Chapter 11. Deploying Your Spring Boot Application

In software development, deployment is the on-ramp to production for an application.

Regardless of any capabilities an application may promise its end users, until said users can actually use the application, it is effectively an academic what-if exercise. Figuratively and often very literally, deployment is the payoff.

Referencing the Spring Initializr, many developers are aware that Spring Boot applications can be created as WAR files or JAR files. Most of those same developers also know that there are many good reasons (several of which were mentioned earlier in this book) to eschew the WAR option and create executable JAR files, and few good reasons to do the opposite. What many developers may not realize is that even when building a Spring Boot executable JAR, there are numerous options for deployment to fulfill various requirements and use cases.

In this chapter, I examine ways to deploy your Spring Boot application with options useful for different target destinations and discuss their relative merits. I then demonstrate how to create these deployment artifacts, explain options for optimal execution, and show how to verify their components and provenance. You almost certainly have more and better tools for deploying your Spring Boot applications than you realized.

CODE CHECKOUT CHECKUP

Please check out branch *chapter11begin* from the code repository to begin.

Revisiting the Spring Boot Executable JAR

As discussed way back in Chapter 1, Spring Boot's executable JAR provides maximum utility and versatility in a single, self-contained, testable, and deployable unit. It's fast to create and iterate, dynamically self-configurable to changes in its environment, and simple in the extreme to distribute and maintain.

Every cloud provider offers an application hosting option that enjoys widespread use for prototyping through production deployments, and most of those application platforms expect a largely self-contained deployable application, offering only the barest of environmental essentials. A Spring Boot JAR fits quite naturally into these clean environments, requiring only the presence of a JDK for frictionless execution; some platforms even specify Spring Boot by name due to its seamless fit for app hosting. By bringing mechanisms with it for external interactions involving HTTP exchanges, messaging, and more, a Spring Boot application can eliminate installation, configuration, and maintenance of an application server or other externalities. This dramatically reduces developer workload and application platform overhead.

Since a Spring Boot application possesses full control over dependent libraries, it eliminates fear of external dependency changes. Scheduled updates to an application server, servlet engine, database or messaging libraries, or any of a number of other critical components have crashed countless non-Boot applications over the years. In those applications that rely on external components maintained by the underlying app platform, developers must be hypervigilant for unplanned outages due to the world shifting under their feet, simply due to a dot-release change of a single dependent library. Exciting times.

With a Spring Boot application, upgrades to any dependencies—whether core Spring libraries or second- (or third- , or fourth- , etc.) tier dependencies—are much less painful and stressful as well. The app developer upgrades and tests the application and deploys an update (typically using a blue-green deployment) only when satisfied that everything is working as expected. Since dependencies are no longer external to the application but instead bundled with it, the developer has full control over dependency versions and upgrade timing.

Spring Boot JARs have another useful trick up their sleeve, courtesy of the Spring Boot Maven and Gradle plug-ins: the ability to create what is sometimes called a “fully executable” JAR. The quotes are intentional and are present in official documentation as well because a JDK is still required for the application to function. So what is meant by a “fully executable” Spring Boot application, and how does one create it?

Let’s begin with the how.

Building a “Fully Executable” Spring Boot JAR

I’ll use PlaneFinder for this example. For purposes of comparison, I build the project from the command line without changes using `mvn clean` `package`. This results in the following JAR being created in the project's *target* directory (result trimmed to fit page):

```
» ls -lb target/*.jar

-rw-r--r--  1 markheckler  staff  27085204 target/planefinder-0.0.1-SNAPSHO
```

This Spring Boot JAR is referred to as an *executable JAR* because it consists of the entire application without need for external downstream dependencies; all it requires to execute is a JVM provided by an installed JDK. Running the app in its current state looks something like this (results trimmed and edited to fit page):

```
» java -jar target/planefinder-0.0.1-SNAPSHOT.jar

  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::                (v2.4.0)

: Starting PlanefinderApplication v0.0.1-SNAPSHOT
: No active profile set, falling back to default profiles: default
: Bootstrapping Spring Data R2DBC repositories in DEFAULT mode.
: Finished Spring Data repository scanning in 132 ms. Found 1 R2DBC
  repository interfaces.
: Netty started on port(s): 7634
: Netty RSocket started on port(s): 7635
: Started PlanefinderApplication in 2.75 seconds (JVM running for 3.106)
```

This works as expected, of course, and it serves as a baseline for what comes next. I now revisit PlaneFinder's *pom.xml* to add the indicated XML snippet to the existing section for the `spring-boot-maven-plug-in`, as shown in Figure 11-1.

```xml
<build>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
            <version>2.4.0</version>
            <configuration>
                <executable>true</executable>
            </configuration>
        </plugin>
    </plugins>
</build>
```

Figure 11-1. Plugins section of PlaneFinder *pom.xml* file

Returning to the terminal, I again build the project from the command line using `mvn clean package`. This time, there is a notable difference in the resultant JAR created within the project's *target* directory, as indicated in the following output (result trimmed to fit page):

```
» ls -lb target/*.jar

-rwxr--r--  1 markheckler  staff  27094314 target/planefinder-0.0.1-SNAPSHO
```

It's ever so slightly larger than Boot's standard executable JAR, to the tune of 9,110 byte, or just under 9 KB. What does this gain you?

Java JAR files are read from end to beginning—yes, you read that correctly—until an end-of-file marker is found. When creating a so-called "fully executable JAR," the Spring Boot Maven plug-in ingeniously prepends a script to the beginning of the usual Spring Boot executable JAR that enables it to be run like any other executable binary (assuming the presence of a JDK) on a Unix- or Linux-based system, including registering it with `init.d` or `systemd`. Examining PlaneFinder's JAR in an editor results in the following (only a portion of the script header is shown for brevity; it is quite extensive):

```
#!/bin/bash
#
#    .   ____          _            __ _ _
#   /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
#  ( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
#   \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
#    '  |____| .__|_| |_|_| |_\__, | / / / /
#   =========|_|==============|___/=/_/_/_/
#   :: Spring Boot Startup Script ::
#
```

```
### BEGIN INIT INFO
# Provides:          planefinder
# Required-Start:    $remote_fs $syslog $network
# Required-Stop:     $remote_fs $syslog $network
# Default-Start:     2 3 4 5
# Default-Stop:      0 1 6
# Short-Description: planefinder
# Description:       Data feed for SBUR
# chkconfig:         2345 99 01
### END INIT INFO

...

# Action functions
start() {
  if [[ -f "$pid_file" ]]; then
    pid=$(cat "$pid_file")
    isRunning "$pid" && { echoYellow "Already running [$pid]"; return 0; }
  fi
  do_start "$@"
}

do_start() {
  working_dir=$(dirname "$jarfile")
  pushd "$working_dir" > /dev/null
  if [[ ! -e "$PID_FOLDER" ]]; then
    mkdir -p "$PID_FOLDER" &> /dev/null
    if [[ -n "$run_user" ]]; then
      chown "$run_user" "$PID_FOLDER"
    fi
  fi
  if [[ ! -e "$log_file" ]]; then
    touch "$log_file" &> /dev/null
    if [[ -n "$run_user" ]]; then
      chown "$run_user" "$log_file"
    fi
  fi
  if [[ -n "$run_user" ]]; then
    checkPermissions || return $?
    if [ $USE_START_STOP_DAEMON = true ] && type start-stop-daemon >
        /dev/null 2>&1; then
      start-stop-daemon --start --quiet \
        --chuid "$run_user" \
        --name "$identity" \
        --make-pidfile --pidfile "$pid_file" \
        --background --no-close \
        --startas "$javaexe" \
        --chdir "$working_dir" \
       —"${arguments[@]}" \
        >> "$log_file" 2>&1
      await_file "$pid_file"
```

```
    else
      su -s /bin/sh -c "$javaexe $(printf "\"%s\" " "${arguments[@]}") >>
        \"$log_file\" 2>&1 & echo \$!" "$run_user" > "$pid_file"
    fi
    pid=$(cat "$pid_file")
  else
    checkPermissions || return $?
    "$javaexe" "${arguments[@]}" >> "$log_file" 2>&1 &
    pid=$!
    disown $pid
    echo "$pid" > "$pid_file"
  fi
  [[ -z $pid ]] && { echoRed "Failed to start"; return 1; }
  echoGreen "Started [$pid]"
}

stop() {
  working_dir=$(dirname "$jarfile")
  pushd "$working_dir" > /dev/null
  [[ -f $pid_file ]] ||
    { echoYellow "Not running (pidfile not found)"; return 0; }
  pid=$(cat "$pid_file")
  isRunning "$pid" || { echoYellow "Not running (process ${pid}).
    Removing stale pid file."; rm -f "$pid_file"; return 0; }
  do_stop "$pid" "$pid_file"
}

do_stop() {
  kill "$1" &> /dev/null || { echoRed "Unable to kill process $1"; return 1
  for ((i = 1; i <= STOP_WAIT_TIME; i++)); do
    isRunning "$1" || { echoGreen "Stopped [$1]"; rm -f "$2"; return 0; }
    [[ $i -eq STOP_WAIT_TIME/2 ]] && kill "$1" &> /dev/null
    sleep 1
  done
  echoRed "Unable to kill process $1";
  return 1;
}

force_stop() {
  [[ -f $pid_file ]] ||
    { echoYellow "Not running (pidfile not found)"; return 0; }
  pid=$(cat "$pid_file")
  isRunning "$pid" ||
    { echoYellow "Not running (process ${pid}). Removing stale pid file.";
    rm -f "$pid_file"; return 0; }
  do_force_stop "$pid" "$pid_file"
}

do_force_stop() {
  kill -9 "$1" &> /dev/null ||
      { echoRed "Unable to kill process $1"; return 1; }
  for ((i = 1; i <= STOP_WAIT_TIME; i++)); do
```

```
    isRunning "$1" || { echoGreen "Stopped [$1]"; rm -f "$2"; return 0; }
    [[ $i -eq STOP_WAIT_TIME/2 ]] && kill -9 "$1" &> /dev/null
    sleep 1
  done
  echoRed "Unable to kill process $1";
  return 1;
}

restart() {
  stop && start
}

force_reload() {
  working_dir=$(dirname "$jarfile")
  pushd "$working_dir" > /dev/null
  [[ -f $pid_file ]] || { echoRed "Not running (pidfile not found)";
      return 7; }
  pid=$(cat "$pid_file")
  rm -f "$pid_file"
  isRunning "$pid" || { echoRed "Not running (process ${pid} not found)";
      return 7; }
  do_stop "$pid" "$pid_file"
  do_start
}

status() {
  working_dir=$(dirname "$jarfile")
  pushd "$working_dir" > /dev/null
  [[ -f "$pid_file" ]] || { echoRed "Not running"; return 3; }
  pid=$(cat "$pid_file")
  isRunning "$pid" || { echoRed "Not running (process ${pid} not found)";
      return 1; }
  echoGreen "Running [$pid]"
  return 0
}

run() {
  pushd "$(dirname "$jarfile")" > /dev/null
  "$javaexe" "${arguments[@]}"
  result=$?
  popd > /dev/null
  return "$result"
}

# Call the appropriate action function
case "$action" in
start)
  start "$@"; exit $?;;
stop)
  stop "$@"; exit $?;;
force-stop)
  force_stop "$@"; exit $?;;
```

```
restart)
  restart "$@"; exit $?;;
force-reload)
  force_reload "$@"; exit $?;;
status)
  status "$@"; exit $?;;
run)
  run "$@"; exit $?;;
*)
  echo "Usage: $0 {start|stop|force-stop|restart|force-reload|status|run}";
    exit 1;
esac

exit 0
<binary portion omitted>
```

The Spring Boot Maven (or Gradle, if chosen as the build system) plug-in also sets file owner permissions to read, write, and execute (rwx) for the output JAR. Doing so enables it to be executed as indicated previously and allows the header script to locate the JDK, prepare the application for execution, and run it as demonstrated here (results trimmed and edited to fit page):

```
» target/planefinder-0.0.1-SNAPSHOT.jar

  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::                (v2.4.0)

: Starting PlanefinderApplication v0.0.1-SNAPSHOT
: No active profile set, falling back to default profiles: default
: Bootstrapping Spring Data R2DBC repositories in DEFAULT mode.
: Finished Spring Data repository scanning in 185 ms.
  Found 1 R2DBC repository interfaces.
: Netty started on port(s): 7634
: Netty RSocket started on port(s): 7635
: Started PlanefinderApplication in 2.938 seconds (JVM running for 3.335)
```

Now that I've demonstrated how, it's time to discuss what this option brings to the table.

What Does It Mean?

The ability to create a Spring Boot "fully executable" JAR is not a solution to all problems, but it does provide a unique capability for deeper integration with underlying Unix- and Linux-based systems when necessary. Adding a Spring Boot application to supply startup functionality becomes trivial thanks to the embedded startup script and execute permissions.

If you don't need or can't make use of that capability in your current application environments, you should continue to simply create typical Spring Boot executable JAR output that makes use of `java -jar`. This is simply another tool in your toolbox, included at no cost and requiring nearly no effort from you to implement, for when you find you need it.

Exploding JARs

Spring Boot's innovative approach of nesting dependent JAR files completely intact and unchanged within the Boot executable JAR lends itself brilliantly to subsequent actions like extraction. Reversing the process that was involved in adding them to the Spring Boot executable JAR produces the component artifacts in their original, unaltered state. It sounds simple because it *is* simple.

There are many reasons you might want to rehydrate a Spring Boot executable JAR into its various, separate parts:

- Extracted Boot applications offer slightly faster execution. This is rarely reason enough to rehydrate, but it is a nice bonus.
- Extracted dependencies are easily replaceable discrete units. App updates can be done more quickly and/or with lower bandwidth because only the changed files must be redeployed.
- Many cloud platforms, such as Heroku and any build or brand/derivative of Cloud Foundry, do this as part of the app deployment process. Mirroring local and remote environments to the maximum extent possible can aid in consistency and, when necessary, diagnosis of any issues.

Both standard Spring Boot executable JARs and "fully executable" JARs can be rehydrated in the following manner, using `jar -xvf <spring_boot_jar>` (most file entries removed for brevity):

```
» mkdir expanded
» cd expanded
» jar -xvf ../target/planefinder-0.0.1-SNAPSHOT.jar
```

```
  created: META-INF/
 inflated: META-INF/MANIFEST.MF
  created: org/
  created: org/springframework/
  created: org/springframework/boot/
  created: org/springframework/boot/loader/
  created: org/springframework/boot/loader/archive/
  created: org/springframework/boot/loader/data/
  created: org/springframework/boot/loader/jar/
  created: org/springframework/boot/loader/jarmode/
  created: org/springframework/boot/loader/util/
  created: BOOT-INF/
  created: BOOT-INF/classes/
  created: BOOT-INF/classes/com/
  created: BOOT-INF/classes/com/thehecklers/
  created: BOOT-INF/classes/com/thehecklers/planefinder/
  created: META-INF/maven/
  created: META-INF/maven/com.thehecklers/
  created: META-INF/maven/com.thehecklers/planefinder/
 inflated: BOOT-INF/classes/schema.sql
 inflated: BOOT-INF/classes/application.properties
 inflated: META-INF/maven/com.thehecklers/planefinder/pom.xml
 inflated: META-INF/maven/com.thehecklers/planefinder/pom.properties
  created: BOOT-INF/lib/
 inflated: BOOT-INF/classpath.idx
 inflated: BOOT-INF/layers.idx
»
```

Once files are extracted, I find it useful to examine the structure a bit more visually using the *nix `tree` command:

```
» tree
.
├── BOOT-INF
│   ├── classes
│   │   ├── application.properties
│   │   ├── com
│   │   │   └── thehecklers
│   │   │       └── planefinder
│   │   │           ├── Aircraft.class
│   │   │           ├── DbConxInit.class
│   │   │           ├── PlaneController.class
│   │   │           ├── PlaneFinderService.class
│   │   │           ├── PlaneRepository.class
│   │   │           ├── PlanefinderApplication.class
│   │   │           └── User.class
│   │   └── schema.sql
│   ├── classpath.idx
│   ├── layers.idx
│   └── lib
```

```
│           ├── h2-1.4.200.jar
│           ├── jackson-annotations-2.11.3.jar
│           ├── jackson-core-2.11.3.jar
│           ├── jackson-databind-2.11.3.jar
│           ├── jackson-dataformat-cbor-2.11.3.jar
│           ├── jackson-datatype-jdk8-2.11.3.jar
│           ├── jackson-datatype-jsr310-2.11.3.jar
│           ├── jackson-module-parameter-names-2.11.3.jar
│           ├── jakarta.annotation-api-1.3.5.jar
│           ├── jul-to-slf4j-1.7.30.jar
│           ├── log4j-api-2.13.3.jar
│           ├── log4j-to-slf4j-2.13.3.jar
│           ├── logback-classic-1.2.3.jar
│           ├── logback-core-1.2.3.jar
│           ├── lombok-1.18.16.jar
│           ├── netty-buffer-4.1.54.Final.jar
│           ├── netty-codec-4.1.54.Final.jar
│           ├── netty-codec-dns-4.1.54.Final.jar
│           ├── netty-codec-http-4.1.54.Final.jar
│           ├── netty-codec-http2-4.1.54.Final.jar
│           ├── netty-codec-socks-4.1.54.Final.jar
│           ├── netty-common-4.1.54.Final.jar
│           ├── netty-handler-4.1.54.Final.jar
│           ├── netty-handler-proxy-4.1.54.Final.jar
│           ├── netty-resolver-4.1.54.Final.jar
│           ├── netty-resolver-dns-4.1.54.Final.jar
│           ├── netty-transport-4.1.54.Final.jar
│           ├── netty-transport-native-epoll-4.1.54.Final-linux-x86_64.jar
│           ├── netty-transport-native-unix-common-4.1.54.Final.jar
│           ├── r2dbc-h2-0.8.4.RELEASE.jar
│           ├── r2dbc-pool-0.8.5.RELEASE.jar
│           ├── r2dbc-spi-0.8.3.RELEASE.jar
│           ├── reactive-streams-1.0.3.jar
│           ├── reactor-core-3.4.0.jar
│           ├── reactor-netty-core-1.0.1.jar
│           ├── reactor-netty-http-1.0.1.jar
│           ├── reactor-pool-0.2.0.jar
│           ├── rsocket-core-1.1.0.jar
│           ├── rsocket-transport-netty-1.1.0.jar
│           ├── slf4j-api-1.7.30.jar
│           ├── snakeyaml-1.27.jar
│           ├── spring-aop-5.3.1.jar
│           ├── spring-beans-5.3.1.jar
│           ├── spring-boot-2.4.0.jar
│           ├── spring-boot-autoconfigure-2.4.0.jar
│           ├── spring-boot-jarmode-layertools-2.4.0.jar
│           ├── spring-context-5.3.1.jar
│           ├── spring-core-5.3.1.jar
│           ├── spring-data-commons-2.4.1.jar
│           ├── spring-data-r2dbc-1.2.1.jar
│           ├── spring-data-relational-2.1.1.jar
│           ├── spring-expression-5.3.1.jar
```

```
│       ├── spring-jcl-5.3.1.jar
│       ├── spring-messaging-5.3.1.jar
│       ├── spring-r2dbc-5.3.1.jar
│       ├── spring-tx-5.3.1.jar
│       ├── spring-web-5.3.1.jar
│       └── spring-webflux-5.3.1.jar
├── META-INF
│   ├── MANIFEST.MF
│   └── maven
│       └── com.thehecklers
│           └── planefinder
│               ├── pom.properties
│               └── pom.xml
└── org
    └── springframework
        └── boot
            └── loader
                ├── ClassPathIndexFile.class
                ├── ExecutableArchiveLauncher.class
                ├── JarLauncher.class
                ├── LaunchedURLClassLoader$DefinePackageCallType.class
                ├── LaunchedURLClassLoader
                    $UseFastConnectionExceptionsEnumeration.class
                ├── LaunchedURLClassLoader.class
                ├── Launcher.class
                ├── MainMethodRunner.class
                ├── PropertiesLauncher$1.class
                ├── PropertiesLauncher$ArchiveEntryFilter.class
                ├── PropertiesLauncher$ClassPathArchives.class
                ├── PropertiesLauncher$PrefixMatchingArchiveFilter.class
                ├── PropertiesLauncher.class
                ├── WarLauncher.class
                ├── archive
                │   ├── Archive$Entry.class
                │   ├── Archive$EntryFilter.class
                │   ├── Archive.class
                │   ├── ExplodedArchive$AbstractIterator.class
                │   ├── ExplodedArchive$ArchiveIterator.class
                │   ├── ExplodedArchive$EntryIterator.class
                │   ├── ExplodedArchive$FileEntry.class
                │   ├── ExplodedArchive$SimpleJarFileArchive.class
                │   ├── ExplodedArchive.class
                │   ├── JarFileArchive$AbstractIterator.class
                │   ├── JarFileArchive$EntryIterator.class
                │   ├── JarFileArchive$JarFileEntry.class
                │   ├── JarFileArchive$NestedArchiveIterator.class
                │   └── JarFileArchive.class
                ├── data
                │   ├── RandomAccessData.class
                │   ├── RandomAccessDataFile$1.class
                │   ├── RandomAccessDataFile$DataInputStream.class
                │   ├── RandomAccessDataFile$FileAccess.class
```

```
        │   └── RandomAccessDataFile.class
        ├── jar
        │   ├── AbstractJarFile$JarFileType.class
        │   ├── AbstractJarFile.class
        │   ├── AsciiBytes.class
        │   ├── Bytes.class
        │   ├── CentralDirectoryEndRecord$1.class
        │   ├── CentralDirectoryEndRecord$Zip64End.class
        │   ├── CentralDirectoryEndRecord$Zip64Locator.class
        │   ├── CentralDirectoryEndRecord.class
        │   ├── CentralDirectoryFileHeader.class
        │   ├── CentralDirectoryParser.class
        │   ├── CentralDirectoryVisitor.class
        │   ├── FileHeader.class
        │   ├── Handler.class
        │   ├── JarEntry.class
        │   ├── JarEntryCertification.class
        │   ├── JarEntryFilter.class
        │   ├── JarFile$1.class
        │   ├── JarFile$JarEntryEnumeration.class
        │   ├── JarFile.class
        │   ├── JarFileEntries$1.class
        │   ├── JarFileEntries$EntryIterator.class
        │   ├── JarFileEntries.class
        │   ├── JarFileWrapper.class
        │   ├── JarURLConnection$1.class
        │   ├── JarURLConnection$JarEntryName.class
        │   ├── JarURLConnection.class
        │   ├── StringSequence.class
        │   └── ZipInflaterInputStream.class
        ├── jarmode
        │   ├── JarMode.class
        │   ├── JarModeLauncher.class
        │   └── TestJarMode.class
        └── util
            └── SystemPropertyUtils.class

19 directories, 137 files
»
```

Viewing the JAR contents using `tree` offers a nice hierarchical display of the application's composition. It also calls out the numerous dependencies that combine to provide the capabilities chosen for this application. Listing the files under *BOOT-INF/lib* confirms that the component libraries remain unchanged through the building of the Spring Boot JAR and subsequent extraction of its contents, even down to original component JAR timestamps, as shown here (most entries removed for brevity):

```
» ls -l BOOT-INF/lib
total 52880
```

```
-rw-r--r--  1 markheckler  staff  2303679 Oct 14  2019 h2-1.4.200.jar
-rw-r--r--  1 markheckler  staff    68215 Oct  1 22:20 jackson-annotations-
 2.11.3.jar
-rw-r--r--  1 markheckler  staff   351495 Oct  1 22:25 jackson-core-
 2.11.3.jar
-rw-r--r--  1 markheckler  staff  1421699 Oct  1 22:38 jackson-databind-
 2.11.3.jar
-rw-r--r--  1 markheckler  staff    58679 Oct  2 00:17 jackson-dataformat-cl
 2.11.3.jar
-rw-r--r--  1 markheckler  staff    34335 Oct  2 00:25 jackson-datatype-jdk
 2.11.3.jar
-rw-r--r--  1 markheckler  staff   111008 Oct  2 00:25 jackson-datatype-jsr
 2.11.3.jar
-rw-r--r--  1 markheckler  staff     9267 Oct  2 00:25 jackson-module-parame
 names-2.11.3.jar
 ...
-rw-r--r--  1 markheckler  staff   374303 Nov 10 09:01 spring-aop-5.3.1.jar
-rw-r--r--  1 markheckler  staff   695851 Nov 10 09:01 spring-beans-5.3.1.j
-rw-r--r--  1 markheckler  staff  1299025 Nov 12 13:56 spring-boot-2.4.0.ja
-rw-r--r--  1 markheckler  staff  1537971 Nov 12 13:55 spring-boot-
 autoconfigure-2.4.0.jar
-rw-r--r--  1 markheckler  staff    32912 Feb  1  1980 spring-boot-jarmode-
 layertools-2.4.0.jar
-rw-r--r--  1 markheckler  staff  1241939 Nov 10 09:01 spring-context-5.3.1
-rw-r--r--  1 markheckler  staff  1464734 Feb  1  1980 spring-core-5.3.1.ja
-rw-r--r--  1 markheckler  staff  1238966 Nov 11 12:03 spring-data-commons-
 2.4.1.jar
-rw-r--r--  1 markheckler  staff   433079 Nov 11 12:08 spring-data-r2dbc-
 1.2.1.jar
-rw-r--r--  1 markheckler  staff   339745 Nov 11 12:05 spring-data-relation
 2.1.1.jar
-rw-r--r--  1 markheckler  staff   282565 Nov 10 09:01 spring-expression-
 5.3.1.jar
-rw-r--r--  1 markheckler  staff    23943 Nov 10 09:01 spring-jcl-5.3.1.jar
-rw-r--r--  1 markheckler  staff   552895 Nov 10 09:01 spring-messaging-
 5.3.1.jar
-rw-r--r--  1 markheckler  staff   133156 Nov 10 09:01 spring-r2dbc-5.3.1.j
-rw-r--r--  1 markheckler  staff   327956 Nov 10 09:01 spring-tx-5.3.1.jar
-rw-r--r--  1 markheckler  staff  1546053 Nov 10 09:01 spring-web-5.3.1.jar
-rw-r--r--  1 markheckler  staff   901591 Nov 10 09:01 spring-webflux-5.3.1
»
```

Once all files are extracted from the Spring Boot JAR, there are a few ways to run the application. The recommended approach is to use `JarLauncher`, which maintains a consistent classloading order across executions, as shown below (results trimmed and edited to fit page):

```
» java org.springframework.boot.loader.JarLauncher

  .   ____          _            __ _ _
```

```
  /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
 ( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
  \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
   '  |____| .__|_| |_|_| |_\__, | / / / /
  =========|_|==============|___/=/_/_/_/
  :: Spring Boot ::                (v2.4.0)

 : Starting PlanefinderApplication v0.0.1-SNAPSHOT
 : No active profile set, falling back to default profiles: default
 : Bootstrapping Spring Data R2DBC repositories in DEFAULT mode.
 : Finished Spring Data repository scanning in 95 ms. Found 1 R2DBC
   repository interfaces.
 : Netty started on port(s): 7634
 : Netty RSocket started on port(s): 7635
 : Started PlanefinderApplication in 1.935 seconds (JVM running for 2.213)
```

In this case, PlaneFinder started just over a full second faster expanded than in the Spring Boot "fully executable" JAR. This positive alone may or may not outweigh the advantages of a single, fully self-contained deployable unit; it likely will not. But combined with the ability to only push deltas when a small number of files change and (if applicable) better alignment between local and remote environments, the ability to run exploded Spring Boot applications can be a very useful option.

Deploying Spring Boot Applications to Containers

As mentioned earlier, some cloud platforms—both on-premises/private and public cloud—take deployable applications and create a container image on a developer's behalf using widely optimized defaults and settings provided by the app's developer. These images are then used to create (and destroy) containers with the running application based on the application's replication settings and utilization. Platforms like Heroku and numerous versions of Cloud Foundry enable a developer to push a Spring Boot executable JAR, and provide any desired configuration settings (or simply accept the defaults), and the rest is handled by the platform. Other platforms like VMware's Tanzu Application Service for Kubernetes incorporate this as well, and the feature list is increasing in both scope and fluid execution.

There are many platforms and deployment targets that don't support this level of frictionless developer enablement. Whether you or your organization has committed to one of those other offerings, or if you have other requirements that guide you in a different direction, Spring Boot has you covered.

While you can handcraft your own container images for your Spring Boot applications, it isn't optimal; doing so adds no value to the application itself and has usually been considered a necessary evil (at best) to go from dev to prod. No more.

Leveraging many of the same tools used by the previously mentioned platforms to intelligently containerize applications, Spring Boot incorporates within its Maven and Gradle plug-ins the capability to build painlessly and frictionlessly fully compliant Open Container Initiative (OCI) images used by Docker, Kubernetes, and every major container engine/mechanism. Built upon industry-leading [Cloud Native Buildpacks] and the [Paketo] buildpacks initiative, the Spring Boot build plug-ins provide the option to create an OCI image using a locally installed and locally running Docker daemon and push it to a local or designated remote image repository.

Using the Spring Boot plug-in to create an image from your application is opinionated in all the best ways as well, using a conceptual "autoconfiguration" to optimize image creation by layering image contents, separating code/libraries based on each code unit's anticipated frequency of change. Staying true to the Spring Boot philosophy behind autoconfiguration and opinions, Boot also provides a way to override and guide the layering process should you need to customize your configuration. This is rarely necessary or even desirable, but it's easily accomplished should your needs fall within one of those rare, exceptional cases.

The default settings produce the following layers for all versions of Spring Boot from 2.4.0 Milestone 2 onward:

`dependencies`
: Includes regularly released dependencies, i.e., GA versions

`spring-boot-loader`
: Includes all files found under *org/springframework/boot/loader*

`snapshot-dependencies`
: Forward-looking releases not yet considered GA

`application`
: Application classes and related resources (templates, properties files, scripts, etc.)

Code volatility, or its propensity and frequency of change, typically increases as you move through this list of layers from top to bottom. By creating separate layers in which to place similarly volatile code, subsequent image creation is much more efficient and thus much faster to complete. This *drastically* reduces the time and resources required to rebuild the deployable artifact over the life of the application.

Creating a Container Image from an IDE

Creating a layered container image from a Spring Boot application can be done from within an IDE very easily. I use IntelliJ for this example, but nearly all major IDEs have similar capabilities.

NOTE

A local version of Docker—Docker Desktop for Mac in my case—must be running to create images.

To create the image, I open the Maven panel by expanding the tab labeled *Maven* in IntelliJ's right margin, then expand *Plugins*, choose and expand the *spring-boot* plug-in, and double-click the *spring-boot:build-image* option to execute the goal, as shown in Figure 11-2.

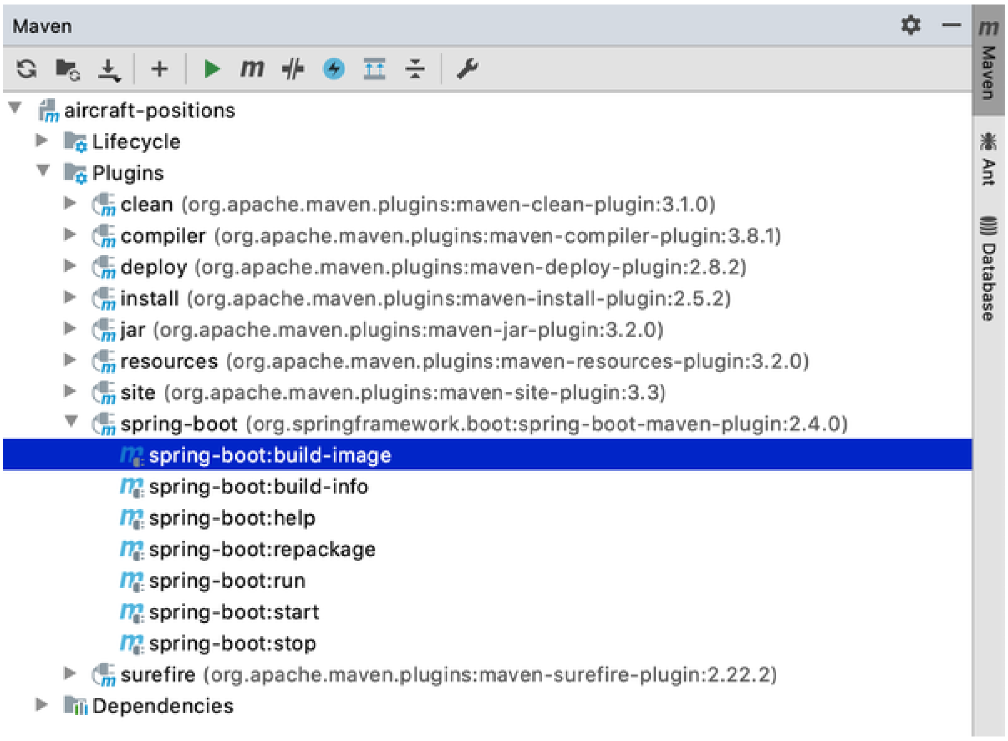

Figure 11-2. Building a Spring Boot application container image from IntelliJ's Maven panel

Creating the image produces a rather lengthy log of actions. Of particular interest are entries listed here:

```
[INFO]     [creator]     Paketo Executable JAR Buildpack 3.1.3
[INFO]     [creator]       https://github.com/paketo-buildpacks/executable-
[INFO]     [creator]         Writing env.launch/CLASSPATH.delim
```

```
[INFO]     [creator]         Writing env.launch/CLASSPATH.prepend
[INFO]     [creator]       Process types:
[INFO]     [creator]         executable-jar: java org.springframework.boot.
    loader.JarLauncher
[INFO]     [creator]         task:           java org.springframework.boot.
    loader.JarLauncher
[INFO]     [creator]         web:            java org.springframework.boot.
    loader.JarLauncher
[INFO]     [creator]
[INFO]     [creator]     Paketo Spring Boot Buildpack 3.5.0
[INFO]     [creator]       https://github.com/paketo-buildpacks/spring-boot
[INFO]     [creator]       Creating slices from layers index
[INFO]     [creator]         dependencies
[INFO]     [creator]         spring-boot-loader
[INFO]     [creator]         snapshot-dependencies
[INFO]     [creator]         application
[INFO]     [creator]       Launch Helper: Contributing to layer
[INFO]     [creator]         Creating /layers/paketo-buildpacks_spring-boot
    helper/exec.d/spring-cloud-bindings
[INFO]     [creator]         Writing profile.d/helper
[INFO]     [creator]       Web Application Type: Contributing to layer
[INFO]     [creator]         Reactive web application detected
[INFO]     [creator]         Writing env.launch/BPL_JVM_THREAD_COUNT.defaul
[INFO]     [creator]       Spring Cloud Bindings 1.7.0: Contributing to lay
[INFO]     [creator]         Downloading from
    https://repo.spring.io/release/org/springframework/cloud/
    spring-cloud-bindings/1.7.0/spring-cloud-bindings-1.7.0.jar
[INFO]     [creator]         Verifying checksum
[INFO]     [creator]         Copying to
    /layers/paketo-buildpacks_spring-boot/spring-cloud-bindings
[INFO]     [creator]       4 application slices
```

As noted earlier, image layers (referred to as *slices* in the preceding listing) and their contents can be modified if necessary for unique circumstances.

Once the image has been created, results like those that follow will complete the log.

```
[INFO] Successfully built image 'docker.io/library/aircraft-positions:
       0.0.1-SNAPSHOT'
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  25.851 s
[INFO] Finished at: 2020-11-28T20:09:48-06:00
[INFO] ------------------------------------------------------------------------
```

Creating a Container Image from the Command Line

It's of course also possible—and simple—to create the same container image from the command line. Prior to doing so, I do want to make a small change to the naming settings for the resultant image.

As a matter of convenience, I prefer to create images that align with my Docker Hub account and naming conventions, and your choice of image repository likely has similar, specific conventions. Spring Boot's build plug-ins accept <*configuration*> section details that smooth the step of pushing the image to repository/catalog. I add a single, properly tagged line to the <*plug-ins*> section of `Aircraft Position` 's *pom.xml* file to match my requirements/preferences:

```
<build>
  <plug-ins>
    <plug-in>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-maven-plug-in</artifactId>
      <configuration>
        <image>
          <name>hecklerm/${project.artifactId}</name>
        </image>
      </configuration>
    </plug-in>
  </plug-ins>
</build>
```

Next, I issue the following command from the project directory in the terminal window to re-create the application container image and soon thereafter receive the results shown:

```
» mvn spring-boot:build-image


... (Intermediate logged results omitted for brevity)


[INFO] Successfully built image 'docker.io/hecklerm/aircraft-positions:late
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  13.257 s
[INFO] Finished at: 2020-11-28T20:23:40-06:00
[INFO] ------------------------------------------------------------------------
```

Notice that the image output is no longer *docker.io/library/aircraft-positions:0.0.1-SNAPSHOT* as it was when I built it using defaults from within the IDE. The new image coordinates match those I specified in the *pom.xml*: *docker.io/hecklerm/aircraft-positions:latest*.

Verifying the Image Exists

To verify that the images created in the prior two sections have been loaded into the local repository, I run the following command from the terminal window, filtering by name to get the following results (and trimmed to fit page):

```
» docker images | grep -in aircraft-positions
aircraft-positions            0.0.1-SNAPSHOT   a7ed39a3d52e    277MB
hecklerm/aircraft-positions  latest           924893a0f1a9    277MB
```

Pushing the image shown last in the preceding output—since it now aligns with expected and desired account and naming conventions—to Docker Hub is accomplished as follows, with the following results:

```
» docker push hecklerm/aircraft-positions
The push refers to repository [docker.io/hecklerm/aircraft-positions]
1dc94a70dbaa: Pushed
4672559507f8: Pushed
e3e9839150af: Pushed
5f70bf18a086: Layer already exists
a3abfb734aa5: Pushed
3c14fe2f1177: Pushed
4cc7b4eb8637: Pushed
fcc507beb4cc: Pushed
c2e9ddddd4ef: Pushed
108b6855c4a6: Pushed
ab39aa8fd003: Layer already exists
0b18b1f120f4: Layer already exists
cf6b3a71f979: Pushed
ec0381c8f321: Layer already exists
7b0fc1578394: Pushed
eb0f7cd0acf8: Pushed
1e5c1d306847: Mounted from paketobuildpacks/run
23c4345364c2: Mounted from paketobuildpacks/run
a1efa53a237c: Mounted from paketobuildpacks/run
fe6d8881187d: Mounted from paketobuildpacks/run
23135df75b44: Mounted from paketobuildpacks/run
b43408d5f11b: Mounted from paketobuildpacks/run
latest: digest:
  sha256:a7e5d536a7426d6244401787b153ebf43277fbadc9f43a789f6c4f0aff6d5011
```

```
    size: 5122
»
```

Visiting the Docker Hub allows me to confirm successful public deployment of the image, as shown in Figure 11-3.

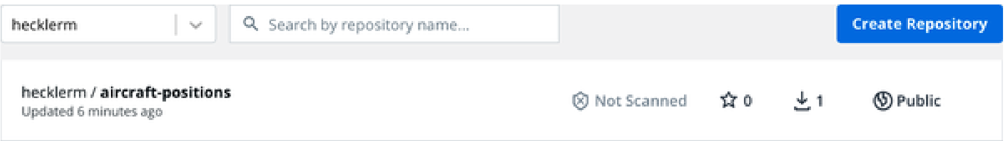

Figure 11-3. Spring Boot application container image in Docker Hub

Deploying to Docker Hub or any other container image repository available from outside of your local machine is the last step prior to wider (and hopefully production) deployment of your Spring Boot containerized application.

Running the Containerized Application

To run the application, I use the `docker run` command. Your organization likely has a deployment pipeline that moves applications from container images (retrieved from image repositories) to running, containerized applications, but the steps performed are likely the same, albeit with more automation and less typing.

Since I already have a local copy of the image, no remote retrieval will be necessary; otherwise, remote access to the image repository is required for the daemon to retrieve the remote image and/or layers to reconstruct it locally prior to starting a container based upon the image specified.

To run the containerized Aircraft Positions application, I execute the following command and see the following results (trimmed and edited to fit page):

```
» docker run --name myaircraftpositions -p8080:8080
  hecklerm/aircraft-positions:latest
Setting Active Processor Count to 6
WARNING: Container memory limit unset. Configuring JVM for 1G container.
Calculated JVM Memory Configuration: -XX:MaxDirectMemorySize=10M -Xmx636688
  -XX:MaxMetaspaceSize=104687K -XX:ReservedCodeCacheSize=240M -Xss1M
  (Total Memory: 1G, Thread Count: 50, Loaded Class Count: 16069, Headroom:
Adding 138 container CA certificates to JVM truststore
Spring Cloud Bindings Enabled
Picked up JAVA_TOOL_OPTIONS:
-Djava.security.properties=/layers/paketo-buildpacks_bellsoft-liberica/
    java-security-properties/java-security.properties
  -agentpath:/layers/paketo-buildpacks_bellsoft-liberica/jvmkill/
```

```
     jvmkill-1.16.0-RELEASE.so=printHeapHistogram=1
   -XX:ActiveProcessorCount=6
   -XX:MaxDirectMemorySize=10M
   -Xmx636688K
   -XX:MaxMetaspaceSize=104687K
   -XX:ReservedCodeCacheSize=240M
   -Xss1M
   -Dorg.springframework.cloud.bindings.boot.enable=true

   .   ____          _            __ _ _
  /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
 ( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
  \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
   '  |____| .__|_| |_|_| |_\__, | / / / /
  =========|_|==============|___/=/_/_/_/
  :: Spring Boot ::                (v2.4.0)

 : Starting AircraftPositionsApplication v0.0.1-SNAPSHOT
 : Netty started on port(s): 8080
 : Started AircraftPositionsApplication in 10.7 seconds (JVM running for 11.
```

Now to take a quick look inside a Spring Boot plug-in-created image.

Utilities for Examining Spring Boot Application Container Images

Numerous utilities exist to work with container images, and much of the functionality provided by them falls well outside the scope of this book. I do want to briefly mention two that I've found useful in certain circumstances: `pack` and `dive`.

Pack

To examine the materials that go into the creation of a Spring Boot application container image using Cloud Native (Paketo) Buildpacks—and the buildpacks themselves—one can use the `pack` utility. `pack` is the designated CLI for building apps using Cloud Native Buildpacks and can be obtained by various means. I used `homebrew` to retrieve and install it with a simple `brew install pack` command on my Mac.

Running `pack` against the image created previously results in the following:

```
» pack inspect-image hecklerm/aircraft-positions
Inspecting image: hecklerm/aircraft-positions
```

```
REMOTE:

Stack: io.buildpacks.stacks.bionic

Base Image:
  Reference: f5caea10feb38ae882a9447b521fd1ea1ee93384438395c7ace2d8cfaf808e
  Top Layer: sha256:1e5c1d306847275caa0d1d367382dfdcfd4d62b634b237f1d7a2e
             746372922cd

Run Images:
  index.docker.io/paketobuildpacks/run:base-cnb
  gcr.io/paketo-buildpacks/run:base-cnb

Buildpacks:
  ID                                           VERSION
  paketo-buildpacks/ca-certificates            1.0.1
  paketo-buildpacks/bellsoft-liberica          5.2.1
  paketo-buildpacks/executable-jar             3.1.3
  paketo-buildpacks/dist-zip                   2.2.2
  paketo-buildpacks/spring-boot                3.5.0

Processes:
  TYPE           SHELL  COMMAND  ARGS
  web (default)  bash   java     org.springframework.boot.loader.JarLaunche
  executable-jar bash   java     org.springframework.boot.loader.JarLaunche
  task           bash   java     org.springframework.boot.loader.JarLaunche


LOCAL:

Stack: io.buildpacks.stacks.bionic

Base Image:
  Reference: f5caea10feb38ae882a9447b521fd1ea1ee93384438395c7ace2d8cfaf808e
  Top Layer: sha256:1e5c1d306847275caa0d1d367382dfdcfd4d62b634b237f1d7a2e
             746372922cd

Run Images:
  index.docker.io/paketobuildpacks/run:base-cnb
  gcr.io/paketo-buildpacks/run:base-cnb

Buildpacks:
  ID                                           VERSION
  paketo-buildpacks/ca-certificates            1.0.1
  paketo-buildpacks/bellsoft-liberica          5.2.1
  paketo-buildpacks/executable-jar             3.1.3
  paketo-buildpacks/dist-zip                   2.2.2
  paketo-buildpacks/spring-boot                3.5.0

Processes:
  TYPE           SHELL  COMMAND  ARGS
  web (default)  bash   java     org.springframework.boot.loader.JarLaunche
```

```
    executable-jar bash    java     org.springframework.boot.loader.JarLaunche
    task           bash    java     org.springframework.boot.loader.JarLaunche
```

Using the `pack` utility's `inspect-image` command provides some key bits of information about the image, particularly the following:

- Which Docker base image/Linux version (bionic) was used as the foundation for this image
- Which buildpacks were used to populate the image (five Paketo buildpacks listed)
- What processes will be run and by what means (Java commands executed by the shell)

Note that both local and remote connected repositories are polled for the specified image, and details are provided for both. This is particularly helpful in diagnosing issues caused by an out-of-date container image in one location or the other.

Dive

The `dive` utility was created by Alex Goodman as a way to "dive" into a container image, viewing the very granular OCI image layers and the tree structure of the entire image filesystem.

`dive` goes far beneath the application-level layers of the Spring Boot layering construct and into the operating system. I find it less useful than `pack` due to its focus on the OS versus the application, but it's ideal for verifying the presence or absence of particular files, file permissions, and other essential low-level concerns. It's a rarely used tool but essential when that level of detail and control is needed.

CODE CHECKOUT CHECKUP

For complete chapter code, please check out branch *chapter11end* from the code repository.

Summary

Until an application's users can actually use that application, it is little more than a what-if exercise. Figuratively and often very literally, deployment is the payoff.

Many developers are aware that Spring Boot applications can be created as WAR files or JAR files. Most of those developers also know that there are many good reasons to skip the WAR option and create executable JAR files and few good reasons to do the opposite. What many developers may not realize is that even when building a Spring Boot executable JAR, there are numerous options for deployment to fulfill various requirements and use cases.

In this chapter, I examined several ways to deploy your Spring Boot application with options useful for different target destinations and discussed their relative merits. I then demonstrated how to create those deployment artifacts, explained options for optimal execution, and showed how to verify their components and provenance. Targets included the standard Spring Boot executable JARs, “fully executable” Spring Boot JARs, exploded/expanded JARs, and container images built using Cloud Native (Paketo) Buildpacks that run on Docker, Kubernetes, and every major container engine/mechanism. Spring Boot gives you numerous frictionless deployment options, extending your development superpowers into deployment superpowers as well.

In the next and final chapter, I round out this book and journey by delving a bit further into two slightly deeper topics. If you’d like to know more about testing and debugging reactive applications, you won’t want to miss it.