

**ĐẠI HỌC QUỐC GIA TP. HỒ CHÍ MINH**  
**TRƯỜNG ĐẠI HỌC BÁCH KHOA**  
**KHOA KHOA HỌC & KỸ THUẬT MÁY TÍNH**



**LUẬN VĂN TỐT NGHIỆP**  
**TĂNG TỐC MÔ HÌNH XỬ LÝ DEVOPS VỚI**  
**DOCKER**

**HỘI ĐỒNG: Hệ thống và Mạng Máy tính 2**

**GVHD: TS. Phạm Trần Vũ**

**—oOo—**

**SVTH 1: Nguyễn Hoàng Anh (51200071)**

**SVTH 2: Nguyễn Ngọc Ân (51200161)**

**TP. HỒ CHÍ MINH, 12/2016**

# LỜI CAM ĐOAN

Báo cáo này là luận văn tốt nghiệp tại Khoa Khoa học và Kỹ thuật Máy Tính, Trường Đại học Bách Khoa Thành Phố Hồ Chí Minh. Đề tài được thực hiện tại công ty TNHH công nghệ thông tin ELCA (Việt Nam) từ tháng 7 đến tháng 12 năm 2016.

Chúng tôi xin cam đoan nội dung báo cáo này là do chúng tôi tự nghiên cứu và thực hiện. Ngoài những nguồn tham khảo đã được chúng tôi ghi rõ, các công việc trình bày trong báo cáo đều do chúng tôi nghiên cứu và không có phần nào của báo cáo là sao chép từ các nguồn tài liệu hay của bất kì ai khác.

Nếu có bất kỳ sai phạm nào chúng tôi xin chịu hoàn toàn trách nhiệm trước Hội đồng bảo vệ, Ban chủ nhiệm khoa và Ban giám hiệu nhà trường.

**TP. HỒ CHÍ MINH, THÁNG 12/2016**

*Nguyễn Ngọc Ẩn      Nguyễn Hoàng Anh*

## LỜI CẢM ƠN

Chúng tôi xin chân thành cảm ơn thầy Phạm Trần Vũ và công ty TNHH công nghệ thông tin ELCA Việt Nam đã đồng hướng dẫn chúng tôi thực hiện đề tài này. Đặc biệt, xin gửi lời cảm ơn anh Lê Đỗ Anh Phong ở công ty ELCA, là người trực tiếp hướng dẫn chúng tôi thực hiện đề tài tại công ty. Sự chỉ bảo tận tình của anh trong suốt thời gian thực hiện là rất quan trọng để chúng tôi có thể hoàn thành luận văn này.

Chúng tôi xin gửi lời cảm ơn đến tất cả các thầy cô trong bộ môn đã nhiệt tình giảng dạy, cung cấp kiến thức cho chúng tôi trong suốt thời gian học tập ở trường.

Chúng tôi cũng xin gửi lời cảm ơn đến gia đình và những người bạn, luôn dành cho chúng tôi sự yêu thương và hỗ trợ tốt nhất.

## TÓM TẮT ĐỀ TÀI

Virtual Machines (VMs) là một phần không thể thiếu trong ngành công nghiệp phần mềm hiện nay. VMs giúp doanh nghiệp sử dụng hiệu quả hơn tài nguyên máy chủ, tận dụng tối đa năng suất của các thiết bị phần cứng, tiết kiệm không gian, nguồn điện.... Ngoài ra, VMs còn giúp giảm thời gian thiết lập máy chủ, tạo các môi trường cho việc kiểm tra phần mềm trước khi đưa vào hoạt động.

Containers (Operating-system-level virtualization) cũng là một phương pháp ảo hóa đã có từ lâu. Không giống như VMs, một hoặc nhiều các máy tính độc lập có thể chạy ảo hóa trên một máy tính vật lý, Containers thay vào đó chạy bên trên một nhân hệ điều hành (operating system's kernel). Nói cách khác, VMs ảo hóa ở mức phần cứng (hardware), Containers chỉ ảo hóa ở mức hệ điều hành (operating system).

Docker là một engine mã nguồn mở sử dụng công nghệ Containers. Docker được xây dựng để hướng tới việc phân phối các ứng dụng nhanh hơn, từ việc vận chuyển (ship) code, kiểm tra (test) code đến triển khai (deploy), rút ngắn quy trình giữa việc viết code và chạy code.

Ở đề tài này, chúng tôi nghiên cứu việc sử dụng Docker để thay thế VMs truyền thống trong quy trình phát triển các ứng dụng Java Platform, Enterprise Edition (Java EE). Cụ thể, chúng tôi nghiên cứu việc sử dụng Docker để hiệu quả hóa hơn nữa việc phát triển (develop) và triển khai (deploy) các ứng dụng Java EE tại công ty ELCA. Đồng thời, đánh giá tính hiệu quả thực tế giữa việc sử dụng Docker và sử dụng VMs truyền thống.

# MỤC LỤC

LỜI CAM ĐOAN	i
LỜI CẢM ƠN	ii
TÓM TẮT ĐỀ TÀI	iii
<b>Chương 1 TỔNG QUAN</b>	<b>1</b>
1.1 Giới thiệu đề tài . . . . .	1
1.2 Mục tiêu đề tài . . . . .	2
1.3 Quy trình thực hiện . . . . .	3
1.4 Tầm quan trọng của nghiên cứu . . . . .	4
1.5 Tổ chức báo cáo . . . . .	4
<b>Chương 2 DOCKER</b>	<b>6</b>
2.1 Giới thiệu . . . . .	6
2.2 Các thành phần cơ bản . . . . .	6
2.2.1 Docker Client và Server . . . . .	7
2.2.2 Registry . . . . .	7
2.2.3 Docker Image . . . . .	8
2.2.4 Docker Container . . . . .	14
2.3 Docker Compose và Docker Machine . . . . .	19
2.3.1 Docker Compose . . . . .	19
2.3.2 Docker Machine . . . . .	23
2.4 Docker networking . . . . .	26
2.4.1 Docker Networking . . . . .	26
2.4.2 Docker Multihost Networking . . . . .	28

2.5	Container Orchestration . . . . .	28
2.5.1	Docker Swarm . . . . .	29
2.5.2	Kubernetes . . . . .	30
2.5.3	Mesos và Marathon . . . . .	31
<b>Chương 3 OPENSIFT</b>		<b>32</b>
3.1	Giới thiệu . . . . .	32
3.2	OpenShift Origin . . . . .	32
3.2.1	OpenShift v3 và Docker . . . . .	32
3.2.2	Kiến trúc . . . . .	33
3.2.3	Các thành phần cơ sở . . . . .	35
3.2.4	Các khái niệm cốt lõi . . . . .	36
3.3	Cài đặt và cấu hình hệ thống . . . . .	42
3.3.1	Thông tin hệ thống . . . . .	42
3.3.2	Cài đặt hệ thống . . . . .	43
3.4	Source-to-Image (S2I) . . . . .	44
3.4.1	Cơ sở kiến thức . . . . .	44
3.4.2	Xây dựng Image với S2I . . . . .	47
<b>Chương 4 CONTINUOUS INTERGRATION - JENKINS</b>		<b>52</b>
4.1	Những điểm mới trên Jenkins 2.0 . . . . .	52
4.2	Một số câu lệnh đơn giản trên Jenkins 2.0 . . . . .	54
<b>Chương 5 TÍCH HỢP DEVOPS PIPELINE VỚI DOCKER</b>		<b>57</b>
5.1	Chuẩn bị . . . . .	57
5.2	Tích hợp Pipeline với Docker . . . . .	58

<b>Chương 6 ĐÁNH GIÁ HIỆU NĂNG CỦA DOCKER SO VỚI VMS TRONG DEVOPS PIPELINE CỦA CÔNG TY ELCA</b>	<b>71</b>
6.1 Thời gian chạy Pipeline . . . . .	72
6.2 Độ ổn định của Pipeline . . . . .	73
6.3 Dung lượng sử dụng . . . . .	73
6.4 Chi phí cấu hình . . . . .	73
6.5 Tái sử dụng image . . . . .	74
6.6 Scalability . . . . .	74
<b>Chương 7 KẾT LUẬN</b>	<b>75</b>
7.1 Kết luận . . . . .	75
7.2 Hướng phát triển . . . . .	76
<b>TÀI LIỆU THAM KHẢO</b>	<b>77</b>
<b>PHẦN PHỤ LỤC</b>	<b>81</b>
<b>Chương A CÀI ĐẶT DOCKER TRÊN HỆ ĐIỀU HÀNH LINUX</b>	<b>81</b>
A.1 Cài đặt Docker Engine . . . . .	81
A.1.1 Tải về Docker Engine binaries cho Linux . . . . .	81
A.1.2 Cài đặt . . . . .	81
A.1.3 Chạy Engine . . . . .	82
A.1.4 Đặt quyền truy cập cho non-root . . . . .	82
A.2 Cài đặt Docker Compose . . . . .	83
A.3 Cài đặt Docker Machine . . . . .	83
<b>Chương B CÀI ĐẶT CÁC CÔNG CỤ ĐỂ GIÁM SÁT VÀ VẬN HÀNH TRÊN OPENSIFT ORIGIN</b>	<b>85</b>
B.1 Origin Metrics . . . . .	85

B.2	Origin Aggregated Logging . . . . .	86
-----	-------------------------------------	----



## DANH SÁCH HÌNH VẼ

2.1	Kiến trúc của Docker . . . . .	7
2.2	Cấu trúc layers của image Ubuntu 15.04 . . . . .	9
2.3	Writable layer . . . . .	9
2.4	Docker bridge mode networking . . . . .	27
2.5	Docker run with host networking . . . . .	27
2.6	Docker host mode networking . . . . .	28
3.1	Kiến trúc tổng quan của OpenShift Origin . . . . .	34
3.2	S2I build workflow . . . . .	46
4.1	Giao diện Pipeline mới . . . . .	53
4.2	Snippet Generator . . . . .	56
5.1	Tạo a-prj trên OpenShift Origin . . . . .	58
5.2	Tạo a-prj-pipeline trên Jenkins . . . . .	59
5.3	Pipeline trên Jenkins 1.x của project A tích hợp với VMs . . . . .	60
5.4	Pipeline trên Jenkins 2.0 của project A tích hợp với Docker . . . . .	61
5.5	Project A đã deploy thành công với version là 3.0.22.9034 trên môi trường AT . . . . .	65
5.6	Project A đã deploy thành công với version là 3.0.22.9034 trên môi trường INT . . . . .	68
5.7	Kết quả sau khi chạy xong pipeline . . . . .	69
5.8	Kết quả test . . . . .	69
5.9	Các Pod của project A đang chạy trên OpenShift Origin . . . . .	70
5.10	Các Docker Image đã build của project A trên OpenShift Origin . . . . .	70
6.1	Thời gian chạy pipeline của Project A với Docker . . . . .	72
6.2	Thời gian chạy pipeline của Project A với VMs . . . . .	72

6.3	Scale project A với 3 pods . . . . .	74
-----	--------------------------------------	----

## **DANH SÁCH BẢNG**

3.1	Thông tin hệ thống OpenShift Origin . . . . .	42
3.2	S2I Scripts . . . . .	47
6.1	Thời gian chạy trong một số stage trên Jenkins . . . . .	72
6.2	Dung lượng lưu trữ của hai môi trường deploy . . . . .	73

# Chương 1 TỔNG QUAN

## 1.1 Giới thiệu đề tài

Ngày nay, ảo hóa máy chủ đã trở thành xu hướng chung của hầu hết các doanh nghiệp. Vì những lợi ích rõ ràng về mặt kinh tế mà việc ảo hóa mang lại. Từ một máy chủ vật lý, nhờ ảo hóa chúng ta có thể tạo ra nhiều các máy chủ ảo khác nhau (cả hệ điều hành lẫn các phần mềm cài đặt trên đó). Các máy chủ ảo hóa hoạt động bình thường như một máy chủ vật lý thực sự. Mặc dù về hiệu suất và khả năng lưu trữ sẽ kém hơn so với máy chủ vật lý, tuy nhiên doanh nghiệp bù lại sẽ tiết kiệm được rất nhiều nguồn tài nguyên.

Ảo hóa phần cứng (hardware virtualization) là phương pháp được chấp nhận và sử dụng rộng rãi từ lâu. Với nhiều các phần mềm hỗ trợ tạo các môi trường ảo bằng ảo hóa phần cứng từ các tập đoàn lớn (VMWare, Microsoft, Oracle..), việc tạo một máy ảo là cực kì dễ dàng. Tuy nhiên ảo hóa (virtualization) trong khoa học máy tính là một khái niệm rộng, mà ảo hóa phần cứng chỉ là một trong số đó. Trong những năm gần đây, ảo hóa mức hệ điều hành (Operating-system-level virtualization hay đơn giản là Container) đang nổi lên trở thành một trong những vấn đề nổi bật nhất của ngành IT nói chung.

Containers virtualization là một phương pháp ảo hoá mà tầng ảo hoá chạy như một ứng dụng trên hệ điều hành. Người ta còn gọi phương pháp này là ảo hóa Container-based, để phân biệt với ảo hóa hypervisor-based (ảo hóa phần cứng). Với Hypervisor-based, một hoặc nhiều các máy tính độc lập ảo hóa có thể chạy trên một phần cứng vật lý thông qua một lớp trung gian (intermediation layer). Containers virtualization, thay vào đó chạy không gian người dùng (user space) trên một nhân hệ điều hành. Phương pháp container cho phép tạo nhiều các user space biệt lập và chạy nó trên một máy đơn.

So với hypervisor virtualization, Containers có những điểm hạn chế. Chúng ta chỉ có thể chạy một guest operating system giống hoặc tương tự với host bên dưới. Ví dụ, chúng ta có thể chạy một Redhat Enterprise Linux trên một Ubuntu server, nhưng không thể chạy một Microsoft Windows trên một Ubuntu server. Ngoài ra, container cũng được xem là có tính biệt lập (isolation) kém hơn so với sự biệt lập hoàn toàn (full isolation) của hypervisor virtualization.

Bỏ qua những hạn chế, Containers có khá nhiều ưu điểm. Containers không cần một hypervisor layer để chạy mà thay vào đó sẽ dùng các lời gọi hệ thống bình thường của hệ điều

hành. Containers có hiệu suất tốt hơn, có tính linh hoạt hơn, chi phí để chạy một container là nhỏ hơn khá nhiều do đó nó cho phép chạy số lượng lớn container trên một host.

Mặc dù vậy, container chưa từng đạt được sự chấp nhận ở quy mô lớn. Một trong những nguyên nhân chính là do nó khá phức tạp: khó để cài đặt, khó để vận hành và tự động hóa.

Docker, một open-source ra đời từ năm 2013, với chức năng tự động đóng gói các ứng dụng thành container. Docker cung cấp thêm một lớp trừu tượng và tự động hóa cho Containers virtualization, đồng thời nâng cao tính biệt lập (isolation) cho container, giảm thiểu các chi phí (overhead) cho việc khởi động (starting) và duy trì (maintaining) của các máy ảo. Docker giúp việc cài đặt, vận hành, tự động hóa Containers virtualization dễ dàng hơn bao giờ hết.

Nếu tận dụng tốt những ưu điểm của container, sử dụng container để thay thế VMs truyền thống tại những điểm phù hợp sẽ mang lại thêm những lợi ích cho doanh nghiệp. Tuy nhiên, đây là một công nghệ mới nổi (dù đã có lịch sử phát triển từ lâu), do đó cần thiết phải có sự nghiên cứu kỹ lưỡng về mặt hiệu quả trước khi áp dụng thực tế. Nhằm đáp ứng phần nào yêu cầu đó, tại luận văn này chúng tôi nghiên cứu sử dụng Docker container thay thế cho Virtual Machines trong quy trình phát triển các ứng dụng Java Platform, Enterprise Edition (Java EE) trong môi trường công ty ELCA. Chúng tôi kỳ vọng việc thay thế này sẽ giúp cho quy trình phát triển (develop) cũng như triển khai triển khai (deploy) các ứng dụng Java EE tại công ty trở nên dễ dàng hơn, nhanh hơn, hiệu quả hơn. Đồng thời với những số liệu thu được, sẽ giúp chúng tôi có một sự so sánh tổng quan về hiệu suất, hiệu quả sử dụng tài nguyên của hai phương pháp ảo hóa này.

Cũng phải nói lại rằng Docker và Virtual Machine là hai công nghệ khác nhau với những đặc điểm riêng biệt. Do đó nghiên cứu của chúng tôi không nhằm nâng cao hay đánh giá chung chung Docker với Virtual Machine, mà đây là sự đánh giá trong một môi trường cụ thể.

## 1.2 Mục tiêu đề tài

Sau khi hoàn thành luận văn, chúng tôi cần đạt được những mục tiêu sau:

- Sử dụng Docker để tạo điều kiện cho việc phát triển, triển khai và kiểm thử các ứng dụng, bao gồm:
  - Tăng tốc mô hình xử lý DevOps với việc sử dụng container.

- Giảm chi phí bảo trì (maintaining) và cấu hình (configuring) phần mềm
- Sử dụng Docker thay thế VMs để giảm thiểu chi phí
- Sử dụng Docker Registry để lưu giữ lâu dài các Image
- Tích hợp Docker vào DevOps Pipeline của ELCA.
- Đánh giá về hiệu suất khi sử dụng so với phương pháp VMs truyền thống.

### 1.3 Quy trình thực hiện

Ở giai đoạn thực tập tốt nghiệp trước luận văn, chúng tôi đã thực hiện:

- Tìm hiểu về kiến trúc, cách cài đặt và sử dụng cơ bản Docker. Tìm hiểu một số công cụ của Docker (Docker Compose, Docker Machine).
- Tìm hiểu kiến trúc Java EE. Các máy chủ ứng dụng phổ biến cho ứng dụng JavaEE bao gồm Apache Tomcat, WildFly (JBoss AS) và Oracle Weblogic cùng với sự triển khai (deployment) ứng dụng lên máy chủ tương ứng.
- Tìm hiểu về quy trình phát triển ứng dụng Java EE, cùng với những công cụ chính nằm trong quy trình này (Maven, Jenkins, Selenium)
- Xây dựng một DevOps pipeline tích hợp với Docker đơn giản cho một project Java EE do chúng tôi tự phát triển.

Tại luận văn này, chúng tôi tiếp tục:

- Tìm hiểu về Docker Ecosystem, Docker Networking, Service Discovery và Container Orchestration.
- Nghiên cứu về một số các framework hỗ trợ quản lý container cluster (Docker Swarm, Kubernetes, Apache Mesos, Marathon). Lựa chọn công nghệ phù hợp với cơ sở hạ tầng mà Công ty ELCA cấp cho chúng tôi.
- Dựa trên DevOps pipeline của một số project Java EE đã có của công ty ELCA, chúng tôi sẽ tiến hành cấu trúc lại sao cho phù hợp với môi trường mới chạy Docker. Đồng thời đánh giá hiệu suất sử dụng Docker so với VMs.

- Cuối cùng, nếu thời gian cho phép, chúng tôi sẽ cố gắng thực hiện performance test những project đã được tích hợp chạy trên Docker để đánh giá khả năng scale với Docker - một trong những lợi thế của Docker so với VMs. Tuy nhiên nếu thời gian không cho phép chúng tôi sẽ không thực hiện phần này.

## 1.4 Tầm quan trọng của nghiên cứu

Tầm quan trọng của thực hiện nghiên cứu này được thể hiện ở hai khía cạnh:

- Với bản thân chúng tôi, tầm quan trọng của nghiên cứu nằm ở khả năng tìm hiểu, tích lũy kinh nghiệm và giải quyết vấn đề trong quá trình nghiên cứu. Bên cạnh đó, chúng tôi còn được làm việc với những người đàn anh có nhiều kinh nghiệm tại công ty ELCA là một trải nghiệm tuyệt vời. Trong quá trình làm việc tại ELCA, chúng tôi có cơ hội để trao đổi và hoàn thiện kỹ năng cá nhân sẽ là một bước đệm lớn trước khi chúng tôi ra trường.
- Với bản thân đề tài nghiên cứu, đây là đề tài được hỗ trợ bởi công ty ELCA, một công ty với gần 50 năm kinh nghiệm hoạt động trong lĩnh vực công nghệ thông tin. Trong đề tài này, chúng tôi nghiên cứu giải pháp tích hợp DevOps pipeline với Docker, nếu giải pháp cho thấy kết quả thành công thì họ sẽ có thể xem xét sử dụng Docker thay thế cho VMs trong việc thiết lập môi trường kiểm thử sản phẩm. Tuy nhiên, nếu kết quả không được như mong đợi thì họ có thể xem đây là giải pháp chưa được tối ưu, điều này sẽ giúp họ tiết kiệm được thời gian cũng như tiền bạc khi họ xem xét cách tiếp cận khác tốt hơn. Cuối cùng, chúng tôi hy vọng đây có thể xem là tài liệu tham khảo hữu ích cho những nghiên cứu khác có liên quan.

## 1.5 Tổ chức báo cáo

Báo cáo thực tập tốt nghiệp của chúng tôi được tổ chức thành 6 chương.

### Chương 1. TỔNG QUAN

Giới thiệu tổng quan về đề tài, mục đích và yêu cầu cần đạt được sau khi thực hiện đề tài này.

### Chương 2. DOCKER

Giới thiệu về Docker, chi tiết về những thành phần cơ bản của Docker. Giới thiệu Docker Com-

pose và Docker Machine. Bên cạnh đó, chúng tôi cũng tìm hiểu về Docker networking và các Container orchestration, là những kiến thức cần thiết khi chạy một hệ thống Docker trên nhiều máy tính (multiple hosts).

### **Chương 3. OPENSIFT**

Giới thiệu về OpenShift, cách cài đặt và cấu hình trên cơ sở hạ tầng của công ty ELCA. Đồng thời tìm hiểu và trình bày cách tạo các image với S2I.

### **Chương 4. CONTINUOUS INTERGRATION - JENKINS**

Giới thiệu công cụ CI phổ biến và được công ty ELCA sử dụng trong quy trình phát triển ứng dụng: Jenkins.

### **Chương 5. TÍCH HỢP DEVOPS PIPELINE VỚI DOCKER**

Mô tả cách chúng tôi tiến hành tích hợp Docker vào DevOps pipeline cho một Java EE project của công ty ELCA.

### **Chương 6. ĐÁNH GIÁ HIỆU NĂNG CỦA DOCKER SO VỚI VMS TRONG DEVOPS PIPELINE CỦA CÔNG TY ELCA**

Đánh giá hiệu năng của Docker và VMs với những số liệu có được sau khi tích hợp DevOps Pipeline với Docker cho project đã thực hiện ở chương 5.

### **Chương 7. KẾT LUẬN**

Những kết quả đạt được và hướng phát triển của luận văn.



## Chương 2 DOCKER

### 2.1 Giới thiệu

Docker là một open-source engine tự động hóa việc triển khai các ứng dụng vào bên trong các container. Được viết bởi Docker, Inc và được phát hành dưới giấy phép Apache 2.0.

Docker triển khai các ứng dụng trên một container ảo hóa từ môi trường thực thi. Được thiết kế để cung cấp các môi trường nhanh và gọn nhẹ (lightweight) cho việc chạy code cũng như tăng tính hiệu quả (efficient) cho cả quy trình (workflow) (từ việc lấy code từ trên máy của laptop của chúng ta đến thực thi trên môi trường test và môi trường production). Docker có những khả năng nổi trội sau đây:

- Docker rất nhanh. Chúng ta có thể đóng gói một ứng dụng Docker trong vài phút và khởi chạy ứng dụng đã đóng gói đó trong vài giây. Những ai đã từng khởi động một máy tính ảo hóa bằng Virtual Machine sẽ dễ dàng nhận ra sự khác biệt lớn này.
- Với Docker, người phát triển (Developers) chỉ quan tâm ứng dụng của họ chạy được trong container và người vận hành (Operations) quan tâm về việc quản lý các container. Docker được thiết kế cho việc tăng cường tính nhất quán (consistency) bằng việc đảm bảo rằng môi trường mà developer viết code sẽ giống với môi trường mà ứng dụng được triển khai. Điều này giúp làm giảm thiểu rủi ro cho việc ứng dụng chạy được trên môi trường dev nhưng khi triển khai lại có vấn đề.
- Mục đích của Docker là giảm thiểu thời gian (cycle time) giữa việc viết code và việc code sau đó được kiểm tra, triển khai và sử dụng. Nâng cao tính di động (portable) cho ứng dụng, dễ dàng build cũng như dễ dàng cho việc cộng tác trên nó.

### 2.2 Các thành phần cơ bản

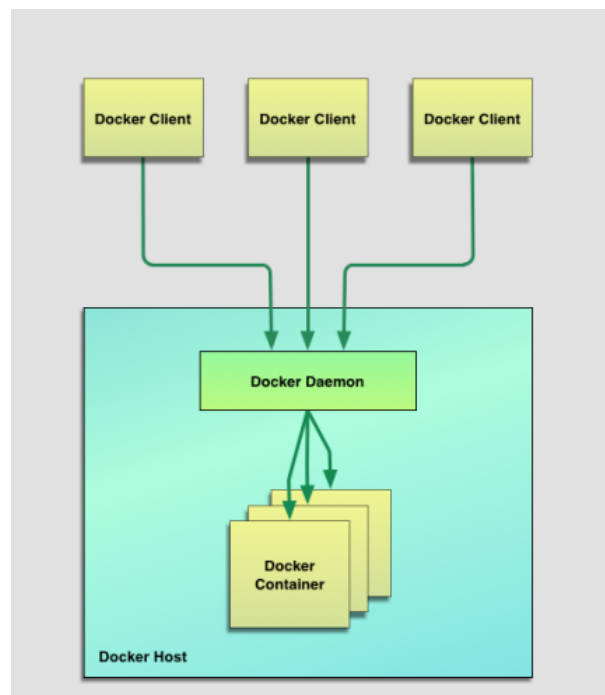
Docker gồm có các thành phần cơ bản sau:

- Docker Client và Server
- Docker Image

- Docker Registry
- Docker Container

### 2.2.1 Docker Client và Server

Docker là một ứng dụng client-server. Docker client sẽ giao tiếp với Docker server hay daemon để làm mọi việc. Docker daemon cung cấp RESTful API, do đó client có thể dễ dàng giao tiếp thông qua dịch vụ này. Ngoài daemon, Docker còn cung cấp RESTful đối với Registry (nơi chúng ta lưu trữ nội bộ các images) và Docker Hub (public registry từ Docker Inc để mọi người có thể dễ dàng chia sẻ các image cho nhau). Do đó chúng ta có thể chạy Docker daemon và client trên cùng một máy hoặc kết nối từ một Docker client chạy ở local đến một remote daemon chạy ở một máy khác. Kiến trúc của Docker được mô tả như hình dưới:



Hình 2.1: Kiến trúc của Docker

### 2.2.2 Registry

Đây đơn giản là nơi mà Docker sẽ lưu trữ các image. Có hai loại registry: public và private.

- Docker Hub: Đây là một public registry lớn được vận hành bởi Docker, Inc. Có thể tạo account trên Docker Hub và sử dụng nó để lưu trữ cũng như chia sẻ image với cộng đồng.

Docker Hub hiện rất lớn mạnh, hầu hết các image đều có thể được tìm thấy ở đây. Một MySQL database, nginx server, hay một hệ điều hành CentOS mới nhất, tất cả đều dễ dàng tìm thấy tại Docker Hub.

- Private registry: Phù hợp với các cá nhân, tổ chức muốn xây dựng và lưu trữ các image một cách riêng tư. Có hai sự lựa chọn:
  - Sử dụng chức năng private của Docker Hub (sẽ phải trả phí).
  - Xây dựng và chạy một registry riêng.

Docker đã cung cấp sẵn cho chúng ta một open-source registry dưới dạng image, giúp ta dễ dàng xây dựng một registry nội bộ chỉ với vài câu lệnh đơn giản, đồng thời cung cấp RESTful APIs cho việc giao tiếp và quản lý registry.

### 2.2.3 Docker Image

Images là một thành phần quan trọng và tạo nên sức mạnh cho Docker. Chúng ta sẽ khởi chạy container từ image. Image có cấu trúc dạng lớp (layered), sử dụng Union file systems, chúng được build từng bước theo các câu lệnh được viết bởi người dùng, file nguồn dùng để build các image có tên là Dockerfile.

Chúng ta có thể xem image như "source code" để khởi chạy các container. Thế mạnh của chúng là có khả năng di động cao (highly portable), có thể chia sẻ, lưu trữ dễ dàng.

#### Cấu trúc của Docker image

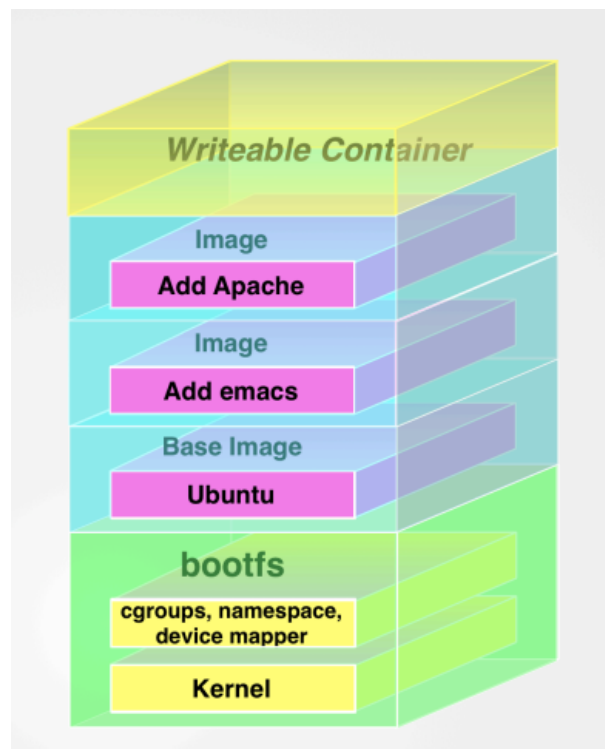
Như đã đề cập ở trên, Docker image có cấu trúc lớp, được tạo nên bởi các lớp filesystem (filesystems layered) nằm chồng lên nhau. Image này có thể là layered trên top của một image khác. Image bên dưới được gọi là image cha và image cuối cùng được gọi là base image. Hình bên dưới sẽ cho chúng ta thấy cấu trúc 4 stacked image layers của hệ điều hành Ubuntu 15.04

91e54dfb1179	0 B
d74508fb6632	1.895 KB
c22013c84729	194.5 KB
d3a1f33e8a5a	188.1 MB
ubuntu:15.04	

Image

Hình 2.2: Cấu trúc layers của image Ubuntu 15.04

Image là một thành phần chỉ đọc (read-only). Khi một container được khởi chạy từ một image, một writable layer sẽ được khởi tạo trên top của stack. Layer này còn được gọi là “container layer”. Tất cả các thay đổi từ việc chạy container - như ghi một file mới, chỉnh sửa các file đã có, xóa các files - sẽ được ghi tại layer này.



Hình 2.3: Writable layer

### Các lệnh thường dùng với Docker image

#### 1. Liệt kê tất cả các image

Để liệt kê tất cả các image, chúng ta sử dụng lệnh `docker images`.

**Danh sách 2.2.1: Liệt kê tất cả các image**

\$ docker images				
REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
anhnguyen/wls-deploy	latest	44b6aa50651a	2 weeks ago	2.382 GB
anhnguyen/debian-weblogic	latest	58bf7b1ec1be	2 weeks ago	2.382 GB
anhnguyen/debian-jdk8	latest	8e52c5069e46	2 weeks ago	777.5 MB
ubuntu	latest	c5f1cf30c96b	3 weeks ago	120.8 MB
anhnguyen/debian-tomcat8	latest	1386c86cb0db	4 weeks ago	799.7 MB
debian	jessie	47af6ca8a14a	7 weeks ago	125.1 MB

Bên trong Docker Hub (hoặc Docker registry do chúng ta quản lý), image được lưu trữ bên trong repository. Mỗi repository có thể chứa nhiều image, và mỗi image được xác định bởi tag.

Có hai loại repository:

- user repository: được quản lý bởi người dùng Docker.
- top-level repository: được quản lý bởi Docker, Inc.

User repository có thêm phần tên user phía trước tên repository. Ví dụ: anhnguyen/debian-weblogic, anhnguyen/debian-jdk8. Còn top-level repository chỉ có phần tên của repository. Ví dụ: ubuntu, debian.

#### 2. Pull image từ Docker Hub

Để pull một image từ Docker Hub, ta sử dụng lệnh `docker pull`

### Danh sách 2.2.2: Pull image ubuntu 15.10

```
$ docker pull ubuntu:15.10
15.10: Pulling from library/ubuntu
1db1b2807a8d: Pull complete
3ed5cfbbd021: Pull complete
d8c663cf1f0d: Pull complete
51d4a9b23a2c: Pull complete
a3ed95caeb02: Pull complete
Digest:
      sha256:61a0d92ca1885613e3d8b57844439a174af72d920fec5a854f38f18746289763
Status: Downloaded newer image for ubuntu:15.10
```

### 3. Tìm kiếm image

Ta dùng lệnh `docker search` để tìm kiếm thông tin về một image

### Danh sách 2.2.3: Tìm kiếm fedora image

```
$ docker search fedora
NAME                DESCRIPTION
STARS              OFFICIAL    AUTOMATED
fedora              Official Docker builds of Fedora
357                [OK]
dockingbay/fedora-rust Trusted build of Rust programming language ...
3                  [OK]
gluster/gluster-fedora Official GlusterFS image [ Fedora 21 + Glu...
2                  [OK]
startx/fedora       Simple container used for all startx based...
2                  [OK]
...
```

### 4. Push image đến Docker Hub

Ta dùng lệnh `docker push` để push image đến Docker Hub

### Danh sách 2.2.4: Push image đến Docker Hub

```
$ docker push anhnguyen/debian-jdk8
The push refers to a repository [docker.io/anhnguyen/debian-jdk8]
14dabc5d55c7: Pushed
82177678b4f3: Pushed
65f842a1ade7: Pushed
73085c79f27c: Pushed
c5d5018be72d: Pushed
48cdbc3e6d4: Pushed
5f70bf18a086: Mounted from library/ubuntu
c12ecfd4861d: Mounted from library/debian
latest: digest:
      sha256:672d68b940935dd6f020a3f30aaf818515010f540db83125d7f55f4ccb116430
      size: 6650
```

## 5. Xóa image

Ta dùng lệnh docker rmi để xóa image

### Danh sách 2.2.5: Xóa image ubuntu:15.10

```
$ docker rmi ubuntu:15.10
Untagged: ubuntu:15.10
Deleted:
      sha256:e8adb3f35815324001b425c89015dbe23596e94676dbcc57d8f4515385ed7945
Deleted:
      sha256:88ff8acbae770b8887a305cd1df2dbfb07423a24b671d51d8a8114bc701ef4f9
...
```

## Build một Docker image

Có hai cách để chúng ta có thể tạo ra một Docker image:

- Dùng lệnh docker commit
- Sử dụng lệnh docker build với một Dockerfile

Với cách sử dụng docker commit, chúng ta cần tạo ra một container, tạo các sự thay đổi cần thiết trên container này, và sau đó commit để tạo thành một image mới. Đây là cách không được khuyến khích sử dụng, bởi build với Dockerfile mạnh mẽ và hiệu quả hơn nhiều.

### Các lệnh của Dockerfile

Dockerfile là một file được Docker sử dụng để định nghĩa việc build các image. Chúng ta sẽ viết các lệnh trong Dockerfile và sử dụng lệnh docker build để build một image mới từ các lệnh này. Một số các lệnh quan trọng của Dockerfile sẽ được giới thiệu sau đây

#### 1. FROM

Xác định base image cho các lệnh tiếp theo. FROM là thường là lệnh đầu tiên của Dockerfile.

#### 2. RUN

Lệnh RUN sẽ thực thi câu lệnh được chỉ ra phía sau nó. Lệnh RUN có hai dạng:

- RUN <command> (dạng shell, câu lệnh sẽ được chạy trên shell - /bin/sh -c)
- RUN ["executable", "param1", "param2"] (dạng exec)

#### 3. CMD

Tương tự như lệnh RUN, nhưng CMD là lệnh được chạy khi container được khởi chạy (launch), lệnh RUN chạy ngay lúc image được build. Chỉ có thể có một lệnh CMD trong Dockerfile, nếu có nhiều hơn một lệnh, lệnh sau cùng sẽ được sử dụng.

Và còn một điều quan trọng là lệnh CMD cho phép chúng ta override khi khởi chạy container bằng lệnh docker run. Lệnh được chỉ ra trong docker run sẽ override lệnh CMD trong Dockerfile.

#### 4. ENTRYPOINT

Tương tự như lệnh CMD, chỉ khác biệt là ENTRYPOINT không cho phép override từ lệnh docker run khi khởi chạy container.

#### 5. EXPOSE

Lệnh EXPOSE sẽ giúp Docker biết container sẽ sử dụng cổng nào trong lúc chạy. Chú ý là các cổng được chỉ ra bởi lệnh EXPOSE không có khả năng truy cập từ host. Để truy cập từ host, chúng ta cần ánh xạ các cổng này ra một cổng của host bằng cách sử dụng các flag -p hoặc -P trong lệnh docker run.

#### 6. ENV

Lệnh ENV dùng để thiết lập các biến môi trường sử dụng trong quá trình build image.



Các biến môi trường này sẽ vẫn tồn tại cho bất kì container nào được tạo nên từ image này.

### 7. **ADD**

Lệnh ADD dùng để thêm file và thư mục từ môi trường build vào image. Các file và thư mục phải được đặt trong thư mục build, bởi lệnh ADD không có khả năng xử lý đối với các file ở ngoài thư mục build.

### 8. **COPY**

Tương tự như lệnh ADD, tuy nhiên lệnh COPY chỉ thuần khiết là copy các file từ môi trường build sang image. Không có khả năng nén hay xả nén, trong khi lệnh ADD có khả năng này. Khi chúng ta thực hiện việc build, thư mục build sẽ được upload đến Docker daemon, do đó cũng giống như lệnh ADD, không thể đặt các file cần copy ở ngoài thư mục build được.

### 9. **VOLUME**

Lệnh VOLUME dùng để tạo ra các volume cho bất kì container nào được tạo ra từ image này. Volume là một thư mục được thiết kế cho việc chia sẻ dữ liệu giữa host và một hoặc nhiều các container. Volume cho phép chúng ta thêm dữ liệu (như source code) vào một image mà không cần phải commit image.

### 10. **WORKDIR**

Dùng để thiết lập working directory cho container và cho các lệnh ENTRYPOINT, CMD. Khi container được launch, các lệnh ENTRYPOINT và CMD sẽ được thực thi từ thư mục này.

## 2.2.4 Docker Container

Container sẽ được khởi chạy (launch) từ image và chứa mọi thứ cần thiết giúp cho một ứng dụng có thể chạy được. Về đơn giản chúng ta có thể xem Image như mặt xây dựng (building) và đóng gói (packing) của Docker và Container là mặt thực thi (execution) của Docker.

Docker container có thể được chạy (run), bắt đầu (started), dừng (stopped), di chuyển (moved), cũng như xóa (deleted). Docker không quan tâm nội dung ở bên trong container khi nó được thực thi, dù đó là một web server, một database, hay một application server. Mỗi container đều được nạp (load) giống nhau như bất kỳ các container nào khác.

### Khởi chạy container từ một image

Chúng ta dùng lệnh *docker run* để khởi chạy container từ một image

#### Danh sách 2.2.6: Khởi chạy container từ ubuntu image

```
$ docker run -i -t ubuntu /bin/bash
Unable to find image 'ubuntu:latest' locally
latest: Pulling from library/ubuntu
6d28225f8d96: Pull complete
166102ec41af: Pull complete
d09bfba2bd6a: Pull complete
c80dad39a6c0: Pull complete
a3ed95caeb02: Pull complete
Digest: sha256:5718d664299eb1db14d87db7bfa6945b28879a67b74f36da3e34f5914866b71c
Status: Downloaded newer image for ubuntu:latest
root@eb0380a38f54:/#
```

Khi thực hiện lệnh này, đầu tiên Docker sẽ kiểm tra xem ubuntu image có ở máy local hay không. Nếu không thể tìm thấy ubuntu image ở local, Docker sẽ tìm kiếm image ở Docker Hub. Và khi tìm thấy image, hệ thống sẽ tải image về, lưu trữ tại máy local, sau đó sẽ tiến hành khởi chạy (launch).

Ở đây chúng ta sử dụng hai flag: *-i* và *-t*. Flag *-i* sẽ giữ STDIN (standard input) mở từ container. Flag *-t* sẽ gán một pseudo-tty vào container chúng ta vừa tạo ra. Hai flag kết hợp với nhau sẽ cung cấp cho chúng ta một interactive shell trong container. Cuối cùng, Docker sẽ launch một Bash shell với câu lệnh */bin/bash* mà chúng ta chỉ ra.

#### Danh sách 2.2.7: Container shell

```
root@eb0380a38f54:/#
```

### Làm việc với container

Khi đã đăng nhập vào container, chúng ta có thể làm bất cứ điều gì chúng ta thích trong nó.

### Danh sách 2.2.8: Làm việc với container

```
root@d666e9316a67:/# hostname
d666e9316a67
root@d666e9316a67:/# apt-get update
Get:1 http://archive.ubuntu.com/ubuntu xenial InRelease
[247 kB]
Get:2 http://archive.ubuntu.com/ubuntu xenial-updates
InRelease [94.5 kB]
...
root@d666e9316a67:/# apt-get install nano
```

### Đặt tên cho container

Docker sẽ tự động tạo ngẫu nhiên một cái tên cho container khi chúng ta tạo. Nếu chúng ta muốn chỉ định một cái tên, có thể dùng flag `--name`

### Danh sách 2.2.9: Đặt tên cho container

```
$ docker run --name myubuntu -i -t ubuntu /bin/bash
```

Tên của container là duy nhất, chúng ta không thể tạo hai container có cùng tên.

### Tạo container chạy ngầm (daemonized container)

Nếu container chúng ta cần chạy là một dịch vụ hoặc một máy chủ, không cần thiết phải tương tác với container. Chúng ta có thể dùng flag `-d` để thông báo với Docker rằng đây là container chạy ở background.

### Danh sách 2.2.10: Daemonized container

```
$ docker run --name mytomcat -d anhnnguyen/tomcat7
a86ffc86cf544815a2733ee4158fe0488436dbc583c9ee6823f7761feabd7a11
```

Có thể kiểm tra trạng thái của container vừa chạy thông qua lệnh

### Danh sách 2.2.11: Kiểm tra trạng thái container

```
$ docker ps -l
```

CONTAINER ID	IMAGE	COMMAND	CREATED
	STATUS	PORTS	NAMES
a86ffc86cf54	anhnguyen/tomcat7	"/usr/share/tomcat7/b"	2 minutes ago
	Up 2 minutes	8080/tcp	mytomcat

Khi chạy container ở chế độ background, việc xem điều gì xảy ra bên trong container là rất quan trọng. Nhờ đó, chúng ta có thể xác định được container có chạy đúng theo ý muốn hay không.

### Danh sách 2.2.12: Xem logs của một daemonized container

```
$ docker logs mytomcat
...
INFO: Deploying web application directory /var/lib/tomcat7/webapps/ROOT
May 26, 2016 12:54:29 AM org.apache.catalina.startup.HostConfig
deployDirectory
INFO: Deployment of web application directory /var/lib/tomcat7/webapps/
ROOT has finished in 764 ms
May 26, 2016 12:54:29 AM org.apache.coyote.AbstractProtocol start
INFO: Starting ProtocolHandler ["http-bio-8080"]
May 26, 2016 12:54:29 AM org.apache.catalina.startup.Catalina start
INFO: Server startup in 820 ms
```

Có thể sử dụng flag `-tail` để xem những dòng cuối của log file

### Danh sách 2.2.13: Xem logs với tail flag

```
$ docker logs --tail 10 mytomcat
```

## Start, stop container

Container có thể được stop, start, restart tương tự như với các service của hệ thống.

### Danh sách 2.2.14: start stop restart container

```
$ docker stop mytomcat
mytomcat
$ docker start mytomcat
mytomcat
$ docker restart mytomcat
mytomcat
```

### Inspecting một container

Đây là một lệnh quan trọng, sử dụng khi chúng ta muốn xem toàn bộ thông tin về một container. Bao gồm các thông tin cấu hình, tên, cấu hình mạng, trạng thái...

### Danh sách 2.2.15: Inspect mytomcat container

```
$ docker inspect mytomcat
[
  {
    "Id": "a86ffc86cf544815a2733ee4158fe0488436dbc583c9ee6823f7761feabd7a1",
    "Created": "2016-05-26T00:17:06.147994574Z",
    "Path": "/usr/share/tomcat7/bin/catalina.sh",
    "Args": [
      "run"
    ],
    "State": {
      "Status": "exited",
      ....
    },
    ...
  }
]
```

### Xóa container

Để xóa một container, chúng ta sử dụng lệnh *docker rm*. Chú ý là chúng ta không thể xóa một container đang chạy, cần phải stop trước bằng lệnh *docker stop* hay *docker kill*

**Danh sách 2.2.16: Xóa mytomcat container**

```
$ docker rm mytomcat
```

## 2.3 Docker Compose và Docker Machine

### 2.3.1 Docker Compose

Docker compose là một công cụ cho việc định nghĩa và chạy một ứng dụng phức tạp với Docker. Một ứng dụng phức tạp sẽ gồm nhiều container liên kết với nhau để chạy. Với Docker Compose, tất cả những gì mà chúng ta cần làm là định nghĩa những containers trong duy nhất một tập tin. Tập tin này được đặt tên là *docker-compose.yml*.

Tập tin *docker-compose.yml* có 2 phiên bản:

- Phiên bản 1 được hỗ trợ bởi Docker Compose đến phiên bản 1.6.x.

### Danh sách 2.3.1: Một tập tin docker-compose.yml phiên bản 1 đơn giản

```
web:
  image: intern/debian-java-tomcat:8
  ports:
    - "8088:8080"
  volumes:
    - app:/opt/tomcat/webapps
  links:
    - db
phpmyadmin:
  image: phpmyadmin/phpmyadmin
  ports:
    - "1234:80"
  links:
    - db
db:
  build: ./mysql
  ports:
    - "3306:3306"
  environment:
    MYSQL_ROOT_PASSWORD: 123456
    MYSQL_USER: dev
    MYSQL_PASSWORD: 123456
    MYSQL_DATABASE: myapp
```

- Phiên bản 2 được hỗ trợ bởi Docker Compose 1.6.0 trở lên và yêu cầu Docker Engine 1.10.0 trở lên.

### Danh sách 2.3.2: Một tập tin docker-compose.yml phiên bản 2 đơn giản

```
version: '2'
services:
  web:
    image: intern/debian-java-tomcat:8
    ports:
      - "8088:8080"
    volumes:
      - app:/opt/tomcat/webapps
    links:
      - db
  phpmyadmin:
    image: phpmyadmin/phpmyadmin
    ports:
      - "1234:80"
    links:
      - db
  db:
    build: ./mysql
    ports:
      - "3306:3306"
    environment:
      MYSQL_ROOT_PASSWORD: 123456
      MYSQL_USER: dev
      MYSQL_PASSWORD: 123456
      MYSQL_DATABASE: myapp
```

Phía trên là tập tin docker-compose.yml cho một ứng dụng đơn giản gồm 3 containers.

- Đầu tiên là container web, là một ứng dụng web đơn giản chạy trên web server Tomcat 8. Image của Tomcat 8 được tạo từ trước.
- Kế đến là container phpmyadmin. Phpmyadmin là phần mềm mã nguồn mở được viết bằng php dùng để quản lý cơ sở dữ liệu MySQL thông qua giao diện web thay vì cửa sổ dòng lệnh. Image của phpmyadmin được pull từ Docker Hub - một public docker registry được dùng để chia sẻ những image. Container được chờ nghe trên cổng 1234 và liên kết đến container db.
- Cuối cùng là container db. Image db được build từ một Dockerfile được đặt trong thư mục mysql. Container này đã được thêm dữ liệu vào từ trước, dữ liệu này được lưu trong tập



tin init.sql

### Danh sách 2.3.3: Dockerfile trong thư mục mysql

```
FROM mysql:5.7
COPY init.sql /docker-entrypoint-initdb.d
```

### Danh sách 2.3.4: Nội dung của tập tin init.sql

```
create table person(
  id int not null auto_increment primary key,
  name varchar(20),
  studentid int,
  university varchar(30)
);

insert into person(name,studentid,university) values('An Nguyen', 51200161,
'HCMUT');
insert into person(name,studentid,university) values('Anh Nguyen', 51200071,
'HCMUT');
```

Để chạy ứng dụng, ta truy cập vào thư mục chứa docker-compose.yml và sử dụng câu lệnh sau:

### Danh sách 2.3.5: Chạy ứng dụng trên với Docker compose

```
$ docker-compose up -d
```

Với câu lệnh trên, Docker compose sẽ đọc tập tin docker-compose.yml từ đó tạo và khởi động những container được định nghĩa. Docker compose đang chạy ngầm với option -d nên để xem ứng dụng chạy xong chưa sử dụng câu lệnh sau:

### Danh sách 2.3.6: Xem logs của Docker compose

```
$ docker-compose logs
```

Mở browser, truy cập vào địa chỉ localhost:8080 để xem kết quả chạy của ứng dụng. Nếu muốn thêm dữ liệu vào cơ sở dữ liệu, truy cập vào địa chỉ localhost:1234/phpmyadmin đăng nhập vào với username: dev và password: 123456.

Để dừng ứng dụng sử dụng câu lệnh sau:

#### **Danh sách 2.3.7: Dừng ứng dụng với Docker compose**

```
$ docker-compose stop
```

Để xóa những containers sử dụng câu lệnh sau:

#### **Danh sách 2.3.8: Xóa những containers với Docker compose**

```
$ docker-compose down
```

### **2.3.2 Docker Machine**

Docker Machine là một công cụ để tạo một máy ảo và cài đặt Docker Engine lên trên nó. Docker machine không thể tự tạo máy ảo mà phải thông qua driver của các nhà cung ứng. Dưới đây là danh sách một số nhà cung ứng hỗ trợ Docker machine:

- Oracle Virtualbox
- VMware Fusion
- Microsoft Hyper-V
- Amazon Web Services
- Digital Ocean
- OpenStack
- ...

Khi đã cài đặt xong Docker Machine. Ta sẽ tiến hành sử dụng Docker Machine để tạo một máy ảo thông qua driver của Oracle Virtualbox và cài đặt Docker Engine lên trên nó bằng câu lệnh sau:

### Danh sách 2.3.9: Tạo một Docker Machine tên là default

```
$ docker-machine create --driver virtualbox default
```

Câu lệnh trên sẽ tiến hành tải về một hệ điều hành Linux Boot2Docker - một hệ điều hành được thiết kế riêng để chạy Docker containers và lưu trong thư mục `~/.docker/machine/cache`. Nó chạy hoàn toàn trên RAM với dung lượng chỉ 24MB và khởi động trong vòng 5s. Trên hệ điều hành này đã cài đặt sẵn Docker. Trong những lần tạo Docker machine khác ta không cần phải tải Boot2Docker về nữa.

Khi đã tải xong Boot2Docker, Docker Machine sẽ tạo một máy ảo tên là default và cài đặt Boot2Docker lên trên nó.

Bây giờ ta sẽ kiểm tra xem đã tạo thành công Docker Machine bằng câu lệnh dưới đây:

### Danh sách 2.3.10: Danh sách các Docker Machine đã tạo

```
$ docker-machine ls
NAME ACTIVE DRIVER STATE URL SWARM DOCKER ERROR
default - virtualbox Running tcp://192.168.99.100:2376
```

Ta thấy rằng Docker Machine default đang chạy. Để sẽ xem các biến môi trường của Docker Machine default sử dụng câu lệnh sau:

### Danh sách 2.3.11: Các biến môi trường của Docker Machine default

```
$ docker-machine env default
export DOCKER_TLS_VERIFY="1"
export DOCKER_HOST="tcp://192.168.99.100:2376"
export DOCKER_CERT_PATH="~/.docker/machine/machines/default"
export DOCKER_MACHINE_NAME="default"
```

Khi đã tạo xong, ta đã có một Docker Server đang chạy từ xa, phần việc còn lại là tiến hành kết nối máy tính của mình (đã cài đặt Docker Client) với Docker Server ta mới vừa tạo bằng Docker Machine.

Tiến hành kết nối đến Docker Machine default. Ta sẽ phải lặp lại bước này trên các cửa sổ terminal khác nếu muốn kết nối terminal đó với Docker Machine default.

### Danh sách 2.3.12: Kết nối với Docker Machine default

```
$ eval $(docker-machine env default)
```

Kiểm tra xem đã kết nối thành công hay chưa:

### Danh sách 2.3.13: Kiểm tra kết nối với Docker Machine default

```
$ docker-machine ls
NAME ACTIVE DRIVER STATE URL SWARM DOCKER ERROR
default * virtualbox Running tcp://192.168.99.100:2376
```

Dấu "\*" ở cột ACTIVE cho ta biết là đã kết nối thành công với Docker Machine default. Bây giờ, ta có thể làm việc với Docker Image, Docker Container thông qua tập lệnh Docker-CLI.

Tắt và khởi động lại Docker Machine default bằng những câu lệnh sau:

### Danh sách 2.3.14: Tắt và khởi động lại Docker Machine default

```
$ docker-machine stop default
$ docker-machine start default
```

Xóa Docker Machine default bằng câu lệnh sau:

### Danh sách 2.3.15: Xóa Docker Machine default

```
$ docker-machine rm -f default
```

Với Docker Machine, ta dễ dàng cài đặt Docker theo mô hình Docker Client/Server trên hai máy khác nhau:

- Docker Client kết nối đến Docker Server thông qua đó để làm việc với Docker Image, Docker Container. Bản thân Docker Client không thể làm việc trực tiếp với Docker Image, Docker Container, mà nó chỉ nói cho Docker Server cần phải làm gì bằng tập lệnh Docker-CLI.

- Docker Server được cài đặt bằng Docker Machine. Nó làm việc trực tiếp với Docker Image, Docker Container như là tạo image, chạy và quản lý container.

## 2.4 Docker networking

Khi chạy Docker Container trên một single-host (máy đơn) hay một cụm máy (cluster of machines), networking luôn là vấn đề mà chúng ta phải đối mặt. Làm sao để các container trên cùng một máy hay các máy khác nhau có thể giao tiếp trao đổi dữ liệu với nhau một cách thuận tiện.

Đối với một máy đơn, câu hỏi được đặt ra là trao đổi dữ liệu qua shared volume hay trao đổi dữ liệu qua networking (HTTP-based hoặc tương tự). Như đã đề cập ở phần trước, shared volume là phương pháp shared dữ liệu đơn giản host và container - container có khả năng mount các folder của host. Đây là cách đơn giản, tốc độ rất nhanh, dễ sử dụng, tuy nhiên sẽ gặp nhiều khó khăn khi chuyển từ một máy đơn sang một cụm máy.

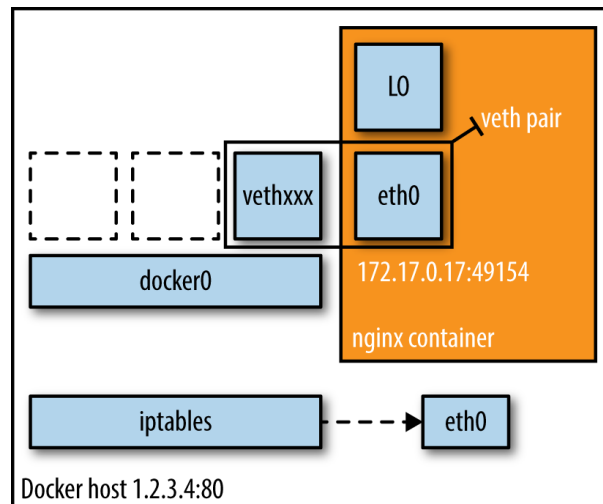
Khi triển khai Docker lên một cụm máy, sẽ có nhiều vấn đề cần xem xét là giao tiếp giữa các container trên cùng một host và trên các host khác nhau. Kể cả các mặt về performance và security cũng phải được xem xét tới.

### 2.4.1 Docker Networking

Docker networking có hai dạng chính đó là bridge networking và host networking.

#### Bridge Networking

Ở chế độ này, Docker daemon tạo ra *docker0*, một Ethernet bridge ảo tự động chuyển tiếp (forward) các package từ bất kỳ network interface nào được attach với nó. Bridge là chế độ mặc định của Docker. Nếu không sử dụng *-P* (publish tất cả các port của container) hoặc *-p host\_port:container\_port* (publish tất cả các port được chỉ định) trong câu lệnh *docker run*, các IP packer sẽ không được định tuyến ra bên ngoài host. Hình dưới đây mô tả bridge mode khi chạy câu lệnh *docker run -d -P nginx:1.9.1*.



Hình 2.4: Docker bridge mode networking

## Host Networking

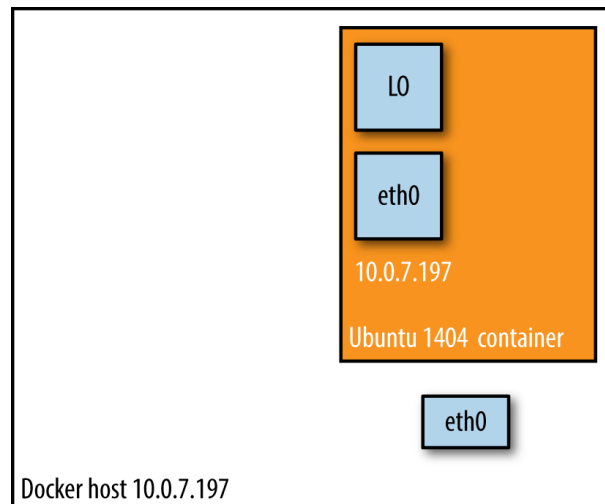
Chế độ này sẽ vô hiệu hóa network isolation của Docker container. Bởi vì container sẽ share networking namespace của host, container sẽ nằm cùng IP với host, chúng ta sẽ phải xác định nó thông qua port mapping.

```
[beo@beo ~]$ sudo docker run -d --net=host ubuntu:14.04 tail -f /dev/null
b77c4f6e54605f6da0e436852b799461d4ab5ba6c5d0984c6591a6c45b196879
[beo@beo ~]$ ip addr | grep -A 2 wlp3s0
3: wlp3s0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UP qlen 1000
    link/ether 74:e5:43:6e:4e:33 brd ff:ff:ff:ff:ff:ff
    inet 192.168.1.10/24 brd 192.168.1.255 scope global dynamic wlp3s0
        valid_lft 86086sec preferred_lft 86086sec
    inet6 fe80::76e5:43ff:fe6e:4e33/64 scope link
[beo@beo ~]$ sudo docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS              NAMES
b77c4f6e5460        ubuntu:14.04       "tail -f /dev/null" About a minute ago  Up About a minute  -                  admiring_ritchie
[beo@beo ~]$ sudo docker exec -it b77c4f6e5460 ip addr | grep wlp3s0
3: wlp3s0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc mq state UP group default qlen 1000
    inet 192.168.1.10/24 brd 192.168.1.255 scope global dynamic wlp3s0
```

Hình 2.5: Docker run with host networking

Ở ví dụ trên, khi chạy ở chế độ container có cùng IP address với host: 192.168.1.10.

Hình ảnh dưới đây mô tả host mode networking khi run container ubuntu:14.04. Container thừa kế IP address từ host của nó. Chế độ này chạy nhanh hơn bridge mode vì nó không có routing overhead, nhưng có tình bảo mật yếu hơn khi container được expose trực tiếp đến public network.



Hình 2.6: Docker host mode networking

### 2.4.2 Docker Multihost Networking

Khi cài đặt hệ thống Docker trên nhiều máy (cluster), có nhiều vấn đề đặt ra như: Làm sao để các container giao tiếp được với nhau khi chạy trên các host khác nhau? Làm sao để quản lý việc giao tiếp giữa các container với bên ngoài? Gán IP address cho container trong cluster? Chính sách bảo mật? v.v..

Thật may là hiện nay có khá nhiều các công cụ mạnh hỗ trợ chúng ta thực hiện những việc trên. Mặc định Docker cung cấp Overlay driver cho multi-host networking. Ngoài ra còn rất nhiều các công nghệ khác như: Flannel từ CoreOS, Weave từ Weaveworks, Open vSwitch, Pipework...

Docker, Inc. mua lại software-defined networking (SDN) từ startup SocketPlane và đổi tên thành Docker Overlay Driver. Từ phiên bản 1.9 trở về sau, Docker mặc định hỗ trợ multihost networking bằng Overlay driver. Overlay Driver mở rộng chế độ bridge mode bằng giao tiếp peer-to-peer. Sử dụng một key-value store backend để phân phối trạng thái của cluster, được hỗ trợ bởi Consul, etcd, và ZooKeeper.

## 2.5 Container Orchestration

Container orchestration là một khái niệm bao gồm các hoạt động quản lý một hệ thống container như:

- Health checks: kiểm tra trạng thái container (running, exited).
- Organizational primitives: tổ chức phân nhóm các container( tương đương label trên Kubernetes hoặc group trên Marathon).
- Autoscaling: Tự động tăng hay giảm số lượng các bản sao của container dựa vào số lượng yêu cầu (request).
- Upgrade/rollback strategies: upgrade, rollback container.
- Service discovery: Bao gồm hai dịch vụ chính là *register* và *lookup*. Container được đăng ký đến service discovery. Các yêu cầu tìm kiếm container hay điều hướng request đến container sẽ do service discovery đảm nhận.
- Quản lý lượng tài nguyên cấp phát cho container.
- ...

Các container orchestrator phổ biến hiện nay:

- Docker Swarm
- Kubernetes
- Mesos Marathon
- Hashicorp Nomad
- Amazon ECS
- Fleet

Dưới đây chúng tôi xin cung cấp thêm thông tin cho ba container orchestrator phổ biến nhất. Trong đó có công nghệ mà chúng tôi chọn sử dụng cho hệ thống của mình - Kubernetes.

### 2.5.1 Docker Swarm

Swarm là công cụ container orchestration của chính Docker. Sử dụng API và networking tiêu chuẩn của Docker. Một Swarm bao gồm một manager và các worker node chạy các service.



Swarm khá đơn giản để cài đặt : sử dụng Docker Engine để initialise một master node và Swarm có tập lệnh để dễ dàng thêm các worker node vào cluster và làm việc với chúng.

Khi một swarm được chạy, các service được đặc tả bởi Docker Compose. Khi service được khởi chạy nó sẽ được deploy qua các host của swarm thay vì chỉ một single-host như Compose thông thường.

Docker Inc đã nỗ lực đầu tư khá nhiều vào Swarm và nó đã có những thay đổi nhanh chóng. Tuy nhiên cho đến nay, Swarm vẫn được coi là chỉ phù hợp để thí nghiệm hay triển khai quy mô nhỏ. Có thể vì những hạn chế nhìn thấy được ở những bản phát hành đầu tiên của Swarm so với Kubernetes and Mesos.

### 2.5.2 Kubernetes

Kubernetes được dựa trên những kinh nghiệm của Google trong việc chạy các tải công việc (workload) ở quy mô rất lớn trong suốt hơn 15 năm. Khác với Docker Swarm, được mở rộng từ single host Docker, Kubernetes xuất phát bản thân nó đã là một cluster.

Kubernetes định nghĩa các khái niệm của chính nó. Nếu muốn làm quen với Kubernetes, nhất thiết phải làm quen với các khái niệm này. Một Kubernetes cluster bao gồm:

- Master: handle các lời gọi API, giao các tải công việc (assigns workloads) và duy trì các trạng thái cấu hình (maintains configuration state).
- Node (tên cũ là Minion): Nơi chạy các tải công việc
- Pod: Đơn vị cơ bản của Kubernetes, bao gồm một hoặc nhiều các container được deploy trên một host, cùng nhau thực hiện một công việc, có một địa chỉ IP và flat networking bên trong pod.
- Service: front end và load balancer cho pod, cung cấp địa chỉ IP để truy cập đến các pod. Sức mạnh của service nằm ở chỗ những thay đổi bên trong pod không làm thay đổi trạng thái interface của service.
- Replication controller: Chịu trách nhiệm luôn duy trì một số lượng cố định các replicas (bản sao) của một pod.

- Label: Thẻ dạng key-value. Được sử dụng trong hệ thống để phân biệt, gom nhóm các pod, replication controller và service.

Kubernetes được xây dựng theo nền tảng module. Ví dụ, ta có thể chọn được nhiều các công cụ networking khác nhau Flannel, Weave, Calico, OpenVSwitch... Tính module còn được mở rộng đến containr. Kubernetes không hoàn toàn chỉ phục vụ cho Docker, nó có thể được mở rộng để sử dụng rkt hay các định dạng container khác.

Kubernetes phù hợp để sử dụng cho các cluster cỡ trung bình đến lớn, chạy các ứng dụng phức tạp. Công việc cài đặt, sử dụng sẽ tốn nhiều nỗ lực hơn so với Docker Swarm nhưng bù lại cluster sẽ có tính linh động cao (flexibility), mô-dun hóa (modularity) và thích hợp cho quy mô lớn hơn.

### 2.5.3 Mesos và Marathon

Apache Mesos ra đời trước Docker và được mô tả như một nhân của hệ thống phân tán (distributed systems kernel). Nó là một cluster manager, giúp các tài nguyên hệ thống trở nên available cho các framework chạy trên Mesos cluster.

Marathon là một framework như vậy, chuyên chạy các ứng dụng, bao gồm các container trên Mesos cluster.

Cùng với nhau, Mesos và Marathon cung cấp những gì mà Kubernetes làm được. Hơn thế nữa, nó còn cho phép chạy các non-containerised workload bên cạnh các container.

Cả Mesos và Marathon đều được thiết kế cho quy mô hàng trăm đến hàng ngàn các node. Nó yêu cầu tài nguyên và nguồn lực quản lý lớn, không phù hợp cho cluster nhỏ. Công việc cài đặt cũng là rất phức tạp, đặc biệt khi so sánh với Docker Swarm.

Mesos đã được chứng minh tại những hệ thống hàng chục ngàn node trong production. Cả Mesos và Marathon đề ra đời trước Kubernetes và Swarm. Họ có thời gian làm việc thực tế và lâu dài hơn với các hệ thống lớn. Mesos và Marathon, có lẽ là sự lựa chọn tốt cho những hệ thống chạy non-containerised workload bên cạnh container và ta muốn một cái gì đó đã được chứng minh ở quy mô rất lớn.

## Chương 3 OPENSIFT

### 3.1 Giới thiệu

OpenShift là một dịch vụ nền tảng điện toán đám mây của hãng Red Hat. Openshift được xây dựng từ Kubernetes của Google, sử dụng Docker containers và các công cụ DevOps giúp tăng tốc việc phát triển ứng dụng.

OpenShift bao gồm các sản phẩm, dịch vụ chính:

- **OpenShift Origin** là một dự án cộng đồng mã nguồn mở được sử dụng để xây dựng OpenShift Online, OpenShift Dedicated, và OpenShift Container Platform. Origin được xây dựng từ nền tảng Docker và Kubernetes container cluster management. Đây cũng là dự án mà chúng tôi tập trung nghiên cứu để xây dựng private Platform-as-a-Service (PaaS) mà chúng tôi sử dụng để tích hợp vào Pipeline.
- **OpenShift Online** là một dịch vụ đám mây của hãng RedHat cho việc phát triển ứng dụng và hosting cho các dịch vụ. Online được xây dựng từ mã nguồn của dự án Origin. Dịch vụ này hỗ trợ nhiều ngôn ngữ, framework cũng như cơ sở dữ liệu. Người phát triển phần mềm có thể sử dụng Git để triển khai ứng dụng bằng các ngôn ngữ khác nhau.
- **OpenShift Container Platform** (tên gọi cũ là **OpenShift Enterprise**) một sản phẩm giúp các nhà phát triển ứng dụng hay các cluster administrator xây dựng nên các PaaS của riêng họ. Tương tự như Origin nhưng đây là sản phẩm thương mại của RedHat.

### 3.2 OpenShift Origin

#### 3.2.1 OpenShift v3 và Docker

OpenShift phiên bản 3 đã mang đến rất nhiều sự thay đổi trong kiến trúc, giới thiệu các khái niệm và thành phần mới. Ở phiên bản này, OpenShift được xây dựng xung quanh việc ứng dụng được chạy trên Docker container, scheduling và management được hỗ trợ bởi Kubernetes - công cụ container orchestration và management đến từ Google.

OpenShift ra mắt lần đầu tiên vào năm 2011 và luôn dựa trên Linux container để triển

khai và chạy các ứng dụng. Ở những phiên bản 1 và 2, RedHat sử dụng các container runtime platform và container orchestration engine của chính họ. Giữa năm 2013, RedHat đã quyết định hỗ trợ Docker cho container runtime và packaging format. Đồng thời họ cũng hoàn toàn hỗ trợ Docker trên Red Hat Enterprise Linux 7 (cũng như Fedora và CentOS), điều này đã đặt nền móng cho dự ra đời của OpenShift 3.

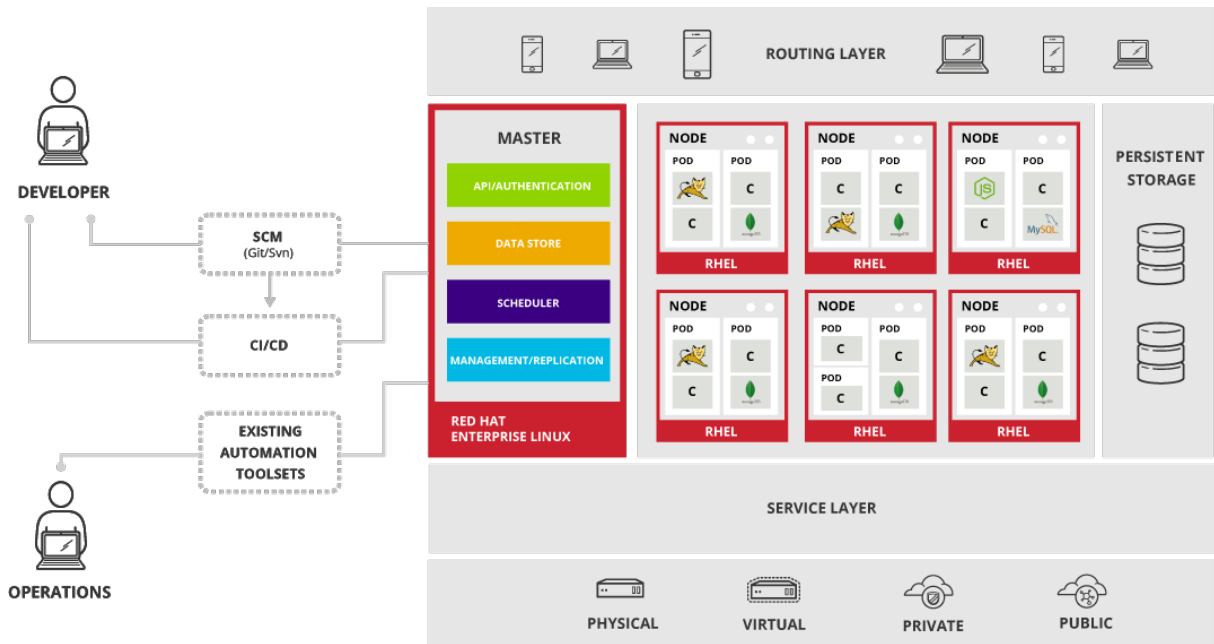
Chỉ có container runtime như Docker là không đủ cho OpenShift. Một ứng dụng không chỉ chạy trên một container và một ứng dụng production không thể chỉ deploy trên một single host. Khả năng orchestrate multiple containers trên multiple hosts là một yêu cầu quan trọng cho OpenShift. Từ cuối năm 2013 đến đầu năm 2014, RedHat đã tìm kiếm một số các giải pháp và họ đã lựa chọn Kubernetes. Theo RedHat, là bởi Kubernetes có "công nghệ tuyệt vời", "một đội ngũ tuyệt vời" và "cộng đồng tuyệt vời".

### 3.2.2 Kiến trúc

OpenShift v3 là một hệ thống phân tầng được thiết kế dựa trên Docker và Kubernetes. OpenShift tập trung chủ yếu vào việc làm sao để dễ dàng hóa việc xây dựng các thành phần ứng dụng trên nền Kubernetes và Docker.

Docker cung cấp trừu tượng hóa cho việc đóng gói (packaging) và tạo (creating) các Linux-based container image. Kubernetes cung cấp cluster management và orchestrates containers trên nhiều host. OpenShift Origin thêm vào đó:

- Quản lý source code, build, và deployment cho nhà phát triển.
- Quản lý các image
- Quản lý ứng dụng khi scale
- Người vận hành hệ thống có thể theo dõi cho việc tổ chức một số lượng lớn các nhà phát triển



Hình 3.1: Kiến trúc tổng quan của OpenShift Origin

OpenShift Origin có kiến trúc microservice dựa trên các thành phần nhỏ hơn, các đơn vị tách rời làm việc cùng nhau. Nó có thể chạy trên top (hoặc alongside) với một Kubernetes cluster, với dữ liệu về các objects được lưu trữ tại etcd, một clustered key-value store. Những dịch vụ này được phân chia theo chức năng:

- REST APIS, được expose các core objects
- Controllers, đọc các API, áp dụng sự thay đổi đến các đối tượng khác, báo cáo trạng thái hoặc ghi trở lại chính đối tượng.

Người sử dụng sẽ gọi các REST API để thay đổi trạng thái của hệ thống. Các controller sử dụng REST API để đọc trạng thái mà người dùng mong muốn, thực hiện các thay đổi và tạo ra sự đồng bộ cho các thành phần khác của hệ thống. Ví dụ, khi người dùng yêu cầu build. Build controller sẽ tạo ra một new build, chạy một process trên cluster để thực hiện build. Sau khi quá trình build hoàn tất, build controller cập nhật lại build object thông qua REST API và người dùng nhìn thấy yêu cầu của họ đã được hoàn thành.

### 3.2.3 Các thành phần cơ sở

#### Kubernetes Infrastructure

Một Kubernetes cluster bao gồm một hoặc nhiều *master* và một tập các *node*. Ở phiên bản hiện tại (v1.3), OpenShift Origin sử dụng Kubernetes 1.3 và Docker 1.10.

- **Masters**

Master gồm một hoặc nhiều máy chứa các thành phần của master, bao gồm API server, controller manager server, và etcd. Master quản lý các nodes trong hệ thống Kubernetes và schedule các pod chạy ở node nào.

#### High Availability Masters

Để giảm thiểu lỗi từ master và nâng cao tính sẵn sàng của hệ thống, chúng ta có thể cấu hình một master trở thành high availability (HA). Khi HA được cấu hình, API server sẽ được quản lý bởi HAProxy và HAProxy lúc này sẽ chịu trách nhiệm cân bằng tải giữa các endpoints.

- **Nodes**

Node là nơi cung cấp môi trường thực thi (runtime environments) cho các containers. Mỗi node trong Kubernetes cluster đều được quản lý bởi master. Node còn bao gồm các dịch vụ cần thiết để chạy các pod, bao gồm Docker service, một kuberlet, và một service proxy. Một node trong OpenShift Origin có thể được tạo ra từ cloud provider, physical system, hoặc virtual system. Mỗi node đều có một **node object** chứa các đặc tả về node đó. Master sử dụng thông tin từ node object để kiểm tra tình trạng (healthy check) cho node. Một node chỉ sẵn sàng hoạt động khi kết quả kiểm tra là thành công.

#### Image Registry

OpenShift Origin có thể sử dụng Docker Hub, private registry chạy bởi bên thứ ba và registry được tích hợp vào OpenShift Origin.

- **Integrated OpenShift Origin Registry**

OpenShift Origin cung cấp một container registry tích hợp trong hệ thống. Điều này cho

phép người dùng mặc định luôn có một nơi để build và push các image của họ.

Khi một image mới được push đến integrated registry, registry sẽ thông báo cho OpenShift Origin về image mới, truyền tất cả các thông tin về nó như namespace, name, image metadata. Các phần khác của OpenShift cũng có thể bị tác động (react) bởi image mới này, như tạo một build mới hoặc một deployment.

- **Third Party Registries**

OpenShift Origin có thể sử dụng registry container từ bên thứ ba, nhưng không giống như integrated registry. Các image khi được push lên registry của bên thứ ba sẽ không tự động thông báo đến OpenShift Origin. Chúng ta sử dụng lệnh `oc import-image <stream>` tạo một image stream. Giúp OpenShift phát hiện, cũng như phản ứng (react) với những thay đổi của image.

### 3.2.4 Các khái niệm cốt lõi

#### Pod và Service

- **Pod**

OpenShift Origin sử dụng khái niệm *pod* từ Kubernetes, là tập hợp một hoặc nhiều container được deploy cùng nhau trên một host, và đây cũng là đơn vị nhỏ nhất có thể được define, deploy, và manage.

Pod cũng tương đương với một machine instance (physical hoặc virtual). Mỗi pod được cấp phát địa chỉ IP nội bộ riêng, sở hữu không gian cổng riêng, và các container trong một pod có thể chia sẻ local storage và networking.

Pod có một lifecycle. Đầu tiên nó được define, sau đó được gán cho một node để chạy, và sẽ chạy cho đến khi kết thúc hoặc bị gỡ bỏ. Tùy thuộc vào policy và exit code, pod có thể được xóa sau khi exiting, hoặc có thể được giữ lại để có thể truy cập logs cho các container.

Số pod tối đa trên mỗi OpenShift Origin node host là khoảng 110.

- **Service**

Một Kubernetes *service* hoạt động như một internal load balancer. Nó xác định một tập replicated pods theo thứ tự để đại diện nhận các kết nối đến các pod này. Pod có thể được thêm vào hoặc gỡ bỏ khỏi service trong khi service vẫn tồn tại và luôn không đổi, điều

này cho phép bất cứ điều gì phụ thuộc vào service có thể thao khảo đến nó tại một địa chỉ cố định.

Mỗi service được gán với một địa chỉ IP và cổng. Khi truy cập qua service, proxy sẽ gọi đến pod thích hợp đằng sau nó.

### Project và User

- **User**

Trong OpenShift Origin, mỗi đối tượng user đại diện cho một tác nhân và có thể được gán các quyền riêng biệt trong hệ thống.

Các kiểu user chính:

<b>Regular users</b>	Người dùng bình thường của hệ thống. Là User tương tác với OpenShift Origin nhiều nhất.
<b>System users</b>	Nhóm quản trị hệ thống: bao gồm cluster administrator, per-node user, etc.
<b>Service accounts</b>	Là các user đặc biệt được gắn liền với project. Một số được tạo ra khi project được tạo. Một số được administrator tạo ra cho phép truy cập vào nội dung các mỗi project.

- **Namespace**

Kubernetes namespace cung cấp một cơ chế để phạm vi các tài nguyên (scope resources) trong cluster. Một project trong OpenShift Origin là một Kubernetes namespace với các chú thích thêm (additional annotations).

- **Project**

Project là cơ bản và quan trọng trong OpenShift Origin. Một project cho phép người dùng tổ chức và quản lý các nội dung của họ và tách biệt với những nội dung khác.

Cluster administrator có thể tạo các project và ủy quyền quản trị cho user. Đồng thời cũng có thể cho phép các developer tự tạo project cho riêng họ.

Developer và administrator có thể tương tác với project qua CLI (OpenShift Client) hoặc giao diện web (web console).



### Build và Image Stream

- **Build**

Build là một process tạo ra các runnable image để sử dụng trong OpenShift. BuildConfig là đối tượng được sử dụng để định nghĩa cho tiến trình build này.

OpenShift Origin gọi xuống Kubernetes để tạo Docker-formatted container từ build image và sau đó push image được tạo đến *container registry*.

OpenShift Origin build system hỗ trợ các *build strategy* khác nhau dựa trên build API khác nhau. Có ba build strategy chính.

- Docker build: Được invoke từ các câu lệnh *docker build*, do đó nó cần một Docker-file và tất cả các artifact cần thiết cho việc build.
- Source-to-Image (S2I) build: Source-to-Image (S2I) là một công cụ được RedHat phát triển riêng cho quá trình build các Docker-formatted container image. Nó xây dựng các ready-to-run image bằng cách injecting application source và container image và đóng gói thành một image mới. S2I còn hỗ trợ tự động số build, sử dụng lại các previously downloaded dependency, previously built artifacts, etc.  
Chúng tôi sử dụng Source-to-Image để build các image được sử dụng trong luận văn này. Do đó, chúng tôi sẽ đề cập cụ thể về Source-to-Image và build image với Source-to-Image hơn trong phần sau.
- Custom build: Cho phép developer định nghĩa cụ thể hơn build process. Sử dụng builder image của chính họ hay tùy chỉnh lại các build process.

Mặc định chỉ có Docker build và S2I build được hỗ trợ.

- **Image Streams**

Một image stream bao gồm một hoặc một nhiều các Docker image được xác định bởi các tag. Image stream đại diện cho một góc nhìn ảo duy nhất (single virtual view) của image. Image Stream được sử dụng trong tự động hóa thực hiện các hành động khi một image mới được tạo ra. Build và deployment có thể theo dõi một image stream và nhận sự thông báo khi một image mới được tạo ra, sau đó có thể phản ứng bằng cách thực hiện một build hoặc deployment tương ứng.

Ví dụ: Một deployment sử dụng image A và khi có version mới hơn của image A được tạo ra, deployment có thể tự động thực hiện lại việc deploy với image mới thông qua thông

báo từ image stream.

### Deployment

- **Replication Controllers**

Replication controller sẽ đảm bảo rằng một số lượng cụ thể các bản sao (replica) của một pod luôn chạy. Khi pod thoát (exit) hoặc bị xóa, replication controller sẽ tạo phiên bản (instantiate) pod mới sao cho số lượng pod đang chạy luôn bằng với một con số được định trước. Tương tự với khi có nhiều hơn số pod cần thiết, nó sẽ xóa bớt đến một số lượng nhất định.

Replication controller không thực hiện auto-scaling dựa trên số lượng load hay lưu lượng. Thay vào đó, số lượng các replica có thể được điều chỉnh bởi một auto-scaler từ bên ngoài.

- **Deployments and Deployment Configurations**

Được xây dựng trên replication controller, deployment hiểu đơn giản là tạo ra một replication controller mới và khởi chạy các pod. OpenShift Origin deploymentnet còn cung cấp khả năng chuyển đổi từ một deployment sang một deployment mới đồng thời định nghĩa các hook có thể chạy trước hoặc sau khi replication controller được tạo ra.

Đối tượng Deployment Configuration được dùng để định nghĩa một deployment:

- Định nghĩa các yếu tố cho một ReplicationController.
- Các trigger cho tự động tạo một deployment mới.
- Chiến lược cho việc chuyển đổi giữa các deployment.
- Life cycle hooks.

Dưới đây là một Deployment Configuration ví dụ:

### Danh sách 3.2.1: Deployment Configuration

```
apiVersion: v1
kind: DeploymentConfig
metadata:
  name: frontend
spec:
  replicas: 5
  selector:
    name: frontend
  template: { ... }
  triggers:
    - type: ConfigChange           (1)
    - imageChangeParams:
        automatic: true
        containerNames:
          - helloworld
        from:
          kind: ImageStreamTag
          name: hello - openshift:latest
      type: ImageChange           (2)
  strategy:
    type: Rolling                 (3)
```

1. *ConfigChange* trigger: deployment mới sẽ được tạo ra mỗi khi Deployment Config thay đổi.
2. *ImageChange* trigger: deployment mới sẽ được tạo ra mỗi khi phiên bản với hơn của image được phát hiện từ image stream.
3. *Rolling* strategy: Chiến lược chuyển đổi giữa các deployment.

### Route

Một OpenShift Origin route expose một service thành host name, ví dụ `www.example.com`, do đó các máy khách bên ngoài có thể truy cập đến service đó thông qua name. DNS phân giải cho host name được tách biệt với route. Để DNS bên ngoài có thể phân giải được host name này, cần cấu hình một entry trở về OpenShift Origin node đang chạy router.

### Template

Template mô tả một tập các đối tượng, có thể được tham số, mục đích là để tạo ra một danh sách các đối tượng (tạo các đối tượng theo một thứ tự). Template có thể được mô tả để tạo ra bất kỳ đối tượng nào miễn là người dùng có quyền tạo chúng trong project (service, build configuration, deployment configuration, etc).

#### Danh sách 3.2.2: Định nghĩa một template đơn giản

```
apiVersion: v1
kind: Template
metadata:
  name: redis-template (1)
  annotations:
    description: "Description"
    iconClass: "icon-redis"
    tags: "database,nosql"
objects: (2)
- apiVersion: v1
  kind: Pod
  metadata:
    name: redis-master
  spec:
    containers:
      - env:
          - name: REDIS_PASSWORD
            value: ${REDIS_PASSWORD} (3)
        image: dockerfile/redis
        name: master
        ports:
          - containerPort: 6379
            protocol: TCP
    parameters: (4)
      - description: Password used for Redis authentication
        from: '[A-Z0-9]{8}' (5)
        generate: expression
        name: REDIS_PASSWORD
    labels: (6)
      redis: master
```

1. Tên của template
2. Danh sách các đối tượng cần tạo (trong trường hợp này đơn giản chỉ có một pod)
3. Giá trị tham số sẽ được thay thế khi template được xử lý

4. Danh sách các tham số cho template
5. Một biểu thức để ngẫu nhiên giá trị cho tham số REDIS\_PASSWORD nếu tham số này không được đặc tả một giá trị cụ thể
6. Danh sách các nhãn áp dụng cho các đối tượng

Template thường dùng để mô tả tập các đối tượng liên quan đến nhau được tạo ra cùng nhau, cũng như tập các tham số cho các đối tượng.

## 3.3 Cài đặt và cấu hình hệ thống

### 3.3.1 Thông tin hệ thống

Hệ thống OpenShift Origin được cấp phát của chúng tôi gồm một Master và một Node:

<b>Master</b>	<ul style="list-style-type: none"><li>• Số lượng: 01</li><li>• Máy ảo.</li><li>• Hệ điều hành: CentOS 7.2 x86_64.</li><li>• 4 GB RAM</li><li>• 40 GB không gian lưu trữ.</li></ul>
<b>Node</b>	<ul style="list-style-type: none"><li>• Số lượng: 01</li><li>• Máy ảo.</li><li>• Hệ điều hành: CentOS 7.2 x86_64.</li><li>• 7 GB RAM</li><li>• 24 GB không gian lưu trữ.</li></ul>

Bảng 3.1: Thông tin hệ thống OpenShift Origin

### 3.3.2 Cài đặt hệ thống

#### Chuẩn bị

OpenShift Origin hỗ trợ cài đặt chính thông qua Ansible - một công cụ tự động hóa cấu hình. Chúng tôi cũng sử dụng cách cài đặt này. Dưới đây là các công việc cần hoàn thành trước khi cài đặt OpenShift Origin

1. Với mỗi Master và Node trong hệ thống, cần cài đặt trước các package sau: *bind-utils*, *net-tools*, *epel-release*.
2. Cấu hình Ansible inventory file (đây là file dùng để cấu hình các máy mà Ansible ssh đến, cùng với các thông số cài đặt) tại */etc/ansible/hosts*. Dưới đây là file cấu hình tham khảo:

#### Danh sách 3.3.1: Cấu hình tham khảo Ansible inventory file

```
# Create an OSEv3 group that contains the masters and nodes groups
[OSEv3:children]
masters
nodes

# Set variables common for all OSEv3 hosts
[OSEv3:vars]
ansible_ssh_user=root
deployment_type=origin

# Proxy (optional)

# host group for masters
[masters]
master.local

# host group for nodes, includes region info
[nodes]
master.local openshift_node_labels="{ 'region': 'infra', 'zone': 'default' }"
           openshift_schedulable=true
node1.local openshift_node_labels="{ 'region': 'primary', 'zone': 'east' }
```

3. Đảm bảo authentication SSH.

Ansible sử dụng ssh để cấu hình các máy. Do đó, cần đảm bảo rằng Ansible Control

Machine (máy cài đặt và chạy Ansible) có thể authentication SSH đến tất cả các Master và Node trong hệ thống.

```
$ ansible all -m ping
master.local | SUCCESS => {
    "changed": false,
    "ping": "pong"
}
node1.local | SUCCESS => {
    "changed": false,
    "ping": "pong"
}
```

### Cài đặt

- Clone Openshift-Ansible repository

```
git clone https://github.com/openshift/openshift-ansible.git
```

- Chạy Ansible playbooks với file cấu hình từ repository trên

```
ansible-playbook openshift-ansible/playbooks/byo/config.yml
```

## 3.4 Source-to-Image (S2I)

### 3.4.1 Cơ sở kiến thức

Source-to-Image (S2I) là một framework giúp tạo các image bằng cách nhận vào mã nguồn ứng dụng, xử lý mã nguồn và tạo ra image mới từ đó.

Ưu điểm chính của việc sử dụng S2I để xây dựng các image là tính dễ sử dụng cho developer. Developer chỉ đưa vào mã nguồn, công đoạn build sẽ diễn ra ngay bên trong image và sau đó image mới sẽ được tạo thành sẵn sàng để deploy.

Tại ELCA, chúng tôi sử dụng S2I để xây dựng các image từ các binary package (các package đã được build sẵn). Project không build trong image mà được build tại một Jenkins server riêng biệt. Sau đó, các binary package sẽ được sử dụng để tạo thành các image. Việc tách rời quá trình build source code và build image là phù hợp với đặc thù của pipeline tại công ty.

S2I có hai khái niệm chính cần được đề cập, đó là *build process* và *S2I scripts*.

### Build process

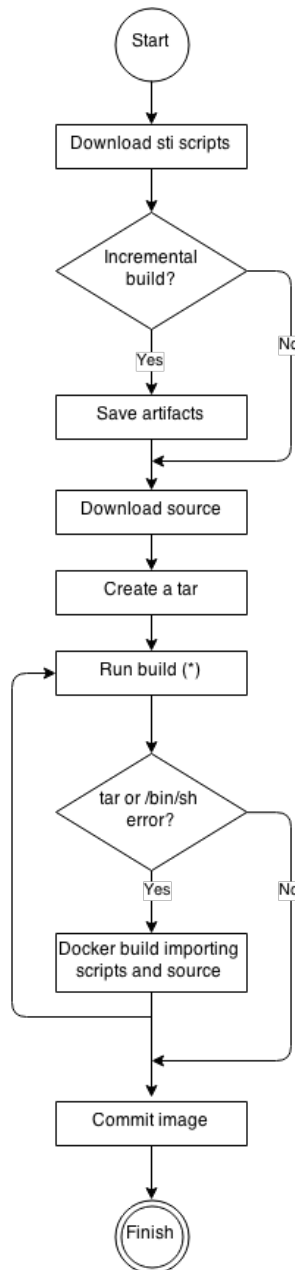
Build process vào gồm ba yếu tố cơ bản, kết hợp tạo thành image cuối cùng:

- sources
- S2I scripts
- builder image

Trong build process, S2I phải đặt sources và scripts vào bên trong một builder image. Để làm điều đó, S2I tạo một file *tar* chứa sources và scripts bên trong, sau đó stream (truyền) vào bên trong builder image. S2I sau đó untar file này vào thư mục được đặc tả bởi *-destination* flag hoặc *io.openshift.s2i.destination* label từ builder image. Sau đó, tiến hành thực thi *assamble* script để tiến hành quá trình build.

Build process được mô tả cụ thể bởi hình bên dưới:





Hình 3.2: S2I build workflow

### S2I scripts

S2I scripts có thể được viết bởi bất cứ ngôn ngữ lập trình nào, miễn sao nó thực thi được trong builder image. Script sẽ được tìm kiếm trong builder image theo thứ tự sau.

1. `--scripts-url` flag
2. Thư mục `.s2i/bin`

3. `io.openshift.s2i.scripts-url` label của builder image.

Dưới đây là danh sách các script và chức năng của nó:

Script	Mô tả
<b>assamble</b> (bắt buộc)	Assamble script build ứng dụng từ source và đặt chúng vào thư mục thích hợp bên trong image.
<b>run</b> (bắt buộc)	Script thực thi ứng dụng.
<b>save-artifacts</b> (không bắt buộc)	Script gather (thu lượm) tất cả các dependency do đó nó có thể đẩy nhanh quá trình build kế tiếp. Các dependency được thu lượm sẽ được nén vào tar file và stream thành output.
<b>usage</b> (không bắt buộc)	Script hiển thị thông báo đến user cách sử dụng image (khi image được run bởi <i>docker run</i> ).
<b>test/run</b> (không bắt buộc)	Script cho phép tạo một simple process để kiểm tra image có hoạt động như ý muốn hay không.

Bảng 3.2: S2I Scripts

### 3.4.2 Xây dựng Image với S2I

Chúng tôi xây dựng các Image cho ba máy chủ ứng dụng Java là Apache Tomcat, Wildfly và Oracle WebLogic, chạy trên hai Java Runtime Environment (JRE) 7 và 8 trên hai OS là Debian và CentOS. Cách build các image này có khá nhiều điểm tương đồng. Do đó, nên ở đây chúng tôi chỉ mô tả chi tiết cho image Tomcat 7 chạy trên JRE 7, CentOS.

Chúng tôi build project từ một Jenkins server riêng và stream các binary package vào builder image. Nhưng như vậy là chưa đủ cho quá trình deploy thành công bởi còn phụ thuộc vào các file configuration cho cả ứng dụng và Tomcat server. Hơn nữa, một yêu cầu đặt ra là image được tạo ra cần deploy được trên nhiều môi trường khác nhau với các tham số cấu hình khác nhau. Image chỉ nên được build một lần. Nhưng khi được deploy, tùy theo tham số môi trường khác nhau mà sẽ có cấu hình cụ thể khác nhau cho ứng dụng cũng như Tomcat server.

Giải pháp mà chúng tôi đưa ra là cùng với các binary package, toàn bộ các file cấu hình cho ứng dụng và Tomcat sẽ đồng thời được stream vào image. Các file cấu hình sẽ được đặt trong thư mục *configuration*, với các thư mục con *tomcat* và *application* tương ứng chứa các

cấu hình cho Tomcat và ứng dụng. Trong trường hợp có nhiều môi trường deploy khác nhau thì các thư mục *tomcat*, *application* sẽ được đặt trong thư mục có tên là môi trường tương ứng. Ví dụ có hai môi trường cần deploy khác nhau là *AT* và *INT* thì thư mục *configuration/at* chứa cấu hình cho môi trường AT, tương tự thư mục *configuration/int* chứa cấu hình cho môi trường INT.

Các binary package sẽ được copy lên thư mục deployment của Tomcat (mặc định là thư mục *webapps*) trong lúc build image. Các configuration sẽ được copy đến các folder thích hợp trong lúc build image khi chỉ có duy nhất một môi trường. Khi có nhiều môi trường deploy, các configuration thích hợp sẽ được copy trong lúc run image tùy thuộc vào biến *DEPLOY\_ENV* được truyền từ lệnh run. Phần dưới đây chúng tôi cung cấp mã nguồn của Dockerfile, script assamble và script run cho mục đích tham khảo:

### Danh sách 3.4.1: Dockerfile của image Tomcat 7

```
FROM anhnghuyen/centos-serverjre-7

MAINTAINER Hoang Anh Nguyen <nghoanganh994@gmail.com>

ENV TOMCAT_VERSION=7.0.72 \
TOMCAT_NUMBER=7

LABEL io.k8s.description="Platform for building and running JEE applications on
Tomcat 7" \
io.k8s.display-name="Tomcat 7" \
io.openshift.expose-services="8080:http" \
io.openshift.expose-services="8443:https" \
io.openshift.tags="builder,tomcat" \
io.openshift.s2i.destination="/opt/s2i/destination"

# Install Tomcat
RUN INSTALL_PKGS="tar unzip bc which lsof" && \
    yum install -y --enablerepo=centosplus $INSTALL_PKGS && \
    rpm -V $INSTALL_PKGS && \
    yum clean all -y && \
    mkdir -p /opt/tomcat && \
    (curl -v
        https://www.apache.org/dist/tomcat/tomcat-$TOMCAT_NUMBER/v$TOMCAT_VERSION/
        bin/apache-tomcat-$TOMCAT_VERSION.tar.gz | tar -zx --strip-components=1 -C
        /opt/tomcat) && \
    mkdir -p /opt/s2i/destination && \
    mkdir -p /opt/configuration

# Copy the S2I scripts from the specific language image to $STI_SCRIPTS_PATH
COPY ./s2i/bin/ $STI_SCRIPTS_PATH

RUN chown -R 1001:0 /opt/tomcat && chown -R 1001:0 $HOME && \
    chmod -R ug+rw /opt/tomcat && \
    chmod -R ug+rw /opt/configuration && \
    chmod -R g+rw /opt/s2i/destination

EXPOSE 8080 8443

USER 1001

CMD $STI_SCRIPTS_PATH/usage
```

### Danh sách 3.4.2: Assamble script

```
#!/bin/bash
function copy_artifacts() {
    dir=$1
    types=
    shift
    while [ $# -gt 0 ]; do
        types="$types;$1"
        shift
    done
    for d in $(echo $dir | tr "," "\n")
    do
        shift
        for t in $(echo $types | tr ";" "\n")
        do
            echo "Copying all $t artifacts from $LOCAL_SOURCE_DIR/$d directory
                into $DEPLOY_DIR for later deployment..."
            cp -v $LOCAL_SOURCE_DIR/$d/*. $t $DEPLOY_DIR 2> /dev/null
            chgrp -fr 0 $LOCAL_SOURCE_DIR/$d/*. $t
            chmod -fr g+rw $LOCAL_SOURCE_DIR/$d/*. $t
        done
    done
}
# Source code provided to S2I is at ${HOME}
LOCAL_SOURCE_DIR=${HOME}
mkdir -p $LOCAL_SOURCE_DIR
# Resulting WAR files will be deployed to /opt/tomcat/webapps
DEPLOY_DIR=/opt/tomcat/webapps
rm -rf $DEPLOY_DIR/ROOT
# Copy the source for compilation
cp -Rf /opt/s2i/destination/src/. $LOCAL_SOURCE_DIR
chgrp -R 0 $LOCAL_SOURCE_DIR
chmod -R g+rw $LOCAL_SOURCE_DIR
if [ -d $LOCAL_SOURCE_DIR/configuration ]; then
    echo -e "Copying config files from project..."
    if [ -d $LOCAL_SOURCE_DIR/configuration/tomcat ]; then
        echo -e "\nCopying Tomcat config files to /opt/tomcat/conf..."
        cp -rfv $LOCAL_SOURCE_DIR/configuration/tomcat/* /opt/tomcat/conf/
    fi
    if [ -d $LOCAL_SOURCE_DIR/configuration/application ]; then
        echo -e "\nCopying Tomcat config files to /opt/configuration..."
        cp -rfv $LOCAL_SOURCE_DIR/configuration/application/* /opt/configuration/
    fi
fi
echo -e "\nCopying binaries in source directory into $DEPLOY_DIR for later
    deployment..."
copy_artifacts "." war ear rar jar
echo -e "\n...done"
```

### Danh sách 3.4.3: Run script

```
#!/bin/bash

LOCAL_SOURCE_DIR=${HOME}

if [ -z "$DEPLOY_ENV" ]; then
    echo "DEPLOY_ENV is unset. Using default configuration."
else
    echo "DEPLOY_ENV is set to '$DEPLOY_ENV'"
    env=$(tr [A-Z] [a-z] <<< "$DEPLOY_ENV")

    if [ -d $LOCAL_SOURCE_DIR/configuration/$env ]; then
        echo "Copy deployment configuration files for $DEPLOY_ENV environment."

        if [ -d $LOCAL_SOURCE_DIR/configuration/$env/tomcat ]; then
            echo -e "\nCopying Tomcat config files to /opt/tomcat/conf..."
            cp -rfv $LOCAL_SOURCE_DIR/configuration/$env/tomcat/* /opt/tomcat/conf/
        fi

        if [ -d $LOCAL_SOURCE_DIR/configuration/$env/application ]; then
            echo -e "\nCopying Tomcat config files to /opt/configuration..."
            cp -rfv $LOCAL_SOURCE_DIR/configuration/$env/application/*
              /opt/configuration/
        fi

        else
            echo "Configuration directory $DEPLOY_ENV doesn't exist."
        fi
    fi

    echo -e "Starting Tomcat..."
    exec /opt/tomcat/bin/catalina.sh run
```

## Chương 4    CONTINUOUS INTEGRATION - JENKINS

Theo Martin Fowler: tích hợp liên tục (Continuous Integration - CI) là một phương pháp phát triển phần mềm đòi hỏi các thành viên trong nhóm tích hợp công việc thường xuyên. Mỗi ngày, các thành viên đều phải theo dõi và phát triển công việc của họ ít nhất một lần. Việc này sẽ được một nhóm khác kiểm tra tự động, nhóm này sẽ tiến hành kiểm thử để phát hiện lỗi nhanh nhất có thể. Cả nhóm thấy rằng phương pháp tiếp cận này giúp giảm bớt vấn đề tích hợp và cho phép nhóm phát triển phần mềm gắn kết nhau hơn.

Jenkins là một web application mã nguồn mở đóng vai trò máy chủ build, test và deploy của hệ thống tích hợp liên tục. Tiền thân là Hudson và được viết bằng Java. Jenkins hỗ trợ đầy đủ các tính năng tự động hóa của hệ thống tích hợp liên tục bằng việc kết hợp giữa các thành phần mở rộng khác (plugin).

Có nhiều lựa chọn khác cho CI tool như Bamboo, GoCD, Team City và nhiều công cụ khác nữa. Tất cả chúng có những điểm mạnh và điểm yếu riêng. Tuy nhiên, Jenkins được sử dụng phổ biến nhờ vào cộng đồng của nó. Jenkins có số lượng lớn những người tình nguyện viên đóng góp các plugin cho nó. Thông qua các plugin, chúng ta có thể dễ dàng đáp ứng hầu hết các nhu cầu cần thiết trong việc build, test, deploy ứng dụng. Thậm chí, nếu bạn không tìm thấy bất kỳ plugin nào phù hợp với nhu cầu sử dụng, thì viết một plugin của riêng bạn cũng là một điều rất dễ dàng.

Công ty ELCA đang sử dụng Jenkins như công cụ để xây dựng một tiến trình build, deploy và test ứng dụng. Tại thời điểm hiện tại, Jenkins đã tung ra Jenkins 2.0, có những thay đổi đáng quan tâm so với phiên bản 1.x mà hiện tại được công ty ELCA sử dụng. Trong phần này chúng tôi sẽ giới thiệu sơ lược về Jenkins 2.0. Chúng tôi sẽ tìm cách để chuyển đổi pipeline một project đã có của công ty ELCA trên Jenkins 1.x thành pipeline trên Jenkins 2.0 và tích hợp với Docker trong chương tiếp theo.

### 4.1 Những điểm mới trên Jenkins 2.0

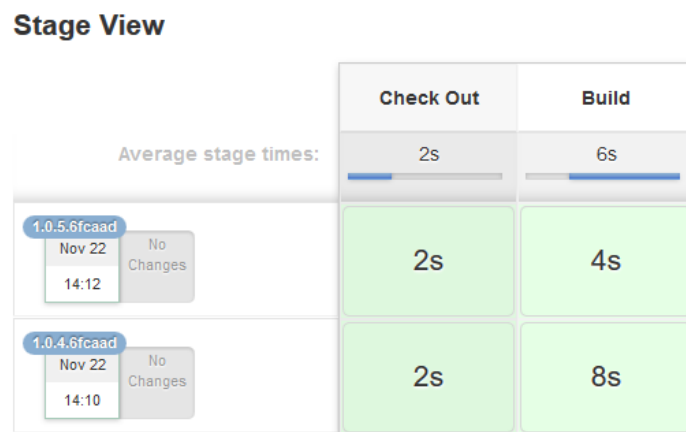
Những điểm nổi bật mà Jenkins 2.0 mang đến:

- Pipeline as code: Jenkins 2.0 giới thiệu một ngôn ngữ đặc tả tên miền (domain-specific language - DSL) giúp người sử dụng Jenkins có thể thoải mái mô tả mô hình pipeline phân phối phần mềm của họ. Ví dụ một pipeline script như sau:

### Danh sách 4.1.1: Ví dụ một pipeline script

```
node {  
    def mvnHome = tool 'maven-3.3.9'  
    env.PATH = "${mvnHome}/bin:${env.PATH}"  
  
    // Git checkout  
    stage 'Check Out'  
    git url: 'https://github.com/anhnguyenbk/sample.git'  
  
    // Build package with maven  
    stage 'Build'  
    sh "$mvn package -DskipTests=true"  
}
```

- Cải thiện về UI, UX design: một giao diện pipeline hoàn toàn mới.



Hình 4.1: Giao diện Pipeline mới

- Cải thiện về security và các plugin. Tuy nhiên, còn nhiều plugin của Jenkins 1.x vẫn chưa tương thích với Jenkins 2.0.



## 4.2 Một số câu lệnh đơn giản trên Jenkins 2.0

- Định nghĩa một stage

```
stage 'Stage name'
```

- Lấy source code từ các source code manager

Với Git SCM:

```
git url: 'https://github.com/anhnguyenbk/sample.git'
```

Với Subversion SCM:

```
svn 'https://svn.example.com/subversion/annguyen/trunk/sample'
```

Sử dụng Checkout SCM plugin. Giả sử checkout source code từ Subversion SCM và đã thêm Credentials cho tài khoản Subversion trên Jenkins:

```
checkout([ $class: 'SubversionSCM',  
    additionalCredentials: [],  
    excludedCommitMessages: '',  
    excludedRegions: '',  
    excludedRevprop: '',  
    excludedUsers: '',  
    filterChangelog: false,  
    ignoreDirPropChanges: false,  
    includedRegions: '',  
    locations: [[ credentialsId: '085f2511-8816-4edd-829c-1557  
        a7835a82',  
    depthOption: 'infinity',  
    ignoreExternalsOption: true,  
    local: '.',  
    remote: 'https://svn.example.com/subversion/annguyen/trunk/  
        sample' ]],  
    workspaceUpdater: [ $class: 'UpdateWithCleanUpdater' ]])
```

- Thực hiện một Shell script. Ví dụ với một câu lệnh remove một thư mục sample trên linux

```
sh 'rm -rf /usr/local/sample'
```

- Định nghĩa một hàm hoặc một biến

Định nghĩa hàm lấy số version trong file pom.xml của một maven project:

```
def version() {  
    def matcher = readFile('pom.xml') =~ '<version>(.)</version>'  
    matcher ? matcher[0][1] : null  
}
```

Định nghĩa một biến appVersion với giá trị là một chuỗi rỗng:

```
def appVersion = ''
```

- Cấu hình maven với java jdk cho một maven project:

```
env.JAVA_HOME = "${tool 'java-7u80'}"  
withEnv(["PATH+MAVEN=${tool 'apache-maven-3.3.9'}/bin:${env.  
    JAVA_HOME}/bin"]) {  
    sh 'mvn --version'  
    sh 'java -version'  
}
```

- Ghi lại kết quả chạy test với maven và lưu lại các file đã được đóng gói bằng maven

```
sh 'mvn -B clean install -Dmaven.test.failure.ignore'  
step([class: 'ArtifactArchiver', artifacts: '**/target/*.war',  
    fingerprint: true])  
step([class: 'JUnitResultArchiver', testResults: '**/target/  
    surefire-reports/TEST-*.xml'])
```

- Chạy các job song song

```
parallel (  
    'Job1': {  
        sh 'echo Run Job 1'  
    },  
    'Job 2': {  
        sh 'echo Run Job 2'  
    }  
)
```

- Gửi mail

```
mail subject: "${env.JOB_NAME} (${env.BUILD_NUMBER}) failed "
body: "It appears that ${env.BUILD_URL} is failing , somebody
      should do something about that",
to: recipient ,
replyTo: recipient ,
from: 'noreply@ci.jenkins.io '
```

- Ngoài ra, Jenkins 2.0 có hỗ trợ công cụ Snippet Generator để tự động tạo Pipeline Script cho các hàm và các plugin mà Jenkins pipeline hỗ trợ .

The screenshot shows the Jenkins Snippet Generator interface. At the top, there's an 'Overview' section with a brief description of the tool. Below that, the 'Steps' section shows a dropdown menu with 'junit: Publish JUnit test result report' selected. The configuration area includes several fields: 'Test report XMLs' with a text input containing '\*\*target/surefire-reports/TEST-\*.xml', a checkbox for 'Retain long standard output/error', 'Health report amplification factor' with a numeric input set to '1', and 'Allow empty results' with a checkbox for 'Do not fail the build on empty test results'. A 'Generate Pipeline Script' button is located below the configuration fields. At the bottom, a text area displays the generated Pipeline Script snippet: 'junit "\*\*target/surefire-reports/TEST-\*.xml'

Hình 4.2: Snippet Generator

## Chương 5 TÍCH HỢP DEVOPS PIPELINE VỚI DOCKER

Trong chương này, chúng tôi sẽ tiến hành làm mới pipeline của một project của công ty ELCA trên Jenkins 1.x thành pipeline trên Jenkins 2.0. Bên cạnh đó tích hợp deploy với Docker để tạo các môi trường kiểm thử cho project.

### 5.1 Chuẩn bị

Đầu tiên, chúng tôi sẽ tạo sẵn một tập Docker image cho các môi trường Java Server khác nhau. Các Docker image này là các builder image sẽ được dùng bởi Source-To-Image (S2I) Toolkit đã được nói đến trong chương 3. Chúng tôi sử dụng cách build dùng S2I trên OpenShift Origin để build Docker image.

- centos-serverjre7-tomcat7
- centos-serverjre8-tomcat8
- centos-serverjre8-wildfly10
- centos-serverjre8-weblogic1221
- debian-serverjre7-tomcat7
- debian-serverjre8-tomcat8
- debian-serverjre8-wildfly10
- debian-serverjre8-weblogic1221

Tất cả các Docker image ở trên sẽ được đẩy lên một integrated Docker registry đã được deploy trên OpenShift Origin. Các image này sẽ là builder image dùng để build Docker image mới cho các project Java trên OpenShift Origin.

Trên Jenkins server, chúng tôi cài đặt oc (OpenShift Client) tool - một command line tool để tương tác với OpenShift Origin master. Từ đó trên Jenkins, chúng tôi sử dụng các lệnh để build và deploy các containers trên OpenShift Origin cluster.

## 5.2 Tích hợp Pipeline với Docker

Sau khi đã tạo sẵn các Docker image cho các Java Server. Tiếp theo chúng tôi sẽ tạo pipeline trên Jenkins 2.0 cho các một project của công ty ELCA. Project này sẽ được deploy với Docker thay vì VMS như trước. Chúng tôi sẽ tạm gọi các project này là project A.

### Project A

Project A là một project java được deploy trên tomcat 7 với java jdk 7. Project A sẽ được kiểm thử trên 2 môi trường được tạo bằng VMware:

- AT (Automation Test): Môi trường này sẽ được kiểm thử tự động với một database với bộ dữ liệu có sẵn.
- INT (Integration Test): Môi trường này sẽ được kiểm thử với một database khác do Tester thêm vào dữ liệu.

Với những đặc tả trên, chúng tôi sẽ sử dụng Docker image centos-serverjre7-tomcat7 làm builder image để build một image mới. Image mới này sẽ tạo một môi trường đồng nhất cho hai môi trường AT và INT.

Trên OpenShift Origin, ngoài việc đã đẩy Docker image centos-serverjre7-tomcat7 lên trên integrated Docker registry, chúng tôi sẽ tạo một project có tên là a-prj:



New Project

\* Name

a-prj

A unique name for the project.

Display Name

My Project

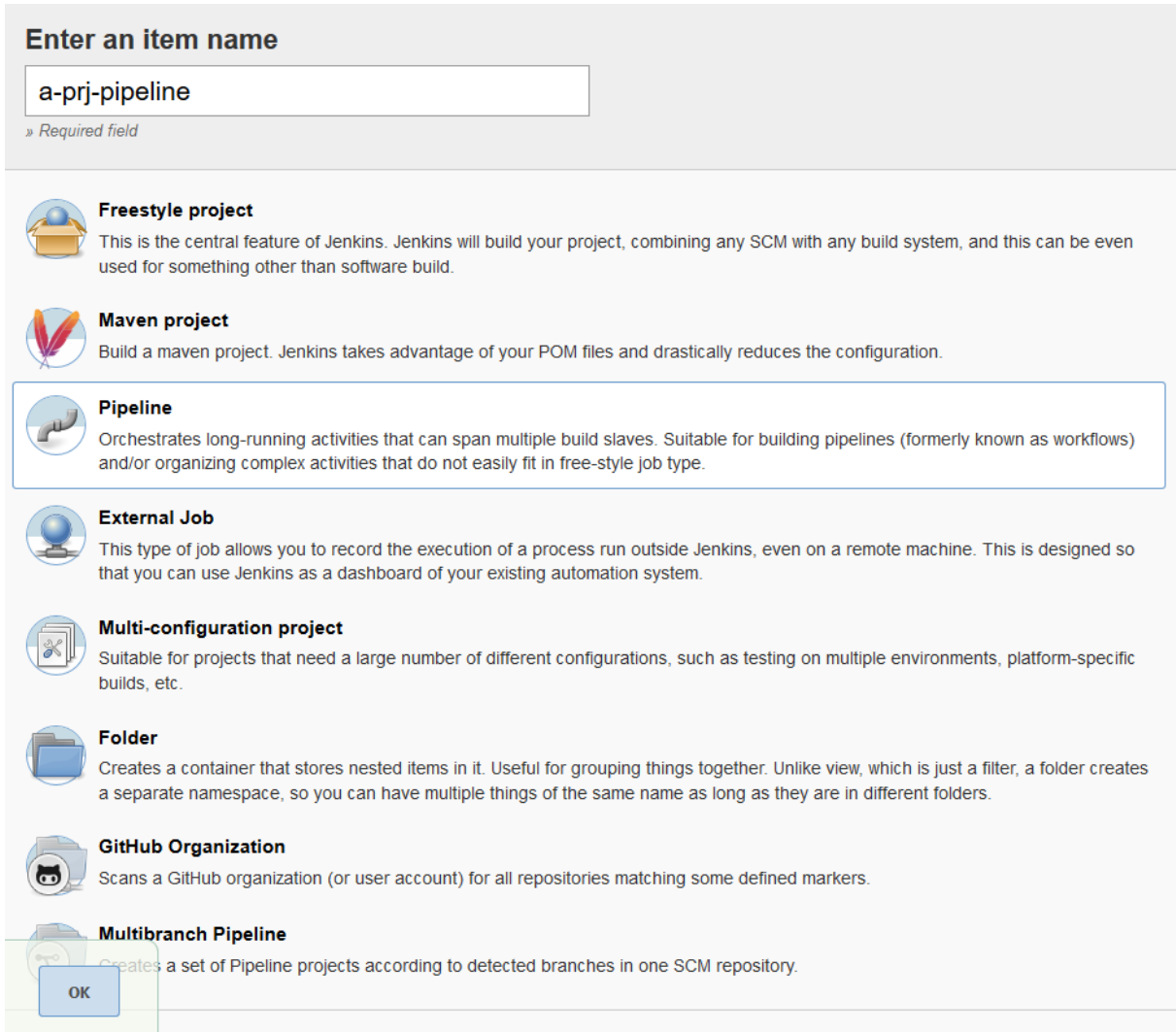
Description

A short description.

Create Cancel

Hình 5.1: Tạo a-prj trên OpenShift Origin

Trên Jenkins, chúng tôi tạo một pipeline project có tên là a-prj-pipeline



**Enter an item name**

a-prj-pipeline

» Required field

- Freestyle project**  
This is the central feature of Jenkins. Jenkins will build your project, combining any SCM with any build system, and this can be even used for something other than software build.
- Maven project**  
Build a maven project. Jenkins takes advantage of your POM files and drastically reduces the configuration.
- Pipeline**  
Orchestrates long-running activities that can span multiple build slaves. Suitable for building pipelines (formerly known as workflows) and/or organizing complex activities that do not easily fit in free-style job type.
- External Job**  
This type of job allows you to record the execution of a process run outside Jenkins, even on a remote machine. This is designed so that you can use Jenkins as a dashboard of your existing automation system.
- Multi-configuration project**  
Suitable for projects that need a large number of different configurations, such as testing on multiple environments, platform-specific builds, etc.
- Folder**  
Creates a container that stores nested items in it. Useful for grouping things together. Unlike view, which is just a filter, a folder creates a separate namespace, so you can have multiple things of the same name as long as they are in different folders.
- GitHub Organization**  
Scans a GitHub organization (or user account) for all repositories matching some defined markers.
- Multibranch Pipeline**  
Creates a set of Pipeline projects according to detected branches in one SCM repository.

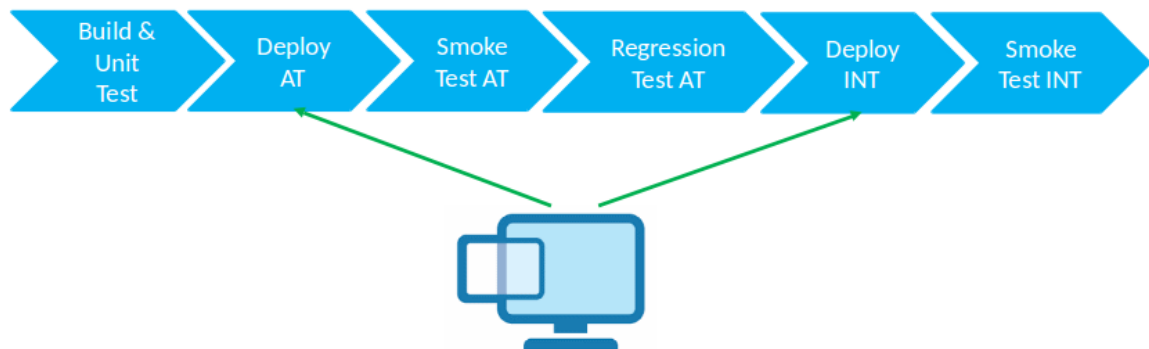
OK

Hình 5.2: Tạo a-prj-pipeline trên Jenkins

Trong Configure, chúng tôi định nghĩa các biến sau trong pipeline script

```
env.OC='/usr/local/bin'
env.OPENSIFT_API='https://openshift-master.example.com:8443'
env.SCM_URL='https://svn.example.com/subversion/trunk/a'
def project = 'a-prj'
def appName = 'a'
def revision = ''
def appVersion = ''
```

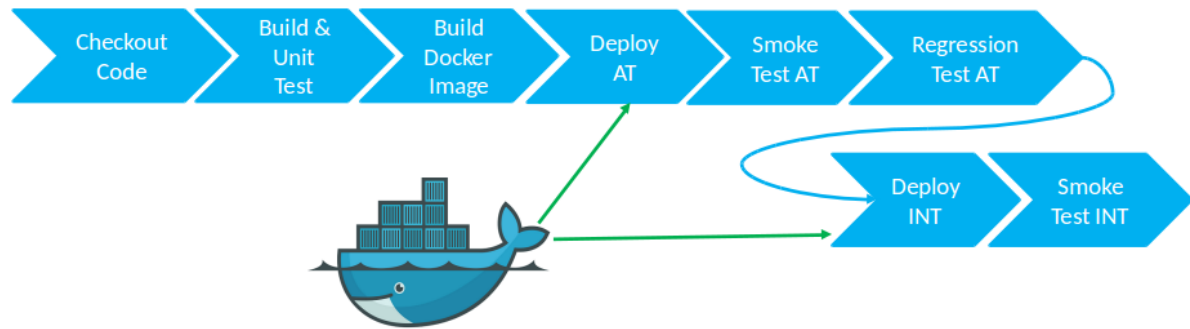
Đầu tiên, chúng tôi sẽ xem xét pipeline trên Jenkins 1.x của project A. Pipeline này gồm các job như sau:



Hình 5.3: Pipeline trên Jenkins 1.x của project A tích hợp với VMs

1. Build and Unit Test
2. Deploy to AT
3. Smoke Test AT
4. Regression Test AT
5. Deploy to INT
6. Smoke Test INT

Mỗi job trên là một freestyle project hoặc maven project trên Jenkins 1.x. Sau đó sử dụng một plugin là Build Pipeline Plugin để liên kết các job với nhau tạo thành pipeline. Trong giai đoạn thực tập tốt nghiệp, chúng tôi cũng đã sử dụng plugin này để tạo pipeline cho một dự án java nhỏ do chúng tôi tự phát triển. Trong Jenkins 2.0, Jenkins đã hỗ trợ tạo ra một pipeline project, thay vì phải tạo nhiều project như trước chúng ta chỉ tạo một project duy nhất. Các job trong pipeline cũ sẽ được gọi là stage trong pipeline mới. Chúng tôi cũng sẽ chỉnh sửa pipeline cũ lại đôi chút cho phù hợp với pipeline project trên Jenkins 2.0 và tích hợp deploy với Docker. Pipeline mới sẽ gồm các stage như sau:



Hình 5.4: Pipeline trên Jenkins 2.0 của project A tích hợp với Docker

1. Checkout code from SCM
2. Build and Unit Test
3. Build Docker Image
4. Deploy to AT
5. Smoke Test AT
6. Regression Test AT
7. Deploy to INT
8. Smoke Test INT

So với pipeline cũ, chúng tôi đã tạo thêm các stage là "Checkout code from SCM" và "Build Docker Image". Với stage 1 "Checkout code from SCM", thật ra thì nhiệm vụ lấy code từ SCM là nằm trong job "Build and Unit Test" của pipeline cũ. Nhưng lý do ở đây chúng tôi tách ra là vì: trong các job sau như "Smoke Test AT", "Regression Test AT", "Smoke Test INT" đều có tác vụ lấy code dùng để test từ SCM về nhưng ở pipeline cũ mỗi job là các Jenkins project khác nhau nên sẽ có một không gian làm việc (workspace) riêng nên việc lấy code về sẽ không ảnh hưởng nhau. Trong khi đối với pipeline project chỉ có một không gian làm việc duy nhất cho toàn bộ pipeline, nên chúng tôi tách ra thành một stage riêng để chỉ cần lấy code một lần duy nhất. Còn với stage 3 "Build Docker Image" đây sẽ một stage mới hoàn toàn so với pipeline cũ. Ở stage này, nhiệm vụ sẽ là tạo ra Docker Image có thể được dùng cho deploy nhiều môi trường.

Cụ thể các stage trong pipeline script như sau:



### Danh sách 5.2.1: Pipeline script: Stage 1 "Checkout code from SCM"

```
stage("1 - Checkout code from SCM") {
    echo "Checkout SCM"
    checkout([
        $class: 'SubversionSCM',
        additionalCredentials: [],
        excludedCommitMessages: '',
        excludedRegions: '',
        excludedRevprop: '',
        excludedUsers: '',
        filterChangelog: false,
        ignoreDirPropChanges: false,
        includedRegions: '',
        locations: [[credentialsId: '085f2511-8816-4edd-829c-1557a7835a82',
        depthOption: 'infinity',
        ignoreExternalsOption: true,
        local: '.',
        remote: "${env.SCM_URL}@${revision}"]],
        workspaceUpdater: [
            $class: 'UpdateWithCleanUpdater'
        ])

    def appVersionPrefix = version(readFile('pom.xml')).take(3)
    sh "svn info -r 'HEAD' --username=${env.USERNAME} --password=${env.PASSWORD} |
        grep Revision | egrep -o '[0-9]+' > SVN_REVISION "
    revision = readFile('SVN_REVISION').take(4)
    appVersion = "${appVersionPrefix}.${env.BUILD_NUMBER}.${revision}"
    currentBuild.displayName = "${appVersion}"
}
```

Trong stage 1, ngoài việc lấy code từ SCM về như đã nói ở trên, chúng tôi còn gán lại tên lần build theo giá trị của appVersion.

### Danh sách 5.2.2: Pipeline script: Stage 2 "Build and Unit Test"

```
stage("2 - Build and Unit Test") {
    sh "mvn versions:set -DnewVersion=${appVersion} -s
        $WORKSPACE/etc/m2/settings-jenkins-VN.xml"

    echo 'Build package and Unit test'
    sh "mvn -B clean install -Dgwt.compile.skip=false -Dgwt.draftCompile=true
        -DskipTests=false -Dmaven.test.failure.ignore -Pa-at,vn-integration -s
        $WORKSPACE/etc/m2/settings-jenkins-VN.xml"
    step([ $class: 'ArtifactArchiver', artifacts: 'a-web/target/registerjp.war,
        etc/appconfig/**/*', fingerprint: true ])
    step([ $class: 'JUnitResultArchiver', testResults:
        '**/target/surefire-reports/TEST-*.xml' ])

    sh "mvn -B flyway:clean flyway:migrate -Pautomation-test,a-at -f
        a-common/pom.xml -s $WORKSPACE/etc/m2/settings-jenkins-VN.xml"
}
```

Dòng lệnh đầu tiên có ý nghĩa là dùng maven để gán lại giá trị của thuộc tính version theo giá trị của biến appVersion trong file pom.xml của project. Các câu lệnh tiếp theo là dùng maven để build và unit test, tạo report kết quả test và lưu lại các package đã build. Câu lệnh cuối cùng là plugin flyway trong maven để reset lại database.

### Danh sách 5.2.3: Pipeline script: Stage 3 "Build Docker Image"

```
stage("3 - Build Docker Image") {
    sh "rm -rf oc-build && mkdir -p oc-build"
    sh "cp -r /opt/jenkins/configuration oc-build/"
    sh "cp zhstregisterjp-web/target/registerjp.war oc-build/registerjp.war"
    withCredentials([ [ $class: 'UsernamePasswordMultiBinding', credentialsId:
        'cbf930eb-39e8-49e5-a09e-182ba8769e9a', usernameVariable: 'USERNAME',
        passwordVariable: 'PASSWORD' ] ]) {
        sh "${env.OC} login --insecure-skip-tls-verify=true ${env.OPENSIFT_API}
            --username=${env.USERNAME} --password=${env.PASSWORD}"
    }
    sh "${env.OC} delete bc,is -l app=${appName}-at -n ${project}";
    sh "${env.OC} new-build --name=${appName}-at
        --image-stream=centos-serverjre7-tomcat7 --binary=true
        --labels=app=${appName}-at -n ${project}"
    sh "${env.OC} start-build ${appName}-at --from-dir=oc-build --follow -n
        ${project}"
    sh "${env.OC} tag ${project}/${appName}-at:latest
        ${project}/${appName}:${appVersion}"
}
```

Ba dòng lệnh đầu tiên dùng để copy các file cấu hình cũng như các package đã được đóng gói từ source code vào một thư mục là oc-build. Thư mục này sẽ được dùng như input- đầu vào cho việc build Docker Image trên Openshift Origin cùng với builder image centos-serverjre7-tomcat7. Câu lệnh thứ 4 là dùng oc client để đăng nhập vào Openshift Origin với tài khoản và mật khẩu chúng tôi đã thêm vào Jenkins trước đó. Các câu lệnh từ 5 - 7 dùng để xóa các build config và image stream đã tạo trước đó nếu có. Khi đã xóa xong, tạo lại build config và image stream mới thông qua câu lệnh oc new-build. Sau đó build Docker image từ thư mục oc-build, Docker image sau khi build thành công sẽ được đẩy lên Docker registry. Mặc định thì Docker image sau khi build xong sẽ được đánh số version là latest, vì thế câu lệnh cuối cùng dùng để đánh lại số image version theo biến appVersion. Việc đánh lại số image version có tác dụng cho việc tái sử dụng Docker image sau này.

### Danh sách 5.2.4: Pipeline script: Stage 4 "Deploy to AT"

```
stage("4 - Deploy to AT") {
  sh "${env.OC} delete dc,svc,route -l app=${appName}-at -n ${project}"
  sh "${env.OC} new-app ${appName}:${appVersion} -e DEPLOY_ENV=at -e
    APP_CONFIG=/opt/configuration -e JAVA_OPTS='-Xms1024M -Xmx2048M
    -XX:MaxPermSize=1024M' --name=${appName}-at -n ${project}"
  sh "${env.OC} expose svc/${appName}-at --hostname=${appName}-at.example.com -n
    ${project}"
}
```

Trong stage 4, chúng tôi sẽ deploy Docker image đã tạo ở stage 3 xong trên Openshift Origin cluster. Cụ thể là dòng lệnh đầu tiên sẽ xóa tất cả các deployment config, service và route của môi trường AT đã tạo trước đó nếu có. Câu lệnh tiếp theo dùng để tạo deployment config và service mới với các tham số đáng chú ý là DEPLOY\_ENV=at và APP\_CONFIG=/opt/configuration. Biến môi trường DEPLOY\_ENV dùng để chạy Docker container với các file cấu hình dành cho môi trường AT và biến môi trường APP\_CONFIG là đường dẫn đến các file cấu hình. Ở đây chúng tôi đã hard code địa chỉ đường dẫn là /opt/configuration để tạo ra template chuẩn cho việc cấu hình cho Java Project. Sau khi thực hiện xong câu lệnh thì Docker container đã bắt đầu chạy. Cuối cùng là tạo một địa chỉ để truy cập ứng dụng đã deploy với Docker. Bây giờ ta có thể truy cập vào địa chỉ <http://a-at.example.com> để xem ứng dụng hoạt động như thế nào. Tuy nhiên sẽ cần một khoảng thời để đợi ứng dụng deploy.

Suche

Aktivität: alle, Status: offen, Kategorie: alle, Typ: , Frist: alle, Datum: von , bis , Benutzer: Nguyen, Man (mmm), ist: zuständig, Priorität: ☒ gering, ☒ mittel, ☒ hoch

Angezeigte Treffer: 0 von insgesamt: 0

Pri.	Frist	Typ	Aktivität	KSARRegister für OV	UID-Nr	Name	Gemeinde	Status	Kategorie	Verantwortlich	Erstellt	Erstellt durch
------	-------	-----	-----------	---------------------	--------	------	----------	--------	-----------	----------------	----------	----------------

Version: 3.0.22.9034 9034@trunk Umgebung: AT

Hình 5.5: Project A đã deploy thành công với version là 3.0.22.9034 trên môi trường AT

### Danh sách 5.2.5: Pipeline script: Stage 5 "Smoke Test AT"

```
stage("5 - Smoke Test AT") {
    echo 'Wait util server is ready'
    timeout(5) {
        waitUntil {
            def r = sh script: "wget -q http://${appName}-at.example.com -O /dev/null",
                returnStatus: true
            return (r == 0);
        }
    }

    echo 'Test with Selenium'
    sh "mvn -B clean install -f a-test/pom.xml -DfailIfNoTests=false
        -Dtest=AuthenticationSmokeTest -Dapplication.url='${appName}-at.example.com'
        -Dmax.retries=1 -Da.webdriver.remote.url=http://192.168.225.150:5555/wd/hub
        -Dthucydides.driver.capabilities='browserName:chrome;version:44.0A'
        -Dmaven.test.failure.ignore=true -Dmaven.test.error.ignore=true
        -Dsvn.revision=${revision} -Pautomatic-ui-test -Pa-at -s
        '$WORKSPACE/a-test/settings-jenkins-VN.xml' -Dstory.timeout.in.secs=30000"
    step([ $class: 'JUnitResultArchiver', testResults:
        '**/target/surefire-reports/TEST-*.xml' ])
}
```

Vì cần phải một khoảng thời gian để deploy ứng dụng nên chúng tôi viết một đoạn script để kiểm ứng dụng đã deploy xong chưa. Sau một khoảng thời gian, đoạn script sẽ gửi một request

đến server thông qua địa chỉ ta đã tạo ở stage 4. Nếu quá thời gian 5 phút, thì pipeline sẽ dừng chạy. Nếu ứng dụng đã deploy thành công, đoạn script tiếp theo sẽ tiến hành tự động Smoke Test ứng dụng và tạo report sau khi test xong.

### Danh sách 5.2.6: Pipeline script: Stage 6 "Regression Test AT"

```
stage("6 - Regression Test AT") {
    echo 'Test with Selenium'
    parallel (
        'RegressionJobPart1SuiteTest': {
            sh "mvn clean install -f a-test/pom.xml -DfailIfNoTests=false
                -Dtest=RegresstionJobPart1SuiteTest
                -Dapplication.url='${appName}-at.example.com' -Dmax.retries=1
                -Da.webdriver.remote.url=http://192.168.225.150:5555/wd/hub
                -Dthucydides.driver.capabilities='browserName:chrome;version:44.0A'
                -Dmaven.test.failure.ignore=true -Dmaven.test.error.ignore=true
                -Dsvn.revision=${revision} -Pautomatic-ui-test -Pa-at -s
                '$WORKSPACE/a-test/settings-jenkins-VN.xml'
                -Dstory.timeout.in.secs=30000"
            step([ $class: 'JUnitResultArchiver', testResults:
                '**/target/surefire-reports/TEST-*.xml' ])
        },
        'RegressionJobPart2SuiteTest': {
            sh "mvn clean install -f a-test/pom.xml -DfailIfNoTests=false
                -Dtest=RegresstionJobPart2SuiteTest
                -Dapplication.url='${appName}-example.com' -Dmax.retries=1
                -Da.webdriver.remote.url=http://192.168.225.150:5555/wd/hub
                -Dthucydides.driver.capabilities='browserName:chrome;version:44.0A'
                -Dmaven.test.failure.ignore=true -Dmaven.test.error.ignore=true
                -Dsvn.revision=${revision} -Pautomatic-ui-test -Pa-at -s
                '$WORKSPACE/a-test/settings-jenkins-VN.xml'
                -Dstory.timeout.in.secs=30000"
            step([ $class: 'JUnitResultArchiver', testResults:
                '**/target/surefire-reports/TEST-*.xml' ])
        },
        'RegressionJobPart3SuiteTest': {
            sh "mvn clean install -f a-test/pom.xml -DfailIfNoTests=false
                -Dtest=RegresstionJobPart3SuiteTest
                -Dapplication.url='${appName}-example.com' -Dmax.retries=1
                -Da.webdriver.remote.url=http://192.168.225.150:5555/wd/hub
                -Dthucydides.driver.capabilities='browserName:chrome;version:44.0A'
                -Dmaven.test.failure.ignore=true -Dmaven.test.error.ignore=true
                -Dsvn.revision=${revision} -Pautomatic-ui-test -Pa-at -s
                '$WORKSPACE/a-test/settings-jenkins-VN.xml'
                -Dstory.timeout.in.secs=30000"
            step([ $class: 'JUnitResultArchiver', testResults:
                '**/target/surefire-reports/TEST-*.xml' ])
        }
    )
}
```

Tương tự như đoạn script dùng để test ở stage 5, chỉ có đều bộ test case của Regression Test sẽ lớn hơn rất nhiều so với Smoke Test. Vì bộ test case quá lớn nên đã được chia thành 3 phần chạy song song để tiết kiệm thời gian.

### Danh sách 5.2.7: Pipeline script: Stage 7 "Deploy to INT"

```
stage("7 - Deploy to INT") {  
    echo "Deploy to INT"  
    sh "${env.OC} delete dc,svc,route -l app=${appName}-int -n ${project}"  
    sh "${env.OC} new-app ${appName}:${appVersion} -e DEPLOY_ENV=int -e  
        APP_CONFIG=/opt/configuration -e JAVA_OPTS='-Xms1024M -Xmx2048M  
        -XX:MaxPermSize=1024M' --name=${appName}-int -n ${project}"  
    sh "${env.OC} expose svc/${appName}-int --hostname=${appName}-int.example.com -n  
        ${project}"  
}
```

Trong stage 7, script tương tự như stage 4, chỉ thay đổi lại các biến môi trường và tham số để đặc tả deploy trên môi trường INT thay vì AT.

The screenshot displays a web application interface with a search bar at the top. Below the search bar, there are several filter sections: 'Suche' (Search), 'Aktivität' (Activity), 'Status' (Status), 'Kategorie' (Category), 'Typ' (Type), 'Frist' (Deadline), 'Datum' (Date), 'Benutzer' (User), and 'Priorität' (Priority). The 'Suche' section includes a search bar and a 'Suchen' button. The 'Aktivität' section has a dropdown menu. The 'Status' section has a dropdown menu. The 'Kategorie' section has a dropdown menu. The 'Typ' section has a dropdown menu. The 'Frist' section has a dropdown menu. The 'Datum' section has a date range selector. The 'Benutzer' section has a dropdown menu. The 'Priorität' section has checkboxes for 'gering', 'mittel', and 'hoch'. The 'Suchen' button is located at the bottom right of the filter sections. Below the filter sections, there is a table with the following columns: 'Pro.', 'Frist', 'Typ', 'Aktivität', 'KSARegistrier OV', 'UID-Nr', 'Name', 'Gemeinde', 'Status', 'Kategorie', 'Verantwortlich', 'Erstellt', and 'Erstellt durch'. The table is currently empty. At the bottom of the page, there is a status bar that shows 'Version: 3.0.22.9034' and '9034@trunk - Umgebungen: int'.

Hình 5.6: Project A đã deploy thành công với version là 3.0.22.9034 trên môi trường INT

### Danh sách 5.2.8: Pipeline script: Stage 8 "Smoke Test INT"

```
stage("8 - Smoke Test INT") {  
    echo 'Wait util server is ready'  
    timeout(5) {  
        waitUntil {  
            def r = sh script: "wget -q http://${appName}-int.exmaple.com-O /dev/null",  
                returnStatus: true  
            return (r == 0);  
        }  
    }  
  
    echo 'Test with Selenium'  
    sh "mvn -B clean install -f a-test/pom.xml -DfailIfNoTests=false  
        -Dtest=AuthenticationSmokeTest  
        -Dapplication.url='${appName}-int.example.com' -Dmax.retries=1  
        -Da.webdriver.remote.url=http://192.168.225.150:5555/wd/hub  
        -Dthucydides.driver.capabilities='browserName:chrome;version:44.0A'  
        -Dmaven.test.failure.ignore=true -Dmaven.test.error.ignore=true  
        -Dsvn.revision=${revision} -Pautomatic-ui-test -Pa-int -s  
        '$WORKSPACE/a-test/settings-jenkins-VN.xml' -Dstory.timeout.in.secs=30000"  
}
```

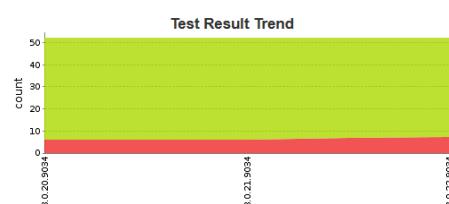
Script ở stage 8 tương tự như script của stage 5.

Bây giờ đặt tất cả các đoạn script trên lại với nhau trong Jenkins pipeline script và bắt đầu chạy thử.

#### Stage View

	1 - Checkout code from SCM	2 - Build and Unit Test	3 - Build Docker Image	4 - Deploy to AT	5 - Smoke Test AT	6 - Regression Test AT	7 - Deploy to INT	8 - Smoke Test INT
Average stage times:	25s	10min 54s	2min 59s	2s	3min 7s	3h 45min	3s	4min 4s
3.0.22.9034 Dec 07 01:00 No Changes	22s	13min 54s	3min 6s	3s	3min 5s	4h 25min	4s	3min 20s

Hình 5.7: Kết quả sau khi chạy xong pipeline

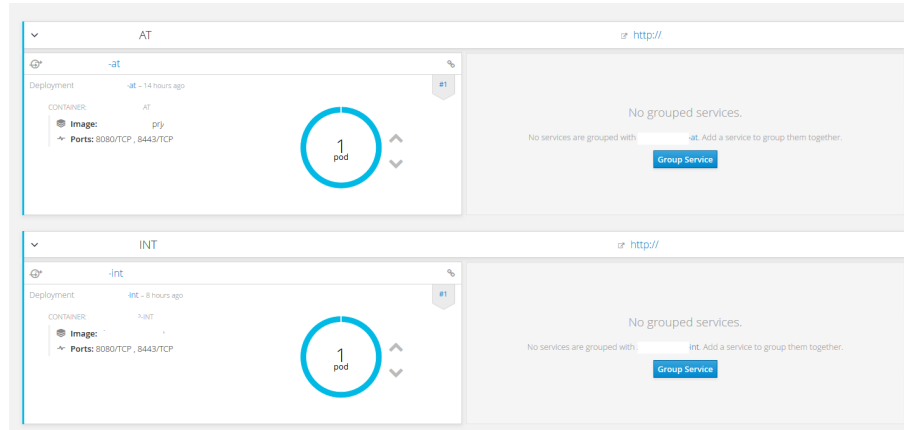


Hình 5.8: Kết quả test



Theo như kết quả trong hình vẽ, pipeline có màu vàng chứng tỏ rằng ứng dụng vẫn chưa hoàn thiện, vẫn còn lỗi trong kiểm tra Regression Test do project còn đang được phát triển.

Dưới đây là một số hình ảnh trên OpenShift Origin Web Console:



Hình 5.9: Các Pod của project A đang chạy trên OpenShift Origin

Image Streams >

created a day ago

Docker pull spec: 172.30.111.254:5000/

Tag	From	Latest Image	Created
3.0.20.9034	@sha256:cffbc67a7983f928c01f69f1742682f468...	cffbc6	a day ago
3.0.21.9034	@sha256:5c8b6bab378f6a772a2d4fa4777f785a...	5c8b6b	15 hours ago
3.0.22.9034	@sha256:b003a439289521787e49070c32a4304...	b003a4	8 hours ago

Hình 5.10: Các Docker Image đã build của project A trên OpenShift Origin

Tổng kết lại trong chương này, chúng tôi đã thành công tích hợp thành công pipeline với Docker cho Project A. Đây cũng là mục tiêu quan trọng đầu tiên của luận văn. Trong chương tiếp theo, chúng tôi sẽ đánh giá về hiệu năng của Docker so với VMs.

## Chương 6 ĐÁNH GIÁ HIỆU NĂNG CỦA DOCKER SO VỚI VMS TRONG DEVOPS PIPELINE CỦA CÔNG TY ELCA

Trong chương này, chúng tôi sẽ trình bày về việc so sánh hiệu năng của Docker sau khi tích hợp vào DevOps Pipeline của công ty ELCA so với VMs. Đây là phần quan trọng nhất của luận văn, kết quả của đánh giá này sẽ phản ánh giá trị thực tiễn của luận văn mà chúng tôi đang thực hiện. Chúng tôi sẽ tiến hành đánh giá theo các tiêu chí sau:

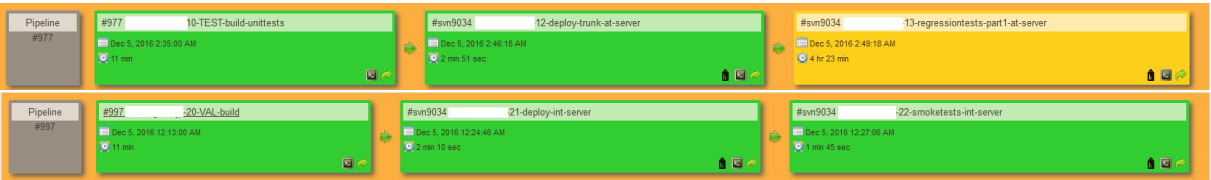
- Thời gian chạy pipeline
- Độ ổn định của pipeline
- Chi phí cấu hình
- Tái sử dụng image
- Khả năng mở rộng (Scalability)

Xin lưu ý, hai kiến trúc hạ tầng cũng như phần cứng chạy Docker (cluster) với VMs (single node) là hoàn toàn khác nhau. Vì thế đánh giá này không phải là đánh giá hiệu năng giữa Docker và VMs mà là đánh giá hiệu năng giữa Docker và VMs trong việc tích hợp vào DevOps Pipeline của công ty ELCA.

## 6.1 Thời gian chạy Pipeline



Hình 6.1: Thời gian chạy pipeline của Project A với Docker



Hình 6.2: Thời gian chạy pipeline của Project A với VMs

Stage	Pipeline với Docker	Pipeline với VMs
Build Docker Image	3min 59s	0s
Deploy + Smoke Test	3min 39s	3min 55s
Regression Test	4h 14min	4h 23min

Bảng 6.1: Thời gian chạy trong một số stage trên Jenkins

Theo kết quả trên, thời gian hoàn thành pipeline của Docker là nhanh hơn một chút so với VMs. Tuy nhiên sự khác biệt này không đáng kể bởi tổng thời gian hoàn thành của một công đoạn (job) là khá lớn.

## 6.2 Độ ổn định của Pipeline

Chúng tôi đã định thời cho Pipeline tự động chạy trong 3 ngày liên tiếp (06/12 - 09/12) với khoảng 6 lần chạy (hoàn thành một Pipeline mất khoảng 5h nên chúng tôi không chạy quá nhiều được) và kết quả là khả quan khi pipeline hoàn thành ổn định.

## 6.3 Dung lượng sử dụng

Môi trường	Docker	VMs
AT	1.55GB	36GB
INT		20GB

Bảng 6.2: Dung lượng lưu trữ của hai môi trường deploy

Về mặt lưu trữ rõ ràng Docker có kết quả ấn tượng hơn nhiều so với VMs. Chúng tôi chỉ sử dụng một Docker image cho cả hai môi trường deploy (AT và INT) và image này có dung lượng nhỏ hơn nhiều so với dung lượng được cấp phát cho VM sử dụng làm môi trường tương ứng.

## 6.4 Chi phí cấu hình

Đối với Docker, chúng tôi tạo ra sẵn những builder image với cấu hình các máy chủ ứng dụng Java trên những hệ điều hành khác nhau. Và khi ứng dụng cần được triển khai trên một môi trường mới, chỉ cần chọn builder image thích hợp.

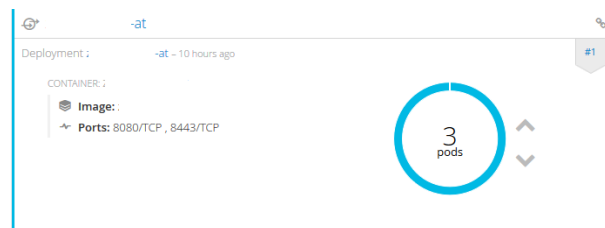
Ví dụ khi cần triển khai ứng dụng lên môi trường PRODUCTION. Yêu cầu môi trường cần phải được cài đặt Tomcat 8 chạy trên Debian. Với VMs, công ty ELCA sẽ phải cấp phát máy ảo, cài đặt Tomcat đúng phiên bản trên hệ điều hành được yêu cầu. Với Docker, chúng tôi đã có builder image theo yêu cầu trên và chúng tôi sẵn sàng deploy ứng dụng.

## 6.5 Tái sử dụng image

Các builder image đã tạo trước đó được lưu trữ trong Docker Registry có thể tái sử dụng cho các project khác. Ví dụ đối với project A, builder image centos-serverjre7-tomcat7 có thể sử dụng lại cho các project Java 7 deploy trên Tomcat 7 khác. Trong khi đó với VMs, các máy ảo được cấp phát một lần và không được sử dụng lại.

## 6.6 Scalability

Với đặc điểm của container, việc scale ứng dụng bằng cách tạo các bản sao (horizontal scaling) là rất dễ dàng. Với OpenShift Origin, nhờ vào Kubernetes, chúng ta dễ dàng scale ứng dụng thông qua command line hoặc ngay trên Web Console.



Hình 6.3: Scale project A với 3 pods

## Chương 7 KẾT LUẬN

### 7.1 Kết luận

Bài toán container để thay thế ảo hóa truyền thống (VMs) là một bài toán thực tế. Hiện nay rất nhiều các công ty, tập đoàn đã sử dụng giải pháp này cho các sản phẩm của họ. Tuy nhiên một lần nữa tôi phải nói lại rằng Container và VMs về bản chất là hai công nghệ khác nhau. Do đó, để có thể sử dụng Container thay thế VMs còn phụ thuộc vào dự án, các hệ điều hành được sử dụng trong dự án. Với ưu thế về hiệu suất và tốc độ, chúng tôi tin rằng Container sẽ tiếp tục phát triển và ứng dụng nhiều hơn nữa trong sản xuất. Chúng tôi cũng tin rằng Container và VMs tốt nhất nên được kết hợp cùng với nhau để tận dụng hết được sức mạnh của từng công nghệ.

Ở giai đoạn thực tập tốt nghiệp, chúng tôi nghiên cứu về Docker, nhằm nắm được những đặc điểm cùng cách sử dụng Docker. Chúng tôi tìm hiểu cơ bản về kiến trúc Java EE, những máy chủ ứng dụng thường sử dụng và cách triển khai ứng dụng Java lên các máy chủ. Đồng thời chúng tôi cũng tìm hiểu về quy trình và những công cụ thường được sử dụng trong quy trình phát triển các ứng dụng Java EE. Mục đích của chúng tôi là hiểu được quy trình phát triển, những công cụ sử dụng và tích hợp Docker vào quy trình này đối với một ứng dụng Java nhỏ do chúng tôi tạo ra.

Trong khoảng thời gian thực hiện luận văn tốt nghiệp này, chúng tôi đã nghiên cứu về OpenShift Origin - một nền tảng mã nguồn mở để build và run Docker container trên một hệ thống nhiều máy tính với những công cụ DevOps mạnh mẽ. Chúng tôi đã thành công trong việc cài đặt, cấu hình và vận hành OpenShift Origin trên cơ sở hạ tầng của công ty ELCA. Đồng thời, chúng tôi cũng nghiên cứu về Jenkins 2.0 để xây dựng DevOps pipeline cho một Java project của công ty ELCA. Đặc biệt, chúng tôi đã hoàn thành hai nhiệm vụ chính của luận văn này: tích hợp DevOps pipeline của công ty ELCA với Docker và đánh giá hiệu năng của giải pháp sử dụng Docker so với VMs.

Với những kết quả đạt được, chúng tôi tin rằng hoàn toàn có thể áp dụng giải pháp sử dụng Docker trong việc tạo môi trường deploy ứng dụng Java cho việc kiểm thử thay cho VMs tại công ty ELCA.

## **7.2 Hướng phát triển**

Luận văn của chúng tôi hiện tại chỉ xây dựng các image hỗ trợ deploy các ứng dụng Java. Trong khi đó, Container không hề bị giới hạn bởi ngôn ngữ, miễn là chúng chạy được trên Linux kernel. Do đó, chúng tôi hoàn toàn có thể mở rộng hệ thống của mình phục vụ cho ứng dụng khác: PHP, NodeJS, Python, Ruby và thậm chí cả ASP.NET (Microsoft hiện đã mã nguồn mở ASP.NET của mình, đồng thời cũng đã xây dựng ASP.NET image cho Docker). Chúng tôi hy vọng rằng trong tương lai chúng tôi sẽ có cơ hội áp dụng Docker vào các dự án thực tế thuộc các ngôn ngữ kể trên.

## TÀI LIỆU THAM KHẢO

- [1] Karl Matthias, Sean P. Kane. (2015). *Docker: Up & Running*. O'Reilly Media.
- [2] Adria Mouat. (2015). *Using Docker*. O'Reilly Media.
- [3] Viktor Farcic. (2016). *The DevOps 2.0 Toolkit*. 1st ed. [ebook]. Available at: <http://leanpub.com/the-devops-2-toolkit>. [Accessed 12 10 2016].
- [4] James Turnbull. (2014). *The Docker Book Containerization Is the New Virtualization*. 1st ed. [ebook]. Available at: <https://www.amazon.com/Docker-Book-Containerization-new-virtualization-ebook/dp/B00LRROT14>. [Accessed 28 05 2016].
- [5] Docker Compose. [Online]. Available at: <https://docs.docker.com/compose/>. [Accessed 28 05 2016].
- [6] Docker Machine. [Online]. Available at: <https://docs.docker.com/machine/>. [Accessed 28 05 2016].
- [7] Docker vs VMs.[Online]. Available at: <http://devops.com/2014/11/24/docker-vs-vm/>. [Accessed 13 03 2016]
- [8] Continuous Integration. [Online]. Available at: <http://martinfowler.com/articles/continuousIntegration.html>. [Accessed 02 05 2016].
- [9] Continuous Integration Platform Using Docker Containers: Jenkins, SonarQube, Nexus, GitLab. [Online]. Available at: <https://blog.codecentric.de/en/2015/10/continuous-integration-platform-using-docker-container-jenkins-sonarqube-nexus-gitlab/>. [Accessed 02 05 2016].
- [10] Continuous Integration and Delivery of Microservices using Jenkins CI, Maven, and Docker Compose. [Online]. Available: <https://programmaticponderings.wordpress.com/2015/06/22/continuous-integration-and-delivery-of-microservices-using-jenkins-ci-maven-and-docker-compose/>. [Accessed 02 05 2016].
- [11] Apache Tomcat. [Online]. Available at: <https://help.ubuntu.com/lts/serverguide/tomcat.html>. [Accessed 23 04 2016]



- [12] Tomcat Web Application Deployment. [Online]. Available at: <https://tomcat.apache.org/tomcat-8.0-doc/deployer-howto.html>. [Accessed 23 04 2016]
- [13] Richard Stones, Neil Matthew. (2000). *Beginning Linux Programming*. 2nd ed. [ebook]. Available at: <http://www.wrox.com/WileyCDA/WroxTitle/Beginning-Linux-Programming-2nd-Edition.productCd-0764543733.html>. [Accessed 28 05 2016].
- [14] Oracle WebLogic Server on Docker Containers. [Online]. Available at: <http://www.oracle.com/technetwork/middleware/weblogic/overview/weblogic-server-docker-containers-2491959.pdf>. [Accessed 23 04 2016]
- [15] Understanding WebLogic Server Deployment. [Online]. Available at: <http://docs.oracle.com/middleware/1221/wls/DEPGD/understanding.htm#DEPGD1141>. [Accessed 10 05 2016]
- [16] Deploying Applications and Modules with weblogic.Deployer. [Online]. Available at: <http://docs.oracle.com/middleware/1221/wls/DEPGD/deploy.htm#DEPGD212>. [Accessed 10 05 2016]
- [17] Understanding the WebLogic Scripting Tool. [Online]. Available at: <http://docs.oracle.com/middleware/1221/wls/WLSTG/toc.htm>. [Accessed 10 05 2016]
- [18] Using the WebLogic Maven Plug-In for Deployment. [Online]. Available at: [http://docs.oracle.com/middleware/1221/wls/DEPGD/maven\\_deployer.htm#DEPGD383](http://docs.oracle.com/middleware/1221/wls/DEPGD/maven_deployer.htm#DEPGD383). [Accessed 10 05 2016]
- [19] WildFly. [Online]. Available at: <https://en.wikipedia.org/wiki/WildFly>. [Accessed 08 05 2016]
- [20] Application deployment. [Online]. Available at: <https://docs.jboss.org/author/display/WFLY10/Application+deployment>. [Accessed 08 05 2016]
- [21] Core management concepts. [Online]. Available at: <https://docs.jboss.org/author/display/WFLY10/Core+management+concepts>. [Accessed 08 05 2016]
- [22] Management Clients. [Online]. Available at: <https://docs.jboss.org/author/display/WFLY10/Management+Clients>. [Accessed 08 05 2016]

- [23] OpenShift Origin Documentation. [Online]. Available at: <https://docs.openshift.org/latest/welcome/index.html>. [Accessed 12 07 2016]
- [24] OpenShift Origin Project Github. [Online]. Available at: <https://github.com/openshift/origin>. [Accessed 12 07 2016]
- [25] OpenShift Ansible Project Github. [Online]. Available at: <https://github.com/openshift/openshift-ansible>. [Accessed 12 07 2016]
- [26] OpenShift Source-To-Image Project Github. [Online]. Available at: <https://github.com/openshift/source-to-image>. [Accessed 12 07 2016]
- [27] OpenShift Origin-Metrics Project Github. [Online]. Available at: <https://github.com/openshift/origin-metrics>. [Accessed 12 07 2016]
- [28] OpenShift Origin-Aggregated-Logging Project Github. [Online]. Available at: <https://github.com/openshift/origin-aggregated-logging>. [Accessed 12 07 2016]
- [29] How to Create an S2I Builder Image. [Online]. Available at: <https://blog.openshift.com/create-s2i-builder-image>. [Accessed 12 07 2016]
- [30] Binary input sources in OpenShift 3.2. [Online]. Available at: <https://blog.openshift.com/binary-input-sources-openshift-3-2>. [Accessed 04 08 2016]
- [31] Binary Deployments with OpenShift 3. [Online]. Available at: <https://blog.openshift.com/binary-deployments-openshift-3>. [Accessed 04 08 2016]
- [32] OpenShift 3.3 Pipelines – Deep Dive. [Online]. Available at: <https://blog.openshift.com/openshift-3-3-pipelines-deep-dive>. [Accessed 04 08 2016]
- [33] OpenShift 3 CI/CD Demo. [Online]. Available at: <https://github.com/OpenShiftDemos/openshift-cd-demo>. [Accessed 04 08 2016]
- [34] Why Red Hat Chose Kubernetes for OpenShift. [Online]. Available at: <https://blog.openshift.com/red-hat-chose-kubernetes-openshift>. [Accessed 17 10 2016]
- [35] Jenkins Documentation. [Online]. Available at: <https://jenkins.io/doc>. [Accessed 24 08 2016]

- [36] Pipeline Examples. [Online]. Available at: <https://jenkins.io/doc/pipeline/examples>. [Accessed 24 08 2016]
- [37] Pipeline Tutorial. [Online]. Available at: <https://github.com/jenkinsci/pipeline-plugin/blob/master/TUTORIAL.md>. [Accessed 24 08 2016]
- [38] Jenkins2 Pipeline jobs using Groovy code in Jenkinsfile. [Online]. Available at: <https://wilsonmar.github.io/jenkins2-pipeline>. [Accessed 24 08 2016]
- [39] Pipeline as Code – Continuous Delivery with Jenkins 2.0. [Online]. Available at: <https://grensesnattet.computas.com/pipeline-as-code-continuous-delivery-with-jenkins-2-0>. [Accessed 24 08 2016]

## Phụ lục A CÀI ĐẶT DOCKER TRÊN HỆ ĐIỀU HÀNH LINUX

Docker hiện hỗ trợ cài đặt trên hầu hết các hệ điều hành phổ biến Windows, Mac OS X và rất nhiều Linux distribution (Ubuntu, Red Hat Enterprise, CentOS, Fedora, Debian, v.v.). Vì chỉ chạy trên Linux kernel, nên với những hệ điều hành không dùng Linux kernel (Windows, Mac OS X), Docker cần chạy qua một lớp máy ảo.

Để xem hướng dẫn cài đặt cụ thể cho từng hệ điều hành được hỗ trợ, có thể xem [tại đây](#). Ở đây chúng tôi hướng dẫn cài đặt Docker chung lên một Linux kernel.

### A.1 Cài đặt Docker Engine

#### A.1.1 Tải về Docker Engine binaries cho Linux

Truy cập các URL sau để tải về bản mới nhất cho Linux

##### Danh sách A.1.1: Tải về bản mới nhất

```
https://get.docker.com/builds/Linux/i386/docker-latest.tgz
https://get.docker.com/builds/Linux/x86_64/docker-latest.tgz
```

Để tải về một phiên bản cụ thể, sử dụng mẫu URL sau

##### Danh sách A.1.2: Tải về một phiên bản cụ thể

```
https://get.docker.com/builds/Linux/i386/docker-<version>.tgz
https://get.docker.com/builds/Linux/x86_64/docker-<version>.tgz
```

#### A.1.2 Cài đặt

Sau khi tải về, dùng lệnh `tar` để xả nén gói nhị phân

### Danh sách A.1.3: Xả nén

```
$ tar -xvzf docker-latest.tgz
docker/
docker/docker-containerd-ctr
docker/docker
docker/docker-containerd
docker/docker-runc
docker/docker-containerd-shim
```

Ta cần cài đặt gói nhị phân với biến môi trường *PATH*. Cách đơn giản là copy gói nhị phân vào thư mục */usr/bin*

### Danh sách A.1.4: Copy vào thư mục */usr/bin*

```
$ mv docker/* /usr/bin/
```

Chú ý: Nếu đã có Docker Engine cài đặt sẵn trên máy, chúng ta phải dừng các Engine lại bằng lệnh trước khi cài đặt bằng lệnh *killall docker* và cài đặt gói nhị phân vào cùng thư mục. Có thể tìm được thư mục cài đặt Docker bằng lệnh *dirname \$(which docker)*

## A.1.3 Chạy Engine

Để chạy Docker Engine, sử dụng lệnh sau:

### Danh sách A.1.5: Chạy Docker Engine

```
$ docker daemon &
```

## A.1.4 Đặt quyền truy cập cho non-root

Docker daemon luôn chạy với quyền root. Nhưng chúng ta có thể chạy docker client dưới quyền user với điều kiện user này nằm trong *docker* group. Do đó, những gì ta cần làm là tạo một group *docker* và thêm user vào đó, sau đó chúng ta có thể sử dụng docker với docker client mà không cần quyền root (docker daemon vẫn sẽ chạy dưới quyền root).

## A.2 Cài đặt Docker Compose

Trước khi cài đặt Docker Compose, chúng ta cần cài đặt Docker Engine trước.

1. Dùng *curl* để tải gói Docker Compose về, phiên bản mới nhất hiện tại (5-2016) là 1.7.1.

### Danh sách A.2.1: Tải về docker compose

```
$ curl -L
https://github.com/docker/compose/releases/download/1.7.1/docker-compose
'uname -s' - 'uname -m' > /usr/local/bin/docker-compose
```

2. Đặt quyền thực thi cho docker compose

### Danh sách A.2.2: Đặt quyền thực thi cho docker compose

```
$ chmod +x /usr/local/bin/docker-compose
```

3. Kiểm tra lại sự cài đặt

### Danh sách A.2.3: Hiện thị Docker Compose version

```
$ docker-compose --version
docker-compose version: 1.7.1
```

## A.3 Cài đặt Docker Machine

1. Tải về và cài đặt Docker Machine.

### Danh sách A.3.1: Tải về docker compose

```
$ curl -L
https://github.com/docker/machine/releases/download/v0.7.0/docker-machine
- 'uname -s' - 'uname -m' > /usr/local/bin/docker-machine && \
$ chmod +x /usr/local/bin/docker-machine
```

2. Kiểm tra cài đặt bằng cách hiển thị Docker Machine version

**Danh sách A.3.2: Hiển thị Docker Machine version**

```
$ docker-machine version  
docker-machine version 0.7.0, build 61388e9
```

# Phụ lục B CÀI ĐẶT CÁC CÔNG CỤ ĐỂ GIÁM SÁT VÀ VẬN HÀNH TRÊN OPENSIFT ORIGIN

## B.1 Origin Metrics

1. Truy cập vào openshift-infra project và tạo metric-deployer service account

### Danh sách B.1.1: Tạo metrics-deployer service account

```
$ oc create -f - <<API
  apiVersion: v1
  kind: ServiceAccount
  metadata:
    name: metrics-deployer
  secrets:
    - name: metrics-deployer
API
```

2. Trước khi deploy các thành phần của origin metrics thì metrics-deployer service account phải được cấp quyền để có thể chỉnh sửa trong openshift-infra project.

### Danh sách B.1.2: Tạo metrics-deployer service account

```
$ oadm policy add-role-to-user edit
  system:serviceaccount:openshift-infra:metrics-deployer
$ oadm policy add-cluster-role-to-user cluster-reader
  system:serviceaccount:openshift-infra:heapster
```

3. Deploy các thành phần của origin metrics

### Danh sách B.1.3: Deploy các thành phần của origin metrics

```
$ oc new-app -f metrics-deployer.yaml \
  -p HAWKULAR_METRICS_HOSTNAME=hawkular-metrics.example.com \
  -p USE_PERSISTENT_STORAGE=false \
```

Với HAWKULAR\_METRICS\_HOSTNAME: địa chỉ để truy cập vào trang web của Hawkular Metrics



USE\_PERSISTENT\_STORAGE: giá trị true cho persistent storage, false có nghĩa không sử dụng persistent storage

4. Cuối cùng, chỉnh sửa lại file master-config.yaml và restart lại service origin master

### Danh sách B.1.4: Chỉnh sửa lại file master-config.yaml và restart lại service origin master

```
$ vi /etc/origin/master/master-config.yaml
...
assetConfig:
  masterPublicURL: https://openshift-master.example.com:8443
  publicURL: https://openshift-master.example.com:8443/console/
  metricsPublicURL: "https://hawkular-metrics.example.com"
...
$ systemctl restart origin-master
```

## B.2 Origin Aggregated Logging

Một hệ thống Aggregated Logging gồm nhiều thành phần được viết tắt là "EFK":

- ElasticSearch: một đối tượng lưu giữ tất cả các file log
- Fluentd: tập hợp tất cả các file log từ các node và đưa cho ElasticSearch
- Kibana: một web UI cho ElasticSearch
- Curator: cho phép admin có thể loại bỏ các log cũ trên ElasticSearch

1. Tạo một project cơ tên logging và truy cập vào project vừa tạo

### Danh sách B.2.1: Tạo một project cơ tên logging và truy cập vào project vừa tạo

```
$ oadm new-project logging --node-selector=""
$ oc project logging
```

2. Tạo các service account hỗ trợ và cấp quyền cho chúng

### Danh sách B.2.2: tạo các service account hỗ trợ và cấp quyền cho chúng

```
$ oc new-app logging-deployer-account-template
$ oadm policy add-cluster-role-to-user oauth-editor \
  system:serviceaccount:logging:logging-deployer
$ oadm policy add-scc-to-user privileged \
  system:serviceaccount:logging:aggregated-logging-fluentd
$ oadm policy add-cluster-role-to-user cluster-reader \
  system:serviceaccount:logging:aggregated-logging-fluentd
```

## 3. Deploy EFK Stack

### Danh sách B.2.3: Deploy EFK Stack

```
$ oc new-app logging-deployer-template \
  -p KIBANA_HOSTNAME=kibana.example.com \
  -p ES_CLUSTER_SIZE=1 \
  -p PUBLIC_MASTER_URL=https://openshift-master.example.com:8443
```

Với KIBANA\_HOSTNAME: địa chỉ để truy cập vào trang web của kibana

ES\_CLUSTER\_SIZE: số lượng container Elasticsearch được deploy

PUBLIC\_MASTER\_URL: địa chỉ của openshift origin master

## 4. Cuối cùng, chỉnh sửa lại file master-config.yaml và restart lại service origin master

### Danh sách B.2.4: Chỉnh sửa lại file master-config.yaml và restart lại service origin master

```
$ vi /etc/origin/master/master-config.yaml
...
assetConfig:
  masterPublicURL: https://openshift-master.example.com:8443
  publicURL: https://openshift-master.example.com:8443/console/
  metricsPublicURL: "https://hawkular-metrics.example.com"
  loggingPublicURL: "https://kibana.example.com"
...
$ systemctl restart origin-master
```