깃허브 역사

GitHub.com 플랫폼은 2007년 10월 19일에 만들기 시작했다. 2008년 4월에 톰 프레스턴-워너, 크리스 완스트라스, P. J. 하이엣, 스콧 샤콘이 베타 버전으로 몇 달 동안 사용 가능하게 만든 후, 이 사이트를 시작했다.

2012년부터 마이크로소프트는 GitHub를 사용하여 GitHub의 주요 사용자가 되었으며, GitHub를 사용하여 오픈 소스 프로젝트 및 와 같은 개발 툴을 호스팅하고 있다. NET Core, Chakra Core, MSBuild, PowerShell, PowerToy, Visual Studio Code, Windows Calculator, Windows 터미널 및 해당 제품 설명서에서 명시한다.

2018년 6월 4일, 마이크로소프트는 75억 달러에 GitHub를 인수할 계획을 발표했다. 이 계약은 2018년 10월 26일에 체결되었고, GitHub는 커뮤니티, 플랫폼 및 비즈니스로서 독립적으로 운영된다.

GitHub의 마스코트는 다섯 개의 문어 같은 팔을 가진 "옥토캣"이다. 이것은 그래픽 디자이너 Simon Oxley가 iStock에서 판매하기 위한 클립 아트로 만든 캐릭터였다. GitHub는 그가 트위터 로고로 디자인한 파란 새 디자인을 본 후 Oxley의 작품에 관심을 갖게 되었다. GitHub는 해당 디자인을 원했기 때문에, 이미지에 대한 독점권을 구입하기 위해 Oxley와 협상했다.

깃허브를 써야하는 이유

버전관리가 쉽다.

소프트웨어에는 버전관리가 필요하다. 꼭 클라이언트의 요청이 없더라도, 버전별로 이전 버전의 기능을 다시 가져오게 될 수도 있고, 누군가 잘못된 코드를 섞으면 복원해야 될 때도 있기 때문이다. 그래서 언제든 특정 시점의 버전을 다시 이용하는데 문제가 없게끔 변경된 내역을 관리해줘야 하는 것이다. 이 때 필요한 것이 버전관리를 위한 툴, Git이다.

다른 개발자와의 협업

어느 정도 규모가 있는 프로젝트를 진행할 때는 혼자가 아닌 여러명의 개발자가 협업하게 된다. 하나의 공유 가능한 공간 안에서 여러 개발자들이 각각 맡은 파트를 가져가서 작업하고, 작업사항을 업데이트하고, 서로의 작업 로그를 살펴볼 수 있어야 하며, 이에 더해 버전별 백업도 실시간으로 진행해야 한다.

하지만 여러 사람이 모듈별로 나누어 실시간으로 작업하는 소프트웨어 개발의 특성상 로컬 저장소라던지 메일이나 클라우드를 통해 관리하려면 어렵고 불편하다. 따라서 작업 중 실시간으로 작업내역을 공유할 수 있는 원격저장소가 필요하다. 이 때 필요한 것이 협업을 위한 저장소, Github이다.

Git 사용방식 이해하기

Git은 분산 저장소 방식으로 버전을 관리한다. 로컬저장소(PC, Git)과 원격저장소(GitHub) 양 쪽에서 관리를 할 수 있으므로 서버 단에 문제가 생기더라도 로컬에서 작업할 수 있어 지속적인 협업이 가능한 것이다. 다시 말해, 로컬에서 깃(Git)을 이용해 실제 개발과 버전 관리를 진행하고 깃헙(Github)에 반영본을 업데이트하는 식으로 작업이 진행된다.

순서도로 살펴보면 Git을 이용한 버전관리는 다음과 같다.

1) 로컬에 저장소를 생성한다 (init)

2) 깃허브 서버와 연결한다 (remote add)

3) 서버에서 전체 내용을 가져온다 (clone)

4) 작업이 진행된 이후에는 일부 변경된 내용만 가져온다 (fetch)

5) 필요에 따라 다른 서버의 내용을 가져올 수도 있다 (fork)

6) 추가작업은 브랜치(branch)를 따서 진행(checkout)하며 파일의 변화는 스냅샷(snapshot)으로 저장한다.

7) 새롭게 추가한 작업이 완성되면 마스터 브랜치와 병합 (merge) 한다

8) 개발이 완료되면 완료된 파일들을 스테이징 영역에 추가 (add) 한다

9) 내용이 안전하게 추가되었다면 로컬에 반영 (Commit) 한다.

Git 커맨드 정리

git init : 현재 디렉토리를 Git이 관리하는 프로젝트 디렉토리(=working directory)로 설정하고 그 안에 레포지토리(.git 디렉토리) 생성

git config user.name 'codeit' : 현재 사용자의 아이디를 'codeit'으로 설정(커밋할 때 필요한 정보)

git config user.email 'teacher@codeit.kr' : 현재 사용자의 이메일 주소를 'teacher@codeit.kr'로 설정(커밋할 때 필요한 정보)

git add [파일 이름] : 수정사항이 있는 특정 파일을 staging area에 올리기

git add [디렉토리명] : 해당 디렉토리 내에서 수정사항이 있는 모든 파일들을 staging area에 올리기

git add . : working directory 내의 수정사항이 있는 모든 파일들을 staging area에 올리기

git reset [파일 이름] : staging area에 올렸던 파일 다시 내리기

git status : Git이 현재 인식하고 있는 프로젝트 관련 내용들 출력(문제 상황이 발생했을 때 현재 상태를 파악하기 위해 활용하면 좋음)

git commit -m "커밋 메시지" : 현재 staging area에 있는 것들 커밋으로 남기기

git help [커맨드 이름] : 사용법이 궁금한 Git 커맨드의 공식 메뉴얼 내용 출력

Git hub 커맨드 정리

git push -u origin master : 로컬 레포지토리의 내용을 처음으로 리모트 레포지토리에 올릴 때 사용합니다.

git push : 로컬 레포지토리의 내용을 리모트 레포지토리에 보내기

git pull : 리모트 레포지토리의 내용을 로컬 레포지토리로 가져오기

git clone [프로젝트의 GitHub 상 주소] : GitHub에 있는 프로젝트를 내 컴퓨터로 가져오기

commit 다루기

git log : 커밋 히스토리를 출력

​git log --pretty=oneline : --pretty 옵션을 사용하면 커밋 히스토리를 다양한 방식으로 출력할 수 있습니다. --pretty 옵션에 oneline이라는 값을 주면 커밋 하나당 한 줄씩 출력해줍니다. --pretty 옵션에 대해 더 자세히 알고싶으면 이 링크를 참고하세요.

git show [커밋 아이디] : 특정 커밋에서 어떤 변경사항이 있었는지 확인

git commit --amend : 최신 커밋을 다시 수정해서 새로운 커밋으로 만듦

git config alias.[별명] [커맨드] : 길이가 긴 커맨드에 별명을 붙여서 이후로 별명으로 해당 커맨드를 실행할 수 있도록 설정

git diff [커밋 A의 아이디] [커밋 B의 아이디] : 두 커밋 간의 차이 비교

git reset [옵션] [커밋 아이디] : 옵션에 따라 하는 작업이 달라짐(옵션을 생략하면 --mixed 옵션이 적용됨)

(1) HEAD가 특정 커밋을 가리키도록 이동시킴(--soft는 여기까지 수행)

(2) staging area도 특정 커밋처럼 리셋(--mixed는 여기까지 수행)

(3) working directory도 특정 커밋처럼 리셋(--hard는 여기까지 수행)

그리고 이때 커밋 아이디 대신 HEAD의 위치를 기준으로 한 표기법(예 : HEAD^, HEAD~3)을 사용해도 됨

git tag [태그 이름] [커밋 아이디] : 특정 커밋에 태그를 붙임

Branch

git branch [새 브랜치 이름] : 새로운 브랜치를 생성

git checkout -b [새 브랜치 이름] : 새로운 브랜치를 생성하고 그 브랜치로 바로 이동

git branch -d [기존 브랜치 이름] : 브랜치 삭제

git checkout [기존 브랜치 이름] : 그 브랜치로 이동

git merge [기존 브랜치 이름] : 현재 브랜치에 다른 브랜치를 머지

git merge --abort : 머지를 하다가 conflict가 발생했을 때, 일단은 머지 작업을 취소하고 이전 상태로 돌아감