Git을 이용해 파일 관리하기

문제

개발자라면 종종 다양한 버전의 코드를 관리해야 하는 상황이 생긴다. 기존 애플리케이션에 영향을 주지 않으면서 새 기능을 추가하고 싶을 때나 애플리케이션이 너무 커서 한 부분을 고치면 전체에 어떤 영향을 미칠지 모를 때 같은 파일의 다른 버전을 만들면 좋다. 개발자들은 이미 모두 버전 컨트롤 시스템을 쓰고 있다. 작업 중인 프로젝트의 복사본을 폴더에 저장하는 것도 다른 버전을 만드는 일종의 파일 관리다. 다만 복사본을 만드는 것으로는 시간이 지나면서 기억이 나지 않거나 실수로 잘못 저장한 경우 다시 되돌릴 방법이 없어 관리하기가 쉽지 않을 뿐이다. 빠르고 강력하면서 현대적인 파일 관리와 여러 사람이 동시에 작업할 수 있는 방법이 필요하다.

재료

- Git[19]

해결책

버전 컨트롤 시스템(Version Control System, VCS)은 다양하다. 그 중에서 Git은 개발자들 사이에서 매우 유명하다. 파일을 로컬에 저장하는데, 폴더를 직접 복사해 다른 사본을 만드는 것보다도 빠르기 때문이다. 또한 여러 명이 작업할 수 있도록 여러 개의 버전을 동시에 만들어도 된다. 속도가 빠르니 저장도 자주 할 수 있고, 자연스럽게 문제가 생겼을 때 돌아갈 복구 지점이 많이 생긴다. 이와 같은 이유들 때문에 많은 오픈소스 프로젝트들이 Git을 버전 컨트롤 시스템으로 선택하고 있다.

아침 회의에서 사장님이 "지난주에 내게 보여 줬던 목(Mock) 중 두 개를 골랐네.

19 http://git-scm.com/

이제 기존 사이트에 기능을 추가한 데모를 만들어 보겠나? 데모니까 기존 사이트 코드와 따로 관리해야 하네."라고 주문했다.

기존 사이트를 위한 버전과 기존 사이트에 새로운 기능을 추가한 데모 버전 2개, 총 3가지 버전을 만들어야 할 것 같다. 이 참에 Git으로 파일 관리를 시작해 보자. Git을 이용하면 여러 개의 버전을 만드는 것도 쉬울 것이다. 데모가 사장님의 마음에 든다면 기존의 사이트에도 데모 버전을 반영해야 할 테니, 각 버전 간 동기화하는 연습도 해 보자.

Git 설정하기

Git을 설치하는 것부터 해 보자. Git의 웹사이트[20]에 방문해 운영체제에 적합한 버전을 다운받는다. Windows를 사용하고 있다면 MsysGit[21]을 받자. MsysGit을 설치하면 Git Bash를 쓸 수 있다. 윈도 사용자가 이 레시피를 따라하기 위해서는 일반 커맨드라인이 아닌 Git Bash가 있어야 한다.

Git은 정의한 사용자 이름으로 파일을 수정한 사람을 추적한다. 이렇게 추적을 할 수 있으면 나중에 누가, 무엇을, 언제 수정했는지 쉽게 알 수 있다. 이름과 이메일을 설정하자. 셸에 다음과 같이 자신의 정보를 이용해 명령어를 입력한다.

```
$ git config --global user.name "정용식"
$ git config --global user.email "sunphiz@gmail.com"
```

이제 Git을 설치하고 설정했으니, 기본적인 동작을 연습해 보자.

Git 기초

먼저 기존 프로젝트의 코드를 Git 저장소로 옮기자. 웹 프로젝트를 위해 git_site 폴더를 만들고, Git 저장소로 등록한다. 커맨드라인(윈도 사용자라면 Git Bash)에서 다음 명령어를 입력한다.

```
$ mkdir git_site
$ cd git_site
$ git init
```

폴더를 등록하면 확인 메시지가 나온다.

---

20 http://git-scm.com/

21 http://code.google.com/p/msysgit/

```
Initialized empty Git repository in /Users/webdev/Sites/git_site/.
git/
```

작업 디렉터리의 최상위에 .git 이라는 숨겨진 폴더가 자동으로 생긴다. 모든 히스토리와 방금 등록한 저장소에 대한 다른 자세한 내용들이 모두 이 폴더에 저장된다. Git이 우리 저장소의 변화를 추적하고 우리 코드의 스냅샷(snapshots)을 저장하려면 어떤 파일을 추적하길 바라는지 Git에게 알려 줘야 한다.

웹사이트 파일들을 git_site 폴더로 모두 옮기자. git 폴더에 넣을 웹사이트 파일들은 이 책의 소스코드 사이트에서 받을 수 있다.

파일을 옮겼으면 모두 저장소에 추가하여 Git으로 관리할 수 있도록 한다. 파일을 저장소에 추가하는 법은 다음의 명령어만 입력하면 된다. 간단하다.

```
$ git add .
```

Add 명령어는 아무런 처리 결과도 보여 주지 않는다. git status 명령어를 써야 결과를 확인할 수 있다. 꼭 추가 명령어의 결과 만이 아니라, Git 저장소의 상태를 확인하고 싶을 땐 git status 명령어를 사용하면 된다.

```
# On branch master
#
# Initial commit
#
# Changes to be committed:
#  (use "Git rm --cached <file>..." to unstage)
#
#    new file:  index.html
#    new file:  javascripts/application.js
#    new file:  styles/site.css
#
```

현재는 파일들을 추가만 하고, 커밋은 하지 않은 상태다. 어떤 파일이 커밋하기 직전인지 확인하고, 커밋하기 전에 한 번 더 생각해 볼 수 있다. 아무런 문제가 없어 보이니, 커밋을 하도록 하자.

```
$ git commit -a -m "첫번째 커밋"
```

이 명령어에서 사용한 옵션은 -a와 -m이다. -a는 커밋하기 전에 모든 변화를 인덱스에 추가하라는 것이고, -m은 커밋 메시지를 넣겠다는 뜻이다. 다른 버전 컨트롤 시스템과 다르게 Git은 커밋할 때마다 메시지를 넣어 줘야 한다. 커밋 메시지는 지금은 귀찮지만 나중에 커밋한 내용을 추적할 때 크게 도움이 되니, 커밋 메시지를

가볍게 여기지 말자. 커밋을 하면 아래처럼 확인 메시지를 볼 수 있다.

```
[master (root-commit) 94c75a2] 첫 번째 커밋
1 files changed, 17 insertions(+), 0 deletions(-)
create mode 100644 index.html
create mode 100644 javascripts/application.js
create mode 100644 styles/site.css
```

파일이 커밋되었는지 git status를 실행해서 확인할 수 있다.

```
# On branch master
nothing to commit (working directory clean)
```

모두 커밋이 되었다. 드디어 파일을 수정하고, 수정 사항을 추적할 스냅샷(snapshot)을 만들었다.

브랜치 작업하기

브랜치를 사용하면 우리 사이트에서 필요한 기능을 쉽게 추가할 수 있다. 현재 서비스 중인 기존 코드가 변경되진 않을까 하는 걱정 없이 말이다. Git은 다른 버전 컨트롤 시스템들보다 브랜치를 만드는 것이 쉬워서 많이 사용하게 된다.

사장님이 두개의 데모를 만들라고 했으니 브랜치도 2개가 추가돼야 한다. 각 브랜치를 layout_a와 layout_b라고 하자. 먼저 layout_a를 위한 브랜치를 만든다.

```
$ git checkout -b layout_a
Switched to a new branch 'layout_a'
```

git status를 실행하면 현재 브랜치가 나온다. 방금 layout_a 브랜치를 체크아웃했으니 수정 사항은 layout_a에 반영돼야 한다. index.html 파일로 가서 〈h1〉 태그 안의 글자를 '레이아웃 A'로 수정하고 저장하자. git status 명령어를 실행하면 다음처럼 나온다.

```
# On branch layout_a
# Changed but not updated:
#   (use "Git add <file>..." to update what will be committed)
#   (use "Git checkout -- <file>..." to discard changes in working
      directory)
#
#     modified:   index.html
#
no changes added to commit (use "Git add" and/or "Git commit -a")
```

변경 사항을 layout_a 브랜치에 커밋하자.

```
$ git commit -a -m "<h1> 값을 레이아웃 A로 변경"
```

layout_a 브랜치에서 작업하는 동안 사장님에게서 이메일이 왔다. "우리 홈페이지에 당일 배송이 가능하다고 써 있네. 그런데 더 이상 당일 배송 행사를 할 수 없을 것 같아. 익일 배송만 가능하다고 수정해 주겠나. 다른 고객이 신청하기 전에 어서 수정해 주게."라고 한다. 어서 마스터 브랜치로 돌아가 사장님이 말한 대로 수정하자.

```
$ git checkout master
```

마스터 브랜치를 체크아웃한 후 index.html 파일을 열면 layout_a 브랜치에서 수정했던 글을 찾을 수 없다. 우리가 변경한 내용은 다른 브랜치에 있고, 브랜치를 바꿀 때 Git이 파일의 내용을 바꾸었기 때문이다. 사장님이 원한 수정 사항을 홈페이지에 적용했으니, 마스터 브랜치에 커밋하자.

```
$ git commit -a -m "당일 배송 가능에서 익일 배송 가능으로 배송 행사 수정"
  [master d00d2de] 당일 배송 가능에서 익일 배송 가능으로 배송 행사 수정
  1 files changed, 1 insertions(+), 1 deletions(-)
```

우리는 마스터 브랜치에 수정했기 때문에 layout_a와 같은 다른 브랜치로 다시 이동하면 변경 사항을 볼 수 없다. 배송에 관한 수정 사항은 매우 중요한 내용이니, layout_a에도 반영하자.

```
$ git checkout layout_a
$ git merge master
```

이 명령어는 layout_a에는 수정되지 않았지만, 마스터 브랜치에서 수정된 내용을 layout_a 브랜치에 반영해 준다.

다음은 layout_b 브랜치를 만든다. 주의할 점은 layout_a 브랜치가 아닌 master 브랜치에서 layout_b 브랜치를 만들고 싶으니, 마스터 브랜치로 돌아간 후 layout_b 브랜치를 만든다.

```
$ git checkout master
$ git checkout -b layout_b
```

이번엔 〈h1〉 태그의 내용을 '레이아웃 B'로 수정한다. 저장하고 변경 사항을 반영하자.

```
$ git commit -a -m "<h1> 값을 레이아웃 B로 변경"
```

방금 만든 layout_b 브랜치에는 products.html 파일과 about_us.html 파일을 추가한다. 커밋하기 위해 먼저 만든 파일들을 추가하자.

```
$ touch products.html
$ touch about_us.html
$ git add .
```

이제 git status 명령어를 실행하면 두 개의 파일이 새로 추가된 것을 확인할 수 있다.

```
# On branch layout_b
# Changes to be committed:
#   (use "Git reset HEAD <file>..." to unstage)
#
#     new file: about_us.html
#     new file: products.html
#
```

이 파일들을 커밋하자.

```
$ git commit -a -m "빈 products.html과 about_us.html 추가"
```

이제, products.html의 〈h1〉 부분에 '판매 중인 제품들'이라는 글을 추가하자.

한참 작업을 하는 동안, 사장님에게서 메일이 왔다. "홈페이지에 당일 배송이 가능하다고 다시 수정해야겠어. 대형 택배 회사와 계약이 성사됐어. 최대한 빨리 수정을 부탁하네!" 당장 수정해야 할 것 같다. 그러나 layout_b에 작업 중이라 중간에 마스터 브랜치로 이동할 수는 없다.

이럴 때는 Git의 stash 명령어를 사용하자. Stash 명령어는 변경 사항을 저장하지만 커밋하지는 않는다. 변경 사항을 저장했으니, 브랜치 간 이동도 할 수 있다. 무엇인가를 커밋 하지는 않고 저장만 하고 싶을 때 쓰자.

커밋을 왜 이렇게 자주하나요?

커밋을 프로젝트를 위한 스냅샷이나 복구 지점이라 생각해 보자. 커밋을 많이 할수록, Git은 더 강력하고 유연해진다. 만약 기능별로 잘게 나누어 커밋을 하면, 다른 브랜치의 커밋을 현재 작업 중인 브랜치로 가져오거나, 그 반대로 할 수도 있다. 만약 자잘하게 쪼갠 커밋들이 지저분하게 보인다면, 기능을 완성한 후에 rebase 명령어를 이용해 커밋들을 합칠 수도 있다.

```
$ git stash
```

git status를 실행하면 커밋을 해야 할 사항이 없다고 나온다. 저장이 제대로 된 것 같다. 이제 마스터 브랜치로 이동하자.

```
$ git checkout master
```

이제 마스터 브랜치의 index.html에 배송 정보를 바꾸고 커밋 한다.

```
$ git commit -a –m "배송 시간 바꿈"
```

git checkout layout_b 명령어로 layout_b 브랜치로 돌아가 우리가 무엇을 저장해 두었는지 다시 확인해 보자. git stash list 명령어로 알 수 있다.

```
$ git stash list
stash@{0}: WIP on layout_b: f8747f4 빈 products.html과 about_us.html 추가
```

지금 products.html 파일을 열면 분명히 수정했는데, 파일이 비어 있다. 우리가 만든 변경 사항은 저장했지만, 숨어 있기 때문이다. 다시 보이게 하자.

```
$ git stash pop
```

다시 products.html 파일을 열어 보면 마스터 브랜치로 가기 전에 만들었던 〈h1〉 태그가 있을 것이다.

layout_a와 layout_b 브랜치에 이것저것 기능을 추가해 데모를 만들었다. 데모를 본 사장님은 layout_b가 더 좋다며 마스터 브랜치에 반영해 서비스에 적용하자고 한다. layout_b 브랜치를 마스터 브랜치에 반영하자.

```
$ git checkout master
$ git merge layout_b
$ git commit -a -m " layout_b를 합침"
```

전통적인 버전 컨트롤 시스템에서는 보통 한 번 만든 브랜치는 저장소에 그대로 둔다. 브랜치를 삭제하면 브랜치 안의 내용까지 지워지기 때문이다. Git은 브랜치와 태그 모두 커밋한 내용을 참조한다는 점이 다르다. Git에서는 브랜치를 삭제해도 커밋한 내용은 삭제되지 않고, 내용에 대한 참조만 삭제된다.

변경 사항을 마스터 브랜치에 반영했기 때문에 개발을 위해 만들었던 브랜치를 삭제할 수 있다. git branch 명령어를 이용해 무엇을 가지고 있는지 먼저 보자. 우리가 마스터 브랜치에 있고 layout_a와 layout_b 브랜치가 보인다. 둘을 다음의 명령

어로 삭제하자.

```
$ git branch -d layout_a
$ git branch -d layout_b
```

Git은 layout_a 브랜치의 변경 사항이 마스터 브랜치로 합쳐져 있지 않다는 것을 알려줄 것이다. 이때 -D 옵션을 이용하면 무시하고 강제로 지울 수 있다.

원격 저장소와 작업하기

지금까지는 로컬 저장소에서만 작업을 했다. 지금도 로컬의 코드를 관리하기에는 충분하지만 원격 저장소를 가지면 다른 사람들과 함께 일할 수 있을 뿐 아니라 안전하게 코드를 두 곳에 저장할 수 있다.

원격 Git 저장소를 '레시피 37. 가상머신 만들기'에서 만든 개발용 가상머신에 만들자. SSH 키를 만들면 로그인하거나 파일을 보낼 때마다 비밀번호를 입력해야 하는 수고를 덜 수 있다. SSH 키를 만들고 서버에 키를 두면 원격 저장소에 파일을 저장하고 싶을 때마다 비밀번호 없이 빠르게 인증받을 수 있다.

SSH 키는 두 부분으로 되어 있다. 우리가 보관할 개인키와 서버에 올릴 공개키다. 서버에 로그인할 때, 서버는 우리의 키가 권한이 있는지 확인하고, 우리 로컬 시스템은 로그인하고자 하는 사람이 누구인지 공개키를 개인키와 비교하여 증명한다. Git으로 로그인 하는 동안 서로 확인하는 과정이 투명하게 처리된다.

계속 진행하려면 여러분의 시스템에 SSH 키가 있는지 먼저 확인해야 한다. 디렉터리를 ~/.ssh로 바꾸자. 폴더가 존재하지 않는다는 메시지를 받으면 키를 생성해야 한다. 만약 폴더가 있고 폴더 안에 id_rsa, id_rsa.pub 파일이 있다면 키를 가지고 있는 것이니, 다음 단계를 진행해도 좋다.

새 SSH 키를 ssh-keygen 명령어로 만들자. 키에 코멘트로 넣을 이메일을 입력해 실행한다.

```
$ ssh-keygen -t rsa -C "sunphiz@gmail.com"
```

코멘트는 공개키가 서버에 올라갔을 때 누구의 공개키인지 쉽게 확인할 수 있게 해 준다.

ssh-keygen 프로그램이 SSH 키를 어디에 저장할지 물을 것이다. 그냥 엔터키를 눌러 기본 폴더에 저장한다. 그리고 passphrase를 입력하라고 물을 것이다. 키에 보

안 레이어를 추가하는 것인데, 이번에는 공백으로 남긴다. 그냥 엔터키를 다시 입력하자.

이제 우리 키가 만들어졌으니, 가상머신에 그 키를 추가하자. 로컬의 공개키를 서버의 authorized_keys 파일에 연결한다. 이것으로 가상머신에 접근할 권한이 생겼다.

```
$ cat ~/.ssh/id_rsa.pub | ssh webdev@192.168.1.100 \
"mkdir ~/.ssh; cat >> ~/.ssh/authorized_keys"
```

이 명령어를 실행하면 서버가 합법적인 요청인지 확인하기 위해 비밀번호를 묻는다. 명령어 실행이 끝나면 SSH로 가상머신에 접근해 보자.

```
$ ssh webdev@192.168.1.100
```

이번에는 비밀번호를 묻지 않을 것이다.

이제 가상머신에 로그인 했으니, 우분투의 패키지 매니저를 이용해 Git을 서버에 설치하자.

```
$ sudo apt-get install git-core
```

이제 가상머신에 저장소를 만들 수 있다. 저장소는 사실 그냥 폴더인데 일반적 .git 확장자로 구분하기 쉽게 해 준다. 폴더 안에서 git 명령어와 bare 스위치를 사용해 폴더를 초기화할 수 있다.

```
$ mkdir website.git
$ cd website.git
$ git init --bare
```

원격 컴퓨터에 저장소를 만들었으니, exit 명령어로 가상머신에서 로그아웃 하자.

컴퓨터로 돌아와서, 원격 저장소의 주소를 추가하고 마스터 브랜치를 올리자.

```
$ git remote add origin ssh://webdev@192.168.1.100/~/website.git
$ git push origin master
```

이제 다른 사람들과 함께 작업할 준비가 되었다. 새로운 기능에 대해 new_feature라는 이름으로 브랜치를 만든 후에 디자인 구현을 그 브랜치에서 작업한다. 디자인 작업이 끝나면 브랜치를 원격 저장소에 올릴 수 있다.

```
$ git checkout -b new_feature
$ git push origin new_feature
```

이제 우리 브랜치를 원격 저장소에 올렸으니, git branch 명령어를 이용해 확인해 보자.

```
$ git branch -r
```

브랜치 목록이 나올 것이다. layout_a, layout_b 브랜치는 원격 저장소로 올리기 전에 지워버렸기 때문에 목록에 없다.

```
origin/HEAD -> origin/master
origin/new_feature
origin/master
```

다른 개발자가 원격 Git에 저장된 전체 프로젝트의 복사본을 가질 수 있다. 복사본에서 new_feature 브랜치를 체크아웃할 수도 있다. 마지막으로 원격 저장소에서 로컬 브랜치로 파일을 당겨와 new_feature 브랜치의 파일들이 최신인지 확인할 수 있다.

```
$ git clone ssh://webdev@192.168.1.100/~/website.git
$ git checkout -b new_feature
$ git pull origin new_feature
```

드디어 원격 저장소에 작업할 수 있는 환경이 준비되었다. 이 레시피의 앞부분에 로컬에서 저장했던 것처럼 Git을 이용하면 같은 파일에 같이 작업하고 변경 사항을 합칠 수 있다.

더 보기

이 레시피에서 Git의 기본적인 기능을 살펴보았다. 여러분은 벌써 이 레시피에서 소개되지 않은 다른 기능들도 보고 있는지도 모른다. 이 레시피에서는 텍스트 파일만 작업했지만, Git은 모든 종류의 파일을 관리할 수 있도록 지원한다. 심지어 포토샵 파일도 관리할 수 있으므로 만든 디자인을 여러 가지 버전으로 관리할 수 있다.

파일의 이전 버전을 가져오는 법은 꼭 살펴 보자. 사장님이 처음엔 마음에 들어 하지 않았던 layout_a 브랜치가 마음에 든다며 다시 한 번 보자고 할지도 모르니까 말이다.

다른 사람과 함께 Git을 이용해 오픈소스 프로젝트를 진행할 수도 있다.