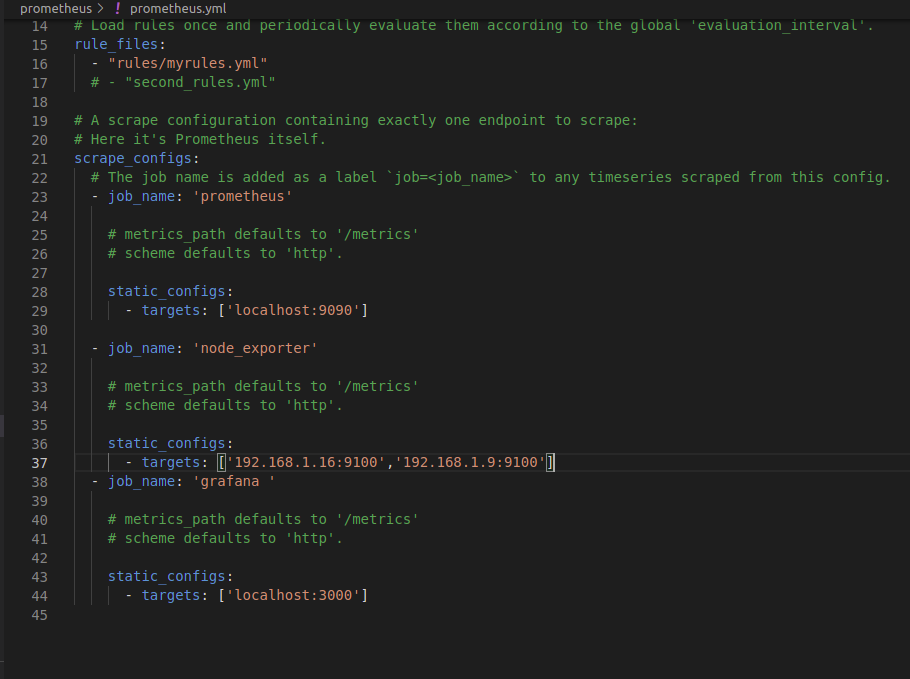
**ĐỒ ÁN THỰC TẬP**

**Thực hành cài đặt Prometheus để giám sát Server và Container**

**1. Cài đặt Prometheus và các Exporter cần thiết cho VM**

Đầu tiên chúng ta sẽ cài đặt prometheus và cài đặt node\_exporter cho VM 2 chương trình này chúng ta có thể tải tại [Prometheus](https://prometheus.io/download/)

Trước khi chạy prometheus và node-exporter thì chúng ta cần chỉnh sửa lại file cấu hình của prometheus là prometheus.yml để có thể liên kết với node-exporter.

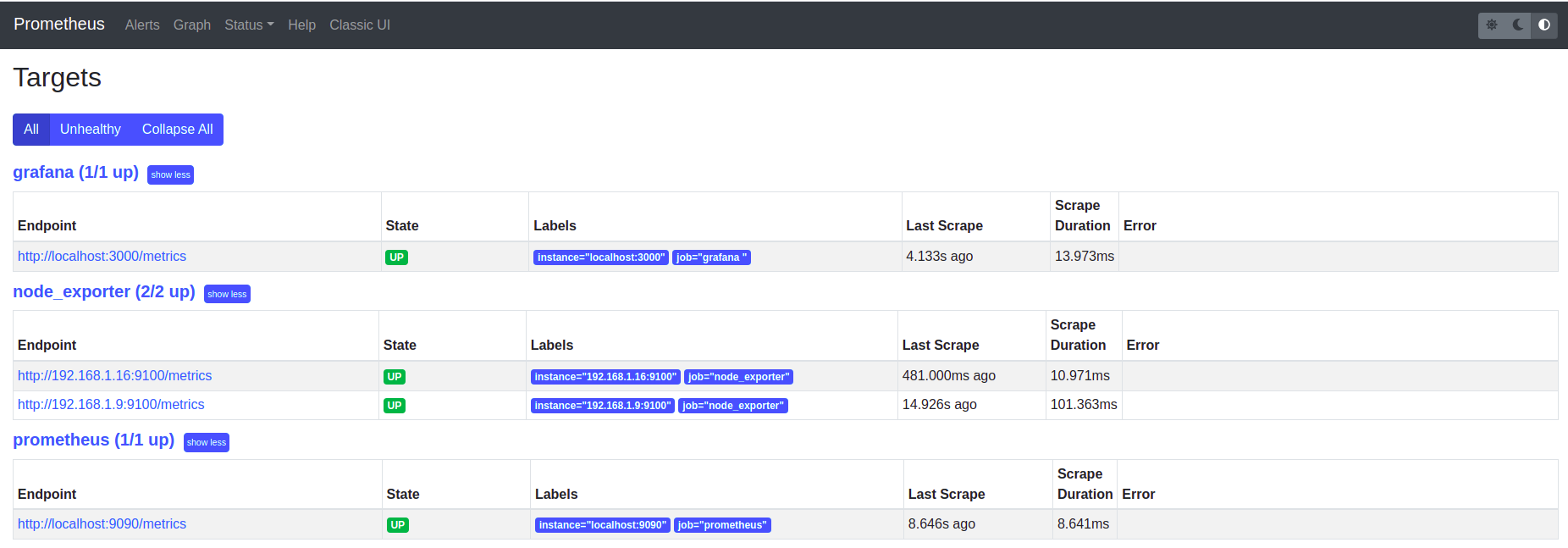
[](https://github.com/vuducthanh0115/Documents/blob/main/Image/12.png)

Tùy vào số lượng VM chúng ta đã tạo mà chúng ta cài đặt node-exporter thì ta sẽ config ip của server đó vào file cấu hình của prometheus.

Để chạy được prometheus và node-exporter thì chúng và truy cập vào nơi chứa file và chạy các file cài đặt.

Sau khi cài đặt chúng ta sẽ truy cập địa chỉ ip mà chúng ta đã khai báo trong file cấu hình của prometheus. Mặc định prometheus sẽ chạy ở port 9090 và node-exporter sẽ chạy ở port 9100.

**Prometheus**

[](https://github.com/vuducthanh0115/Documents/blob/main/Image/13.png)

**Node\_exporter**

[](https://github.com/vuducthanh0115/Documents/blob/main/Image/14.png)

**2. Cài đặt Grafana**

***Cài đặt theo hình dưới đây:***

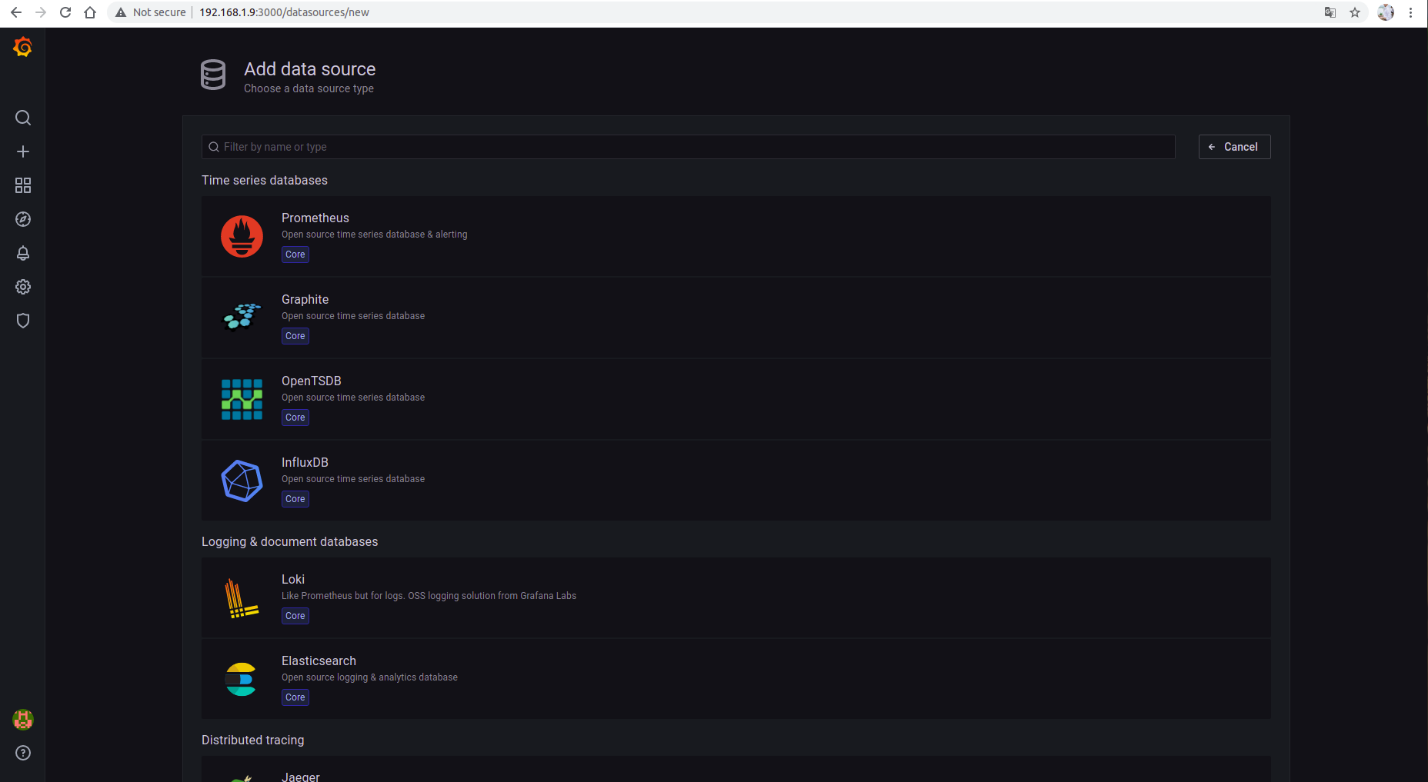
[](https://github.com/vuducthanh0115/Documents/blob/main/Image/15.png)

Grafana mặc định sẽ chạy ở port 3000. Khi lần đầu chúng ta chạy grafana thì username và password mặc định sẽ là admin.

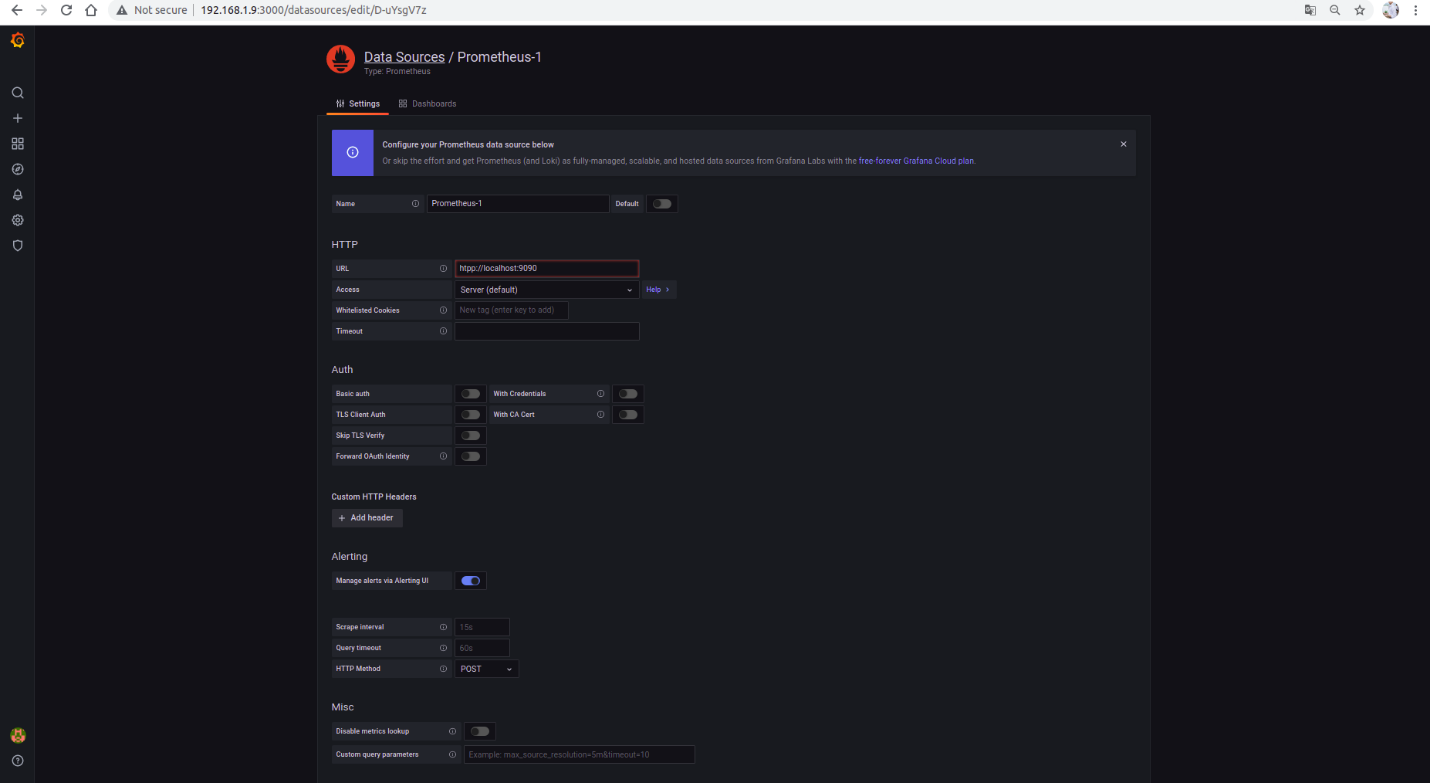
[](https://github.com/vuducthanh0115/Documents/blob/main/Image/16.png)

***Setup Grafana***

Sau khi cài đặt thành công grafana thì chúng ta cần tạo 1 data source cho prometheus trong grafana để có thể kết nối và lấy dữ liệu.

[](https://github.com/vuducthanh0115/Documents/blob/main/Image/17.png)

Grafana hỗ trợ rất nhiều data source, ở đây chúng ta sử dụng prometheus nên chọn prometheus.

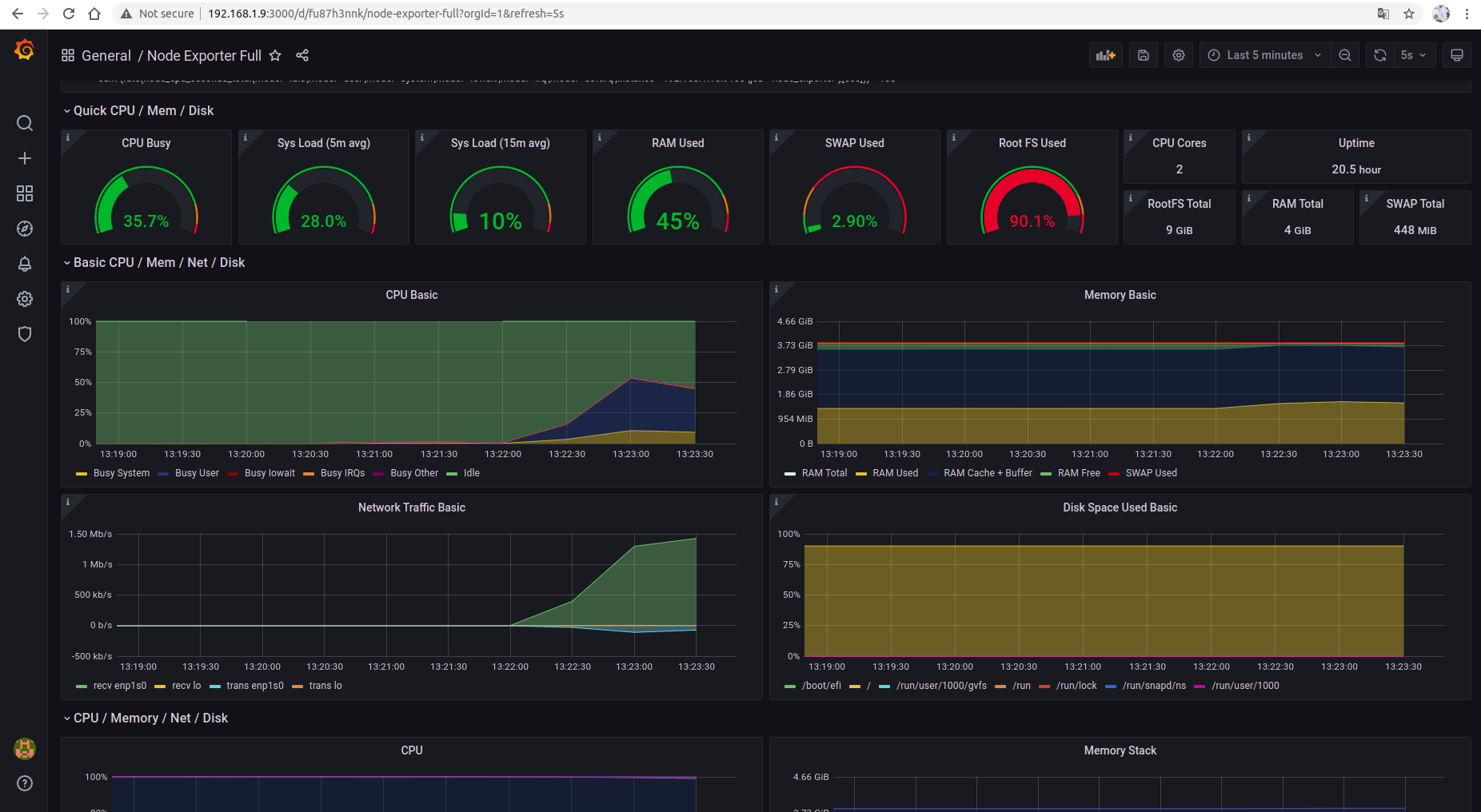
[](https://github.com/vuducthanh0115/Documents/blob/main/Image/18.png)

Sau khi chọn data source thì chúng sẽ điền thông tin URL của grafana vào và chọn save & test để kiểm tra data source của chúng ta có hoạt động hay không.

Khi data source hoạt động thì chúng ta chỉ cần cài đặt dashboard để hiển các thông số ra nữa thôi. Grafana có hỗ trợ chúng ta rất nhiều dashboard có cấu hình sẵn. Ta chỉ tìm cái nào mà chúng ta thích và grafana nhập mã của dashboard đó vào rồi ấn load để load cái dashboard lên là được.

Sau khi chúng load dashboard lên thì chúng ta cần chọn data source cho dashboard đấy.

***Và đây là kết quả chúng ta có được:***

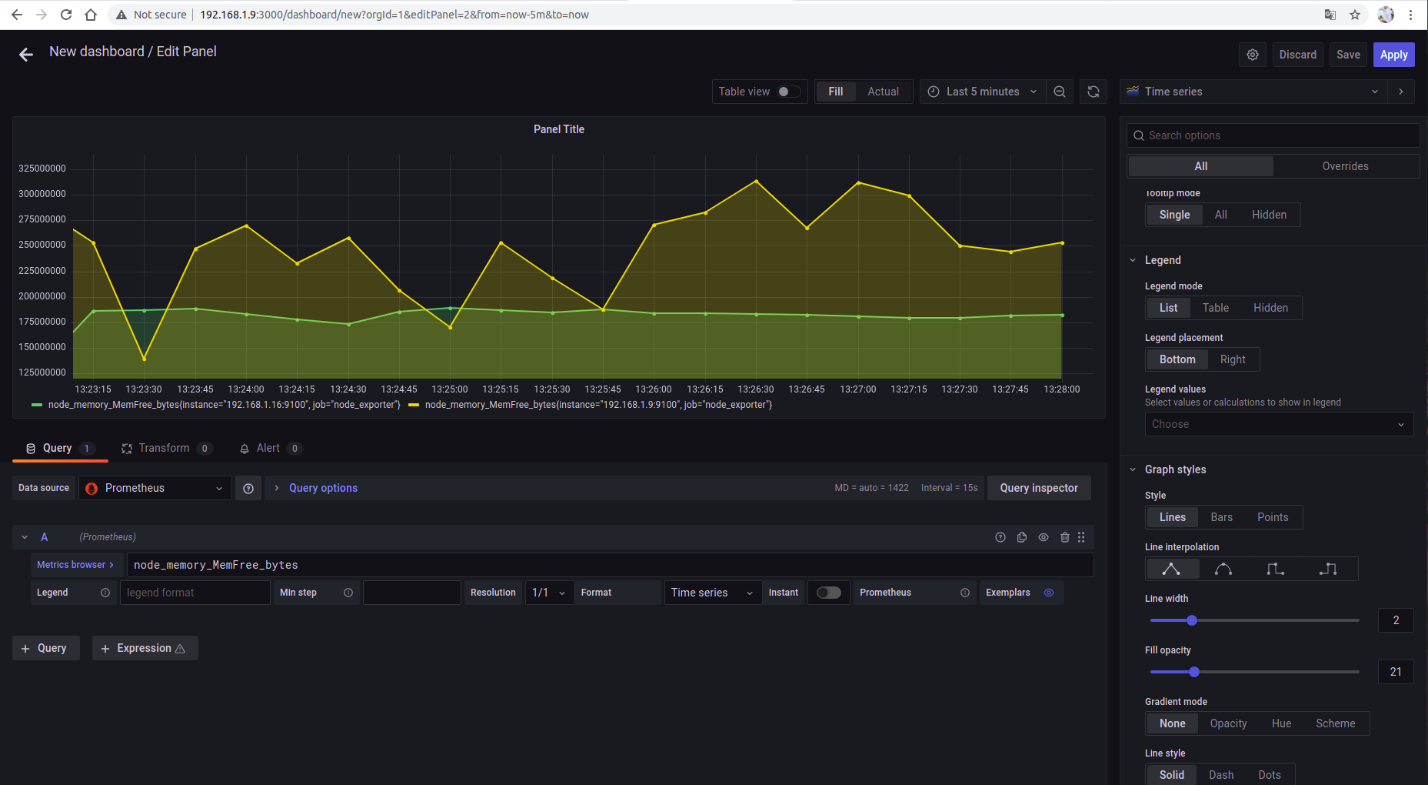
[](https://github.com/vuducthanh0115/Documents/blob/main/Image/19.png)

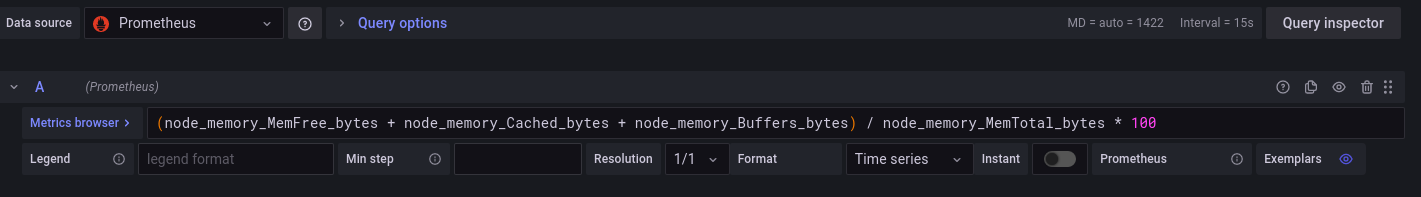
Ngoài cách dùng dashboard có sẵn chúng ta cũng có thể tự tay mình thiết kế những khung hiển thị tùy ý.

Để tạo được dashboard thì chúng ta chọn create và chọn ***Add new panel*** để có thể tùy chỉnh.

[](https://github.com/vuducthanh0115/Documents/blob/main/Image/20.png)

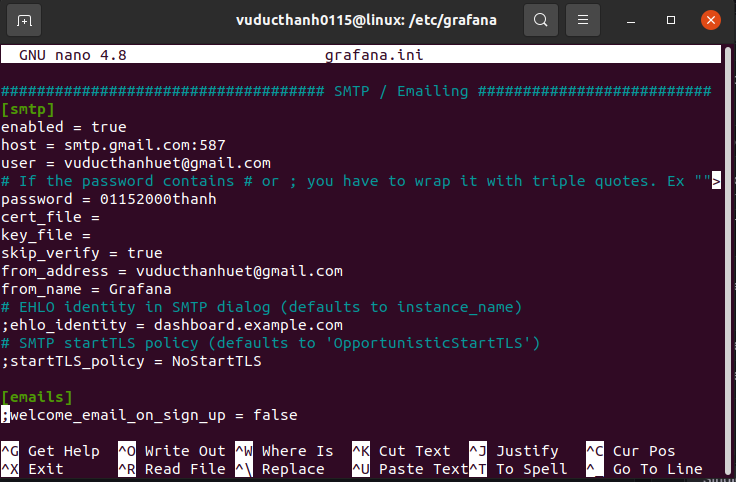
Vào giao diện setting thì chúng ta sẽ dùng các đoạn metric của node-exporter để hiển thị ra thông tin, và chúng ta có thể thay đổi màu sắc cũng như các biểu đồ hiển thị một cách rất đơn giản

[](https://github.com/vuducthanh0115/Documents/blob/main/Image/21.png)

[](https://github.com/vuducthanh0115/Documents/blob/main/Image/22.png)

**Cấu hình Alert (cảnh báo)**

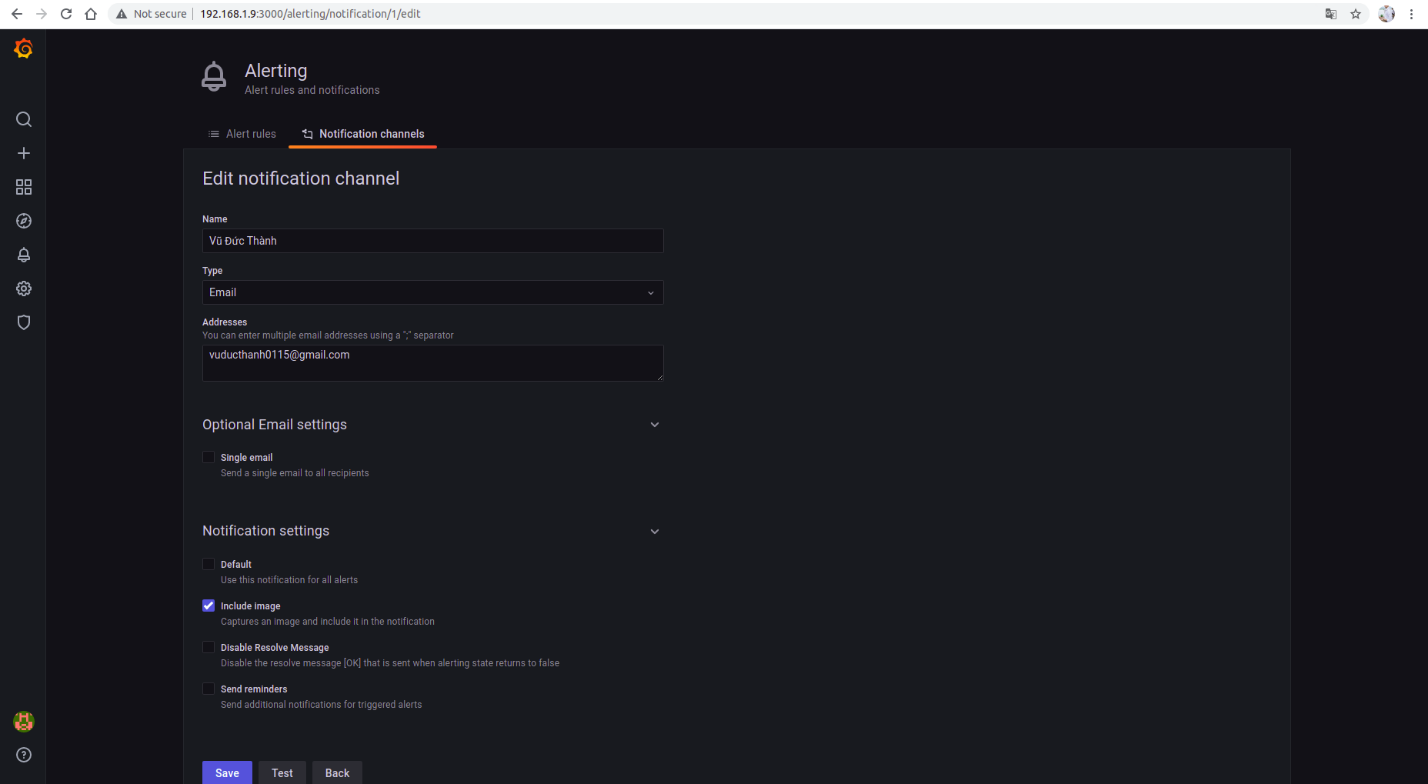
Tiếp theo chúng ta cần cấu hình cảnh báo cho grafana, có 1 điều cần chú ý là: khi sử dụng dashboard được cấu hình sẵn thì chúng ta sẽ không cấu hình alert được,lý do là khi gọi các metric có sử dụng biến để có thể phân biệt được thời gian cũng như của server nào, vì thế nếu chúng ta có nhiều server thì chúng ta phải tạo ra nhiều khung hiển thị để hiển thị từng thông tin tài nguyên của server đó và bỏ đi phần biến lúc gọi metric để có thể cấu hình alert.

[](https://github.com/vuducthanh0115/Documents/blob/main/Image/23.png)Chúng ta cần phải cấu hình lại grafana.ini để có thể gửi cảnh báo về mail của chúng ta, đường dẫn để truy cập file grafana.ini là: /etc/grafana/grafana.ini

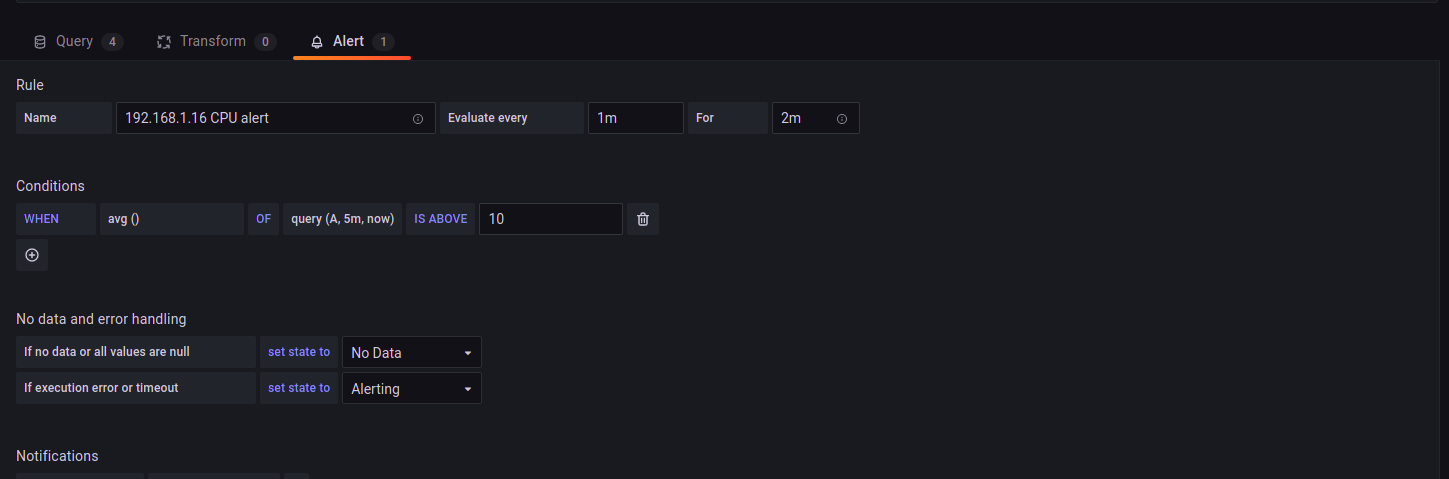
Ta cần thêm 1 số thông tin ở phần smtp, cần chú ý là gmail mà chúng ta nhập vào ở đây, grafana sẽ dùng gmail đó để viết và gửi cảnh báo cho chúng ta.

Sau khi chỉnh sửa thì chạy lệnh: *$ sudo service grafana-server restart để restart lại grafana*

Trước khi thiết lập cảnh báo thì chúng ta cần tạo 1 kênh thông báo với 1 email cụ thể để có thể gửi cảnh báo đến email đó. Ta ấn test để kiểm tra thử email có được gửi thành công hay không.

[](https://github.com/vuducthanh0115/Documents/blob/main/Image/24.png)

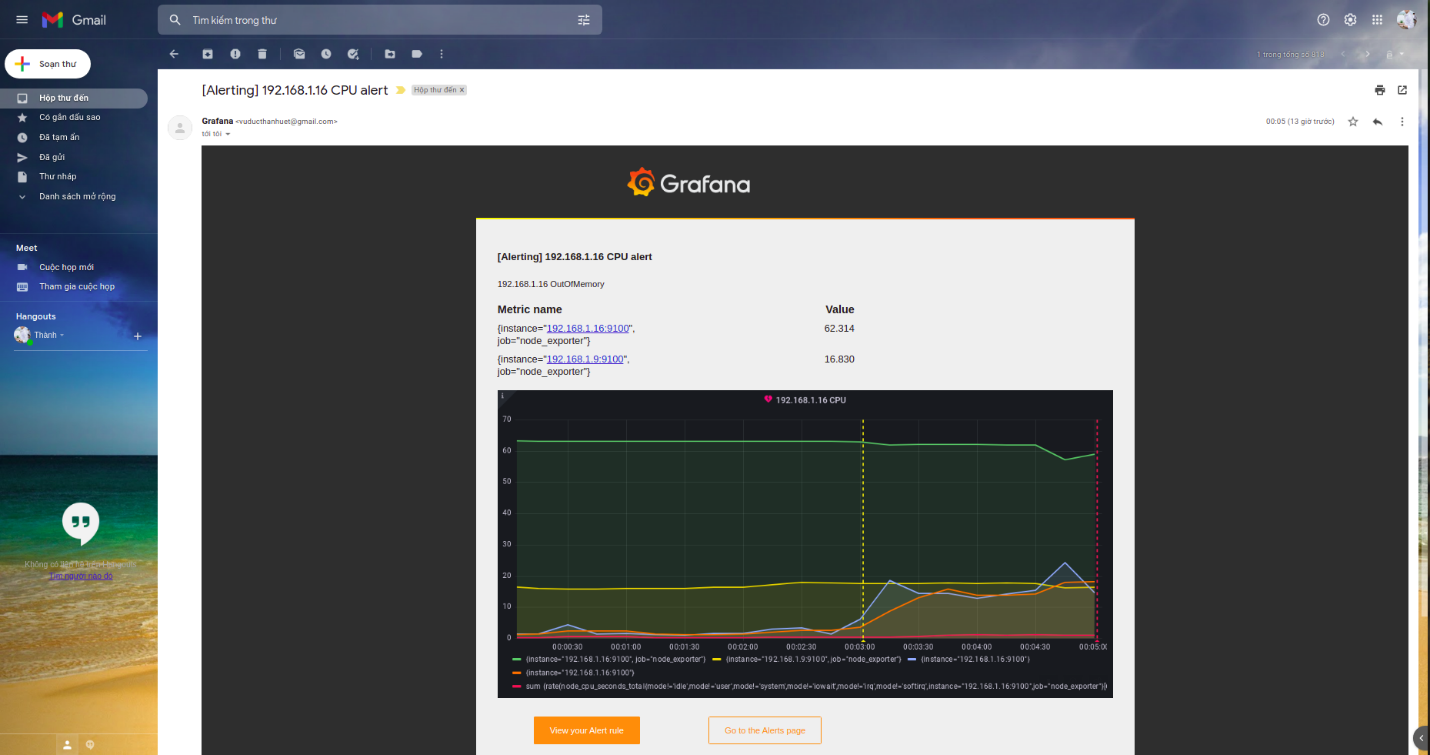
Để cấu hình cảnh báo thì chúng chọn ***Alert*** và chọn ***Create Alert*** để bắt.

[](https://github.com/vuducthanh0115/Documents/blob/main/Image/25.png)

Tại đây chúng ta điền vào các thông số mà chúng ta quy định, ví dụ như trong ảnh, hệ thống sẽ cảnh báo khi CPU vượt ngưỡng 1, sau đó chúng ta chọn test rule để kiểm tra alert chúng ta cài đặt có hoạt động không, sao đó chọn save để có thể lưu lại

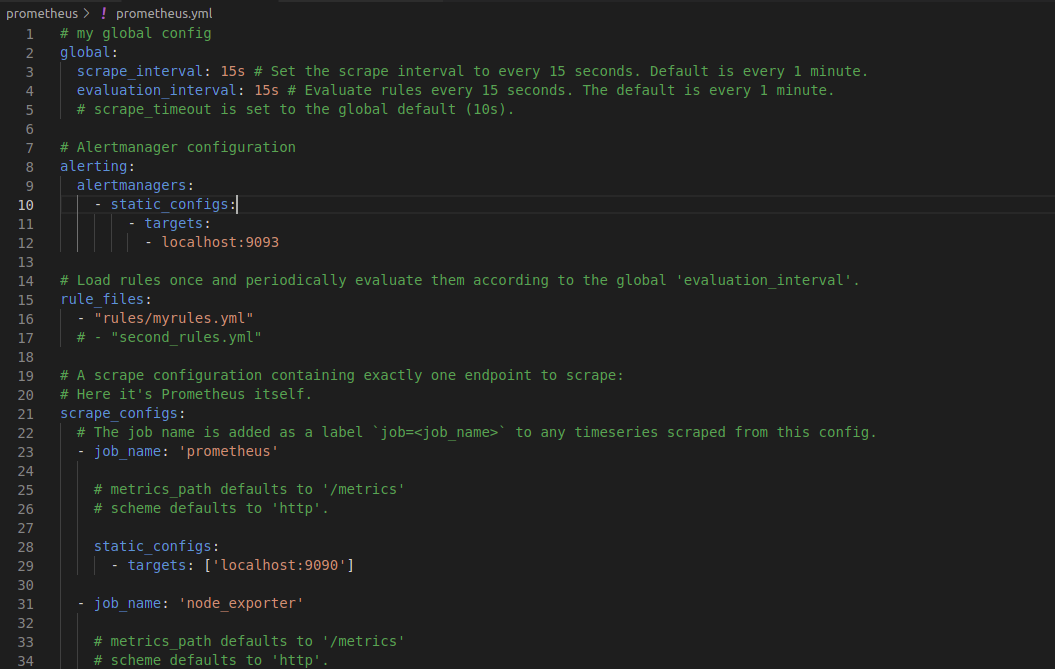
Khi chúng ta thiết lập cảnh báo thành công là phía trên mỗi khung dữ liệu điều có hình trái tim, màu xanh là an toàn các tài nguyên thấp hơn mức quy định, màu vàng là khi tài nguyên vượt mức quy định, nhưng chưa đủ thời gian gây ra nguy hiểm mà chúng ta thiết lập, màu đỏ là vượt qua thời gian an toàn thì hệ thống sẽ gửi cảnh báo qua email của chúng ta kèm theo hình ảnh khi vượt ngưỡng an toàn.

***Và đây là email chúng ta nhận được :***

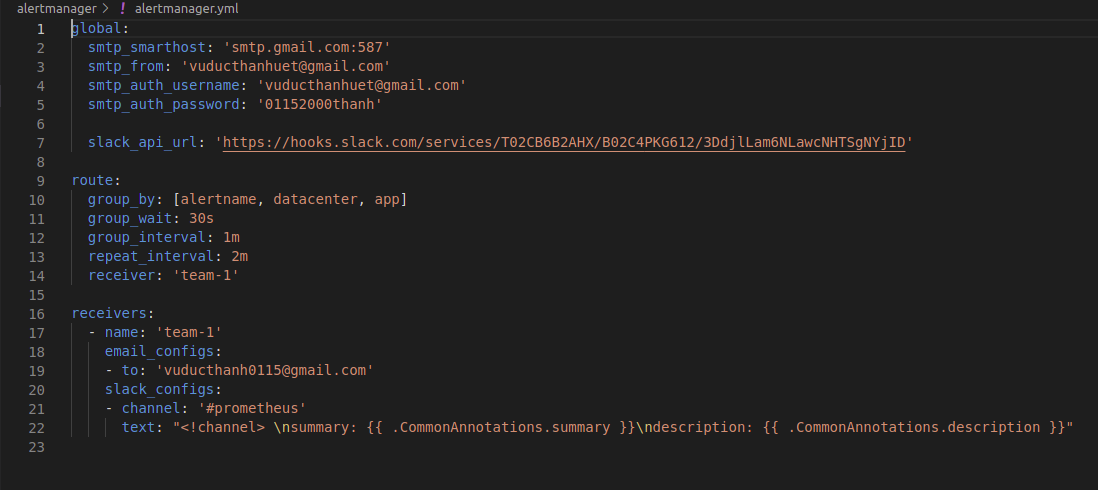
[](https://github.com/vuducthanh0115/Documents/blob/main/Image/26.png)

**3. Ta cũng có thể dùng Alertmanager để bắn cảnh báo về mail, webhook, telegram, ....**

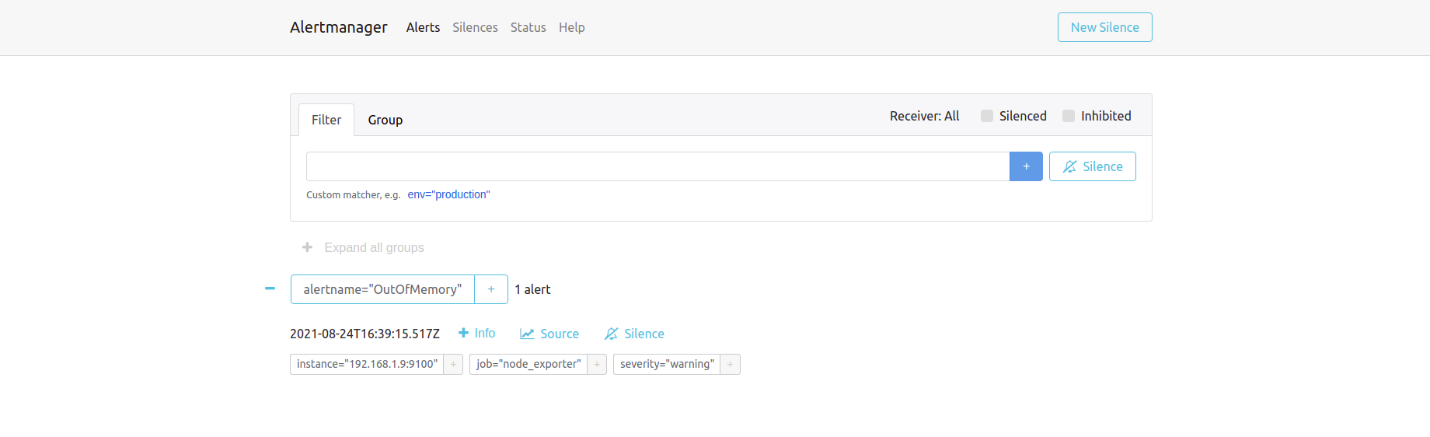
Đầu tiên chúng ta sẽ cài đặt alertmanager chương trình này chúng ta có thể tải tại [Prometheus](https://prometheus.io/download/)  
Trước khi chạy alertmanager thì chúng ta cần chỉnh sửa lại file cấu hình của prometheus là prometheus.yml để có thể liên kết với alertmanager.

[](https://github.com/vuducthanh0115/Documents/blob/main/Image/27.png)

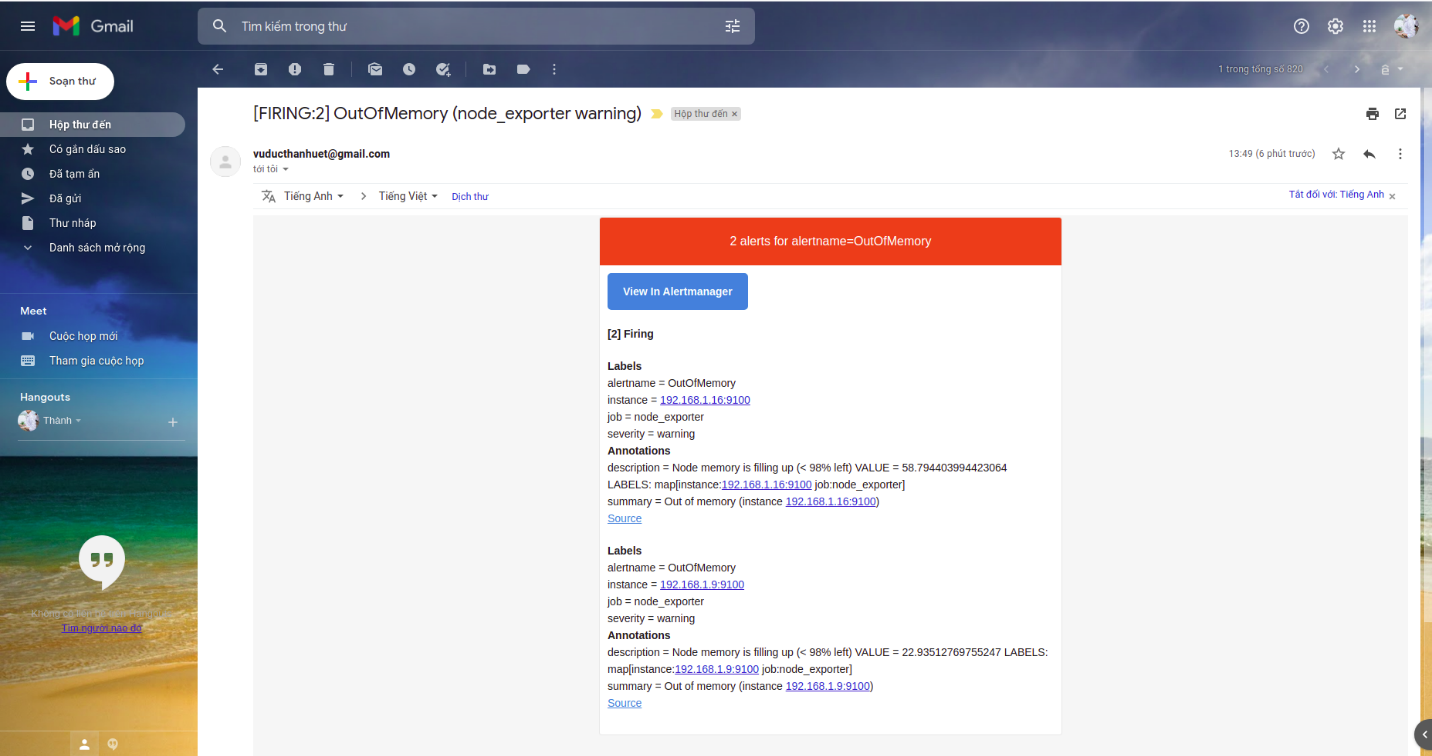
*Sau đó trước khi chạy Alertmanager thì ta cần cấu hình như sau :*

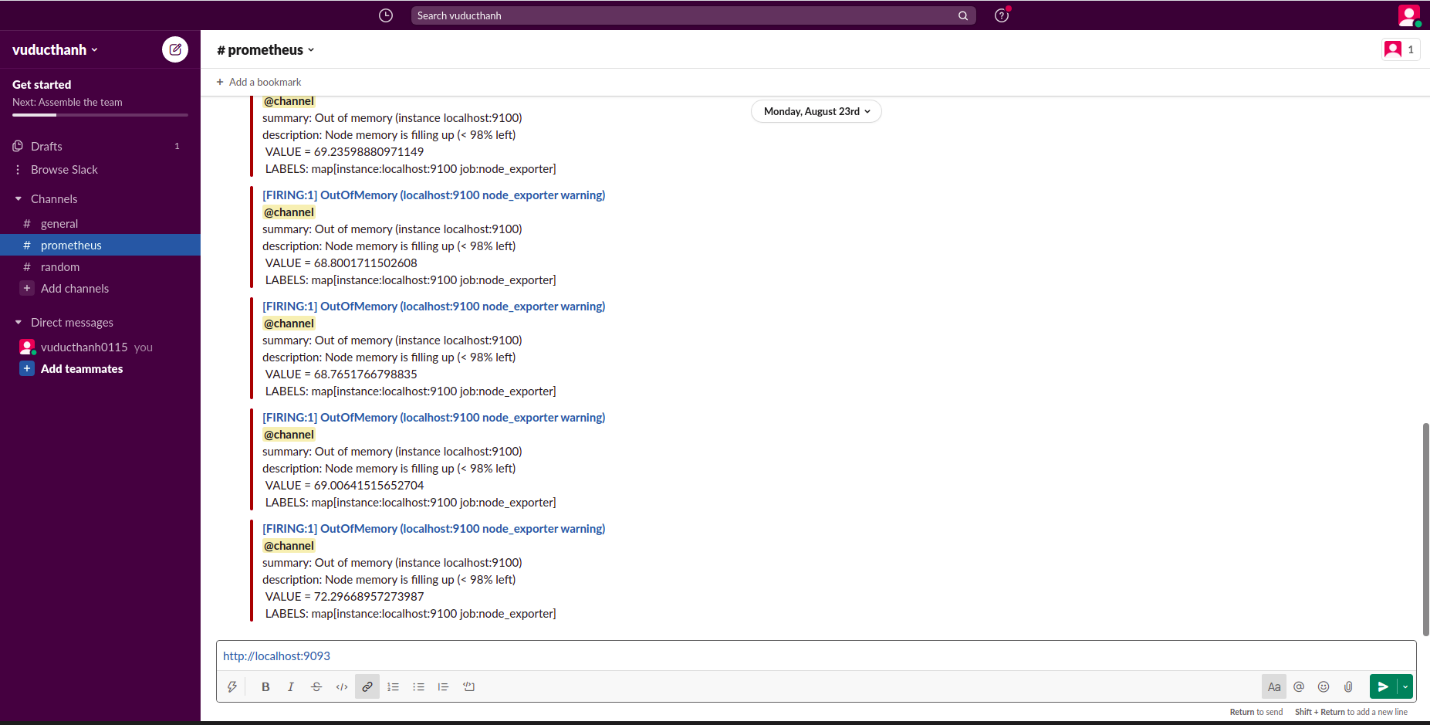
[](https://github.com/vuducthanh0115/Documents/blob/main/Image/28.png)

***Và đây là giao diện của Alertmanager:***

[](https://github.com/vuducthanh0115/Documents/blob/main/Image/29.png)

***Và đây là alertmanager bắn cảnh báo về mail + webhook :***

[](https://github.com/vuducthanh0115/Documents/blob/main/Image/33.png)

[](https://github.com/vuducthanh0115/Documents/blob/main/Image/31.png)

**4. Một số cảnh báo hệ thống**

**Node\_exporter**

***Memory còn trống dưới 10%***

*- alert: LowMemory*

*expr: (node\_memory\_MemFree\_bytes + node\_memory\_Cached\_bytes + node\_memory\_Buffers\_bytes) / node\_memory\_MemTotal\_bytes \* 100 < 10*

*for: 30m*

*labels:*

*severity: warning*

*annotations:*

*summary: "Out of memory (instance {{ $labels.instance }})"*

*description: "Node memory is filling up (< 10% left)\n VALUE = {{ $value }}\n LABELS: {{ $labels }}"*

**Ổ đĩa gần hết dung lượng**

Nhỏ hơn 10% thì cảnh báo

*- alert: LowDiskSpace*

*expr: node\_filesystem\_free\_bytes{mountpoint ="/rootfs"} / node\_filesystem\_size\_bytes{mountpoint ="/rootfs"} \* 100 < 10*

*for: 30m*

*labels:*

*severity: warning*

*annotations:*

*summary: "Out of disk space (instance {{ $labels.instance }})"*

*description: "Disk is almost full (< 10% left)\n VALUE = {{ $value }}\n LABELS: {{ $labels }}"*

**CPU cao**

*- alert: HighCpu*

*expr: node\_load15 / (count without (cpu, mode) (node\_cpu\_seconds\_total{mode="system"})) > 2*

*for: 30m*

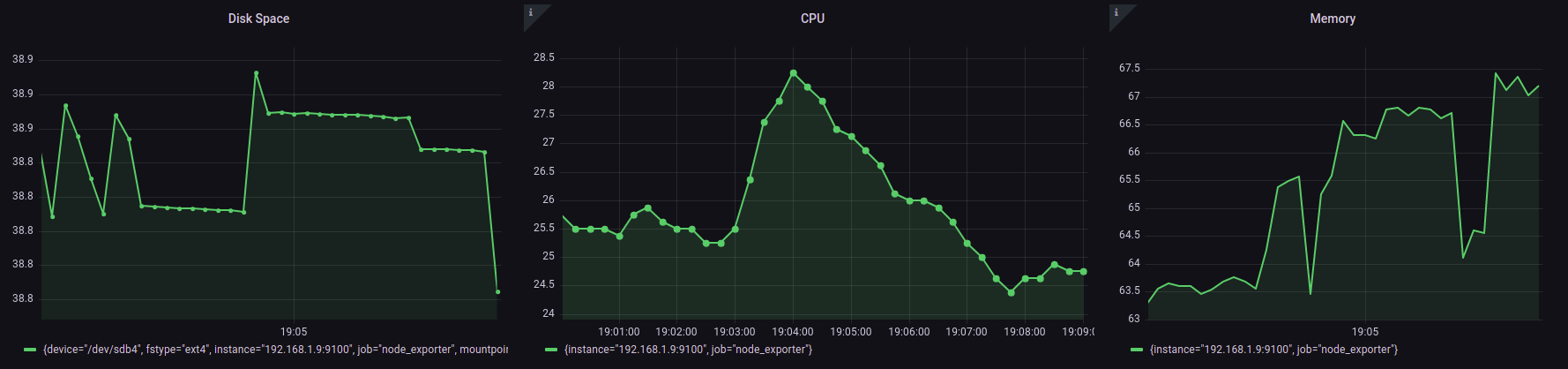
*labels:*

*severity: warning*

*annotations:*

*summary: "CPU load (instance {{ $labels.instance }})"*

*description: "CPU load (15m) is high\n VALUE = {{ $value }}\n LABELS:*

[](https://github.com/vuducthanh0115/Documents/blob/main/Image/34.png)

**cAdvisor**

*- alert: jenkins\_down*

*expr: absent(container\_memory\_usage\_bytes{name="jenkins"})*

*for: 30s*

*labels:*

*severity: critical*

*annotations:*

*summary: "Jenkins down"*

*description: "Jenkins container is down for more than 30 seconds."*

*- alert: jenkins\_high\_cpu*

*expr: sum(rate(container\_cpu\_usage\_seconds\_total{name="jenkins"}[1m])) / count(node\_cpu\_seconds\_total{mode="system"}) \* 100 > 10*

*for: 30s*

*labels:*

*severity: warning*

*annotations:*

*summary: "Jenkins high CPU usage"*

*description: "Jenkins CPU usage is {{ humanize $value}}%."*

*- alert: jenkins\_high\_memory*

*expr: sum(container\_memory\_usage\_bytes{name="jenkins"}) > 1200000000*

*for: 30s*

*labels:*

*severity: warning*

*annotations:*

*summary: "Jenkins high memory usage"*

*description: "Jenkins memory consumption is at {{ humanize $value}}."*

**Các kiến thức đã được dạy**

[Tổng quan về GIT 13](#_Toc81075656)

[1. Những khái niệm chung 14](#_Toc81075657)

[2. Các trạng thái có thể có trong git repository 15](#_Toc81075658)

[3. Các thao tác ban đầu 17](#_Toc81075659)

[4. Các thao tác liên quan khác 20](#_Toc81075660)

[5. Các thao tác với git remote 23](#_Toc81075661)

[6. Các thao tác với Stash 24](#_Toc81075662)

[Tổng quan về docker 25](#_Toc81075663)

[1. Giới thiệu Docker 25](#_Toc81075664)

[2. Các thành phần cơ bản của Docker 25](#_Toc81075665)

[3. Mạng nội bộ trong container 30](#_Toc81075666)

[Tổng quan về Linux 35](#_Toc81075667)

[1. Sự khác biệt của Linux với Windows 35](#_Toc81075668)

[2. Kiến trúc 36](#_Toc81075669)

[3. Các lệnh cơ bản 37](#_Toc81075670)

[4. Files and Directories 39](#_Toc81075671)

[5. Quản lý thư mục, quản lý tập tin 41](#_Toc81075672)

[6. Vim & Nano Editor 43](#_Toc81075673)

[7. Các biến môi trường 45](#_Toc81075674)

[8. Truyền thông giữa các máy 47](#_Toc81075675)

[9. Lập trình Bash Shell 48](#_Toc81075676)

[Tổng quan Prometheus Server 56](#_Toc81075677)

[1. Giới thiệu cơ bản về Prometheus 56](#_Toc81075678)

[2. Các khái niệm cơ bản của Prometheus 59](#_Toc81075679)

[3. Kiến trúc của Prometheus stack và các thành phần cơ bản. 60](#_Toc81075680)

[4. Cấu hình Prometheus 62](#_Toc81075681)

[5. Tìm hiểu về PromQL (Operators, Functions... + Example) 65](#_Toc81075682)

[6. Tìm hiểu về Labels 71](#_Toc81075683)

[7. Lưu trữ 72](#_Toc81075684)

[8. Tìm hiểu về Alertmanager 73](#_Toc81075685)

# Tổng quan về GIT

## 1. Những khái niệm chung

**Git là gì?**

Git là 1 phần mềm cài đặt trên máy tính giúp cho chúng ta có thể quản lí file, có thể biết được khi nào thì 1 file nào trong folder đó bị thay đổi bằng 1 lệnh git

Git cung cấp cho mỗi người kho lưu trữ (repository) riêng chứa toàn bộ lịch sử thay đổi.

Git cung cấp cho mỗi lập trình viên kho lưu trữ (repository) riêng chứa toàn bộ lịch sử thay đổi.

Lưu lại được các phiên bản khác nhau của mã nguồn dự án phần mềm

Khôi phục lại mã nguồn từ một phiên bản bất kỳ

Dễ dàng so sánh giữa các phiên bản

Phát hiện được ai đã sửa phần nào làm phát sinh lỗiKhôi phục lại tập tin bị mất

Dễ dàng thử nghiệm, mở rộng tính năng của dự án mà không làm ảnh hưởng đến phiên bản chính

Giúp phối hợp thực hiện dự án trong nhóm một cách hiệu quả

**Git repository là gì?**

Repository hay còn được gọi là repo.

Git repository là 1 nơi chứa tất cả các mã nguồn cho 1 dự án được quản lí bởi Git.

Trong Repo có 2 cấu trúc dữ liệu chính là Object Store và Index. Tất cả dữ liệu của Repo đều được chứa trong thư mục bạn đang làm việc dưới dạng folder ẩn có tên là .git.

Repository của Git được phân thành 2 loại : remote repository và local repository.

* Remote repository: Là repository để chia sẻ giữa nhiều người và bố trí trên server chuyên dụng.
* Local repository: Là repository bố trí trên máy của bản thân mình, dành cho một người dùng sử dụng.

**Git & Github & Gitlab giống/khác gì nhau?**

Git là một hệ thống kiểm soát phiên bản phân tán mã nguồn mở có sẵn cho tất cả mọi người với chi phí bằng không. Nó được thiết kế để xử lý các dự án từ nhỏ đến lớn với tốc độ và hiệu quả. Nó được phát triển để điều phối công việc giữa các lập trình viên. Kiểm soát phiên bản cho phép bạn theo dõi và làm việc cùng với thành viên trong nhóm của mình tại cùng một không gian làm việc.

GitHub cũng là một nền tảng lưu trữ online lớn về các dự án nhiều người làm.

GitLab là hệ thống self-hosted mã nguồn mở dựa trên hệ thống máy chủ Git dùng để quản lý mã nguồn của bạn. GitLab cung cấp giải pháp server một cách hoàn hảo

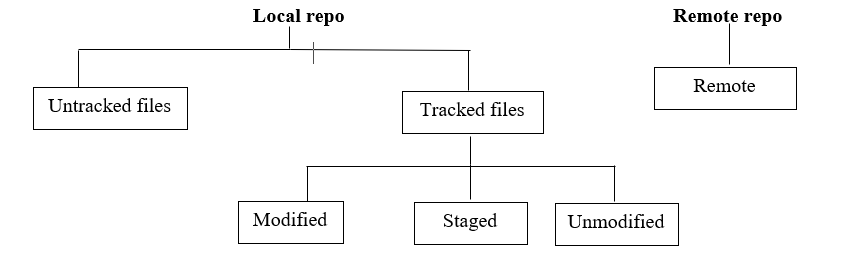
**Phân biệt :**

* Git là một công cụ kiểm soát phiên bản phân tán có thể quản lý lịch sử mã nguồn còn GitHub và GitLab là một công cụ dựa trên đám mây được phát triển xung quanh công cụ Git.
* Git là 1 công cụ cục bộ còn GitHub và GitLab là một dịch vụ trực tuyến để lưu trữ code và đẩy từ máy tính chạy công cụ Git.
* Git tập trung vào kiểm soát phiên bản và chia sẻ code còn GitHub và GitLab tập trung vào lưu trữ code
* Git là 1 công cụ dòng lệnh còn GitHub và GitLab được quản lí thông qua trang web.
* Git không cung cấp bất kỳ tính năng quản lý người dùng nào còn GitHub và GitLab được tích hợp sẵn tính năng quản lý người dùng.

## 2. Các trạng thái có thể có trong git repository

Repo của Git được phân thành 2 loại : remote repo và local repo.

Các trạng thái có thể có của file trong repo :

[](https://github.com/vuducthanh0115/img/blob/main/anh/1.png)

**Untracked Files**: Nếu tạo ra hoặc thêm vào một tập tin mới vào trong thư mục làm việc của mình thì nó sẽ ở trạng thái Untracked.

**Tracked Files**: Những files đã được thêm vào danh sách theo dõi của git được gọi là Tracked Files, những file này khi chúng ta thay đổi / thêm / xoá thì git sẽ nhận biết được điều đó và lưu các thay đổi này lại theo yêu cầu của chúng ta.

**Modified Files**: Ta thay đổi tập tin nhưng chưa commit vào cơ sở dữ liệu có nghĩa chưa sử dụng lệnh git add và git commit.

**Staged Files**: Staged là đã đánh dấu sẽ commit phiên bản hiện tại của một tập tin đã chỉnh sửa trong lần commit sắp tới. VD sử dụng lệnh git add <file\_name> nhưng chưa commit.  
**Unmodified Files**: File được Git quản lí nhưng không có thay đổi gì.

**Config cho git với git config**

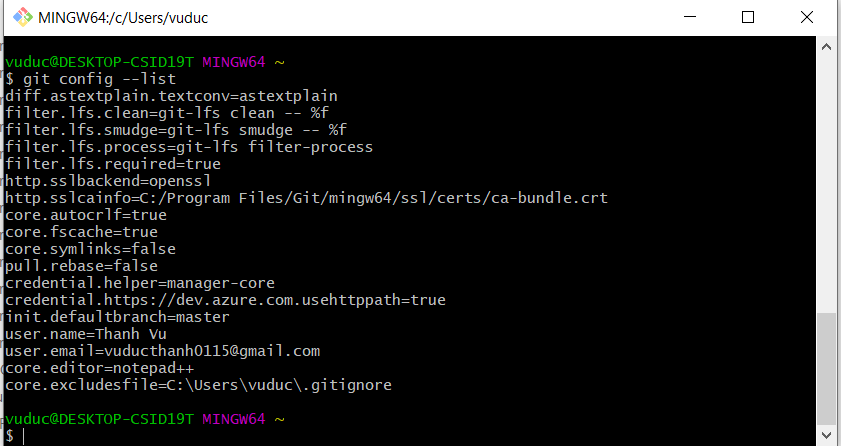
* Config thông tin cá nhân
* Config default/prefered editor
* Config global .gitignore

***Các lệnh cấu hình :***

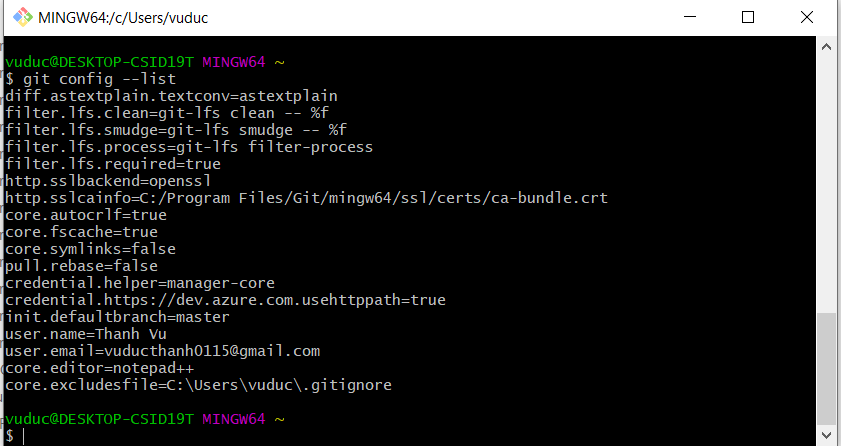
$ git config --global user.name "<name>"

$ git config --global user.email <mail>

$ git config --global core.editor "'C:\Program Files\Notepad++\notepad++.exe' -multiIns -nosession"

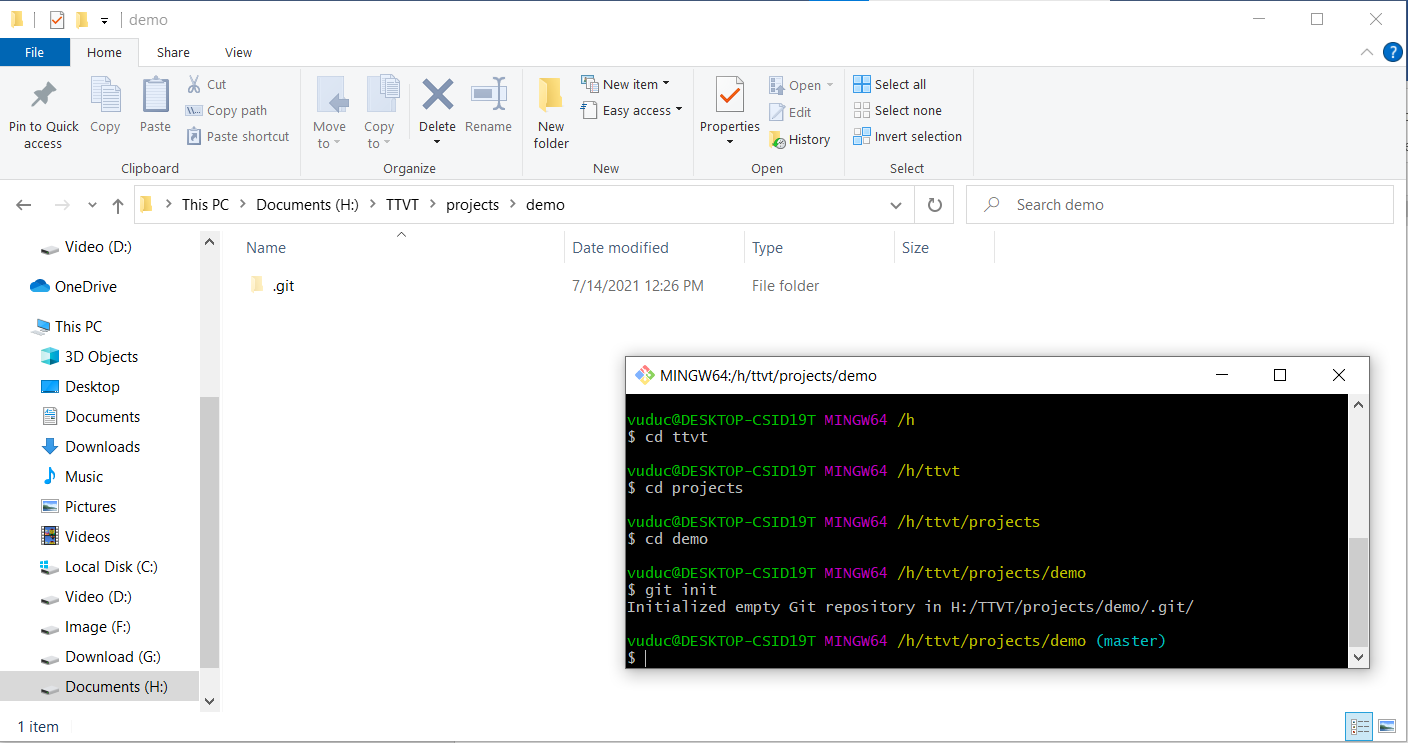
[](https://github.com/vuducthanh0115/img/blob/main/anh/2.png)

## 3. Các thao tác ban đầu

[](https://github.com/vuducthanh0115/img/blob/main/anh/2.png)

***Khởi tạo repo với git init***

Lệnh git init được sử dưng để tạo, khởi tạo một kho chứa Git mới (Git Repo) ở local. Khi đang trong thư mục dự án chạy lệnh git init nó sẽ tạo ra một thư mục con (ẩn) tên .git, thư mục này chứa tất cả thông tin mô tả cho kho chứa dự án (Repo) mới - những thông tin này gọi là metadata gồm các thư mục như objects, refs ... Có một file tên HEAD cũng được tạo ra - nó trỏ đến commit hiện tại.  
Lệnh: $ git init

[](https://github.com/vuducthanh0115/img/blob/main/anh/3.png)

**Git commit**

**Commit trong git là gì ?**

Commit là thao tác báo cho hệ thống biết mình muốn lưu lại trạng thái hiện hành, ghi nhận lại lịch sử các xử lý như đã thêm, xóa, cập nhật các file hay thư mục nào đó trên repo.  
Khi thực hiện commit, trong repo sẽ ghi lại sự khác biệt từ lần commit trước với trạng thái hiện tại. Các commit ghi nối tiếp với nhau theo thứ tự thời gian.

**Một commit chứa những thông tin gì ?**

Một commit chứa tất cả những thay đổi của mình tất cả những gì mình đã làm và có ngày giờ cụ thể tại thời điểm chúng ta thay đổi.

**Amending - Thay Đổi Commit Cuối Cùng**

* Khi chúng ta commit nhưng bị quên add một số file nào đó và không muốn tạo ra một commit mới thì có thể sử dụng lệnh $ git commit --amend để gộp các file đó và bổ sung vào commit cuối cùng, vì vậy không tạo ra commit mới.
* Muốn loại bỏ một file ra khỏi stage thì sử dụng lệnh $ git reset HEAD <file\_name>.

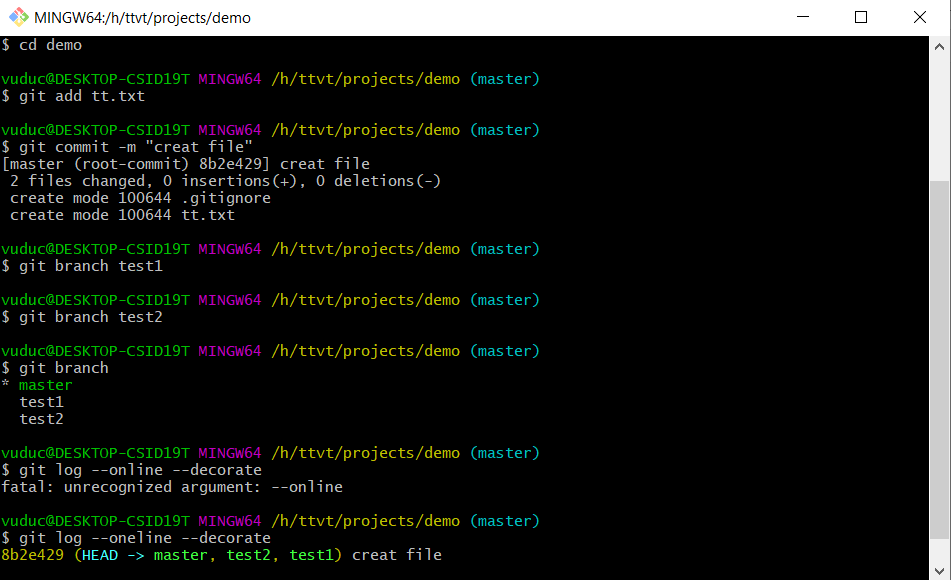
**Làm việc với branch**

Branch là những phân nhánh ghi lại luồng thay đổi của lịch sử, các hoạt động trên mỗi branch sẽ không ảnh hưởng lên các branch khác nên có thể tiến hành nhiều thay đổi đồng thời trên một repo, giúp giải quyết được nhiều nhiệm vụ cùng lúc.

Khi tham gia dự án với mỗi nhiệm vụ chúng ta sẽ tạo một branch và làm việc trên đó, các branch này sẽ hoạt động riêng lẻ và không ảnh hưởng lẫn nhau.

**Tạo mới branch**

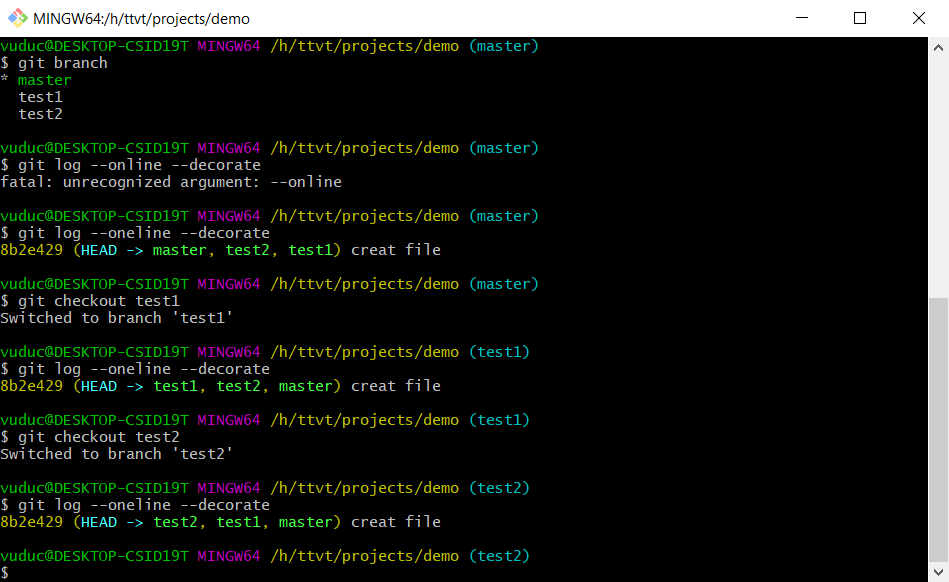
Để tạo 1 branch mới ta sử dụng lệnh : $ git branch <branchname>

[](https://github.com/vuducthanh0115/img/blob/main/anh/4.png)

==> Khi tạo mới nhánh thì nhánh master và test đều trỏ tới cái commit hiện tại.

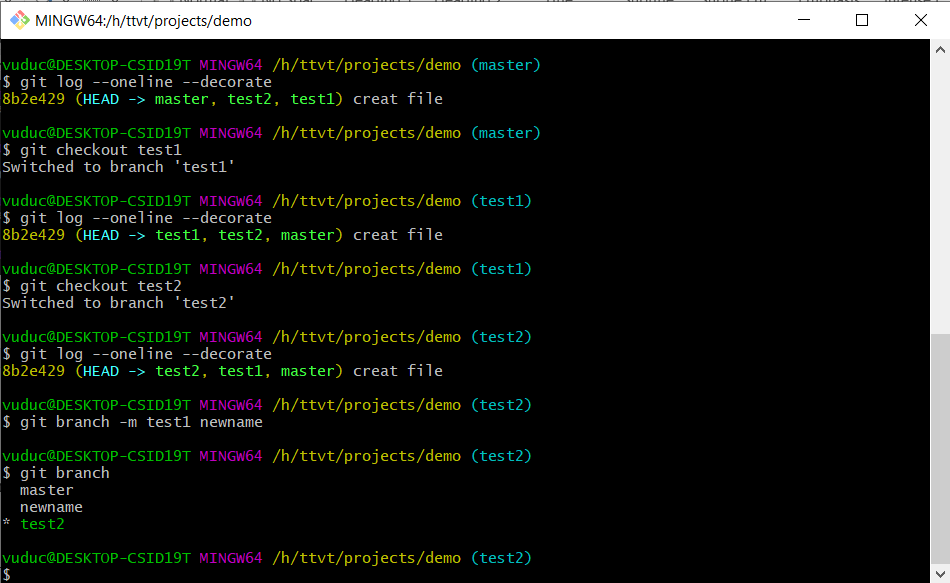
**Nhảy từ branch này sang branch khác**

Khi nhảy từ branch này sang branch khác ta dùng câu lệnh: $ git checkout <branch\_name>

[](https://github.com/vuducthanh0115/img/blob/main/anh/5.png)

**Đổi tên branch**

Muốn đổi tên branch ta dùng lệnh: $ git branch -m

[](https://github.com/vuducthanh0115/img/blob/main/anh/6.png)

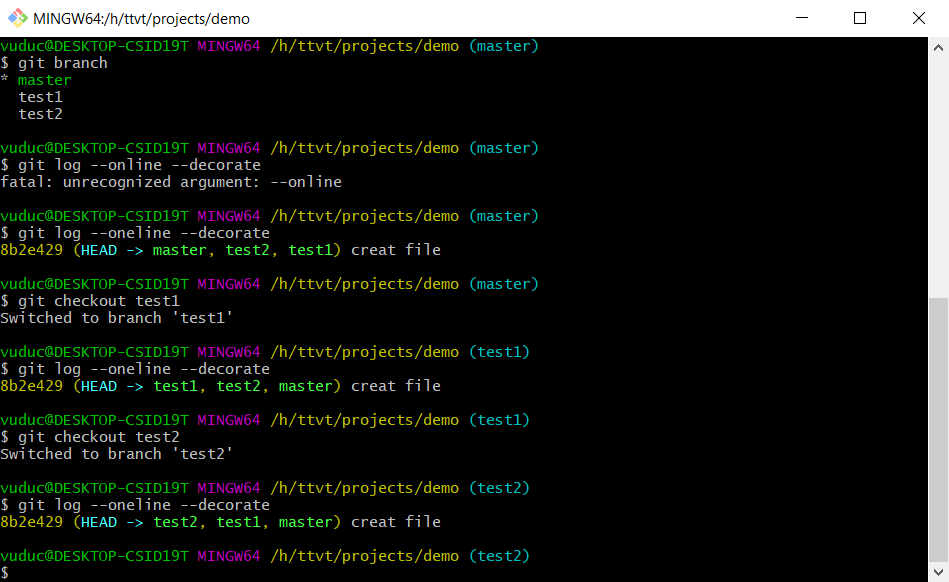
**Git checkout**

Lệnh git checkout được dùng để chuyển nhánh hoặc để phục hồi file trong thư mục làm việc từ một commit trước đây .

VD: Ta đang ở nhánh nào đó muốn chuyển sang nhánh master thì thực hiện lệnh:

$ git checkout master

Lúc này nhánh master hoạt động, và thư mục làm việc là các file tương ứng với nhánh này

[](https://github.com/vuducthanh0115/img/blob/main/anh/7.png)

## 4. Các thao tác liên quan khác

**Git Merge**

Git Merge là một lệnh dùng để hợp nhất các chi nhánh độc lập thành một nhánh duy nhất trong Git.

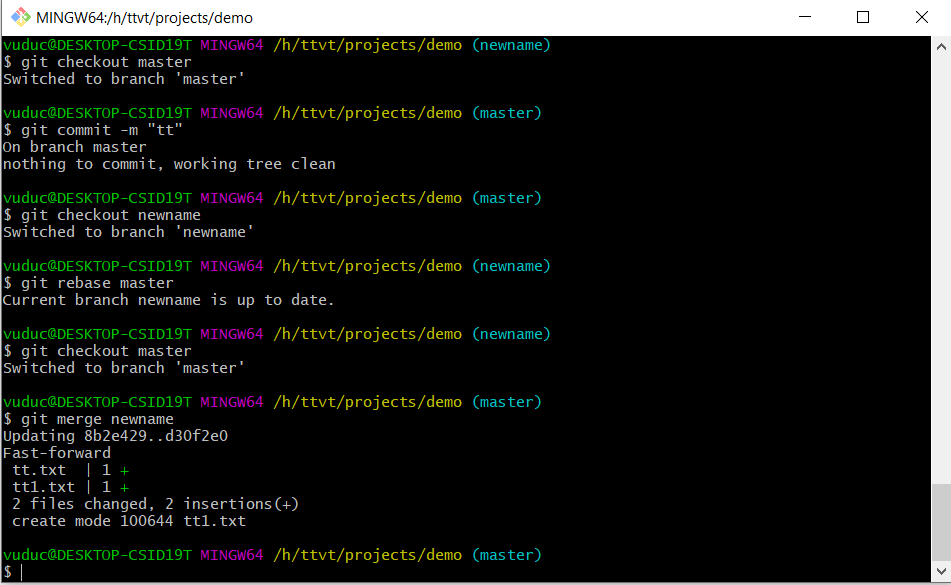
Khi sử dụng lệnh hợp nhất trong Git, chỉ có nhánh hiện tại được cập nhật để phản ánh sự hợp nhất, còn nhánh đích sẽ không bị ảnh hưởng.

Để merge một branch bất kì vào branch hiện tại thì bạn sử dụng lệnh: $ git merge <branch\_name>

**Git Rebase**

Git Rebase là một chức năng được dùng khi gắn nhánh đã hoàn thành công việc vào nhánh gốc . Việc điều chỉnh nhánh công việc gắn vào với nhánh gốc nên các commit sẽ được đăng kí theo thứ tự gắn vào .

Câu lệnh : $ git rebase <namebranch>

[](https://github.com/vuducthanh0115/img/blob/main/anh/8.png)

**Git Pull**

Git Pull là một lệnh dùng để tải xuống dữ liệu từ một Remote repo và cập nhật Local repo phù hợp với dữ liệu đó. Có nghĩa là Git Pull là lệnh hợp nhất các thay đổi từ Remote repo vào Local repo.

**Git Push**

Lệnh git push được sử dụng để đẩy các commit mới ở local repo lên remote repo. Nguồn để đẩy lên là nhánh mà con trỏ HEAD đang trỏ tới (nhánh làm việc).

Lệnh git push cũng có thể xóa đi các nhánh của remote.

Một số tham số hay dùng như:

* all đẩy tất cả các nhánh lên server
* tags đẩy tất cả các tag lên server
* delete xóa một nhánh chỉ ra trên server
* u đẩy và tạo một upstream (luồng upload tương ứng với nhánh của local), hay sử dụng cho lần đầu đẩy lên server

***Đẩy lên server lần đầu tiên***

Nếu là lần đầu tiên đẩy Local Repo lên Remote Repo mới khởi tạo thì cần tạo ra một theo dõi kết nối, upstream giữa local và remote, vậy hãy dùng tham số -u. Ví dụ đẩy lên remote có tên origin và tạo upstream cho nhánh master

Lệnh : $ git push -u origin master

***Đẩy lên server***

Sau khi có upstream, mỗi lần cần đẩy dữ liệu lên remote của nhánh master, chỉ việc thực hiện lệnh  
Lệnh : $ git push

***Đẩy lên server tất cả các nhánh***

Đẩy tất cả các nhánh ở local lên server có tên origin:  
Lệnh : $ git push origin --all

***Xóa một nhánh trên remote***

VD xóa nhánh test, trên remote có tên origin  
Lệnh : $ git push origin --delete test

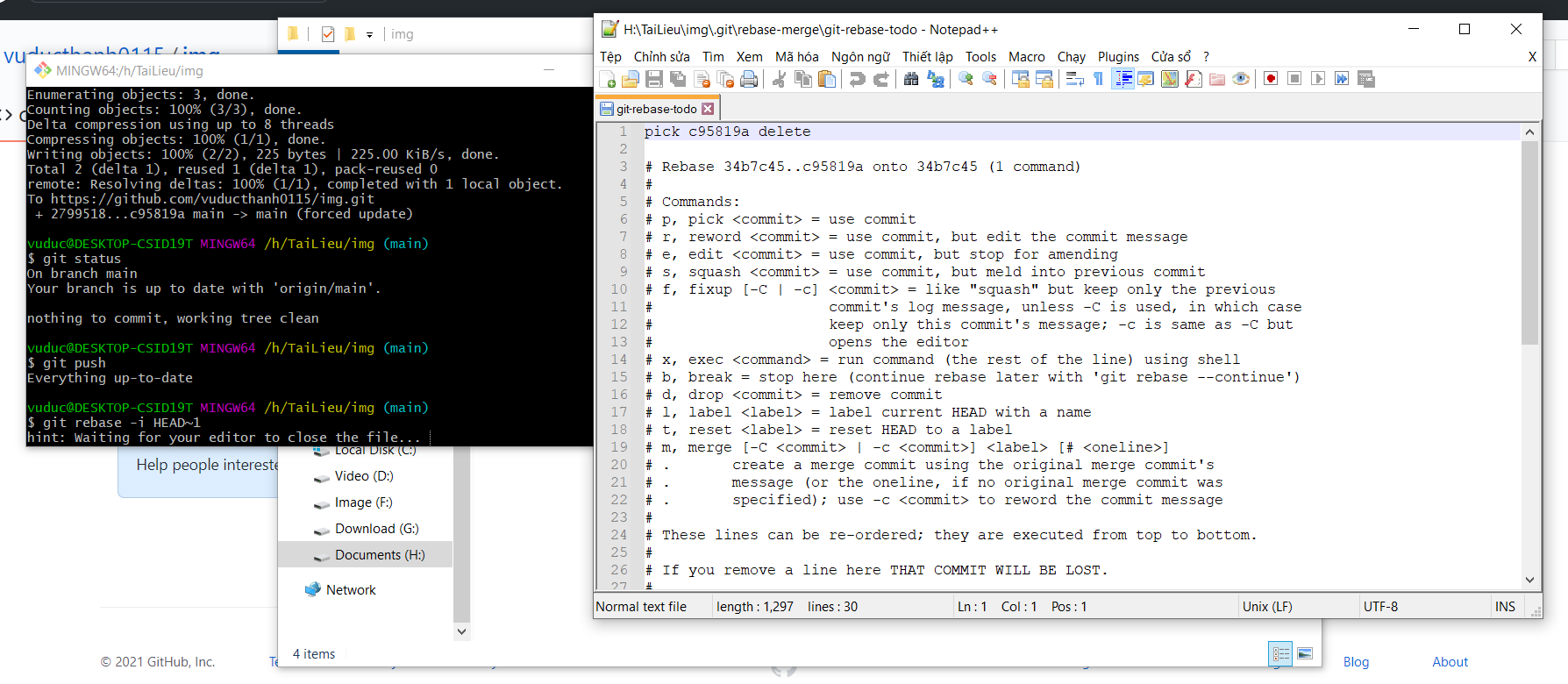
***Ghi đè nhánh với --force***

Có thể ghi đè toàn bộ một nhánh ở remote bởi một nhánh ở master

VD ghi đè toàn nhánh master ở remote, giống với master của local  
Lệnh : $ git push --force origin master

**Git Rebase interactive**

Khi chúng ta commit thiếu file lên git server hoặc ghi nội dung comment chưa được như ý muốn hoặc nội dung comment chưa đúng theo quy tắc chung đã đưa ra. Thì để sửa nội dung đã commit chúng ra có thể sử dụng lệnh : $ git rebase -i

[](https://github.com/vuducthanh0115/img/blob/main/anh/9.png)

**Git Cherry-pick**

Là một cách để checkout 1 commit bất kỳ tại 1 branch được chỉ định về brach hiện tại. Hay chính là git cherry-pick sẽ lấy thay đổi của 1 commit trên 1 nhánh nào đó áp dụng vào nhánh hiện tại.

**Tùy chọn force**

Lệnh : git push -f origin master cho phép push kho lưu trữ cục bộ đến từ xa mà không cần giải quyết xung đột. Cách này khá nguy hiểm vì theo cơ chế, nó sẽ ghi đè lên remote repo bằng code ở local của mình, mà không cần quan tâm đến việc bên phía remote đang chứa thứ gì.

Cách push an toàn : sử dụng push --force-with-lease, git sẽ từ chối việc update lên branch trừ khi branch đó nằm trong trạng thái "được coi là an toàn".

**Git reset**

Git reset được dùng để quay về một điểm commit nào đó, đồng thời xóa lịch sử của các commit trước nó.

Sử dụng reset để xóa commit : $ git reset --hard HEAD  
Sử dụng reset để xóa commit :$ git reset --hard HEAD

## 5. Các thao tác với git remote

**Add remote**

Để thêm một remote repo vào local repo thì bạn sử lệnh: $ git remote add [shortname] [url]

Trong đó:

* shortname: là tên mà bạn muốn đặt cho remote repo
* url: là đường dẫn trỏ đến repo, phần đuôi của URL sẽ là .git

**Rename remote**

Để đổi tên cho remote thì bạn sử dụng lệnh sau : $ git remote rename old\_name new\_name

**Update remote**

Để cập nhật dữ liệu từ remote repo về local repo thì ta sử dụng lệnh git fetch hoặc lệnh git pull.

Lệnh git fetch tải về dữ liệu từ Remote Repo (các dữ liệu như các commit, các file, refs), dữ liệu tải về để chúng ta theo dõi sự thay đổi của remote, tuy nhiên tải về bằng git fetch nó chưa tích hợp thay đổi ngay local repository, mục đích để theo dõi các commit người khác đã cập nhật lên server, để có được thông thông tin khác nhau giữa remote và local.

Lệnh git pull lấy về thông tin từ remote và cập nhật vào các nhánh của local repo.

**Prune remote branch**

Lệnh : $ git remote prune

Xóa các ref cho các nhánh không tồn tại trên remote.

Nó loại bỏ các nhánh từ xa không có đối tác cục bộ. Ví dụ : nếu có một nhánh từ xa nói bản demo, nếu nhánh này không tồn tại cục bộ, thì nó sẽ bị xóa.

## 6. Các thao tác với Stash

**Tổng quan về git stash**

Lệnh git stash sẽ có tác dụng với tất cả dữ liệu đang hoạt động trong working directory với điều kiện là dữ liệu đó đã được đưa vào trạng thái Staged hoặc đã từng được committed.

**git stash list**

Xem danh sách bao nhiêu lệnh stash đã dùng.

Nếu chúng ta đã stash rất nhiều lần và muốn xem danh sách và địa chỉ của nó thì sử dụng lệnh : $ git stash list

**git stash clear**

Nếu chúng ta muốn xóa tất cả stash ra khỏi history thì sử dụng tham số clear Lệnh : $ git stash clear

==> Lúc này danh sách stash sẽ rỗng.

# Tổng quan về docker

## 1. Giới thiệu Docker

Docker là một nền tảng cho phép bạn đóng gói, triển khai và chạy các ứng dụng một cách nhanh chóng. Ứng dụng Docker chạy trong vùng chứa (container) có thể được sử dụng trên bất kỳ hệ thống nào: máy tính xách tay của nhà phát triển, hệ thống trên cơ sở hoặc trong hệ thống đám mây. Và là một công cụ tạo môi trường được "đóng gói" (còn gọi là Container) trên máy tính mà không làm tác động tới môi trường hiện tại của máy, môi trường trong Docker sẽ chạy độc lập.

Docker được tạo ra để làm việc trên nền tảng Linux , nhưng đã mở rộng để cung cấp hỗ trợ lớn hơn cho các hệ điều hành không phải Linux, bao gồm Microsoft Windows và Apple OS X

## 2. Các thành phần cơ bản của Docker

**Docker Engine**

Docker Engine là công cụ Client - Server hỗ trợ công nghệ container để xử lý các nhiệm vụ và quy trình công việc liên quan đến việc xây dựng các ứng dụng dựa trên vùng chứa (container). Engine tạo ra một quy trình daemon phía máy chủ lưu trữ images, containers, networks và storage volumes. Daemon cũng cung cấp giao diện dòng lệnh phía máy khách (CLI) cho phép người dùng tương tác với daemon thông qua giao diện lập trình ứng dụng Docker.

4 đối tượng của Engine là: images, containers, network, volume chúng đều có ID để xác định và phối hợp với nhau để có thể build, ship và run application ở bất cứ đâu

**Images**: là thành phần để đóng gói ứng dụng và các thành phần mà ứng dụng phụ thuộc để chạy. Và image được lưu trữ ở trên local hoặc trên một Registry (là nơi lưu trữ và cung cấp kho chứa các image)

**Containers**: là một instance của image, và nó hoạt động như một thư mục, chứa tất cả những thứ cần thiết để chạy một ứng dụng

**Network**: cung cấp một private network chỉ tồn tại giữa container và host

**Volume**: Volume trong Docker được dùng để chia sẻ dữ liệu cho container

**Docker Container**

Một docker container là một môi trường bị tách biệt để đóng gói và chạy ứng dụng. Docker cung cấp một tùy chọn để chạy một ứng dụng trong các container nằm bên cạnh nhau để tăng hiệu quả tính toán. Bạn có thể chạy nhiều container trên cùng một host. Và bạn cũng có thể dễ dàng di chuyển những container này từ host này sang host khác.  
Docker container là một ví dụ của image. Một container chỉ cần kết hợp với các thư viện và thiết lập cần thiết để làm cho ứng dụng hoạt động. Nó là một môi trường đóng gói gọn nhẹ và di động cho một ứng dụng.

***Cách chạy một docker container***

Sử dụng lệnh docker để khởi chạy docker container trên hệ thống. Ví dụ lệnh bên dưới sẽ tạo một Docker Container từ image có tên “hello-world”  
docker run hello-world

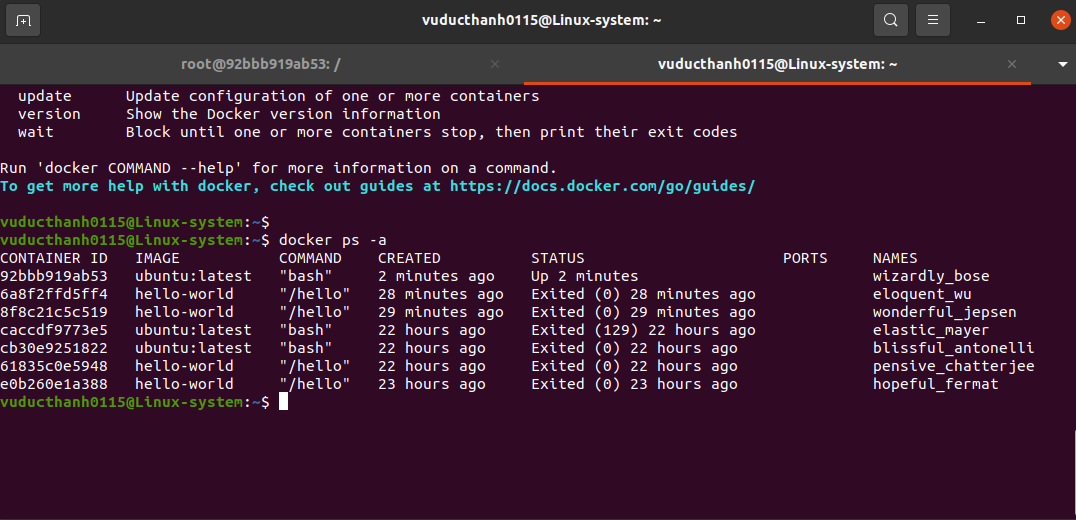
[](https://github.com/vuducthanh0115/Documents/blob/main/test.txt/1_3.png)

***Liệt kê danh sách docker container***

Dùng lệnh docker ps để liệt kê các container đang chạy trên hệ thống hiện tại. Nó sẽ không liệt kê các container bị dừng. Nó sẽ hiển thị Container ID, name và các thông tin khác về container.  
docker ps

[](https://github.com/vuducthanh0115/Documents/blob/main/test.txt/2_3.png)

Dùng tùy chọn -a với lệnh ở trên để liệt kê tất cả các container bao gồm cả container bị dừng.  
$ docker ps -a

[](https://github.com/vuducthanh0115/Documents/blob/main/test.txt/3_3.png)

Tìm kiếm tất cả thông tin chi tiết về container

$ docker inspect <ID container>

***Xóa Docker container***

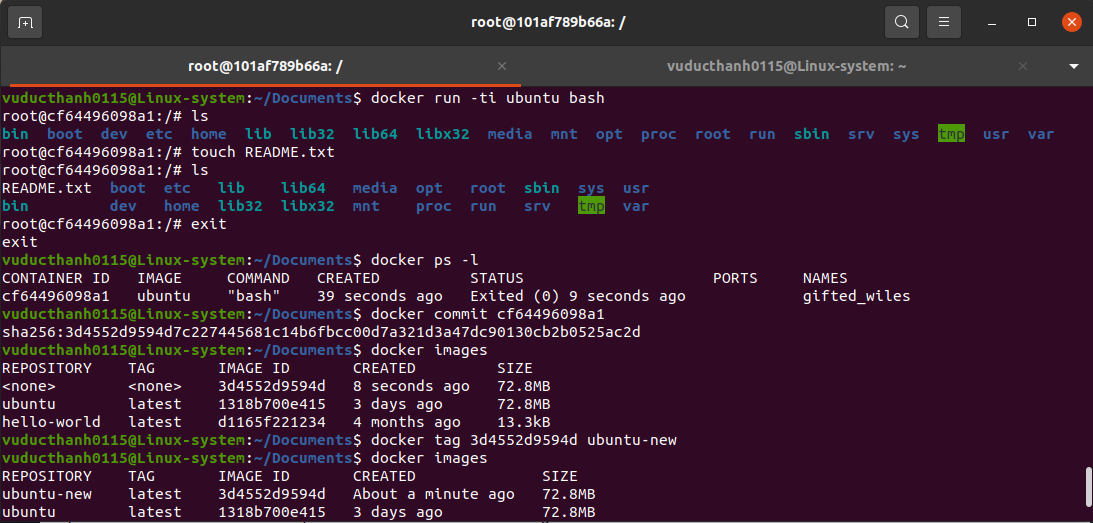
Dùng lệnh docker rm để xóa docker container đang tồn tại. Bạn cần cung cấp docker container id hoặc container name để xóa một container cụ thể.

docker stop <ID container>

docker rm <ID container>

**Lưu container thành images**

Ta sử dụng lệnh : docker commit <containeri ID>

[](https://github.com/vuducthanh0115/Documents/blob/main/test.txt/4_3.png)

**Cơ chế lưu trữ của docker**

Cơ chế lưu trữ copy-on-write

Tạo 1 hệ thống mới ngay lập tức mà không cần copy tất cả file hệ thống

Hệ thống lưu trữ lưu lại mỗi thay đổi

**Chạy các tiến trình trong docker contaier**

**docker run**

Khởi tại 1 container bằng cách truyền vào tên 1 images và 1 tiến trình để yêu cầu chạy trong container.

Container có tiến trình chính

Container dừng khi tiến trình chính kết thúc

Container có thể được đặt tên

**docker attach**

Truy cập container đang chạy thông qua ID hoặc tên

Có thể truy cập vào cùng 1 container từ nhiều nơi tại cùng 1 thời điểm

Ctrl-p/Ctrl-Q để ra khỏi container và vẫn để tiến trình chạy

**docker exec**

Khỏi tại 1 process khác bên trong container đã có

Tiện lợi cho việc debug và quản lí database

Không thể thêm port, volume, ...

**Một số lệnh thường dùng với container**

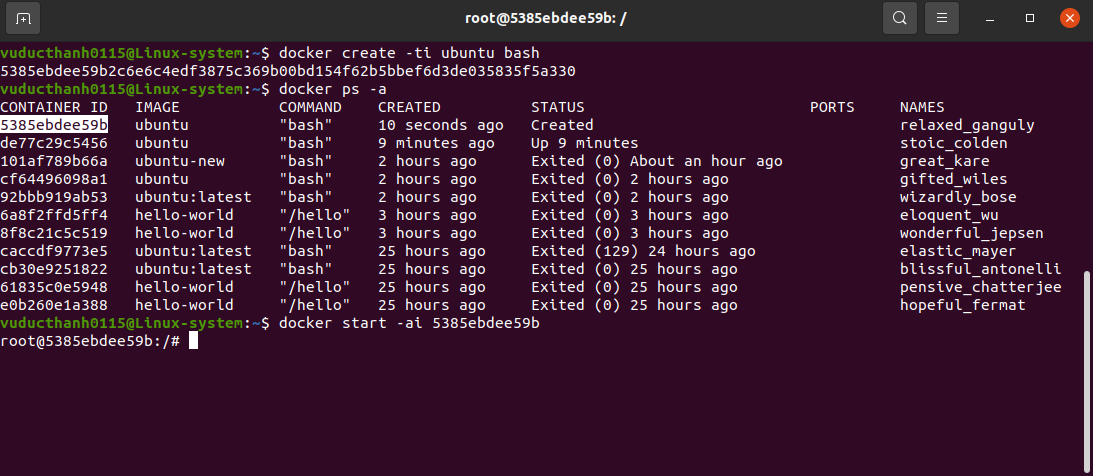
**docker create**

Tạo ra 1 container với các config tương tự docker run

Container không được chạy ngay từ đầu

Dùng lệnh docker start để chạy container

Cú pháp: docker create [options] images [command][arg...]

[](https://github.com/vuducthanh0115/Documents/blob/main/test.txt/5_3.png)

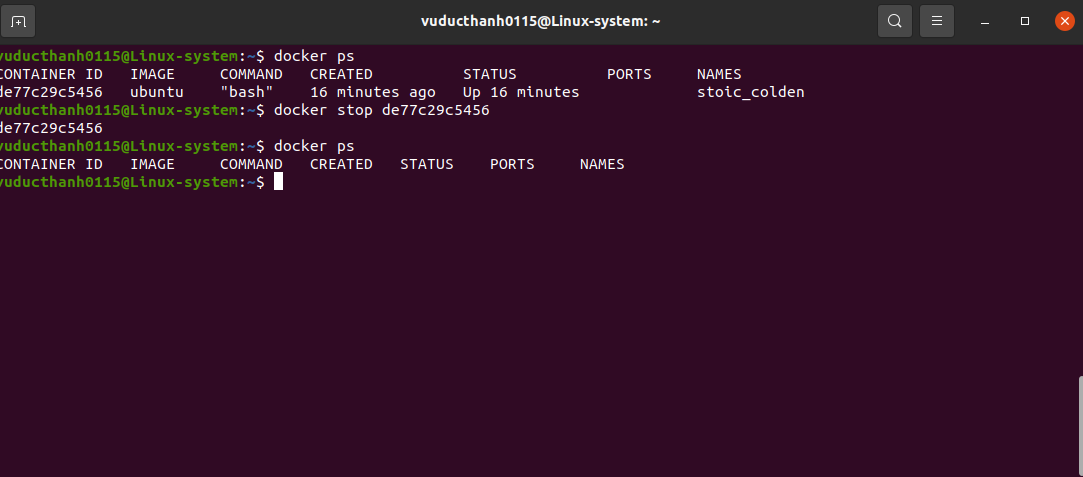
**docker start/stop/restart**

Thay đổi trạng thái cảu container, từ running sang stopped hoặc khởi động lại container

Stop :

* Gửi tín hiệu SIGTERM để yêu cầu tắt tiến trình
* Gửi tín hiệu SIGKILL nếu chưa tắt sau 1 khoảng thời gian nhất định

Cú pháp: docker start/stop/restart [options] container [container]

[](https://github.com/vuducthanh0115/Documents/blob/main/test.txt/6_3.png)

**docker cp**

Copy dữ liệu (file/filder) từ container tới local machine và ngược lại

Áp dụng cho tất cả stopped và running container

Đường dẫn relative

Cú pháp: docker cp [options] container:src\_path dest\_path

**docker inspect**

Kiểm tra các thiết lập của Docker (network, driver, ...)

In ra các thông tin dưới dạng JSON

Cú pháp: docker inspect [options] container|image|task [container|image|task...]

**docker rm**

Xóa container dựa theo name hoặc ID

Cần stop container trước khi xóa bỏ

Cú pháp: docker rm [options] container [container...]

**docker logs**

Xem output của container

docker logs [option] container

Không nên để cho dung lượng log quá lớn

**docker kill**

Dừng container  
docker kill [option] container [container...]

**docker run --memory**  
Giới hạn tài nguyên

**docker run --cpu-shares=**  
Giới hạn CPU tương đối

## 3. Mạng nội bộ trong container

Các chương trình trong container được mặc định tách biệt hoàn toàn khỏi Internet

Nhóm các container lại thành 1 mạng nội bộ

Kiểm soát sự liên kết giữa các container này với container kia

Sử dụng phương pháp expose cổng và link container

Docker có cơ chế đặc biệt giúp container tìm và liên kết với nhau

Xác định cổng bên trong và bên ngoài container

Không giới hạn số lượng cổng để mở

**Mở cổng container**

Cần sự phối hợp giữa các container

Dễ dàng tìm kiếm cổng được mở

Cổng bên trong container là cố định

Cổng bên ngoài container được lựa chọn ngẫu nhiên

Cho phép nhiều container chạy các chương trình với cổng cố định

Thường được sử dụng với các chương trình tìm kiếm dịch vụ

**Liên kết giữa các container**

Liên kết tr tiếp các container

* Dùng để kiểm tra xem container đang chạy gì và ở chỗ nào
* Kết nối tất cả các cổng, nhưng 1 chiều
* Chỉ dành cho các dịch vụ mà không thể chạy trên nhiều máy khác nhau
* Tự động gán hostname
* Liên kết có thể bị đứt nếu container khởi động lại

Liên kết động giữa các container

* Docker cung cấp tính năng tạo mạng riêng nội bộ
* Name server với cơ chế quản lý địa chỉ IP và name
* Cần tạo lập mạng trước rồi chạy các container liên quan
* Lập mạng riêng : docker network create [option] network

**Cách cài đặt docker**

***Bước 1:*** Update các package và cài các package cần thiết

sudo apt-get update

sudo apt-get install apt-transport-https ca-certificates curl gnupg lsb-release

***Bước 2:*** Thêm Docker’s official GPG key:

curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg --dearmor -o /usr/share/keyrings/docker-archive-keyring.gpg

***Bước 3:*** Thêm Docker Repository

echo "deb [arch=amd64 signed-by=/usr/share/keyrings/docker-archive-keyring.gpg] https://download.docker.com/linux/ubuntu $(lsb\_release -cs) stable"

***Bước 4:*** Cài đặt docker

sudo apt-get update

sudo apt-get install docker-ce docker-ce-cli containerd.io

***Bước 5:*** Kiểm tra

sudo docker run hello-world

**Các câu lệnh cơ bản**

***Docker tag***

Gắn tag cho TARGET\_IMAGE tương ứng với SOURCE\_IMAGE.

docker tag SOURCE\_IMAGE[:TAG] TARGET\_IMAGE[:TAG]

***Docker run***

Đây là câu lệnh dùng để khởi tạo một container dựa vào Image có sẵn, cú pháp:

docker run [OPTIONS] IMAGE[:TAG|@DIGEST] [COMMAND] [ARG...]

Option  
--name cho phép đặt tên container.  
-t cung cấp một giao diện để gõ command bên trong container.  
-i giúp cho các chương trình bên trong container nhận được những command đã viết.

***Docker build***

Thực hiện lệnh trong thư mục chứa file Dockerfile  
docker build [OPTIONS] PATH | URL | -

***Docker push***

Đẩy image, repository lên Docker HUB.

docker push [OPTIONS] NAME[:TAG]

***Docker pull***

Pull một image hoặc repository từ Docker HUB đăng ký.

docker pull [OPTIONS] NAME[:TAG|@DIGEST]

**Docker CE/Docker EE**

Docker Community Edition (CE): Là phiên bản miễn phí và chủ yếu dựa vào các sản phầm nguồn mở khác.

Docker Enterprise(EE): Khi sử dụng phiên bản này bạn sẽ nhận được sự support của nhà phát hành, có thêm các tính năng quản lý và security.

**Docker Compose**

Docker compose là công cụ dùng để định nghĩa và run multi-container cho Docker application. Với compose sử dụng file YAML để config các services cho application. Sau đó dùng command để create và run từ những config đó.

Docker compose cho phép tạo nhiều service(container) giống nhau bằng lệnh : $ docker-compose scale <tên service> = <số lượng>

Sử dụng chỉ với ba bước:  
Khai báo app’s environment trong Dockerfile.  
Khai báo các services cần thiết để chạy application trong file docker-compose.yml.  
Run docker-compose up để start và run app.

Cú pháp :

* Tạo file docker-compose.yml
* Viết theo cú pháp của ngôn ngữ yaml
* Viết theo đúng cú pháp cảu từng version cảu docker-compose

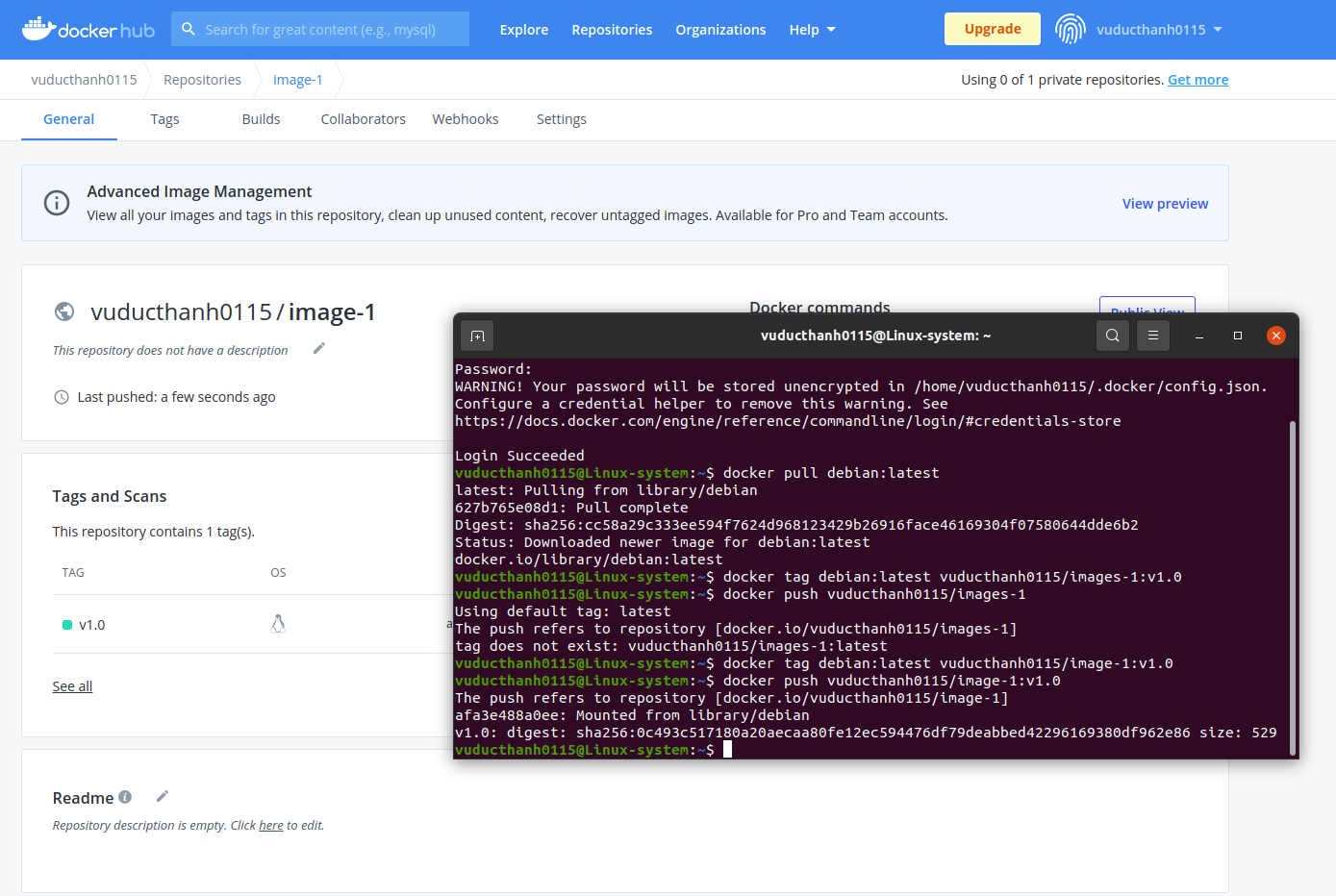
Các lệnh trong docker compose :

* version: chỉ ra phiên bản docker-compose đã sử dụng
* services: thiết lập các services(containers) muốn cài đặt và chạy
* image: chỉ ra image được sử dụng trong lúc tạo container
* build: dùng để tạo container
* port: thiết lập ports chạy tại máy host và trong container
* restart: tự động khởi chạy container bị tắt
* environment: thiết lập biến môi trường
* depends\_on: chỉ ra sự phụ thuộc. Tức là service nào phải dduwwocj cài đặt và chạy trước thì service dduwwocj config tại đói mới được chạy
* volumes: dùng để mount 2 thư mục trên host và container với nhau
* links: Cho phép đăng kí 1 cái tên(bí danh) để cho các Server khác có thể gọi nó
* context: Xác định thư mục chứa gốc, dựa vào thư mục này chúng ta sẽ khai báo tiếp đường dẫn dockerfile

**Docker Registry**

Docker Registry là một dịch vụ máy chủ cho phép lưu trữ các docker image của cá nhân, công ty, team,… Dịch vụ Docker Registry có thể được cung cấp bởi tổ chức thứ 3 hoặc là dịch vụ nội bộ được xây dựng riêng nếu bạn muốn. Một số dịch vụ Docker Registry phổ biến như :

* Azure Container Registry
* Docker Hub
* Quay Enterprise
* Google Container Registry.
* AWS Container Registry

[](https://github.com/vuducthanh0115/Documents/blob/main/test.txt/7_3.png)

**Docker Daemon**

Docker Daemon: là server Docker cho yêu cầu từ Docker API. Nó quản lý images, containers, networks và volume.

**Docker volume**

Là 1 nơi lưu trữ data nằm ngoài container, mục đích không làm mất data khi xóa container

Sử dụng volume khi :

* Khi chia sẻ dữ liệu giữa nhiều container đang chạy
* Lưu dữ liệu tới 1 server remote hoặc cloud
* Khi cần backup, restore hoặc migrate dữ liệu từ Docker Host này sang Docker Host khác

Lệnh volume:

* docker volume create: Tạo mạng mới
* docker volume inspect: Xem chi tiết mạng
* docker volume ls: Hiện thị những mạng đã có
* docker volume rm: Xóa mạng
* docker volume prune: Xóa các volume rỗng

**Docker network**

Để kết nối các container trong cùng 1 mạng hoặc khác mạng với nhau.

Các câu lệnh thao tác với mạng

* ocker network create: Tạo mạng mới
* docker network inspect: Xe, chi tiết mạng
* docker network ls: Hiện thị những mạng đã có
* docker network rm: Xóa mạng
* docker network prune: Xóa đông loạt các mạng không sử dụng
* docker network connect: Tạo kết nối mạng
* docker network disconnect: Ngắt kết nối mạng

# Tổng quan về Linux

## 1. Sự khác biệt của Linux với Windows

***Về lịch sử phát triển:***

* Linux miễn phí và là mã nguồn mở ngay từ khi mới ra đời năm 1991. Linux bắt đầu như một dự án, nhưng nó đã nhanh chóng trở thành một trong những dự án mã nguồn mở lớn nhất từ trước đến nay.
* Còn đối với Windows, phiên bản Windows 1.0 của Microsoft được phát hành vào năm 1985 và không giống như Linux, nó là một sản phẩm mã nguồn đóng hoàn toàn được Microsoft bán theo chương trình cấp phép.

***Về xử lý, can thiệp vào mã nguồn:***

* Có thể chỉnh sửa, thay đổi các tính năng đối với Linux, nhưng với Windows thì không.
* Linux được cấp phép GNU Public License nên nó cho phép người dùng truy cập mã nguồn đến tận lõi của hệ điều hành.Còn đối với hệ điều hành Windows thì không, chỉ trừ khi là kỹ sư trong nhóm phát triển hệ điều hành Windows, còn không thì không có quyền truy cập vào mã nguồn này. Nó được bảo mật vô cùng cẩn thận.

***Về vấn đề bản quyền:***

* Để có quyền truy cập mã nguồn Linux thì phải được cấp phép. Cấp phép nghĩa là phần mềm sẽ có quyền được phân phối. Với giấy phép GPL của Linux, bạn có thể tự do sửa đổi OS đó, tái xuất bản nó và thậm chí bạn có thể thương mại nó, miễn là bạn cung cấp mã nguồn của nó, không giữ bí mật cho riêng mình.Với giấy phép GPL, bạn cũng có thể tải xuống bản sao của Linux và cài đặt nó trên bao nhiêu máy tùy ý.
* Còn giấy phép của Microsoft thì sẽ rất khác với giấy phép này, tức là bạn không thể sửa đổi mã nguồn vì Microsoft không bao giờ công bố mã nguồn của hệ điều hành này.
* Khi bạn mua bản quyền Windows (đối với các cá nhân) thì chỉ sử dụng giấy phép đó cho cho một máy tính duy nhất. Còn các giấy phép dành cho máy doanh nghiệp thì khác,có thể sử dụng 1 key để kích hoạt cho nhiều máy tính.

***Về ứng dụng – phần mềm:***

* Với hầu hết các bản phối của hệ điều hành Linux,có một trung tâm để cài đặt các ứng dụng. Điều này giúp dễ dàng thêm các ứng dụng mới và xóa chúng đi khi không còn dùng đến.Tính năng quản lý gói của Linux cực kỳ hữu ích vì có thể tìm kiếm và cài đặt ứng dụng trực tiếp mà không cần phải dò tìm trên mạng. Và tất cả đều miễn phí.Trước đây, Windows không có kho ứng dụng của riêng mình. Với Windows, bạn phải lên Google và tìm kiếm những phần mềm của bên thứ 3 để cài đặt.Sau đó đến quá trình tải về máy rồi chạy file thực thi .exe để tiến hành cài đặt. Sau khi ứng dụng đã được cài đặt,cũng không biết nó đã thay đổi bao nhiêu tập tin hệ thống. Nhưng kể từ phiên bản Windows 8.x, Microsoft đã có kho ứng dụng riêng của mình – nó mang tên Windows Store, mặc dù còn hạn chế so với Linux nhưng nó cũng cung cấp khá nhiều phần mềm (cả miễn phí và trả phí), đáp ứng nhu cầu cơ bản của người dùng.

***Về khả năng sử dụng:***

* Linux rất phức tạp để cài đặt nhưng có khả năng hoàn thành các tác vụ phức tạp dễ dàng hơn.
* Windows cung cấp cho người dùng một hệ thống đơn giản để vận hành nhưng sẽ mất nhiều thời gian hơn để cài đặt.

***Về đối tượng sử dụng***

***Về bảo mật:*** Cả 2 đều có tính bảo mật nhưng linux có tính bảo mật cao hơn.

***Về khả năng hỗ trợ:*** Windows là một sản phẩm thương mại hoàn toàn nên về khâu support sẽ tốt hơn.

***Về cập nhật:***

* Linux, người dùng có toàn quyền kiểm soát các bản cập nhật, chúng tôi có thể cài đặt bất cứ khi nào chúng tôi cần và sẽ mất ít thời gian hơn mà không cần khởi động lại.
* Windows, các bản cập nhật sẽ đến vào những thời điểm bất tiện như bạn đang in một bản in cho máy in nhưng đột nhiên bản cập nhật bật lên sẽ khiến người dùng bực bội và mất nhiều thời gian hơn để cài đặt.

## 2. Kiến trúc

Kiến trúc hệ điều hành Linux chia làm 3 phần:

* Kernel
* Shell
* Applications

**Thành phần:**

* **Kernel** : Đây là phần quan trọng của HĐH, phần kernel chứa các module, thư viện để quản lý và giao tiếp với phần cứng và các ứng dụng.

Dùng để quản lý phần cứng và các ứng dụng thực thi

* Linux xem mỗi thiết bị phần cứng tương đương với một tập tin
* Khi khởi động máy tính, Kernel được nạp vào trong bộ nhớ chính, và nó hoạt động cho đến khi tắt máy. Thực hiện chức năng mức thấp và chức năng mức hệ thống.
* Kernel chịu trách nhiệm thông dịch và gửi các chỉ thị tới bộ vi xử lý máy tính.
* Kernel cũng chịu trách nhiệm về các tiến trình và cung cấp các đầu vào và ra cho các tiến trình.
* **Shell**: là một chương trình có chức năng thực thi các lệnh từ người dùng hoặc từ các ứng dụng – tiện ích yêu cầu chuyển đến cho Kernel xử lý.

***Các loại shell:***

* Sh (the Bourne Shell)
* Bash(Bourne-again shell): đây là shell mặc định trên linux.
* csh (the C shell): shell được viết bằng ngôn ngữ lập trình C.
* ash (the Almquist shell).
* tsh (the TENEX C shell).
* zsh (the Z shell).

***Chức năng của Shell:***

* Thông dịch lệnh.
* Khởi tạo chương trình.
* Định hướng vào ra.
* Kết nối đường ống.
* Thao tác trên tập tin.
* Duy trì các biến.
* Điều khiển môi trường.
* Lập trình shell.
* **Applications**: Là các ứng dụng và tiện ích mà người dùng cài đặt trên Server. Ví dụ: ftp, samba, Proxy, …
* **Commands and Utilities**: Khi làm việc với HĐH Linux hầu như tất cả người dùng đều sử dụng lệnh để làm việc. Lệnh là một chương trình hoặc một script dùng để thực thi một nhiệm vụ nào đó. Lệnh được gõ sau dấu nhắc shell.

Dòng lệnh shell tổng quát có dạng như sau: command [options]

Trong đó:

* command Lệnh
* options Tùy chọn, thường bắt đầu bằng - hoặc - -.Nhiều tùy chọn có thể kết hợp bằng một ký hiệu -. Ví dụ: -lF thay vì -l -F

Dòng lệnh shell có phân biệt chữ thường và chữ hoa.

## 3. Các lệnh cơ bản

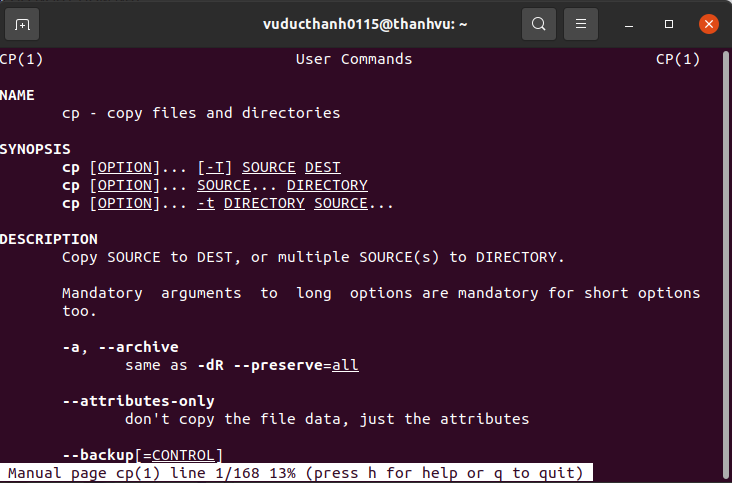
**Lệnh trợ giúp: man**

Mỗi lệnh trên Linux có rất nhiều options, mỗi options thực hiện các chức năng khác nhau, người quản trị cũng không cần nhớ hết các options của lệnh mà chỉ cần nhớ một vài options thông dụng. Để biết một lệnh có bao nhiêu options cũng như chức năng của từng options thì lệnh đầu tiên cần phải biết là lệnh: man

Cấu trúc lệnh: $ man <tên lệnh>

Ví dụ: Để xem hướng dẫn sử dụng lệnh cp (copy) có thể nhập lệnh $man cp

Để thoát khỏi man ta bấm phím “q”

[](https://github.com/vuducthanh0115/img/blob/main/anh/1_2.png)

**Các lệnh kiểm tra performance**

Lệnh xem thông tin RAM: Xem tổng dung lượng, dung lượng hiện tại đang dùng, dung lượng còn trống. có 2 lệnh đó là:

* cat /proc/meminfo

Lệnh cat: Dùng để đọc nội dung của file text  
/proc/meminfo: đây là đường dẫn (đường dẫn tuyệt đối) tới file chứa thông tin RAM có tên là meminfo

* free

các options:

* -b Hiển thị theo bytes
* -k Hiển thị theo kilobytes
* -m Hiển thị theo megabytes
* -g Hiển thị theo gigabytes
* --tera Hiển thị theo terabytes
* -h Hiển thị theo kiểu tự động gom theo khối dữ liệu

Lệnh xem thông tin CPU: cat /proc/cpuinfo

Lệnh hiển thị thông tin kernel: uname -a

options:  
-a : all information

Lệnh hiển thị phiên bản: cat /etc/redhat-release

Lệnh xem dung lượng ổ cứng: Xem dung lượng ổ cứng đã dùng và còn trống bao nhiêu: df-h

options: -h : in kích thước mà người dùng có thể đọc

Lệnh xem các tiến trình: top

Lệnh xem dung lượng của thư mục: du

options:

* -s : xuất kết quả theo summarize (tổng dung lượng)
* -h: in kích thước mà người dùng có thể đọc

Ví dụ: Xem dung lượng của thư mục /etc  
du -sh /etc

Lệnh xem tên server: hostname

Lệnh xem địa chỉ ip: ifconfig

## 4. Files and Directories

Trên linux tất cả mọi thứ đều được xem dưới dạng là file. Có 3 loại file: file thông thường (Regular files), file thư mục (Directory files), file đặc biệt (Special files).

File thông thường: một chương trình, file text, library, file nhạc …

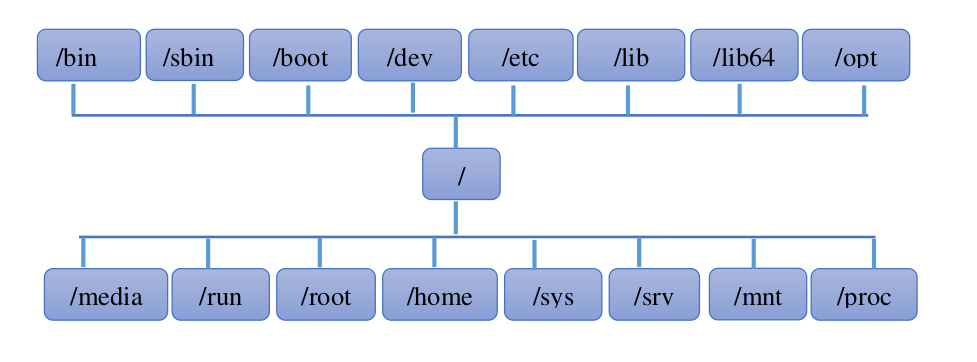
Thư mục: thành phần dùng để chứa các file khác (container).

File đặc biệt: (device, socket, pipe, symbolic links …).

* -: Regular file
* d: Directory
* l: Link
* c: Special File
* s: Socket
* p: Named Pipe
* b: Block Device

Các file ẩn thường bắt đầu bằng dấu “.”

Cấu trúc hệ thống file được bố trí theo dạng hình cây(tree) :

[](https://github.com/vuducthanh0115/img/blob/main/anh/2_2.png)

Bắt đầu là thư mục gốc “/”, sau đó là các thư mục con (hay còn gọi là nhánh): /bin, /sbin, /home, /mnt …

Mỗi thư mục con của thư mục gốc có các chức năng khác nhau.

/bin: Chứa các file binary của các tập lệnh trong Linux.

/sbin: Tương tự như /bin, nhưng là những lệnh chỉ được dùng bởi quản trị hệ thống - tương đương root user.

/boot: Chứa các thư viện cần thiết cho quá trình khởi động.

/dev: Chứa thông tin chứa các file thiết bị. Trong Linux, mỗi thiết bị đều có file đại diện và được đặt tên theo 1 Logic nhất định:

* cdrom : đĩa CDRom / DVD
* fd\* : Đĩa mềm
* hd\* : Đĩa cứng IDE
* sd\* : Đĩa cứng SCSI
* st\* : Băng từ
* tty\* : cổng giao tiếp (COM,...)
* eth\* : card ethenet

/etc: Chứa file cấu hình hệ thống và ứng dụng.

/lib: Chứa thư viện chia sẻ được dùng bởi các tiến trình, các lệnh boot, lệnh hệ thống như trong /bin và /sbin.

/lib64: Tương như như lib nhưng dành cho 64bit.

/opt: Nơi dành riêng cho các tiện ích chương trình được cài đặt.

/media: Thư mục có vai trò như đích đến của quá trình mount point. Khi gắn 1 thiết bị lưu trữ bên ngoài, để sử dụng, cần mount thiết bị này vào /media, từ đó, các thư mục, tập tin sẽ được chuyển vào đây (lúc này /media có thể coi như ảnh chiếu của thiết bị).

/run:  
/root: Thư mục home của user root.  
/home: Thư mục chứa các thư mục home của các user được tạo.

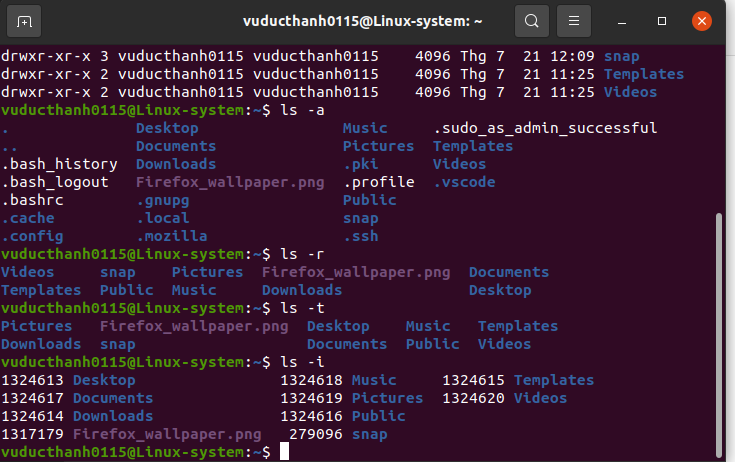
/sys:  
/srv : chứa dữ liệu, các file của các dịch vụ trên hệ thống.  
/mnt: Thư mục này được dùng để gắn các hệ thống tập tin tạm thời (mounted filesystems).  
/proc: Lưu các thông tin về tình trạng của hệ thống.

## 5. Quản lý thư mục, quản lý tập tin

Lệnh ls: dùng để xem (liệt kê) nội dung thư mục  
Cấu trúc lệnh: ls [options] [Path]

Options:

* -l: liệt kê chi tiết.
* -a: liệt kê tất cả các file ẩn
* -t: liệt kê các file được sắp xếp theo ngày sửa đổi
* -r: liệt kê các dile theo kiểu đảo ngược lại
* -i: liệt kê số node của các file

[](https://github.com/vuducthanh0115/img/blob/main/anh/3_2.png)

cd Lệnh cd: dùng để chuyển thư mục  
Cấu trúc lệnh: cd[Path]

Ví dụ: cd /etc Chuyển đến thư mục /etc.  
cd usr Chuyển vào thư mục usr là con của thư mục hiện tại.  
cd .. Chuyển lên thư mục cấp cao hơn (cha)  
cd Chuyển về thư mục home  
cd ~ Chuyển về thư mục home

Lệnh pwd: cho biết thư mục làm việc hiện tại.  
Cấu trúc lệnh: $ pwd

Lệnh passwd: đổi mật khẩu đăng nhập của user đang login.  
Cấu trúc lệnh: $ passwd

Lệnh cp: dùng để sao chép file  
Cấu trúc lệnh: cp [Options] Source Dest

Options:  
-R, -r : Sao chép toàn bộ thư mục.  
Source, Dest: Lần lượt là tên thư mục/tập tin nguồn, đích

Lệnh mv: dùng để đổi tên / di chuyển thư mục hoặc file từ nơi này sang nơi khác  
Cấu trúc lệnh: mv [options] Source Dest

Options:  
-i : Nhắc trước khi di chuyển với tập tin/thư mục đích đã có rồi.  
-f: Ghi đè khi di chuyển với tập tin/thư mục đích đã có rồi.

Lệnh mkdir: dùng để tạo thư mục  
Cấu trúc lệnh: mkdir [Options] Directory

Options:  
-p : Cho phép tạo thư mục con ngay cả khi chưa có thư mục cha.  
Directory: Tên thư mục muốn tạo.

Lệnh rmdir: dùng để xóa thư mục rỗng. Thư mục rỗng là thư mục không chứa bất kỳ thành phần nào.  
Cấu trúc lệnh: rmdir [options] directory

Options:  
-p : xóa thư mục và cả thư mục cha.  
Directory: Tên thư mục muốn xóa.

Lệnh rm: dùng để xoá file/thư mục. Lệnh này được xem là một trong những lệnh nguy hiểm của Linux.  
Cấu trúc lệnh: rm [options] file

Options:  
-f: xóa không cần hỏi  
-i: hỏi trước khi xóa  
-R, -r: xóa toàn bộ thư mục, kể cả thư mục con

Lệnh gedit: dùng để mở file với trình soạn thảo gedit.

Lệnh vi: dùng để mở trình soạn thảo thảo vi.

## 6. Vim & Nano Editor

Là trình soan thảo Terminal

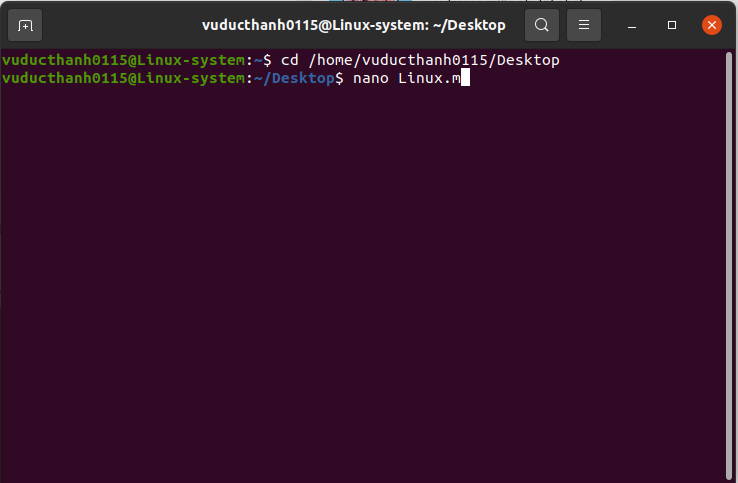
***GNU nano***

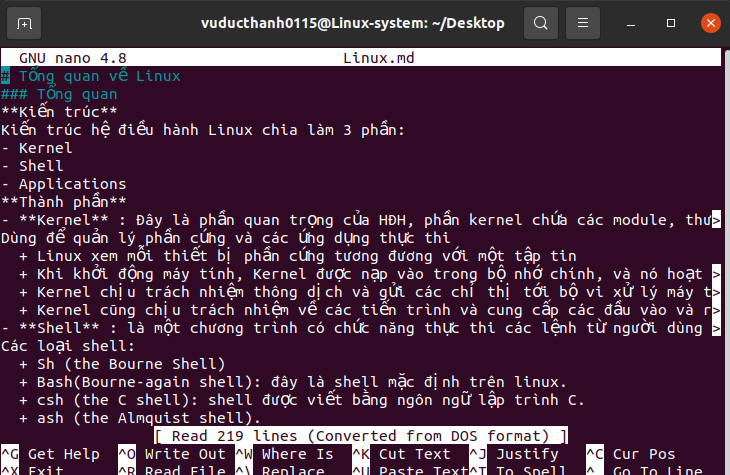
GNU nano là một trình soạn thảo văn bản dòng lệnh dễ sử dụng cho các hệ điều hành Unix và Linux. Nó bao gồm tất cả các chức năng cơ bản của một trình soạn thảo văn bản thông thường, như Syntax Highlighting, bộ đệm, tìm kiếm và thay thế văn bản, kiểm tra chính tả, mã hóa UTF-8,...

Cài đặt Nano trên Ubuntu  
sudo apt install nano -y

Các tính năng của GNU nano bao gồm:

* Hỗ trợ Autoconf
* Chức năng tìm kiếm phân biệt chữ hoa chữ thường
* Tìm kiếm và thay thế tương tác
* Khả năng tự động thụt lề
* Tùy chọn chiều rộng tab được hiển thị
* Tìm kiếm và thay thế biểu thức thông thường
* Công tắc chuyển đổi cho flag cmdline thông qua các phím meta
* Tab completion (nhấn Tab trong khi nhập lệnh, tùy chọn hoặc tên file và môi trường shell sẽ tự động hoàn thành những gì bạn đang nhập) khi đọc/ghi file
* Soft text wrapping (văn bản trông giống như đã kết thúc ở phần rìa màn hình, nhưng thực tế, nó là một dòng rất dài. Các phần tiếp theo được chỉ định bằng $)

[](https://github.com/vuducthanh0115/img/blob/main/anh/5_2.png)

[](https://github.com/vuducthanh0115/img/blob/main/anh/6_2.png)

***Vim***

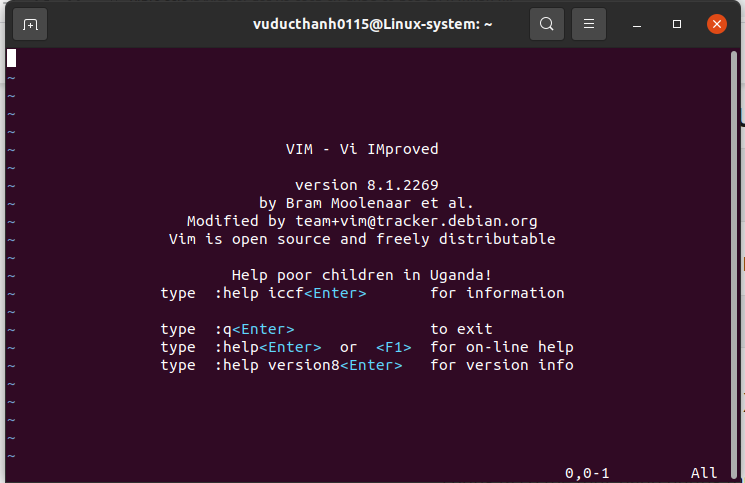
Vim - viết tắt của từ Vi IMproved là một bản sao, với một số bổ sung, của trình soạn thảo vi của Bill Joy cho Unix. Nó được viết bởi Bram Moolenaar dựa trên mã nguồn của một port của Stevie editor lên Amiga và phát hành lần đầu vào năm 1991. Vim được sử dụng rất mạnh mẽ trong CLI (command-line interface). Linux sử dụng rất nhiều file cấu hình, mình thường sẽ cần chỉnh sửa chúng và vim là một công cụ để làm điều đó.

Cài đặt Vim trên linux  
sudo apt-get install vim

Vim có lợi thế là mạnh hơn GNU nano. Vim không chỉ chứa nhiều tính năng hơn, mà bạn còn có thể tùy chỉnh chương trình với các plugin và script.

Các tính năng của Vim bao gồm:

* Các lệnh tự động
* Các lệnh bổ sung
* Đầu vào
* Giới hạn bộ nhớ cao hơn vanilla vi
* Chia nhỏ màn hình
* Khôi phục phiên
* Mở rộng tab
* Hệ thống tag
* Thêm màu cho cú pháp

[](https://github.com/vuducthanh0115/img/blob/main/anh/4_2.png)

## 7. Các biến môi trường

Biến môi trường là các đại lượng có các giá trị cụ thể. Một số biến môi trường được cung cấp các giá trị đặt trước của hệ thống và các biến khác được đặt trực tiếp bởi người dùng, tại dòng lệnh hoặc trong khi khởi động các tập lệnh khác.

Biến môi trường là một chuỗi ký tự chứa thông tin được sử dụng bởi một hoặc nhiều ứng dụng.

*Một số lệnh có sẵn cho phép liệt kê và đặt các biến môi trường trong Linux*:

* env – Lệnh cho phép bạn chạy một chương trình khác trong môi trường tùy chỉnh mà không cần sửa đổi chương trình hiện tại. Khi được sử dụng mà không có đối số, nó sẽ in một danh sách các biến môi trường hiện tại.
* printenv – Lệnh in tất cả hoặc các biến môi trường được chỉ định.
* set– Lệnh dùng để set hoặc unset biến shell. Khi được sử dụng mà không có đối số, nó sẽ in một danh sách tất cả các biến bao gồm các biến môi trường và shell và các hàm shell.
* unset – Lệnh xóa các biến shell và môi trường.
* export – Lệnh đặt các biến môi trường.

*Một số biến môi trường phổ biến nhất*:

* USER – Người dùng đăng nhập hiện tại.
* HOME – Thư mục home của người dùng hiện tại.
* EDITOR– Trình chỉnh sửa tập tin mặc định sẽ được sử dụng. Đây là trình chỉnh sửa sẽ được sử dụng khi bạn nhập edit vào terminal của bạn.
* SHELL – Đường dẫn của shell hiện tại của người dùng, chẳng hạn như bash hoặc zsh.
* LOGNAME – Tên của người dùng hiện tại.
* PATH– Một danh sách các thư mục sẽ được tìm kiếm khi thực hiện các lệnh. Khi bạn chạy một lệnh, hệ thống sẽ tìm kiếm các thư mục theo thứ tự này và sử dụng tệp thực thi đầu tiên được tìm thấy.
* LANG – Các cài đặt local hiện tại.
* TERM – Terminal emulation hiện tại.
* MAIL – Vị trí lưu trữ thư của người dùng hiện tại.

**Quản lý tiến trình**

Tiến trình - process là một khái niệm cơ bản trong bất kì một hệ điều hành nào. Một tiến trình có thể được định nghĩa là một thực thể chương trình đang được chạy trong hệ thống.  
VD mở 5 chương trình vim để thao tác với 5 file khác nhau, thì cho 5 tiến trình chạy trong hệ thống.

--> Tiến trình là một thực thể chương trình đang được thực thi. Nó là một tập hợp các cấu trúc dữ liệu mô tả đầy đủ quá trình một chương trình được chạy trong hệ thống.

Để có thể quản lý một tiến trình, kernel cần phải biết được tiến trình đó đang làm những gì. Ví dụ, kernel cần phải biết mức độ ưu tiên của tiến trình – biết tiến trình đang chạy (sử dụng CPU) hay đang bị blocked – khóa để chờ một sự kiện nào đó. Kernel cũng cần biết không gian địa chỉ nào đang được tiến trình đó sử dụng, những file nào mà tiến trình được truy cập …

**Trạng thái của tiến ttrình**

**TASK\_RRUNNING** – Trạng thái đang chạy  
Tiến trình đang được thực thi (đang sử dụng CPU) hoặc đang chờ được thực thi (chờ để được lập lịch thực hiện).

**TASK\_INTEERRUPTIBLE** – Trạng thái có thể làm gián đoạn  
Tiến trình đang được tạm dừng (đang ngủ) để chờ một số điều kiện xảy ra để thực thi tiếp. Một ngắt phần cứng được tạo ra (hardware interupt), hoặc sự giải phóng một tài nguyên hệ thống mà tiến trình đang chờ đợi để được sử dụng, hoặc chờ một tín hiệu – signal là những ví dụ về những điều kiện có thể đánh thức một tiến trình đang tạm dừng. Việc “đảnh thức” sẽ thực hiện hành động chuyển trạng thái tiến trình từ TASK\_INTERRUPTIBLE sang TASK\_RUNNING.

**TASK\_UNINTERRUPTIBLE** – Trạng thái không thể làm gián đoạn  
Trạng thái này giống trạng thái TASK\_INTERRUPTIBLE, trừ việc tiến trình trong trạng thái này không thể chuyển trạng thái khi nhận được signal. Trạng thái này hiếm khi được sử dụng, thường thì sẽ sử dụng trạng thái TASK\_INTERRUPTIBLE. TASK\_UNINTERRUPTIBLE được sử dụng trong vài điều kiện đặc trưng của tiến trình, khi tiến trình đang đợi một sự kiện gì đó mà không muốn bị ngắt, bị gián đoạn.

**TASK\_SSTOPPED** – Trạng thái dừng  
Tiến trình sẽ chuyển sang trạng thái dừng nếu nhận được một trong các tín hiệu – signal: SIGSTOP, SIGTSTP, SIGTTIN, hoặc SIGTTOU.

**TASK\_TRACED** – Trạng thái truy tìm  
Tiến trình đang thực thi thì bị dừng lại bởi một công cụ tìm lỗi – debugger. Một công cụ tìm lỗi thực hiện ptrace() system call để theo dõi một chương trình test, nó sẽ gửi các signal cho chương trình test đó, và trạng thái của chương trình test sẽ chuyển sang TASK\_TRACED.  
Tóm lại, trạng thái TASK\_TRACED được sử dụng khi các công cụ tìm lỗi muốn đặt các điểm debug, muốn dừng chương trình test lại để thay đổi giá trị đầu vào …

**EXIT\_ZZOMBIE** – Trạng thái zombie  
Điều kiện để một tiến trình chuyển sang trạng thái EXIT\_ZOMBIE là:  
Nó là tiến trình con  
Nó kết thúc khi tiến trình cha của nó vẫn đang được thực thi  
Tiến trình cha không gọi wait4() system call  
Với những điều kiện trên, khi tiến trình con kết thúc, tài nguyên mà nó sử dụng không được kernel thu hồi vì kernel nghĩ rằng tiến trình cha có thể sẽ cần dùng chúng. Điều này gây lãng phí tài nguyên hệ thống.

**EXIT\_DEAD** – Trạng thái chết  
Đây là trạng thái cuối cùng của một tiến trình. Tiến trình ở trạng thái này sẽ thực sự được kernel xóa khỏi hệ thống do tiến trình cha đã gọi wait4() system call.  
Trạng thái của một tiến trình được gán – thay đổi một cách rất đơn giản bằng việc thay đổi trường state của task\_struct.

## 8. Truyền thông giữa các máy

Là trao đổi các dữ liệu giữa các máy với nhau.

Các lệnh truyền thông:  
Terminal 1: Sender  
$write<tên người nhận>[<tên trạm cuối>]

* is not logged in
* màn hình trắng để soạn thảo
* -o: Kết thúc 1 dòng
* -ô: Kết thúc dòng cuối cùng(hết thông báo).
* crt-d: Để kết thúc nối.

Terminal 2: Receiver

Message from user1

Từ chối hoặc chấp nhận các message

* $mesg<y|n>

Mỗi người dùng đều được cấp 1 hộp thư riêng.

$ mail: hỗ trợ việc gửi mail giữa các users

* Message sau khi login: "you have mail"
* Có 2 chế độ làm việc:
* command mode
* compose mode

## 9. Lập trình Bash Shell

**Khái niệm**

*Shell script* là một chương trình được sử dụng cho nhiều mục đích khác nhau, chẳng hạn như thực thi lệnh shell, chạy nhiều lệnh cùng nhau, tùy chỉnh các tác vụ quản trị, thực hiện tự động hóa,... mà bạn thường xuyên thực hiện trên máy tính của mình.

Shell script cho phép chúng ta lập trình các lệnh theo một chuỗi và hệ thống sẽ thực thi chúng. Viết shell script cho phép bạn sử dụng các chức năng lập trình như các vòng lặp for, các câu lệnh if/then/else...

***Tiện ích của shell script***:

* Nó có thể nhận đầu vào từ người dùng, tệp, hoặc kết quả từ màn hình.
* Giúp cho chúng ta có thể tạo nhóm lệnh riêng.
* Shell script giúp chúng ta tiết kiệm thời gian.
* Có khả năng thực hiện tự động một số công việc mà bạn thường xuyên trên máy tính của mình.

Các tập lệnh được lưu trữ dưới dạng các tập tin có thể đặt tên tùy ý cho tập lệnh shell. Và nó bắt đầu với một shebang ngay dòng đầu tiên: #!/bin/bash

Tiếp theo thì nó phải là một tập tin thực thi. Để có thể phân quyền cho tập tin là thực thi thì chúng ta sử dụng lệnh chmod: chmod u+x myscript

Lệnh trên giúp cho tập tin myscript có thể thực thi được cho người dùng.

Để comment cho 1 nội dung nào đó t sử dụng # <nội dung cần comment>

**Biến**

**Biến** là một chuỗi ký tự mà chúng ta có thể gán giá trị cho chúng. Giá trị được gán có thể là một số, văn bản, hoặc bất kỳ kiểu dữ liệu nào. Shell cho phép bạn tạo ra biến, gán giá trị, và xóa chúng.

1. Đặt tên biến

Tên biến bắt buộc phải đặt theo chuẩn mà Shell đưa ra.

Tên biến chỉ chứa các ký tự từ a-z, các số từ 0-9 và ký tự gạch dưới \_. Có thể dùng chữ in hoa hoặc in thường, tuy nhiên người ta quy ước nên đặt chữ in hoa vì nó là chuẩn chung.

1. Gán giá trị cho biến

Để gán giá trị cho biến thì bạn dùng cú pháp sau: variable\_name=variable\_value

1. Truy xuất giá trị của biến

Khi bạn muốn truy xuất đến một biến thì phải dùng ký tự đô la $ đặt trước biến đó.

1. Biến chỉ đọc (giống hằng số)

Biến chỉ đọc là biến chỉ cho phép đọc, không được phép thay đổi giá trị của biến.

Để thiết lập một biến là chỉ đọc thì ta sử dụng từ khóa readonly.

1. Xóa biến

Nếu một biến không còn tác dụng gì cho chương trình thì nên xóa nó đi bằng cách sử dụng lệnh shell để xóa nó ra khỏi dành sách biến, sau khi xóa xong thì mất hoàn toàn biến đó.  
Để xóa biên thì ta dùng hàm unset, cú pháp như sau: unset variable\_name

**Có các loại biến như:**

***Local Variables***

Một biến nội bộ là một biến mà tồn tại trong trong quá trình thực thi của shell. Nó không có sẵn trong các chương trình mà được bắt đầu bởi Shell. Chúng được thiết lập tại dòng nhắc lệnh.

***Environment Variables***

Biến môi trường là giá trị động ảnh hưởng đến phần mềm và tiến trình hoạt động trên server. Biến môi trường – environment variable có trên mọi hệ điều hành và có nhiều loại khác nhau. Biến môi trường có thể được tạo, chỉnh sửa, lưu hay xóa.

***Shell Variables***

Một biến shell là một biến đặc biệt mà được thiết lập bởi Shell và được yêu cầu bởi Shell để thực thi các hàm một cách chính xác. Một số biến này là biến môi trường, trong khi số khác là biến nội bộ.

***Special Variables***

Là biến được sinh ra bởi shell, giá trị của nó sẽ được cấp phát dựa vào dữ liệu đầu vào của chương trình shell, hoặc dữ liệu của chính hệ thống Linux.

**Toán tử và toán hạng**

Trong toán học, toán tử là các dấu hoặc ký hiệu như: cộng (+), trừ (-), nhân (x), chia (:), căn bậc 2 (√), lớn (>), bé (<), bằng (=),...

Toán tử là công cụ dùng để thực hiện các phép tính hoặc chức năng với dữ liệu trong lập trình được thể hiện dưới dạng ký hiệu, biểu tượng nhằm thực hiện các biểu thức.

**Các toán tử cần bbiết**:

***Toán tử số học***

* +(phép cộng): Cộng 2 giá trị
* -(phép trừ): Trừ 2 giá trị
* \*(phép nhân): Nhân 2 giá trị, phải thêm \* vì đây là ký tự đặc biệt
* /(phép chia): Chia 2 giá trị
* %(chia 2 giá trị): Chia lấy dư
* =(phép gán): Phép gán
* ==(phép so sánh): Phép so sánh bằng
* !=(phép so sánh không bằng): Phép so sánh không bằng

Cú pháp:  
expr toán\_hạng\_1 toán\_tử toán\_hạng\_2

***Toán tử quan hệ***

Toán tử quan hệ là toán tử so sánh giá trị giữa hai biểu thức, biểu thức có thể là một giá trị đơn hoặc một biểu thức dài với nhiều phép toán trong đó.

* -eq So sánh bằng
* -ne So sánh không bằng
* -gt So sánh lớn hơn
* -lt So sánh bé hơn
* -ge So sánh lớn hơn hoặc bằng
* -le So sánh bé hơn hoặc bằng

***Toán tử Boolean***

Boolean là kiểu dữ liệu có một trong hai giá trị là đúng hoặc sai, tức true hoặc false.

***Toán tử xử lí chuỗi***

* = So sánh hai chuỗi bằng nhau hay không
* != So sánh hai chuỗi khác nhau hay không
* -z Kiểm tra chiều dài chuỗi có bằng 0 hay không (chuỗi rỗng)
* -n Kiểm tra chiều dài chuỗi có lớn hơn 0 hay không
* str Kiểm tra chuỗi str có khác rỗng hay không

***Toán tử kiểm tra file***

Dùng để kiểm tra file như kiểm tra định dạng, đường dẫn, kích thước file ...

Khi sử dụng các toán tử này bạn phải chú ý đến quyền của file có hợp lệ hay không.

* -b file Kiểm tra có phải là file khối đặc biệt
* -c file Kiểm tra có phải file là file ký tự đặc biệt
* -d file Kiểm tra file có phải là một thư mục
* -f file Kiểm tra có phải là một file thông thường
* -g file Kiểm tra file có được thiết lập group ID (SGID)
* -k file Kiểm tra file có được thiết lập sticky bit
* -p file Kiểm tra file có phải là file named pipe
* -t file Kiểm tra trạng thái của file có đang mở bởi một terminal hay không
* -u file Kiẻm tra file có được thiết lập User ID (SUID)
* -r file Kiểm tra user có quyền đọc file hay không
* -w file Kiểm tra user có quyền write trên file hay không
* -x file Kiểm tra user có quyền execute file không
* -s file Kiểm tra kích thước file có lớn hơn 0 hay không
* -e file Kiểm tra file có tồn tại hay không

**Câu lệnh điều khiển**

**Câu lệnh rẽ nhánh If else**

Lệnh If else đây là lệnh rẽ nhánh chương trình

1. Lệnh if then fi  
Cú pháp

then

# Thực hiện lệnh nếu expression true

fi

Shell sẽ kiểm tra điều kiện đầu vào expression có giá trị true hay false, nếu true thì thực thi chương trình bên trong lệnh then, sau đó kết thúc tại lệnh fi.

2. Lệnh if else fi

Nếu muốn expression trả về false sẽ thực hiện một chương trình khác thì có thể kết hợp thêm lệnh else.

Cú pháp

then

Chạy nếu expression true

else

Chạy nếu expression false

fi

Với chương trình này thì chương trình sẽ rẻ sang hai hướng và phụ thuộc vào biểu thức expression.

3. Lệnh if elif fi

Để có thể rẽ là 3 nhánh, 4 nhánh, ... bằng cách sử dụng kết hợp thêm lệnh elif.

Cú pháp

then

Chạy nếu expression1 đúng

elif [ expression 2 ]

then

Chạy nếu expression2 đúng

elif [ expression 3 ]

then

Chạy nếu expression3 đúng

else

Chạy nếu cả 3 biểu thức trên sai

fi

Có thể tạo nhiều hơn 3 nhánh và điều này phụ thuộc vào chương trình mà muốn viết.

**Lệnh case...esac**

Đây là câu lệnh rẽ nhánh có công dụng giống như if...else

Cú pháp lệnh case .. esac trong shell script

Thực tế bạn có thể sử dụng lệnh if else để xử lý theo nhiều luồng khác nhau, tuy nhiên không phải lúc nào nó cũng tốt, nhất là trường hợp tất cả các luồng đều phụ thuộc vào một giá trị. Và trong trường hợp này bạn nên sử dụng lệnh case .. esac.

Cú pháp:

pattern1)

Statement(s)

;;

pattern2)

Statement(s)

;;

pattern3)

Statement(s)

;;

\*)

Default

;;

esac

Trong đó:

* word chính là giá trị mà bạn muốn dùng để rẻ nhánh chương trình thành nhiều luồng.
* pattern1), pattern2), ... chính là các nhánh cho mỗi trường hợp.
* Default sẽ được chạy nếu không có nhánh nào ở trên phù hợp.

**Vòng lặp**

***Vòng lặp while***

Đây là vòng lặp cho phép lặp lại một nhóm lệnh cho đến khi điều kiện lặp trả về False.

Vòng lặp while được sử dụng trong trường hợp ta chưa biết trước được tổng số lần lặp.

Cú pháp vòng lặp while

Vòng lặp while có cú pháp như sau:

do

Các lệnh sẽ chạy khi command trả về true

done

Điểm quan trọng nhất của vòng lặp while đó là kết quả trả về của command:

* Nếu command trả về true thì vòng lặp sẽ được chạy.
* Nếu command trả về false thì vòng lặp kết thúc.

Có một số trường hợp các bạn chạy thuật toán bị sai dẫn đến command luôn trả về true, lúc này chương trình sẽ dẫn đến lặp vô hạn.

***Vòng lặp for***

Dùng để lặp một dãy số, một danh sách các thư mục / các file.

Cú pháp:

for { tên biến } in { danh sách }

do

# Khối lệnh

# Thực hiện từng mục trong danh sách cho đến cho đến hết

# (Và lặp lại tất cả các lệnh nằm trong "do" và "done")

done

#hoặc sử dụng for

for (( expr1; expr2; expr3 ))

do

# Lặp cho đến khi biểu thức expr2 trả về giá trị TRUE

done

***Vòng lặp until***

Vòng lặp này tương tự với while

Cú pháp:

until command

do

Các lệnh sẽ chạy khi command true

done

Trước khi lặp thì vòng lặp sẽ kiểm tra điều kiện command trả về true hay là false. Nếu trả về true thì lần lặp đó sẽ được lặp, ngược lại nếu trả về false thì vòng lặp sẽ kết thúc.

Khi vòng lặp kết thúc thì trình biên dịch sẽ nhảy xuống phía dưới lệnh done, tức là chương trình vẫn tiếp tục chạy nếu phía dưới vòng lặp vẫn còn mã code.

***Vòng lặp select***

Vòng lặp select cung cấp một cách dễ dàng để tạo một menu được đánh số từ đó người sử dụng có thể chọn lựa. Nó là hữu ích khi bạn muốn hỏi người sử dụng để chọn một hoặc nhiều mục từ một danh sách các lựa chọn.

Cú pháp:

select { tên biến } in { danh sách }

do

cac lenh de thuc thi cho từng mục trong danh sách.

done

**Hàm**

**1. Khai báo hàm**

Để khai báo hàm trong Bash shell, ta thực hiện như sau:

function fname()

{

statements;

}

hoặc

fname()

{

statements;

}

với fname là tên của hàm.

**2. Gọi hàm**

Sau khi hoàn thành việc khai báo hàm, mỗi khi cần sử dụng đến hàm, ta gọi hàm:

Đối với các hàm không có đối số $ fname

Đối với các hàm có đối số, ta thêm danh sách các đối số phía sau tên hàm như sau: $ fname arg1 arg2 với arg1, arg2 là tên của các đối số.

**3. Lấy giá trị đối số truyền vào hàm**

Sau khi truyền các đối số vào hàm ở bước trên, để có thể sử dụng chúng bên trong hàm, chúng ta sử dụng các biến cục bộ:

* $1 => Đối số thứ 1
* $2 => Đối số thứ 2
* $n => Đối số thứ n
* “$@” => Danh sách tất cả các đối số được truyền vào.
* “$\*” => In ra danh sách tất cả các đối số truyền vào dưới dạng 1 chuỗi

# Tổng quan Prometheus Server

## 1. Giới thiệu cơ bản về Prometheus

**Monitoring là gì? Khác gì với Logging và Tracing**

Monitoring là quá trình thu thập, tổng hợp và phân tích các số liệu(metrics) để cải thiện hệ thống.

Logging là hệ thống thu thập, tìm kiếm, phân tích,... log của ứng dụng

Tracing là các hệ thống theo dõi đường đi của 1 request tới hệ thống. Nó đi qua những service nào, dừng ở mỗi service bao lâu, lỗi ở step nào,...

**Prometheus là gì?**

Prometheus là một bộ công cụ giám sát và cảnh báo hệ thống mã nguồn mở ban đầu được xây dựng bởi công ty SoundCloud.

Prometheus bây giờ đã trở thành một dự án mã nguồn mở độc lập và được duy trì độc lập với bất kỳ công ty nào.

Prometheus có khả năng thu thập thông số/số liệu (metric) từ các mục tiêu được cấu hình theo các khoảng thời gian nhất định, đánh giá các biểu thức quy tắc, hiển thị kết quả và có thể kích hoạt cảnh báo nếu một số điều kiện được thảo mãn yêu cầu.

Prometheus là 100% mã nguồn mở. Có thể xem mã nguồn tại Git : <https://github.com/prometheus/prometheus/>

Phần lớn các core tính năng của Prometheus được viết bằng ngôn ngữ Go. Một số còn lại thì được viết bằng Java, Python hoặc Ruby.

Prometheus không dùng để lấy dữ liệu log, thay vì vậy thì Prometheus là dịch vụ giám sát, thu thập và xử lý dữ liệu dạng metric (thông số).

Prometheus sử dụng cơ chế pull dữ liệu từ máy chủ remote là chính, chứ không sử dụng cơ chế đợi remote push dữ liệu lên ngoại trừ trường hợp sử dụng PushGateway.

Prometheus sử dụng chương trình cảnh báo Alertmanager để xử lý và gửi cảnh báo đi.

***Một số tính năng của Prometheus***

* Mô hình dữ liệu đa chiều với chuỗi dữ liệu theo thời gian.
* Cung cấp ngôn ngữ truy vấn dữ liệu - PromQL: tổng hợp dữ liệu chuỗi thời gian trong thời gian thực.
* Triển khai đơn giản: bắt đầu từ 1 server duy nhất, có thể mở rộng theo nhu cầu.
* Dữ liệu chuỗi thời gian được thu thập qua giao thức HTTP.
* Hỗ trợ đẩy dữ liệu thông qua bên thứ 3 trung gian.
* Tự động nhận diện mục tiêu hoặc cấu hình tĩnh.
* Giao diện người dùng hỗ trợ phong phú, nhiều loại biểu đồ.
* Quản lý được trên Cloud và máy chủ vật lý. Được dùng phổ biến để giám sát các hệ thống container và microservices.

***Mô hình dữ liệu đa chiều***

Prometheus lưu trữ dữ liệu dưới dạng chuỗi thời gian time series . Chuỗi thời gian là luồng các giá trị theo timestamp của các giá trị được đánh dấu cùng một label hoặc metrics. Label chính là điều tạo nên khả năng đa chiều các dữ liệu trong Prometheus ( phân chia theo host, machine, service ...)

Ví dụ: muốn giám sát được số lượng HTTP request trên API, chúng ta có thể tạo metric api\_http\_request\_total. Để metric trở nên đa chiều ta sử dụng các label. Label chỉ là các giá trị key-value. Trong trường hợp này ta có thể thêm label method để nhận HTTP method làm giá trị trong metric mới.

***Ưu điểm và nhược điểm của Prometheus***

**Ưu điểm**

* Độ tin cậy cao. Vì được tích hợp sẵn bằng cách làm cho mỗi máy chủ Prometheus trở nên độc lập với lưu trữ cơ sở dữ liệu chuỗi thời gian cục bộ, để tránh phụ thuộc vào bất kỳ dịch vụ từ xa nào.
* Độc lập và không phụ thuộc vào bên ngoài: do đó khi hệ thống có bất kỳ sự có gì, Prometheus vẫn có thể tiến hành giám sát mà không bị gián đoạn.

**Nhược điểm**

* Hơi phức tạp nếu cần mở rộng trong trường hợp đối tượng cần giám sát tăng lên.
* Do hoạt động độc lập, nên số lượng các chỉ số giám sát bị phụ thuộc vào sức mạnh cũng như khả năng lưu trữ của máy chủ.
* Để giải quyết vấn đề tăng khả năng xử lý cho Prometheus, chúng ta có thể: Nâng cấp phần cứng máy chủ Prometheus. Hoặc giới hạn số lượng chỉ số cần giám sát.

**Tại sao nó được sinh ra?**

Do công việc Development và Operations ngày càng trở lên áp lực vì hệ thống là sự tổng hợp vô số các thành phần khác nhau và có các nhu cầu quản trị, điều hành, bảo trì,... Ví dụ trong môi trường containerized như Docker, Kubernetes, khi có vô số máy chủ chạy hàng ngàn container và có sự liên kết lẫn nhau thì việc đảm bảo cho hệ thống ổn định, vận hành trơn tru là một nhiệm vụ khá khó.

Prometheus được sinh ra để giải quyết 1 số vấn để sau:

* Giám sát hệ thống phân tán.
* Giám sát tình trạng máy chủ: CPU, memory, hdd, lỗi hệ thống, server ngừng hoạt động,...
* Giám sát các chương trình phần mềm: lỗi phần mềm, độ trễ,...
* Nhanh chóng nhận diện sự cố, nguồn gốc phát sinh, cung cấp thông tin cụ thể cho developer kịp thời xử lý.
* Tự động giám sát và đưa ra cảnh báo.

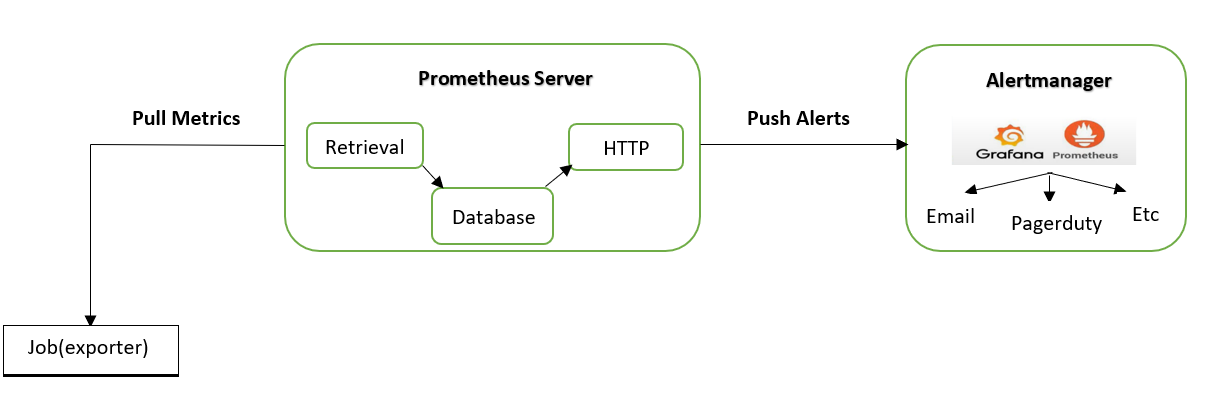
**Hệ thống phân tán là gì ?**

Hệ thống phân tán là một hệ thống bao gồm nhiều thành phần hoạt động đôc lập, trao đổi và làm việc với nhau (qua mạng) để giải quyết một vấn đề cụ thể.  
Các thành phần trong hệ thống phân tán có thể là một máy tính (máy vật lý hoặc máy ảo), container, hay bất kỳ thiết bị kết nối mạng nào, có bộ nhớ local và có thể giao tiếp với nhau thông qua mạng máy tính.

**Nó hoạt động như thế nào? Luồng hoạt động với các thành phần khác trong stack ra sao?**

***Cách Prometheus hoạt động:***

Prometheus thực hiện quá trình kéo(pull) các thông số/số liệu(metric) từ các job(exporter) được chỉ định qua kênh trực tiếp hoặc thông qua dịch vụ Pushgateway trung gian. Sau đó Prometheus sẽ kưu các dữ liệu thu thập được ở local máy chủ. Tiếp đến sẽ chạy các rule để xử lý các dữ liệu theo nhu cầu cũng như kiểm tra thực hiện các cảnh báo mà ta mong muốn.

[](https://github.com/vuducthanh0115/img/blob/main/anh/img1.png)

Đầu tiên thì job(exporter) sẽ giúp lấy các thông số sau đó theo định kì 1 thời gian nhất định thì Prometheus sẽ thu thập bằng cách kéo các thông số đã được lấy ra về và lưu vào trong database. Từ dữ liệu của database Prometheus có thể xây dựng những biểu đồ trên trang web để ta có thể dễ dàng theo dõi và Prometheus hỗ trợ việc theo dõi và cảnh báo giúp ta giám sát hệ thống. Chúng ta có thể sử dụng Grafana hoặc Prometheus để gửi cảnh báo qua mail, pagerduty, sky, ...

**Exporters**

Exporter về cơ bản như là 1 loại hình dịch vụ (Services) có khả năng thu thập các chỉ số từ đối tượng cần giám sát, sau đó chuyển đổi thành định dạng mà Prometheus có thể hiểu. Exporter cũng cung cấp 1 địa chỉ Endpoint đúng theo yêu cầu mà Prometheus cần.

Danh sách exporter trải dài từ cơ sở dữ liệu (MongoDB, MySql, PostgreSQL,...), phần cứng (NVIDIA GPU, Fortigate, Netgear Router, IoT Edison,...), lưu trữ (Hadoop HDFS FSImage, ScaleIO, GPFS, Gluster,...), máy chủ web (Apache, Nginx, HAProxy,...), hoặc là các nền tảng đám mây như AWS, Cloudflare, DigitalOcean,...

**Exporter lấy metrics rồi đẩy ra đâu?**

Exporter giúp thu thập các chỉ số từ đối tượng cần giám sát, sau đó chuyển đổi thành định dạng mà Prometheus có thể hiểu. Sau đó đẩy các metrics về service của Prometheus.

**Client Libraries**

Prometheus cung cấp bộ thư viện được viết bằng rất nhiều ngôn ngữ lập trình như Go, Java, NodeJS, Ruby, C++, C#,...

Thư viện này giúp chúng ta tạo ra endpoint “/metrics” theo nhu cầu riêng, và định dạng dữ liệu theo chuẩn mà Prometheus có thể hiểu và xử lý.

**Endpoint**: nguồn dữ liệu của các chỉ số (metric) mà Prometheus sẽ đi lấy thông tin.

**Push Gateway Prometheus**: sử dụng để hỗ trợ các job có thời gian thực hiện ngắn (tạm thời). Đơn giản là các tác vụ công việc này không tồn tại lâu đủ để Prometheus chủ động lấy dữ liệu. Vì vậy là mà các dữ liệu chỉ số (metric) sẽ được đẩy về Push Gateway rồi đẩy về Prometheus Server.

## 2. Các khái niệm cơ bản của Prometheus

**Data model**

Prometheus về cơ bản lưu trữ tất cả dữ liệu dưới dạng time series. Time series là luồng các giá trị theo timestamp của các giá trị được đánh dấu cùng một label hoặc metrics.

***Metric names and labels***

Mỗi time series được xác định duy nhất bởi tên metric name và một bộ các cặp key-value được gọi là labels(nhãn).

Metric name chỉ định các thông số của của hệ thống. Vd: http\_requests\_total - tổng số request HTTP đã nhận). Nó có thể chứa các chữ cái và chữ số ASCII cũng như dấu gạch dưới và dấu hai chấm. Nó phải được match với [a-zA-Z\_:][a-zA-Z0-9\_:]\*.

Labels: bất kỳ sự kết hợp nhãn đã cho nào cho cùng một metric name xác định particular dimensional instantiation of that metric( Vd: tất cả HTTP requests đã sử dụng phương thức POST tới /api/tracks xử lý). Ngôn ngữ truy vấn cho phép lọc và tổng hợp dựa trên các labels này. Khi thay đổi giá tri label, bao gồm cả thêm hoặc xóa label sẽ tạo ta một time series mới.

Ví dụ:

api\_http\_requests\_total{method="POST", handler="/messages"}

Một time series có metric name là api\_http\_requests\_total và lable method="POST", handler="/messages

**Metric types**

***Có 4 kiểu metric được sử dụng trong prometheus:***

* **Counter** là một bộ đếm tích lũy, được đặt về 0 khi restart. Ví dụ, có thể dùng counter để đếm số request được phục vụ, số lỗi, số task hoàn thành,... Không sử dụng cho gía trị có thể giảm như số tiến trình đang chạy. Trong trường hợp này, ta có thể sử dụng gauge.
* **Gauge** đại diện cho số liệu duy nhất, nó có thể lên hoặc xuống. Nó thường được sử dụng cho các giá trị đo.
* **Histogram** lấy mẫu quan sát (thường là những thứ như là thời lượng yêu cầu, kích thước phản hồi). Nó cũng cung cấp tổng của các giá trị đó.
* **Summary** tương tự histogram, nó cung cấp tổng số các quan sát và tổng các giá trị đó, nó tính toán số lượng có thể cấu hình qua sliding time window.

**Jobs và Instances**

Trong prometheus quy định một endpoint mà chúng ta có thế thể scrape được gọi là một **instance**.

Tập hợp của các instances có cùng một mục đích được gọi là một **job**.

Ví dụ:  
job: api-server  
instance 1: 1.2.3.4:5670  
instance 2: 1.2.3.4:5671  
instance 3: 5.6.7.8:5670  
instance 4: 5.6.7.8:5671

## 3. Kiến trúc của Prometheus stack và các thành phần cơ bản.

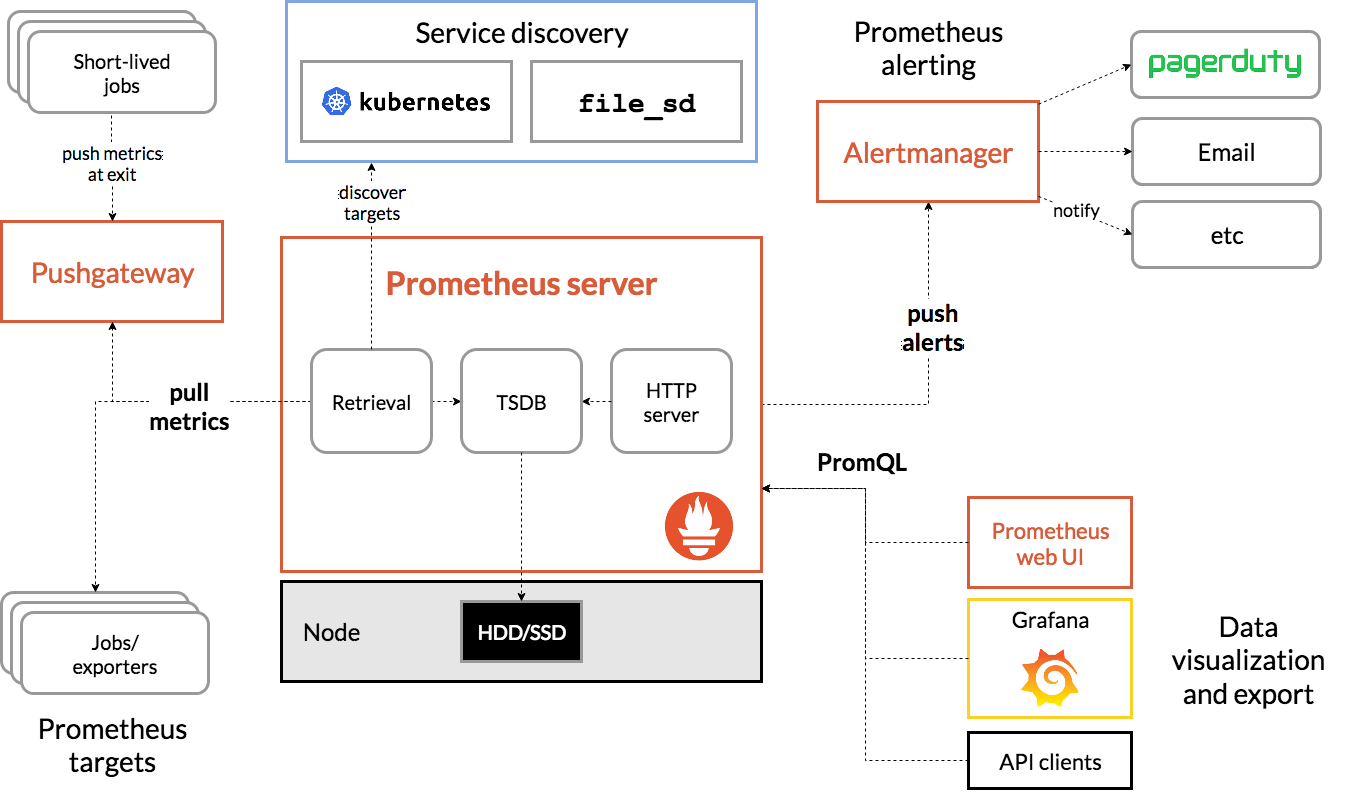
**Các thành phần trong promtheus**

Hệ sinh thái của giải pháp bao gồm nhiều thành phần trong đó có một vài thành phần là tùy chọn ( tức là có thể có hoặc không )

* **Prometheus server** để scrapes và lưu trữ dữ liệu
* **Client libraries** hỗ trợ các ngôn ngữ lập trình khác tương tác với prom
* **Push gateway** là một gateway trung gian hỗ trợ short-lived jobs
* Các **exporter** để giám sát các services như HAProxy, StatsD, Graphite, ...
* Một **alertmanager** phụ vụ cho việc cảnh báo

**Kiến trúc của Prometheus**

Sơ đồ này minh họa kiến ​​trúc của Prometheus và một số thành phần trong hệ sinh thái của nó:

[](https://camo.githubusercontent.com/2cb8b2d19c897fef44f30b6a0c8ba5d0c6ea7ffde700cdd84d3c053ad0406815/68747470733a2f2f70726f6d6574686575732e696f2f6173736574732f6172636869746563747572652e706e67)

Cách làm việc của promtheus được mô tả như sau:

* Metrics sẽ thu thập thông qua exporter
* Prometheus sẽ tự discovery ra target của các jobs hoặc trong file cấu hình thủ công để pull metrics về
* Sau đó prom server sẽ xử lý và lưu trữ xuống database của mình
* Cùng lúc đó AlertManager sẽ tiến hành kiểm tra điều kiện nếu match với các rules đã được người quản trị định nghĩa thì sẽ gửi cảnh báo thông qua các kênh mail, slack, ...
* Prom UI là nơi hiển thị graph của các metrics đẩy về hệ thống. Ngoài ra garafa cũng hỗ trợ data source prom.

Prom được viết bằng ngôn ngữ Go và được thiết kế theo kiến trúc microservices. Các thầnh phần của nó giao tiếp với nhau thông qua api

**Trường hợp nào sử dụng Prometheus ?**

Prometheus hoạt động tốt để ghi các time series data dạng số. Nó phù hợp với việc giám sát các máy chủ cũng như các kiến trúc highly dynamic service-oriented.

Đối với microservice, nó hỗ trợ thu thập và truy vấn dữ liệu đa chiều.

**Trường hợp nào không nên sử dụng?**

Prometheus là đáng tin cậy. Có thể xem thống kê nào có sẵn về hệ thống của chúng ta, ngay cả trong điều kiện lỗi. Nếu chúng ta cần độ chính xác 100%, ví dụ cho mục đích thanh toán, Prometheus không phải là một lựa chọn tốt vì nó không chi tiết và đầy đủ.

## 4. Cấu hình Prometheus

**Cấu hình Prometheus server**

Xem phần thực hành phía trên

**Recording rules**

Các recording rules cho phép ta định tuyến trước các biểu thức thường xuyên cần thiết hoặc expensive về mặt tính toán và lưu kết quả của chúng dưới dạng một chuỗi time series mới. Truy vấn kết quả được tính trước sau đó thường sẽ nhanh hơn nhiều so với thực hiện biểu thức ban đầu mỗi khi cần thiết. Điều này đặc biệt hữu ích cho bảng điều khiển, cần truy vấn cùng một biểu thức lặp đi lặp lại mỗi khi chúng làm mới.

Các recording rules và alerting rules tồn tại trong 1 nhóm rule. Các rule trong một nhóm được chạy tuần tự trong một khoảng thời gian đều đặn, với cùng một evaluation time. Tên của các recording rules phải là metric name hợp lệ. Tên của các alerting rules phải là giá trị label(nhãn) hợp lệ.

Cú pháp của tệp quy tắc là:

groups:

[ - <rule\_group> ]

Một tệp quy tắc ví dụ đơn giản :

groups:

- name: example

rules:

- record: job:http\_inprogress\_requests:sum

expr: sum by (job) (http\_inprogress\_requests)

<rule\_group>

name: <string>

[ interval: <duration> | default = global.evaluation\_interval ]

rules:

[ - <rule> ... ]

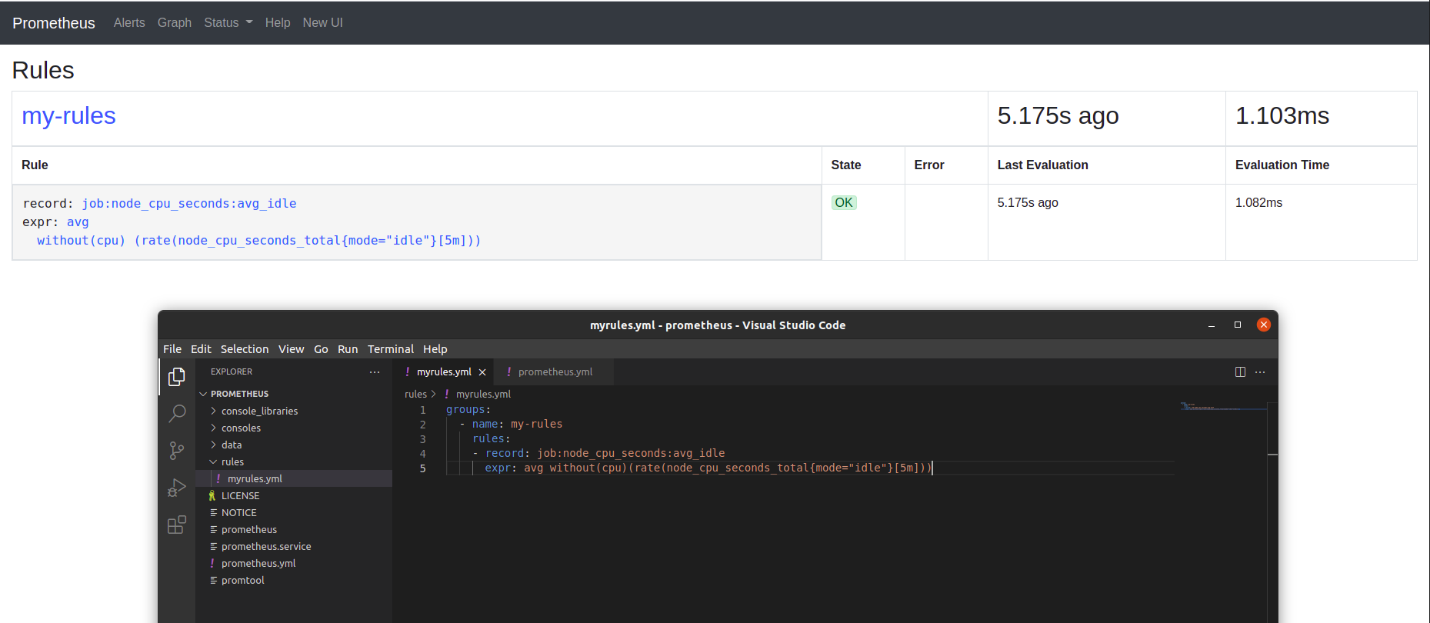
Cú pháp cho recording rules là:

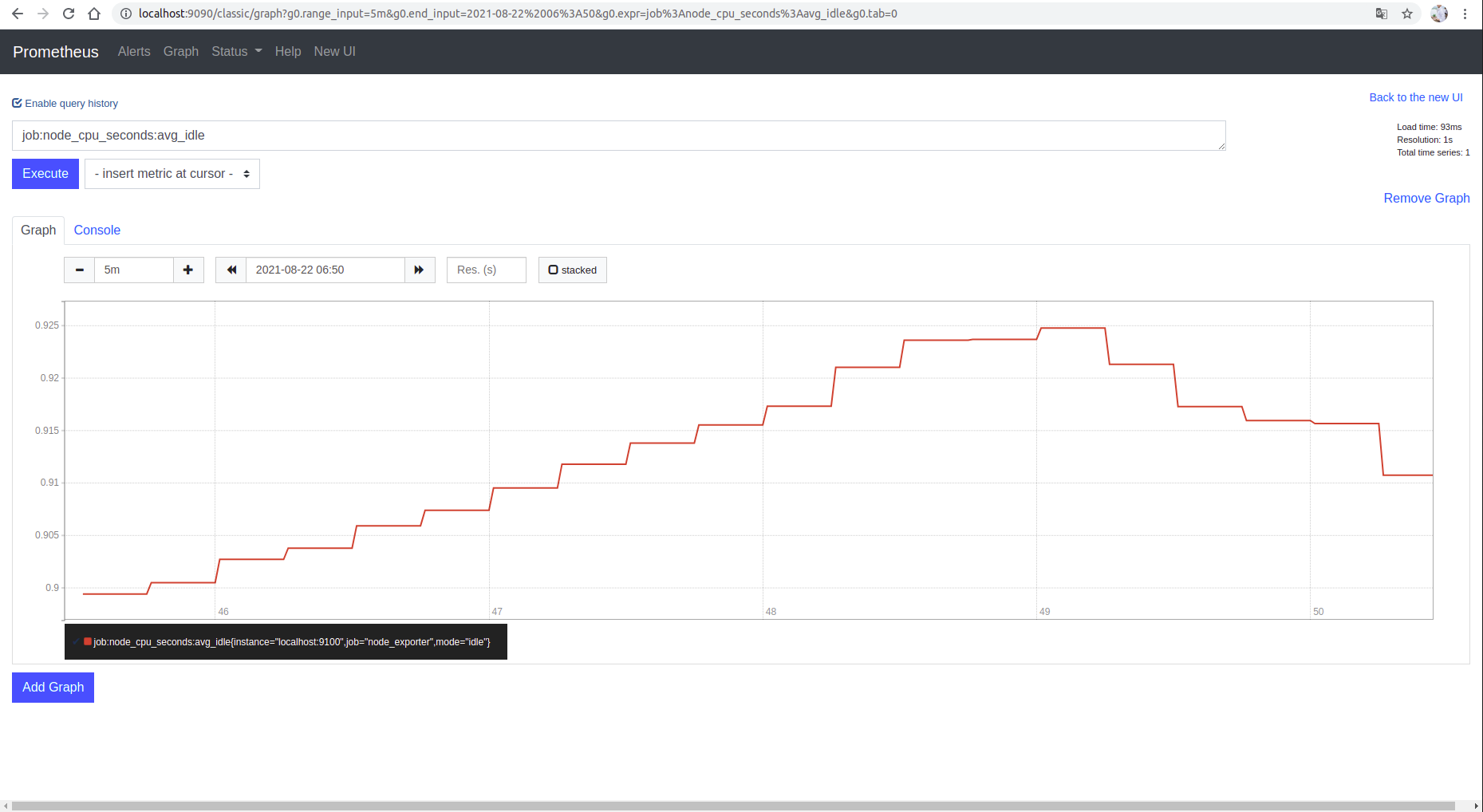
record: <string>

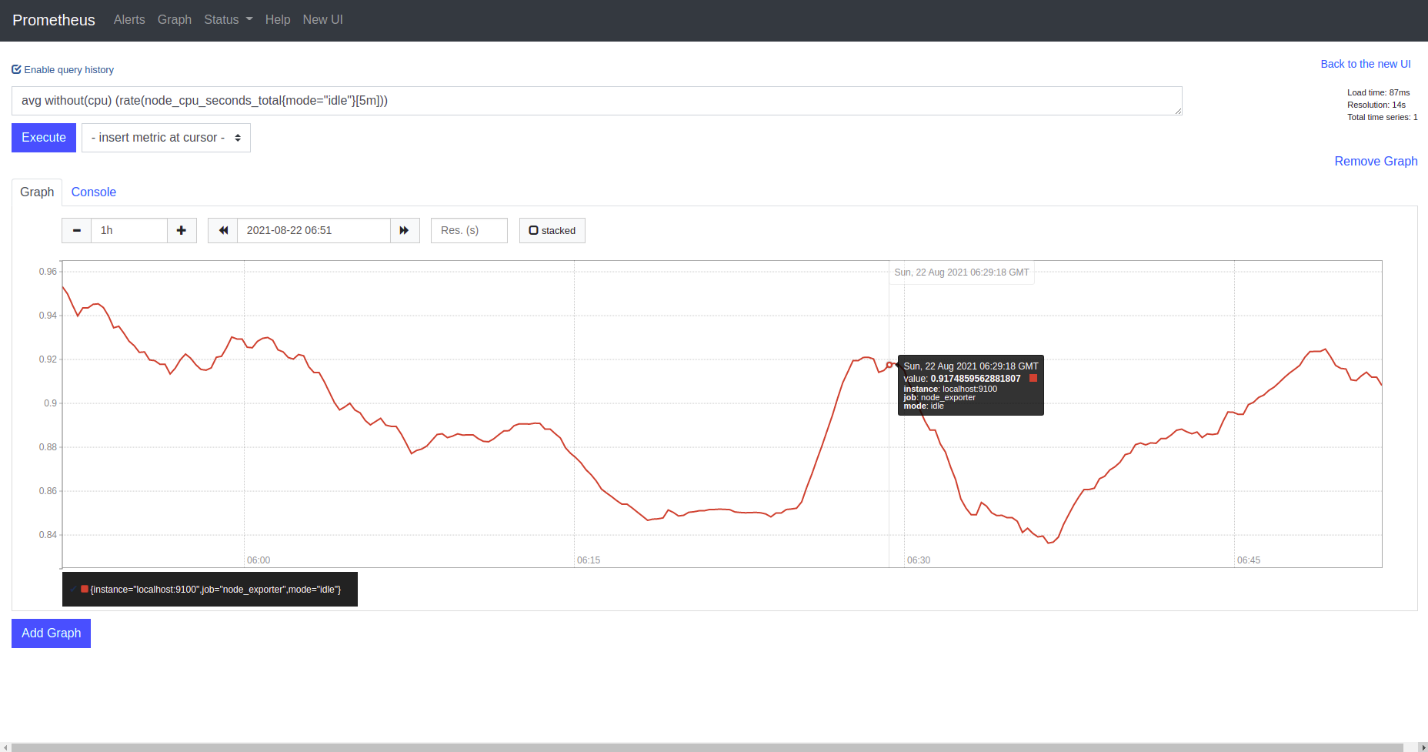
expr: <string>

labels:

[ <labelname>: <labelvalue> ]

[](https://github.com/vuducthanh0115/Anh/blob/master/Learn%20about%20GIT/recording%20rules.png)

[](https://github.com/vuducthanh0115/Anh/blob/master/Learn%20about%20GIT/recording%20rules_2.png)

[](https://github.com/vuducthanh0115/Anh/blob/master/Learn%20about%20GIT/recording%20rules_3.png)

**Alerting rules**

Cho phép ta xác định các điều kiện cảnh báo dựa trên biểu thức ngôn ngữ của Prometheus và gửi thông báo về firing alerts đến dịch vụ bên ngoài.

Xác định các alerting rules:  
Các alerting rules được cấu hình trong Prometheus giống như các recording rules.

Một tệp quy tắc ví dụ với cảnh báo sẽ là:

groups:

- name: example

rules:

- alert: HighRequestLatency

expr: job:request\_latency\_seconds:mean5m{job="myjob"} > 0.5

for: 10m

labels:

severity: page

annotations:

summary: High request latency

**Templating**

Prometheus hỗ trợ templating trong các chú thích và labels của cảnh báo, cũng như trong các trang bảng điều khiển được cung cấp. Template có khả năng chạy các truy vấn dựa trên cơ sở dữ liệu cục bộ, lặp qua dữ liệu, sử dụng các điều kiện, định dạng dữ liệu, ...

## 5. Tìm hiểu về PromQL (Operators, Functions... + Example)

Prometheus cung cấp ngôn ngữ truy vấn được gọi là PromQL( viết tắt bởi Promtheus Query Language) cho phép người dụng select và tổng hợp time series data theo thời gian thực. Kết quả của một biểu thức có thể được hiển thị dưới dạng biểu đồ được xem dưới dạng dữ liệu bảng trên dashboard của Prom hoặc từ các ứng dụng thứ ba thông qua HTTP API.

***Expression language data types:***

**Instant vector**: Truy vấn một chuỗi các time series có dùng một timestamp

VD: Nếu query http\_request\_count thì kết quả có dạng như sau: http\_request\_count{status=“200”} 20 http\_request\_count{status=“404”} 3 http\_request\_count{status=“500”} 5

**Range Vector**: Truy vấn một chuỗi các time series trong khoảng thời gian

VD: Nếu query http\_request\_count[5m] thì kết quả: http\_request\_count{status=“200”} **Scalar**: 1 dấu phảy động đơn giản

**String** : 1 chuối giá trị đơn giản

***Literals***

**String literals**

Có thể được chỉ định là chuỗi kí tự trong dấu ngoặc kép, ngoặc đơn hoặc or backticks.  
Đối với những kí tự đặc biệt cần thêm dấu . Vd như \n, \t.

**Float literals**

Các giá trị float được viết dưới dạng chữ số [-](https://github.com/vuducthanh0115/Documents/blob/main/digits)[.(digits)].

VD: -2.43

***Time series Selectors***

**Instant vector selectors**

http\_requests\_total  
Select tất cả time series của metric name http\_requests\_total  
hoặc  
http\_requests\_total{job="prometheus",group="canary"}

Select time series metric name là http\_requests\_total có job lable là prometheus và group lable là canary

Ngòai ra ta còn có thể sử dụng các biểu thức chính quy:  
= : Chọn nhãn chính xác so với string khai báo.  
!= : Ngược lại với =  
=~ : Select lable regex-match với chuỗi khai báo.  
!~ : Ngược lại với =~

***Range Vector Selectors***

Thời lượng được chỉ định là một số được biểu diễn trong [], theo sau là một trong các đơn vị sau:

s - seconds  
m - minutes  
h - hours  
w - weeks  
y - years  
VD select tất cả các giá giá trị trong 5 phút gần đây của tất cả time series metric name http\_requests\_total và job lable là prometheus

http\_requests\_total{job="prometheus"}[5m]

***Offset modifier***

Ví dụ: biểu thức sau đây trả về giá trị của http\_quests\_total 5 phút trong quá khứ so với thời gian truy vấn hiện tại:

http\_requests\_total offset 5m

**Operators**

Toán tử số học :  
+ (phép cộng)  
- (phép trừ)  
\* (phép nhân)  
/ (phép chia lấy phần nguyên)  
% (phép chia lây phần dư)) ^ (lũy thừa)

Toán tử so sánh:  
== (so sánh bằng)  
!= (so sánh khác)  
> (lớn hơn)  
< (nh hơn)  
>=(lớn hơn hoặc bằng)  
<=(nhỏ hơn hoặc bằng)

Toán tử logic: and, or, unless

One to One vector matching: Phương pháp này cố gắng tìm tất cả các cặp phần tử duy nhất từ ​​mỗi vectơ. Các mẫu được chọn hoặc loại bỏ khỏi vectơ kết quả dựa trên các từ khóa “ignoring” và “on”

Many to one / One to many vector matching: các mẫu được chọn bằng cách sử dụng các từ khóa như “group\_left” hoặc “group\_right”.

Toán tử tổng hợp :  
sum (calculate sum over dimensions) min (select minimum over dimensions) max (select maximum over dimensions) avg (calculate the average over dimensions) group (all values in the resulting vector are 1) stddev (calculate population standard deviation over dimensions) stdvar (calculate population standard variance over dimensions) count (count number of elements in the vector) count\_values (count number of elements with the same value) bottomk (smallest k elements by sample value) topk (largest k elements by sample value)

**Functions**

Trong Prometheus query language, chúng ta có thể sử dụng các hàm được xây dựng sẵn để tạo ra các giá trị mong muốn  
Một số hàm có các đối số mặc định. Ví dụ: year(v=vector(time()) instant-vector).  
Nghĩa là, nếu nó có một đối số v nhận giá trị là một instance vector, nếu nó không được cung cấp thì sẽ nhận giá trị của biểu thức vector(time())

**abs()**

abs(v instant-vector) trả về giá trị tuyệt đối của input

**absent()**

absent(v instant-vector) trả về vector rỗng nếu

**ceil()**

ceil(v instant-vector) làm tròn các giá trị của các phần tử v lên số nguyên gần nhất

**changes()**

changes(v range-vector) trả về số lần giá trị của nó đã thay đổi trong phạm vi thời gian đã cho dưới dạng vectơ tức thời.

**clamp(), clamp\_max(), clamp\_min()**  
clamp(v instant-vector, min scalar, max scalar) clamp\_max(v instant-vector, max scalar) clamp\_min(v instant-vector, min scalar)  
clamps các giá trị mẫu của tất cả các phần tử trong đó v có điều kiện

**day\_of\_month()**

day\_of\_month(v=vector(time()) instant-vector)trả về ngày trong tháng cho mỗi thời điểm nhất định trong UTC. Giá trị trả về là từ 1 đến 31.

**day\_of\_week()**

day\_of\_week(v=vector(time()) instant-vector)trả về ngày trong tuần cho từng thời điểm nhất định theo giờ UTC. Các giá trị trả về là từ 0 đến 6, trong đó 0 có nghĩa là Chủ nhật, v.v.

**days\_in\_month()**

days\_in\_month(v=vector(time()) instant-vector)trả về số ngày trong tháng cho mỗi khoảng thời gian nhất định theo giờ UTC. Giá trị trả về là từ 28 đến 31.

**delta()**

delta(v range-vector)tính toán sự khác biệt giữa giá trị đầu tiên và giá trị cuối cùng của mỗi phần tử time series trong một khoảng v, trả về một vectơ tức thời với các delta đã cho và các labels tương đương.

VD trả về sự khác biệt về nhiệt độ CPU từ bây giờ đến 2 giờ trước:

delta(cpu\_temp\_celsius{host="zeus"}[2h])  
Notes: delta chỉ nên sử dụng với đồng hồ đo.

**deriv()**

deriv(v range-vector)tính toán đạo hàm trên giây của time series trong một khoảng v, sử dụng simple linear regression .  
Notes: deriv chỉ nên sử dụng với đồng hồ đo.

**exp()**

exp(v instant-vector)tính hàm số mũ cho tất cả các phần tử trong v.  
Các trường hợp đặc biệt là:  
Exp(+Inf) = +Inf  
Exp(NaN) = NaN

**floor()**

floor(v instant-vector)làm tròn xuống các giá trị mẫu của tất cả các phần tử v .

**histogram\_quantile()**

histogram\_quantile(φ scalar, b instant-vector)tính toán lượng tử φ (0 ≤ φ ≤ 1) từ các nhóm bcủa biểu đồ .

Sử dụng hàm rate() để chỉ định cửa sổ thời gian cho phép tính lượng tử.

**holt\_winters()**

holt\_winters(v range-vector, sf scalar, tf scalar)tạo ra một giá trị smoothed cho time series dựa trên khoảng của v. sf và tf nằm trong khoảng từ 0 đến 1.  
Notes:holt\_winters chỉ nên sử dụng với đồng hồ đo.

**hour()**

hour(v=vector(time()) instant-vector)trả về giờ trong ngày cho mỗi thời gian nhất định trong UTC. Giá trị trả về là từ 0 đến 23.

**idelta()**

idelta(v range-vector)tính toán sự khác biệt giữa hai mẫu cuối cùng trong khoảng của v, trả về một vectơ tức thời với các delta đã cho và các labels tương đương.  
Notes: idelta chỉ nên sử dụng với đồng hồ đo.

**increase()**

increase(v range-vector)tính toán sự gia tăng của chuỗi thời gian trong khoảng v.

VD trả về số lượng yêu cầu HTTP được đo trong 5 phút qua, trên mỗi time series trong khoảng v :

increase(http\_requests\_total{job="api-server"}[5m])  
Notes: increasechỉ nên sử dụng với bộ đếm.

**irate()**

irate(v range-vector)tính toán tốc độ tăng tức thì trên giây của time series trong khoảng v. Điều này dựa trên hai điểm dữ liệu cuối cùng.

VD trả về tốc độ mỗi giây của các yêu cầu HTTP tìm kiếm trở lại 5 phút cho hai điểm dữ liệu gần đây nhất, trên mỗi chuỗi thời gian trong khoảng v:

irate(http\_requests\_total{job="api-server"}[5m])  
Notes: iratechỉ nên sử dụng khi vẽ đồ thị bộ đếm biến động, chuyển động nhanh.

**label\_join()**

Đối với mỗi khoảng thời gian trong v, label\_join(v instant-vector, dst\_label string, separator string, src\_label\_1 string, src\_label\_2 string, ...) kết hợp tất cả các giá trị của tất cả các thời gian đang src\_labelssử dụng separator và trả về time series với labels dst\_labelchứa giá trị được kết hợp.

VD: label\_join(up{job="api-server",src1="a",src2="b",src3="c"}, "foo", ",", "src1", "src2", "src3")

**label\_replace()**

Đối với mỗi time series trong v, label\_replace(v instant-vector, dst\_label string, replacement string, src\_label string, regex string) đối sánh biểu thức chính quy regex với giá trị của labels src\_label. Nếu nó khớp, giá trị của labels dst\_label trong các time series được trả về sẽ là phần mở rộng replacement cùng với các labels ban đầu trong đầu vào.

VD:  
label\_replace(up{job="api-server",service="a:c"}, "foo", "$1", "service", "(.\*):.\*")

**ln()**

ln(v instant-vector)tính lôgarit tự nhiên cho tất cả các phần tử trong v.  
Các trường hợp đặc biệt là:  
ln(+Inf) = +Inf  
ln(0) = -Inf  
ln(x < 0) = NaN  
ln(NaN) = NaN

**log2()**

log2(v instant-vector)tính toán lôgarit nhị phân cho tất cả các phần tử trong v.

**log10()**

log10(v instant-vector)tính toán lôgarit thập phân cho tất cả các phần tử trong v.

**minute()**

minute(v=vector(time()) instant-vector)trả về phút cho mỗi thời gian nhất định trong UTC. Giá trị trả về là từ 0 đến 59.

**month()**

month(v=vector(time()) instant-vector)trả về tháng trong năm cho mỗi thời điểm nhất định trong UTC. Các giá trị được trả về là từ 1 đến 12, trong đó 1 có nghĩa là tháng 1,...

**predict\_linear()**

predict\_linear(v range-vector, t scalar) dự đoán giá trị của time series t(s) kể từ bây giờ, dựa trên khoảng của v.  
Notes: predict\_linear chỉ nên sử dụng với đồng hồ đo.

**rate()**

rate(v range-vector) tính toán tốc độ tăng trung bình trên giây của chuỗi thời gian trong khoảng v.

VD:  
rate(http\_requests\_total{job="api-server"}[5m])  
Notes: rate chỉ nên sử dụng với bộ đếm. Nó phù hợp nhất để cảnh báo và vẽ đồ thị cho các bộ đếm chuyển động chậm.

**resets()**

Đối với mỗi time series đầu vào, resets(v range-vector) trả về số lượng bộ đếm được đặt lại trong khoảng thời gian đã cho dưới dạng vectơ tức thời. Bất kỳ sự giảm giá trị nào giữa hai mẫu liên tiếp được hiểu là sự thiết lập lại bộ đếm.  
Notes: resets chỉ nên sử dụng với bộ đếm.

**round()**

round(v instant-vector, to\_nearest=1 scalar) làm tròn các giá trị.

**scalar()**

Cho một vectơ đầu vào một phần tử, scalar(v instant-vector) trả về giá trị mẫu của phần tử đơn đó dưới dạng một đại lượng vô hướng. Nếu vectơ đầu vào không xác định thì scalar sẽ trả về NaN.

**sgn()**

sgn(v instant-vector) trả về một vectơ với tất cả các giá trị mẫu được chuyển đổi 1 nếu v dương, -1 nếu v âm và 0 nếu v bằng không.

**sort()**

sort(v instant-vector) trả về các phần tử vectơ được sắp xếp theo giá trị mẫu của chúng, theo thứ tự tăng dần.

**sort\_desc()**

Sắp xếp theo thứ tự giảm dần.

**sqrt()**  
sqrt(v instant-vector)tính căn bậc hai của tất cả các phần tử trong v.

**vector()**  
vector(s scalar) trả về scalar s dưới dạng vectơ không có labels.

**year()**

year(v=vector(time()) instant-vector) trả về năm cho mỗi thời điểm nhất định trong UTC.

**\_over\_time()**

Các hàm sau cho phép tổng hợp từng chuỗi của một vectơ phạm vi nhất định theo thời gian và trả về một vectơ tức thì với kết quả tổng hợp theo từng chuỗi:

* avg\_over\_time(range-vector): the average value of all points in the specified interval.
* min\_over\_time(range-vector): the minimum value of all points in the specified interval.
* max\_over\_time(range-vector): the maximum value of all points in the specified interval.
* sum\_over\_time(range-vector): the sum of all values in the specified interval.
* count\_over\_time(range-vector): the count of all values in the specified interval.
* quantile\_over\_time(scalar, range-vector): the φ-quantile (0 ≤ φ ≤ 1) of the values in the specified interval.
* stddev\_over\_time(range-vector): the population standard deviation of the values in the specified interval.
* stdvar\_over\_time(range-vector): the population standard variance of the values in the specified interval.
* last\_over\_time(range-vector): the most recent point value in specified interval.
* present\_over\_time(range-vector): the value 1 for any series in the specified interval.

## 6. Tìm hiểu về Labels

**Labels là gì và để làm gì?**

Labels: bất kỳ sự kết hợp nhãn đã cho nào cho cùng một metric name xác định particular dimensional instantiation of that metric( Vd: tất cả HTTP requests đã sử dụng phương thức POST tới /api/tracks xử lý) Ngôn ngữ truy vấn cho phép lọc và tổng hợp dựa trên các labels này. Khi thay đổi giá tri label, bao gồm cả thêm hoặc xóa label sẽ tạo ta một time series mới.

Sử dụng labels để phân biệt các đặc điểm của đối tượng đang được đo:

api\_http\_requests\_total - phân biệt các loại yêu cầu: operation="create|update|delete"

api\_request\_duration\_seconds - phân biệt các giai đoạn yêu cầu: stage="extract|transform|load"

Không đặt tên nhãn trong tên chỉ số, vì điều này dẫn đến dư thừa và sẽ gây nhầm lẫn nếu các nhãn tương ứng được tổng hợp lại.

**Instrumentation & Target labels**

Phần khả năng về tạo công cụ đo đạc là một lơi thế rất lớn của Prometheus. Prometheus cung cấp khả năng đo đạc (lấy các thông số ) bằng các công cụ xuất trực tiếp (node, database) hoặc các thư viện ngôn ngữ như: Python, go, Ruby ...

Cách tiếp cận sử dụng Instrumentation

Khi instument, nên cân nhắc xem là instrument service hoặc libraries (thư viện )

**Service Instumentation:**

online-serving systems: Các dịch vụ có người dùng hoặc service khác đợi phản hồi: web server, databases ..

Thông thường các dịch vụ này thường có các metrics theo nguyên tắc RED ( Request, Error, Duration ) offline-serving systems: Các dịch vụ này không cần phải phản hồi ngay lập tức. Chúng thường tham một hàng đợi công việc: xử lý log . Cho mỗi trạng thái nên có metrics về: lượng công việc trong hàng đợi, công việc đang tiến hành, tốc độ xử lý tiến trình,, lỗi xảy ra. USE method ( Utilisation, Saturation, Error,)

Utilisation: Lượng công việc đang phải xử lý ( processing)

Saturation: Lượng công việc còn lại đang đợi ( waiting )

Error: Lỗi

batch jobs: cũng giống như offline-serving nhưng batch-job chạy theo chu kí còn offline-serving chạy liên tục. Do batch jobs không chạy liên tục nên scrape không hoạt động tốt lắm, nên kỹ thuật nên dùng là Pushgateway. Khi kết thúc batch-jobs nên nên có metrics: how long it took to run ? how long each stage of job took ? time job succeed ? Alert when failed .

**Library instrumentation**

Libraries liên quan tới các mini services: functions, method, cho phép tính toán các thông tin ở mức độ backend

**Relabel**

Relabeling là công cụ mạnh mẽ cho việc tự động viết lại bộ label cho target trước khi nó được scrape. Nhiều bước relabeling có thể được cấu hình với mỗi cấu hình scrape. Chúng áp dụng cho bộ lablel của từng target theo thứ tự xuất hiện của chúng trong file cấu hình.

## 7. Lưu trữ

Prometheus lưu trữ dữ liệu lên ổ đĩa local trong cơ sở dữ liệu tùy chỉnh (PromQL) có dạng TSDB(time series database). Độ tin cậy là một thách thức của hệ thống phân tán do đó để duy trì tín tin cậy cao (reliable) Prometheus vẫn chưa hỗ trợ kiến trúc phân tán chính thức. Ngoài ra việc không phân cụm giúp việc chạy các Prometheus dạng binary dễ dàng không phụ thuộc vào các thành phần bên ngoài. Từ lúc phát triển tới nay, cơ chế lưu trữ luôn là phần được chú trong trong Prometheus. Hệ thống lưu trữ được tối ưu để có thể lưu trữ và xử lý hàng triệu mẫu dữ liệu mỗi giây, tương đương hàng chục nghìn exporter trên thực tế, với mỗi mẫu dữ liệu chỉ tốn 1,3 byte. thực tế

**Các loại lưu trữ Prometheus**

**Local storage**

Cơ sở dữ liệu chuỗi thời gian địa phương của Prometheus lưu trữ dữ liệu ở định dạng tùy chỉnh, hiệu quả cao trên bộ nhớ cục bộ.

Remote storage

Lưu trữ cục bộ của Prometheus được giới hạn ở khả năng mở rộng và độ bền của một nút duy nhất. Thay vì cố gắng giải quyết lưu trữ theo cụm trong chính Prometheus, Prometheus cung cấp một tập hợp các giao diện cho phép tích hợp với các hệ thống lưu trữ từ xa.

## 8. Tìm hiểu về Alertmanager

Alerting và Prometheus tách thành 2 phần. Các Alerting rule được Prometheus gửi đến Alertmanager, Alertmanager quản lý việc cảnh báo, bao gồm silencing, inhibition, aggregation và gửi cảnh báo đi qua các kênh như email, HipChat, PagerDuty.

3 bước để thiết lập cảnh báo:

* Cài đặt và cấu hình Alertmanager
* Cấu hình Prometheus liên kết với Alertmanager
* Tạo các rule alert trong Prometheus

**Alertmanager**

Alertmanager xử lý cảnh báo được gửi bởi ứng dụng như là Prometheus server. Nó có các cơ chế Grouping, Inhibition, Silience

* Grouping :

Phân loại cảnh báo có những tính chất tương tự. Điều này thực sự hữu ích trong một hệ thống lớn với nhiều thông báo được gửi đông thời.

Ví dụ: Một hệ thống với nhiều server mất kết nối đến cơ sở dữ liệu, thay vì rất nhiều cảnh báo được gửi về Alertmanager thì Grouping giúp cho việc giảm số lượng cảnh báo trùng lặp, thay vào đó là một cảnh báo để chúng ta có thể biết được chuyện gì đang xảy ra với hệ thống của bạn.

* Inhibition :

Inhibition là một khái niệm về việc chặn thông báo cho một số cảnh báo nhất định nếu các cảnh báo khác đã được kích hoạt.

Ví dụ: Một cảnh báo đang kích hoạt, thông báo là cluster không thể truy cập (not reachable). Alertmanager có thể được cấu hình là tắt các cảnh báo khác liên quan đến cluster này nếu cảnh báo đó đang kích hoạt. Điều này lọc bớt những cảnh báo không liên quan đến vấn đề hiện tại.

* Silience :

Silience là tắt cảnh báo trong một thời gian nhất định. Nó được cấu hình dựa trên các match, nếu nó match với các điều kiện thì sẽ không có cảnh báo nào được gửi khi đó.

* High avability:

Alertmanager hỗ trợ cấu hình để tạo một cluster với độ khả dụng cao.

**Silences Alert**: Tắt cảnh báo trong một thời gian nhất định, cấu hình trên giao diện alertmanager nền web. Alertmanager được cấu hình với các thông tin như: Routes: Định tuyến đường đi của notification. Có các route con với các match của nó. Nếu notification trùng với match của route nào đó, thì sẽ được gửi đi theo đường đó. Còn không match với route nào, nó sẽ được gửi theo đường đi mặc định. Receivers: Cấu hình thông tin các nơi nhận. Ví dụ như tên đăng nhập, mật khẩu, tên mail sẽ gửi đến,....

**Service Discovery**

Khi các exporter monitor đã được cài đặt, Prometheus cần phải biết các exporter ở đâu ? để tiến hành scrape metrics và có thể thông báo tình trạng sẵn sàng của exporter đó thông qua việc truy vấn tới. Với môi trường mạng ổn định, có thể 1 list danh sách địa chỉ các instance đủ để Prometheus scape dữ liệu. Nhưng với các môi trường mạng động: docker, kubernetes , aws, .. thì cần có service discovery để cung cấp cơ chế cho Prometheus cập nhật thông tin các exporter trong mạng một cách tự động.

Prometheus có hỗ trợ Service Discovery cho EC2, Kubernetes, Consul, Azure .. hoặc có thể tự tạo custom Service Discovery

**cAdvisor**

cAdvisor là 1 dự án Open source của Google mục đích để phân tích mức độ sử dụng, hiệu năng, và rất nhiều thông số khác từ các ứng dụng Container, cung cấp cho người dùng cái nhìn tổng quan về toàn bộ các container đang chạy.

**Node-exporter**

Prometheus Node Exporter là một chương trình exporter viết bằng ngôn ngữ Golang. Exporter là một chương trình được sử dụng với mục đích thu thập, chuyển đổi các metric không ở dạng kiểu dữ liệu chuẩn Prometheus sang chuẩn dữ liệu Prometheus. Sau đấy exporter sẽ expose web service api chứa thông tin các metrics hoặc đẩy về Prometheus.  
Node Exporter này sẽ đi thu thập các thông số về máy chủ Linux như : ram, load, cpu, disk, network,…. từ đó tổng hợp và xuất ra kênh truy cập các metrics hệ thống này ở port TCP 9100 để Prometheus đi lấy dữ liệu metric cho việc giám sát.