Optimizing JAVA Based Docker Containers

For Openshift Deployments

By Veerendra Akula, Principal Software Engineer, ATPCO; Anandhi Navaneethakrishnan, Solution Architect, ATPCO; Vikas Grover, Sr. Solution Architect, Red Hat

Background and overview

ATPCO is an airline-owned organization at the center of a global network of airlines, data distributors, online travel agents, and other industry groups. ATPCO manages complex pricing and shopping data from more than 430 airlines in 160 countries and distributes it to various industry companies—for example, technology companies like Amadeus and Sabre, online travel agencies like Expedia, Google, and Kayak, and airlines' central reservation systems. ATPCO manages nearly 4 million fare changes in the market every day.

ATPCO's customers needed faster releases to ensure they remain competitive and sought data from the company in multiple channels to integrate with their internal business process tools. Traditional ATPCO applications, based on a monolithic architectural approach representative of the last decade's strategy, was not up to the task. Also, the applications were tightly coupled to presentation and custom designed for a UI focused interaction with the customers. ATPCO sought a solution to provide faster feedback to its application development teams to speed releases and to support multiple interfaces to integrate with customers' business processes.

A microservices-based architecture combined with container technology running on Red Hat® OpenShift® Container Platform was the solution adopted by ATPCO to address the requirement for an agile IT environment that provides rapid, continuous delivery of highly scalable solutions with multiple interfaces to its customers. During this transition to microservices and container-based architecture, the ATPCO team realized that building the services were easy, but tuning them for optimal resource footprint and achieving higher efficiencies was challenging. While addressing these challenges, the team gained an in-depth understanding of Docker and the behavior of Java™ virtual machines (JVMs) inside containers. This document shares some of the insights learned from ATPCO's experience for building, deploying and configuring high performance container applications running on OpenShift platform.

Problems, solutions, and lessons learned

Here are the main problems we encountered and lessons learned from them:

1. Docker image size
2. Running JVMs in containers
3. Build/Deployment optimization

Problem 1: Docker image size

We created OpenShift builder images using the Atomic Docker file approach. The builder image was composed of 4 distinct Atomic images consisting of a base OS (Red Hat® Enterprise Linux®), utilities (like *wget , tar, unzip, bzip2, which,* etc.), application runtime, and build tools. With Java-based workloads (Spring Boot), we had Java and Maven as runtime and build tools.

The idea was to build the final composite builder image from Atomic images in case if we needed to reuse the Atomic images. As we started building our images, we discovered that our custom images were ballooning in size quickly. It's not too big of a deal to have a couple of gigs worth of images on your local system, but it becomes a bit of pain when you start pushing/pulling these images across the network on a regular basis and the number of images grows. Initially, we didn't pay attention to these images, as our goal was to get the application up and running quickly, and there were only a few of them. However, once we had lots of applications and services running, we were compelled to inspect the images closely.

Docker image size matters

In the world of Docker and OpenShift, the size of the Docker image does matter. Large Docker images can have the following side effects:

1. Larger images take longer to download, which makes the images harder to distribute.
2. Each pull/push of a large image can take more time and puts unnecessary load on the network.
3. Larger images take up more disk space, both in the node and in the registry, which will call for frequent purge cycles and add more storage overhead.
4. Larger images usually contain unwanted components, reducing image hygiene and increasing potentially unknown vulnerabilities.
5. Large image size can make the OpenShift platform unstable.

A common assumption is that if you use an OS Docker image with a smaller footprint (i.e., a few megabytes), your final application image will be small. But in reality, the customization we do to the initial OS image can cause the image to bloat, and the size can reach up to few GB, which is exactly what happened in our case.

Understanding layers in a Docker image

A Docker image is built from a series of layers. An instruction in the Docker file translates into a layer in the image. To understand why and how the layers are created, understanding of lower-level details such as root filesystems, copy-on-write, and union mounts are needed. This is covered in this tutorial and explains about the part, that the images and layers play in storage drivers.

Each layer itself is an image. Any changes to a layer creates a new layer with the updated content.

What makes an image bigger

The possible ways an image can get bigger are:

1. Having multiple layers.
2. Using an older version of the base image and making an update to it instead of using the latest version available.
3. Not cleaning the yum caches.
4. Having multiple instructions, which causes multiple layers instead of chaining the instructions into a single line.
5. Having unnecessary components that get bundled during install, update, etc.
6. Installing all the recommended packages instead of just what is required.

Steps taken to optimize the size of a Docker image—general

Here are the steps we took to optimize our image size:

1. Inspected base images to understand what is packaged inside each image
2. Cleaned the yum caches immediately after installing the packages to ensure that the cleanup was applied on the same layer.
3. Removed unnecessary yum caches and associated yum cache directory
4. Created a composite builder image and chained the individual commands into a single line, which reduced the number of layers

Steps taken to optimize the size of a Docker image—details

Step 1: Inspecting the image

Here are the initial sizes of our custom base images used to compose the builder image.

```
rhel-utils                          559.1 MB
jdk-1.8.0.131                       919.7 MB
maven                               929.2 MB
java-sti                            929.2 MB

No of layers                        33
java-sti                            930 MB
```

We noticed that there were 33 layers in total.

Here are the main Docker instructions in each image:

```
rhel-utils

FROM rhel:7.3
RUN yum install -y --setopt=tsflags=nodocs --enablerepo rhel-7-
server-optional-rpms --enablerepo rhel-7-server-rpms wget tar unzip
bzip2 which

jdk-1.8.0.131

FROM rhel-utils
RUN wget --no-cookies --no-check-certificate --header "Cookie:
oraclelicense=accept-securebackup-cookie"
"http://download.oracle.com/otn-pub/java/jdk/8u131-b11/${auth-
token}/jdk-8u131-linux-x64.rpm" -O /tmp/jdk-8-linux-x64.rpm  && yum
-y --setopt=tsflags=nodocs install /tmp/jdk-8-linux-x64.rpm  && yum
clean all  && rm  /tmp/jdk-8-linux-x64.rpm

Maven

FROM jdk-1.8.0.131
RUN curl -fsSL http://archive.apache.org/dist/maven/maven-
3/$MAVEN_VERSION/binaries/apache-maven-$MAVEN_VERSION-bin.tar.gz |
tar xzf - -C /usr/share && mv /usr/share/apache-maven-
$MAVEN_VERSION /usr/share/maven && ln -s /usr/share/maven/bin/mvn
/usr/bin/mvn

#...more
```

The base Red Hat Enterprise Linux image is around 200 MB and we did not expect the utils to exceed 5 MB. But the size of the rhel-utils image hovered around 560 MB. That led us to wonder how the image size get so big?

Step 2: Yum clean all

In this iteration, we added *yum clean all* to the RUN command on the rhel-utils:

```
rhel-utils

FROM rhel:7.3
RUN yum install -y --setopt=tsflags=nodocs --enablerepo rhel-7-
server-optional-rpms --enablerepo rhel-7-server-rpms wget tar unzip
bzip2 which && yum clean all && rm -rf /var/cache/yum
#...more
```

```
rhel-utils                              252.1 MB
```

We immediately noticed a reduction of more than 50% in the image size.

Step 3: Remove yum cache

On further inspection, we found that the /var/cache/yum directory in the container was occupying 34 MB. This provided further opportunity to reduce the size by clearing the directory.

```
rhel-utils

FROM rhel:7.3
RUN yum install -y --setopt=tsflags=nodocs --enablerepo rhel-7-
server-optional-rpms --enablerepo rhel-7-server-rpms wget tar unzip
bzip2 which && yum clean all && rm -rf /var/cache/yum
#...more
```

```
my-rhel-utils                           213.5 MB
```

We applied these three improvements across all the images and noticed the following optimized sizes.

Size after Optimization:

Images	**Original Size**	**Optimized Size**
rhel-utils	559.1 MB	213.5 MB
jdk-1.8.0.131	919.7 MB	574.1 MB
Maven	929.2 MB	583.7 MB
java-sti	929.2 MB	583.7 MB

This was close to a 50% reduction in size.

Step 4: Create a composite file—using single FROM and single RUN

In this step we combined all images [rhel-utils , jdk and maven] into a composite Docker file with a single FROM & RUN instruction to create the final image.

```
FROM rhel:7.3
    RUN yum repolist > /dev/null && \
    yum-config-manager --enable rhel-7-server-optional-rpms && \
    INSTALL_PKGS="tar \
    unzip \
    wget \
    which \
    bzip2" && \
    yum install -y --setopt=tsflags=nodocs install $INSTALL_PKGS && \

# Install Java
    wget --no-cookies --no-check-certificate --header "Cookie:
oraclelicense=accept-securebackup-cookie"
"http://download.oracle.com/otn-pub/java/jdk/8u13b11/${auth-
token}/jdk8u131-linux-x64.rpm" -O /tmp/jdk-8-linux-x64.rpm  && \
   yum -y --setopt=tsflags=nodocs install /tmp/jdk-8-linux-x64.rpm && \
   yum clean all && rm -rf /var/cache/yum && rm -rf /var/lib/yum && \
   rm -rf /var/lib/rpm && rm /tmp/jdk-8-linux-x64.rpm && \

# Install Maven
    curl https://archive.apache.org/dist/maven/maven-
3/${MAVEN_VERSION}/binaries/apache-maven-${MAVEN_VERSION}-bin.tar.gz | \
    tar -xzf - -C /opt && \
    ln -s /opt/apache-maven-${MAVEN_VERSION} /opt/maven && \
    ln -s /opt/maven/bin/mvn /usr/bin/mvn
#...more
```

```
No of layers                          14
java-sti                              560 MB
```

Summary

a) Used a single RUN command and chained all the commands together on the same RUN instruction.

b) Applied the remove and clean actions with the same command to apply the cleanup on the same layer.

c) Created a composite Docker file.

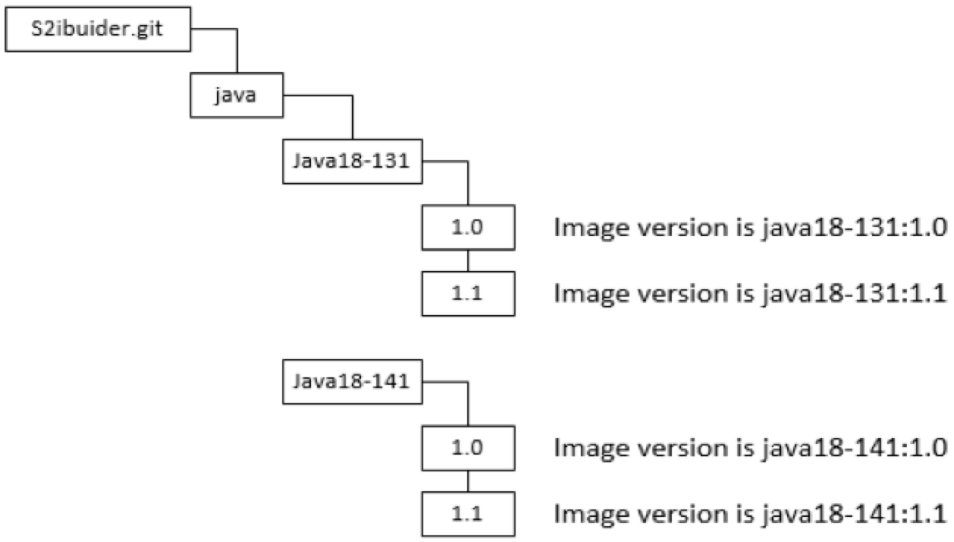
Advantages of using composite image

a) Reduces the number of image layers

b) Gives visibility to what is in the image package

c) Provides better organization of builder images for OpenShift, since the repository organization aligns with the image versioning strategy

Example



Conclusion

We reduced the image to an optimal size. This helps us maintain image hygiene with a smaller foot print and more security due to reduced attack area. We could further reduce image size by adopting a different OS, like Alpine or by using OpenJDK. We can always further optimize an image size, but we chose to stop here.

Problem 2: Running JVMs in containers

Initially, all our Java pods were unstable and frequently restarted. We noticed that the pod was killed due to being out of memory. The applications running on the OpenShift Container Platform were unstable. Following are the steps we took to address the problem.

Step 1: Added more resources

As the pods were terminated with out of memory errors, the first thought that occurred to us was to add more hardware. So, we went ahead increased the cluster capacity by two more application nodes with sizable memory. That seemed to relieve the out-of-memory issue for a period of time, but then the problem patterns began to reappear.

Step 2: Analyzed what was happening

Following are snapshots of the Spring Boot application pod using Oracle Java 8. Each request to this application created 10 MB objects.

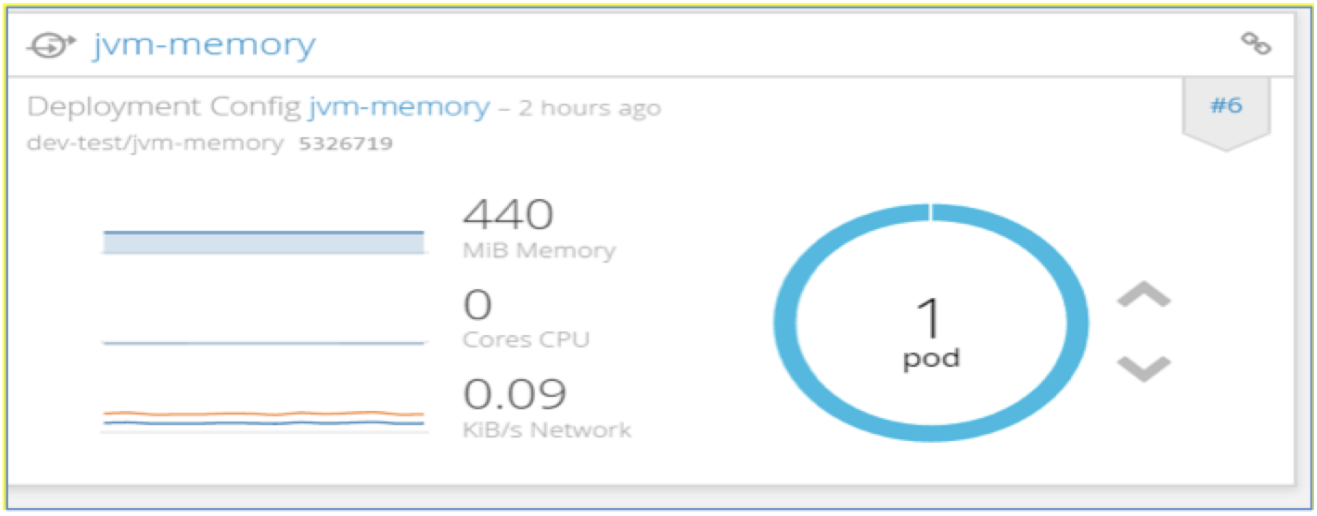


Figure 1. Memory usage after initial deployment

We did load testing on this service with 10 concurrent users with 100,000 requests. The memory usage of the pod began to increase as seen in these snapshots:

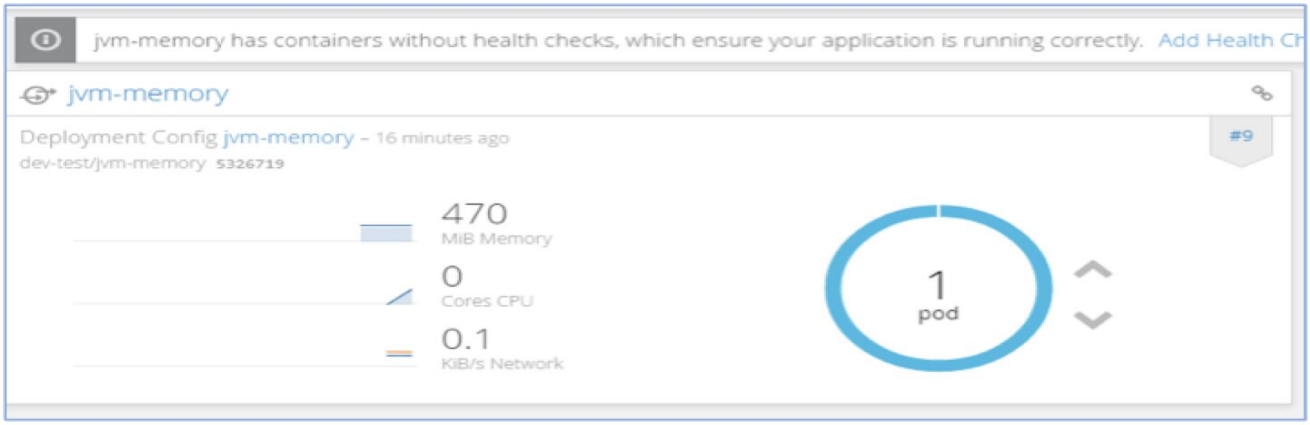


Figure 2. Memory usage increase to 470 MB

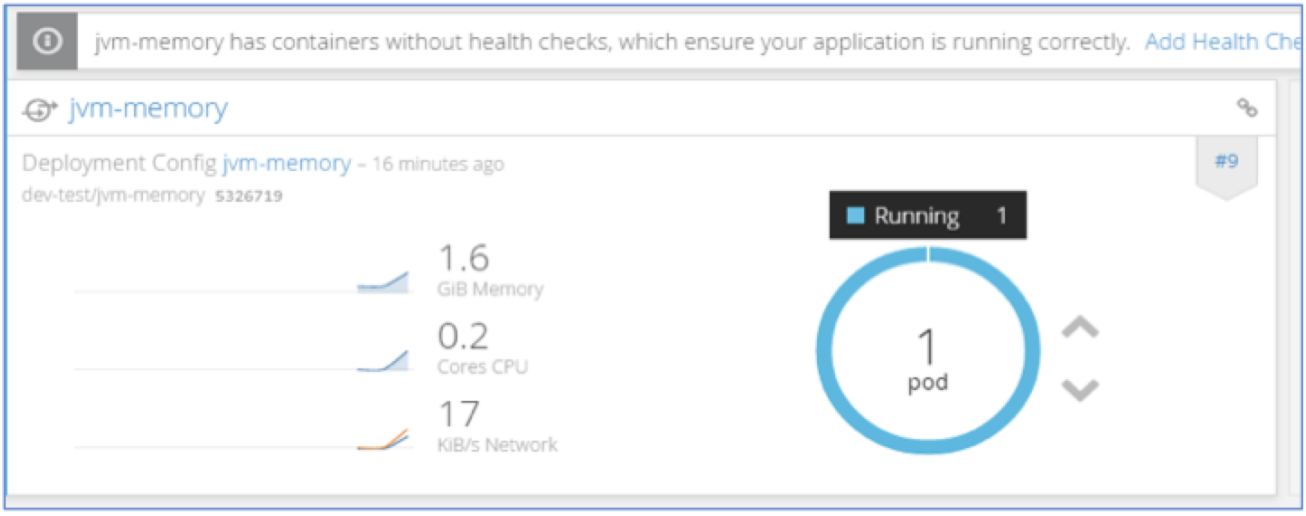


Figure 3. Memory usage increase to 1.6 GB

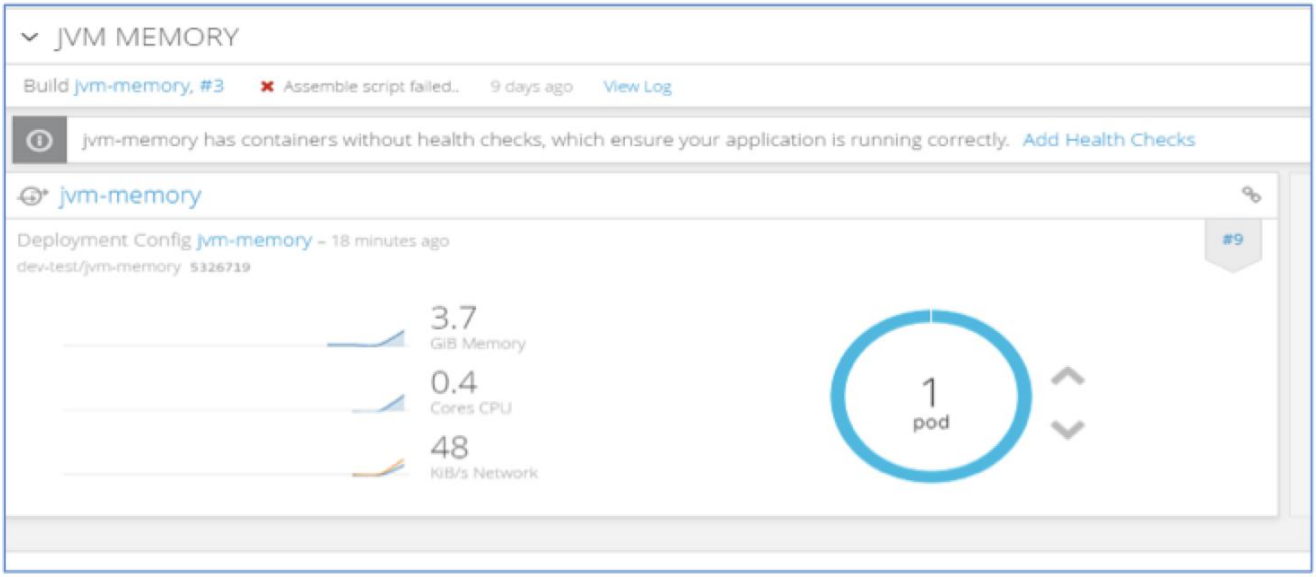


Figure 4. Memory usage increase to 3.7 GB

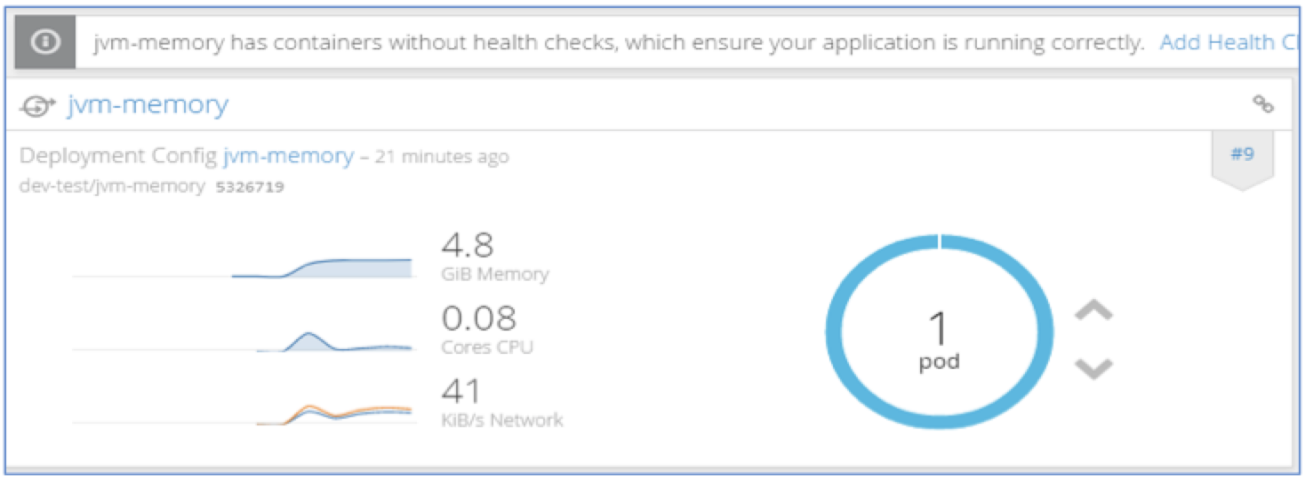


Figure 5. Memory usage increase to 4.8 GB

Another behavior we noticed was that once the heavy usage was over, the consumed memory was

never released, even when the pod was idle. Even after a month with no usage the memory used was not released, indicating that the garbage collection function did not work.

Figure 6. Memory still in use one month later

We noticed that the pods became unstable because the JVM in the pods began to hold huge amount of memory and reached the memory limit of the nodes, which caused frequent restarts.

We realized that our pods did not have any compute resource limit set; that is, CPU/memory limits on the pod. Per RedHat OpenShift recommendations, we set the request/limit on compute resources on each of our application pods to see if it limited the memory usage.

Step 3: Adjusted the quality of service (QoS) tiers

We set the pod limit of 250 MB to 1 GB for memory thinking it would be the JVM's memory limit and ran the similar load test on the Spring Boot application pod.

The application started with a memory usage of 440 MB.

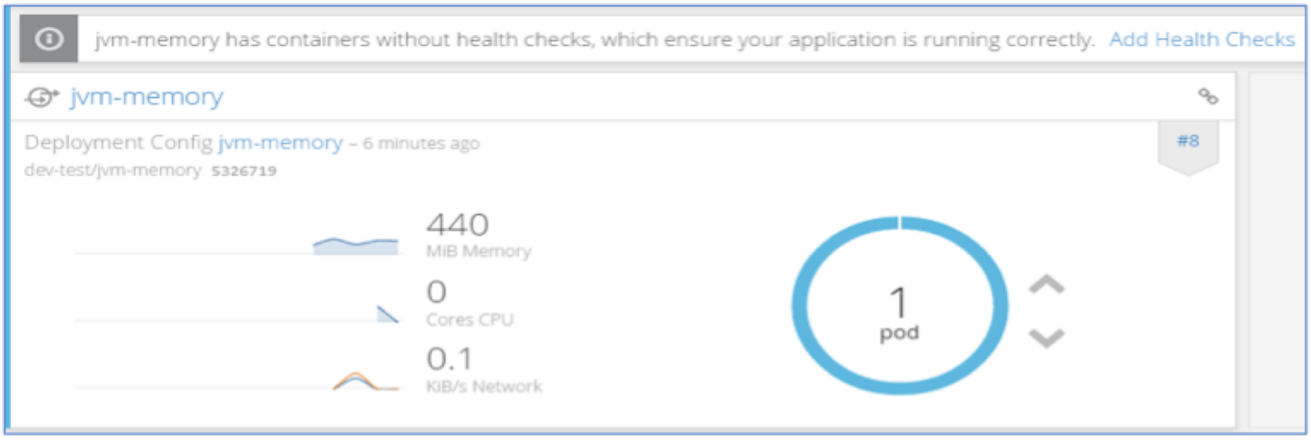


Figure 7. Initial memory usage of 440 MB

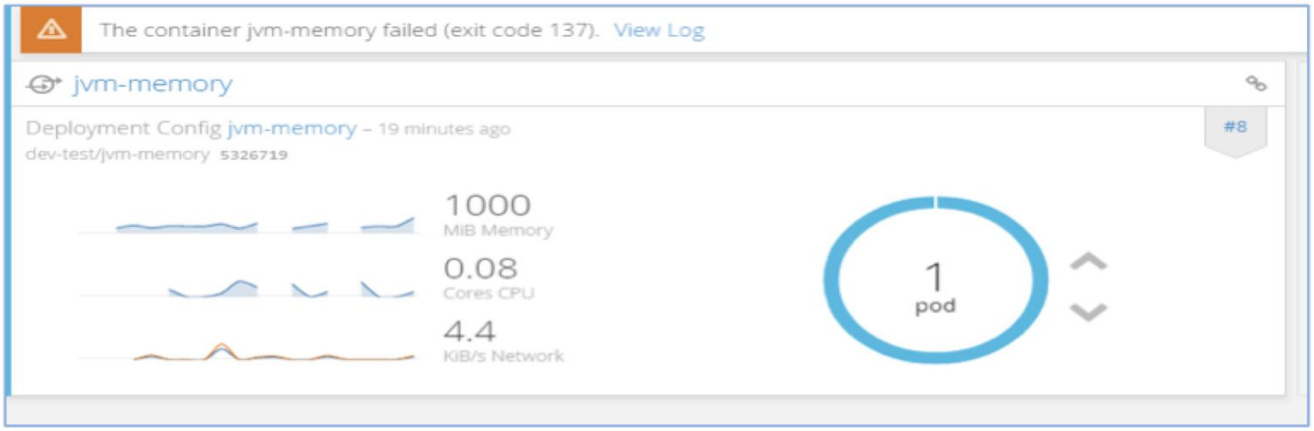


Figure 8. Memory usage increase to 1000 MB

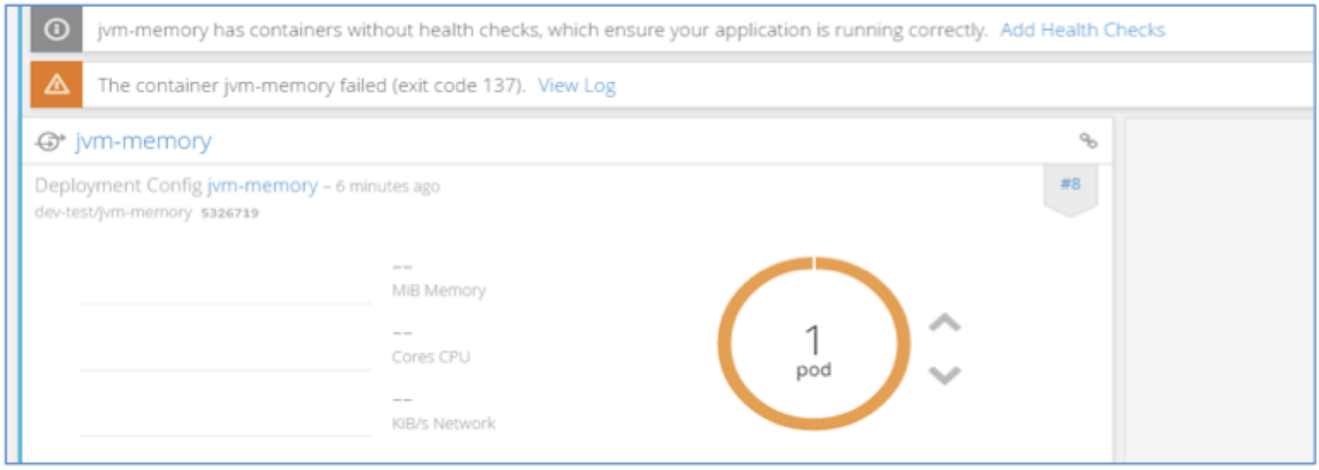


Figure 9. Memory failure

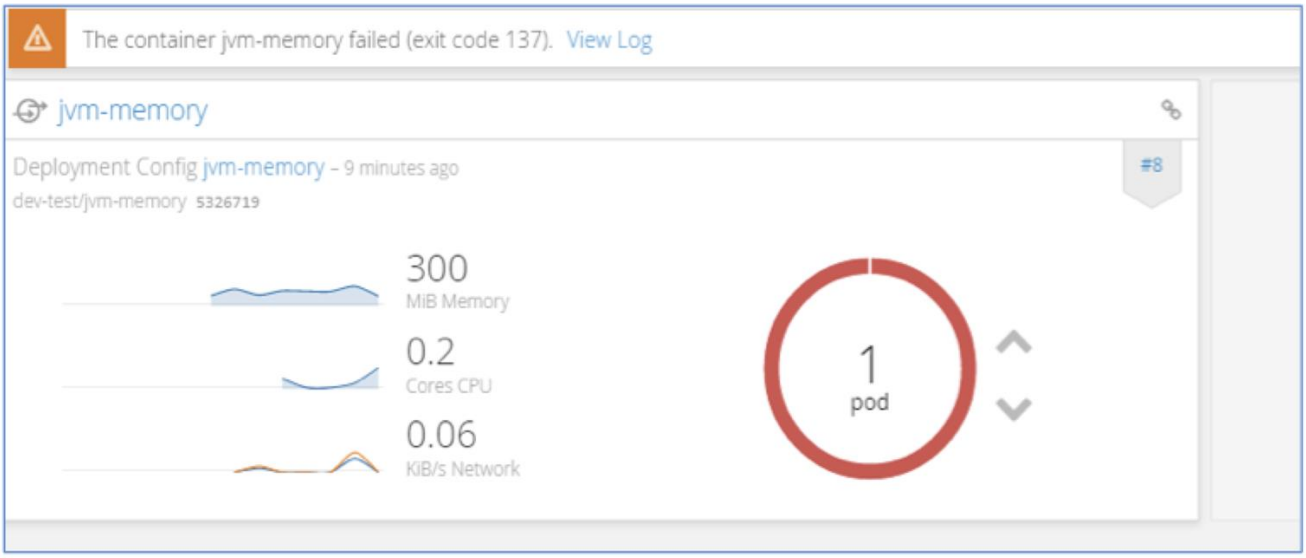


Figure 10. Crash looping because of memory issues.

Seeing the pods being unstable, we thought setting the QoS to guaranteed might help. Setting the pod limits to a burstable or guaranteed QoS did not make any difference. Pods still got terminated, often due to being out of memory.

Step 4: Looked into the pod/container

```
oc rsh <podname>      # Remote shell into the container
sh-4.2$ free -h
              total  used  free   shared  buff/cache   available
Mem:          62G    3.6G  53G    1.2G    5.3G         57G
Swap:         0B     0B    0B
sh-4.2$
```

As further analysis, we wanted to see the actual JVM allocation and usage with in a live pod. Doing a remote shell into the container/pod, we observed:

```
sh-4.2$ java -XX:+PrintFlagsFinal version | grep HeapSize

    uintx ErgoHeapSizeLimit              = 0           {product}
    uintx HeapSizePerGCThread            = 87241520    {product}
    uintx InitialHeapSize               := 790626304   {product}
    uintx LargePageHeapSizeThreshold     = 134217728   {product}
    uintx MaxHeapSize                   := 12620660736 {product}
```

The capacity of the worker node, where the application pod is placed, is 48 GB. The assumption that setting the pod limit to 1 GB would limit pod JVM's MaxHeapSize to 1 GB was not correct. If that was the case, the MaxHeapSize would be 1 GB instead of 12 GB, which is actually a fourth of the total node memory and in line with JVM's ergonomics.

Looking at the free memory, it was evident that the pod/JVM had visibility into the node's memory.

We were faced with the following questions,

1. Why does the pod/JVM have visibility into the node's memory?
2. What happened to the memory limit that we set on the pod?

Step 5: Summarized our understanding of containers and cgroups

A container is not a virtual machine

- Containers, for all intents and purposes, *look* like virtual machine, because they:
  - Have a private space for processing
  - Can execute commands as root.
  - Have a private network interface.
  - Have an IP address.

- Allow custom routes and iptable rules.
- Can mount file systems, etc.

- But the big differences between containers and virtual machines (VMs) are:
  - Containers share the host system's kernel with other containers.
  - A VM runs in its own operating system (OS), whereas all containers share a host OS.
  - VMs give full isolation, whereas containers only give process-level isolation.

JVM (version 7/8) is not control groups (cgroups) aware

- As containers share the host OS and its kernel, its process isolation is via namespaces and cgroups. While namespaces provide isolated views, cgroups are used to isolate access to resources. So cgroups can be used for either limiting access (e.g., a process group can only use a maximum of 2 GB of memory) or accounting (e.g., keeping track of how many CPU cycles a certain process group consumed over the last minute).
- Most of the commonly used Linux command tools are not cgroups aware and might show CPU and memory usage at the VM level rather than at the cgroups level. This is true for JVM.
- When the pod limits are not set, the cgroup limits are unbounded. So, the JVM is free to use the resources up to the VM's limit. If multiple such JVMs are running at a high memory usage, it soon reaches the VM's memory limit and they can begin to crash.
- When the pod limits are set, the cgroups limit is then bounded to the pod limit. Since the JVM is not cgroups aware, it tries to expand beyond the cgroups limit, which causes the pod to crash.

Step 6: Made our JVM cgroups-aware

Starting a JVM without providing any parameters will make the JVM use the VM resources as its limit rather than the pod's limits, because JVMs are not cgroups-aware. We can make JVM cgroups-aware by passing cgroups values via JVM parameters.

These items need sizing for JVM resource usage:

- Heap
- Threads
- Metaspace
- Garbage collection

How to set defaults for JVM parameters:

- Calculate the max number of core(s) that can be utilized using CPU cgroups limits from "/sys/fs/cgroup/cpu/cpu.cfs_period_us" and "/sys/fs/cgroup/cpu/cpu.cfs_quota_us"

  For details refer: **Fabric8 Java base image OpenJDK 8** https://github.com/fabric8io-images/java/tree/master/images/jboss/openjdk8/jdk

- Calculate the max memory that can be utilized using cgroups limits from "/sys/fs/cgroup/memory/memory.limit_in_bytes"
- jvm can be tuned with the following settings to the max number of cores previously calculated using CPU cgroups limits:

- -XX:ParallelGCThreads
- -XX:ConcGCThread
- -Djava.util.concurrent.ForkJoinPool.common.parallelism

- Memory limits can be set using: -Xmx It is good practice to set the -Xmx limit to a default value. (If this option is not set explicitly then provide a percentage value via environmental variable to the percentage value of the cgroups memory limit.)
- Set garbage collection (GC) type explicitly: -XX:+UseParallelGC
- Specify the min and max for the mapped pages. These parameters ensure that if the mapped heap space is less than the min, then it is extended by mapping more physical pages. If the mapped heap space is more than the max, then it is trimmed by removing the mapped pages. These limits are constrained by -Xms and Xmx settings:
  - -XX:MinHeapFreeRatio=20
  - -XX:MaxHeapFreeRatio=40
- Setting GC time goals via 2 parameters:
  - The GCTimeRatio tells, how much time should be spent on GC. With setting this parameter the time goal no longer dominates and heap size stays between the MinHeapFreeRatio and MaxHeapFreeRatio.
  - Setting AdaptiveSizePolicyWeight to 90 means that GC timing goals check is mostly based on current GC time.
    - -XX:GCTimeRatio=4
    - -XX:AdaptiveSizePolicyWeight=90
- Setting max limit for the metaspace will control the unbounded memory growth and will release memory if not used: -XX:MaxMetaspaceSize=100m

*The above calculations and settings can be sourced in as **startup script** to the container and are used when starting the jvm.*

After setting pods limits along with the JVM parameters, bounding to cgroup limits, the pods began to behave as predicted. Here are snapshots of the same Spring Boot application after the initial deployment with the changes:

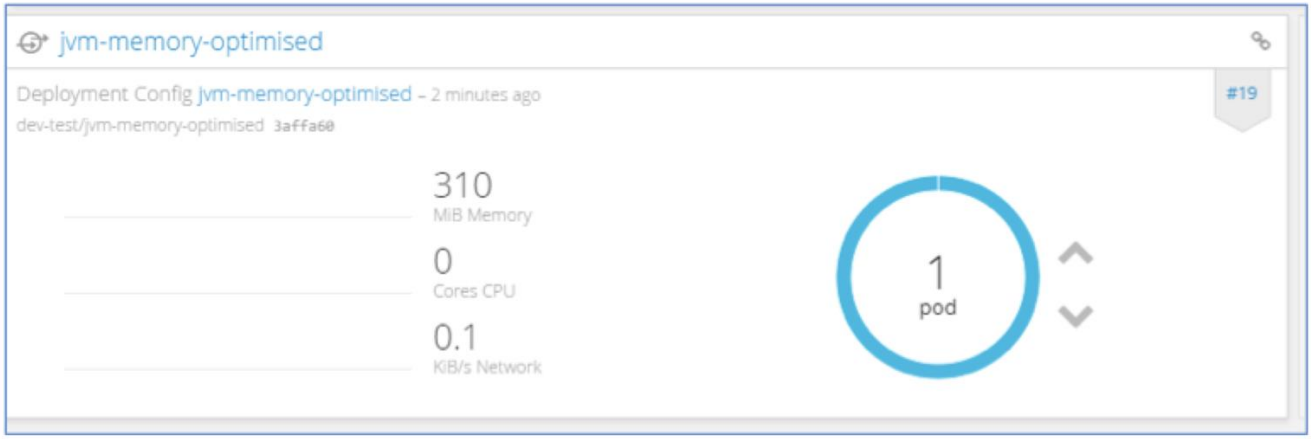


Figure 11: Snapshot after 100,000 request with 100 concurrent users with memory usage very stable

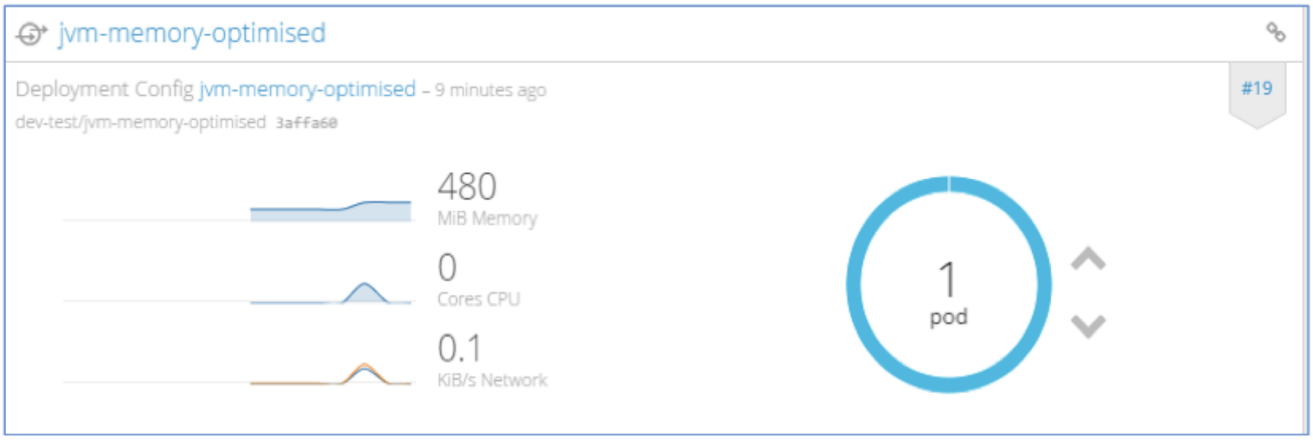


Figure 12: Snapshot after 1,000 requests with 10 concurrent users

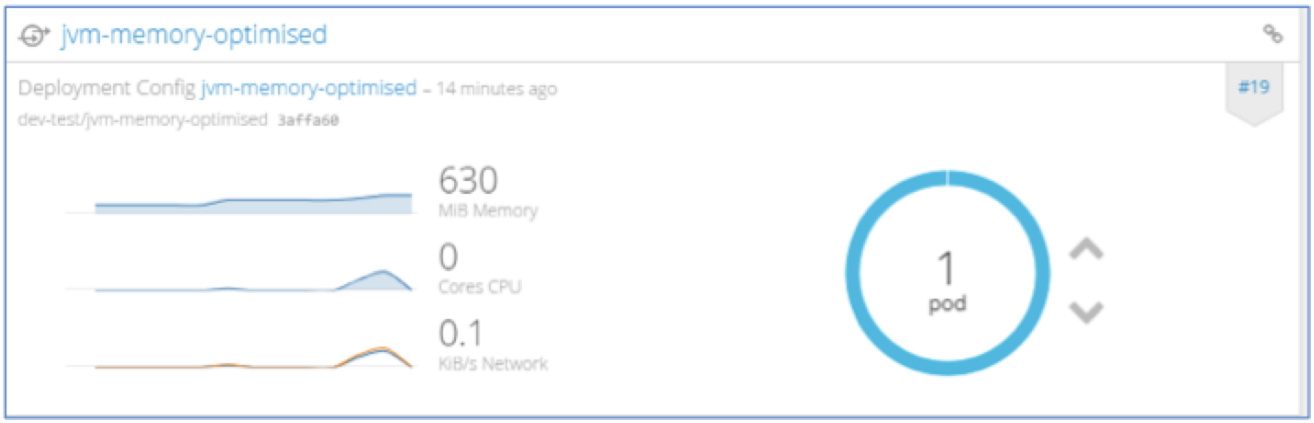


Figure 13: Snapshot after 100,000 requests with 100 concurrent users with memory usage very stable

Conclusion

While running a JVM in a pod, be aware of the cgroups. First set the pod limit, so that the cgroups limits are set, then make the JVM aware of the cgroup limits by passing the necessary JVM parameters.

As a best practice, always set defaults for pod limits and Java (default) options for the JVM. With this you ensure that if a JVM container runs without any limits, the defaults can set some level boundaries. Each JVM should test the thread usage and memory usage and set its limit accordingly.

Problem 3: Build/deployment optimization

The present Java base image helped us bring stability to workloads running on OpenShift Container Platform. The base image footprint has been reduced by close to 50%, with the previous enhancement adopting composite Docker file approach and cleanup.

There are further opportunities to reduce the size of base image footprint adopting alpine OS, OpenJDK /JRE, or by changing the build strategy to package the application artifact with JRE instead of building the application artifact with JDK and Maven using the base image.

After exploring the options and getting the opinion of Red Hat experts, we were advised to adopt Red Hat® Enterprise Linux® Atomic Base image with JRE and change the build strategy to reduce the image footprint instead of using non-supported opensource software (Alpine OS / OpenJDK).

We then introspected the way we were doing the builds and noticed the "Double Build" issue.

Build and deployment—current approach

1. We use Jenkins Executor to checkout code, build an application artifact and perform unit test, integration test, etc. Once the artifacts were validated, they were pushed to Nexus.
2. We initiate S2I build, which used a source type git. In this process the code was checked out and built again, and the image was created and then deployed.

Present build and deployment process:

code checkout → compile → unit test → initiate S2I source build → code checkout → compile → create the new app image → push app image to nexus → deploy image from nexus

Pitfalls

1. We were building twice, which causes duplication, with no guarantee of what was validated in unit tests and integration tests against what was deployed.
2. The image needs to have full JDK and Maven for a successful build, but for application runtime, just the JRE would suffice.

Best practices

1. Instead of using S2I build with a source type git, use S2I build with a binary source type with the available artifact previously built and unit tested.
2. The final builder image should only have components that are needed for the runtime, which in our case was Red Hat Enterprise Linux Atomic Base image and Oracle JRE.

Implementation

This is what the implementation of the approach looked like:

```
FROM registry.access.redhat.com/rhel7-atomic:7.4-54

# install Base Packages and JRE
RUN microdnf --nodocs --enablerepo=rhel-7-server-rpms install wget
rsync which tar unzip && microdnf clean all && \

# install oracle JRE
wget --no-cookies --no-check-certificate --header "Cookie:
oraclelicense=accept-securebackup-cookie"
"http://download.oracle.com/otn-pub/java/jdk/8u131-b11/${auth-
token}/jre-8u131-linux-x64.rpm" -O /tmp/jre-8u131-linux-x64.rpm &&
rpm -ivh /tmp/jre-8u131-linux-x64.rpm --nodocs && rm /tmp/jre-
8u131-linux-x64.rpm
#...more
```

The current image size was further reduced:

```
No of layers                    14
jre-sti                         364 MB
```

s

Proposed build and deployment process:

code checkout → compile → unit test → initiate S2I binary build from Jenkins executor local app target directory → push image in nexus → deploy image from nexus

initiate S2I binary build from Jenkins executor local app target directory → oc start-build <build-name> --from-file = < Jenkins executor local app target directory/artifact.jar >

Conclusion

Putting in the effort to understand and use the right build strategy may not seem like a big deal initially when the number of builds on the platform are low. But, when the number of builds per day increases, then any reduction in build time will be beneficial, as it reduces the wait and increases the efficacy of resource usage.

References

- **The JVM and Docker: A good idea?** [video] https://www.youtube.com/watch?v=6ePUiQuaUos
- **Gracefully shutting down Java in containers** [article] https://dzone.com/articles/gracefully-shutting-down-java-in-containers
- **Running a JVM in a container without getting killed** [article] https://dzone.com/articles/running-a-jvm-in-a-container-without-getting-kille
- **Analyzing a Java memory usage in a Docker container** [blog entry] http://trustmeiamadeveloper.com/2016/03/18/where-is-my-memory-java/
- **Java inside Docker: What you must know to not FAIL** [blog entry] https://developers.redhat.com/blog/2017/03/14/java-inside-docker/#comment-1906
- **Java memory consumption in Docker and how we employed Spring Boot** [article] https://dzone.com/articles/how-to-decrease-jvm-memory-consumption-in-docker-u
- **Java RAM usage in containers: Top 5 tips not to lose your memory** [blog entry] https://blog.jelastic.com/2017/04/13/java-ram-usage-in-containers-top-5-tips-not-to-lose-your-memory/
- **Fabric8 Java base image OpenJDK 8** [documentation] https://github.com/fabric8io-images/java/tree/master/images/jboss/openjdk8/jdk
- **Keep it small: A closer look at Docker image sizing** [blog entry] https://developers.redhat.com/blog/2016/03/09/more-about-docker-images-size/
- **10 things to avoid in Docker containers [blog entry]** https://developers.redhat.com/blog/2016/02/24/10-things-to-avoid-in-docker-containers/
- **Dude, where's my PaaS memory? Tuning Java's footprint in OpenShift (Part 2)** [blog entry] https://developers.redhat.com/blog/2014/07/22/dude-wheres-my-paas-memory-tuning-javas-footprint-in-openshift-part-2/ https://github.com/redhat-cop/containers-quickstarts
- **Introducing the Red Hat Enterprise Linux Atomic base image** [blog entry] http://rhelblog.redhat.com/2017/03/13/introducing-the-red-hat-enterprise-linux-atomic-base-image/
- **About storage drivers** [documentation] https://docs.docker.com/engine/userguide/storagedriver/imagesandcontainers/
- **Refactoring a Dockerfile for image size** [blog entry] https://blog.replicated.com/refactoring-a-dockerfile-for-image-size/