**Deploying Spring Boot Applications**

[Back to index](https://docs.spring.io/spring-boot/docs/current/reference/html/index.html)

1. [1. Deploying to Containers](https://docs.spring.io/spring-boot/docs/current/reference/html/deployment.html#containers-deployment)
2. [2. Deploying to the Cloud](https://docs.spring.io/spring-boot/docs/current/reference/html/deployment.html#cloud-deployment)
   1. [2.1. Cloud Foundry](https://docs.spring.io/spring-boot/docs/current/reference/html/deployment.html#cloud-deployment-cloud-foundry)
      1. [2.1.1. Binding to Services](https://docs.spring.io/spring-boot/docs/current/reference/html/deployment.html#cloud-deployment-cloud-foundry-services)
   2. [2.2. Heroku](https://docs.spring.io/spring-boot/docs/current/reference/html/deployment.html#cloud-deployment-heroku)
   3. [2.3. OpenShift](https://docs.spring.io/spring-boot/docs/current/reference/html/deployment.html#cloud-deployment-openshift)
   4. [2.4. Amazon Web Services (AWS)](https://docs.spring.io/spring-boot/docs/current/reference/html/deployment.html#cloud-deployment-aws)
      1. [2.4.1. AWS Elastic Beanstalk](https://docs.spring.io/spring-boot/docs/current/reference/html/deployment.html#aws-elastic-beanstalk)
         1. [Using the Tomcat Platform](https://docs.spring.io/spring-boot/docs/current/reference/html/deployment.html#using-the-tomcat-platform)
         2. [Using the Java SE Platform](https://docs.spring.io/spring-boot/docs/current/reference/html/deployment.html#using-the-java-se-platform)
      2. [2.4.2. Summary](https://docs.spring.io/spring-boot/docs/current/reference/html/deployment.html#summary)
   5. [2.5. Boxfuse and Amazon Web Services](https://docs.spring.io/spring-boot/docs/current/reference/html/deployment.html#cloud-deployment-boxfuse)
   6. [2.6. Google Cloud](https://docs.spring.io/spring-boot/docs/current/reference/html/deployment.html#cloud-deployment-gae)
3. [3. Installing Spring Boot Applications](https://docs.spring.io/spring-boot/docs/current/reference/html/deployment.html#deployment-install)
   1. [3.1. Supported Operating Systems](https://docs.spring.io/spring-boot/docs/current/reference/html/deployment.html#deployment-install-supported-operating-systems)
   2. [3.2. Unix/Linux Services](https://docs.spring.io/spring-boot/docs/current/reference/html/deployment.html#deployment-service)
      1. [3.2.1. Installation as an init.d Service (System V)](https://docs.spring.io/spring-boot/docs/current/reference/html/deployment.html#deployment-initd-service)
         1. [Securing an init.d Service](https://docs.spring.io/spring-boot/docs/current/reference/html/deployment.html#deployment-initd-service-securing)
      2. [3.2.2. Installation as a systemd Service](https://docs.spring.io/spring-boot/docs/current/reference/html/deployment.html#deployment-systemd-service)
      3. [3.2.3. Customizing the Startup Script](https://docs.spring.io/spring-boot/docs/current/reference/html/deployment.html#deployment-script-customization)
         1. [Customizing the Start Script when It Is Written](https://docs.spring.io/spring-boot/docs/current/reference/html/deployment.html#deployment-script-customization-when-it-written)
         2. [Customizing a Script When It Runs](https://docs.spring.io/spring-boot/docs/current/reference/html/deployment.html#deployment-script-customization-when-it-runs)
   3. [3.3. Microsoft Windows Services](https://docs.spring.io/spring-boot/docs/current/reference/html/deployment.html#deployment-windows)
4. [4. What to Read Next](https://docs.spring.io/spring-boot/docs/current/reference/html/deployment.html#deployment-whats-next)

Spring Boot’s flexible packaging options provide a great deal of choice when it comes to deploying your application. You can deploy Spring Boot applications to a variety of cloud platforms, to container images (such as Docker), or to virtual/real machines.

This section covers some of the more common deployment scenarios.

**1. Deploying to Containers**

If you are running your application from a container, you can use an executable jar, but it is also often an advantage to explode it and run it in a different way. Certain PaaS implementations may also choose to unpack archives before they run. For example, Cloud Foundry operates this way. The simplest way to run an unpacked archive is by starting the appropriate launcher, as follows:

$ jar -xf myapp.jar

$ java org.springframework.boot.loader.JarLauncher

This is actually slightly faster on startup (depending on the size of the jar) than running from an unexploded archive. At runtime you shouldn’t expect any differences.

Once you have unpacked the jar file, you can also get an extra boost to startup time by running the app with its "natural" main method instead of the JarLauncher. For example:

$ jar -xf myapp.jar

$ java -cp BOOT-INF/classes:BOOT-INF/lib/\* com.example.MyApplication

More efficient container images can also be created by copying the dependencies to the image as a separate layer from the application classes and resources (which normally change more frequently). There is more than one way to achieve this layer separation. For example, using a Dockerfile you could express it in this form:

FROM openjdk:8-jdk-alpine AS builder

WORKDIR target/dependency

ARG APPJAR=target/\*.jar

COPY ${APPJAR} app.jar

RUN jar -xf ./app.jar

FROM openjdk:8-jre-alpine

VOLUME /tmp

ARG DEPENDENCY=target/dependency

COPY --from=builder ${DEPENDENCY}/BOOT-INF/lib /app/lib

COPY --from=builder ${DEPENDENCY}/META-INF /app/META-INF

COPY --from=builder ${DEPENDENCY}/BOOT-INF/classes /app

ENTRYPOINT ["java","-cp","app:app/lib/\*","com.example.MyApplication"]

Assuming the above Dockerfile is the current directory, your docker image can be built with docker build ., or optionally specifying the path to your application jar, as shown in the following example:

docker build --build-arg APPJAR=path/to/myapp.jar .

**2. Deploying to the Cloud**

Spring Boot’s executable jars are ready-made for most popular cloud PaaS (Platform-as-a-Service) providers. These providers tend to require that you “bring your own container”. They manage application processes (not Java applications specifically), so they need an intermediary layer that adapts *your* application to the *cloud’s* notion of a running process.

Two popular cloud providers, Heroku and Cloud Foundry, employ a “buildpack” approach. The buildpack wraps your deployed code in whatever is needed to *start* your application. It might be a JDK and a call to java, an embedded web server, or a full-fledged application server. A buildpack is pluggable, but ideally you should be able to get by with as few customizations to it as possible. This reduces the footprint of functionality that is not under your control. It minimizes divergence between development and production environments.

Ideally, your application, like a Spring Boot executable jar, has everything that it needs to run packaged within it.

In this section, we look at what it takes to get the [simple application that we developed](https://docs.spring.io/spring-boot/docs/current/reference/html/getting-started.html#getting-started-first-application) in the “Getting Started” section up and running in the Cloud.

**2.1. Cloud Foundry**

Cloud Foundry provides default buildpacks that come into play if no other buildpack is specified. The Cloud Foundry [Java buildpack](https://github.com/cloudfoundry/java-buildpack) has excellent support for Spring applications, including Spring Boot. You can deploy stand-alone executable jar applications as well as traditional .war packaged applications.

Once you have built your application (by using, for example, mvn clean package) and have [installed the cf command line tool](https://docs.cloudfoundry.org/cf-cli/install-go-cli.html), deploy your application by using the cf push command, substituting the path to your compiled .jar. Be sure to have [logged in with your cf command line client](https://docs.cloudfoundry.org/cf-cli/getting-started.html#login) before pushing an application. The following line shows using the cf push command to deploy an application:

$ cf push acloudyspringtime -p target/demo-0.0.1-SNAPSHOT.jar

|  |  |
| --- | --- |
|  | In the preceding example, we substitute acloudyspringtime for whatever value you give cf as the name of your application. |

See the [cf push documentation](https://docs.cloudfoundry.org/cf-cli/getting-started.html#push) for more options. If there is a Cloud Foundry [manifest.yml](https://docs.cloudfoundry.org/devguide/deploy-apps/manifest.html) file present in the same directory, it is considered.

At this point, cf starts uploading your application, producing output similar to the following example:

Uploading acloudyspringtime... **OK**

Preparing to start acloudyspringtime... **OK**

-----> Downloaded app package (**8.9M**)

-----> Java Buildpack Version: v3.12 (offline) | https://github.com/cloudfoundry/java-buildpack.git#6f25b7e

-----> Downloading Open Jdk JRE 1.8.0\_121 from https://java-buildpack.cloudfoundry.org/openjdk/trusty/x86\_64/openjdk-1.8.0\_121.tar.gz (found in cache)

Expanding Open Jdk JRE to .java-buildpack/open\_jdk\_jre (1.6s)

-----> Downloading Open JDK Like Memory Calculator 2.0.2\_RELEASE from https://java-buildpack.cloudfoundry.org/memory-calculator/trusty/x86\_64/memory-calculator-2.0.2\_RELEASE.tar.gz (found in cache)

Memory Settings: -Xss349K -Xmx681574K -XX:MaxMetaspaceSize=104857K -Xms681574K -XX:MetaspaceSize=104857K

-----> Downloading Container Certificate Trust Store 1.0.0\_RELEASE from https://java-buildpack.cloudfoundry.org/container-certificate-trust-store/container-certificate-trust-store-1.0.0\_RELEASE.jar (found in cache)

Adding certificates to .java-buildpack/container\_certificate\_trust\_store/truststore.jks (0.6s)

-----> Downloading Spring Auto Reconfiguration 1.10.0\_RELEASE from https://java-buildpack.cloudfoundry.org/auto-reconfiguration/auto-reconfiguration-1.10.0\_RELEASE.jar (found in cache)

Checking status of app 'acloudyspringtime'...

0 of 1 instances running (1 starting)

...

0 of 1 instances running (1 starting)

...

0 of 1 instances running (1 starting)

...

1 of 1 instances running (1 running)

App started

Congratulations! The application is now live!

Once your application is live, you can verify the status of the deployed application by using the cf apps command, as shown in the following example:

$ cf apps

Getting applications in ...

OK

name requested state instances memory disk urls

...

acloudyspringtime started 1/1 512M 1G acloudyspringtime.cfapps.io

...

Once Cloud Foundry acknowledges that your application has been deployed, you should be able to find the application at the URI given. In the preceding example, you could find it at https://acloudyspringtime.cfapps.io/.

**2.1.1. Binding to Services**

By default, metadata about the running application as well as service connection information is exposed to the application as environment variables (for example: $VCAP\_SERVICES). This architecture decision is due to Cloud Foundry’s polyglot (any language and platform can be supported as a buildpack) nature. Process-scoped environment variables are language agnostic.

Environment variables do not always make for the easiest API, so Spring Boot automatically extracts them and flattens the data into properties that can be accessed through Spring’s Environment abstraction, as shown in the following example:

@Component

class MyBean implements EnvironmentAware {

private String instanceId;

@Override

public void setEnvironment(Environment environment) {

this.instanceId = environment.getProperty("vcap.application.instance\_id");

}

// ...

}

All Cloud Foundry properties are prefixed with vcap. You can use vcap properties to access application information (such as the public URL of the application) and service information (such as database credentials). See the [‘CloudFoundryVcapEnvironmentPostProcessor’](https://docs.spring.io/spring-boot/docs/2.2.2.RELEASE/api/org/springframework/boot/cloud/CloudFoundryVcapEnvironmentPostProcessor.html) Javadoc for complete details.

|  |  |
| --- | --- |
|  | The [Java CFEnv](https://github.com/pivotal-cf/java-cfenv/) project is a better fit for tasks such as configuring a DataSource. |

**2.2. Heroku**

Heroku is another popular PaaS platform. To customize Heroku builds, you provide a Procfile, which provides the incantation required to deploy an application. Heroku assigns a port for the Java application to use and then ensures that routing to the external URI works.

You must configure your application to listen on the correct port. The following example shows the Procfile for our starter REST application:

web: java -Dserver.port=$PORT -jar target/demo-0.0.1-SNAPSHOT.jar

Spring Boot makes -D arguments available as properties accessible from a Spring Environment instance. The server.port configuration property is fed to the embedded Tomcat, Jetty, or Undertow instance, which then uses the port when it starts up. The $PORT environment variable is assigned to us by the Heroku PaaS.

This should be everything you need. The most common deployment workflow for Heroku deployments is to git push the code to production, as shown in the following example:

$ git push heroku master

Initializing repository, **done**.

Counting objects: 95, **done**.

Delta compression using up to 8 threads.

Compressing objects: 100% (78/78), **done**.

Writing objects: 100% (95/95), 8.66 MiB | 606.00 KiB/s, **done**.

Total 95 (delta 31), reused 0 (delta 0)

-----> Java app detected

-----> Installing OpenJDK 1.8... **done**

-----> Installing Maven 3.3.1... **done**

-----> Installing settings.xml... **done**

-----> Executing: mvn -B -DskipTests=true clean install

[INFO] Scanning for projects...

Downloading: https://repo.spring.io/...

Downloaded: https://repo.spring.io/... (818 B at 1.8 KB/sec)

....

Downloaded: https://s3pository.heroku.com/jvm/... (152 KB at 595.3 KB/sec)

[INFO] Installing /tmp/build\_0c35a5d2-a067-4abc-a232-14b1fb7a8229/target/...

[INFO] Installing /tmp/build\_0c35a5d2-a067-4abc-a232-14b1fb7a8229/pom.xml ...

[INFO] ------------------------------------------------------------------------

[INFO] **BUILD SUCCESS**

[INFO] ------------------------------------------------------------------------

[INFO] Total time: 59.358s

[INFO] Finished at: Fri Mar 07 07:28:25 UTC 2014

[INFO] Final Memory: 20M/493M

[INFO] ------------------------------------------------------------------------

-----> Discovering process types

Procfile declares types -> **web**

-----> Compressing... **done**, 70.4MB

-----> Launching... **done**, v6

https://agile-sierra-1405.herokuapp.com/ **deployed to Heroku**

To git@heroku.com:agile-sierra-1405.git

\* [new branch] master -> master

Your application should now be up and running on Heroku. For more details, refer to [Deploying Spring Boot Applications to Heroku](https://devcenter.heroku.com/articles/deploying-spring-boot-apps-to-heroku).

**2.3. OpenShift**

[OpenShift](https://www.openshift.com/) is the Red Hat public (and enterprise) extension of the Kubernetes container orchestration platform. Similarly to Kubernetes, OpenShift has many options for installing Spring Boot based applications.

OpenShift has many resources describing how to deploy Spring Boot applications, including:

* [Using the S2I builder](https://blog.openshift.com/using-openshift-enterprise-grade-spring-boot-deployments/)
* [Architecture guide](https://access.redhat.com/documentation/en-us/reference_architectures/2017/html-single/spring_boot_microservices_on_red_hat_openshift_container_platform_3/)
* [Running as a traditional web application on Wildfly](https://blog.openshift.com/using-spring-boot-on-openshift/)
* [OpenShift Commons Briefing](https://blog.openshift.com/openshift-commons-briefing-96-cloud-native-applications-spring-rhoar/)

**2.4. Amazon Web Services (AWS)**

Amazon Web Services offers multiple ways to install Spring Boot-based applications, either as traditional web applications (war) or as executable jar files with an embedded web server. The options include:

* AWS Elastic Beanstalk
* AWS Code Deploy
* AWS OPS Works
* AWS Cloud Formation
* AWS Container Registry

Each has different features and pricing models. In this document, we describe only the simplest option: AWS Elastic Beanstalk.

**2.4.1. AWS Elastic Beanstalk**

As described in the official [Elastic Beanstalk Java guide](https://docs.aws.amazon.com/elasticbeanstalk/latest/dg/create_deploy_Java.html), there are two main options to deploy a Java application. You can either use the “Tomcat Platform” or the “Java SE platform”.

**Using the Tomcat Platform**

This option applies to Spring Boot projects that produce a war file. No special configuration is required. You need only follow the official guide.

**Using the Java SE Platform**

This option applies to Spring Boot projects that produce a jar file and run an embedded web container. Elastic Beanstalk environments run an nginx instance on port 80 to proxy the actual application, running on port 5000. To configure it, add the following line to your application.properties file:

server.port=5000

|  |  |
| --- | --- |
|  | Upload binaries instead of sources  By default, Elastic Beanstalk uploads sources and compiles them in AWS. However, it is best to upload the binaries instead. To do so, add lines similar to the following to your .elasticbeanstalk/config.yml file:  deploy:  artifact: target/demo-0.0.1-SNAPSHOT.jar |
|  | Reduce costs by setting the environment type  By default an Elastic Beanstalk environment is load balanced. The load balancer has a significant cost. To avoid that cost, set the environment type to “Single instance”, as described in [the Amazon documentation](https://docs.aws.amazon.com/elasticbeanstalk/latest/dg/environments-create-wizard.html#environments-create-wizard-capacity). You can also create single instance environments by using the CLI and the following command:  eb create -s |

**2.4.2. Summary**

This is one of the easiest ways to get to AWS, but there are more things to cover, such as how to integrate Elastic Beanstalk into any CI / CD tool, use the Elastic Beanstalk Maven plugin instead of the CLI, and others. There is a [blog post](https://exampledriven.wordpress.com/2017/01/09/spring-boot-aws-elastic-beanstalk-example/) covering these topics more in detail.

**2.5. Boxfuse and Amazon Web Services**

[Boxfuse](https://boxfuse.com/) works by turning your Spring Boot executable jar or war into a minimal VM image that can be deployed unchanged either on VirtualBox or on AWS. Boxfuse comes with deep integration for Spring Boot and uses the information from your Spring Boot configuration file to automatically configure ports and health check URLs. Boxfuse leverages this information both for the images it produces as well as for all the resources it provisions (instances, security groups, elastic load balancers, and so on).

Once you have created a [Boxfuse account](https://console.boxfuse.com), connected it to your AWS account, installed the latest version of the Boxfuse Client, and ensured that the application has been built by Maven or Gradle (by using, for example, mvn clean package), you can deploy your Spring Boot application to AWS with a command similar to the following:

$ boxfuse run myapp-1.0.jar -env=prod

See the [boxfuse run documentation](https://boxfuse.com/docs/commandline/run.html) for more options. If there is a [boxfuse.conf](https://boxfuse.com/docs/commandline/#configuration) file present in the current directory, it is considered.

|  |  |
| --- | --- |
|  | By default, Boxfuse activates a Spring profile named boxfuse on startup. If your executable jar or war contains an [application-boxfuse.properties](https://boxfuse.com/docs/payloads/springboot.html#configuration) file, Boxfuse bases its configuration on the properties it contains. |

At this point, boxfuse creates an image for your application, uploads it, and configures and starts the necessary resources on AWS, resulting in output similar to the following example:

Fusing Image for myapp-1.0.jar ...

Image fused in 00:06.838s (53937 K) -> axelfontaine/myapp:1.0

Creating axelfontaine/myapp ...

Pushing axelfontaine/myapp:1.0 ...

Verifying axelfontaine/myapp:1.0 ...

Creating Elastic IP ...

Mapping myapp-axelfontaine.boxfuse.io to 52.28.233.167 ...

Waiting for AWS to create an AMI for axelfontaine/myapp:1.0 in eu-central-1 (this may take up to 50 seconds) ...

AMI created in 00:23.557s -> ami-d23f38cf

Creating security group boxfuse-sg\_axelfontaine/myapp:1.0 ...

Launching t2.micro instance of axelfontaine/myapp:1.0 (ami-d23f38cf) in eu-central-1 ...

Instance launched in 00:30.306s -> i-92ef9f53

Waiting for AWS to boot Instance i-92ef9f53 and Payload to start at https://52.28.235.61/ ...

Payload started in 00:29.266s -> https://52.28.235.61/

Remapping Elastic IP 52.28.233.167 to i-92ef9f53 ...

Waiting 15s for AWS to complete Elastic IP Zero Downtime transition ...

Deployment completed successfully. axelfontaine/myapp:1.0 is up and running at https://myapp-axelfontaine.boxfuse.io/

Your application should now be up and running on AWS.

See the blog post on [deploying Spring Boot apps on EC2](https://boxfuse.com/blog/spring-boot-ec2.html) as well as the [documentation for the Boxfuse Spring Boot integration](https://boxfuse.com/docs/payloads/springboot.html) to get started with a Maven build to run the app.

**2.6. Google Cloud**

Google Cloud has several options that can be used to launch Spring Boot applications. The easiest to get started with is probably App Engine, but you could also find ways to run Spring Boot in a container with Container Engine or on a virtual machine with Compute Engine.

To run in App Engine, you can create a project in the UI first, which sets up a unique identifier for you and also sets up HTTP routes. Add a Java app to the project and leave it empty and then use the [Google Cloud SDK](https://cloud.google.com/sdk/install) to push your Spring Boot app into that slot from the command line or CI build.

App Engine Standard requires you to use WAR packaging. Follow [these steps](https://github.com/GoogleCloudPlatform/getting-started-java/blob/master/appengine-standard-java8/springboot-appengine-standard/README.md) to deploy App Engine Standard application to Google Cloud.

Alternatively, App Engine Flex requires you to create an app.yaml file to describe the resources your app requires. Normally, you put this file in src/main/appengine, and it should resemble the following file:

service: default

runtime: java

env: flex

runtime\_config:

jdk: openjdk8

handlers:

- url: /.\*

script: this field is required, but ignored

manual\_scaling:

instances: 1

health\_check:

enable\_health\_check: False

env\_variables:

ENCRYPT\_KEY: your\_encryption\_key\_here

You can deploy the app (for example, with a Maven plugin) by adding the project ID to the build configuration, as shown in the following example:

<plugin>

<groupId>com.google.cloud.tools</groupId>

<artifactId>appengine-maven-plugin</artifactId>

<version>1.3.0</version>

<configuration>

<project>myproject</project>

</configuration>

</plugin>

Then deploy with mvn appengine:deploy (if you need to authenticate first, the build fails).

**3. Installing Spring Boot Applications**

In addition to running Spring Boot applications by using java -jar, it is also possible to make fully executable applications for Unix systems. A fully executable jar can be executed like any other executable binary or it can be [registered with init.d or systemd](https://docs.spring.io/spring-boot/docs/current/reference/html/deployment.html#deployment-service). This makes it very easy to install and manage Spring Boot applications in common production environments.

|  |  |
| --- | --- |
|  | Fully executable jars work by embedding an extra script at the front of the file. 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 make your jar or war fully executable only if you intend to execute it directly, rather than running it with `java -jar`or deploying it to a servlet container. |
|  | A zip64-format jar file cannot be made fully executable. Attempting to do so will result in a jar file that is reported as corrupt when executed directly or with java -jar. A standard-format jar file that contains one or more zip64-format nested jars can be fully executable. |

To create a ‘fully executable’ jar with Maven, use the following plugin configuration:

<plugin>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-maven-plugin</artifactId>

<configuration>

<executable>true</executable>

</configuration>

</plugin>

The following example shows the equivalent Gradle configuration:

bootJar {

launchScript()

}

You can then run your application by typing ./my-application.jar (where my-application is the name of your artifact). The directory containing the jar is used as your application’s working directory.

**3.1. Supported Operating Systems**

The default script supports most Linux distributions and is tested on CentOS and Ubuntu. Other platforms, such as OS X and FreeBSD, require the use of a custom embeddedLaunchScript.

**3.2. Unix/Linux Services**

Spring Boot application can be easily started as Unix/Linux services by using either init.d or systemd.

**3.2.1. Installation as an init.d Service (System V)**

If you configured Spring Boot’s Maven or Gradle plugin to generate a [fully executable jar](https://docs.spring.io/spring-boot/docs/current/reference/html/deployment.html#deployment-install), and you do not use a custom embeddedLaunchScript, your application can be used as an init.d service. To do so, symlink the jar to init.d to support the standard start, stop, restart, and status commands.

The script supports the following features:

* Starts the services as the user that owns the jar file
* Tracks the application’s PID by using /var/run/<appname>/<appname>.pid
* Writes console logs to /var/log/<appname>.log

Assuming that you have a Spring Boot application installed in /var/myapp, to install a Spring Boot application as an init.d service, create a symlink, as follows:

$ sudo ln -s /var/myapp/myapp.jar /etc/init.d/myapp

Once installed, you can start and stop the service in the usual way. For example, on a Debian-based system, you could start it with the following command:

$ service myapp start

|  |  |
| --- | --- |
|  | If your application fails to start, check the log file written to /var/log/<appname>.log for errors. |

You can also flag the application to start automatically by using your standard operating system tools. For example, on Debian, you could use the following command:

$ update-rc.d myapp defaults <priority>

**Securing an init.d Service**

|  |  |
| --- | --- |
|  | The following is a set of guidelines on how to secure a Spring Boot application that runs as an init.d service. It is not intended to be an exhaustive list of everything that should be done to harden an application and the environment in which it runs. |

When executed as root, as is the case when root is being used to start an init.d service, the default executable script runs the application as the user specified in the RUN\_AS\_USER environment variable. When the environment variable is not set, the user who owns the jar file is used instead. You should never run a Spring Boot application as root, so RUN\_AS\_USER should never be root and your application’s jar file should never be owned by root. Instead, create a specific user to run your application and set the RUN\_AS\_USER environment variable or use chown to make it the owner of the jar file, as shown in the following example:

$ chown bootapp:bootapp your-app.jar

In this case, the default executable script runs the application as the bootapp user.

|  |  |
| --- | --- |
|  | To reduce the chances of the application’s user account being compromised, you should consider preventing it from using a login shell. For example, you can set the account’s shell to /usr/sbin/nologin. |

You should also take steps to prevent the modification of your application’s jar file. Firstly, configure its permissions so that it cannot be written and can only be read or executed by its owner, as shown in the following example:

$ chmod 500 your-app.jar

Second, you should also take steps to limit the damage if your application or the account that’s running it is compromised. If an attacker does gain access, they could make the jar file writable and change its contents. One way to protect against this is to make it immutable by using chattr, as shown in the following example:

$ sudo chattr +i your-app.jar

This will prevent any user, including root, from modifying the jar.

If root is used to control the application’s service and you [use a .conf file](https://docs.spring.io/spring-boot/docs/current/reference/html/deployment.html#deployment-script-customization-conf-file) to customize its startup, the .conf file is read and evaluated by the root user. It should be secured accordingly. Use chmod so that the file can only be read by the owner and use chown to make root the owner, as shown in the following example:

$ chmod 400 your-app.conf

$ sudo chown root:root your-app.conf

**3.2.2. Installation as a systemd Service**

systemd is the successor of the System V init system and is now being used by many modern Linux distributions. Although you can continue to use init.d scripts with systemd, it is also possible to launch Spring Boot applications by using systemd ‘service’ scripts.

Assuming that you have a Spring Boot application installed in /var/myapp, to install a Spring Boot application as a systemd service, create a script named myapp.service and place it in /etc/systemd/system directory. The following script offers an example:

[Unit]

Description=myapp

After=syslog.target

[Service]

User=myapp

ExecStart=/var/myapp/myapp.jar

SuccessExitStatus=143

[Install]

WantedBy=multi-user.target

|  |  |
| --- | --- |
|  | Remember to change the Description, User, and ExecStart fields for your application. |
|  | The ExecStart field does not declare the script action command, which means that the run command is used by default. | |

Note that, unlike when running as an init.d service, the user that runs the application, the PID file, and the console log file are managed by systemd itself and therefore must be configured by using appropriate fields in the ‘service’ script. Consult the [service unit configuration man page](https://www.freedesktop.org/software/systemd/man/systemd.service.html) for more details.

To flag the application to start automatically on system boot, use the following command:

$ systemctl enable myapp.service

Refer to man systemctl for more details.

**3.2.3. Customizing the Startup Script**

The default embedded startup script written by the Maven or Gradle plugin can be customized in a number of ways. For most people, using the default script along with a few customizations is usually enough. If you find you cannot customize something that you need to, use the embeddedLaunchScript option to write your own file entirely.

**Customizing the Start Script when It Is Written**

It often makes sense to customize elements of the start script as it is written into the jar file. For example, init.d scripts can provide a “description”. Since you know the description up front (and it need not change), you may as well provide it when the jar is generated.

To customize written elements, use the embeddedLaunchScriptProperties option of the Spring Boot Maven plugin or the [properties property of the Spring Boot Gradle plugin’s launchScript](https://docs.spring.io/spring-boot/docs/2.2.2.RELEASE/gradle-plugin/reference/html/#packaging-executable-configuring-launch-script).

The following property substitutions are supported with the default script:

| **Name** | **Description** | **Gradle default** | **Maven default** |
| --- | --- | --- | --- |
| mode | The script mode. | auto | auto |
| initInfoProvides | The Provides section of “INIT INFO” | ${task.baseName} | ${project.artifactId} |
| initInfoRequiredStart | Required-Start section of “INIT INFO”. | $remote\_fs $syslog $network | $remote\_fs $syslog $network |
| initInfoRequiredStop | Required-Stop section of “INIT INFO”. | $remote\_fs $syslog $network | $remote\_fs $syslog $network |
| initInfoDefaultStart | Default-Start section of “INIT INFO”. | 2 3 4 5 | 2 3 4 5 |
| initInfoDefaultStop | Default-Stop section of “INIT INFO”. | 0 1 6 | 0 1 6 |
| initInfoShortDescription | Short-Description section of “INIT INFO”. | Single-line version of ${project.description} (falling back to ${task.baseName}) | ${project.name} |
| initInfoDescription | Description section of “INIT INFO”. | ${project.description} (falling back to ${task.baseName}) | ${project.description} (falling back to ${project.name}) |
| initInfoChkconfig | chkconfig section of “INIT INFO” | 2345 99 01 | 2345 99 01 |
| confFolder | The default value for CONF\_FOLDER | Folder containing the jar | Folder containing the jar |
| inlinedConfScript | Reference to a file script that should be inlined in the default launch script. This can be used to set environmental variables such as JAVA\_OPTS before any external config files are loaded |  |  |
| logFolder | Default value for LOG\_FOLDER. Only valid for an init.d service |  |  |
| logFilename | Default value for LOG\_FILENAME. Only valid for an init.d service |  |  |
| pidFolder | Default value for PID\_FOLDER. Only valid for an init.d service |  |  |
| pidFilename | Default value for the name of the PID file in PID\_FOLDER. Only valid for an init.d service |  |  |
| useStartStopDaemon | Whether the start-stop-daemon command, when it’s available, should be used to control the process | true | true |
| stopWaitTime | Default value for STOP\_WAIT\_TIME in seconds. Only valid for an init.d service | 60 | 60 |

**Customizing a Script When It Runs**

For items of the script that need to be customized *after* the jar has been written, you can use environment variables or a [config file](https://docs.spring.io/spring-boot/docs/current/reference/html/deployment.html#deployment-script-customization-conf-file).

The following environment properties are supported with the default script:

| **Variable** | **Description** |
| --- | --- |
| MODE | The “mode” of operation. The default depends on the way the jar was built but is usually auto (meaning it tries to guess if it is an init script by checking if it is a symlink in a directory called init.d). You can explicitly set it to service so that the stop|start|status|restart commands work or to run if you want to run the script in the foreground. |
| RUN\_AS\_USER | The user that will be used to run the application. When not set, the user that owns the jar file will be used. |
| USE\_START\_STOP\_DAEMON | Whether the start-stop-daemon command, when it’s available, should be used to control the process. Defaults to true. |
| PID\_FOLDER | The root name of the pid folder (/var/run by default). |
| LOG\_FOLDER | The name of the folder in which to put log files (/var/log by default). |
| CONF\_FOLDER | The name of the folder from which to read .conf files (same folder as jar-file by default). |
| LOG\_FILENAME | The name of the log file in the LOG\_FOLDER (<appname>.log by default). |
| APP\_NAME | The name of the app. If the jar is run from a symlink, the script guesses the app name. If it is not a symlink or you want to explicitly set the app name, this can be useful. |
| RUN\_ARGS | The arguments to pass to the program (the Spring Boot app). |
| JAVA\_HOME | The location of the java executable is discovered by using the PATH by default, but you can set it explicitly if there is an executable file at $JAVA\_HOME/bin/java. |
| JAVA\_OPTS | Options that are passed to the JVM when it is launched. |
| JARFILE | The explicit location of the jar file, in case the script is being used to launch a jar that it is not actually embedded. |
| DEBUG | If not empty, sets the -x flag on the shell process, making it easy to see the logic in the script. |
| STOP\_WAIT\_TIME | The time in seconds to wait when stopping the application before forcing a shutdown (60 by default). |

|  |  |
| --- | --- |
|  | The PID\_FOLDER, LOG\_FOLDER, and LOG\_FILENAME variables are only valid for an init.d service. For systemd, the equivalent customizations are made by using the ‘service’ script. See the [service unit configuration man page](https://www.freedesktop.org/software/systemd/man/systemd.service.html) for more details. |

With the exception of JARFILE and APP\_NAME, the settings listed in the preceding section can be configured by using a .conf file. The file is expected to be next to the jar file and have the same name but suffixed with .conf rather than .jar. For example, a jar named /var/myapp/myapp.jar uses the configuration file named /var/myapp/myapp.conf, as shown in the following example:

myapp.conf

JAVA\_OPTS=-Xmx1024M

LOG\_FOLDER=/custom/log/folder

|  |  |
| --- | --- |
|  | If you do not like having the config file next to the jar file, you can set a CONF\_FOLDER environment variable to customize the location of the config file. |

To learn about securing this file appropriately, see [the guidelines for securing an init.d service](https://docs.spring.io/spring-boot/docs/current/reference/html/deployment.html#deployment-initd-service-securing).

**3.3. Microsoft Windows Services**

A Spring Boot application can be started as a Windows service by using [winsw](https://github.com/kohsuke/winsw).

A ([separately maintained sample](https://github.com/snicoll-scratches/spring-boot-daemon)) describes step-by-step how you can create a Windows service for your Spring Boot application.

**4. What to Read Next**

Check out the [Cloud Foundry](https://www.cloudfoundry.org/), [Heroku](https://www.heroku.com/), [OpenShift](https://www.openshift.com), and [Boxfuse](https://boxfuse.com) web sites for more information about the kinds of features that a PaaS can offer. These are just four of the most popular Java PaaS providers. Since Spring Boot is so amenable to cloud-based deployment, you can freely consider other providers as well.

The next section goes on to cover the [*Spring Boot CLI*](https://docs.spring.io/spring-boot/docs/current/reference/html/spring-boot-cli.html#cli), or you can jump ahead to read about [*build tool plugins*](https://docs.spring.io/spring-boot/docs/current/reference/html/build-tool-plugins.html#build-tool-plugins).

Last updated 2019-12-06 06:37:25 UTC