How to deploy and run Real-Time Java Application with Maven in Docker

Wasin Waeosri

.

Follow

Published in

LSEG Developer Community

.

13 min read

.

Nov 2, 2023

--

Listen

Share

The original article on the Refinitiv Developer Community is available here.

Introduction

Update: January 2022

As of December 2021: There are new serious vulnerabilities that were identified impacting the Apache Log4j utility. Please update the library to the latest version. You can find more detail regarding the vulnerability and the fix from the Apache Log4j Security Vulnerabilities page.

Docker is an open containerization platform for developing, testing, deploying, and running any software application. In Docker, the applications are presented as lightweight, portable, and self-sustaining containers which can be simultaneously run in a loosely isolated and virtual environment on a given host. Developers can use Docker to automate repetitive chores, such as setting up and configuring controlled development environments. Each environment or container has its resources that are independent of other containers. Numerous containers of separate applications are running on completely different stacks. Therefore, developers can avoid common problems, such as run-time library conflicts, and unsupported environments, and focus totally on developing software. Moreover, they can simply delete the containers without any footprints left on the host machine.

This article aims for helping Java developers who already familiar with Maven to use Docker container to build and the Real-Time Java application dynamically with Maven. This scenario is suitable for the Java developer team who already uses Maven in their project to set up a Development and Build environment via Docker.

What is Refinitiv Real-Time SDK?

The Refinitiv Real-Time SDK Family (RTSDK C++ and Java Editions, formerly known as Elektron SDK) is a suite of modern and open APIs that aim to simplify development through a strong focus on ease of use and standardized access to a broad set of Refinitiv and proprietary content and services via the proprietary TCP connection named RSSL and proprietary binary message encoding format named OMM Message. The capabilities range from low latency/high performance APIs right through to simple streaming Web APIs.

This SDK is also available on GitHub with instructions to build the libraries.

The Real-Time SDK stack contains a set of capabilities ranging from low level ‘Transport’ interfaces to very high-level content aware stateful interfaces. The SDK consists of two following APIs

1. **Enterprise Transport API (ETA)**: Formerly known as UPA, or Ultra Performance API. The ETA API is the open source, high performance, low latency, foundation of the Refinitiv Real-Time SDK. This API provides the highest level of performance, scalability, tune-ability, low memory utilization, and low CPU utilization.
2. **Enterprise Message API (EMA)**: Formerly known as Elektron Message API, this API is a data-neutral, multi-threaded, ease-of-use API providing access to Refinitiv Real-Time data. As part of the Real-Time SDK, the Enterprise Message API allows applications to consume and provide OMM data at the message level of the API stack.

You can find more detail regarding the EMA Java from the following resources

- Refinitiv Real-Time SDK Java page on the Refinitiv Developer Community web site.
- Refinitiv Real-Time SDK Family page.
- Enterprise Message API Java Quick Start
- Developer Webinar: Introduction to Enterprise App Creation With Open-Source Enterprise Message API
- Developer Article: 10 important things you need to know before you write an Enterprise Real Time application

Note: Please note that the Refinitiv Real-Time SDK isn't qualified on the Docker platform. This article and example projects aim for Development and Testing purposes only. If you find any problems while running it on the Docker platform, the issues must be replicated on bare metal machines before contacting the helpdesk support.

RTSDK Java with Maven

The Refinitiv Real-Time SDK Java is now available in Maven Central Repository. You can define the following dependency in Maven's pom.xml file to let Maven automatically download the EMA Java library and ETA Java library for the application.

For more detail regarding how to setup RTSDK Java project with Maven, please see How to Run Real-Time SDK Java Application with Maven article and GitHub Project pages

Note:

- This article is based on EMA Java version 3.6.1 L1 (RTSDK Java Edition 2.0.1 L1).

Simple EMA Java Consumer on Docker

Let's start with a simple EMA Java Consumer application that connects and consume data from Refinitiv Real-Time — Optimized (RRTO) which is Refinitiv Real-Time infrastructure on the Cloud (currently is AWS). The application *CloudConsumer* is based on the EMA Java Consumer *ex450_MP_QueryServiceDiscovery* example application.

Project Maven file

The application project is in Maven project layout and the Maven *pom.xml* file is the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd"

<modelVersion>4.0.0</modelVersion
  <groupId>com.refinitiv.ema</groupId
  <artifactId>cloud_consumer</artifactId>
  <version>1.0</version>
  <name>cloud_consumer</name>

<properties>
     <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding
     <maven.compiler.source>1.8</maven.compiler.source
     <maven.compiler.target>1.8</maven.compiler.target
     <rtsdk.version>3.6.1.0</rtsdk.version>
  </properties>

<dependencies>
     <!-- RTSDK -->
     <dependency>
        <groupId>com.refinitiv.ema</groupId
        <artifactId>ema</artifactId>
        <version>${rtsdk.version}</version>
     </dependency>
  </dependencies>

<build>
     <plugins>
        <plugin>
           <groupId>org.apache.maven.plugins</groupId
           <artifactId>maven-assembly-plugin</artifactId
           <executions>
           <execution>
           <phase>package</phase>
           <goals>
              <goal>single</goal>
```

```
          </goals>
          <configuration>
          <archive>
          <manifest>
          <mainClass>
             com.refinitiv.ema.cloud.CloudConsumer
          </mainClass>
          </manifest>
          </archive>
          <descriptorRefs>
          <descriptorRef>jar-with-dependencies</descriptorRef>
          </descriptorRefs>
          </configuration>
          </execution>
          </executions>
        </plugin>
     </plugins>
  </build>

</project>
```

The above Maven *pom.xml* file resolves the RTSDK Java library and dependencies from Maven Central, and then builds the application and the RTSDK library into a single-all-dependencies jar file named *cloud_consumer-1.0-jar-with-dependencies.jar*. Please see How to Set Up Refinitiv Real-Time SDK Java Application with Maven article about the pom.xml setting for the RTSDK Java library.

Project Docker file

The Dockerfile for building this simple application's image is the following:

```
# Maven
FROM maven:3.8.1-openjdk-11-slim AS builder
WORKDIR /app
COPY pom.xml .
RUN mvn -e -B dependency:resolve
COPY src ./src
RUN mvn clean -e -B package

# RTSDK Java
FROM openjdk:11-jre-slim-buster
WORKDIR /app
COPY --from=builder /app/target/cloud_consumer-1.0-jar-with-dependencies.jar .
COPY EmaConfig.xml .
COPY etc ./etc
COPY run.sh ./run.sh
# Use shell script to support passing application name and its arguments to the ENTRYPOINT
ENTRYPOINT [ "./run.sh" ]
```

This *Dockerfile* utilizes the best practices from Intro Guide to Dockerfile Best Practices blog post. It uses multi-stage builds to separate the Maven build process from the RTSDK Java application run process.

You may notice that the Dockerfile calls the **run.sh** script with **ENTRYPOINT** instruction instead calls **java** command directly. The reason is the CloudConsumer application requires the RRTO credentials information input via the command line arguments (**-username**, **-password**, **-clientId**, etc), so we create the **run.sh** shell script to run the cloud_consumer-1.0-jar-with-dependencies.jar file and accept the command line arguments for Docker.

The content of *run.sh* script is the following:

```
#!/bin/sh

java -jar ./cloud_consumer-1.0-jar-with-dependencies.jar "$@"
```

Update: September 2021.

You do not need the *run.sh* script to run the application in a Docker container. You can set both `ENTRYPOINT` and `CMD` instructions instead.

```
# Maven
FROM maven:3.8.1-openjdk-11-slim AS builder
WORKDIR /app
COPY pom.xml .
RUN mvn -e -B dependency:resolve
COPY src ./src
```

```
RUN mvn clean -e -B package

# RTSDK Java
FROM openjdk:11-jre-slim-buster
WORKDIR /app
COPY --from=builder /app/target/cloud_consumer-1.0-jar-with-dependencies.jar .
COPY EmaConfig.xml .
COPY etc ./etc
#COPY run.sh ./run.sh #comment the COPY command
ENTRYPOINT ["java", "-jar", "./cloud_consumer-1.0-jar-with-dependencies.jar"]
CMD ["-ric", "/EUR="]
```

Then you can pass the RRTO credentials and the application command line arguments after the `docker run` command as usual.

Advance EMA Consumer and IProvider applications on Docker

The next example is replicating the EMA Consumer and Interactive-Provider applications projects. The Consumer and Provider applications are developed and maintained in a different project, each project has its Maven dependencies that match requirements of Consumer and Provider tasks/requirements.

We will run the Consumer and Provider applications in separate Docker containers to simulate the real-life scenario that the applications run in a different machine (or even network).

Project Maven file

The IProvider application is based on EMA Java Interactive-Provider ex200_MP_Streaming example with additional logic to display application message with the Cowsay library (if you do not know what cowsay is), so we define the Cowsay dependency in Maven pom.xml too.

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd"

<modelVersion>4.0.0</modelVersion
  <groupId>com.refinitiv</groupId
  <artifactId>IProvider</artifactId>
  <version>1.0</version>
  <name>IProvider</name>

<properties>
     <maven.compiler.source>8</maven.compiler.source
     <maven.compiler.target>8</maven.compiler.target
     <rtsdk.version>3.6.1.0</rtsdk.version
     <cowsay.version>1.1.0</cowsay.version>
  </properties>

<dependencies>
     <!-- RTSDK -->
     ...
     <dependency>
        <groupId>com.github.ricksbrown</groupId
        <artifactId>cowsay</artifactId>
        <version>${cowsay.version}</version
        <classifier>lib</classifier>
     </dependency>
  </dependencies>

<!-- Build single-all-dependencies jar file named IProvider-1.0-jar-with-dependencies.jar -->
  <build>
  ...
  </build>
</project>
```

The Consumer application is based on EMA Java Consumer ex200_MP_Streaming example with additional logic to print/log application and EMA Java API messages with Apache Log4j library. Please refer to Enterprise Message API Java with Log4j article for more detail about EMA Java application and Log4j integration.

The pom.xml file for the Consumer project is the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
```

```
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion
  <groupId>com.refinitiv</groupId
  <artifactId>Consumer</artifactId>
  <version>1.0</version>
  <name>Consumer</name>

<properties>
     <maven.compiler.source>8</maven.compiler.source
     <maven.compiler.target>8</maven.compiler.target
     <rtsdk.version>3.6.1.0</rtsdk.version
     <log4j.version>2.17.1</log4j.version>
  </properties>

  <dependencies>
     <!-- RTSDK -->
     ...
     <dependency>
        <groupId>com.refinitiv.eta.valueadd</groupId
        <artifactId>etaValueAdd</artifactId
        <version>${rtsdk.version}</version>
        <exclusions>
        <exclusion>
           <groupId>org.slf4j</groupId>
           <artifactId>slf4j-jdk14</artifactId>
        </exclusion>
        </exclusions>
     </dependency>
     <!-- log4j -->
     <dependency>
        <groupId>org.apache.logging.log4j</groupId>
        <artifactId>log4j-api</artifactId>
        <version>${log4j.version}</version>
     </dependency>
     <dependency>
        <groupId>org.apache.logging.log4j</groupId>
        <artifactId>log4j-core</artifactId>
        <version>${log4j.version}</version>
     </dependency>
     <dependency>
        <groupId>org.apache.logging.log4j</groupId>
        <artifactId>log4j-slf4j-impl</artifactId>
        <version>${log4j.version}</version>
     </dependency>
  </dependencies>

<!-- Build a single-all-dependencies jar file named Consumer-1.0-jar-with-dependencies.jar  -->
  <build>
     ...
  </build>
</project>
```

Consumer Log4j configuration file

The Consumer application's *log4j2.xml* configuration file is available in the Consumer project's *resources* folder. The configurations set the Consumer application to log messages with the following logic:

- Application log message: Log messages to both console and a log file in */logs/consumer_log4j.log* location
- EMA log message: log messages to a log file in */logs/ema_log4j.log* location

So, we need to map the */log/* folder inside the containers to the host machine directory as well.

```
<?xml version="1.0" encoding="UTF-8"?>
<Configuration status="INFO">

<Appenders>
     <Console name="LogToConsole" target="SYSTEM_OUT"
```

```
        <PatternLayout pattern="%d Class name-%C Message-%m%n"/>
     </Console>
     <File name="emaLogFile" fileName="/logs/ema_log4j.log">
        <PatternLayout>
        <Pattern>%d LEVEL-%-5p Thread-[%t]  Method-%M()   Class name-%C   Message-%m%n
        </Pattern>
        </PatternLayout>
     </File>
     <File name="consumerLogFile" fileName="/logs/consumer_log4j.log">
        <PatternLayout>
        <Pattern>%d{HH:mm:ss.SSS} [%t] %-5level %logger{36} - %msg%n
        </Pattern>
        </PatternLayout>
     </File>
  </Appenders>

<Loggers>
     <!-- avoid duplicated logs with additivity=false -->
     <Logger name="com.refinitiv.ema" level="trace" additivity="false">
        <AppenderRef ref="emaLogFile"/>
     </Logger>
     <Logger name="com.refinitiv.ema.consumer" level="info" additivity="false">
     <AppenderRef ref="LogToConsole"/>
     <AppenderRef ref="consumerLogFile"/>
  </Logger>
  </Loggers>
</Configuration>
```

Projects Docker files

The *Dockerfile-provider* for building the Provider application's Image is almost identical to the CloudConsumer's Dockerfile except it called `java -jar ./IProvider-1.0-jar-with-dependencies.jar` command with the `CMD` instruction instead:

```
## Dockerfile-provider

# Maven
FROM maven:3.8.1-openjdk-11-slim AS builder
WORKDIR /app
COPY pom.xml .
RUN mvn -e -B dependency:resolve
COPY src ./src
RUN mvn clean -e -B package

# RTSDK Java
FROM openjdk:11-jre-slim-buster
WORKDIR /app
COPY --from=builder /app/target/IProvider-1.0-jar-with-dependencies.jar .
COPY EmaConfig.xml .
COPY etc ./etc
# run IProvider-1.0-jar-with-dependencies.jar with CMD
CMD ["java", "-jar", "./IProvider-1.0-jar-with-dependencies.jar"]
```

The *Dockerfile-consumer* for building the Consumer application's Image adds the *resource* folder with *log4j2.xml* Log4j configuration file into the container for Log4j setting during runtime:

```
## Dockerfile-consumer

# Maven
FROM maven:3.8.1-openjdk-11-slim AS builder
WORKDIR /app
COPY pom.xml .
RUN mvn -e -B dependency:resolve
COPY src ./src
RUN mvn clean -e -B package

# RTSDK Java
FROM openjdk:11-jre-slim-buster
WORKDIR /app
COPY --from=builder /app/target/Consumer-1.0-jar-with-dependencies.jar .
COPY EmaConfig.xml .
```

```
COPY etc ./etc
COPY resources ./resources

# run Consumer-1.0-jar-with-dependencies.jar with CMD
CMD ["java", "-jar","-Dlog4j.configurationFile=./resources/log4j2.xml", "./Consumer-1.0-jar-with-dependencies.jar"]
```

Since we need to run both Provider and Consumer Containers simultaneously, so we use the Docker-Compose tool to build/run multi-container Docker applications. We define the containers configurations in the docker-compose.yml (version 3) configuration file. Please note that the file is in YAML file format.

The consumer service for Consumer contains is mapped the current *log* folder to the container working directory's */logs* folder with the volumes configuration to store the EMA and Consumer application log files in the Host machine.

```
version: "3.9"
services:
   provider:
       image: developers/provider
       build:
           context: ./IProvider
           dockerfile: Dockerfile-provider
   consumer:
       image: developers/consumer
       build:
          context: ./Consumer
          dockerfile: Dockerfile-consumer
      depends_on:
          - provider
       volumes:
            - ./Consumer/logs:/logs
```

Demo prerequisite

This example requires the following dependencies software and libraries.

1. Oracle/Open JDK 8 or Oracle JDK 11.
2. Apache Maven project management and comprehension tool.
3. Internet connection.
4. Docker Desktop/Engine version 20.10.x and Docker-Compose version 1.29.x
5. Access to the Refinitiv Refinitiv Data Platform and Refinitiv Real-Time — Optimized. (for the *CloudConsumer.java* example only)

Please contact your Refinitiv's representative to help you to access the Refinitiv Real-Time Distribution System, or RDP account, and services. You can follow the step-by-step guide to complete your RDP credentials setup from the *Getting Started for Machine ID* section of the Getting Start with Refinitiv Data Platform article.

Note: The RTSDK Java version 2.0.1 L1 (EMA Java 3.6.1) supports Oracle JDK versions 8, 11, and Open JDK version 8. If you are using other versions of RTSDK Java, please check the SDK's README.md file regarding the supported Java version.

Running the demo applications

Please unzip or download the example application from the GitHub repository into a directory of your choosing. The Docker Desktop/Engine application should be installed and run properly on your machine. For Windows 10, please refer to this Install Docker Desktop on Windows page.

Running the CloudConsumer Example

Please note that since RTSDK 1.5.1 (EMA and ETA Java API version 3.5.1), the SDK does not require the Keystore file (.jsk) to connect to the Refinitiv Real-Time — Optimized HTTPS/ENCRYPTED connection anymore. However, if you need to use the Keystore file, please see the step-by-step guide from the Building a Keystore file to be used with an HTTPS (or ENCRYPTED) connection type for real-time Java-based APIs article first and store it in the Host directory, then pass it to Docker with *-v* Docker run parameter.

Firstly, open the project folder in the command prompt and go to the *simpleCloudConsumer* subfolder. Next, run the Docker build command to build the Docker Image name *developers/cloudconsumer*:

```
$>simpleCloudConsumer> docker build . -t developers/cloudconsumer
```

Then Docker will start building the application image. Please note that this process may take time for the first build and run. Once Docker builds success, you can use `docker images` command to check the image detail.

To start and run the cloudconsumer container, run the following Docker run command in a command prompt.

```
$>simpleCloudConsumer> docker run --name <Container Name> -it developers/cloudconsumer -username <machine-id> -password <passw
```

The result is the following:

Running the IProvider and Consumer examples

Firstly, open the project folder in the command prompt and go to the *advanceExamples* subfolder. Then we run the following Docker-Compose command to build and run all applications containers:

```
$>advanceExamples> docker-compose up
```

Then Docker will start building the application's images, please note that this process may take time for the first build and run.

Then Docker will run Provider and Consumer containers, the Consumer container will connect and consume real-time streaming data from the Provider container.

The log files are generated by Log4j in the containers which are mapped to the Host folder too.

If you use `docker images` command, Docker will show that it already creates two images for you.

For other Docker-Compose commands which can interact with these containers/images (such as `docker-compose build`), please see the Docker-Compose CLI reference page.

Conclusion

Docker is an open containerization platform for developing, testing, deploying, and running any software application. The combination of Docker and Maven provides a consistent development environment for Java developers and the team. The developer does not need to manually maintain jar file dependencies, the OS, and toolsets for the project. The Docker also help building, testing, deployment, and packaging on various environment easier than on the physical or virtual machine because the container already contains its configurations and dependencies.

The RTSDK Java developers can fully gain benefits from both Docker and Maven. The SDK is now available in Maven central repository and the refinitivapis/realtimesdk_java Docker Hub repository. This project helps Java developers integrate the Maven development environment with the Docker container to simplify the real-time development process for both simple and advanced use cases.

If you interested in other Refinitiv APIs and Docker integration, please see the following resources:

- Deploy and Run Refinitiv Real-Time SDK in Docker article.
- Introduction to the refinitivapis/realtimesdk_java Docker Image article.
- Introduction to the refinitivapis/realtimesdk_c Docker Image article.
- Introduction to the refinitivrealtime/adspop docker image article
- refinitivapis/websocket_api Repository.

References

For further details, please check out the following resources:

- Refinitiv Real-Time SDK Java page on the Refinitiv Developer Community website.
- Refinitiv Real-Time SDK Family page.
- Enterprise Message API Java Quick Start
- Developer Webinar: Introduction to Enterprise App Creation With Open-Source Enterprise Message API
- Developer Article: 10 important things you need to know before you write an Enterprise Real Time application
- How to Set Up Refinitiv Real-Time SDK Java Application with Maven
- Introduction to the refinitivapis/realtimesdk_java Docker Image article.
- Deploy and Run Refinitiv Real-Time SDK in Docker article.
- Building a Keystore file to be used with an HTTPS (or ENCRYPTED) connection type for real-time Java-based APIs article.
- Intro Guide to Dockerfile Best Practices.
- Get Started with Docker page.
- Get started with Docker Compose page.

For any questions related to this article or the RTSDK page, please use the Developer Community Q&A Forum.