

초심자를 위한 Github 협업 튜토리얼 (with 토끼와 거북이)

대상

본 자료는 간단한 git-flow로 Github에서 협업을 하는 과정을 설명한 글입니다.  
초심자를 대상으로 하며, SourceTree를 사용합니다.

익힐 수 있는 개념

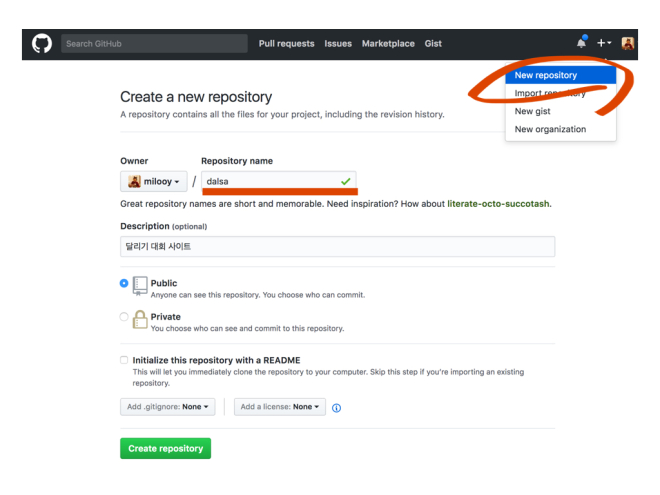
commit, push, branch, merge, rebase, pull request, release, fork

PART 1. 단일 저장소에서 협업하기

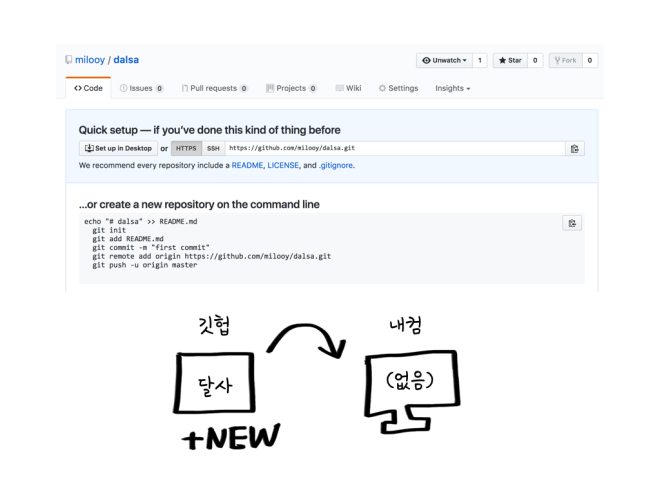


개발자 토끼와 거북이가 달리기 대회 사이트 ‘달사’를 만들려고 합니다  
토끼는 **메인 페이지**를, 거북이는 **대회 신청 페이지**를 만들기로 했습니다.

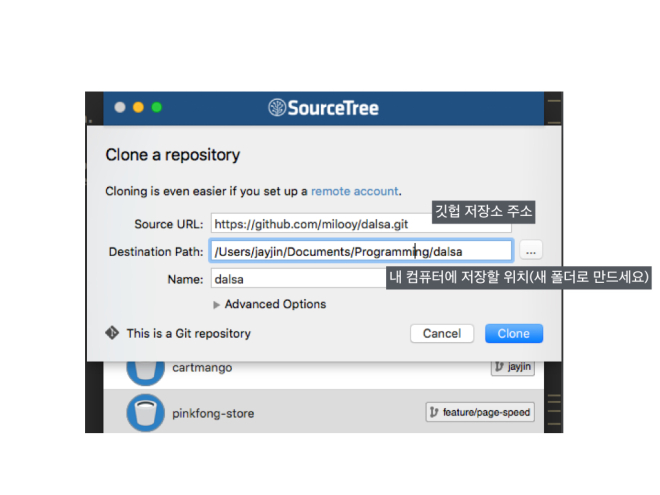
그런데 각자 코딩을 하면, 이를 **어떻게 합쳐서** 하나의 ‘달사’로 만들까요?  
토끼와 거북이는 이에 대한 수단으로 [git](https://git-scm.com/book/ko/v2/%EC%8B%9C%EC%9E%91%ED%95%98%EA%B8%B0-CLI)을 사용하기로 했습니다.



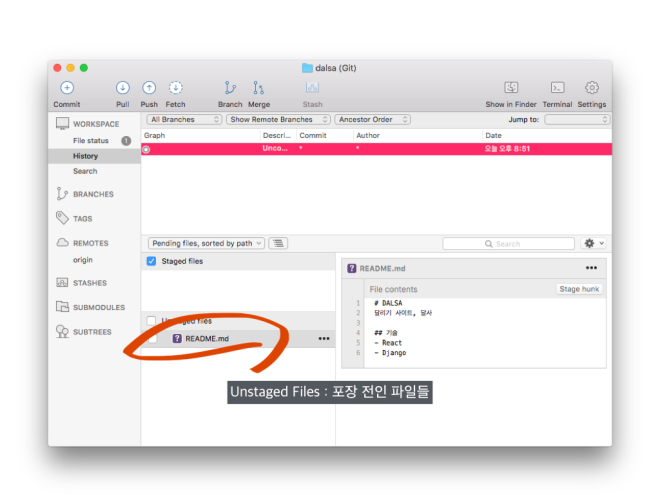
토끼는 <https://github.com/> 에 가입해서  ‘dalsa’라는 이름의 새로운 저장소를 만들었습니다.  
Public으로 만들어 누구나 이 프로젝트에 참여할 수 있게 할 거예요.  
*\* Organization 을 만들어 단체의 이름으로 저장소를 만들 수도 있습니다*



**원격**(Github 사이트)에 dalsa 저장소가 생겼습니다.  
이젠 여기에 코딩을 하기 위해 **로컬**(내 컴퓨터)에 저장소를 다운받아야 합니다.



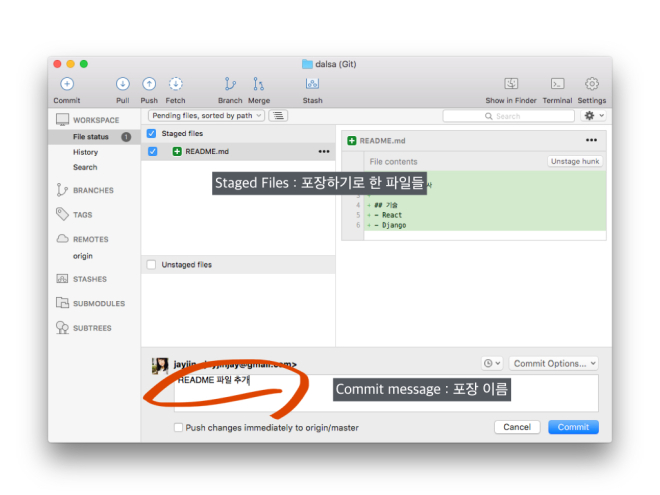
토끼는 먼저 git을 설치하고, ([https://git-scm.com/book/ko/v2/시작하기-Git-설치](https://git-scm.com/book/ko/v2/%EC%8B%9C%EC%9E%91%ED%95%98%EA%B8%B0-Git-%EC%84%A4%EC%B9%98)) [SourceTree](https://www.sourcetreeapp.com/)의 ‘+New Repository > Clone from URL’을 사용해서 저장소를 받아옵니다.  
SourceTree, Github Desktop같은 GUI 프로그램은 시각적으로 상태를 확인할 수 있다는 장점이 있지만 CLI의 모든 기능을 지원하지 않습니다.  
개인이 이해하기 쉬운 것 먼저 배우셔도 되지만, CLI로 하는 법은 처음이든 나중이든 필수로 익히셔야 합니다.  
CLI로 git을 설치하고 사용하는 방법은 [git 간편 안내서](https://rogerdudler.github.io/git-guide/index.ko.html) 를 참고해주세요.



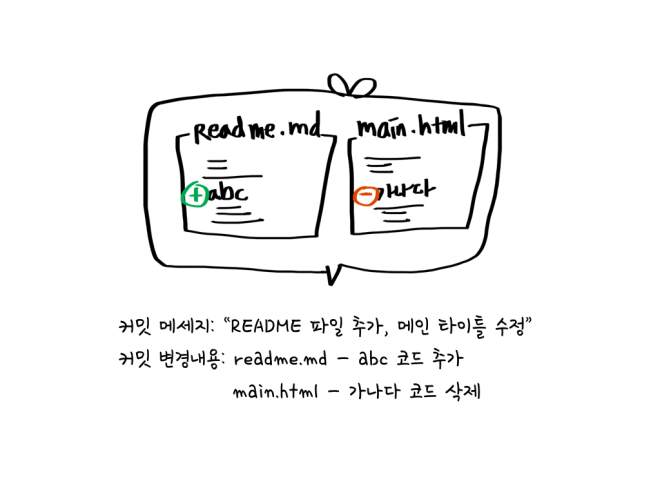
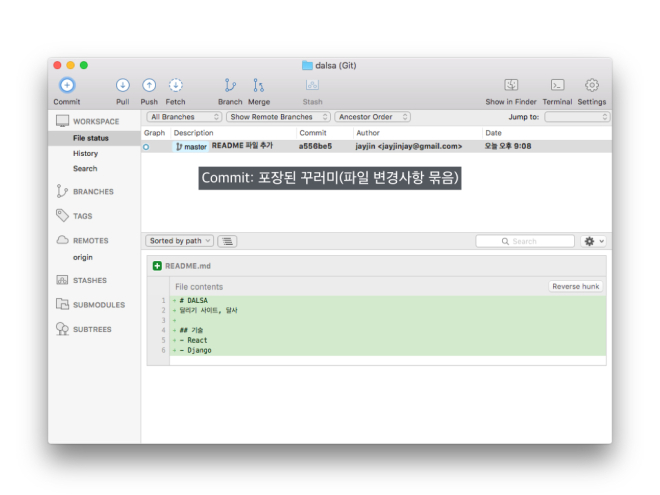
토끼는 제일 먼저 저장소에 대한 설명인 README.md 파일을 만들었어요.  
이를 원격에 올릴거예요.

깃에 커밋을 하는 올릴 파일은 3가지 상태가 있습니다.  
이를 ‘포장하는 과정’이라고 생각해 볼게요,

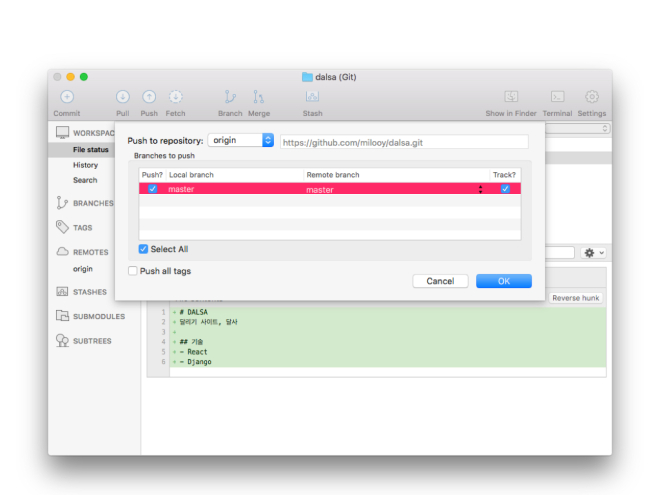
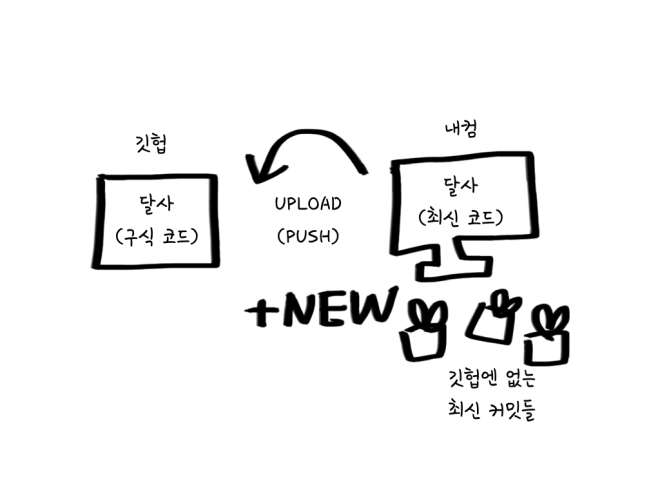
1. 포장 전인 파일들: Unstaged Files
2. 포장하기로 한 파일들: Staged Files
3. 포장 완료 파일 묶음: Commit



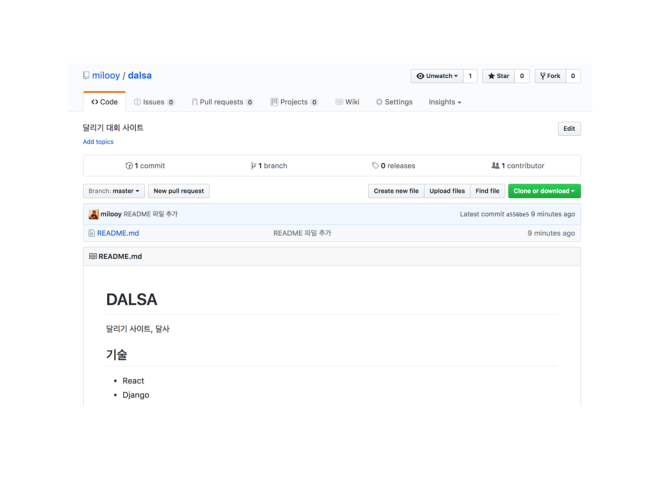
README파일을 체크해서 ‘이 파일을 포장할거다’라는 *Staged*단계로 올립니다.  
지금은 한 파일밖에 없지만, 나중에 변경 내역이 여러 파일에 있으면 내가 원하는 것들만 선택해서 staged로 올릴 수 있겠죠?  
이는 ‘README 파일 추가’란 이름을 붙여서(*commit message*) 선물로 포장할거예요(*commit*).



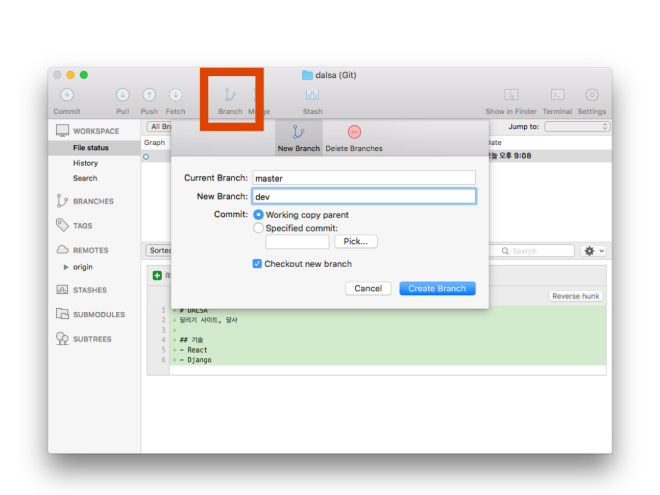
이렇게 코드를 의미있는 단위로 쪼개서 포장하는 것이 *Commit* 입니다.  
각 커밋별로 변경된 이력을 편하게 볼 수 있고, 예전에 올렸던 커밋으로 코드를 돌려 시간여행을 할 수 있습니다.  
오프라인 상태에서도 가능해서 비행기나 산 꼭대기에서도 커밋할 수 있죠(?)



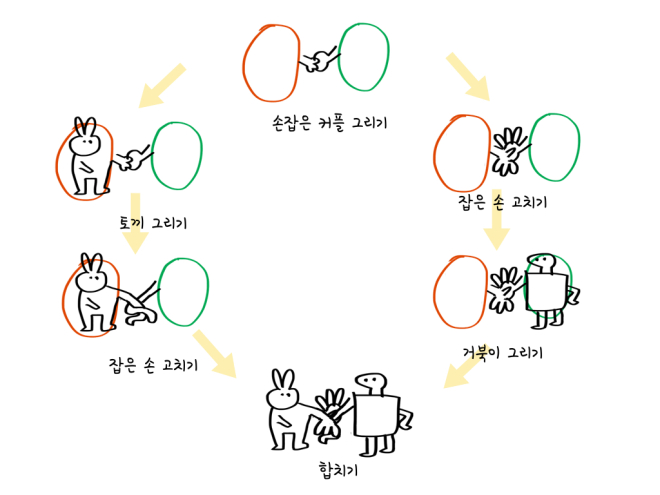
이렇게 커밋만 하면 내 컴퓨터에만 저장되어 있고 Github사이트엔 아직 올라가지 않은 상태입니다.  
이 커밋들을 진짜 올리기 위해 Push를 합니다.  
Push를 누르면 git이 우리 컴퓨터 폴더에 숨겨져 있는 .git파일을 참조해 그 곳에 저장된 달사 저장소 url에 코드를 올려줍니다.



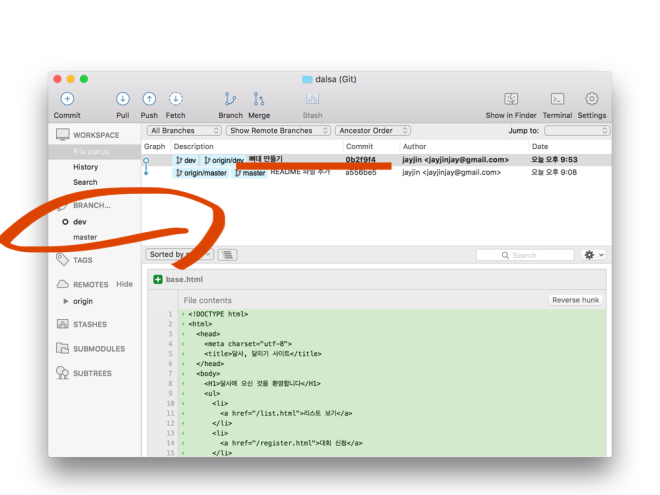
Github 저장소에 가 보면 잘 올라와 있는 것을 확인할 수 있습니다.

이렇게 바로 올린 코드는 *Master 브랜치*(기본적으로 생김)에 올라갔습니다.  
브랜치는 단어 그대로 **가지**를 뻗어 나가는 것입니다.  


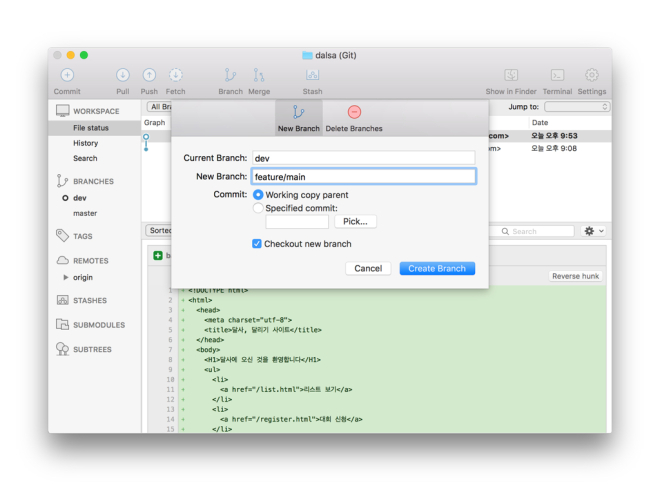
토끼와 거북이는 Master 브랜치엔 **완벽한 코드만(언제든지 배포할 수 있는) 두자**는 약속을 했습니다.  
그래서 **‘개발 중’인 코드**를 올리기 위한 dev브랜치를 새로 만들었습니다.



특정한 코드 상태에서 브랜치를 따면 새로운 브랜치에서 독립적으로 작업하고 나중에 합칠 수 있습니다.  
위의 예시에서 브랜치 없이 둘이 한 종이에 작업했다면 서로 같은 곳을 수정했을 때 (잡은 손 고치기) 에러가 났겠죠.  
브랜치로 작업하면 추후에 합칠 때 원하는 손 그림을 택하면 됩니다.

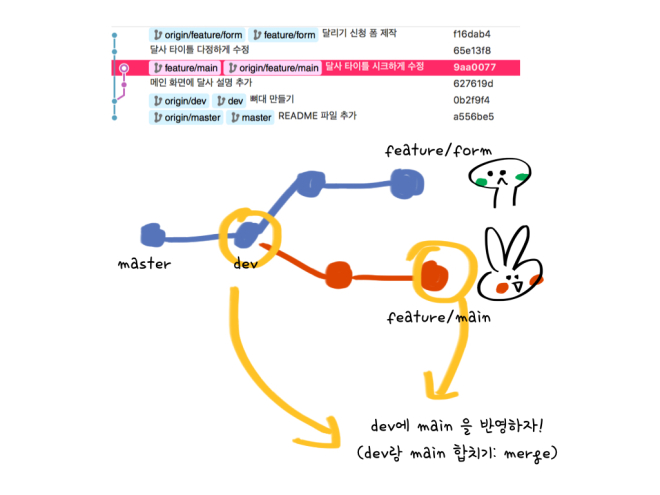


둘은 짝 코딩으로 사이트 뼈대를 만들어 dev 브랜치에 커밋 후 푸시했습니다.  
좌측 메뉴의 branch에서 dev, master를 더블 클릭해서 각 브랜치를 옮겨다닐(checkout) 수 있습니다.



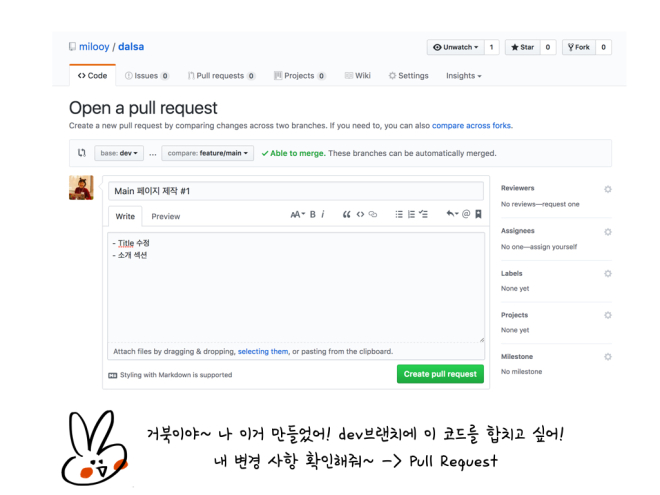
초기 세팅을 마친 거북이와 토끼는 서로 나뉘어서 코딩을 하기로 했습니다.

토끼는 메인 페이지를 만들기 위해 *‘feature/main’*이란 이름으로, 거북이는 달리기 신청 폼을 만들기 위해 *‘feature/form’*이란 이름으로 dev브랜치의 최신 커밋으로부터 새로운 브랜치를 만들었습니다.  
브랜치 앞에 ‘feature/‘처럼 슬래시로 구분된 이름을 달아주면 이 **구분별로 브랜치를 묶어볼 수 있는 장점**이 있습니다.

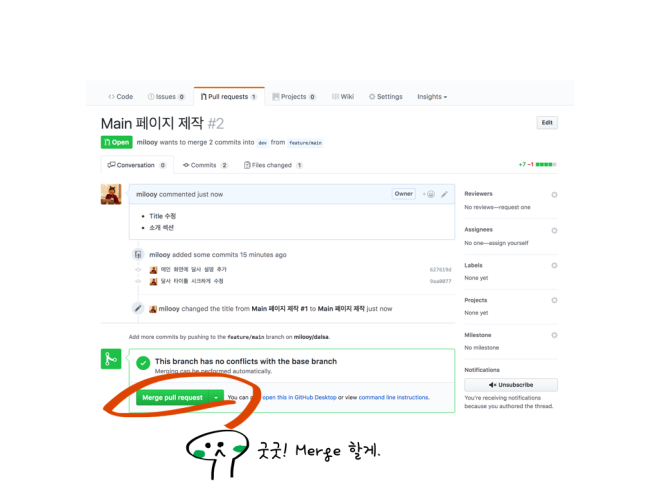


토끼와 거북이가 각자 브랜치에서 코딩을 합니다.  
시간이 흘러 토끼가 main 페이지를 다 만들어 dev브랜치에 ‘feature/main’브랜치를 합치려 합니다.  
하지만 예고 없이 바로 합치기보다는 거북이에게 **리뷰를 먼저 받고 합치고**싶습니다.

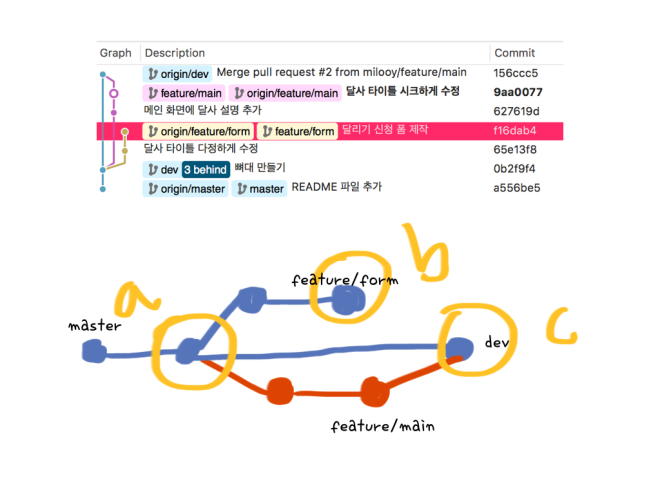
그러기 위해 하는 것이 ***Pull Request***입니다.



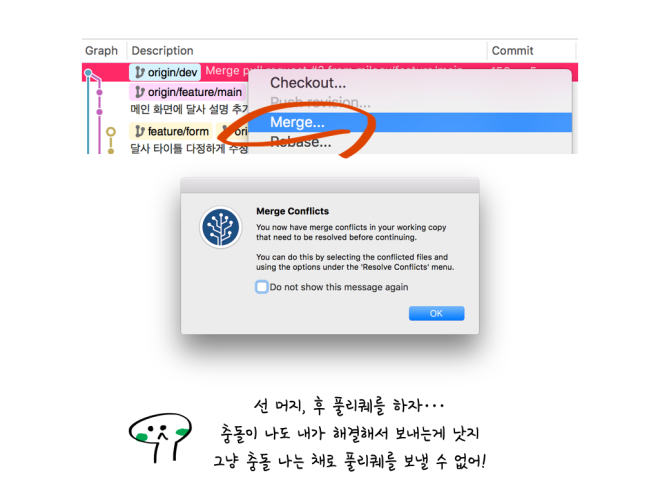
Github저장소에 있는 ‘Pull Request’버튼을 눌러  
‘feature/main’에서 ‘dev’브랜치로 Pull Request(땡김 요청)를 보냅니다.



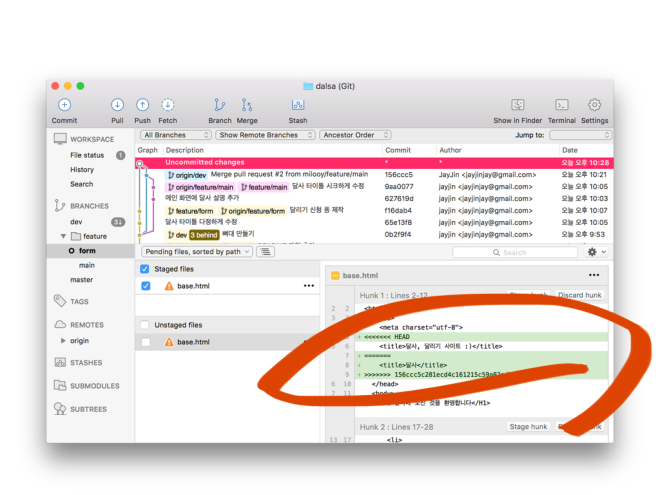
그럼 저장소의 ‘Pull requests’섹션에서 토끼가 보낸 풀리퀘를 확인할 수 있습니다.  
Files changed를 보니 잘 만들었네요 (맘에 안 들면 거부할 수 있습니다).  
하단의 ‘**Merge pull request**’를 눌러 토끼의 커밋들을 dev브랜치에 합칩니다.



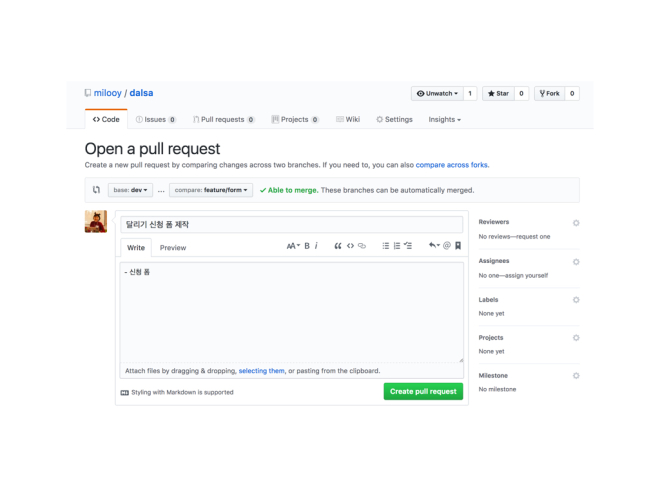
그 와중에 거북이도 폼 제작을 마쳤습니다.  
dev 브랜치(c)에 Pull Request를 보내려 보니 토끼의 새로운 코드가 추가되어 거북이가 처음에 땄던 dev 커밋(a)과 상태가 달라졌군요.  
a와 b는 바로 합칠 수 있는데 b와 c를 섣불리 Merge 했다가는 서로의 코드가 충돌날 수 있겠는데요?  
만약 거북이와 토끼가 같은 코드를 다르게 고쳤다면 에러가 나겠죠.

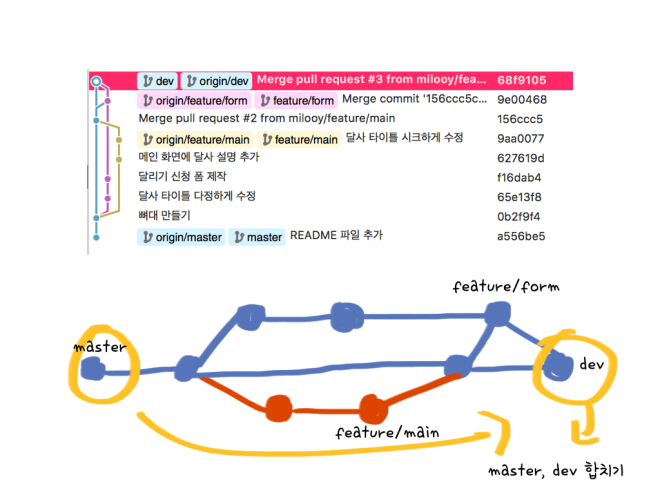


사려깊은 거북이는 ‘**선-머지 후-풀리퀘**’를 보내기로 합니다.  
dev의 최신 커밋(c)에서 마우스 오른쪽 버튼을 눌러 Merge를 합니다.  
그랬더니 **Merge Conflicts**가 나네요.

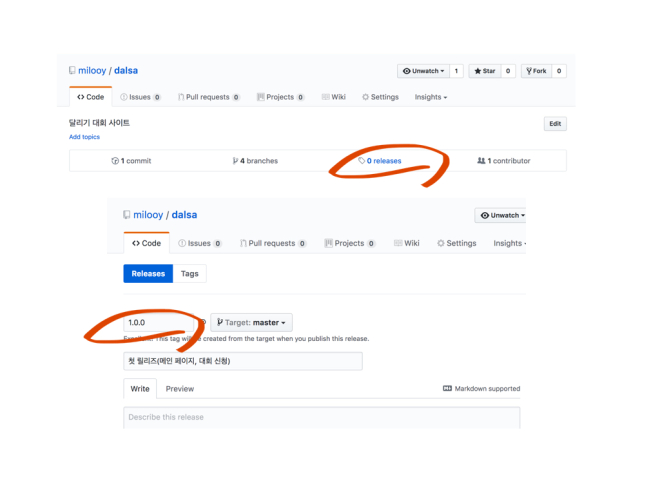


토끼와 거북이 둘 다 타이틀을 고쳤네요.  
코드를 보니 **거북이의 코드가 맞는 버전**입니다. 회의에서 다정 버전으로 하자고 결론이 났거든요.  
수동으로 토끼의 코드를 지우고 거북이의 코드를 남겨 commit+push합니다.



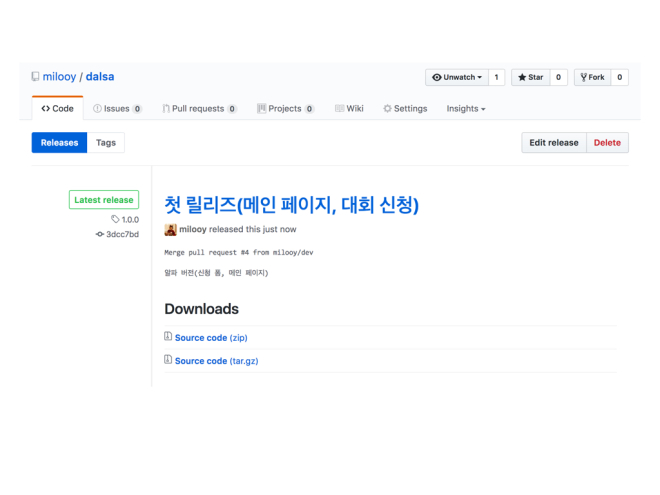
거북이도 dev로 Pull request를 보냅니다.  
토끼가 확인하고 merge하니 오류 없이 잘 됩니다.  
거북이의 ‘feature/form’에는 dev브랜치의 코드도 잘 섞여있는 버전이 들어가 있었기 때문이죠.  


이제 dev에 토끼의 ‘메인 페이지’와 거북이의 ‘신청 폼’이 모두 반영된 깔끔한  
코드가 올라왔습니다.  
이 정도면 베타 버전으로 출시해도 되겠어요!

배포를 위해 dev에서 master 브랜치로 pull request를 보내고 머지를 합니다.  
출시 가능한 버전이니까 master에 두면 되겠지요?  


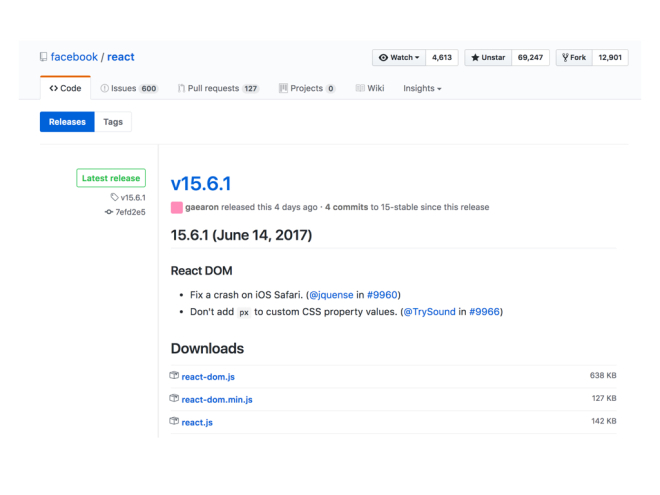
그럼 한번 출시를 해볼까요?

저장소에서 ‘**releases**’를 누르고, 이번 업데이트의 버전과 세부 내용을 적어준 뒤 Publish Release 버튼을 누릅니다. ([일반적인 버전 관리 규칙](http://kimsq.cweb.co.kr/b/tinylogs/10))

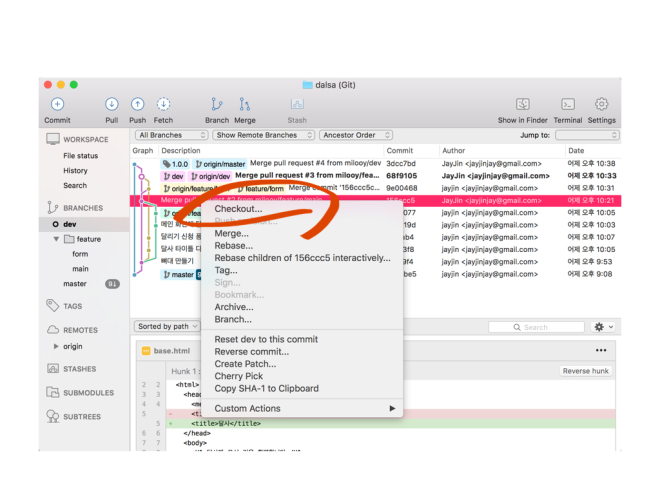


첫 번째 릴리즈가 완성되었습니다!

우리 서비스가 업데이트 되는 기록을 남길 수 있고,  그 당시의 소스코드를 다운받을 수도 있습니다. 만약 이번 업데이트에 치명적인 버그가 있다면 지난 버전으로 빠르게 롤백할 수도 있습니다.



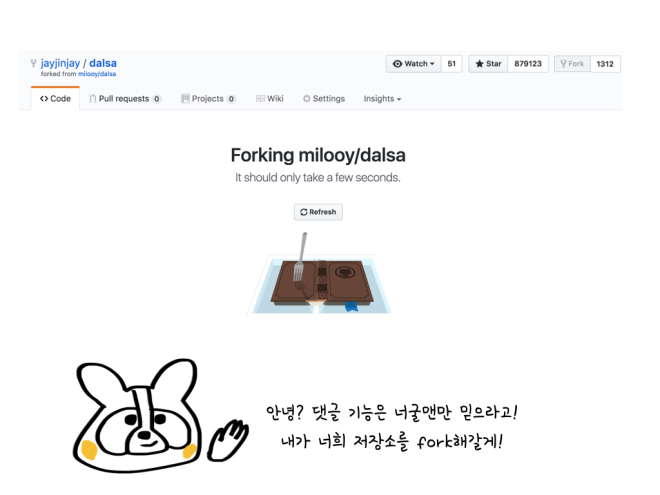
React처럼 잘 관리되는 오픈소스의 저장소 release 노트를 보면  
그 소스가 어떻게 관리되고 있는지 깔끔하게 확인할 수 있겠죠



이렇게 커밋을 하고 git에 올렸을 때의 또 하나의 장점은 언제든지 어떤 상태로든 **시간여행**을 할 수 있다는 것입니다.

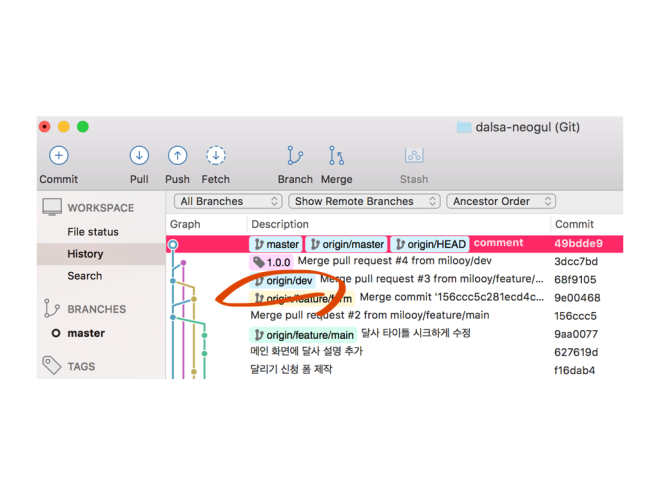
‘feature/main’브랜치에서 메인 페이지를 만들다가 ‘feature/comment’브랜치로 넘어가서 댓글 기능을 개발할 수도 있고, 거북이가 개발중인 ‘feature/request’로 넘어가서 거북이의 코드를 돌려볼 수도 있고, 내가 코딩을 이상하게 했을 때 내 브랜치의 기존 버전으로 reset할 수도 있습니다.

PART 2. Fork해와서 협업하기

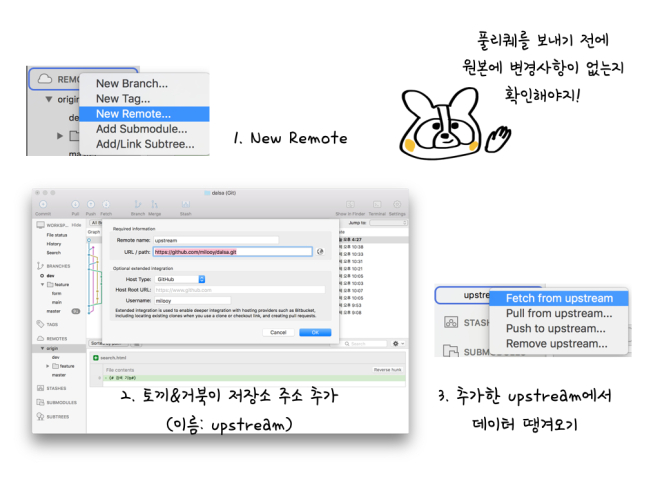


토끼와 거북이가 만든 ‘달사’가 흥했습니다. 너굴맨이 사이트를 써 보니 ‘댓글’기능이 있으면 좋을 것 같다는 생각이 들었습니다. 달사는 오픈소스니까 너굴맨이 기능을 추가해서 기여(**contribution**) 할 수 있죠.

달사 github repository페이지에서 **Fork**버튼을 누릅니다. 그러면 너굴맨의 계정으로 리포지토리가 복사됩니다.

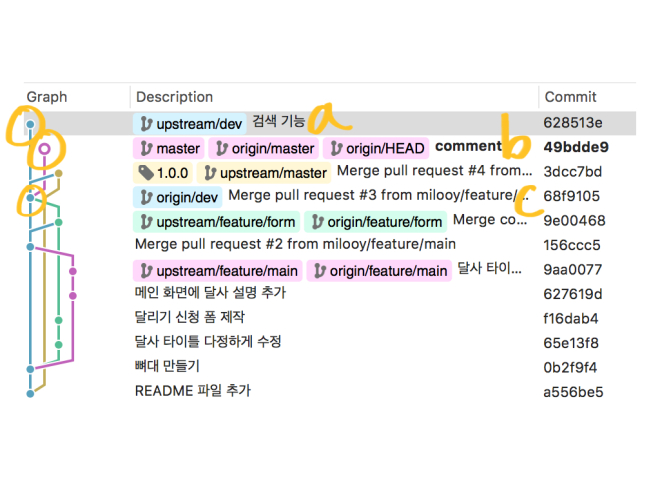


토끼와 거북이가 했던 방법처럼 원격 repo를 로컬에 받아오고, 댓글 기능을 커밋합니다.  
Master 말고 Dev의 최신 커밋에서부터 작업해야 나중에 Pull Request를 Dev로 보낼 수 있겠죠? 토끼와 거북이의 Master에는 배포 가능한 코드만 들어가야 한다는 규칙이 있으니까요.

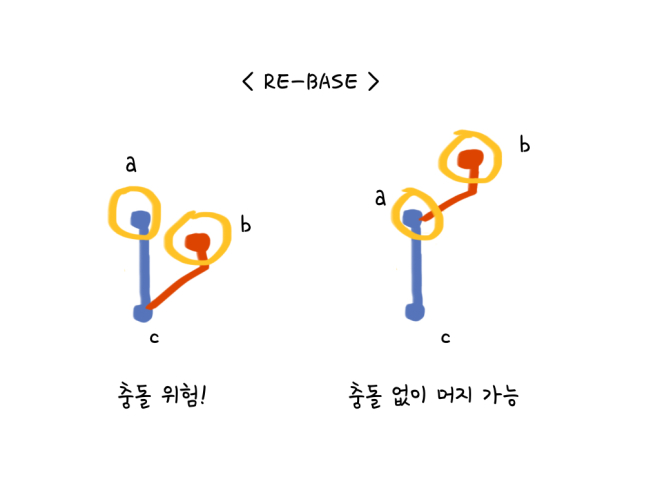


믿음직스러운 너굴맨은 Pull request를 보내기 전에 토끼와 거북이의 **원본 저장소에 변경 사항**이 없나 확인합니다. 여기는 너굴맨이 fork뜬 저장소이기에 원본 저장소의 최신 커밋들을 볼 수 없습니다.

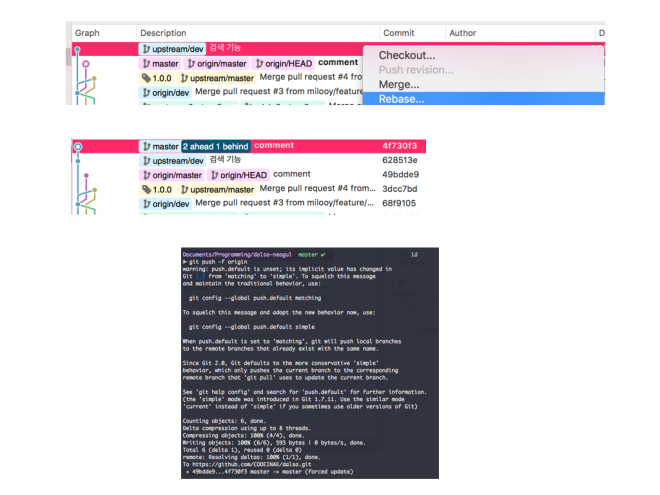
너굴맨은 **Remotes > New Remote**로 원본 저장소의 url을 ‘**upstream**’이라는 이름으로 추가해줍니다.



그 와중에 토끼와 거북이가 열일해서 너굴맨이 딴 c커밋에서부터 검색 기능(a커밋)을 dev브랜치에 새로 추가했네요. 그냥 Pull request를 보내면 충돌이 일어날 수도 있겠습니다.  
여기서 선머지 후풀리퀘스트를 할 수도 있겠지만, 깔끔한 코드 히스토리를 남기고 싶은 너굴맨은 ‘**Rebase**‘를 하기로 합니다.



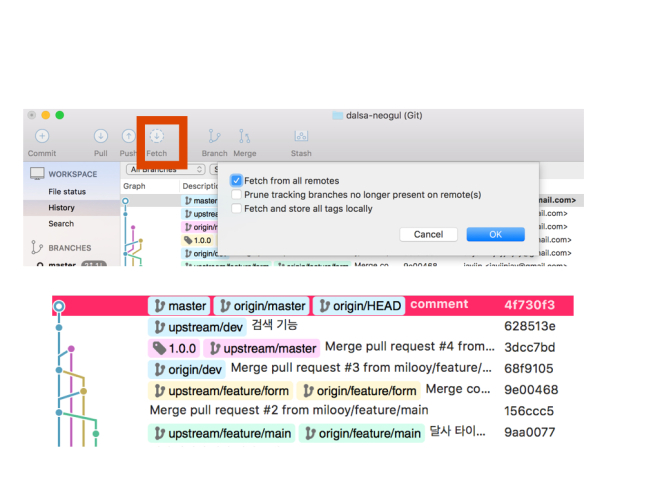
Rebase는 단어 그대로, 다시 베이스를 정하는 것입니다. 너굴맨의 코드는 C커밋에서 따서 현재 a커밋과 머지했을 때 충돌이 일어날 수 있습니다. 하지만 a커밋에서 브랜치를 땄다면 충돌 없이 한 번에 머지를 할 수 있겠죠? 여기서 a커밋에서 브랜치를 딴 것처럼 히스토리를 조작하는 것이 Rebase입니다.



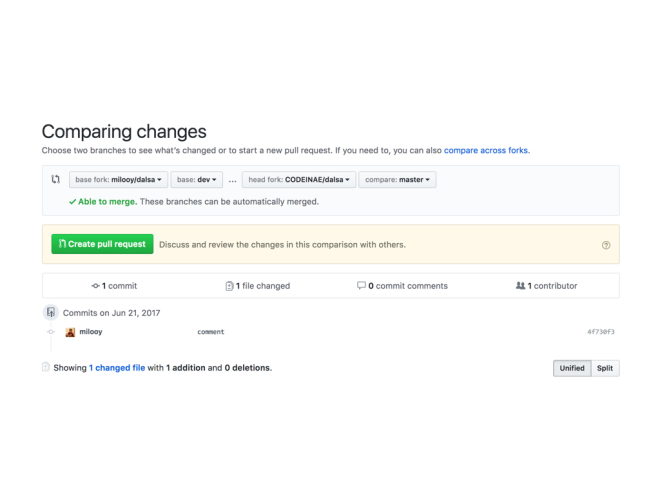
새로 베이스로 삼고 싶은 a커밋에서 ‘Rebase’를 눌러줍니다.  
(충돌이 난다면 고쳐줍니다. Merge할 때 충돌 해결하는것과 동일하게 해 주시면 됩니다. 충돌 해결을 하면 다시 commit을 눌러주세요.)

그러면 내 컴퓨터에는 b커밋이 c에서 딴 게 아닌 a에서 땄던 것처럼 **히스토리가 조작**됩니다. 근데 이를 원격 저장소에도 push하려면, 그냥 push로는 에러가 납니다. 히스토리를 망가뜨리는 위험한 push니까요. 그래서 콘솔 창에서 ‘**git push -f origin**’이라는 명령어로 force push를 해 줍니다(소스트리에는 force push 기능이 없습니다).

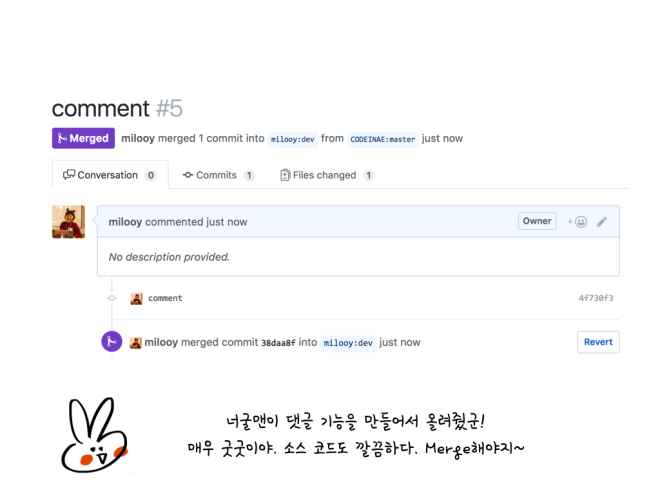
이는 다른 사람이 이 브랜치를 수정하고 있을 가능성이 제로일 때만 해야합니다.



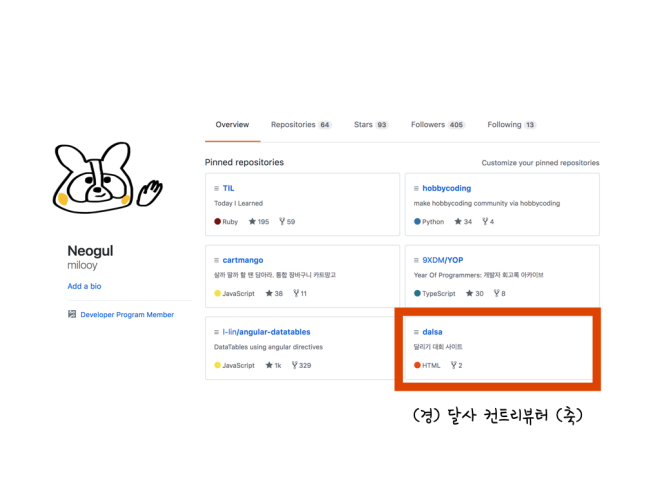
fetch를 한 번 눌러 원격과 내 소스트리를 동기화 해줍니다.  
베이스가 옮겨졌네요! 기존에 거북이가 머지를 했을 때는 ‘Merge pull request 어쩌고’ 이런 머지 커밋이 안 예쁘게 남았는데, 이제는 바로 깨끗하게 머지를 할 수 있는 상태가 되었습니다.



포크된 내 리포지토리에서 pull request버튼을 누릅니다.  
base fork를 토끼와 거북이의 dev브랜치, head fork를 너굴맨의 master브랜치로 해 줍니다.



토끼가 확인했네요! 맘에 들었나 봅니다. 너굴맨의 코드를 머지했습니다.



이제 너굴맨은 달사의 contributer로 등록되고, 너굴맨의 개인 페이지에도 달사 저장소가 뜨게 됩니다. 너굴맨 달사 커미터 축하!



이렇게 토끼와 거북이가 단일 리포지토리에서 협업하는 것, 너굴맨이 오픈소스에 기여하는 것을 ‘달사’ 서비스 만들기를 예를 들어서 설명하였습니다.

개선점이나 피드백 해주시면 감사하겠습니다   
모쪼록 처음 Github으로 협업을 하시는 분들께 도움이 되었으면 합니다.

토끼&거북이의 저장소: <https://github.com/milooy/dalsa>  
너굴맨의 저장소: <https://github.com/CODEINAE/dalsa>