LTIMindtree

WHITEPAPER

Why and how should you optimize Docker images with Jib?

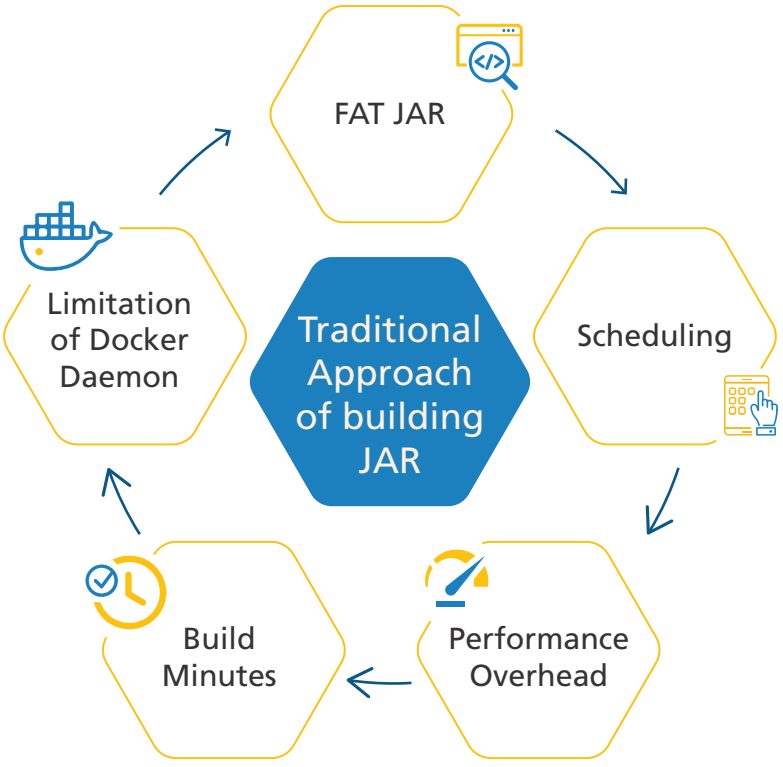
As enterprises across industries look to fast-track their digital transformation journeys, business leaders are increasingly looking to reduce their application startup time and reduce general costs over unwanted network and internet usage. Towards that end, software developers and DevOps engineers are now looking to build, run, and share applications with containers. However, as they do so, they are running into some issues.

The key challenges for Java developers, storage and long build times

Currently, most Java based applications are using Spring Boot in Docker with Maven and are creating standard fat Boot jars with all the dependencies packed inside. In this process, a new 250 MB image is created each time anything is built, even if it's very little code change, resulting in **inefficient storage space usage in our private repository.** This is because the fat jar contains both shared dependencies (which change infrequently) and our code.

In a container orchestration system, the container image is pulled from the image registry to a host running a container engine. This process is called scheduling. Pulling large-sized images from the registry result in long scheduling times in container orchestration systems and **long build times** in CI pipelines.

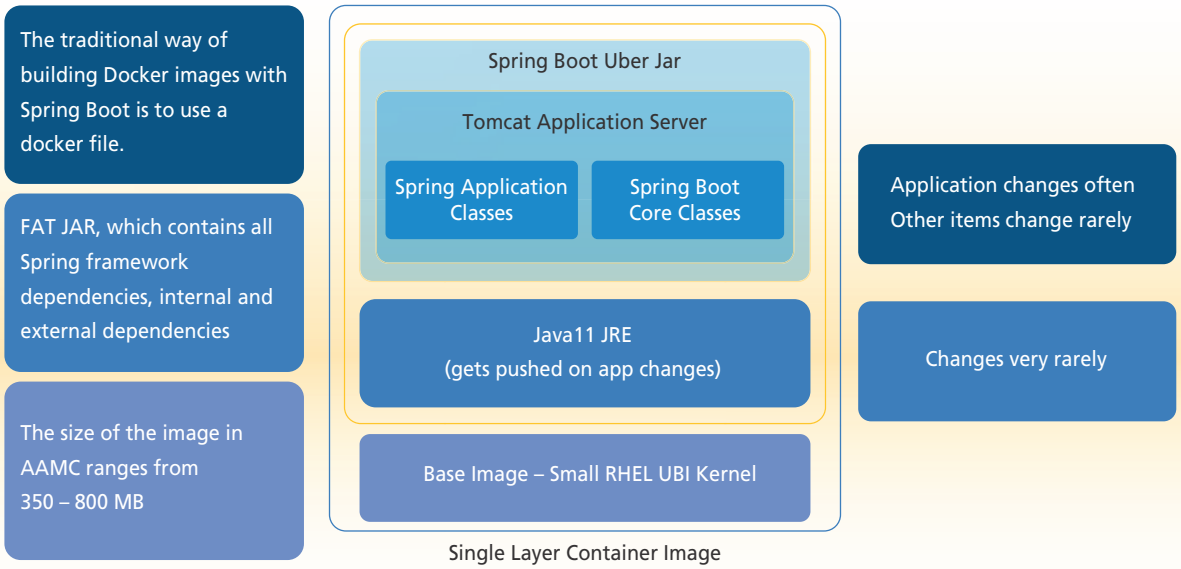
Large docker images and frequent release cycles **can eat up a lot of gigabytes of storage space and increase network traffic.** This can be potentially poor design, especially for cloud designs where you pay for traffic usage and storage. Pulling 1GB of data from your registry every single time results in long scheduling processes and long build times in CI pipelines.



When you multiply this scenario in context to all our pipelines, think about the **time the CI systems will need** to place this container and then multiply with the number of applications you have. Further, you have an additional step of first creating a docker image and then pushing it to Artifactory/ECR, again resulting in the consumption of increased build minutes.

Most organizations also have a **problem installing docker on the developer workspace.** Hence testing these builds will need to be deployed into systems that have docker installed and then validated, which can be time-consuming during critical times.

Traditional Builds



Single Layer Container Image

So, what's the solution?

We can resolve the above problem using two different approaches.

- Building and deploying only the delta
- Abstract image creation

1. Building and deploying only the delta

The process of deploying changes to an existing application is called delta deployment. Delta deployment supports several scenarios in which a full deployment process would be inefficient. Typically, we deploy the parts of the application that are frequently updated.

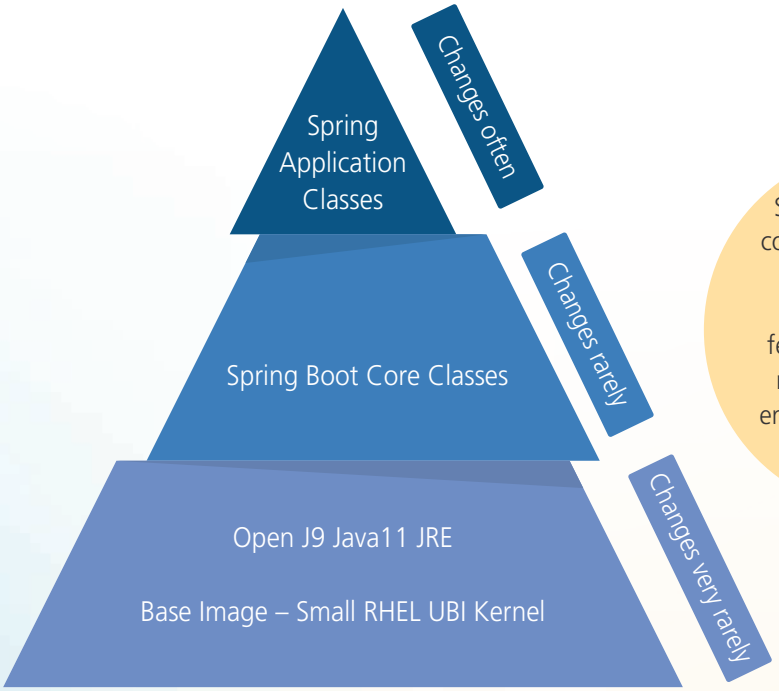
Using Layering Feature in Spring

Spring Boot splits the content of *fat Jar* into 4 layers. However, by default, this layering feature is disabled and needs to be explicitly enabled with the Spring Boot Maven plugin.

Maven Plugin

```
<plugin>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-maven-plugin</artifactId>
  <configuration>
     <layers>
        <enabled>true</enabled>
     </layers>
  </configuration>
</plugin>
```

Layering to the Solution



Spring Boot splits the content of FAT JAR into four layers, but by default, this layering feature is disabled and needs to be explicitly enabled with the Spring Boot Maven Plugin.

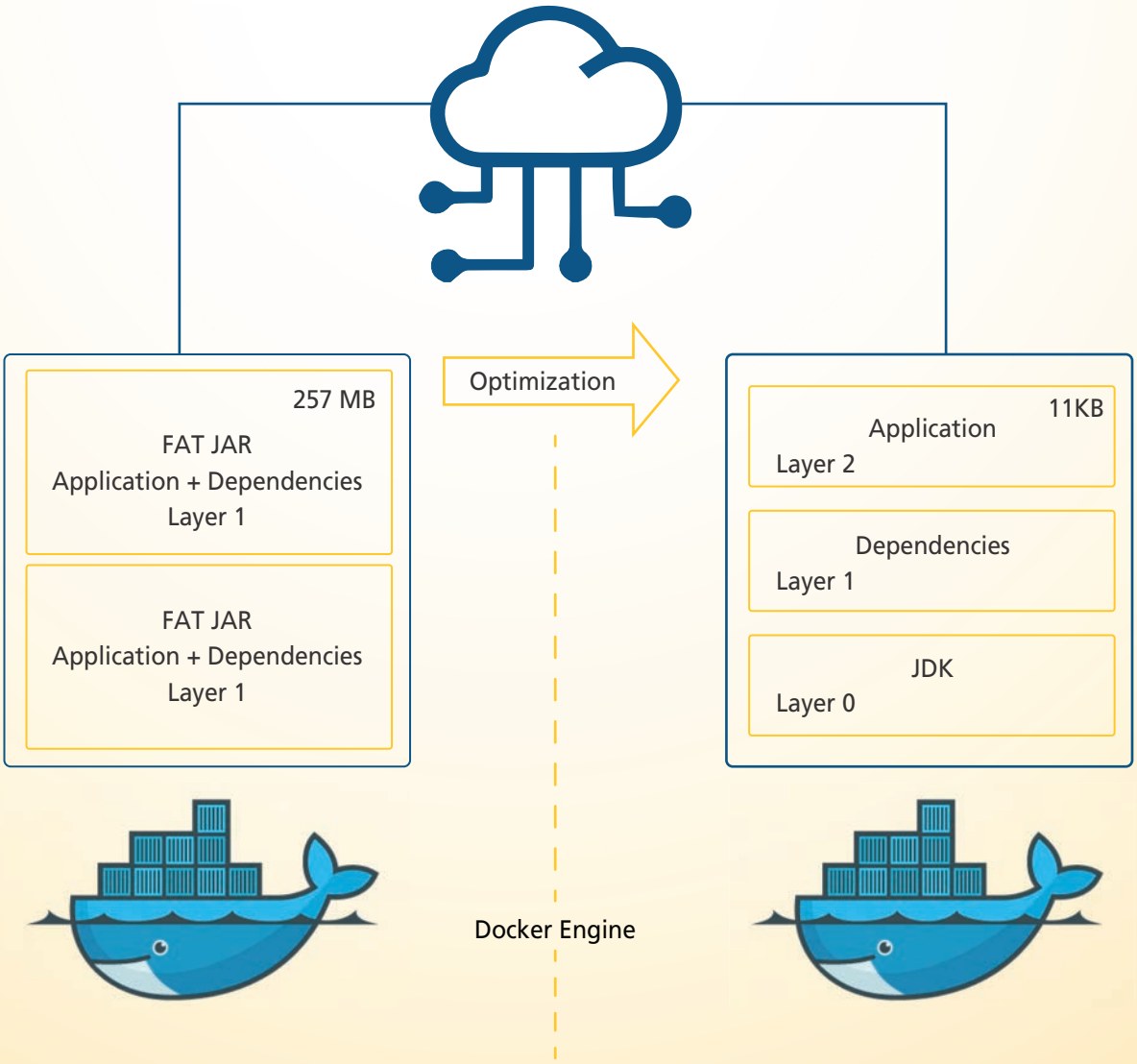
The default layers are:

Layer Name	Contents
Application	Application classes and resources
Snapshot Dependencies	Any dependency, whose version contains SNAPSHOT
Spring-Boot-Loader	JAR loader classes
Dependencies	Any dependency, whose version does not contain SNAPSHOT

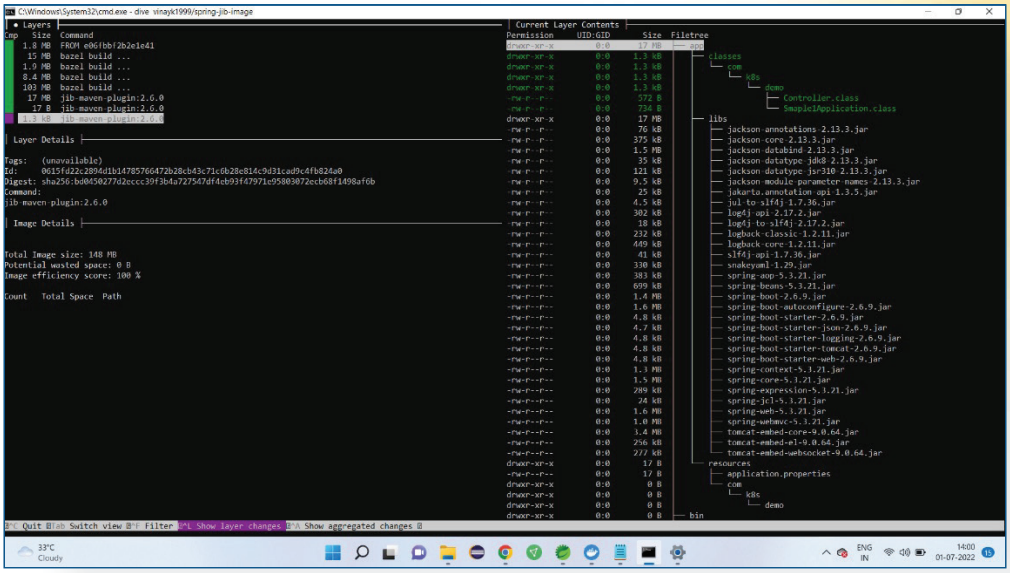
The layers are defined in a layers.idx file in the order that they should be added to the Docker image. These layers get cached in the host after the first pull since they do not change. Only the updated application layer is downloaded to the host, which is faster because of the reduced size.

Building the Image with Dependencies Extracted in Separate Layers

We will build the final image in two stages using a method called multi-stage build. In the first stage, we will extract the dependencies, and in the second stage, we will copy the extracted dependencies to the final image.



Shrinking data volume



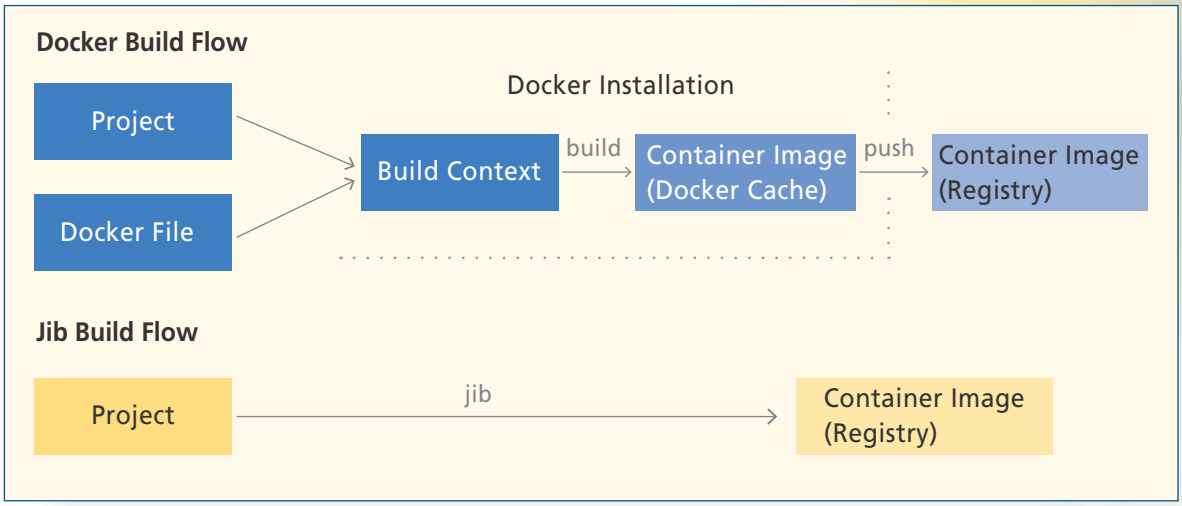
Benefits of this approach:

- Improves application performance by reducing startup time.
- Package size is reduced, thus reducing network bandwidth and cost.
- CI -Build minutes are decreased, which makes this approach cost-effective.
- Container security is addressed by JIB

2. Abstracting image creation

Google Jib builds containers without using a Docker file or requiring a Docker installation. You can use Jib in the Jib plugins for Maven (https://github.com/GoogleContainerTools/jib/tree/master/jib-maven-plugin) or Gradle (https://github.com/GoogleContainerTools/jib/tree/master/jib-gradle-plugin), or you can use the Jib Java library (https://github.com/GoogleContainerTools/jib/tree/master/jib-core).

Google Jib: Jib organizes your application into distinct layers, dependencies, resources, and classes and utilizes the Docker image layer caching to keep builds fast by only rebuilding changes. Jib's layer organization and the small base image keep the overall image size small which improves performance and portability. You don't need to know the best practices for creating Docker files or having Docker installed.



Jib is available as a plugin for Maven (https://github.com/GoogleContainerTools/jib/tree/master/jib-maven-plugin) and Gradle (https://github.com/GoogleContainerTools/jib/tree/master/jib-gradle-plugin) and requires minimal configuration. Simply add the plugin to your build definition and configure the target image. If you are building to a private registry, make sure to configure Jib with credentials for your registry. (https://github.com/GoogleContainerTools/jib/tree/master/jib-maven-plugin#authentication-methods) The easiest way to do this is to use credential helpers like docker-credential-gcr (https://github.com/GoogleContainerTools/jib/blob/master/docs/configure-gcp-credentials.md). Jib also provides additional rules for building an image to a Docker daemon if you need it.

Use Case

For this example, we'll provide our Docker Hub credentials to ***.m2/settings.xml:***

```
<servers>
    <server>
        <id>registry.hub.docker.com</id>
        <username><DockerHub Username></username>
        <password><DockerHub Password></password>
    </server>
</servers>
```

Deploying to Jfrog with Jib:

Now, we can use jib-maven-plugin, or the Gradle equivalent, to containerize our application with a simple command:

```
mvn compile com.google.cloud.tools:jib-maven-plugin:2.5.0:build -Dimage=$IMAGE_PATH
```

where IMAGE_PATH is the target path in the container registry.

For example, to upload the image spring-jib-app to Jfrog, we would do:

```
export IMAGE_PATH=registry.hub.docker.com/spring-jib-app
```

And that's it! This will build the docker image of our application and push it to the Jfrog.
We can, of course, upload the image to Google Container Registry (https://cloud.google.com/container-registry/) or Amazon Elastic Container Registry (https://aws.amazon.com/ecr/) in a similar way.

Simplifying the Maven Command

Also, we can shorten our initial command by configuring the plugin in our pom instead, like any other Maven plugin.

```
<project>
    ...
    <build>
        <plugins>
            ...
            <plugin>
                <groupId>com.google.cloud.tools</groupId>
                <artifactId>jib-maven-plugin</artifactId>
                <version>2.5.0</version>
                <configuration>
                    <to>
                        <image>${image.path}</image>
                    </to>
                </configuration>
            </plugin>
            ...
        </plugins>
    </build>
    ...
</project>
```

With this change, we can simplify our maven command:

Now, we can use jib-maven-plugin, or the Gradle equivalent, to containerize our application with a simple command:

```
mvn compile jib:build
```

When we build the application, an image is created, and it is stored in Jfrog.

Benefits of this approach:

- Optimization will **help teams share smaller images,** improve performance, and make it easier to debug problems.
- Docker caches images. If you need to create a custom base image, layers optimization, and multiple instances of the same layers, it will **speed up the load times and make it easier to track.**
- **Bypassing Docker daemon** to be installed on developer machines is a boon for companies that have restrictions to install docker on developer machines.
- JIB layers application binaries and thus deploys only layers only that change, thus **reducing network bandwidth** and cost it incurred before to download the entire fat jar and deploy.
- All of the image creation and downloading are usually done via the CI platform. This further increases the build minutes. Hence since JIB takes over and does this for us, we are **effectively reducing the build minutes,** which makes this approach **cost-effective.**

In a nutshell:

Effective use of Jib containerizing applications has proven to reduce the scheduling time of the container orchestration system. This has further resulted in reducing the application start-up time. Further reducing the application start-up time. This approach also removes manual docker file creation, which is not straightforward and can bring in adversities in application security. It also optimizes the containers by enabling the layering feature, which extracts the dependencies in separate layers that get cached in the host, and the thin layer of the application is downloaded during scheduling in container runtime engines.

Santosh Malimath

Sr Project Manager/Technical Architect

Over the last 16 years, Santosh has worked with several large enterprises. His experience spans as a Full-stack development, DevSecOps engineering and IOT solutioning. During the last 6 years he has engaged in analyzing and creating project scope and milestones for several technical company initiatives. Has been contributing to help define key partnerships to targets, establishing technical relationships, and managing the day-to-day interactions to build long-term business and marketing opportunities.