Mule ESB in Docker

By Ivan Krizsan | January 18, 2015 4 Comments

In this article I will attempt to run the Mule ESB community edition in Docker in order to see whether it is feasible without any greater inconvenience. My goal is to be able to use Docker both when testing as well as in a production environment in order to gain better control over the environment and to separate different types of environments.

I imagine that most of the Docker-related information can be applied to other applications – I have used Mule since it is what I usually work with.

The conclusion I have made after having completed my experiments is that it is possible to run Mule ESB in Docker without any inconvenience. In addition, Docker will indeed allow me to have better control over the different environments and also allow me to separate them as I find appropriate.

Finally, I just want to mention that I have used Docker in an Ubuntu environment. I have not attempted any of the exercises in Docker running on Windows or Mac OS X.

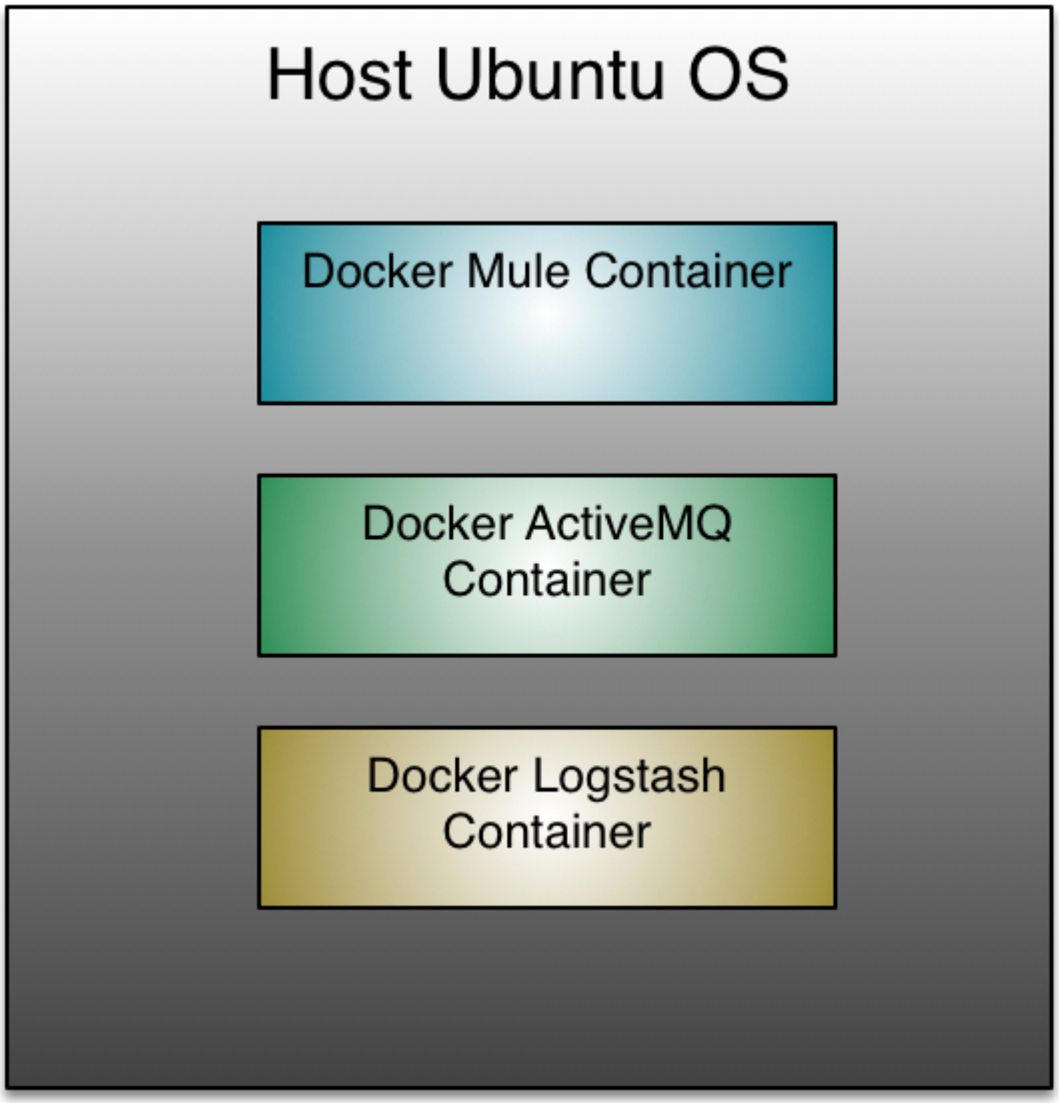
Docker Briefly

In short, Docker allows for creating of images that serve as blueprints for containers. A Docker container is an instance of a Docker image in the same way a Java object is an instance of a Java class.

```
FROM codingtony/java

MAINTAINER tony(dot)bussieres(at)ticksmith(dot)com

RUN wget https://repository.mulesoft.org/nexus/content/repositories/releases/org/mul
RUN cd /opt && tar xvzf ~/mule-standalone-3.5.0.tar.gz
RUN echo "4a94356f7401ac8be30a992a414ca9b9 /mule-standalone-3.5.0.tar.gz" | md5sum -
RUN rm ~/mule-standalone-3.5.0.tar.gz
RUN ln -s /opt/mule-standalone-3.5.0 /opt/mule

CMD [ "/opt/mule/bin/mule" ]
```

The resource isolation features of Linux are used to create Docker containers, which are more lightweight than virtual machines and are separated from the environment in which Docker runs, the host.

Using Docker an image can be created that, every time it is started has a known state. In order to remove any doubts about whether the environment has been altered in any way, the container can be stopped and a new container started. I can even run multiple Docker containers on one and the same computer to simulate a multi-server production environment. Applications can also be run in their own Docker containers, as shown in this figure.



Three Docker containers, each containing a specific application, running in one host.

The main entry point to the Docker documentation can be found here.

Motivation

Some of the motivations I have for using Docker in both testing and production environments are:

- The environment in which I test my application should be as similar as the final deployment environment as possible, if not identical.
- Making the deployment environment easy to scale up and down.
  If it is easy to start a new processing node when need arise and stop it if it is no longer used, I will be able to adapt to changes rather quickly and thus reduce errors caused by, for instance, load peaks.
- Maintain an increased number of nodes to which applications can be deployed.
  Instead of running one instance of some kind of application server, Mule ESB in my case, on a computer, I want multiple instances that are partitioned, for instance, according to importance. High-priority applications run on one separate instance, which have higher priority both as far as resources (CPU, memory, disk etc) are concerned but also as far as support is concerned. Applications which are less critical run on another instance.
- Enable quick replacement of instances in the deployment environment.
  Reasons for having to replace instances may be hardware failure etc.
- Better control over the contents of the different environments.
  The concept of an environment that, at any time, may be disposed (and restarted) discourages hacks in the environment, which are usually poorly documented and sometimes difficult to trace. Using Docker, I need to change the appropriate Docker image if I want to make changes to some application environment. The Docker image file, commonly known as Dockerfile, can be checked into any ordinary revision control system, such as Git, Subversion etc, making changes reversible and traceable.
- Automate the creation of a testing environment.
  An example could be a nightly job that runs on my build server which creates a test environment, deploys one or more applications to it and then performs tests, such as load-testing.

Prerequisites

To get the best possible experience when running Docker, I run it under Ubuntu.
According to the current documentation, Docker is supported under the following versions of Ubuntu:

- 12.04 LTS (64-bit)
- 13.04 (64-bit)

Against my usual conservative self, I chose Ubuntu 14.10, which at the time of writing this article is the latest version. While I haven't run into any issues, I cannot promise anything regarding compatibility with Docker as far as this version of Ubuntu is concerned.

Installing Docker

Before we install anything, those who have the Docker version from the Ubuntu repository should remove this version before installing a newer version of Docker, since the Ubuntu repository does not contain the most recent version and the package does not have the same name as the Docker package we will install:

```
sudo apt-get remove docker.io
```

The simplest way to install Docker is to use an installation script made available at the Docker website:

```
curl -sSL https://get.docker.com/ubuntu/ | sudo sh
```

If you are not running Ubuntu or if you do not want to use the above way of installing Docker, please refer to this page containing instructions on how to install Docker on various platforms.

To verify the Docker installation, open a terminal window and enter:

```
sudo docker version
```

Output similar to the following should appear:

```
Client version: 1.4.1
Client API version: 1.16
Go version (client): go1.3.3
Git commit (client): 5bc2ff8
OS/Arch (client): linux/amd64
Server version: 1.4.1
Server API version: 1.16
Go version (server): go1.3.3
Git commit (server): 5bc2ff8
```

We are now ready to start a Mule instance in Docker.

Running Mule in Docker

One of the advantages with Docker is that there is a large repository of Docker images that are ready to be used, and even extended if one so wishes. This Docker image is the one that I will use in this article. It is well documented, there is a source repository and it contains a recent version of the Mule ESB Community Edition. Some additional details on the Docker image:

- Oracle JavaSE 1.7.0_65.
  This version will change as the PPA containing the package is updated.
- Mule ESB CE 3.5.0

Note that the image may change at any time and the specifications above may have changed.

If you intend to use Docker in your organization, I would suspect that the best alternative is to create your own Docker images that are totally under your control. The Docker image repository is an excellent source of inspiration and aid even in this case.

Starting a Docker Container

To start a Docker container using this image, open a terminal window and write:

```
sudo docker run codingtony/mule
```

The first time an image is used it needs to be downloaded and created. This usually takes quite some time, so I suggest a short break here – perhaps for a cup of coffee or tea.
If you just want to download an image without starting it, exchange the Docker command “run” with “pull”.

Once the container is started, you will see some output to the console. If you are familiar with Mule, you will recognize the log output:

```
MULE_HOME is set to /opt/mule-standalone-3.5.0
Running in console (foreground) mode by default, use Ctrl-C to exit...
MULE_HOME is set to /opt/mule-standalone-3.5.0
Running Mule...
--> Wrapper Started as Console
Launching a JVM...
Starting the Mule Container...
Wrapper (Version 3.2.3) http://wrapper.tanukisoftware.org
  Copyright 1999-2006 Tanuki Software, Inc.  All Rights Reserved.

INFO  2015-01-05 04:41:42,302 [WrapperListener_start_runner] org.mule.module.launche
**********************************************************************
* Mule ESB and Integration Platform                                  *
* Version: 3.5.0 Build: ff1df1f3                                     *
* MuleSoft, Inc.                                                     *
* For more information go to http://www.mulesoft.org                 *
*                                                                    *
* Server started: 1/5/15 4:41 AM                                     *
* JDK: 1.7.0_65 (mixed mode)                                         *
* OS: Linux (3.16.0-28-generic, amd64)                               *
* Host: f95698cfb796 (172.17.0.2)                                    *
**********************************************************************
```

- In the text-box containing information about the Mule ESB and Integration Platform, there is a row which starts with "Host:".
  The hexadecimal digit that follows is the Docker container id and the IP-address is the external IP-address of the Docker container in which Mule is running.

Before we do anything with the Mule instance running in Docker, let's take a look at Docker containers.

Docker Containers

We can verify that there is a Docker container running by opening another terminal window, or a tab in the first terminal window, and running the command:

```
sudo docker ps
```

As a result, you will see output similar to the following (I have edited the output in order for the columns to be aligned with the column titles):

```
CONTAINER ID  IMAGE                    COMMAND                   CREATED    STATUS    PORTS
f95698cfb796  codingtony/mule:latest "/opt/mule/bin/mule" 7 min ago  Up 7 min
```

From this output we can see that:

- The ID of the container is f95698cfb796.
  This ID can be used when performing operations on the container, such as stopping it, restarting it etc.
- The name of the image used to created the container.
- The command that is currently executing.
  If we look at the Dockerfile for the image, we can see that the last line in this file is:
  CMD ["/opt/mule/bin/mule"]
  This is the command that is executed whenever an instance of the Docker image is launched and it matches what we see in the COMMAND column for the Docker container.
- The CREATED column shows how much time has passed since the container was created.
- The STATUS column shows the current status of the image.
  When you have used Docker for a while, you can view all the containers using:
  sudo docker ps -a
  This will show you containers that are not running, in addition to the running ones. Containers that are not running can be restarted.
- The PORTS column shows any port mappings for the container.
  More about port mappings later.
- Finally, the NAMES column contain a more human-friendly container name.

Docker containers will consume disk-space and if you want to determine how much disk-space each of the containers on your computer use, issue the following command:

```
sudo docker ps -a -s
```

An additional column, SIZE, will be shown and in this column I see that my Mule container consumes 41,76kB. Note that this is in addition to the disk-space consumed by the Docker image. This number will grow if you use the container under a longer period of time, as the container retains any files written to disk.
To completely remove a stopped Docker container, find the id or name of the container and use the command:

```
sudo docker rm [container id or name here]
```

Before going further, let's stop the running container and remove it:

```
sudo docker stop [container id or name here]
sudo docker rm [container id or name here]
```

Files and Docker Containers

So far we have managed to start a Mule instance running inside a Docker container, but there were no Mule applications deployed to it and the logs that were generated were only visible in the terminal window. I want to be able to deploy my applications to the Mule instance and examine the logs in a convenient way.
In this section I will show how to:

- Share one or more directories in the host file-system with a Docker container.
- Access the files in a Docker container from the host.

As the first step in looking at sharing directories between the host operating system and a Docker container, we are going to look at Mule logs. As part of this exercise we also set up the directories in the host operating system that are going to be shared with the Docker container.

- In your home directory, create a directory named "mule-root".
- In the "mule-root" directory, create three directories named "apps", "conf" and "logs".
- Download the Mule CE 3.5.0 standalone distribution from this link.
- From the Mule CE 3.5.0 distribution, copy the files in the "apps" directory to the "mule-root/apps" directory you just created.
- From the Mule CE 3.5.0 distribution, copy the files in the "conf" directory to the "mule-root/conf"

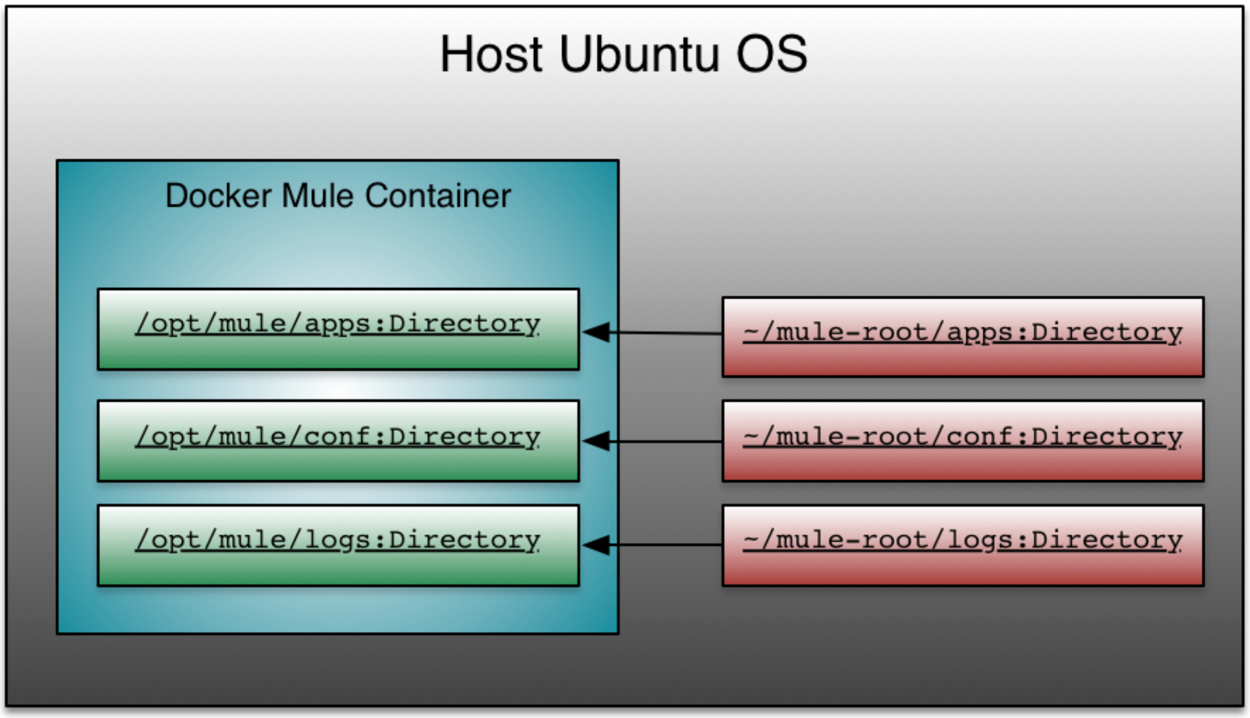
```
~/mule-root/
├── apps
|    └── default
|        └── mule-config.xml
├── conf
|    ├── log4j.properties
|    ├── tls-default.conf
|    ├── tls-fips140-2.conf
|    ├── wrapper-additional.conf
|    └── wrapper.conf
└── logs
```

- Edit the log4j.properties file in the “mule-root/conf” directory and set the log-level on the last line in the file to “DEBUG”. This modification has nothing to do with sharing directories, but is in order for us to be able to see some more output from Mule when we run it later. The last two lines should now look like this:

```
# Mule classes
log4j.logger.org.mule=DEBUG
```

Binding Volumes

We are now ready to launch a new Docker container and when we do, we will tell Docker to map three directories in the Docker container to three directories in the host operating system.



This website uses cookies to improve your experience. We'll assume you're ok with this, but you can opt-out if you wish. Accept Read More

- Launch the Docker container with the command below.
  The -v option tells Docker that we want to make the contents of a directory in the host available at a certain path in the Docker container file-system.
  The -d option runs the container in the background and the terminal prompt will be available as soon as the id of the newly launched Docker container has been printed.

```
sudo docker run -d -v ~/mule-root/apps:/opt/mule/apps -v ~/mule-root/conf:/opt/mule/c
```

- Examine the "mule-root" directory and its subdirectories in the host, which should now look like below.
  The files on the highlighted rows have been created by Mule.

```
mule-root/
├── apps
|   ├── default
|   |   └── mule-config.xml
|   └── default-anchor.txt
├── conf
|   ├── log4j.properties
|   ├── tls-default.conf
|   ├── tls-fips140-2.conf
|   ├── wrapper-additional.conf
|   └── wrapper.conf
└── logs
    ├── mule-app-default.log
    ├── mule-domain-default.log
    └── mule.log
```

- Examine the "mule.log" file using the command "tail -f ~/mule-root/logs/mule.log".
  There should be periodic output written to the log file similar to the following:

```
DEBUG 2015-01-05 12:05:37,216 [Mule.app.deployer.monitor.1.thread.1] org.mule.module.
DEBUG 2015-01-05 12:05:37,216 [Mule.app.deployer.monitor.1.thread.1] org.mule.module.
 default-anchor.txt
DEBUG 2015-01-05 12:05:37,216 [Mule.app.deployer.monitor.1.thread.1] org.mule.module.
```

- Stop and remove the container:

```
sudo docker stop [container id or name here]
sudo docker rm [container id or name here]
```

Direct Access to Docker Container Files

When running Docker under the Ubuntu OS it is also possible to access the file-system of a Docker container from the host file-system. It may be possible to do this under other operating systems too, but I haven't bound any volumes.

Note!
If given the choice to use either volume binding, as seen above, or direct access to container files as we will look at in this section for something more than a temporary file access, I would chose to use volume binding.
Direct access to Docker container files relies on implementation details that I suspect may change in future versions of Docker if the developers find it suitable.

With all that said, lets get the action started:

- Start a new Docker container:

```
sudo docker run -d codingtony/mule
```

- Find the id of the newly launched Docker container:

```
sudo docker ps
```

- Examine low-level information about the newly launched Docker container:

```
sudo docker inspect [container id or name here]
```

Output similar to this will be printed to the console (portions removed to conserve space):

```
[{
    "AppArmorProfile": "",
    "Args": [],
    "Config": {
        ...
    },
    "Created": "2015-01-12T07:58:47.913905369Z",
    "Driver": "aufs",
    "ExecDriver": "native-0.2",
    "HostConfig": {
        ...
    },
    "HostnamePath": "/var/lib/docker/containers/68b40def7ad6a7f819bd654d5627ad1c3a0f
    "HostsPath": "/var/lib/docker/containers/68b40def7ad6a7f819bd654d5627ad1c3a0f40c
    "Id": "68b40def7ad6a7f819bd654d5627ad1c3a0f40c84e0fb0f875760f1bd6790eef",
    "Image": "bcd0f37d48d4501ad64bae941d95446b157a6f15e31251e26918dbac542d731f",
    "MountLabel": "",
    "Name": "/thirsty_darwin",
    "NetworkSettings": {
        ...
    },
    "Path": "/opt/mule/bin/mule",
        ...
    },
    "Volumes": {},
    "VolumesRW": {}
}]
```

- Locate the “Driver” node (highlighted in the above output) and ensure that its value is “aufs”.
  If it is not, you may need to modify the directory paths below replacing “aufs” with the value of this node. Personally I have only seen the “aufs” value at this node so anything else is uncharted territory to me.
- Copy the long hexadecimal value that can be found at the “Id” node (also highlighted in the above output).
  This is the long id of the Docker container.
- In a terminal window, issue the following command, inserting the long id of your container where noted:

```
sudo ls -al /var/lib/docker/aufs/mnt/[long container id here]
```

You are now looking at the root of the volume used by the Docker container you just launched.

- In the same terminal window, issue the following command:

```
sudo ls -al /var/lib/docker/aufs/mnt/[long container id here]/opt
```

The output from this command should look like this:

```
total 12
drwxr-xr-x  4 root root 4096 jan 12 15:58 .
drwxr-xr-x 75 root root 4096 jan 12 15:58 ..
lrwxrwxrwx  1 root root   26 aug 10 04:19 mule -> /opt/mule-standalone-3.5.0
drwxr-xr-x 17  409  409 4096 jan 12 15:58 mule-standalone-3.5.0
```

- Examine this line in the Dockerfile :
  RUN ln -s /opt/mule-standalone-3.5.0 /opt/mule
  We see that a symbolic link is created and that the directory name and the name of the symbolic link matches the output we saw earlier. This matches the directory output in the previous step.
- To examine the Mule log file that we looked at when binding volumes earlier, use the following command:

```
sudo cat /var/lib/docker/aufs/mnt/[long container id here]/opt/mule-standalone-3.5.0/
```

- Next we create a new file in the Docker container using *vi*:

```
sudo vi /var/lib/docker/aufs/mnt/[long container id here]/opt/mule-standalone-3.5.0/t
```

When you are finished entering the text, press the Escape key and write the file to disk by typing the characters ":wq" without quotes. This writes the new contents of the file to disk and quits the editor.

- Leave the Docker container running after you are finished.
  In the next section, we are going to look at the file we just created from inside the Docker container.

We have seen that we can examine the file system of a Docker container without binding volumes. It is also possible to copy or move files from the host file-system to the container's file system using the regular commands. Root privileges are required both when examining and writing to the Docker container's file system.

Entering a Docker Container

In order to verify that the file we just created in the host was indeed written to the Docker container, we are going to start a bash shell in the running Docker container and examine the location where the new file is expected to be located and the contents of the file.
In the process we will see how we can execute commands in a Docker container from the host.

- Issue the command below in a terminal window.
  The *exec* Docker command is used to run a command, *bash* in this case, in a running Docker container. The -i flags tell Docker to keep the input stream open while the command is being executed. In this example, it allows us to enter commands into the bash shell running inside the Docker container. The -t flag cause Docker to allocate a text terminal to which the output from the command execution is printed.

```
sudo docker exec -i -t [container id or name here] bash
```

Note the prompt, which should change to [user]@[Docker container id].
In my case it looks like this:

```
root@3ea374a280da:/#
```

- Go to the Mule installation directory using this command:

```
cd /opt/mule-standalone-3.5.0/
```

- Examine the contents of the directory:

Among the other files, you should see the "test.txt" file:

```
-rw-r--r--  1 root root    53 Jan 14 03:19 test.txt
```

- Examine the contents of the "text.txt" file.
  The contents of the file should match what you entered earlier.

```
cat text.txt
```

- Exit to the host OS:

```
exit
```

- Stop and remove the container:

```
sudo docker stop [container id or name here]
sudo docker rm [container id or name here]
```

We have seen that we can execute commands in a running Docker container. In this particular example, we used it to execute the bash shell and examine a file.
I draw the conclusion that I should be able to set up a Docker image that contains a very controlled environment for some type of test and then create a container from that image and start the test from the host.

Deploying a Mule Application

In this section we will look at deploying a Mule application to an instance of the Mule ESB running in a Docker container. We will use volume binding, that we looked at in the section on files and Docker containers, to share directories in the host with the Docker container in order to make it easy to deploy applications, modify running applications, examine logs etc.

Preparations

Before deploying the application, we need to make some preparations:

First of all, we restore the original log-level that we changed earlier. In this example, there will be log output when the applications we will deploy is run and we can limit the log generated by Mule.

- Edit the log4j.properties file in the "mule-root/conf" directory in the host and set the log-level on the last line in the file back to "INFO" and add one line, as in the listing below. The last three lines

```
# Mule classes
log4j.logger.org.mule=INFO
log4j.logger.org.mule.tck.functional=DEBUG
```

Next, we create the Mule application which we will deploy to the Mule ESB running in Docker:

- In some directory, create a file named “mule-deploy.properties” with the following contents:

```
redeployment.enabled=true
encoding=UTF-8
domain=default
config.resources=HelloWorld.xml
```

- In the same directory create a file named “HelloWorld.xml”.
  This file contains the Mule configuration for our example application:

```
<?xml version="1.0" encoding="UTF-8"?>
<mule xmlns="http://www.mulesoft.org/schema/mule/core"
    xmlns:core="http://www.mulesoft.org/schema/mule/core"
    xmlns:http="http://www.mulesoft.org/schema/mule/http"
    xmlns:spring="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:test="http://www.mulesoft.org/schema/mule/test"
    xsi:schemaLocation="
http://www.mulesoft.org/schema/mule/http http://www.mulesoft.org/schema/mule/http/cu
http://www.springframework.org/schema/beans http://www.springframework.org/schema/be
http://www.mulesoft.org/schema/mule/core http://www.mulesoft.org/schema/mule/core/cu
http://www.mulesoft.org/schema/mule/test http://www.mulesoft.org/schema/mule/test/cu

    <flow name="HelloWorldFlow">
        <http:inbound-endpoint
            exchange-pattern="request-response"
            host="0.0.0.0"
            port="8181" />

        <test:component logMessageDetails="true"/>
        <set-payload
            value="#[string:Hello World! It is now: #[server.dateTime]]"/>
    </flow>
</mule>
```

- Create a zip-archive named “mule-hello.zip” containing the two files created above:

```
zip mule-hello.zip mule-deploy.properties HelloWorld.xml
```

Deploy the Mule Application

Before you start the Docker container in which the Mule EBS will run, make sure that you have created and prepared the directories in the host as described in the section Files and Docker Containers above.

```
sudo docker run -d -v ~/mule-root/apps:/opt/mule/apps -v ~/mule-root/conf:/opt/mule/c
```

As before, the -v option tells Docker to bind three directories in the host to three locations in the Docker container's file system.

- Find the IP-address of the Docker container:

```
sudo docker inspect [container id or name here] | grep IPAddress
```

In my case, I see the following line which reveals the IP-address of the Docker container:

```
"IPAddress": "172.0.17.2",
```

- Open a terminal window or tab and examine the Mule log.
  Leave this window or tab open during the exercise, in order to be able to verify the output from Mule.

```
tail -f ~/mule-root/logs/mule.log
```

- Copy the zip-archive "mule-hello.zip" created earlier to the host directory ~/mule-root/apps/.
  Verify that the application has been deployed without errors in the Mule log:

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
+ Started app 'mule-hello'                                 +
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

- Leave the Docker container running after you are finished.
  In the next section we will look at how to access endpoints exposed by applications running in Docker containers.

By binding directories in the host thus making them available in the Docker container, it becomes very simple to deploy Mule applications to an instance of Mule ESB running in a Docker container.
I am considering this setup for a production environment as well, since it will enable me to perform back-ups of the directories containing Mule applications and configuration without having to access the Docker container's file system. It is also in accord with the idea that a Docker container should be able to be quickly and easily restarted, which I feel it would not be if I had to deploy a number of Mule applications to it in order to recreate its previous state.

Accessing Endpoints

endpoints exposed by applications running in a Docker container.

This section assumes that the Mule application we deployed to Mule in the previous section is still running.

- In the host, open a web-browser and issue a request to the Docker container's IP-address at port 8181. In my case, the URL is http://172.17.0.2:8181
  Alternatively use the *curl* command in a terminal window. In my case I would write: *curl 172.17.0.2:8181*
  The result should be a greeting in the following format:

```
Hello World! It is now: 2015-01-14T07:39:03.942Z
```

In addition, you should be able to see that a message was received in the Mule log.

- Now try the URL http://localhost:8181
  You will get a message saying that the connection was refused, provided that you do not already have a service listening at that port.
- If you have another computer available that is connected to the same network as the host computer running Ubuntu, do the following:
  – Find the IP-address of the Ubuntu host computer using the *ifconfig* command.
  – In a web-browser on the other computer, try accessing port 8181 at the IP-address of the Ubuntu host computer.
  Again you will get a message saying that the connection was refused.
- Stop and remove the container:

```
sudo docker stop [container id or name here]
sudo docker rm [container id or name here]
```

Without any particular measures taken, we see that we can access a service exposed in a Docker container from the Docker host but we did not succeed in accessing the service from another computer.
To make a service exposed in a Docker container reachable from outside of the host, we need to tell Docker to publish a port from the Docker container to a port in the host using the -p flag:

- Launch a new Docker container using the following command:

```
sudo docker run -d -p 8181:8181 -v ~/mule-root/apps:/opt/mule/apps -v ~/mule-root/con
```

The added flag -p 8181:8181 makes the service exposed at port 8181 in the Docker container available at port 8181 in the host.

should be a greeting of the form we have seen earlier.

- Try accessing port 8181 at the IP-address of the Ubuntu host computer from another computer.This should also result in a greeting message.
- Stop and remove the container:

```
sudo docker stop [container id or name here]
sudo docker rm [container id or name here]
```

Using the -p flag, we have seen that we can expose a service in a Docker container so that it becomes accessible from outside of the host computer. However, we also see that this information need to be supplied at the time of launching the Docker container.
The conclusions that I draw from this is that:

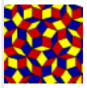
1. I can test and develop against a Mule ESB instance running in a Docker container without having to publish any ports, provided that my development computer is the Docker host computer.
2. In a production environment or any other environment that need to expose services running in a Docker container to "the outside world" and where services will be added over time, I would consider deploying an Apache HTTP Server or NGINX on the Docker host computer and use it to proxy the services that are to be exposed.
   This way I can avoid re-launching the Docker container each time a new service is added and I can even (temporarily) redirect the proxy to some other computer if I need to perform some maintenance.

Is There More?

Of course! This article should only be considered an introduction and I am just a beginner with Docker. I hope I will have the time and inspiration to write more about Docker as I learn more.

Category: Mule Operations Tags: docker, mule

4 thoughts on "Mule ESB in Docker"


granthbr
March 19, 2015

Great blog post! I've been using a similar setup but inside of Mesos. Using Mesos you can generate your

Mesos/Karaf/Camel (as well as with Mule). Would be great to see a comparison of those models.
One suggestion, you might want to make this available on github and allow collaboration or forking.

Ivan Krizsan Post author
March 19, 2015

Thanks a lot for the input!
I have been thinking about where to go next with Docker and these hints are really appreciated!

Nicolasc
April 9, 2016

Hello.

Your post is very use full and clear. I try to access the mule container from another container (to call Rest api) but port 8181 seams to be not exposed. The URL is OK when I use a ping command but KO with a wget mule:8181. The port 8181 is bind to 8181 in my docker-compose file.

Thanks.

Ivan Krizsan Post author
April 9, 2016

Hello!
When you want to access one container(A) from another container(B) you need to use the –link option when starting container B. As soon as you have more than one single container, I really recommend using Docker Compose. For an example on how to use Docker Compose and linking between containers, see the docker-compose.yml file here: https://github.com/krizsan/alerting-with-elk-and-elastalert.
Notice how I set an environment variable containing the URL of the Elasticsearch server (running in a separate container) for the Kibana container.
Happy coding!