* **Chapter 1. Git과 Github의 이해**
* **Git**
  + Git은 분산 버전 관리 시스템이다.
    - Git으로 프로젝트를 개발하는 사람은 모두 현재 상태의 파일뿐만 아니라 그 프로젝트의 전체 이력을 가지고 있게 된다는 뜻이다.
    - 버전관리 시스템 : 파일의 변경 내역을 계속 추적하도록 개발된 소프트웨어
* **GitHub**
  + GitHub는 Git 저장소(repository)를 업로드 할 수 있는 웹사이트를 말한다.
  + GitHub는 다른 사람들과의 협업을 매우 용이하게 해준다.
    - 리포지토리를 공유할 수 있는 중앙저장소
      * 웹 기반 인터페이스
      * forking
      * pull requests
      * issues
      * wikis
  + Github는 위와 같은 기능을 제공하여 팀원들과 보다 효율적으로 변경안을 구체화하고 토론하며 검토할 수 있게 해준다.

**Git의 장점**

본질적으로 깃의 사용은 1인 프로젝트를 진행하는 사람에게도 다양한 가치가 있다.

* 변경취소 가능 : 실수 했을 때 구 버전의 작업 파일로 돌아갈 수 있음.
* 모든 변경에 대한 완벽한 이력 History : 짧게는 하루, 길게는 1년 전에 프로젝트가 어떤 형태였는지도 알 수 있다.
* 변경한 이유를 기록 : 협업을 하다보면(때로는 내가 작성한 코드라할지라도) 왜 변경했는지 모르겠을 때가 많다. 이때 Git의 커밋 메시지(commit message)를 이용하면 변경한 이유를 쉽게 기록할 수 있으며, 추후에 참조할 수 있다.
* 변경에 대한 확신 : 이전 버전으로의 복귀가 쉽기 때문에 자신을 원하는 대로 다 변경가능한다. 잘 안되면 언제든 이전 버전으로 복귀하면 된다.
* 여러 갈래의 히스토리(History) : 콘텐츠의 변경 내용을 테스트해보거나 기능을 독립적으로 실험해보기 위해 별도의 브런치(branch)를 생성할 수 있다. 완료되면 변경 내용을 마스터 브랜치(Master branch)로 병합할 수 있고, 잘 작동하지 않을 경우 삭제 가능하다.
* 충돌 해결 능력 : Git을 이용하면 여러사람이 동시에 같은 파일을 작업할 수 있다. Git은 대개 자동으로 변경사항을 병합할 수 있다. 그렇지 못할 경우에 충돌이 무엇인지 알려주고 이를 해결하기 쉽게 해준다.
* 독립된 히스토리(History) : 여러 사람들이 다른 브랜치(branch)에서 작업이 가능하다. 기능을 독립적으로 개발하고, 완료되었을 때 그 기능을 병합할 수 있다.

**왜 GitHub를 사용하는가?**

**GitHub는 단순히 Git 저장소를 제공하는 것 이상의 가치를 제공한다.**

* 기록요구 : Issue(이슈)를 사용해 버그를 기록하거나 개발하고 싶은 새로운 기능을 구체화할 수 있다.
* 독립된 히스토리(History)에 대한 협력 : branch와 pull requests를 이용해 다른 브랜치 또는 기능에 협력할 수 있다.
* 진행 중인 작업 검토 : pull requests 목록을 통해 현재 무슨 작업이 진행되고 있는지 모두 볼 수 있다. 그리고 특정 pull request를 클릭하여 최근의 변경 내용과 변경에 관한 모든 논의 내용을 볼 수 있다.
* 팀의 작업 진척 상황 확인 : 펄스(pulse)를 훑어보거나 commit histroy를 살펴보면 팀의 작업 진척 상황을 알 수 있다.

**Git & Github 주요개념**

* commit : 하나 또는 다수의 파일에 변경 내용을 저장할 때마다 새로운 commit 생성한다.

ex) ”이 변경 내용을 commit하고 이를 GitHub로 push 합시다.”

* commit message : commit을 생서할 때 마다 왜 변결을 했는지에 대한 이유를 설명하는 메시지를 제공해야한다. 이 commit message는 변경을 한 이유를 추후에 이해할 때 매우유용하다.

ex) ”새로운 SEC 가이드라인에 대한 수잔의 의견을 commit 메시지에 꼭 넣으세요.”

* branch : 테스트를 해보거나 새로운 기능을 개발하기 위해 사용할 수 있는 따로 떨어진 독립적인 commit들을 말한다.

ex) “새로운 검색기능을 구현하기 위해 branch를 생성합시다.”

* master branch : 새로운 Git 프로젝트를 만들 때마다 ‘master’라고 불리는 기본 branch가 생성된다. 배포할 준비가 되면 작업이 최종적으로 마무리되는 branch이다.

ex) “master로 바로 commit 하면 안 된다는 것을 기억하세요”

* feature or topic branch : 새로운 기능을 개발할 때마다 작업할 브랜치를 만드는데, 이를 feature branch라고 한다.

ex) “feature branch가 너무 많습니다. 이들 중 하나나 두개를 완료해서 출시하는데 주력합시다.”

* release branch : 직접 QA(품질보증) 작업을 하거나 고객의 요구로 구 버전의 소프트웨어를 지원해야 한다면 모든 수정이나 업데이트를 위한 장소로 release branch가 필요하다. feature branch와 release branch는 기능적 차이는 없지만, 팀월들과 프로젝트에 대해 이야기할 때 확연히 구별할 수 있어 유용하다.

ex) “release branch 전체에 대한 보안 버그를 수정해야 합니다.”

* Merge : 병합(merge)은 한 branch에서 완성된 작업을 가져와 다른 branch에 포함하는 방법이다. 흔히 feature branch를 master branch로 merge한다.

ex) “‘내 계정’기능이 훌륭합니다. 배포할 수 있게 master로 merge 해줄 수 있습니까?”

* Tag : 특정 이력이 있는 commit에 대한 참조. 어떤 버전의 코드가 언제 배포(release)되었는지 정확히 알 수 있도록 제품 배포 기록에 가장 자주 사용된다.

ex) “이번 배포판에 tag를 달고 출시합시다.”

* Check out : 프로젝트 history의 다른 버전으로 이동해 해당 시점의 파일을 보기 위해 ‘checkout’ 한다. 가장 일반적으로 branch에서 완료된 작업을 모두 보기 위해 branch를 checkout하지만, commit도 checkout할 수 있다.

ex) ”최종 릴리즈 태그(release tag)를 checkout 해주시겠어요? 제품에 버그가 있어서 검증하고 수정해야 합니다.”

* Pull request : 원래 pull request는 branch에서 완료된 작업을 다른 사람이 리뷰하고 master로 merge요청을 하기 위해 사용되었다. 요즘은 개발 가능한 기능에 대한 논의를 시작하는 초기 단계에서 자주 사용한다.

ex) “다른 팀원들이 어떻게 생각하는지 알 수 있게 새로운 투표 기능에 대한 pull request를 만드십시오.”

* Issue : GitHub는 기능에 대해 논의하거나 버그를 추적하거나 혹은 두 가지 경우 모두 사용될 수 있는 Issue라는 기능을 갖고 있다.

ex) “당신이 옳아요, 아이폰에서는 로그인이 도지 않네요. 버그를 검증하기 위한 단계를 기록하면서 GitHub에 issue를 생성해 주시겠습니까?”

* Wiki : Ward Cunningham이 최초로 개발하였다. wiki는 링크들 간을 연결해 간단하게 웹페이지를 만드는 방법이다. GitHub 프로젝트는 문서 작성에 자주 Wiki를 사용한다.

ex) “복수의 서버에서 작동하게끔 프로젝트 환경 설정 방법을 설명하는 페이지를 추가해 주시겠습니까?”

* Clone : 종종 로컬로 작업하기 위해 프로젝트 복사본을 GitHub에서 다운로드 한다. repository를 사용자의 컴퓨터로 복사하는 과정을 복제(cloning)라고 한다.

ex) “repository를 clone하고, 버그를 수정한 다음 오늘 밤 수정본을 다시 GitHub로 push 해주시겠습니까?”

* Fork : 때로는 프로젝트를 직접 변경하는 데 필요한 권한을 보유하지 못할 때가 있다. 잘 알지 못하는 사람이 작성한 오픈소스 프로젝트이거나, 회사에서 작업을 같이 많이 하지 않는 다른 그룹이 작성한 프로젝트일 수도 있다. 그러한 프로젝트를 변경하고 싶다면 먼저 GitHub의 사용자 계정에 프로젝트 복사본을 만들어야한다. 그 과정을 repository를 fork한다고 한다. 그런다음 복제(clone)하고, 변경하고, pull request를 이용해 원본 프로젝트에 변경 내용을 반영할 수 있다.

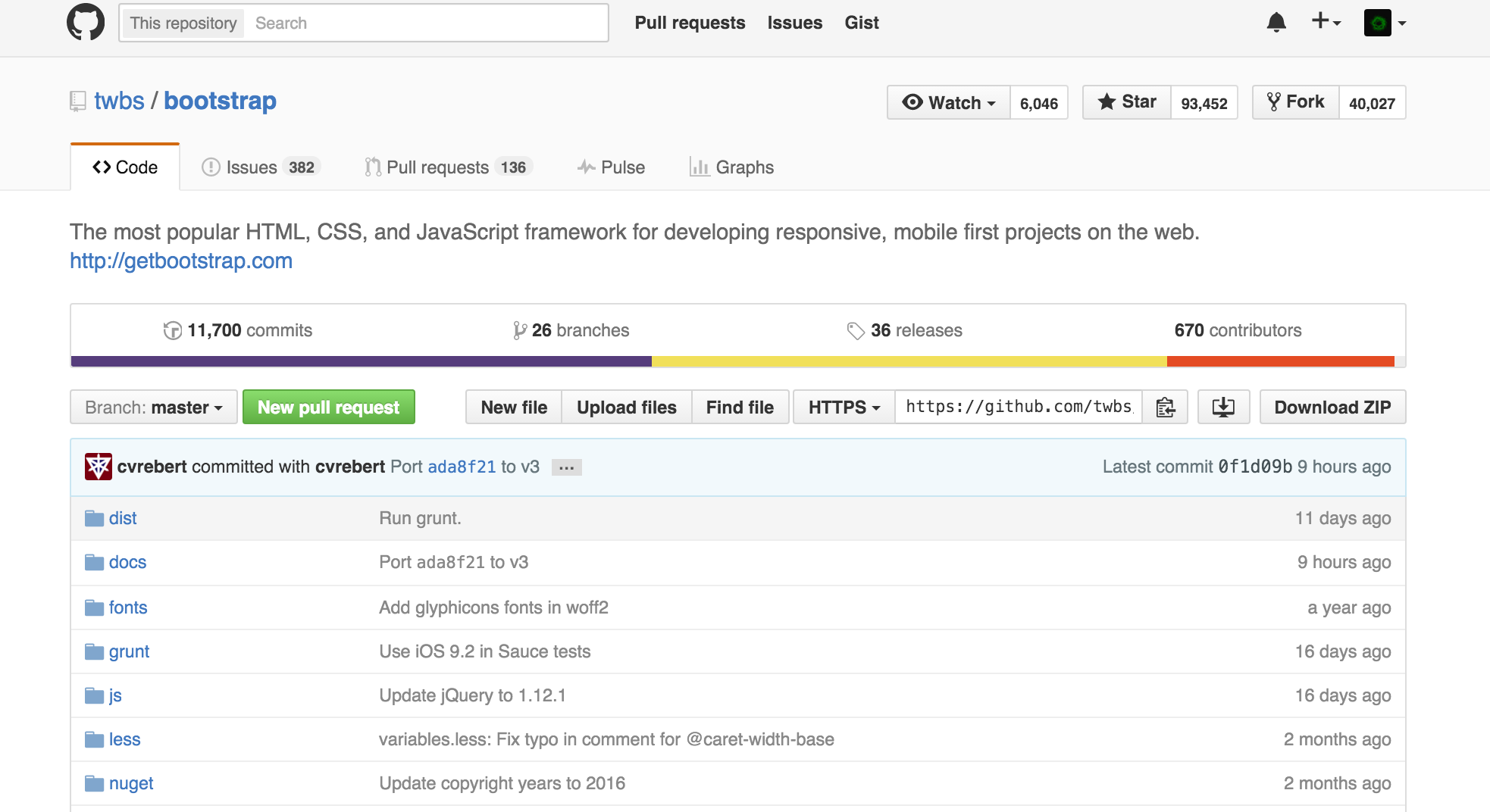
ex) “홈페이지 마케팅 원고를 당신이 어떻게 재작성 했는지 보고 싶습니다. repository를 fork하고 수정안과 함께 pull request를 제출해주세요.”

**Chapter 2. 프로젝트 보기**

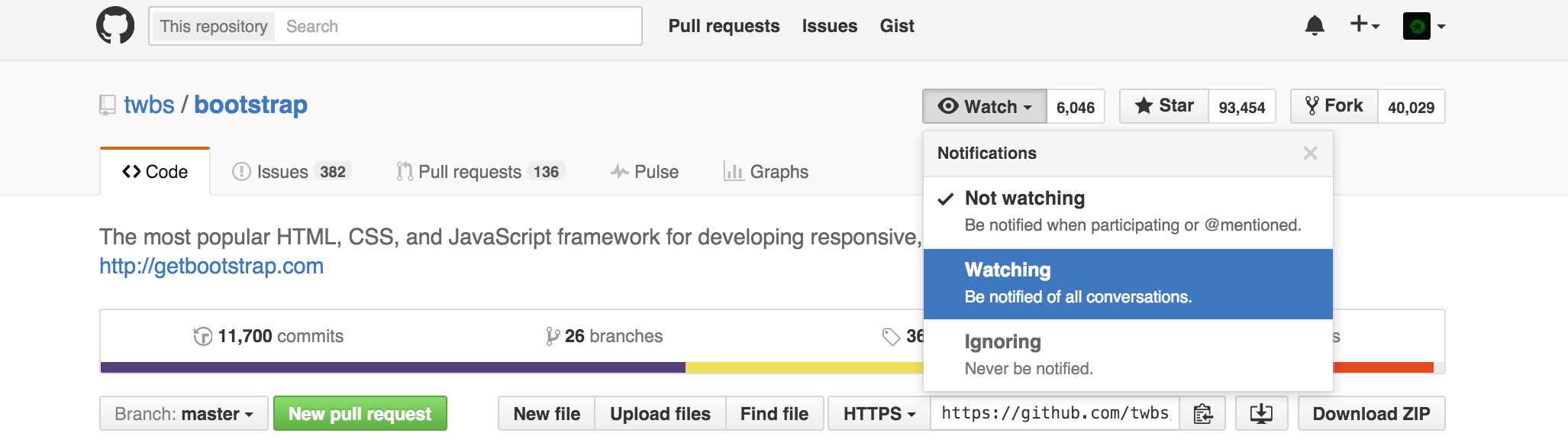
이 장에서는 프로젝트의 상태를 확인할 수 있는 방법을 살펴볼 것이다. 잘 알려진 부트스트랩(Bootstrap)의 오픈 소스 프로젝트를 예제로 사용하겠다.

**2.1 프로젝트 페이지소개**

부트스트랩은 개발자들이 웹앱을 빠르게 개발할 수 있게 해주는 프로젝트이다. GitHub의 프로젝트 페이지로 이동한다. 홈페이지에 아주 많은 정보가 있지만, 먼저 가장 중요한 요소들을 살펴보자.

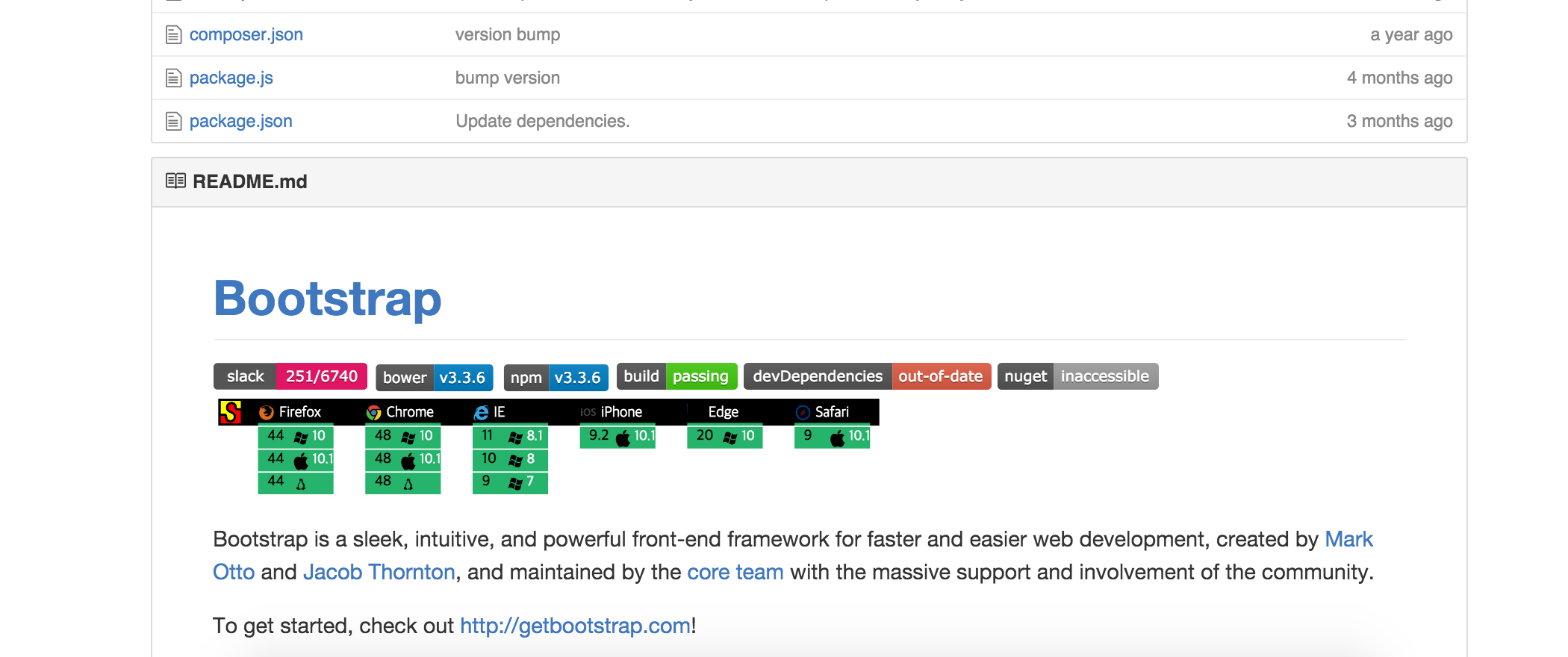


위 사진 왼쪽 위를 보면 twbs/bootstrap라고 적혀있는데, 이는 twbs라는 사용자가 bootstrap이라는 프로젝트를 소유하고있다는 것을 뜻한다.

GitHub를 그동안 오래 보아왔지만, 눈, 별, 나무줄기(?)형태의 아이콘이 정확히 어떤의미인지는 모르고있었다.

* ‘눈’은 이 프로젝트에 새로운 변화가 생길 때 마다 알림을 받겠다는 뜻이다. ‘구독’정도의 의미로 이해하면 좋겠다.
* ‘별’은 즐겨찾기 또는 별점을 부여했다는 것이고,
* ‘나무줄기’는 아이콘에도 나와있지만, 이 프로젝트를 fork한 수를 의미한다.

**2.2 README.md파일 보기**

프로젝트의 루트에 README.md라는 파일이 있으면 그 파일의 콘텐츠가 프로젝트 홈페이지의 폴더와 파일 목록 바로 아래에 표시된다.

이 파일은 프로젝트에 대한 소개와 협력자들에게 유용한 추가 정보(소프트웨어를 어떻게 설치하는지, 자동화된 테스트를 어떻게 실행하는지, 코드를 어떻게 사용하는지, 프로젝트에 어떻게 기여할 수 있는지 등)를 제공한다.

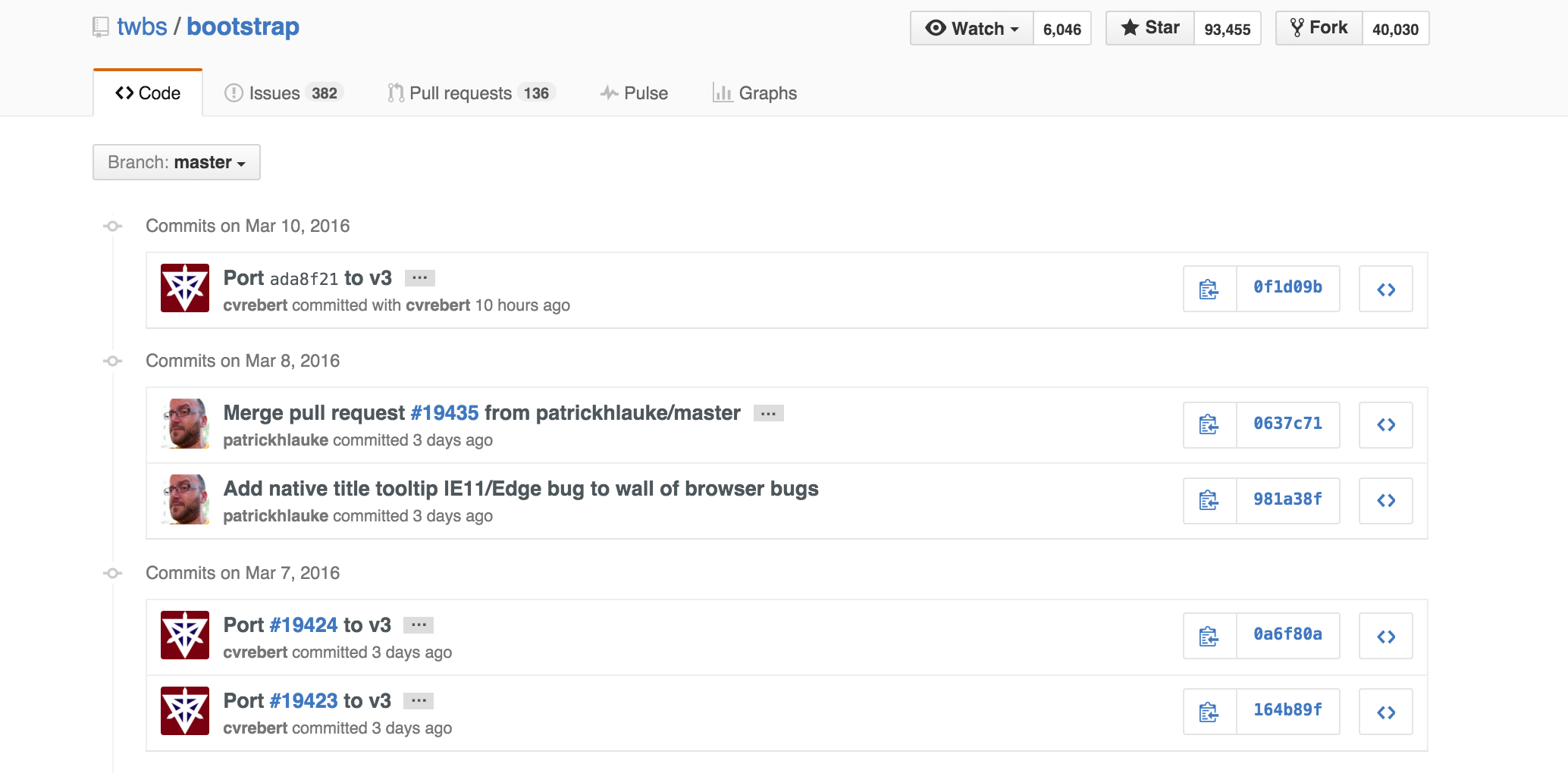
요즘들어 README.md 파일은 badge도 자주 포함하는데, badge는 자동화된 test suite같이 프로젝트의 현 상태를 알려주기 위해 사용되는 이미지다.

[그림2–1–2]에서는 부트스트랩이 동작하는 두 개의 다른 프로젝트의 버전을 표시하고 있다. 또한 자동화된 테스트를 통과했고, 종속관계(dependencies)가 업데이트 되었다는 것과, 브라우저와 운영체제의 버전을 보여주고있다.

**2.3 Commit history 보기**

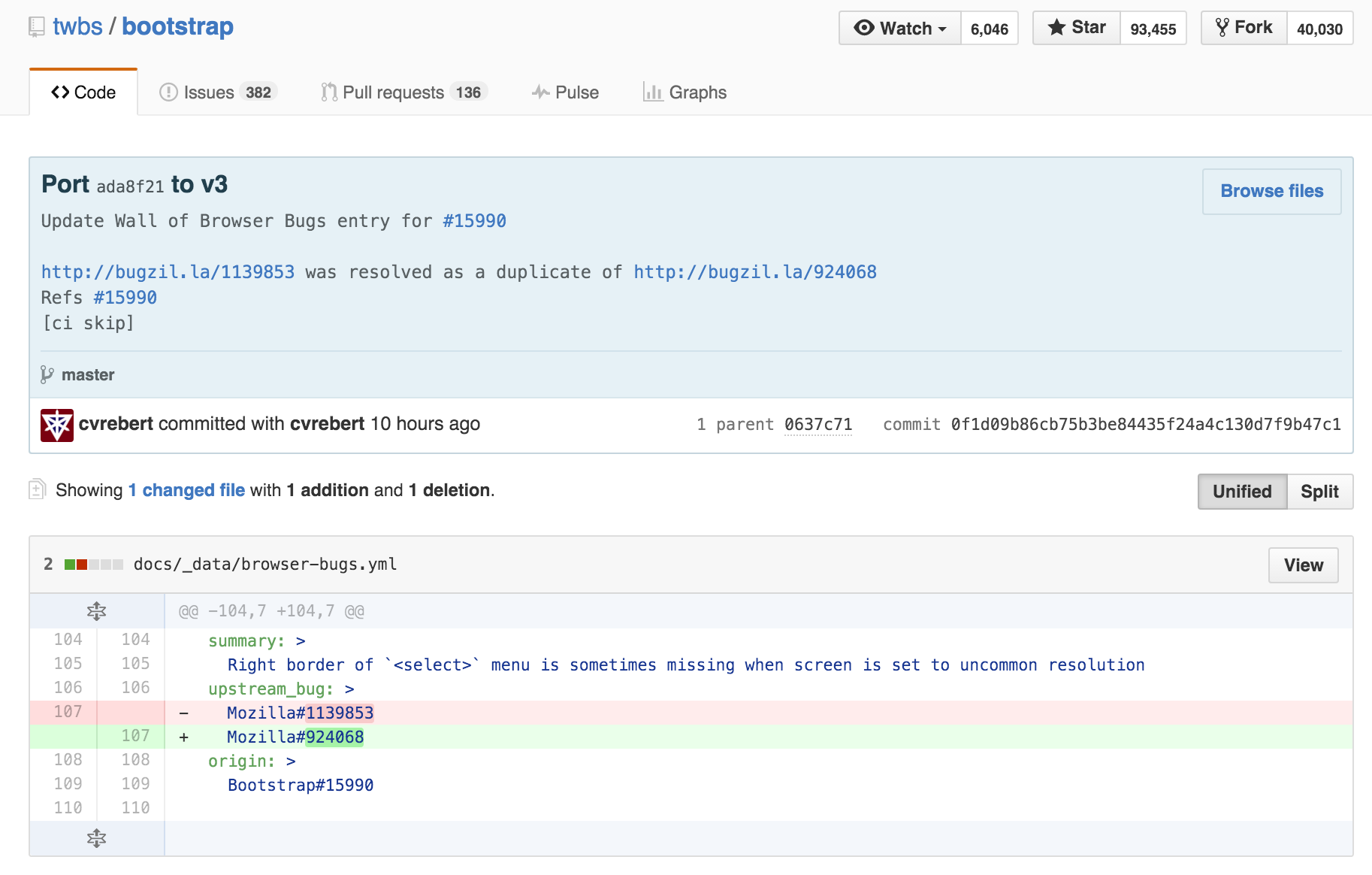
다음으로 살펴 볼 commit history는 어떤 특정 branch에서 작어빙 완료되었을 때 가장 최근 작업이 무엇인지 알아보는 좋은 방법이다.

이를 확인하기위해 GitHub의 부트스트랩페이지로 이동해 “9,448 commits”링크를 클릭한다.

(물론, 부트스트랩 프로젝트 페이지에 접속한 시점을 기준으로 commit 수가 달라질 것이다.)

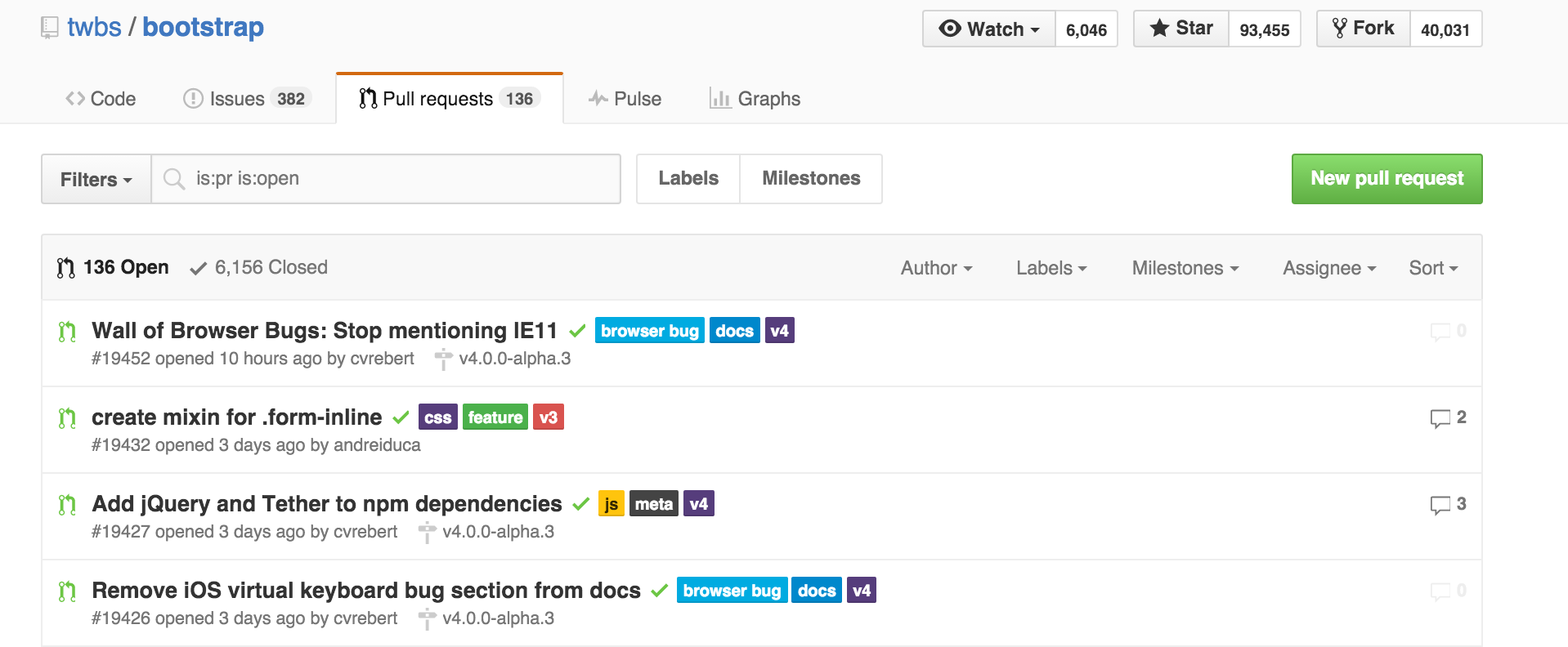
가장 최근의 작업이 목록 위쪽으로 오는 순서로 commit 목록이 표시된다.([그림 2–3]참조)

각 commit을 클릭하면, 변경이 된 이유를 설명하는 commit message가 표시된다.



commit message 아래로 commit의 일부분으로 추가되거나 삭제 또는 수정된 각각의 파일을 볼 수 있다. 삭제된 콘텐츠는 빨간색으로, 추가된 콘텐츠는 녹색으로 표시된다.

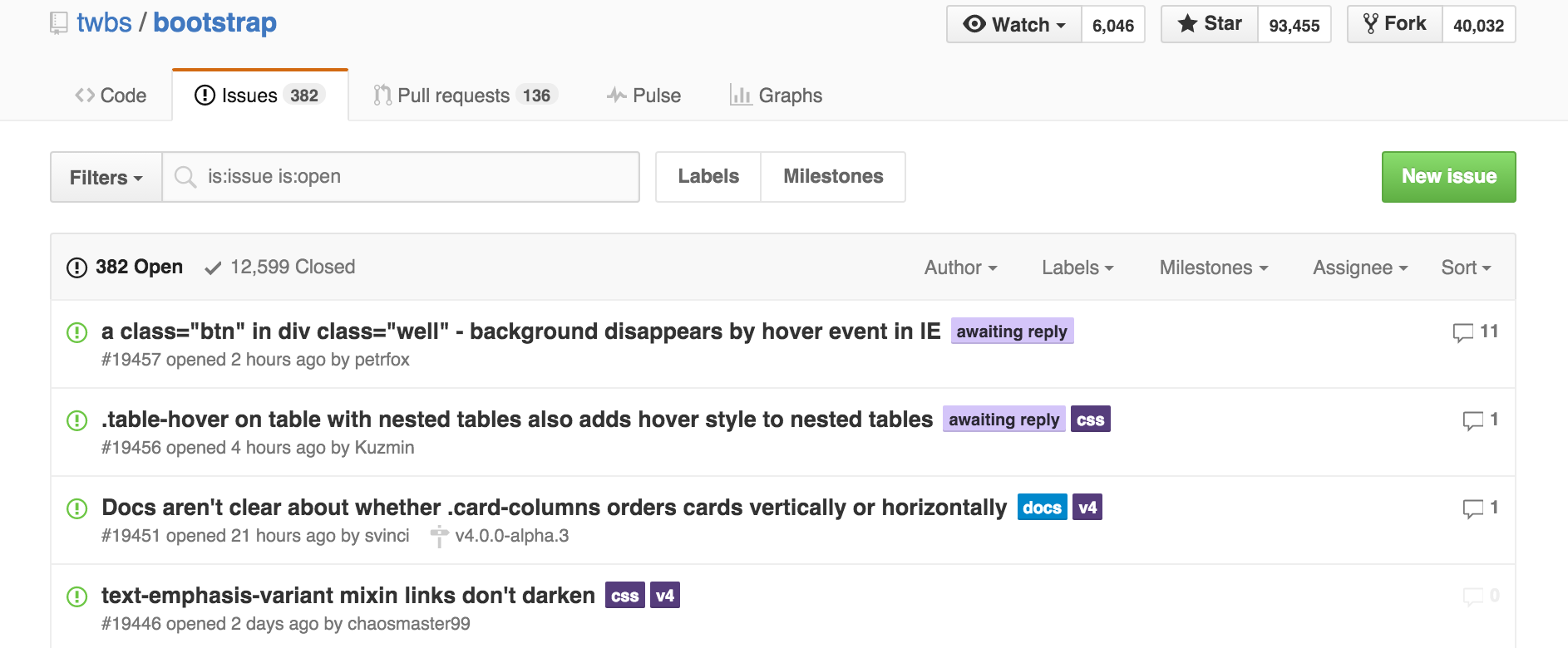
**2.4 Pull Request 보기**



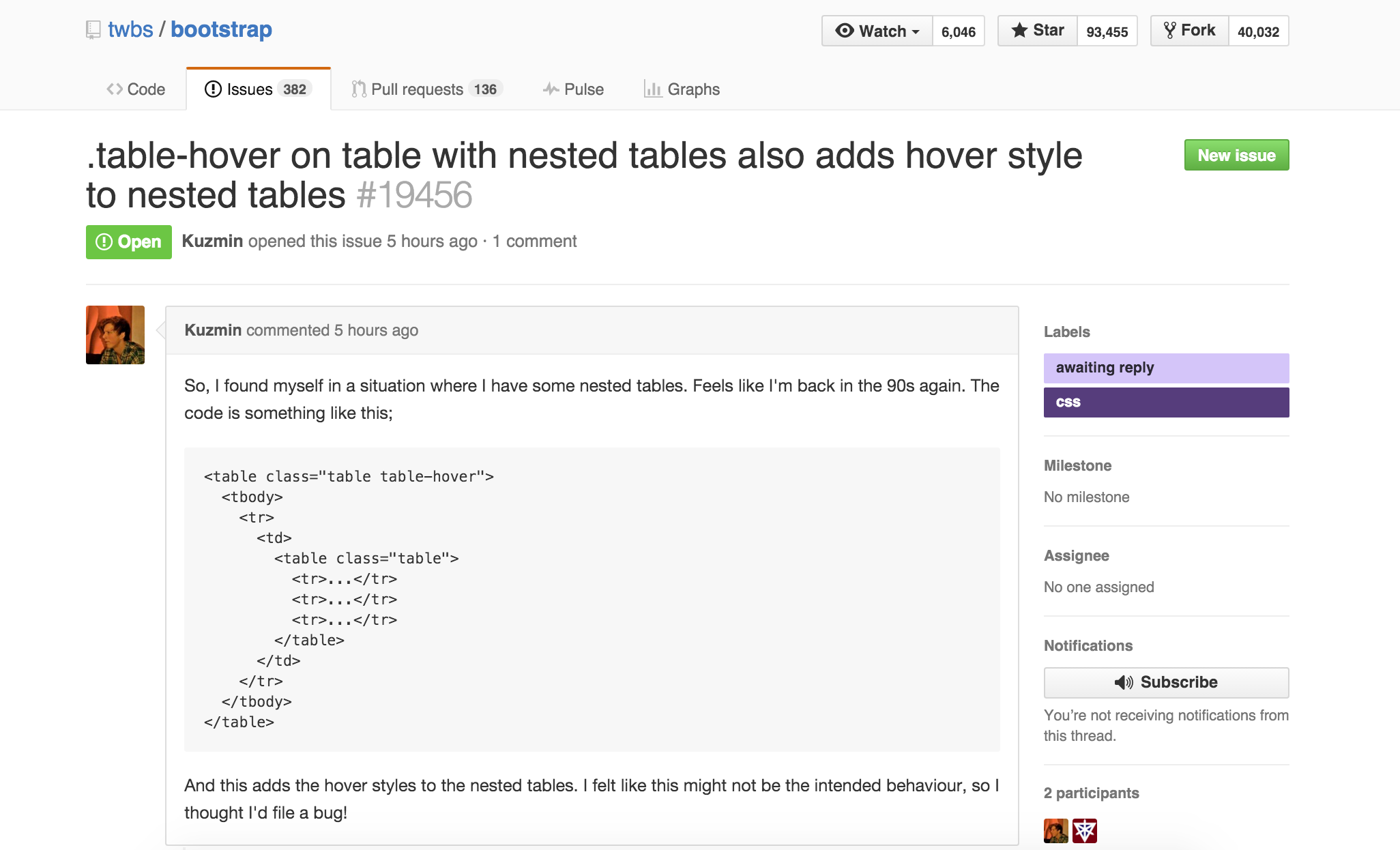
* pull request는 현재 진행 중인 작업이 무엇인지 알 수 있게 해준다. 프로젝트 페이지 화면에서 오른쪽위의 pull request를 클릭하면 공개된 pull request 목록이 표시된다. 이것은 사람들이 현재 작업하고 있는 기능이나 수정사항을 나타낸다.([그림 2–5]참조)

pull request를 살펴보면 사람들이 현재 무슨 작업을 하고 있으며, 버그 수정을 하든 기능 개발을 하든 각가의 변경 사항에 대해 어떤 역할을 하고 있는지 알 수 있다.

**2.5 Issue 보기**

pull request로 현재 진행되고 있는 버그 수정과 기능을 알 수 있다면, issue를 통해서는 프로젝트에 필요한 작업을 넓은 관점에서 볼 수 있다.

pull request는 주로 issue에 링크되어 있지만, 아무도 작업을 시작하지 않아 pull request가 없는 issue또한 존재한다. issue 목록을 보기 위해 링크를 클릭하면 모든 공개 issue목록이 자동으로 나타난다([그림 2–7] 참조).

누군가 Bootstrap 사용 중 겪은 어려움을 적어 공유하는 모습이다.

GitHub 페이지를 이용해 협력하는 방법에 대해 살펴보기

#Github 협업 #1 Branch

**브랜치로 커밋하기**

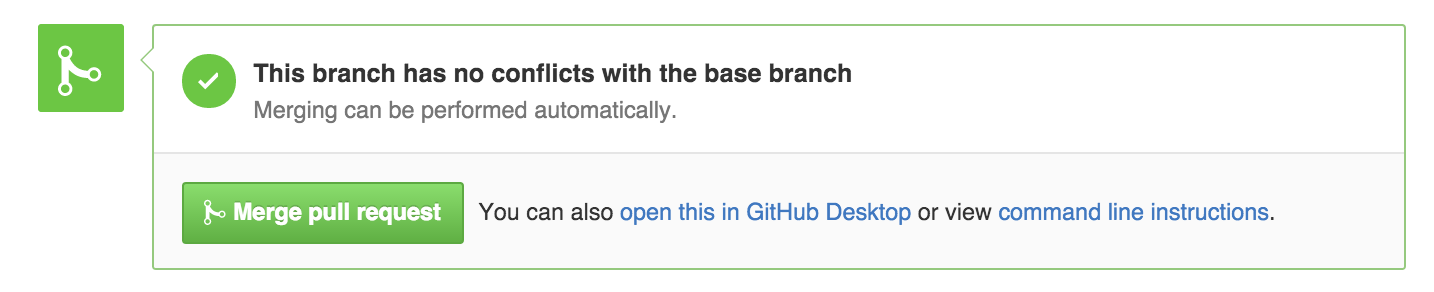
README.md 파일 확장

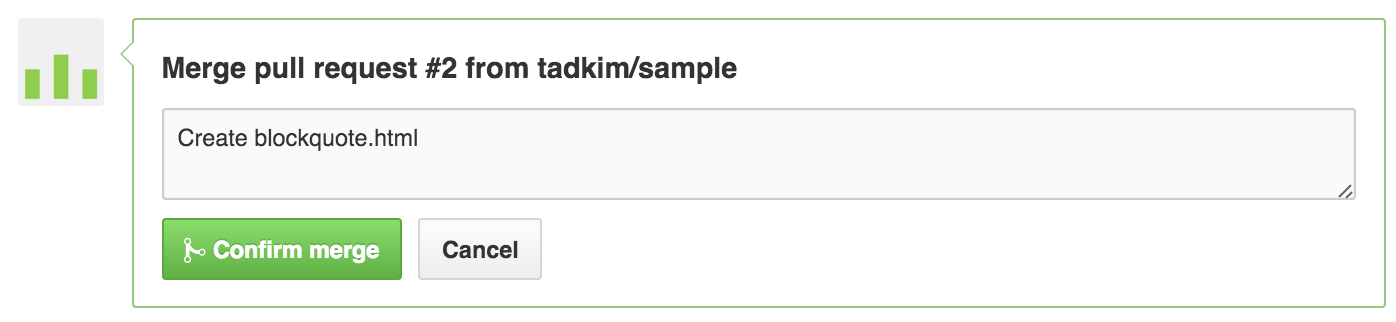
1. 브랜치 생성
2. 브랜치에서 Pull requset 하기 2-1. 프로젝트 페이지 우측 Pull Request탭의 New pull request를 클릭한다.이때 바로 생성이 되지 않는 것은, Github가 어느 브랜치에 생성해야할지를 몰라서이다. 2-2. 왼쪽에 base:master가 있으면 올바른 것. pull request를 생성하고 이것이 받아들여지면 마스터 브랜치로 병합된다. 2-3. 이어 compare:master 버튼을 클릭해 어떤 브랜치에 pull request를 생성해야하는지 지정해주어야 한다. compare:브랜치는 마스터 브랜치로 병합되었으면 하는 브랜치이다.

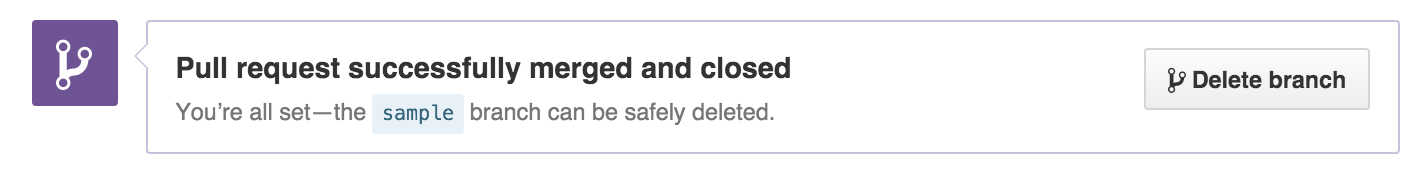
######브랜치 병합하기

**브랜치 병합은 누가 하는가**? pull request를 생성한 사람이 병합할지, 아니면 생성자가 아닌 다른 사람이 할지에 대한 질문이 많다. 정답은 없지만 다음 사항들을 고려해보자.

* 보통 pull request는 생성한 사람이 병합하도록 조언한다. 대부분의 경우 pull request를 생성한 사람이 해당 코드에 대해 가장 잘 알고 있는 사람이기 때문이다. 단, 반드시 다른 사람들로부터 :+:[^1]를 최소한 둘 이상 받았을 때 한다.
* 많은 회사에서는 pull request를 생성한 사람은 병합 할 수 없다는 원칙을 세운다. 그 이유는 아무 피드백 없이 pull request를 생성하고 병합하지 않도록 하기 위함이다. 의도는 좋지만 그다지 이상적인 권고사항은 아니다.







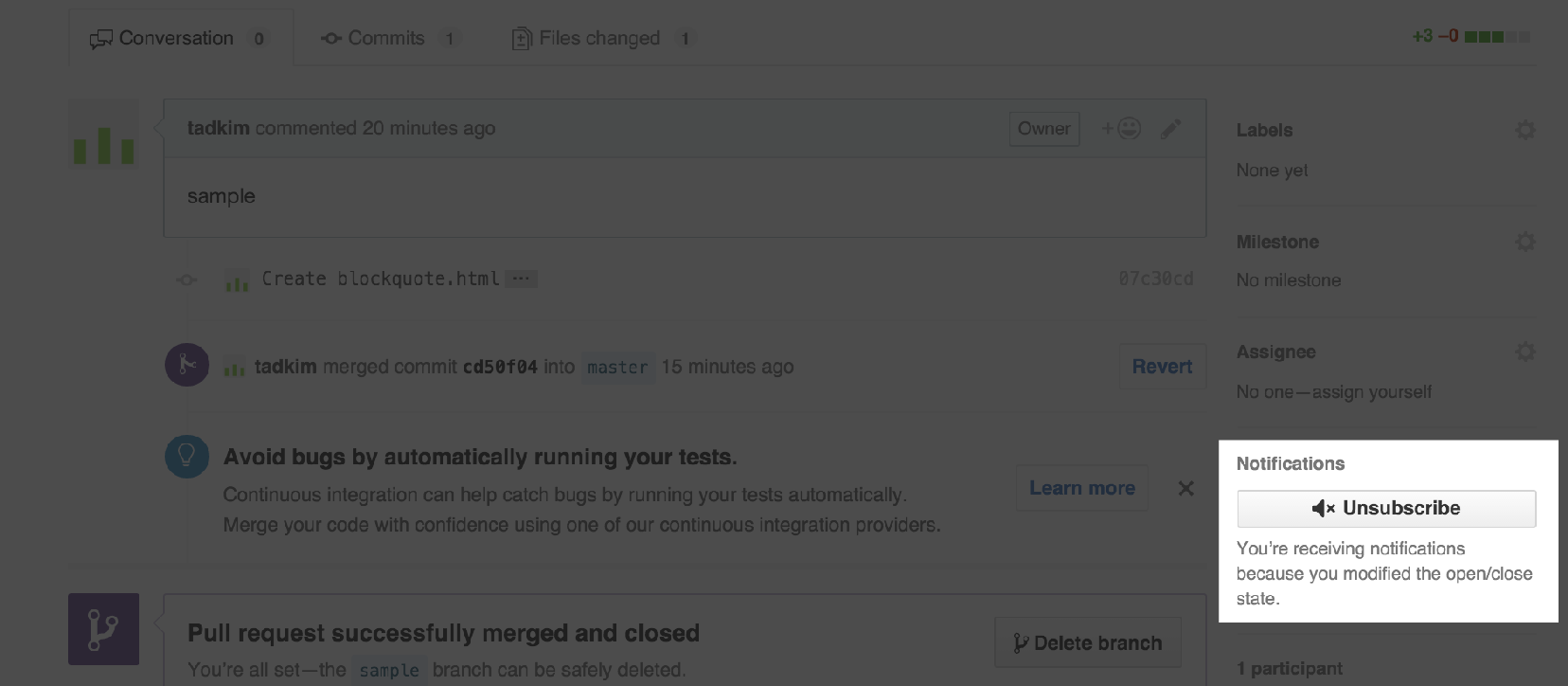
######Pull request 알림 기능(Notification)

다음 사항에 해당되는 사람은 기본적으로 pull request에 가입된다. **Pull request 알림대상**

* pull request를 작성한 사람
* pull request에 댓글을 단 사람
* (해당사항에 대해)commit을 생성한 사람
* @mention이 된 사람

**Pull request 알림 기능**

* pull request에 새로운 댓글 등록시
* pull request에 대한 commit 생성시
* pull request가 병합 또는 닫힐 때



**pull request가입 확인 및 알람기능 해제방법** 페이지의 오른쪽을 보면 pull request에 가입되어 있음을 알 수 있으며, 만약 가입되어 있는 pull request에 더이상 관심이 없다면 Unsubscribe버튼을 클릭해 해당 pull request에 대한 알람을 끄면 된다.

**pull request가입 방법** 반대로 눈여겨보고 싶은 pull request인데 가입되어 있지 않을 경우 Subscribe 버튼을 클릭하여 해당 pull request에 대한 모든 활동 내용에 대한 알림을 받으면 된다.

######Pull Request의 좋은 연습사례

개발환경에 익숙해지는데 도움이 될 만한 연습 방법을 정리&공유한다.

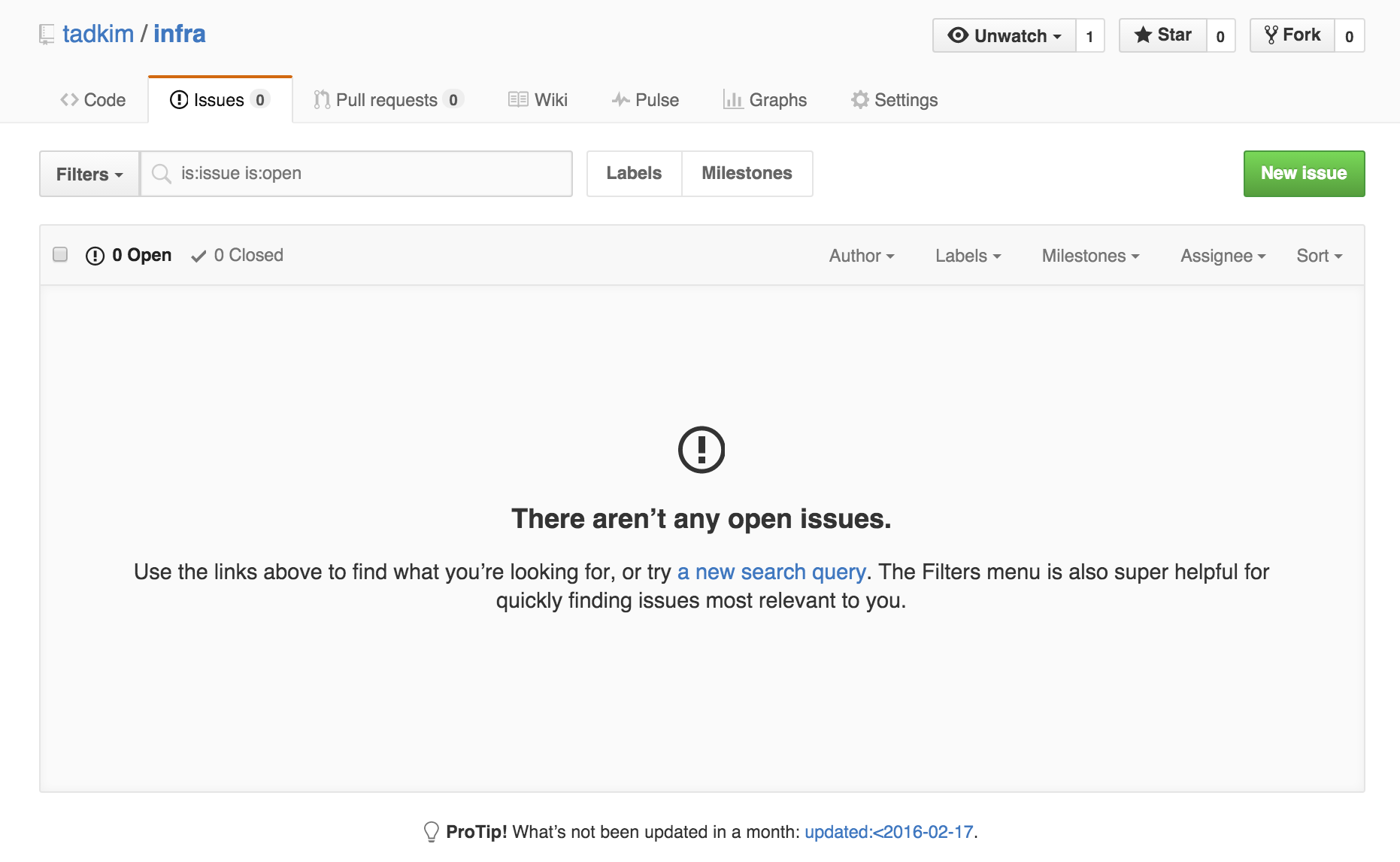
* **모든 것에 pull request 만들기** : 버그를 수정하거나 새 기능을 추가할 때마다 반드시 브랜치에서 작업하고 마스터로 병합하기 전에 다른 사람들에 의견을 받기 위한 pull request를 만든다.
* **알아보기 쉬운 제목 짓기** : 작업이 어떻게 진행되는지 알고자 다른 팀원들이 pull request 페이지를 들여다 볼 것이다. 제목을 보고 무슨 작업인지 알아챌 수 있어야 한다.
* **댓글에 공들이기** : @mention되지 않았더라도 공들여 댓글을 작성한다. 프로젝트의 진행상황을 잘 알 수 있을 뿐만 아니라 작업의 전체적인 질을 향상시킬 것이다.
* **주요인물에 대한 @mention** : 마케팅, 법무, 운영팀의 피드백을 받고 싶으면 필요한 사용자를 @mention해서 그들이 pull request를 보도록 하고, 피드백 할 수 있는 환경을 만들어준다.
* **테스트 시행** : 최소한 한 명의 개발자라도 pull request에서 최근 변경을 다운로드하고, 적합한 브랜치인지 확인하며, 자동화된 테스트를 반드시 실행하도록 한다. 중대한 변경 내용을 그저 눈으로 코드를 봐서는 충분히 알 수 없다.
* **pull request를 승인하는 명확한 정책 갖기** : 대부분의 회사에서는 pull request가 병합되기 전에 pull request의 원 작성자 외에 한 두명의 사람들이 검토를 하고 :+1:을 제공하도록 하고 있다.
* **pull request와 issue의 차이** : pull request가 완료도니 작업의 통합에 이용되는 반면, Issue는 버그 또는 노의하거나 개발해야하는 새로운 기능 요구사항을 설명하는 데 이용한다.

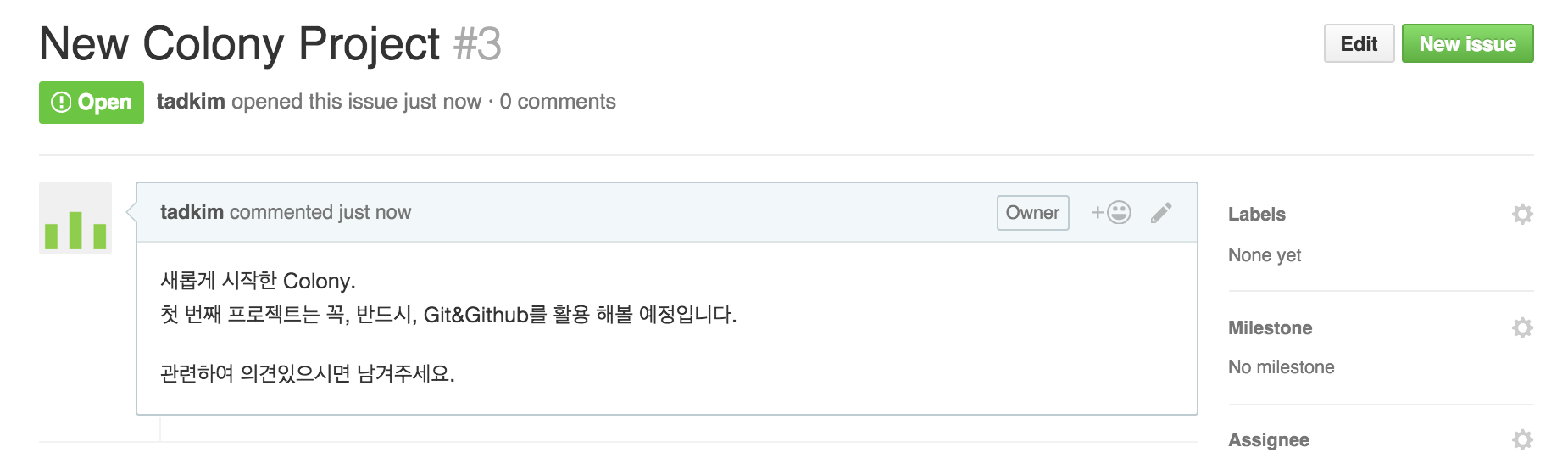
#Github 협업 #2 Issue

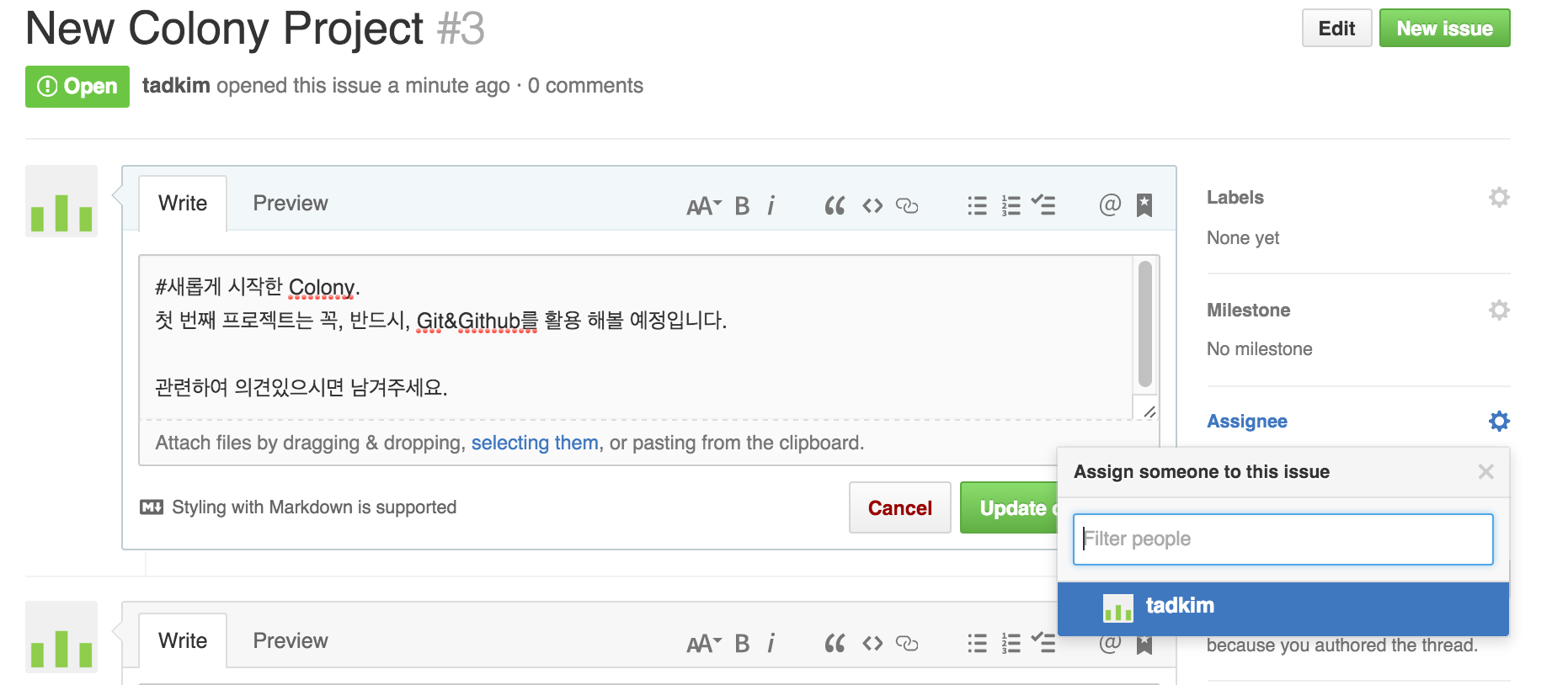
Issue는 수정해야 하는 버그나 새로 개발해야 하는 기능을 관리하는 가볍고 사용하기 편리한 도구이다.

보통 새로운 프로젝트를 시작할 때 GitHub Issue를 사용해 버그나 기능을 관리한다. 그리고 Issue가 제공하지 않는 기능이 있을 때만 Pivotal tracker, JIRA, Light-House, Trello, Asama 등과 같응ㄴ 툴을 이용한다.

**Issue 생성하기**

새로운 이슈를 생성하기위해서 Github의 프로젝트 페이지에서 Issue 탭을 클릭 한다. 





**Issue에서 설정가능한 다양한 옵션**

* **Label** : 프로젝트에 대한 사용자 지정 라벨을 만들 때 사용한다.
* **Milestone** : issue의 milestone은 “7월 29일 투자자 PT”등과 같은 외부 마감날짜나 스프린트[^2]를 지정할 때 사용한다. New Milestone을 클릭하면 제목, 설명, 기한을 묻는 양식이 나타난다.
* **Assignee** : 누가 Issue에서 작업하고 있는지 알면 No one is assigned라는 버튼을 클릭하여

[^1]: 사용자들 사이에서 이루어지는 비공식적인 룰. pull request가 병합되기 전에 pull request 작성자 외에 한두 명에게 :+:를 해줄 것을 요구하는 팀이 많다. 닫기 위한 프로세스는 갖되, 간단하게 한다. 병합한 후에도 언제든 되돌릴 수 있다. [^2]: 반복적인 개발 주기, 계획회의부터 제품 리뷰까지를 보통 1 스프린트라 한다.

#Github 협업 #3 Wiki

Wiki기능은 협력자들이 다량의 연결된 페이지를 개발하기 쉽게 해주는 간단한 콘텐츠관리 시스템이다.

보통 Github wiki는 사용자 문서 개발자 문서 또는 모두를 캡처해 프로젝트와 관련된 모든 정보가 GitHub를 통해 접근 가능하도록 하는 데 사용된다.

1. **Git 작업환경 시작하기 (see also: git help tutorial)**
   * clone 새로운 디렉터리에 Repository를 복제한다.
   * init 새로운 빈 Repository를 생성하고, 기존 존재하던 것들을 초기화한다.
2. **현재 변경작업 (see also: git help everyday)**
   * add 인덱스에 파일 내용을 추가 Add file contents to the index
   * mv 파일 이동 또는 파일 명 변경 또는 심볼릭 링크의 이름변경 (Move or rename a file, a directory, or a symlink)
   * reset 지정된 상태로현재 HEAD 리셋 (Reset current HEAD to the specified state)
   * rm 작업 트리(Working tree)에서 인덱스 파일을 제거(Remove files from the working tree and from the index)
3. **작업이력과 현재 상태를 검사 (see also: git help revisions**)
   * bisect 이진 검색으로 버그 변경을? 찾는다.( Find by binary search the change that introduced a bug)
   * grep Print lines matching a pattern
   * log Commit 로그를 보이게한다. (Show commit logs)
   * show 다양한 종류의 객체를 보이게 한다.(Show various types of objects)
   * status 작업트리(Working tree)를 보이게한다.(Show the working tree status)
4. **성장, 표시, 당신의 최근 변경 이력**
   * branch 브런치를 생성하거나 목록을 만든다. (List, create, or delete branches)
   * checkout ??Switch branches or restore working tree files
   * commit 저장소(Repository)에 변경사항을 기록한다.(Record changes to the repository)
   * diff 두 개 이상의 변경사항 (Commit과 작업트리의 Commit등)을 표시 변경? (Show changes between commits, commit and working tree, etc)
   * merge 두 개 이상의 개발 과정을 병합 (Join two or more development histories together)
   * rebase Forward-port 지역 저장소의 Commit을 업데이트 된 상위 HEAD에 전달. (Forward-port local commits to the updated upstream head)
   * tag 다양한 태그 객체를 생성, 삭제 또는 확인한다. (Create, list, delete or verify a tag object signed with GPG)
5. **협업 관련 (see also: git help workflows)**
   * fetch 다른 저장소에서 개체를 다운로드하고 바로 병합하지는 않는 상태(Download objects and refs from another repository)
   * pull 원격 저장소(Repository)에서 로컬저장소로 가져와 로컬지점과 통합한다.)Fetch from and integrate with another repository or a local branch)
   * push 관련 객체를 가지고 원격저장소에 업데이트 위해 보낸다. 바로업데이트는 되지않는다.(Update remote refs along with associated objects)

#Git :: 원격저장소

##Git :: 원격저장소 1. 기본환경 세팅

다른 사람의 gitgub repositiory clone 후 branch 정보가져오기

git pull origin 'branch이름'

**리모트 브랜치 remote branch** : remote 저장소에서 만들어지는 branch

리모트 저장소는 협업에 참여하고 있는 작업자들이 접속할 수 있는 중립된 저장소를 의미

**SSH Keys**

SSH Key: push와 같은 쓰기작업을 시도했을 때 공개키와 로컬PC의 비공개키를 비교해 ‘자동로그인’과 같은 기능을 적용해주게된다.

$ ssh-keygen

//\*Your identification has been saved in \*\*`/Users/admin/.ssh/`\*\*id\_rsa.\*

\*\*/Users/admin/.ssh/\*\*를 카피해서 해당 디렉토리로 바꿔준다.

$ ls -al

id\_rsa, id\_rsa.pub을 기억해야하고 유출해서도 안된다.

* id\_rsa : 비공개키.
* d\_rsa.pub : 공개키

원격 저장소 연결하는 등의 작업시 비공개키와 공개키를 매칭해서 맞으면 별도의 로그인없이 작업을 진행할 수 있게 된다.

$ cat id\_rsa.pub //ssh key 출력

//ssh key 출력 결과

ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABAQDQbM29wqgX+RpKLyG+MtdFOeC7lb3rIID1fTiW5HKATj38Fta7VRRiiPUba/YwvCqWz7r4eejAjbDf3YsSV+4gijHeT5XoLiA1PPEd1+/zk+UeEwkvA/41zPUniQgwGFhuqvM6d5BF/gPy3bu9zlfl+rBuBiNO0iG2WaeCCfADmNW6KF8IzLZ82PIwJsTxfrqwk1TpKoQOaGZiP5ZLFJ1fiOQCRaTsQvoRBOMdJmVV71/InyXIAIookGYhgHpvyRWsftNuGitrnDyoGn3byBa65bL8Ah4Mf681BQQlCdE/iwDu6u3C1N8zqUL3O2D6Hy13qzL9ugbdNMxz336KKe/B admin@adminui-MacBook-Pro.local

출력 된 키를 드래그해서 복사한 후 다음의 과정에 따른다.

1. Github 로그인
2. Account Settings > SSH keys
3. new SSH Key (또는 Add SSH Key)
4. title에는 자유롭게 SSH KEY 이름을 입력 (예 : admin everybody computing)
5. key영역에 드래그로 복사했던 sshkey를 불어넣기.

**Git console에서의 협업 1(리모트 저장소)**

**리모트 저장소 복제**

\*\*$ git clone (SSH KEY)\*\*

//\*git clone git@github.com:tadkim/openeverybody.git\*

\*\*$ls -al\*\*

\*\*$cd openeverybody\*\* //\*openeverybody는 저장소이름\*

\*\*$ls -al\*\*

//\*total 16\*

//\*drwxr-xr-x 5 admin staff 170 3 22 12:08 .\*

//\*drwxr-xr-x 5 admin staff 170 3 22 12:08 ..\*

//\*drwxr-xr-x 13 admin staff 442 3 22 12:08 .git\*

//\*-rw-r--r-- 1 admin staff 106 3 22 12:08 .gitignore\*

//\*-rw-r--r-- 1 admin staff 32 3 22 12:08 README.md\*

//\*adminui-MacBook-Pro:openeverybody admin$ cat .gitignore\*

//\*\*/config/development\*

//\*\*/logs/log-\*.php\*

//\*!\*/logs/index.html\*

//\*\*/cache/\*\*

//\*!\*/cache/index.html\*

//\*!\*/cache/.htaccess\*

그다음에는

\*\*$ git add helloworld.html\*\* //\*tracked 상태로 전환\*

\*\*$ git commit -m 'first commit\*\*'

//\*[master 4479b64] first commit\*

//\*1 file changed, 165 insertions(+)\*

//\*create mode 100644 helloworld.html\*

commit을 했으나 원격저장소에 바로 가는 것이 아니다. 커밋은 오로지 로컬저장소에 저장되고, push했을 경우 반영된다.

\*\*$git remote\*\* // \*리모트 저장소의 별명 확인하는 코드\*

//\*origin : 원격저장소를 clone해서 그렇다.\*

**git push : 서버에 작업내용을 업로드**

*서버에있는 내용을 먼저 다운로드해야한다. 다운로드 후, 자신의 파일의 내용과 충돌하는 게 없는지 확인후 push해줘야한다.(동기화작업)*

**리모트 브랜치와 동기화**

* git fetch (리모트 저장소의 별명) (리모트 브랜치) (리모트 저장소 별명)에 해당하는 저장소의 (리모트 브랜치)를 현재 선택된 로컬 브랜치로 가져온다.
* git pull (리모트 저장소의 별명) (리모트 브랜치)
* git fetch명령을 실행하고 자동으로 merge(병합) 하기
* git push (리모트 저장소의 별명) (리모트 브랜치) (리모트 저장소 별명)의 (리모트 브랜치)로 로컬 브랜치를 전송한다. 이 명령을 수행하기 전에 git fetch나 git pull을 수행한다.

**$git pull origin master** //*From github.com:tadkim/openeverybody* //\* branch master -> FETCH\_HEAD\* //*Already up-to-date.*

**리모트 브랜치 생성**

* git checkout -b (새로운 브랜치) : (새로운 브랜치)를 로컬 브랜치로 만든다.
* git push (리모트 저장소) (리모트 브랜치) :(리모트 저장소)에 (리모트 브랜치)를 생성하고 현재의 로컬 브랜치와 추적상태를 만든다.

**$git push origin master**

//\*Counting objects: 3, done.\*

//\*Delta compression using up to 4 threads.\*

//\*Compressing objects: 100% (3/3), done.\*

//\*Writing objects: 100% (3/3), 2.63 KiB | 0 bytes/s, done.\*

//\*Total 3 (delta 0), reused 0 (delta 0)\*

//\*To git@github.com:tadkim/openeverybody.git\*

//\*f05bfaa..4479b64 master -> master\*

별도 설정하지않으면 git에서 ‘master`라는 이름을 자동으로 생성한다.

**리모트 저장소 복제**

* git clone (리모트 저장소 URL)
* git add (리모트 저장소의 별명) (리모트 저장소의 URL)

**리모트 저장소의 삭제**

* git remote rm (리모트 저장소의 별명)

**리모트 브랜치 현황보기**

* git remote : 로컬 브랜치에 연결되어 있는 리모트 브랜치 리스트 보기
* git remote show (리모트 브랜치) : (리모트 브랜치)와 로컬브랜치의 관계를 상세히 보기 **리모트 브랜치 삭제**
* git push (리모트 저장소의 별명) :(리모트 브랜치) (리모트 저장소의 별명)의 (리모트 브랜치를 삭제) ':' 앞에 뛰어쓰기해야 함

**git fetch VS git pull**

**git fetch와 git pull 중 어떤 것이 더 나은 방법일까**? 정답은 없지만 git fetch가 더 바람직하다. git fetch는 원격저장소의 내용을 모두 가져오지만 그것을 working copy에 합치지 않는다. 다시 말해서 소스가 변경되지 않는다는 것이다. 그럼 어떻게 해야 코드가 최신 상태가 될까? git merge 명령을 실행시키면 된다. 그럼 fetch를 통해서 가져온 원격 저장소의 소스와 자신이 작업하고 있었던 소스가 병합(merge) 된다.

**간단하게 pull하면 되는데 fetch를 하는 이유는 무엇일까**? fetch를 하면 실제로는 working copy에 반영되지 않기 때문에 원격 저장소와 로컬 저장소 사이의 차이점을 비교 할 수 있다. 문제점이 있는지 여부를 미리 검토 할 수 있는 것이다. 필자의 경우 fetch를 이용하는 경우 아래와 같은 흐름으로 작업을 한다.

* git fetch; #원격 저장소에서 로컬 저장소로 변경점을 가져온다. 단, Working Copy에는 반영하지 않는다.
* git log git log ..origin/master; #fetch를 이용해서 가져온 변경점과 Working Copy 사이의 차이점만 보여준다.
* git merge; #fetch를 이용해서 가져온 내용을 Working Copy에 반영한다. 만약 fetch를 이용해서 가져온 변경 점 중에 특정 커밋 이전만 Working Copy에 반영하고 싶다면 git merge 뒤에 커밋 ID를 붙인다.

[생활코딩 - Git console에서의 협업 1(리모트 저장소)](https://www.youtube.com/watch?v=NCHhXvvX4OU)

##Git :: 원격저장소 2. commit

[Git Remote Branch 1]에서 어떻게 Github에 가입하고, 원격 저장소와 로컬저장소를 연결하는지. 원격 저장소에서 pull을 받고 로컬저장소에서 원격저장소로의 push를 날리는 방법에 대해서 살펴보았다.

이번 글에서는, 지난 글에 이어 commit하는 방법에대해서 살펴보려고한다.

1. 지난 번 과 동일한 방법으로, 원격저장소를 clone하여 새로운 로컬저장소를 만든다.
2. 기존 로컬 저장소 A와 새로운 로컬 저장소 B를 각각 다른 터미널 창에서 연다.
3. 저장소 A에서 파일 수정 후 커밋을 날린다.
4. 저장소 A에서 수정 파일을 push한다.
5. 저장소 B에서 원격저장소의 변경사항을 저장하기위해 git pull을 해준다.
6. 저장소 B에서 helloworld.html을 수정 후, 3번과 4번에서의 방법으로 commit, push를 날려준다.
7. Github에서 helloworld.html파일의 형태를 확인한다.

STEP1. 새로운 터미널 창을 연 후, 원격저장소를 clone한다. $git clone [git@github.com](mailto:git@github.com):tadkim/openeverybody.git openeverybody2 //*$git clone SSHKEY 새로운로컬저장소이름*

이렇게하면 새로운 폴더가 생성되고 최신 원격저장소의 파일들이 세팅된다.

**STEP2** 각각의 독립된 창에서 기존 로컬저장소 A와 새롭게 clone하여 만든 로컬저장소 B를 연다. **STEP3**기존 저장소인 A에서 만들어둔 helloworld.html파일을 임의로 수정하고, commit을 날린다. $git commit -am ‘change hello world.html`;

**STEP4** 저장소 A에서 수정된 파일을 push한다. $git push

**STEP5** 저장소 B에서 원격저장소의 최신 정보를 동기화하기 위해 git pull을 실행한다. $git pull origin master

**STEP6** 저장소 B에서 helloworld.html을 수정 후, 3번과 4번에서의 방법으로 commit, push를 날려준다.

**STEP7** Github에서 helloworld.html파일의 형태를 확인한다.