[**https://stackoverflow.com/questions/71179660/spring-boot-3-jakarta-and-javax**](https://stackoverflow.com/questions/71179660/spring-boot-3-jakarta-and-javax)

**Từ Spring Boot ver 3.x trở đi, sẽ dùng thư viện jakarta của Jakarta EE 9 thay vì javax. Và sẽ dùng Java 17.**

**-> khi import các thư viện vẫn xài thư viện của javax, thì down parent version của Spring Boot xuống ver 2.x.**

**Hoặc đơn giản hơn là cứ vào luôn solution copy file pom.xml cho nhanh.**

# Authentication? Authorization

## Authentication

## Authorization

# Encoding? Encrypting? Hashing? Salting?

* + **Encoding** đơn giảnlà mã hoá 1 input chuỗi kí tự thành một output bằng một thuật thoán. Output này có thể dễ dàng dịch ngược lại được bằng cách sử dụng lại chính thuật toán đã encode trước đó.
  + **Encrypting** nôm na giống như encoding, nhưng chúng sẽ đi kèm theo secret key. Khi encrypt 1 input thì sẽ đi kèm theo một secret key để cho ra output, muốn dịch ngược lại output thì ta chỉ cần thuật toán ban đầu kèm theo secret key là được
  + **Hashing** là mã hoá 1 chuỗi kí tự nhưng không thể dịch ngược lại được (hoặc rất rất khó có thể dịch ngược lại vì bản chất thuận toán hash là hash đi hash lại nhiều lần output đó sau mỗi lần hash rồi mới ra output cuối cùng)
    - Thuật toán hashing ko hoàn hảo, chúng vẫn có tỉ lệ hash ra 2 output giống nhau từ 2 input ban đầu khác nhau. Việc hash 2 input ban đầu khác nhau ra output giống nhau đó gọi là đụng độ giá trị băm (**collision)**
    - Điều này sẽ sinh ra vấn đề khi 2 user có password giống nhau sẽ hash ra cùng output giống nhau. Điều này là nên tránh vì khi ta lưu password trên hệ thống, mọi hashing value nên là unique (hashing value rất khó có thể dịch ngược lại nhưng ko có nghĩa là ko thể). **Salt** sinh ra để khắc phục vấn đề này.
  + **Salting** là hành động thêm 1 chuỗi kí tự bất kì vào cuối password trước khi hashing. Với việc ta định nghĩa trước mỗi user sẽ có một **salt** riêng biệt, ta sẽ đảm bảo rằng password của từng user nếu kết hợp với salt sẽ luôn hash ra output ko trùng nhau. Việc này sẽ làm tăng độ phức tạp của hashing value nên rất nhiều lần và việc dịch ngược lại gần như là ko thể.

# Basic auth? Session-based auth? Token-based auth?

* + <https://kipalog.com/posts/Authentication-story-part-2--Authentication-co-ban>

## Basic auth

* là cơ chế authen cổ điển nhất, đơn giản là gửi username và password kèm theo mỗi lần gửi request.
* Dấu hiệu nhận biết là ở những website cũ, khi truy cập lần đầu sẽ hiện lên dialog của browser yêu cầu nhập username và password.
* Cơ chế: Sau khi đã nhập username và password và xác thực thành công thì browser sẽ lưu lại username và pasword này và sẽ tự động gửi kèm nó trong phần header (**Authorization**) cho mỗi lần gửi request sau. Server mỗi lần nhận request thì sẽ kiểm tra lại username và password trong database.
* Nhược điểm: Điều này có nghĩa là trong mỗi request đó sẽ luôn bao gồm thông tin của username và password

=> Lỗi thời, kém bảo mật và ko thể logout vì việc username và password được lưu là do trình duyệt làm và server ko thể can thiệp được, muốn logout chỉ có cách là xoá lịch sử duyệt web hoặc đợi thời gian lưu của browser này kết thúc.

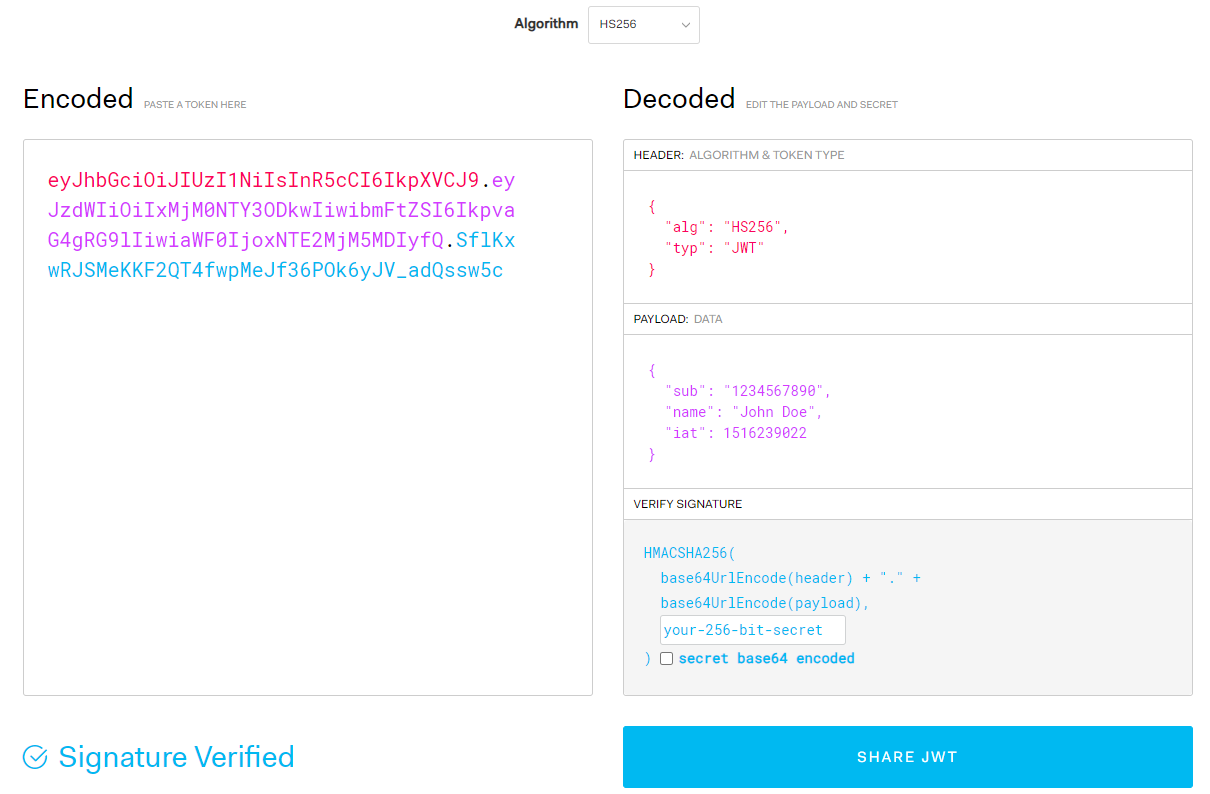
## Session-base auth

* là cơ chế đăng nhập người dùng dựa trên việc tạo ra session của người dùng ở phía server. Sau quá trình xác thực người dùng thành công (username/password,...) thì phía server sẽ tạo và lưu ra một session chứa thông tin của người dùng đang đăng nhập và trả lại cho client session ID ngẫu nhiên để truy cập session cho những request sau.
* Cơ chế: Sau khi đã nhập username và password và xác thực thành công thì server sẽ tạo ra SessionID và lưu lại trên server, sau đó gửi SessionID này về phía client. Mỗi lần user gửi request sẽ kèm theo SessionID này trong phần header (**Cookie**). Server mỗi lần nhận request thì sẽ so sánh SessionID này và SessionID do server đã tạo ra trước đó.
* Ưu điểm:
  + Vì SessionID do server gửi về là 1 chuỗi ngẫu nhiên, ko bao gồm thông tin gì nên quá trình truyển tải dữ liệu sẽ giấu thông tin về username và password.

## Token-base auth

* là cơ chế đăng nhập người dùng dựa trên việc tạo ra **token** - một chuỗi ký tự mang thông tin xác định người dùng được server tạo ra và lưu ở client. Server sau đó có thể không lưu lại token này.
* Cơ chế: Sau khi đã nhập username và password và xác thực thành công thì server sẽ tạo ra token bao gồm những thông tin cơ bản của user, encode nó, sau đó gửi token này về phía client. Mỗi lần user gửi request sẽ kèm theo token này trong phần header (thường là **Authorization**). Server mỗi lần nhận request thì sẽ decode token này và sẽ có thông tin cơ bản của user.
* Token này bản chất đã được verify khi tạo ra rồi vậy nên chúng sẽ ko cần kèm theo thông tin password nữa. Và server cũng ko cần lưu lại chúng.
* Token này đã được encode cùng với secret key. Và chúng sẽ ko thể dịch ngược lại được nếu như ko biết secret key, và secret key này chỉ có server biết => chỉ server mới biết được trong token này chứa thông tin gì.

# JWT

* + <https://viblo.asia/p/jwt-va-ung-dung-cua-no-vyDZOMRk5wj>
  + <https://viblo.asia/p/jwt-tu-co-ban-den-chi-tiet-LzD5dXwe5jY>
  + ****
  + <base64-encoded header>.<base64-encoded payload>.<HMACSHA256(base64-encoded signature)>
  + 
    - **Header:** bao gồm thuật toán encode chuỗi JWT này, và loại token (ở đây là JWT)
    - **Payload:** chứa các claims, chứa thông tin của user và vài thông tin về token như issue at,…
    - **Signature:** bao gồm header và payload đã được encode theo kiểu base64, secret key, và thuật toán encode được chỉ định trong header => nếu thay đổi payload thì signature cũng sẽ thay đổi theo.
  + Payload trong JWT thường ko bao gồm password (và cũng ko cần thiết) vì bản chất JWT được tạo ra từ việc encoding và decoding, ta hoàn toàn có thể đọc được nội dung bên trong nếu như biết được thuật toán encoding và chuỗi secret key.

# Testing

## JUnit

## Selenium

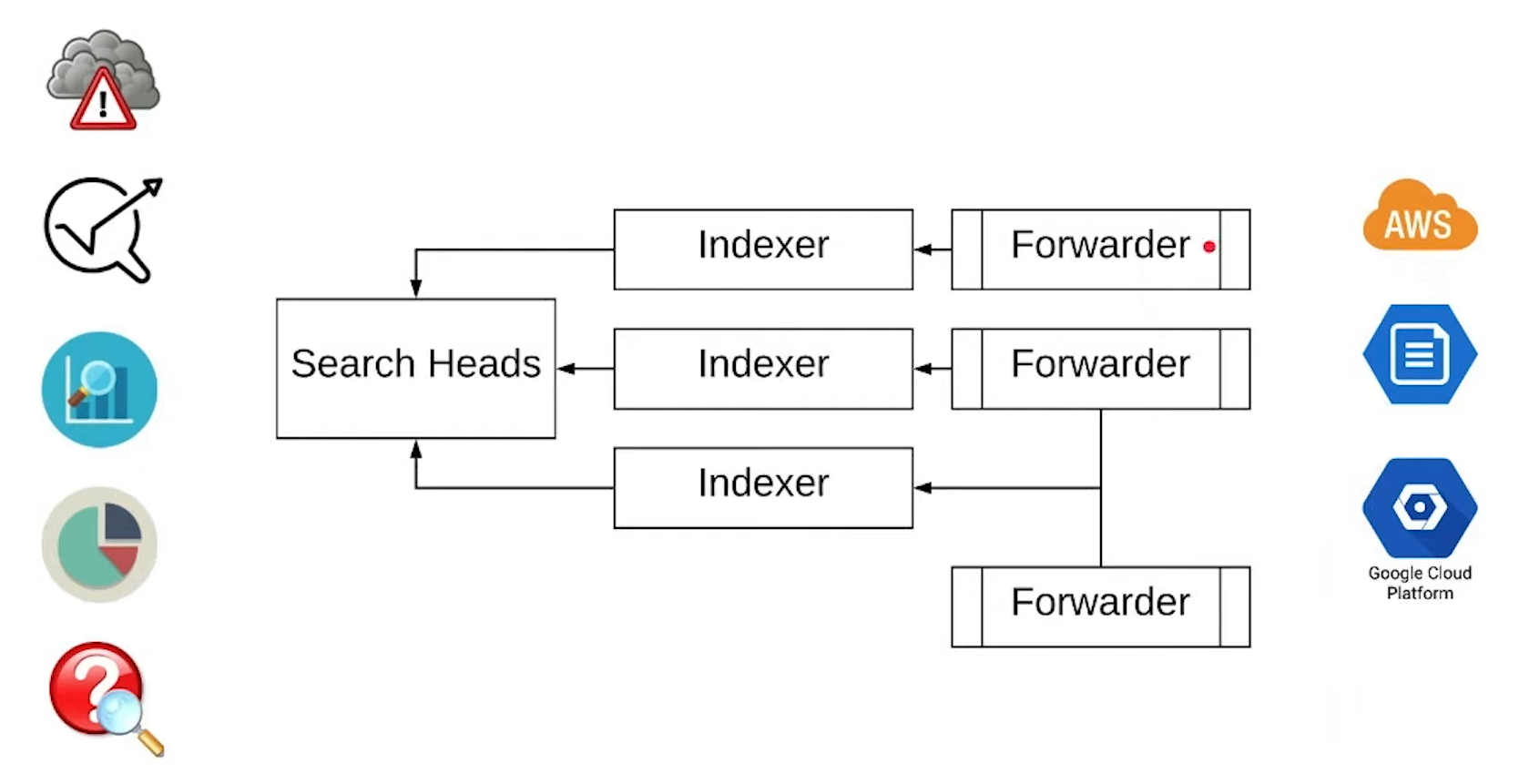
## Mockito

* + **Mockito** là một framework cho phép ta tạo các đối tượng giả (mock object) để có thể tập trung vào việc test những thứ khác bằng cách bỏ qua việc khởi tạo các object (nếu object đó có nhiều dependency). Bằng cách này ta có thể giả lập hành vi của các object đó.
    - **@Mock/mock():** dùng để tạo một mock object **hoàn toàn giả** -> những method được gọi từ object này sẽ ko thực hiện code gì trong method cả nếu chúng ta ko giả hành vi của chúng.
    - **when()/given():** dùng để định nghĩa hành vi của object đó, có arg là một method cụ thể.
    - **thenReturn():** để giả hành vi rằng method đó sẽ return về value gì.
    - **thenThrow():** để giả hành vi rằng method đó sẽ throw exception gì.
    - **@Spy/spy():** dùng để tạo một object **thật** nhưng ta vẫn có thể giả hành vi của chúng -> những method được gọi từ object này sẽ thực sự thực hiện code bên trong các method đó nếu method đó ko được giả hành vi.
    - **@InjectMock:**
  + Mockito ko thể mock method private. Mockito ngầm định rằng method private ko tồn tại trong viewpoint của việc test.

# SLF4J

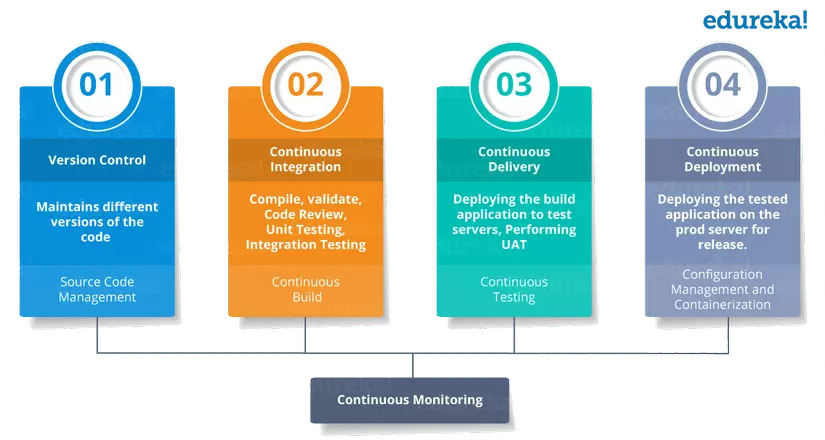
* + <https://viblo.asia/p/hoc-facade-design-pattern-qua-cau-chuyen-RnB5pxBr5PG>
  + **Facade pattern** là một design pattern. **Facade** (mặt tiền) có nghĩa là ta chỉ cần 1 lớp đại diện thực thi tất cả các công việc phức tạp của hệ thống con (subsystem).
    - Có nghĩa là trong hệ thống của ta có nhiều class con và thực hiện nhiều method phức tạp, khó hiểu. Việc thực hiện 1 chức năng cần call nhiều method ở nhiều class khác nhau theo một thứ tự nào đó. Khi đó để đơn giản hơn ta sẽ gom những method nằm trong class con đó lại vào một method duy nhất trong 1 class gọi là **facade**. Khi đó ta chỉ cần gọi method trong class **facade** là những method trong class con sẽ được gọi theo đúng thứ tự ta muốn.
    - Kể cả khi ta cần gọi trực tiếp method trong class con ta vẫn có thể gọi trực tiếp. Điều này giúp ta tách biệt phần code của client và code thực thi. Khi có thay đổi, ta chỉ cần chỉnh sửa lại code facade mà không ảnh hưởng tới client.
  + **SLF4J (Simple logging fascase for Java)** được sử dụng như một facade hay lớp abstraction cho những logging framework khác nhau như **java.util.logging**, **logback** và **Log4j.**
  + Logging levels: TRACE, DEBUG, INFO, WARN, ERROR, FATAL. Cấp độ nguy hiểm tăng dần.

# Splunk

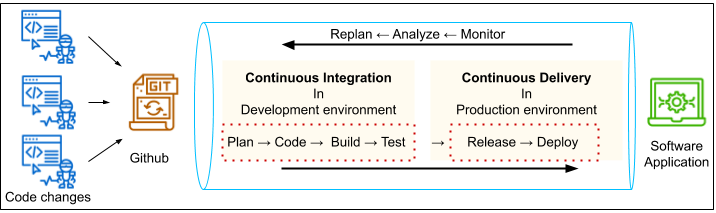
* + [**https://securityzone.vn/t/tong-hop-kien-thuc-co-ban-ve-splunk-danh-cho-nguoi-moi-bat-dau.11792/**](https://securityzone.vn/t/tong-hop-kien-thuc-co-ban-ve-splunk-danh-cho-nguoi-moi-bat-dau.11792/)
  + **Splunk** là một search engine dùng để phân tích dữ liệu được sinh ra từ ứng dụng (machine-generated data), bao gồm application logs, server logs, network devices, sensors, IoT, cloud services, or mobile services. Hiểu đơn giản Splunk là một tool tìm kiếm, giám sát và phân tích log.
  + **Splunk** giúp tạo ra thông tin dựa vào machine-generated data và phân tích chúng, giúp build những hệ thống auto-alert, đánh giá hiệu suất hệ thống, phân tích lỗi, mô phỏng data (vẽ biểu đồ, report),…
  + **Splunk** sẽ thu thập data trực tiếp từ các thiết bị được cấu hình đẩy log về splunk hoặc bằng cách cài các agent lên trên thiết bị. Các log được xử lí và lưu lại dưới các **indexer** để phục vụ cho việc tìm kiếm của người dùng
  + ****
  + **Index** (đánh chỉ mục) là hành động đánh chỉ mục raw data để biến nó thành một event để có thể tìm kiếm (searchable event).
  + **Forwarder** sẽ thực hiện việc lấy raw data và chuyển nó tới indexer để xử lí.
  + **Indexer** là nơi xử lí raw data và lưu lại các kết quả đó dưới dạng các events. Hiểu đơn giản giống như một nhà máy, nó sẽ nhìn vào data và xử lí chúng. Indexer sau đó sẽ gán **timestamp** cho các event và gánvào một **index** để người dùng có thể tìm kiếm. Người dùng muốn truy vấn event trong khoảng thời gian nào thì chỉ việc mở các **index** trong khoảng thời gian đó.
  + **Search head** là nơi cho phép người dùng có thể sử dụng ngôn ngữ Splunk **(SPL – Splunk Processing Language)** để có thể tìm kiếm dữ liệu đã được indexed. Search head xử lí search request của người dùng và phân tán nó đến các indexer, sau khi nhận được kết quả từ các indexer, Search head thực hiện hợp nhất kết quả và trả về response cho người dùng. Từ đây ta có được report hay biểu đồ dữ liệu ta cần xem.
  + Splunk components (Splunk instance) được chia thành 2 loại:
    - **Processing-components**: Forwarder, Indexer, Search head.
    - **Managing-component**: Deployement server, Monitoring console,…
  + Raw data sẽ đi vào pipeline, sau đó chúng được xử lí bởi Processing-components. Pipeline được chia thành những phân đoạn nhỏ hơn, mỗi phân đoạn map với một hoặc nhiều Processing-components. Những phân đoạn đó là data input, parsing, indexing và searching.
    - Data input: indexer, universal forwarder, heavy forwarder
    - Parsing: indexer, heavy forwarder
    - Indexing: Indexer
    - Search: indexer, search head

# DevOps

## DevOps

* DevOps là sự kết hợp của 2 từ Development (phát triển) và Operation (vận hành)
  + Development bao gồm phần việc của designer, developer, QA QC…
  + Operations có sự tham gia của system engineer, system administrator, operation executive, release engineer, DBA, network engineer, security engineer…
* DevOps là phương pháp phát triển phần mềm mà việc phát triển, kiểm thử, triển khai, theo dõi phần mềm một cách liên tục suốt vòng đời phát triển của nó.

## CI/CD

* **CI/CD** là viết tắt của **Continous integration** (tích hợp liên tục) và **Continous delivery/Continous deployment** (triển khai liên tục), tức là tự động hoá quy trình development của phần mềm và delivery sản phẩm, giúp rút ngắn thời gian tích hợp và phần mềm được đưa tới user một cách nhanh nhất.
  + Continuous Integration nghĩa là những code mới của một project sẽ được build, validate, test và merge vào một repository ví dụ như git.
  + Continuous Delivery là quá trình tự động hoá việc release code đã được merge và release lên test server.
  + Continuous Deployment là quá trình tự động hoá việc release code đã được test UAT và release lên production.
* **Continuous Delivery** sẽ triển khai phiên bản pre-release của sản phẩm, yêu cầu ta kiểm thử mọi thứ đều ổn thì mới tiến hành triển khai. Còn **Continuous Deployment** thì sẽ triển khai thẳng lên production luôn.
* CI/CD pipeline là xương sống của môi trường DevOps. Nó là cầu nối giúp team Dev và Ops (có thể làm cả dev và ops) tự động quá trình build, test, và deploy app. Nhưng ko phải dự án nào cũng có mô hình này, thường là tách team Dev và team DevOps riêng.

## Git

* Repository của Git được chia thành 2 loại:
  + **Remote repository**: Là repository để chia sẻ giữa nhiều người và bố trí trên server chuyên dụng. Code của mọi người sẽ được share công khai tại đây.
  + **Local repository**: Là repository bố trí trên máy của bản thân mình, dành cho một người dùng sử dụng. Ta sẽ code trên repo này và commit lên remote repo.
* Upstream là remote trên repo gốc, origin mà remote trên repo đã fork
* Khi tạo một repository sẽ được cấp một main branch mặc định, gọi là **branch master.**
* Commands:
  + **git fork** là bản copy của một repository, tức là ta duplicate một toàn bộ code của một repository, ko bao gồm history. Tức là ta tạo một **remote repository**
  + **git clone** khác với git fork, là một bản copy ở trên local (trên máy) của một repository, tức là ta đã sao chép toàn bộ code của một repository, bao gồm cả branch và history. Tức là ta tạo một **local repository**
  + **git status** để track những changes.
  + **git diff** để xem code thay đổi
  + **git add** để add file vào commit. **git add .** để commit toàn bộ file
  + **git commit** để commit. **-m “comment”** để add comment
  + **git reset** để revert commit
  + **git push** để push
  + **git merge** để merge những changes từ nhánh này sang nhánh khác
  + **git remote** để tạo, xem và xoá liên kết giữa các repo. **git remove -v** để xem toàn bộ remote
  + **git checkout** để chuyển giữa các branch. **git checkout master** là trở về branch chính
  + **git fetch** để fetch ở branch hiện tại.
  + **git pull** để pull code mới nhật ở branch hiện tại. Docker
* Container?
* Image?
* EC2 instance?

## Jenkins

## AWS

* Amazon Web Services (AWS) là một cloud service provider