Getting Started Building Docker Images with Gradle

1. Getting Started

In this Gradle Guide, you will learn how to build a simple Docker image using the Gradle Docker Plugin.

1.1. What You'll Build

You will build a Java application that generates a simple `Dockerfile`, builds the Docker image and creates the Docker container. You will be using the Gradle Docker Plugin to achieve this goal.

1.2. What You'll Need

- A text editor or IDE such as IntelliJ IDEA
- A Java Development Kit (JDK), version 11+
- The latest version of Docker
- The latest Gradle distribution
- The Docker Java library

The Gradle Docker Plugin requires Gradle >= 7.4.0.

1.3. Create a Project Folder

Gradle provides a built-in task, `init`, that initializes a new Gradle project in an empty folder. The `init` task also uses the built-in `wrapper` task to create a Gradle wrapper script, `gradlew`.

The first step is to create a folder for the new project and change directory into it.

```
$ mkdir demo
$ cd demo
```

1.4. Execute the `init` Task

From inside your new project directory, run the `init` task using the following command in a terminal:

```
$ gradle init
```

You will be presented with an interactive application to select various options. For this example application, select the options as shown in the output:

```
Starting a Gradle Daemon (subsequent builds will be faster)

Select type of project to generate:
  1: basic
  2: application
  3: library
  4: Gradle plugin
Enter selection (default: basic) [1..4] 2

Select implementation language:
  1: C++
  2: Groovy
  3: Java
  4: Kotlin
  5: Scala
  6: Swift
Enter selection (default: Java) [1..6] 3

Split functionality across multiple subprojects?:
  1: no - only one application project
  2: yes - application and library projects
Enter selection (default: no - only one application project) [1..2] 1

Select build script DSL:
  1: Groovy
  2: Kotlin
Enter selection (default: Groovy) [1..2] 1

Generate build using new APIs and behavior (some features may change in the next minor
release)? (default: no) [yes,

Select test framework:
  1: JUnit 4
  2: TestNG
  3: Spock
  4: JUnit Jupiter
Enter selection (default: JUnit Jupiter) [1..4] 1

Project name (default: test-application):
Source package (default: test.application): org.gradle

> Task :init
Get more help with your project:
https://docs.gradle.org/7.6/samples/sample_building_java_applications.html

BUILD SUCCESSFUL in 1m 31s
2 actionable tasks: 2 executed
```

The `init` task generates the new project with the following structure:

```
.
├──── app
│     ├──── build.gradle
│     └──── src
│         ├──── main
│         │     ├──── java
│         │     │     └──── org
│         │     │         └──── gradle
│         │     │             └──── App.java
│         │     └──── resources
│         └──── test
│             ├──── java
│             │     └──── org
│             │         └──── gradle
│             │             └──── AppTest.java
│             └──── resources
├──── gradle
│     └──── wrapper
│         ├──── gradle-wrapper.jar
│         └──── gradle-wrapper.properties
├──── gradlew
├──── gradlew.bat
└──── settings.gradle

14 directories, 8 files
```

1.5. Project Files

As you can see, the `init` task provides a comprehensive project complete with a basic application, `App.java`, and corresponing test, `AppTest.java`. However, this example application will primarily focus on building upon the `build.gradle` file as we will be working with Gradle Tasks. You will learn more about Gradle Tasks later in this guide.

The generated `build.gradle` file contains:

```
plugins {
    id 'application'
    }

repositories {
    mavenCentral()
    }

dependencies {
    testImplementation 'junit:junit:4.13.2'
    implementation 'com.google.guava:guava:31.1-jre'
    }

application {
    mainClass = 'org.gradle.App'
    }
```

1.6. Update the `build.gradle` File

Now that you have an application structure in place, let's build upon the `build.gradle` file that was generated by `gradle init` command.

First, you will need to import the required classes, which are Gradle Tasks, and place these statements at the very top of the `build.gradle` file:

```
import com.bmuschko.gradle.docker.tasks.DockerInfo
import com.bmuschko.gradle.docker.tasks.DockerVersion

import com.bmuschko.gradle.docker.tasks.container.DockerCreateContainer
import com.bmuschko.gradle.docker.tasks.container.DockerExecContainer
import com.bmuschko.gradle.docker.tasks.container.DockerStartContainer
import com.bmuschko.gradle.docker.tasks.container.DockerStopContainer

import com.bmuschko.gradle.docker.tasks.image.Dockerfile
import com.bmuschko.gradle.docker.tasks.image.DockerBuildImage
import com.bmuschko.gradle.docker.tasks.image.DockerListImages
```

You will need to provide additional plugins for this example application. You can do so by editing the `plugins` block:

```
plugins {
    id 'java'
    id 'application'
    id 'java-gradle-plugin'
    id 'com.bmuschko.docker-java-application' version '9.1.0'
    id 'com.bmuschko.docker-remote-api' version '9.1.0'
}
```

You will also need to provide additional dependencies for this example application. You can do so by editing the `dependencies` block:

```
dependencies {
    testImplementation 'junit:junit:4.13.2'
    implementation group: 'com.bmuschko', name: 'gradle-docker-plugin', version:
'6.7.0'
    implementation group: 'com.bmuschko', name: 'asciidoctorj-tabbed-code-extension',
version: '0.3'
    }
```

Now that you have initially updated your `build.gradle` file, it's time to define the tasks.

2. Plugins

The Gradle Docker Plugin provides three specific-use plugins to create and build a Docker image:

- `DockerRemoteApiPlugin`
- `DockerJavaApplicationPlugin`
- `DockerSpringBootApplicationPlugin`

For our example application, you will be using the `DockerJavaApplicationPlugin` as described in the Java Application Plugin section of the user guide.

Details on use cases and corresponding details on all three of these plugins may be found in the Provided Plugins section of the Gradle Docker Plugin User Guide and Examples guide.

3. Tasks

A Gradle Task represents a single atomic piece of work for a Gradle build, such as compiling classes or generating JavaDocs. Tasks are comprised by series of actions as defined by implementations of the `Action` interface. Tasks are allowed to depend on other tasks.

You can create your own custom tasks by extending the `DefaultTask` class which implements the `Task` interface.

In the Getting Started section, you imported all the required Gradle Tasks for this example application.

Directly instantiating these classes is **not** supported. You can only instantiate them in the Gradle API or DSL, such as the `build.gradle` file. Attempting to directly instantiate these classes will result in an exception of type `TaskInstantiationException`.

3.1. Defined Tasks

Let's review a list of defined task names and their dependencies that you will use for the example application:

- `createMyAppDockerfile` generates a working `Dockerfile` file based on the template, `Dockefile.tmpl`.
- `buildMyAppImage` builds the Docker image from the generated `Dockerfile`.
  - depends on `createMyAppDockerfile`
- `createMyAppContainer` creates the Docker container.
  - depends on `buildMyAppImage`
- `startMyAppContainer` starts the Docker container.
  - depends on `createMyAppContainer`
- `stopMyAppContainer`
- `executeMyAppContainer`
- `getMyDockerInfo`
- `getMyDockerVersion`
- `getMyDockerImageList`

You will be providing the definitions to all of these tasks within your `build.gradle` file as we review each one.

3.2. Dockerfile Template

As a starting point, you will need a Docker file template for this application. In the root directory of your project, create a file named `Dockerfile.tmpl` and add the following comment:

```
# template for generated Dockerfile
```

3.3. Create the `Dockerfile` File

This task instantiates the `Dockerfile` class to generate a standard `Dockerfile` based on a template file, `Dockerfile.tmpl` that you just created. You can add the following content to your `build.gradle` file:

```
task createMyAppDockerfile(type: Dockerfile) {
    instruction('FROM openjdk:11.0.15-jre-slim')
    instruction('LABEL maintainer=\"Michael Redlich\"')
    instruction('WORKDIR /app2')
    instruction('ENTRYPOINT ["java", "-Xms256m", "-Xmx2048m", "-cp",
"/app2/resources:/muDockerApp/classes:/app2/libs/*", "org.gradle.MainApp"]')
    instruction('EXPOSE 9090 5701')
    instruction('RUN ls -la')
    environmentVariable('JAVA_OPTS', '-XX:+UnlockExperimentalVMOptions
-XX:+UseCGroupMemoryLimitForHeap')
    instructionsFromTemplate(file('Dockerfile.tmpl'))
    }
```

There are three methods defined in this task:

- The `instruction()` method specifies Docker commands
- The `environmentalVariable()` method specifies any environmental variables
- The `instructionsFromTemplate()` method specifies the `Dockerfile.tmpl` file as a template

3.4. Build the Docker Image

The `buildMyAppImage` task instantiates the `DockerBuildImage` class to build the Docker image. You can add the following content to your `build.gradle` file:

```
task buildMyAppImage(type: DockerBuildImage) {
    dependsOn(createMyAppDockerfile)
    inputDir.set(file('build/docker'))
    images.add('test/app2:latest')
    }
```

There are three methods define in this task:

- The `dependsOn()` method specifies a task that should be executed before this task
- The `inputDir()` method specifies a location where to place the generated `Dockerfile`
- The `images.add()` method specifies the name of the Docker image

3.5. Create the Docker Container

The `createMyAppContainer` task instantiates the `DockerCreateContainer` class to create a Docker container. You can add the following content to your `build.gradle` file:

```
task createMyAppContainer(type: DockerCreateContainer) {
    dependsOn(buildMyAppImage)
    targetImageId(buildMyAppImage.getImageId())
    }
```

There are two methods defined in this task:

- The `dependsOn()` method specifies a task that should be execute before this task
- The `targetImageId()` method specifies a Docker image ID

3.6. Start the Docker Container

The `startMyAppContainer` task instantiates the `DockerStartContainer` class to start a Docker container. You can add the following content to your `build.gradle` file:

```
task startMyAppContainer(type: DockerStartContainer) {
    dependsOn(createMyAppContainer)
    targetContainerId(createMyAppContainer.getContainerId())
    }
```

There are two methods defined in this task:

- The `dependsOn()` method specifies a task that should be execute before this task
- The `targetContainerId()` method specifies the Docker image ID to use to start the Docker container

3.7. Stop the Docker Container

The `stopMyAppcontainer` task instantiates the `DockerStopContainer` class to stop a running Docker container. You can add the following content to your `build.gradle` file:

```
task stopMyAppContainer(type: DockerStopContainer) {
    targetContainerId(createMyAppContainer.getContainerId())
    }
```

There is only one method defined in this task:

- The `targetContainerId()` method specifies the Docker image ID to use to stop the running container

3.8. Execute the Docker Container

The `executeMyAppContainer` task instantiates the `DockerExecContainer` class to execute a Docker container. You can add the following content to your `build.gradle` file:

```
task executeMyAppContainer(type: DockerExecContainer) {
    targetContainerId(createMyAppContainer.getContainerId())
    }
```

There is only one method defined in this task:

- The `targetContainerId()` method specifies the Docker image ID to use for executing a Docker container.

3.9. Obtain Information about Docker Installation

The `getMyDockerInfo` task instantiates the `DockerInfo` class to retrieve your local Docker installation. You can add the following content to your `build.gradle` file:

```
task getMyDockerInfo(type: DockerInfo) {
    }
```

There are no defined methods in this task, however, executing this task invokes the `runRemoteCommand()` method defined in the `DockerInfo` class.

3.10. Obtain the Docker Version

The `getMyDockerVersion` task instantiates the `DockerVersion` class to retrieve the version of your local Docker installation. You can add the following content to your `build.gradle` file:

```
task getMyDockerVersion(type: DockerVersion) {
    }
```

There are no defined methods in this task, however, executing this task invokes the `runRemoteCommand()` method defined in the `DockerVersion` class.

3.11. Obtain the List of Docker Images

The `getMyDockerImageList` task instantiates the `DockerListImages` class to retrieve a list of your local Docker images. You can add the following content to your `build.gradle` file:

```
task getMyDockerImageList(type: DockerListImages) {
    }
```

There are no defined methods in this task, however, executing this task invokes the `runRemoteCommand()` method defined in the `DockerListImages` class.

Now that you have added these tasks to your `build.gradle` file, it's time to exercise the example application!

4. Build and Execute the Application

Applications built with Gradle are launched using the `gradle` command along with built-in and user-defined tasks.

4.1. Execute Java Applications

While you won't be required to execute the `public static void main(String[] args)` method defined in the `App.java` file for this example application, you can see at least how this gets executed by issuing the following command:

```
$ gradle run --warning-mode all
```

The `warning-mode` flag is used for listing any deprecated Gradle features that may be incompatible with the upcoming release of Gradle 8.0.

The output will look like:

```
> Task :run
Getting Started Building Docker Images with Gradle

BUILD SUCCESSFUL in 1s
4 actionable tasks: 3 executed, 1 from cache
```

This displays the string, "Getting Started Building Docker Images with Gradle," defined in the `getGreeting()` method in `App.java` that generated using the `gradle init` command.

4.2. Execute Tasks

This example application was designed to initiate the `startMyAppContainer` task and allow the dependency tasks to be executed first, but let's individually execute those task dependencies.

4.2.1. `createMyAppDockerfile`

```
$ gradle createMyAppDockerfile
```

The only output is verification that the task successfully completed:

```
BUILD SUCCESSFUL in 540ms
2 actionable tasks: 1 executed, 1 from cache
```

However, a `Dockerfile` was quietly generated based on the `Dockerfile.tmpl` template. You can find the generated `Dockerfile` file in the `/build/docker` folder.

Let's take a look:

```
FROM openjdk:11.0.15-jre-slim
LABEL maintainer="Michael Redlich"
WORKDIR /app2
ENTRYPOINT ["java", "-Xms256m", "-Xmx2048m", "-cp",
"/app2/resources:/muDockerApp/classes:/app2/libs/*", "org.gradle.MainApp"]
EXPOSE 9090 5701
RUN ls -la
ENV JAVA_OPTS="-XX:+UnlockExperimentalVMOptions -XX:+UseCGroupMemoryLimitForHeap"
# template for generated Dockerfile
```

Notice how the Docker commands were prepended on top of the original comment, `# template for generated Dockerfile`.

4.2.2. buildMyAppImage

Now you can execute the `buildMyAppImage` task:

```
$ gradle buildMyAppImage
```

Keep in mind that this task depends on the `createMyAppDockerfile` that you just executed, so the `Dockerfile` will once-again be generated.

Let's review the output:

```
> Task :buildMyAppImage
Building image using context '/usr/local/apps/gradle-apps/getting-started-building-
docker-images-with-gradle/build/docker'.
Using images 'test/app2:latest'.
Step 1/7 : FROM openjdk:11.0.15-jre-slim
 ---> 699c24828c34
Step 2/7 : LABEL maintainer="Michael Redlich"
 ---> Using cache
 ---> c7f43ff98289
Step 3/7 : WORKDIR /app2
 ---> Using cache
 ---> 7eb1b1aaa358
Step 4/7 : ENTRYPOINT ["java", "-Xms256m", "-Xmx2048m", "-cp",
"/app2/resources:/muDockerApp/classes:/app2/libs/*", "org.gradle.MainApp"]
 ---> Using cache
 ---> 7a6087ee375c
Step 5/7 : EXPOSE 9090 5701
 ---> Using cache
 ---> c2752b3bb2be
Step 6/7 : RUN ls -la
 ---> Using cache
 ---> 8b5c28802e00
Step 7/7 : ENV JAVA_OPTS="-XX:+UnlockExperimentalVMOptions
-XX:+UseCGroupMemoryLimitForHeap"
 ---> Using cache
 ---> c2f34a8c1df6
Successfully built c2f34a8c1df6
Successfully tagged test/app2:latest
Created image with ID 'c2f34a8c1df6'.

BUILD SUCCESSFUL in 1s
3 actionable tasks: 2 executed, 1 from cache
```

You can verify creation of the new image by executing the `docker` command:

```
$ docker images
```

Let's review the output:

```
test/app2                                       latest              c2f34a8c1df6
28 hours ago    227MB
test/app                                        latest              13e86aa2705c
29 hours ago    227MB
test/myapp                                      latest              ec013200a08d
42 hours ago    426MB
mongo                                           latest              0850fead9327   4
weeks ago     700MB
cassandra                                       latest              5b647422e184   2
months ago    353MB
arangodb/arangodb                               latest              d81cf81aaa4b   3
months ago    440MB
jakartaee-cafe                                  v1                  f03eac10057c   3
months ago    484MB
```

4.2.3. createMyAppContainer

Now, let's execute the `createMyAppContainer` task:

```
$ gradle createMyAppContainer
```

Again, keep in mind that this task depends on the `buildMyAppImage` task.

Let's review the output:

```
> Task :createMyAppContainer
Created container with ID
'ea83da3cc17681424105d4c3602789b858cb0daee45b40efd2d36bac1b22f59c'.

BUILD SUCCESSFUL in 1s
4 actionable tasks: 3 executed, 1 from cache
```

4.2.4. startMyAppContainer

Now let's run the entire process by executing the `startMyAppContainer` task that executes all the dependencies:

```
$ gradle startMyAppContainer
```

Let's review the output:

```
> Task :startMyAppContainer
Starting container with ID
'0f150837001528439c9957b7b077b3d85be25fd86b98c3f491044f0a65083e29'.

BUILD SUCCESSFUL in 2s
5 actionable tasks: 4 executed, 1 from cache
```

The next few tasks may be executed independently from the `startMyAppContainer` task.

4.2.5. getMyDockerVersion

```
$ gradle getMyDockerVersion
```

```
> Task :getMyDockerVersion
Retrieving Docker version.
Version          : 20.10.21
Git Commit       : 3056208
Go Version       : go1.18.7
Kernel Version   : 5.15.49-linuxkit
Architecture     : amd64
Operating System : linux

BUILD SUCCESSFUL in 1s
1 actionable task: 1 executed
```

4.2.6. getMyDockerInfo

```
$ gradle getMyDockerInfo
```

```
> Task :getMyDockerInfo
Retrieving Docker info.
Debug                : false
Containers           : 25
Driver               : overlay2
Driver Statuses      : [[Backing Filesystem, extfs], [Supports d_type, true], [Native
Overlay Diff, true], [userxattr, false]]
Images               : 53
IPv4 Forwarding      : true
Index Server Address : https://index.docker.io/v1/
Init Path            : null
Init SHA1            : null
Kernel Version       : 5.15.49-linuxkit
Sockets              : null
Memory Limit         : true
nEvent Listener      : 4
NFd                  : 46
NGoroutines          : 49
Swap Limit           : true
Execution Driver     : null

BUILD SUCCESSFUL in 1s
1 actionable task: 1 executed
```

4.2.7. getMyDockerImageList

```
$ gradle getMyDockerImageList
```

```
> Task :getMyDockerImageList
Repository Tags : test/app2:latest
Image ID        :
sha256:c2f34a8c1df67b413537f9d706c85724bec78cfcaa85374194b10fda76e2fbae
Created         : Sun Jan 08 10:41:03 EST 2023
Virtual Size    : 227459049
-------------------------------------------------
Repository Tags : test/app:latest
Image ID        :
sha256:13e86aa2705c759b869ecc3c4de2e3d3a0c44cd13e7d1fa573693e34ebcbefdf
Created         : Sun Jan 08 09:25:07 EST 2023
Virtual Size    : 227459049
-------------------------------------------------
Repository Tags : test/myapp:latest
Image ID        :
sha256:ec013200a08d606abdf922e4941d5d557ed7a1718ad0a7f83a32e236503a70f4
Created         : Sat Jan 07 20:14:53 EST 2023
Virtual Size    : 425724205
-------------------------------------------------
```

```
Repository Tags : mongo:latest
Image ID        :
sha256:0850fead9327a6d88722c27116309022d78e9daf526b407a88de09762c32e620
Created         : Thu Dec 08 21:37:35 EST 2022
Virtual Size    : 699901543
-------------------------------------------------
Repository Tags : cassandra:latest
Image ID        :
sha256:5b647422e184fb0fd8f6d5513541e85c06876bdaa68decc026abcb65c3fe4ec5
Created         : Fri Nov 04 19:47:02 EDT 2022
Virtual Size    : 353334938
-------------------------------------------------
Repository Tags : arangodb/arangodb:latest
Image ID        :
sha256:d81cf81aaa4b1b637874eab9b877e34dcdee97a00800aaece837fd3d32f7eb56
Created         : Thu Sep 29 10:22:06 EDT 2022
Virtual Size    : 439791606
-------------------------------------------------
Repository Tags : jakartaee-cafe:v1
Image ID        :
sha256:f03eac10057c875306561572ce9ce338aa14f4d73917689ec911e891d0a9ce4d
Created         : Tue Sep 13 13:38:25 EDT 2022
Virtual Size    : 484008628
-------------------------------------------------
Repository Tags : openjdk:11.0.15-jre-slim
Image ID        :
sha256:699c24828c341c27d15a4f62b5c8fa3c5c986bf52fa76228906c50634f430311
Created         : Mon Jul 11 22:00:45 EDT 2022
Virtual Size    : 227459049
-------------------------------------------------

BUILD SUCCESSFUL in 1s
1 actionable task: 1 executed
```

5. Summary

That's it! You've now successfully configured and built a Java application project with Gradle. You've learned how to:

- Initialize a Java project using the `gradle init` command
- Add additonal plugins and dependencies in the generated `build.gradle` file
- Define custom tasks that apply the classes defined within the Gradle Docker Plugin
- Build run the application

6. Next Steps

If you are interested in learning more, please visit these resources.

6.1. Resources

- JavaDocs: Gradle Docker Plugin
- User Guide: Gradle Docker Plugin User Guide and Examples
- User Guide: Java Application Plugin for this example application.
- Gradle Plugin Portal
- Documentation: Using Gradle Plugins