Docker Integration Testing

jim stafford

Fall 2024 v2024-01-13: Built: 2024-11-19 21:39 EST

Table of Contents

Chapter 1. Introduction

We have just seen how we can package our application within a Docker image and run the image in a container. It is highly likely that the applications you develop will be deployed in production within a Docker container and you will want to test the overall packaging. Before we add any external resource dependencies, I want to cover how we can automate the build and execution to perform a heavy-weight Maven Failsafe integration test against this image versus the Spring Boot executable JAR alone. The purpose of this could be to automate a test of how you are defining and building your Docker image with the application — as close as deployment as possible.

One unique aspect at the end of this lecture is coverage of developing a test to operate within a Docker-based CI/CD environment. Running a test within your native environment where localhost (relative to the IT test) is also the Docker host (where containers are running) is one thing. However, CI/CD test environments commonly run builds within Docker containers where localhost is not the Docker host. We must be aware of and account for that.

1.1. Goals

You will learn:

- to automate the build of a Docker image within a module
- to implement a heavyweight Maven Failsafe integration test of that Docker image
- to implement a Docker integration test that can run concurrently with other tests of the same module on the same Docker host
- to implement a Docker integration test that can run outside of and within a Docker-based CI/CD environment

1.2. Objectives

At the conclusion of this lecture and related exercises, you will be able to:

1. configure a Maven pom to automatically build a Docker image using a Dockerfile
2. configure a Maven pom to automatically manage the start and stop of that Docker image during a Maven integration test
3. configure a Maven pom to uniquely allocate and/or name resources so that concurrent tests can be run on the same Docker host
4. identify the hostname of the Docker host running the Docker containers
5. configure a Docker container and IT test to communicate with a variable Docker host

Chapter 2. Disable Spring Boot Plugin Start/Stop

In a previous lecture, we introduced the Maven `it` profile and how to structure it so that it would activate the Maven integration test infrastructure we needed to test a standalone Spring Boot application with a dynamically assigned server port#. In this lecture, we are using a Docker image versus a Spring Boot executable JAR. Everything else is still the same, so we would like to make use of the Maven `it` profile but turn off the start/stop of the Spring Boot executable — and start/stop our Docker container instead.

it Profile Activation Trigger

```
src/test/resources
|-- application-it.properties  ①
...
```

① triggers activation of `it` profile

We will need to disable the start/stop of the native Spring Boot executable JAR, because we cannot have that process and our Docker container allocating the same port number for the test. To disable the start/ stop of the Spring Boot executable JAR and allow the Docker container to be the target of the test, we can add a few properties to cause the plugin to "skip" those goals.

Disable Spring Boot Plugin run/stop

```
<properties>
    <!-- turn off launch of unwrapped Spring Boot server during integration-test;
using Docker instead -->
    <spring-boot.run.skip>true</spring-boot.run.skip> ①
    <spring-boot.stop.skip>true</spring-boot.stop.skip> ②
```

① property triggers Spring Boot Maven plugin to skip the start goal and leave the `${server.http.port}` unallocated

② property triggers Spring Boot Maven plugin to skip the unnecessary stop goal

If we stopped there, the `${server.http.port}` port would still be identified and our IT test would be executed by failsafe and fail because we have no server yet listening on the test port.

IT Connection Error

```
[ERROR] Errors:
[ERROR]   DockerHelloIT.can_authenticate_with_server:59 » ResourceAccess I/O error on
GET request for "http://localhost:59416/api/hello": Connection refused
[ERROR]   DockerHelloIT.can_contact_server:46 » ResourceAccess I/O error on GET
request for "http://localhost:59416/api/hello": Connection refused
```

Lets work on building and managing the execution of the Docker image.

Chapter 3. Docker Maven Plugin

A google search will come up with several Maven plugins designed to manage the building and execution of a Docker image. However, many of them are end-of-life and no longer maintained. I found one (io.fabric8:docker-maven-plugin) in 2024 that has current activity and the features we need to accomplish our goals.

The following snippet shows the outer shell of the Docker Maven Plugin. We have 2 to 3 ways to fill in the configuration details: XML, properties, and a combination of both. Both will require defining portions of the `image` element of `configuration.images`.

io.fabric8:docker-maven-plugin Outer Shell

```
<properties>
...
</properties>

<build>
    <plugins>
        <plugin>
            <groupId>io.fabric8</groupId>
            <artifactId>docker-maven-plugin</artifactId>
            <configuration> ①
                <images>
                    <image>
                        ...
                    </image>
                </images>
            </configuration>
            <executions> ②
                ...
            </executions>
        </plugin>
...
```

① defines how to build the image and run the image's container

② defines which plugin goals will be applied to the build

3.1. Executions

We can first enable the goals we need for the Docker Maven Plugin. The `start` and `stop` goals automatically associate themselves with the `pre/post-integration-test` phases. I have assigned the building of the Docker image to the `package` build phase, which fires just before `pre-integration-test`.

Enable Docker Maven Plugin Goals

```
<build>
    <plugins>
```

```
        <plugin>
            <groupId>io.fabric8</groupId>
            <artifactId>docker-maven-plugin</artifactId>
                ...
                <executions>
                    <execution>
                        <id>build-image</id>
                        <phase>package</phase> ①
                        <goals>
                            <goal>build</goal>
                        </goals>
                    </execution>
                    <execution>
                        <id>start-container</id>
                        <goals> ②
                            <goal>start</goal>
                        </goals>
                    </execution>
                    <execution>
                        <id>stop-container</id>
                        <goals> ③
                            <goal>stop</goal>
                        </goals>
                    </execution>
```

① `build` goal must be manually bound to the `package` phase

② `start` goal is automatically bound to the `pre-integration` phase

③ `stop` goal is automatically bound to the `post-integration` phase

If we stopped here, we would have the Docker Maven Plugin activating the `build`, `start`, and `stop` goals when we need them, but without any configuration — they will do nothing but activate when we configured them to.

Docker Maven Plugin Goals Activating

```
[INFO] --- docker-maven-plugin:0.43.4:build (build-image) @ docker-hello-example ---
...
[INFO] --- docker-maven-plugin:0.43.4:start (start-container) @ docker-hello-example
---
...
[INFO] --- docker-maven-plugin:0.43.4:stop (stop-container) @ docker-hello-example ---
```

3.2. Configuring with Properties

The Docker Maven Plugin XML configuration approach is more expressive, but the properties approach is more concise. I will be using the properties approach to take advantage of the more concise definition. We can use the plugin-default property names that start with `docker`. However, I want to highlight which properties are for my docker plugin configuration and will use a `my.docker`

prefix as show in the snippet below.

Configure Use of Property Overrides

```
<properties>
    <!-- configure the fabric8:docker plugin using properties-->
     <my.docker.verbose>true</my.docker.verbose> ②

...
<build>
    <plugins>
        <plugin>
            <groupId>io.fabric8</groupId>
            <artifactId>docker-maven-plugin</artifactId>
            <configuration>
                <images>
                    <image>
                        <external>
                            <type>properties</type> ①
                            <prefix>my.docker</prefix>
                            <mode>override</mode>
                        </external>
```

① plugin uses default property prefix `docker`. Overriding with custom `my.docker` prefix

② supplying an example `my.docker` property override

3.3. Configuration Properties

The first few properties are pretty straight forward:

(my.docker.)dockerFile

the path of the Dockerfile used to build the image. One can alternately define the all the build details within the XML elements of the `image` element if desired — but I want to stay consistent with using the Dockerfile.

(my.docker.)name

the full name:tag of the Docker image built and stored in the repository

(my.docker.)ports.api.port

external:internal port mappings that will allow us to map the internal Spring Boot 8080 port to a randomly selected external port accessible by the IT test

(my.docker.)wait.url

URL to wait for before considering the image in a running state and turning control back to the Maven lifecycle to transition from `pre-integration-test` to `integration-test`

- **server.http.port** was generated by the Build Helper Maven Plugin
- **ejava-parent.docker.hostname** will be discussed shortly. It names the Docker host, which

will be `localhost` for most development environments and different for CI/CD builds.

Example Configuration Properties

```
<properties>
    ...
    <my.docker.dockerFile>
${basedir}/Dockerfile.${docker.imageTag}</my.docker.dockerFile> ①
    <my.docker.name>${project.artifactId}:${docker.imageTag}</my.docker.name>
    <my.docker.ports.api.port>${server.http.port}:8080</my.docker.ports.api.port>
    <my.docker.wait.url>http://${ejava-
parent.docker.hostname}:${server.http.port}/api/hello?name=jim</my.docker.wait.url> ②
```

① `docker.imageTag` (2 values: `execjar` and `layered`) helps reference specific Dockerfile

② `ejava-parent.docker.hostname` Maven property definition will be shown later in this lecture. You can just think `localhost` for now.

Chapter 4. Concurrent Testing

Without additional configuration, each running instance will have a basename of the artifactId and a one-up number incremented locally, starting with 1. That means that two builds running this test and using the same Docker host could collide when using the `docker-hello-example-1` name.

Default Name Pattern

```
IMAGE                          PORTS                     NAMES
docker-hello-example:execjar   0.0.0.0:52007->8080/tcp   docker-hello-example-1
```

To override this behavior, we can assign a `containerNamingPattern` to include a random number. Once we are setting the `containerNamingPattern`, we need to explicitly set the image `alias`. I am assigning it to the same `artifactId` value we saw earlier.

Plugin with Container Name Configuration

```
<build>
    <plugins>
        <plugin>
            <groupId>io.fabric8</groupId>
            <artifactId>docker-maven-plugin</artifactId>
            <configuration>
                <containerNamePattern>%a-%t</containerNamePattern> ②
                <images>
                    <alias>${project.artifactId}</alias> ①
                    <image>
                        <external>
            ...
```

① define an alias that will be used to reference the Docker image

② define a container name pattern that will be used to track running container(s) of the image

With the above configuration, we can have a randomly unique name generated that still makes some sense to us when we see it in the Docker host status.

Configured Name Pattern

```
IMAGE                          PORTS                     NAMES
docker-hello-example:execjar   0.0.0.0:52913->8080/tcp   docker-hello-example-
1705249438850 ①
```

① random number has been assigned to artifactId to make container name unique (required) within the Docker host

Chapter 5. Non-local Docker Host

For cases where we are running CI/CD builds within a Docker container, we will not have a local Docker host accessible via localhost. That means that if our IT test uses "http://localhost:...", it will not locate the server. To do so would require running Docker within Docker (termed "DinD") — which is not a popular setting due to elevated permissions required for the CI/CD image. Instead, special settings will be applied to the CI/CD container to identify the non-local location of the Docker host (termed "wormhole pattern"). The server is running on the Docker host as a sibling of the CI/CD build image. The IT test and `my.docker.wait.url` need to reference that non-localhost value.

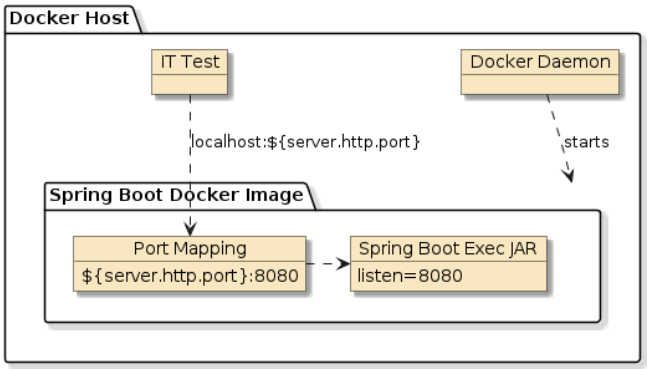

Figure 1. IT Test Running Locally in Native Environment

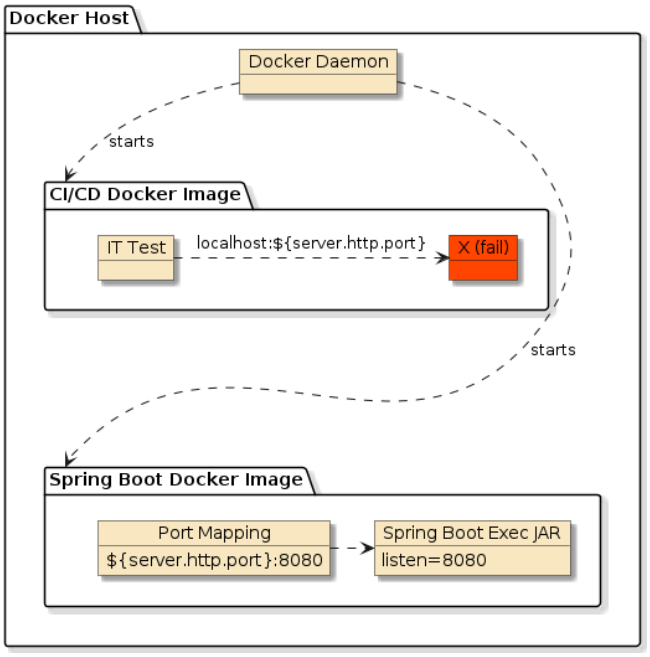

Figure 2. IT Test Running within CI/CD Docker Image

- IT Test running native on Docker host (`localhost` == Docker host)
- Spring Boot Docker image `server.http.port` is exposed on the Docker host
- IT Test locates Docker host using `localhost`

- IT Test and Spring Boot server running within sibling Docker images (`localhost` != Docker host)
- Spring Boot Docker image `server.http.port` is exposed on the Docker host
- IT test `localhost` is now within its local CI/CD Docker image and will fail to find the server running in the sibling Docker image

Tim van Baarsen wrote a nice explanation of our problem/solution in an article. He states that both Windows and MacOS-based Docker installations inherently have a `host.docker.internal` hostname added to images (that does not show up in /etc/hosts).

Any container launched by the Docker host will have its exposed port(s) available on the Docker host and referenced by `docker.host.internal`.

- IT Test and Spring Boot server running within sibling Docker images (`localhost` != Docker host)
- Spring Boot Docker image `server.http.port` is exposed on the Docker host
- IT test uses `docker.host.internal` to locate Docker host (`docker.host.internal` == Docker host)

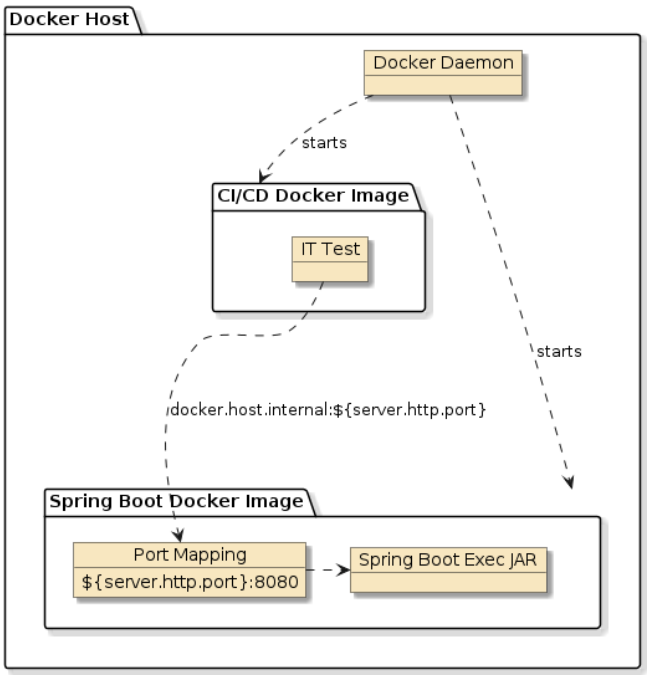

Figure 3. CI/CD Docker Image Configured with Docker Host Address

The following snippet shows that even though `docker.host.internal` is not exposed in the `/etc/hosts` file, a `ping` command is able to resolve `host.docker.internal` to an IP address. This command was run on MacOS.

Resolving host.docker.internal

```
$ docker run --rm mbentley/healthbomb grep -c host.docker.internal /etc/hosts
0 ①

$ docker run --rm mbentley/healthbomb ping host.docker.internal
PING host.docker.internal (192.168.65.254): 56 data bytes ②
```

① running image on MacOS, the `host.docker.internal` name does not show in /etc/hosts

② using an image with `ping` command, we can show that `host.docker.internal` is resolvable

The following snippet shows a curl command resolving the `host.docker.internal` name to the Docker host where our test image is running. The curl command successfully reaches our server — which again — is not running on `localhost` relative the to curl command/client. Curl, in the case is simulating the conditions of the IT test.

Client Completes Call Using Special Hostname

```
$ docker run --rm -p 8080:8080 docker-hello-example:execjar ①
# or
$ mvn docker:run -Dserver.http.port=8080 ②
...
IMAGE                          PORTS                    NAMES
docker-hello-example:execjar   0.0.0.0:8080->8080/tcp   thirsty_bose
```

```
...
$ docker run --rm curlimages/curl curl
http://host.docker.internal:8080/api/hello?name=jim ③
hello, jim
```

① start Docker container using raw Docker command (listening on port 8080 on Docker host)

② start Docker container using Maven plugin (listening on port 8080 on Docker host)

③ Curl client running within sibling Docker image (`localhost` != Docker host; `host.docker.internal` == Docker host)

5.1. Linux Work-around

As Tim van Baarsen points out, the automatic feature provided by Docker Desktop on Windows and MacOs is not automatically provided within Linux installations. We can manually configure the execution to define a hostname with the value of the network between the Docker host and container(s) — obtained by resolving host-gateway. `host-gateway` is set to the IP address of the gateway put in place between the container and the Docker host.

Manually Adding host.docker.internal Using host-gateway Thru Docker Command

```
$ docker run --rm --add-host=host.docker.internal:host-gateway curlimages/curl grep
host.docker.internal /etc/hosts ①

192.168.65.254  host.docker.internal
```

① command line `--add-host=host.docker.internal:host-gateway` maps the `host.docker.internal` hostname to the network between the container and Docker host

The `docker-compose.yml` file provided in the root of the example source tree supplies that value using the `extra_hosts` element.

Manually Adding host.docker.internal Using host-gateway Thru Docker Compose File

```
extra_hosts:
  - host.docker.internal:host-gateway
```

With the `add-host`/`extra_hosts` configured, we are able to resolve the Docker host in Windows, MacOS, and Linux environments.

5.2. Failsafe

Our IT test will need to know the Spring Boot server's hostname in order to properly resolve. We can configure the server's hostname using the `it.server.host` Spring Boot property in the IT test using Failsafe configuration.

it.server Properties Mapped to ServerConfig

Remember that `it.server` properties are mapped to the `ServerConfig`

> `@ConfigurationProperties` instance for IT tests. This is a product of the `ejava` libraries, enabled by Spring Boot but not part of Spring Boot

The Spring Boot property can be added to the Failsafe configuration by appending to the systemPropertyVariables. The snippet below shows the child pom appending new properties to the parent definition (which supplied `it.server.port`). The options are to:

- `combine.children="append"` - add child values to parent-provided values
- `combine.self="override"` - replace parent-provided values with child values

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-failsafe-plugin</artifactId>
    <executions>
        <execution>
            <id>integration-test</id>
            <configuration> <!-- account for CD/CD environment when server will not be
localhost -->
                <systemPropertyVariables combine.children="append">
                    <it.server.host>${ejava-parent.docker.hostname}</it.server.host>
                </systemPropertyVariables>
            </configuration>
        </execution>
    </executions>
</plugin>
```

5.3. Resolving docker.hostname

We have all configurations referencing `${ejava-parent.docker.hostname}`. We now need to make sure this value is either set to `host.docker.internal` (within Docker) or `localhost` (within native environment) depending on our runtime environment.

To make this decision, I am leveraging the fact that we control the CI/CD Docker image and have knowledge of an environment variable called `TESTCONTAINERS_HOST_OVERRIDE` that exists to guide another Docker-based test tool. For this example, it does not matter what we call it as long as we know what to look for.

Root-level docker-compose.yml

```
environment:
  - TESTCONTAINERS_HOST_OVERRIDE=host.docker.internal ①
extra_hosts:
  - host.docker.internal:host-gateway ②
```

① explicitly setting a well-known environment variable within CI/CD environment

② explicitly defining `host.docker.internal` for all CI/CD environments

We can use the presence or absence of the `TESTCONTAINERS_HOST_OVERRIDE` environment variable to

provide a value for an `ejava-parent.docker.hostname` Maven build property.

Inside CI/CD Docker image

host.docker.internal

Outside CI/CD Docker image / Native Environment

localhost

```
<profile> <!-- build is running within Docker-based CI/CD via root level docker-
compose.yml -->
    <id>wormhole-build</id>
    <activation>
        <property>
            <name>env.TESTCONTAINERS_HOST_OVERRIDE</name> ①
        </property>
    </activation>
    <properties> <!-- this hostname is mapped to "host-gateway", used by
testcontainers, but generically usable -->
        <ejava-parent.docker.hostname>${env.TESTCONTAINERS_HOST_OVERRIDE}</ejava-
parent.docker.hostname> ②
    </properties>
</profile>
<profile> <!-- build is running outside of Docker/root-level docker-compose.yml -->
    <id>native-build</id>
    <activation>
        <property>
            <name>!env.TESTCONTAINERS_HOST_OVERRIDE</name> ③
        </property>
    </activation>
    <properties> <!-- localhost outside of Docker CI/CD -->
        <ejava-parent.docker.hostname>localhost</ejava-parent.docker.hostname>④
    </properties>
</profile>
```

① we know out CI/CD container will have `TESTCONTAINERS_HOST_OVERRIDE` defined in all cases

② in our CI/CD container, environment variable `TESTCONTAINERS_HOST_OVERRIDE` will resolve to `host.docker.internal`

③ we assume the lack of `TESTCONTAINERS_HOST_OVERRIDE` means we are in native environment

④ in native environment, Docker containers should be accessible via `localhost` in normal cases

We have the option to use the `docker.host.address` property supplied by Docker Maven Plugin for cases when Docker host is truly remote and `localhost` is not correct. However, I wanted to keep this part simple and independent of the Docker Maven Plugin.

5.4. Example Output

With everything setup, we can now run our IT test against the Docker image. The Docker Compose file used in this example is at the root of the class examples tree. It hosts the ability to run Maven commands within a Docker container. We will discuss Docker Compose in a follow-on lecture.

- running locally in the native environment

Maven/IT Test Running in Native Environment

```
$ env | grep -c TESTCONTAINERS_HOST_OVERRIDE
0 ①

$ mvn clean verify -DitOnly
...
DockerHelloIT#init:34 baseUrl=http://localhost:54132 ②
RestTemplate#debug:127 HTTP GET http://localhost:54132/api/hello?name=jim
...
[INFO] Tests run: 2, Failures: 0, Errors: 0, Skipped: 0
```

① `TESTCONTAINERS_HOST_OVERRIDE` system property is not present

② host defaults to `localhost`

- running within the CI/CD Docker image

Maven/IT Test Running within Docker CI/CD Environment

```
$ docker-compose -f ../../../docker-compose.yml run --rm mvn env | grep
TESTCONTAINERS_HOST_OVERRIDE
TESTCONTAINERS_HOST_OVERRIDE=host.docker.internal ①

$ docker-compose -f ../../../docker-compose.yml run --rm mvn mvn clean verify
-DitOnly
...
DockerHelloIT#init:34 baseUrl=http://host.docker.internal:35423 ②
RestTemplate#debug:127 HTTP GET
http://host.docker.internal:35423/api/hello?name=jim
...
[INFO] Tests run: 2, Failures: 0, Errors: 0, Skipped: 0
```

① `TESTCONTAINERS_HOST_OVERRIDE` system property is present

② host is assigned to value of `TESTCONTAINERS_HOST_OVERRIDE` — host.docker.internal

Chapter 6. Summary

In this module, we learned:

- to automate the build of a Docker image within a module using a Maven plugin
- to implement a heavyweight integration test of that Docker image using the integration goals of a Docker plugin
- to address some singleton matters when running the Docker images simultaneously on the same Docker host.
- to configure a Docker image to communicate with another Docker image running on the same non-local Docker host. This is something common in CI/CD environments.