Tuần 2 - Tổng hợp kiến thức Buổi học số 5

Time-Series Team

Ngày 17 tháng 6 năm 2025

Buổi học số 5 (Thứ 7, 14/06/2025) bao gồm bốn nội dung chính:

- ***Phần I: Lịch sử hình thành Git và Tính chất Git***
- ***Phần II: Quy trình làm việc với Git cơ bản***
- ***Phần III: Quản lý nhánh và lịch sử***
- ***Phần IV: Thiết lập tài khoản và bảo mật trên Github***
- ***Phần V: Tùy chỉnh và Mở rộng Git***

Phần I: Lịch sử hình thành Git và Tính chất của Git

1 Version Control System (VCS)

1.1 Khái niệm

Version Control System (VCS) hay hệ thống quản lý phiên bản là một công cụ giúp theo dõi và quản lý tất cả các thay đổi được thực hiện lên mã nguồn hoặc tài liệu theo thời gian. VCS cho phép bạn quay lại các phiên bản trước đó, xem lịch sử chỉnh sửa, so sánh sự khác biệt, và làm việc cộng tác một cách an toàn.

1.2 Vì sao cần có VCS?

- **Phục hồi khi xảy ra lỗi:** Nếu bạn mắc lỗi hoặc hệ thống bị sự cố, bạn có thể quay lại phiên bản ổn định trước đó một cách dễ dàng.
- **Làm việc nhóm hiệu quả:** VCS cho phép nhiều người làm việc trên cùng một dự án mà không ghi đè lên công việc của nhau.
- **Theo dõi thay đổi:** Dễ dàng xác định ai đã thay đổi gì, khi nào và tại sao (thông qua commit message).
- **Thử nghiệm dễ dàng:** Bạn có thể tạo các nhánh (branch) để thử nghiệm tính năng mới mà không ảnh hưởng đến phiên bản chính (main branch).

1.3 Ví dụ giải thích vì sao cần có VCS

Giả sử bạn đang viết một luận văn và lưu nội dung trong file `luanvan.docx`. Mỗi khi bạn sửa, bạn lại lưu thành một bản mới như:

- `luanvan_v1.docx`
- `luanvan_v2.docx`
- `luanvan_final.docx`

- `luanvan_final_real.docx`

Cách làm này dễ gây nhầm lẫn, không rõ đâu là bản chính xác, và khó hợp tác với người khác.

Nếu bạn sử dụng một hệ thống VCS như Git, bạn chỉ cần một tệp duy nhất và mọi thay đổi được theo dõi rõ ràng, có thể quay lại bất kỳ thời điểm nào, so sánh nội dung, hoặc hợp nhất nội dung từ các cộng tác viên khác.

1.4 Phân loại VCS

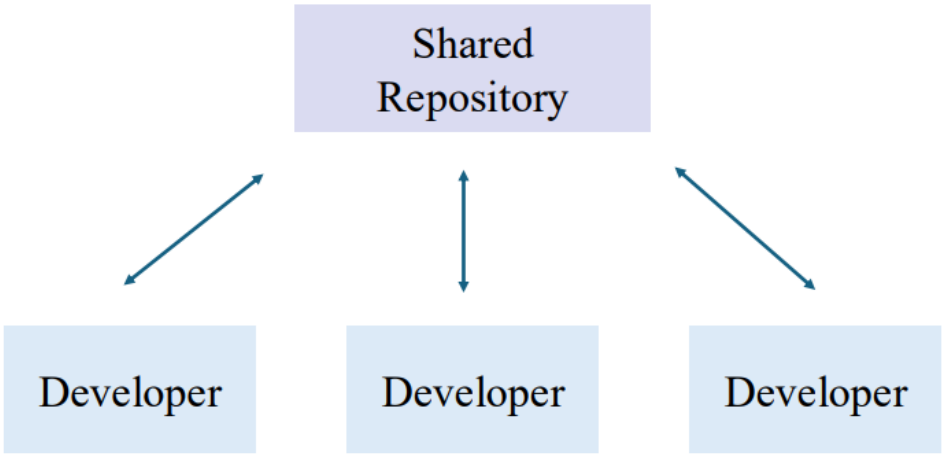
Version Control System được chia thành ba loại chính:

1.4.1 Local Version Control System

- Quản lý phiên bản cục bộ, lưu thông tin thay đổi ngay trên máy cá nhân.
- Không hỗ trợ làm việc nhóm từ xa.
- Ví dụ: `RCS (Revision Control System)`.

1.4.2 Centralized Version Control System (CVCS)

- Có một máy chủ trung tâm lưu trữ toàn bộ mã nguồn và lịch sử thay đổi.
- Các lập trình viên kết nối tới server để lấy hoặc gửi mã.
- Nhược điểm: nếu server gặp sự cố, có thể mất toàn bộ dữ liệu.
- Ví dụ: `CVS`, `Subversion (SVN)`, `Perforce`.

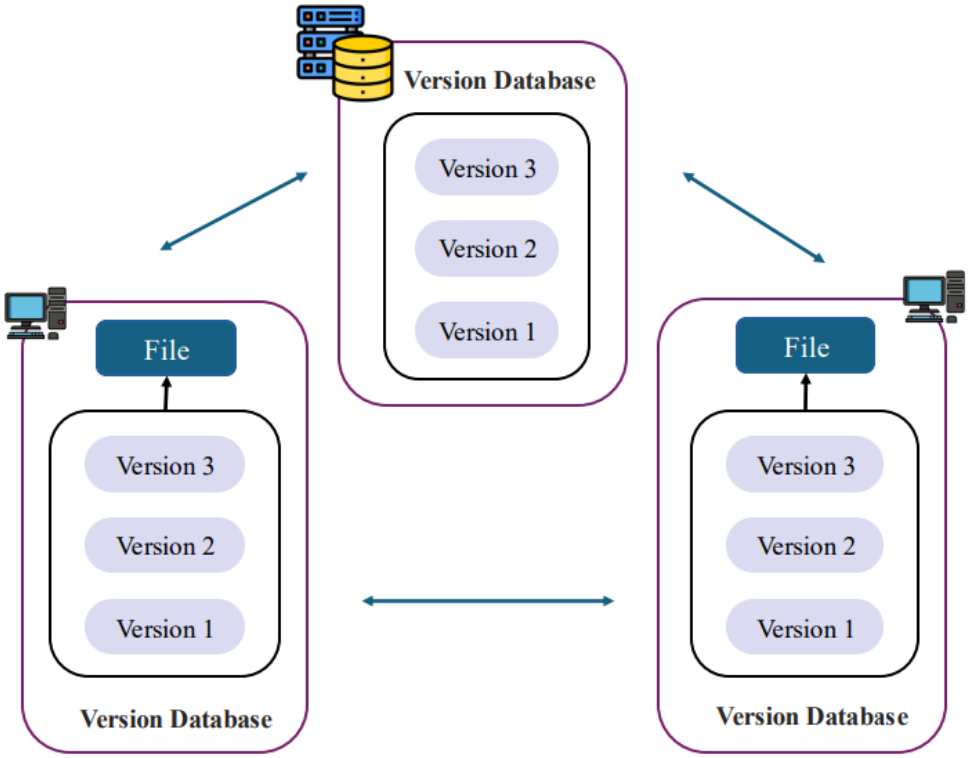


Hình 1: Centralized VCS

1.4.3 Distributed Version Control System (DVCS)

- Mỗi máy khách đều có một bản sao đầy đủ của toàn bộ repository.
- Cho phép làm việc offline và đồng bộ sau.

- An toàn, hiệu quả, phù hợp với dự án lớn và cộng đồng mở.
- Ví dụ: `Git`, `Mercurial`, `Bazaar`.



Hình 2: Distributed VCS

2 Lược sử Git

2.1 Bối cảnh ra đời (2005)

Vào năm 2005, nhóm phát triển nhân Linux gặp mâu thuẫn với công ty cung cấp công cụ quản lý mã nguồn BitKeeper về quyền sử dụng. Điều này khiến nhóm Linux không còn có thể sử dụng BitKeeper miễn phí nữa.

2.2 Người sáng lập

Trước tình thế đó, **Linus Torvalds** – người sáng lập Linux – đã quyết định tạo ra một hệ thống quản lý phiên bản mới mang tên **Git**, nhằm phục vụ riêng cho nhu cầu phát triển mã nguồn mở của dự án Linux.

2.3 Mục tiêu thiết kế

Git được Linus thiết kế với các mục tiêu chính:

- Nhanh (performance cao)
- Đơn giản (simple usage)

- Dễ phân nhánh và gộp nhánh (branching/merging dễ dàng)
- Hỗ trợ làm việc phân tán (distributed system)

2.4 Quá trình phát triển

Chỉ sau vài tháng phát triển, Git đã thay thế hoàn toàn BitKeeper trong dự án Linux. Sau đó, nó nhanh chóng trở thành công cụ phổ biến trong cộng đồng mã nguồn mở và cả doanh nghiệp.

3 Nguyên lý hoạt động của Git

3.1 Snapshot thay vì Diff

- Khác với các hệ thống VCS cũ thường lưu các thay đổi (diff) theo dòng, Git lưu một bản **chụp toàn bộ (snapshot)** dự án mỗi khi thực hiện `commit`.
- Nếu một tệp không thay đổi, Git chỉ tạo liên kết đến bản trước đó, giúp tiết kiệm dung lượng và tăng hiệu suất truy xuất.

Ví dụ: Bạn có một thư mục dự án chứa 3 tệp: `index.html`, `style.css`, và `script.js`. Sau lần commit đầu tiên, Git tạo bản snapshot đầy đủ của cả ba tệp. Giả sử bạn chỉ thay đổi nội dung trong `index.html`, và thực hiện commit lần 2. Git sẽ:

- Tạo bản snapshot mới cho `index.html`
- Tái sử dụng liên kết của `style.css` và `script.js` từ snapshot trước (vì không thay đổi)

Điều này giúp tiết kiệm dung lượng và cải thiện hiệu suất hơn so với việc ghi lại từng dòng thay đổi như các VCS kiểu cũ.

3.2 Toàn vẹn dữ liệu với SHA-1

- Git sử dụng hàm băm `SHA-1` để định danh tất cả nội dung: tệp, thư mục, commit, v.v.
- Mỗi đối tượng có một mã hash 40 ký tự, giúp bảo đảm nội dung không bị thay đổi ngoài ý muốn.
- Nếu một tệp bị thay đổi dù chỉ một ký tự, mã SHA-1 cũng sẽ hoàn toàn khác.

Ví dụ: Tệp `index.html` có nội dung:

```
<h1>Hello World</h1>
```

Git tính toán mã băm SHA-1 cho nội dung trên, ví dụ:

```
fa49b077972391ad58037050f2a75f74e3671e92
```

Nếu bạn thêm một dấu chấm:

```
<h1>Hello World.</h1>
```

Thì SHA-1 mới sẽ hoàn toàn khác:

```
2fd4e1c67a2d28fced849ee1bb76e7391b93eb12
```

Điều này đảm bảo rằng bất kỳ thay đổi nào trong nội dung đều được Git phát hiện và ghi nhận.

3.3 Nguyên lý "Offline-first"

- Git cho phép hầu hết thao tác (commit, branch, merge, v.v.) thực hiện **offline** mà không cần kết nối Internet.
- Chỉ khi cần `push` hoặc `pull` từ kho lưu trữ trung tâm thì mới cần mạng.

Ví dụ: Bạn đang làm việc tại quán cà phê không có Wi-Fi. Với Git, bạn vẫn có thể:

- Thêm file mới (`git add`)
- Ghi lại thay đổi (`git commit`)
- Tạo nhánh thử nghiệm (`git branch new-feature`)
- Chuyển nhánh và gộp (`git checkout`, `git merge`)

Chỉ khi có kết nối Internet, bạn mới cần sử dụng `git push` hoặc `git pull` để đồng bộ với repository trên GitHub hoặc GitLab.

4 Cài đặt và cấu hình Git

4.1 Cài đặt Git

- **Linux (Debian/Ubuntu):** `sudo apt install git`
- **Linux (Fedora/RHEL):** `sudo dnf install git`
- **macOS:** `xcode-select --install` hoặc tải từ https://git-scm.com
- **Windows:** Tải từ https://git-scm.com hoặc dùng Chocolatey: `choco install git`

4.2 Cấu hình cơ bản

```
git config --global user.name "Tên Của Bạn"
git config --global user.email "email@example.com"
git config --global core.editor "code --wait"
git config --global color.ui auto
```

4.3 Thiết lập Alias hữu ích

Giúp rút gọn các lệnh Git thường dùng:

```
git config --global alias.st status
git config --global alias.co checkout
git config --global alias.br branch
git config --global alias.cm "commit -m"
```

5 Hỗ trợ và trợ giúp

5.1 Tài liệu chính thức

- **Trang chủ:** https://git-scm.com – cung cấp đầy đủ tài liệu, tải phần mềm và các tài nguyên học Git.
- **Sách "Pro Git":** Miễn phí, chi tiết và thực hành tốt, có thể đọc tại https://git-scm.com/book.

5.2 Trợ giúp từ dòng lệnh

- `git help`
- `git --help`
- `man git-<lệnh>`

Giúp hiển thị tài liệu của bất kỳ lệnh Git nào.

5.3 Cộng đồng hỗ trợ

- Stack Overflow (https://stackoverflow.com)
- GitHub Discussions
- Reddit #git hoặc các diễn đàn kỹ thuật

5.4 Tài nguyên học tập khác

- Khóa học online: Udemy, Coursera, freeCodeCamp
- Video hướng dẫn trên YouTube
- Các trang blog như Atlassian Git tutorials, Git Tower, Dev.to

Phần II: Quy trình làm việc Git cơ bản

1 Khởi tạo và sao chép kho (Repository)

1.1 Khởi tạo repository mới

```
mkdir <new_project_name>
cd <new_project_name>
git init
```

Lệnh `git init` tạo thư mục ẩn `.git/` để Git lưu trữ dữ liệu và theo dõi dự án.

1.2 Sao chép repository từ xa

```
git clone https://github.com/username/repository.git
git clone git@github.com:username/repository.git
git clone https://github.com/username/repository.git <new_name>
```

Git sẽ tải về toàn bộ lịch sử và các nhánh của dự án.

2 Theo dõi và lưu trữ thay đổi

2.1 Trạng thái file trong Git

Mỗi file trong thư mục làm việc có thể thuộc một trong bốn trạng thái:

- **Untracked**: File mới chưa được Git theo dõi.
- **Modified**: File đã thay đổi nhưng chưa được chuẩn bị để commit.
- **Staged**: File đã được đưa vào vùng chuẩn bị để commit.
- **Committed**: File đã được lưu trữ an toàn trong Git.

2.2 Các lệnh theo dõi và lưu trữ

Xem trạng thái hiện tại:

```
git status
```

Đưa file vào vùng chuẩn bị (staging):

```
git add tên-file.txt      # Add one file
git add .                 # Add all update files
```

Lưu các thay đổi đã chuẩn bị:

```
git commit -m "<commit message>"
```

2.3 Sử dụng file .gitignore

File `.gitignore` liệt kê các file và thư mục mà Git sẽ bỏ qua, ví dụ:

```
# Ignore build folder
/build/
# Ignore files with ending .log
*.log
# Ignore files with specific type
config.ini
.env
```

3 Xem lịch sử commit

```
git log -p                      # Show detailed differences for each commit
git log --stat                  # Show number of files and lines changed in each commit
git log --since="2 weeks ago" # Filter commits from the last 2 weeks
git log --author="Name"         # Filter commits by author
git log --oneline --graph --all # Display concise graphical history of all branches
```

4 Hoàn tác thay đổi

Tình huống	Lệnh	Mô tả
Đã sửa file nhưng chưa staged	`git restore <file>`	Khôi phục file về trạng thái commit trước
Đã staged nhưng muốn bỏ stage	`git restore --staged <file>`	Đưa file từ staged về modified
Muốn sửa commit gần nhất	`git commit --amend`	Sửa message hoặc thêm file vào commit cuối
Muốn hoàn tác commit	`git revert <commit-hash>`	Tạo commit mới để hoàn tác thay đổi

Lưu ý: `git restore` sẽ xóa thay đổi chưa commit và không thể khôi phục! Dùng `git stash` để lưu tạm:

```
git stash       # Temporarily save changes
git stash pop   # Reapply the saved changes
```

5 Làm việc với remote repository

Remote repository là phiên bản lưu trữ dự án từ xa trên máy chủ như GitHub, GitLab, Bitbucket.

5.1 Xem danh sách remote

```
git remote -v
git remote add origin https://github.com/username/repository.git
git remote remove origin
```

5.2 Đẩy và kéo thay đổi

```
git push origin main
git pull origin main
git fetch
git merge origin/main
```

6 Tagging – Đánh dấu phiên bản

6.1 Annotated Tag

Có thông tin người tạo, ngày, message. Dùng cho bản phát hành chính thức.

```
git tag -a v1.0 -m "Version 1.0 - Phát hành chính ứthc"
```

6.2 Lightweight Tag

Chỉ là con trỏ đơn giản tới commit.

```
git tag v1.0-beta
```

6.3 Đẩy tag lên remote

```
git push origin v1.0
git push origin --tags
```

Xem nội dung tag:

```
git show v1.0
```

Liệt kê tất cả tag:

```
git tag
```

Tìm kiếm tag theo mẫu:

```
git tag -l "v1.*"
```

7 Tạo branch và merge

7.1 Giải thích đơn giản về việc tạo branch và merge

Tưởng tượng dự án như một cây lớn. - **Branch** là các nhánh nhỏ của cây, mỗi nhánh phát triển riêng biệt. - Bạn làm việc trên nhánh riêng để tránh ảnh hưởng nhánh chính. - Khi xong việc, bạn gộp nhánh nhỏ lại với nhánh chính (merge). - Nếu có xung đột (conflict) như hai người cùng sửa một nhánh cây khác nhau, bạn cần chọn hoặc hòa giải sao cho cây khỏe mạnh.

Ví dụ: Bạn và bạn bè cùng viết một cuốn sách, mỗi người viết trên bản nháp riêng. Sau đó hợp nhất các chương lại thành bản chính. Nếu hai người viết đoạn văn khác nhau ở cùng chỗ, bạn phải đọc và chỉnh sửa lại cho hợp lý.

7.2 Tạo branch

```
git branch <branch_name>           # Create a new branch
git checkout <branch_name>         # Switch to the branch
git checkout -b <branch_name>      # Create and switch to the branch at the same time
```

Ví dụ:

```
git checkout -b feature-login
```

7.3 Merge branch

Chuyển về branch chính (ví dụ `main`):

```
git checkout main
```

Merge branch phụ (ví dụ `feature-login`):

```
git merge feature-login
```

8 Xử lý xung đột (Conflict) khi merge

8.1 Khi nào xảy ra conflict?

Conflict (xung đột) xảy ra khi hai nhánh (`branch`) cùng sửa một phần giống nhau trong cùng một file nhưng với nội dung khác nhau. Git không thể tự quyết định nên giữ phiên bản nào, nên sẽ dừng lại và yêu cầu bạn xử lý thủ công.

8.2 Cách xử lý conflict:

Giả sử bạn đang ở nhánh `main` và thực hiện merge với nhánh `feature-login`:

```
git merge feature-login
```

Nếu có conflict, Git sẽ hiện thông báo lỗi và đánh dấu những vùng bị xung đột trong file như sau:

```
<<<<<<< HEAD
Nội dung trong nhánh hiện tại (main)
=======
Nội dung trong nhánh được merge (feature-login)
>>>>>>> feature-login
```

1. Mở file bị conflict.
2. Sửa lại nội dung theo ý muốn (giữ phiên bản nào hoặc kết hợp cả hai), sau đó xóa các dòng đánh dấu `<<<<<<<`, `=======`, `>>>>>>>`.

   Ví dụ sau khi chỉnh sửa:

   ```
   Nội dung đã được chỉnh sửa phù hợp (kết hợp từ cả hai nhánh)
   ```

3. Lưu lại file như bình thường.
4. Đánh dấu rằng file đã được xử lý xong conflict:

   ```
   git add <tên_file>
   ```

5. Cuối cùng, tiếp tục commit để hoàn tất merge:

   ```
   git commit
   ```

Lưu ý: Nếu sử dụng VS Code hoặc các IDE hiện đại, bạn có thể xử lý conflict thông qua giao diện trực quan dễ dàng hơn.

Phần III: Quản lý nhánh và lịch sử

1 Nhánh là gì?

Nhánh là Con trỏ

Nhánh trong Git giống như một **con trỏ** trỏ đến một **commit cụ thể** trong lịch sử thay đổi của dự án.

- Khi tạo một commit mới, con trỏ nhánh sẽ tự động cập nhật đến commit đó.
- Vì nhánh chỉ là một file nhỏ chứa mã băm SHA-1, nên việc tạo và xóa nhánh diễn ra rất nhanh chóng.

Phát triển Song song

- Nhánh tạo ra môi trường làm việc riêng biệt để phát triển tính năng mới, sửa lỗi hoặc thử nghiệm mà không ảnh hưởng đến mã nguồn chính.
- Giúp nhiều người có thể làm việc đồng thời trên cùng một codebase mà không xung đột.

Lịch sử Phân nhánh

- Lịch sử của Git có thể được hình dung như một cái cây với nhiều nhánh phát triển riêng biệt.
- Các nhánh này sau đó được **gộp (merge)** hoặc **tái cơ sở (rebase)** để tạo thành lịch sử thống nhất, giúp theo dõi quá trình phát triển.

Lưu ý: Git tự động tạo nhánh `master` hoặc `main` khi khởi tạo repository. Đây là nhánh chính chứa phiên bản ổn định. Khi phát triển tính năng mới hoặc sửa lỗi, bạn nên tạo nhánh mới từ nhánh chính.

Ví dụ minh họa

1. Bạn đang làm việc trên nhánh chính `main`.
2. Tạo nhánh mới để phát triển tính năng đăng nhập:

   ```
   git checkout -b feature/login
   ```

3. Làm việc và commit trên nhánh `feature/login` mà không ảnh hưởng đến nhánh `main`.
4. Khi hoàn tất, chuyển về nhánh `main` và gộp nhánh:

   ```
   git checkout main
   git merge feature/login
   ```

2 Tại sao cần sử dụng nhánh?

- Phát triển tính năng mới hoặc sửa lỗi độc lập.
- Tự do thử nghiệm mà không ảnh hưởng đến mã ổn định.
- Làm việc nhóm hiệu quả, tránh xung đột.

3 Ví dụ: Tạo và sử dụng nhánh

3.1 Tạo và chuyển nhánh

```
git checkout -b feature/login
# hoặc:
git branch feature/login
git checkout feature/login
```

3.2 Commit trên nhánh mới

```
git add .
git commit -m "Thêm tính năng đăng nhập"
```

3.3 Gộp nhánh

```
git checkout main
git merge feature/login
```

4 Giải quyết xung đột

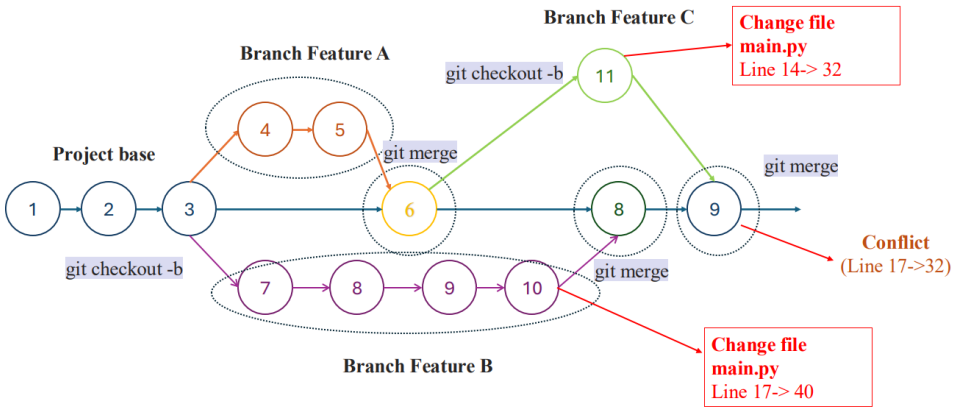
Khi xung đột xảy ra do cùng chỉnh sửa một dòng, Git đánh dấu như sau:

```
<<<<<<< HEAD
Nội dung từ nhánh main
=======
Nội dung từ feature/login
>>>>>>> feature/login
```

Cách xử lý:

1. Chỉnh sửa lại nội dung.
2. Xoá các ký hiệu xung đột.
3. Lưu file và chạy:

   ```
   git add <tên_file>
   git commit
   ```



Hình 3: Tạo, chuyển và gộp nhánh

5 Quản lý Nhánh và Nhánh Remote trong Git

5.1 Quản lý Nhánh

Liệt kê nhánh

Một số lệnh phổ biến để xem danh sách nhánh:

- `git branch` : Liệt kê các nhánh **local**, đánh dấu nhánh hiện tại bằng dấu *.
- `git branch -r` : Liệt kê các nhánh **remote-tracking**.
- `git branch -a` : Liệt kê **tất cả** nhánh (local và remote).
- `git branch -vv` : Hiển thị chi tiết trạng thái theo dõi remote của các nhánh local.

Xóa nhánh

- `git branch -d feature/login` : Xóa nhánh **đã merge** vào nhánh khác.
- `git branch -D feature/login` : Ép xóa nhánh **chưa merge** (cẩn trọng khi dùng).
- `git push origin --delete feature/login` : Xóa nhánh trên **remote repository**.

Đổi tên nhánh

- `git branch -m tên-cũ tên-mới` : Đổi tên một nhánh cụ thể.
- `git branch -m tên-mới` : Đổi tên **nhánh hiện tại**.

Sau khi đổi tên, cập nhật nhánh trên remote bằng lệnh:

```
git push origin :tên-cũ tên-mới
```

Tạo nhánh không có lịch sử chung

Đôi khi bạn muốn tạo một nhánh mới hoàn toàn **không có lịch sử commit chung** với nhánh khác:

```
git checkout --orphan tên-nhánh-mới
```

Điều này hữu ích khi muốn bắt đầu lại từ đầu nhưng vẫn giữ nguyên mã nguồn.

5.2 Nhánh Remote

Remote-tracking branches

- Là các tham chiếu tới trạng thái nhánh trên repository từ xa, có dạng: `[remote-name]/[branch-name]` (ví dụ: `origin/main`).
- Không thể chỉnh sửa trực tiếp, các nhánh này tự động cập nhật khi tương tác với remote.
- Lệnh `git fetch` sẽ cập nhật tất cả các remote-tracking branch theo trạng thái mới nhất của remote repository mà không thay đổi nhánh local.

Theo dõi nhánh từ xa

Để thiết lập mối quan hệ theo dõi giữa nhánh local và nhánh remote:

```
git branch --track branch-name origin/branch-name
```

Khi clone repository, Git tự động thiết lập nhánh `main` local theo dõi `origin/main`, cho phép dùng `git pull` và `git push` mà không cần chỉ định nhánh.

Để thay đổi nhánh remote mà nhánh local theo dõi:

```
git branch -u origin/dev-feature
```

Tạo nhánh local từ remote

- Tạo nhánh local mới từ nhánh remote:

  ```
  git fetch origin
  git checkout -b mybranch origin/feature-x
  ```

- Hoặc dùng cú pháp rút gọn:

  ```
  git checkout feature-x
  ```

  Nếu nhánh local chưa tồn tại nhưng có nhánh remote-tracking cùng tên, Git sẽ tự tạo nhánh local và thiết lập theo dõi.

- Đẩy nhánh mới lên remote lần đầu và thiết lập theo dõi:

  ```
  git push -u origin feature-branch
  ```

6 Các mô hình làm việc với nhánh

- GitFlow – rõ ràng, phù hợp phát hành định kỳ
- GitHub Flow – đơn giản, CI/CD nhanh
- GitLab Flow – thêm môi trường staging/production
- Trunk-Based – phát triển trực tiếp trên nhánh chính

7 Rebase và Các Workflow Mô Hình Nhánh Phổ Biến

7.1 Rebase

Rebase là tính năng mạnh mẽ của Git, tạo lịch sử commit gọn gàng và tuyến tính bằng cách "tái áp dụng" các commit từ nhánh hiện tại lên đầu nhánh đích, thay vì tạo commit merge.

7.2 Thực hiện Rebase cơ bản:

```
git checkout feature-branch
git rebase main
```

Lệnh này tạm gỡ bỏ các commit từ `feature-branch`, áp dụng các commit từ `main`, rồi áp dụng lại các commit của `feature-branch`. Kết quả là lịch sử commit tuyến tính như thể các thay đổi được thực hiện liên tiếp.

7.3 Rebase tương tác:

```
git rebase -i HEAD~3
```

Cho phép điều chỉnh, kết hợp, hoặc xóa commit trong quá trình rebase, giúp "dọn dẹp" lịch sử trước khi chia sẻ.

Lưu ý quan trọng: Không rebase các nhánh đã chia sẻ công khai vì sẽ thay đổi lịch sử commit, gây rắc rối cho người khác. Chỉ rebase nhánh local hoặc nhánh bạn làm việc một mình.

7.4 Các Workflow & Mô Hình Nhánh Phổ Biến

Tầm quan trọng của mô hình nhánh

Mô hình nhánh là quy ước sử dụng nhánh giúp đội phát triển làm việc nhất quán và hiệu quả. Việc lựa chọn mô hình phù hợp phụ thuộc vào quy mô dự án, số lượng thành viên và chu kỳ phát hành.

So sánh các mô hình

Mỗi mô hình có ưu điểm riêng:

- GitFlow phù hợp với chu kỳ phát hành dài,
- GitHub Flow đơn giản hơn cho triển khai liên tục,
- Trunk-based thúc đẩy tích hợp liên tục,
- GitLab Flow cân bằng giữa đơn giản và kiểm soát.

Áp dụng cho dự án

Khi chọn mô hình, cần cân nhắc văn hóa đội, quy trình CI/CD và yêu cầu kinh doanh. Có thể bắt đầu đơn giản với GitHub Flow và phát triển theo thời gian. Quan trọng là mọi thành viên hiểu và tuân thủ mô hình đã chọn.

7.5 GitFlow Branching Model

GitFlow được Vincent Driessen giới thiệu năm 2010, định nghĩa các loại nhánh và quy tắc tương tác. Mô hình này phù hợp với dự án có chu kỳ phát hành rõ ràng, nhưng có thể phức tạp cho dự án nhỏ hoặc triển khai liên tục.

- **Main branch:** Chứa mã nguồn sẵn sàng triển khai. Mỗi commit được gắn tag số phiên bản và đại diện cho bản phát hành chính thức.
- **Develop branch:** Tạo từ main, là nhánh tích hợp cho tất cả tính năng mới đã hoàn thành nhưng chưa sẵn sàng phát hành.
- **Feature branches:** Tách từ develop (`feature/*`), mỗi tính năng một nhánh. Sau khi hoàn thành, merge lại vào develop.
- **Release branches:** Tạo từ develop (`release/*`) khi chuẩn bị phát hành. Chỉ sửa lỗi, không thêm tính năng. Sau đó merge vào main và develop.
- **Hotfix branches:** Tạo từ main (`hotfix/*`) để sửa lỗi khẩn cấp. Sau khi sửa, merge vào main và develop.

7.6 Các Workflow Hiện Đại

Chọn workflow dựa trên yêu cầu dự án và văn hóa đội. Xu hướng hiện nay là sử dụng workflow đơn giản hỗ trợ tốt cho triển khai và tích hợp liên tục.

Trunk-based Development

Phương pháp phát triển đơn giản với công việc diễn ra trực tiếp trên nhánh `trunk` (main):

- Commit trực tiếp vào `main` hoặc qua nhánh tính năng ngắn hạn.
- Sử dụng feature flags để kiểm soát tính năng mới.
- Yêu cầu tích hợp liên tục (CI).
- Phù hợp với DevOps và triển khai nhiều lần mỗi ngày.

GitHub Flow

Quy trình đơn giản cho triển khai liên tục:

1. Tạo nhánh từ `main` cho mỗi tính năng hoặc lỗi.
2. Commit và push thường xuyên.
3. Mở Pull Request để review.
4. Merge sau khi pass kiểm tra và được approve.
5. Triển khai ngay sau khi merge.

Lý tưởng cho các dự án web và ứng dụng có chu kỳ phát hành liên tục.

GitLab Flow

Kết hợp GitHub Flow với quản lý môi trường:

- Sử dụng nhánh `production` thay vì chỉ dùng tags.
- Thêm các nhánh môi trường (staging, pre-production).
- Thay đổi di chuyển: main → môi trường kiểm thử → production.
- Hai biến thể: nhánh môi trường hoặc nhánh phát hành.

Giải quyết hạn chế của GitHub Flow với nhiều môi trường/phiên bản, đơn giản hơn GitFlow.

Phần IV: Thiết lập tài khoản và bảo mật trên Github

1 Đăng ký tài khoản

- Truy cập trang github.com để tạo tài khoản mới.
- Chọn tên người dùng (username) cẩn thận vì username này sẽ xuất hiện trong URL của các repository bạn tạo hoặc fork, ví dụ: `https://github.com/username/project`.
- Sau khi đăng ký, GitHub sẽ gửi email xác minh để kích hoạt tài khoản, bạn cần xác nhận email để có thể sử dụng đầy đủ các tính năng.

Ví dụ:
Bạn đăng ký username là `ha-nguyen`, khi tạo repository `my-project` thì URL sẽ là `https://github.com/ha-nguyen/my-project`

2 Thiết lập phương thức xác thực

- GitHub hỗ trợ xác thực bằng HTTPS và SSH.
- **HTTPS:** Dễ dùng nhưng mỗi lần push/pull sẽ phải nhập mật khẩu (hoặc token).
- **SSH:** Bảo mật hơn và bạn không cần nhập mật khẩu liên tục.

Cách tạo SSH key:
Mở terminal (command line) và gõ lệnh:

```
ssh-keygen -t ed25519 -C "email@example.com"
```

- Lệnh này tạo cặp khóa SSH (private và public).
- Sau đó, bạn mở file public key (ví dụ: `/.ssh/id_ed25519.pub`) và copy toàn bộ nội dung.
- Vào GitHub > Settings > SSH and GPG keys > New SSH key, dán key vào và lưu lại.

Ví dụ:
Bạn tạo SSH key, rồi dùng lệnh:

```
git clone git@github.com:ha-nguyen/my-project.git
```

thay vì clone HTTPS, và bạn không cần nhập mật khẩu mỗi lần thao tác.

3 Tùy chỉnh hồ sơ cá nhân

- Cập nhật ảnh đại diện, email, tên hiển thị để người khác dễ nhận diện bạn.
- Vào GitHub > Your profile > Edit profile.

4 Bảo mật tài khoản

- Bật **Xác thực hai yếu tố (2FA)** để tăng cường bảo mật. Có thể dùng app Authenticator (Google Authenticator, Authy), SMS, hoặc khóa bảo mật vật lý (YubiKey).
- Quản lý quyền truy cập các ứng dụng bên thứ ba trong phần `Settings > Applications`.

5 Quy trình Fork & Pull Request

Đây là quy trình chuẩn khi bạn muốn đóng góp mã nguồn vào một dự án mà bạn không phải là thành viên trực tiếp.

1. **Fork repository**
   - Fork là tạo bản sao repository gốc về tài khoản của bạn.
   - Vào repository gốc trên GitHub, nhấn nút **Fork** ở góc trên bên phải.
   - Bạn có quyền chỉnh sửa và push vào repository fork mà không ảnh hưởng đến repository gốc.
2. **Clone về máy local**

   Clone repository fork về máy để làm việc:

   ```
   git clone https://github.com/username-cua-ban/ten-repository.git
   cd ten-repository
   ```

   Thiết lập remote `upstream` để liên kết với repository gốc, giúp bạn cập nhật thay đổi mới nhất:

   ```
   git remote add upstream https://github.com/owner-goc/ten-repository.git
   ```

3. **Tạo nhánh chủ đề (feature branch)**

   Tạo nhánh mới từ `main` hoặc `master` để làm việc riêng cho tính năng hoặc sửa lỗi:

   ```
   git checkout -b ten-nhanh-moi
   ```

   Ví dụ:

   ```
   git checkout -b fix-login-bug
   ```

4. **Thực hiện thay đổi và push**

   Sau khi chỉnh sửa code, commit các thay đổi:

   ```
   git add .
   git commit -m "Sua loi dang nhap khong kiem tra mat khau dung"
   git push origin fix-login-bug
   ```

5. **Tạo Pull Request (PR)**
   - Vào repository gốc trên GitHub, nhấn nút **New pull request**.
   - Chọn nhánh bạn vừa push làm nguồn, và nhánh chính (main/master) của repo gốc làm đích.
   - Viết tiêu đề và mô tả chi tiết thay đổi.

- Nhấn **Create pull request** để gửi yêu cầu hợp nhất mã nguồn.

6. **Thảo luận và cập nhật**

- Chủ sở hữu repo gốc sẽ xem xét PR, có thể yêu cầu bạn chỉnh sửa.
- Bạn cập nhật lại trên nhánh đã push, PR sẽ tự động cập nhật theo.
- Khi được chấp thuận, PR sẽ được merge vào repo gốc.

6 Quản lý repository trên GitHub

6.1 Tạo và cấu hình repository

- Click nút `"+"` trên thanh menu, chọn **New repository**.
- Đặt tên, mô tả, chọn quyền riêng tư (public/private).
- Có thể khởi tạo `README.md`, `.gitignore`, `LICENSE` ngay khi tạo repo.
- Thêm collaborator (người cùng làm) trong phần `Settings > Manage access`.

6.2 Tài liệu và hướng dẫn

- Tạo file `README.md` mô tả dự án.
- Tạo `CONTRIBUTING.md` hướng dẫn cách đóng góp.
- Tạo `CODE_OF_CONDUCT.md` quy định ứng xử trong dự án.

6.3 Mẫu Issue và Pull Request

- Đặt mẫu trong thư mục `.github/` như `ISSUE_TEMPLATE.md` hoặc `PULL_REQUEST_TEMPLATE.md`.
- Mẫu giúp chuẩn hóa thông tin khi tạo issue hoặc PR.

6.4 Thiết lập branch và bảo vệ

- Trong `Settings > Branches`, cấu hình nhánh mặc định.
- Kích hoạt tính năng **Branch protection rules** để yêu cầu review, chạy CI trước khi merge, ngăn push trực tiếp lên nhánh chính.

7 Organization & Teams

7.1 Tạo và quản lý Organization

- Organization quản lý nhiều repo và thành viên cùng lúc, phù hợp công ty hoặc dự án lớn.
- Tạo Organization qua nút `"+"` > **New organization**.
- Chọn gói dịch vụ (Teams hoặc Enterprise).
- Nhập tên, email và người quản lý chính.

7.2 Chuyển repository cá nhân vào Organization

- Vào repository > `Settings > Options > Transfer ownership`.

7.3 Quản lý Teams và phân quyền

- Trong Organization, vào `Teams`, tạo team mới.
- Đặt tên, mô tả, chọn quyền riêng tư.
- Thêm thành viên và cấp quyền truy cập repo:
  - **Read:** xem và clone
  - **Write:** push code
  - **Admin:** quản lý repo
- Teams có thể lồng nhau, thành viên thừa hưởng quyền từ team cha.

7.4 Nhật ký hoạt động và giám sát (Audit log)

- Theo dõi hoạt động người dùng, thay đổi quyền, truy cập repo.
- Phục vụ mục đích bảo mật và tuân thủ.
- Có thể xuất báo cáo để phân tích.

7.5 Security alerts và Dependabot

- GitHub tự động cảnh báo lỗ hổng bảo mật trong dependencies.
- Dependabot giúp cập nhật thư viện tự động.

8 Mẹo làm việc hiệu quả trên GitHub

1. **Sử dụng Draft Pull Request**
   - Tạo pull request dạng draft để xin góp ý sớm mà không làm reviewer phải xem xét hoàn chỉnh.
   - Khi đã sẵn sàng, chuyển draft thành pull request bình thường.
2. **Tạo review checklist**
   - Checklist giúp reviewer kiểm tra đầy đủ các tiêu chuẩn (coding style, test coverage, bảo mật...).
   - Thêm checklist vào mẫu Pull Request để mọi người tự kiểm tra trước khi gửi review.
3. **Đồng bộ hóa fork thường xuyên**

   Giữ fork cập nhật với repository gốc để tránh xung đột:

```
git remote add upstream https://github.com/original/repo.git
git fetch upstream
git checkout main
git merge upstream/main
git push origin main
```

9 GitHub Desktop

9.1 Giới thiệu

- Ứng dụng GUI giúp thao tác Git/GitHub dễ dàng cho người không quen dùng dòng lệnh.
- Hỗ trợ quản lý nhánh, commit, push/pull, xử lý xung đột.

9.2 Tính năng chính

- Giao diện trực quan, theo dõi thay đổi file bằng hình ảnh.
- Quản lý repository, nhánh đơn giản.
- Tích hợp trực tiếp với GitHub.com.

9.3 Cách sử dụng cơ bản

1. Tải và cài đặt từ desktop.github.com
2. Đăng nhập với tài khoản GitHub.
3. Clone repository hoặc tạo mới.
4. Thực hiện thay đổi, commit.
5. Push thay đổi lên GitHub.

Phần V: Tùy chỉnh và Mở rộng Git

1 Cấu hình nâng cao (.gitconfig)

1.1 Thiết lập nhánh mặc định

Từ Git phiên bản 2.28 trở đi, bạn có thể thay đổi tên nhánh mặc định khi khởi tạo repository. Mặc định nhánh này thường là `master`, bạn có thể đổi sang `main` để phù hợp với xu hướng hiện nay:

```
git config --global init.defaultBranch main
```

Giải thích: Lệnh trên sẽ thiết lập nhánh mặc định thành `main` khi bạn chạy lệnh `git init`.
Ví dụ:

- Trước khi cấu hình, lệnh `git init` sẽ tạo ra nhánh `master`.
- Sau khi chạy lệnh trên, `git init` sẽ tạo nhánh `main`.

1.2 Mẫu commit message

Để đảm bảo tính nhất quán cho các *commit message*, bạn có thể thiết lập mẫu *template* cho commit:

```
git config --global commit.template ~/.gitmessage.txt
```

Cấu trúc mẫu trong file `.gitmessage.txt`:

```
# Tiêu đề: Tóm tắt ngắn gọn (< 50 ký tự)
# Nội dung: Giải thích thay đổi và lý do
# (Tối đa 72 ký tự/dòng)
# Issue liên quan: #123
```

Giải thích: Khi commit, Git sẽ mở mẫu này để bạn điền nội dung, giúp đảm bảo mọi người viết commit theo chuẩn.
Ví dụ:

```
Add user login validation

Thêm các kiểm tra dữ liệu đầu vào để đảm bảo username và password
không được để trống, đồng thời hiển thị lỗi rõ ràng cho người dùng.

Issue liên quan: #45
```

1.3 Công cụ xử lý merge và diff

Bạn có thể sử dụng Visual Studio Code làm công cụ so sánh và hợp nhất:

```
git config --global merge.tool vscode
git config --global diff.tool vscode
git config --global difftool.vscode.cmd "code --wait --diff $LOCAL $REMOTE"
git config --global mergetool.vscode.cmd "code --wait $MERGED"
```

Giải thích: Khi xảy ra xung đột, Git sẽ gọi VSCode mở để bạn so sánh và chỉnh sửa file dễ dàng hơn.
Ví dụ: Khi bạn chạy `git mergetool`, VSCode sẽ tự động mở file có xung đột.

1.4 Thiết lập proxy và xác thực

Nếu bạn làm việc trong môi trường mạng doanh nghiệp cần proxy, thiết lập proxy cho Git như sau:

```
git config --global http.proxy http://proxy.example.com:8080
git config --global https.proxy https://proxy.example.com:8080
```

Lưu thông tin đăng nhập để không phải nhập lại nhiều lần:

```
git config --global credential.helper store    # ưLu ĩvnh ễvin
git config --global credential.helper cache    # ưLu ạtm trong ộb ởnh
git config --global credential.helper 'cache --timeout=3600' # ưLu theo giây
```

Giải thích:

- `store`: lưu thông tin đăng nhập vĩnh viễn trên đĩa.
- `cache`: lưu trong bộ nhớ RAM trong khoảng thời gian mặc định (15 phút).
- `cache --timeout=3600`: lưu trong RAM trong 3600 giây (1 giờ).

Ví dụ: Sau khi cấu hình, lần đầu bạn nhập mật khẩu khi push, những lần sau sẽ tự động sử dụng lại.

2 Git Attributes (.gitattributes)

2.1 Khái niệm Git Attributes

File `.gitattributes` quy định cách Git xử lý các loại file cụ thể. Cấu trúc đơn giản:

pattern attr1 attr2 ...

pattern là mẫu tên file (ví dụ `*.txt`, `docs/*.md`) và **attr** là thuộc tính áp dụng cho file.
Ví dụ:

```
# Chuẩn hóa line endings
* text=auto

# File binary
*.png binary
*.jpg binary

# Xác định merge
database.xml merge=ours
```

2.2 Xử lý kết thúc dòng (Line Endings)

Giúp quản lý sự khác biệt line endings giữa Windows (CRLF) và Unix/Mac (LF), tránh thay đổi không cần thiết:

```
# Chuẩn hóa thành LF trong repository
*.txt text
*.java text
*.md text

# Chuyển sang CRLF khi checkout trên Windows
```

```
*.bat text eol=crlf

# Giữ nguyên line ending
*.sh text eol=lf
```

Giải thích: Khi bạn làm việc trên Windows, Git tự động chuyển đổi để tránh xung đột với Linux/Mac.

2.3 Đánh dấu file binary và chiến lược merge

Thuộc tính `binary` giúp Git không xử lý nội dung file:

```
*.png binary
*.jpg binary
*.zip binary
```

Quy định cách xử lý khi merge file:

```
# Giữ file cấu hình local khi merge
config.xml merge=ours

# Sử dụng phiên bản từ nhánh đang merge
generated-code.js merge=theirs
```

Giải thích:

- `merge=ours`: Giữ nguyên file hiện tại, bỏ thay đổi từ nhánh khác.
- `merge=theirs`: Chấp nhận thay đổi từ nhánh khác, bỏ thay đổi hiện tại.

3 Clean và Smudge Filters

Biến đổi file khi thêm vào hoặc lấy ra từ repository:

```
# Áp dụng filter "indent" cho file C
*.c filter=indent

# Mở rộng từ khóa cho file nguồn Java
*.java filter=keyword
```

Cấu hình filter trong `.gitconfig`:

```
[filter "indent"]
clean = indent
smudge = cat

[filter "keyword"]
clean = sed "s/\\$Date[^\\$]*\\$/\\$Date\\$/"
smudge = sed "s/\\$Date\\$/\\$Date: $(date)\\$/"
```

Giải thích:

- `clean`: chạy khi thêm file vào repo, thường dùng để chuẩn hóa.
- `smudge`: chạy khi lấy file ra, có thể thêm thông tin động như ngày giờ.

Ví dụ: Filter `keyword` sẽ tự động cập nhật trường ngày tháng mỗi khi lấy file ra làm việc.

4 Git Hooks – Tự động hóa

4.1 Giới thiệu về Git Hooks

Git Hooks là các script tự động chạy ở các thời điểm cụ thể trong quy trình Git. Các hook nằm trong thư mục `.git/hooks/`.

Để sử dụng, bạn:

- Bỏ đuôi `.sample` khỏi file hook mẫu.
- Cấp quyền thực thi: `chmod +x hook-name`.

4.2 Client-side Hooks

Chạy trên máy của người phát triển:

- `pre-commit`: kiểm tra code, chạy test trước commit.
- `prepare-commit-msg`: chỉnh sửa nội dung commit trước khi soạn.
- `commit-msg`: kiểm tra định dạng commit message.
- `post-commit`: chạy sau commit, ví dụ gửi thông báo.
- `pre-push`: chạy trước khi push code lên remote.

4.3 Server-side Hooks

Chạy trên máy chủ Git:

- `pre-receive`: chạy khi server nhận yêu cầu push.
- `update`: chạy riêng cho từng ref thay đổi.
- `post-receive`: chạy sau khi push hoàn thành, thường kích hoạt CI/CD.

Lưu ý: Server-side hooks giúp kiểm soát chặt chẽ quy trình vì người dùng không thể bỏ qua.

5 Alias & Scripts

5.1 Tạo Git Alias cơ bản

Alias giúp tạo lệnh tắt cho các lệnh dài, tiết kiệm thời gian:

```
[alias]
st = status
co = checkout
br = branch -v
cm = commit -m
l = log --oneline --graph --decorate
unstage = restore --staged
last = log -1 HEAD
visual = !gitk
```

Ví dụ: Gõ `git st` thay vì `git status`.

5.2 Alias nâng cao và hàm

Bạn có thể tạo alias phức tạp, kết hợp nhiều lệnh:

```
[alias]
 # Tạo nhánh mới và chuyển sang nhánh đó
 nb = "!f() { git checkout -b $1; }; f"

 # Xóa tất cả nhánh đã merge vào nhánh hiện tại
 cleanup = "!git branch --merged | grep -v '\\*' | xargs -n 1 git branch -d"

 # Cập nhật nhánh hiện tại từ upstream
 sync = "!f() { git pull upstream $(git rev-parse --abbrev-ref HEAD); }; f"

 # Xem lịch sử với giao diện đẹp
 graph = "log --graph --pretty=format:'%Cred%h%Creset - %C(yellow)%d%Creset %s %Cgreen(%cr) %C
```