**The Report Committee for Robert Sandoval**
**Certifies that this is the approved version of the following report:**

# A Case Study in Enabling DevOps Using Docker

**APPROVED BY**
**SUPERVISING COMMITTEE:**

**Supervisor:**

Craig Chase

Adnan Aziz

# A Case Study in Enabling DevOps Using Docker

**by**

**Robert Sandoval, B.S.**

**Report**

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

**Master of Science in Engineering**

**The University of Texas at Austin**

**August 2015**

# Dedication

I would like to dedicate this to my parents, Juan and Lupe for being there to support me in all my endeavors.

## Acknowledgements

I am deeply grateful to Dr. Craig Chase for acting as my supervisor as well as Dr. Adnan Aziz for acting as my reader. I would also like to express my gratitude to the entire staff of the CLEE organization whom where always professional and extremely helpful.

# Abstract

## A Case Study in Enabling DevOps Using Docker

Robert Sandoval, M.S.E

The University of Texas at Austin, 2015

Supervisor: Craig Chase

There are varying definitions for DevOps, but at its core, it is a software engineering methodology. The concept of DevOps was born from the need to create synergy between the people who develop software (Dev), and the operations people who manage production systems (Ops). The result is a methodology that strives to create a streamlined approach to the full lifecycle of software-from development to production support and maintenance. DevOps methodologies do so by implementing automation that enables a repeatable process for building, testing, deploying and managing software components. Docker is a platform that allows for the packaging of these software components, as well as any dependencies, into isolated virtual containers. These Docker containers are portable across any Linux based infrastructure that has Docker installed. DevOps processes that are supported with Docker enable developers to produce self-contained applications that are delivered in a fast, repeatable way.

Docker is relatively new within the technology community and is gaining wide support. In this report I evaluate features available within Docker. I also implement and

describe a system to build and deploy Docker containers. These containers could be used for deployment to any Linux based server or public cloud offering that supports Docker. Finally, I identify issues that would need to be considered when using Docker, as well as provide topics for future work.

# Table of Contents

# List of Figures

# Chapter 1: Introduction

Modern software development is going through a transition. Developers are asked to deliver projects faster and also adapt to changes much more quickly. There is also a push to give the development community access to functions that would normally be handled by operations teams. These functions include deployment, installation and day-to-day management of application running on production systems. These functions are enabled through automation and allow developers to accomplish software project efforts more efficiently. These requirements have caused a wide range of enabling technologies to be developed.

Several changes to the IT industry are coming about and gaining popularity. Agile development and Extreme programming methodologies are becoming widely popular as a way to deliver changes more quickly. From an operational perspective, virtualization is widely used in datacenters. Servers are thought of as software and can be created, destroyed, enabled or disabled the same way a software application can be. This paper attempts to look at developing a process to enable DevOps within an enterprise by using virtualized containers for application deployment.

DevOps is a methodology that creates a partnership between the development and operations communities. This methodology involves implementing processes and automation to streamline the software development lifecycle. DevOps is a relatively new concept. In essence it “aims to help an organization rapidly produce software products

and services and to improve operations performance" [2]. There are several technologies that aid the development of a DevOps methodology. I will discuss some of them below.

### 1.1 Software Development Lifecycle

The historical approach to software development projects typically involved large efforts that utilized the Waterfall method or something similar as their software development lifecycle methodology. Agile programming is gaining popularity within the IT community. It attempts to develop artifacts rapidly and push smaller changes at a more frequent pace. Most agile development methods break the tasks into small increments with minimal planning and do not directly involve long-term planning [3]. Typically software components are implemented in smaller units and deployed to production more frequently. This process is repeated to include new features into the software. The DevOps approach is similar. DevOps attempts to allow developers to make changes more frequently. DevOps and Agile development share similar methodologies in this sense.

### 1.2 Cloud computing

Cloud computing is a term used to describe the abstraction of the datacenter from developers. "At the foundation of cloud computing is the broader concept of converged infrastructure and shared services." [4]. The overall approach is to allow developers the ability to deploy applications to a cloud platform without needing to know anything about the underlying infrastructure. Virtualization has enabled cloud platforms to come to fruition. This result is primarily due to the fact that virtual machines are treated as

software. Virtual machines created at scale to absorb workload and shut down when they are not needed. The primary limitation is developers still need knowledge of the underlying virtual machine in most cases. The need for system level configurations or knowledge of the underlying servers takes away from this cloud abstraction model.

Linux Containers (LXC) is "an operating-system-level virtualization environment for running multiple isolated Linux systems (containers) on a single Linux control host" [7]. Linux containers give us an opportunity to implement a cloud model that allows software to be independent of where that software is deployed. In essence, it allows for what would seem is a very small virtual machine instance. That instance could be massively clustered if required. Due to the shared kernel (as well as operating system libraries), an advantage of container-based solutions is that they can achieve a higher density of virtualized instances, and disk images are smaller compared to hypervisor-based solutions. [5]

### 1.3 Project Scope

The goal of this project is to evaluate some key technologies to enable DevOps. This project will do so by evaluating the Docker platform[6]. Docker is a Linux container framework that allows developers to package an artifact along with its dependencies. The resulting container could be deployed to any system containing the Docker platform. There are several objectives addressed by the evaluation in this paper. We will show how containers can aid Agile development practices and enable true cloud enabled artifacts for use in the datacenter. The report will also develop a system for the enablement of

DevOps processes to evaluate key aspects of using of the framework. Some of the key features evaluated will be:

- Aspects of building containers
- Container Management
- Target environment deployment issues

The project will describe a system for deploying and managing J2EE-based applications within a JBoss Application Server container. Primarily the report will address how to build and manage containers regardless of being deployed to a test or production system.

### 1.4 Report Outline

The remainder of the report is broken down as follows: Chapter 2 will deal with an overview of the technology stack used for the project. We will cover specifically Linux Container, the Docker framework and aspects of JBoss that were taken into account for evaluation of the technologies. Chapter 3 will expand on the current architecture of the proposed system. Specifically it will delve into the container build and deployment process as well as aspects of the using JBoss Application Server for the reference application. Chapter 4 will provide the results of the project. It will address how and why certain things were done and some alternatives that were explored. Chapter 5 provides the conclusion, along with sections for future and related work.

# Chapter 2: Technology Overview

Various technologies were evaluated for the implementation of this report. Here I will outline the technologies used in the Docker container build and deploy system.

## 2.1 Linux

One of the requirements of the evaluation was the use of the Docker platform for containers. Docker relies on several key features of Linux and is currently not available on any other Operating System. For the scope of this report and evaluation, CentOS 7 was used as the primary operating system used. Docker recommends a kernel version of 3.10 or higher.

## 2.2 Linux Containers

Container technology is built around a few features available as part of the Linux kernel. Specifically LXC provides for an operating system level virtualization environment for Linux. "LXC relies on the Linux kernel cgroups functionality that was released in version 2.6.24. It also relies on other kinds of namespace-isolation functionality, which were developed and integrated into the mainline Linux kernel" [7].

## 2.3 Docker

Docker is a relatively new technology built on top of LXC. It was released in March of 2013 as an open source project under the Apache License 2.0. The current version for Docker is 1.6. From Docker's documentation: "Docker is an open platform for developers and sysadmins to build, ship, and run distributed applications" [6]. Docker has several aspects that will be covered in sections 2.3.1 through 2.3.9. Specifically:

- Images
- Dockerfiles
- File Systems - AUFS
- Containers
- Networking
- Management
- Security

### 2.3.1 Docker Images

Docker images are the basis of all containers. They provide a read-only file system that contains the base operating system. Docker images are readily provided by various vendors and contain stripped down versions of the operating system. The file system is not however based on the host server's file system. They are self-contained. One unique aspect about Docker is it allows any Linux-based operating system to be used as a container, provided they are based off of the same kernel. You can run an Ubuntu container on a host running CentOS without issue. You can build images on top of one

another to create new images. You will hear the term “layers” as well when working with Docker. The terms refer to the same thing. The layers that make up an image are images in and of themselves. See Figure 1 to show the base image root file system.

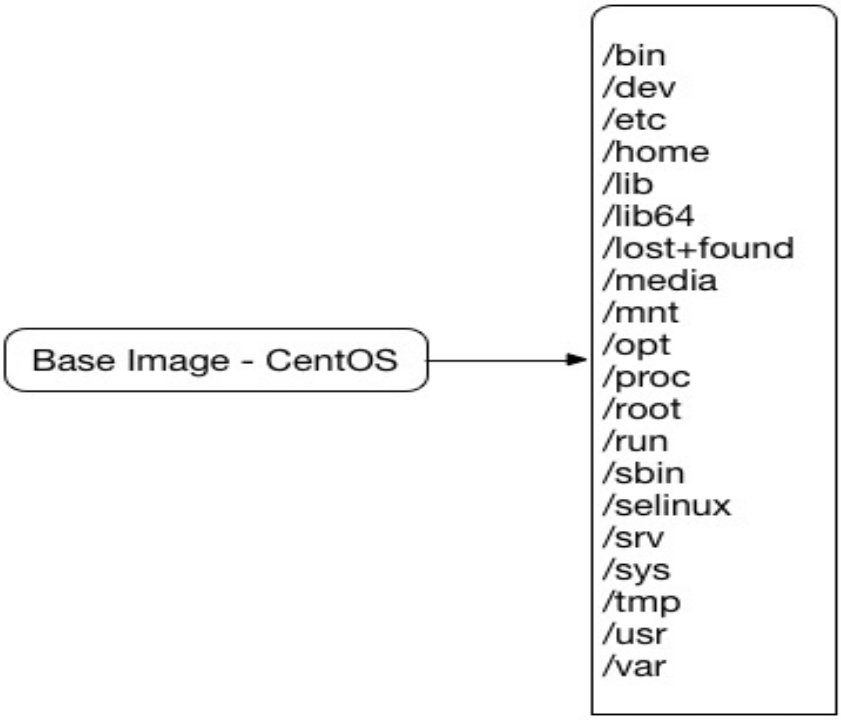


Figure 1: CentOS 7 Base Image

### 2.3.2 Dockerfiles

Dockerfiles are developer-defined blueprints to build custom Docker images. Some of the key features allowed by Dockerfiles are listed below:

- Add software
- Set environment variables for your container
- Define TCP ports that will be exposed outside of the container

- Run commands inside the container during build time (executed in bash shell).
- Optionally set the entry point for the container. An entry point is the defined startup process for the container.

The process of building an image involves executing each instruction within a Dockerfile in its own layer and at the end merging those layers into a final image. The process is illustrated below.

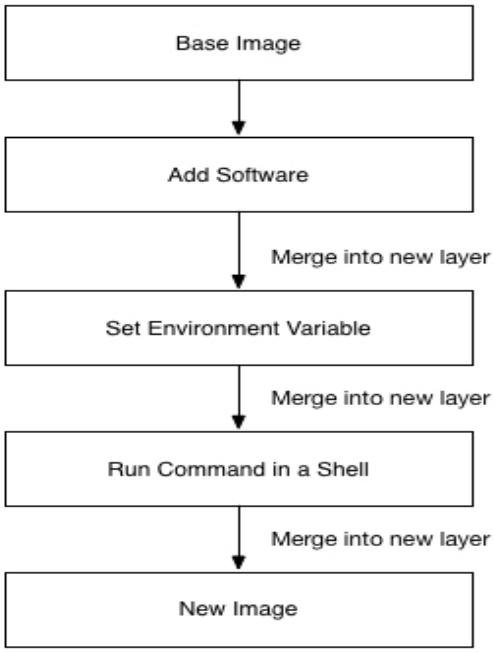


Figure 2: Dockerfile Layering Process

### 2.3.3 File Systems - AUFS

AUFS is a union file system. It is the basis of how Docker creates layers to build a complete image. In respect to Docker, the base image's file system is loaded as read-only. Each layer that is added when building a child image from the base image creates another file system over the base one. All the file systems within an image are read-only. It is not until a container is instantiated that a read-write layer is added.

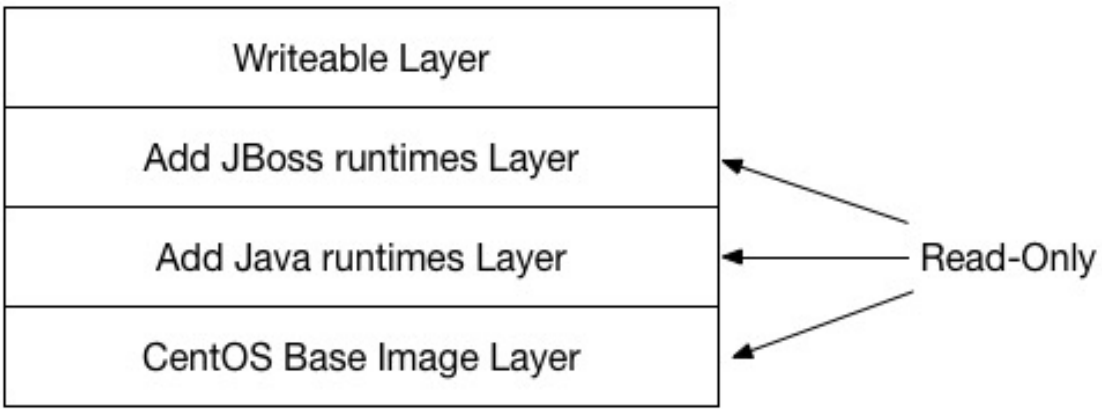


Figure 3: AUFS Layers

### 2.3.4 Containers

Docker containers are runtime instances of images. You can start many containers based off the same image. Each container has a unique identifier either provided by the Docker daemon or the identifier can be user-defined. If an entry point is defined for the container that process will be initiated at startup. If no entry point is defined, you can define one when you start the container.

A container only has access to its own instance of the layered file system. The container does get a read-write layer when its instantiated. If a container needs to edit a file that is part of one of the read-only lower layers, that file is copied up into the top level read-write layer. The container is able to edit the file, however the file is not persisted back to the base image. This provides isolation for each container. It is worth noting that containers share layers only in the sense that they can be modified in one artifact. That artifact is the image created when building the image from the Dockerfile. Each container gets a copy of the image at runtime.

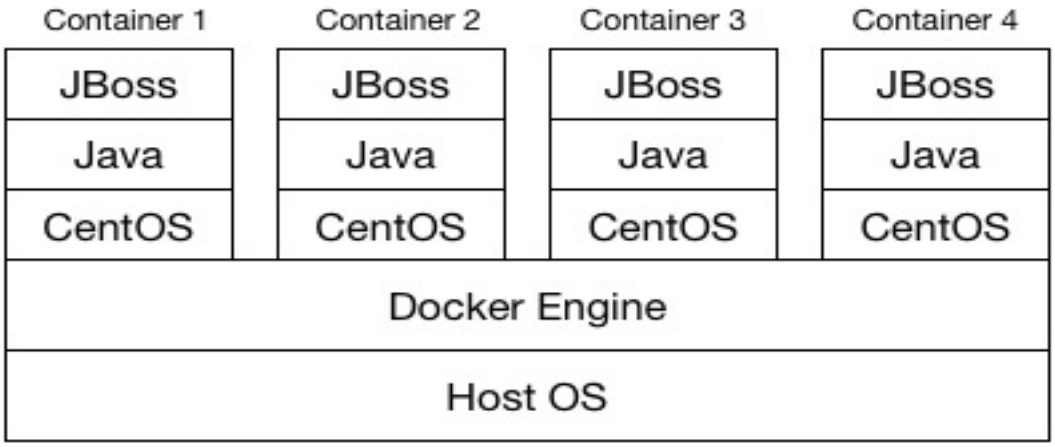


Figure 4: Docker Containers

### 2.3.5 Networking

Containers are started with their own network stack. By default each container is provided with an interface that allows it to communicate with the host system and other Docker containers. You can also configure the Docker container to use the network stack of the host system or manually configure the entire networking stack to your needs. There are several features provided by the Docker platform enable you to create custom networking configurations.

### 2.3.6 Container ports

Container ports by default do not expose themselves outside of the container. You must specify each port you want to have access outside the container at container creation time. You do this by defining a host-to-container port mapping. As you can see [Figure 5] every container is the same and advertises port 80, but each is mapped differently to list of advertised ports by the host. This is a key aspect to running multiple copies of the same image.

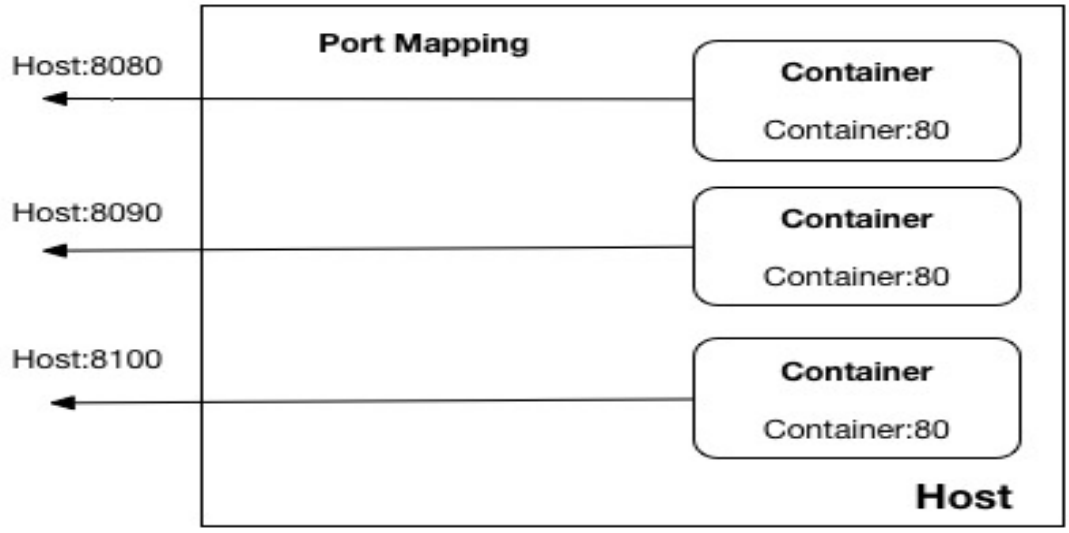


Figure 5: Port Mappings

### 2.3.7 Container DNS and Host file entries

Containers use the host system's configuration to determine it's primary DNS server. Each container can override this setting if a specific startup option is passed. This setting allows for the possibility of each container to resolve names differently. There is also the option to inject host file entries to resolve network addresses at container creation time. Host files are useful for overriding hostnames and provides much of the same functionality of pointing to different DNS servers for name resolution.

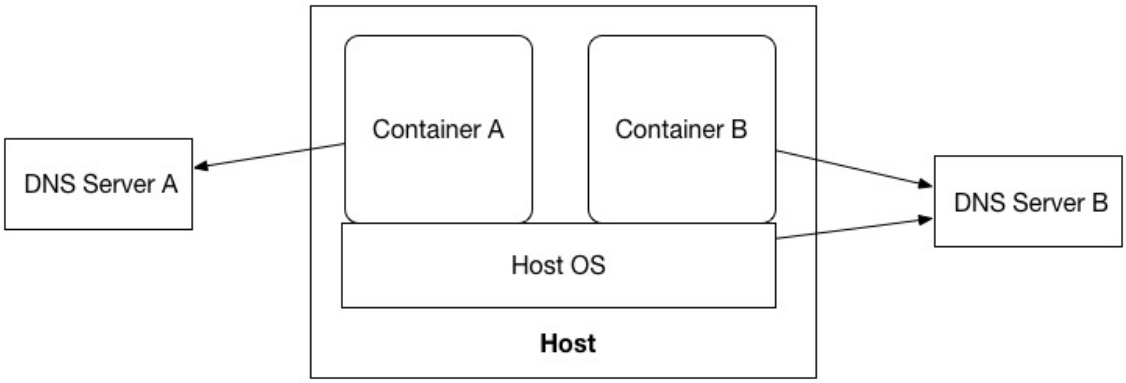


Figure 6: Containers using different DNS servers

### 2.3.8 Management

The Docker framework provides mechanisms for image and container management. You have the ability to pull base images from Docker's own repository. Docker provides tools to build new images, import images, or export images. It also provides for versioning of images.

### 2.3.9 Security

Docker security is based on the technologies that Docker uses to create containers. Specifically they use kernel namespaces, which isolate the container from the host system. The also use cgroups, or control groups, to help limit what a container can do. Cgroups specifically define the amount of resources a container can use. Specifically it allows for restrictions on the amount of CPU and memory resources utilized. This is

beneficial in that a container is limited and can't exhaust the host system's resources if the container becomes compromised.

### 2.4 Application Layer

A reference application was created as part of the research for the project. The application is a J2EE application written in Java. This was chosen in part to test the aspects of implementing a muti-tier system. As three-tier application was created with an Apache HTTP Server, JBoss middleware application and MongoDB database backend. The reference application generates a static HTML page with the output of a single record from the database server. This record is the name of the database server and was implemented to show how a container can point to various database servers without modification. It strictly uses container startup options to alter what database server to target.

### 2.5 Apache HTTP Web Server/ ModCluster

Apache HTTP web server was used as the primary load balancer for the reference application. ModCluster is an Apache package that is regularly used for integration with JBoss Application Server. One of the reasons ModCluster was chosen is that the application sever can be set up to dynamically connect to and register itself with the ModCluster module installed in Apache HTTP web server. This provides for a more dynamic runtime. JBoss Application Server instances can engage and disengage themselves from the Apache HTTP web servers load balancing process as needed.

### 2.6 JBoss Application Server

JBoss application server is a J2EE-compliant web container supporting Java-based web applications. I am using JBoss Enterprise Application Platform version 6.3. It was chosen specifically for its dynamic integration with Apache and ModCluster. The JBoss Application Server platform also provides for two different runtime configurations. The first is defined as Domain mode and is used for multi-node installations. JBoss Application Server also has a Standalone mode, which was used for this effort. Standalone mode provides a well-encapsulated version of the application server and is meant to run as a single instance. Standalone mode provides a single directory structure for installation and fit will with Dockers layered process for image creation.

# Chapter 3: Architecture

The architecture used to evaluate the Docker ecosystem is described in this section. The primary use-case states that a developer should be able to build a Docker container running an instance of JBoss Application Server. The developer should be able to deploy that container to either a test or production system. The Docker container should not need to be modified to run on a particular system.

## 3.1 Development Platform

The development platform used was an Eclipse Integrated Development Environment. Eclipse provides the Ant[1] libraries used in building and deploying the Docker containers. Figure 7 depicts that development and unit testing of the reference application are done on a local instance of JBoss Application Server.

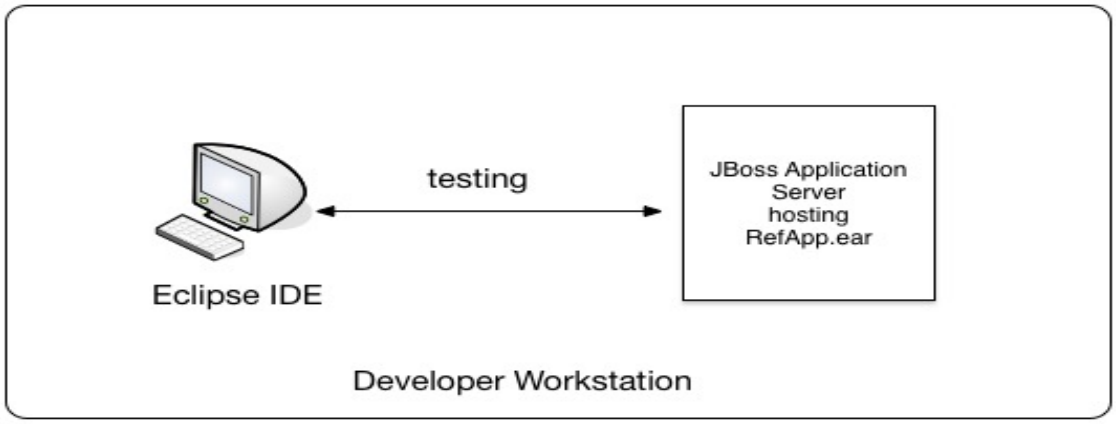


Figure 7: Developer Workstation Development Environment

[1] https://ant.apache.org

### 3.2 Reference Application

The reference application developed was a Java-based web application. The application created was used to connect to a database and display the contents of a single database record. The record provides the name of the physical server the database is running on. The application's output was used to confirm the Docker container connected to the correct database.

### 3.3 Server Layout

Virtual Machines running CentOS version 7 were created to test various the deployment process. The following servers were created:

- prodbuild11 – primary Docker build server
- proddoc11 – production Docker container server
- prodmongo11 – production database server
- testdoc11 – test Docker container server
- testmongo11 – test database server

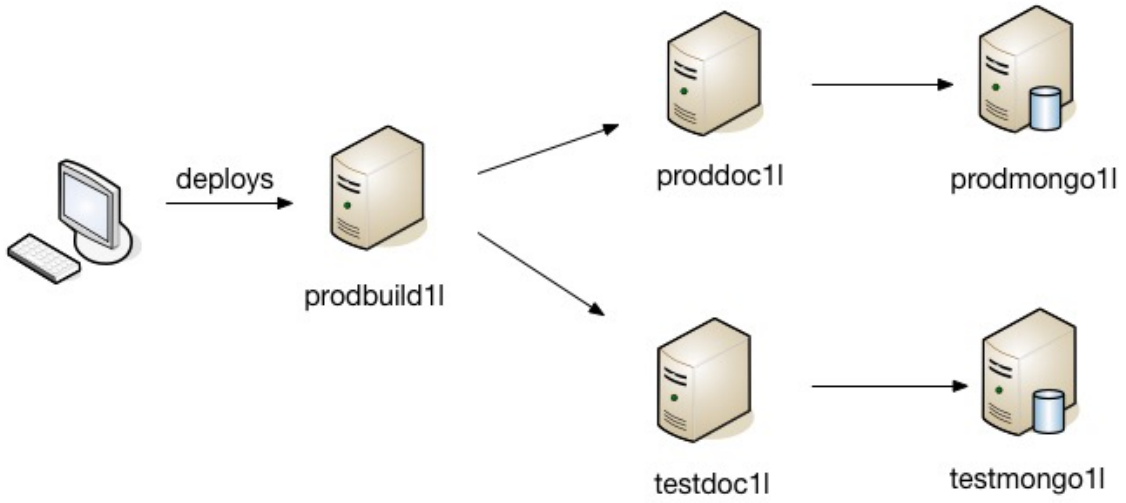


Figure 8: Server layout

The Docker build server is an intermediary server used only to build the Docker image that will be used for deployment to the target servers. In this system the build server is a separate server, but creating the container could easily be done locally on any Linux workstation provided the Docker platform was installed.

The Docker container server, proddoc1l, also acts as an Apache HTTP web server that was used to load balance traffic to JBoss containers running on the system. See Figure 9 below.

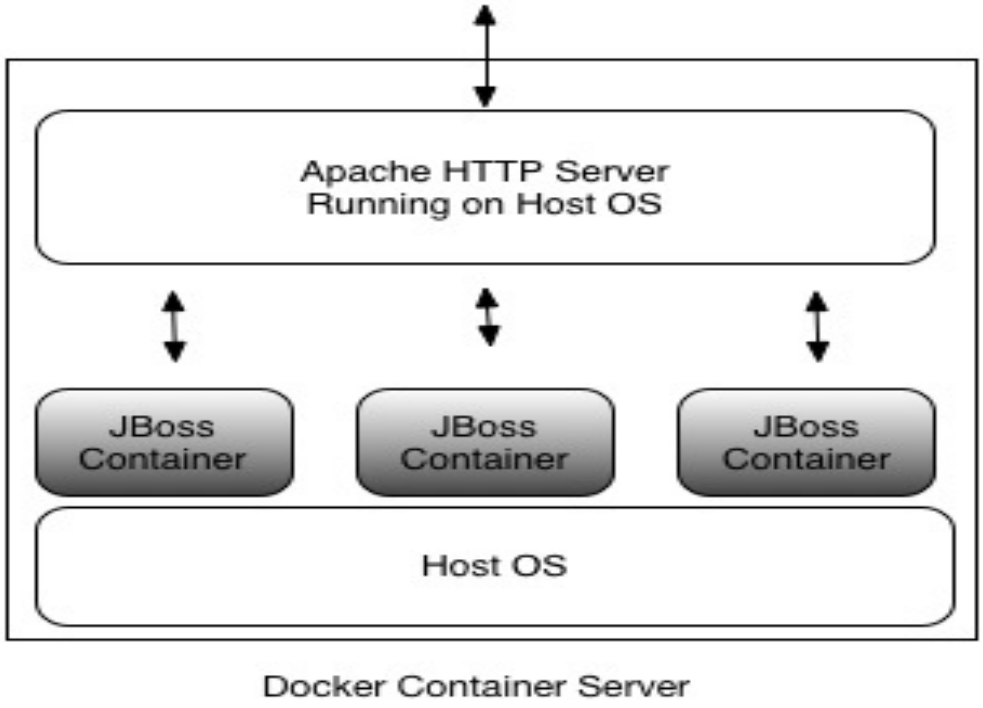


Figure 9: Docker Container Server Architecture

The database servers were installed with MongoDB and only used as a backend layer to test the viability that the same Docker container could connect to a production or test database without modification.

### 3.4 Container Operations

The container build process has two parts. The first is the process of packaging the JBoss Application Server used on the developer's local workstation into a compressed file. The compressed file was then copied over to the build server. The second process

packages the compressed file into the Docker image format. The Docker image was used to create the running containers.

### 3.4.1 Developer Workstation Application Packaging

The developer should build and test their applications on a local instance of JBoss Application Server. JBoss Application Server is contained in a single directory structure. In this case JBoss Application Server resides in the following location:

- /opt/jboss-eap-6.3/

The application deployed to JBoss Application Server running in Standalone mode is located at this location:

- /opt/jboss-eap-6.3/standalone/deployments/RefApp.ear

The application packaging process used an Ant script to package the application and copy it to the build server. The Ant script provides several functions listed below:

- Builds a tar archive file of the /opt/jboss-eap-6.3 directory
- Copies the tar file to the build server
- Remotely executes a container build
- Remotely executes a deployment to either a test or production Docker Container Server.
- Remotely starts a container on a test or production system.

### 3.4.2 Container Build

The container build process happens on the build server. Scripts take an existing base image and create a new image that contains the JBoss Application Server instance that was copied from the developer's workstation. The base CentOS image lacks the Java Runtime Environment needed to run the application. A new image was created to use as the base system for our JBoss container. The container build process is illustrated in Figure 10. The Dockerfiles used to create the Java-based CentOS image as well as the one used to create the JBoss containers are listed in Figure 11 and Figure 12.

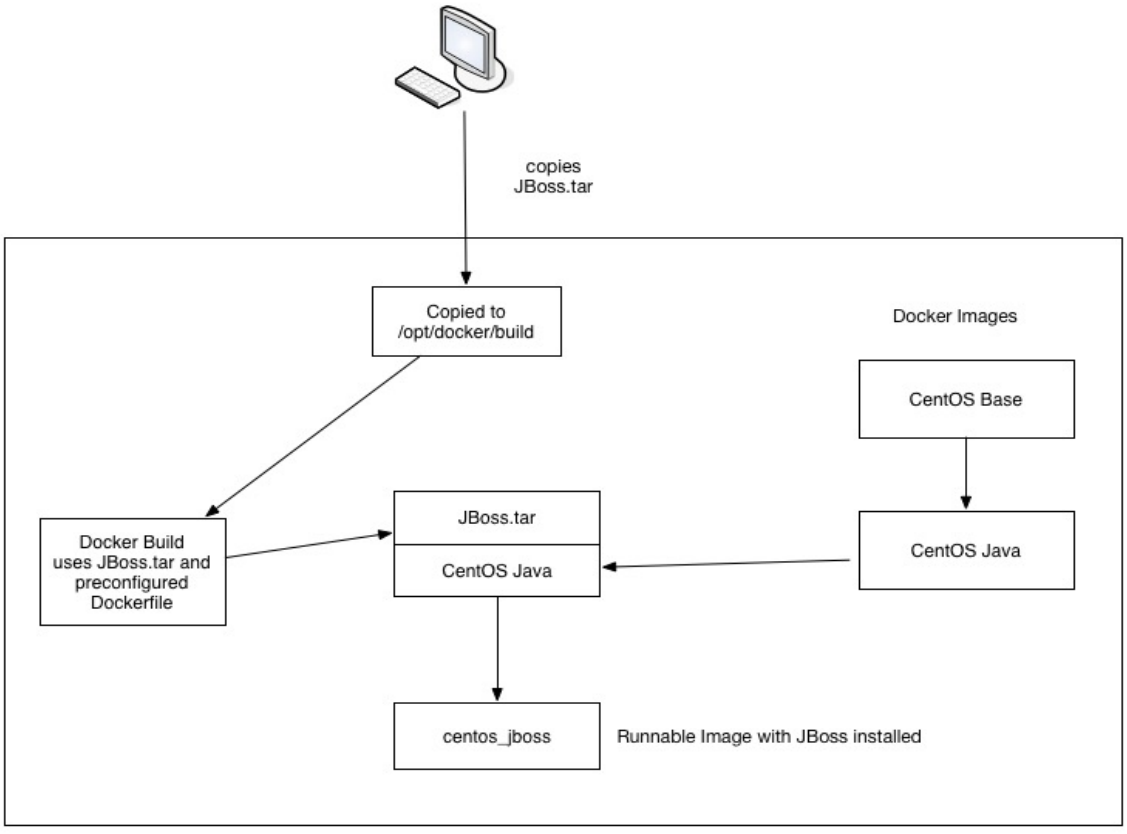


Figure 10: Container Build Process

Docker file used to generate the centos_java image

```
# Use base image CentOS, latest version
FROM centos:latest

# Install java, executes in a bash shell
RUN yum -y install java-1.7.0-openjdk.x86_64

#Sets home directory for Java
ENV JAVA_HOME /usr/lib/jvm/java-1.7.0-openjdk-1.7.0.75-2.5.4.2.el7_0.x86_64/jre
```

Figure 11: centos_java Dockerfile

Docker file used to generate centos_jboss image

```
#Use previously built centos_java image as base
FROM centos_java

# Adds JBoss tar to image. This command extracts the tar into /jboss
ADD jboss.tar /jboss

#Sets JBOSS_HOME environment variable
ENV JBOSS_HOME /jboss

#Updates permissions on /jboss. This executes in a bash shell
RUN chmod -R 755 /jboss

#Startup command for the container. Any container started will run this process
CMD ["/jboss/bin/standalone.sh", "--server-config=standalone-ha.xml", "-b", "0.0.0.0"]

#Defines what ports the container can expose to the host system
EXPOSE 8080
```

Figure 12: centos_jboss Dockerfile

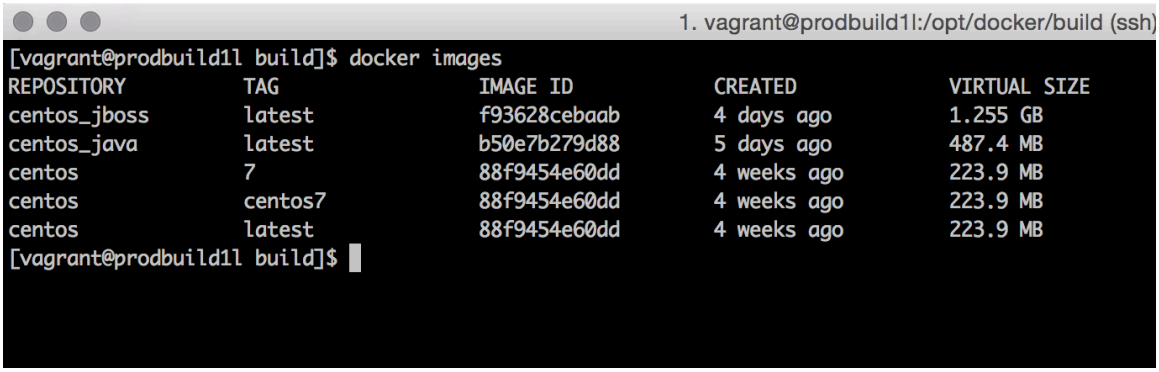


Figure 13: Docker image list

### 3.4.3 Container Deploy

The container deployment process is initiated from the developer workstation through the use of Ant scripts. The servers have been setup to allow secure copies from the production build servers to the target Docker container servers. The developer initiates the deployment by remotely executing a script with the target server passed as an argument.

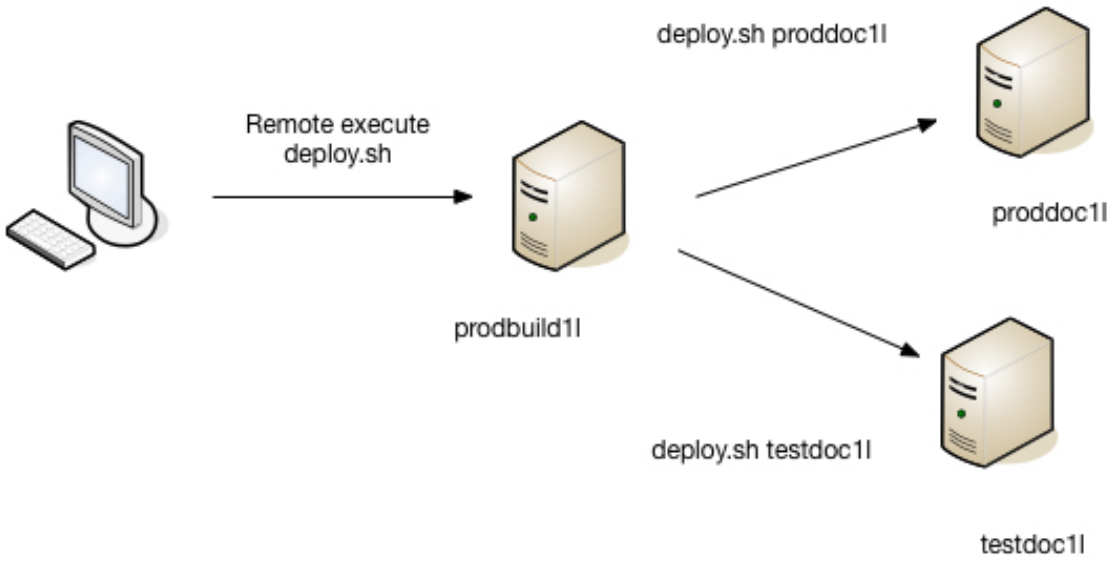


Figure 14: Container Deploy Flow

### 3.4.4 Container Startup

The Docker container startup process is initiated by the execution of an Ant script. There are some aspects to the startup that require some knowledge of the target server. Specifically the target server name must be defined in the Ant script. Also any specific custom host file entries must be passed in to the startup command listed in the Ant Script. A typical startup command would look similar to the commands shown in Figure 15.

Production Docker Container start up command

```
docker run -d --name prodcontainer1 -p 8080:8080 --add-
host=mongodb:192.168.88.10 centos_jboss
```

Test Docker Container start up command

```
docker run -d --name testcontainer1 -p 8080:8080 --add-
host=mongodb:192.168.77.10 centos_jboss
```

Command reference:

```
          -d : Run in Daemon mode
      --name : Assign the container a name
           -p : defines host:container port mappings. Exposes the 8080
                port from the container as 8080 on the host
  --add-host : Adds an entry into containers /etc/host file for local DNS
                resolution
centos_jboss : base image the container should be started from
```

Figure 15: Docker Container Startup Commands

Each container is given a name, port mapping definition and custom entries to include in its host file. Both containers are instantiated from the centos_jboss image described in section 3.4.2.

# Chapter 4: Project Results

The project was able to successfully build and deploy JBoss Application Server packaged inside a Docker container. The process allows for the developer to have control of the build and deploy process. The deployable Docker image was environment agnostic. The same Docker image was used on both a test and production system.

## 4.1 Build Process Results

The build process required some initial setup. The base CentOS Docker image was not usable in its initial configuration. The main component missing from the initial configuration was the Java Runtime Environment. To get the functionality we needed, a new image was created from the base CentOS image. The Java Runtime Environment was installed and a new Docker image was created. The resulting Docker image was used as the base for creation of the Docker image with JBoss Application Server installed.

The JBoss Application Server Docker image was over 1 Gigabyte in size. Figure 13 illustrates that our initial base image was 224 Megabytes. Adding the Java JRE layer the image size increased to 486 Megabytes. Adding the JBoss Application Server layer further increased the image size to 1.26 Gigabytes. Networking performance must be taken into consideration when transferring a large number of Docker images. However, once a Docker image is copied to a target server, several instances of the Docker container can be instantiated from a single Docker image.

Some features provided by the Docker platform would enable us to minimize the image size. Docker has the ability to mount volumes from the host server's file system. It is possible to have the JBoss Application Server runtime mounted on the host server's file system and have the container pull in the volume at runtime. The approach in Figure 15 was not pursued as part of this implementation, primarily because it breaks the self-containment of the Docker container. The resulting architecture is illustrated just for reference.

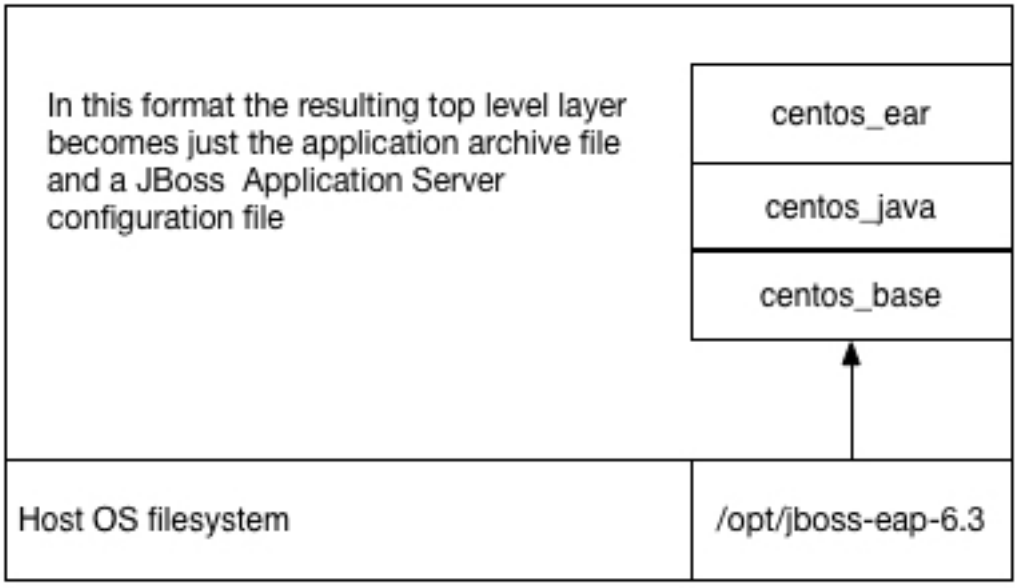


Figure 16: Docker Container with JBoss mounted as a volume

### 4.2 Deploy Process and Container Management

The deploy process is relatively straightforward. Utilizing Ant scripts on the developer workstation gives control of the entire process to the developer. Ideally the

functions provided by the Ant scripts would be re-implemented in a framework that provides better security. This would also prevent erroneous commands from being inadvertently executed on a production server.

The goal of creating a container that could run in any environment was achieved primarily through the use of several Docker features. The ability to add host file entries or point to an alternate DNS server enables the goal of the project. The reference application's connection parameters were abstracted. The Java-based web application does not reference the database server directly. It uses an alias illustrated in Figure 17. The alias used is "mongodb" which would point to either the test or production database server.

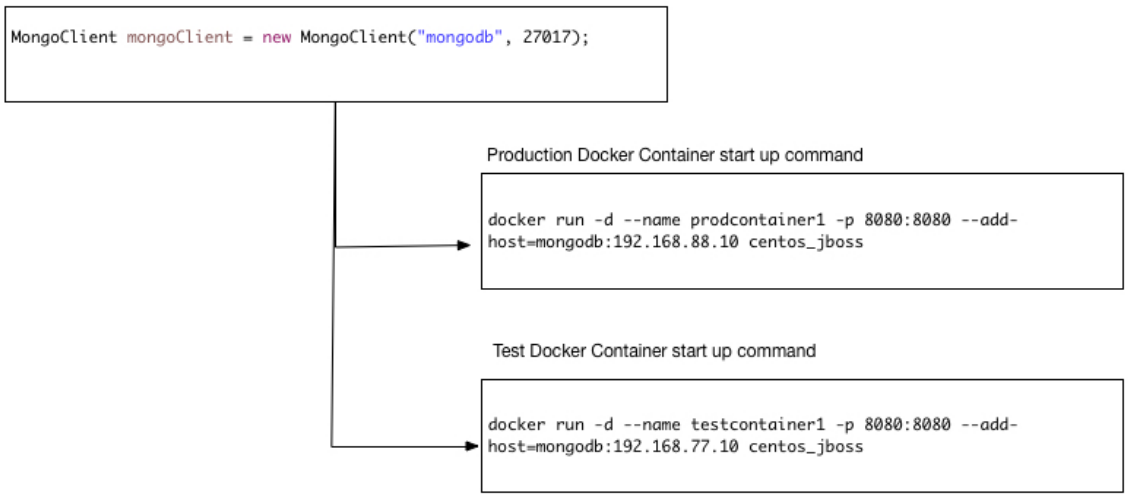


Figure 17: Application with database alias

One caveat to this approach is the port number must be static. There is no way to change the port number in the application without building a separate version of the application for a test versus a production deployment. The flexibility of adding host file

entries at the container level allows test and productions instance of the container to run together on the same host if needed. The host server in this case would become both a test and a production server.

# Chapter 5: Conclusion

The goal of the report was to evaluate the Docker platform as a suitable technology to enable the building and deploying a single Docker container to a production system. The goal was to enable a portion of a DevOps methodology. In this case the developers handle the deployment of an application that would normally be handled by the operations team members. In this case the operations team provides the infrastructure and processes and the development team gains control of the application on the production system. The system provides to the developer the ability to define a container for any target environment (test or production). It allows the development community to manage the Docker container creation, deployment and lifecycle management. It empowers the developer to completely manage the execution environment for their application.

## 5.1 Current Limitations and Challenges

The system that was developed works as designed for deployment to targeted Docker container servers. A user is able to deploy when required to a system but the developer must target each system directly. There currently is not concept of clustering across servers in the current system design. That will be addressed in the future and related work sections of this report.

One of the main challenges with the system was centered on the networking stack within Docker containers. In its initial design, traffic was to be routed to each JBoss

instance from a separate server that was running Apache HTTP. There was an initial problem with this design. The problem manifested as a result of the dynamic nature of using ModCluster for application registration within Apache HTTP server. The infrastructure within JBoss Application Server does not allow it to use a specific network interface when sending the connection information to the Apache HTTP server. Thus resulting in the Apache HTTP web server registering an IP that is used internally by the Docker platform. This networking configuration did not allow traffic to be routed to the Docker container from an external server. One work around for the networking issue was for the Docker container to use the host systems network stack. This allowed the Docker container to obtain the routable IP from the Docker container server. This workaround negated the requirement for a server-agnostic Docker container that could be used regardless of the host server and was not developed.

## 5.2 Future Work

The project only touched what was a wide range of aspects to working with Docker containers. That being said it allows for several avenues of future work for the system.

### 5.2.1 Networking

As stated in the previous section, the networking configuration used when working with Docker containers has challenges. Future work will be done to solve some of the problems encountered. Specifically the default networking provided by Docker is

insufficient. This is a “best use” setup for typical installations. However, the system is fully configurable. A custom network configuration will be implemented to allow routing to a container from an external source server.

### 5.2.2 Eclipse Plugin

The system as tested used Ant scripts for deployment and some management actions. Future work would involve creating a plugin for the Eclipse IDE so that developers will not have to manage scripts within their source code.

### 5.2.3 Port Management

The aspect of managing ports is not addressed in this report. It is implied that the developers manage ports definitions. Ideally port management should be hidden from the developers. It would be beneficial to have some infrastructure that would manage Docker container ports that are mapped to the host system’s networking stack. A simple system that defines a static range of ports should be used. The system could check for the availability of ports and assigns them to the Docker container at startup.

### 5.2.4 Clustering capabilities

Clustering applications means that multiple instances of the same Docker container can be instantiated. Work will be done to mature the current system so that deployed containers can be clustered if need be. That work will be done using third party frameworks addressed in the following section.

### 5.3 Related Work

One aspect to the report that was not really addressed was the issue of clustering the application. I will point out a few technologies that would benefit the system and address application clustering.

There are currently two clustering platforms being developed for Docker containers. Both are similar and will be presented together. The first is Kubernetes [15], an open-sourced platform developed by Google. Kubernetes is based on their internal, container management system. The second is Docker Swarm [16], which is a new tool being offered by Docker and is currently in beta. Both would allow for clustering across Docker container servers and provide cluster management features. Both Kubernetes and Docker Swarm provide features for clustering, orchestration and management of Docker containers. Kubernetes is more mature and has additional features not yet offered by Swarm, such as container scheduling and a more advanced set of networking tools.

## Bibliography


[1] Waller, Jan, Nils C. Ehmke, and Wilhelm Hasselbring. "Including Performance Benchmarks into Continuous Integration to Enable DevOps." *ACM SIGSOFT Software Engineering Notes* 40.2 (2015): 1-4.

[2] Mueller, Earnest, James Wickett, Karthik Gaekwad, and Peco Karayanev. "What Is DevOps?" *The Agile Admin*. 2 Aug. 2010. Web. 22 Apr. 2015. <http://theagileadmin.com/what-is-devops/>.

[3] "Agile Software Development." *Wikipedia*. Wikimedia Foundation. Web. 30 Apr. 2015. <http://en.wikipedia.org/wiki/Agile_software_development>.

[4] "Cloud Computing." *Wikipedia*. Wikimedia Foundation. Web. 22 Apr. 2015. <http://en.wikipedia.org/wiki/Cloud_computing>.

[5] Morabito, Roberto, Jimmy Kjällman, and Miika Komu. "Hypervisors vs. Lightweight Virtualization: a Performance Comparison."

[6] "Build, Ship and Run Any App, Anywhere." *Docker*. Web. 22 Apr. 2015. <http://www.docker.com>.

[7] "LXC." *Wikipedia*. Wikimedia Foundation. Web. 22 Apr. 2015. <http://en.wikipedia.org/wiki/LXC>.

[8] "What Is Docker?" *What Is Docker? An Open Platform for Distributed Apps*. Web. 22 Apr. 2015. <https://www.docker.com/whatisdocker/>.

[9] "Aufs." *Wikipedia*. Wikimedia Foundation. Web. 22 Apr. 2015. <http://en.wikipedia.org/wiki/Aufs>.

[10] "CentOS 7 Updates." *CentOS Project*. Web. 22 Apr. 2015. <https://www.centos.org>.

[11] "JBoss EAP - Overview." *JBoss Developer*. Web. 22 Apr. 2015. <http://www.jboss.org/products/eap/overview/>.

[12] "Essentials." *Welcome!* Web. 22 Apr. 2015. <http://httpd.apache.org>.

[13] "Mod_cluster - JBoss Community." *Mod_cluster - JBoss Community*. Web. 22 Apr. 2015. <http://mod-cluster.jboss.org>.

[14] "Agility, Scalability, Performance. Pick Three." *MongoDB*. Web. 22 Apr. 2015. <https://www.mongodb.org>.

[15] "Kubernetes by Google." *Kubernetes by Google*. Web. 22 Apr. 2015. <http://kubernetes.io>.

[16] "Docker Swarm." - *Docker Documentation*. Web. 22 Apr. 2015. <https://docs.docker.com/swarm/>.