Traditional way of Deployment - Spring Boot Application: -

**Traditional deployment**

**Create a deployable war file**

The first step in producing a deployable war file is to provide a SpringBootServletInitializer subclass and override its configure method. This makes use of Spring Framework’s Servlet 3.0 support and allows you to configure your application when it’s launched by the servlet container. Typically, you update your application’s main class to extend SpringBootServletInitializer:

*@SpringBootApplication*

**public** **class** Application **extends** SpringBootServletInitializer {

*@Override*

**protected** SpringApplicationBuilder configure(SpringApplicationBuilder application) {

**return** application.sources(Application.**class**);

}

**public** **static** **void** main(String[] args) **throws** Exception {

SpringApplication.run(Application.**class**, args);

}

}

The next step is to update your build configuration so that your project produces a war file rather than a jar file. If you’re using Maven and usingspring-boot-starter-parent (which configures Maven’s war plugin for you) all you need to do is modify pom.xml to change the packaging to war:

<packaging>war</packaging>

If you’re using Gradle, you need to modify build.gradle to apply the war plugin to the project:

apply plugin: 'war'

The final step in the process is to ensure that the embedded servlet container doesn’t interfere with the servlet container to which the war file will be deployed. To do so, you need to mark the embedded servlet container dependency as provided.

If you’re using Maven:

<dependencies>

*<!-- … -->*

<dependency>

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

<artifactId>spring-boot-starter-tomcat</artifactId>

<scope>provided</scope>

</dependency>

*<!-- … -->*

</dependencies>

And if you’re using Gradle:

dependencies {

*// …*

providedRuntime 'org.springframework.boot:spring-boot-starter-tomcat'

*// …*

}

If you’re using the [Spring Boot build tools](http://docs.spring.io/spring-boot/docs/current/reference/html/build-tool-plugins.html), marking the embedded servlet container dependency as provided will produce an executable war file with the provided dependencies packaged in a lib-provided directory. This means that, in addition to being deployable to a servlet container, you can also run your application usingjava -jar on the command line.

|  |
| --- |
| Tip] |
| Take a look at Spring Boot’s sample applications for a [Maven-based example](http://github.com/spring-projects/spring-boot/tree/v1.3.6.RELEASE/spring-boot-samples/spring-boot-sample-traditional/pom.xml) of the above-described configuration. |

**80.2 Create a deployable war file for older servlet containers**

Older Servlet containers don’t have support for the ServletContextInitializer bootstrap process used in Servlet 3.0. You can still use Spring and Spring Boot in these containers but you are going to need to add a web.xml to your application and configure it to load an ApplicationContext via a DispatcherServlet.

**80.3 Convert an existing application to Spring Boot**

For a non-web application it should be easy (throw away the code that creates your ApplicationContext and replace it with calls to SpringApplication orSpringApplicationBuilder). Spring MVC web applications are generally amenable to first creating a deployable war application, and then migrating it later to an executable war and/or jar. Useful reading is in the [Getting Started Guide on Converting a jar to a war](http://spring.io/guides/gs/convert-jar-to-war/).

Create a deployable war by extending SpringBootServletInitializer (e.g. in a class called Application), and add the Spring Boot@EnableAutoConfiguration annotation. Example:

*@Configuration*

*@EnableAutoConfiguration*

*@ComponentScan*

**public** **class** Application **extends** SpringBootServletInitializer {

*@Override*

**protected** SpringApplicationBuilder configure(SpringApplicationBuilder application) {

*// Customize the application or call application.sources(...) to add sources*

*// Since our example is itself a @Configuration class we actually don't*

*// need to override this method.*

**return** application;

}

}

Remember that whatever you put in the sources is just a Spring ApplicationContext and normally anything that already works should work here. There might be some beans you can remove later and let Spring Boot provide its own defaults for them, but it should be possible to get something working first.

Static resources can be moved to /public (or /static or /resources or /META-INF/resources) in the classpath root. Same for messages.properties(Spring Boot detects this automatically in the root of the classpath).

Vanilla usage of Spring DispatcherServlet and Spring Security should require no further changes. If you have other features in your application, using other servlets or filters for instance, then you may need to add some configuration to your Application context, replacing those elements from the web.xml as follows:

* A @Bean of type Servlet or ServletRegistrationBean installs that bean in the container as if it was a <servlet/> and <servlet-mapping/> inweb.xml.
* A @Bean of type Filter or FilterRegistrationBean behaves similarly (like a <filter/> and <filter-mapping/>.
* An ApplicationContext in an XML file can be added to an @Import in your Application. Or simple cases where annotation configuration is heavily used already can be recreated in a few lines as @Bean definitions.

Once the war is working we make it executable by adding a main method to our Application, e.g.

**public** **static** **void** main(String[] args) {

SpringApplication.run(Application.**class**, args);

}

Applications can fall into more than one category:

* Servlet 3.0+ applications with no web.xml.
* Applications with a web.xml.
* Applications with a context hierarchy.
* Applications without a context hierarchy.

All of these should be amenable to translation, but each might require slightly different tricks.

Servlet 3.0+ applications might translate pretty easily if they already use the Spring Servlet 3.0+ initializer support classes. Normally all the code from an existingWebApplicationInitializer can be moved into a SpringBootServletInitializer. If your existing application has more than one ApplicationContext(e.g. if it uses AbstractDispatcherServletInitializer) then you might be able to squash all your context sources into a single SpringApplication. The main complication you might encounter is if that doesn’t work and you need to maintain the context hierarchy. See the [entry on building a hierarchy](http://docs.spring.io/spring-boot/docs/current/reference/html/howto-spring-boot-application.html#howto-build-an-application-context-hierarchy) for examples. An existing parent context that contains web-specific features will usually need to be broken up so that all the ServletContextAware components are in the child context.

Applications that are not already Spring applications might be convertible to a Spring Boot application, and the guidance above might help, but your mileage may vary.

**80.4 Deploying a WAR to WebLogic**

To deploy a Spring Boot application to WebLogic you must ensure that your servlet initializer **directly** implements WebApplicationInitializer (even if you extend from a base class that already implements it).

A typical initializer for WebLogic would be something like this:

**import** org.springframework.boot.autoconfigure.SpringBootApplication;

**import** org.springframework.boot.context.web.SpringBootServletInitializer;

**import** org.springframework.web.WebApplicationInitializer;

*@SpringBootApplication*

**public** **class** MyApplication **extends** SpringBootServletInitializer **implements** WebApplicationInitializer {

}

If you use logback, you will also need to tell WebLogic to prefer the packaged version rather than the version that pre-installed with the server. You can do this by adding a WEB-INF/weblogic.xml file with the following contents:

<?xml version="1.0" encoding="UTF-8"?>

<wls:weblogic-web-app

xmlns:wls="http://xmlns.oracle.com/weblogic/weblogic-web-app"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://java.sun.com/xml/ns/javaee

http://java.sun.com/xml/ns/javaee/ejb-jar\_3\_0.xsd

http://xmlns.oracle.com/weblogic/weblogic-web-app

http://xmlns.oracle.com/weblogic/weblogic-web-app/1.4/weblogic-web-app.xsd">

<wls:container-descriptor>

<wls:prefer-application-packages>

<wls:package-name>org.slf4j</wls:package-name>

</wls:prefer-application-packages>

</wls:container-descriptor>

</wls:weblogic-web-app>

**80.5 Deploying a WAR in an Old (Servlet 2.5) Container**

Spring Boot uses Servlet 3.0 APIs to initialize the ServletContext (register Servlets etc.) so you can’t use the same application out of the box in a Servlet 2.5 container. It **is** however possible to run a Spring Boot application on an older container with some special tools. If you includeorg.springframework.boot:spring-boot-legacy as a dependency ([maintained separately](https://github.com/scratches/spring-boot-legacy) to the core of Spring Boot and currently available at 1.0.0.RELEASE), all you should need to do is create a web.xml and declare a context listener to create the application context and your filters and servlets. The context listener is a special purpose one for Spring Boot, but the rest of it is normal for a Spring application in Servlet 2.5. Example:

<?xml version="1.0" encoding="UTF-8"?>

<web-app version="2.5" xmlns="http://java.sun.com/xml/ns/javaee"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app\_2\_5.xsd">

<context-param>

<param-name>contextConfigLocation</param-name>

<param-value>demo.Application</param-value>

</context-param>

<listener>

<listener-class>org.springframework.boot.legacy.context.web.SpringBootContextLoaderListener</listener-class>

</listener>

<filter>

<filter-name>metricFilter</filter-name>

<filter-class>org.springframework.web.filter.DelegatingFilterProxy</filter-class>

</filter>

<filter-mapping>

<filter-name>metricFilter</filter-name>

<url-pattern>/\*</url-pattern>

</filter-mapping>

<servlet>

<servlet-name>appServlet</servlet-name>

<servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>

<init-param>

<param-name>contextAttribute</param-name>

<param-value>org.springframework.web.context.WebApplicationContext.ROOT</param-value>

</init-param>

<load-on-startup>1</load-on-startup>

</servlet>

<servlet-mapping>

<servlet-name>appServlet</servlet-name>

<url-pattern>/</url-pattern>

</servlet-mapping>

</web-app>

In this example we are using a single application context (the one created by the context listener) and attaching it to the DispatcherServlet using an init parameter. This is normal in a Spring Boot application (you normally only have one application context).