**HƯỚNG DẪN SỬ DỤNG GITHUB**

# **I Giới thiệu về git và github**

## **1 Git là gì ?**

**Git** là tên gọi của một **Hệ thống quản lý phiên bản phân tán** (*Distributed Version Control System –****DVCS***) là một trong những hệ thống quản lý phiên bản phân tán phổ biến nhất hiện nay. DVCS nghĩa là hệ thống giúp mỗi máy tính có thể lưu trữ nhiều phiên bản khác nhau của một mã nguồn được nhân bản (**clone**) từ một kho chứa mã nguồn (**repository**), mỗi thay đổi vào mã nguồn trên máy tính sẽ có thể ủy thác (**commit**) rồi đưa lên máy chủ nơi đặt kho chứa chính. Và một máy tính khác (nếu họ có quyền truy cập) cũng có thể clone lại mã nguồn từ kho chứa hoặc clone lại một tập hợp các thay đổi mới nhất trên máy tính kia. Trong Git, thư mục làm việc trên máy tính gọi là **Working Tree**.

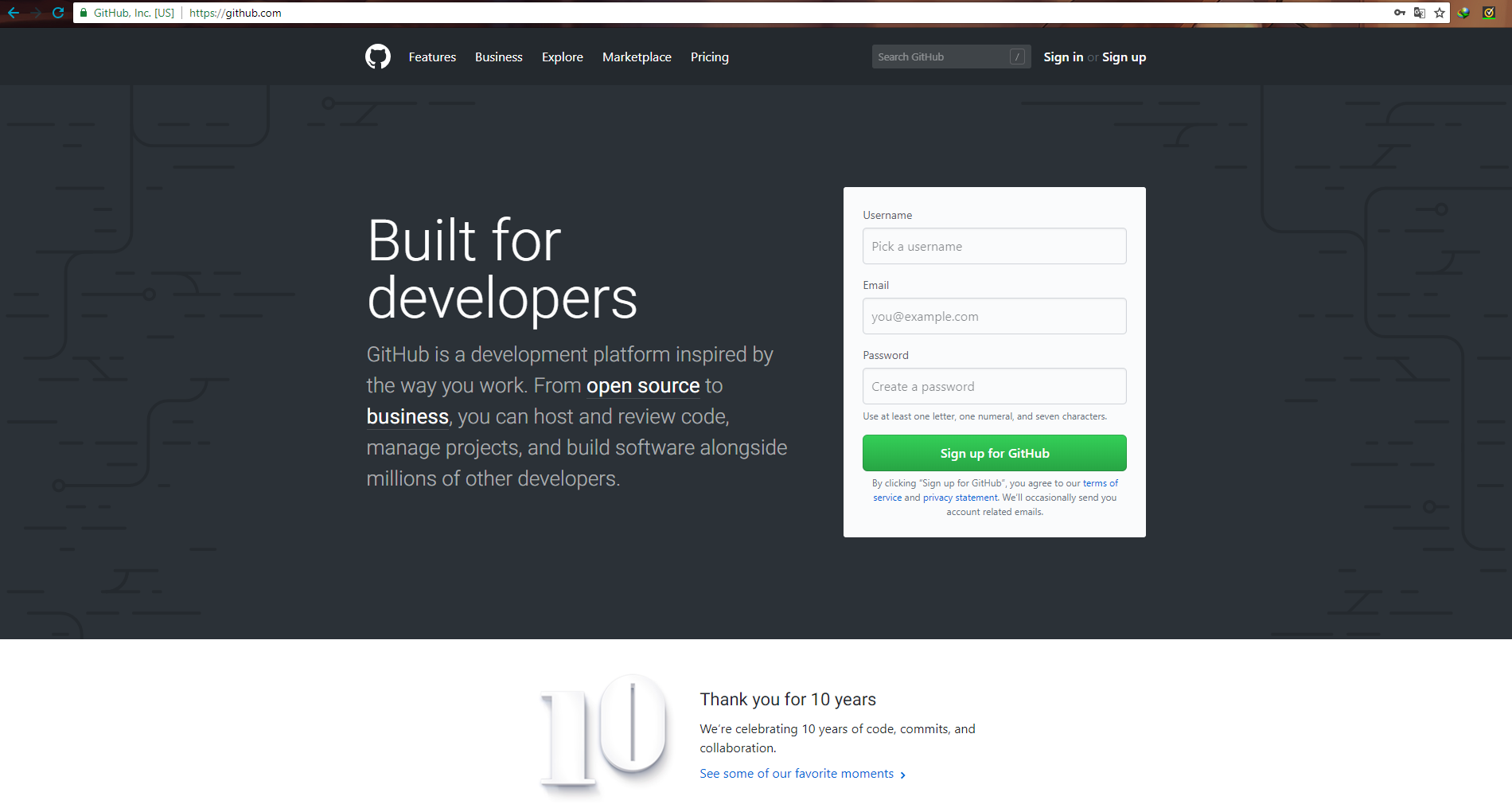
[](https://thachpham.com/wp-content/uploads/2015/04/dvcs.png)

Ngoài ra, có một cách hiểu khác về Git đơn giản hơn đó là nó sẽ giúp bạn lưu lại các phiên bản của những lần thay đổi vào mã nguồn và có thể dễ dàng khôi phục lại dễ dàng mà không cần copy lại mã nguồn rồi cất vào đâu đó. Và một người khác có thể xem các thay đổi của bạn ở từng phiên bản,  họ cũng có thể đối chiếu các thay đổi của bạn rồi gộp phiên bản của bạn vào phiên bản của họ. Cuối cùng là tất cả có thể đưa các thay đổi vào mã nguồn của mình lên một kho chứa mã nguồn.

Cơ chế lưu trữ phiên bản của Git là nó sẽ tạo ra một “*ảnh chụp*” (*snapshot*) trên mỗi tập tin và thư mục sau khi commit, từ đó nó có thể cho phép bạn tái sử dụng lại một ảnh chụp nào đó mà bạn có thể hiểu đó là một phiên bản. Đây cũng chính là lợi thế của Git so với các DVCS khác khi nó không “lưu cứng” dữ liệu mà sẽ lưu với dạng snapshot.

## **2 Github là gì ?**

Git và github là không phải là một và có thể nhiều bạn sẽ bị hiểu lầm. Git là tên gọi của một mô hình hệ thống và như đã giải thích ở trên, **các máy tính có thể clone lại mã nguồn từ một repository** và **Github chính là một dịch vụ máy chủ repository** công cộng, mỗi người có thể tạo tài khoản trên đó để tạo ra các kho chứa của riêng mình để có thể làm việc.



## **3 Mô hình quản lý phân tán của git**

Quản lý phân tán của [Git](http://expressmagazine.net/taxonomy/term/308)là một repositories không cần có chung một nơi để lưu trữ, mà mỗi thành viên sẽ có một repository ở local của họ. Tất cả thao tác ta làm việc với Git đều ở trên máy của ta, local repository, khi quyết định đưa những thay đổi đó lên server ta chỉ cần một thao tác "push" nó lên server. Chúng ta vẫn có thể share thay đổi của chúng ta cho thành viên khác, bằng cách commit hoặc update trực tiếp từ máy của họ mà không phải thông qua repositories gốc trên server (thông qua share ssh cho nhau). Và dĩ nhiên là mọi thao tác đều mang theo thông tin history với Git.

Tính phân tán thì an toàn hơn so với tập trung, vì mỗi bản copy của thành viên đều là full copy từ repository gốc, khi server bị down, các thành viên vẫn có thể làm việc offline, họ vẫn có thể commit và update trên local của họ hoặc thậm chí với nhau mà không cần thông qua server. Khi server hoạt động trở lại, họ có thể cập nhật tất cả lên lại server.

## **4 Cách thức làm việc với github**

Làm việc với GitHub nói riêng hay hệ thống GIT nói chung có 2 workflow chính là local workflow và server workflow.

Bạn có thể làm mọi chuyện thay đổi source code ở local, sau khi đã thay đổi xong, bạn sẽ commit nhưng thay đổi đó lên server và bản lên server phải là bản hoàn chỉnh một tính năng nào đó, hoặc fix bug xong, test xong hoặc ít nhất bản đó phải chạy được. Không được commit code dở dang, chưa qua test lên repository server sẽ làm ảnh hưởng đến các thành viên khác, ngược lại bạn có thể làm điều đó ở repository local (Bạn cũng có thể tạo một branch ở server cho việc commit code dở dang hay tính năng chưa hoàn thành, nó sẽ chiếm space ở server cũng như làm mất thời gian của bạn vào việc tương tác kết nối với server, vậy nên commit nó lên repository local vừa nhanh thao tác lại không mất space của server.)

1) Làm việc với r**epository** ở local: với 2 command thường dùng là git add và git commit

+ Git add: add file đã thay đổi vào stage

Stage là nơi bfile đã thay đổi vào stagegit add và git commitry vào trưi bfile đã thay đổi vào stagegit add và git commit xét l i những thay đổi chắc chắn đã đúng hay chưa, cân nhắc lần cuối trước khi commit).

+ Git commit: commit các file đã add vào stage lên repository local.

2) Làm việc với repository ở server github:

Sau khi đã làm xong ở local , cuối cùng khi có một bản ổn định và hoàn tất (có thông qua test) ta sẽ quyết định cập nhật nó lên repository server với:

+ push: push thay đổi từ repository local lên repository server

+ fetch: cập nhật thay đổi từ repository server về repository local

+ pull: sao chép source code từ server về local workspace

## **5 Tại sao nên sử dụng github?**

Một số lợi ích từ việc sử dụng github có thể kể đến ở đây như :

+ Git dễ dàng sử dụng, an toàn và nhanh chóng

+ Có thể khiến quy trình làm việc code theo nhóm trở nên đơn giản hơn bằng việc kết hợp các phân nhánh

+ Có thể làm việc ở bất cứ đâu vì chỉ cần clone mã nguồn từ kho chứa.

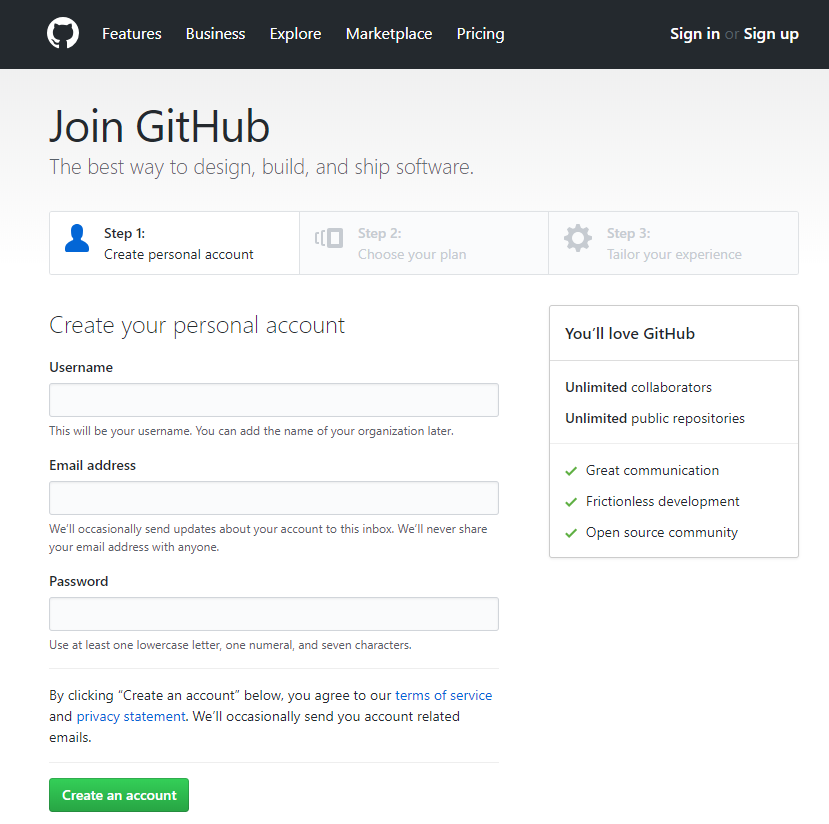
+ Dễ dàng trong việc triển khai sản phẩm

# **II Hướng dẫn sử dụng github**

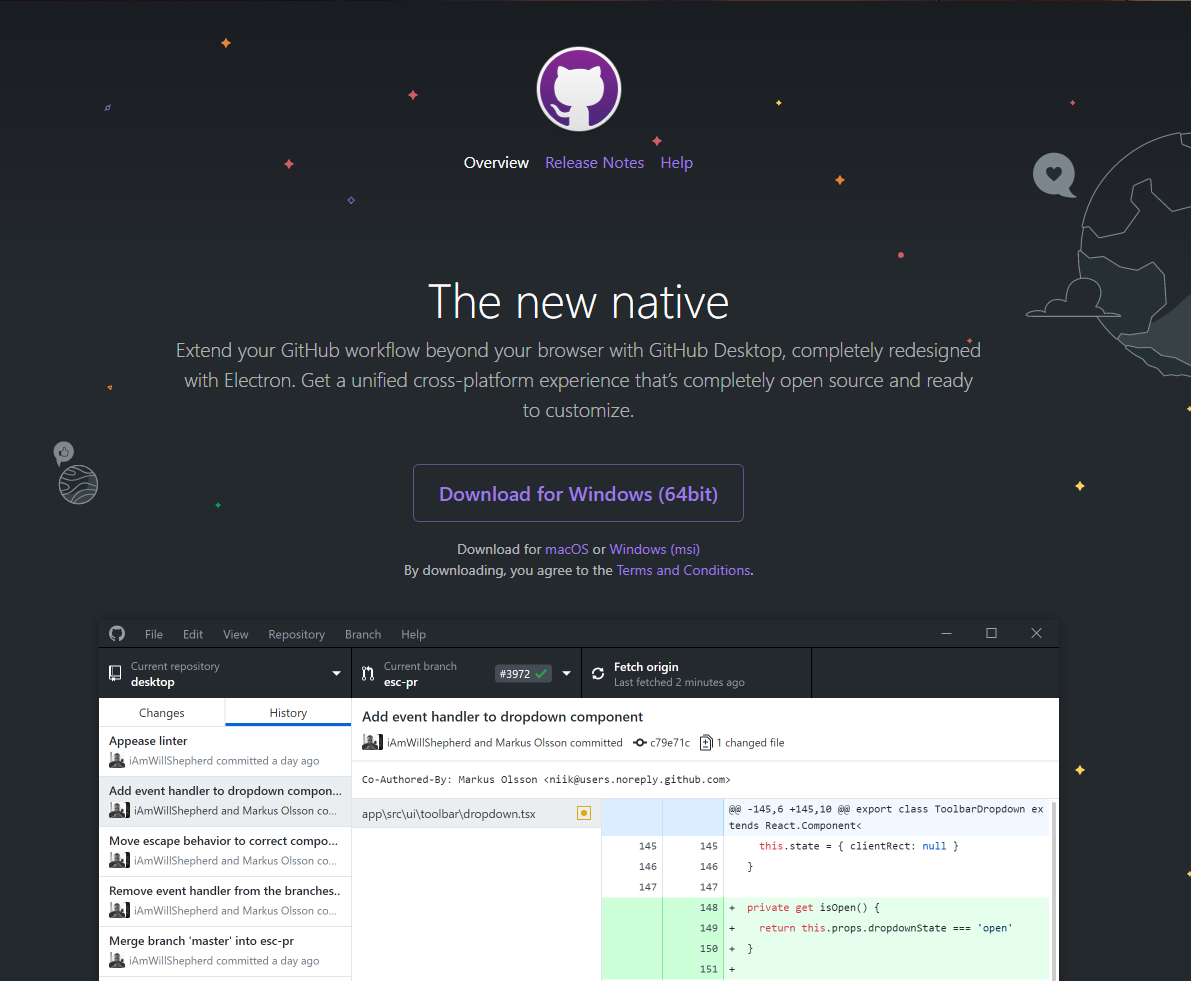
## **1 Bước chuẩn bị**

+ Đầu tiên đăng ký một tài khoản github

>>> Link đăng ký tài khoản github: <https://github.com/join>



+ Sau đó cài đặt Github for windows ( GFW ) để có thể sử dụng trên máy tính của mình



>>> Link tải GFW: <https://desktop.github.com/>

## **2 Sử dụng github**

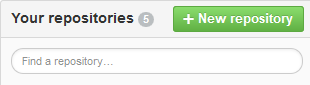
Github có 2 cách để sử dụng :

+ Thứ nhất là tạo một kho lưu code ( repositories ) cho project của mình

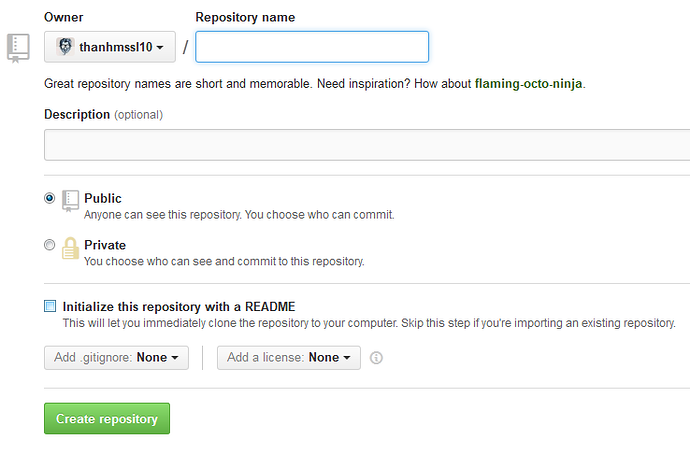
+ Thứ hai là có thể tham gia vào một project nào đó

### **2.1 Tạo repositories**

Trên giao diện web của github ta ấn nút New Repository

  
**New Repository**

Sau khi ấn chọn **New Repository** sẽ ra giao diện như dưới:

[](https://daynhauhoc.s3-ap-southeast-1.amazonaws.com/original/2X/7/76a4800ae5002b93adffbf520f94c13ef214f265.png)

Trong đó:

+ **Repository name:** là tên dự án mà bạn muốn đặt

+ **Description:** miêu tả chi tiết dự án, có thể điền hoặc không

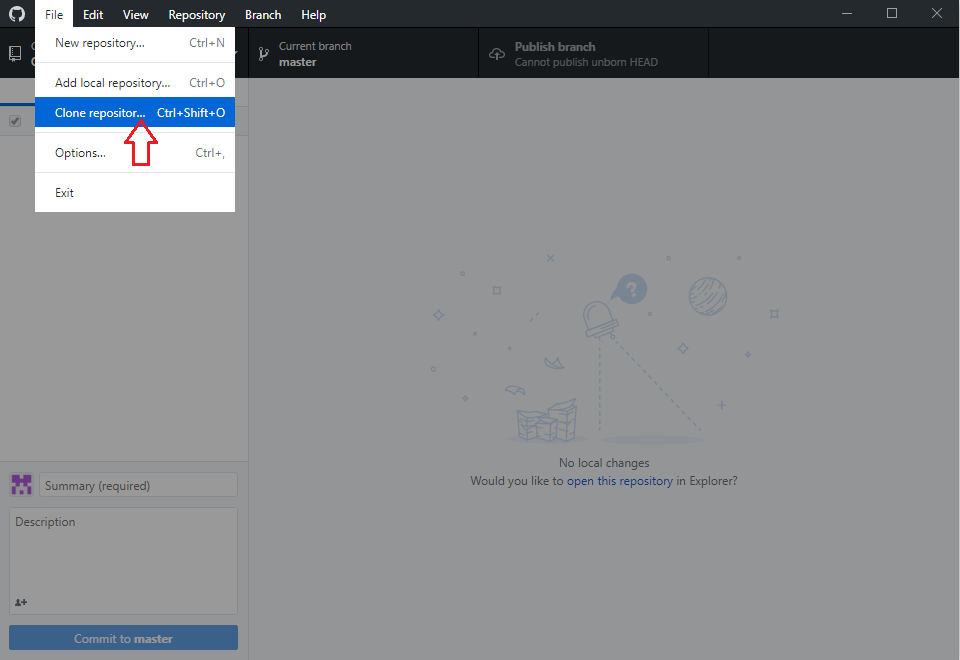
+ **Public or Private:** nếu chọn chế độ private thì người khác sẽ không thấy code của bạn nhưng bạn sẽ phải tốn phí cho nó.

+ **Intialize this repository with a README:** Có thể chọn hoặc không

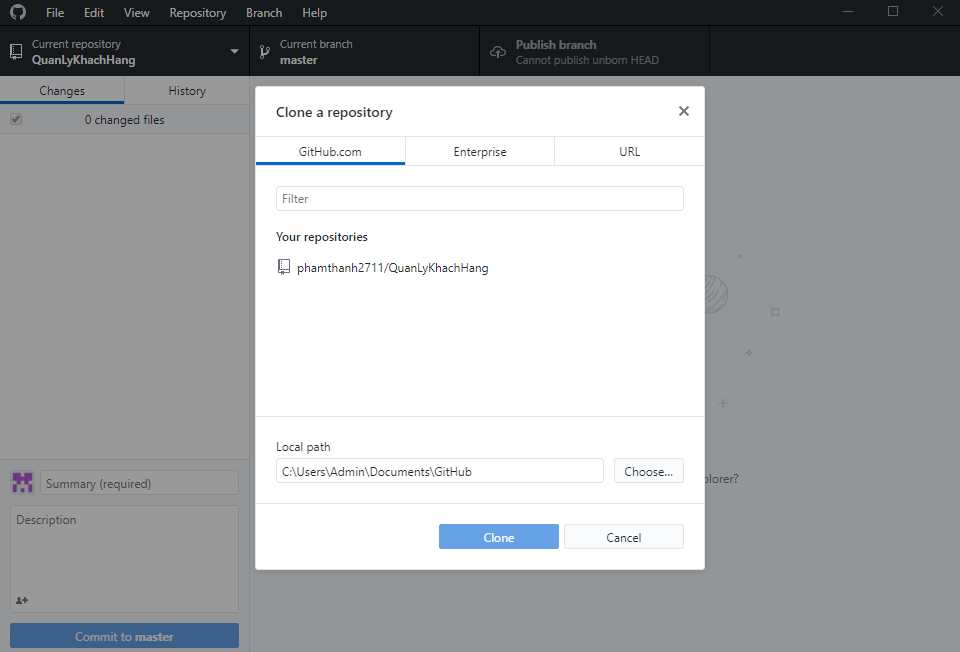
Sau khi điền xong thông tin bạn ấn create repository là xong. Bạn sẽ thu được remote repository vừa tạo, và link này có dạng **git@github.com:user-name/repository-name.**

### **2.2 Tạo clone repository trong github desktop**

Mở Github for Window để tạo clone repository về máy tính

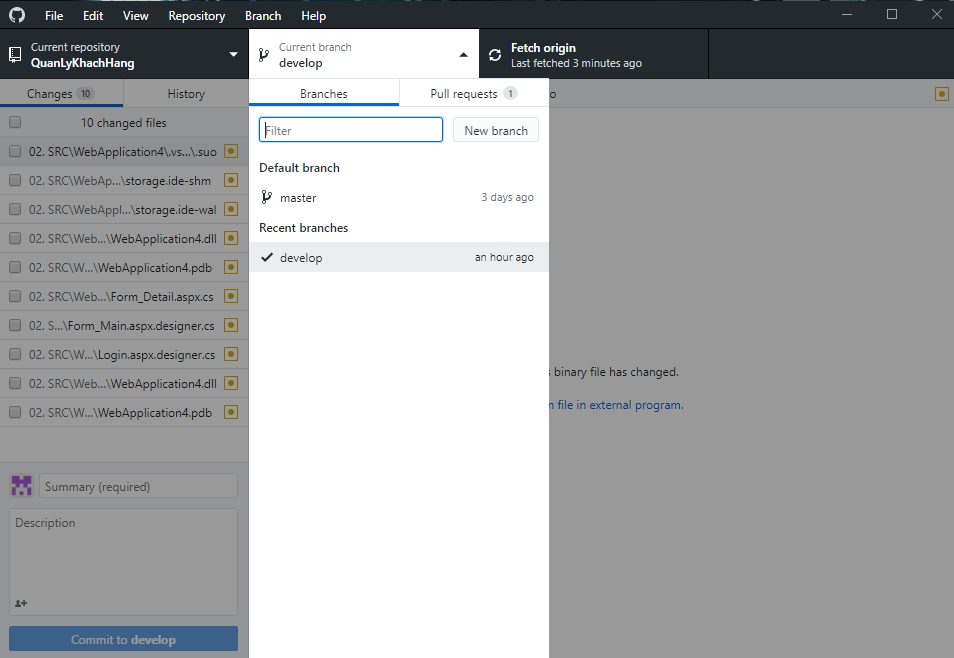


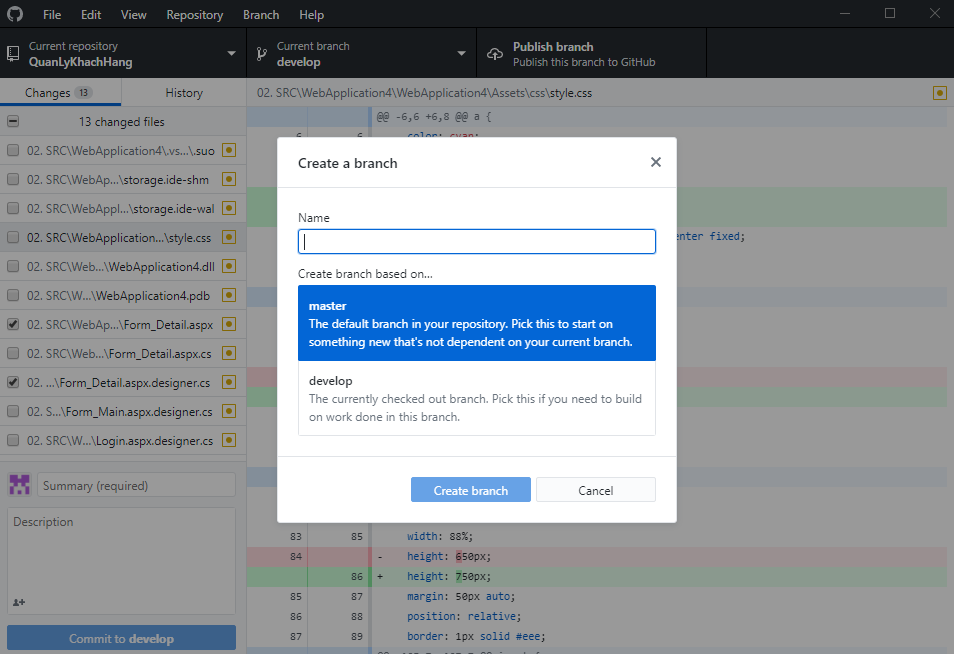
Sau khi tạo clone repository xong thì sẽ xuất hiện



Và từ bây giờ bạn có thể cập nhật các dự án của mình vào trong thư mục github trên máy tính

### **2.3 Tạo phân nhánh trong github desktop**





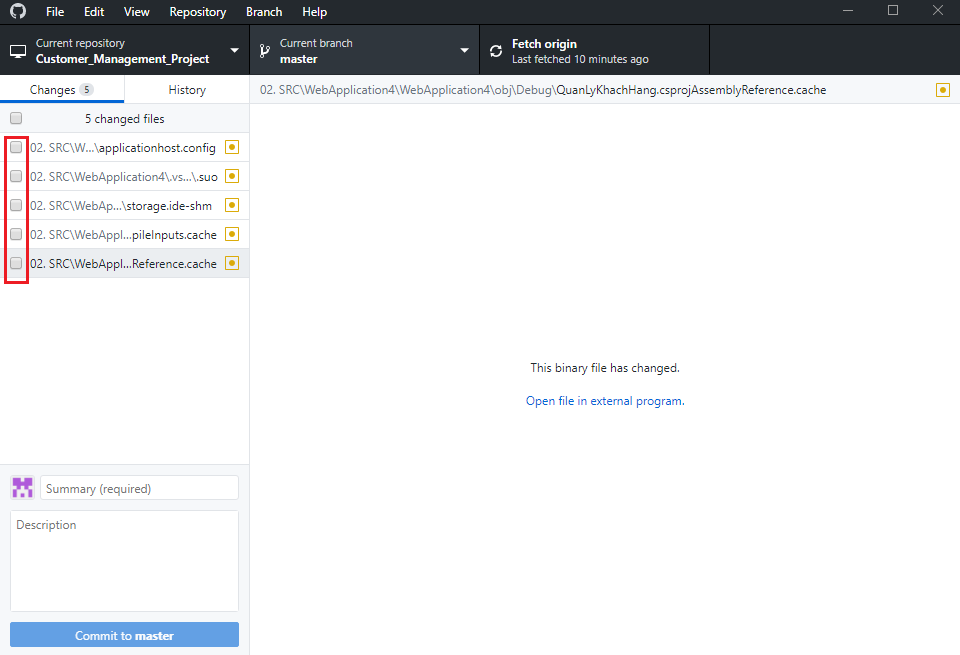
Chọn new branch và sau đó sẽ hiện lên box **create a branch**.

Trong đó:

+ Name: điền tên của nhánh định tạo.

+ Create branch based on: Chọn nhánh cha của nhánh định tạo.

### **2.4 Đưa những thay đổi lên github**



Khi có bất cứ sự thay đổi gì trong project thì sẽ được hiện lên trong ô danh sách bên trái

Bạn có thể tích vào check box để chọn những file thay đổi sẽ được đưa lên github.

Sau đó điền tiêu đề tóm tắt vào ô textbox summary bên dưới và kích nút commit to master.

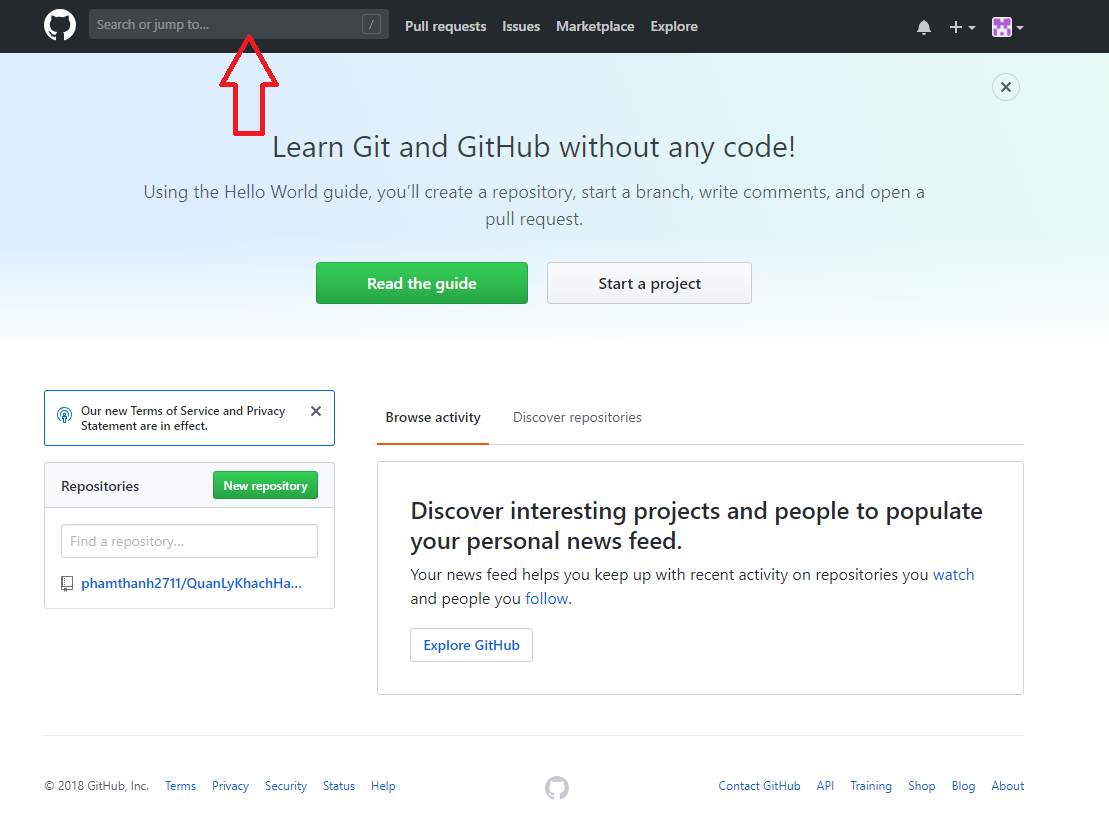
Cuối cùng nhấn vào fetch origin để đưa commit lên trên github.

Lưu ý:

* Commit to master mới chỉ update trạng thái trên local repository của bạn

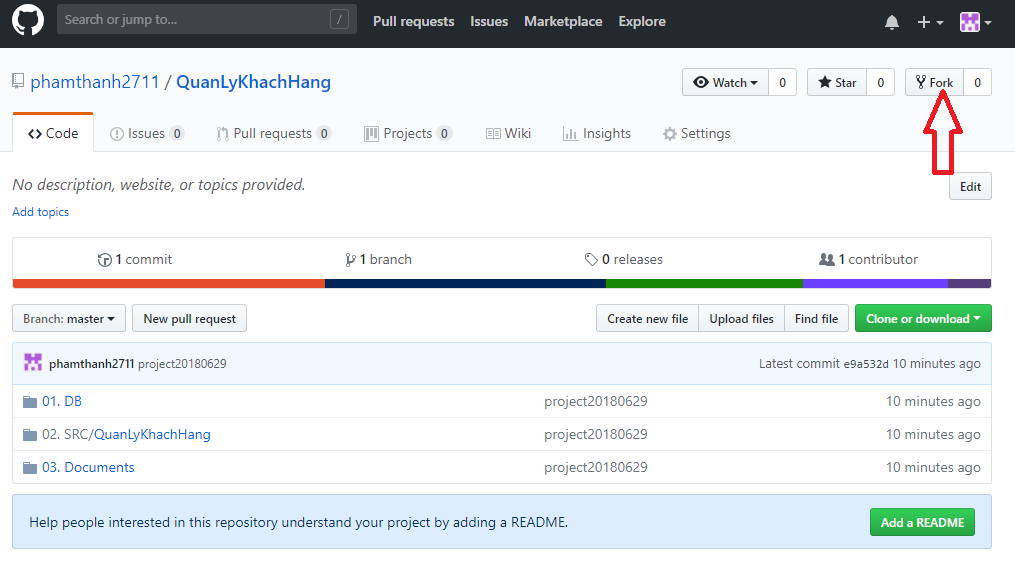
- Fetch origin để update thay đổi lên repository trên github.com

### **2.5 Tham gia vào một project khác**



Để tham gia vào một dự án khác thì trên ô tìm kiếm bạn gõ tên dự án.

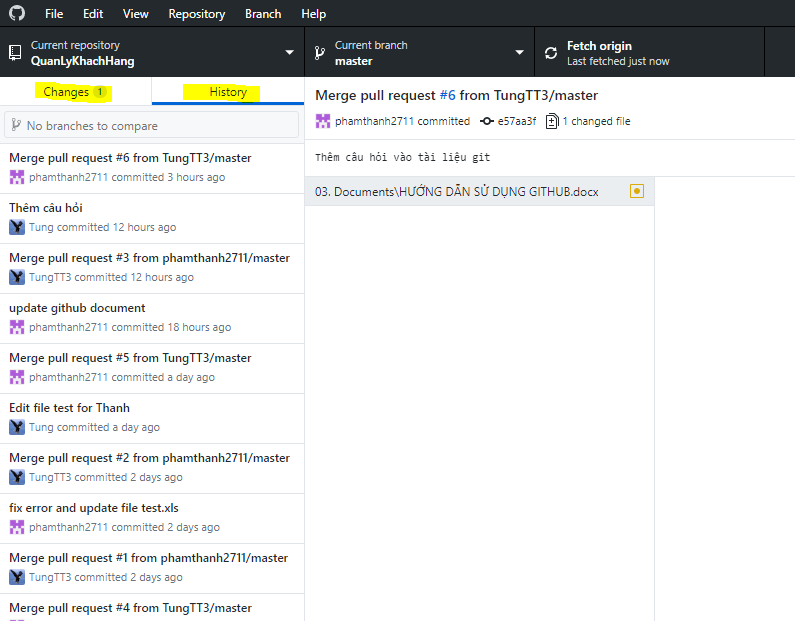
VD: để tham gia dự án của mình , bạn gõ phamthanh2711/QuanLyKhachHang ,sau đó chọn fork

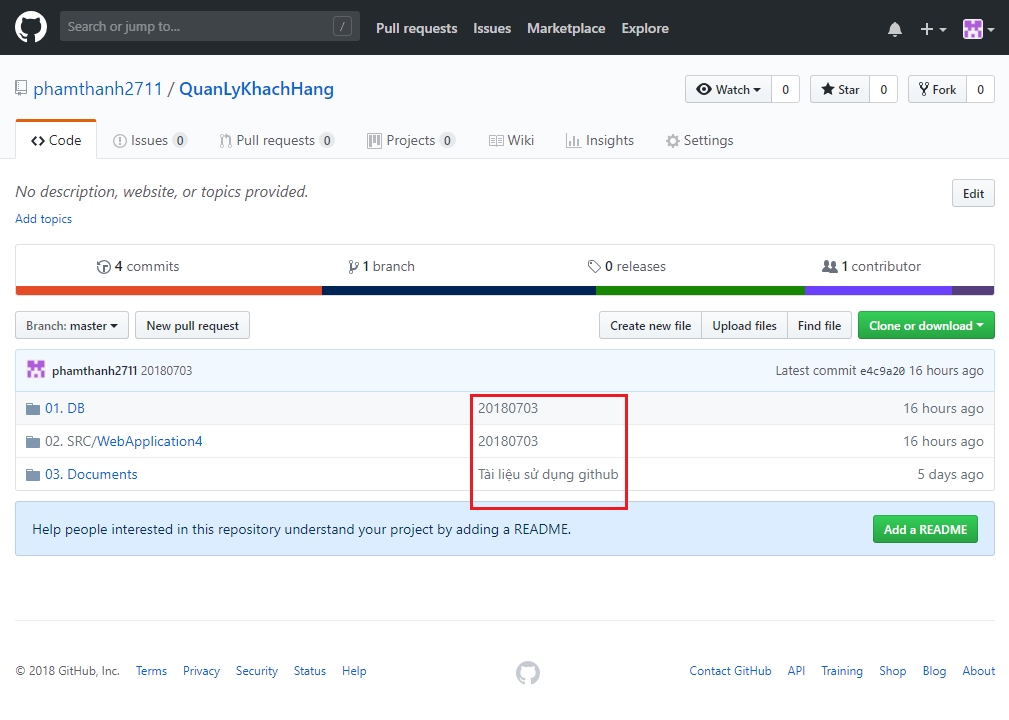


Sau khi fork thì bạn đã copy được dự án về repo online của mình. Giờ muốn lấy project về local repo trên máy tính đơn giản chỉ cần tạo clone để pull project về ( Xem hướng dẫn cách clone ở mục 2.2 ).

### **2.6 Xem những thay đổi trong project**

Trên github desktop bạn có thể dễ dàng xem những thay đổi mới nhất trên working folder khi nhấn vào Changes hoặc có thể xem cả những commit chứa những thay đổi trước đó khi nhấn vào History.



Còn khi thay đ mới nhất trên working folder khi nhấn vào Changes hoặc có thể xem cả những commit

Vùng khoanh đỏ trên đây là những thay đổi mới nhất của project và bên cạnh là thời gian thay đổi được đưa lên github.

Hoặc có thể nhấn vào commits để xem cả những thay đổi được đưa lên trước đó.



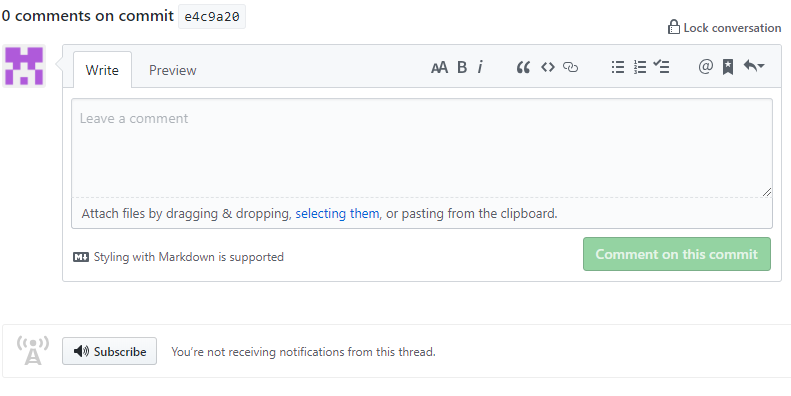
Sau khi nhấn vào , bạn có thể thấy được các sự thay đổi trong chương trình

Trong đó :

+ Trên cùng là tên file có sự thay đổi, bên trái là tương quan tỉ lệ giữa sự thêm và bớt, được thể hiện bằng các ô chấm màu xanh(thêm vào) và màu đỏ(bớt đi)

+ Những dòng có dấu “+” là những dòng mới được thêm vào

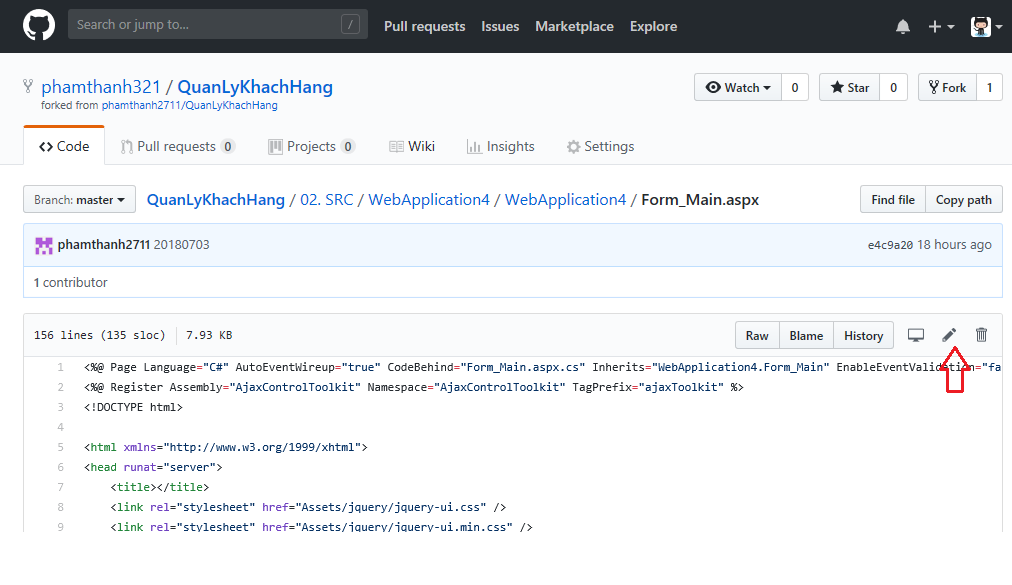
+ Những dòng có dấu “-” là những dòng mới được bớt đi



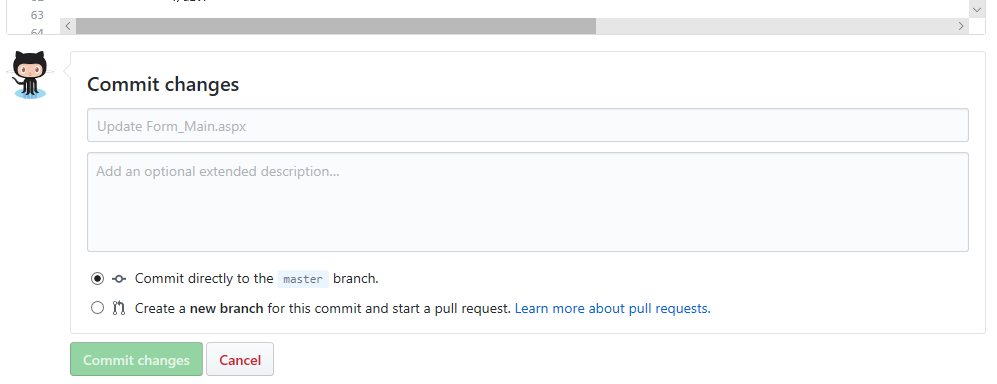
Kéo xuống dưới cùng sẽ có một box cho phép các bạn có thể để lại comment cho commit này.

### **2.7 Cách pull request**

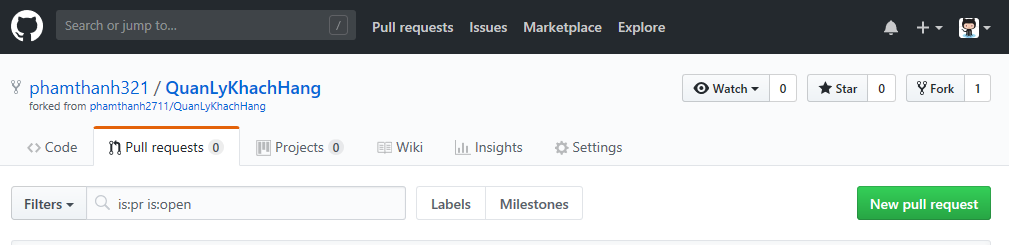
Bây giờ bên B (người tham gia dự án) nếu chỉnh sửa file, khi ấy file đó giữa người A( người tạo repository ban đầu) và B đã có sự khác nhau và B có thể gửi pull request (PR) tới master. PR nghĩa là partner nói với master:" B đã sửa file, yêu cầu A hợp nhất vào repo của A", nếu như A đồng ý với yêu cầu đó,thì file trên repo của A sẽ thay đổi theo file trên repo của B.

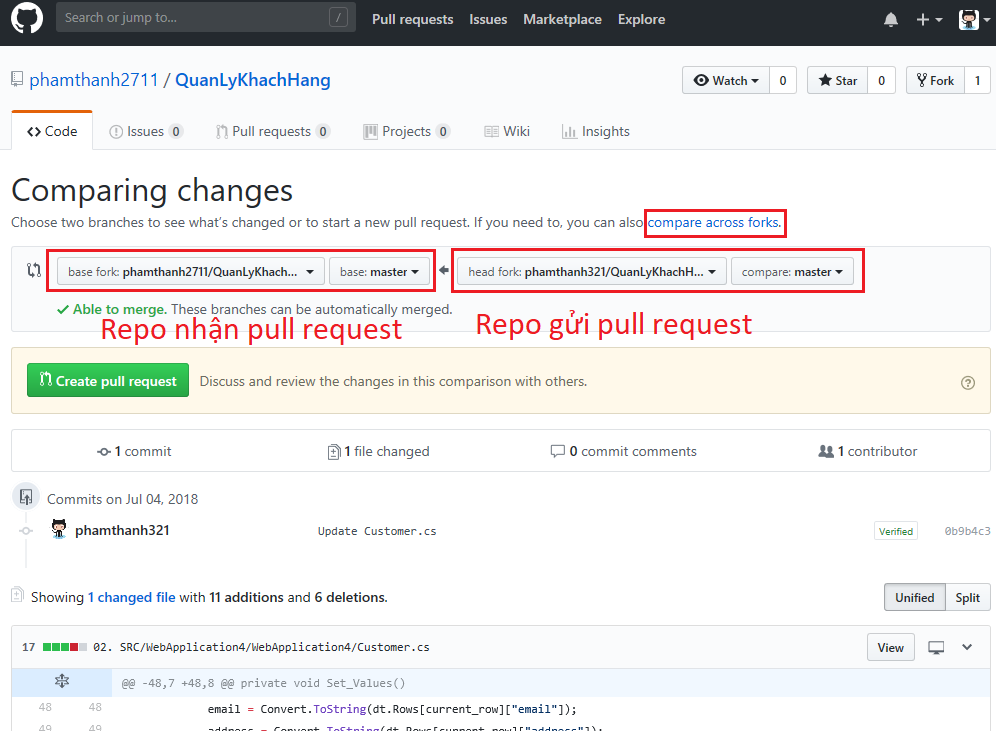


Nhấn vào chọn chỉnh sửa file trực tiếp trên web



Sau khi **commit changes**, ta bắt đầu pull request. Vào **pull requests** chọn **new pull request**.

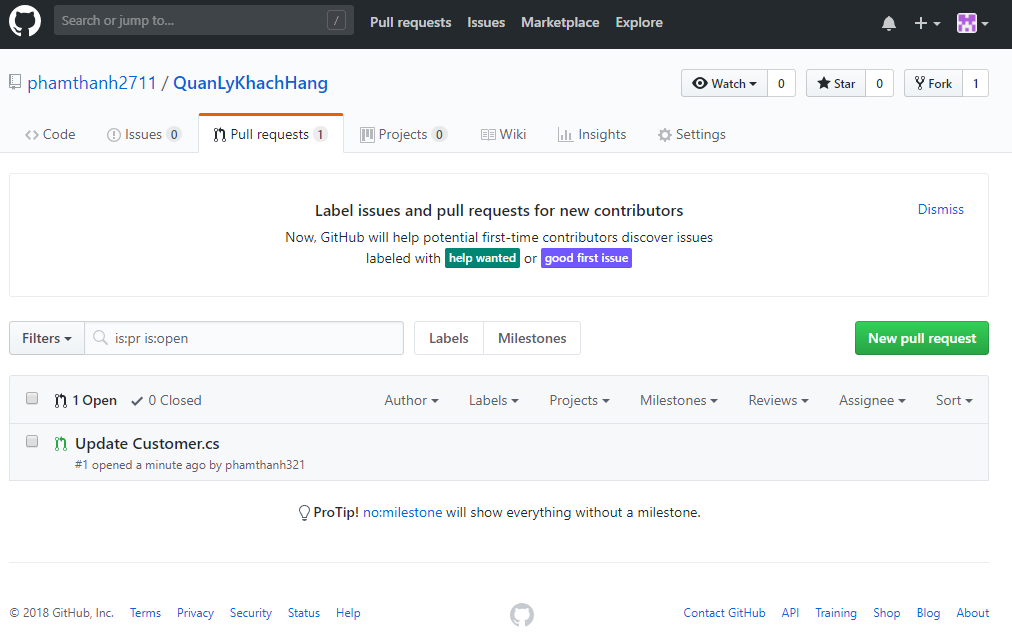


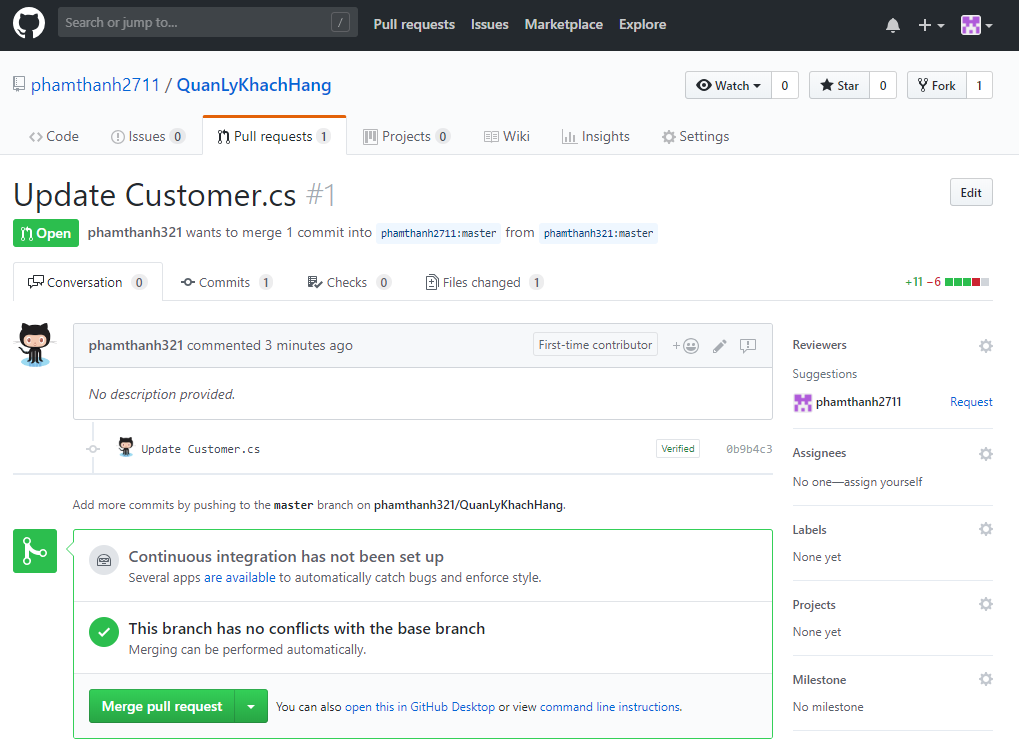


Thông thường thì github tự động hiện bên nhận PR là repo của A và bên gửi PR là repo của B, nhưng nếu không hiện đúng như vậy thì B phải tự chỉnh bằng tay, trong lúc đổi bên nhận gửi PR, thì sẽ có thời điểm là bên nhận và bên gửi là cùng 1 repo, khi đó phải dùng nút **compare across forks** để giải quyết.

### **2.8 Hợp nhất pull request**

Sau khi B gửi pull request, thì bên A sẽ có thông báo có Pull Request như hình bên dưới.

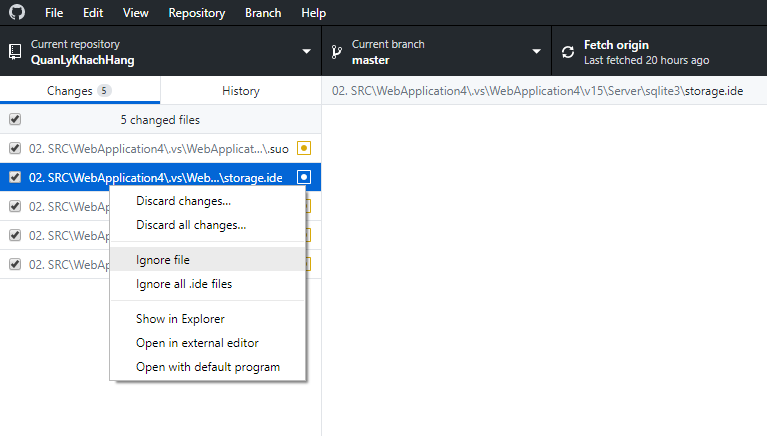




Sau khi chọn **Merge pull request** để chấp thuận hợp nhất thì thay đổi sẽ được hợp nhất vào repo của A.

### **2.9 Tạo file gitignore**

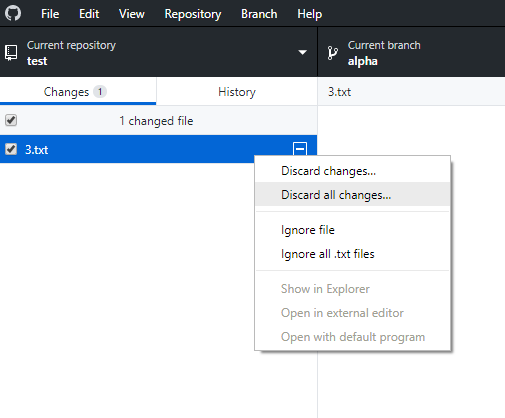
Khi mà bạn không muốn một file bất kỳ nào đó thể hiện sự thay đổi trên git bạn có thể cho nó vào trong file gitignore và để làm được điều đó bạn click chuột phải vào file thay đổi đó.



Bạn chọn ignore file này thì khi có sự thay đổi ở file này nó sẽ không xuất hiện nữa hoặc chọn ignore all nếu muốn tất cả những file có đuôi mở rộng giống file này không xuất hiện.

### **2.10 Khôi phục file khi mới xóa hay thay đổi**

Sau khi xóa file trong thư mục git trên máy bạn hoặc thay đổi nội dung của nó, nếu như muốn khôi phục lại file như cũ, bạn có thể click chuột phải vào sự thay đổi đó và chọn Discard changes hoặc discard all changes nếu như muốn khôi phục lại hết các sự thay đổi. Lưu ý điều này chỉ thực hiện khi bạn vẫn chưa push commit có sự thay đổi đó lên remote repository ở website.



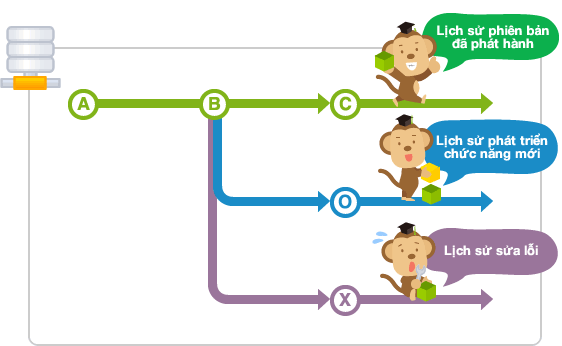
## **3 Tìm hiểu về branch**

### **3.1 Branch là gì ?**

Trong việc phát triển phần mềm, thì ứng với một phần mềm có nhiều thành viên đồng thời tiến hành thêm chức năng hay là tiến hành chỉnh sửa lỗi cùng một lúc. Và ở tình trạng tồn tại của nhiều phiên bản đã phát hành thì cũng phải lưu giữ từng phiên bản.

Chính vì vậy để hỗ trợ quản lý phiên bản hay thêm nhiều chức năng được tiến hành song song, một chức năng được trang bị thêm được gọi là branch ở Git.

Branch là cái dùng để phân nhánh và ghi lại luồng của lịch sử. Branch đã phân nhánh sẽ không ảnh hưởng đến branch khác nên có thể tiến hành nhiều thay đổi đồng thời trong cùng 1 repository.

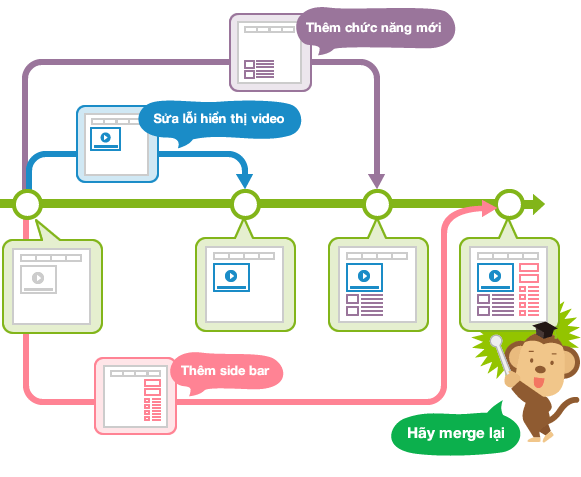


Hơn nữa, branch đã phân nhánh có thể chỉnh sửa tổng hợp lại thành 1 branch bằng việc hợp lại (merge) với branch khác.

Sơ đồ bên dưới là mô hình của thao tác song song đã sử dụng branch.

Các thành viên của nhóm sẽ tạo branch dùng riêng cho công việc của mình từ branch chính để không ảnh hưởng đến công việc của các thành viên khác. Sau đó, những thành viên đã hoàn thành công việc của mình sẽ thực hiện đưa thay đổi của mình vào branch chính. Theo cách như vậy, sẽ không bị ảnh hưởng từ công việc của các thành viên khác, và bản thân mình có thể thực hiện công việc của mình.

Hơn nữa, bằng việc để lại lịch sử theo đơn vị công việc, trong trường hợp có phát sinh vấn đề thì việc điều tra nguyên nhân ở những vị trí thay đổi cũng như việc tiến hành đối sách khắc phục sẽ trở nên dễ dàng hơn.



Khi tiến hành commit lần đầu trong repository thì Git sẽ tạo ra một branch có tên là master. Vì thế những lần commit sau sẽ được thêm vào branch master cho đến khi chuyển đổi branch.

### **3.2 Vận dụng branch**

Trên Git thì có thể tự do tạo branch. Thế nhưng để sử dụng branch hiệu quả thì trước hết cần thiết phải thiết lập qui tắc vận dụng.

Ở đây sẽ giới thiệu phương pháp vận dụng đã sử dụng với 2 loại branch là branch chủ đề (Topic branch) và branch tích hợp (Intergration branch).

**A, Branch tích hợp ( Integration branch )**

Branch tích hợp là branch có thể tạo ra bản phát hành bất cứ khi nào. Hơn nữa, nó cũng được sử dụng như là nguồn phân branch của branch chủ đề. Vì thế, việc duy trì trạng thái ổn định là điều cần thiết.

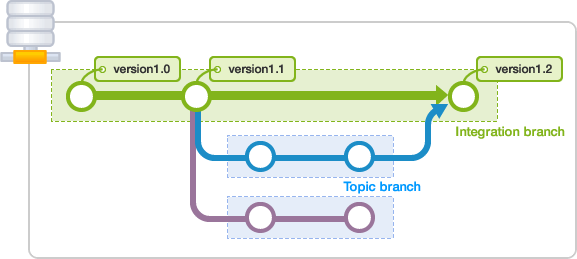
Trường hợp tiến hành thay đổi gì đó thì sẽ thường tạo ra branch chủ đề rồi thực hiện thay đổi. Và việc kiểm tra và build tự động sử dụng công cụ CI như Jenkins chẳng hạn thì sẽ sử dụng branch này để tiến hành.

Thông thường sẽ sử dụng branch master như là một branch tích hợp.

**B, Branch chủ đề ( Topic branch )**

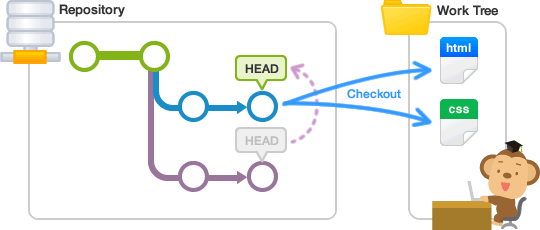
Branch chủ đề là branch tạo ra nhằm tiến hành công việc liên quan đến chủ đề như là chỉnh sửa lỗi hay là thêm chức năng. Khi tiến hành cùng lúc những công việc có liên quan đến nhiều chủ đề thì số lượng branch chủ đề tương ứng sẽ được tạo ra.

Branch chủ đề sẽ tạo ra bằng cách phân branch từ branch tích hợp đã ổn định, khi đã hoàn thành xong công việc sẽ sử dụng đưa vào branch tích hợp.



### **3.3 Chuyển đổi branch**

Để chuyển đổi branch làm việc thì sẽ thực hiện thao tác gọi là checkout. Khi thực hiện checkout, trước tiên nội dung của lần commit cuối cùng trong branch chuyển đến sẽ được mở ra trong work tree. Và commit đã tiến hành sau khi check out thì sẽ được thêm vào branch sau khi di chuyển đến.



**+ Head**

HEAD là tên hiển thị phần đầu của branch đang sử dụng hiện tại. Mặc định là đang hiển thị phần đầu của master. Bằng việc di chuyển HEAD thì branch đang sử dụng sẽ được thay đổi.

**+ Stash**

Khi những file được thêm mới hay nội dung thay đổi vẫn chưa commit vẫn còn lưu lại Index và Work tree, mà thực hiện checkout đến branch khác thì nội dung thay đổi đó sẽ di chuyển từ branch ban đầu đến branch chuyển đến.

Tuy nhiên ở branch di chuyển đến, trường hợp có file giống như vậy đã được tiến hành thay đổi cái gì đó thì checkout sẽ thất bại. Khi xảy ra trường hợp này thì sẽ commit lại nội dung thay đổi 1 lần nữa hoặc là sử dụng stash để lưu tạm thời nội dung thay đổi, sau đó phải thực hiện commit.

Stash là khu vực ghi lại tạm thời nội dung thay đổi của file. Bằng việc sử dụng stash, trong work tree và index, những thay đổi chưa được commit có thể lưu lại tạm thời, thưsh là khu vực ghi lại tạm thời nội dung thay đổi củc mà lại đang làm dở ở branch hiện tại.

Muưsh là khu vực ghi lại tạm thời nội dung thay đổi củc mà lại đang git stash như sau:

$ git stash save *# or just "git stash"*

Những thay đổi lưu tạm này về sau có thể lấy ra và hiển thị trên branch ban đầu hay là phản ánh lên branch khác.

**Lhững thay đổi lư**Sau khi đã git stash 1 hoặc vài lần, bạn có thể xem lại danh sách các lần lưu thay đổi bằng câu lệnh

$ git stash list

stash@{0}: WIP on <branch-name>: <lastest commit>

stash@{1}: WIP on <branch-name>: <lastest commit>

stash@{2}: WIP on <branch-name>: <lastest commit>

Nếu muốn xem cả nội dung của từng thay đổi thì thêm option -p

$ git stash list -p

hoặc xem nội dung cụ thể hơn nữa của lần thay đổi thứ 1:

$ git stash show stash@{1}

Khi muốn apply lại thay đổi từ stash lần 1 bạn có thể

$ git stash apply stash@{1}

**Xoá các thay đply stash@{ổi từ s**

Đôi khi bạn muốn lấy lại thay đổi và xoá nội dung thay đổi lưu trong stack đi, khi đó bạn có thể

$ git stash apply stash@{1}

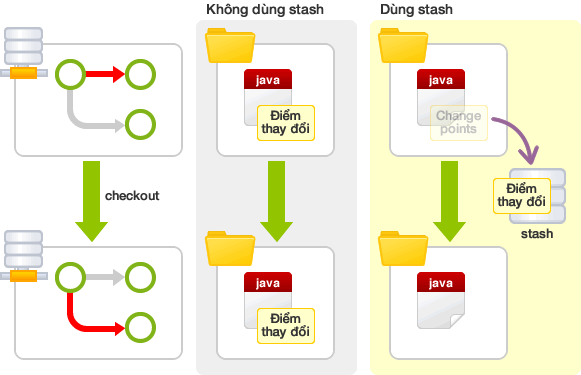
$ git stash drop stash@{1}

hoặc đơn giản hơn là

$ git stash pop stash@{1}

Thậm chí nếu muốn xoá *toàn bộ stack* thì có thể dùng clear

$ git stash clear



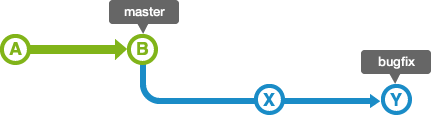
### **3.4 Tích hợp branches**

Branch chủ đề đã hoàn thành công việc, cuối cùng sẽ được tích hợp vào branch tích hợp. Việc tích hợp branch thì có hai phương pháp, phương pháp sử dụng merge và phương pháp sử dụng rebase. Dựa vào việc sử dụng phương pháp nào mà lịch sử của branch sau khi tích hợp sẽ có sự khác biệt lớn.

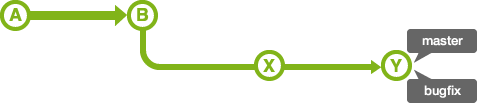
**+ Merge**

Khi sử dụng merge, có thể tổng hợp nhiều luồng lịch sử.

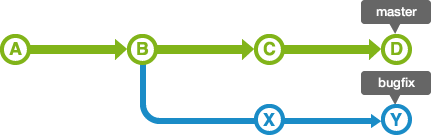
Ví dụ, có branch bugfix phân nhánh ra từ branch master như sơ đồ bên dưới.



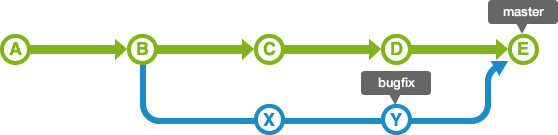
Khi merge branch bugfix này vào branch master, nếu trạng thái branch master không có gì thay đổi thì có thể thực hiện merge hết sức đơn giản. Vì lịch sử của branch bugfix sẽ bao gồm tất cả lịch sử của branch master, nên branch master chỉ có việc di chuyển đơn giản là có thể lấy được nội dung của branch bugfix. Và, người ta gọi merge như thế này là merge fast-forward (chuyển tiếp nhanh).



Nhưng, cũng có trường hợp lịch sử của branch master vẫn đang tiến triển sau khi phân branch bugfix. Trường hợp này thì cần thiết phải tổng hợp nội dung thay đổi của branch master và nội dung thay đổi của branch bugfix thành một.

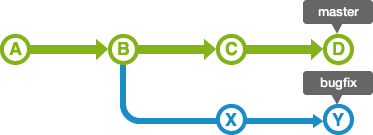


Vì thế, merge commit đã lấy thay đổi của cả hai branch sẽ được tạo ra. Đầu branch master sẽ di chuyển đến commit đó.

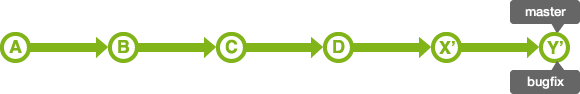


**+ Rebase**

Giống với ví dụ merge, sẽ có branch bugfix phân nhánh từ branch master như sơ đồ bên dưới.

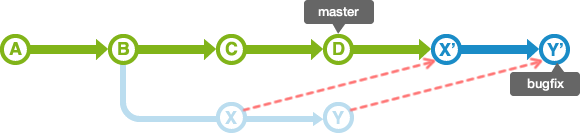


Trường hợp sử dụng rebase ở đây rồi tiến hành tích hợp branch thì lịch sử sẽ giống như sơ đồ tiếp theo. Bây giờ, sẽ trình bày đơn giản trình tự rebase như thế nào.



Trước hết, khi rebase branch bugfix vào branch master, lịch sử branch bugfix sẽ được thay đổi đính kèm sau branch master. Cho nên, lịch sử sẽ thành 1 đường như trong sơ đồ.

Khi này sẽ có trường hợp phát sinh xung đột tại commit di chuyển X và Y. Lúc đó tại từng commit cần thiết phải chỉnh sửa lại những chỗ phát sinh xung đột.



Nếu chỉ rebase không thôi thì vị trí đầu master vẫn cứ như vậy. Vì thế, merge branch bugfix từ branch master rồi di chuyển master đến phần đầu của bugfix.



**Khác nhau giữa merge và rebase:**

+ Merge: Thao tác merge là kết quả của phép trộn giữa ba commit, đó là commit cha chung và 2 commit mới nhất của 2 nhánh (hai điểm kết thúc), kết quả sẽ tạo ra một commit chứa kết quả của phép trộn. Vì vậy lịch sử tại nhánh đang làm việc sẽ bị gián đoạn.

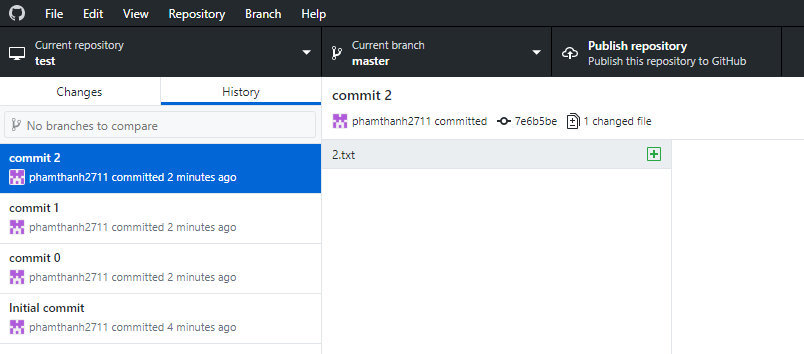
+ Rebase: Cho lịch sử rõ ràng hơn, bởi nó sẽ duyệt qua từng commit của nhánh đang làm việc và thực hiện thay đổi, vì vậy nhìn vào ta sẽ thấy giống như một đường thẳng.

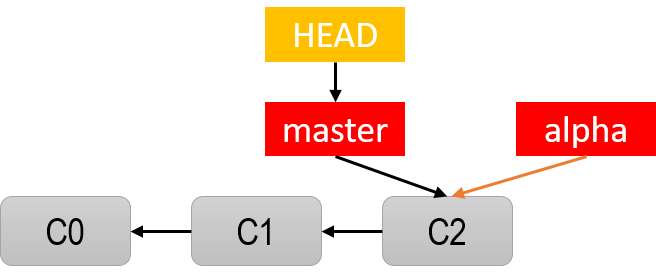
Khi làm việc bạn có thể chọn một trong hai đều được, nhưng có một số trường hợp bạn nên sử dụng rebase thì tốt hơn. Ví dụ bạn đang làm việc nhóm, mỗi thành viên làm trên một branch, và khi bạn merge với nhau thì các thành viên sẽ không thấy lịch sử thay đổi của dự án một cách rõ ràng, nhưng nếu bạn sử dụng rebase thì họ sẽ dễ dàng thấy hơn.

### **3.5 Ví dụ về phân nhánh và gộp nhánh**

**Khởi tạo dự án với 3 commit**

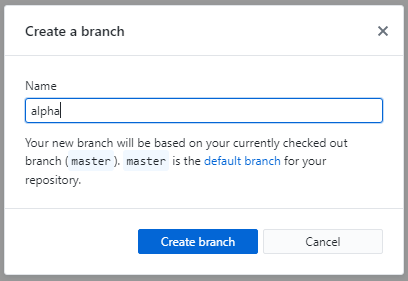
Giả sử tạo repo với tên là test. Lúc này sẽ mặc định sẽ làm việc trên nhánh chính là nhánh master. Sau khi có repo, ta tạo 3 file 0.txt, 1.txt, 2.txt và lần lượt tạo 3 commit đưa lên git.

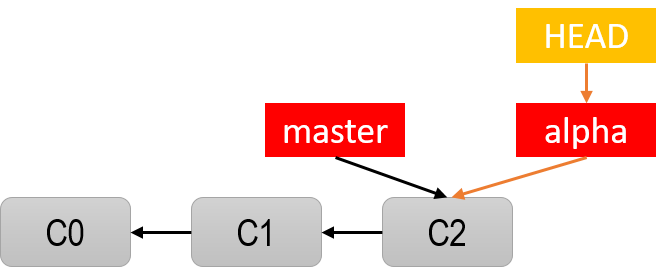




**Tạo một nhánh mới**

Ta tạo một nhánh mới với tên là alpha và chuyển nhánh làm việc sang nhánh alpha

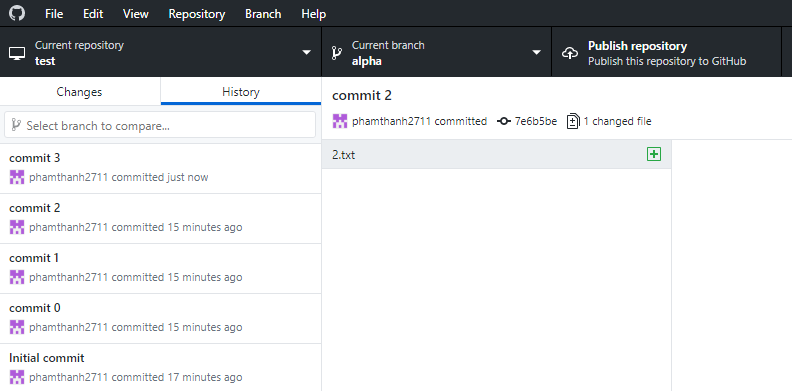


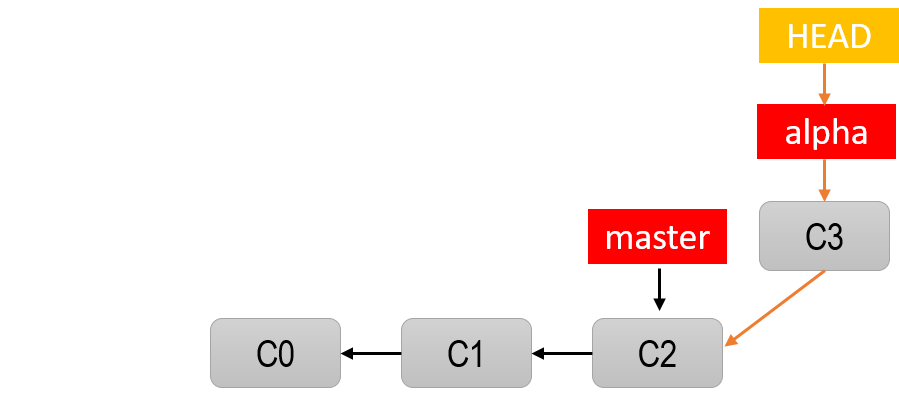


Cả 2 nhánh master và alpha đều trỏ đến commit 2 ( C2 )

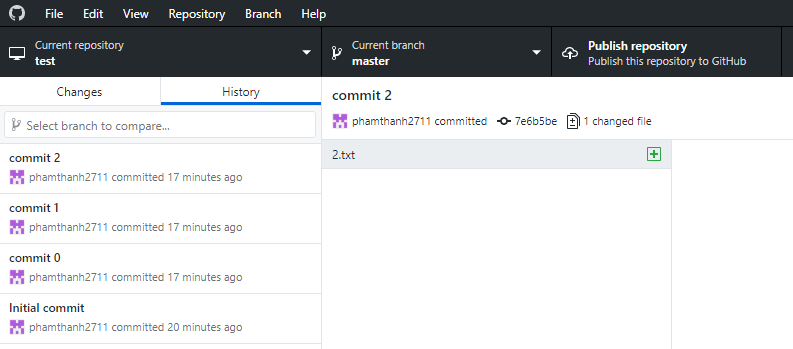
**Tạo sự thay đổi và commit lên nhánh alpha**

Trên nhánh alpha ta thực hiện thay đổi (tạo mới file 3.txt) và hiện tại ở nhánh alpha đã có commit 3 và các commit cũ kế thừa từ nhánh master.



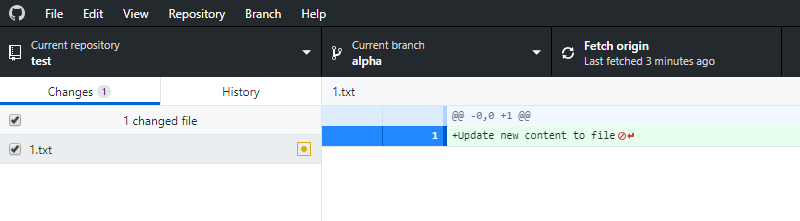


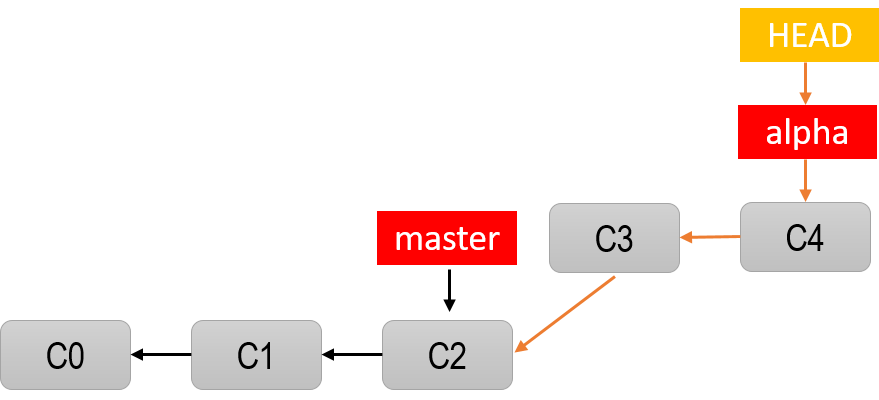
Quay lại nhánh master, ta thấy mới chỉ dừng lại ở commit 2 và các commit cũ hơn , trong này chúng ta sẽ không thấy commit 3 khi tạo ở trên nhánh alpha.



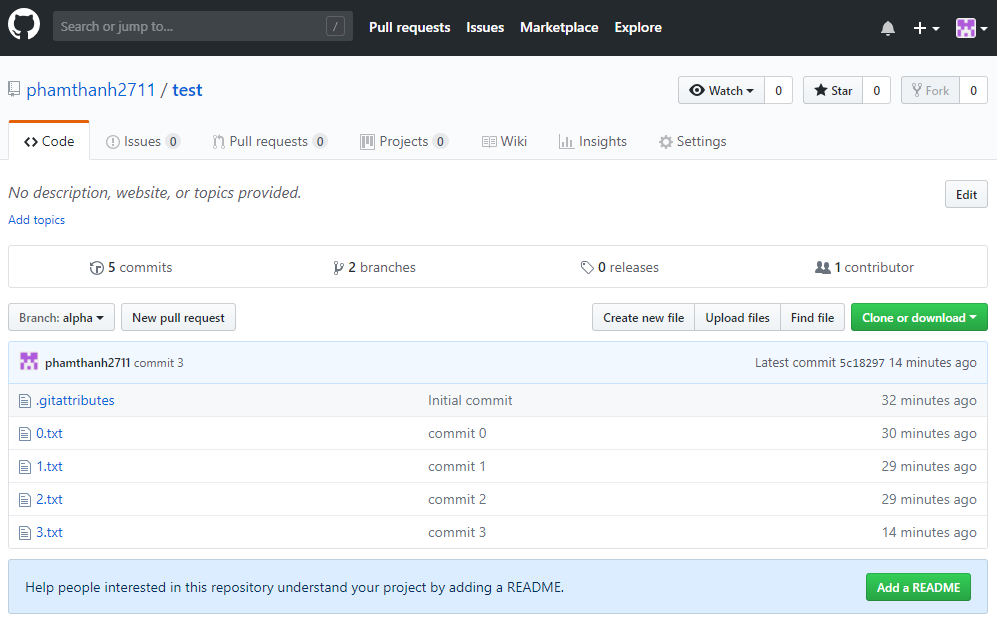
**Tiếp tục tạo sự thay đổi trên nhánh alpha**

Quay lại nhánh alpha, ta tiếp tục thay đổi nội dung file 1.txt (thêm nội dung ‘Update new content to file’) và tạo commit 4 đưa lên nhánh alpha.



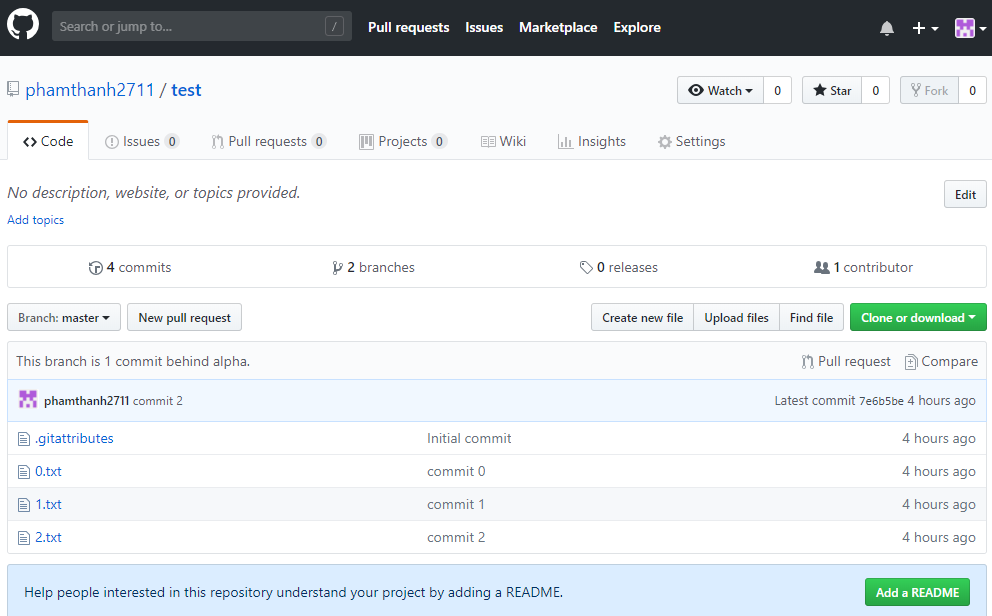


Như vậy ở commit 4, tại nhánh alpha, trong thư mục làm việc có các file: 0.txt, 1.txt, 2.txt, 3.txt, so với master nó có thêm file 3.txt và có chèn thêm nội dung mới trong file 1.txt



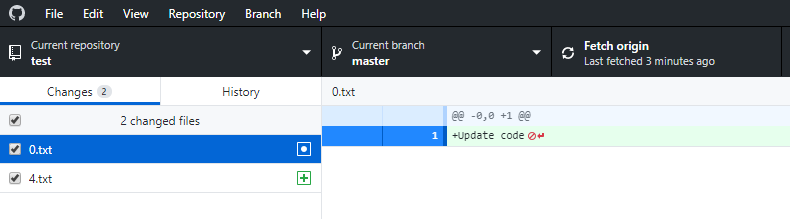
**Quay trở lại làm việc với nhánh master**

Trở về nhánh master, ta thấy có các file 0.txt, 1.txt và 2.txt, trong đó file 1.txt chưa có sự thay đổi nào cả.

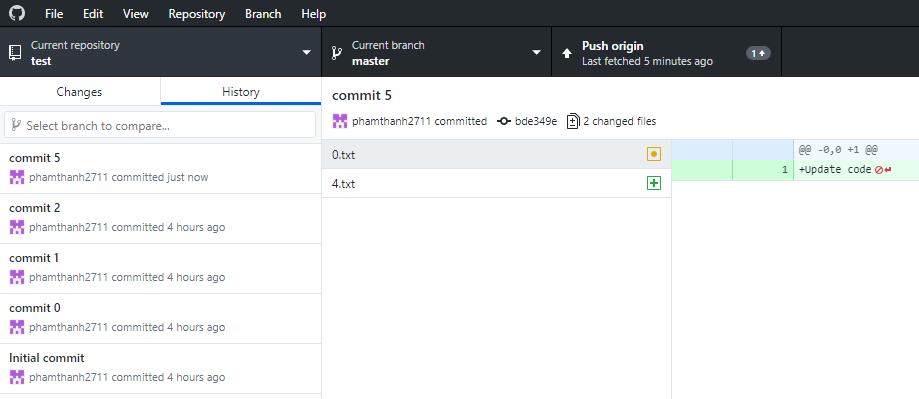


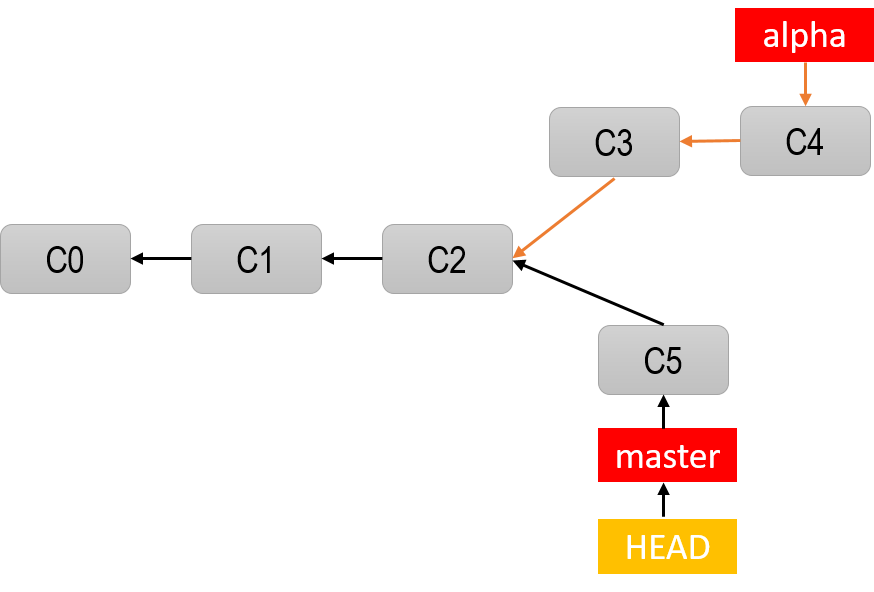
**Tạo sự thay đổi và commit cho nhánh master**

Tại nhánh master, ta bắt đầu tạo sự thay đổi bằng cách tạo thêm file 4.txt và sửa nội dung file 0.txt (thêm nội dung: “Update code” ) .



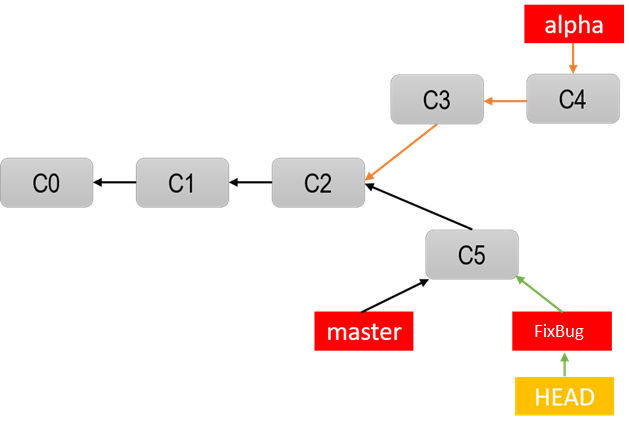
Commit lên nhánh master và đặt tiêu đề là commit 5.





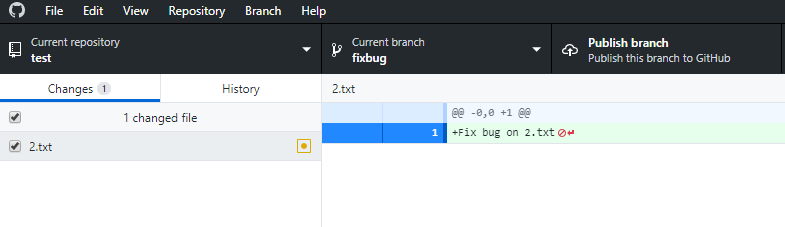
**Tạo nhánh mới rẽ từ nhánh master**

Hiện giờ đang ở nhánh master và phát triển code theo kế hoạch nhưng code trong master cần sửa và kiểm tra ngay. Để quá trình sửa lỗi code không làm ảnh hưởng đến các code hiện tại trong nhánh master thì ta có thể tạo thêm một nhánh mới đặt tên là **fixbug,** sau đó chuyển sang nhánh fixbug để làm việc.

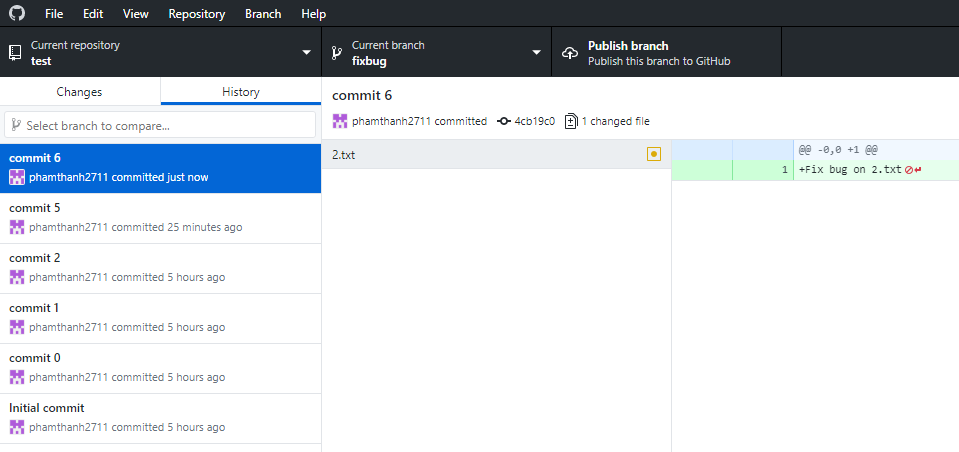


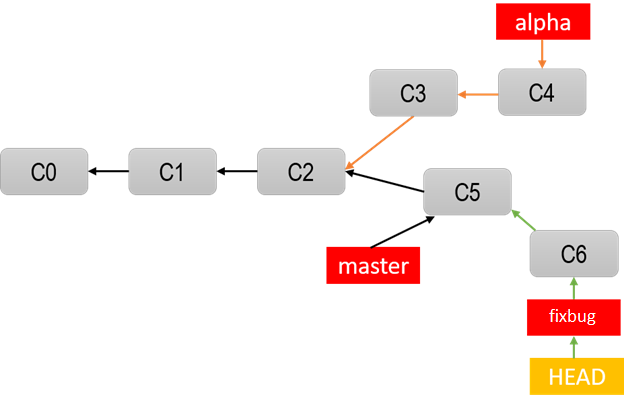
**Tạo sự thay đổi và commit lên nhánh fixbug**

Tạo sự thay đổi trên file 2.txt (thêm nội dung: ‘Fix bug on 2.txt’) và tạo commit 6 lên nhánh fixbug

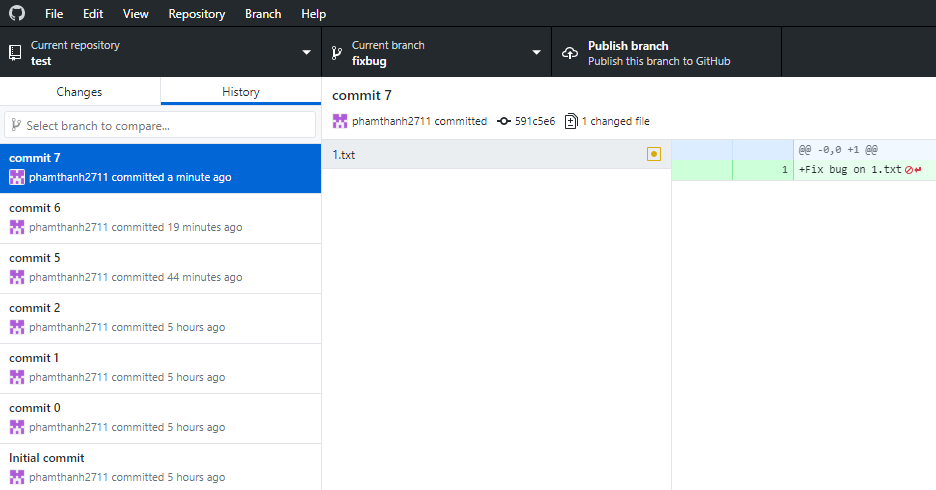


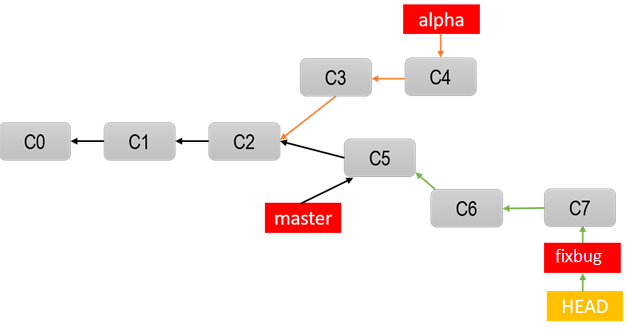
Nhìn vào nhánh fixbug ta sẽ thấy có thêm commit 6 và kế thừa các commit trước kể từ commit 5 của nhánh master





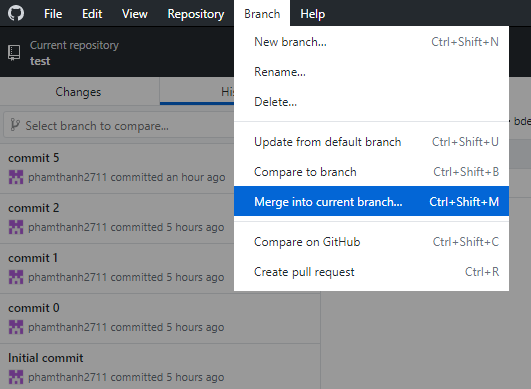
Ta có thể tiếp tục sửa đổi trên file 1.txt (thêm nội dung: ‘Fix bug on 1.txt’) và commit lên nhánh fixbug (commit 7)



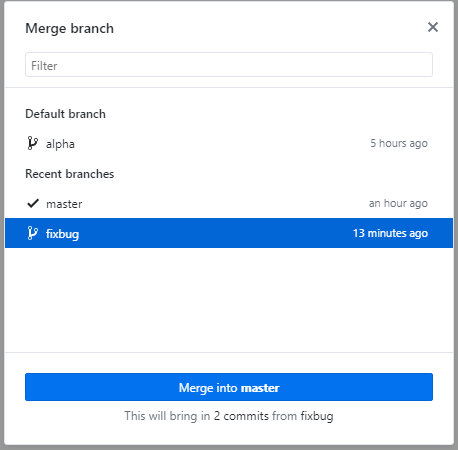


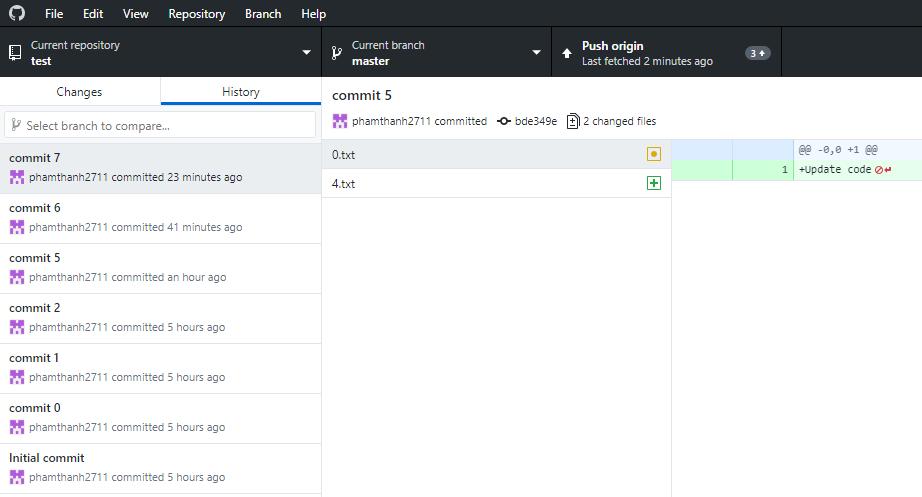
**Trộn nhánh fixbug vào nhánh master**

Sau khi đã hoàn thành nhiệm vụ trên nhánh fixbug nếu muốn các kết quả của nhánh này tích hợp thay đổi vào nhánh master thì tiến hành gộp 2 nhánh này lại ( Merge ). Để làm được điều này, cần phải chuyển về nhánh master rồi cho nhánh master gộp lại với nhánh fixbug bằng chức năng merge.

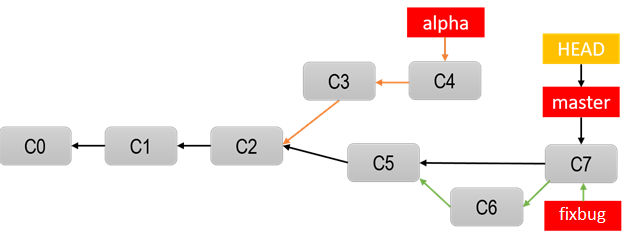


Sau đó chọn nhánh cần merge





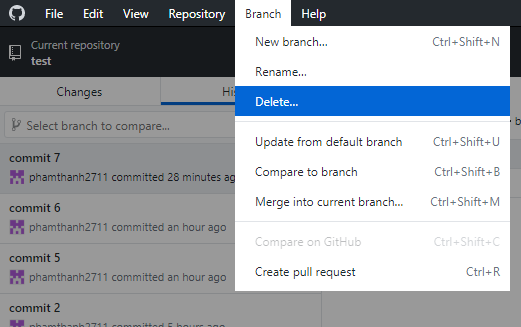
Hiện giờ nhánh master đã gộp vào với nhánh fixbug và cả 2 nhánh đều trỏ vào commit 7 như hình bên dưới



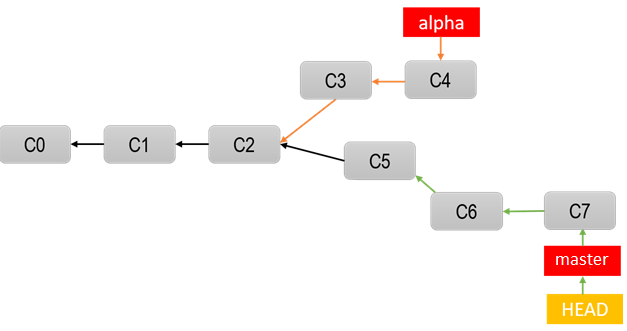
**Xóa nhánh khi không còn sử dụng**

Sau khi nhánh fixbug đã gộp lại vào nhánh master, nếu ta không còn nhu cầu làm việc trên nhánh này thì có thể xóa nhánh này đi.

Ta quay trở lại nhánh fixbug sau đó vào Branch -> Delete.

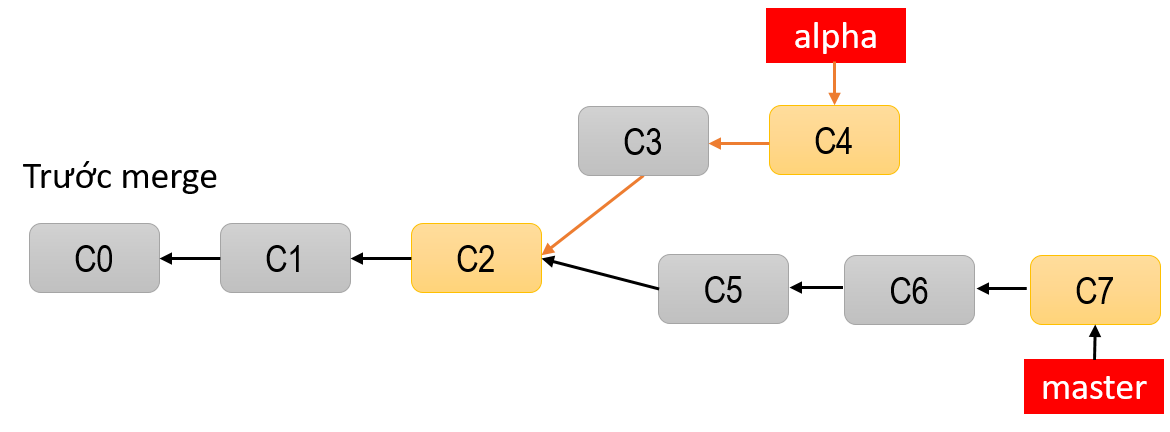


Vậy là giờ nhánh master đã có các commit C0, C1, C2,C5,C6,C7 và hình ảnh trên repo hiện tại như sau:



**Xử lý xung đột khi gộp nhánh**

Bây giờ giả sử có nhu cầu gộp code tại nhánh alpha vào nhánh master, trường hợp này không đơn giản như cách gộp nhánh fixbug ở trên. Do cả 2 nhánh có nhiều commit kể từ thời điểm rẽ nhánh nên khi gộp nó sẽ xem xét sự thay đổi trên cả 2 nhánh tại ba điểm (three-way), thời điểm commit cuối của các nhánh và thời điểm rẽ nhánh, đó là các commit với snapshot C2, C4, C7 như hình dưới:

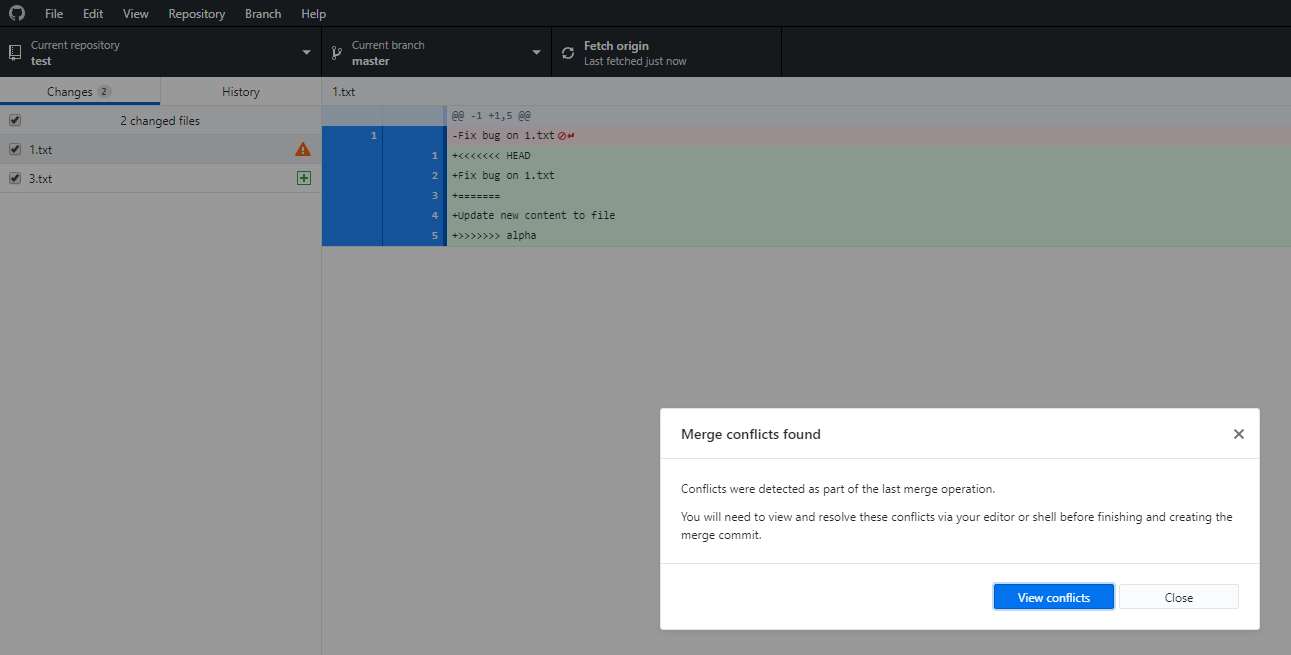


Như hình trên, repo có 2 nhánh và nhánh rẽ bắt đầu từ C2, tức là bắt đầu từ C2 là 2 nhánh bắt đầu có sự thay đổi.

So sánh giữa 2 nhánh master và alpha sau commit cuối ta có bảng so sánh như sau:

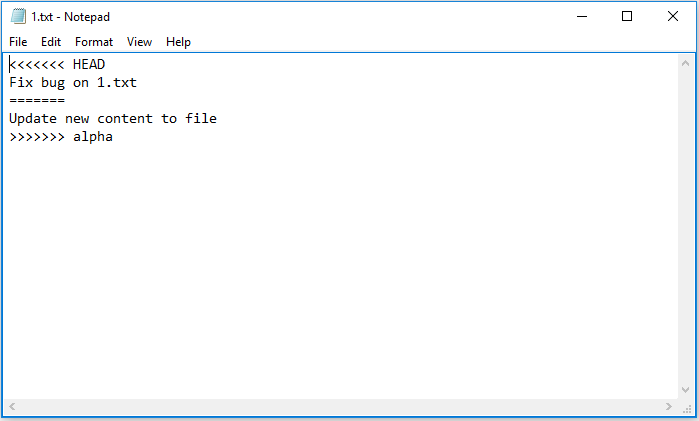
|  |  |  |  |
| --- | --- | --- | --- |
| **File** | **Nhánh master** | **Nhánh alpha** | **Sau khi merge alpha vào master** |
| 0.txt | Có sửa đổi:”Update code” | Không sửa đổi | Không xung đột (lấy theo bản bên nhánh master) |
| 1.txt | Có sửa đổi:”fix bug on 1.txt” | Có sửa đổi:”Update new content to file” | Xung đột vì không biết lấy theo bản ở nhánh nào |
| 2.txt | Có sửa đổi:”fix bug on 2.txt” | Không sửa đổi | Không xung đột (lấy theo bản bên nhánh master) |
| 3.txt | Không sửa đổi | Tạo mới file | Không xung đột (lấy theo bản bên nhánh alpha) |
| 4.txt | Tạo mới file | Không sửa đổi | Không xung đột (lấy theo bản bên nhánh master) |

Theo bảng phân tích trên, sau khi merge nhánh alpha vào nhánh master sẽ xảy ra xung đột ở file 1.txt vì có sự thay đổi trên cả 2 nhánh này và git không biết sẽ lấy sự thay đổi bên nhánh nào.



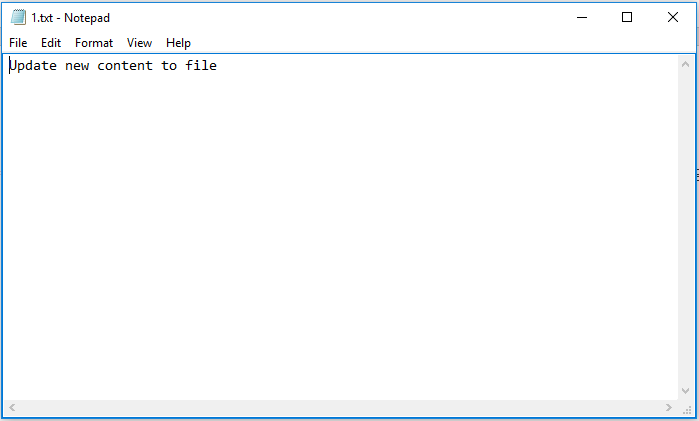
**Xử lý xung đột khi gộp nhánh.**

Ta thấy file xung đột ở đây là file 1.txt và sau khi mở file 1.txt ra chúng ta thấy như sau:

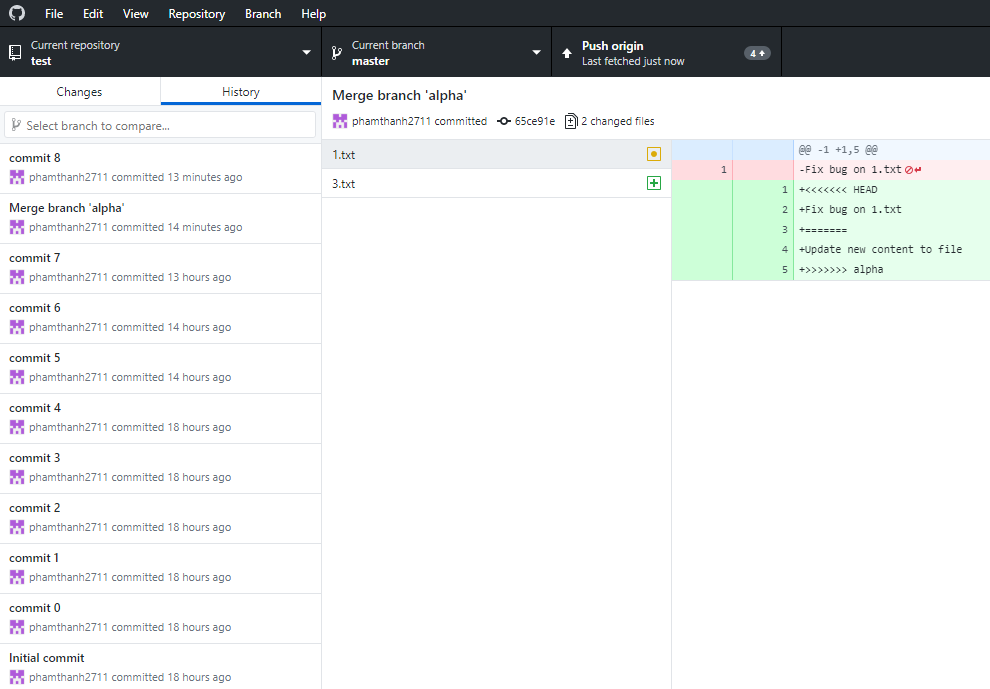


Bạn thấy nội dung xung đột nó được đánh dấu bên trong các ký hiệu. Nội dung nằm giữa <<<<<<< HEAD và ======= là nội dung có từ master, còn nội dung nằm giữa

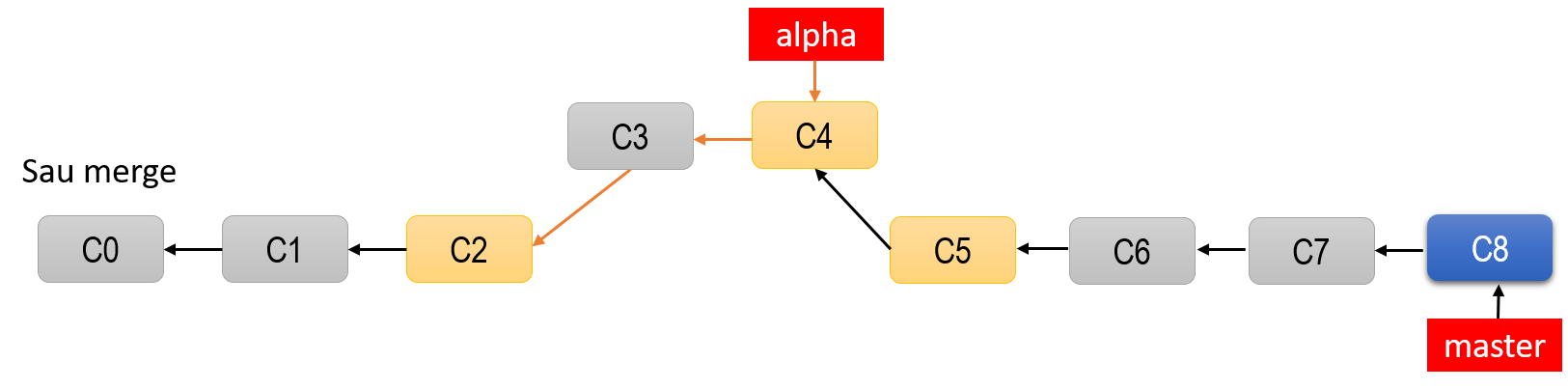
======= và >>>>>>> alpha là nội dung từ alpha. Giữa hai nội dung đó lấy nội dung nào thì giữ, còn lại thì xóa hết. Ví dụ file trên giữ lại nội dung như sau (nội dung từ alpha)



Sau khi xử lý xong xung đột trong file 1.txt ta tạo commit để hoàn tất việc gộp nhánh



Hình ảnh repo sau khi gộp nhánh sẽ như sau:



## **4 Quản lý nhóm làm việc trên github**

### **4.1 Git flow là gì ?**

Git Flow được đưa ra nhằm cải thiện quá trình làm việc cùng Git. Thực chất, đấy là cách chia nhánh và merge nhánh vào khi hoàn thành một tập hợp tính năng hoặc fix.

Git Flow làm việc dựa trên merge, chứ không phải thao tác *cherry-pick* thường thấy.

Git Flow đưa ra các quy ước để triển khai công việc. Mục đích là các nhóm công việc triển khai song song nhưng không ảnh hưởng tới nhau. Các  môi trường development, staging và production tách biệt giúp quá trình kiểm thử (QA), trả lại feedback và xử lý các issue được gọn gàng và thống nhất hơn nhiều.

Ý tưởng của nó thực ra là duy trì các nhánh branch không đổi, tức không xoá (có tính cố định) trong suốt dòng đời sản phẩm. Branch master sẽ luôn là branch chính áp dụng cho production, trong khi các branch hotfix, features hay develop cung cấp các bản để phục vụ QA và hoàn thiện trước khi được đẩy lên master.

Khác với cách thông thường tạo ra nhiều vấn đề xảy ra ngay trên production, thứ mà chúng ta hay gọi là “rút kinh nghiệm từ những sai lầm thực tiễn”, Git Flow đẩy quá trình QA vào một phần bắt buộc cho cả developer và team QA và yêu cầu sự hoàn thiện cao hơn về chất lượng đầu ra.

**Phân tán nhưng tập trung**

Mô hình git sẽ hoạt động xoay quanh một repo trung tâm. Chú ý rằng nó chỉ được xem là repo trung tâm chứ thực tế về mặt kĩ thuật git không có một repo nào gọi là repo trung tâm.

Mỗi developer sẽ pull và push với repo trung tâm. Bên cạnh đó mỗi người có thể pull những thay đổi từ những người khác để tạo thành các sub team. Điều này thực sự có ích khi phải làm việc nhóm nhiều người trở lên và phải hoàn thành một feature lớn mà không phải đẩy source code đang dang dở lên repo trung tâm

### **4.2 Những branch chính**

Repo trung tâm (gọi tạm là origin) sẽ chứa hai branches chính hoạt động mãi mãi là master và develop.



Nhánh origin/master được coi là nhánh chính phản ánh trạng thái sẵn sàng để sản xuất. Đây sẽ là nhánh chứa chương trình ổn định nhất.

Nhánh origin/develop cũng được coi là nhánh chính song song với master chứa code mới nhất trong quá trình phát triển và để chuẩn bị cho đợt release tiếp theo

Khi source code bên develop đạt đến một mức độ ổn định sẵn sàng để release thì sẽ được merge sang master và được đánh dấu với release number

### **4.3 Những branch phụ**

Bên cạnh 2 branches chính ở trên thì sẽ có thêm những branches phụ để giúp các thành viên trong team có thể phát triển song song, dễ dàng theo dõi qua các tính năng cụ thể, chuẩn bị cho release hoặc fix nhanh các vấn đề trước khi sản xuất. Khác với 2 branches chính kia, các branches phụ này chỉ tồn tại trong một thời gian rồi có thể sẽ được xóa đi.

Những branches phụ này sẽ gồm: feature branches , release branches , hotfix branches.

Về mặt kĩ thuật thì không có branches nào là đặc biệt hơn branches nào, tất cả chỉ là các git branches thông thường và chúng chỉ được phân loại theo cách mà ta sử dụng ra sao thôi.

**+ Feature branches**

Tách ra từ : develop branch

Merge vào: develop branch

Name: feature-\* (Ví dụ: feature-login)



Hay còn được gọi là topic branches được sử dụng để phát triển các feature mới phục vụ cho release sau này. Nó sẽ được tồn tại trong quá trình chức năng được phát triển và cuối cùng sẽ được merge vào nhánh develop( khi quyết định lần release tới sẽ bao gồm chức năng đó ) hoặc sẽ được xóa đi khi không còn cần thiết nữa. Về cơ bản thì feature chỉ tồn tại ở repo của developer chứ không trong origin.

**+ Release branches**

Tách ra từ: Develop branches

Merge vào: Develop branches và Master branches

Name: release-\* (ví dụ release-1.1)

Nhánh này được sử dụng để chuẩn bị cho release bản production mới. Tất cả các công việc cuối cùng trước khi release được thực hiện ở đây, ngoài ra còn để fix các bug lẻ tẻ, chuẩn bị cho meta-data(version number, build date, …). Nhờ việc tách từ develop, chúng ta có thể tiếp tục phát triển các feature cho đợt release khác một cách bình thường.

Thời điểm được lựa chọn để tách ra từ nhánh develop là khi nhánh develop phản ánh trạng thái sẵn sàng cho việc release mới, ít nhất là lúc đó các features dành cho đợt release phải được merge vào nhánh develop rồi. Những feature được nhắm đến các lần release sau thì chưa được merge vào.

Chúng ta sẽ tiến hành đánh version theo quy định của dự án ngay sau khi tạo release branches.

Nhánh này sẽ tồn tại cho đến khi việc release được thực hiện xong. Trong khoảng thời gian đó, có thể thực hiện fix bug ở nhánh này , tuy nhiên sẽ cấm việc bổ sung feature mới lên đó. Nếu có feature mới thì hãy merge vào nhánh develop và đợi đợt release sau.

Khi source trên release branches sẵn sàng để release, đầu tiên ta merge vào master, sau đó merge vào develop để những lần release sau cũng chứa những thay đổi ở lần lần này.

**+ Hotfix branches**

Tách ra từ: Master branches

Merge vào: Develop branches và Master branches

Name: hotfix-\* (ví dụ hotfix-1.1)

Nhánh này được tạo ra từ nhánh master . Ví dụ hiện tại version 1.1 là phiên bản product đang chạy và bỗng xuất hiện lỗi nghiêm trọng. Tuy nhiên soure code trên develop vẫn chưa ổn định, vì thế chúng ta phải tách nhánh hotfix và tiến hành sửa lỗi.

Sau khi tách nhánh tiến hành up version và tiến hành commit khi sửa xong

Kếkhi sửa xongcommitững thay đổi phải được merge lại vào master , đồng thời cũng phải merge vào develop đg thay đổi phải được merge lại vào master , đồng thời cũng phải mer trên release branch phải không. Tuy nhiên, có mhải được merge lại vào m**khi đang tcó mhải được merge lại vào master , đồng thời cũng phải mer trên release branch phải không.**. Khi release branch đưmerge lại vào master , đồng thời cũng phải mer trên release ix cũng được merge vào develop, nên không có vấn đề gì cả. Trừ khi thực sự công việc ở develop cvelopelease branch đưmerge lại vào master , đồng thời cũng phải mer trên release ix cũng được merge vo develop.

## **5 Các câu lệnh cơ bản trong git-scm**

Bên câu github desktop bạn có thể sử dụng git bằng những câu lệnh và cần phải có một phần mềm để thn câu github desktop bạn có thể sử dụng <https://git-scm.com/download/win>

### **5.1 Cấu hình git lần đầu**

**Danh Tính Của Bạn**

Việc đầu tiên bạn nên làm khi cấu hình Git là chỉ định tên tài khoản và địa chỉ e-mail. Điều này rất quan trọng vì mỗi Git sẽ sử dụng chúng cho mỗi lần commit, những thông tin này được gắn bất di bất dịch vào các commit:

$ git config --global user.name "phamthanh"

$ git config --global user.email phamthanh@gmail.com

Bạn chỉ phải làm việc này một lần duy nhất nếu như sử dụng --global, vì Git sẽ sử dụng các thông tin đó cho tất cả những gì bạn làm trên hệ thống. Nếu bạn muốn sử dụng tên và địa chỉ e-mail khác cho một dự án riêng biệt nào đó, bạn có thể chạy lại lệnh trên không sử dụng --global trên dự án đó

**Trình Soạn Thảo**

Bây giờ danh tính của bạn đã được cấu hình xong, bạn có thể lựa chọn trình soạn thảo mặc định sử dụng để soạn thảo các dòng lệnh. Mặc định, Git sử dụng trình soạn thảo mặc địch của hệ điều hành, thường là Vi hoặc Vim. Nếu bạn muốn sử dụng một trình soạn thảo khác, như Emacs, bạn có thể sửa như sau:

$ git config --global core.editor emacs

**Công Cụ So Sánh Thay Đổi**

Một lựa chọn hữu ích khác mà bạn có thể muốn thay đổi đó là chương trình so sánh sự thay đổi để giải quyết các trường hợp xung đột nội dung. Ví dụ bạn muốn sử dụng vimdiff:

$ git config --global merge.tool vimdiff

Git chấp nhận kdiff3, tkdiff, meld, xxdiff, emerge, vimdiff, gvimdiff, ecmerge, và opendiff là các công cụ trộn/sát nhập (merge) hợp lệ.

**Kiểm Tra Cấu Hình**

Nếu như bạn muốn kiểm tra các cấu hình cài đặt, bạn có thể sử dụng lệnh git config --list để liệt kê tất cả các cài đặt của Git:

$ git config --list

user.name=phamthanh

user.email=phamthanh@gmail.com

color.status=auto

color.branch=auto

color.interactive=auto

color.diff=auto

...

Bạn có thể thấy các từ khoá xuất hiện nhiều hơn một lần, bởi vì Git đọc chúng từ các tập tin khác nhau (ví dụ, /etc/gitconfig và ~/.gitconfig). Trong trường hợp này Git sử dụng giá trị xuất hiện cuối cùng cho mỗi từ khoá duy nhất.

Bạn cũng có thể kiểm tra giá trị của một từ khoá riêng biệt nào đó bằng cách sử dụng git config {key}:

$ git config user.name

phamthanh

### **5.1 Tạo một kho chứa git**

Bạn có thể tạo một dự án có sử dụng Git dựa theo hai phương pháp chính. Thứ nhất là dùng một dự án hay một thư mục đã có sẵn để nhập (import) vào Git. Thứ hai là tạo bản sao của một kho chứa Git đang hoạt động trên một máy chủ khác.

**Khởi tạo một kho chứa từ thư mục cũ**

Nếu như bạn muốn theo dõi một dự án cũ trong Git, bạn cần ở trong thư mục của dự án đó và gõ lệnh sau:

$ git init

Lệnh này sẽ tạo một thư mục mới có tên .git, thư mục này chứa tất cả các tập tin cần thiết cho kho chứa - đó chính là bộ khung/xương của kho chứa Git. Cho tới thời điểm hiện tại, vẫn chưa có gì trong dự án của bạn được theo dõi (track) hết.

Nếu bạn muốn kiếm soát phiên bản cho các tập tin có sẵn (đối lập với một thư mục trống), chắc chắn bạn nên bắt đầu theo dõi các tập tin đó và thực hiện commit đầu tiên/khởi tạo (initial commit). Bạn có thể hoàn thành việc này bằng cách chỉ định tập tin bạn muốn theo dõi trong mỗi lần commit sử dụng câu lệnh git add:

$ git add \*.c

$ git add README

$ git commit -m 'phiên bản đầu tiên/khởi tạo của dự án'