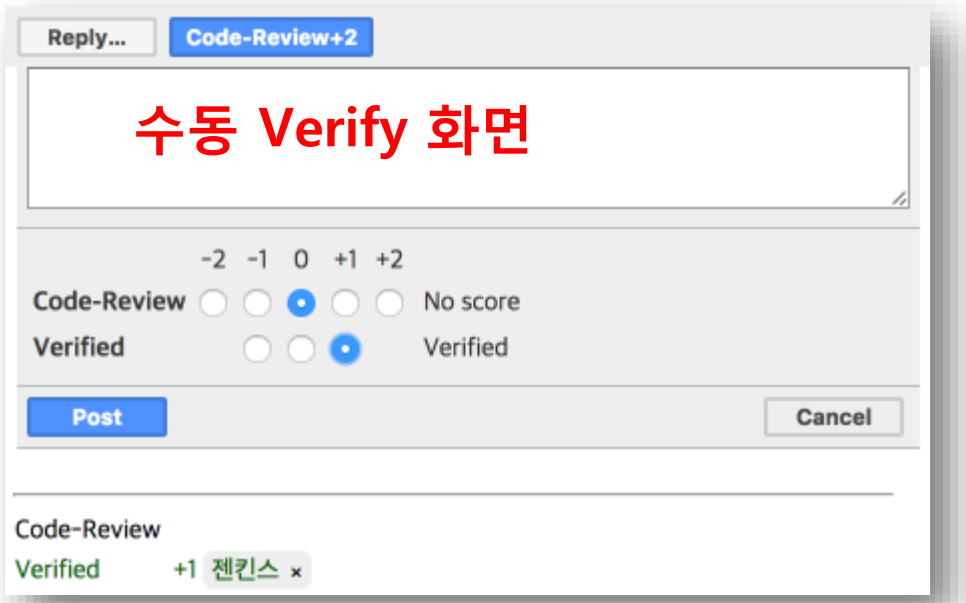


✓ verify 과정에 코드가 컴파일, 실행 등이 정상적으로 되는지 검사

✓ verify는 주로 CI(Jenkins 등)에 의해 자동으로 수행 됨

✓ Review는 리뷰어에 의해 코드가 프로젝트의 가이드라인에 따라 작성 되었는지 검사

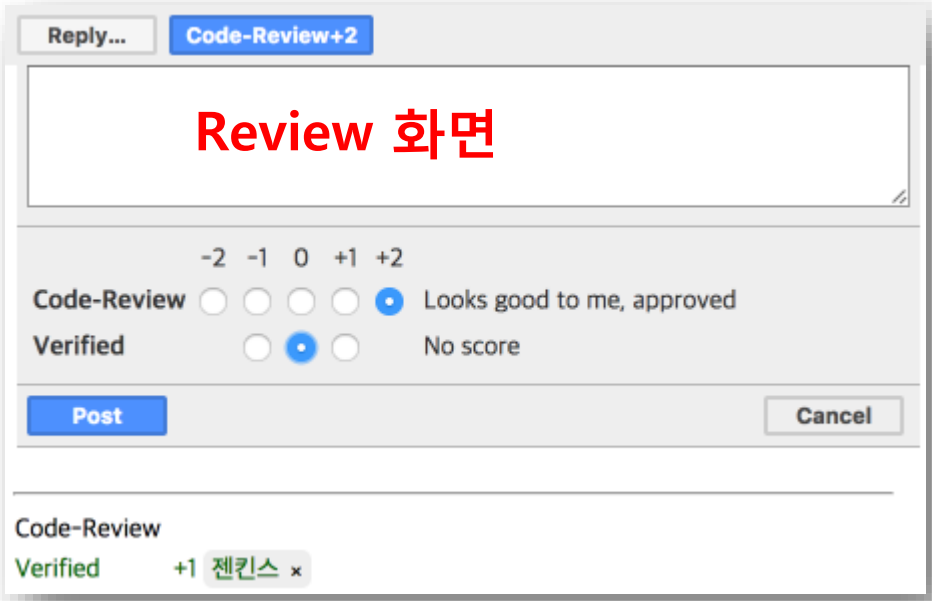
Verify와 Review



✓일반적으로 자동으로 처리

✓수동 verify 가능

✓+1이면 통과

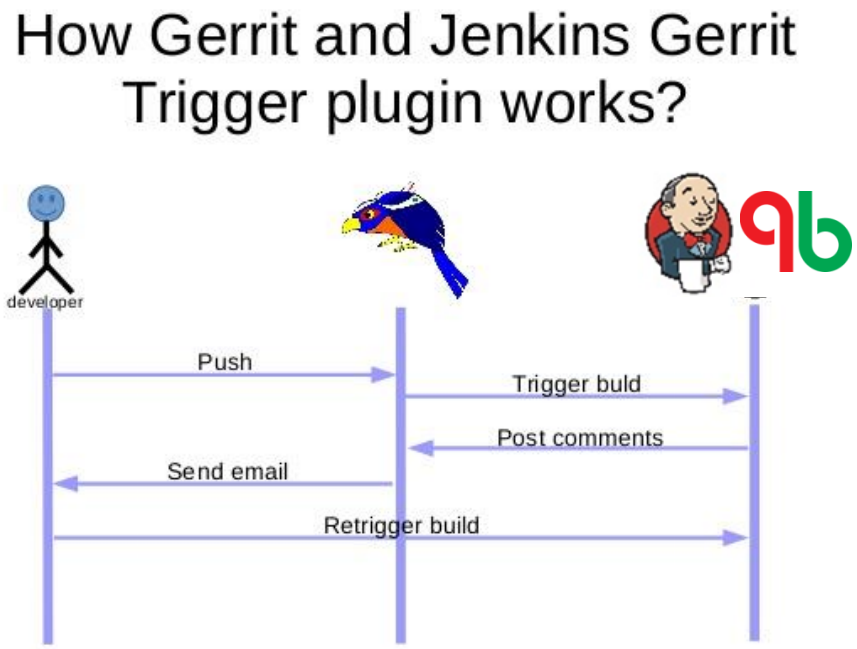


✓리뷰어가 하는 일반적인 Review

✓일반적으로 결과가 +2이면 통과

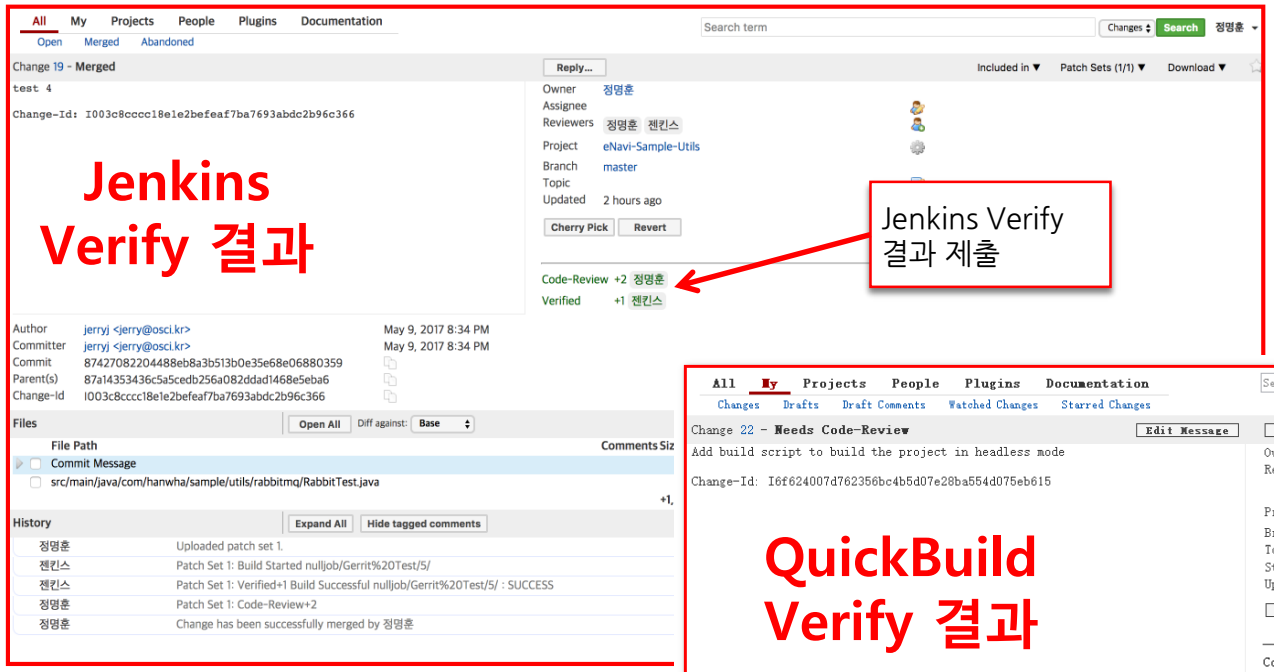
✓결과가 -2이면 Merge 하면 안 됨

CI(Jenkins, Quck Build)를 통한 Verify 과정

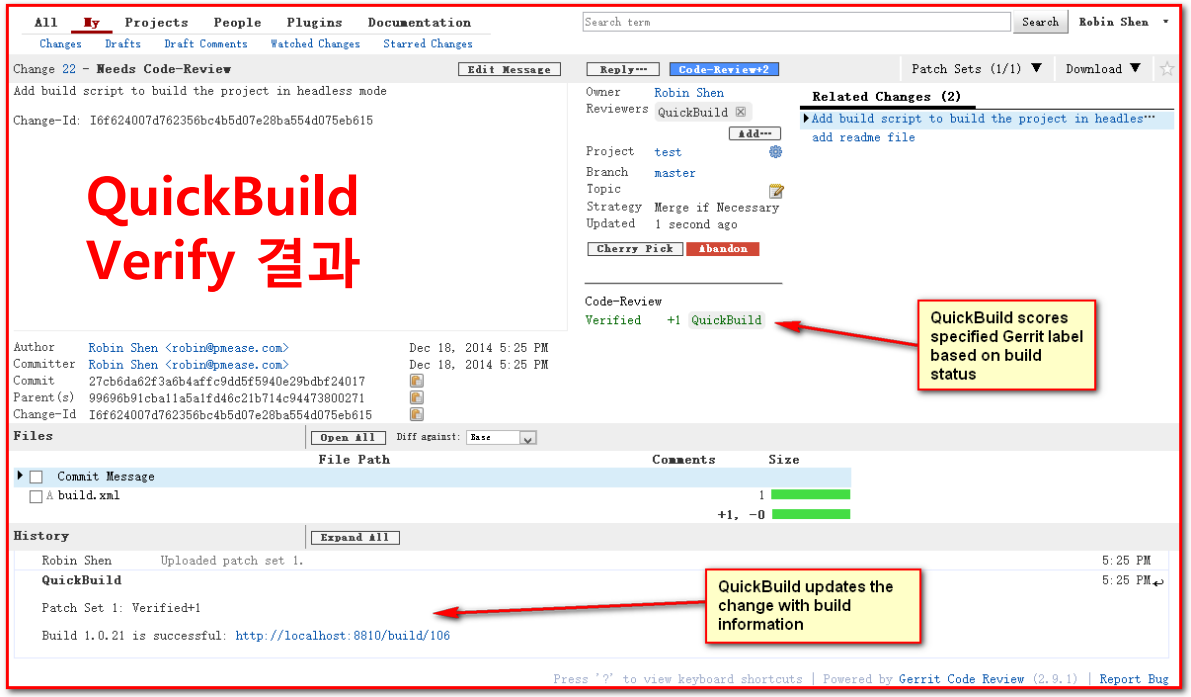


- ✓ Jenkins는 Git Client Plugin, Git Plugin, Gerrit Trigger Plugin을 사용
  – Gerrit에 리뷰가 제출되면 빌드 트리거가 동작하여 Git 저장소로부터 소스를 가져와 빌드/테스트 후 결과를 제출
- ✓ QuickBuild는 Gerrit이 제공하는 REST API를 통해 리뷰 제출을 감지하여 빌드를 트리거하며, 빌드/테스트 결과를 제출
- ✓ 빌드에 성공하면 Verified/+1을 부여

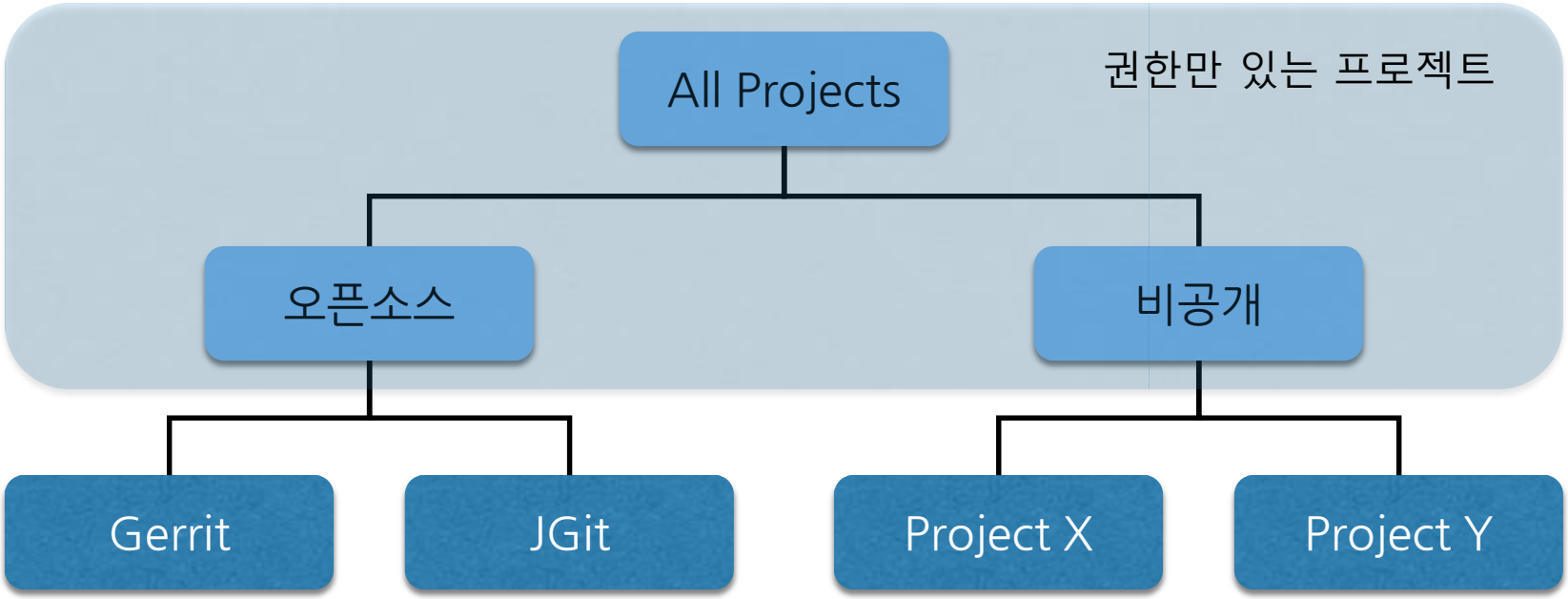
CI(Jenkins, Quck Build)를 통한 Verify 과정



✓ verify 결과에 CI 도구가
사용한 Gerrit 계정 표시

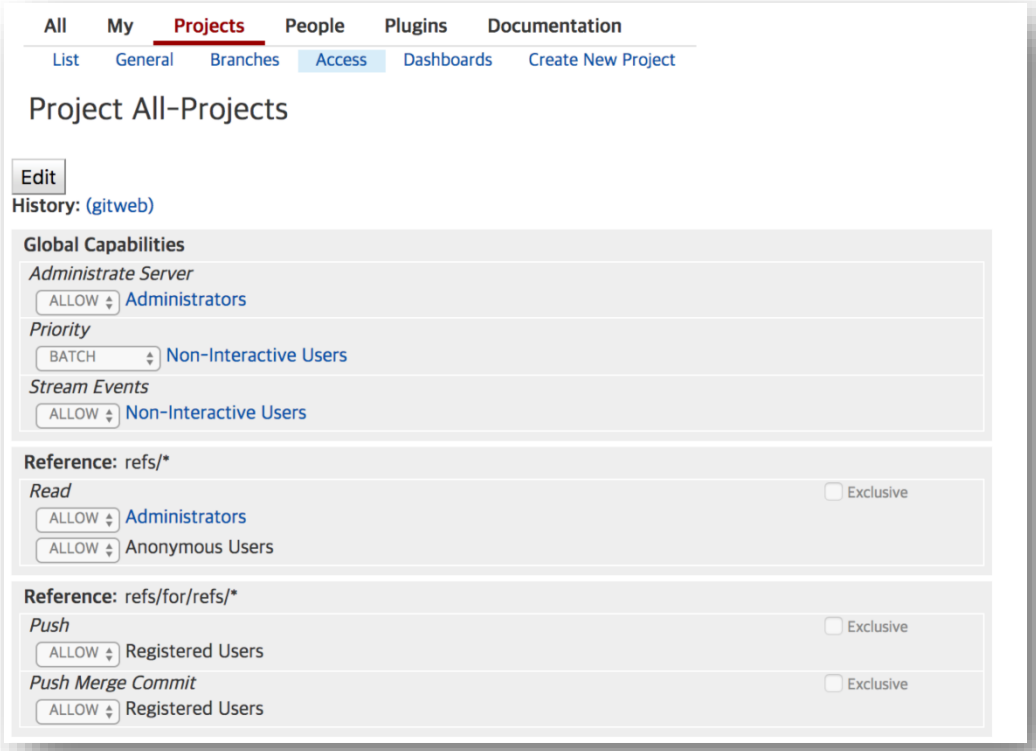


Gerrit 권한 구조의 이해



- ✓ 상속을 이용한 권한 구조 – 자식 프로젝트나 하위 그룹은 부모의 설정을 상속
- ✓ 그룹에 기반한 권한 – 개별 사용자는 역할을 가진 그룹을 통해 권한 부여
- ✓ 자원 – 주로 Git Reference인 권한 제어의 대상
- ✓ 제어 - Push, Merge 등 저장소에 대한 접근 권한과 코드 리뷰 권한을 포함

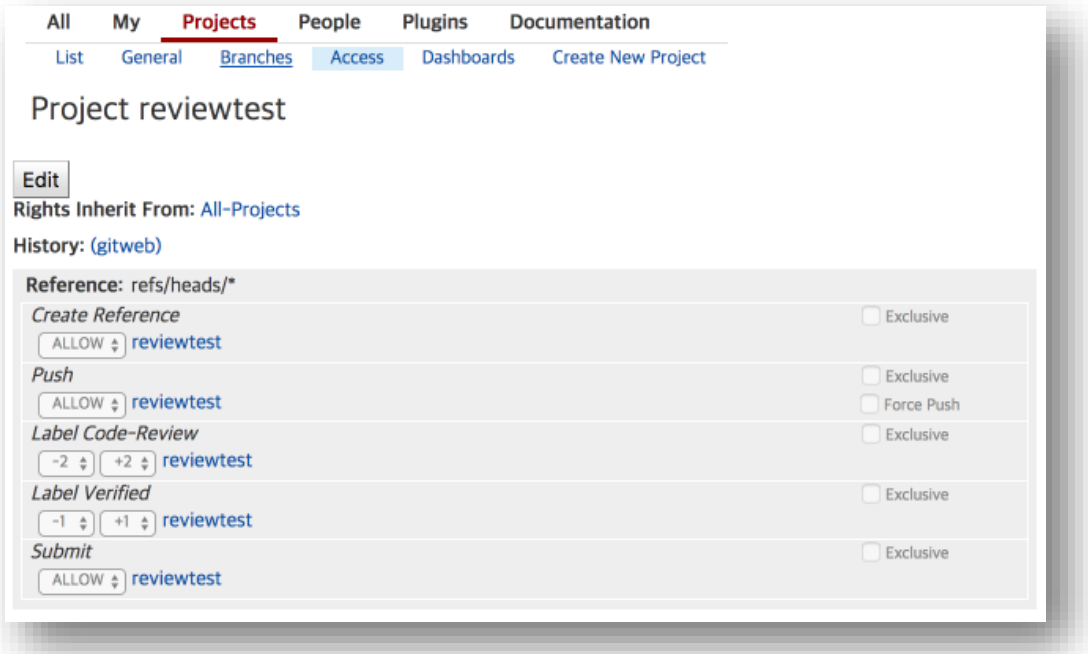
기본 권한을 설정하는 All-Projects



All-Projects 권한

- ✓ Gerrit이 초기화 될 때 자동으로 생성
- ✓ 실제 프로젝트가 아닌 Gerrit 내 모든 프로젝트들의 권한에 대한 기본 설정
- ✓ Global Capabilities – Gerrit 시스템에 대한 관리 기능(예: 사용자 생성)으로 여기서만 설정 가능
- ✓ Reference – Git Reference 단위의 권한 제어로 하위 프로젝트에서 오버라이드 가능

Project 별 상세 권한 설정



개별 Project 권한

- ✓ 상속 받은 상위 프로젝트의 권한을 기본적으로 사용
- ✓ 프로젝트 별로 권한을 오버라이드 가능
- ✓ 왼쪽의 예에서 reviewtest 프로젝트는 기본적으로 All-Projects의 권한을 상속하지만, 특별히 reviewtest 그룹(프로젝트가 아님에 주의)에게 목표 브랜치를 생성하거나 리뷰(-2 ~ 2사이 점수)를 하고 승인 결과를 제출할 수 있는 권한 부여

Git 저장소 관련 권한

Git 저장소에 대한 권한

Read	브랜치 복제 권한
Push(+Force)	존재하는 브랜치에 대한 Fast-Forward (또는 강제) 푸쉬 권한
Create Reference	새로운 브랜치를 생성할 수 있는 권한
Create Signed Tag Create Annotated Tag	태그를 생성할 수 있는 권한
Forge Author Identity Forge Committer Identity	변경의 저자 및 커미터 저자 정보를 수정할 수 있는 권한
Owner	refs/*에 적용하면 프로젝트에 대한 전체 관리 권한 부여

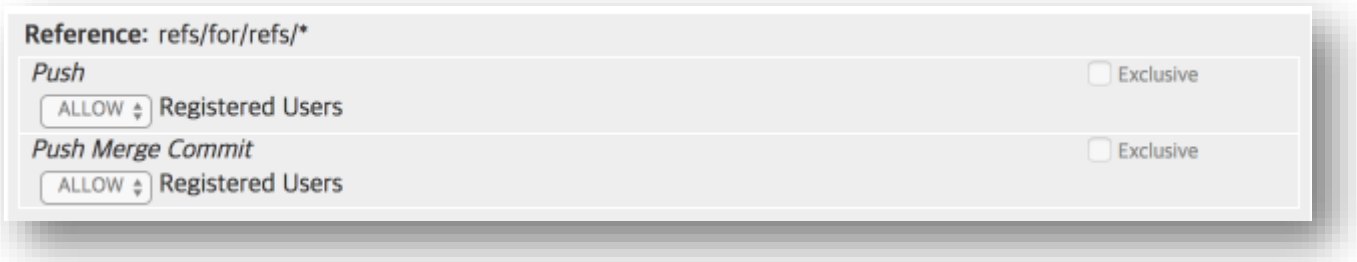
✓Project 권한 설정에서 위의 Git 저장소 관련 권한을 어떤 그룹에게 부여할지 지정 가능

코드 리뷰 권한

코드 리뷰 관련 권한 정의

Push	refs/for/refs/*	리뷰 요청 권한
Push Merge Commit	refs/for/refs/*	Merge Commit에 대한 리뷰 요청 권한

코드 리뷰 관련 권한 예



✓코드 리뷰를 위해서는 개발자가 리뷰어가 해당 Project의 관련 Reference에 적절한 권한이 있어야 함

✓위의 예에서 Gerrit 사용자는 누구나 Review를 요청할 권한이 있음

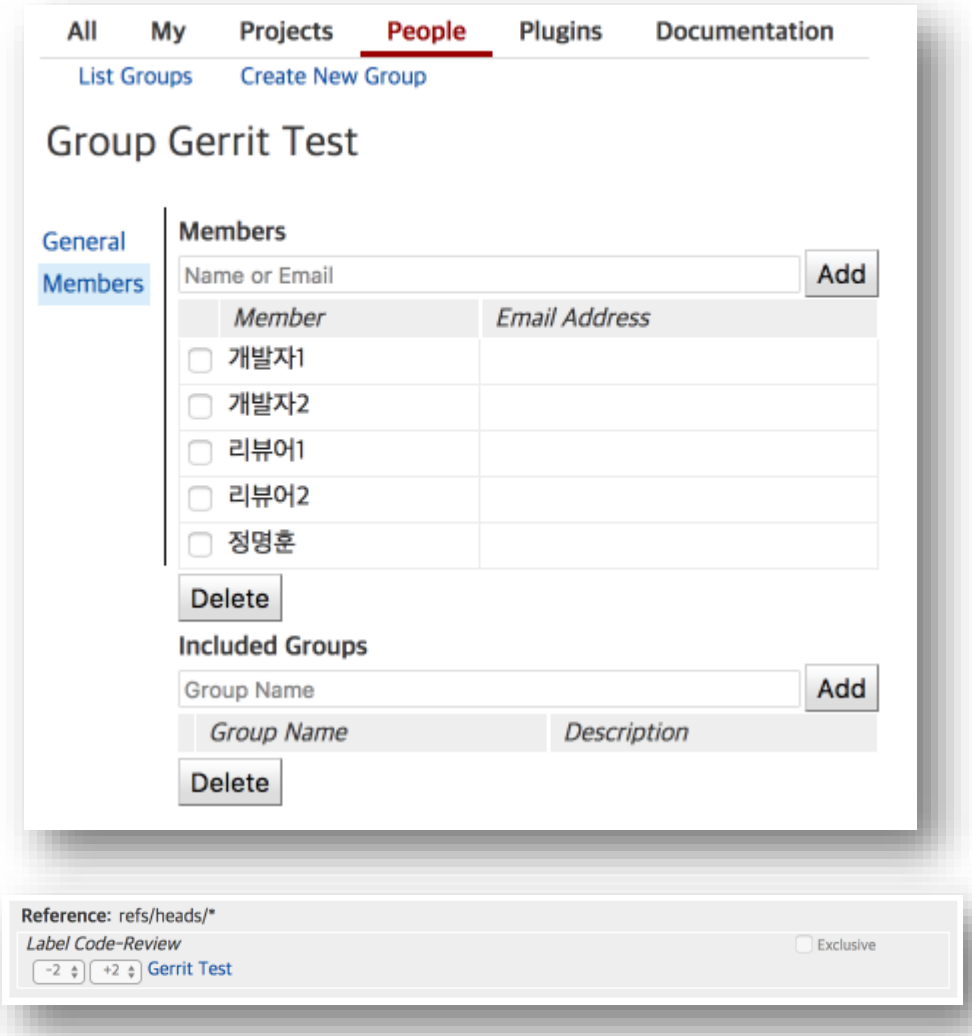
코드 리뷰 권한

코드 리뷰 관련 특수한 권한

Label-<LabelType>-X..+Y	refs/heads/*	진행 중인 리뷰에 대해 리뷰를 올리고 -X에서 +Y 사이의 점수를 부여할 권한 LabelType에는 Review와 Verify가 있음
Submit	refs/heads/*	리뷰 결과로 변경(Change)에 대한 중앙저장소 브랜치 병합 (Merge) 요청 권한 병합 요청은 바로 처리되어 병합이 이루어짐
Abandon	refs/heads/*	리뷰 결과로 변경에 대한 폐기 권한
Rebase	refs/heads/*	리뷰 중인 변경을 목표 브랜치로 리베이스(Rebase)할 수 있는 권한

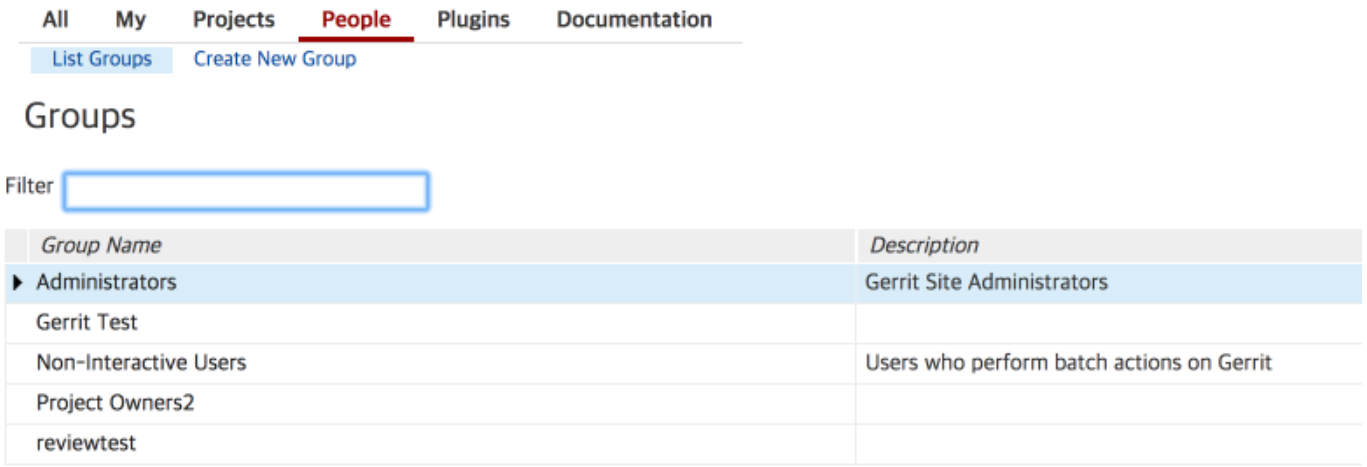
✓Project 권한 설정에서 위의 특수한 리뷰 관련 권한을 어떤 그룹에게 부여할지 지정 가능

사용자와 그룹



- ✓ 모든 Gerrit 사용자는 그룹에 속해야 해당 권한을 부여 받을 수 있음
- ✓ 그룹은 프로젝트의 권한 설정에서 역할을 부여 받음
- ✓ 그룹은 다른 그룹을 포함할 수 있음
- ✓ LDAP과 같은 외부 시스템의 그룹 정보를 가져와 재정의 해서 사용 가능
- ✓ 사용자별 권한 제어를 위해서는 특수한 플러그인(singleusergroup)을 통해서 가능

기본 그룹과 역할



✓ Administrators - Gerrit 관리자 (Gerrit 설치 후 최초 생성된 계정)

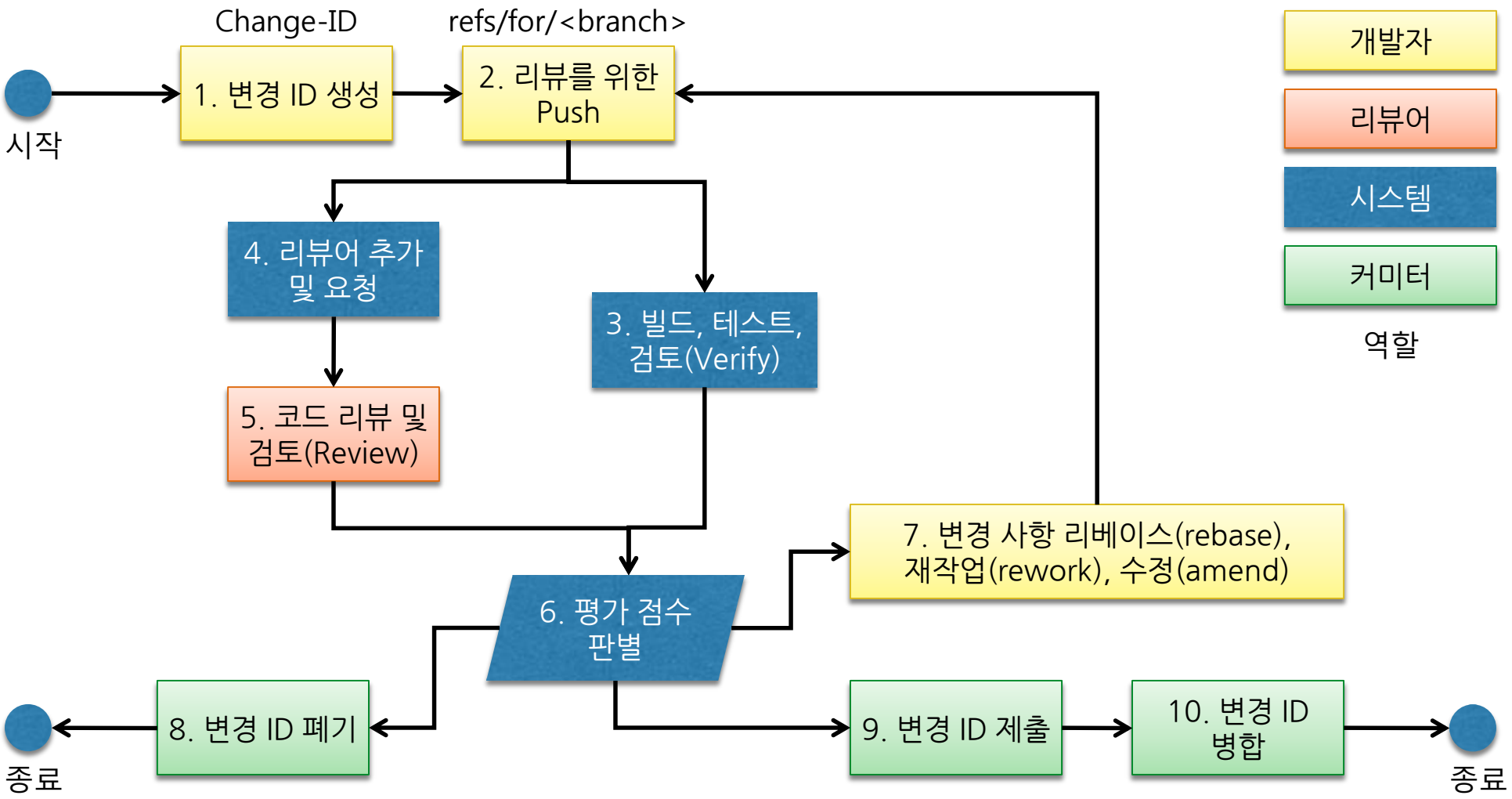
✓ Anonymouse users - 익명으로 접근 가능한 사용자 (여기에 권한을 부여했다는 의미는 공개 프로젝트임을 의미)

✓ Project Owners - 프로젝트의 소유자로 브랜치 생성 등 프로젝트에 대한 전반적 관리 권한 가짐

2.3 Gerrit 의 사용

실습

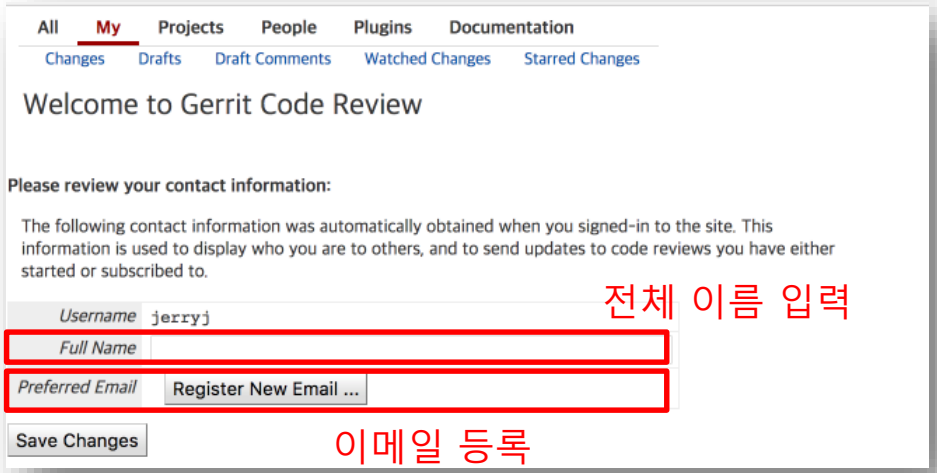
Gerrit 코드 리뷰의 작업 흐름

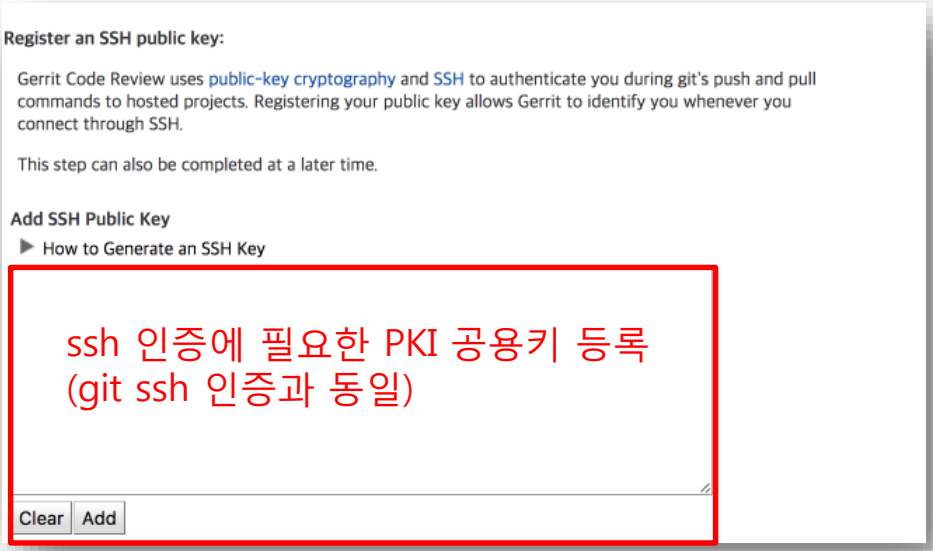


Gerrit 프로젝트 시작 – 사용자 생성

- Apache HTTPD에 계정 생성 및 로그인
- Gerrit 사용자 정보 입력
  - ✓ 최초 생성된 사용자는 자동으로 관리자 그룹에 등록 됨

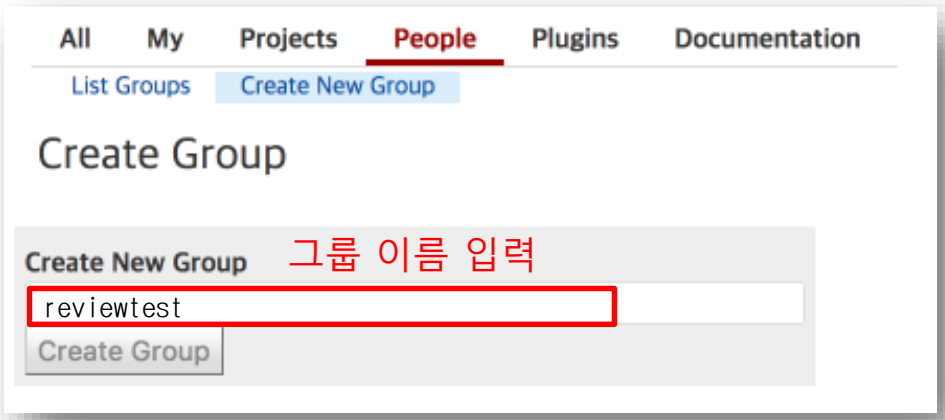
```
> sudo htpasswd -c /etc/apache2/passwords admin      (passwords 파일 새로 생성 시에는 -c 옵션 사용)
> sudo htpasswd /etc/apache2/passwords dev01
> sudo htpasswd /etc/apache2/passwords rev01
```

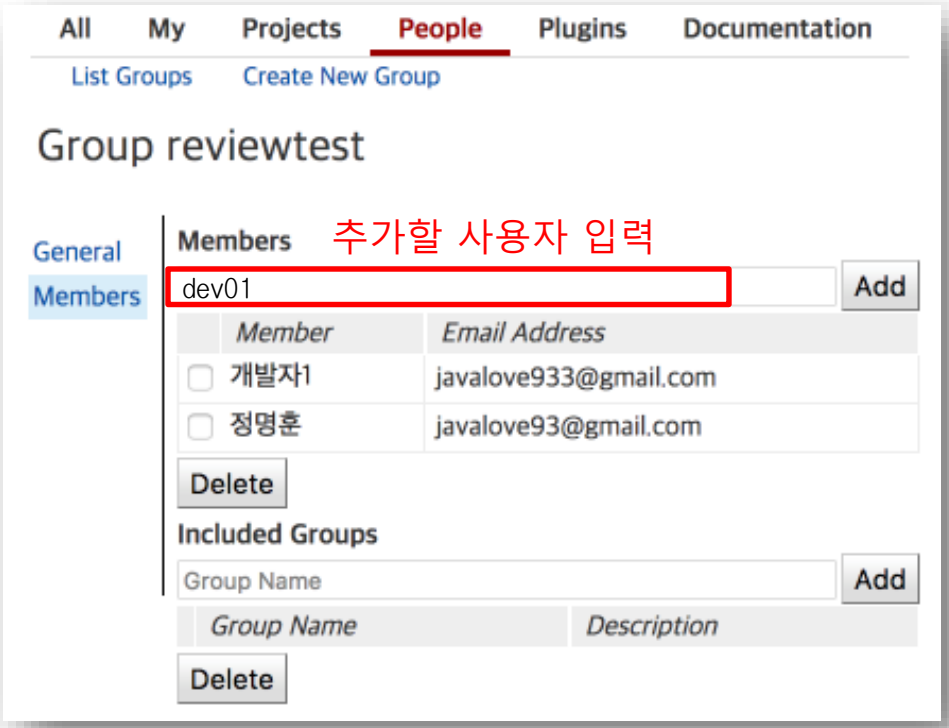




Gerrit 프로젝트 시작 – 그룹 지정

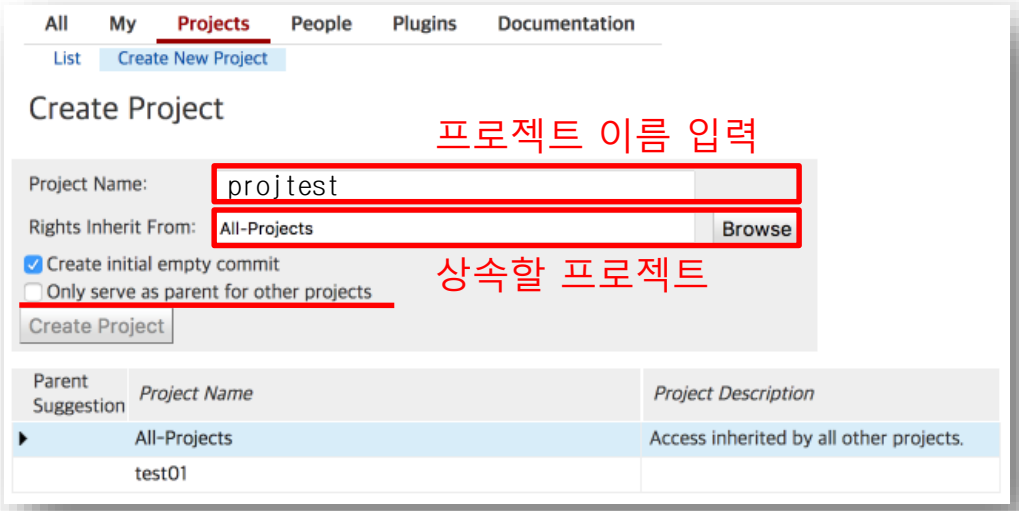
- Gerrit 그룹 생성
- 그룹에 사용자 추가
  - ✓ 개발자와 리뷰어 모두 추가

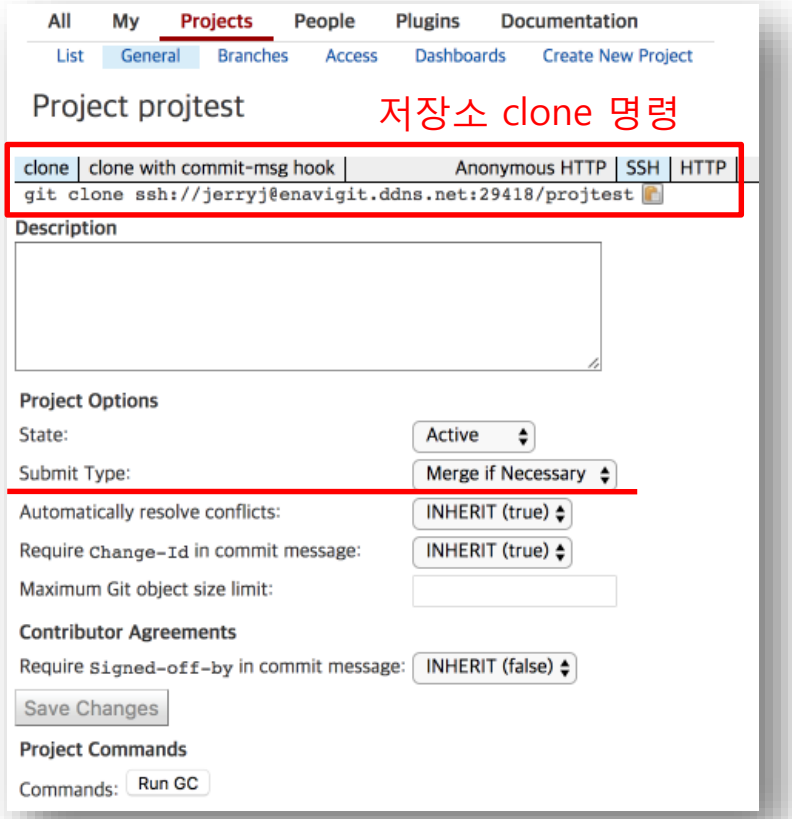




Gerrit 프로젝트 시작 – 프로젝트 생성

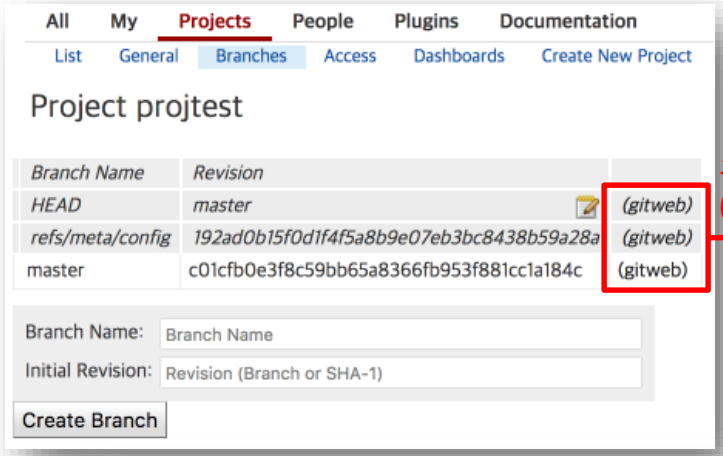
- Gerrit 프로젝트 생성
  - ✓ Only serve as parent for other projects: 권한만 있는 프로젝트
  - ✓ Submit Type: 변경을 저장소에 병합하는 방법



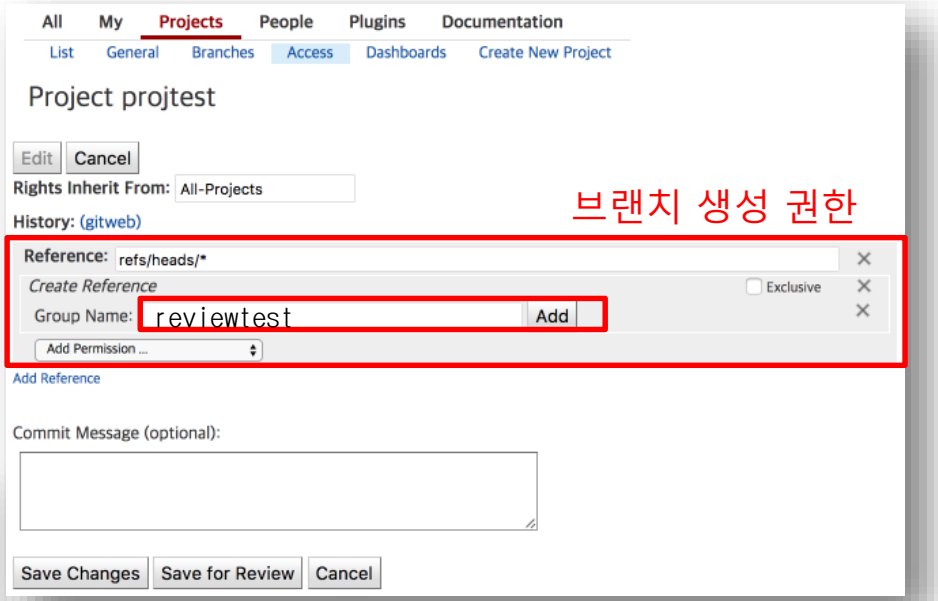


Gerrit 프로젝트 시작 – 프로젝트 구성

- 프로젝트 소스 및 브랜치 조회
- 프로젝트 권한 설정
  - ✓ 브랜치 생성 권한 부여: refs/heads/* 에 대한 Create Reference 권한을 그룹에 부여



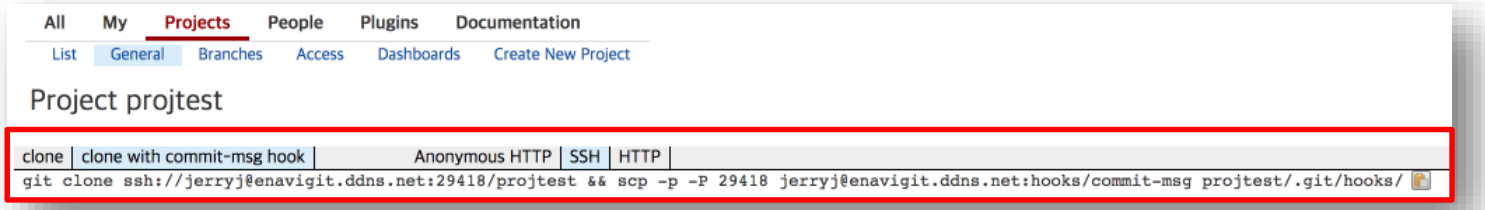
소스 조회 (gitweb)



브랜치 생성 권한

Gerrit 프로젝트 시작 – 저장소 clone

- 프로젝트 저장소 clone
  - ✓ 프로젝트 정보에 있는 git clone 명령 사용 (git config로 개발자 ID 지정)
  - ✓ Change-Id 자동 생성을 위한 커밋훅 스크립트도 같이 복사



저장소 clone 명령 with commit-msg hook | ssl

```
> git clone ssh://dev01@yourdomain.net:29418/projtest && scp -p -P 29418 dev01@yourdomain.net:hooks/commit-msg
projtest/.git/hooks/          (git 저장소 clone 및 commit-msg 커밋훅 스크립트 복사)

> git log
commit c01cfb0e3f8c59bb65a8366fb953f881cc1a184c
Author: 정명훈 <javalove93@gmail.com>
Date:   Fri May 1 13:15:59 2017 +0900

    Initial empty repository
```

1. 변경 ID 생성 (개발자)

- 브랜치 생성
  - ✓ 로컬에서만 브랜치 작업하여 병합하는 방법
  - ✓ 서버에 브랜치 생성하는 방법 (프로젝트에 대한 브랜치 생성 권한 필요)
- 소스 변경 작업

```
> git checkout -b mybranch            (로컬 브랜치 생성)
Switched to a new branch 'mybranch ‘

> ssh -p 29418 dev01@yourdomain.net gerrit create-branch projtest mybranch master          (master 브랜치를
기반으로 mybranch라는 브랜치 서버에 생성)

> vi branch.txt
  This is a file in new branch

> git add branch.txt
> git commit -a               (Change-Id는 commit-msg 스크립트에 의해 자동 부여됨)
  [mybranch b9d536f] mybranch file
  1 file changed, 1 insertion(+)
  create mode 100644 branch.txt
```

1. 변경 ID 생성 (개발자)

- 로컬 브랜치 병합
  - ✓ 로컬에서만 브랜치 작업하여 병합하는 방법
  - ✓ 로컬에만 브랜치 흔적이 남음

```
> git checkout master
Switched to branch 'master ‘

> git merge mybranch
Updating 2dc8723..b9d536f
Fast-forward
 branch.txt | 1 +
 1 file changed, 1 insertion(+)
 create mode 100644 branch.txt
```

2. 리뷰를 위한 Push (개발자)

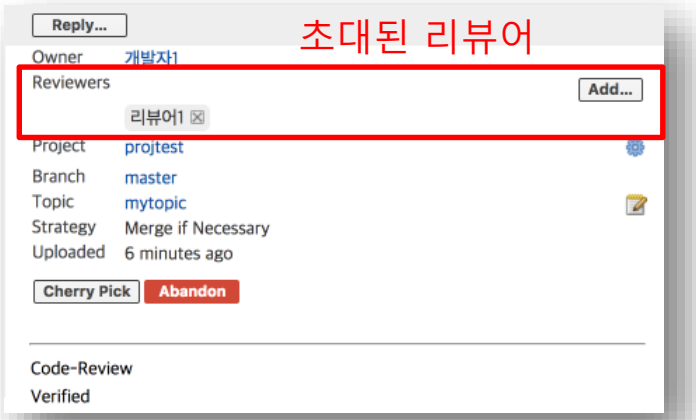
- 리뷰 전용 브랜치로 Push
  - ✓ 프로젝트 정보에 있는 git clone 명령 사용 (git config로 개발자 ID 지정)
  - ✓ Change-Id 자동 생성을 위한 커밋훅 스크립트도 같이 복사

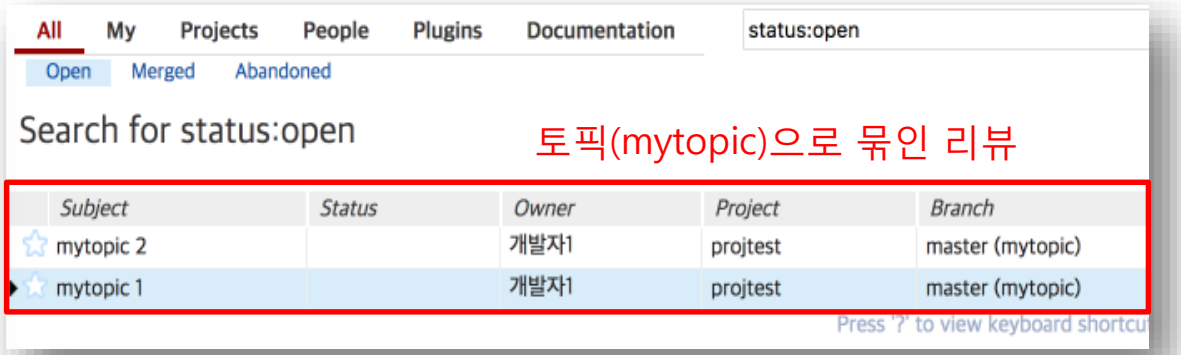
```
> git push origin HEAD:refs/for/mybranch          (서버 브랜치를 사용하지 않는 경우에는 refs/for/master)
Counting objects: 4, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 342 bytes | 0 bytes/s, done.
Total 3 (delta 0), reused 0 (delta 0)
remote: Processing changes: new: 1, refs: 1, done
remote:
remote: New Changes:
remote:   http://yourdomain.net/7
remote:
To ssh://yourdomain.net:29418/projtest
 * [new branch]      HEAD -> refs/for/mybranch
```

2. 리뷰를 위한 Push (개발자)

- Push 과정에 리뷰자 지정
- Push 과정에 토픽 지정
  - ✓ 관련된 변경들을 묶어서 하나의 리뷰로 지정

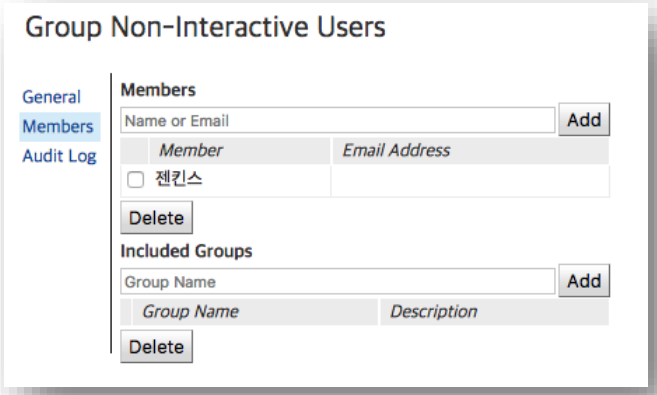
```
> git push origin HEAD:refs/for/mybranch%r=rev01@yourmail.net          (rev01 리뷰어로 초대)

> git push origin HEAD:refs/for/mybranch%topic=mytopic                 (mytopic 토픽 지정)

> git push origin HEAD:refs/for/mybranch%topic=mytopic,r=rev01@yourmail.net          (토픽과 리뷰어 동시 지정)
```



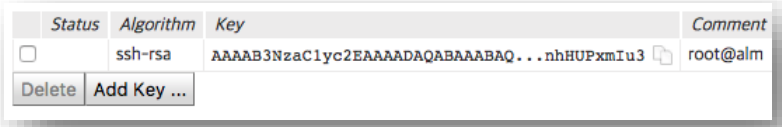


3. 빌드, 테스트, 검토 (시스템)

- CI 도구(Jenkins, Quick Build)를 이용하여 자동 빌드/테스트 및 결과를 Gerrit으로 제출
- Jenkins 설정
  - ✓ Gerrit 계정 생성 및 SSH Key 등록
  - ✓ 계정을 Non-Interactive Users 그룹에 등록하고 Project 권한 부여

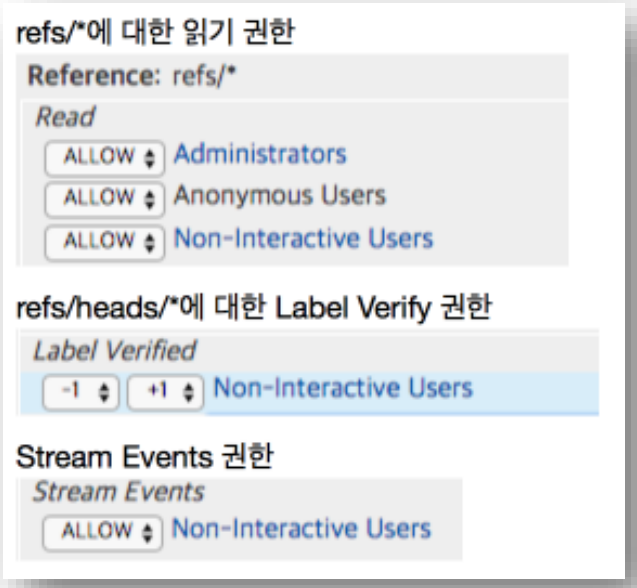


Gerrit에 Jenkins용 계정 생성



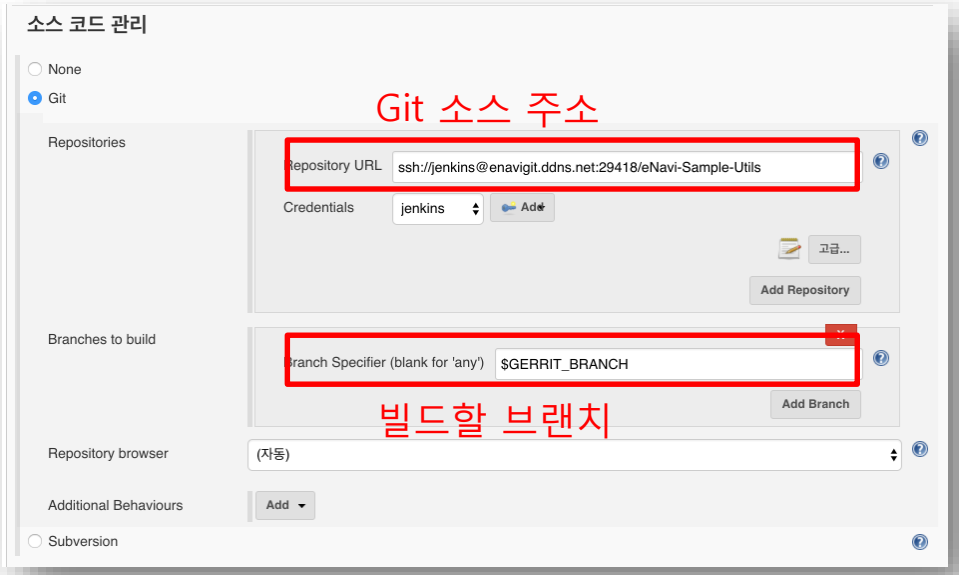
Gerrit의 Jenkins 계정에 Jenkins 시스템의 SSH Key 등록

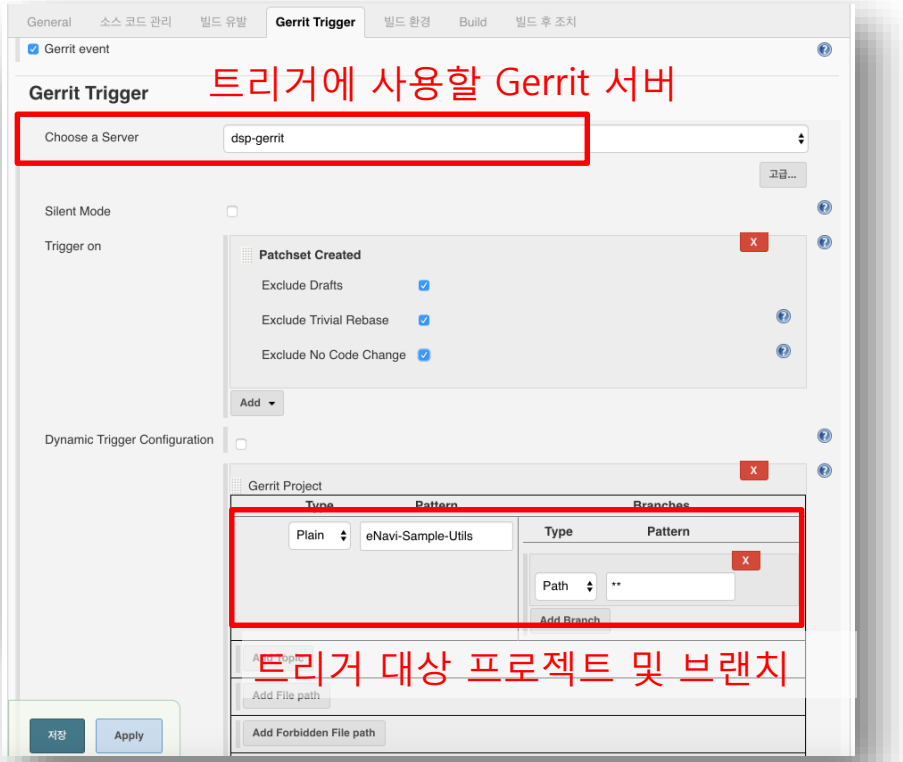
Project에 대한 Verify 관련 권한 부여



3. 빌드, 테스트, 검토 (시스템)

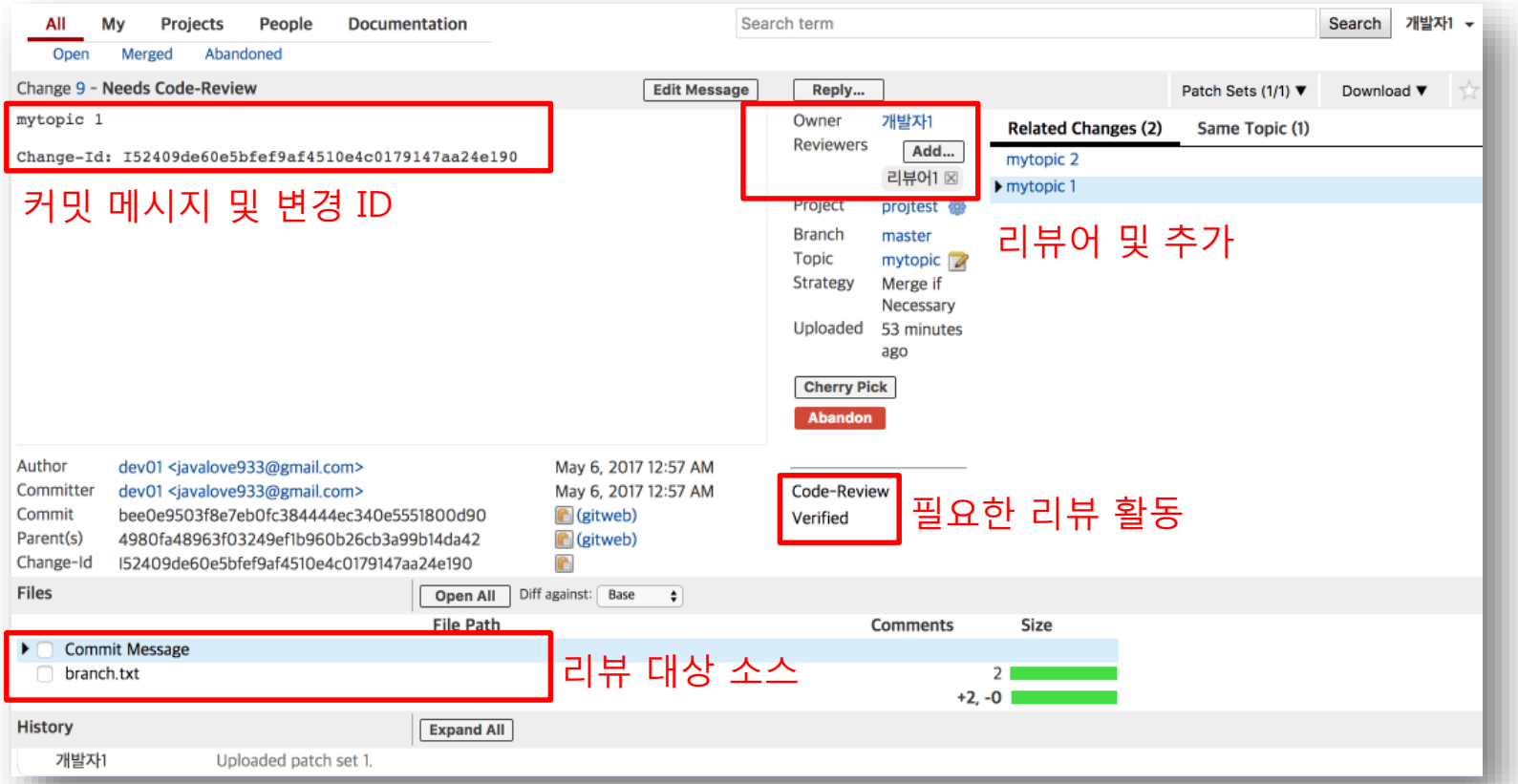
- Jenkins 설정
  - ✓ Gerrit Git 저장소 설정 - 빌드를 위한 소스 Fetch
  - ✓ Gerrit 트리거 설정 - 리뷰 제출 시 자동으로 빌드/테스트 동작





4. 리뷰어 추가 및 요청 (시스템)

- 웹 UI에서 생성된 리뷰 조회
- 리뷰어 추가(초대)

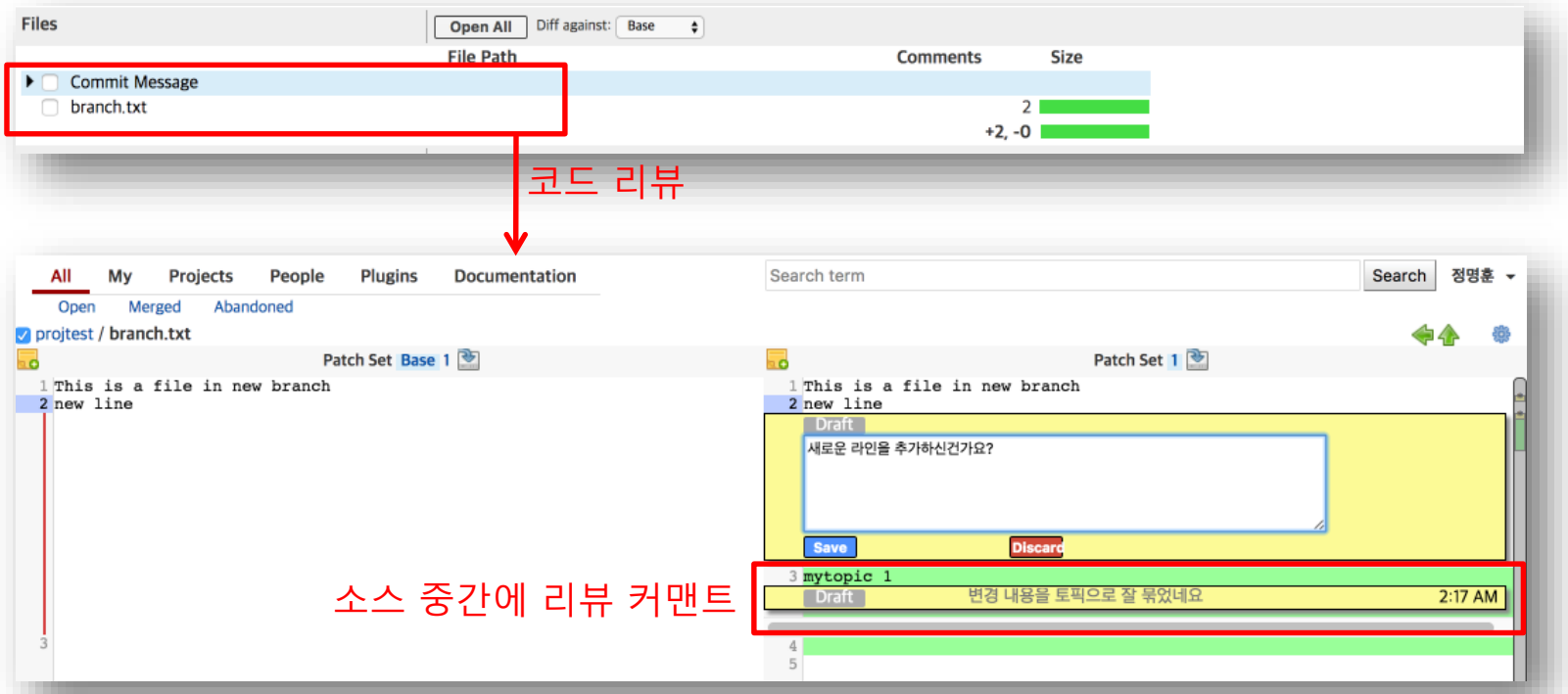


TwoSeed

SAMSUNG

5. 코드 리뷰 및 검토 (리뷰어)

- 웹 UI에서 변경된 코드에 대한 리뷰 및 검토 활동
  - ✓ 기존 소스와 변경된 소스를 비교할 수 있는 리뷰 도구를 활용한 코드 리뷰
  - ✓ 소스 중간에 인라인 의견(커맨트) 추가



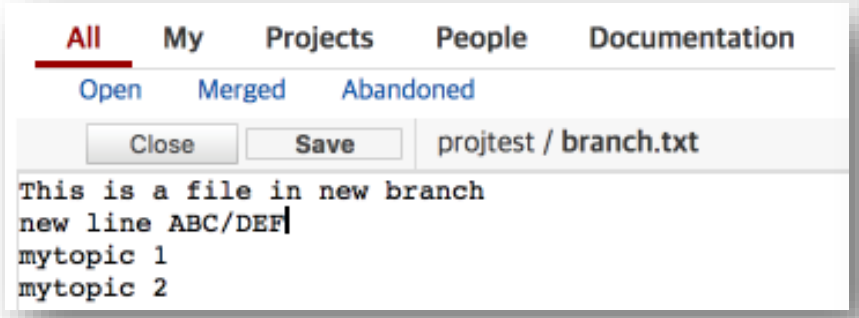
Side-by-Sde 코드 리뷰 도구

5. 코드 리뷰 및 검토 (리뷰어)

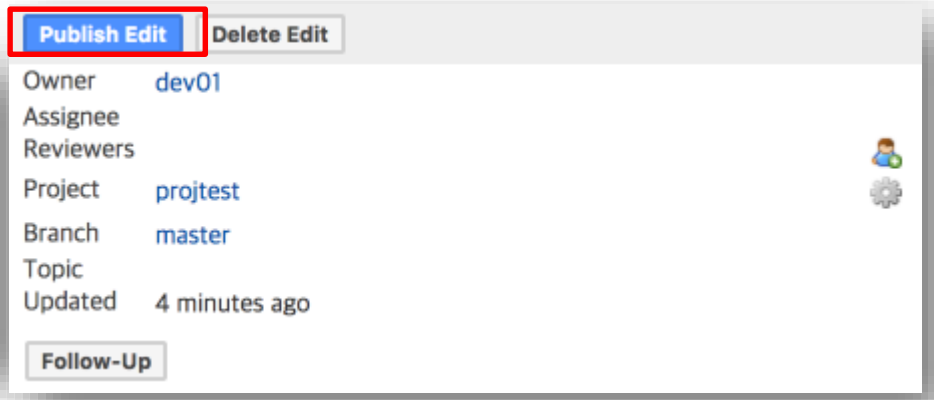
- 웹 UI에서 소스 인라인 편집
  - ✓ 리뷰 도중 UI 내에서 바로 코드를 편집하여 새로운 패치셋 생성

소스 편집 모드

인라인 소스 편집



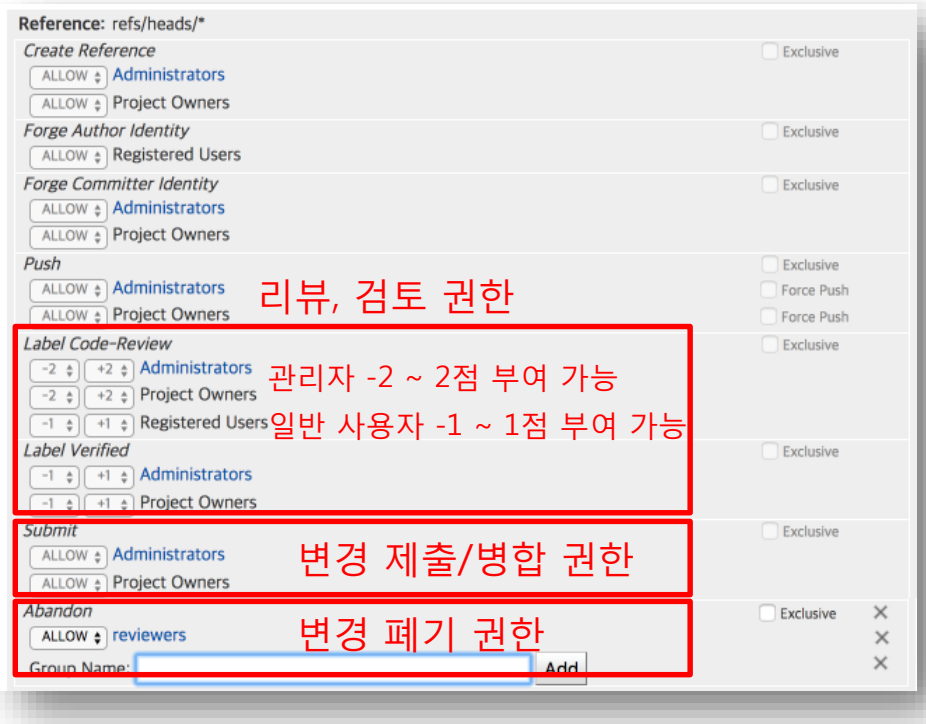
편집 내용 반영



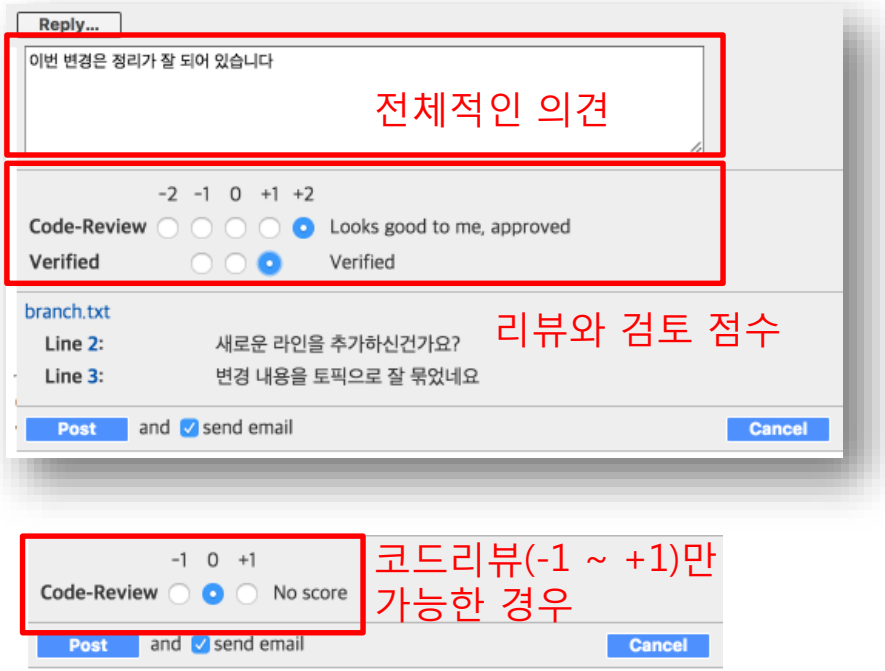
5. 코드 리뷰 및 검토 (리뷰어)

- 리뷰 피드백 제출
  - ✓ 부여된 리뷰 권한(프로젝트 권한에서 설정)에 따른 피드백 제공
  - ✓ 개발자(리뷰 요청)에게 전체적인 의견 제공 가능

프로젝트 권한 설정



리뷰 피드백



5. 코드 리뷰 및 검토 (리뷰어)

리뷰 예절 및 프랙티스

- ✓ 리뷰 및 토론 내용 공유: 모든 의견은 Gerrit을 통해 변경 내용과 같이 관리 및 공유 함
- ✓ 모든 파일을 리뷰: 다른 사람의 코드를 완전히 이해하기 위해 변경된 모든 파일을 리뷰
- ✓ 코드에 대한 의견: 리뷰는 다른 사람의 수준을 평가하는 것이 아닌 코드 개선 목적을 위해 코드에 대한 의견만을 제공하는 것임
- ✓ 모든 의견에 대답: 모든 의견과 질문에 대해 반영 여부와 함께 답변을 제공
- ✓ 소스 코드로 의견 제시: 가능하면 변경에 대한 대안이 될 수 있는 구체적인 코드를 제시
- ✓ 한 번에 한 가지씩 변경: 리뷰의 근본적인 목적은 변경을 반영하는 것으로, 리뷰에 너무 많은 시간이 들어가지 않도록 리뷰 요청하는 변경 단위를 최소화
- ✓ 토픽 사용: 큰 변경은 작게 나누어서 토픽으로 구분

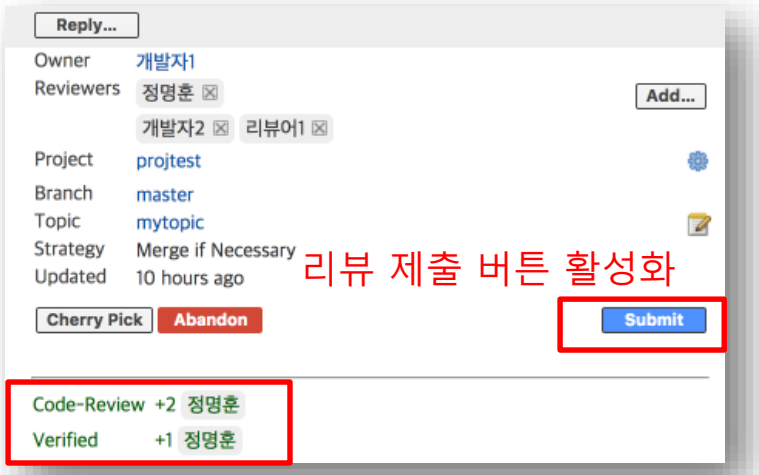
5. 코드 리뷰 및 검토 (리뷰어)

코드 리뷰 용어

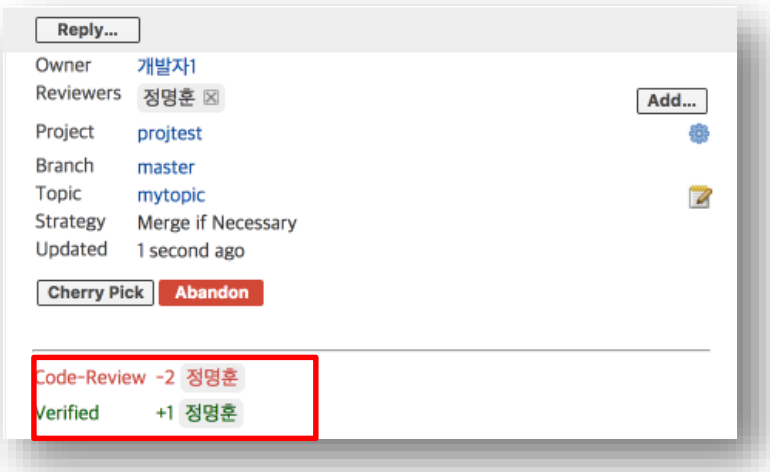
- ✓ NIT(picking): 작은 변경을 필요로 하는 사소한 문제
- ✓ Optional: 리뷰자의 개인 취향으로 원저자가 무시할 수 있음
- ✓ RFC(Request For Comments): 반드시 병합될 필요 없이 아이디어를 공유하기 위한 리뷰
- ✓ WIP(Work In Progress): 초안 성격의 변경으로 리뷰를 통해 사전 피드백을 수집하기 위함

6. 평가 점수 판별 (시스템)

- 시스템을 통한 평가 점수 판별
  - ✓ 저장소에 커미터 권한을 가진 담당자가 +2점을 부여할 경우 리뷰는 통과된 것으로 봄
  - ✓ 커미터 권한을 가진 담당자가 -2점을 부여할 경우에 리뷰는 거부된 것으로 봄
  - ✓ 대체로 +1점 두 개가 모인다 해서 +2점으로 간주하지는 않음
  - ✓ 검토 실패 시 -1점을 부여하고 저장소에 병합되어서는 안됨
  - ✓ 조직 내부에서 규칙을 정해야 함



리뷰와 검토 점수 통과
요건(각 +2, +1) 만족

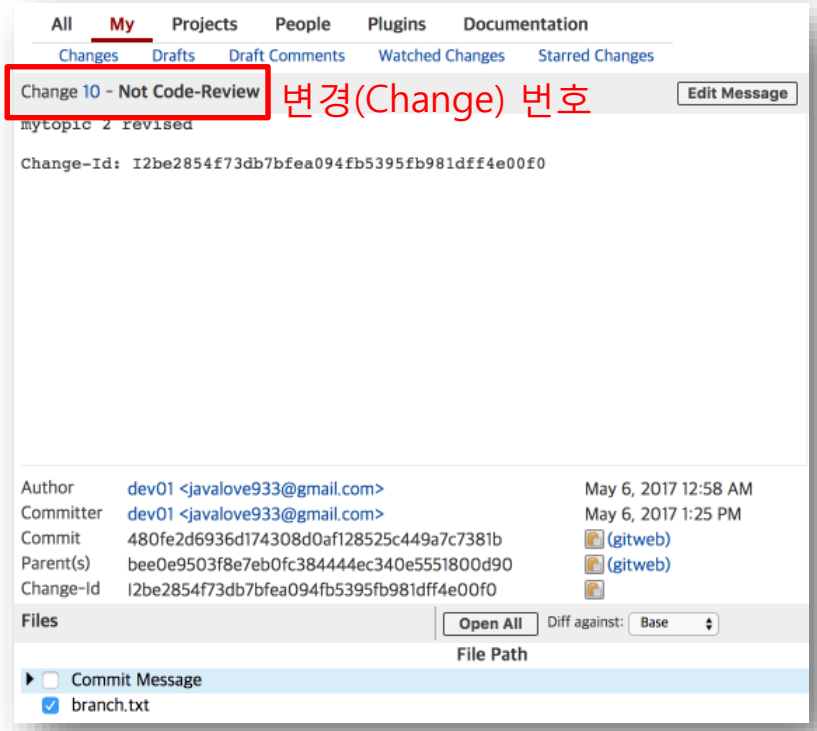


검토는 통과했으나 리뷰
통과 실패

7. 변경 사항 리베이스, 재작업, 수정 (개발자)

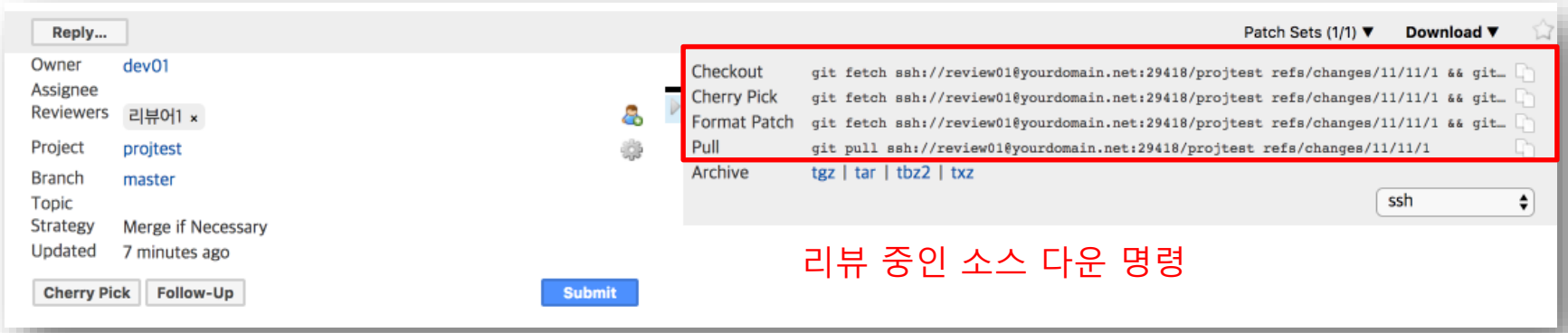
- 부정적인 점수를 받은 리뷰에 대한 재 작업
- git-review를 이용한 리뷰 중인 소스 다운로드
  - ✓ 리뷰 중인 변경(change)과 패치셋 기준으로 다운로드하여 로컬에 브랜치 생성

```
> sudo apt-get install git-review
> vi .gitreview                    (프로젝트 작업 위치에 생성)
  [gerrit]
  host=yourdomain.net
  port=29418
  project=projtest.git
  defaultbranch=master

> git review -l                    (진행 중인 리뷰 목록 조회)
10  master  mytopic 2
Found 1 items for review

> git review -d 10      (Change 10번 소스 다운로드하여 로컬에 브랜치 생성)
Downloading refs/changes/10/10/1 from gerrit
Switched to branch "review/_1/mytopic"
```



7. 변경 사항 리베이스, 재작업, 수정 (개발자)

- git을 fetch를 이용한 리뷰 중인 소스 다운로드
  - ✓ 리뷰 중인 변경(change)과 패치셋 기준으로 다운로드하여 로컬에 브랜치 생성

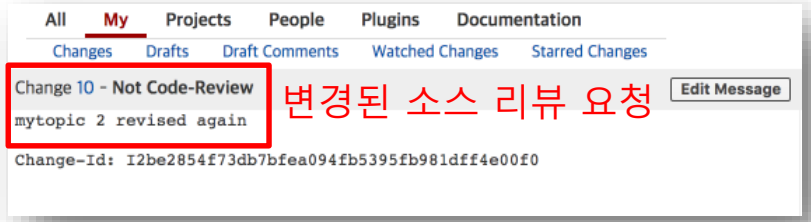


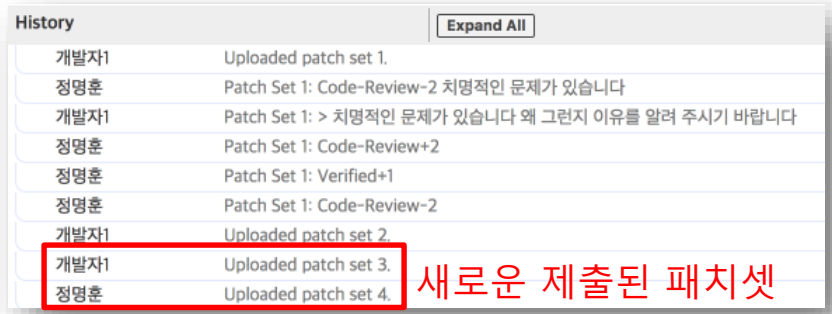

```
> git fetch ssh://dev01@yourdomain.net:29418/projtest refs/changes/12/12/1 && git checkout FETCH_HEAD
From ssh://yourdomain.net:29418/projtest
 * branch            refs/changes/12/12/1 -> FETCH_HEAD
Note: checking out 'FETCH_HEAD'.

> git branch
* (detached from FETCH_HEAD)
  master
```

7. 변경 사항 리베이스, 재작업, 수정 (개발자)

- 리뷰에 따라 소스 수정
- 동일한 Change-Id로 커밋
- 다시 리뷰 제출 (git push 또는 git review 명령 사용)

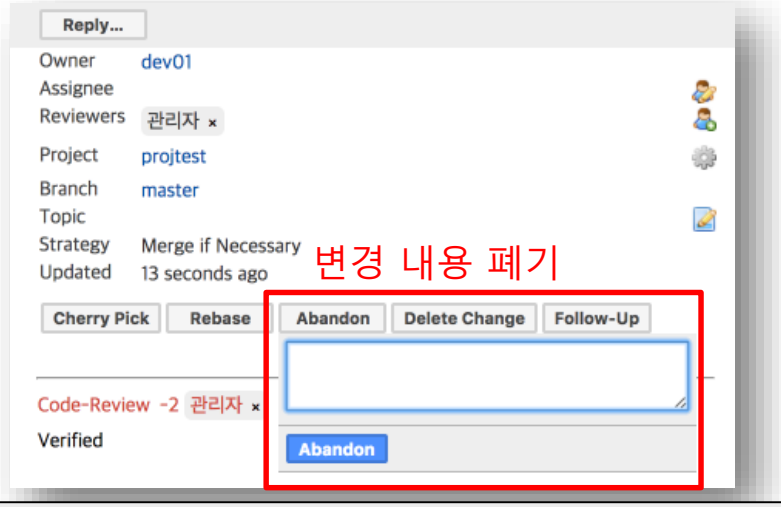
```
> vi branch.txt                    (소스 수정 작업)
> git add branch.txt

> git commit --amend               (동일 Change-Id 유지하면서 커밋)
[review/_1/mytopic 48815e8] mytopic 2 revised
 1 file changed, 1 insertion(+), 1 deletion(-)
 (-m option을 사용 시 Change ID가 변경 되므로 유의)

> git push origin HEAD:refs/for/master          (리뷰 제출)

> git review -f          (git review 이용한 리뷰 제출 및 브랜치 삭제)
remote: Processing changes: updated: 1, refs: 1, done
........
* [new branch]      HEAD -> refs/publish/master/mytopic
Switched to branch 'master'
Deleted branch 'review/_1/mytopic'
```

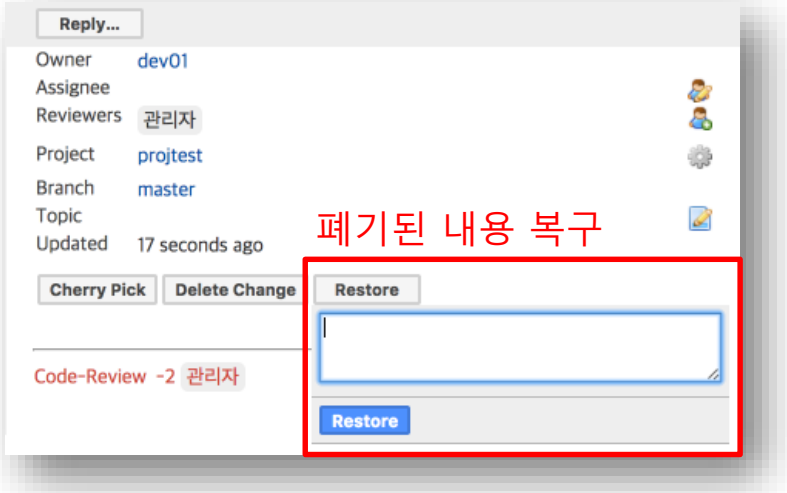




8. 변경 ID 폐기 (커미터)

- 변경에 대한 재작업을 했음에도 리뷰 평가 점수가 기준에 미달할 경우 변경 내용을 폐기
- 폐기된 내용은 목록(Abandoned)에 남아 있으므로 다시 복구 가능
- 개발자는 로컬 저장소의 커밋 내용을 취소

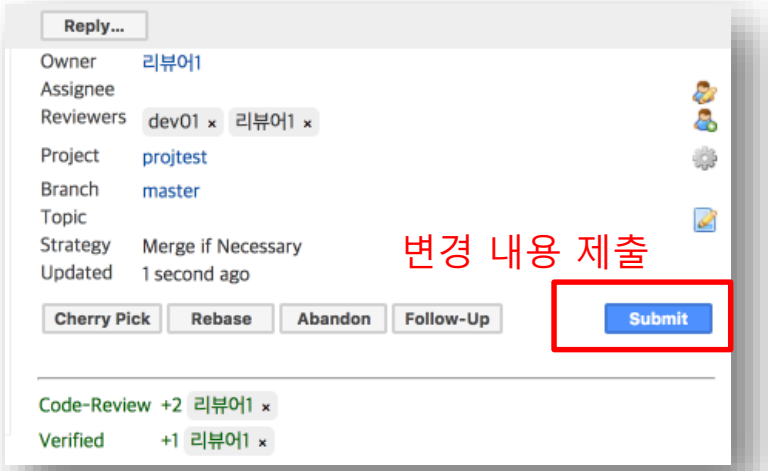


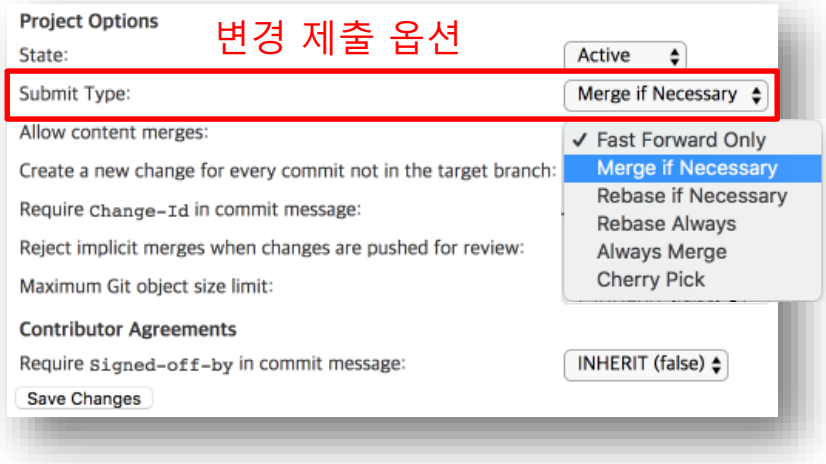



```
> git status
# On branch master
# Your branch is ahead of 'origin/master' by 1 commit.
#   (use "git push" to publish your local commits)

> git reset --hard HEAD^
```

9. 변경 ID 제출 (커미터)

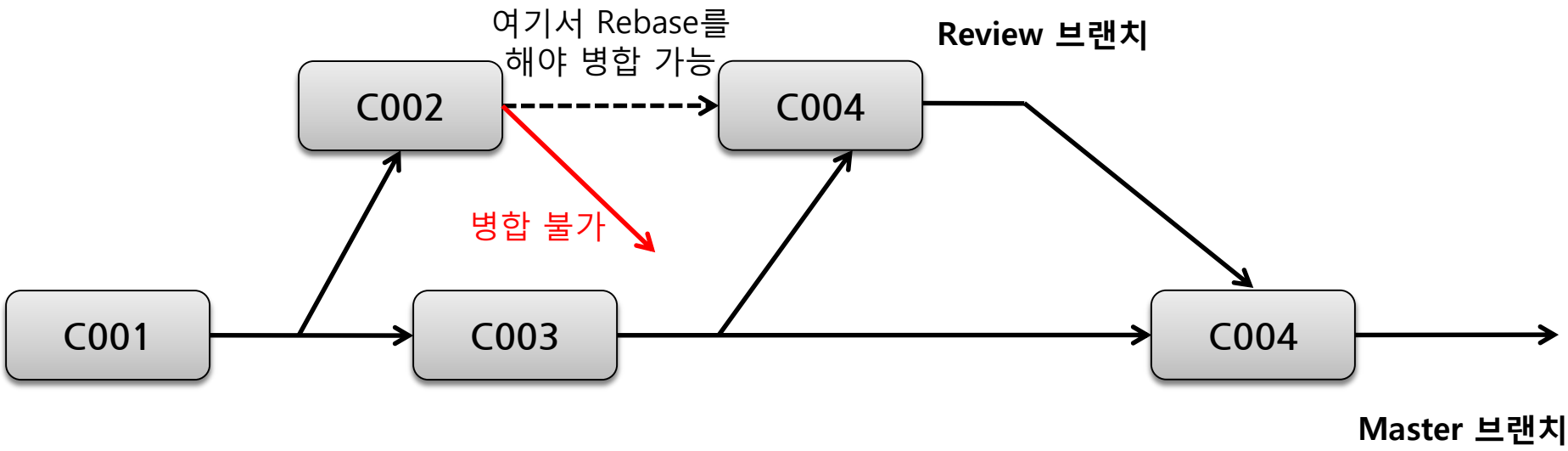
- 변경 내용에 평가 점수가 기준 이상일 경우 변경 내용을 제출
- 프로젝트의 변경 제출 옵션에 따라 중앙저장소에 병합 됨
  - ✓ Fast Forward Only
  - ✓ Merge If Necessary
  - ✓ Rebase If Necessary
  - ✓ Always Merge
  - ✓ Cherry Pick





10. 변경 ID 병합 (커미터)

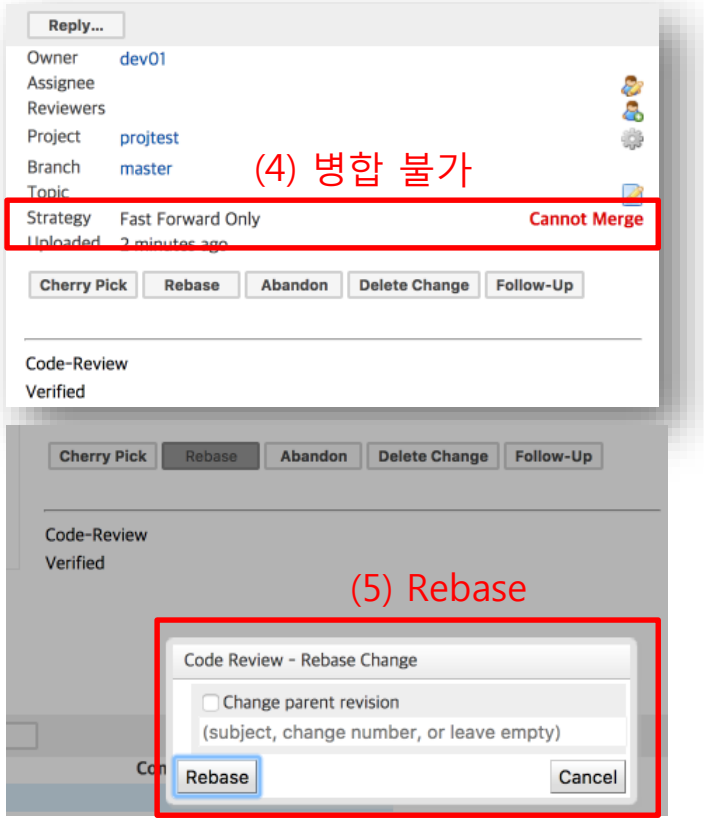
- Fast Forward Only 방식의 병합
  - ✓ 해당 커밋이 최신 HEAD보다 더 최신일 때만 병합을 허용
  - ✓ 리뷰 중에 다른 커밋이 있을 경우 코드를 병합하기 위해서는 Rebase를 해야만 함



10. 변경 ID 병합 (커미터)

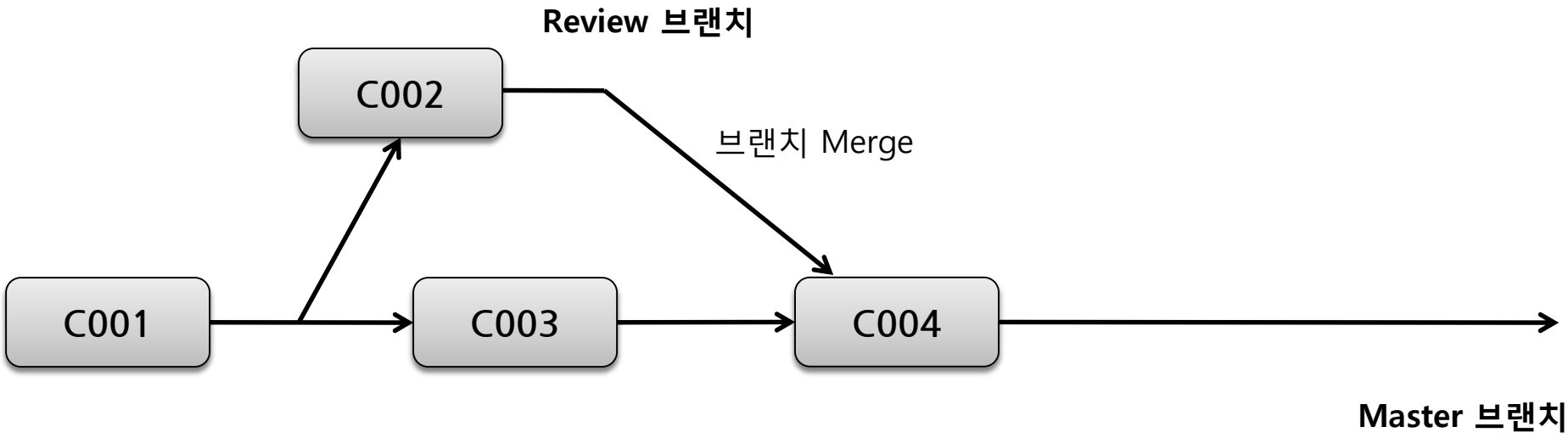
- Fast Forward Only 방식의 병합
  - ✓ 해당 커밋이 최신 HEAD보다 더 최신일 때만 병합을 허용
  - ✓ 리뷰 중에 다른 커밋이 있을 경우 코드를 병합하기 위해서는 Rebase를 해야만 함

```
(1) 개발자1, C002 변경 및 리뷰 제출
> vi branch.txt
> git commit -a
> git review

(2) 개발자2, C003 변경 및 리뷰 제출
> vi dev02.txt
> git commit -a
> git review

(3) 리뷰어, C003 리뷰 및 병합
(4) C002 병합 불가능
(5) 리뷰어, C002를 rebase      (C004로 이름 변경)
(6) 리뷰어, C004 리뷰 및 병합
(7) 개발자2, git pull
(8) 개발자1, git fetch & git rebase
```



10. 변경 ID 병합 (커미터)

- Merge If Necessary, Always Merge
  - ✓ 해당 커밋이 최신 HEAD보다 더 최신일 때는 Fast Forward 병합 수행
  - ✓ 그렇지 않은 경우에는 리뷰 브랜치를 포함한 Merge 시도 (Conflict Resolution 없다면)



10. 변경 ID 병합 (커미터)

- Merge If Necessary, Always Merge
  - ✓ 해당 커밋이 최신 HEAD보다 더 최신일 때는 Fast Forward 병합 수행
  - ✓ 그렇지 않은 경우에는 리뷰 브랜치를 포함한 Merge 시도 (Conflict Resolution 없다면)

```
(1) 개발자1, C002 변경 및 리뷰 제출
> vi branch.txt
> git commit -a
> git review

(2) 개발자2, C003 변경 및 리뷰 제출
> vi dev02.txt                (다른 파일 변경)
> git commit -a
> git review

(3) 리뷰어, C003 리뷰 및 병합
(4) 리뷰어, C004 리뷰 및 병합 (Conflict 없음)
(5) 개발자2, git pull
(6) 개발자1, git pull
```

10. 변경 ID 병합 (커미터)

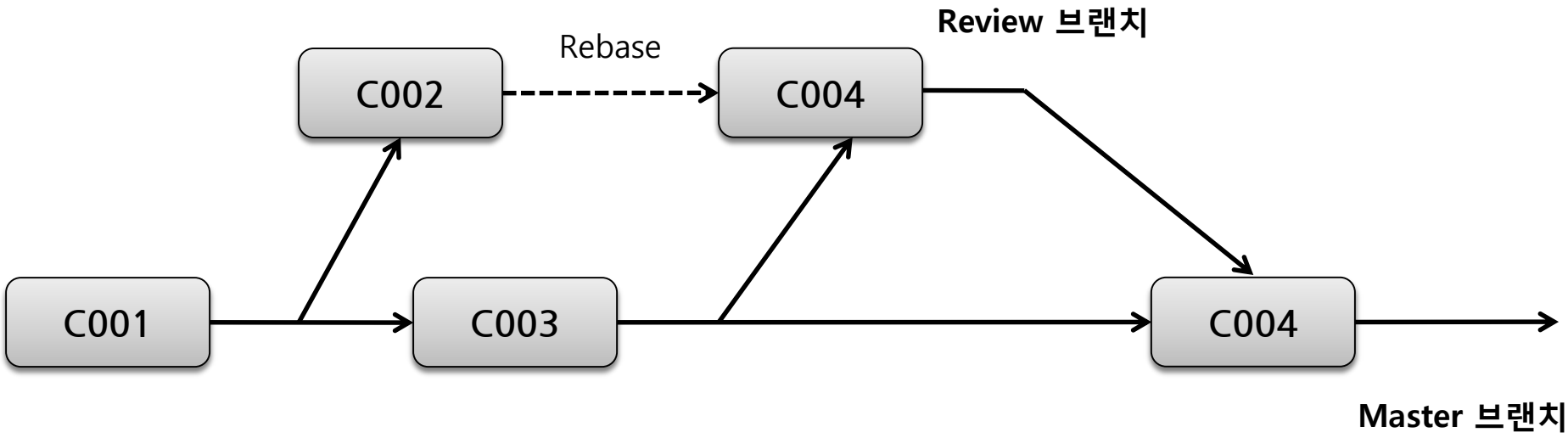
- Merge If Necessary, Always Merge
  - ✓ 해당 커밋이 최신 HEAD보다 더 최신일 때는 Fast Forward 병합 수행
  - ✓ 그렇지 않은 경우에는 리뷰 브랜치를 포함한 Merge 시도 (Conflict Resolution 없다면)

```
(1) 개발자1, C002 변경 및 리뷰 제출
> vi branch.txt
> git commit -a
> git review

(2) 개발자2, C003 변경 및 리뷰 제출
> vi branch.txt                (동일한 파일 변경)
> git commit -a
> git review

(3) 리뷰어, C003 리뷰 및 병합
(4) C002 병합 불가능           (Conflict 발생)
```

```
(5) 리뷰어, 리뷰 중인 변경 git fetch
(6) 리뷰어, git rebase 및 merge
> git rebase master
> vi branch.txt                (Conflict 해결)
> git add branch.txt
> git rebase --continue
(7) 리뷰어, git review 리뷰 제출
(8) 리뷰어, C004 리뷰 및 병합
(9) 개발자2, git pull
(10) 개발자1, git fetch & git rebase
```

10. 변경 ID 병합 (커미터)

- Rebase If Necessary, Rebase Always 방식의 병합
  - ✓ 해당 커밋이 최신 HEAD보다 더 최신일 때는 Fast Forward 병합 수행
  - ✓ 그렇지 않은 경우에는 Rebase 시도 (Conflict Resolution 없다면)



10. 변경 ID 병합 (커미터)

- Rebase If Necessary, Rebase Always 방식의 병합
  - ✓ 해당 커밋이 최신 HEAD보다 더 최신일 때는 Fast Forward 병합 수행
  - ✓ 그렇지 않은 경우에는 Rebase 시도 (Conflict Resolution 없다면)

```
(1) 개발자1, C002 변경 및 리뷰 제출
> vi branch.txt
> git commit -a
> git review

(2) 개발자2, C003 변경 및 리뷰 제출
> vi dev02.txt                (다른 파일 변경)
> git commit -a
> git review

(3) 리뷰어, C003 리뷰 및 병합
(4) 리뷰어, C004 리뷰 및 병합 (Conflict 없음)
(5) 개발자2, git pull
(6) 개발자1, git pull
```

END OF DOCUMENT.

http://www.twoseed.co.kr