**TẤT TẦN TẬT VỀ**

**GIT CƠ BẢN**

**NOTEBOOK của Thug24**

**Cập nhật: 10/1/2023**

**Facebook: Tuấn Hưng**

**Github: gnuh24**

**Lời mở đầu**

Xin chào mọi người, tui là Hưng hiện tại tui chỉ đang là sinh viên bập bẹ tập code mà thôi. Và tui hay có thói quen là với mỗi môn hay là 1 chủ đề gì đó hay thì tui sẽ Note lại trên file Word rồi share cho anh em dĩ nhiên là sẽ không có liên quan tới buôn bán tài liệu gì ở đây. Đây chỉ là Notebook của tui trong quá trình học cho nên nếu có bất cứ sai xót nào thì mong anh em thông cảm và góp ý thông qua mạng xã hội tui để link bên dưới.

Về nội dung của file Notebook này thì chủ yếu về GIT. Các nguồn tham khảo thì mình sẽ để nó ở trang cuối và có đi kèm theo link. Notebook này sẽ tóm tắt lại những kiến thức mà mình thấy nó quan trọng. Notebook này cho anh em biết về các khái niệm về GIT, GITHUB và cách sử dụng. Đồng thời một số hình ảnh minh họa mình có lấy từ khóa học GIT của Udemy.

Facebook: <https://www.facebook.com/profile.php?id=100036421866670>

Github: <https://github.com/gnuh24>

**Mục lục**

[BÀI 1: GIỚI THIỆU VỀ GIT / GITHUB 5](#_Toc156058251)

[**I.** **GIT LÀ GÌ ?** 5](#_Toc156058252)

[**a)** **Version Management/Controll** 5](#_Toc156058253)

[**b)** **Ưu điểm của GIT** 6](#_Toc156058254)

[**c)** **Nhược điểm của GIT** 7](#_Toc156058255)

[**II.** **GITHUB LÀ GÌ ?** 7](#_Toc156058256)

[BÀI 2: VERSION MANAGEMENT VỚI GIT 8](#_Toc156058257)

[**I.** **GIT HOẠT ĐỘNG NHƯ THẾ NÀO ?** 8](#_Toc156058258)

[**a.** **Các khái niệm cơ bản** 8](#_Toc156058259)

[b. Luồng chạy 9](#_Toc156058260)

[c. Branch (Nhánh) 10](#_Toc156058261)

[d. Con trỏ HEAD 13](#_Toc156058262)

[e. Các thao tác CRUD đối với các file trong GIT 20](#_Toc156058263)

[**II.** **CÁC CÂU LỆNH CƠ BẢN VÀ CHỨC NĂNG** 21](#_Toc156058264)

[a. Thao tác với file 21](#_Toc156058265)

[b. Branching and Merging 23](#_Toc156058266)

[TÀI LIỆU THAM KHẢO 25](#_Toc156058267)

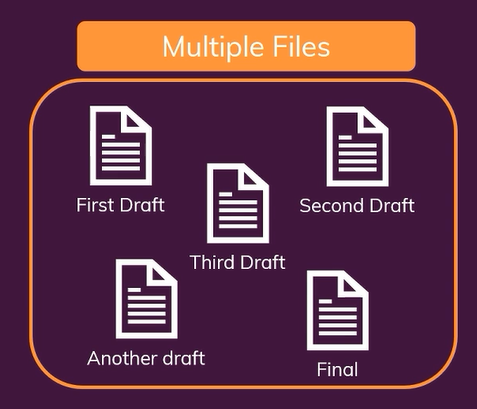
# BÀI 1: GIỚI THIỆU VỀ GIT / GITHUB

1. **GIT LÀ GÌ ?**

* Git là 1 Version Controll System (VCS) (Hệ thống quản lý các phiên bản.
  1. **Version Management/Controll**
* Đầu tiên giả sử như bạn đang phải code 1 project mà giảng viên giao cho và bạn đang muốn tìm cách quản lý project một cách hiệu quả. Lúc này ta sẽ thường sử dụng theo 2 cách truyền thống là

Multiple files hoặc Single File

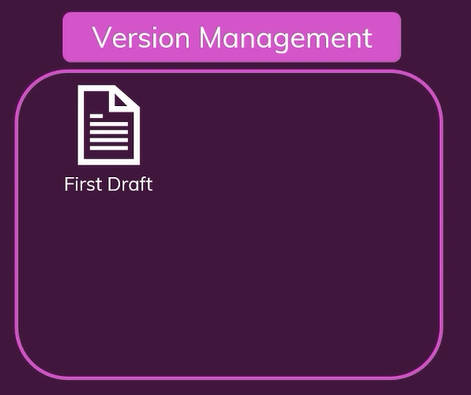
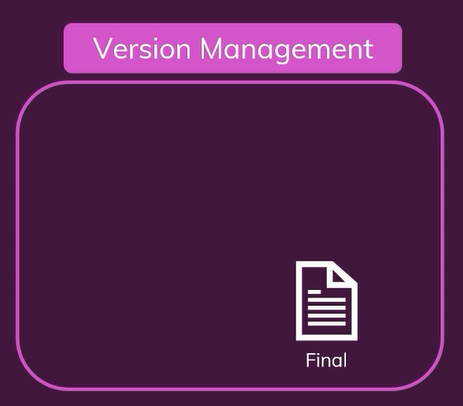
* Giải pháp 1 (Multiple files): Bạn phải code rất nhiều phiên bản khác nhau rồi sau cùng mới cho ra phiên bản cuối cùng của project đó. Lúc này bạn vẫn có thể tìm lại được các phiên bản cũ nếu như final project của bạn bị lỗi gì đó. Tuy nhiên cách làm này sẽ khiến cho folder của bạn bị rối và khá khó chịu trong việc quản lý. Vì vậy ta đi đến giải pháp thứ 2



* Giải pháp 2 (Single File): Lúc này thay vì copy tạo ra nhiều phiên bản khác nhau thì bạn chỉ dùng đúng 1 phiên bản. Lúc này folder của bạn sẽ rất gọn gàng vì chỉ có duy nhất 1 file project nhưng nếu project đó bị hỏng thì coi như ta không có bản back-up nào của phiên bản trước đó để dự trù.



* Sự hạn chế của 2 giải pháp trên khiến ta cần dùng đến giải pháp thử 3 **Version Management/Controll**
* Giải pháp 3 (**Version Management/Controll** ): Kỹ thuật này sẽ phép ta vừa code trên 1 file duy nhất mà không cần phải copy ra, nhưng khác với Single File là ta sẽ giữ được tất cả các phiên bản trước của project và ta có thể back up lại bất cứ lúc nào, bất cứ phiên bản nào mà ta muốn. Và chỉ có phiên bản cụ thể hiện tại của bạn đang dùng là có trên máy của chúng ta. Cụ thể cơ chế này hoạt động thế nào thì sang bài 2 chúng ta sẽ rõ.

**** ****

* **Để thực hiện giải pháp thứ 3 ta cần dùng đến các Version Controll System. Và Git là 1 trong số chúng.**
  1. **Ưu điểm của GIT**
* Là 1 (**VCS**) miễn phí và cục bộ
* Lưu được tất cả lịch sử:

+ Cái gì thay đổi ?

+ Thay đổi khi nào ?

+ Thay đổi ở đâu ?

* Theo dỏi và phát hiện những sự thay đổi của code.
* So sánh được các phiên bản khác nhau của code.
  1. **Nhược điểm của GIT**
* Vì là 1 hệ thống cục bộ cho nên ta chỉ có thể sử dụng nó trên 1 thiết bị.  
  **VD**: Ta có Project A trên máy A thì ta chỉ có thể sử dụng GIT trên máy A để tương tác với project A. Ta không thể dùng máy B (Có GIT) để quản lý project A được.
* Nếu ta bị mất thiết bị thì những dữ liệu cục bộ trong máy đó cũng sẽ bay mất và GIT không giúp ta khôi phục được.
* **Để khắc phục điểm yếu cục bộ này thì ta sẽ cần 1 thứ gì đó có thể đồng bộ và liên kết toàn bộ trong các thiết bị dữ liệu cục bộ lại với nhau.**
* **GITHUB là thứ ta cần tìm.**

1. **GITHUB LÀ GÌ ?**

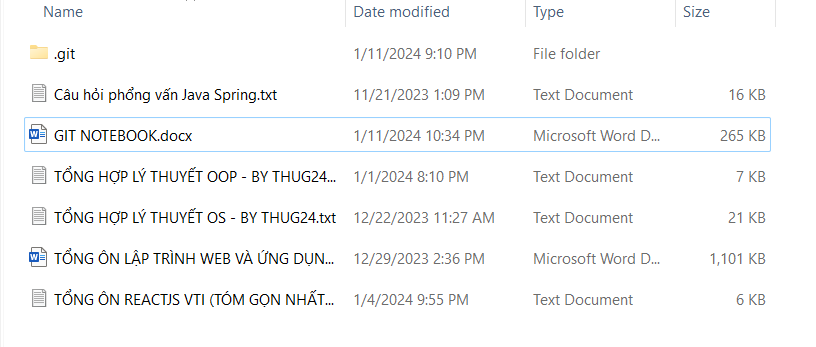
* Là một dịch vụ lưu trữ đám mây miễn phí
* Là một nền tảng hợp tác và quản lý phát triển các phần mềm.
* Cung cấp 1 kho lưu trữ GIT ngay trên đám mây (K
* Giải quyết được nhược điểm của GIT truyền thống
* Cho phép nhiều người dùng có thể truy cập vào cùng 1 Repository (Kho lưu trữ GIT) để cùng tương tác với 1 dự án.
* Github sử dụng Git để quản lý các repository vì thế mà ta có thể lưu lại lịch sử thay đổi của Repository

# BÀI 2: VERSION MANAGEMENT VỚI GIT

1. **GIT HOẠT ĐỘNG NHƯ THẾ NÀO ?**

* Dưới đây là 1 số khái niệm cơ bản mà ta cần phải biết về git. Lưu ý là chúng ta **chưa hề đụng tới github** cho nên hãy đọc kỹ để tránh hiểu nhầm khái niệm nhé ^^
  1. **Các khái niệm cơ bản**
* **Working Directory/Tree**: Là folder do git quản lý (repository nội bộ trên máy)
* Folder **.git**: Đây là dấu hiệu nhận biết xem folder đó có phải là **Working Directory/Tree** hay là folder bình thường. File này rất quan trọng vì nó lưu lại tất cả những gì ta thao tác với git bên trong. File này được chia thành 2 phần chính là **Staging Area** **(Index file)** và **Commits (Objects folder)**

VD: Đây là folder “Notebook” mà mình dùng để lưu các notebook lại và trong folder “Notebook” này chứa các thành phần dưới đây. Như mọi người đã thấy thì có 1 folder ẩn tên là .git . Đây là dấu hiệu nhận biết Folder “Notebook” của mình hiện tại đang là **Working Directory/Tree.** Thấy folder .git này rồi cũng đừng mở ra =)) Hiện tại chúng ta chưa hiểu rõ những gì bên trong đó đâu. Chỉ cần biết trong đó có 2 thành phần như mình đã liệt kê là được.



* **Staging Area (Index file):** Bạn có thể xem đây là 1 vùng nhớ tạm dùng để chuẩn bị cho sự việc commit. Các thao tác “git add” của chúng ta sẽ đều đưa những file thay đổi vào trong này.
* **Commits (Objects folder):** Đây là nơi dùng để lưu trữ những commit của chúng ta. Cứ mỗi lần bạn commit thì 1 commit mới sẽ được sinh ra và sẽ lưu **những thay đổi** so với lần commit trước đó.
* **NOTE!!:** Hãy chú ý về cụm từ “**những thay đổi**” với kiến thức hiện tại của mình thì mình chỉ biết cứ mỗi lần commit thì git sẽ lưu những thay đổi (sự khác biệt) của folder hiện tại so với đợt commit trước đó dưới 1 dạng đặc biệt nào đó khiến cho những commit này rất nhẹ trên máy nhưng không phải là copy lại các bản sau của file. Tất nhiên là git vẫn sẽ phải lưu bản copy đầu tiên của project để có thể làm mốc so sánh với các commit sau này.

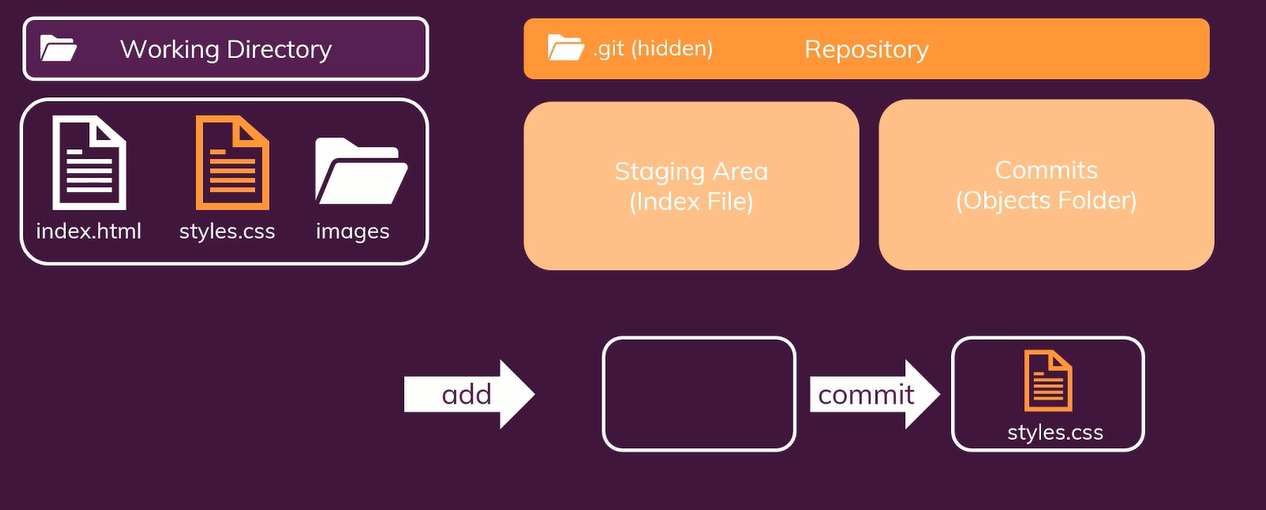
### Luồng chạy

**VD:** Ban đầu ta có Working Directory như hình và folder .git  
****

* Lúc này tôi đã có những thay đổi trong file styles.css. Sau đó tôi dùng lệnh **“git add .”** hoặc **“git add styles.css”** để thêm file này vào **Staging Area**

****

* Sau đó tôi sẽ tạo commit mới bằng “git commit -m `Commit đợt 1` ”. Lúc này git sẽ chính thức tạo 1 commit dựa trên những sự thay đổi bên trong **Staging Area** mà bạn đã add vào ban nảy.

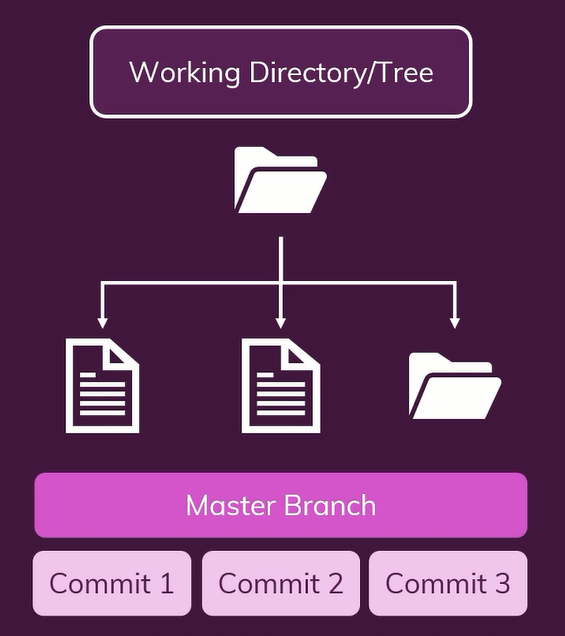
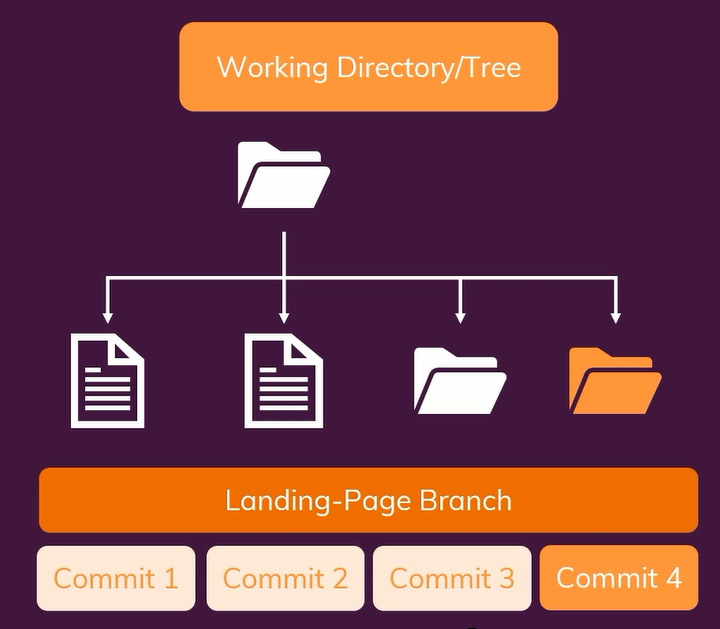
****

* **Quá trình trên sẽ lặp đi lặp lại liên tục cho tới khi project được hoàn tất.**

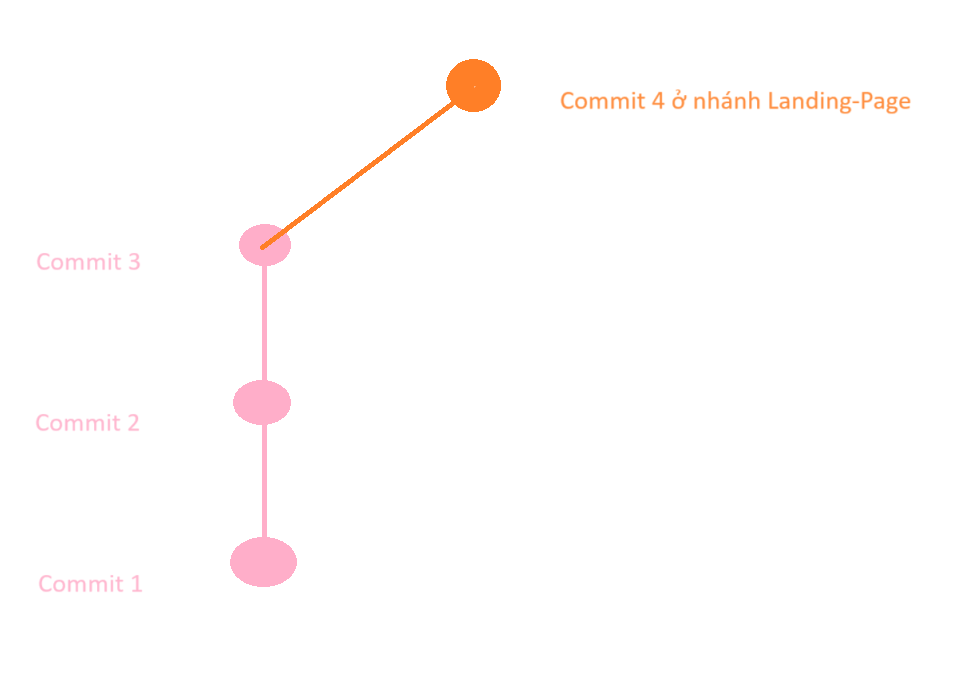
### Branch (Nhánh)

* Tất cả những commit của chúng ta ban đầu mặc định sẽ là của nhánh **master** . Cơ chế **branch** này được tạo ra chủ yếu giúp ta có thể phát triển những phiên bản khác nhau của code mà không đụng tới phiên bản chính nằm trong nhánh master

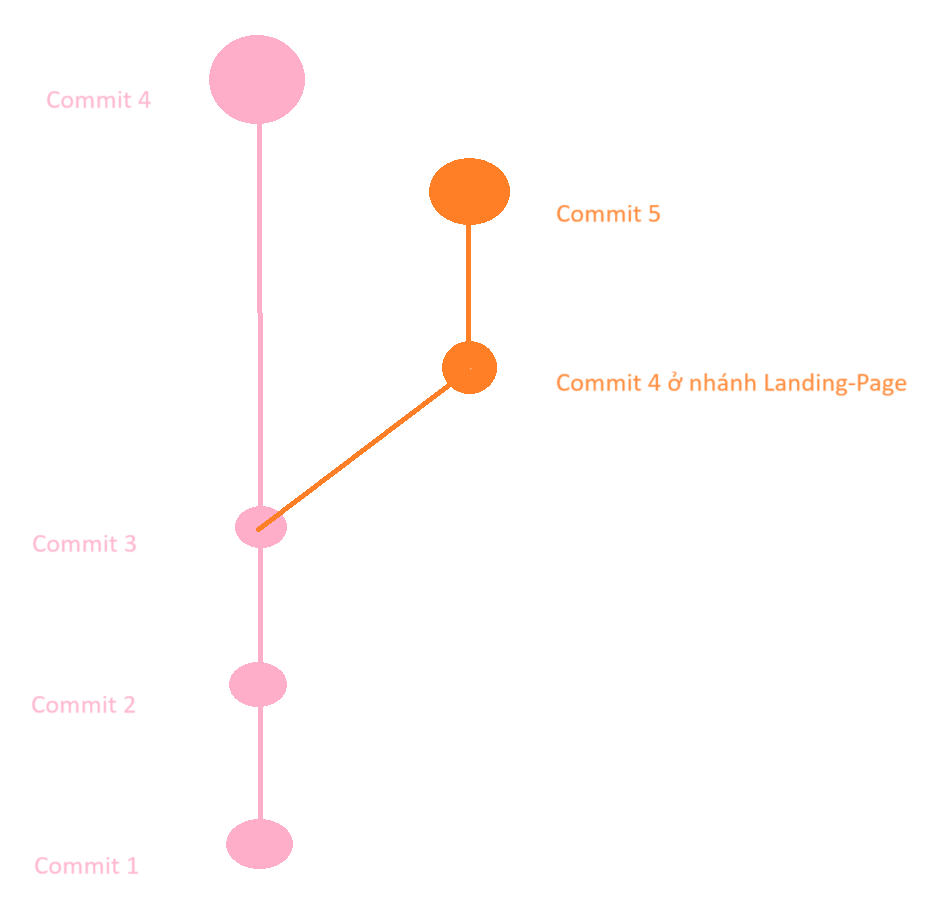
**VD:** Ta có nhánh master với 3 commit và từ commit thứ 3 này ta muốn phát triển thêm 1 chức năng mới nhưng không muốn đụng tới bản gốc. Lúc này ta tạo thêm 1 branch mới là Landing-Page để phát triển chức năng đó với commit thứ 4. Điều thú vị là ở nhánh này ta vẫn sẽ được kế thừa lại hết tất cả các commit trước đó như những commit 1 2 3 đều thuộc sở hữu của Landing-Page này.

**** ****

* Để dễ hình dung ta sử dụng Git graph để vẽ lại ví dụ trên như sau:



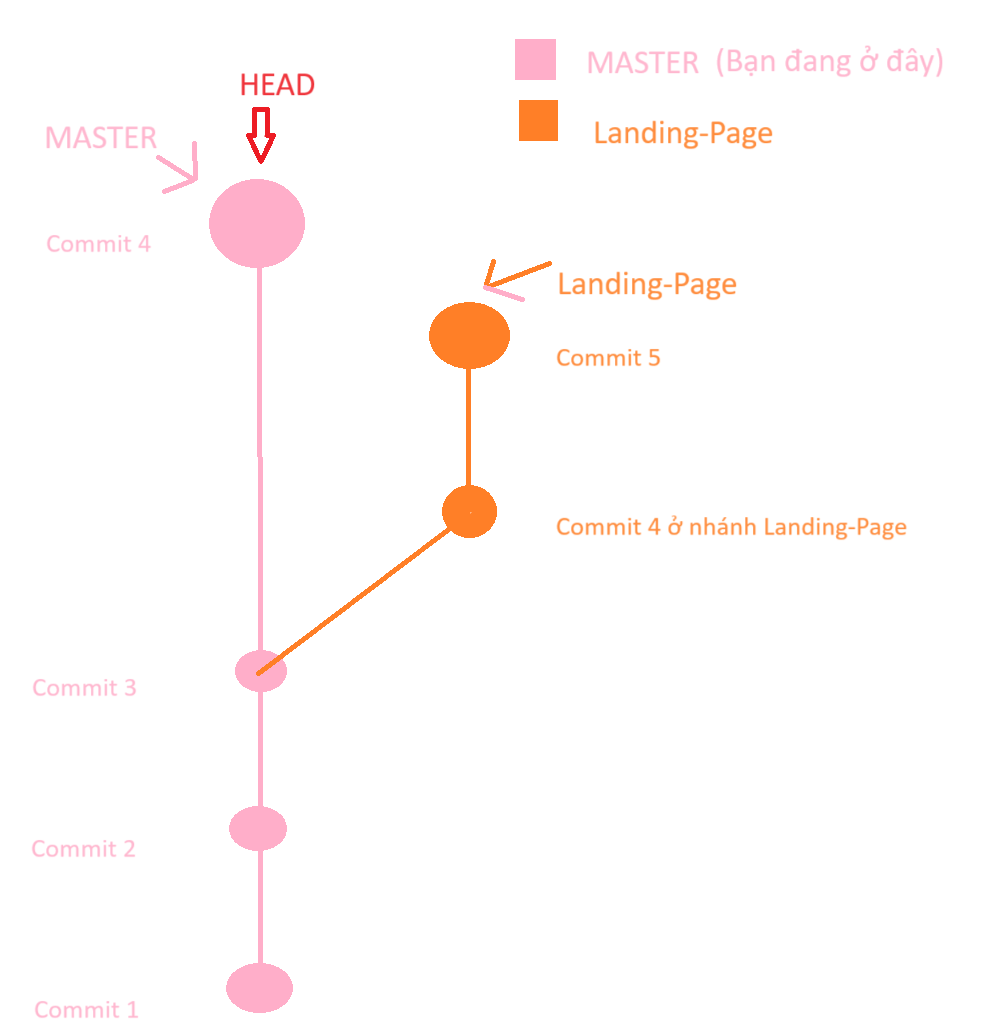
Bây giờ giả sử nhánh Master phát triển thêm commit mới và đồng thời nhánh Landing-Page cũng vậy thì ta sẽ có mô hình sau



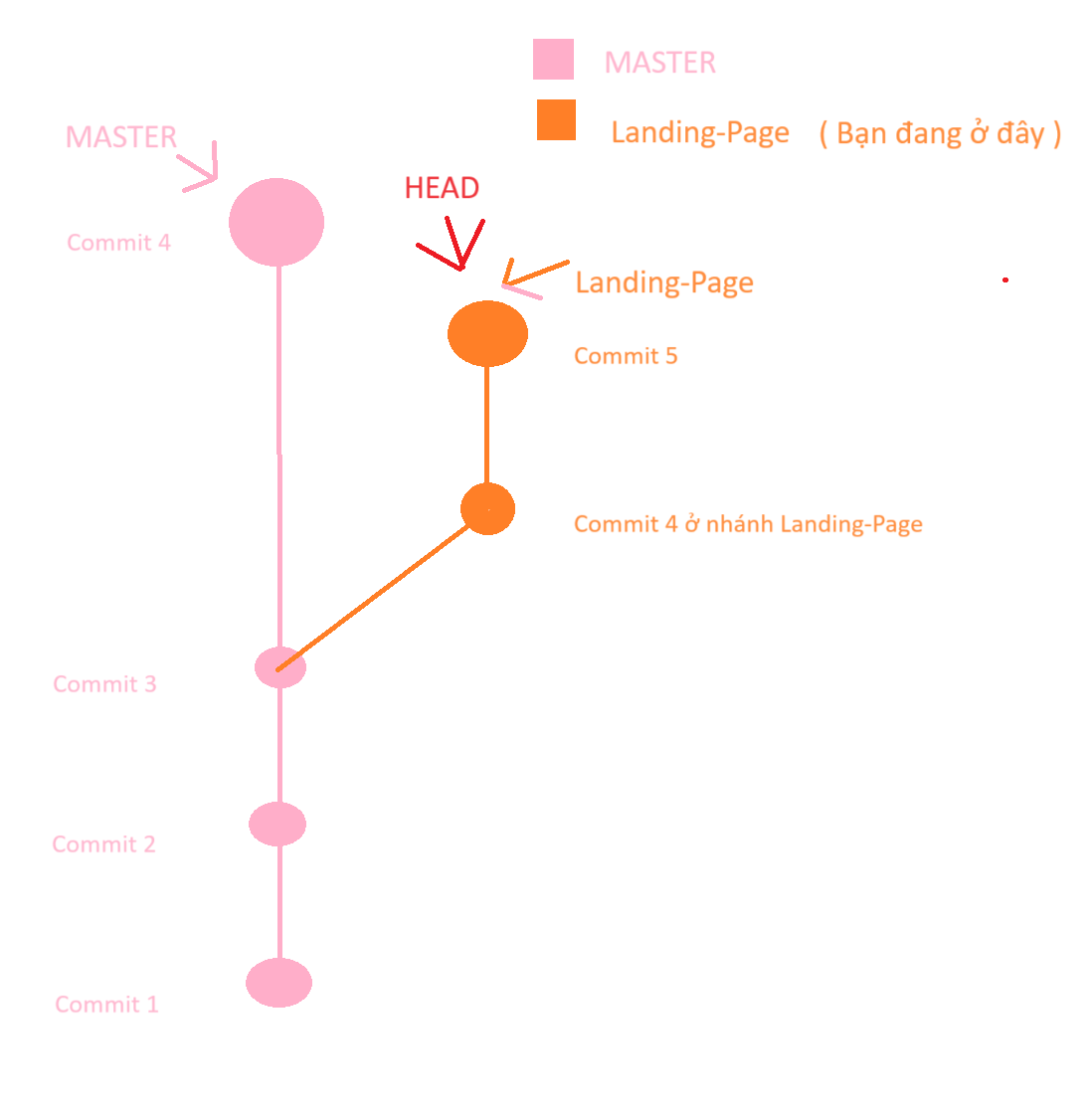
Lúc này ta đã thấy rõ sự khác biệt giữa 2 nhánh. Những dữ liệu ở Commit 5 và Commit thứ 4 bên nhánh Master sẽ không còn liên quan gì tới nhau nữa mà phát triển theo 2 hướng khác. Và khi nào nhánh Landing-Page này phát triển xong sẽ được merge lại với nhánh master. Kỹ thuật merge này ta sẽ tìm hiểu sau.

### Con trỏ HEAD

* Con trỏ HEAD này trỏ vào đâu thì Working Directory của chúng ta sẽ chứa data ở commit đó.
* Con trỏ HEAD này có 2 chế độ
* **Attached**
* Đây là chế độ mà con trỏ HEAD sẽ phụ thuộc vào nhánh hiện tại bạn đang đứng.
* Mặc định con trỏ HEAD sẽ luôn luôn trỏ vào Commit mà con trỏ của nhánh đó đang chạy. Hãy quay lại ví dụ ban nảy ta sẽ dễ hiểu hơn.
* Hiện tại mặc định ta đang ở nhánh master như bên dưới. Vì con trỏ Master đang ở Commit mới nhất của nhánh Master đó cho nên HEAD sẽ chạy theo con trỏ Master tới commit 4.

****

* Con trỏ HEAD sẽ dịch chuyển qua Commit mới nhất của Branch Landing-Page (Commit 5) nếu ta chuyển qua nhánh đó

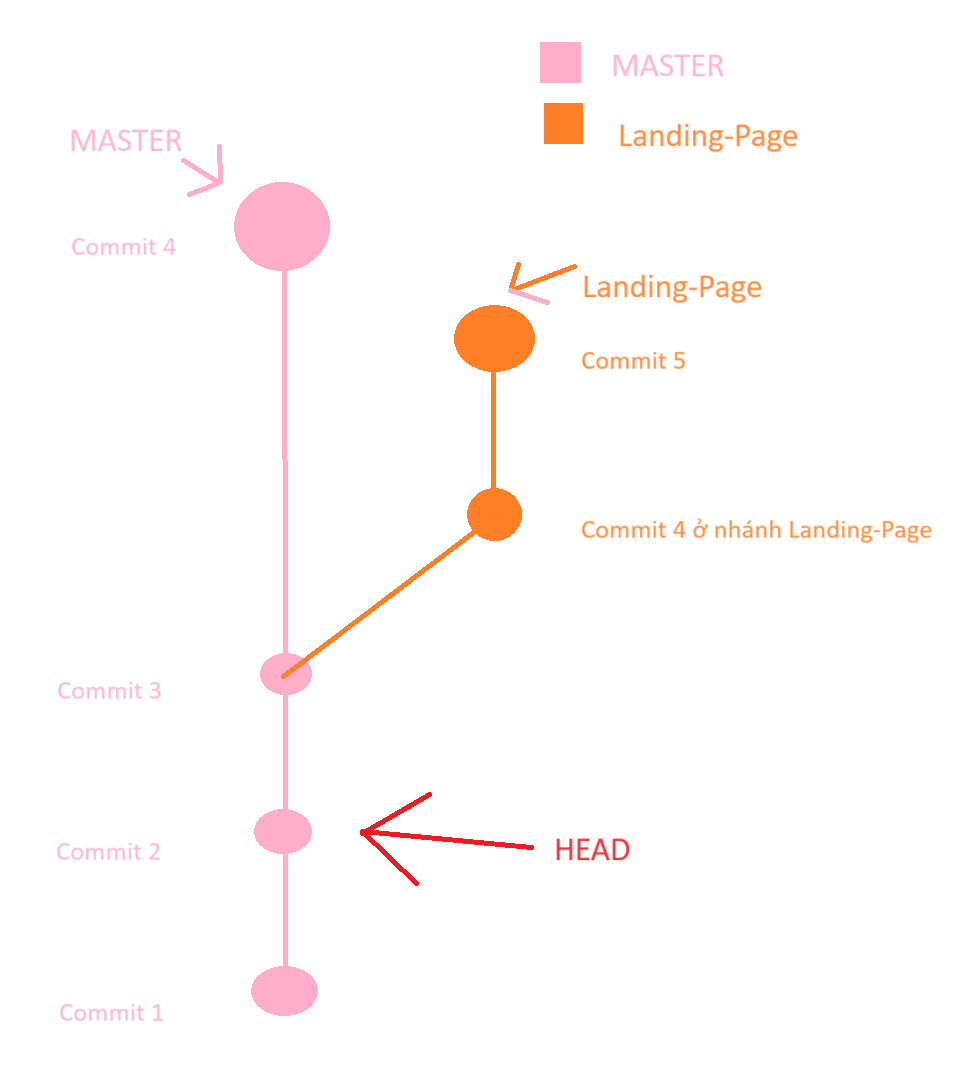


* **Detached**
* Đây là chế độ mà con trỏ HEAD sẽ được chạy tự do không phụ thuộc vào nhánh. Mà con trỏ HEAD sẽ nhảy thẳng tới ô commit mà chúng ta yêu cầu.
* Để vào chế độ này thì ta chỉ cần dùng  
  **git checkout <id của commit ta muốn chuyển tới>**

**VD:** Với ví dụ trên ta sẽ minh họa cho điều đó. Giả sử bạn đang ở nhánh master và bạn muốn chuyển sang commit 2

**git checkout <id của commit2>**

Lúc này con trỏ HEAD sẽ nhảy vào commit 2 và Working Directory của chúng ta sẽ chứ những thứ mà commit 2 đang chứa



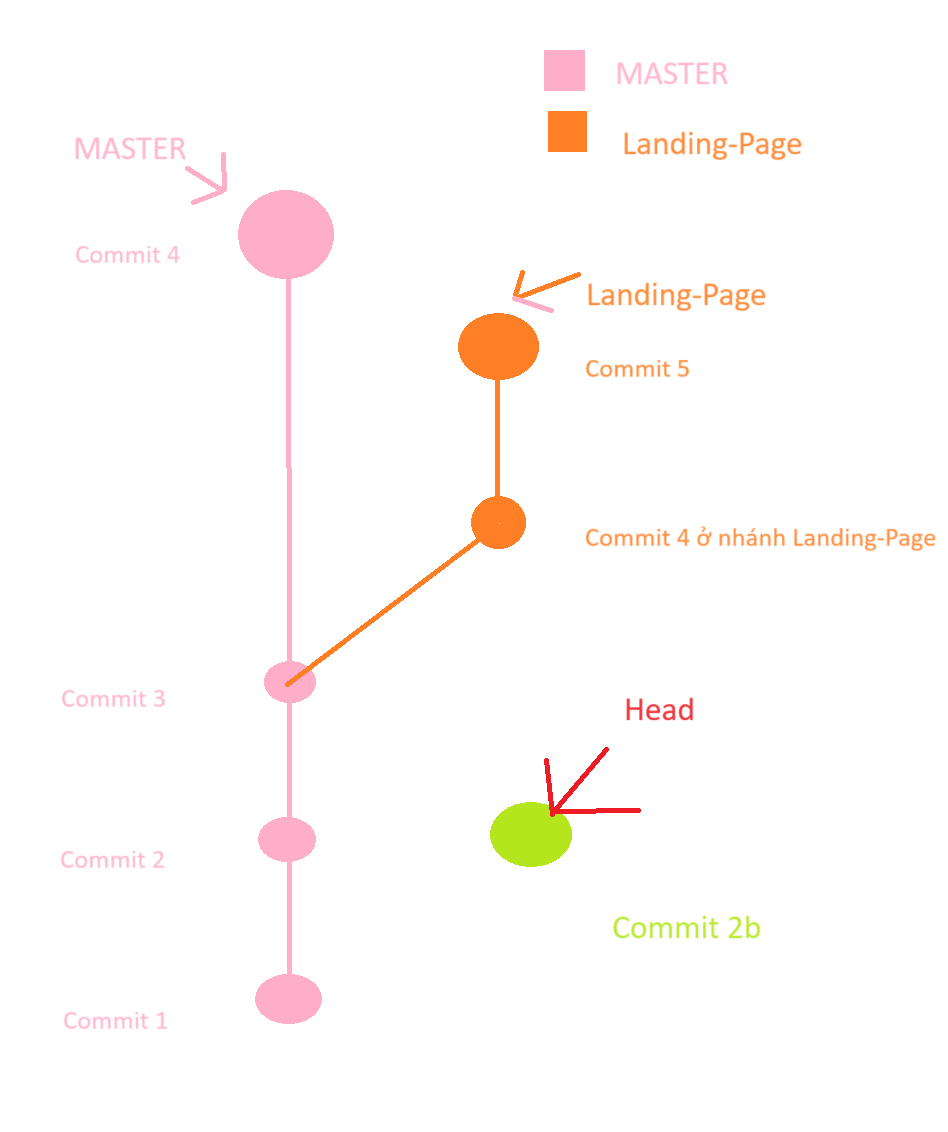
* Để thoát khỏi chế độ này thì ta chỉ cần dùng

**git checkout <tên nhánh>**

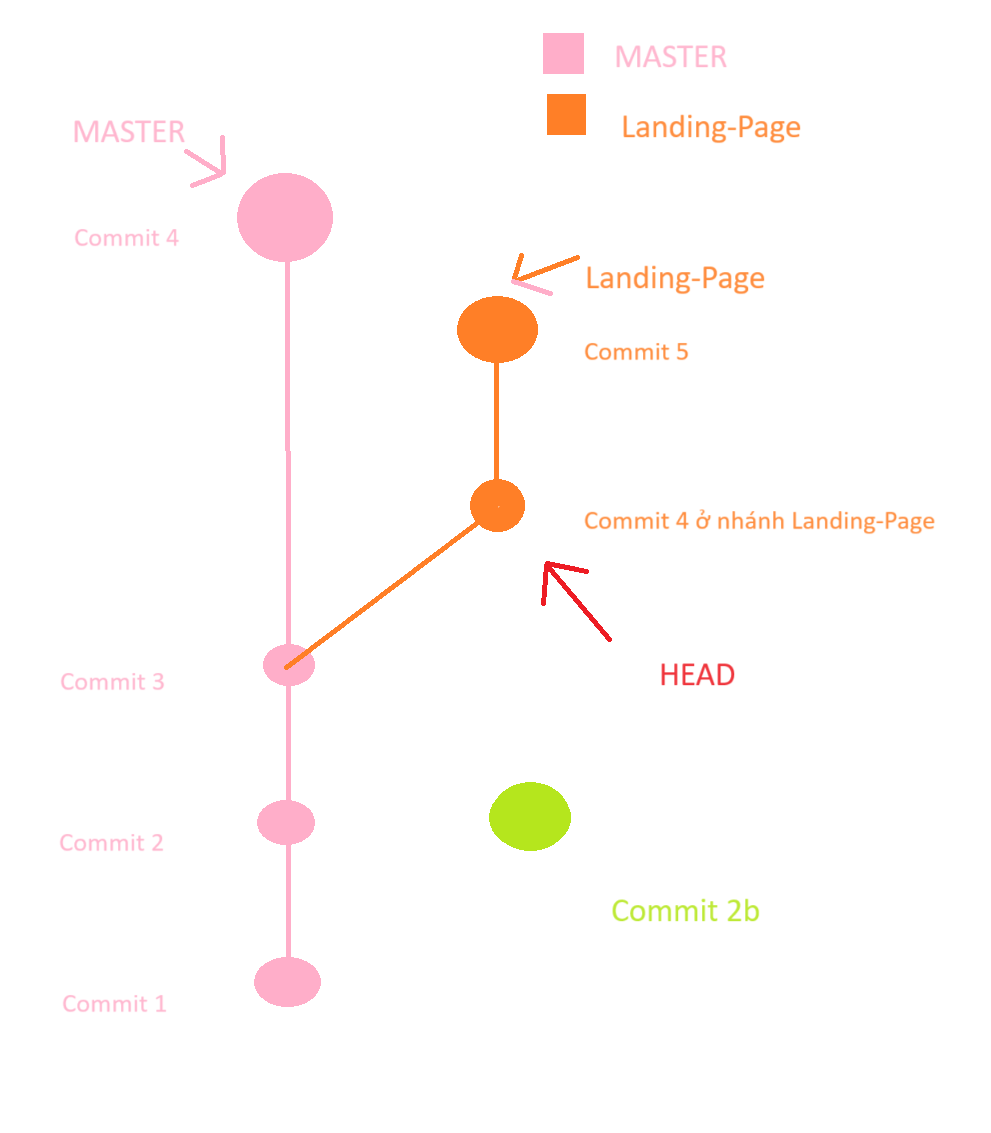
để chuyển tới nhánh khác, lúc này con trỏ HEAD sẽ vào lại trạng thái Attached và sẽ trỏ theo con trỏ của nhánh đó.

⚠️ **NOTE !!!**

* Nếu bạn đang ở Commit 2 nhờ chế độ Detached Head. Mọi commit của bạn lúc đó sẽ đều tạo ra các commit rời rạc (các commit không thuộc bất cứ branch nào).

VD: Ta đang ở Commit 2 và ta thay đổi 1 số thứ sau đó ta **git commit -m “Tạo commit 2b”** lúc này commit 2b này sẽ nằm biệt lập bên ngoài và chỉ do HEAD trỏ tới 

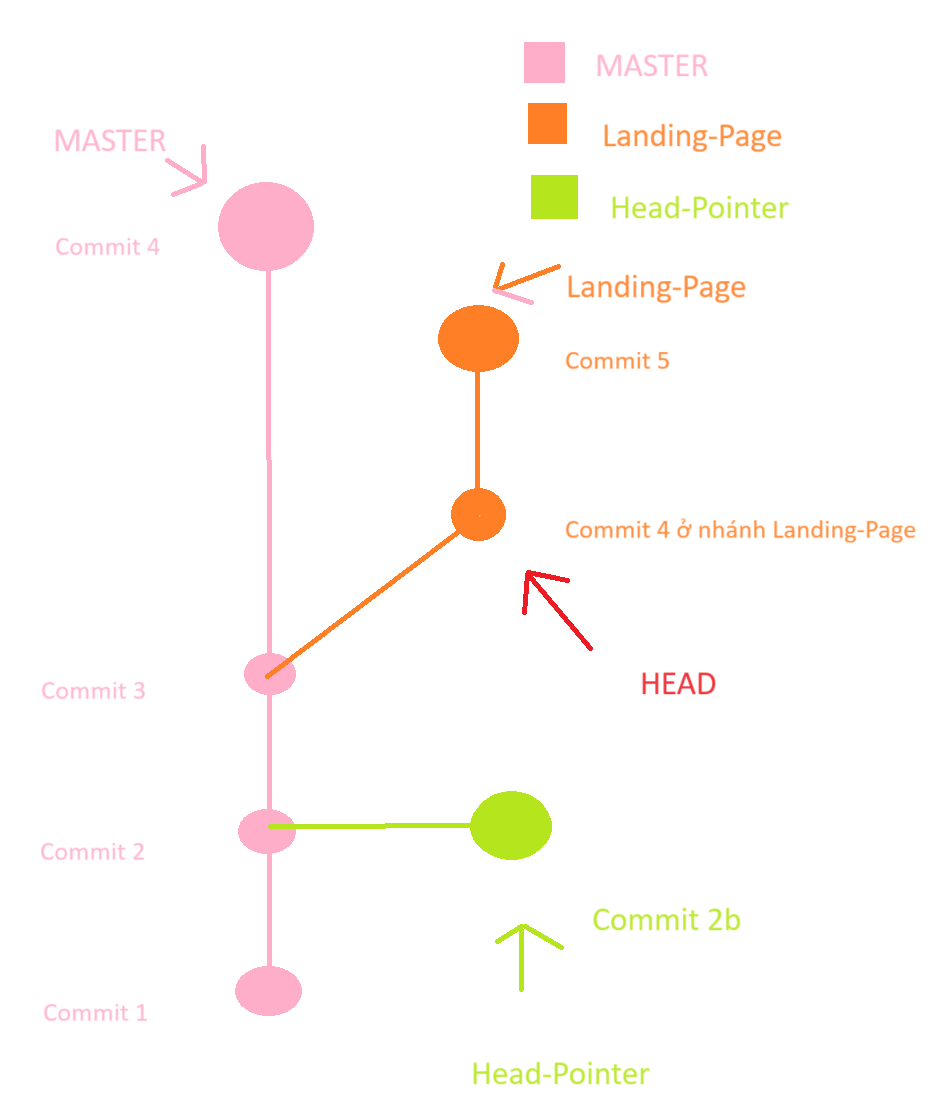
* Và khi bạn git checkgout con trỏ Head đi chỗ khác thì ta sẽ bị mất mối liên kết với commit 2b đó. Dĩ nhiên là vẫn sẽ có cách để xử lý vấn đề này mình sẽ trình bầy sau.



* Giải pháp:
  + - 1. **Vấn đề tạo 1 commit mới tại Detached Head**
* Sau khi tạo commit 2b xong ta sẽ tạo thêm 1 branch nữa để không bị mất đường liên kết.
* Dùng lệnh:

**git branch <Tên branch mới> <id của commit vừa tạo>**

Lúc này sơ đồ git của chúng ta sẽ như bên dưới. Giả sử ta tạo thêm nhánh Head-Pointer cho commit 2b



* + - 1. **Vấn đề lạc mất commit 2b**
* Dùng lệnh **git reflog**
* Lệnh này sẽ hiện lại toàn bộ những commit (Đầy đủ thông tin) mà con trỏ HEAD đã đi qua .
* Lúc này ta chỉ cần dùng **git checkout <id của commit bị lạc>** là ta đã có thể đưa con trỏ Head quay trở lại commit đó rồi quay lại giải pháp của vấn đề (1) là xong.

### Các thao tác CRUD đối với các file trong GIT

#### Vấn đề khi xóa file

* Có 2 cách xóa file chính:
* **rm <Tên file>**
* Đây là lệnh xóa file thông thường của hệ điều hành. Tuy nhiên nếu ta xóa file theo cách này thì ta buộc phải “**git add <Tên file>”** để add những sự thay đổi vào Staging Areasau đó **“git commit -m `Message` ”** để tạo ra một commit mới.
* **git rm <Tên file>**
* Đây là lệnh xóa file của git tuy nhiên flow bên dưới hệ thống của nó tương tự **git add**. Nếu **git add** thêm vào Staging Area về sự thay đổi so với commit gần nhất. Thì **git rm** sẽ báo cho Staging Area về một sự thay đổi tại file mà bạn đã khai báo trong lệnh đó.

Tức là với **rm <Tên file>** thì ta còn phải git add các thứ còn với **git rm** thì nó tương đương với bước xóa bằng **rm <Tên file> + git add <Tên file>**

#### Vấn đề khi thay đổi nội dung

* Giả sử bây giờ tôi có 1 file test.txt và tôi đã viết thêm 1 vài thứ mới vào (Đã có sự thay đổi so với commit gần nhất). Nhưng bây giờ tôi nội dung trong file của tôi quay trở lại so với commit gần nhất thì làm thế nào ?
* Có 2 cách:
* **git checkout <Tên file>**
* Đây là cách đầu tiên mà ta dùng để back nội dung của 1 file về với commit gần nhất. Tuy nhiên nó có 1 nhược điểm là nó chỉ có thể back về commit gần nhất.
* **git restore --source=<ID commit> --worktree <tên file>**
* Lệnh này được phát triển để xử lý khuyết điểm của **git checkout <Tên file>** . Lệnh này có thể đưa nội dung của 1 file về bất cứ call back nào trong quá khứ.

#### Vấn đề khi xóa 1 commit

**git revert <ID commit>**

* Lệnh trên dùng để xóa 1 commit

1. **CÁC CÂU LỆNH CƠ BẢN VÀ CHỨC NĂNG**

### Thao tác với file

#Nếu một folder bình sử dụng các lệnh của git hệ thống sẽ báo lỗi như hình bên dưới. Vì thế ta cần phải biến folder này thành **Working Directory/Tree** trước  

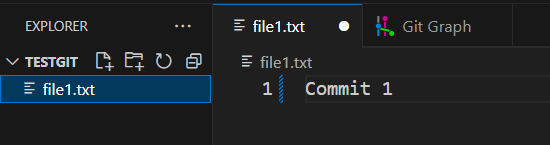

# Lệnh này dùng để biến 1 folder bình thường trở thành một **Working Directory/Tree**

**git init**

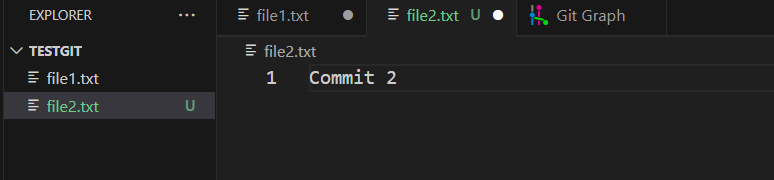
# Show ra trạng thái hiện tại của Working Directory của bạn đồng thời sẽ chỉ ra cụ thể file nào đã có sự thay đổi so với lần commit gần nhất

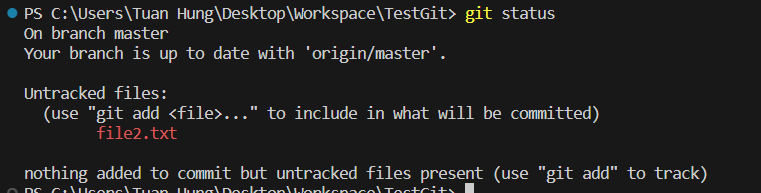
**git status**

**VD:** Hiện tại tôi đang có một file1.txt và tôi đã tạo xong 1 commit cho folder này.



Sau đó tôi quyết định tạo thêm 1 file mới là file2.txt và dùng lệnh git status để kiểm tra

****

****

Lúc này hệ thống báo rằng so với commit cũ thì đã có sự thay đổi tại file2.txt và gợi ý cho ta nên dùng “git add .”

# Dùng để add tất cả các file có sự thay đổi vào bên trong Staging Area.

**git add <file(s)>**

# Remove files from working tree and staging area

**git rm <file(s)>**

# Move or rename a file

**git mv <old\_file> <new\_file>**

# Tạo 1 commit với chú thích trong đó

**git commit -m "commit message"**

### Branching and Merging

# In ra danh sách tất cả các branch hiện tại trong local của bạn

**git branch**

# Tạo một nhánh mới

**git branch <branch\_name>**

# Di chuyển tới 1 nhánh khác

**git checkout <branch\_name>**

# Merge 1 nhánh khác vào nhánh hiện tại

**git merge <branch\_name>**

# Xóa một nhánh

**git branch -d <branch\_name>**

* Git sẽ đưa ra cảnh báo cho bạn nếu branch bạn xóa có biến)

**git branch -D <branch\_name>**

* (Lệnh này bỏ qua luôn cả cảnh báo)

#Xem lịch sử các commit của branch hiện tại

**git log**

Hiện tại thì ta chỉ cần biết nhiêu đây thôi. Nếu bạn muốn tìm hiểu thêm các lệnh nâng cao hãy kéo xuống phần tài liệu tham khảo và sang mục số 4 (Các lệnh cơ bản) của Tài liệu tham khảo

# TÀI LIỆU THAM KHẢO

* Khóa học GIT Udemy: <https://www.udemy.com/course/git-github-practical-guide/>
* Kênh youtube: <https://www.youtube.com/@TITVvn>
* Web thực hành: <https://learngitbranching.js.org/?locale=vi>
* Các lệnh cơ bản: <https://cs.fyi/guide/git-cheatsheet>
* Software Development: <https://www.atlassian.com/git>
* Tìm hiểu về Staging và Commits: <https://thachpham.com/tools/hieu-them-ve-commit-va-staging-area-git.html>