Deploying Spring Boot-based microservices with Docker – part 3

Posted on November 16, 2014

In part 1 and part 2 of this series, we looked at the benefits of the microservice architecture and showed how Spring Boot simplifies the development of microservices. We also built a couple of simple Spring Boot-based services for user registration: a web application and a RESTful backend service (github repo). In this article, we will look at how to deploy Spring Boot-based services with an exciting new technology called Docker.

Microservice deployment options

One of the neat things about Spring Boot is that it builds a self-contained, executable JAR file. We don't need to install and correctly configure an application server. All that's required is the appropriate version of Java. This means that you just need to tell whoever is responsible for deploying the application in production (or a QA environment) the JAR file along with the following information:

1. What version of Java to install.
2. What external configuration parameters to specify. For example, the coordinates of infrastructure services, such as MongoDB and RabbitMQ, and of other application services.
3. The command for running the application.

It's straightforward to deploy an individual service. But deploying all of an application's services is likely to be much more complicated. Different services might use different Java versions. For example, the Restful web service developed in part 1 used Java 7 where as the web application build in part 2 uses Java 8. Some services might be Play or Tomcat-based web applications. Services might not even be written in Java and instead use, for example, Ruby or NodeJS. The details of how each service is configured and run depend very much on the framework and language. Services might even have conflicting requirements. As a result, whoever is responsible to deploying an application in test and production environments has to deal with a lot of complexity.

Overview of Docker

The journey of services from development to production mirrors how cargo was shipped prior to the invention of intermodal shipping containers. At the origin, cargo was manually loaded one piece at a time onto the truck or train that carried it to the port. At the port of origin the cargo was unloaded onto the dock. Longshoremen then carefully loaded the tens of thousands of individual pieces of cargo one at time – boxes of bananas, ingots, coffee, etc. – onto the ship in a way that optimized space while simultaneously ensuring that the ship was stable. All of these steps then happened in reverse at the destination. Not only was shipping a time consuming and labor intensive process but repeated handling often broke fragile goods.

Containerization dramatically changed the ship industry (see the excellent book The Box for more details). All non-bulk cargo is now packed into standard shipping containers, which can be carried by truck, trains and ships. Crane operators now rapidly load and unload containers regardless of their contents. The contents of the container are never touched in transit. In other words, the shipping container encapsulates it's contents. It has become the standardized API of cargo.

One way of containerizing a service is to package it as a VM machine image. For example, your continuous integration server can run a VM building tool such as Packer to create a VM image that can be deployed where needed. The virtual machine's standardized API becomes the service's management API. Starting and stopping the VM starts and stops the service regardless of the technology used to implement it. The contents of the VM are never touched. The service-as-a-VM approach is popular way of packaging and deploying services. For example, the Netflix video streaming service consists of many services each packaged as an AMI and deployed on Amazon EC2.

There are, however, various downsides to the service-as-a-VM approach. It assumes that your application runs in a virtualized environment, which is not always the case. Also, virtual machines are a heavyweight technology. It's not practical to deploy more than a couple of VMs on a developer laptop. Building and booting an VM machine is usually I/O intensive and consequently slow. Also, some IaaS offerings such AWS EC2, don't let you specify arbitrary VM sizes and so it's likely that a given service doesn't fully utilize its VM. Furthermore, the cost of deploying many small services as individual VMs on a public IaaS such as Amazon EC2 can rapidly add up.

Introduction to Docker

Docker is a new way to containerize applications that is becomingly increasingly popular. It allows you to package a microservice in a standardized portable format that's independent of the technology used to implement the service. At runtime it provides a high degree of isolation between different services. However, unlike virtual machines, Docker containers are extremely lightweight and as a result can be built and started extremely quickly. A container can typically be built in just a few seconds and starting a container simply consists of starting the service's process(es).

Docker runs on a variety of platforms. It runs natively on Linux. You can also run Docker on Windows and Mac OSX using Boot2Docker, which runs the Docker daemon in a VirtualBox VM. Some clouds also have added extra support for Docker. For example, not only can you run Docker inside your EC2 instances but you can also use Elastic Beanstalk to run Docker containers. Amazon also recently announced the Amazon EC2 Container Service, which is a hosted Docker container management service. Google Cloud also has support for Docker.

The two main Docker concepts are image, which is a portable application packaging format, and container, which is a running image and consists of one or more sandboxed processes. Let's first look at how images work.

About Docker images

A Docker image is read-only file system image of an operating system and an application. It's analogous to an AWS EC2 AMI. An image is self-contained and will run on any Docker installation. You can create an image from scratch but normally an image is created by starting a container from existing base image, installing applications by executing the same kinds of commands you would use when configuring a regular machine, such as apt-get install –y and then saving the container as a new image. For example, to create an image containing a Spring Boot based application, you could start from a vanilla Ubuntu image, install the JDK and

then install the executable JAR.

In many ways, building a Docker image is similar to building an AMI. However, while an AMI is a blob of bits, a Docker image has a layered structure that dramatically reduces the amount of time needed to build and deploy a Docker image. An image consists of a sequence of layers. When building an image, each command that changes the file system (e.g. *apt-get install*) create a new layer that references it's parent layer.

This layered structure has two important benefits. First it enables of sharing of layers between images, which means that Docker does not need to move an entire image over the network. Only those layers that don't exist on the destination machine need to be copied, which usually results in a dramatic speedup. Another important benefit of the layered structure is that Docker aggressively caches layers when building an image. When re-executing a command against an input layer Docker tries to skip executing the command and instead reuses the already built output layer. As a result, building an image is usually extremely fast.

About Docker containers

A Docker container is a running image consisting of one or more sandboxed processes. Docker isolates a container's processes using a variety of mechanisms including relatively mature OS-level virtualization mechanisms such as control groups and namespaces. Each process group has its own root file-system. Process groups can be assigned resource limits, e.g. CPU and memory limits. In the same way that a hypervisor divides up the hardware amongst virtual machines, this mechanism divides up the OS between process groups. Each Docker container is a process group.

Docker also isolates the networking portion of each container. When Docker is installed, it creates a virtual interface called docker0 on the host and sets up subnet. Each container is given it's own virtual interface called eth0 (within the container's namespace), which is assigned an available IP address from the Docker subnet. This means, for example, that a Spring Boot application running in a container listens on port 8080 of the virtual interface that's specific to its container. Later on we will look how you can enable a service to be accessed from outside its container by setting up a port mapping that associates a host port with a container port.

It's important to remember that even though an image contains an entire OS a Docker container often only consists of the application's processes. You often don't need to start any of the typical OS processes such as initd. For example, a Docker container that runs a Spring Boot application might only start Java. As a result, a Docker container has a minimal runtime overhead and its startup time is the startup time of your application.

Now that we have looked at basic Docker concepts let's look at using Docker to package Spring Boot applications.

Spring Boot and Docker

Let's now build a Docker image that runs the Spring Boot application. Because Spring Boot packages the application as a self-contained executable JAR, we just need to build an image containing that JAR file and Java. One option is to take a vanilla Ubuntu Docker image, install Java and install the JAR. Fortunately, we can skip the first step because it's already been done. One of the great features of the Docker ecosystem is https://hub.docker.com, which is a website where the community shares Docker images. There are a huge number of images available including dockerfile/java, which provides Java images for Oracle and OpenJDK versions 6, 7, and 8.

Once we have identified a suitable base image the next step is to build a new image that runs the Spring Boot application. You could build an image manually by launching the base image and entering shell commands in pretty much the same way that you would configure a regular OS. However, it's much better to automate image creation. To do that we need to create a Dockerfile, which is a text file containing series of commands that tell Docker how to build an image. Once we have written a Dockerfile, we can then repeatedly build an image by running *docker build*.

Here is the Dockerfile (see github repo) that builds an image that runs the application

```
FROM dockerfile/java:oracle-java7
MAINTAINER chris@chrisrichardson.net
EXPOSE 8080
CMD java -jar spring-boot-restful-service.jar
ADD build/spring-boot-restful-service.jar /data/spring-boot-restful-service.jar
```

As you can see, the Dockerfile is very simple. It consists of the following instructions:

- FROM – the FROM instruction specifies the starting image, which in this example is the Java 7 image mentioned above. The first time you build this image, Docker will download the Java 7 image from the central Docker registry
- MAINTAINER – this instruction simply specifies the author
- EXPOSE – this instruction tells Docker that this server process will listen on port 8080
- CMD – the CMD instruction specifies the command to run (by default) when the container is started, i.e. the Spring Boot application.
- ADD – this instruction copies the JAR file to the specified location in the image. Note that working directory for the dockerfile/java image is /data so that's why we are putting the JAR file there.

Here is the shell script that builds the image:

```
rm -fr build
mkdir build
cp ../build/libs/spring-boot-restful-service.jar build
docker build -t sb_rest_svc .
```

This script builds the image using the *docker build* command. The –t argument specifies the name give to the new image. The "." argument tells Docker to build the image using the current working directory as what is called the context of the build. The context defines the set of files that are uploaded to the Docker daemon and used to build the image. At the root of the context is the Dockerfile, which contains commands such as ADD that reference the other files in the context. We could specify the Gradle project root as the context and upload the entire project to the Docker daemon. But since the only files needed to build the image are the Dockerfile and the JAR it's much more efficient to copy the JAR file to a docker/build subdirectory.

Now that have packaged the application as a Docker image we need to run it. To do that we use the *docker run* command:

```
docker run -d -p 8080:8080 --name sb_rest_svc sb_rest_svc
```

The arguments to the *run* command are as follows:

- -d – tells Docker to run the service as a daemon
- -p – specifies the port mapping for the container. In this particular case, it specifies that the container's port 8080 should be mapped to port 8080 on the host. In other words, clients can access this service via http://host:8080
- –name – specifies the name of the newly created container
- sb_rest_svc – the name of the image to run

If we execute this command, the service starts up but the Spring application context initialization fails because it doesn't know how connect to MongoDB and RabbitMQ.

Fortunately, this problem is easy to fix because of how Spring Boot and Docker support environment variables. One of the nice features of Spring Boot is that it let's you specify configuration properties using OS environment variables. Specifically, for this service we need to supply values for SPRING_DATA_MONGODB_URI, which specifies the MongoDB database, and SPRING_RABBITMQ_HOST, which specifies the RabbitMQ host.

This mechanism works extremely well with Docker because you can use the *docker run* command's –e option to specify values for a container's environment variables. For example, let's suppose that RabbitMQ and Mongo are running on a machine with an IP address of 192.168.59.103. You can then run the container with the following command:

```
docker run -d -p 8080:8080 -e SPRING_DATA_MONGODB_URI=mongodb://192.168.59.103/userregistration
-e SPRING_RABBITMQ_HOST=192.168.59.103
--name sb_rest_svc
sb_rest_svc
```

This command starts the service, which connects to MongoDB and RabbitMQ. You can examine the output of the process using the docker log command:

```
docker logs sb_rest_svc
```

This command outputs the stdout/stderr of the service.

Summary

In this article, we saw how containerization is an excellent way to simplify deployment. Rather than giving operations your application and detailed deployment instructions, you simply package your application as a container. Docker is an extremely lightweight and efficient container technology. You can package your services as Docker containers and run them unchanged in any Docker environment: development, test and production. Later articles will look at various aspects of deploying Docker-based applications including a Jenkins-based deployment pipeline that builds and tests Docker images. We will also look at using Docker to simplify the setup of test environments.

Next steps

- Learn more about microservices at Microservices.io
- Look at the user registration service code on github

- Read up on the Spring Boot project
- Read up on the Docker

Share this:

This entry was posted in architecture, microservices, spring boot and tagged architecture, microservices, spring boot. Bookmark the permalink.

5 Responses to *Deploying Spring Boot-based microservices with Docker – part 3*

Pingback: *Need to install MongoDB, RabbitMQ, or MySQL? Use Docker to simplify dev and test | plain old objects*

VCS *says:*
September 21, 2015 at 8:34 am

Awesome !! Thank you for the detailed article. I'm building a spring boot application, which is not going to be deployed as jar, but war and depoyed to tomcat server. My application also access MySQL DB. In this scenario how can i build the docker image?

Reply

ceracm *says:*
September 22, 2015 at 1:41 pm

Glad you liked the article. I haven't deployed a traditional WAR on Tomcat using Docker.
You would need to use this as your base image: https://hub.docker.com/_/tomcat/
Your Dockerfile would copy the the WAR file into the /usr/local/tomcat/webapps dir
Something like:

```
FROM tomcat:8.0
EXPOSE 8080
COPY yourwebapp.war /usr/local/tomcat/webapps/yourwebapp.war
```

I hope helps.

Reply

VCS *says:*
October 2, 2015 at 11:50 am

Thank you for detailed article. I'm building a spring boot multi maven module application. I'm trying to create a docker image. Should I need to create docker file for each module ?or creating docker file just for web is sufficient?

The project structure is like this.
+ parent
+web
+module 1
+module 2

Reply

Pingback: *Spring Boot Services using Netflix OSS | Tales from a Trading Desk*

plain old objects