Spring Boot Gradle Plugin Reference Guide

Andy Wilkinson, Scott Frederick

3.1.5

Table of Contents

Chapter 1. Introduction

The Spring Boot Gradle Plugin provides Spring Boot support in Gradle. It allows you to package executable jar or war archives, run Spring Boot applications, and use the dependency management provided by `spring-boot-dependencies`. Spring Boot's Gradle plugin requires Gradle 7.x (7.5 or later) or 8.x and can be used with Gradle's configuration cache.

In addition to this user guide, API documentation is also available.

Chapter 2. Getting Started

To get started with the plugin it needs to be applied to your project.

The plugin is published to Gradle's plugin portal and can be applied using the `plugins` block:

Groovy

```
plugins {
    id 'org.springframework.boot' version '3.1.5'
}
```

Kotlin

```
plugins {
    id("org.springframework.boot") version "3.1.5"
}
```

Applied in isolation the plugin makes few changes to a project. Instead, the plugin detects when certain other plugins are applied and reacts accordingly. For example, when the `java` plugin is applied a task for building an executable jar is automatically configured. A typical Spring Boot project will apply the `groovy`, `java`, or `org.jetbrains.kotlin.jvm` plugin as a minimum and also use the `io.spring.dependency-management` plugin or Gradle's native bom support for dependency management. For example:

Groovy

```
plugins {
    id 'java'
    id 'org.springframework.boot' version '3.1.5'
}

apply plugin: 'io.spring.dependency-management'
```

Kotlin

```
plugins {
    java
    id("org.springframework.boot") version "3.1.5"
}

apply(plugin = "io.spring.dependency-management")
```

To learn more about how the Spring Boot plugin behaves when other plugins are applied please see the section on reacting to other plugins.

Chapter 3. Managing Dependencies

To manage dependencies in your Spring Boot application, you can either apply the `io.spring.dependency-management` plugin or use Gradle's native bom support. The primary benefit of the former is that it offers property-based customization of managed versions, while using the latter will likely result in faster builds.

3.1. Managing Dependencies with the Dependency Management Plugin

When you apply the `io.spring.dependency-management` plugin, Spring Boot's plugin will automatically import the `spring-boot-dependencies` bom from the version of Spring Boot that you are using. This provides a similar dependency management experience to the one that's enjoyed by Maven users. For example, it allows you to omit version numbers when declaring dependencies that are managed in the bom. To make use of this functionality, declare dependencies in the usual way but omit the version number:

Groovy

```
dependencies {
    implementation('org.springframework.boot:spring-boot-starter-web')
    implementation('org.springframework.boot:spring-boot-starter-data-jpa')
}
```

Kotlin

```
dependencies {
    implementation("org.springframework.boot:spring-boot-starter-web")
    implementation("org.springframework.boot:spring-boot-starter-data-jpa")
}
```

3.1.1. Customizing Managed Versions

The `spring-boot-dependencies` bom that is automatically imported when the dependency management plugin is applied uses properties to control the versions of the dependencies that it manages. Browse the `Dependency versions Appendix` in the Spring Boot reference for a complete list of these properties.

To customize a managed version you set its corresponding property. For example, to customize the version of SLF4J which is controlled by the `slf4j.version` property:

Groovy

```
ext['slf4j.version'] = '1.7.20'
```

Kotlin

```
extra["slf4j.version"] = "1.7.20"
```

> ⚠ Each Spring Boot release is designed and tested against a specific set of third-party dependencies. Overriding versions may cause compatibility issues and should be done with care.

3.1.2. Using Spring Boot's Dependency Management in Isolation

Spring Boot's dependency management can be used in a project without applying Spring Boot's plugin to that project. The `SpringBootPlugin` class provides a `BOM_COORDINATES` constant that can be used to import the bom without having to know its group ID, artifact ID, or version.

First, configure the project to depend on the Spring Boot plugin but do not apply it:

Groovy

```
plugins {
    id 'org.springframework.boot' version '3.1.5' apply false
}
```

Kotlin

```
plugins {
    id("org.springframework.boot") version "3.1.5" apply false
}
```

The Spring Boot plugin's dependency on the dependency management plugin means that you can use the dependency management plugin without having to declare a dependency on it. This also means that you will automatically use the same version of the dependency management plugin as Spring Boot uses.

Apply the dependency management plugin and then configure it to import Spring Boot's bom:

Groovy

```
apply plugin: 'io.spring.dependency-management'

dependencyManagement {
    imports {
        mavenBom
org.springframework.boot.gradle.plugin.SpringBootPlugin.BOM_COORDINATES
    }
}
```

Kotlin

```
apply(plugin = "io.spring.dependency-management")

the<DependencyManagementExtension>().apply {
    imports {

mavenBom(org.springframework.boot.gradle.plugin.SpringBootPlugin.BOM_COORDINATES)
    }
}
```

The Kotlin code above is a bit awkward. That's because we're using the imperative way of applying the dependency management plugin.

We can make the code less awkward by applying the plugin from the root parent project, or by using the `plugins` block as we're doing for the Spring Boot plugin. A downside of this method is that it forces us to specify the version of the dependency management plugin:

```
plugins {
    java
    id("org.springframework.boot") version "3.1.5" apply false
    id("io.spring.dependency-management") version "1.1.3"
}

dependencyManagement {
    imports {

mavenBom(org.springframework.boot.gradle.plugin.SpringBootPlugin.BOM_COORDINATES)
    }
}
```

3.1.3. Learning More

To learn more about the capabilities of the dependency management plugin, please refer to its documentation.

3.2. Managing Dependencies with Gradle's Bom Support

Gradle allows a bom to be used to manage a project's versions by declaring it as a `platform` or `enforcedPlatform` dependency. A `platform` dependency treats the versions in the bom as recommendations and other versions and constraints in the dependency graph may cause a version of a dependency other than that declared in the bom to be used. An `enforcedPlatform` dependency treats the versions in the bom as requirements and they will override any other version found in the dependency graph.

The `SpringBootPlugin` class provides a `BOM_COORDINATES` constant that can be used to declare a dependency upon Spring Boot's bom without having to know its group ID, artifact ID, or version, as shown in the following example:

Groovy

```
dependencies {
    implementation
platform(org.springframework.boot.gradle.plugin.SpringBootPlugin.BOM_COORDINATES)
}
```

Kotlin

```
dependencies {

implementation(platform(org.springframework.boot.gradle.plugin.SpringBootPlugin.BOM_CO
ORDINATES))
}
```

A platform or enforced platform will only constrain the versions of the configuration in which it has been declared or that extend from the configuration in which it has been declared. As a result, in may be necessary to declare the same dependency in more than one configuration.

3.2.1. Customizing Managed Versions

When using Gradle's bom support, you cannot use the properties from `spring-boot-dependencies` to control the versions of the dependencies that it manages. Instead, you must use one of the mechanisms that Gradle provides. One such mechanism is a resolution strategy. SLF4J's modules are all in the `org.slf4j` group so their version can be controlled by configuring every dependency in that group to use a particular version, as shown in the following example:

Groovy

```
configurations.all {
    resolutionStrategy.eachDependency { DependencyResolveDetails details ->
        if (details.requested.group == 'org.slf4j') {
            details.useVersion '1.7.20'
        }
    }
}
```

Kotlin

```
configurations.all {
    resolutionStrategy.eachDependency {
        if (requested.group == "org.slf4j") {
            useVersion("1.7.20")
        }
    }
}
```

Each Spring Boot release is designed and tested against a specific set of third-party dependencies. Overriding versions may cause compatibility issues and should be done with care.

Chapter 4. Packaging Executable Archives

The plugin can create executable archives (jar files and war files) that contain all of an application's dependencies and can then be run with `java -jar`.

4.1. Packaging Executable Jars

Executable jars can be built using the `bootJar` task. The task is automatically created when the `java` plugin is applied and is an instance of `BootJar`. The `assemble` task is automatically configured to depend upon the `bootJar` task so running `assemble` (or `build`) will also run the `bootJar` task.

4.2. Packaging Executable Wars

Executable wars can be built using the `bootWar` task. The task is automatically created when the `war` plugin is applied and is an instance of `BootWar`. The `assemble` task is automatically configured to depend upon the `bootWar` task so running `assemble` (or `build`) will also run the `bootWar` task.

4.2.1. Packaging Executable and Deployable Wars

A war file can be packaged such that it can be executed using `java -jar` and deployed to an external container. To do so, the embedded servlet container dependencies should be added to the `providedRuntime` configuration, for example:

Groovy

```
dependencies {
    implementation('org.springframework.boot:spring-boot-starter-web')
    providedRuntime('org.springframework.boot:spring-boot-starter-tomcat')
}
```

Kotlin

```
dependencies {
    implementation("org.springframework.boot:spring-boot-starter-web")
    providedRuntime("org.springframework.boot:spring-boot-starter-tomcat")
}
```

This ensures that they are package in the war file's `WEB-INF/lib-provided` directory from where they will not conflict with the external container's own classes.

> `providedRuntime` is preferred to Gradle's `compileOnly` configuration as, among other limitations, `compileOnly` dependencies are not on the test classpath so any web-based integration tests will fail.

4.3. Packaging Executable and Plain Archives

By default, when the `bootJar` or `bootWar` tasks are configured, the `jar` or `war` tasks are configured to use `plain` as the convention for their archive classifier. This ensures that `bootJar` and `jar` or `bootWar` and `war` have different output locations, allowing both the executable archive and the plain archive to be built at the same time.

If you prefer that the executable archive, rather than the plain archive, uses a classifier, configure the classifiers as shown in the following example for the `jar` and `bootJar` tasks:

Groovy

```
tasks.named("bootJar") {
    archiveClassifier = 'boot'
}

tasks.named("jar") {
    archiveClassifier = ''
}
```

Kotlin

```
tasks.named<BootJar>("bootJar") {
    archiveClassifier.set("boot")
}

tasks.named<Jar>("jar") {
    archiveClassifier.set("")
}
```

Alternatively, if you prefer that the plain archive isn't built at all, disable its task as shown in the following example for the `jar` task:

Groovy

```
tasks.named("jar") {
    enabled = false
}
```

Kotlin

```
tasks.named<Jar>("jar") {
    enabled = false
}
```

Do not disable the `jar` task when creating native images. See #33238 for details.

4.4. Configuring Executable Archive Packaging

The `BootJar` and `BootWar` tasks are subclasses of Gradle's `Jar` and `War` tasks respectively. As a result, all of the standard configuration options that are available when packaging a jar or war are also available when packaging an executable jar or war. A number of configuration options that are specific to executable jars and wars are also provided.

4.4.1. Configuring the Main Class

By default, the executable archive's main class will be configured automatically by looking for a class with a `public static void main(String[])` method in the main source set's output.

The main class can also be configured explicitly using the task's `mainClass` property:

Groovy

```
tasks.named("bootJar") {
    mainClass = 'com.example.ExampleApplication'
}
```

Kotlin

```
tasks.named<BootJar>("bootJar") {
    mainClass.set("com.example.ExampleApplication")
}
```

Alternatively, the main class name can be configured project-wide using the `mainClass` property of the Spring Boot DSL:

Groovy

```
springBoot {
    mainClass = 'com.example.ExampleApplication'
}
```

Kotlin

```
springBoot {
    mainClass.set("com.example.ExampleApplication")
}
```

If the `application` plugin has been applied its `mainClass` property must be configured and can be used for the same purpose:

Groovy

```
application {
    mainClass = 'com.example.ExampleApplication'
}
```

Kotlin

```
application {
    mainClass.set("com.example.ExampleApplication")
}
```

Lastly, the `Start-Class` attribute can be configured on the task's manifest:

Groovy

```
tasks.named("bootJar") {
    manifest {
        attributes 'Start-Class': 'com.example.ExampleApplication'
    }
}
```

Kotlin

```
tasks.named<BootJar>("bootJar") {
    manifest {
        attributes("Start-Class" to "com.example.ExampleApplication")
    }
}
```

> If the main class is written in Kotlin, the name of the generated Java class should be used. By default, this is the name of the Kotlin class with the `Kt` suffix added. For example, `ExampleApplication` becomes `ExampleApplicationKt`. If another name is defined using `@JvmName` then that name should be used.

4.4.2. Including Development-only Dependencies

By default all dependencies declared in the `developmentOnly` configuration will be excluded from an executable jar or war.

If you want to include dependencies declared in the `developmentOnly` configuration in your archive, configure the classpath of its task to include the configuration, as shown in the following example for the `bootWar` task:

Groovy

```
tasks.named("bootWar") {
    classpath configurations.developmentOnly
}
```

Kotlin

```
tasks.named<BootWar>("bootWar") {
    classpath(configurations["developmentOnly"])
}
```

4.4.3. Configuring Libraries that Require Unpacking

Most libraries can be used directly when nested in an executable archive, however certain libraries can have problems. For example, JRuby includes its own nested jar support which assumes that `jruby-complete.jar` is always directly available on the file system.

To deal with any problematic libraries, an executable archive can be configured to unpack specific nested jars to a temporary directory when the executable archive is run. Libraries can be identified as requiring unpacking using Ant-style patterns that match against the absolute path of the source jar file:

Groovy

```
tasks.named("bootJar") {
    requiresUnpack '**/jruby-complete-*.jar'
}
```

Kotlin

```
tasks.named<BootJar>("bootJar") {
    requiresUnpack("**/jruby-complete-*.jar")
}
```

For more control a closure can also be used. The closure is passed a `FileTreeElement` and should return a `boolean` indicating whether or not unpacking is required.

4.4.4. Making an Archive Fully Executable

Spring Boot provides support for fully executable archives. An archive is made fully executable by prepending a shell script that knows how to launch the application. On Unix-like platforms, this launch script allows the archive to be run directly like any other executable or to be installed as a service.

> Currently, some tools do not accept this format so you may not always be able to use this technique. For example, `jar -xf` may silently fail to extract a jar or war that has been made fully-executable. It is recommended that you only enable this option if you intend to execute it directly, rather than running it with `java -jar` or deploying it to a servlet container.

To use this feature, the inclusion of the launch script must be enabled:

Groovy

```
tasks.named("bootJar") {
    launchScript()
}
```

Kotlin

```
tasks.named<BootJar>("bootJar") {
    launchScript()
}
```

This will add Spring Boot's default launch script to the archive. The default launch script includes several properties with sensible default values. The values can be customized using the `properties` property:

Groovy

```
tasks.named("bootJar") {
    launchScript {
        properties 'logFilename': 'example-app.log'
    }
}
```

Kotlin

```
tasks.named<BootJar>("bootJar") {
    launchScript {
        properties(mapOf("logFilename" to "example-app.log"))
    }
}
```

If the default launch script does not meet your needs, the `script` property can be used to provide a custom launch script:

Groovy

```
tasks.named("bootJar") {
    launchScript {
        script = file('src/custom.script')
    }
}
```

Kotlin

```
tasks.named<BootJar>("bootJar") {
    launchScript {
        script = file("src/custom.script")
    }
}
```

4.4.5. Using the PropertiesLauncher

To use the `PropertiesLauncher` to launch an executable jar or war, configure the task's manifest to set the `Main-Class` attribute:

Groovy

```
tasks.named("bootWar") {
    manifest {
        attributes 'Main-Class': 'org.springframework.boot.loader.PropertiesLauncher'
    }
}
```

Kotlin

```
tasks.named<BootWar>("bootWar") {
    manifest {
        attributes("Main-Class" to
"org.springframework.boot.loader.PropertiesLauncher")
    }
}
```

4.4.6. Packaging Layered Jar or War

By default, the `bootJar` task builds an archive that contains the application's classes and dependencies in `BOOT-INF/classes` and `BOOT-INF/lib` respectively. Similarly, `bootWar` builds an archive that contains the application's classes in `WEB-INF/classes` and dependencies in `WEB-INF/lib` and `WEB-INF/lib-provided`. For cases where a docker image needs to be built from the contents of the jar, it's useful to be able to separate these directories further so that they can be written into distinct layers.

Layered jars use the same layout as regular boot packaged jars, but include an additional meta-data file that describes each layer.

By default, the following layers are defined:

- `dependencies` for any non-project dependency whose version does not contain `SNAPSHOT`.
- `spring-boot-loader` for the jar loader classes.
- `snapshot-dependencies` for any non-project dependency whose version contains `SNAPSHOT`.
- `application` for project dependencies, application classes, and resources.

The layers order is important as it determines how likely previous layers can be cached when part of the application changes. The default order is `dependencies`, `spring-boot-loader`, `snapshot-dependencies`, `application`. Content that is least likely to change should be added first, followed by layers that are more likely to change.

To disable this feature, you can do so in the following manner:

Groovy

```
tasks.named("bootJar") {
    layered {
        enabled = false
    }
}
```

Kotlin

```
tasks.named<BootJar>("bootJar") {
    layered {
        enabled.set(false)
    }
}
```

When a layered jar or war is created, the `spring-boot-jarmode-layertools` jar will be added as a dependency to your archive. With this jar on the classpath, you can launch your application in a special mode which allows the bootstrap code to run something entirely different from your application, for example, something that extracts the layers. If you wish to exclude this dependency, you can do so in the following manner:

Groovy

```
tasks.named("bootJar") {
    layered {
        includeLayerTools = false
    }
}
```

Kotlin

```
tasks.named<BootJar>("bootJar") {
    layered {
        includeLayerTools.set(false)
    }
}
```

Custom Layers Configuration

Depending on your application, you may want to tune how layers are created and add new ones.

This can be done using configuration that describes how the jar or war can be separated into layers, and the order of those layers. The following example shows how the default ordering described above can be defined explicitly:

Groovy

```
tasks.named("bootJar") {
    layered {
        application {
            intoLayer("spring-boot-loader") {
                include "org/springframework/boot/loader/**"
            }
            intoLayer("application")
        }
        dependencies {
            intoLayer("application") {
                includeProjectDependencies()
            }
            intoLayer("snapshot-dependencies") {
                include "*:*:*SNAPSHOT"
            }
            intoLayer("dependencies")
        }
        layerOrder = ["dependencies", "spring-boot-loader", "snapshot-dependencies",
"application"]
    }
}
```

Kotlin

```
tasks.named<BootJar>("bootJar") {
    layered {
        application {
            intoLayer("spring-boot-loader") {
                include("org/springframework/boot/loader/**")
            }
            intoLayer("application")
        }
        dependencies {
            intoLayer("application") {
                includeProjectDependencies()
            }
            intoLayer("snapshot-dependencies") {
                include("*:*:*SNAPSHOT")
            }
            intoLayer("dependencies")
        }
        layerOrder.set(listOf("dependencies", "spring-boot-loader", "snapshot-
dependencies", "application"))
    }
}
```

The `layered` DSL is defined using three parts:

- The `application` closure defines how the application classes and resources should be layered.
- The `dependencies` closure defines how dependencies should be layered.
- The `layerOrder` method defines the order that the layers should be written.

Nested `intoLayer` closures are used within `application` and `dependencies` sections to claim content for a layer. These closures are evaluated in the order that they are defined, from top to bottom. Any content not claimed by an earlier `intoLayer` closure remains available for subsequent ones to consider.

The `intoLayer` closure claims content using nested `include` and `exclude` calls. The `application` closure uses Ant-style path matching for include/exclude parameters. The `dependencies` section uses `group:artifact[:version]` patterns. It also provides `includeProjectDependencies()` and `excludeProjectDependencies()` methods that can be used to include or exclude project dependencies.

If no `include` call is made, then all content (not claimed by an earlier closure) is considered.

If no `exclude` call is made, then no exclusions are applied.

Looking at the `dependencies` closure in the example above, we can see that the first `intoLayer` will claim all project dependencies for the `application` layer. The next `intoLayer` will claim all SNAPSHOT dependencies for the `snapshot-dependencies` layer. The third and final `intoLayer` will claim anything left (in this case, any dependency that is not a project dependency or a SNAPSHOT) for the `dependencies` layer.

The `application` closure has similar rules. First claiming `org/springframework/boot/loader/**` content for the `spring-boot-loader` layer. Then claiming any remaining classes and resources for the `application` layer.

> The order that `intoLayer` closures are added is often different from the order that the layers are written. For this reason the `layerOrder` method must always be called and *must* cover all layers referenced by the `intoLayer` calls.

Chapter 5. Packaging OCI Images

The plugin can create an OCI image from a jar or war file using Cloud Native Buildpacks (CNB). Images can be built using the `bootBuildImage` task.

> For security reasons, images build and run as non-root users. See the CNB specification for more details.

The task is automatically created when the `java` or `war` plugin is applied and is an instance of `BootBuildImage`.

5.1. Docker Daemon

The `bootBuildImage` task requires access to a Docker daemon. By default, it will communicate with a Docker daemon over a local connection. This works with Docker Engine on all supported platforms without configuration.

Environment variables can be set to configure the `bootBuildImage` task to use an alternative local or remote connection. The following table shows the environment variables and their values:

Environment variable	Description
DOCKER_HOST	URL containing the host and port for the Docker daemon - for example `tcp://192.168.99.100:2376`
DOCKER_TLS_VERIFY	Enable secure HTTPS protocol when set to `1` (optional)
DOCKER_CERT_PATH	Path to certificate and key files for HTTPS (required if `DOCKER_TLS_VERIFY=1`, ignored otherwise)

Docker daemon connection information can also be provided using `docker` properties in the plugin configuration. The following table summarizes the available properties:

Property	Description
`host`	URL containing the host and port for the Docker daemon - for example `tcp://192.168.99.100:2376`
`tlsVerify`	Enable secure HTTPS protocol when set to `true` (optional)
`certPath`	Path to certificate and key files for HTTPS (required if `tlsVerify` is `true`, ignored otherwise)
`bindHostToBuilder`	When `true`, the value of the `host` property will be provided to the container that is created for the CNB builder (optional)

For more details, see also examples.

5.2. Docker Registry

If the Docker images specified by the `builder` or `runImage` properties are stored in a private Docker image registry that requires authentication, the authentication credentials can be provided using `docker.builderRegistry` properties.

If the generated Docker image is to be published to a Docker image registry, the authentication credentials can be provided using `docker.publishRegistry` properties.

Properties are provided for user authentication or identity token authentication. Consult the documentation for the Docker registry being used to store images for further information on supported authentication methods.

The following table summarizes the available properties for `docker.builderRegistry` and `docker.publishRegistry`:

Property	Description
`username`	Username for the Docker image registry user. Required for user authentication.
`password`	Password for the Docker image registry user. Required for user authentication.
`url`	Address of the Docker image registry. Optional for user authentication.
`email`	E-mail address for the Docker image registry user. Optional for user authentication.
`token`	Identity token for the Docker image registry user. Required for token authentication.

For more details, see also examples.

5.3. Image Customizations

The plugin invokes a builder to orchestrate the generation of an image. The builder includes multiple buildpacks that can inspect the application to influence the generated image. By default, the plugin chooses a builder image. The name of the generated image is deduced from project properties.

Task properties can be used to configure how the builder should operate on the project. The following table summarizes the available properties and their default values:

Property	Command-line option	Description	Default value
builder	--builder	Name of the Builder image to use.	paketobuildpacks/builder:base or paketobuildpacks/builder:tiny when GraalVM Native Image plugin is applied.
runImage	--runImage	Name of the run image to use.	No default value, indicating the run image specified in Builder metadata should be used.
imageName	--imageName	Image name for the generated image.	docker.io/library/${project.name}:${project.version}
pullPolicy	--pullPolicy	Policy used to determine when to pull the builder and run images from the registry. Acceptable values are ALWAYS, NEVER, and IF_NOT_PRESENT.	ALWAYS
environment		Environment variables that should be passed to the builder.	Empty or ['BP_NATIVE_IMAGE': 'true'] when GraalVM Native Image plugin is applied.

Property	Command-line option	Description	Default value
`buildpacks`		Buildpacks that the builder should use when building the image. Only the specified buildpacks will be used, overriding the default buildpacks included in the builder. Buildpack references must be in one of the following forms: • Buildpack in the builder - `[urn:cnb:builder:]<buildpack ID>[@<version>]` • Buildpack in a directory on the file system - `[file://]<path>` • Buildpack in a gzipped tar (.tgz) file on the file system - `[file://]<path>/<file name>` • Buildpack in an OCI image - `[docker://]<host>/<repo>[:<tag>][@<digest>]`	None, indicating the builder should use the buildpacks included in it.

Property	Command-line option	Description	Default value
`bindings`		Volume bind mounts that should be mounted to the builder container when building the image. The bindings will be passed unparsed and unvalidated to Docker when creating the builder container. Bindings must be in one of the following forms: • `<host source path>:<container destination path>[:<options>]` • `<host volume name>:<container destination path>[:<options>]` Where `<options>` can contain: • `ro` to mount the volume as read-only in the container • `rw` to mount the volume as readable and writable in the container • `volume-opt=key=value` to specify key-value pairs consisting of an option name and its value	

Property	Command-line option	Description	Default value
`network`	`--network`	The network driver the builder container will be configured to use. The value supplied will be passed unvalidated to Docker when creating the builder container.	
`cleanCache`	`--cleanCache`	Whether to clean the cache before building.	`false`
`verboseLogging`		Enables verbose logging of builder operations.	`false`
`publish`	`--publishImage`	Whether to publish the generated image to a Docker registry.	`false`
`tags`		A list of one or more additional tags to apply to the generated image. The values provided to the `tags` option should be full image references in the form of `[image name]:[tag]` or `[repository]/[image name]:[tag]`.	
`buildCache`		A cache containing layers created by buildpacks and used by the image building process.	A named volume in the Docker daemon, with a name derived from the image name.
`launchCache`		A cache containing layers created by buildpacks and used by the image launching process.	A named volume in the Docker daemon, with a name derived from the image name.

Property	Command-line option	Description	Default value
`createdDate`	`--createdDate`	A date that will be used to set the `Created` field in the generated image's metadata. The value must be a string in the ISO 8601 instant format, or `now` to use the current date and time.	A fixed date that enables build reproducibility.
`applicationDirectory`	`--applicationDirectory`	The path to a directory that application contents will be uploaded to in the builder image. Application contents will also be in this location in the generated image.	`/workspace`

> The plugin detects the target Java compatibility of the project using the JavaPlugin's `targetCompatibility` property. When using the default Paketo builder and buildpacks, the plugin instructs the buildpacks to install the same Java version. You can override this behaviour as shown in the builder configuration examples.

5.4. Examples

5.4.1. Custom Image Builder and Run Image

If you need to customize the builder used to create the image or the run image used to launch the built image, configure the task as shown in the following example:

Groovy

```
tasks.named("bootBuildImage") {
    builder = "mine/java-cnb-builder"
    runImage = "mine/java-cnb-run"
}
```

Kotlin

```
tasks.named<BootBuildImage>("bootBuildImage") {
    builder.set("mine/java-cnb-builder")
    runImage.set("mine/java-cnb-run")
}
```

This configuration will use a builder image with the name `mine/java-cnb-builder` and the tag `latest`, and the run image named `mine/java-cnb-run` and the tag `latest`.

The builder and run image can be specified on the command line as well, as shown in this example:

```
$ gradle bootBuildImage --builder=mine/java-cnb-builder --runImage=mine/java-cnb-run
```

5.4.2. Builder Configuration

If the builder exposes configuration options, those can be set using the `environment` property.

The following is an example of configuring the JVM version used by the Paketo Java buildpacks at build time:

Groovy

```
tasks.named("bootBuildImage") {
    environment["BP_JVM_VERSION"] = "17"
}
```

Kotlin

```
tasks.named<BootBuildImage>("bootBuildImage") {
    environment.set(environment.get() + mapOf("BP_JVM_VERSION" to "17"))
}
```

If there is a network proxy between the Docker daemon the builder runs in and network locations that buildpacks download artifacts from, you will need to configure the builder to use the proxy. When using the Paketo builder, this can be accomplished by setting the `HTTPS_PROXY` and/or `HTTP_PROXY` environment variables as show in the following example:

Groovy

```
tasks.named("bootBuildImage") {
    environment["HTTP_PROXY"] = "http://proxy.example.com"
    environment["HTTPS_PROXY"] = "https://proxy.example.com"
}
```

Kotlin

```
tasks.named<BootBuildImage>("bootBuildImage") {
    environment.set(mapOf("HTTP_PROXY" to "http://proxy.example.com",
                          "HTTPS_PROXY" to "https://proxy.example.com"))
}
```

5.4.3. Runtime JVM Configuration

Paketo Java buildpacks configure the JVM runtime environment by setting the `JAVA_TOOL_OPTIONS` environment variable. The buildpack-provided `JAVA_TOOL_OPTIONS` value can be modified to customize JVM runtime behavior when the application image is launched in a container.

Environment variable modifications that should be stored in the image and applied to every deployment can be set as described in the Paketo documentation and shown in the following example:

Groovy

```
tasks.named("bootBuildImage") {
    environment["BPE_DELIM_JAVA_TOOL_OPTIONS"] = " "
    environment["BPE_APPEND_JAVA_TOOL_OPTIONS"] = "-XX:+HeapDumpOnOutOfMemoryError"
}
```

Kotlin

```
tasks.named<BootBuildImage>("bootBuildImage") {
    environment.set(mapOf(
        "BPE_DELIM_JAVA_TOOL_OPTIONS" to " ",
        "BPE_APPEND_JAVA_TOOL_OPTIONS" to "-XX:+HeapDumpOnOutOfMemoryError"
    ))
}
```

5.4.4. Custom Image Name

By default, the image name is inferred from the `name` and the `version` of the project, something like `docker.io/library/${project.name}:${project.version}`. You can take control over the name by setting task properties, as shown in the following example:

Groovy

```
tasks.named("bootBuildImage") {
    imageName = "example.com/library/${project.name}"
}
```

Kotlin

```
tasks.named<BootBuildImage>("bootBuildImage") {
    imageName.set("example.com/library/${project.name}")
}
```

Note that this configuration does not provide an explicit tag so `latest` is used. It is possible to specify a tag as well, either using `${project.version}`, any property available in the build or a hardcoded version.

The image name can be specified on the command line as well, as shown in this example:

```
$ gradle bootBuildImage --imageName=example.com/library/my-app:v1
```

5.4.5. Buildpacks

By default, the builder will use buildpacks included in the builder image and apply them in a pre-defined order. An alternative set of buildpacks can be provided to apply buildpacks that are not included in the builder, or to change the order of included buildpacks. When one or more buildpacks are provided, only the specified buildpacks will be applied.

The following example instructs the builder to use a custom buildpack packaged in a `.tgz` file, followed by a buildpack included in the builder.

Groovy

```
tasks.named("bootBuildImage") {
    buildpacks = ["file:///path/to/example-buildpack.tgz", "urn:cnb:builder:paketo-
buildpacks/java"]
}
```

Kotlin

```
tasks.named<BootBuildImage>("bootBuildImage") {
    buildpacks.set(listOf("file:///path/to/example-buildpack.tgz",
"urn:cnb:builder:paketo-buildpacks/java"))
}
```

Buildpacks can be specified in any of the forms shown below.

A buildpack located in a CNB Builder (version may be omitted if there is only one buildpack in the builder matching the `buildpack-id`):

- `urn:cnb:builder:buildpack-id`
- `urn:cnb:builder:buildpack-id@0.0.1`
- `buildpack-id`
- `buildpack-id@0.0.1`

A path to a directory containing buildpack content (not supported on Windows):

- `file:///path/to/buildpack/`
- `/path/to/buildpack/`

A path to a gzipped tar file containing buildpack content:

- `file:///path/to/buildpack.tgz`

- `/path/to/buildpack.tgz`

An OCI image containing a packaged buildpack:

- `docker://example/buildpack`
- `docker:///example/buildpack:latest`
- `docker:///example/buildpack@sha256:45b23dee08…`
- `example/buildpack`
- `example/buildpack:latest`
- `example/buildpack@sha256:45b23dee08…`

5.4.6. Image Publishing

The generated image can be published to a Docker registry by enabling a `publish` option.

If the Docker registry requires authentication, the credentials can be configured using `docker.publishRegistry` properties. If the Docker registry does not require authentication, the `docker.publishRegistry` configuration can be omitted.

> The registry that the image will be published to is determined by the registry part of the image name (`docker.example.com` in these examples). If `docker.publishRegistry` credentials are configured and include a `url` property, this value is passed to the registry but is not used to determine the publishing registry location.

Groovy

```
tasks.named("bootBuildImage") {
    imageName.set("docker.example.com/library/${project.name}")
    publish = true
    docker {
        publishRegistry {
            username = "user"
            password = "secret"
        }
    }
}
```

Kotlin

```
tasks.named<BootBuildImage>("bootBuildImage") {
    imageName.set("docker.example.com/library/${project.name}")
    publish.set(true)
    docker {
        publishRegistry {
            username.set("user")
            password.set("secret")
        }
    }
}
```

The publish option can be specified on the command line as well, as shown in this example:

```
$ gradle bootBuildImage --imageName=docker.example.com/library/my-app:v1
--publishImage
```

5.4.7. Builder Cache Configuration

The CNB builder caches layers that are used when building and launching an image. By default, these caches are stored as named volumes in the Docker daemon with names that are derived from the full name of the target image. If the image name changes frequently, for example when the project version is used as a tag in the image name, then the caches can be invalidated frequently.

The cache volumes can be configured to use alternative names to give more control over cache lifecycle as shown in the following example:

Groovy

```
tasks.named("bootBuildImage") {
    buildCache {
        volume {
            name = "cache-${rootProject.name}.build"
        }
    }
    launchCache {
        volume {
            name = "cache-${rootProject.name}.launch"
        }
    }
}
```

Kotlin

```
tasks.named<BootBuildImage>("bootBuildImage") {
    buildCache {
        volume {
            name.set("cache-${rootProject.name}.build")
        }
    }
    launchCache {
        volume {
            name.set("cache-${rootProject.name}.launch")
        }
    }
}
```

5.4.8. Docker Configuration

Docker Configuration for minikube

The plugin can communicate with the Docker daemon provided by minikube instead of the default local connection.

On Linux and macOS, environment variables can be set using the command `eval $(minikube docker-env)` after minikube has been started.

The plugin can also be configured to use the minikube daemon by providing connection details similar to those shown in the following example:

Groovy

```
tasks.named("bootBuildImage") {
    docker {
        host = "tcp://192.168.99.100:2376"
        tlsVerify = true
        certPath = "/home/user/.minikube/certs"
    }
}
```

Kotlin

```
tasks.named<BootBuildImage>("bootBuildImage") {
    docker {
        host.set("tcp://192.168.99.100:2376")
        tlsVerify.set(true)
        certPath.set("/home/user/.minikube/certs")
    }
}
```

Docker Configuration for podman

The plugin can communicate with a podman container engine.

The plugin can be configured to use podman local connection by providing connection details similar to those shown in the following example:

Groovy

```
tasks.named("bootBuildImage") {
    docker {
        host = "unix:///run/user/1000/podman/podman.sock"
        bindHostToBuilder = true
    }
}
```

Kotlin

```
tasks.named<BootBuildImage>("bootBuildImage") {
    docker {
        host.set("unix:///run/user/1000/podman/podman.sock")
        bindHostToBuilder.set(true)
    }
}
```

With the `podman` CLI installed, the command `podman info --format='{{.Host.RemoteSocket.Path}}'` can be used to get the value for the `docker.host` configuration property shown in this example.

Docker Configuration for Authentication

If the builder or run image are stored in a private Docker registry that supports user authentication, authentication details can be provided using `docker.builderRegistry` properties as shown in the following example:

Groovy

```
tasks.named("bootBuildImage") {
    docker {
        builderRegistry {
            username = "user"
            password = "secret"
            url = "https://docker.example.com/v1/"
            email = "user@example.com"
        }
    }
}
```

Kotlin

```
tasks.named<BootBuildImage>("bootBuildImage") {
    docker {
        builderRegistry {
            username.set("user")
            password.set("secret")
            url.set("https://docker.example.com/v1/")
            email.set("user@example.com")
        }
    }
}
```

If the builder or run image is stored in a private Docker registry that supports token authentication, the token value can be provided using `docker.builderRegistry` as shown in the following example:

Groovy

```
tasks.named("bootBuildImage") {
    docker {
        builderRegistry {
            token = "9cbaf023786cd7..."
        }
    }
}
```

Kotlin

```
tasks.named<BootBuildImage>("bootBuildImage") {
    docker {
        builderRegistry {
            token.set("9cbaf023786cd7...")
        }
    }
}
```

Chapter 6. Publishing your Application

6.1. Publishing with the Maven-publish Plugin

To publish your Spring Boot jar or war, add it to the publication using the `artifact` method on `MavenPublication`. Pass the task that produces that artifact that you wish to publish to the `artifact` method. For example, to publish the artifact produced by the default `bootJar` task:

Groovy

```
publishing {
    publications {
        bootJava(MavenPublication) {
            artifact tasks.named("bootJar")
        }
    }
    repositories {
        maven {
            url 'https://repo.example.com'
        }
    }
}
```

Kotlin

```
publishing {
    publications {
        create<MavenPublication>("bootJava") {
            artifact(tasks.named("bootJar"))
        }
    }
    repositories {
        maven {
            url = uri("https://repo.example.com")
        }
    }
}
```

6.2. Distributing with the Application Plugin

When the `application` plugin is applied a distribution named `boot` is created. This distribution contains the archive produced by the `bootJar` or `bootWar` task and scripts to launch it on Unix-like platforms and Windows. Zip and tar distributions can be built by the `bootDistZip` and `bootDistTar` tasks respectively. To use the `application` plugin, its `mainClassName` property must be configured with the name of your application's main class.

Chapter 7. Running your Application with Gradle

To run your application without first building an archive use the `bootRun` task:

```
$ ./gradlew bootRun
```

The `bootRun` task is an instance of `BootRun` which is a `JavaExec` subclass. As such, all of the usual configuration options for executing a Java process in Gradle are available to you. The task is automatically configured to use the runtime classpath of the main source set.

By default, the main class will be configured automatically by looking for a class with a `public static void main(String[])` method in the main source set's output.

The main class can also be configured explicitly using the task's `main` property:

Groovy

```
tasks.named("bootRun") {
    mainClass = 'com.example.ExampleApplication'
}
```

Kotlin

```
tasks.named<BootRun>("bootRun") {
    mainClass.set("com.example.ExampleApplication")
}
```

Alternatively, the main class name can be configured project-wide using the `mainClass` property of the Spring Boot DSL:

Groovy

```
springBoot {
    mainClass = 'com.example.ExampleApplication'
}
```

Kotlin

```
springBoot {
    mainClass.set("com.example.ExampleApplication")
}
```

By default, `bootRun` will configure the JVM to optimize its launch for faster startup during development. This behavior can be disabled by using the `optimizedLaunch` property, as shown in the following example:

Groovy

```
tasks.named("bootRun") {
    optimizedLaunch = false
}
```

Kotlin

```
tasks.named<BootRun>("bootRun") {
    optimizedLaunch.set(false)
}
```

If the `application` plugin has been applied, its `mainClass` property must be configured and can be used for the same purpose:

Groovy

```
application {
    mainClass = 'com.example.ExampleApplication'
}
```

Kotlin

```
application {
    mainClass.set("com.example.ExampleApplication")
}
```

7.1. Passing Arguments to your Application

Like all `JavaExec` tasks, arguments can be passed into `bootRun` from the command line using `--args='<arguments>'` when using Gradle 4.9 or later. For example, to run your application with a profile named `dev` active the following command can be used:

```
$ ./gradlew bootRun --args='--spring.profiles.active=dev'
```

See the javadoc for `JavaExec.setArgsString` for further details.

7.2. Passing System properties to your application

Since `bootRun` is a standard `JavaExec` task, system properties can be passed to the application's JVM by specifying them in the build script. To make that value of a system property to be configurable set its value using a project property. To allow a project property to be optional, reference it using `findProperty`. Doing so also allows a default value to be provided using the `?:` Elvis operator, as shown in the following example:

Groovy

```
tasks.named("bootRun") {
    systemProperty 'com.example.property', findProperty('example') ?: 'default'
}
```

Kotlin

```
tasks.named<BootRun>("bootRun") {
    systemProperty("com.example.property", findProperty("example") ?: "default")
}
```

The preceding example sets that `com.example.property` system property to the value of the `example` project property. If the `example` project property has not been set, the value of the system property will be `default`.

Gradle allows project properties to be set in a variety of ways, including on the command line using the `-P` flag, as shown in the following example:

```
$ ./gradlew bootRun -Pexample=custom
```

The preceding example sets the value of the `example` project property to `custom`. `bootRun` will then use this as the value of the `com.example.property` system property.

7.3. Reloading Resources

If devtools has been added to your project it will automatically monitor your application's classpath for changes. Note that modified files need to be recompiled for the classpath to update in order to trigger reloading with devtools. For more details on using devtools, refer to this section of the reference documentation.

Alternatively, you can configure `bootRun` such that your application's static resources are loaded from their source location:

Groovy

```
tasks.named("bootRun") {
    sourceResources sourceSets.main
}
```

Kotlin

```
tasks.named<BootRun>("bootRun") {
    sourceResources(sourceSets["main"])
}
```

This makes them reloadable in the live application which can be helpful at development time.

7.4. Using a Test Main Class

In addition to `bootRun` a `bootTestRun` task is also registered. Like `bootRun`, `bootTestRun` is an instance of `BootRun` but it's configured to use a main class found in the output of the test source set rather than the main source set. It also uses the test source set's runtime classpath rather than the main source set's runtime classpath. As `bootTestRun` is an instance of `BootRun`, all of the configuration options described above for `bootRun` can also be used with `bootTestRun`.

Chapter 8. Ahead-of-Time Processing

Spring AOT is a process that analyzes your code at build-time in order to generate an optimized version of it. It is most often used to help generate GraalVM native images.

The Spring Boot Gradle plugin provides tasks that can be used to perform AOT processing on both application and test code. The tasks are configured automatically when the GraalVM Native Image plugin is applied:

Groovy

```
plugins {
    id 'org.springframework.boot' version '3.1.5'
    id 'org.graalvm.buildtools.native' version '0.9.27'
    id 'java'
}
```

Kotlin

```
plugins {
    id("org.springframework.boot") version "3.1.5"
    id("org.graalvm.buildtools.native") version "0.9.27"
    java
}
```

8.1. Processing Applications

Based on your `@SpringBootApplication`-annotated main class, the `processAot` task generates a persistent view of the beans that are going to be contributed at runtime in a way that bean instantiation is as straightforward as possible. Additional post-processing of the factory is possible using callbacks. For instance, these are used to generate the necessary reflection configuration that GraalVM needs to initialize the context in a native image.

As the `BeanFactory` is fully prepared at build-time, conditions are also evaluated. This has an important difference compared to what a regular Spring Boot application does at runtime. For instance, if you want to opt-in or opt-out for certain features, you need to configure the environment used at build time to do so. To this end, the `processAot` task is a `JavaExec` task and can be configured with environment variables, system properties, and arguments as needed.

The `nativeCompile` task of the GraalVM Native Image plugin is automatically configured to use the output of the `processAot` task.

8.2. Processing Tests

The AOT engine can be applied to JUnit 5 tests that use Spring's Test Context Framework. Suitable tests are processed by the `processTestAot` task to generate `ApplicationContextInitialzer` code. As with application AOT processing, the `BeanFactory` is fully prepared at build-time. As with `processAot`, the `processTestAot` task is `JavaExec` subclass and can be configured as needed to influence this processing.

The `nativeTest` task of the GraalVM Native Image plugin is automatically configured to use the output of the `processAot` and `processTestAot` tasks.

Chapter 9. Integrating with Actuator

9.1. Generating Build Information

Spring Boot Actuator's `info` endpoint automatically publishes information about your build in the presence of a `META-INF/build-info.properties` file. A `BuildInfo` task is provided to generate this file. The easiest way to use the task is through the plugin's DSL:

Groovy

```
springBoot {
    buildInfo()
}
```

Kotlin

```
springBoot {
    buildInfo()
}
```

This will configure a `BuildInfo` task named `bootBuildInfo` and, if it exists, make the Java plugin's `classes` task depend upon it. The task's destination directory will be `META-INF` in the output directory of the main source set's resources (typically `build/resources/main`).

By default, the generated build information is derived from the project:

Property	Default value
`build.artifact`	The base name of the `bootJar` or `bootWar` task
`build.group`	The group of the project
`build.name`	The name of the project
`build.version`	The version of the project
`build.time`	The time at which the project is being built

The properties can be customized using the DSL:

Groovy

```
springBoot {
    buildInfo {
        properties {
            artifact = 'example-app'
            version = '1.2.3'
            group = 'com.example'
            name = 'Example application'
        }
    }
}
```

Kotlin

```
springBoot {
    buildInfo {
        properties {
            artifact.set("example-app")
            version.set("1.2.3")
            group.set("com.example")
            name.set("Example application")
        }
    }
}
```

To exclude any of the default properties from the generated build information, add its name to the excludes. For example, the `time` property can be excluded as follows:

Groovy

```
springBoot {
    buildInfo {
        excludes = ['time']
    }
}
```

Kotlin

```
springBoot {
    buildInfo {
        excludes.set(setOf("time"))
    }
}
```

The default value for `build.time` is the instant at which the project is being built. A side-effect of this is that the task will never be up-to-date. As a result, builds will take longer as more tasks, including the project's tests, will have to be executed. Another side-effect is that the task's output will always change and, therefore, the build will not be truly repeatable. If you value build performance or repeatability more highly than the accuracy of the `build.time` property, exclude the `time` property as shown in the preceding example.

Additional properties can also be added to the build information:

Groovy

```
springBoot {
    buildInfo {
        properties {
            additional = [
                'a': 'alpha',
                'b': 'bravo'
            ]
        }
    }
}
```

Kotlin

```
springBoot {
    buildInfo {
        properties {
            additional.set(mapOf(
                "a" to "alpha",
                "b" to "bravo"
            ))
        }
    }
}
```

Chapter 10. Reacting to Other Plugins

When another plugin is applied the Spring Boot plugin reacts by making various changes to the project's configuration. This section describes those changes.

10.1. Reacting to the Java Plugin

When Gradle's `java` plugin is applied to a project, the Spring Boot plugin:

1. Creates a `BootJar` task named `bootJar` that will create an executable, fat jar for the project. The jar will contain everything on the runtime classpath of the main source set; classes are packaged in `BOOT-INF/classes` and jars are packaged in `BOOT-INF/lib`
2. Configures the `assemble` task to depend on the `bootJar` task.
3. Configures the `jar` task to use `plain` as the convention for its archive classifier.
4. Creates a `BootBuildImage` task named `bootBuildImage` that will create a OCI image using a buildpack.
5. Creates a `BootRun` task named `bootRun` that can be used to run your application using the `main` source set to find its main method and provide its runtime classpath.
6. Creates a 'BootRun` task named `bootTestRun` that can be used to run your application using the `test` source set to find its main method and provide its runtime classpath.
7. Creates a configuration named `bootArchives` that contains the artifact produced by the `bootJar` task.
8. Creates a configuration named `developmentOnly` for dependencies that are only required at development time, such as Spring Boot's Devtools, and should not be packaged in executable jars and wars.
9. Creates a configuration named `productionRuntimeClasspath`. It is equivalent to `runtimeClasspath` minus any dependencies that only appear in the `developmentOnly` configuration.
10. Configures any `JavaCompile` tasks with no configured encoding to use `UTF-8`.
11. Configures any `JavaCompile` tasks to use the `-parameters` compiler argument.

10.2. Reacting to the Kotlin Plugin

When Kotlin's Gradle plugin is applied to a project, the Spring Boot plugin:

1. Aligns the Kotlin version used in Spring Boot's dependency management with the version of the plugin. This is achieved by setting the `kotlin.version` property with a value that matches the version of the Kotlin plugin.
2. Configures any `KotlinCompile` tasks to use the `-java-parameters` compiler argument.

10.3. Reacting to the War Plugin

When Gradle's `war` plugin is applied to a project, the Spring Boot plugin:

1. Creates a `BootWar` task named `bootWar` that will create an executable, fat war for the project. In addition to the standard packaging, everything in the `providedRuntime` configuration will be packaged in `WEB-INF/lib-provided`.
2. Configures the `assemble` task to depend on the `bootWar` task.
3. Configures the `war` task to use `plain` as the convention for its archive classifier.
4. Configures the `bootArchives` configuration to contain the artifact produced by the `bootWar` task.

10.4. Reacting to the Dependency Management Plugin

When the `io.spring.dependency-management` plugin is applied to a project, the Spring Boot plugin will automatically import the `spring-boot-dependencies` bom.

10.5. Reacting to the Application Plugin

When Gradle's `application` plugin is applied to a project, the Spring Boot plugin:

1. Creates a `CreateStartScripts` task named `bootStartScripts` that will create scripts that launch the artifact in the `bootArchives` configuration using `java -jar`. The task is configured to use the `applicationDefaultJvmArgs` property as a convention for its `defaultJvmOpts` property.
2. Creates a new distribution named `boot` and configures it to contain the artifact in the `bootArchives` configuration in its `lib` directory and the start scripts in its `bin` directory.
3. Configures the `bootRun` task to use the `mainClassName` property as a convention for its `main` property.
4. Configures the `bootRun` and `bootTestRun` tasks to use the `applicationDefaultJvmArgs` property as a convention for their `jvmArgs` property.
5. Configures the `bootJar` task to use the `mainClassName` property as a convention for the `Start-Class` entry in its manifest.
6. Configures the `bootWar` task to use the `mainClassName` property as a convention for the `Start-Class` entry in its manifest.

10.6. Reacting to the GraalVM Native Image Plugin

When the GraalVM Native Image plugin is applied to a project, the Spring Boot plugin:

1. Applies the `org.springframework.boot.aot` plugin that:
    a. Registers `aot` and `aotTest` source sets.
    b. Registers a `ProcessAot` task named `processAot` that will generate AOT-optimized source for the application in the `aot` source set.
    c. Configures the Java compilation and process resources tasks for the `aot` source set to depend upon `processAot`.
    d. Registers a `ProcessTestAot` task named `processTestAot` that will generated AOT-optimized source for the application's tests in the `aotTest` source set.

e. Configures the Java compilation and process resources tasks for the `aotTest` source set to depend upon `processTestAot`.

2. Adds the output of the `aot` source set to the classpath of the `main` GraalVM native binary.
3. Adds the output of the `aotTest` source set to the classpath of the `test` GraalVM native binary.
4. Configures the GraalVM extension to disable Toolchain detection.
5. Configures each GraalVM native binary to require GraalVM 22.3 or later.
6. Configures the `bootJar` task to include the reachability metadata produced by the `collectReachabilityMetadata` task in its jar.
7. Configures the `bootBuildImage` task to use `paketobuildpacks/builder:tiny` as its builder and to set `BP_NATIVE_IMAGE` to `true` in its environment.