KIP-975: Docker Image for Apache Kafka

Status

Current state: *Accepted*

Discussion thread: *here*

Voting thread: *here*

JIRA: KAFKA-15445

Please keep the discussion on the mailing list rather than commenting on the wiki (wiki discussions get unwieldy fast).

Motivation

Docker is a widely adopted tool among developers for its ease of deployment. Popular Apache projects like Hadoop, Spark, and Flink publish a docker image. However, Apache Kafka does not have an official docker image currently. This KIP aims to publish Apache Kafka docker image through the ASF docker profile.

This KIP targets to publish docker image for JVM based Apache Kafka.

POC

We did an initial POC of our image and compared with other existing popular Apache Kafka docker images

Image Name	Image size (Uncompressed)	Startup total time (in seconds)	Time to start the kafka server inside container (in seconds)	Memory Used	Java version	Apache Kafka version
Proposed apache/kafka with Java 21	457.86 MB	3.885	0.816	382.4 MB	21	3.6.0
Proposed apache/kafka with Java 17	460.76 MB	2.174	0.679	326 MB	17	3.5.1
bitnami/kafka	517.16 MB	3.469	0.889	344 MB	17	3.5.1
confluentinc/cp-kafka	876.4 MB	3.341	0.701	390.6 MB	11	7.5.0-ccs
ubuntu/kafka + ubuntu/zookeeper (Image doesn't support kraft and source code is not public)	359.03 MB + 359.03 MB	1.425 + 0.636	0.827	335 MB + 78 MB	11	3.1.0

Note: ubuntu/kafka is smaller as it is making use of Apache Kafka:3.1.0 and Java 11. With same AK and java versions, our image size was 367MB

Public Interfaces

- There will be a new additional artifact i.e. Apache Kafka docker image for every Apache Kafka release.
- Sample `docker-compose.yml`

- Quick start examples
- Add public documentation

Proposed Changes

- There will be a docker image as an additional artifact for every Apache Kafka release.
- This docker image will consist of JVM based Kafka and will have support for Linux based AMD and ARM architectures.
- Add sanity tests to run against the new docker image for each tag.
- Add release script to extend the Apache release process to publish a new docker image to public DockerHub.

Scala Version

Currently Kafka has support for two flavours of binaries based on Scala version: 2.12 and Scala 2.13.
We will support **Docker Image only for Scala 2.13**

- Less maintenance overhead.
- Recommended by Apache Kafka.
- Should anyone require a Docker image with Scala version 2.12, it can be conveniently generated by making a minor adjustment to the Apache Kafka tar file specified in the provided Dockerfiles. This modification will align the image with the 2.12 version of Apache Kafka.

JAVA Version

- We will support only the latest LTS JAVA supported by Apache Kafka.
- Apache Kafka is already being built and tested with JAVA 21.
- By release 3.7.0 (the next release), Apache Kafka will have JAVA 21 as one of the officially supported versions.
- Hence, **we will support JAVA 21** in our docker image.

What if users want Docker Image with a different Java version?

- For users seeking a Docker image with an alternative Java version, they will have the flexibility to build their own Docker image utilising the Dockerfiles we provide. In our documentation, we will provide clear guidance on the designated base images for various Java versions.
- We will update the Java major version as part of minor Apache Kafka releases. The implication is that users who include broker plugins alongside the broker should use custom images to ensure their custom code is not broken by Java upgrades.

Docker Base Image

To run Apache Kafka, only JRE, and not JDK, is needed in the Docker image.
We have decided to utilise the **`eclipse-temurin:<JRE-version>`** as the base image for docker image.

- This is one of the best images for JAVA JRE which is maintained by a reputed organisation.
- Flink relies on eclipse-temurin base images, affirming their authenticity and increasing our confidence in their reliability.
- This image has support for arm64 and amd64 both.
- Since it's just JRE it is lightweight in terms of image size of 263MB.

Image Naming

Image naming should transparently communicate the packaged Kafka version.

Adhering to the outlined constraints, image tagging can follow this format
<image-name>:<kafka-version>

- Name of the image: **kafka**
- For example, for 3.7.0 version of kafka, the image name with tagging would be **`apache/kafka:3.7.0`**

Directory Structure

A new directory named `docker` will be added to the repository. This directory will contain all the Docker related code.
Directory Structure:

```
kafka/
        - docker/
                - jvm/
                        - Dockerfile          #Dockerfile for the JVM-based Apache Kafka Docker image.
                - resources/              #Contains resources needed to create the Docker image.
                - test/                   #Contains sanity tests for the Docker image.
                - docker_build_test.py    #Python script for building and testing the Docker image.
                - docker_release.py       #Python script for building the Docker image and pushing it to Docker
Hub.
```

NOTE: This structure is designed with the anticipation of introducing another Docker image based on the native Apache Kafka Broker (as per KIP-974). Both images will share the same resources for image building.

Configuring Properties

We offer two methods for passing the above properties to the container:

1. **File Mounting**: Users can mount a properties file to a specific path within the container (we will clearly document this path). This file will then be utilized to start up Kafka.
2. **Using Environment Variables**: Alternatively, users have the option to provide configurations via environment variables. Here's how to structure these variables:

    - Replace **.** with _
    - Replace _ with __`(double underscore)`
    - Replace **-** with ___`(triple underscore)`
    - Prefix the result with **`KAFKA_`**

    Examples:

    - For **`abc.def`**, use **`KAFKA_ABC_DEF`**
    - For **`abc-def`**, use **`KAFKA_ABC___DEF`**
    - For **`abc_def`**, use **`KAFKA_ABC__DEF`**

This way, you have flexibility in how you pass configurations to the container, making it more adaptable to various user preferences and requirements.
NOTE:

1. Secrets will be provided to the container using folder mount.
2. If a property is provided both in the mounted file and as an environment variable, the value from the environment variable will take precedence.

Code changes

- We are introducing a Docker wrapper into the Kafka codebase. This wrapper will encapsulate the logic required to set up property files, adhering to the rules outlined in the previous section.
- This Docker wrapper will be supported by bash scripts, which will be situated within the Docker container. These scripts will serve as the entry point for the Docker image, streamlining the process.
- Previously, we utilized Golang for the setup of the property files. However, once the Docker wrapper changes are integrated, the Golang code will be removed. This change will eliminate the need for maintaining support for an additional language in the codebase, reducing overhead.

NOTE:

1. Having the logic to configure property files in the kafka codebase, means that there will be a dependency on the kafka binary tarball that is containerised inside the docker image.
2. Hence, previous versions of kafka which do not contain this Docker wrapper will not be supported for Docker image generation.

Class Data Sharing(CDS)

We did a small POC with class data sharing (CDS) feature that can help reduce the startup time and memory footprints for Java applications.

```
System details: M2 Mac, 32GB memory.

JSA file created for bringing up Kafka (default Kraft configs): ~45MB
Local Kafka startup time (without JSA): 1.592 secs
Local Kafka startup time (with JSA): 1.016 secs
Local Kafka startup memory usage (without JSA): 440MB
Local Kafka startup memory usage (with JSA): 380MB

NOTE: The jsa files were created only by covering the Kafka Startup code path and resulted in ~45MB of size.
```

We can see a ~**30% reduction** in the startup time.

Following are the points to consider:

1. Docker Image Size**:** Adding the *jsa* files will increase the size of the Docker image. Classes stored in the CDS are a few times (e.g., 2 – 5x) larger than classes stored in JAR files
2. Choosing Application Usage for JSA: For our POC, we focused on Kafka startup using default configurations. It's crucial to decide how broadly the application should be used to generate *jsa* files.
3. Startup Memory Footprint: While we did observe a reduction in startup memory footprint, we believe this difference may not significantly scale with production load.
4. CDS Archive Exclusions: The CDS archive doesn't include pre-JDK 6 classes, which are present in our codebase.
5. Limitations of CDS:
    a. The OpenJDK version used for building the *jsa* files and during runtime must be the same. If they are not, application wont fail but the expected optimisation won't be achieved.
    b. If a custom JAR is provided by the user, it's currently prepended to the existing classpath. However, CDS requires that the classpath used during *jsa* file generation should either be the same or a prefix of the classpath used during runtime. Otherwise, it will be disabled, and the expected optimisation won't be achieved.

Given the notable reduction in the startup time, we've made the decision to incorporate the CDS in the Docker image.
The *jsa* file will be generated dynamically as a docker layer using the following workflow in the kraft mode:

1. Start Apache Kafka
2. Create a topic, produce messages, and consume messages
3. Stop Kafka

Compatibility, Deprecation, and Migration Plan

- For existing apache kafka users there will be no impact as JVM based docker image will be a new feature.

Test Plan

- **Testing the functionality of the Apache Kafka packaged in the image**
    - The image will consist of the official tarball released by Apache Kafka.
    - The above tarball is pre tested as the part of Apache Kafka release.
    - Hence no extra testing is required for the Apache Kafka packaged in the image.
- **Testing the Docker Image - Integration of the Apache Kafka with the Docker**
    - Dockerizing Apache Kafka requires additional steps like, passing the configs from the user to the properties file in the container, passing credentials etc.
    - Sanity tests will be added to test the proper functionality of the docker image.

Build, Test and Scanning Pipeline

Build and Test

Prior to release, the Docker images must undergo building, testing, and vulnerability scanning. To streamline this process, we'll be setting up a GitHub Actions workflow. This workflow will generate two reports: one for test results and another for scanning results. These reports will be available for community review before voting.

Scanning Previously Released Images

We intend to setup a nightly cron job using GitHub Actions and leverage an open-source vulnerability scanning tool like trivy (https://github.com/aquasecurity/trivy), to get vulnerability reports on all supported images. This tool offers a straightforward way to integrate vulnerability checks directly into our GitHub Actions workflow.

Release Process

Following is the plan to release the Docker image:

1. RM would have generated and pushed Apache Kafka's Release Candidate artifacts to apache sftp server hosted in blocked URLhome.apache.org by release.py script
2. Run the automation to build the docker image(**using the above Release Candidate tarball URL**) and test the image.
3. The docker image needs to be pushed to some Dockerhub repo(eg. Release Manager's) for the evaluation of RC Docker image.
4. Start the Voting for RC, which will include the Docker image as well as docker sanity tests report.
5. In case any docker image specific issue is detected, that will be evaluated by the community, if it’s a release blocker or not.
6. Once the vote passes, the image will be pushed to `apache/kafka` with the version as tag.
7. Steps for the Docker image release will be included in the Release Process doc of Apache Kafka

8. eg. for AK release 3.7.0 and image released will be `apache/kafka:3.7.0` (=> image contains AK 3.7.0)

What if we observe a bug or a critical CVE in the released Apache Kafka Docker Image?

We should consider the software in the docker image in the same way as consider third party dependencies today - if there is a high severity CVE affecting any of them, we aim to release a new version of Kafka for the affected branch. It would include the latest Kafka code from the branch.

Ownership of the Docker Images' Release

- **Suggestion:** The docker image release should be owned by the Release Manager.
- As per the current release process, only PMC members should be allowed to push to apache/kafka docker image.
- If the RM is not a PMC member, they'll need to take help from a PMC member to release the image.

Rejected Alternatives

Docker Image release post AK Release(eg CVE in the base image)

Apart form the aforementioned approach we also considered the following approach

Docker Image release during CVEs
Only Docker Image Release

1. This step will be followed in case only Docker Image need to be released(eg CVE in the base image).
2. Execute the script to build the docker image(**using the already publicly released AK tarball URL**) and test the image locally.
3. Once the Docker image artifact is ready, it will get reviewed by the community and voting will be conducted, just for the Docker image release.
4. This image will then be pushed to `apache/kafka` with proper tagging to communicate kafka version.
5. No AK release and image released will be `apache/kafka:3.7.0-1` (=> image contains AK 3.7.0)