DISCOVER **METEOR**

Git과 GitHub 사용하기 Sidebar

3.5

이 장에서는:

- 책을 따라해보며 Github 사용법을 배운다.

GitHub는 **Git** 버전 컨트롤 시스템에 기반한 오픈소스 프로젝트를 위한 소셜 저장소이다. 이것의 주요 기능은 프로젝트에서 코드의 공유와 협업을 쉽게 하는 것이다. 하지만 또한 훌륭한 학습도구이기도 하다. 이 사이드바 장에서는 독자가 *Discover Meteor*를 따라하는 과정에서 GitHub를 사용하는 몇 가지 방법을 빠르게 훑어볼 것이다.

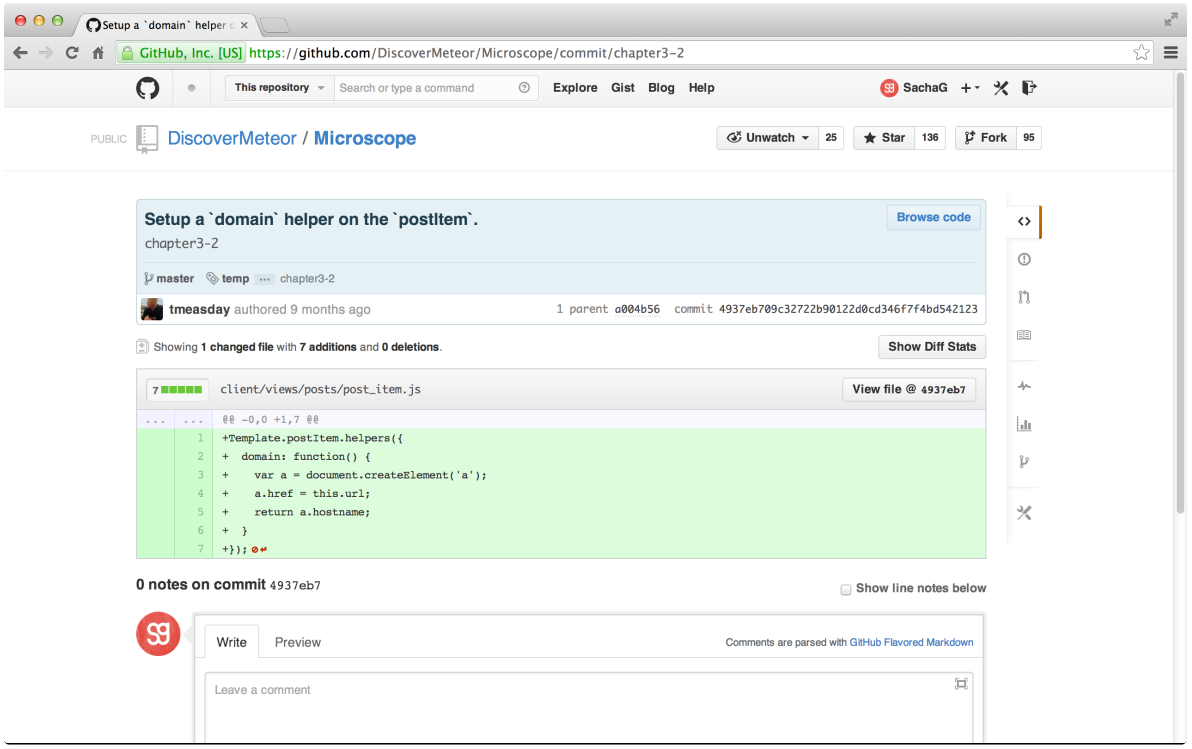
이 사이드바 장은 독자가 Git과 GitHub에 익숙하지 않다고 가정한다. 만약 독자가 이들에 익숙하다면 다음장으로 바로 넘어가도 된다!

커밋(Commit)하기

Git 저장소의 기본 작업단위는 커밋*(commit)*이다. 커밋은 주어진 시점에서의 코드 상태에 대한 스냅사진이라고 생각하면 된다.

우리는 단순하게 Microscope에 대한 완성된 코드를 제공하는 대신에, 각 단계별 스냅사진을 만들고 GitHub에서 이 모두를 온라인으로 볼 수 있도록 하였다.

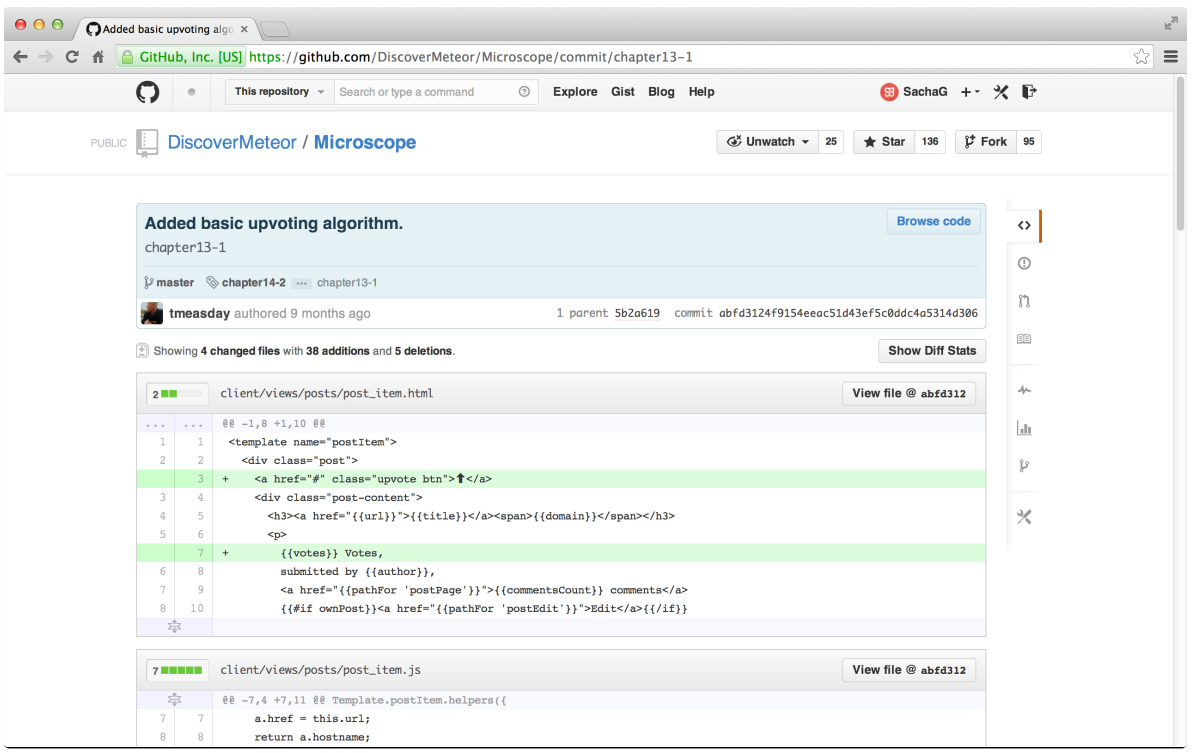
예를 들면, 이것은 이전 장의 마지막 커밋의 모습이다:



*GitHub*에서의 *Git* 커밋 화면.

아래에 보이는 것은 `post_item.js` 파일의 "diff" ("difference"를 의미)로서, 다른 표현으로는 이 커밋에 따른 변경 사항을 의미한다. 이 경우, `post_item.js` 파일은 처음부터 작성되었으므로, 그 내용 전체가 녹색으로 나타난다.

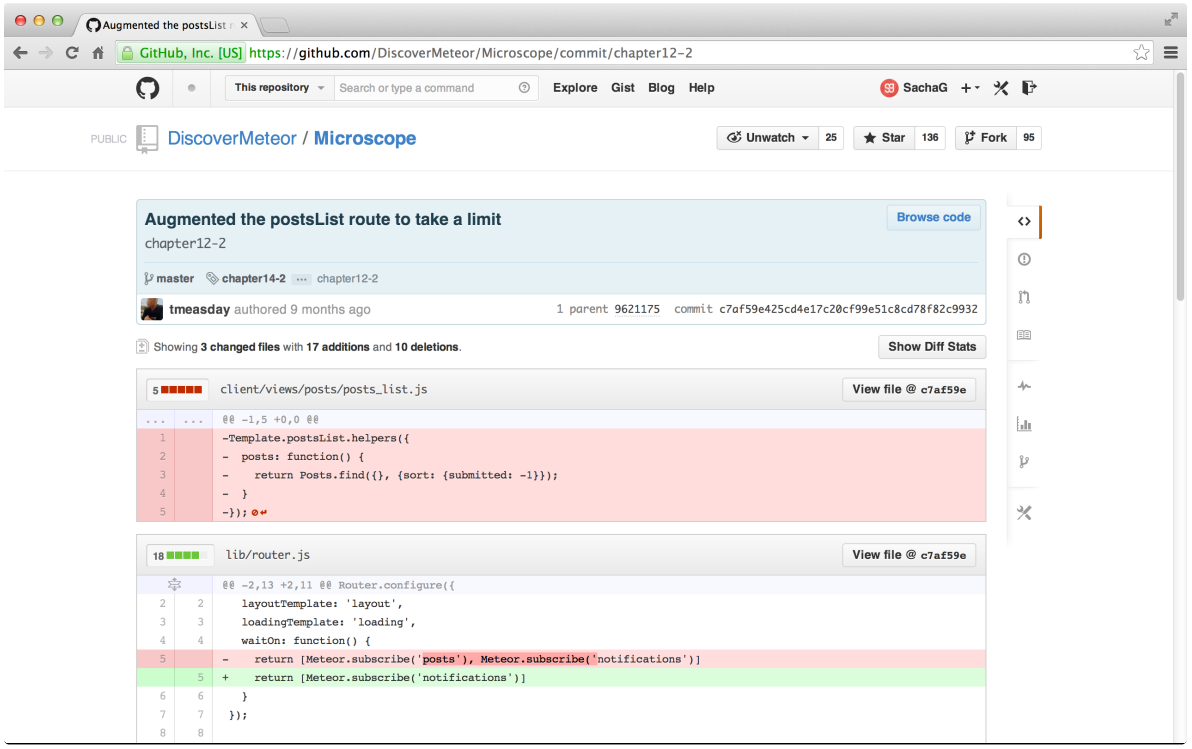
이 책의 나중 부분의 예제와 비교해 보기 바란다:



코드 수정.

이번에는, 수정된 라인들만 녹색으로 나타난다.

물론, 때로는 코드 라인을 추가하거나 수정하는 것이 아니라 삭제하기도 한다:



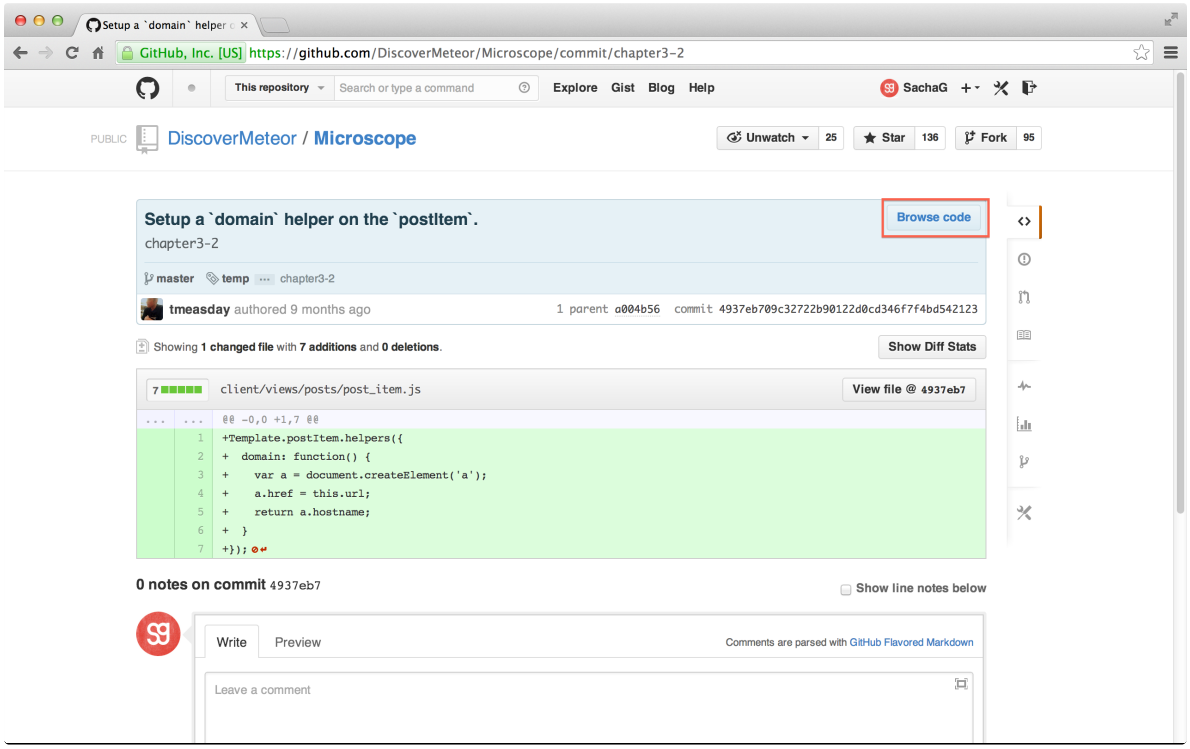
코드 삭제.

이와같이 우리는 GitHub를 처음 사용해보았다: 변경 사항을 한 눈에 보기

커밋한 코드 살펴보기

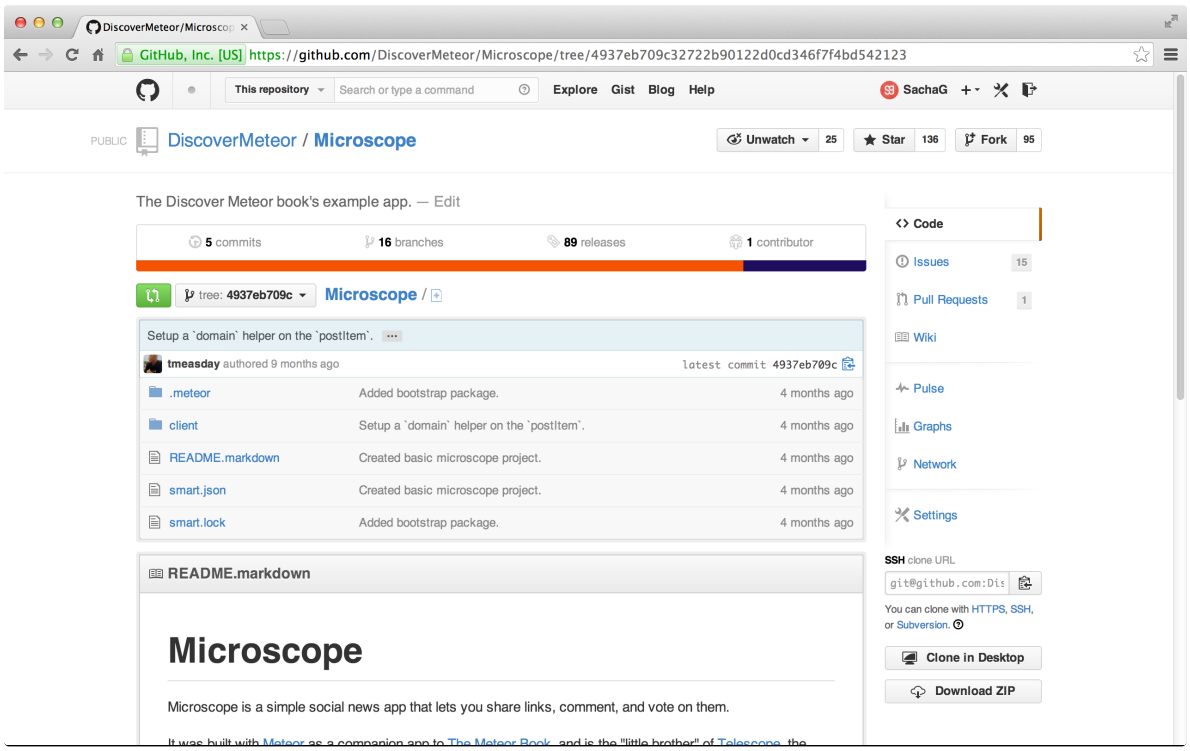
Git의 커밋 뷰는 이 커밋에 반영된 변경사항들을 보여주지만, 때때로 변경되지 않은 파일들을 보고 싶은 경우도 있는데, 이는 그 진행 단계에서 코드가 어떻게 보이는 지를 확인하기 위함이다.

다시 GitHub로 돌아가자. 커밋 페이지에 있을 때, **Browse code** 버튼을 눌러보라:



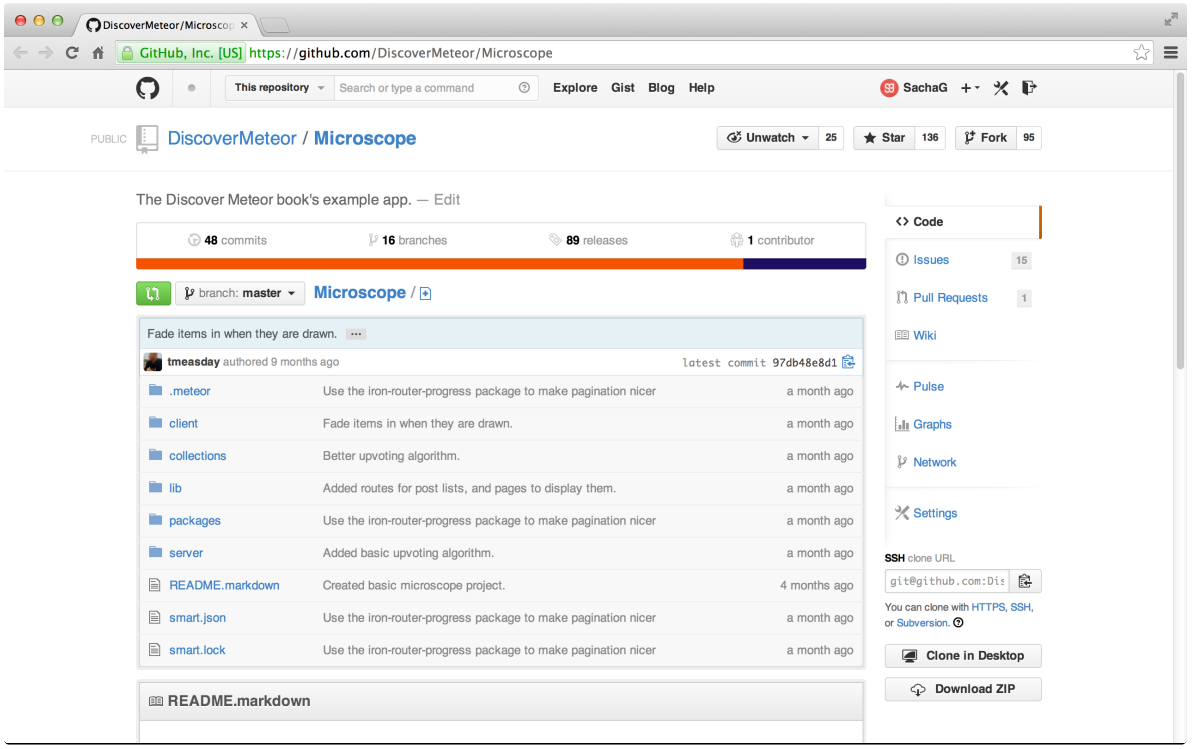
Browse code 버튼.

이제 특정한 커밋상태의 저장소에 접근할 수 있을 것이다.



*commit 3-2*의 저장소.

GitHub에서 우리가 해당 커밋에서 찾는 시각적 단서를 많이 주지는 않지만, “normal” master view와 비교하여 그 파일 구조가 달라진 것을 한 눈에 볼 수 있다.



*commit 14-2*의 저장소.

로컬로 커밋에 접근하기

지금까지 우리는 GitHub에서 커밋한 전체 코드를 온라인으로 살펴보는 방법을 알아보았다. 그러나 같은 일을 로컬에서 하고 싶다면 어떻게 해야 할까? 예를 들면, 특정한 커밋 상태에서 이것이 어떻게 작동하는 지를 보기 위하여 로컬에서 앱을 실행하고자 하는 경우가 있다.

이를 위해, 우리는 첫 단계(최소한 이 책에서는 그렇다)를 `git` 커맨드 라인 유틸리티로 시작할 것이다. 우선, **Git**이 설치된 것을 확인하라. 그리고 Microscope repository를 복제하라(바꾸어 말하면, 복사본을 로컬에 다운로드 하라).

```
$ git clone git@github.com:DiscoverMeteor/Microscope.git gi
thub_microscope
```

끝부분의 `github_microscope` 는 앱을 복제하여 저장할 로컬 디렉토리의 이름이다. 이미 `microscope` 라는 이름의 디렉토리가 존재한다면, 그저 다른 이름으로 바꾸면 된다(GitHub repo에 있는 이름과 똑같을 필요는 없다).

`git` 커맨드 라인 유틸리티를 사용할 수 있도록 `cd` 명령으로 저장소 디렉토리로 이동한다:

```
$ cd github_microscope
```

이제 GitHub로부터 저장소를 복제하여 그 앱의 모든 코드를 다운로드하였다. 이는 커밋한 마지막 코드를 보고 있다는 의미이다.

고맙게도, 다른 것들에는 영향을 주지 않으면서 시간을 되돌려 특정한 커밋을 "check out" 하는 방법이 있다. 한 번 시도해보자:

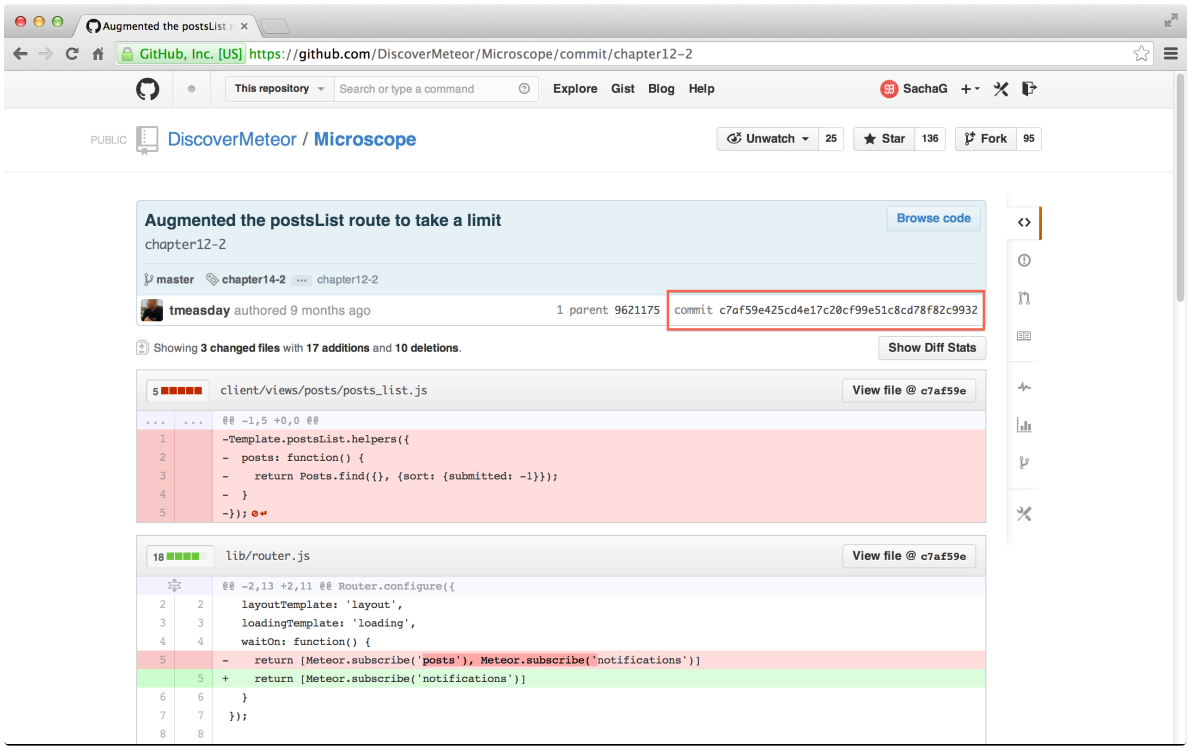
```
$ git checkout chapter3-1
Note: checking out 'chapter3-1'.

You are in 'detached HEAD' state. You can look around, make
 experimental
changes and commit them, and you can discard any commits yo
u make in this
state without impacting any branches by performing another
checkout.

If you want to create a new branch to retain commits you cr
eate, you may
do so (now or later) by using -b with the checkout command
again. Example:

  git checkout -b new_branch_name

HEAD is now at a004b56... Added basic posts list template a
nd static data.
```

Git은 우리가 "detached HEAD" 상태에 있다는 것을 알려주고 있는데, 이는 Git에 관한 한, 지나간 커밋을 볼 수는 있지만 수정할 수는 없다는 것을 의미한다. 마치 이것은 유리구슬을 통해서 과거를 보는 마법사에 비유할 수 있다.

(Git에는 과거의 커밋을 변경할 수 있는 명령어들도 있다는 점에 유의하기 바란다. 이는 마치 시간 여행자가 과거로 돌아가서 나비를 밟는 것과 같은 일이 일어날 수 있다는 의미이나 이것은 여기서의 간단한 소개의 범위를 넘어선다.)

우리가 단순하게 `chapter3-1` 이라고 입력할 수 있었던 이유는 모든 Microscope의 커밋마다 각 장에 대한 표식으로 미리 꼬리표를 달아 놓았기 때문이다. 이렇게 하지 않았다면, 먼저 해당 커밋의 해시**(hash)**나 또는 unique identifier를 찾아야 했을 것이다.

다시 한 번, GitHub는 우리의 삶을 보다 편안하게 해준다. 커밋의 해시는 파란 커밋 헤더 박스의 오른쪽 하단에서 찾을 수 있다:



Commit hash 찾기.

그러면 태그 대신 해시로 시도해보자:

```
$ git checkout c7af59e425cd4e17c20cf99e51c8cd78f82c9932
Previous HEAD position was a004b56... Added basic posts lis
t template and static data.
HEAD is now at c7af59e... Augmented the postsList route to
take a limit
```

그리고 마지막으로, 우리가 마술 유리구슬을 보는 것을 멈추고 현재로 돌아오려면 어떻게 해야 할까? Git에게 **master** branch를 `check out` 하고 싶다고 말하면 된다:

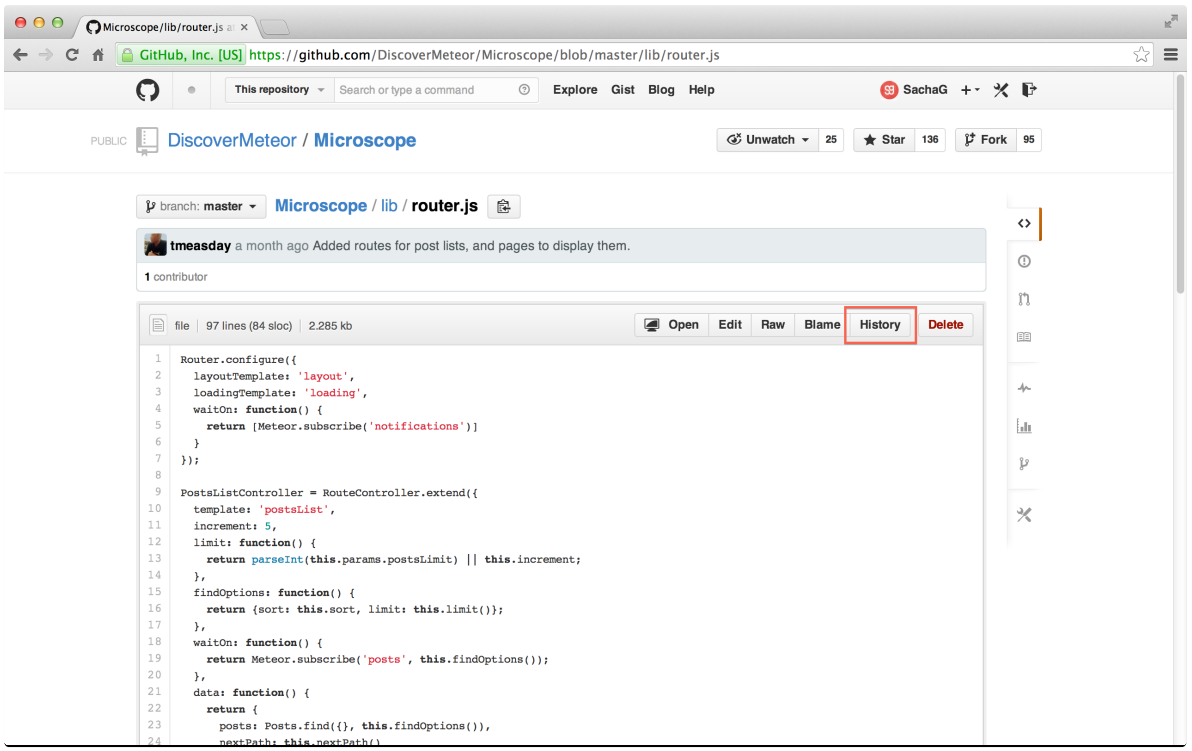
```
$ git checkout master
```

이 과정의 어느 시점에서나, "detached HEAD" 상태에서도, 앱을 `meteor` 명령어로 실행시킬 수도 있다. 만약 Meteor가 패키지가 누락되었다는 메시지를 보이면 `meteor update` 명령을 바로 실행시키면 되는데, 이것은 Microscope의 Git 저장소에는 패키지가 포함되어 있지 않기 때문이다.

History 관점

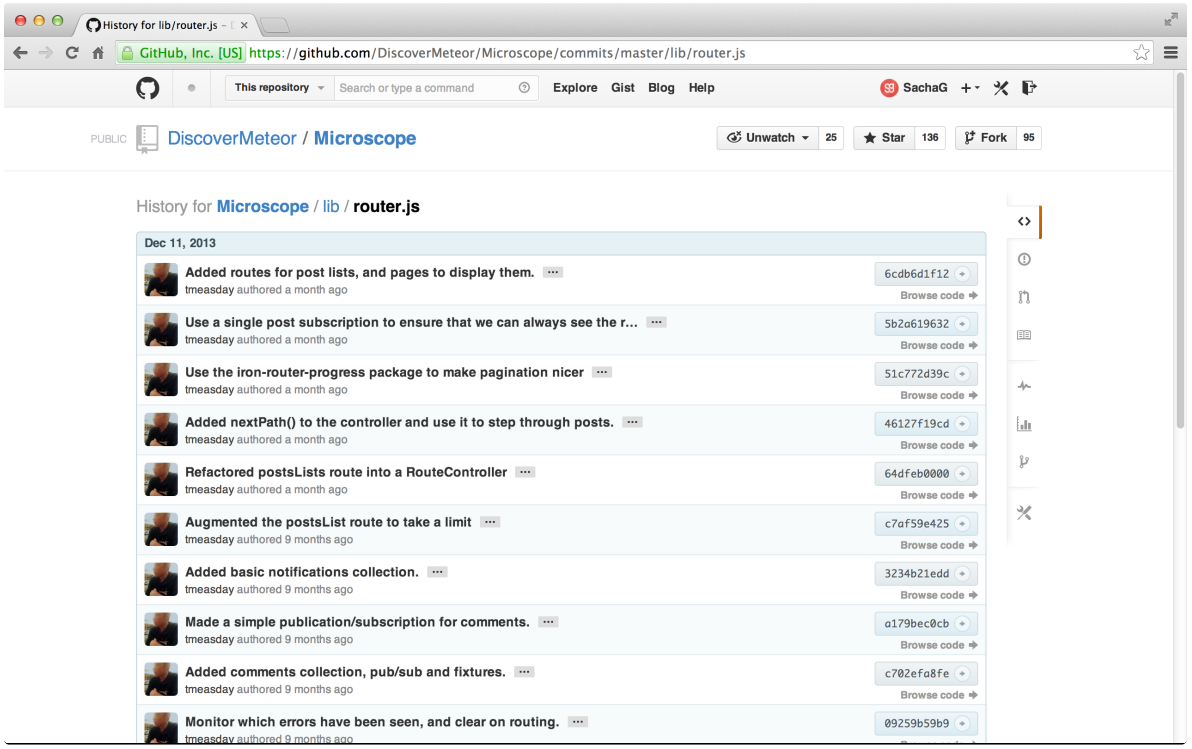
또 다른 일반적인 시나리오가 있다: 독자가 어떤 파일을 보고 있는데 독자가 전에 보지 못했던 변경 사항이 있는 것을 발견했다. 문제는 언제 파일이 변경되었는지 기억을 못한다는 것이다. 그러면 올바른 버전을 찾을 때까지 각 커밋을 하나씩 찾아보려고 할 것이다. 하지만, GitHub의 **History** 기능 덕분에 쉽게 할 수 있다.

우선, GitHub에 있는 저장소의 파일중의 하나에 접근하여, "History" 버튼을 누른다:



GitHub의 History 버튼.

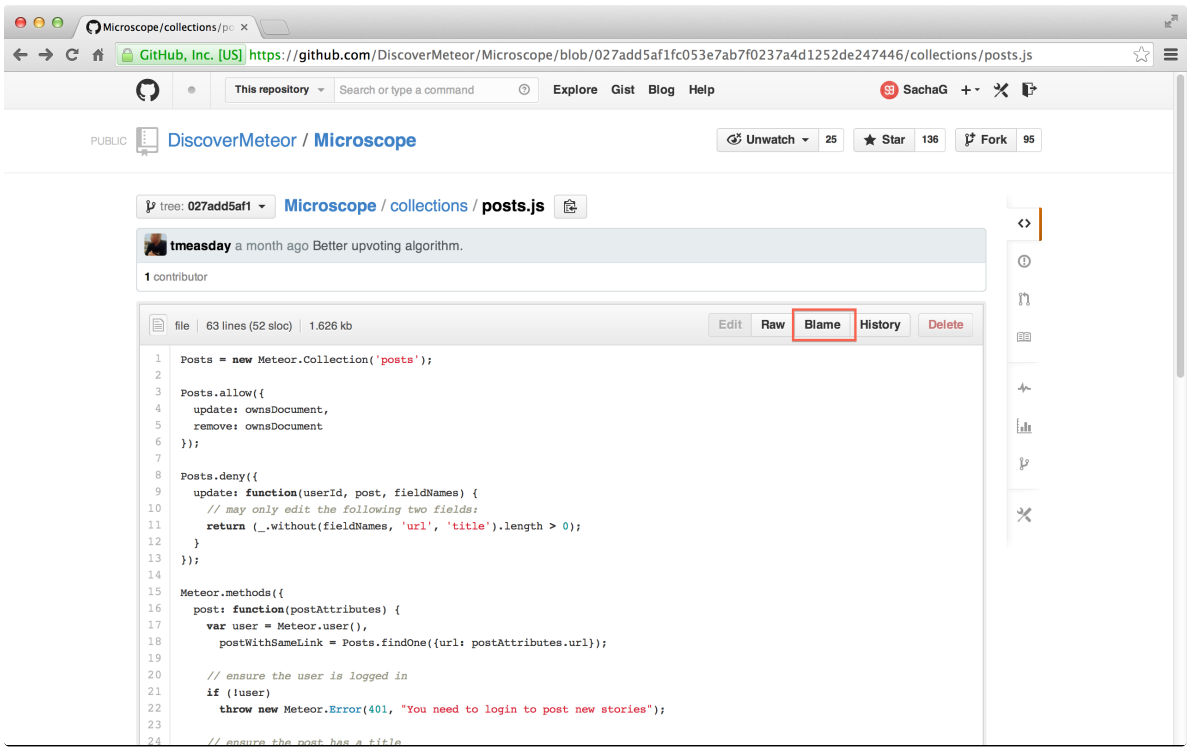
이제 이 특정한 파일에 영향을 준 모든 커밋 목록이 정돈되어 나타난다.



파일의 *history* 보기.

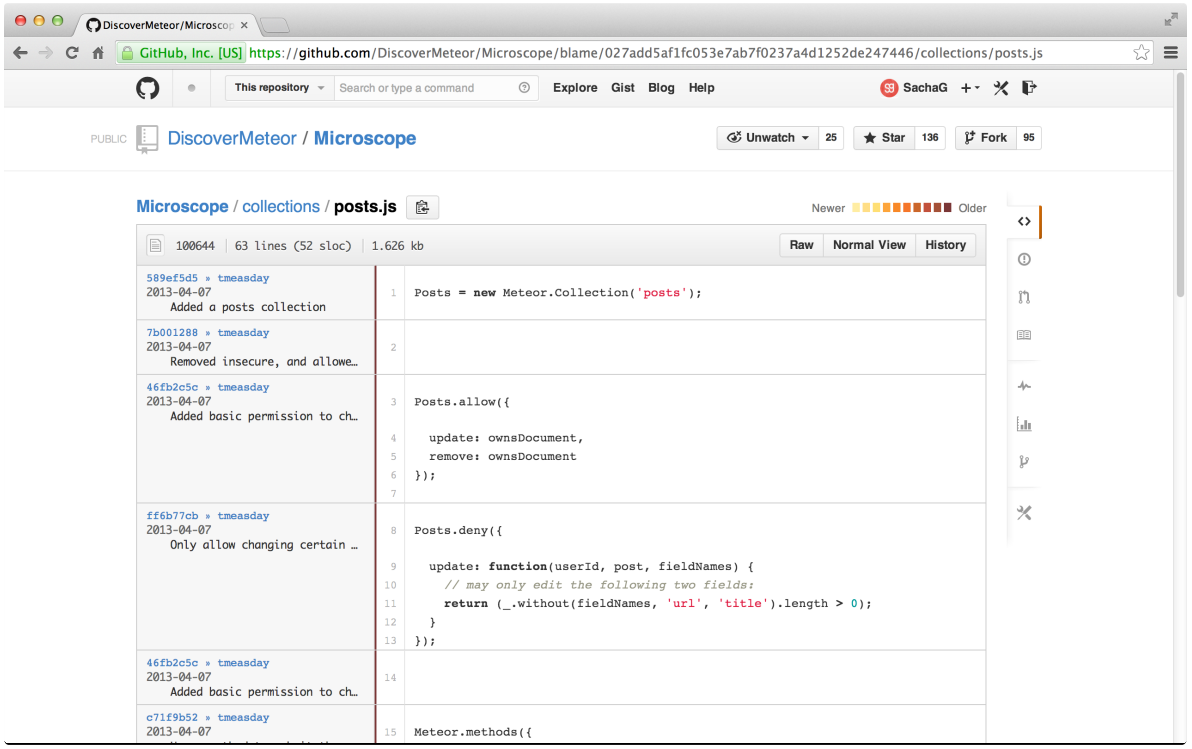
책임 공방(Blame Game)

마무리 단계로 **Blame**을 살펴보자:



*GitHub*의 *Blame* 버튼.

이 깔끔한 뷰는 어느 커밋에서, 어떤 파일의 라인마다 누가 수정하였는 지를(바꾸어 말하면, 누구의 잘못으로 작동하지 않는 지를) 보여준다:



GitHub의 Blame 열람.

Git은 매우 복잡한 도구이다 - GitHub도 그렇다 -, 그러므로 이 단일 장에서 모두 다루기를 기대할 수는 없다. 사실, 이들 도구로 가능한 것들에 대한 겉핥기만 한 것이다. 하지만, 바라건데, 이 정도만으로도 이 책의 나머지를 따라가는데에 도움이 될 것이다.