**Git Starting**

Version 1.0

2016.09.05

작성자 : 문해민

**Table of Contents**

[**1.** **개요** - 3 -](#_Toc461198022)

[**1.1.** **버전 관리 시스템 [ VCS ] ?** - 3 -](#_Toc461198023)

[**1.2.** **Git** - 4 -](#_Toc461198024)

[**1.2.1.** **Git이란?** - 4 -](#_Toc461198025)

[**1.2.2.** **Git 기초 개념** - 5 -](#_Toc461198026)

[**2.** **작업 환경 구축 및 절차** - 6 -](#_Toc461198027)

[**2.1.** **환경 설정 및 설치** - 6 -](#_Toc461198028)

[**2.1.1.** **Git 설치 및 초기설정** - 6 -](#_Toc461198029)

[**2.1.2.** **GitHub 기본 설정** - 8 -](#_Toc461198030)

[**3.** **GitHub / Git 기초 사용법** - 10 -](#_Toc461198031)

[**3.1.** **Git 저장소 생성** - 10 -](#_Toc461198032)

[**3.2.** **Git 저장소 복제** - 12 -](#_Toc461198033)

[**3.3.** **파일 수정과 상태 변경** - 13 -](#_Toc461198034)

[**3.4.** **원격 저장소(리모트 저장소) 활용 튜토리얼** - 15 -](#_Toc461198035)

[**3.4.1.** **Commit** - 15 -](#_Toc461198036)

[**3.4.2.** **Push / Pull** - 17 -](#_Toc461198037)

[**3.4.3.** **Branch** - 19 -](#_Toc461198038)

[**4.** **참고자료 및 다운로드 링크** - 24 -](#_Toc461198039)

[**4.1.** **학습자료** - 24 -](#_Toc461198040)

[**4.1.1.** **누구나 쉽게 이해할 수 있는 Git 입문** - 24 -](#_Toc461198041)

[**4.1.2.** **Pro Git** - 24 -](#_Toc461198042)

[**4.2.** **원격 저장소 사이트** - 24 -](#_Toc461198043)

[**4.2.1.** **Git Hub** - 24 -](#_Toc461198044)

[**4.2.2.** **Bit-Bucket** - 24 -](#_Toc461198045)

[**4.3.** **Git 유틸리티 툴 ( GUI )** - 24 -](#_Toc461198046)

[**4.3.1.** **Source Tree** - 25 -](#_Toc461198047)

[**4.3.2.** **GitHub for Windows or Mac** - 25 -](#_Toc461198048)

[**4.3.3.** **Tortoise Git** - 25 -](#_Toc461198049)

[**4.3.4.** **Egit ( Eclipse-Plug in)** - 25 -](#_Toc461198050)

[**4.3.5.** **Git 내장 툴** - 25 -](#_Toc461198051)

1. **개요**

**Abstract**

본 문서는 Git을 처음으로 사용하는 개발자들에게 ‘기초적인 개념과 사용법’을 전달하기 위한 목적으로 작성된 문서입니다. 기본적으로 구성된 내용이 ‘Git에 대한 이해(개념, 명령어), 설치 방법, 사용 방법’등 기초적인 내용으로 구성되어 있기 때문에, 이미 Git에 대한 사전 지식이 충분하고, 본인의 Git 설정을 보유하신 분들은 프로젝트 CM문서를 참고하시면 됩니다.

* 1. **버전 관리 시스템 [ VCS ] ?**

Git을 한마디로 표현하자면 버전 관리 시스템입니다. 버전관리 시스템은 파일 변화를 시간에 따라 기록했다가 나중에 특정 시점의 버전을 다시 꺼내올 수 있는 시스템을 말합니다. 이는 소스코드 뿐만 아니라 실제로 거의 모든 컴퓨터 파일의 버전을 관리 할 수 있는 시스템이라고 생각하시면 됩니다.

버전관리 시스템이 필요한 이유는 다음과 같습니다.

**개발 하는 동안 소스 코드의 변경 사항을 보존하기 위해**

- 버그 및 문제점이 발생했을 때 추적에 유용

- 과거 특정 시점의 소스 파일 및 디렉토리의 내용을 손쉽게 확인 가능

- 과거 특정 시점의 소스 파일로 손쉽게 되돌릴 수 있음

**협동 작업을 가능하게 하기 위해**

- 대부분의 프로젝트는 팀 단위이기 때문에 전체 팀원이 하나의 소스를 가지고 효율적으로 작업할 수 있는 도구가 필요함

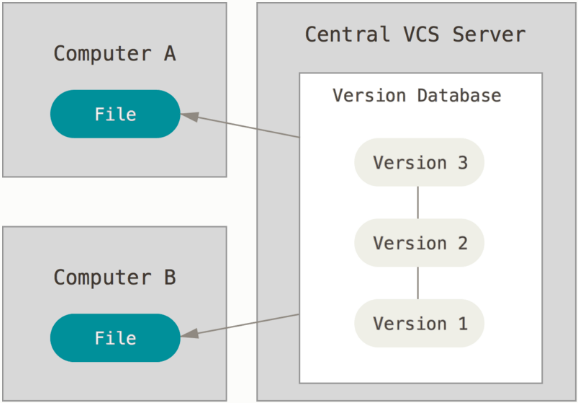
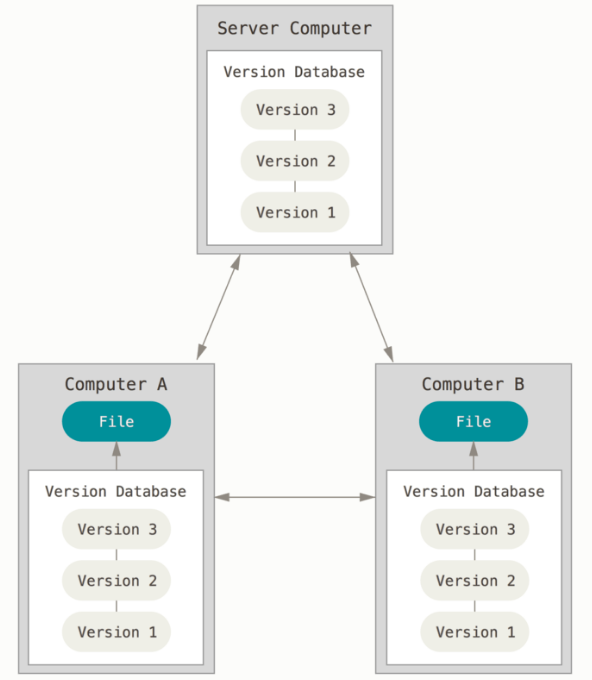
- 인터넷을 이용한 전세계 개발자들이 협업을 하여 개발하는 오픈 소스 프로젝트에 필수

프로그램을 만들다 보면, 잘못 만들어서 다시 소스코드를 이전 상태로 되돌릴 필요도 있고, 변경된 이력을 확인할 필요도 있습니다. 그리고 동시에 같은 소스코드를 개발하면서 발생하는 충돌에 대한 처리도 필요합니다. 이러한 일련의 상황들을 대비하고 대응하기 위해 버전관리 시스템을 사용하는 것입니다.

* 1. **Git**
     1. **Git이란?**

Git은 분산형 버전 관리 시스템(DVCS)입니다. 기존 중앙 관리형 버전 관리 시스템(CVCS)에서는 서버에서 파일의 마지막 스냅샷(Snapshot)을 체크아웃(Checkout)해 로컬에서 사용하는 방식입니다. 이는 서버에서 발생하는 장애에 대해 치명적이라는 단점이 있습니다. 그러나 **DVCS는 저장소(Repository)를 전부 복제해서 전체 파일 시스템을 하나의 클라이언트(Client)처럼 사용하는 방식입니다. 그래서 서버에 문제가 생겼을 때, 소스 변경 작업이나 복구가 용이하다는 점에서 큰 강점을 가지고 있습니다.** 이 외에도 Git은 ‘속도, 무결성, 코드 관리’ 측면에서 뛰어난 성능을 보장하고 있습니다.

* **속도 : 모든 작업을 로컬에서 사용하기 때문에 서버에 제약 받지 않음**
* **무결성 : 해시를 사용하여 체크섬을 구하여 활용함, git의 원자 데이터 단위는 체크섬**
* **코드관리 : Git은 모든 작업 로그를 데이터베이스에 추가한다. (삭제 불가) -> 손실 복구에 용이**

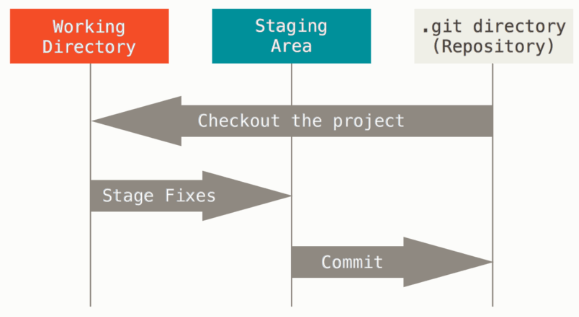


***[그림#1 중앙 관리형 버전관리 원리] [그림#2 분산 관리형 버전관리 원리]***

* + 1. **Git 기초 개념**
* Git의 세가지 상태 (파일을 Committed, Modified, Staged 상태로 관리함 )
  + Committed란, 데이터가 로컬 데이터베이스에 안전하게 저장됐다는 것
  + Modified란, 수정한 파일을 아직 로컬 데이터베이스에 Commit하지 않은 것
  + Staged란, 현재 수정한 파일을 곧 Commit할 것이라고 표시한 상태

위의 세 가지 상태는 Git 프로젝트의

**‘Git dir, Wokring dir, Staging area’로** 연결되어 있습니다.



***[그림#3 ‘Git dir, Wokring dir, Staging area’]***

* **Git dir은,** Git이 프로젝트의 메타데이터와 객체 데이터베이스를 저장하는 곳을 말합니다. 다른 컴퓨터에 있는 저장소를 Clone 하면 Git 디렉토리가 생성 됩니다.
* **Working dir은,** 프로젝트의 특정 버전을 Checkout한 것입니다. Git dir이 현재 작업하는 디스크에 있고 그 디렉토리 안에 압축된 데이터베이스에서 파일을 가져와서 Working dir을 만듭니다.
* **Staging area는,** Git dir에 존재하며 단순히 곧 Commit 할 파일에 대한 정보를 저장합니다. 종종 ‘Index’라고 불리기도 하지만, Staging area라는 명칭이 표준입니다.

**이러한 연결 구도에서 우리가 Git으로 하는 일은 아래와 같습니다.**

* *Working dir에서 파일을 수정한다.*
* *Staging area에 파일을 Stage해서 Commit할 스냅샷을 만든다.*
* *Staging area에 있는 파일들을 Commit해서 Git dir에 영구적인 스냅샷으로 저장한다.*

Git dir에 있는 파일들은 Committed 상태이며, 파일을 수정하고 Staging area에 추가했다면 Staged가 됩니다. 그리고 Checkout 하고 나서 수정했지만, 아직 Staging area에 추가하지 않았으면 Modified 상태라고 합니다.

1. **작업 환경 구축 및 절차**
   1. **환경 설정 및 설치**

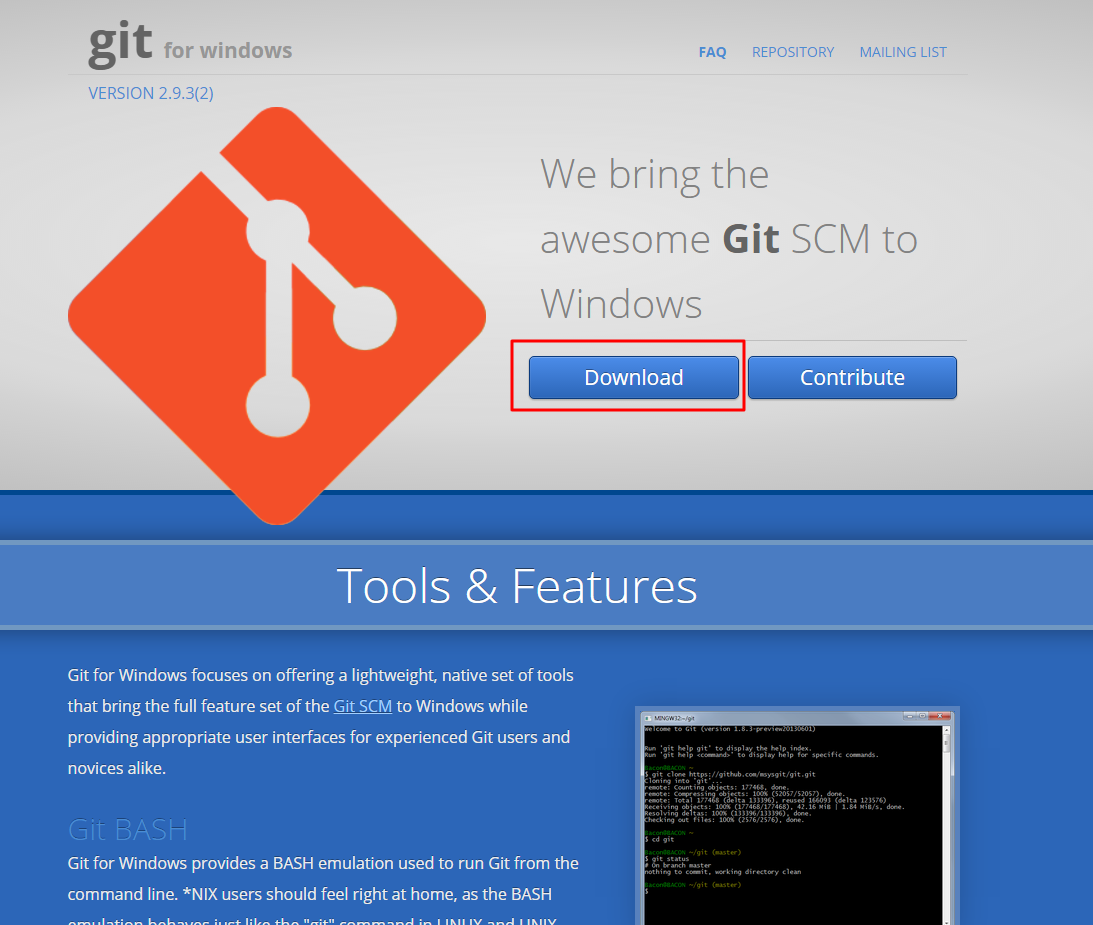
본 문서는 Git에 대한 기초 설명 및 사용법을 설명합니다. OS에 제약 받지 않고 원활한 학습을 위해 CLI 환경을 기준으로 Git환경을 구축하고, 작업은 서버 작업과 로컬 작업으로 구분하여 설명하기 위해 원격 저장소인 ‘Git Hub’를 설치합니다.

* + 1. **Git 설치 및 초기설정**
* Windows 설치

Windows용 Git에는 여러 가지가 있습니다. 그러나 본 문서에서는 Git, Git bash( CLI ), Git GUI를 통합으로 제공하는 “Git For windows(msysGit)”를 설치합니다.

**다운로드 링크**

<http://msysgit.github.io/>



***[그림#4 msysGit – Download view]***

해당 URL에 접속하면 ‘그림#4’와 같은 웹 페이지가 보여집니다. 이 페이지에서 별도의 설정 없이 다운로드하시고 ‘.exe’파일을 설치하면 자동으로 Git과 Git유틸들이 설치됩니다.

**Git 공식 공식 버전 다운로드 [windows, linux, mac]**

**Windows** : <https://git-scm.com/download/win>

**Linux** : https://git-scm.com/download/linux

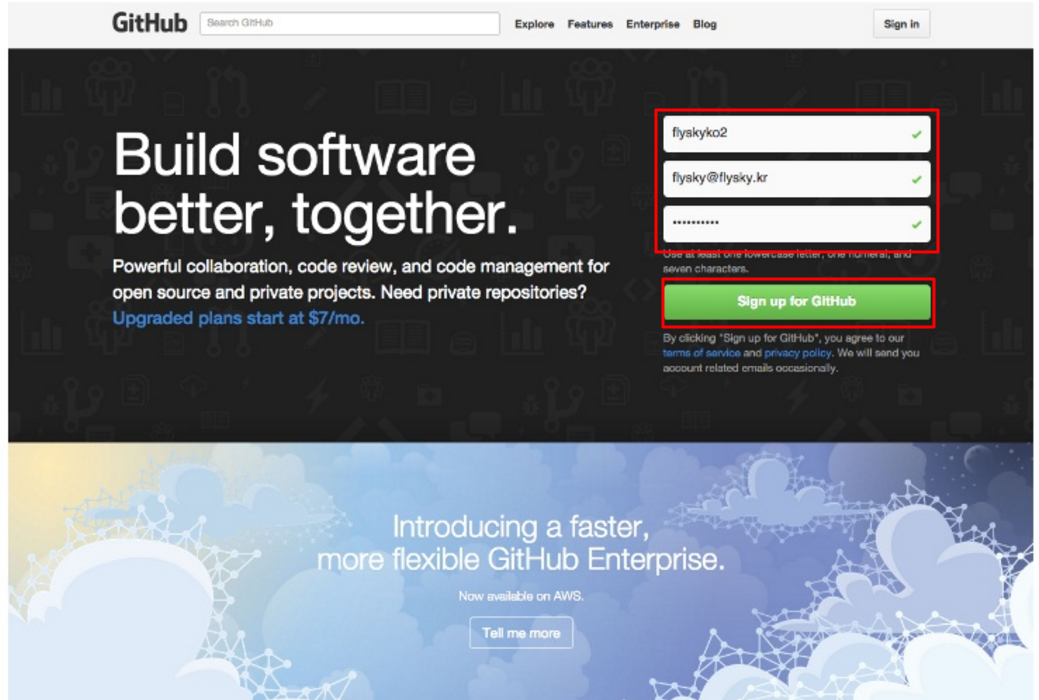
**Mac** : https://git-scm.com/download/mac

* + 1. **GitHub 기본 설정**

실제 작업 서버에서 Git을 실습 할 수 없기 때문에, GtiHub라는 무료 원격 저장소 활용합니다. GitHub는 회원가입 및 기본 설정이 필요합니다.

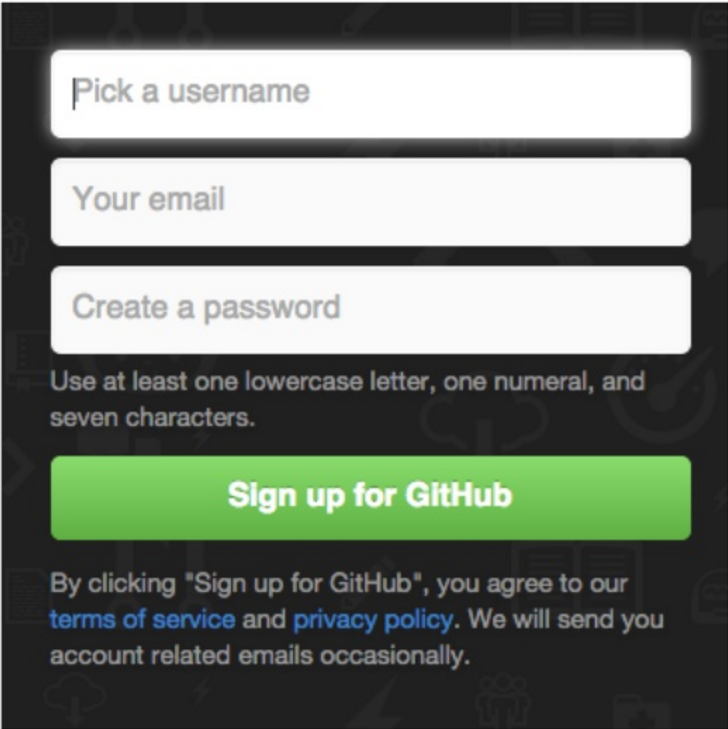
**GitHub URL**

<https://github.com>

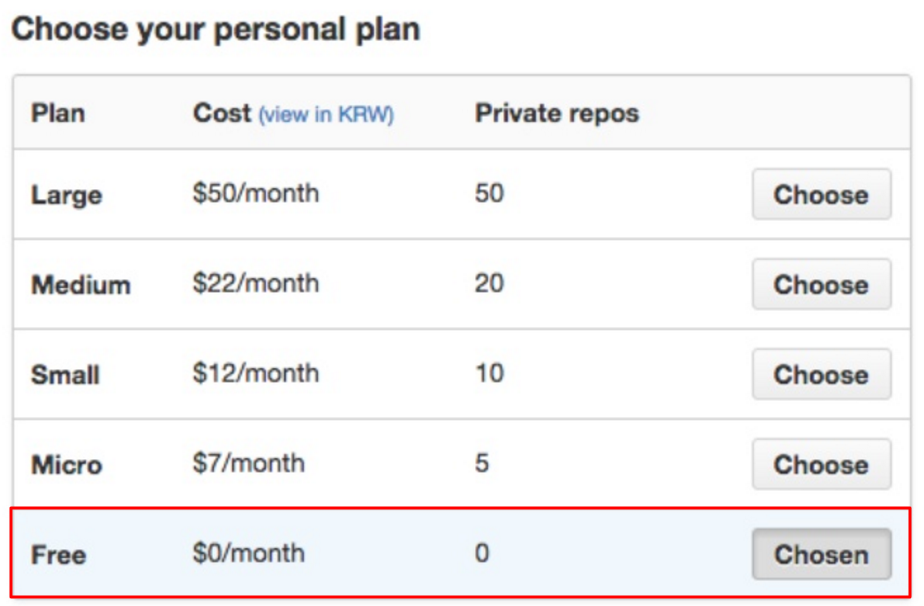


***[그림#5 GitHub main view]***

* 해당 URL에 접속 후 원하시는 아이디와 비밀번호 그리고 이메일을 입력한 후 회원가입을 진행합니다.

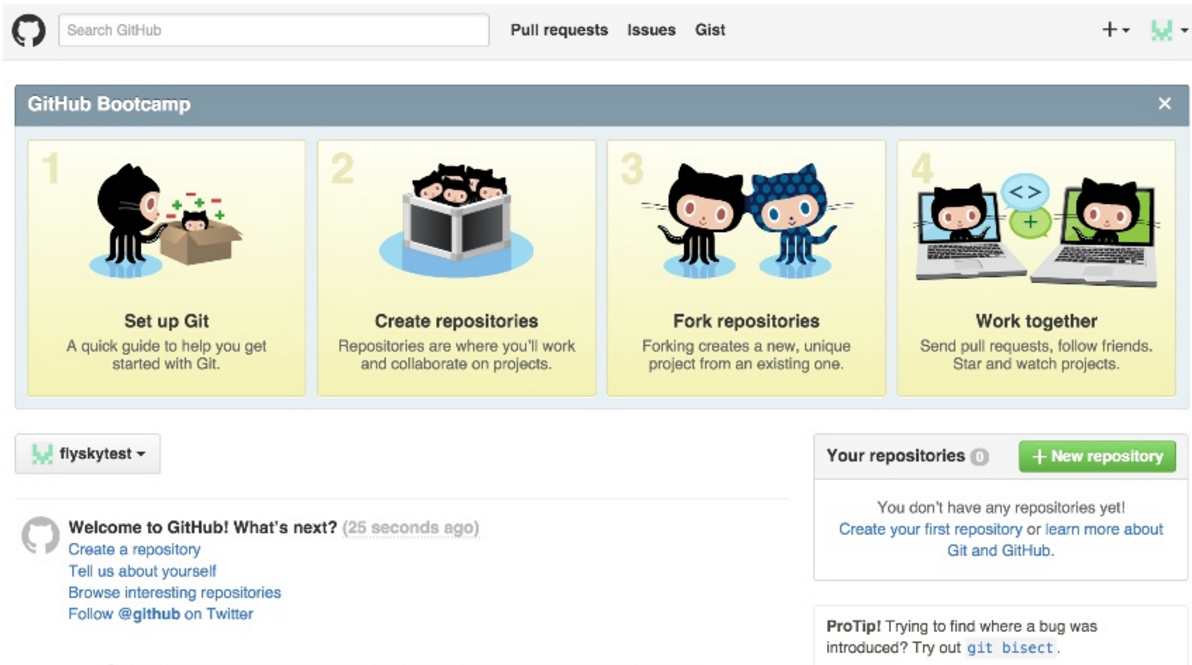
******

***[그림#6 Sign up view]***



***[그림#7 User cost view]***

Free를 선택하신 후, 계속 진행하시면 회원가입이 완료됩니다.



***[그림#8 Complete sign up]***

**회원가입이 완료 된 후, 반드시 기입하신 이메일에서 인증을 받으셔야 합니다.**

( 이메일 인증을 받지 않으면 계정이 잠기게 됩니다. )

1. **GitHub / Git 기초 사용법**

저장소를 만들고 설정하는 방법, 변경 내용을 Stage하고 Commit하는 방법을 알아봅니다. 또한 버전관리의 핵심인 히스토리를 조회하고 Commit을 비교하는 방법, 원격 저장소에 Push 하고 Pull하는 방법 등, Git의 기초적인 사용법에 대해 알아볼 것입니다.

* 1. **Git 저장소 생성**
* **기존의 디렉토리를 Git 저장소로 만들기**

기존의 프로젝트를 Git으로 관리하고 싶을 때, 프로젝트의 디렉토리로 이동해서 “git init”이라는 명령을 실행합니다.

* 실습을 위해 Git Bash 프로그램을 실행합니다. 그 후 ‘cd’ 명령어를 이용하여 원하는 디렉토리에서 실습용 저장소를 생성합니다.





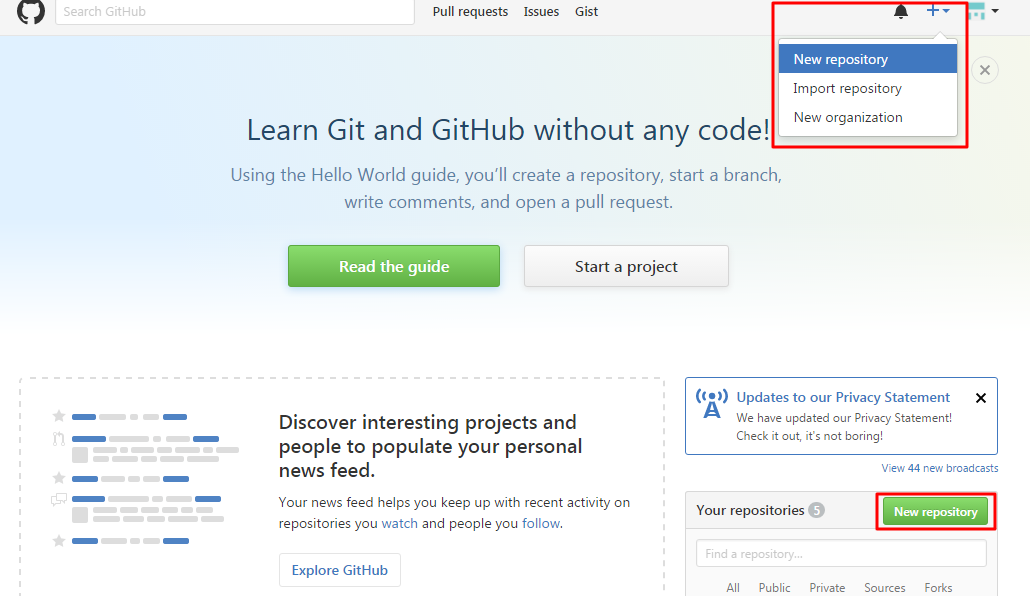
* 이동 완료 후 “git init” 명령어를 실행합니다.



* 위 이미지와 같은 문구가 보인다면 성공적으로 Git 저장소가 생성된 것입니다.
* **GitHub를 활용하여 원격 저장소 만들기**

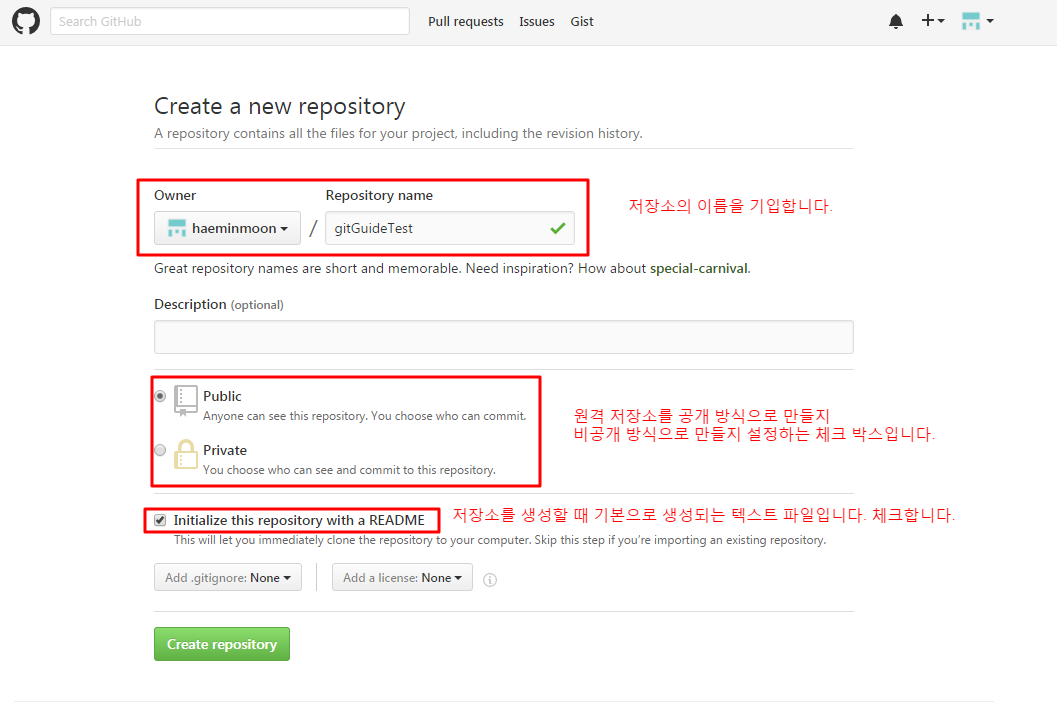
이번에는 GitHub를 활용해 원격 저장소를 생성해보도록 하겠습니다. 사전에 가입한 GitHub 사이트에 접속하신 후 로그인 합니다.

* 아래와 같은 화면 처럼 ‘New repository’ 버튼을 클릭합니다.

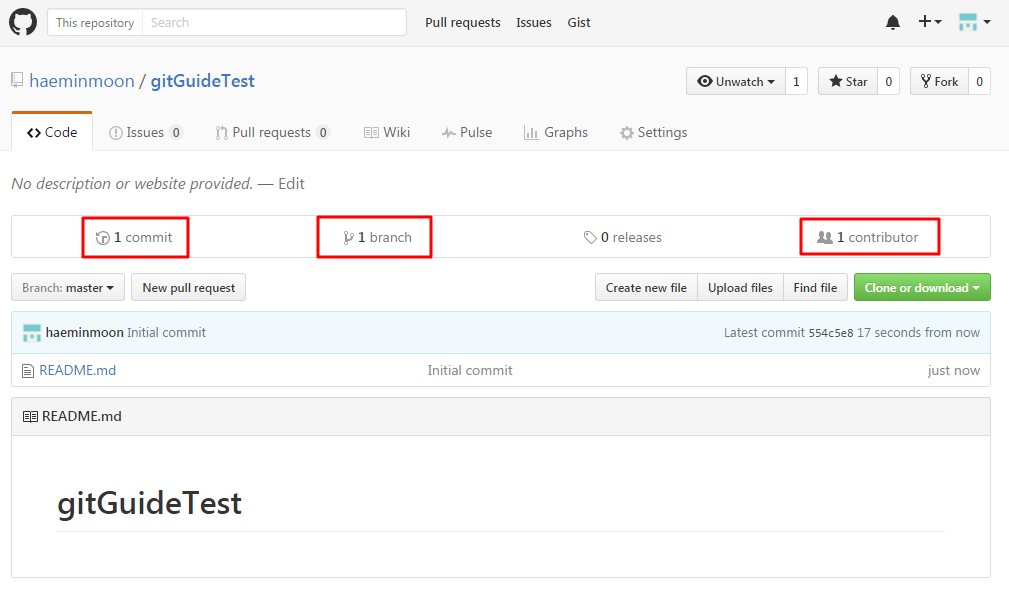


***[그림#9 GitHub logined view]***

* 아래와 같은 화면에서, 저장소의 이름을 기입하여 저장소를 생성합니다.



***[그림#10 GitHub create a new repository]***

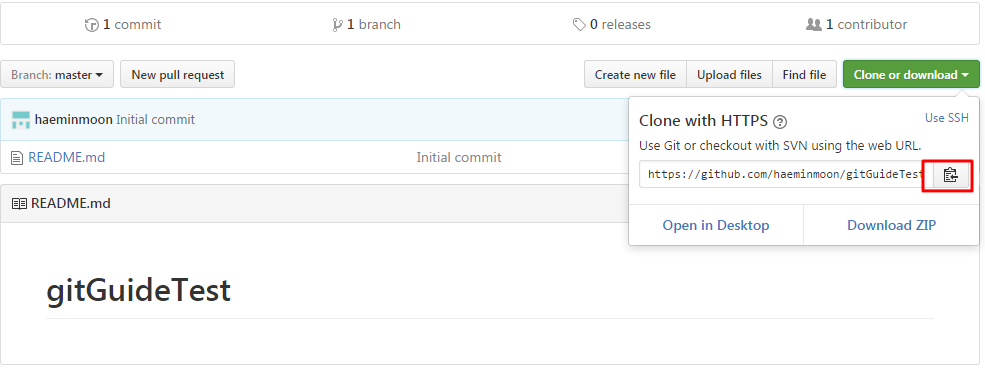


***[그림#10 GitHub create a new repository]***

위와 같은 화면이 보이시면, 원격 저장소 생성이 성공적으로 완료 된 것입니다.

* 1. **Git 저장소 복제**

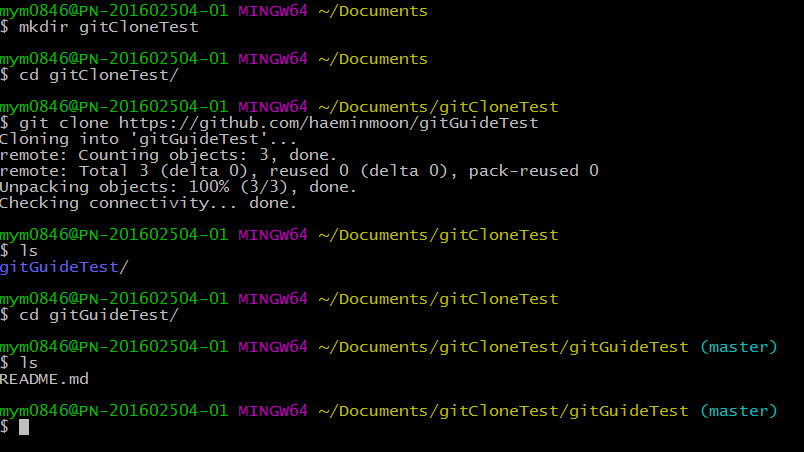
실질적인 협업 프로젝트를 진행하기 위해 개발자들은 원격 서버에서 프로젝트 관련 코드를 내려 받습니다. 이처럼 원격서버에 존재하는 저장소를 개인의 PC에 복제( Clone ) 해보도록 하겠습니다.



***[그림#11 Repository clone]***

* GitHub 저장소 페이지에서 오른쪽 상단에 위치한 ‘Clone or Download’ 버튼을 클릭합니다. 그 후 보여지는 팝업에서 URL을 클립보드에 복사합니다. 복사가 완료된 후 Git Bash를 실행시키신 후 다음과 같이 입력합니다.

1. **원격 저장소를 복제하기 위해 새로운 디렉토리 생성합니다.**
2. **해당 디렉토리로 이동 후 “git clone + 저장소 페이지 URL” 입력합니다.**



***[그림#11 Clone a repository – Flow]***

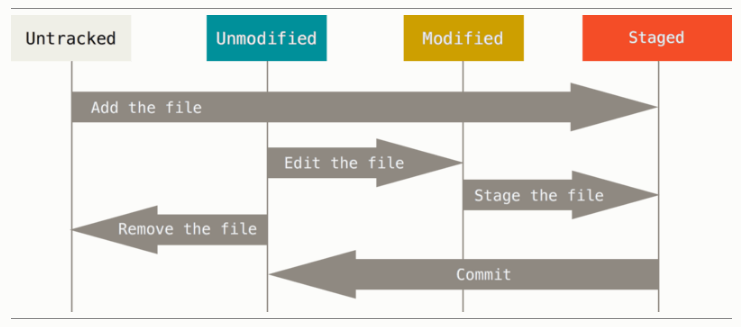
위 이미지는 서버에 존재하는 원격 저장소를 자신의 로컬 저장소로 복제하는 플로우을 보여줍니다. 이러한 방식으로 개발자들은 프로젝트를 공유받고, 자신의 소스코드를 로컬에서 작업한 후 서버에 업로드 하는 싸이클로 버전관리를 진행합니다.

* 1. **파일 수정과 상태 변경**

현재까지, 작업할 수 있는 원격 저장소를 만들었고 워킹 디렉토리에 Checkout도 했습니다. 본격적으로 Git을 활용하기 전에 Git에서 파일들의 상태 변화에 대해 알아보도록 하겠습니다.

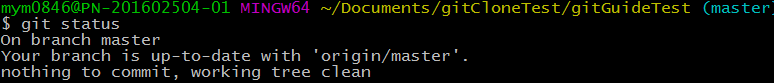
워킹 디렉토리의 모든 파일은 크게 Tracked(관리대상임)와 Untracked(관리대상이 아님)로 나뉩니다. Tracked 파일은 이미 스냅샷에 포함돼 있던 파일인데, Tracked 파일은 또 Unmodified(수정하지 않음)와 Modified(수정함) 그리고 Staged(Commit으로 저장소에 기록할) 상태 중 하나로 분류할 수 있습니다. 그리고 나머지 파일은 모두 Untracked 파일 입니다. Untracked 파일은 워킹 디렉토리에 있는 파일 중 스냅샷에도, Staging Area에도 포함되지 않은 파일인데, 처음 저장소를 Clone 하면 모든 파일은 Tracked이면서 Unmodified 상태가 됩니다. 파일을 Checkout 하고 나서 아무것도 수정하지 않았기 때문입니다.

마지막 Commit 이후 아직 아무것도 수정하지 않은 상태에서 어떤 파일을 수정하면 Git은 그 파일을 **Modified** 상태로 인식하게 되는데, 실제로 Commit을 하기 위해서는 이 수정한 파일을 Staged 상태로 만들고, Staged 상태의 파일을 Commit하는 라이프 사이클을 반복하게 됩니다.



***[그림#12 파일의 라이프사이클]***

* 작업을 진행하기 전 파일의 상태를 확인하는 방법에 대해 알아보겠습니다.



‘ git status ‘라는 명령어를 워킹 디렉토리에서 입력하면 위와 같이 Branch에 대한 정보와 Commit과 워킹 플로우에 대한 설명을 확인할 수 있습니다. 위 이미지의 내용은 파일을 수정하지 않았다는 것을 말해주는데 Tracked나 Modified 상태인 파일이 없다는 것입니다. 아직 워킹 디렉토리에서 아무 작업도 하지 않았기 때문에 위와 같은 상태 확인 메시지를 확인하게 되는 것 입니다.

**이처럼 Git 파일들의 상태 특징 및 Git 워킹 방식을 잘 이해하면 향후**

**Git 숙련 지식 습득 및 발생하는 문제해결 측면에서 큰 도움이 될 것입니다.**

* 1. **원격 저장소(리모트 저장소) 활용 튜토리얼**

지금부터는 로컬 디렉토리에서 작업한 결과를 원격 저장소(리모트 저장소)에 반영하고, 이를 다시 복제하며, 협업하는 과정들을 실습합니다.

* + 1. **Commit**
* 워킹 디렉토리에서 vi편집기를 이용하여 test 파일을 생성합니다.

C:\Users\한화에스엔씨\Desktop\5.PNG

* 파일을 생성한 후 내용을 입력합니다.

C:\Users\한화에스엔씨\Desktop\1.PNG

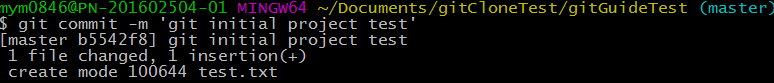
현재 워킹 디렉토리 안에는 test.txt라는 파일이 생성되었습니다. 이는 현재 로컬 저장소에서 작업이 이루어졌기 때문에 원격 저장소와 차이가 있는 상태가 된 것입니다. 이때 작업자의 로컬 저장소에서 작업이 이루어졌다는 것을 알리기 위해 ‘Commit’이라는 명령어를 사용합니다.

**“3.3 파일의 상태”**에서 설명했듯이 작성한 파일이나 수정한 파일들을 Commit하기 위해선 파일들을 ‘Staged’ 상태로 만들어줘야 정상적인 Commit 규칙에 의해 Commit이 가능해집니다.

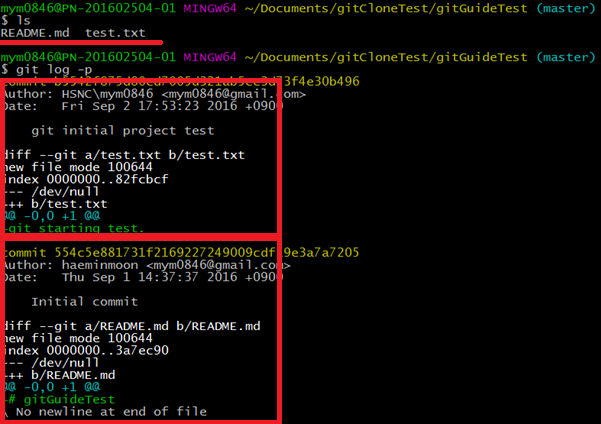
* 이때 파일들을 ‘Staged’ 상태로 만들어 주기 위해선 ‘ git add 파일명 ‘으로 상태를 변경 해줘야 합니다.

C:\Users\한화에스엔씨\Desktop\4.PNG

* 파일들이 ‘Staged’ 된 후, ‘ git commit –m ‘ 자신이 원하는 ‘Commit message‘ 를 입력 합니다.



* Commit의 상태 확인을 위해 ‘ git log –p ‘ 명령어로 Commit 로그를 확인할 수 있습니다.



확인 결과, 현재 워킹 디렉토리에는 “최초 Commit, README.md”, “두번째 Commit과, test.txt” 작업들로 구성되었으며, 이를 통해 성공적으로 작업이 Commit 되었음을 확인 할 수 있습니다.

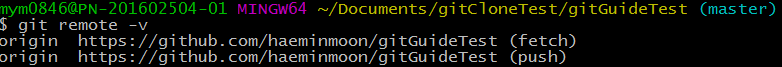
* + 1. **Push / Pull**

Commit까지는 완료되었지만, 워킹 디렉토리에서 작업한 내용이 아직 원격 저장소에 반영되지 않았기 때문에 ‘자신이 작업한 내용’을 다른 사람들과 공유할 수 없는 상태입니다. 원격 저장소는 인터넷이나 네트워크 어딘가에 있는 저장소를 말하는데, Push, Pull 명령어로 작업 내용을 반영하고 공유 할 수 있습니다.

**원격 저장소 관리는 작업을 공유하는 것뿐만 아니라**

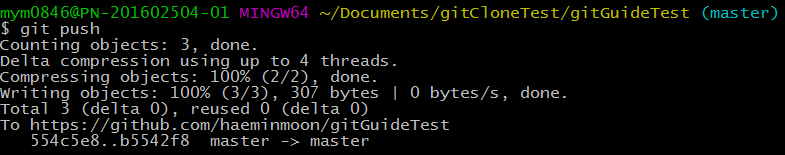
**저장소를 추가, 삭제하고 로그 추적, 컨트리뷰터(접근자) 제어 등을 통해 버전 관리가 가능합니다.**

* 먼저 ‘ git remote -v‘ 라는 명령어를 통해 현재 작업 영역에 등록된 원격 저장소를 확인해 볼 것입니다.



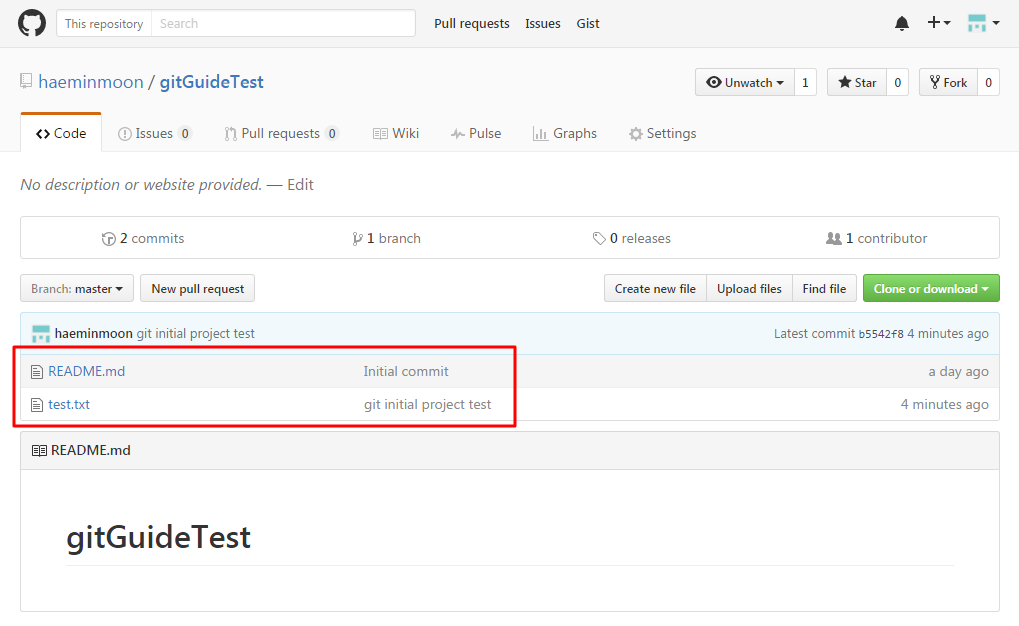
정상적으로 워킹 디렉토리에 원격 저장소가 연결된 것을 확인했으니, 워킹 디렉토리에서 작업한 Commit을 원격 저장소로 업로드 할 것입니다.

* ‘ git push ‘ 명령으로 원격 저장소에 작업 내용을 업로드 합니다.



**\*만약 원격 저장소가 여러 개일 경우 해당 원격 저장소를 선택 후 push 해야 합니다.**

* 아래 이미지와 같이 Push 명령어가 성공적으로 수행되어, GitHub의 원격 저장소에 작업내용이 반영되었음을 확인 할 수 있습니다.



***[그림#13 GitHub 원격 저장소]***

다음은 Pull 명령어를 활용하여 원격 저장소에서 최신의 작업내용을 복제해 보도록 하겠습니다. Pull의 개념은 ‘자신 말고 다른 작업자’가 작업을 한 후 원격 저장소에 Push 하면 그 내용을 이어서 작업하기 위해 ‘최신 버전의 작업내용’을 내려 받는 것 입니다.

* ‘ git pull ‘ 명령어를 이용하여 원격 저장소의 내용을 내려 받습니다.

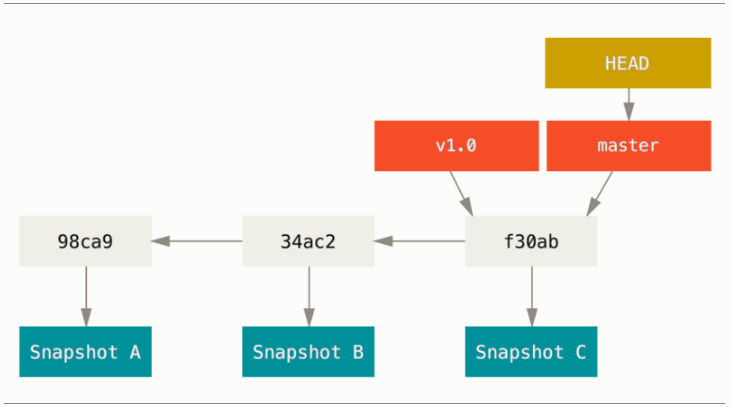


위 이미지는 현재 다른 작업자가 아무도 Push하지 않았기 때문에, 현재 원격 저장소가 최신 상태라는 메시지 입니다. ‘ git pull ‘ 명령어는 만약 현재 자신의 로컬 저장소가 최신 버전이 아니라면 원격 저장소의 최신 버전의 작업을 ‘Merge’해 주는 역할을 합니다.

* + 1. **Branch**

Git의 강력한 기능 중 하나인 Branch는 Commit 사이를 가볍게 이동할 수 있는 어떤 포인터 같은 것입니다. 이 Branch를 활용하여 “분류 -> 병합” 과정을 반복하면서 작업을 진행 할 수 있습니다. 우리는 이를 활용하여 개발자 구분, 모듈 구분, 작업 영역 구분 등의 작업들을 분류하여 형상관리를 진행할 수 있습니다.

기본적으로 Git은 Master branch를 만듭니다. 최초로 Commit하면 Git은 Master라는 이름의 Branch를 만드는데, Commit을 만들 때 마다 Branch가 자동으로 가장 마지막 Commit을 가리키게 됩니다. 즉, Master branch는 ‘git init‘ 명령어로 워킹 디렉토리를 만들 때, 최초의 Commit이 이루어지기 때문에 최초의 Commit을 가리키고 있는 것입니다.

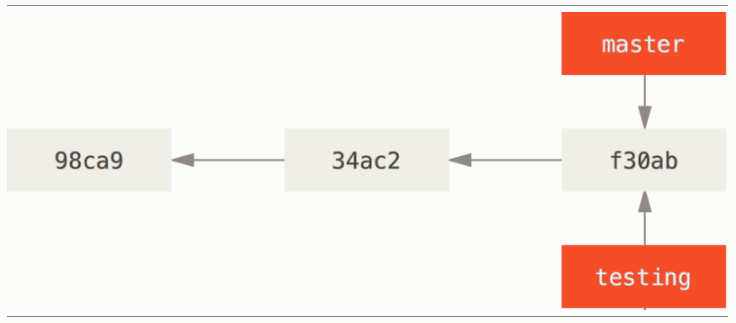


***[그림#14 Branch와 Commit 히스토리]***

* **Branch 생성하기**
* Branch는 ‘git branch’ 명령으로 Branch를 생성할 수 있습니다. 해당 디렉토리에서 ‘ git branch testing ‘ 이라는 명령어를 입력해보시길 바랍니다.



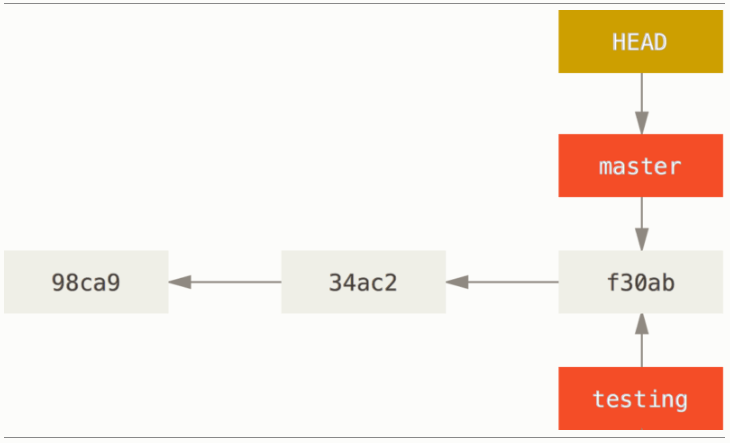
명령어를 입력한 후 새로 생성된 Branch는 다음과 같이 마지막 Commit을 가리키게 됩니다.



***[그림#15 한 개의 Commit 히스토리를 가리키는 두 개의 Branch]***

**‘그림#15’**를 참고하면 현재 현재 작업중인 두 개의 Branch(master, testing)가 보일 것입니다. 여기서 우리는 현재 작업 중인 Branch가 무엇인지 구분할 필요가 있습니다. Git에는 다른 버전 관리 시스템과는 달리 ‘ HEAD ‘ 라는 특수한 포인터가 있습니다. 이 포인터는 지금 작업하는 로컬 Branch를 가리키고 있습니다.

* 현재는 Branch만 생성하고 Branch를 옮기지 않았기 때문에 HEAD 포인터는 master를 가리키고 있습니다.

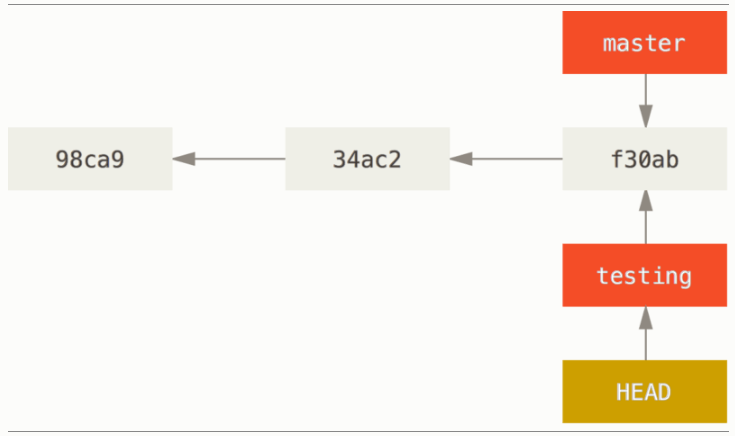


***[그림#16 현재 작업중인 Branch를 가리키는 HEAD]***

* **Branch 이동하기**
* ‘ git checkout ‘ 명령으로 Branch간 이동이 가능합니다. ‘ git checkout testing ‘ 명령을 입력해 보시길 바랍니다.

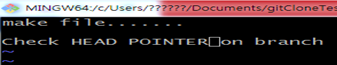


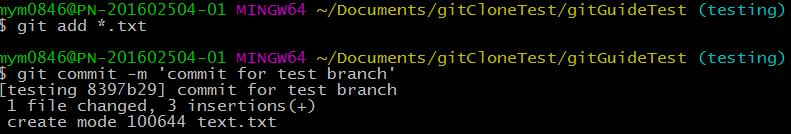
위 이미지 처럼 Branch가 성공적으로 이동된다면 HEAD 포인터는 testing branch를 가리키게 됩니다.



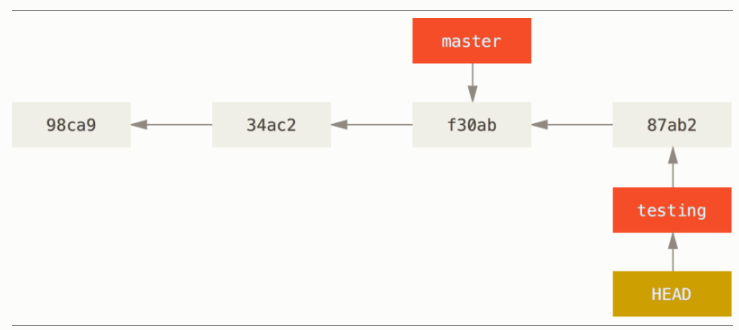
***[그림#17 HEAD는 testing을 가리킴]***

* Branch의 이동까지 완료했으니 파일을 수정한 후 Commit을 진행하도록 하겠습니다.
* vi 편집기를 활용하여 test.txt 파일 수정 후 commit



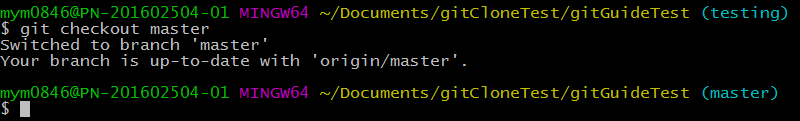


이렇게 Branch를 생성하여, 그 Branch로 작업을 하면 아래와 같이 HEAD 포인터가 새로운 Commit을 가리키게 됩니다.



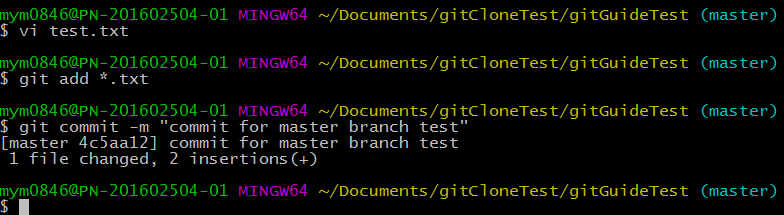
***[그림#18 HEAD 포인터가 새로운 Commit을 가리킴]***

* ‘ git checkout master ‘ 명령을 이용하여 다시 HEAD 포인터가 Master를 가리키게 합니다.

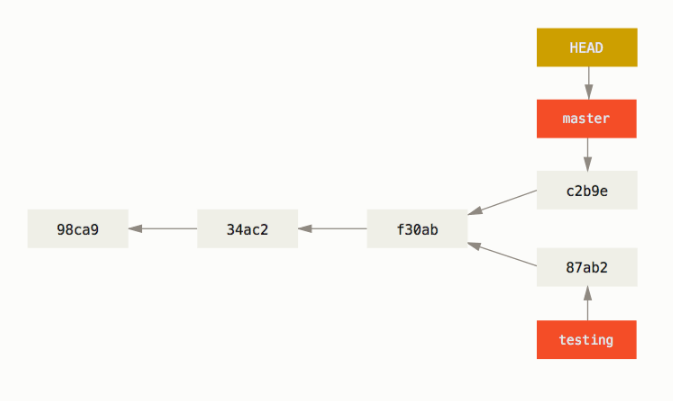


지금까지 실행한 일은 총 두 가지 입니다. master branch가 가리키는 Commit을 HEAD가 가리키게 하고 워킹 디렉토리의 파일도 그 시점으로 되돌려 놓았습니다. 앞으로 Commit을 하면 다른 Branch의 작업들과 별개로 진행되기 때문에 testing branch에서 임시로 작업하고 원래 master branch로 돌아와서 하던 일을 계속할 수 있게 되는 것입니다.

* **브랜치를 이동하면 워킹 디렉토리의 파일이 변경된다는 점을 기억해야함 \***
* 다시 한번 파일을 수정하고 Commit을 진행하도록 하겠습니다.



위와 같이 명령을 수행하면 다음과 같은 플로우가 생성됩니다.



***[그림#19 Branch가 갈라짐]***

이처럼 Branch를 활용하여 워킹 디렉토리를 구분하여 작업 영역을 분할하여 누가 어떤 작업을 했는지를 알 수 있으며, 이러한 로그들을 통해 문제를 보다 쉽게 해결하기 위한 기반으로 활용합니다. 또한 Branch의 ‘Git Flow’라는 Branch 개념을 활용하면 **“ 개발-> 디버깅 -> 릴리즈 -> 빌드 “** 등 소프트웨어 공학적 프로세스를 보다 효율적으로 적용시킬 수 있는 기능을 하기도 합니다.

1. **참고자료 및 다운로드 링크**

Git에 대한 가장 ‘원론적인 지식과 Git이 어떻게 워킹하는지’ 이해를 돕기 위해 간단한 실습과 이론을 위주로 설명하였습니다. Git이라는 버전 관리 시스템의 강력한 기능을 알아보기 위해서는 Git에서 제공하는 다양한 제어 명령어와 개념들을 이해할 수 있어야 합니다. 이를 돕기 위해 도움이 될 만한 학습자료 및 유틸 사이트를 첨부하였습니다.

* 1. **학습자료**
     1. **누구나 쉽게 이해할 수 있는 Git 입문**

<https://backlogtool.com/git-guide/kr/>

* + 1. **Pro Git**

<https://git-scm.com/doc>

* 1. **원격 저장소 사이트**
     1. **Git Hub**

<https://www.github.com/>

* + 1. **Bit-Bucket**

<https://www.bitbucket.org/>

* 1. **Git 유틸리티 툴 ( GUI )**

GUI 환경에서 작업하면 ‘Git flow monitoring, Push, Pull’ 등 다양한 명령어를 손쉽게 사용할 수 있습니다. 즉, 편리한 버전관리 환경을 구축할 수 있는 것입니다. 하지만 편리함 때문에 처음부터 GUI 환경에 익숙해지고, 의존하다 보면 Git의 워킹 프로세스에 익숙해질 수 없기 때문에, 향후 장애발생시 문제점을 찾는다거나 대처할 수 있는 능력이 떨어질 수도 있습니다. 또한 GUI 환경에서는 Git의 모든 기능을 지원하지 않기 때문에, 세부적인 사항들을 컨트롤 할 때 제약이 생길 수 있습니다.

**☞ CLI 환경에서 충분히 연습하고, 기초 지식을 습득한 후 사용하시길 권장합니다.**

* + 1. **Source Tree**

<https://www.sourcetreeapp.com/>

* + 1. **GitHub for Windows or Mac**

https://desktop.github.com/

* + 1. **Tortoise Git**

<https://tortoisegit.org/>

* + 1. **Egit ( Eclipse-Plug in)**

<http://www.eclipse.org/egit/>

* + 1. **Git 내장 툴**

**Git for Windows를 설치하신 경우**

CLI 환경에서 git 로컬 저장소로 디렉토리 이동 ->

git gui 입력 -> Git GUI 팝업 실행