

Getting Started: Gradle Plug-in for Docker

Docker is the popular platform for creating containerized cloud-native applications. It allows you to deploy, test, and scale your applications with ease as Docker images in containers.

Creating and managing Docker images and containers requires running multiple Docker commands. Gradle provides a better alternative as Gradle Docker plug-ins, which help automate the creation and management of Docker images and containers. These plug-ins add special tasks for Docker to the existing Gradle tasks. The special Gradle tasks automatically perform Docker tasks for creating application images, running them into containers, pushing them to repositories, and much more.

This guide walks you through the steps for containerizing and running a simple Java application using a popular Gradle Docker plug-in, the [bmuschko plug-in](#).

What You'll Build

Create and run a Docker image from a simple Java application using the bmuschko Gradle Docker plug-in.

What You Need

- Basic understanding of Gradle and Docker tasks.
- A Java Development Kit (JDK), version 8.0 or higher, which is compatible with both Gradle and bmuschko Gradle Docker plug-in. The example in this guide is built with [OpenJDK 17](#).
- Latest Gradle and Docker distributions. The example in this guide uses Gradle 7.3.3 and Docker 20.10.9.

Note: If installing Docker on Windows, refer to the [installation instructions and limitations](#).

The Java Application

First, you need to create the Java application that you'll containerize. Follow these steps to quickly create a simple 'Hello World!' Java application for this purpose.

1. On your machine's C: drive, create a DemoApp\src\main\java\com\demoapp directory.
2. Create an App.java file in the demoapp directory using a text editor or Java IDE of your choice.

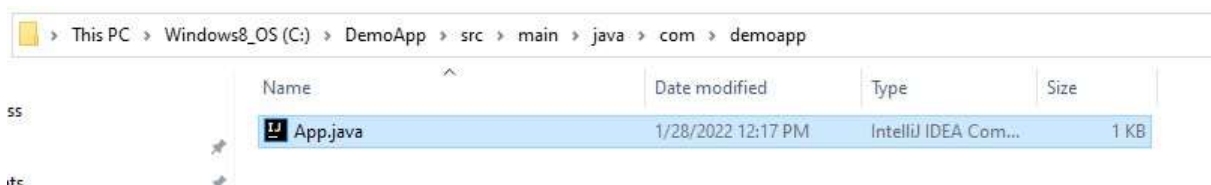


Figure 1 The Java Application Directory

3. Copy the following code into the App.java file and save the file.

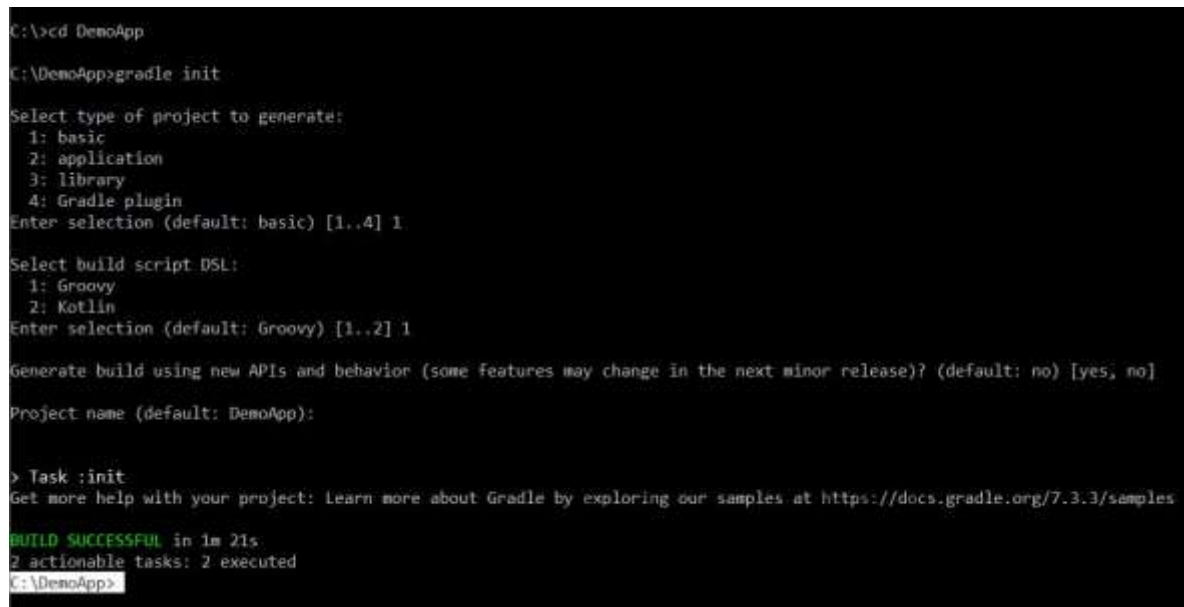
```
package com.demoapp;
public class App {
    public String greet() {
        return "Hello World!";
    }
}
```

```

        public static void main(String[] args) {
            System.out.println(new App().greet());
        }
    }
}

```

4. Navigate to the DemoApp directory and run the `gradle init` command from the command line. When prompted, enter your choice for the project type and DSL as shown in the following figure. For other options, enter to accept the default.



```

C:\>cd DemoApp
C:\DemoApp>gradle init
Select type of project to generate:
  1: basic
  2: application
  3: library
  4: Gradle plugin
Enter selection (default: basic) [1..4] 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, no]
Project name (default: DemoApp):

> Task :init
Get more help with your project: learn more about Gradle by exploring our samples at https://docs.gradle.org/7.3.3/samples
BUILD SUCCESSFUL in 1m 21s
2 actionable tasks: 2 executed
C:\DemoApp>

```

Figure 2 Gradle Init in the Application directory

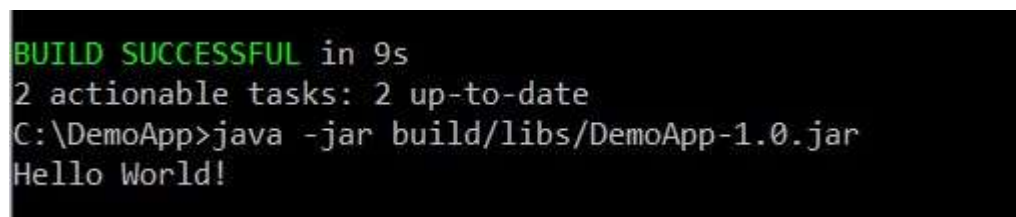
5. Copy the following code in the build.gradle file created in the DemoApp directory and save the file.

```

plugins {
    id 'java'
}
jar {
    manifest {
        attributes 'Main-Class': 'com.demoapp.App'
    }
}
group 'com.demoapp'
version '1.0'

```

6. Run the `gradle build` command to compile the application. Then run the application JAR created in the DemoApp\build\libs directory. The application should print "Hello World!".



```

BUILD SUCCESSFUL in 9s
2 actionable tasks: 2 up-to-date
C:\DemoApp>java -jar build/libs/DemoApp-1.0.jar
Hello World!

```

This is an optional step.

Applying the Docker Plug-in

Now you need to apply the bmuschko plug-in to your application. To do so, you'll specify this plug-in in the build.gradle file of your application.

1. In the plugins section of the build.gradle file, add the plug-in ID and version as follows, and save the file.

```
plugins {  
    id 'java'  
    id 'com.bmuschko.docker-java-application' version '7.2.0'  
}  
  
jar {  
    manifest {  
        attributes 'Main-Class': 'com.demoapp.App'  
    }  
}  
  
group 'com.demoapp'  
version '1.0'
```

2. Run the `gradlew tasks` command. Now in the displayed list of Gradle tasks, you can see the additional tasks for Docker that the plug-in provides. For example, `dockerBuildImage` and `dockerCreateDockerfile`.

Creating and Running Docker Image

Now you are ready to run Gradle tasks to create an image of your application and a Docker container to run the image in. Follow these steps to create and run the image in the container:

1. Run the `gradlew dockerBuildImage` command. The `dockerBuildImage` task runs and creates an image of the Java application and sets its properties. Finally, a message like the following example is displayed, including the ID of the newly-created image.

```
Created image with ID '3e7b7d0a7a75'.  
BUILD SUCCESSFUL in 3s
```

2. Run the `docker run` command, providing the image ID as argument. The application should print "Hello World!".

```
docker run --rm '3e7b7d0a7a75'  
  
Hello World!
```

Summary

Now you have learned how to apply the bmuschko Gradle Docker plug-in to your Java applications. You have also successfully created and run a Docker image of your Java application using the plug-in.

The Next Steps

- Apart from the `dockerBuildImage` task used in this example, the bmuschko plug-in provides more [tasks](#) and [extensions](#) that help automate Docker tasks.
- In addition to containerizing desktop Java applications, the bmuschko plug-in also supports containerizing Spring Boot applications. You can [explore an example](#) and try creating Docker image of a simple Spring Boot application.

- Many community Docker plug-ins are available for Gradle, some of them supporting more Docker tasks and less limitations than the rest. Explore the [community plug-ins](#) available at Gradle.org.

Mugdha V
Work Sample