EBOOK

Docker From Code to Container

How Docker Enables DevOps, Continuous Integration and Continuous Delivery

sumologic®

Table of Contents

Docker and DevOps – Enabling DevOps Teams Through Containerization

By Michael Floyd

Software containers are a form of OS virtualization where the running container includes just the minimum operating system resources, memory and services required to run an application or service. Containers enable developers to work with identical development environments and stacks. But they also facilitate DevOps by encouraging the use of stateless designs.

The primary usage for containers has been focused on simplifying DevOps with easy developer to test to production flows for services deployed, often in the cloud. A Docker image can be created that can be deployed identically across any environment in seconds. Containers offer developers benefits in three areas:

1. Instant startup of operating system resources
2. Container Environments can be replicated, template-ized and blessed for production deployments.
3. Small footprint leads to greater performance with higher security profile.

The combination of instant startup that comes from OS virtualization, and the reliable execution that comes from namespace isolation and resource governance makes containers ideal for application development and testing. During the development process, developers can quickly iterate. Because its environment and resource usage are consistent across systems, a containerized application that works on a developer's system will work the same way in a production system.

The instant startup and small footprint also benefits cloud scenarios.

The instant startup and small footprint also benefits cloud scenarios. More application instances can fit onto a machine than if they were each in their own VM, which allows applications to scale-out quickly.

Composition and Clustering

For efficiency, many of the operating system files, directories and running services are shared between containers and projected into each container's namespace. This sharing makes deploying multiple containers on a single host extremely efficient. That's great for a single application running in a container. In practice, though, containers making up an application may be distributed across machines and cloud environments.

The magic for making this happen is composition and clustering. Computer clustering is where a set of computers are loosely or tightly connected and work together so that they can be viewed as a single system. Similarly container cluster managers handle the communication between containers, manage resources (memory, CPU, and storage), and manage task execution. Cluster managers also include schedulers that manage dependencies between the tasks that make up jobs, and assign tasks to nodes.

Docker

Docker needs no introduction. Containerization has been around for decades, but it is Docker that has reinvigorated this ancient technology. Docker's appeal is that it provides a common toolset, packaging model and deployment mechanism that greatly simplifies the containerization and distribution of applications. These "Dockerized" applications can run anywhere on any Linux host. But as support for Docker grows organizations like AWS, Google, Microsoft, and Apache are building in support.

Software developers are challenged with log files that may be scattered in a variety of different isolated containers each with its own log system dependencies.

To manage composition and clustering, Docker offers Docker Compose that gives you a way of defining and running multi-container distributed applications. (We'll look at Docker Compose in-depth in a later chapter.) Then developers can use Docker Swarm to turn a pool of Docker hosts into a single, virtual Docker host. Swarm silently manages the scaling of your application to multiple hosts.
Another benefit of Docker is Dockerhub, the massive and growing ecosystem of applications packaged in Docker containers. Dockerhub is a registry for Dockerized applications with currently well over 235,000 public repositories. Need a Web server in a container? Pull Apache httpd. Need a database? Pull the MySQL image. Whatever major service you need, there's probably an image for it on DockerHub. Docker has also formed the Open Container Initiative (OCI) to ensure the packaging format remains universal and open.

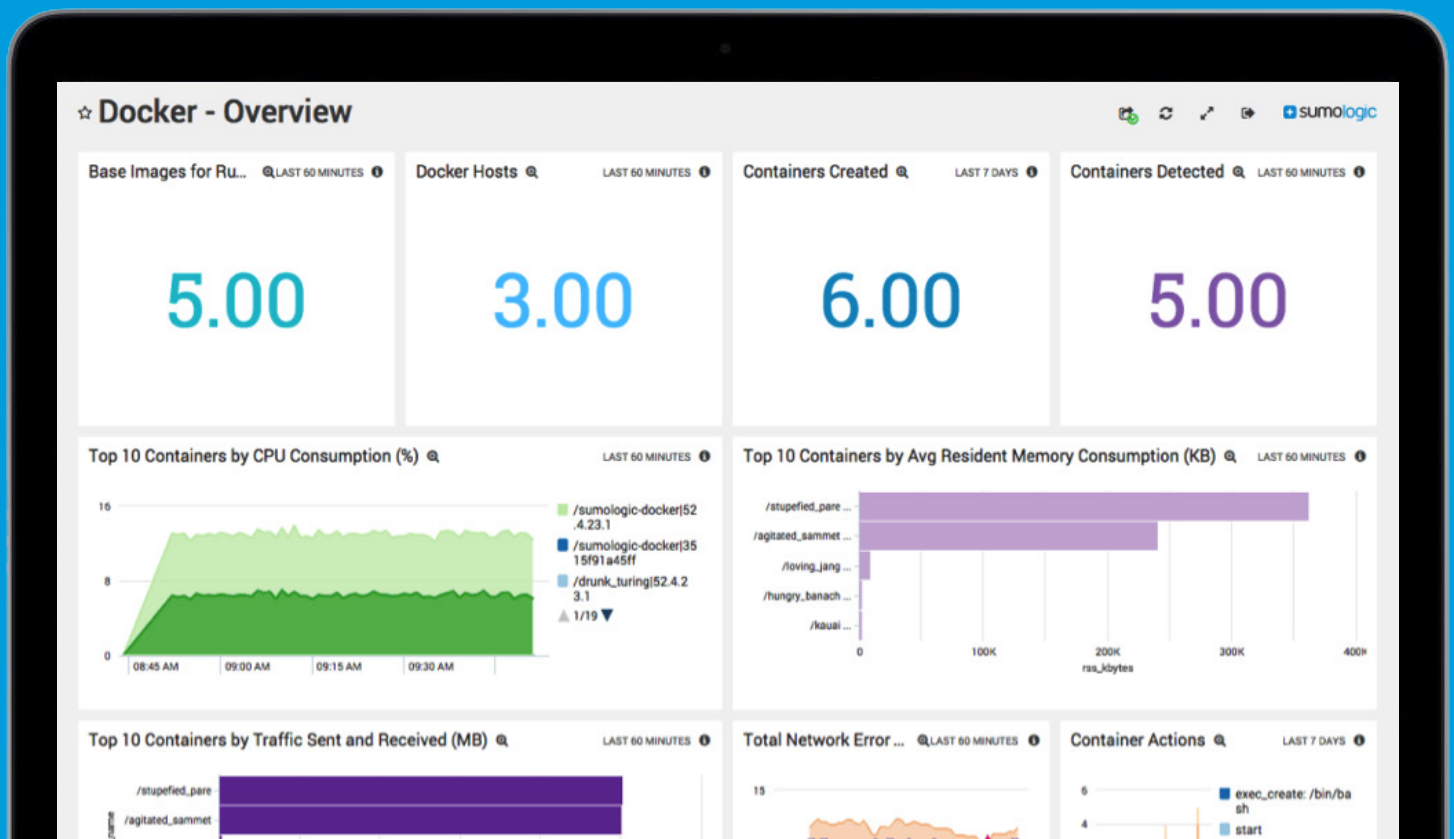
Amazon EC2 Container Service

If you running on AWS, Amazon EC2 Container Service (ECS) is a container management service that supports Docker containers and allows you to run applications on a managed cluster of Amazon EC2 instances. ECS provides cluster management including task management and scheduling so you can scale your applications dynamically. Amazon ECS also eliminates the need to install and manage your own cluster manager. ECS allows you to launch and kill Docker-enabled applications, query the state of your cluster, and access other AWS services (e.g., CloudTrail, ELB, EBS volumes) and features like security groups via API calls.

With Change Comes New Challenges

While both DevOps and containers are helping improve software quality and breaking down monolithic applications, the emphasis on automation and continuous delivery also leads to new issues.

The Sumo Logic App for Docker provides a complete overview of your Docker environment including container consumption, actions, traffic and network errors.



Software developers are challenged with log files that may be scattered in a variety of different isolated containers each with its own log system dependencies. Developers often implement their own logging solutions, and with them language dependencies. As Christian Beedgen notes later in this book, this is particularly true of containers built with earlier versions of Docker. To summarize, organizations are faced with:

- Organizing applications made up of different components to run across multiple containers and servers.
- Container security – (namespace isolation)
- Containers deployed to production are difficult to update with patches.
- Logs are no longer stored in one uniform place, they are scattered in a variety of different isolated containers.

A Model for Comprehensive Monitoring

In a later chapter, we'll show how the Sumo Logic App for Docker uses a container that includes a collector and a script source to gather statistics and events from the Docker Remote API on each host. The app basically wraps events into JSON messages, then enumerates over all running containers and listens to the event stream. This essentially creates a log for container events. In addition, the app collects configuration information obtained using Docker's Inspect API. The app also collects host and daemon logs, giving developers and DevOps teams a way to monitor their entire Docker infrastructure in real time.

Using this approach, developers no longer have to synchronize between different logging systems (that might require Java or Node.js), agree on specific dependencies, or risk of breaking code in other containers.

If you're running Docker on AWS, you can of course monitor your container environment as described above. But Sumo Logic also provides a collection of Apps supporting all things AWS including out-of-the-box solutions for Sumo Logic App for CloudTrail, AWS Config, AWS ELB, and many others, thus giving you a comprehensive view of your entire environment.

Taking Docker to Production with Confidence

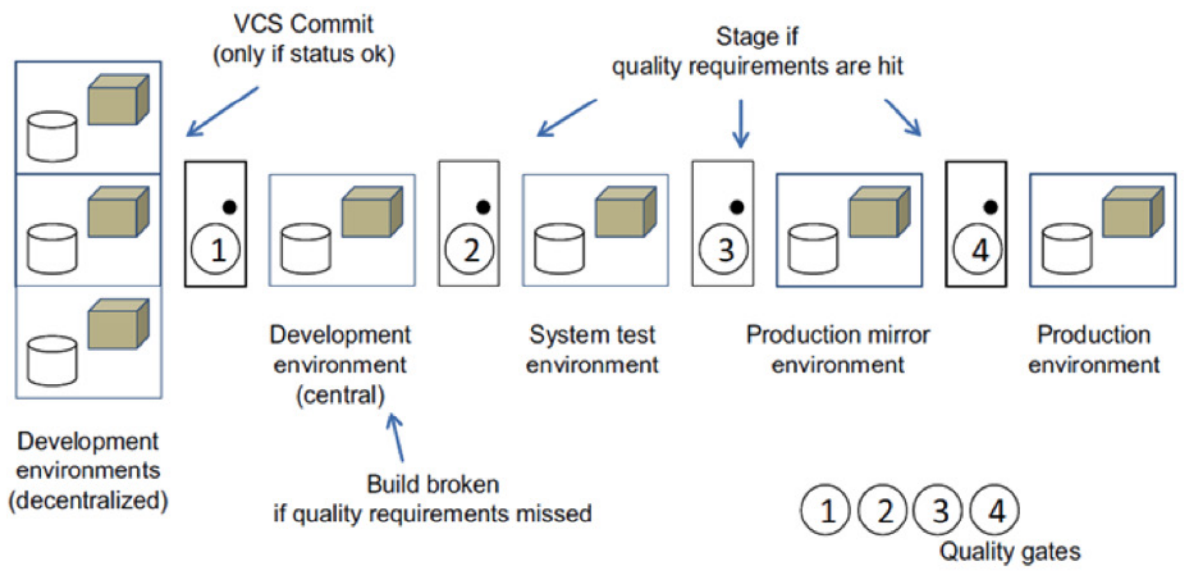
By Baruch Sadogursky

Many organizations developing software today use Docker in one way or another. If you go to any software development or DevOps conference and ask a big crowd of people "Who uses Docker?", most people in the room will raise their hands. But if you now ask the crowd, "Who uses Docker in production?", most hands will fall immediately. Why is it, that such a popular technology that has enjoyed meteoric growth is so widely used at early phases of the development pipeline, but rarely used in production?

Software Quality: Developer Tested, Ops Approved

A typical software delivery pipeline looks something like this (and has done for over a decade!)

At each phase in the pipeline, the representative build is tested, and the binary outcome of this build can only pass through to the next phase if it passes all the criteria of the relevant quality gate. By promoting the original binary we guarantee that the same binary we built in the CI server is the one deployed or distributed. By implementing rigid quality gates we guarantee the access control to untested, tested and production-ready artifacts.



Source: Hüttermann, Michael. Agile ALM. Shelter Island, N.Y.: Manning, 2012. Print.

The Unbearable Lightness of $ Docker Build

Since running a Docker build is so easy, instead of a build passing through a quality gate to the next phase…

… it is REBUILT at each phase.

"So what," you say? So plenty. Let's look at a typical build script.

```
1  FROM ubuntu                                          Latest version
2                                                       Latest version
3  RUN apt-get install -y python-software-properties python
4  RUN apt-get install -y nodejs
5  RUN mkdir /var/www                                   Latest version
6
7  ADD app.js /var/www/app.js                           Latest version
8
9  CMD ["/usr/bin/node", "/var/www/app.js"]
```

To build your product, you need a set of dependencies, and the build will normally download the latest versions of each dependency you need. But since each phase of the development pipeline is built at a different time, …

> *…you can't be sure that the same version of each dependency in the development version also got into your production version.*

But we can fix that. Let's use:

```
FROM ubuntu:14.04.
Done.
```

Or are we?

Can we be sure that the Ubuntu version 14.04 downloaded in development will be exactly the same as the one built for production? No, we can't. What about security patches or other changes that don't affect the version number? But wait; there IS a way. Let's use the fingerprint of the image. That's rock solid! We'll specify the base image as:

```
FROM
ubuntu:0bf3461984f2fb18d237995e81faa657aff260a52a795367e6725f0617f7a56c
```

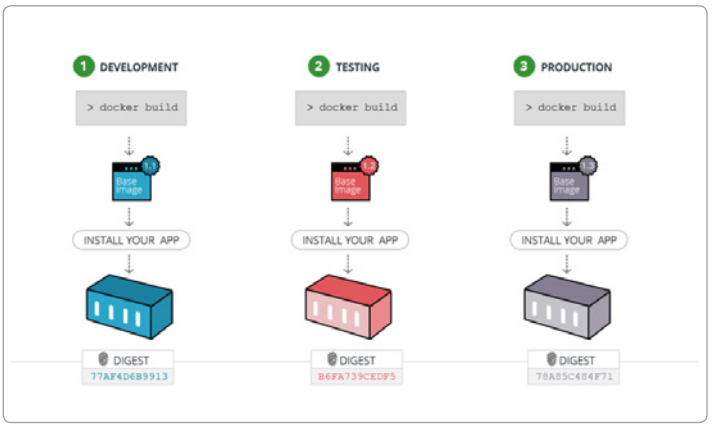
But, what was that version again? Is it older or newer than the one I was using last week?

You get the picture. Using fingerprints is neither readable nor maintainable, and in the end, nobody really knows what went into the Docker image.

Using fingerprints is neither readable nor maintainable…

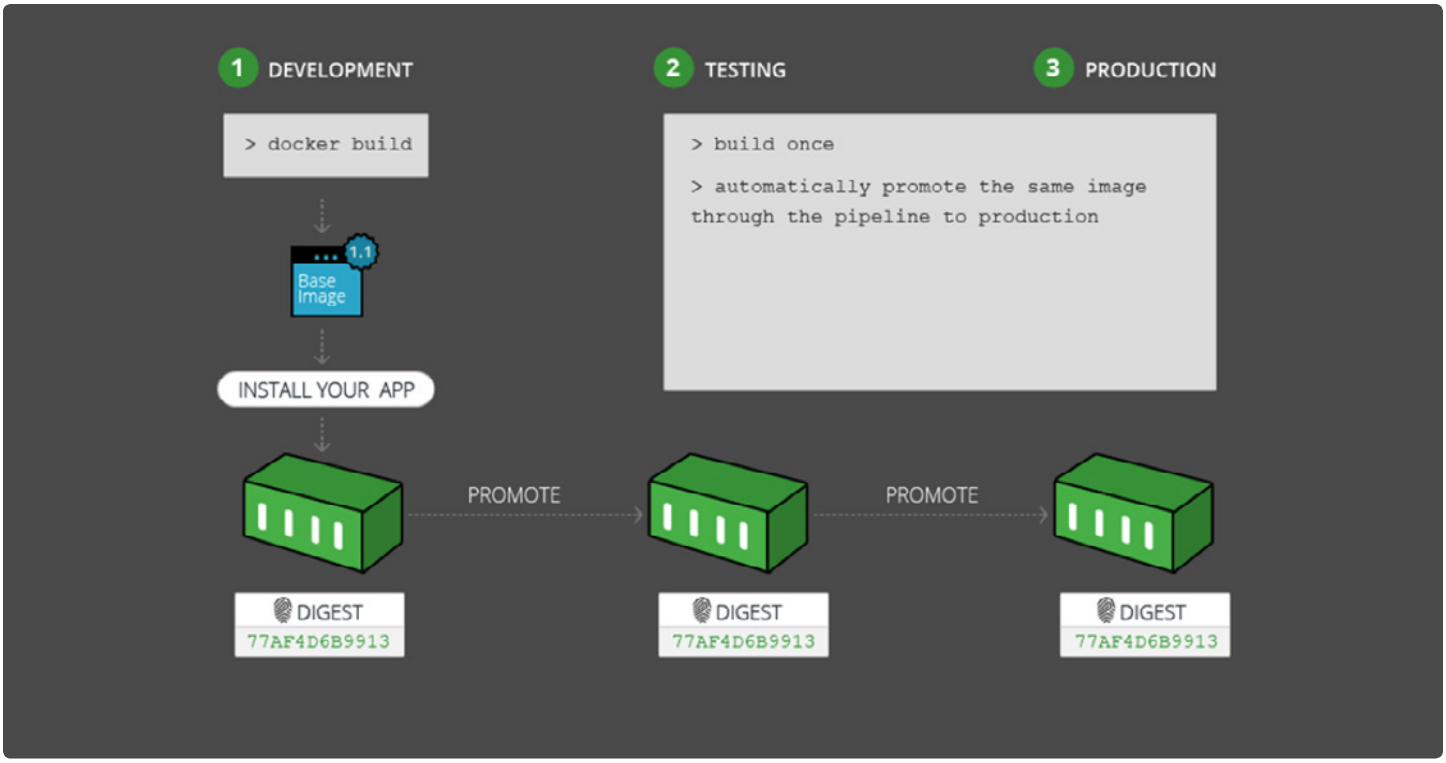
And what about the rest of the dockerfile? Most of it is just a bunch of implicit or explicit dependency resolution, either in the form of apt-get, or wget commands to download files from arbitrary locations. For some of the commands you can nail down the version, but with others, you aren't even sure they do dependency resolution! And what about transitive dependencies?

So you end up with this:



Rebuilding Docker images at each stage of the development cycle makes dependency resolution difficult.

Basically, by rebuilding the Docker image at each phase in the pipeline, you are actually changing it, so you can't be sure that the image that passed all the quality gates is the one that got to production.



Promoting your Docker image as an immutable and stable binary through the quality gates to production is a better option.

Stop rebuilding, start promoting

What we should be doing, is taking our development build, and rather than rebuilding the image at each stage, we should be promoting it, as an immutable and stable binary through the quality gates to production.

Sounds good. Let's do it with Docker.

Wait, not so fast.

Docker tag is a drag

This is what a Docker tag looks like:

```
Usage: docker tag [OPTIONS] IMAGE[:TAG] [REGISTRYHOST/][USERNAME/]NAME[:TAG]

Tag an image into a repository
```

The Docker tag limits us to one registry per host.

How do you build a promotion pipeline if you can only work with one registry?

"I'll promote using labels," you say. "That way I only need one Docker registry per host." That will work, of course, to some extent. Docker labels (plain key:value properties) may be a fair solution for promoting images through minor quality gates, but are they strong enough to guard your production deployment? Considering you can't manage permissions on labels, probably not. What's the name of the property? Did QA update it? Can developers still access (and change) the release candidate? The questions go on and on. Instead, let's look at promotion for a more robust solution. After all, we've been doing it for years with Artifactory.

Virtual repositories tried and true

Virtual repositories have been in Artifactory since version 1.0. More recently, we also added the capability to deploy artifacts to a virtual repository. This means that virtual repositories can be a single entry point for both upload and download of Docker images. Like the figure on the top right.

Here's what we're going to do:

- Deploy our build to a virtual repository which functions as our development Docker registry
- Promote the build within Artifactory through the pipeline
- Resolve production ready images from the same (or even a different) virtual repository now functioning as our production Docker registry

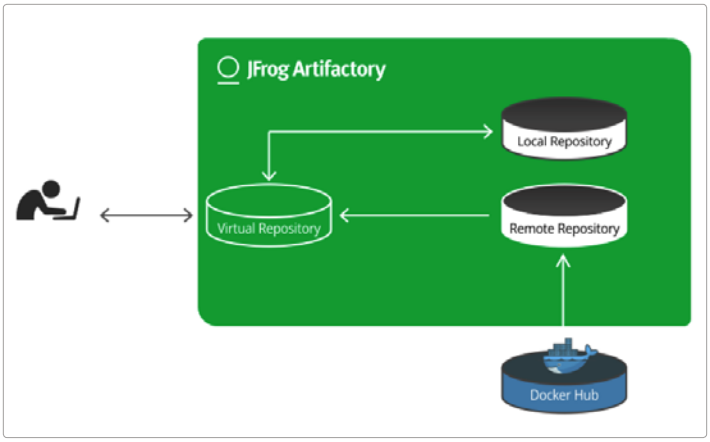
This is how it works:

Our developer (or our Jenkins) works with a virtual repository that wraps a local development repository, a local production repository, and a remote repository that proxies Docker Hub (as the first step in the pipeline, our developer may need access to Docker Hub in order to create our image). Once our image is built, it's deployed through the docker-virtual repository to docker-dev-local. See the center image on the right.

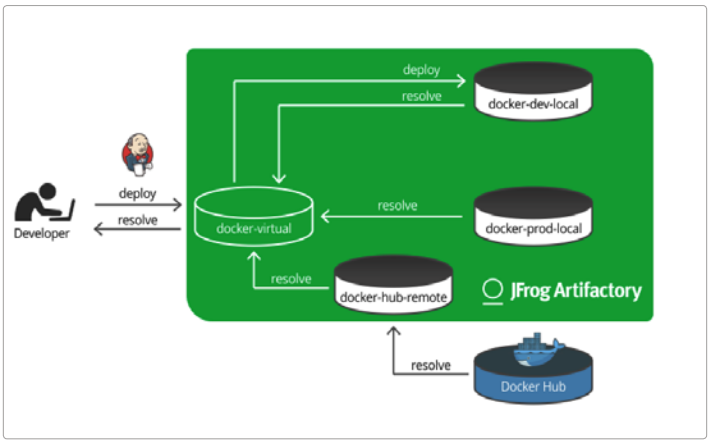
Now, Jenkins steps in again and promotes our image through the pipeline to production. See the bottom image on the right.

At any step along the way, you can point a Docker client at any of the intermediate repositories, and extract the image for testing or staging before promoting to production.

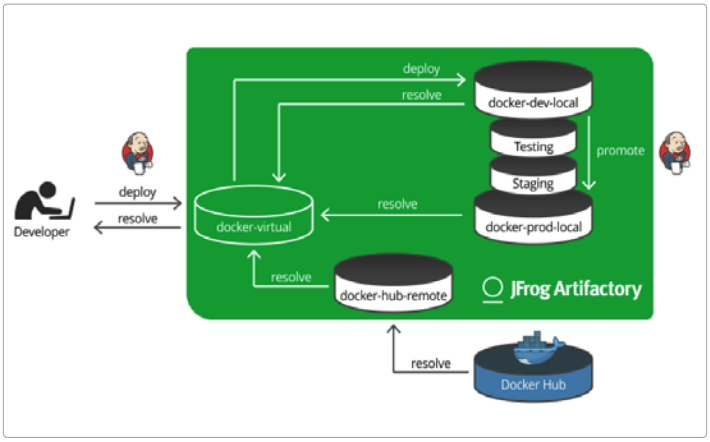
Once your Docker image is in production, you can expose it to your customers through another virtual repository functioning as your production Docker registry. You don't want customers accessing your development registry or any of the others in your pipeline. Only the production Docker registry. There is no need for any other repositories, because unlike other package formats, the point of a docker image is that it has everything it needs.



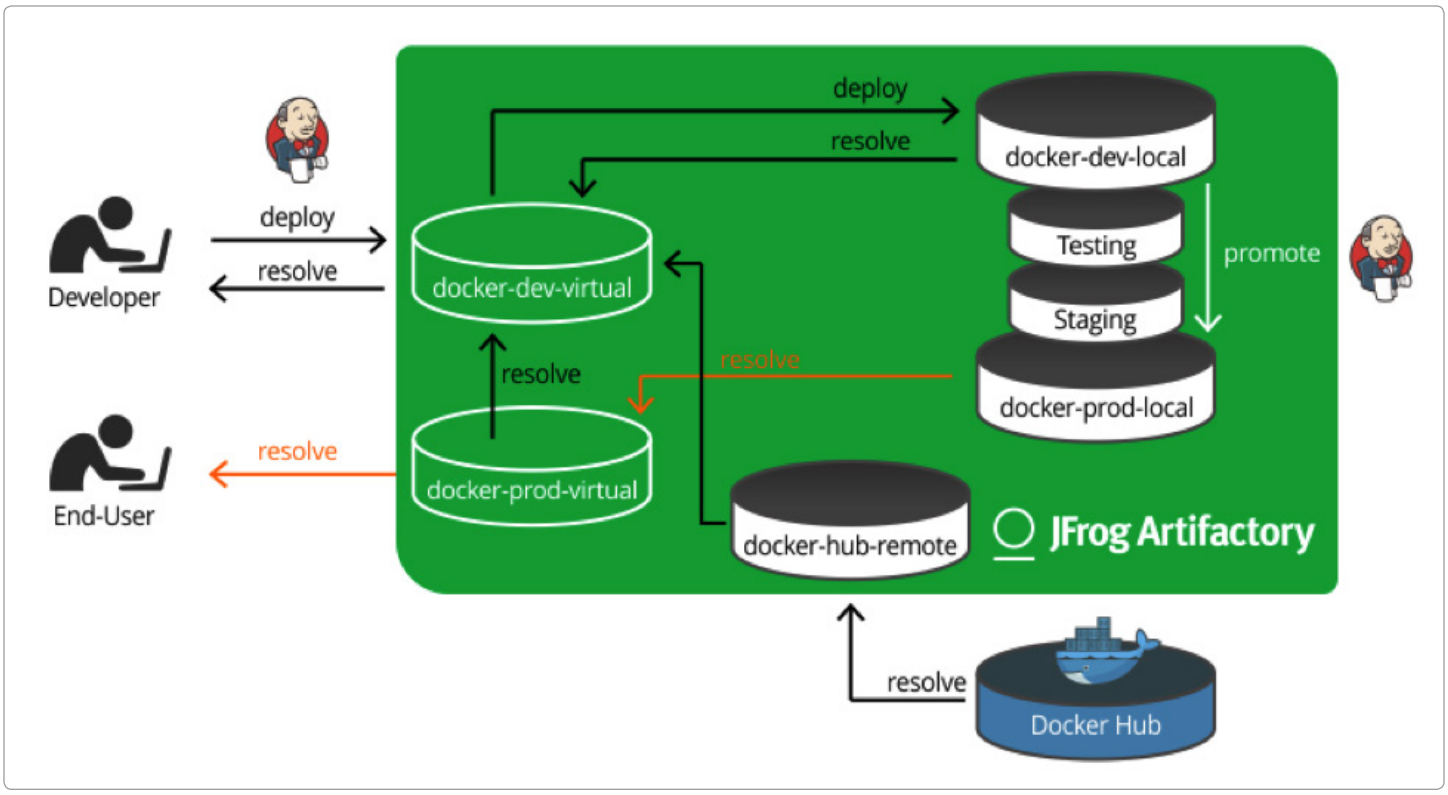
Deploying to a virtual repository.



The Docker image is promoted through the build, and deployed through the docker-virtual repository to docker-dev-local.



You can point a Docker client at any of the intermediate repositories.



Exposing your image to customers through another virtual repository functioning as your production Docker registry.

So we've done it. We built a Docker image, promoted it through all phases of testing and staging, and once it passed all those quality gates, the exact same image we created in development is now available for download by the end user or deployed to production servers, without risk of a non-curated image being received.

What about setup?

You might ask if getting Docker to work with all these repositories in JFrog Artifactory is easy to setup? Well, it's now easier than ever with our new Reverse Proxy Configuration Generator. Stick with Artifactory and NGINX or Apache and you can easily access all of your Docker registries to start promoting Docker images to production.

How to Build Applications with Docker Compose

By Faisal Puthuparackat

Application development for the cloud has always been challenging. Cloud applications tend to run on headless Linux machines, with little or no development tools installed. According to a recent survey, most developers either use Windows or Mac OS X as their primary platform. Statistically, only 21% of all developers appear to use Linux as their primary OS. About 26% use Mac OS X, and the remaining 53% of developers use various versions of Microsoft Windows. So for developers who use Windows or Mac as their primary OS, developing for Linux would require running a Linux VM to test their code. While this isn't difficult in itself, replicating this VM environment for new team members isn't easy, especially if there are a lot of tools and libraries that need to be installed to run the application code.

Docker For Development

Docker is a container mechanism that runs on Linux and allows you to package an application with all of its dependencies into a standardized unit for software development. While it is meant to act primarily as a delivery platform, it also makes for a nice standard development platform. Recent versions of the Docker toolbox aimed at Windows and Mac provide an easy path to running Docker in a VM while simultaneously providing access to the host machine's filesystem for shared access from within the Docker containers running in the VM. For applications that require extraneous services like MySQL, Postgres, Redis, Nginx, HAProxy, etc., Docker provides a simple way to abstract these away into containers that are easy to manage and deploy for development or production. This allows you to focus on writing and testing your application using the OS of your choice while still being able to easily run and debug the full application stack using Docker.

Docker Compose

Docker Compose is an orchestration tool for Docker that allows you to define a set of containers and their interdependencies in the form of a YAML file. You can then use Docker Compose to bring up part or the whole of your application stack, as well as track application output, etc. Setting up the Docker toolbox on Mac OSX or Windows is fairly easy. Head over to https://www.docker.com/products/docker-toolbox to download the installer for your platform. On Linux, you simply install Docker and Docker Compose using your native packaging tools.

An Example Application

For the sake of this exercise, let's look at a simple Python app that uses a web framework, with Nginx acting as a reverse proxy sitting in front. Our aim is to run this application stack in Docker using the Docker Compose tool. This is a simple "Hello World" application. Let's start off with just the application. This is a single Python script that uses the Pyramid framework. Let's create a directory and add the application code there. Here's what the directory structure looks like:

```
helloworld
└── app.py
```

I have created a directory called helloworld, in which there's a single Python script called app.py. helloworld here represents my checked out code tree.

This makes up the contents of my example application app.py:

```
from wsgiref.simple_server import make_server
from pyramid.config import Configurator
from pyramid.response import Response


def hello_world(request):
    print('Incoming request')
    return Response('&lt;body&gt;

&lt;h1&gt;Hello World!&lt;/h1&gt;


&lt;/body&gt;')


if __name__ == '__main__':
    config = Configurator()
    config.add_route('hello', '/')
    config.add_view(hello_world, route_name='hello')
    app = config.make_wsgi_app()
    server = make_server('0.0.0.0', 5000, app)
    server.serve_forever()
```

```
14
15
16  if __name__ == '__main__':
17      config = Configurator()
18      config.add_route('hello', '/')
19      config.add_view(hello_world, route_name='hello')
20      app = config.make_wsgi_app()
21      server = make_server('0.0.0.0', 5000, app)
22      server.serve_forever()
```

It simply listens on port 5000 and responds to all HTTP requests with "Hello World!" If you wanted to run this natively on your Windows or Mac machine, you would need to install Python, and then the Pyramid module, along with all dependencies. Let's run this under Docker instead.

It's always a good idea to keep the infrastructure code separate from the application code.

It's always a good idea to keep the infrastructure code separate from the application code. Let's create another directory here called compose and add files here to containerize this application.

Here's what my file structure now looks like. The text in bold represents new files and folders:

```
├── compose
│   └── docker-compose.yml
└── helloworld
    └── app.py
```

This makes up the contents of the docker-compose.yml:
Let's break this down to understand what our docker-compose definition means. We start off with the line "version: '2'", which tells

```
1  version: '2'
2  services:
3    helloworld:
4      image: helloworld:1.0
5      ports:
6      - &quot;5000:5000&quot;
7      volumes:
8      - ../helloworld:/code
```

Let's break this down to understand what our docker-compose definition means. We start off with the line "version: '2'", which tells Docker Compose we are using the new Docker Compose syntax.

We define a single service called helloworld, which runs from an image called helloworld:1.0. (This of course doesn't exist. We'll come to that later.) It exposes a single port 5000 on the docker host that maps to port 5000 inside the container. It maps the helloworld directory that holds our app.py to /code inside the container.

Now if you tried to run this as-is, using "docker-compose up", docker could complain that it couldn't find helloworld:1.0. That's because it's looking on the docker hub for a container image called helloworld:1.0. We haven't created it yet. So now, let's add the recipe to create this container image. Here's what the file tree now looks like:

```
├── compose
│   └── docker-compose.yml
│   └── helloworld
    └── Dockerfile
└── helloworld
└── app.py
```

We've added a new directory called helloworld inside the compose directory and added a file called Dockerfile there. The following makes up the contents of Dockerfile:

```
FROM ubuntu:14.04
MAINTAINER Your Name &amp;amp;lt;your-email@somedomain.com&amp;amp;gt;

ENV HOME /root
ENV DEBIAN_FRONTEND noninteractive

RUN apt-get -yqq update
RUN apt-get install -yqq python python-dev python-pip
RUN pip install pyramid

WORKDIR /code
CMD [&quot;python&quot;, &quot;app.py&quot;]
```

This isn't a very optimal Dockerfile, but it will do for us. It's derived from Ubuntu 14.04, and it contains the environment needed to run our Python app. It has the Python interpreter and the Pyramid Python module installed. It also defines /code as the working directory and defines an entry point to the container, namely: "python app.py". It assumes that /code will contain a file called app.py that will then be executed by the Python interpreter.

We'll now change our docker-compose.yml to add a single line that tells Docker Compose to build the application container for us if needed. This is what it now looks like:

```
version: '2'
services:
  helloworld:
    build: ./helloworld
    image: helloworld:1.0
    ports:
    - &quot;5000:5000&quot;
    volumes:
    - ../helloworld:/code
```

We've added a single line "build: ./helloworld" to the helloworld service. It instructs Docker Compose to enter the compose/helloworld directory, run a docker build there, and tag the resultant image as helloworld:1.0. It's very concise. You'll notice that we haven't added the application app.py into the container. Instead, we're actually mapping the helloworld directory that contains app.py to /code inside the container, and asking docker to run it from there. What that means is that you are free to modify the code using the developer IDE or editor of your choice on your host platform, and all you need to do is restart the docker container to run new code. So let's fire this up for the first time.

Before we start, let's find out the IP address of the docker machine so we can connect to our application when it's up. To do that, type

```
$ docker-compose up
```

You should see something like the following:

```
NAME      ACTIVE   DRIVER       STATE     URL
SWARM   DOCKER    ERRORS
default     *      virtualbox   Running   tcp://1
92.168.99.100:2376
v1.11.0
```

This tells us that the Docker VM is running on 192.168.99.100. Inside the Docker terminal, navigate to the compose directory and run:

```
$ docker-compose up
```

You're running docker-compose in the foreground. You should see something similar to this:

```
$ docker-compose up
Building helloworld
Step 1 : FROM ubuntu:14.04
 ---> b72889fa879c
Step 2 : MAINTAINER Your Name <your-email@
somedomain.com>
 ---> Running in d40e1c4e45d8
 ---> f0d1fe4ec198
Removing intermediate container d40e1c4e45d8
Step 3 : ENV HOME /root
 ---> Running in d6808a44f46f
 ---> b382d600d584
Removing intermediate container d6808a44f46f
Step 4 : ENV DEBIAN_FRONTEND noninteractive
 ---> Running in d25def6b366b
 ---> b5d310716d1f
Removing intermediate container d25def6b366b
Step 5 : RUN apt-get -yqq update
 ---> Running in 198faaac5c1b
 ---> fb86cbdcbe2e
Removing intermediate container 198faaac5c1b
Step 6 : RUN apt-get install -yqq python python-
dev python-pip
 ---> Running in 0ce70f832459
Extracting templates from packages: 100%
Preconfiguring packages ...
Selecting previously unselected package
libasan0:amd64.
...

 ---> 4a9ac1adb7a2
Removing intermediate container 0ce70f832459
Step 7 : RUN pip install pyramid
 ---> Running in 0907fb066fce
Downloading/unpacking pyramid
...

Cleaning up...
 ---> 48ef0b2c3674
Removing intermediate container 0907fb066fce
Step 8 : WORKDIR /code
 ---> Running in 5c691ab4d6ec
 ---> 860dd36ee7f6
Removing intermediate container 5c691ab4d6ec
Step 9 : CMD python app.py
 ---> Running in 8230b8989501
 ---> 7b6d773a2eae
Removing intermediate container 8230b8989501
Successfully built 7b6d773a2eae
Creating compose_helloworld_1
Attaching to compose_helloworld_1
```

…And it stays stuck there. This is now the application running inside Docker. Don't be overwhelmed by what you see when you run it for the first time. The long output is Docker attempting to build and tag the container image for you since it doesn't already exist. After it's built once, it will reuse this image the next time you run it.

Now open up a browser and try navigating to http://192.168.99.100:5000.

You should be greeted by a page that says Hello World!

So, that's our first application running under Docker. To stop the application, simply type Ctrl-C at the terminal prompt and Docker Compose will stop the container and exit. You can go ahead and change the code in the helloworld directory, add new code or modify existing code, and test it out using "docker-compose up" again.

To run it in the background:

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

To tail the container standard output:

```
$ docker-compose logs -f
```

This is a minimal application. Let's now add a commodity container to the mix. Let's pull in Nginx to act as the front-end to our application. Here, Nginx listens on port 80 and forwards all requests to helloworld:5000. This isn't useful in itself, but helps us demonstrate a few key concepts, primarily inter-container communication. It also demonstrates the container dependency that Docker Compose can handle for you, ensuring that your application comes up before Nginx comes up, so it can then forward connections to the application correctly. Here's the new docker-compose.yml file:

```
version: '2'
services:
  helloworld:
    build: ./helloworld
    image: helloworld:1.0
    volumes:
    - ../helloworld:/code
    - ./logs:/var/log
    - ./config:/etc/appconfig

  nginx:
    image: nginx:alpine
    ports:
    - &quot;80:80&quot;
    volumes:
    - ./nginx/conf.d:/etc/nginx/conf.d
    links:
    - helloworld
```

As you can see, we've added a new service here called nginx. We've also removed the port's entry for helloworld, and instead we've added a link to it from nginx. What this means is that the nginx service can now communicate with the helloworld service using the name helloworld. Then, we also map the new nginx/conf.d directory to /etc/nginx/conf.d inside the container. This is what the tree now looks like:

```
├── compose
│ ├── docker-compose.yml
│ ├── helloworld
│ │ └── Dockerfile
│ └── nginx
│ └── conf.d
│ └── helloworld.conf
└── helloworld
└── app.py
```

The following makes up the contents of compose/nginx/conf.d

```
server {

    listen 80;
    server_name helloworld.org;
    charset utf-8;

    location / {
        proxy_pass http://helloworld:5000;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_fo
    }
}
```

This tells nginx to listen on port 80 and forward all requests for / to helloworld:5000. Although port 5000 is no longer being forwarded to by Docker, it's still exposed on the helloworld container and is accessible from all other containers on the machine. This is how the connections now work:

```
browser -> 192.168.99.100(docker machine) ->
nginx:80 -> nginx-process -> hellworld:5000
```

Commodity Containers And Docker Hub

The nginx container for this example comes from the official Nginx image on the Docker Hub. This version uses Alpine Linux as its base OS, instead of Ubuntu. Not only is the Alpine Linux version smaller in size, it also demonstrates one of the advantages of dockerization—running commodity containers without worrying about underlying distribution. I could swap it out for the Debian version tomorrow without breaking a sweat.

It's possible that your cloud application actually uses cloud services like Amazon's RDS for the database, or S3 for the object store, etc. You could of course let your local instance of the application talk to the services too, but the latency and the cost involved may beg for a more

developer-friendly solution. The easy way out is to abstract the access to these services via some configuration and point the application to local containers that offer the same service instead. So instead of Amazon's RDS, spin up a MySQL container and let your application talk to that. For Amazon S3, use LeoFS or minio.io in containers, for example.

Container Configuration

Unless you've created your own images for the commodity services, you might need to pass on configuration information in the form of files or environment variables. This can usually be expressed in the form of environment variables defined in the docker-compose.yml file, or as mapped directories inside the container for configuration files. We've already seen an example of overriding configuration in the nginx section of the docker-compose.yml file.

Managing Data, Configuration and Logs

For a real-world application, it's very likely that you have some persistent storage in the form of RDBMS or NoSQL storage. This will typically store your application state. Keeping this data inside the commodity container would mean you couldn't really swap it out for a different version or entity later without losing your application data. That's where data volumes come in. Data volumes allow you to keep state separately in a different container volume. Here's a snippet from the official Docker Compose documentation about how to use data volumes:

```
version: '2'
services:
  db:
    image: postgres
    volumes:
      - mydata:/var/lib/postgresql/data
      - ./logs:/var/log
volumes:
  mydata: {}
```

The volume is defined in the top level volumes section as mydata. It's then used in the volumes section of the db service and maps the mydata volume to /var/lib/postgesql/data, so that when the postgres container starts, it actually writes to a separate container volume named mydata.

While our example only mapped code into the application container, you could potentially get data out of the container just as easily. In our data volume example, we map a directory called logs to /var/log inside the postgres container. So all postgres logs should end up in the logs directory, which we could then analyze using our native Windows/Mac tools. The Docker toolbox maps volumes into the VM running the docker daemon using vboxfs, Virtualbox's shared filesystem implementation. It does so transparently, so it's easy to use without any extra setup.

Docker is constantly evolving, and each version of the core Docker Engine, as well as the associated tools, are constantly improving. Utilizing them effectively for development should result in a dramatic improvement in productivity for your team.

References

Developer OS split statistics
Docker toolbox
Docker Compose reference

Docker Logging and Comprehensive Monitoring

By Michael Floyd

Support for Docker logging has evolved over the past two years, and the improvements made from Docker 1.6 to today have greatly simplified both the process and the options for logging. However, DevOps teams are still challenged with monitoring, tracking and troubleshooting issues in a context where each container emits its own logging data. Machine data can come from numerous sources, and containers may not agree on a common method. Once log data has been acquired, assembling meaningful real-time metrics such as the condition of your host environment, the number of running containers, CPU usage, memory consumption and network performance can be arduous. And if a logging method fails, even temporarily, that data is lost.

Docker Logging

When it comes to logging in Docker, the recommended pathway for developers has been for the container to write to its standard output, and let Docker collect the output. Then you configure Docker to either store it in files, or send it to syslog. Another option is to write to a directory, so the plain log file is the typical /var/log thing, and then you share that directory with another container.

In practice, When you stop the first container, you indicate that /var/log will be a "volume," essentially a special directory, that can then be shared with another container. Then you can run tail -f in a separate container to inspect those logs. Running tail by itself isn't extremely exciting, but it becomes much more meaningful if you want to run a log collector that takes those logs and ships them somewhere. The reason is you shouldn't have to synchronize between application and logging containers (for example, where the logging system needs Java or Node.js because it ships logs that way). The application and logging containers should not have to agree on specific dependencies, and risk breaking each others' code.

The application and logging containers should not have to agree on specific dependencies, and risk breaking each others' code.

Another issue is aligning logging methods. Prior to Docker 1.6, there were no less than 10 options for container logging - some better than others. You could:

1. Log Directly from an Application
2. Install a File Collector in the Container
3. Install a File as a Container
4. Install a Syslog Collector as a Container
5. Use Host Syslog for Local Syslog
6. Use a Syslog Container for Local Syslog
7. Log to Stdout and use a file collector
8. Log to StdOut and use Logspout
9. Collect from the Docker File systems (Not recommended)
10. Inject Collector via Docker Exec

Logging drivers have been a very large step forward in the last 12 months, and there have been incremental logging enhancements since.

However, there are still different methods to log in Docker, and there's little guarantee that containers will agree on a method, especially in a distributed microservices environment. Following the principles of the 12-Factor app, a methodology for building SaaS applications, we recommend as a best practice that you limit to one process per container, with each running unbuffered and sending data to `Stdout`.

Logging Drivers In Docker Engine

Docker 1.6 added 3 new log drivers: docker logs, syslog, and log-driver null. The driver interface was meant to support the smallest subset available for logging drivers to implement their functionality. Stdout and stderr would still be the source of logging for containers, but Docker takes the raw streams from the containers to create discrete messages delimited by writes that are then sent to the logging drivers. Version 1.7 added the ability to pass in parameters to drivers, and in Docker 1.9 tags were made available to other drivers. Importantly, Docker 1.10 allows syslog to run encrypted, thus allowing companies like Sumo Logic to send securely to the cloud.

More recently we've seen proposals for Google Cloud Logging driver, and the TCP, UDP, Unix Domain Socket driver. As Sumo Logic co-founder and CTO, Christian Beedgen, points out "As part of the Docker engine, you need to go through the engine commit protocol. The benefit is there's a lot of review stability. But it can also be suboptimal because it is not really modular, and it adds more and more dependencies on third party libraries." So he poses the question of whether this should be decoupled.

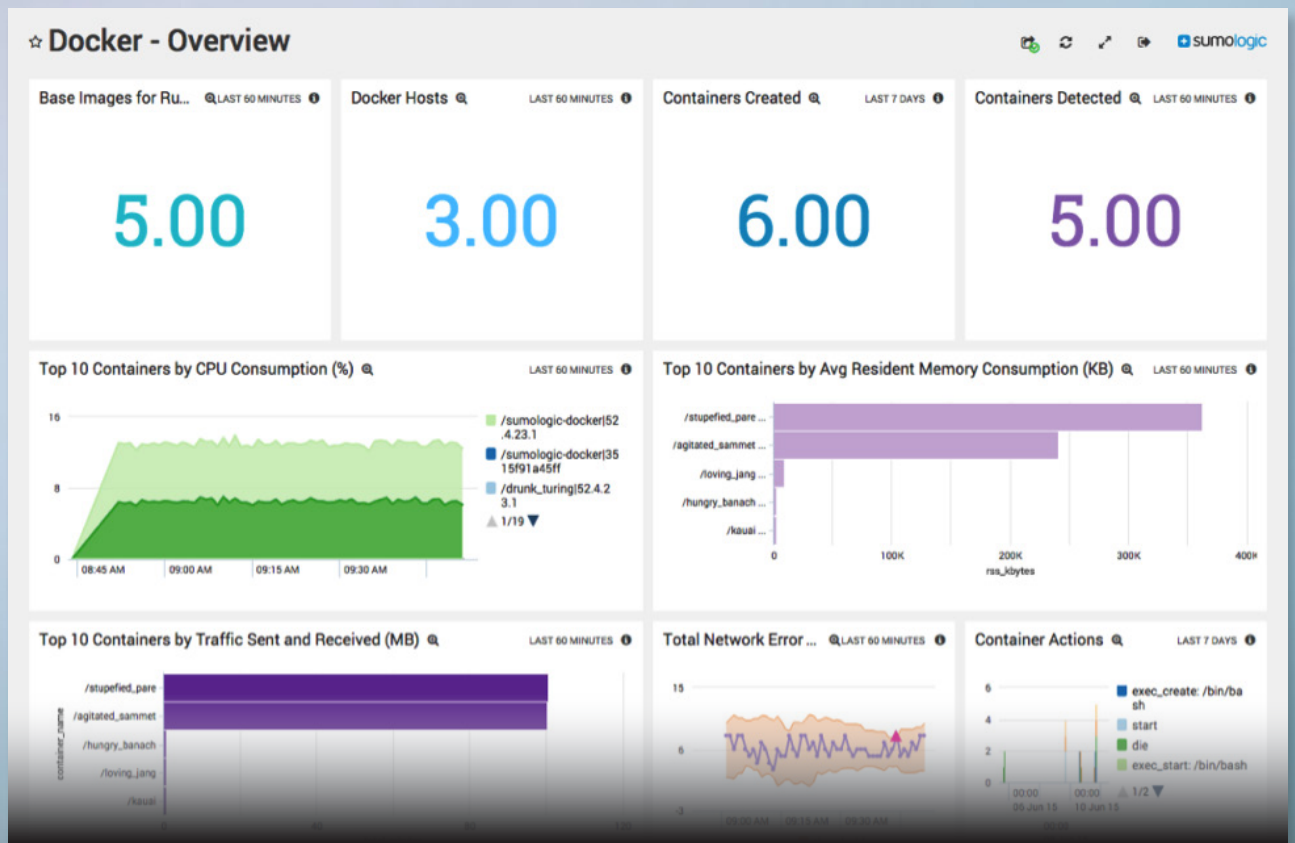
In fact, others have suggested the drivers be external plugins, similar to how volumes and networks work. Plugins would allow developers to write custom drivers for their specific infrastructure, and it would enable third-party developers to build drivers without having to get them merged upstream and wait for the next Docker release.

A Comprehensive Approach For Monitoring And Logging

As the saying goes, you can't live on logs alone. To get real value from machine-generated data, you need to look at "comprehensive monitoring." There are five requirements to enable comprehensive monitoring:

- Events
- Configurations
- Logs
- Stats
- Host and daemon logs

For events, you can send each event as a JSON message, which means you can use JSON as a way of logging each event. You



enumerate all running containers, then start listening to the event stream. Then you start collecting each running container and each start event. For configurations, you call the inspect API and send that in JSON, as well. "Now you have a record," he said. "Now we have all the configurations in the logs, and we can quickly search for them when we troubleshoot." For logs, you simply call the logs API to open a stream and send each log as, well, a log.

Similarly for statistics, you call the stats API to open a stream for each running container and each start event, and send each received JSON message as a log. "Now we have monitoring," says Christian. "For host and daemon logs, you can include a collector into host images or run a collector as a container. This is what Sumo Logic is already doing, thanks to the API."

Monitor Your Entire Docker Ecosystem

Sumo Logic delivers a comprehensive strategy for monitoring Docker infrastructure with a native collection source for events, stats, configurations and logs. Sumo Logic's advanced machine-learning and analytics capabilities enable DevOps teams to analyze, troubleshoot, and perform root cause analysis of issues surfacing from distributed container-based applications and Docker containers themselves. There's no need to parse different log formats, or manage logging dependencies between containers.

The App for Docker comes with pre-built dashboards and search queries that enable you to correlate container events, configuration information, host and daemon logs to get a complete overview your Docker environment. Quickly view Top 10 Active containers by memory consumption, CPU consumption or by traffic sent and received.

The Sumo Logic App for Docker provides:

- Native collection source for entire Docker infrastructure
- Real-time monitoring of Docker infrastructure including stats, events and container logs
- Ability to troubleshoot issues and set alerts on abnormal container or application behavior
- Visualizations of key metrics and KPIs, including image usage, container actions and faults, as well as CPU/Memory/Network statistics
- Ability to easily create custom and aggregate KPIs and metrics using Sumo Logic's powerful query language
- Advanced analytics powered by Log Reduce, Anomaly Detection, Transaction Analytics, and Outlier Detection

The Sumo Logic App for Docker uses a container that includes a collector and a script source to gather statistics and events from the Docker Remote API on each host. The app wraps events into JSON messages, then enumerates over all running containers and listens to the event stream. This essentially creates a log for container events. In addition, the app collects configuration information obtained using Docker's Inspect API, and collects host and daemon logs, giving developers and DevOps teams a way to monitor their entire Docker infrastructure in real time.

Sumo Logic's advanced machine-learning and analytics capabilities enable DevOps teams to analyze, troubleshoot, and perform root cause analysis of issues surfacing from distributed container-based applications and Docker containers themselves.

To learn more, please visit any of the following resources for more information:

- Official SumoLogic Collector Docker Image
- Sumo Logic Docker Blog
- Collect Docker Logs w/Sumo Logic
- Docker Website

Download Christian's Docker presentation on Slideshare.

sumologic®

Toll-Free: 1.855.LOG.SUMO | **Int'l:** 1.650.810.8700
305 Main Street, Redwood City, CA 9460

www.sumologic.com