Alpaquita Linux
Modifying images without a rebuild

Alpaquita Linux
Revision 1.0
June 2024

bellsoft

BellSoft software contains open source software. Additional information about third party code is available at https://bell-sw.com/third_party_licenses. You can also get more information on how to get a copy of source code by contacting info@bell-sw.com.

The use of any software product referenced in this document is governed by the applicable license agreement, which is not modified in any way by the terms of this notice.

Contents

1. Overview

BellSoft maintains a number of Alpaquita Linux container images. The full list of images is also available in the Alpaquita Linux: Getting started with images document.

This document describes several ways of customizing the existing Alpaquita images for building and running Java applications. As an example, we use a Spring Boot application.

All the examples below are performed on a Linux system with Docker installed.

2. Prerequisites

If you want to use a Spring Boot application to test image modification as in this document, make sure your system complies with the following prerequisites before proceeding to the next chapters.

Go to the spring initializer page and perform the following:

1. Specify the following values:
    - **Project**: Gradle - Groovy
    - **Language**: Java
    - **Spring Boot**: 3.0.6
    - **Packaging**: Jar
    - **Java**: 17
    - Leave all other fields with default values.
2. Click **ADD DEPENDENCIES** and select two dependencies:
    - Spring Web
    - GraalVM Native Support
3. Click GENERATE to create the .zip file.
4. If the created .zip file is not downloaded automatically after it was generated, download the `demo.zip`.

We will be creating some files during the course of this article. We recommend keeping the files in one place, separately from other parts of your file system. Create a separate directory, for example, `customizing-images` and unpack `demo.zip` to the created directory. All the next steps are performed inside the `customizing-images` directory.

Update the `DemoApplication` class in `demo/src/main/java/com/example/demo/DemoApplication.java` to include the `home` method serving request to the `/` path as follows:

```
package com.example.demo;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.web.bind.annotation.RequestMapping;
```

```
import org.springframework.web.bind.annotation.RestController;

@RestController
@SpringBootApplication
public class DemoApplication {
    @RequestMapping("/")
    String home() {
        return "Hello from the web!";
    }
    public static void main(String[] args) {
        SpringApplication.run(DemoApplication.class, args);
    }
}
```

3. Building with JDK, running with JRE

JDK provides all the tools for compiling Java applications. It also serves as a runtime environment for applications requiring the full JDK JVM. JRE is designed for running compiled Java applications.

Each image comes in 2 types: regular and slim. The slim version does not include Alpaquita Linux package management software (`apk-tools`) and has a single image layer. These features make the slim image smaller, more secure (with a reduced attack surface), and a good candidate to be used as the `final' image for your applications.

Our application does not require the full JDK to run. We only need JDK tools to compile it, but you can use a JRE JVM to run it. Therefore, we can use the Docker multi-stage builds approach and build our application with a JDK image and copy it to a JRE image. Perform the following steps:

1. Create a `Dockerfile` file with the following content:

   ```
   # Build our application with a JDK
   FROM bellsoft/liberica-runtime-container:jdk-17-musl AS builder
   WORKDIR /build
   COPY demo .
   RUN ./gradlew clean bootJar

   # But run it with a JRE
   FROM bellsoft/liberica-runtime-container:jre-17-slim-musl
   WORKDIR /app
   COPY --from=builder /build/build/libs/*.jar application.jar
   EXPOSE 8080/tcp
   CMD ["java", "-jar", "application.jar"]
   ```

2. Build the image as follows:

   ```
   docker build -t jdk-build-jre-run .
   ```

3. To verify the image, start a new container based on the newly created image.

   ```
   docker run -rm -p 8080:8080 jdk-build-jre-run
   ```

4. After the container starts, open the http://127.0.0.1:8080 address in a web browser.

4. Modifying images with buildah

In the previous steps, we created an image with our application. Let's assume that some time later we decide that our application must work over HTTPS instead of HTTP as it does now. Should we change the Java code and execute the build procedure again? Actually, no. Fortunately, Spring applications have a feature called "externalized configuration". This feature helps us modify the behavior of the application by setting some properties in a config file, which is not a part of the application code, and making the application read this file.

To enable HTTPS for the application in the `jdk-build-jre-run` image, you only need to create a derivative image with a config file, SSL certificates, and a modified `CMD`.

For our test purposes, it is enough to use a "self-signed" certificate, that is, a certificate not signed by a trusted certificate authority. Let's create it with the `openssl` tool using the following command:

```
openssl req -x509 -newkey rsa:4096 -keyout key.pem -out certificate.pem -sha256 -days 365 -nodes -subj '/CN=localhost'
```

`certificate.pem` contains the certificate, and `key.pem` contains its private key.

Now it is the time to create a new image with necessary customizations.

Besides Docker, there are other tools for working with images. Each has its pros and cons. In this article, we use buildah. Buildah supports Dockerfiles. Also, you can use it to iteratively modify the image by calling `buildah` with certain arguments. This feature may help in situations when your image creation logic is hard (or even impossible) to implement in a Dockerfile, because you can describe the entire creation procedure in a shell script or even in a program in another language. Follow the steps below.

1. Install buildah by following the official installation guide.
2. Buildah keeps container images separately from the Docker daemon. To get access to our `jdk-build-jre-run` image, pull it from the local Docker daemon to the buildah storage location.

   ```
   buildah pull docker-daemon:jdk-build-jre-run:latest
   ```

3. Create a working container from the image.

   ```
   newcontainer=$(buildah from jdk-build-jre-run)
   ```

4. Mount its root file system.

   ```
   export newcontainer
   buildah unshare
   mntpoint=$(buildah mount $newcontainer)
   ```

5. Now `$mntpoint` points to the root file system of our working container. For example, `$mntpoint/app` contains our `application.jar`.

6. Copy certificate and key files to `/app` in the container.

   ```
   cp certificate.pem key.pem $mntpoint/app/
   ```

7. Create `$mntpoint/app/application.yaml` configuration file for our application with the following content:

   ```
   server:
     ssl:
       certificate: "/app/certificate.pem"
       certificate-private-key: "/app/key.pem"
   ```

8. We do not need to modify the file system anymore, so let's unmount it.

   ```
   buildah umount $newcontainer
   exit # to exit from buildah unshare session
   ```

9. Update the image `CMD` to make the application read the config from `/app/application.yaml`.

   ```
   buildah config --cmd='["java", "-jar", "application.jar", "--
   spring.config.location=/app/application.yaml" ]' $newcontainer
   ```

10. Commit the working container as a new image - `jdk-jre-run-with-ssl`.

    ```
    buildah commit $newcontainer jdk-jre-run-with-ssl
    ```

11. The new image is available only in the buildah storage. You can verify it by running the following command:

    ```
    buildah images
    ```

12. To make it available for `docker` commands, push it to the local Docker daemon.

    ```
    buildah push localhost/jdk-jre-run-with-ssl:latest docker-daemon:jdk-jre-
    run-with-ssl:latest
    ```

13. To verify that the new image is indeed works over HTTPS, start it with Docker.

    ```
    docker run -it --rm -p 8080:8080 jdk-jre-run-with-ssl
    ```

Once the image is running, open https://127.0.0.1:8080 in a browser. After accepting the certificate warning, you can see the page generated by the application.

5. Running native applications with base images

Liberica Native Image Kit, one of BellSoft products, can transform a Java application to a standalone execution binary. The binary version of the program does not require a JVM and has almost instant startup time. Alpaquita Linux images with Liberica Native Image Kit are available in the Docker Hub repository.

Even if you have an execution binary, you still need an image to run it in. Images in the **alpaquita-linux-base** repository are small and do not include any unnecessary software as they provide Alpaquita Linux images with the minimum amount of packages.

We can transform our application to an execution binary and run it with a base image. A `Dockerfile` for this scenario may look like the following:

```
# Build our application with Liberica Native Image Kit
FROM bellsoft/liberica-native-image-kit-container:jdk-17-nik-22-musl AS builder
WORKDIR /build
COPY demo .
RUN ./gradlew clean nativeCompile

# Create a new image based on the base image and the binary
FROM bellsoft/alpaquita-linux-base:stream-musl
WORKDIR /app
COPY --from=builder /build/build/native/nativeCompile/demo .
EXPOSE 8080/tcp
RUN apk add --no-cache zlib && \
        apk del --no-cache apk-tools
CMD ["./demo"]
```

Binaries generated by Liberica Native Image Kit have a dependency on zlib shared libraries. Make sure that this package is installed in the runtime environment.

```
apk add --no-cache zlib
```

The following optional command removes tools to add or delete packages in Alpaquita Linux:

```
apk del --no-cache apk-tools
```

You can remove the tools if you want to prevent accidental modification of packages installed in this or any derivative image.

Building and verification of the generated image is identical to the previous approach.

This document provides an overview of customizing Alpaquita containers for Java applications. However, apart from Java, our repositories contain images for Python and C/C++ applications. The approaches described in this document can be applied to them as well.

Alpaquita Linux
Modifying images without a rebuild

bellsoft