Using the Docker Gradle Plugin

This guide demonstrates using the family of Gradle plugins for Docker that can be found at github.com/bmuschko/gradle-docker-plugin

Introduction

We will show you how to use the `docker-spring-boot-application` plugin to create a Docker image and the `docker-remote-api` plugin to create, start, test, stop and remove a container.

We are assuming you have a project for a Spring Boot Application and that you used Gradle to build the application. Now you want to create a Docker image to run as a container.

The Spring Boot Gradle plugin since 2.3 includes support for creating a Docker image. So we are assuming you have a project that is using an older version of Spring Boot.

Our sample application exposes a simple API. We want to verify that the application starts up and responds to HTTP requests after the container is started.

What you'll build

You will add plugins and tasks to an existing Spring Boot Gradle project to create an image. You will also add tasks to create, start, test and destroy a container from the image.

What you'll need

- A text editor or IDE like IntelliJ IDEA, Spring Tools 4 for Eclipse or Eclipse with Buildship
- A Java Development Kit (JDK) version 8 or higher. You will find an extensive list at foojay

Adding the plugins

Add the Spring Boot Application and the Docker Remote API plugins

```
plugins {
    id 'org.springframework.boot' version '2.2.7.RELEASE' ①
    id 'io.spring.dependency-management' version '1.0.11.RELEASE'
    id 'org.jetbrains.kotlin.jvm' version '1.3.72'
    id 'org.jetbrains.kotlin.plugin.spring' version '1.3.72'
    id 'io.jumpco.open.kfsm.viz-plugin' version '1.0.9' ②
    id 'com.bmuschko.docker-spring-boot-application' version '7.1.0' ③
    id 'com.bmuschko.docker-remote-api' version '7.1.0' ④
}
```

① Spring Boot, Spring Dependency Management, and Kotlin plugins

② Application specific plugin

③ Docker Spring Boot Application plugin

④ Docker Remote API plugin

Adding imports

Add imports for the Docker Remote API tasks.

```
import com.bmuschko.gradle.docker.tasks.container.DockerCreateContainer
import com.bmuschko.gradle.docker.tasks.container.DockerRemoveContainer
import com.bmuschko.gradle.docker.tasks.container.DockerStartContainer
import com.bmuschko.gradle.docker.tasks.container.DockerStopContainer
```

Create Tasks

We want to create and configure tasks to create and verify the image and test the application running in the container. This will also require tasks to create a container from the image, start the container, stop the container, and remove the container.

Configure the Docker image names

The Spring Boot Application plugin will provide a task named `dockerBuildImage`, to configure the `springBootApplication` plugin with the names to tag the image we use:

```
def dockerImageNames = [ ①
  'guide/sample-app:1.0',
  'guide/sample-app:latest'
] as Set<String>

docker {
    springBootApplication {
        images = dockerImageNames ②
    }
}
```

① We define a variable to hold a set of names for the image tags.

② Assign the names to the `images` property.

NOTE	We declared a variable `dockerImagesNames` with the names for use in the configuration of the `springBootApplication` plugin and the `verifyImage` task

Create verifyImage task

This is a simple task that verifies the image tags.

```
task verifyImage(dependsOn: dockerBuildImage) { ①
    group 'verification'
    doLast {
        assert dockerBuildImage.images.get() == dockerImageNames ②
        logger.lifecycle "$name:verified $dockerImageNames"
    }
}
check.dependsOn(verifyImage) ③
```

① This task depends on the `dockerBuildImage` task.

② We use the variable previously defined to verify that the image has been tagged properly.

> **TIP** Use `$name:` when logging to ensure the task name is included.

Create createContainer task

This task creates a Docker container from the built image.

```
task createContainer(
    type: DockerCreateContainer, ①
    dependsOn: dockerBuildImage ②
) {
    targetImageId dockerBuildImage.imageId ③
}
```

① The task type is `DockerCreateContainer`.

② The task depends on the `dockerBuildImage` task.

③ We assign `dockerBuildImage.imageId` to `targetImageId`

Create startContainer task

This task will start the container.

```
task startContainer(
    type: DockerStartContainer, ①
    dependsOn: createContainer ②
) {
    targetContainerId createContainer.containerId ③
    onComplete {
        logger.lifecycle "$name:started:${createContainer.containerId.get()}"
    }
}
```

① The task type is `DockerStartContainer`

② The task depends on `createContainer`

③ We assign `createContainer.contrainerId` to `targetContainerId`

NOTE	`onComplete` is invoked when the specific task has completed.

Create stopContainer task

This task will stop the container.

```
task stopContainer(type: DockerStopContainer) { ①
    targetContainerId createContainer.containerId ②
    waitTime = 15 ③
    onComplete {
        logger.lifecycle "$name:stopped:${createContainer.containerId.get()}"
    }
}
```

① The task type is `DockerStopContainer`.

② We assign `createContainer.containerId` to `targetContainerId`.

③ We chose a wait time of 15 seconds for the container to stop.

Create removeContainer task

This task removes the container.

```
task removeContainer(
    type: DockerRemoveContainer, ①
    dependsOn: stopContainer ②
) {
    targetContainerId createContainer.containerId ③
    onComplete {
        logger.lifecycle "$name:removed:${createContainer.containerId.get()}"
    }
}
```

① The task type is `DockerRemoveContainer`.

② The task depends on `stopContainer`

③ We assign `createContainer.containerId` to the `targetContainerId`.

Create testContainer task

This task tests a connection to the container and then invokes the API and verifies the outcome.

```
task testContainer(dependsOn: [verifyImage, startContainer]) { ①
    group 'verification'   ②
    finalizedBy removeContainer ③
    doFirst {
        def localGet = new URL('http://localhost:8080')
        def getConnection = localGet.openConnection()
        getConnection.requestMethod = 'GET'
        assert getConnection.responseCode == 200 ④
        logger.lifecycle "$name:responds"
    }
    doLast {
        def localGet = new URL('http://localhost:8080/turnstile')
        def connection = localGet.openConnection()
        connection.requestMethod = 'POST'
        assert connection.responseCode == 200 ⑤
        def jsonSlurper = new groovy.json.JsonSlurper()
        def text = connection.inputStream.withReader {
            it.readLines().join('\n')
        }
        def response = jsonSlurper.parseText(text)
        logger.info "response:$response"
        assert response._links != null ⑥
        assert response._links.self != null
        assert response._links.self.href.contains('/turnstile/')
        logger.lifecycle "$name:passed"
    }
}
check.dependsOn(testContainer) ⑦
```

① The task depends on the `verifyImage` and `startContainer` tasks.

② We assign the task to the group named `verification`. This is a conventional Gradle group for tasks related to unit testing and integration testing.

③ We want to stop and remove the container even if this task fails. Use `finalizedBy` to ensure `removeContainer` is added to the task graph when this task is going to run.

④ Verify that we can connect to the running container.

⑤ Verify that the POST operation responds as expected.

⑥ Verify that the JSON representation of the response matches the expectations.

⑦ Adding this task as a dependency of `check` ensures it will run when using `check` or `build` as Gradle tasks.

Execute the build

```
./gradlew build
```

What else?

You can create integration tests that use TestContainers with your image to perform a range of tests. The integration tests will execute after the image has been built.

You can publish the image to a container register using the `DockerPushImage` task.

Summary

You are now able to create an image and test the application inside a container based on the image.

The family of Docker plugins for Gradle consist of 3 plugins that use the Docker Java Library to invoke the Docker Remote API.

- docker-remote-api plugin
- docker-spring-boot-application plugin
- docker-java-application plugin

The user guide will provide a lot more information how to extend you configuration.

If you create integration tests you should add `startContainer` as a dependency of your task and finalize the task using `removeContainer`.

You can use the docker-remote-api plugin to create any kind of docker image and execute end-to-end, or integration tests as needed.