# Part VI. Deploying Spring Boot applications

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

This section covers some of the more common deployment scenarios.

## 58. 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 some 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, it might be an embedded web server, or it might be 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’ll look at what it takes to get the [simple application that we developed](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#getting-started-first-application) in the “Getting Started” section up and running in the Cloud.

## 58.1 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’ve built your application (using, for example, mvn clean package) and [installed the cf command line tool](http://docs.cloudfoundry.org/devguide/installcf/install-go-cli.html), simply deploy your application using the cf push command as follows, substituting the path to your compiled .jar. Be sure to have [logged in with your cf command line client](http://docs.cloudfoundry.org/devguide/installcf/whats-new-v6.html#login) before pushing an application.

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

See the [cf push documentation](http://docs.cloudfoundry.org/devguide/installcf/whats-new-v6.html#push) for more options. If there is a Cloud Foundry [manifest.yml](http://docs.cloudfoundry.org/devguide/deploy-apps/manifest.html) file present in the same directory, it will be consulted.

|  |
| --- |
| [Note] |
| Here we are substituting acloudyspringtime for whatever value you give cf as the name of your application. |

At this point cf will start uploading your application:

Uploading acloudyspringtime... **OK**

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

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

-----> Java Buildpack source: system

-----> Downloading Open JDK 1.7.0\_51 from .../x86\_64/openjdk-1.7.0\_51.tar.gz (**1.8s**)

Expanding Open JDK to .java-buildpack/open\_jdk (**1.2s**)

-----> Downloading Spring Auto Reconfiguration from 0.8.7 .../auto-reconfiguration-0.8.7.jar (**0.1s**)

-----> Uploading droplet (**44M**)

Checking status of app 'acloudyspringtime'...

0 of 1 instances running (1 starting)

...

0 of 1 instances running (1 down)

...

0 of 1 instances running (1 starting)

...

1 of 1 instances running (1 running)

App started

Congratulations! The application is now live!

It’s easy to then verify the status of the deployed application:

$ 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 hit the application at the URI given, in this casehttp://acloudyspringtime.cfapps.io/.

### 58.1.1 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 don’t 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:

*@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 CloudFoundryVcapEnvironmentPostProcessor Javadoc for complete details.

|  |
| --- |
| [Tip] |
| The [Spring Cloud Connectors](http://cloud.spring.io/spring-cloud-connectors/) project is a better fit for tasks such as configuring a DataSource. Spring Boot includes auto-configuration support and a spring-boot-starter-cloud-connectors starter. |

## 58.2 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. Here’s 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 it when it starts up. The $PORT environment variable is assigned to us by the Heroku PaaS.

Heroku by default will use Java 1.8. This is fine as long as your Maven or Gradle build is set to use the same version (Maven users can use the java.version property). If you want to use JDK 1.7, create a new file adjacent to your pom.xml and Procfile, called system.properties. In this file add the following:

java.runtime.version=1.7

This should be everything you need. The most common workflow for Heroku deployments is to git push the code to production.

$ 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: http://repo.spring.io/...

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

....

Downloaded: http://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

http://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.

## 58.3 OpenShift

[OpenShift](https://www.openshift.com/) is the RedHat public (and enterprise) PaaS solution. Like Heroku, it works by running scripts triggered by git commits, so you can script the launching of a Spring Boot application in pretty much any way you like as long as the Java runtime is available (which is a standard feature you can ask for at OpenShift). To do this you can use the [DIY Cartridge](https://www.openshift.com/developers/do-it-yourself) and hooks in your repository under .openshift/action\_hooks:

The basic model is to:

1. Ensure Java and your build tool are installed remotely, e.g. using a pre\_build hook (Java and Maven are installed by default, Gradle is not)
2. Use a build hook to build your jar (using Maven or Gradle), e.g.
3. #!/bin/bash
4. cd $OPENSHIFT\_REPO\_DIR

mvn package -s .openshift/settings.xml -DskipTests=true

1. Add a start hook that calls java -jar …​
2. #!/bin/bash
3. cd $OPENSHIFT\_REPO\_DIR

nohup java -jar target/\*.jar --server.port=${OPENSHIFT\_DIY\_PORT} --server.address=${OPENSHIFT\_DIY\_IP} &

1. Use a stop hook (since the start is supposed to return cleanly), e.g.
2. #!/bin/bash
3. source $OPENSHIFT\_CARTRIDGE\_SDK\_BASH
4. PID=$(ps -ef | grep java.\*\.jar | grep -v grep | awk '{ print $2 }')
5. if [ -z "$PID" ]
6. then
7. client\_result "Application is already stopped"
8. else
9. kill $PID

fi

1. Embed service bindings from environment variables provided by the platform in your application.properties, e.g.
2. spring.datasource.url: jdbc:mysql://${OPENSHIFT\_MYSQL\_DB\_HOST}:${OPENSHIFT\_MYSQL\_DB\_PORT}/${OPENSHIFT\_APP\_NAME}
3. spring.datasource.username: ${OPENSHIFT\_MYSQL\_DB\_USERNAME}

spring.datasource.password: ${OPENSHIFT\_MYSQL\_DB\_PASSWORD}

There’s a blog on [running Gradle in OpenShift](https://www.openshift.com/blogs/run-gradle-builds-on-openshift) on their website that will get you started with a gradle build to run the app.

## 58.4 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. Options include :

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

Each has different features and pricing model, here we will describe only the simplest option : AWS Elastic Beanstalk.

### 58.4.1 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 producing a war file. There is no any special configuration required, just follow the official guide.

#### Using the Java SE platform

This option applies to Spring Boot projects producing a jar file and running 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 to your application.properties:

server.port=5000

#### Best practices

##### Uploading binaries instead of sources

By default Elastic Beanstalk uploads sources and compiles them in AWS. To upload the binaries instead, add the following to your .elasticbeanstalk/config.ymlfile:

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 cost perspective, to avoid it, 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). Single instance environments can be created using the CLI as well using the following command:

eb create -s

### 58.4.2 Summary

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

## 58.5 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 will use 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, etc).

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

$ 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.com/docs/commandline/#configuration](https://boxfuse.com/docs/commandline/#configuration) [boxfuse.conf] file present in the current directory, it will be consulted.

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

At this point boxfuse will create an image for your application, upload it, and then configure and start the necessary resources on AWS:

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 http://52.28.235.61/ ...

Payload started in 00:29.266s -> http://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 http://myapp-axelfontaine.boxfuse.io/

Your application should now be up and running on AWS.

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

## 58.6 Google Cloud

Google Cloud has several options that could 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 using 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 HTTP routes. Add a Java app to the project and leave it empty, then use the [Google Cloud SDK](https://cloud.google.com/sdk/downloads) to push your Spring Boot app into that slot from the command line or CI build.

App Engine needs you to create an app.yaml file to describe the resources your app requires. Normally you put this in src/main/appengine, and it looks something like this:

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 simply adding the project ID to the build configuration:

<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 will fail).

|  |
| --- |
| [Note] |
| Google App Engine Classic is tied to the Servlet 2.5 API, so you can’t deploy a Spring Application there without some modifications. See the [Servlet 2.5 section](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto-servlet-2-5) of this guide. |

## 59. Installing Spring Boot applications

In additional to running Spring Boot applications 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/htmlsingle/#deployment-service). This makes it very easy to install and manage Spring Boot applications in common production environments.

|  |
| --- |
| [Warning] |
| 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 only make your jar or war fully executable if you intend to execute it directly, rather than running it with java -jar or deploying it to a servlet container. |

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>

With Gradle, the equivalent configuration is:

springBoot {

executable = true

}

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 will be used as your application’s working directory.

## 59.1 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, will require the use of a customembeddedLaunchScript.

## 59.2 Unix/Linux services

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

### 59.2.1 Installation as an init.d service (System V)

If you’ve configured Spring Boot’s Maven or Gradle plugin to generate a [fully executable jar](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#deployment-install), and you’re not using a custom embeddedLaunchScript, then your application can be used as an init.d service. Simply 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 application’s PID 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 simply create a symlink:

$ 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:

$ service myapp start

|  |
| --- |
| [Tip] |
| 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 using your standard operating system tools. For example, on Debian:

$ update-rc.d myapp defaults <priority>

#### Securing an init.d service

|  |
| --- |
| [Note] |
| The following is a set of guidelines on how to secure a Spring Boot application that’s being run 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 will run the application as the user which owns the jar file. You should never run a Spring Boot application as root so your application’s jar file should never be owned by root. Instead, create a specific user to run your application and use chown to make it the owner of the jar file. For example:

$ chown bootapp:bootapp your-app.jar

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

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

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:

$ chmod 500 your-app.jar

Secondly, 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 using chattr:

$ 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/htmlsingle/#deployment-script-customization-conf-file) to customize its startup, the .conf file will be 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:

$ chmod 400 your-app.conf

$ sudo chown root:root your-app.conf

### 59.2.2 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 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 using the following example and place it in /etc/systemd/system directory:

[Unit]

Description=myapp

After=syslog.target

[Service]

User=myapp

ExecStart=/var/myapp/myapp.jar

SuccessExitStatus=143

[Install]

WantedBy=multi-user.target

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

Note that unlike when running as an init.d service, user that runs the application, PID file and console log file are managed by systemd itself and therefore must be configured using appropriate fields in ‘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.

### 59.2.3 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 can’t customize something that you need to, you can always use the embeddedLaunchScript option to write your own file entirely.

#### Customizing script when it’s written

It often makes sense to customize elements of the start script as it’s written into the jar file. For example, init.d scripts can provide a “description” and, since you know this up front (and it won’t 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 or Gradle plugins.

The following property substitutions are supported with the default script:

| **Name** | **Description** |
| --- | --- |
| mode | The script mode. Defaults to auto. |
| initInfoProvides | The Provides section of “INIT INFO”. Defaults to spring-boot-application for Gradle and to ${project.artifactId} for Maven. |
| initInfoRequiredStart | The Required-Start section of “INIT INFO”. Defaults to $remote\_fs $syslog $network. |
| initInfoRequiredStop | The Required-Stop section of “INIT INFO”. Defaults to $remote\_fs $syslog $network. |
| initInfoDefaultStart | The Default-Start section of “INIT INFO”. Defaults to 2 3 4 5. |
| initInfoDefaultStop | The Default-Stop section of “INIT INFO”. Defaults to 0 1 6. |
| initInfoShortDescription | The Short-Description section of “INIT INFO”. Defaults to Spring Boot Application for Gradle and to ${project.name}for Maven. |
| initInfoDescription | The Description section of “INIT INFO”. Defaults to Spring Boot Application for Gradle and to ${project.description} (falling back to ${project.name}) for Maven. |
| initInfoChkconfig | The chkconfig section of “INIT INFO”. Defaults to 2345 99 01. |
| confFolder | The default value for CONF\_FOLDER. Defaults to the folder containing the jar. |
| logFolder | The default value for LOG\_FOLDER. Only valid for an init.d service. |
| logFilename | The default value for LOG\_FILENAME. Only valid for an init.d service. |
| pidFolder | The default value for PID\_FOLDER. Only valid for an init.d service. |
| pidFilename | The default value for the name of the pid file in PID\_FOLDER. Only valid for an init.d service. |
| useStartStopDaemon | If the start-stop-daemon command, when it’s available, should be used to control the process. Defaults to true. |
| stopWaitTime | The default value for STOP\_WAIT\_TIME. Only valid for an init.d service. Defaults to 60 seconds. |

#### Customizing 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/htmlsingle/#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 will usually be 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 just want to run the script in the foreground. |
| USE\_START\_STOP\_DAEMON | If 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 to put log files in (/var/log by default). |
| CONF\_FOLDER | The name of the folder to read .conf files from (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, but 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 in. |
| DEBUG | if not empty will set 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). |
| [Note] |
| The PID\_FOLDER, LOG\_FOLDER and LOG\_FILENAME variables are only valid for an init.d service. With systemd the equivalent customizations are made using ‘service’ script. Check 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 above settings can be configured using a .conf file. The file is expected 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 will use the configuration file named /var/myapp/myapp.conf.

**myapp.conf.**

JAVA\_OPTS=-Xmx1024M

LOG\_FOLDER=/custom/log/folder

|  |
| --- |
| [Tip] |
| You can use a CONF\_FOLDER environment variable to customize the location of the config file if you don’t like it living next to the jar. |

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

## 59.3 Microsoft Windows services

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

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

## 60. What to read next

Check out the [Cloud Foundry](https://www.cloudfoundry.com/), [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’re free to 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/htmlsingle/#cli); or you can jump ahead to read about [*build tool plugins*](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#build-tool-plugins).

# Part VII. Spring Boot CLI

The Spring Boot CLI is a command line tool that can be used if you want to quickly develop with Spring. It allows you to run Groovy scripts, which means that you have a familiar Java-like syntax, without so much boilerplate code. You can also bootstrap a new project or write your own command for it.

## 61. Installing the CLI

The Spring Boot CLI can be installed manually; using SDKMAN! (the SDK Manager) or using Homebrew or MacPorts if you are an OSX user. See [*Section 10.2, “Installing the Spring Boot CLI”*](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#getting-started-installing-the-cli) in the “Getting started” section for comprehensive installation instructions.

## 62. Using the CLI

Once you have installed the CLI you can run it by typing spring. If you run spring without any arguments, a simple help screen is displayed:

$ spring

usage: spring [--help] [--version]

<command> [<args>]

Available commands are:

run [options] <files> [--] [args]

Run a spring groovy script

... more command help is shown here

You can use help to get more details about any of the supported commands. For example:

$ spring help run

spring run - Run a spring groovy script

usage: spring run [options] <files> [--] [args]

Option Description

------ -----------

--autoconfigure [Boolean] Add autoconfigure compiler

transformations (default: true)

--classpath, -cp Additional classpath entries

-e, --edit Open the file with the default system

editor

--no-guess-dependencies Do not attempt to guess dependencies

--no-guess-imports Do not attempt to guess imports

-q, --quiet Quiet logging

-v, --verbose Verbose logging of dependency

resolution

--watch Watch the specified file for changes

The version command provides a quick way to check which version of Spring Boot you are using.

$ spring version

Spring CLI v1.5.7.RELEASE

## 62.1 Running applications using the CLI

You can compile and run Groovy source code using the run command. The Spring Boot CLI is completely self-contained so you don’t need any external Groovy installation.

Here is an example “hello world” web application written in Groovy:

**hello.groovy.**

*@RestController*

**class** WebApplication {

*@RequestMapping("/")*

String home() {

"Hello World!"

}

}

To compile and run the application type:

$ spring run hello.groovy

To pass command line arguments to the application, you need to use a -- to separate them from the “spring” command arguments, e.g.

$ spring run hello.groovy -- --server.port=9000

To set JVM command line arguments you can use the JAVA\_OPTS environment variable, e.g.

$ JAVA\_OPTS=-Xmx1024m spring run hello.groovy

### 62.1.1 Deduced “grab” dependencies

Standard Groovy includes a @Grab annotation which allows you to declare dependencies on a third-party libraries. This useful technique allows Groovy to download jars in the same way as Maven or Gradle would, but without requiring you to use a build tool.

Spring Boot extends this technique further, and will attempt to deduce which libraries to “grab” based on your code. For example, since the WebApplication code above uses @RestController annotations, “Tomcat” and “Spring MVC” will be grabbed.

The following items are used as “grab hints”:

| **Items** | **Grabs** |
| --- | --- |
| JdbcTemplate, NamedParameterJdbcTemplate, DataSource | JDBC Application. |
| @EnableJms | JMS Application. |
| @EnableCaching | Caching abstraction. |
| @Test | JUnit. |
| @EnableRabbit | RabbitMQ. |
| @EnableReactor | Project Reactor. |
| extends Specification | Spock test. |
| @EnableBatchProcessing | Spring Batch. |
| @MessageEndpoint @EnableIntegrationPatterns | Spring Integration. |
| @EnableDeviceResolver | Spring Mobile. |
| @Controller @RestController @EnableWebMvc | Spring MVC + Embedded Tomcat. |
| @EnableWebSecurity | Spring Security. |
| @EnableTransactionManagement | Spring Transaction Management. |
| [Tip] |
| See subclasses of [CompilerAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot-cli/src/main/java/org/springframework/boot/cli/compiler/CompilerAutoConfiguration.java) in the Spring Boot CLI source code to understand exactly how customizations are applied. |

### 62.1.2 Deduced “grab” coordinates

Spring Boot extends Groovy’s standard @Grab support by allowing you to specify a dependency without a group or version, for example @Grab('freemarker'). This will consult Spring Boot’s default dependency metadata to deduce the artifact’s group and version. Note that the default metadata is tied to the version of the CLI that you’re using – it will only change when you move to a new version of the CLI, putting you in control of when the versions of your dependencies may change. A table showing the dependencies and their versions that are included in the default metadata can be found in the [appendix](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#appendix-dependency-versions).

### 62.1.3 Default import statements

To help reduce the size of your Groovy code, several import statements are automatically included. Notice how the example above refers to @Component,@RestController and @RequestMapping without needing to use fully-qualified names or import statements.

|  |
| --- |
| [Tip] |
| Many Spring annotations will work without using import statements. Try running your application to see what fails before adding imports. |

### 62.1.4 Automatic main method

Unlike the equivalent Java application, you do not need to include a public static void main(String[] args) method with your Groovy scripts. ASpringApplication is automatically created, with your compiled code acting as the source.

### 62.1.5 Custom dependency management

By default, the CLI uses the dependency management declared in spring-boot-dependencies when resolving @Grab dependencies. Additional dependency management, that will override the default dependency management, can be configured using the @DependencyManagementBom annotation. The annotation’s value should specify the coordinates (groupId:artifactId:version) of one or more Maven BOMs.

For example, the following declaration:

@DependencyManagementBom("com.example.custom-bom:1.0.0")

Will pick up custom-bom-1.0.0.pom in a Maven repository under com/example/custom-versions/1.0.0/.

When multiple BOMs are specified they are applied in the order that they’re declared. For example:

@DependencyManagementBom(["com.example.custom-bom:1.0.0",

"com.example.another-bom:1.0.0"])

indicates that dependency management in another-bom will override the dependency management in custom-bom.

You can use @DependencyManagementBom anywhere that you can use @Grab, however, to ensure consistent ordering of the dependency management, you can only use @DependencyManagementBom at most once in your application. A useful source of dependency management (that is a superset of Spring Boot’s dependency management) is the [Spring IO Platform](http://platform.spring.io/), e.g. @DependencyManagementBom('io.spring.platform:platform-bom:1.1.2.RELEASE').

## 62.2 Testing your code

The test command allows you to compile and run tests for your application. Typical usage looks like this:

$ spring test app.groovy tests.groovy

Total: 1, Success: 1, : Failures: 0

Passed? true

In this example, tests.groovy contains JUnit @Test methods or Spock Specification classes. All the common framework annotations and static methods should be available to you without having to import them.

Here is the tests.groovy file that we used above (with a JUnit test):

**class** ApplicationTests {

*@Test*

**void** homeSaysHello() {

assertEquals("Hello World!", **new** WebApplication().home())

}

}

|  |
| --- |
| [Tip] |
| If you have more than one test source files, you might prefer to organize them into a test directory. |

## 62.3 Applications with multiple source files

You can use “shell globbing” with all commands that accept file input. This allows you to easily use multiple files from a single directory, e.g.

$ spring run \*.groovy

This technique can also be useful if you want to segregate your “test” or “spec” code from the main application code:

$ spring test app/\*.groovy test/\*.groovy

## 62.4 Packaging your application

You can use the jar command to package your application into a self-contained executable jar file. For example:

$ spring jar my-app.jar \*.groovy

The resulting jar will contain the classes produced by compiling the application and all of the application’s dependencies so that it can then be run using java -jar. The jar file will also contain entries from the application’s classpath. You can add explicit paths to the jar using --include and --exclude (both are comma-separated, and both accept prefixes to the values “+” and “-” to signify that they should be removed from the defaults). The default includes are

public/\*\*, resources/\*\*, static/\*\*, templates/\*\*, META-INF/\*\*, \*

and the default excludes are

.\*, repository/\*\*, build/\*\*, target/\*\*, \*\*/\*.jar, \*\*/\*.groovy

See the output of spring help jar for more information.

## 62.5 Initialize a new project

The init command allows you to create a new project using [start.spring.io](https://start.spring.io/) without leaving the shell. For example:

$ spring init --dependencies=web,data-jpa my-project

Using service at https://start.spring.io

Project extracted to '/Users/developer/example/my-project'

This creates a my-project directory with a Maven-based project using spring-boot-starter-web and spring-boot-starter-data-jpa. You can list the capabilities of the service using the --list flag

$ spring init --list

=======================================

Capabilities of https://start.spring.io

=======================================

Available dependencies:

-----------------------

actuator - Actuator: Production ready features to help you monitor and manage your application

...

web - Web: Support for full-stack web development, including Tomcat and spring-webmvc

websocket - Websocket: Support for WebSocket development

ws - WS: Support for Spring Web Services

Available project types:

------------------------

gradle-build - Gradle Config [format:build, build:gradle]

gradle-project - Gradle Project [format:project, build:gradle]

maven-build - Maven POM [format:build, build:maven]

maven-project - Maven Project [format:project, build:maven] (default)

...

The init command supports many options, check the help output for more details. For instance, the following command creates a gradle project using Java 8 and war packaging:

$ spring init --build=gradle --java-version=1.8 --dependencies=websocket --packaging=war sample-app.zip

Using service at https://start.spring.io

Content saved to 'sample-app.zip'

## 62.6 Using the embedded shell

Spring Boot includes command-line completion scripts for BASH and zsh shells. If you don’t use either of these shells (perhaps you are a Windows user) then you can use the shell command to launch an integrated shell.

$ spring shell

**Spring Boot** (v1.5.7.RELEASE)

Hit TAB to complete. Type \'help' and hit RETURN for help, and \'exit' to quit.

From inside the embedded shell you can run other commands directly:

$ version

Spring CLI v1.5.7.RELEASE

The embedded shell supports ANSI color output as well as tab completion. If you need to run a native command you can use the ! prefix. Hitting ctrl-c will exit the embedded shell.

## 62.7 Adding extensions to the CLI

You can add extensions to the CLI using the install command. The command takes one or more sets of artifact coordinates in the format group:artifact:version. For example:

$ spring install com.example:spring-boot-cli-extension:1.0.0.RELEASE

In addition to installing the artifacts identified by the coordinates you supply, all of the artifacts' dependencies will also be installed.

To uninstall a dependency use the uninstall command. As with the install command, it takes one or more sets of artifact coordinates in the format group:artifact:version. For example:

$ spring uninstall com.example:spring-boot-cli-extension:1.0.0.RELEASE

It will uninstall the artifacts identified by the coordinates you supply and their dependencies.

To uninstall all additional dependencies you can use the --all option. For example:

$ spring uninstall --all

## 63. Developing application with the Groovy beans DSL

Spring Framework 4.0 has native support for a beans{} “DSL” (borrowed from [Grails](http://grails.org/)), and you can embed bean definitions in your Groovy application scripts using the same format. This is sometimes a good way to include external features like middleware declarations. For example:

*@Configuration*

**class** Application **implements** CommandLineRunner {

*@Autowired*

SharedService service

*@Override*

**void** run(String... args) {

println service.message

}

}

**import** my.company.SharedService

beans {

service(SharedService) {

message = "Hello World"

}

}

You can mix class declarations with beans{} in the same file as long as they stay at the top level, or you can put the beans DSL in a separate file if you prefer.

## 64. Configuring the CLI with settings.xml

The Spring Boot CLI uses Aether, Maven’s dependency resolution engine, to resolve dependencies. The CLI makes use of the Maven configuration found in ~/.m2/settings.xml to configure Aether. The following configuration settings are honored by the CLI:

* Offline
* Mirrors
* Servers
* Proxies
* Profiles
  + Activation
  + Repositories
* Active profiles

Please refer to [Maven’s settings documentation](https://maven.apache.org/settings.html) for further information.

## 65. What to read next

There are some [sample groovy scripts](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot-cli/samples) available from the GitHub repository that you can use to try out the Spring Boot CLI. There is also extensive Javadoc throughout the [source code](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot-cli/src/main/java/org/springframework/boot/cli).

If you find that you reach the limit of the CLI tool, you will probably want to look at converting your application to full Gradle or Maven built “groovy project”. The next section covers Spring Boot’s [*Build tool plugins*](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#build-tool-plugins) that you can use with Gradle or Maven.

# Part VIII. Build tool plugins

Spring Boot provides build tool plugins for Maven and Gradle. The plugins offer a variety of features, including the packaging of executable jars. This section provides more details on both plugins, as well as some help should you need to extend an unsupported build system. If you are just getting started, you might want to read “[Chapter 13, *Build systems*](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#using-boot-build-systems)” from the [Part III, “Using Spring Boot”](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#using-boot) section first.

## 66. Spring Boot Maven plugin

The [Spring Boot Maven Plugin](http://docs.spring.io/spring-boot/docs/1.5.7.RELEASE/maven-plugin/) provides Spring Boot support in Maven, allowing you to package executable jar or war archives and run an application “in-place”. To use it you must be using Maven 3.2 (or better).

|  |
| --- |
| [Note] |
| Refer to the [Spring Boot Maven Plugin Site](http://docs.spring.io/spring-boot/docs/1.5.7.RELEASE/maven-plugin/) for complete plugin documentation. |

## 66.1 Including the plugin

To use the Spring Boot Maven Plugin simply include the appropriate XML in the plugins section of your pom.xml

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

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">

<modelVersion>4.0.0</modelVersion>

*<!-- ... -->*

<build>

<plugins>

<plugin>

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

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

<version>1.5.7.RELEASE</version>

<executions>

<execution>

<goals>

<goal>repackage</goal>

</goals>

</execution>

</executions>

</plugin>

</plugins>

</build>

</project>

This configuration will repackage a jar or war that is built during the package phase of the Maven lifecycle. The following example shows both the repackaged jar, as well as the original jar, in the target directory:

$ mvn package

$ ls target/\*.jar

target/myproject-1.0.0.jar target/myproject-1.0.0.jar.original

If you don’t include the <execution/> configuration as above, you can run the plugin on its own (but only if the package goal is used as well). For example:

$ mvn package spring-boot:repackage

$ ls target/\*.jar

target/myproject-1.0.0.jar target/myproject-1.0.0.jar.original

If you are using a milestone or snapshot release you will also need to add appropriate pluginRepository elements:

<pluginRepositories>

<pluginRepository>

<id>spring-snapshots</id>

<url>http://repo.spring.io/snapshot</url>

</pluginRepository>

<pluginRepository>

<id>spring-milestones</id>

<url>http://repo.spring.io/milestone</url>

</pluginRepository>

</pluginRepositories>

## 66.2 Packaging executable jar and war files

Once spring-boot-maven-plugin has been included in your pom.xml it will automatically attempt to rewrite archives to make them executable using the spring-boot:repackage goal. You should configure your project to build a jar or war (as appropriate) using the usual packaging element:

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

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">

*<!-- ... -->*

<packaging>jar</packaging>

*<!-- ... -->*

</project>

Your existing archive will be enhanced by Spring Boot during the package phase. The main class that you want to launch can either be specified using a configuration option, or by adding a Main-Class attribute to the manifest in the usual way. If you don’t specify a main class the plugin will search for a class with apublic static void main(String[] args) method.

To build and run a project artifact, you can type the following:

$ mvn package

$ java -jar target/mymodule-0.0.1-SNAPSHOT.jar

To build a war file that is both executable and deployable into an external container you need to mark the embedded container dependencies as “provided”, e.g:

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

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">

*<!-- ... -->*

<packaging>war</packaging>

*<!-- ... -->*

<dependencies>

<dependency>

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

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

</dependency>

<dependency>

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

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

<scope>provided</scope>

</dependency>

*<!-- ... -->*

</dependencies>

</project>

|  |
| --- |
| [Tip] |
| See the “[Section 85.1, “Create a deployable war file”](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto-create-a-deployable-war-file)” section for more details on how to create a deployable war file. |

Advanced configuration options and examples are available in the [plugin info page](http://docs.spring.io/spring-boot/docs/1.5.7.RELEASE/maven-plugin/).

## 67. Spring Boot Gradle plugin

The Spring Boot Gradle Plugin provides Spring Boot support in Gradle, allowing you to package executable jar or war archives, run Spring Boot applications and use the dependency management provided by spring-boot-dependencies.

## 67.1 Including the plugin

To use the Spring Boot Gradle Plugin configure it using the plugins block:

plugins {

id 'org.springframework.boot' version '1.5.7.RELEASE'

}

## 67.2 Gradle dependency management

The spring-boot plugin automatically applies the [Dependency Management Plugin](https://github.com/spring-gradle-plugins/dependency-management-plugin/) and configures it to import the spring-boot-starter-parent bom. This provides a similar dependency management experience to the one that is enjoyed by Maven users. For example, it allows you to omit version numbers when declaring dependencies that are managed in the bom. To make use of this functionality, simply declare dependencies in the usual way, but leave the version number empty:

dependencies {

compile("org.springframework.boot:spring-boot-starter-web")

compile("org.thymeleaf:thymeleaf-spring4")

compile("nz.net.ultraq.thymeleaf:thymeleaf-layout-dialect")

}

|  |
| --- |
| [Note] |
| The version of the spring-boot gradle plugin that you declare determines the version of the spring-boot-starter-parent bom that is imported (this ensures that builds are always repeatable). You should always set the version of the spring-boot gradle plugin to the actual Spring Boot version that you wish to use. Details of the versions that are provided can be found in the [appendix](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#appendix-dependency-versions). |

To learn more about the capabilities of the Dependency Management Plugin, please refer to its [documentation](https://github.com/spring-gradle-plugins/dependency-management-plugin/blob/master/README.md).

## 67.3 Packaging executable jar and war files

Once the spring-boot plugin has been applied to your project it will automatically attempt to rewrite archives to make them executable using the [bootRepackagetask](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#build-tool-plugins-gradle-repackage-configuration). You should configure your project to build a jar or war (as appropriate) in the usual way.

The main class that you want to launch can either be specified using a configuration option, or by adding a Main-Class attribute to the manifest. If you don’t specify a main class the plugin will search for a class with a public static void main(String[] args) method.

|  |
| --- |
| [Tip] |
| Check [Section 67.6, “Repackage configuration”](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#build-tool-plugins-gradle-repackage-configuration) for a full list of configuration options. |

To build and run a project artifact, you can type the following:

$ gradle build

$ java -jar build/libs/mymodule-0.0.1-SNAPSHOT.jar

To build a war file that is both executable and deployable into an external container, you need to mark the embedded container dependencies as belonging to the war plugin’s providedRuntime configuration, e.g.:

...

apply plugin: 'war'

war {

baseName = 'myapp'

version = '0.5.0'

}

repositories {

jcenter()

maven { url "http://repo.spring.io/libs-snapshot" }

}

dependencies {

compile("org.springframework.boot:spring-boot-starter-web")

providedRuntime("org.springframework.boot:spring-boot-starter-tomcat")

...

}

|  |
| --- |
| [Tip] |
| See the “[Section 85.1, “Create a deployable war file”](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto-create-a-deployable-war-file)” section for more details on how to create a deployable war file. |

## 67.4 Running a project in-place

To run a project in place without building a jar first you can use the “bootRun” task:

$ gradle bootRun

If [devtools](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#using-boot-devtools) has been added to your project it will automatically monitor your application for changes. Alternatively, you can also run the application so that your static classpath resources (i.e. in src/main/resources by default) are reloadable in the live application, which can be helpful at development time.

bootRun {

addResources = true

}

Making static classpath resources reloadable means that bootRun does not use the output of the processResources task, i.e., when invoked using bootRun, your application will use the resources in their unprocessed form.

## 67.5 Spring Boot plugin configuration

The gradle plugin automatically extends your build script DSL with a springBoot element for global configuration of the Boot plugin. Set the appropriate properties as you would with any other Gradle extension (see below for a list of configuration options):

springBoot {

backupSource = false

}

## 67.6 Repackage configuration

The plugin adds a bootRepackage task which you can also configure directly, e.g.:

bootRepackage {

mainClass = 'demo.Application'

}

The following configuration options are available:

| **Name** | **Description** |
| --- | --- |
| enabled | Boolean flag to switch the repackager off (sometimes useful if you want the other Boot features but not this one) |
| mainClass | The main class that should be run. If not specified, and you have applied the application plugin, the mainClassNameproject property will be used. If the application plugin has not been applied or no mainClassName has been specified, the archive will be searched for a suitable class. "Suitable" means a unique class with a well-formed main() method (if more than one is found the build will fail). If you have applied the application plugin, the main class can also be specified via its "run" task (main property) and/or its "startScripts" task (mainClassName property) as an alternative to using the "springBoot" configuration. |
| classifier | A file name segment (before the extension) to add to the archive, so that the original is preserved in its original location. Defaults to null in which case the archive is repackaged in place. The default is convenient for many purposes, but if you want to use the original jar as a dependency in another project you must use a classifier to define the executable archive. |
| withJarTask | The name or value of the Jar task (defaults to all tasks of type Jar) which is used to locate the archive to repackage. |
| customConfiguration | The name of the custom configuration which is used to populate the nested lib directory (without specifying this you get all compile and runtime dependencies). |
| executable | Boolean flag to indicate if jar files are fully executable on Unix like operating systems. Defaults to false. |
| embeddedLaunchScript | The embedded launch script to prepend to the front of the jar if it is fully executable. If not specified the 'Spring Boot' default script will be used. |
| embeddedLaunchScriptProperties | Additional properties that to be expanded in the launch script. The default script supports a mode property which can contain the values auto, service or run. |
| excludeDevtools | Boolean flag to indicate if the devtools jar should be excluded from the repackaged archives. Defaults to true. |

## 67.7 Repackage with custom Gradle configuration

Sometimes it may be more appropriate to not package default dependencies resolved from compile, runtime and provided scopes. If the created executable jar file is intended to be run as it is, you need to have all dependencies nested inside it; however, if the plan is to explode a jar file and run the main class manually, you may already have some of the libraries available via CLASSPATH. This is a situation where you can repackage your jar with a different set of dependencies.

Using a custom configuration will automatically disable dependency resolving from compile, runtime and provided scopes. Custom configuration can be either defined globally (inside the springBoot section) or per task.

task clientJar(type: Jar) {

appendix = 'client'

from sourceSets.main.output

exclude('\*\*/\*Something\*')

}

task clientBoot(type: BootRepackage, dependsOn: clientJar) {

withJarTask = clientJar

customConfiguration = "mycustomconfiguration"

}

In above example, we created a new clientJar Jar task to package a customized file set from your compiled sources. Then we created a new clientBootBootRepackage task and instructed it to work with only clientJar task and mycustomconfiguration.

configurations {

mycustomconfiguration.exclude group: 'log4j'

}

dependencies {

mycustomconfiguration configurations.runtime

}

The configuration that we are referring to in BootRepackage is a normal [Gradle configuration](https://docs.gradle.org/2.14.1/dsl/org.gradle.api.artifacts.Configuration.html). In the above example we created a new configuration namedmycustomconfiguration instructing it to derive from a runtime and exclude the log4j group. If the clientBoot task is executed, the repackaged boot jar will have all dependencies from runtime but no log4j jars.

### 67.7.1 Configuration options

The following configuration options are available:

| **Name** | **Description** |
| --- | --- |
| mainClass | The main class that should be run by the executable archive. |
| providedConfiguration | The name of the provided configuration (defaults to providedRuntime). |
| backupSource | If the original source archive should be backed-up before being repackaged (defaults to true). |
| customConfiguration | The name of the custom configuration. |
| layout | The type of archive, corresponding to how the dependencies are laid out inside (defaults to a guess based on the archive type). See[available layouts for more details](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#build-tool-plugins-gradle-configuration-layouts). |
| layoutFactory | A layout factory that can be used if a custom layout is required. Alternative layouts can be provided by 3rd parties. Layout factories are only used when layout is not specified. |
| requiresUnpack | A list of dependencies (in the form “groupId:artifactId” that must be unpacked from fat jars in order to run. Items are still packaged into the fat jar, but they will be automatically unpacked when it runs. |

### 67.7.2 Available layouts

The layout attribute configures the format of the archive and whether the bootstrap loader should be included or not. The following layouts are available:

| **Name** | **Description** | **Executable** |
| --- | --- | --- |
| JAR | Regular executable [JAR layout](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#executable-jar-jar-file-structure). | Yes |
| WAR | Executable [WAR layout](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#executable-jar-war-file-structure). provided dependencies are placed in WEB-INF/lib-provided to avoid any clash when the war is deployed in a servlet container. | Yes |
| ZIP (alias to DIR) | Similar to JAR layout, using [PropertiesLauncher](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#executable-jar-property-launcher-features). | Yes |
| MODULE | Bundle dependencies (excluding those with provided scope) and project resources. | No |
| NONE | Bundle all dependencies and project resources. | No |

### 67.7.3 Using a custom layout

If you have custom requirements for how to arrange the dependencies and loader classes inside the repackaged jar, you can use a custom layout. Any library which defines one or more LayoutFactory implementations can be added to the build script dependencies and then the layout factory becomes available in the springBootconfiguration. For example:

buildscript {

repositories {

mavenCentral()

}

dependencies {

classpath("org.springframework.boot:spring-boot-gradle-plugin:1.5.7.RELEASE")

classpath("com.example:custom-layout:1.0.0")

}

}

springBoot {

layoutFactory = **new** com.example.CustomLayoutFactory()

}

|  |
| --- |
| [Note] |
| If there is only one custom LayoutFactory on the build classpath and it is listed in META-INF/spring.factories then it is unnecessary to explicitly set it in the springBoot configuration. Layout factories are only used when no explicit layout is specified. |

## 67.8 Understanding how the Gradle plugin works

When spring-boot is applied to your Gradle project a default task named bootRepackage is created automatically. The bootRepackage task depends on Gradle assemble task, and when executed, it tries to find all jar artifacts whose qualifier is empty (i.e. tests and sources jars are automatically skipped).

Due to the fact that bootRepackage finds 'all' created jar artifacts, the order of Gradle task execution is important. Most projects only create a single jar file, so usually this is not an issue; however, if you are planning to create a more complex project setup, with custom Jar and BootRepackage tasks, there are few tweaks to consider.

If you are 'just' creating custom jar files from your project you can simply disable default jar and bootRepackage tasks:

jar.enabled = false

bootRepackage.enabled = false

Another option is to instruct the default bootRepackage task to only work with a default jar task.

bootRepackage.withJarTask = jar

If you have a default project setup where the main jar file is created and repackaged, 'and' you still want to create additional custom jars, you can combine your custom repackage tasks together and use dependsOn so that the bootJars task will run after the default bootRepackage task is executed:

task bootJars

bootJars.dependsOn = [clientBoot1,clientBoot2,clientBoot3]

build.dependsOn(bootJars)

All the above tweaks are usually used to avoid situations where an already created boot jar is repackaged again. Repackaging an existing boot jar will not break anything, but you may find that it includes unnecessary dependencies.

## 67.9 Publishing artifacts to a Maven repository using Gradle

If you are [declaring dependencies without versions](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/) and you want to publish artifacts to a Maven repository you will need to configure the Maven publication with details of Spring Boot’s dependency management. This can be achieved by configuring it to publish poms that inherit from spring-boot-starter-parent or that import dependency management from spring-boot-dependencies. The exact details of this configuration depend on how you’re using Gradle and how you’re trying to publish the artifacts.

### 67.9.1 Configuring Gradle to produce a pom that inherits dependency management

The following is an example of configuring Gradle to generate a pom that inherits from spring-boot-starter-parent. Please refer to the [Gradle User Guide](https://docs.gradle.org/2.14.1/userguide/userguide.html) for further information.

uploadArchives {

repositories {

mavenDeployer {

pom {

project {

parent {

groupId "org.springframework.boot"

artifactId "spring-boot-starter-parent"

version "1.5.7.RELEASE"

}

}

}

}

}

}

### 67.9.2 Configuring Gradle to produce a pom that imports dependency management

The following is an example of configuring Gradle to generate a pom that imports the dependency management provided by spring-boot-dependencies. Please refer to the [Gradle User Guide](http://www.gradle.org/docs/current/userguide/userguide.html) for further information.

uploadArchives {

repositories {

mavenDeployer {

pom {

project {

dependencyManagement {

dependencies {

dependency {

groupId "org.springframework.boot"

artifactId "spring-boot-dependencies"

version "1.5.7.RELEASE"

type "pom"

scope "import"

}

}

}

}

}

}

}

}

## 68. Spring Boot AntLib module

The Spring Boot AntLib module provides basic Spring Boot support for Apache Ant. You can use the module to create executable jars. To use the module you need to declare an additional spring-boot namespace in your build.xml:

<project xmlns:ivy="antlib:org.apache.ivy.ant"

xmlns:spring-boot="antlib:org.springframework.boot.ant"

name="myapp" default="build">

...

</project>

You’ll need to remember to start Ant using the -lib option, for example:

$ ant -lib <folder containing spring-boot-antlib-1.5.7.RELEASE.jar>

|  |
| --- |
| [Tip] |
| The “Using Spring Boot” section includes a more complete example of [using Apache Ant with spring-boot-antlib](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#using-boot-ant) |

## 68.1 Spring Boot Ant tasks

Once the spring-boot-antlib namespace has been declared, the following additional tasks are available.

### 68.1.1 spring-boot:exejar

The exejar task can be used to creates a Spring Boot executable jar. The following attributes are supported by the task:

| **Attribute** | **Description** | **Required** |
| --- | --- | --- |
| destfile | The destination jar file to create | Yes |
| classes | The root directory of Java class files | Yes |
| start-class | The main application class to run | No (default is first class found declaring a *main* method) |

The following nested elements can be used with the task:

| **Element** | **Description** |
| --- | --- |
| resources | One or more [Resource Collections](https://ant.apache.org/manual/Types/resources.html#collection) describing a set of [Resources](https://ant.apache.org/manual/Types/resources.html) that should be added to the content of the created jar file. |
| lib | One or more [Resource Collections](https://ant.apache.org/manual/Types/resources.html#collection) that should be added to the set of jar libraries that make up the runtime dependency classpath of the application. |

### 68.1.2 Examples

**Specify start-class.**

<spring-boot:exejar destfile="target/my-application.jar"

classes="target/classes" start-class="com.foo.MyApplication">

<resources>

<fileset dir="src/main/resources" />

</resources>

<lib>

<fileset dir="lib" />

</lib>

</spring-boot:exejar>

**Detect start-class.**

<exejar destfile="target/my-application.jar" classes="target/classes">

<lib>

<fileset dir="lib" />

</lib>

</exejar>

## 68.2 spring-boot:findmainclass

The findmainclass task is used internally by exejar to locate a class declaring a main. You can also use this task directly in your build if needed. The following attributes are supported

| **Attribute** | **Description** | **Required** |
| --- | --- | --- |
| classesroot | The root directory of Java class files | Yes (unless *mainclass* is specified) |
| mainclass | Can be used to short-circuit the main class search | No |
| property | The Ant property that should be set with the result | No (result will be logged if unspecified) |

### 68.2.1 Examples

**Find and log.**

<findmainclass classesroot="target/classes" />

**Find and set.**

<findmainclass classesroot="target/classes" property="main-class" />

**Override and set.**

<findmainclass mainclass="com.foo.MainClass" property="main-class" />

## 69. Supporting other build systems

If you want to use a build tool other than Maven, Gradle or Ant, you will likely need to develop your own plugin. Executable jars need to follow a specific format and certain entries need to be written in an uncompressed form (see the [*executable jar format*](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#executable-jar) section in the appendix for details).

The Spring Boot Maven and Gradle plugins both make use of spring-boot-loader-tools to actually generate jars. You are also free to use this library directly yourself if you need to.

## 69.1 Repackaging archives

To repackage an existing archive so that it becomes a self-contained executable archive use org.springframework.boot.loader.tools.Repackager. The Repackager class takes a single constructor argument that refers to an existing jar or war archive. Use one of the two available repackage() methods to either replace the original file or write to a new destination. Various settings can also be configured on the repackager before it is run.

## 69.2 Nested libraries

When repackaging an archive you can include references to dependency files using the org.springframework.boot.loader.tools.Libraries interface. We don’t provide any concrete implementations of Libraries here as they are usually build system specific.

If your archive already includes libraries you can use Libraries.NONE.

## 69.3 Finding a main class

If you don’t use Repackager.setMainClass() to specify a main class, the repackager will use [ASM](http://asm.ow2.org/) to read class files and attempt to find a suitable class with a public static void main(String[] args) method. An exception is thrown if more than one candidate is found.

## 69.4 Example repackage implementation

Here is a typical example repackage:

Repackager repackager = **new** Repackager(sourceJarFile);

repackager.setBackupSource(false);

repackager.repackage(**new** Libraries() {

*@Override*

**public** **void** doWithLibraries(LibraryCallback callback) **throws** IOException {

*// Build system specific implementation, callback for each dependency*

*// callback.library(new Library(nestedFile, LibraryScope.COMPILE));*

}

});

## 70. What to read next

If you’re interested in how the build tool plugins work you can look at the [spring-boot-tools](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot-tools) module on GitHub. More technical details of the [executable jar format](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#executable-jar) are covered in the appendix.

If you have specific build-related questions you can check out the “[how-to](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto)” guides.

# Part IX. ‘How-to’ guides

This section provides answers to some common ‘how do I do that…​’ type of questions that often arise when using Spring Boot. This is by no means an exhaustive list, but it does cover quite a lot.

If you are having a specific problem that we don’t cover here, you might want to check out [stackoverflow.com](https://stackoverflow.com/tags/spring-boot) to see if someone has already provided an answer; this is also a great place to ask new questions (please use the spring-boot tag).

We’re also more than happy to extend this section; If you want to add a ‘how-to’ you can send us a [pull request](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE).

## 71. Spring Boot application

## 71.1 Create your own FailureAnalyzer

[FailureAnalyzer](http://docs.spring.io/spring-boot/docs/1.5.7.RELEASE/api/org/springframework/boot/diagnostics/FailureAnalyzer.html) is a great way to intercept an exception on startup and turn it into a human-readable message, wrapped into a [FailureAnalysis](http://docs.spring.io/spring-boot/docs/1.5.7.RELEASE/api/org/springframework/boot/diagnostics/FailureAnalysis.html). Spring Boot provides such analyzer for application context related exceptions, JSR-303 validations and more. It is actually very easy to create your own.

AbstractFailureAnalyzer is a convenient extension of FailureAnalyzer that checks the presence of a specified exception type in the exception to handle. You can extend from that so that your implementation gets a chance to handle the exception only when it is actually present. If for whatever reason you can’t handle the exception, return null to give another implementation a chance to handle the exception.

FailureAnalyzer implementations are to be registered in a META-INF/spring.factories: the following registers ProjectConstraintViolationFailureAnalyzer:

org.springframework.boot.diagnostics.FailureAnalyzer=\

com.example.ProjectConstraintViolationFailureAnalyzer

## 71.2 Troubleshoot auto-configuration

The Spring Boot auto-configuration tries its best to ‘do the right thing’, but sometimes things fail and it can be hard to tell why.

There is a really useful ConditionEvaluationReport available in any Spring Boot ApplicationContext. You will see it if you enable DEBUG logging output. If you use the spring-boot-actuator there is also an autoconfig endpoint that renders the report in JSON. Use that to debug the application and see what features have been added (and which not) by Spring Boot at runtime.

Many more questions can be answered by looking at the source code and the Javadoc. Some rules of thumb:

* Look for classes called \*AutoConfiguration and read their sources, in particular the @Conditional\* annotations to find out what features they enable and when. Add --debug to the command line or a System property -Ddebug to get a log on the console of all the auto-configuration decisions that were made in your app. In a running Actuator app look at the autoconfig endpoint (‘/autoconfig’ or the JMX equivalent) for the same information.
* Look for classes that are @ConfigurationProperties (e.g. [ServerProperties](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/web/ServerProperties.java)) and read from there the available external configuration options. The@ConfigurationProperties has a name attribute which acts as a prefix to external properties, thus ServerProperties has prefix="server" and its configuration properties are server.port, server.address etc. In a running Actuator app look at the configprops endpoint.
* Look for use of RelaxedPropertyResolver to pull configuration values explicitly out of the Environment. It often is used with a prefix.
* Look for @Value annotations that bind directly to the Environment. This is less flexible than the RelaxedPropertyResolver approach, but does allow some relaxed binding, specifically for OS environment variables (so CAPITALS\_AND\_UNDERSCORES are synonyms for period.separated).
* Look for @ConditionalOnExpression annotations that switch features on and off in response to SpEL expressions, normally evaluated with placeholders resolved from the Environment.

## 71.3 Customize the Environment or ApplicationContext before it starts

A SpringApplication has ApplicationListeners and ApplicationContextInitializers that are used to apply customizations to the context or environment. Spring Boot loads a number of such customizations for use internally from META-INF/spring.factories. There is more than one way to register additional ones:

* Programmatically per application by calling the addListeners and addInitializers methods on SpringApplication before you run it.
* Declaratively per application by setting context.initializer.classes or context.listener.classes.
* Declaratively for all applications by adding a META-INF/spring.factories and packaging a jar file that the applications all use as a library.

The SpringApplication sends some special ApplicationEvents to the listeners (even some before the context is created), and then registers the listeners for events published by the ApplicationContext as well. See [*Section 23.5, “Application events and listeners”*](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-application-events-and-listeners) in the ‘Spring Boot features’ section for a complete list.

It is also possible to customize the Environment before the application context is refreshed using EnvironmentPostProcessor. Each implementation should be registered in META-INF/spring.factories:

org.springframework.boot.env.EnvironmentPostProcessor=com.example.YourEnvironmentPostProcessor

The implementation can load arbitrary files and add them to the Environment. For instance, this example loads a YAML configuration file from the classpath:

**public** **class** EnvironmentPostProcessorExample **implements** EnvironmentPostProcessor {

**private** **final** YamlPropertySourceLoader loader = **new** YamlPropertySourceLoader();

*@Override*

**public** **void** postProcessEnvironment(ConfigurableEnvironment environment,

SpringApplication application) {

Resource path = **new** ClassPathResource("com/example/myapp/config.yml");

PropertySource<?> propertySource = loadYaml(path);

environment.getPropertySources().addLast(propertySource);

}

**private** PropertySource<?> loadYaml(Resource path) {

**if** (!path.exists()) {

**throw** **new** IllegalArgumentException("Resource " + path + " does not exist");

}

**try** {

**return** **this**.loader.load("custom-resource", path, null);

}

**catch** (IOException ex) {

**throw** **new** IllegalStateException(

"Failed to load yaml configuration from " + path, ex);

}

}

}

|  |
| --- |
| [Tip] |
| The Environment will already have been prepared with all the usual property sources that Spring Boot loads by default. It is therefore possible to get the location of the file from the environment. This example adds the custom-resource property source at the end of the list so that a key defined in any of the usual other locations takes precedence. A custom implementation may obviously defines another order. |
| [Note] |
| While using @PropertySource on your @SpringBootApplication seems convenient and easy enough to load a custom resource in the Environment, we do not recommend it as Spring Boot prepares the Environment before the ApplicationContext is refreshed. Any key defined via @PropertySource will be loaded too late to have any effect on auto-configuration. |

## 71.4 Build an ApplicationContext hierarchy (adding a parent or root context)

You can use the ApplicationBuilder class to create parent/child ApplicationContext hierarchies. See [*Section 23.4, “Fluent builder API”*](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-fluent-builder-api) in the ‘Spring Boot features’ section for more information.

## 71.5 Create a non-web application

Not all Spring applications have to be web applications (or web services). If you want to execute some code in a main method, but also bootstrap a Spring application to set up the infrastructure to use, then it’s easy with the SpringApplication features of Spring Boot. A SpringApplication changes its ApplicationContext class depending on whether it thinks it needs a web application or not. The first thing you can do to help it is to just leave the servlet API dependencies off the classpath. If you can’t do that (e.g. you are running 2 applications from the same code base) then you can explicitly call setWebEnvironment(false) on your SpringApplicationinstance, or set the applicationContextClass property (through the Java API or with external properties). Application code that you want to run as your business logic can be implemented as a CommandLineRunner and dropped into the context as a @Bean definition.

## 72. Properties & configuration

## 72.1 Automatically expand properties at build time

Rather than hardcoding some properties that are also specified in your project’s build configuration, you can automatically expand them using the existing build configuration instead. This is possible in both Maven and Gradle.

### 72.1.1 Automatic property expansion using Maven

You can automatically expand properties from the Maven project using resource filtering. If you use the spring-boot-starter-parent you can then refer to your Maven ‘project properties’ via @..@ placeholders, e.g.

app.encoding=@project.build.sourceEncoding@

app.java.version=@java.version@

|  |
| --- |
| [Note] |
| Only production configuration is filtered that way (i.e. no filtering is applied on src/test/resources). |
| [Tip] |
| The spring-boot:run can add src/main/resources directly to the classpath (for hot reloading purposes) if you enable the addResources flag. This circumvents the resource filtering and this feature. You can use the exec:java goal instead or customize the plugin’s configuration, see the [plugin usage page](http://docs.spring.io/spring-boot/docs/1.5.7.RELEASE/maven-plugin/usage.html) for more details. |

If you don’t use the starter parent, in your pom.xml you need (inside the <build/> element):

<resources>

<resource>

<directory>src/main/resources</directory>

<filtering>true</filtering>

</resource>

</resources>

and (inside <plugins/>):

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-resources-plugin</artifactId>

<version>2.7</version>

<configuration>

<delimiters>

<delimiter>@</delimiter>

</delimiters>

<useDefaultDelimiters>false</useDefaultDelimiters>

</configuration>

</plugin>

|  |
| --- |
| [Note] |
| The useDefaultDelimiters property is important if you are using standard Spring placeholders in your configuration (e.g. ${foo}). These may be expanded by the build if that property is not set to false. |

### 72.1.2 Automatic property expansion using Gradle

You can automatically expand properties from the Gradle project by configuring the Java plugin’s processResources task to do so:

processResources {

expand(project.properties)

}

You can then refer to your Gradle project’s properties via placeholders, e.g.

app.name=${name}

app.description=${description}

|  |
| --- |
| [Note] |
| Gradle’s expand method uses Groovy’s SimpleTemplateEngine which transforms ${..} tokens. The ${..} style conflicts with Spring’s own property placeholder mechanism. To use Spring property placeholders together with automatic expansion the Spring property placeholders need to be escaped like \${..}. |

## 72.2 Externalize the configuration of SpringApplication

A SpringApplication has bean properties (mainly setters) so you can use its Java API as you create the application to modify its behavior. Or you can externalize the configuration using properties in spring.main.\*. E.g. in application.properties you might have.

spring.main.web-environment=false

spring.main.banner-mode=off

and then the Spring Boot banner will not be printed on startup, and the application will not be a web application.

|  |
| --- |
| [Note] |
| The example above also demonstrates how flexible binding allows the use of underscores (\_) as well as dashes (-) in property names. |

Properties defined in external configuration overrides the values specified via the Java API with the notable exception of the sources used to create the ApplicationContext. Let’s consider this application

**new** SpringApplicationBuilder()

.bannerMode(Banner.Mode.OFF)

.sources(demo.MyApp.**class**)

.run(args);

used with the following configuration:

spring.main.sources=com.acme.Config,com.acme.ExtraConfig

spring.main.banner-mode=console

The actual application will now show the banner (as overridden by configuration) and use three sources for the ApplicationContext (in that order): demo.MyApp, com.acme.Config, com.acme.ExtraConfig.

## 72.3 Change the location of external properties of an application

By default properties from different sources are added to the Spring Environment in a defined order (see [*Chapter 24, Externalized Configuration*](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-external-config) in the ‘Spring Boot features’ section for the exact order).

A nice way to augment and modify this is to add @PropertySource annotations to your application sources. Classes passed to the SpringApplication static convenience methods, and those added using setSources() are inspected to see if they have @PropertySources, and if they do, those properties are added to the Environment early enough to be used in all phases of the ApplicationContext lifecycle. Properties added in this way have lower priority than any added using the default locations (e.g. application.properties), system properties, environment variables or the command line.

You can also provide System properties (or environment variables) to change the behavior:

* spring.config.name (SPRING\_CONFIG\_NAME), defaults to application as the root of the file name.
* spring.config.location (SPRING\_CONFIG\_LOCATION) is the file to load (e.g. a classpath resource or a URL). A separate Environment property source is set up for this document and it can be overridden by system properties, environment variables or the command line.

No matter what you set in the environment, Spring Boot will always load application.properties as described above. If YAML is used then files with the ‘.yml’ extension are also added to the list by default.

Spring Boot logs the configuration files that are loaded at DEBUG level and the candidates it has not found at TRACE level.

See [ConfigFileApplicationListener](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot/src/main/java/org/springframework/boot/context/config/ConfigFileApplicationListener.java) for more detail.

## 72.4 Use ‘short’ command line arguments

Some people like to use (for example) --port=9000 instead of --server.port=9000 to set configuration properties on the command line. You can easily enable this by using placeholders in application.properties, e.g.

server.port=${port:8080}

|  |
| --- |
| [Tip] |
| If you are inheriting from the spring-boot-starter-parent POM, the default filter token of the maven-resources-plugins has been changed from ${\*} to @ (i.e. @maven.token@ instead of ${maven.token}) to prevent conflicts with Spring-style placeholders. If you have enabled maven filtering for the application.properties directly, you may want to also change the default filter token to use [other delimiters](https://maven.apache.org/plugins/maven-resources-plugin/resources-mojo.html#delimiters). |
| [Note] |
| In this specific case the port binding will work in a PaaS environment like Heroku and Cloud Foundry, since in those two platforms the PORT environment variable is set automatically and Spring can bind to capitalized synonyms for Environment properties. |

## 72.5 Use YAML for external properties

YAML is a superset of JSON and as such is a very convenient syntax for storing external properties in a hierarchical format. E.g.

spring:

application:

name: cruncher

datasource:

driverClassName: com.mysql.jdbc.Driver

url: jdbc:mysql://localhost/test

server:

port: 9000

Create a file called application.yml and stick it in the root of your classpath, and also add snakeyaml to your dependencies (Maven coordinates org.yaml:snakeyaml, already included if you use the spring-boot-starter). A YAML file is parsed to a Java Map<String,Object> (like a JSON object), and Spring Boot flattens the map so that it is 1-level deep and has period-separated keys, a lot like people are used to with Properties files in Java.

The example YAML above corresponds to an application.properties file

spring.application.name=cruncher

spring.datasource.driverClassName=com.mysql.jdbc.Driver

spring.datasource.url=jdbc:mysql://localhost/test

server.port=9000

See [*Section 24.6, “Using YAML instead of Properties”*](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-external-config-yaml) in the ‘Spring Boot features’ section for more information about YAML.

## 72.6 Set the active Spring profiles

The Spring Environment has an API for this, but normally you would set a System property (spring.profiles.active) or an OS environment variable (SPRING\_PROFILES\_ACTIVE). E.g. launch your application with a -D argument (remember to put it before the main class or jar archive):

$ java -jar -Dspring.profiles.active=production demo-0.0.1-SNAPSHOT.jar

In Spring Boot you can also set the active profile in application.properties, e.g.

spring.profiles.active=production

A value set this way is replaced by the System property or environment variable setting, but not by the SpringApplicationBuilder.profiles() method. Thus the latter Java API can be used to augment the profiles without changing the defaults.

See [*Chapter 25, Profiles*](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-profiles) in the ‘Spring Boot features’ section for more information.

## 72.7 Change configuration depending on the environment

A YAML file is actually a sequence of documents separated by --- lines, and each document is parsed separately to a flattened map.

If a YAML document contains a spring.profiles key, then the profiles value (comma-separated list of profiles) is fed into the SpringEnvironment.acceptsProfiles() and if any of those profiles is active that document is included in the final merge (otherwise not).

Example:

server:

port: 9000

*---*

spring:

profiles: development

server:

port: 9001

*---*

spring:

profiles: production

server:

port: 0

In this example the default port is 9000, but if the Spring profile ‘development’ is active then the port is 9001, and if ‘production’ is active then it is 0.

The YAML documents are merged in the order they are encountered (so later values override earlier ones).

To do the same thing with properties files you can use application-${profile}.properties to specify profile-specific values.

## 72.8 Discover built-in options for external properties

Spring Boot binds external properties from application.properties (or .yml) (and other places) into an application at runtime. There is not (and technically cannot be) an exhaustive list of all supported properties in a single location because contributions can come from additional jar files on your classpath.

A running application with the Actuator features has a configprops endpoint that shows all the bound and bindable properties available through @ConfigurationProperties.

The appendix includes an [application.properties](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#common-application-properties) example with a list of the most common properties supported by Spring Boot. The definitive list comes from searching the source code for @ConfigurationProperties and @Value annotations, as well as the occasional use of RelaxedPropertyResolver.

## 73. Embedded servlet containers

## 73.1 Add a Servlet, Filter or Listener to an application

There are two ways to add Servlet, Filter, ServletContextListener and the other listeners supported by the Servlet spec to your application. You can either provide Spring beans for them, or enable scanning for Servlet components.

### 73.1.1 Add a Servlet, Filter or Listener using a Spring bean

To add a Servlet, Filter, or Servlet \*Listener provide a @Bean definition for it. This can be very useful when you want to inject configuration or dependencies. However, you must be very careful that they don’t cause eager initialization of too many other beans because they have to be installed in the container very early in the application lifecycle (e.g. it’s not a good idea to have them depend on your DataSource or JPA configuration). You can work around restrictions like that by initializing them lazily when first used instead of on initialization.

In the case of Filters and Servlets you can also add mappings and init parameters by adding a FilterRegistrationBean or ServletRegistrationBeaninstead of or as well as the underlying component.

|  |
| --- |
| [Note] |
| If no dispatcherType is specified on a filter registration, it will match FORWARD,INCLUDE and REQUEST. If async has been enabled, it will match ASYNCas well.  If you are migrating a filter that has no dispatcher element in web.xml you will need to specify a dispatcherType yourself:  *@Bean*  **public** FilterRegistrationBean myFilterRegistration() {  FilterRegistrationBean registration = **new** FilterRegistrationBean();  registration.setDispatcherTypes(DispatcherType.REQUEST);  ....  **return** registration;  } |

#### Disable registration of a Servlet or Filter

As [described above](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto-add-a-servlet-filter-or-listener-as-spring-bean) any Servlet or Filter beans will be registered with the servlet container automatically. To disable registration of a particular Filter or Servlet bean create a registration bean for it and mark it as disabled. For example:

*@Bean*

**public** FilterRegistrationBean registration(MyFilter filter) {

FilterRegistrationBean registration = **new** FilterRegistrationBean(filter);

registration.setEnabled(false);

**return** registration;

}

### 73.1.2 Add Servlets, Filters, and Listeners using classpath scanning

@WebServlet, @WebFilter, and @WebListener annotated classes can be automatically registered with an embedded servlet container by annotating a @Configuration class with @ServletComponentScan and specifying the package(s) containing the components that you want to register. By default, @ServletComponentScan will scan from the package of the annotated class.

## 73.2 Change the HTTP port

In a standalone application the main HTTP port defaults to 8080, but can be set with server.port (e.g. in application.properties or as a System property). Thanks to relaxed binding of Environment values you can also use SERVER\_PORT (e.g. as an OS environment variable).

To switch off the HTTP endpoints completely, but still create a WebApplicationContext, use server.port=-1 (this is sometimes useful for testing).

For more details look at [*Section 27.3.4, “Customizing embedded servlet containers”*](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-customizing-embedded-containers) in the ‘Spring Boot features’ section, or the [ServerProperties](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/web/ServerProperties.java) source code.

## 73.3 Use a random unassigned HTTP port

To scan for a free port (using OS natives to prevent clashes) use server.port=0.

## 73.4 Discover the HTTP port at runtime

You can access the port the server is running on from log output or from the EmbeddedWebApplicationContext via its EmbeddedServletContainer. The best way to get that and be sure that it has initialized is to add a @Bean of type ApplicationListener<EmbeddedServletContainerInitializedEvent> and pull the container out of the event when it is published.

Tests that use @SpringBootTest(webEnvironment=WebEnvironment.RANDOM\_PORT) can also inject the actual port into a field using the @LocalServerPortannotation. For example:

*@RunWith(SpringJUnit4ClassRunner.class)*

*@SpringBootTest(webEnvironment=WebEnvironment.RANDOM\_PORT)*

**public** **class** MyWebIntegrationTests {

*@Autowired*

EmbeddedWebApplicationContext server;

*@LocalServerPort*

**int** port;

*// ...*

}

|  |
| --- |
| [Note] |
| @LocalServerPort is a meta-annotation for @Value("${local.server.port}"). Don’t try to inject the port in a regular application. As we just saw, the value is only set once the container has initialized; contrary to a test, application code callbacks are processed early (i.e. before the value is actually available). |

## 73.5 Configure SSL

SSL can be configured declaratively by setting the various server.ssl.\* properties, typically in application.properties or application.yml. For example:

server.port=8443

server.ssl.key-store=classpath:keystore.jks

server.ssl.key-store-password=secret

server.ssl.key-password=another-secret

See [Ssl](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot/src/main/java/org/springframework/boot/context/embedded/Ssl.java) for details of all of the supported properties.

Using configuration like the example above means the application will no longer support plain HTTP connector at port 8080. Spring Boot doesn’t support the configuration of both an HTTP connector and an HTTPS connector via application.properties. If you want to have both then you’ll need to configure one of them programmatically. It’s recommended to use application.properties to configure HTTPS as the HTTP connector is the easier of the two to configure programmatically. See the [spring-boot-sample-tomcat-multi-connectors](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot-samples/spring-boot-sample-tomcat-multi-connectors) sample project for an example.

## 73.6 Configure Access Logging

Access logs can be configured for Tomcat and Undertow via their respective namespaces.

For instance, the following logs access on Tomcat with a [custom pattern](https://tomcat.apache.org/tomcat-8.0-doc/config/valve.html#Access_Logging).

server.tomcat.basedir=my-tomcat

server.tomcat.accesslog.enabled=true

server.tomcat.accesslog.pattern=%t %a "%r" %s (%D ms)

|  |
| --- |
| [Note] |
| The default location for logs is a logs directory relative to the tomcat base dir and said directory is a temp directory by default so you may want to fix Tomcat’s base directory or use an absolute path for the logs. In the example above, the logs will be available in my-tomcat/logs relative to the working directory of the application. |

Access logging for undertow can be configured in a similar fashion

server.undertow.accesslog.enabled=true

server.undertow.accesslog.pattern=%t %a "%r" %s (%D ms)

Logs are stored in a logs directory relative to the working directory of the application. This can be customized via server.undertow.accesslog.directory.

## 73.7 Use behind a front-end proxy server

Your application might need to send 302 redirects or render content with absolute links back to itself. When running behind a proxy, the caller wants a link to the proxy, and not to the physical address of the machine hosting your app. Typically such situations are handled via a contract with the proxy, which will add headers to tell the back end how to construct links to itself.

If the proxy adds conventional X-Forwarded-For and X-Forwarded-Proto headers (most do this out of the box) the absolute links should be rendered correctly as long as server.use-forward-headers is set to true in your application.properties.

|  |
| --- |
| [Note] |
| If your application is running in Cloud Foundry or Heroku the server.use-forward-headers property will default to true if not specified. In all other instances it defaults to false. |

### 73.7.1 Customize Tomcat’s proxy configuration

If you are using Tomcat you can additionally configure the names of the headers used to carry “forwarded” information:

server.tomcat.remote-ip-header=x-your-remote-ip-header

server.tomcat.protocol-header=x-your-protocol-header

Tomcat is also configured with a default regular expression that matches internal proxies that are to be trusted. By default, IP addresses in 10/8, 192.168/16,169.254/16 and 127/8 are trusted. You can customize the valve’s configuration by adding an entry to application.properties, e.g.

server.tomcat.internal-proxies=192\\.168\\.\\d{1,3}\\.\\d{1,3}

|  |
| --- |
| [Note] |
| The double backslashes are only required when you’re using a properties file for configuration. If you are using YAML, single backslashes are sufficient and a value that’s equivalent to the one shown above would be 192\.168\.\d{1,3}\.\d{1,3}. |
| [Note] |
| You can trust all proxies by setting the internal-proxies to empty (but don’t do this in production). |

You can take complete control of the configuration of Tomcat’s RemoteIpValve by switching the automatic one off (i.e. set server.use-forward-headers=false) and adding a new valve instance in a TomcatEmbeddedServletContainerFactory bean.

## 73.8 Configure Tomcat

Generally you can follow the advice from [*Section 72.8, “Discover built-in options for external properties”*](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto-discover-build-in-options-for-external-properties) about @ConfigurationProperties (ServerProperties is the main one here), but also look at EmbeddedServletContainerCustomizer and various Tomcat-specific \*Customizers that you can add in one of those. The Tomcat APIs are quite rich so once you have access to the TomcatEmbeddedServletContainerFactory you can modify it in a number of ways. Or the nuclear option is to add your own TomcatEmbeddedServletContainerFactory.

## 73.9 Enable Multiple Connectors with Tomcat

Add a org.apache.catalina.connector.Connector to the TomcatEmbeddedServletContainerFactory which can allow multiple connectors, e.g. HTTP and HTTPS connector:

*@Bean*

**public** EmbeddedServletContainerFactory servletContainer() {

TomcatEmbeddedServletContainerFactory tomcat = **new** TomcatEmbeddedServletContainerFactory();

tomcat.addAdditionalTomcatConnectors(createSslConnector());

**return** tomcat;

}

**private** Connector createSslConnector() {

Connector connector = **new** Connector("org.apache.coyote.http11.Http11NioProtocol");

Http11NioProtocol protocol = (Http11NioProtocol) connector.getProtocolHandler();

**try** {

File keystore = **new** ClassPathResource("keystore").getFile();

File truststore = **new** ClassPathResource("keystore").getFile();

connector.setScheme("https");

connector.setSecure(true);

connector.setPort(8443);

protocol.setSSLEnabled(true);

protocol.setKeystoreFile(keystore.getAbsolutePath());

protocol.setKeystorePass("changeit");

protocol.setTruststoreFile(truststore.getAbsolutePath());

protocol.setTruststorePass("changeit");

protocol.setKeyAlias("apitester");

**return** connector;

}

**catch** (IOException ex) {

**throw** **new** IllegalStateException("can't access keystore: [" + "keystore"

+ "] or truststore: [" + "keystore" + "]", ex);

}

}

## 73.10 Use Tomcat’s LegacyCookieProcessor

The embedded Tomcat used by Spring Boot does not support "Version 0" of the Cookie format out of the box, and you may see the following error:

java.lang.IllegalArgumentException: An invalid character [32] was present in the Cookie value

If at all possible, you should consider updating your code to only store values compliant with later Cookie specifications. If, however, you’re unable to change the way that cookies are written, you can instead configure Tomcat to use a LegacyCookieProcessor. To switch to the LegacyCookieProcessor use anEmbeddedServletContainerCustomizer bean that adds a TomcatContextCustomizer:

*@Bean*

**public** EmbeddedServletContainerCustomizer cookieProcessorCustomizer() {

**return** **new** EmbeddedServletContainerCustomizer() {

*@Override*

**public** **void** customize(ConfigurableEmbeddedServletContainer container) {

**if** (container **instanceof** TomcatEmbeddedServletContainerFactory) {

((TomcatEmbeddedServletContainerFactory) container)

.addContextCustomizers(**new** TomcatContextCustomizer() {

*@Override*

**public** **void** customize(Context context) {

context.setCookieProcessor(**new** LegacyCookieProcessor());

}

});

}

}

};

}

## 73.11 Use Jetty instead of Tomcat

The Spring Boot starters (spring-boot-starter-web in particular) use Tomcat as an embedded container by default. You need to exclude those dependencies and include the Jetty one instead. Spring Boot provides Tomcat and Jetty dependencies bundled together as separate starters to help make this process as easy as possible.

Example in Maven:

<dependency>

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

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

<exclusions>

<exclusion>

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

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

</exclusion>

</exclusions>

</dependency>

<dependency>

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

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

</dependency>

Example in Gradle:

configurations {

compile.exclude module: "spring-boot-starter-tomcat"

}

dependencies {

compile("org.springframework.boot:spring-boot-starter-web:1.5.7.RELEASE")

compile("org.springframework.boot:spring-boot-starter-jetty:1.5.7.RELEASE")

*// ...*

}

## 73.12 Configure Jetty

Generally you can follow the advice from [*Section 72.8, “Discover built-in options for external properties”*](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto-discover-build-in-options-for-external-properties) about @ConfigurationProperties (ServerProperties is the main one here), but also look at EmbeddedServletContainerCustomizer. The Jetty APIs are quite rich so once you have access to the JettyEmbeddedServletContainerFactory you can modify it in a number of ways. Or the nuclear option is to add your own JettyEmbeddedServletContainerFactory.

## 73.13 Use Undertow instead of Tomcat

Using Undertow instead of Tomcat is very similar to [using Jetty instead of Tomcat](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto-use-jetty-instead-of-tomcat). You need to exclude the Tomcat dependencies and include the Undertow starter instead.

Example in Maven:

<dependency>

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

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

<exclusions>

<exclusion>

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

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

</exclusion>

</exclusions>

</dependency>

<dependency>

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

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

</dependency>

Example in Gradle:

configurations {

compile.exclude module: "spring-boot-starter-tomcat"

}

dependencies {

compile("org.springframework.boot:spring-boot-starter-web:1.5.7.RELEASE")

compile("org.springframework.boot:spring-boot-starter-undertow:1.5.7.RELEASE")

*// ...*

}

## 73.14 Configure Undertow

Generally you can follow the advice from [*Section 72.8, “Discover built-in options for external properties”*](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto-discover-build-in-options-for-external-properties) about @ConfigurationProperties (ServerProperties and ServerProperties.Undertow are the main ones here), but also look at EmbeddedServletContainerCustomizer. Once you have access to theUndertowEmbeddedServletContainerFactory you can use an UndertowBuilderCustomizer to modify Undertow’s configuration to meet your needs. Or the nuclear option is to add your own UndertowEmbeddedServletContainerFactory.

## 73.15 Enable Multiple Listeners with Undertow

Add an UndertowBuilderCustomizer to the UndertowEmbeddedServletContainerFactory and add a listener to the Builder:

*@Bean*

**public** UndertowEmbeddedServletContainerFactory embeddedServletContainerFactory() {

UndertowEmbeddedServletContainerFactory factory = **new** UndertowEmbeddedServletContainerFactory();

factory.addBuilderCustomizers(**new** UndertowBuilderCustomizer() {

*@Override*

**public** **void** customize(Builder builder) {

builder.addHttpListener(8080, "0.0.0.0");

}

});

**return** factory;

}

## 73.16 Use Tomcat 7.x or 8.0

Tomcat 7 & 8.0 work with Spring Boot, but the default is to use Tomcat 8.5. If you cannot use Tomcat 8.5 (for example, because you are using Java 1.6) you will need to change your classpath to reference a different version.

### 73.16.1 Use Tomcat 7.x or 8.0 with Maven

If you are using the starters and parent you can change the Tomcat version property and additionally import tomcat-juli. E.g. for a simple webapp or service:

<properties>

<tomcat.version>7.0.59</tomcat.version>

</properties>

<dependencies>

...

<dependency>

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

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

</dependency>

<dependency>

<groupId>org.apache.tomcat</groupId>

<artifactId>tomcat-juli</artifactId>

<version>${tomcat.version}</version>

</dependency>

...

</dependencies>

### 73.16.2 Use Tomcat 7.x or 8.0 with Gradle

With Gradle, you can change the Tomcat version by setting the tomcat.version property and then additionally include tomcat-juli:

ext['tomcat.version'] = '7.0.59'

dependencies {

compile 'org.springframework.boot:spring-boot-starter-web'

compile group:'org.apache.tomcat', name:'tomcat-juli', version:property('tomcat.version')

}

## 73.17 Use Jetty 9.2

Jetty 9.2 works with Spring Boot, but the default is to use Jetty 9.3. If you cannot use Jetty 9.3 (for example, because you are using Java 7) you will need to change your classpath to reference Jetty 9.2.

### 73.17.1 Use Jetty 9.2 with Maven

If you are using the starters and parent you can just add the Jetty starter and override the jetty.version property:

<properties>

<jetty.version>9.2.17.v20160517</jetty.version>

</properties>

<dependencies>

<dependency>

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

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

<exclusions>

<exclusion>

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

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

</exclusion>

</exclusions>

</dependency>

<dependency>

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

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

</dependency>

</dependencies>

### 73.17.2 Use Jetty 9.2 with Gradle

You can set the jetty.version property. For example, for a simple webapp or service:

ext['jetty.version'] = '9.2.17.v20160517'

dependencies {

compile ('org.springframework.boot:spring-boot-starter-web') {

exclude group: 'org.springframework.boot', module: 'spring-boot-starter-tomcat'

}

compile ('org.springframework.boot:spring-boot-starter-jetty')

}

## 73.18 Use Jetty 8

Jetty 8 works with Spring Boot, but the default is to use Jetty 9.3. If you cannot use Jetty 9.3 (for example, because you are using Java 1.6) you will need to change your classpath to reference Jetty 8. You will also need to exclude Jetty’s WebSocket-related dependencies.

### 73.18.1 Use Jetty 8 with Maven

If you are using the starters and parent you can just add the Jetty starter with the required WebSocket exclusion and change the version properties, e.g. for a simple webapp or service:

<properties>

<jetty.version>8.1.15.v20140411</jetty.version>

<jetty-jsp.version>2.2.0.v201112011158</jetty-jsp.version>

</properties>

<dependencies>

<dependency>

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

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

<exclusions>

<exclusion>

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

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

</exclusion>

</exclusions>

</dependency>

<dependency>

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

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

<exclusions>

<exclusion>

<groupId>org.eclipse.jetty.websocket</groupId>

<artifactId>\*</artifactId>

</exclusion>

</exclusions>

</dependency>

</dependencies>

### 73.18.2 Use Jetty 8 with Gradle

You can set the jetty.version property and exclude the WebSocket dependency, e.g. for a simple webapp or service:

ext['jetty.version'] = '8.1.15.v20140411'

dependencies {

compile ('org.springframework.boot:spring-boot-starter-web') {

exclude group: 'org.springframework.boot', module: 'spring-boot-starter-tomcat'

}

compile ('org.springframework.boot:spring-boot-starter-jetty') {

exclude group: 'org.eclipse.jetty.websocket'

}

}

## 73.19 Create WebSocket endpoints using @ServerEndpoint

If you want to use @ServerEndpoint in a Spring Boot application that used an embedded container, you must declare a single ServerEndpointExporter @Bean:

*@Bean*

**public** ServerEndpointExporter serverEndpointExporter() {

**return** **new** ServerEndpointExporter();

}

This bean will register any @ServerEndpoint annotated beans with the underlying WebSocket container. When deployed to a standalone servlet container this role is performed by a servlet container initializer and the ServerEndpointExporter bean is not required.

## 73.20 Enable HTTP response compression

HTTP response compression is supported by Jetty, Tomcat, and Undertow. It can be enabled via application.properties:

server.compression.enabled=true

By default, responses must be at least 2048 bytes in length for compression to be performed. This can be configured using the server.compression.min-response-size property.

By default, responses will only be compressed if their content type is one of the following:

* text/html
* text/xml
* text/plain
* text/css

This can be configured using the server.compression.mime-types property.

## 74. Spring MVC

## 74.1 Write a JSON REST service

Any Spring @RestController in a Spring Boot application should render JSON response by default as long as Jackson2 is on the classpath. For example:

*@RestController*

**public** **class** MyController {

*@RequestMapping("/thing")*

**public** MyThing thing() {

**return** **new** MyThing();

}

}

As long as MyThing can be serialized by Jackson2 (e.g. a normal POJO or Groovy object) then [localhost:8080/thing](http://localhost:8080/thing) will serve a JSON representation of it by default. Sometimes in a browser you might see XML responses because browsers tend to send accept headers that prefer XML.

## 74.2 Write an XML REST service

If you have the Jackson XML extension (jackson-dataformat-xml) on the classpath, it will be used to render XML responses and the very same example as we used for JSON would work. To use it, add the following dependency to your project:

<dependency>

<groupId>com.fasterxml.jackson.dataformat</groupId>

<artifactId>jackson-dataformat-xml</artifactId>

</dependency>

You may also want to add a dependency on Woodstox. It’s faster than the default StAX implementation provided by the JDK and also adds pretty print support and improved namespace handling:

<dependency>

<groupId>org.codehaus.woodstox</groupId>

<artifactId>woodstox-core-asl</artifactId>

</dependency>

If Jackson’s XML extension is not available, JAXB (provided by default in the JDK) will be used, with the additional requirement to have MyThing annotated as@XmlRootElement:

*@XmlRootElement*

**public** **class** MyThing {

**private** String name;

*// .. getters and setters*

}

To get the server to render XML instead of JSON you might have to send an Accept: text/xml header (or use a browser).

## 74.3 Customize the Jackson ObjectMapper

Spring MVC (client and server side) uses HttpMessageConverters to negotiate content conversion in an HTTP exchange. If Jackson is on the classpath you already get the default converter(s) provided by Jackson2ObjectMapperBuilder, an instance of which is auto-configured for you.

The ObjectMapper (or XmlMapper for Jackson XML converter) instance created by default has the following customized properties:

* MapperFeature.DEFAULT\_VIEW\_INCLUSION is disabled
* DeserializationFeature.FAIL\_ON\_UNKNOWN\_PROPERTIES is disabled

Spring Boot has also some features to make it easier to customize this behavior.

You can configure the ObjectMapper and XmlMapper instances using the environment. Jackson provides an extensive suite of simple on/off features that can be used to configure various aspects of its processing. These features are described in six enums in Jackson which map onto properties in the environment:

| **Jackson enum** | **Environment property** |
| --- | --- |
| com.fasterxml.jackson.databind.DeserializationFeature | spring.jackson.deserialization.<feature\_name>=true|false |
| com.fasterxml.jackson.core.JsonGenerator.Feature | spring.jackson.generator.<feature\_name>=true|false |
| com.fasterxml.jackson.databind.MapperFeature | spring.jackson.mapper.<feature\_name>=true|false |
| com.fasterxml.jackson.core.JsonParser.Feature | spring.jackson.parser.<feature\_name>=true|false |
| com.fasterxml.jackson.databind.SerializationFeature | spring.jackson.serialization.<feature\_name>=true|false |
| com.fasterxml.jackson.annotation.JsonInclude.Include | spring.jackson.default-property-inclusion=always|non\_null|non\_absent|non\_default|non\_empty |

For example, to enable pretty print, set spring.jackson.serialization.indent\_output=true. Note that, thanks to the use of [relaxed binding](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-external-config-relaxed-binding), the case of indent\_output doesn’t have to match the case of the corresponding enum constant which is INDENT\_OUTPUT.

This environment-based configuration is applied to the auto-configured Jackson2ObjectMapperBuilder bean, and will apply to any mappers created using the builder, including the auto-configured ObjectMapper bean.

The context’s Jackson2ObjectMapperBuilder can be customized by one or more Jackson2ObjectMapperBuilderCustomizer beans. Such customizer beans can be ordered and Boot’s own customizer has an order of 0, allowing additional customization to be applied both before and after Boot’s customization.

Any beans of type com.fasterxml.jackson.databind.Module will be automatically registered with the auto-configured Jackson2ObjectMapperBuilder and applied to any ObjectMapper instances that it creates. This provides a global mechanism for contributing custom modules when you add new features to your application.

If you want to replace the default ObjectMapper completely, either define a @Bean of that type and mark it as @Primary, or, if you prefer the builder-based approach, define a Jackson2ObjectMapperBuilder @Bean. Note that in either case this will disable all auto-configuration of the ObjectMapper.

If you provide any @Beans of type MappingJackson2HttpMessageConverter then they will replace the default value in the MVC configuration. Also, a convenience bean is provided of type HttpMessageConverters (always available if you use the default MVC configuration) which has some useful methods to access the default and user-enhanced message converters.

See also the [*Section 74.4, “Customize the @ResponseBody rendering”*](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto-customize-the-responsebody-rendering) section and the [WebMvcAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/web/WebMvcAutoConfiguration.java) source code for more details.

## 74.4 Customize the @ResponseBody rendering

Spring uses HttpMessageConverters to render @ResponseBody (or responses from @RestController). You can contribute additional converters by simply adding beans of that type in a Spring Boot context. If a bean you add is of a type that would have been included by default anyway (like MappingJackson2HttpMessageConverter for JSON conversions) then it will replace the default value. A convenience bean is provided of type HttpMessageConverters (always available if you use the default MVC configuration) which has some useful methods to access the default and user-enhanced message converters (useful, for example if you want to manually inject them into a custom RestTemplate).

As in normal MVC usage, any WebMvcConfigurerAdapter beans that you provide can also contribute converters by overriding the configureMessageConvertersmethod, but unlike with normal MVC, you can supply only additional converters that you need (because Spring Boot uses the same mechanism to contribute its defaults). Finally, if you opt-out of the Spring Boot default MVC configuration by providing your own @EnableWebMvc configuration, then you can take control completely and do everything manually using getMessageConverters from WebMvcConfigurationSupport.

See the [WebMvcAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/web/WebMvcAutoConfiguration.java) source code for more details.

## 74.5 Handling Multipart File Uploads

Spring Boot embraces the Servlet 3 javax.servlet.http.Part API to support uploading files. By default Spring Boot configures Spring MVC with a maximum file of 1MB per file and a maximum of 10MB of file data in a single request. You may override these values, as well as the location to which intermediate data is stored (e.g., to the /tmp directory) and the threshold past which data is flushed to disk by using the properties exposed in the MultipartProperties class. If you want to specify that files be unlimited, for example, set the spring.http.multipart.max-file-size property to -1.

The multipart support is helpful when you want to receive multipart encoded file data as a @RequestParam-annotated parameter of type MultipartFile in a Spring MVC controller handler method.

See the [MultipartAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/web/MultipartAutoConfiguration.java) source for more details.

## 74.6 Switch off the Spring MVC DispatcherServlet

Spring Boot wants to serve all content from the root of your application / down. If you would rather map your own servlet to that URL you can do it, but of course you may lose some of the other Boot MVC features. To add your own servlet and map it to the root resource just declare a @Bean of type Servlet and give it the special bean name dispatcherServlet (You can also create a bean of a different type with that name if you want to switch it off and not replace it).

## 74.7 Switch off the Default MVC configuration

The easiest way to take complete control over MVC configuration is to provide your own @Configuration with the @EnableWebMvc annotation. This will leave all MVC configuration in your hands.

## 74.8 Customize ViewResolvers

A ViewResolver is a core component of Spring MVC, translating view names in @Controller to actual View implementations. Note that ViewResolvers are mainly used in UI applications, rather than REST-style services (a View is not used to render a @ResponseBody). There are many implementations of ViewResolver to choose from, and Spring on its own is not opinionated about which ones you should use. Spring Boot, on the other hand, installs one or two for you depending on what it finds on the classpath and in the application context. The DispatcherServlet uses all the resolvers it finds in the application context, trying each one in turn until it gets a result, so if you are adding your own you have to be aware of the order and in which position your resolver is added.

WebMvcAutoConfiguration adds the following ViewResolvers to your context:

* An InternalResourceViewResolver with bean id ‘defaultViewResolver’. This one locates physical resources that can be rendered using the DefaultServlet(e.g. static resources and JSP pages if you are using those). It applies a prefix and a suffix to the view name and then looks for a physical resource with that path in the servlet context (defaults are both empty, but accessible for external configuration via spring.mvc.view.prefix and spring.mvc.view.suffix). It can be overridden by providing a bean of the same type.
* A BeanNameViewResolver with id ‘beanNameViewResolver’. This is a useful member of the view resolver chain and will pick up any beans with the same name as the View being resolved. It shouldn’t be necessary to override or replace it.
* A ContentNegotiatingViewResolver with id ‘viewResolver’ is only added if there **are** actually beans of type View present. This is a ‘master’ resolver, delegating to all the others and attempting to find a match to the ‘Accept’ HTTP header sent by the client. There is a useful [blog about ContentNegotiatingViewResolver](https://spring.io/blog/2013/06/03/content-negotiation-using-views)that you might like to study to learn more, and also look at the source code for detail. You can switch off the auto-configured ContentNegotiatingViewResolver by defining a bean named ‘viewResolver’.
* If you use Thymeleaf you will also have a ThymeleafViewResolver with id ‘thymeleafViewResolver’. It looks for resources by surrounding the view name with a prefix and suffix (externalized to spring.thymeleaf.prefix and spring.thymeleaf.suffix, defaults ‘classpath:/templates/’ and ‘.html’ respectively). It can be overridden by providing a bean of the same name.
* If you use FreeMarker you will also have a FreeMarkerViewResolver with id ‘freeMarkerViewResolver’. It looks for resources in a loader path (externalized tospring.freemarker.templateLoaderPath, default ‘classpath:/templates/’) by surrounding the view name with a prefix and suffix (externalized to spring.freemarker.prefix and spring.freemarker.suffix, with empty and ‘.ftl’ defaults respectively). It can be overridden by providing a bean of the same name.
* If you use Groovy templates (actually if groovy-templates is on your classpath) you will also have a GroovyMarkupViewResolver with id ‘groovyMarkupViewResolver’. It looks for resources in a loader path by surrounding the view name with a prefix and suffix (externalized to spring.groovy.template.prefix and spring.groovy.template.suffix, defaults ‘classpath:/templates/’ and ‘.tpl’ respectively). It can be overridden by providing a bean of the same name.

Check out [WebMvcAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/web/WebMvcAutoConfiguration.java), [ThymeleafAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/thymeleaf/ThymeleafAutoConfiguration.java), [FreeMarkerAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/freemarker/FreeMarkerAutoConfiguration.java) and [GroovyTemplateAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/groovy/template/GroovyTemplateAutoConfiguration.java)

## 74.9 Use Thymeleaf 3

By default, spring-boot-starter-thymeleaf uses Thymeleaf 2.1. If you are using the spring-boot-starter-parent, you can use Thymeleaf 3 by overriding thethymeleaf.version and thymeleaf-layout-dialect.version properties, for example:

<properties>

<thymeleaf.version>3.0.2.RELEASE</thymeleaf.version>

<thymeleaf-layout-dialect.version>2.1.1</thymeleaf-layout-dialect.version>

</properties>

|  |
| --- |
| [Note] |
| if you are managing dependencies yourself, look at spring-boot-dependencies for the list of artifacts that are related to those two versions. |

To avoid a warning message about the HTML 5 template mode being deprecated and the HTML template mode being used instead, you may also want to explicitly configure spring.thymeleaf.mode to be HTML, for example:

spring.thymeleaf.mode: HTML

Please refer to the [Thymeleaf 3 sample](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot-samples/spring-boot-sample-web-thymeleaf3) to see this in action.

If you are using any of the other auto-configured Thymeleaf Extras (Spring Security, Data Attribute, or Java 8 Time) you should also override each of their versions to one that is compatible with Thymeleaf 3.0.

## 75. HTTP clients

## 75.1 Configure RestTemplate to use a proxy

As described in [Section 33.1, “RestTemplate customization”](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-restclient-customization), a RestTemplateCustomizer can be used with RestTemplateBuilder to build a customizedRestTemplate. This is the recommended approach for creating a RestTemplate configured to use a proxy.

The exact details of the proxy configuration depend on the underlying client request factory that is being used. Here’s an example of configuringHttpComponentsClientRequestFactory with an HttpClient that uses a proxy for all hosts except 192.168.0.5.

**static** **class** ProxyCustomizer **implements** RestTemplateCustomizer {

*@Override*

**public** **void** customize(RestTemplate restTemplate) {

HttpHost proxy = **new** HttpHost("proxy.example.com");

HttpClient httpClient = HttpClientBuilder.create()

.setRoutePlanner(**new** DefaultProxyRoutePlanner(proxy) {

*@Override*

**public** HttpHost determineProxy(HttpHost target,

HttpRequest request, HttpContext context)

**throws** HttpException {

**if** (target.getHostName().equals("192.168.0.5")) {

**return** null;

}

**return** **super**.determineProxy(target, request, context);

}

}).build();

restTemplate.setRequestFactory(

**new** HttpComponentsClientHttpRequestFactory(httpClient));

}

}

## 76. Logging

Spring Boot has no mandatory logging dependency, except for the Commons Logging API, of which there are many implementations to choose from. To use [Logback](http://logback.qos.ch/) you need to include it and jcl-over-slf4j (which implements the Commons Logging API) on the classpath. The simplest way to do that is through the starters which all depend on spring-boot-starter-logging. For a web application you only need spring-boot-starter-web since it depends transitively on the logging starter. For example, using Maven:

<dependency>

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

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

</dependency>

Spring Boot has a LoggingSystem abstraction that attempts to configure logging based on the content of the classpath. If Logback is available it is the first choice.

If the only change you need to make to logging is to set the levels of various loggers then you can do that in application.properties using the "logging.level" prefix, e.g.

logging.level.org.springframework.web=DEBUG

logging.level.org.hibernate=ERROR

You can also set the location of a file to log to (in addition to the console) using "logging.file".

To configure the more fine-grained settings of a logging system you need to use the native configuration format supported by the LoggingSystem in question. By default Spring Boot picks up the native configuration from its default location for the system (e.g. classpath:logback.xml for Logback), but you can set the location of the config file using the "logging.config" property.

## 76.1 Configure Logback for logging

If you put a logback.xml in the root of your classpath it will be picked up from there (or logback-spring.xml to take advantage of the templating features provided by Boot). Spring Boot provides a default base configuration that you can include if you just want to set levels, e.g.

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

<configuration>

<include resource="org/springframework/boot/logging/logback/base.xml"/>

<logger name="org.springframework.web" level="DEBUG"/>

</configuration>

If you look at that base.xml in the spring-boot jar, you will see that it uses some useful System properties which the LoggingSystem takes care of creating for you. These are:

* ${PID} the current process ID.
* ${LOG\_FILE} if logging.file was set in Boot’s external configuration.
* ${LOG\_PATH} if logging.path was set (representing a directory for log files to live in).
* ${LOG\_EXCEPTION\_CONVERSION\_WORD} if logging.exception-conversion-word was set in Boot’s external configuration.

Spring Boot also provides some nice ANSI colour terminal output on a console (but not in a log file) using a custom Logback converter. See the default base.xmlconfiguration for details.

If Groovy is on the classpath you should be able to configure Logback with logback.groovy as well (it will be given preference if present).

### 76.1.1 Configure logback for file only output

If you want to disable console logging and write output only to a file you need a custom logback-spring.xml that imports file-appender.xml but not console-appender.xml:

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

<configuration>

<include resource="org/springframework/boot/logging/logback/defaults.xml" />

<property name="LOG\_FILE" value="${LOG\_FILE:-${LOG\_PATH:-${LOG\_TEMP:-${java.io.tmpdir:-/tmp}}/}spring.log}"/>

<include resource="org/springframework/boot/logging/logback/file-appender.xml" />

<root level="INFO">

<appender-ref ref="FILE" />

</root>

</configuration>

You also need to add logging.file to your application.properties:

logging.file=myapplication.log

## 76.2 Configure Log4j for logging

Spring Boot supports [Log4j 2](https://logging.apache.org/log4j/2.x) for logging configuration if it is on the classpath. If you are using the starters for assembling dependencies that means you have to exclude Logback and then include log4j 2 instead. If you aren’t using the starters then you need to provide jcl-over-slf4j (at least) in addition to Log4j 2.

The simplest path is probably through the starters, even though it requires some jiggling with excludes, .e.g. in Maven:

<dependency>

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

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

</dependency>

<dependency>

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

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

<exclusions>

<exclusion>

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

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

</exclusion>

</exclusions>

</dependency>

<dependency>

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

<artifactId>spring-boot-starter-log4j2</artifactId>

</dependency>

|  |
| --- |
| [Note] |
| The use of the Log4j starters gathers together the dependencies for common logging requirements (e.g. including having Tomcat use java.util.logging but configuring the output using Log4j 2). See the Actuator Log4j 2 samples for more detail and to see it in action. |

### 76.2.1 Use YAML or JSON to configure Log4j 2

In addition to its default XML configuration format, Log4j 2 also supports YAML and JSON configuration files. To configure Log4j 2 to use an alternative configuration file format, add the appropriate dependencies to the classpath and name your configuration files to match your chosen file format:

| **Format** | **Dependencies** | **File names** |
| --- | --- | --- |
| YAML | com.fasterxml.jackson.core:jackson-databindcom.fasterxml.jackson.dataformat:jackson-dataformat-yaml | log4j2.yamllog4j2.yml |
| JSON | com.fasterxml.jackson.core:jackson-databind | log4j2.jsonlog4j2.jsn |

## 77. Data Access

## 77.1 Configure a custom DataSource

To configure your own DataSource define a @Bean of that type in your configuration. Spring Boot will reuse your DataSource anywhere one is required, including database initialization. If you need to externalize some settings, you can easily bind your DataSource to the environment (see [Section 24.7.1, “Third-party configuration”](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-external-config-3rd-party-configuration)).

*@Bean*

*@ConfigurationProperties(prefix="app.datasource")*

**public** DataSource dataSource() {

**return** **new** FancyDataSource();

}

app.datasource.url=jdbc:h2:mem:mydb

app.datasource.username=sa

app.datasource.pool-size=30

Assuming that your FancyDataSource has regular JavaBean properties for the url, the username and the pool size, these settings will be bound automatically before the DataSource is made available to other components. The regular [database initialization](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto-initialize-a-database-using-spring-jdbc) will also happen (so the relevant sub-set of spring.datasource.\* can still be used with your custom configuration).

You can apply the same principle if you are configuring a custom JNDI DataSource:

*@Bean(destroyMethod="")*

*@ConfigurationProperties(prefix="app.datasource")*

**public** DataSource dataSource() **throws** Exception {

JndiDataSourceLookup dataSourceLookup = **new** JndiDataSourceLookup();

**return** dataSourceLookup.getDataSource("java:comp/env/jdbc/YourDS");

}

Spring Boot also provides a utility builder class DataSourceBuilder that can be used to create one of the standard data sources (if it is on the classpath). The builder can detect the one to use based on what’s available on the classpath. It also auto detects the driver based on the JDBC url.

*@Bean*

*@ConfigurationProperties("app.datasource")*

**public** DataSource dataSource() {

**return** DataSourceBuilder.create().build();

}

To run an app with that DataSource, all that is needed really is the connection information; pool-specific settings can also be provided, check the implementation that is going to be used at runtime for more details.

app.datasource.url=jdbc:mysql://localhost/test

app.datasource.username=dbuser

app.datasource.password=dbpass

app.datasource.pool-size=30

There is a catch however. Because the actual type of the connection pool is not exposed, no keys are generated in the metadata for your custom DataSource and no completion is available in your IDE (The DataSource interface doesn’t expose any property). Also, if you happen to only have Hikari on the classpath, this basic setup will not work because Hikari has no url parameter (but a jdbcUrl parameter). You will have to rewrite your configuration as follows:

app.datasource.jdbc-url=jdbc:mysql://localhost/test

app.datasource.username=dbuser

app.datasource.password=dbpass

app.datasource.maximum-pool-size=30

You can fix that by forcing the connection pool to use and return a dedicated implementation rather than DataSource. You won’t be able to change the implementation at runtime but the list of options will be explicit.

*@Bean*

*@ConfigurationProperties("app.datasource")*

**public** HikariDataSource dataSource() {

**return** (HikariDataSource) DataSourceBuilder.create()

.type(HikariDataSource.**class**).build();

}

You can even go further by leveraging what DataSourceProperties does for you, that is providing a default embedded database if no url is provided with a sensible username and password for it. You can easily initialize a DataSourceBuilder from the state of any DataSourceProperties so you could just as well inject the one Spring Boot creates automatically. However, that would split your configuration in two namespaces: url, username, password, type and driver on spring.datasourceand the rest on your custom namespace (app.datasource). To avoid that, you can redefine a custom DataSourceProperties on your custom namespace:

*@Bean*

*@Primary*

*@ConfigurationProperties("app.datasource")*

**public** DataSourceProperties dataSourceProperties() {

**return** **new** DataSourceProperties();

}

*@Bean*

*@ConfigurationProperties("app.datasource")*

**public** HikariDataSource dataSource(DataSourceProperties properties) {

**return** (HikariDataSource) properties.initializeDataSourceBuilder()

.type(HikariDataSource.**class**).build();

}

This setup puts you in pair with what Spring Boot does for you by default, except that a dedicated connection pool is chosen (in code) and its settings are exposed in the same namespace. Because DataSourceProperties is taking care of the url/jdbcUrl translation for you, you can configure it like this:

app.datasource.url=jdbc:mysql://localhost/test

app.datasource.username=dbuser

app.datasource.password=dbpass

app.datasource.maximum-pool-size=30

|  |
| --- |
| [Note] |
| Because your custom configuration chooses to go with Hikari, app.datasource.type will have no effect. In practice the builder will be initialized with whatever value you might set there and then overridden by the call to .type(). |

See [*Section 29.1, “Configure a DataSource”*](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-configure-datasource) in the ‘Spring Boot features’ section and the [DataSourceAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/jdbc/DataSourceAutoConfiguration.java) class for more details.

## 77.2 Configure Two DataSources

If you need to configure multiple data sources, you can apply the same tricks that are described in the previous section. You must, however, mark one of the DataSource @Primary as various auto-configurations down the road expect to be able to get one by type.

If you create your own DataSource, the auto-configuration will back off. In the example below, we provide the exact same features set than what the auto-configuration provides on the primary data source:

*@Bean*

*@Primary*

*@ConfigurationProperties("app.datasource.foo")*

**public** DataSourceProperties fooDataSourceProperties() {

**return** **new** DataSourceProperties();

}

*@Bean*

*@Primary*

*@ConfigurationProperties("app.datasource.foo")*

**public** DataSource fooDataSource() {

**return** fooDataSourceProperties().initializeDataSourceBuilder().build();

}

*@Bean*

*@ConfigurationProperties("app.datasource.bar")*

**public** BasicDataSource barDataSource() {

**return** (BasicDataSource) DataSourceBuilder.create()

.type(BasicDataSource.**class**).build();

}

|  |
| --- |
| [Tip] |
| fooDataSourceProperties has to be flagged @Primary so that the database initializer feature uses your copy (should you use that). |

Both data sources are also bound for advanced customizations. For instance you could configure them as follows:

app.datasource.foo.type=com.zaxxer.hikari.HikariDataSource

app.datasource.foo.maximum-pool-size=30

app.datasource.bar.url=jdbc:mysql://localhost/test

app.datasource.bar.username=dbuser

app.datasource.bar.password=dbpass

app.datasource.bar.max-total=30

Of course, you can apply the same concept to the secondary DataSource as well:

*@Bean*

*@Primary*

*@ConfigurationProperties("app.datasource.foo")*

**public** DataSourceProperties fooDataSourceProperties() {

**return** **new** DataSourceProperties();

}

*@Bean*

*@Primary*

*@ConfigurationProperties("app.datasource.foo")*

**public** DataSource fooDataSource() {

**return** fooDataSourceProperties().initializeDataSourceBuilder().build();

}

*@Bean*

*@ConfigurationProperties("app.datasource.bar")*

**public** DataSourceProperties barDataSourceProperties() {

**return** **new** DataSourceProperties();

}

*@Bean*

*@ConfigurationProperties("app.datasource.bar")*

**public** DataSource barDataSource() {

**return** barDataSourceProperties().initializeDataSourceBuilder().build();

}

This final example configures two data sources on custom namespaces with the same logic than what Spring Boot would do in auto-configuration.

## 77.3 Use Spring Data repositories

Spring Data can create implementations for you of @Repository interfaces of various flavors. Spring Boot will handle all of that for you as long as those @Repositories are included in the same package (or a sub-package) of your @EnableAutoConfiguration class.

For many applications all you will need is to put the right Spring Data dependencies on your classpath (there is a spring-boot-starter-data-jpa for JPA and aspring-boot-starter-data-mongodb for Mongodb), create some repository interfaces to handle your @Entity objects. Examples are in the [JPA sample](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot-samples/spring-boot-sample-data-jpa) or the [Mongodb sample](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot-samples/spring-boot-sample-data-mongodb).

Spring Boot tries to guess the location of your @Repository definitions, based on the @EnableAutoConfiguration it finds. To get more control, use the @EnableJpaRepositories annotation (from Spring Data JPA).

## 77.4 Separate @Entity definitions from Spring configuration

Spring Boot tries to guess the location of your @Entity definitions, based on the @EnableAutoConfiguration it finds. To get more control, you can use the @EntityScan annotation, e.g.

*@Configuration*

*@EnableAutoConfiguration*

*@EntityScan(basePackageClasses=City.class)*

**public** **class** Application {

*//...*

}

## 77.5 Configure JPA properties

Spring Data JPA already provides some vendor-independent configuration options (e.g. for SQL logging) and Spring Boot exposes those, and a few more for hibernate as external configuration properties. Some of them are automatically detected according to the context so you shouldn’t have to set them.

The spring.jpa.hibernate.ddl-auto is a special case in that it has different defaults depending on whether you are using an embedded database (create-drop) or not (none). The dialect to use is also automatically detected based on the current DataSource but you can set spring.jpa.database yourself if you want to be explicit and bypass that check on startup.

|  |
| --- |
| [Note] |
| Specifying a database leads to the configuration of a well-defined Hibernate dialect. Several databases have more than one Dialect and this may not suit your need. In that case, you can either set spring.jpa.database to default to let Hibernate figure things out or set the dialect using the spring.jpa.database-platform property. |

The most common options to set are:

spring.jpa.hibernate.naming.physical-strategy=com.example.MyPhysicalNamingStrategy

spring.jpa.show-sql=true

In addition all properties in spring.jpa.properties.\* are passed through as normal JPA properties (with the prefix stripped) when the local EntityManagerFactoryis created.

## 77.6 Configure Hibernate Naming Strategy

Spring Boot provides a consistent naming strategy regardless of the Hibernate generation that you are using. If you are using Hibernate 4, you can customize it usingspring.jpa.hibernate.naming.strategy; Hibernate 5 defines a Physical and Implicit naming strategies.

Spring Boot configures SpringPhysicalNamingStrategy by default. This implementation provides the same table structure as Hibernate 4: all dots are replaced by underscores and camel cases are replaced by underscores as well. By default, all table names are generated in lower case but it is possible to override that flag if your schema requires it.

Concretely, a TelephoneNumber entity will be mapped to the telephone\_number table.

If you’d rather use Hibernate 5’s default instead, set the following property:

spring.jpa.hibernate.naming.physical-strategy=org.hibernate.boot.model.naming.PhysicalNamingStrategyStandardImpl

See [HibernateJpaAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/orm/jpa/HibernateJpaAutoConfiguration.java) and [JpaBaseConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/orm/jpa/JpaBaseConfiguration.java) for more details.

## 77.7 Use a custom EntityManagerFactory

To take full control of the configuration of the EntityManagerFactory, you need to add a @Bean named ‘entityManagerFactory’. Spring Boot auto-configuration switches off its entity manager based on the presence of a bean of that type.

## 77.8 Use Two EntityManagers

Even if the default EntityManagerFactory works fine, you will need to define a new one because otherwise the presence of the second bean of that type will switch off the default. To make it easy to do that you can use the convenient EntityManagerBuilder provided by Spring Boot, or if you prefer you can just use theLocalContainerEntityManagerFactoryBean directly from Spring ORM.

Example:

*// add two data sources configured as above*

*@Bean*

**public** LocalContainerEntityManagerFactoryBean customerEntityManagerFactory(

EntityManagerFactoryBuilder builder) {

**return** builder

.dataSource(customerDataSource())

.packages(Customer.**class**)

.persistenceUnit("customers")

.build();

}

*@Bean*

**public** LocalContainerEntityManagerFactoryBean orderEntityManagerFactory(

EntityManagerFactoryBuilder builder) {

**return** builder

.dataSource(orderDataSource())

.packages(Order.**class**)

.persistenceUnit("orders")

.build();

}

The configuration above almost works on its own. To complete the picture you need to configure TransactionManagers for the two EntityManagers as well. One of them could be picked up by the default JpaTransactionManager in Spring Boot if you mark it as @Primary. The other would have to be explicitly injected into a new instance. Or you might be able to use a JTA transaction manager spanning both.

If you are using Spring Data, you need to configure @EnableJpaRepositories accordingly:

*@Configuration*

*@EnableJpaRepositories(basePackageClasses = Customer.class,*

*entityManagerFactoryRef = "customerEntityManagerFactory")*

**public** **class** CustomerConfiguration {

...

}

*@Configuration*

*@EnableJpaRepositories(basePackageClasses = Order.class,*

*entityManagerFactoryRef = "orderEntityManagerFactory")*

**public** **class** OrderConfiguration {

...

}

## 77.9 Use a traditional persistence.xml

Spring doesn’t require the use of XML to configure the JPA provider, and Spring Boot assumes you want to take advantage of that feature. If you prefer to use persistence.xml then you need to define your own @Bean of type LocalEntityManagerFactoryBean (with id ‘entityManagerFactory’, and set the persistence unit name there.

See [JpaBaseConfiguration](https://github.com/spring-projects/spring-boot/blob/master/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/orm/jpa/JpaBaseConfiguration.java) for the default settings.

## 77.10 Use Spring Data JPA and Mongo repositories

Spring Data JPA and Spring Data Mongo can both create Repository implementations for you automatically. If they are both present on the classpath, you might have to do some extra configuration to tell Spring Boot which one (or both) you want to create repositories for you. The most explicit way to do that is to use the standard Spring Data @Enable\*Repositories and tell it the location of your Repository interfaces (where ‘\*’ is ‘Jpa’ or ‘Mongo’ or both).

There are also flags spring.data.\*.repositories.enabled that you can use to switch the auto-configured repositories on and off in external configuration. This is useful for instance in case you want to switch off the Mongo repositories and still use the auto-configured MongoTemplate.

The same obstacle and the same features exist for other auto-configured Spring Data repository types (Elasticsearch, Solr). Just change the names of the annotations and flags respectively.

## 77.11 Expose Spring Data repositories as REST endpoint

Spring Data REST can expose the Repository implementations as REST endpoints for you as long as Spring MVC has been enabled for the application.

Spring Boot exposes as set of useful properties from the spring.data.rest namespace that customize the [RepositoryRestConfiguration](http://docs.spring.io/spring-data/rest/docs/current/api/org/springframework/data/rest/core/config/RepositoryRestConfiguration.html). If you need to provide additional customization, you should use a [RepositoryRestConfigurer](http://docs.spring.io/spring-data/rest/docs/current/api/org/springframework/data/rest/webmvc/config/RepositoryRestConfigurer.html) bean.

|  |
| --- |
| [Note] |
| If you don’t specify any order on your custom RepositoryRestConfigurer it will run after the one Spring Boot uses internally. If you need to specify an order, make sure it is higher than 0. |

## 77.12 Configure a component that is used by JPA

If you want to configure a component that will be used by JPA then you need to ensure that the component is initialized before JPA. Where the component is auto-configured Spring Boot will take care of this for you. For example, when Flyway is auto-configured, Hibernate is configured to depend upon Flyway so that the latter has a chance to initialize the database before Hibernate tries to use it.

If you are configuring a component yourself, you can use an EntityManagerFactoryDependsOnPostProcessor subclass as a convenient way of setting up the necessary dependencies. For example, if you are using Hibernate Search with Elasticsearch as its index manager then any EntityManagerFactory beans must be configured to depend on the elasticsearchClient bean:

**/\*\***

**\* {@link EntityManagerFactoryDependsOnPostProcessor} that ensures that**

**\* {@link EntityManagerFactory} beans depend on the {@code elasticsearchClient} bean.**

**\*/**

*@Configuration*

**static** **class** ElasticsearchJpaDependencyConfiguration

**extends** EntityManagerFactoryDependsOnPostProcessor {

ElasticsearchJpaDependencyConfiguration() {

**super**("elasticsearchClient");

}

}

## 78. Database initialization

An SQL database can be initialized in different ways depending on what your stack is. Or of course you can do it manually as long as the database is a separate process.

## 78.1 Initialize a database using JPA

JPA has features for DDL generation, and these can be set up to run on startup against the database. This is controlled through two external properties:

* spring.jpa.generate-ddl (boolean) switches the feature on and off and is vendor independent.
* spring.jpa.hibernate.ddl-auto (enum) is a Hibernate feature that controls the behavior in a more fine-grained way. See below for more detail.

## 78.2 Initialize a database using Hibernate

You can set spring.jpa.hibernate.ddl-auto explicitly and the standard Hibernate property values are none, validate, update, create, create-drop. Spring Boot chooses a default value for you based on whether it thinks your database is embedded (default create-drop) or not (default none). An embedded database is detected by looking at the Connection type: hsqldb, h2 and derby are embedded, the rest are not. Be careful when switching from in-memory to a ‘real’ database that you don’t make assumptions about the existence of the tables and data in the new platform. You either have to set ddl-auto explicitly, or use one of the other mechanisms to initialize the database.

|  |
| --- |
| [Note] |
| You can output the schema creation by enabling the org.hibernate.SQL logger. This is done for you automatically if you enable the [debug mode](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-logging-console-output). |

In addition, a file named import.sql in the root of the classpath will be executed on startup if Hibernate creates the schema from scratch (that is if the ddl-autoproperty is set to create or create-drop). This can be useful for demos and for testing if you are careful, but probably not something you want to be on the classpath in production. It is a Hibernate feature (nothing to do with Spring).

## 78.3 Initialize a database using Spring JDBC

Spring JDBC has a DataSource initializer feature. Spring Boot enables it by default and loads SQL from the standard locations schema.sql and data.sql (in the root of the classpath). In addition Spring Boot will load the schema-${platform}.sql and data-${platform}.sql files (if present), where platform is the value of spring.datasource.platform, e.g. you might choose to set it to the vendor name of the database (hsqldb, h2, oracle, mysql, postgresql etc.). Spring Boot enables the fail-fast feature of the Spring JDBC initializer by default, so if the scripts cause exceptions the application will fail to start. The script locations can be changed by setting spring.datasource.schema and spring.datasource.data, and neither location will be processed if spring.datasource.initialize=false.

To disable the fail-fast you can set spring.datasource.continue-on-error=true. This can be useful once an application has matured and been deployed a few times, since the scripts can act as ‘poor man’s migrations’ — inserts that fail mean that the data is already there, so there would be no need to prevent the application from running, for instance.

If you want to use the schema.sql initialization in a JPA app (with Hibernate) then ddl-auto=create-drop will lead to errors if Hibernate tries to create the same tables. To avoid those errors set ddl-auto explicitly to "" (preferable) or "none". Whether or not you use ddl-auto=create-drop you can always use data.sql to initialize new data.

## 78.4 Initialize a Spring Batch database

If you are using Spring Batch then it comes pre-packaged with SQL initialization scripts for most popular database platforms. Spring Boot will detect your database type, and execute those scripts by default, and in this case will switch the fail fast setting to false (errors are logged but do not prevent the application from starting). This is because the scripts are known to be reliable and generally do not contain bugs, so errors are ignorable, and ignoring them makes the scripts idempotent. You can switch off the initialization explicitly using spring.batch.initializer.enabled=false.

## 78.5 Use a higher-level database migration tool

Spring Boot supports two higher-level migration tools: [Flyway](http://flywaydb.org/) and [Liquibase](http://www.liquibase.org/).

### 78.5.1 Execute Flyway database migrations on startup

To automatically run Flyway database migrations on startup, add the org.flywaydb:flyway-core to your classpath.

The migrations are scripts in the form V<VERSION>\_\_<NAME>.sql (with <VERSION> an underscore-separated version, e.g. ‘1’ or ‘2\_1’). By default they live in a folderclasspath:db/migration but you can modify that using flyway.locations. You can also add a special {vendor} placeholder to use vendor-specific scripts. Assume the following:

flyway.locations=db/migration/{vendor}

Rather than using db/migration, this configuration will set the folder to use according to the type of the database (i.e. db/migration/mysql for MySQL). The list of supported database are available in [DatabaseDriver](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot/src/main/java/org/springframework/boot/jdbc/DatabaseDriver.java).

See also the Flyway class from flyway-core for details of available settings like schemas etc. In addition Spring Boot provides a small set of properties in[FlywayProperties](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/flyway/FlywayProperties.java) that can be used to disable the migrations, or switch off the location checking. Spring Boot will call Flyway.migrate() to perform the database migration. If you would like more control, provide a @Bean that implements [FlywayMigrationStrategy](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/flyway/FlywayMigrationStrategy.java).

|  |
| --- |
| [Tip] |
| If you want to make use of [Flyway callbacks](http://flywaydb.org/documentation/callbacks.html), those scripts should also live in the classpath:db/migration folder. |

By default Flyway will autowire the (@Primary) DataSource in your context and use that for migrations. If you like to use a different DataSource you can create one and mark its @Bean as @FlywayDataSource - if you do that remember to create another one and mark it as @Primary if you want two data sources. Or you can use Flyway’s native DataSource by setting flyway.[url,user,password] in external properties.

There is a [Flyway sample](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot-samples/spring-boot-sample-flyway) so you can see how to set things up.

You can also use Flyway to provide data for specific scenarios. For example, you can place test-specific migrations in src/test/resources and they will only be run when your application starts for testing. If you want to be more sophisticated you can use profile-specific configuration to customize flyway.locations so that certain migrations will only run when a particular profile is active. For example, in application-dev.properties you could set flyway.locations toclasspath:/db/migration, classpath:/dev/db/migration and migrations in dev/db/migration will only run when the dev profile is active.

### 78.5.2 Execute Liquibase database migrations on startup

To automatically run Liquibase database migrations on startup, add the org.liquibase:liquibase-core to your classpath.

The master change log is by default read from db/changelog/db.changelog-master.yaml but can be set using liquibase.change-log. In addition to YAML, Liquibase also supports JSON, XML, and SQL change log formats.

By default Liquibase will autowire the (@Primary) DataSource in your context and use that for migrations. If you like to use a different DataSource you can create one and mark its @Bean as @LiquibaseDataSource - if you do that remember to create another one and mark it as @Primary if you want two data sources. Or you can use Liquibase’s native DataSource by setting liquibase.[url,user,password] in external properties.

See [LiquibaseProperties](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/liquibase/LiquibaseProperties.java) for details of available settings like contexts, default schema etc.

There is a [Liquibase sample](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot-samples/spring-boot-sample-liquibase) so you can see how to set things up.

## 79. Messaging

## 79.1 Disable transacted JMS session

If your JMS broker does not support transacted session, you will have to disable the support of transactions altogether. If you create your own JmsListenerContainerFactory there is nothing to do since it won’t be transacted by default. If you want to use the DefaultJmsListenerContainerFactoryConfigurer to reuse Spring Boot’s default, you can disable transacted session as follows:

*@Bean*

**public** DefaultJmsListenerContainerFactory jmsListenerContainerFactory(

ConnectionFactory connectionFactory,

DefaultJmsListenerContainerFactoryConfigurer configurer) {

DefaultJmsListenerContainerFactory listenerFactory =

**new** DefaultJmsListenerContainerFactory();

configurer.configure(listenerFactory, connectionFactory);

listenerFactory.setTransactionManager(null);

listenerFactory.setSessionTransacted(false);

**return** listenerFactory;

}

This overrides the default factory and this should be applied to any other factory that your application defines, if any.

## 80. Batch applications

|  |
| --- |
| [Note] |
| By default, batch applications require a DataSource to store job details. If you want to deviate from that, you’ll need to implement BatchConfigurer, see[The Javadoc of @EnableBatchProcessing](http://docs.spring.io/spring-batch/apidocs/org/springframework/batch/core/configuration/annotation/EnableBatchProcessing.html) for more details. |

## 80.1 Execute Spring Batch jobs on startup

Spring Batch auto-configuration is enabled by adding @EnableBatchProcessing (from Spring Batch) somewhere in your context.

By default it executes **all** Jobs in the application context on startup (see [JobLauncherCommandLineRunner](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/batch/JobLauncherCommandLineRunner.java) for details). You can narrow down to a specific job or jobs by specifying spring.batch.job.names (comma-separated job name patterns).

If the application context includes a JobRegistry then the jobs in spring.batch.job.names are looked up in the registry instead of being autowired from the context. This is a common pattern with more complex systems where multiple jobs are defined in child contexts and registered centrally.

See [BatchAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/batch/BatchAutoConfiguration.java) and [@EnableBatchProcessing](https://github.com/spring-projects/spring-batch/blob/master/spring-batch-core/src/main/java/org/springframework/batch/core/configuration/annotation/EnableBatchProcessing.java) for more details.

## 81. Actuator

## 81.1 Change the HTTP port or address of the actuator endpoints

In a standalone application the Actuator HTTP port defaults to the same as the main HTTP port. To make the application listen on a different port set the external propertymanagement.port. To listen on a completely different network address (e.g. if you have an internal network for management and an external one for user applications) you can also set management.address to a valid IP address that the server is able to bind to.

For more detail look at the [ManagementServerProperties](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot-actuator/src/main/java/org/springframework/boot/actuate/autoconfigure/ManagementServerProperties.java) source code and [*Section 48.3, “Customizing the management server port”*](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#production-ready-customizing-management-server-port) in the ‘Production-ready features’ section.

## 81.2 Customize the ‘whitelabel’ error page

Spring Boot installs a ‘whitelabel’ error page that you will see in browser client if you encounter a server error (machine clients consuming JSON and other media types should see a sensible response with the right error code).

|  |
| --- |
| [Note] |
| Set server.error.whitelabel.enabled=false to switch the default error page off which will restore the default of the servlet container that you are using. Note that Spring Boot will still attempt to resolve the error view so you’d probably add you own error page rather than disabling it completely. |

Overriding the error page with your own depends on the templating technology that you are using. For example, if you are using Thymeleaf you would add an error.html template and if you are using FreeMarker you would add an error.ftl template. In general what you need is a View that resolves with a name of error, and/or a @Controller that handles the /error path. Unless you replaced some of the default configuration you should find a BeanNameViewResolver in your ApplicationContext so a @Bean with id error would be a simple way of doing that. Look at [ErrorMvcAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/web/ErrorMvcAutoConfiguration.java) for more options.

See also the section on [Error Handling](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#boot-features-error-handling) for details of how to register handlers in the servlet container.

## 81.3 Actuator and Jersey

Actuator HTTP endpoints are only available for Spring MVC-based applications. If you want to use Jersey and still use the actuator you will need to enable Spring MVC (by depending on spring-boot-starter-web, for example). By default, both Jersey and the Spring MVC dispatcher servlet are mapped to the same path (/). You will need to change the path for one of them (by configuring server.servlet-path for Spring MVC or spring.jersey.application-path for Jersey). For example, if you add server.servlet-path=/system into application.properties, the actuator HTTP endpoints will be available under /system.

## 82. Security

## 82.1 Switch off the Spring Boot security configuration

If you define a @Configuration with @EnableWebSecurity anywhere in your application it will switch off the default webapp security settings in Spring Boot (but leave the Actuator’s security enabled). To tweak the defaults try setting properties in security.\* (see [SecurityProperties](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/security/SecurityProperties.java) for details of available settings) and SECURITYsection of [Common application properties](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#common-application-properties-security).

## 82.2 Change the AuthenticationManager and add user accounts

If you provide a @Bean of type AuthenticationManager the default one will not be created, so you have the full feature set of Spring Security available (e.g. [various authentication options](http://docs.spring.io/spring-security/site/docs/current/reference/htmlsingle/#jc-authentication)).

Spring Security also provides a convenient AuthenticationManagerBuilder which can be used to build an AuthenticationManager with common options. The recommended way to use this in a webapp is to inject it into a void method in a WebSecurityConfigurerAdapter, e.g.

*@Configuration*

**public** **class** SecurityConfiguration **extends** WebSecurityConfigurerAdapter {

*@Autowired*

**public** **void** configureGlobal(AuthenticationManagerBuilder auth) **throws** Exception {

auth.inMemoryAuthentication()

.withUser("barry").password("password").roles("USER"); *// ... etc.*

}

*// ... other stuff for application security*

}

You will get the best results if you put this in a nested class, or a standalone class (i.e. not mixed in with a lot of other @Beans that might be allowed to influence the order of instantiation). The [secure web sample](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot-samples/spring-boot-sample-web-secure) is a useful template to follow.

If you experience instantiation issues (e.g. using JDBC or JPA for the user detail store) it might be worth extracting the AuthenticationManagerBuilder callback into aGlobalAuthenticationConfigurerAdapter (in the init() method so it happens before the authentication manager is needed elsewhere), e.g.

*@Configuration*

**public** **class** AuthenticationManagerConfiguration **extends**

GlobalAuthenticationConfigurerAdapter {

*@Override*

**public** **void** init(AuthenticationManagerBuilder auth) {

auth.inMemoryAuthentication() *// ... etc.*

}

}

## 82.3 Enable HTTPS when running behind a proxy server

Ensuring that all your main endpoints are only available over HTTPS is an important chore for any application. If you are using Tomcat as a servlet container, then Spring Boot will add Tomcat’s own RemoteIpValve automatically if it detects some environment settings, and you should be able to rely on the HttpServletRequest to report whether it is secure or not (even downstream of a proxy server that handles the real SSL termination). The standard behavior is determined by the presence or absence of certain request headers (x-forwarded-for and x-forwarded-proto), whose names are conventional, so it should work with most front end proxies. You can switch on the valve by adding some entries to application.properties, e.g.

server.tomcat.remote-ip-header=x-forwarded-for

server.tomcat.protocol-header=x-forwarded-proto

(The presence of either of those properties will switch on the valve. Or you can add the RemoteIpValve yourself by adding a TomcatEmbeddedServletContainerFactory bean.)

Spring Security can also be configured to require a secure channel for all (or some requests). To switch that on in a Spring Boot application you just need to setsecurity.require\_ssl to true in application.properties.

## 83. Hot swapping

## 83.1 Reload static content

There are several options for hot reloading. The recommended approach is to use [spring-boot-devtools](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#using-boot-devtools) as it provides additional development-time features such as support for fast application restarts and LiveReload as well as sensible development-time configuration (e.g. template caching). Devtools works by monitoring the classpath for changes. This means that static resource changes must be "built" for the change to take affect. By default, this happens automatically in Eclipse when you save your changes. In IntelliJ IDEA, Make Project will trigger the necessary build. Due to the [default restart exclusions](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#using-boot-devtools-restart-exclude), changes to static resources will not trigger a restart of your application. They will, however, trigger a live reload.

Alternatively, running in an IDE (especially with debugging on) is a good way to do development (all modern IDEs allow reloading of static resources and usually also hot-swapping of Java class changes).

Finally, the [Maven and Gradle plugins](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#build-tool-plugins) can be configured (see the addResources property) to support running from the command line with reloading of static files directly from source. You can use that with an external css/js compiler process if you are writing that code with higher level tools.

## 83.2 Reload templates without restarting the container

Most of the templating technologies supported by Spring Boot include a configuration option to disable caching (see below for details). If you’re using thespring-boot-devtools module these properties will be [automatically configured](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#using-boot-devtools-property-defaults) for you at development time.

### 83.2.1 Thymeleaf templates

If you are using Thymeleaf, then set spring.thymeleaf.cache to false. See [ThymeleafAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/thymeleaf/ThymeleafAutoConfiguration.java) for other Thymeleaf customization options.

### 83.2.2 FreeMarker templates

If you are using FreeMarker, then set spring.freemarker.cache to false. See [FreeMarkerAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/freemarker/FreeMarkerAutoConfiguration.java) for other FreeMarker customization options.

### 83.2.3 Groovy templates

If you are using Groovy templates, then set spring.groovy.template.cache to false. See [GroovyTemplateAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/groovy/template/GroovyTemplateAutoConfiguration.java) for other Groovy customization options.

## 83.3 Fast application restarts

The spring-boot-devtools module includes support for automatic application restarts. Whilst not as fast as technologies such as [JRebel](http://zeroturnaround.com/software/jrebel/) or [Spring Loaded](https://github.com/spring-projects/spring-loaded) it’s usually significantly faster than a “cold start”. You should probably give it a try before investigating some of the more complex reload options discussed below.

For more details see the [Chapter 20, *Developer tools*](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#using-boot-devtools) section.

## 83.4 Reload Java classes without restarting the container

Modern IDEs (Eclipse, IDEA, etc.) all support hot swapping of bytecode, so if you make a change that doesn’t affect class or method signatures it should reload cleanly with no side effects.

[Spring Loaded](https://github.com/spring-projects/spring-loaded) goes a little further in that it can reload class definitions with changes in the method signatures. With some customization it can force an ApplicationContext to refresh itself (but there is no general mechanism to ensure that would be safe for a running application anyway, so it would only ever be a development time trick probably).

### 83.4.1 Configuring Spring Loaded for use with Maven

To use Spring Loaded with the Maven command line, just add it as a dependency in the Spring Boot plugin declaration, e.g.

<plugin>

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

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

<dependencies>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>springloaded</artifactId>

<version>1.2.6.RELEASE</version>

</dependency>

</dependencies>

</plugin>

This normally works pretty well with Eclipse and IntelliJ IDEA as long as they have their build configuration aligned with the Maven defaults (Eclipse m2e does this out of the box).

### 83.4.2 Configuring Spring Loaded for use with Gradle and IntelliJ IDEA

You need to jump through a few hoops if you want to use Spring Loaded in combination with Gradle and IntelliJ IDEA. By default, IntelliJ IDEA will compile classes into a different location than Gradle, causing Spring Loaded monitoring to fail.

To configure IntelliJ IDEA correctly you can use the idea Gradle plugin:

buildscript {

repositories { jcenter() }

dependencies {

classpath "org.springframework.boot:spring-boot-gradle-plugin:1.5.7.RELEASE"

classpath 'org.springframework:springloaded:1.2.6.RELEASE'

}

}

apply plugin: 'idea'

idea {

module {

inheritOutputDirs = false

outputDir = file("$buildDir/classes/main/")

}

}

*// ...*

|  |
| --- |
| [Note] |
| IntelliJ IDEA must be configured to use the same Java version as the command line Gradle task and springloaded **must** be included as a buildscriptdependency. |

You can also additionally enable ‘Make Project Automatically’ inside IntelliJ IDEA to automatically compile your code whenever a file is saved.

## 84. Build

## 84.1 Generate build information

Both the Maven and Gradle plugin allow to generate build information containing the coordinates, name and version of the project. The plugin can also be configured to add additional properties through configuration. When such file is present, Spring Boot auto-configures a BuildProperties bean.

To generate build information with Maven, add an execution for the build-info goal:

<build>

<plugins>

<plugin>

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

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

<version>1.5.7.RELEASE</version>

<executions>

<execution>

<goals>

<goal>build-info</goal>

</goals>

</execution>

</executions>

</plugin>

</plugins>

</build>

|  |
| --- |
| [Tip] |
| Check the [Spring Boot Maven Plugin documentation](http://docs.spring.io/spring-boot/docs/1.5.7.RELEASE/maven-plugin/) for more details. |

And to do the same with Gradle:

springBoot {

buildInfo()

}

Additional properties can be added using the DSL:

springBoot {

buildInfo {

additionalProperties = [

'foo': 'bar'

]

}

}

## 84.2 Generate git information

Both Maven and Gradle allow to generate a git.properties file containing information about the state of your git source code repository when the project was built.

For Maven users the spring-boot-starter-parent POM includes a pre-configured plugin to generate a git.properties file. Simply add the following declaration to your POM:

<build>

<plugins>

<plugin>

<groupId>pl.project13.maven</groupId>

<artifactId>git-commit-id-plugin</artifactId>

</plugin>

</plugins>

</build>

Gradle users can achieve the same result using the [gradle-git-properties](https://plugins.gradle.org/plugin/com.gorylenko.gradle-git-properties) plugin

plugins {

id "com.gorylenko.gradle-git-properties" version "1.4.17"

}

## 84.3 Customize dependency versions

If you use a Maven build that inherits directly or indirectly from spring-boot-dependencies (for instance spring-boot-starter-parent) but you want to override a specific third-party dependency you can add appropriate <properties> elements. Browse the [spring-boot-dependencies](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot-dependencies/pom.xml) POM for a complete list of properties. For example, to pick a different slf4j version you would add the following:

<properties>

<slf4j.version>1.7.5<slf4j.version>

</properties>

|  |
| --- |
| [Note] |
| This only works if your Maven project inherits (directly or indirectly) from spring-boot-dependencies. If you have added spring-boot-dependenciesin your own dependencyManagement section with <scope>import</scope> you have to redefine the artifact yourself instead of overriding the property. |
| [Warning] |
| Each Spring Boot release is designed and tested against a specific set of third-party dependencies. Overriding versions may cause compatibility issues. |

To override dependency versions in Gradle, you can specify a version as shown below:

ext['slf4j.version'] = '1.7.5'

For additional information, please refer to the [Gradle Dependency Management Plugin documentation](https://github.com/spring-gradle-plugins/dependency-management-plugin).

## 84.4 Create an executable JAR with Maven

The spring-boot-maven-plugin can be used to create an executable ‘fat’ JAR. If you are using the spring-boot-starter-parent POM you can simply declare the plugin and your jars will be repackaged:

<build>

<plugins>

<plugin>

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

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

</plugin>

</plugins>

</build>

If you are not using the parent POM you can still use the plugin, however, you must additionally add an <executions> section:

<build>

<plugins>

<plugin>

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

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

<version>1.5.7.RELEASE</version>

<executions>

<execution>

<goals>

<goal>repackage</goal>

</goals>

</execution>

</executions>

</plugin>

</plugins>

</build>

See the [plugin documentation](http://docs.spring.io/spring-boot/docs/1.5.7.RELEASE/maven-plugin/usage.html) for full usage details.

## 84.5 Use a Spring Boot application as a dependency

Like a war file, a Spring Boot application is not intended to be used as a dependency. If your application contains classes that you want to share with other projects, the recommended approach is to move that code into a separate module. The separate module can then be depended upon by your application and other projects.

If you cannot rearrange your code as recommended above, Spring Boot’s Maven and Gradle plugins must be configured to produce a separate artifact that is suitable for use as a dependency. The executable archive cannot be used as a dependency as the [executable jar format](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#executable-jar-jar-file-structure) packages application classes in BOOT-INF/classes. This means that they cannot be found when the executable jar is used as a dependency.

To produce the two artifacts, one that can be used as a dependency and one that is executable, a classifier must be specified. This classifier is applied to the name of the executable archive, leaving the default archive for use as dependency.

To configure a classifier of exec in Maven, the following configuration can be used:

<build>

<plugins>

<plugin>

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

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

<configuration>

<classifier>exec</classifier>

</configuration>

</plugin>

</plugins>

</build>

And when using Gradle, the following configuration can be used:

bootRepackage {

classifier = 'exec'

}

## 84.6 Extract specific libraries when an executable jar runs

Most nested libraries in an executable jar do not need to be unpacked in order to run, however, certain libraries can have problems. For example, JRuby includes its own nested jar support which assumes that the jruby-complete.jar is always directly available as a file in its own right.

To deal with any problematic libraries, you can flag that specific nested jars should be automatically unpacked to the ‘temp folder’ when the executable jar first runs.

For example, to indicate that JRuby should be flagged for unpack using the Maven Plugin you would add the following configuration:

<build>

<plugins>

<plugin>

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

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

<configuration>

<requiresUnpack>

<dependency>

<groupId>org.jruby</groupId>

<artifactId>jruby-complete</artifactId>

</dependency>

</requiresUnpack>

</configuration>

</plugin>

</plugins>

</build>

And to do that same with Gradle:

springBoot {

requiresUnpack = ['org.jruby:jruby-complete']

}

## 84.7 Create a non-executable JAR with exclusions

Often if you have an executable and a non-executable jar as build products, the executable version will have additional configuration files that are not needed in a library jar. E.g. the application.yml configuration file might excluded from the non-executable JAR.

Here’s how to do that in Maven:

<build>

<plugins>

<plugin>

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

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

<configuration>

<classifier>exec</classifier>

</configuration>

</plugin>

<plugin>

<artifactId>maven-jar-plugin</artifactId>

<executions>

<execution>

<id>exec</id>

<phase>package</phase>

<goals>

<goal>jar</goal>

</goals>

<configuration>

<classifier>exec</classifier>

</configuration>

</execution>

<execution>

<phase>package</phase>

<goals>

<goal>jar</goal>

</goals>

<configuration>

*<!-- Need this to ensure application.yml is excluded -->*

<forceCreation>true</forceCreation>

<excludes>

<exclude>application.yml</exclude>

</excludes>

</configuration>

</execution>

</executions>

</plugin>

</plugins>

</build>

In Gradle you can create a new JAR archive with standard task DSL features, and then have the bootRepackage task depend on that one using its withJarTaskproperty:

jar {

baseName = 'spring-boot-sample-profile'

version = '0.0.0'

excludes = ['\*\*/application.yml']

}

task('execJar', type:Jar, dependsOn: 'jar') {

baseName = 'spring-boot-sample-profile'

version = '0.0.0'

classifier = 'exec'

from sourceSets.main.output

}

bootRepackage {

withJarTask = tasks['execJar']

}

## 84.8 Remote debug a Spring Boot application started with Maven

To attach a remote debugger to a Spring Boot application started with Maven you can use the jvmArguments property of the [maven plugin](http://docs.spring.io/spring-boot/docs/1.5.7.RELEASE/maven-plugin/).

Check [this example](http://docs.spring.io/spring-boot/docs/1.5.7.RELEASE/maven-plugin/examples/run-debug.html) for more details.

## 84.9 Remote debug a Spring Boot application started with Gradle

To attach a remote debugger to a Spring Boot application started with Gradle you can use the jvmArgs property of bootRun task or --debug-jvm command line option.

build.gradle:

bootRun {

jvmArgs "-agentlib:jdwp=transport=dt\_socket,server=y,suspend=y,address=5005"

}

Command line:

$ gradle bootRun --debug-jvm

Check [Gradle Application Plugin](http://www.gradle.org/docs/current/userguide/application_plugin.html) for more details.

## 84.10 Build an executable archive from Ant without using spring-boot-antlib

To build with Ant you need to grab dependencies, compile and then create a jar or war archive. To make it executable you can either use the spring-boot-antlibmodule, or you can follow these instructions:

1. If you are building a jar, package the application’s classes and resources in a nested BOOT-INF/classes directory. If you are building a war, package the application’s classes in a nested WEB-INF/classes directory as usual.
2. Add the runtime dependencies in a nested BOOT-INF/lib directory for a jar or WEB-INF/lib for a war. Remember **not** to compress the entries in the archive.
3. Add the provided (embedded container) dependencies in a nested BOOT-INF/lib directory for jar or WEB-INF/lib-provided for a war. Remember **not** to compress the entries in the archive.
4. Add the spring-boot-loader classes at the root of the archive (so the Main-Class is available).
5. Use the appropriate launcher, e.g. JarLauncher for a jar file, as a Main-Class attribute in the manifest and specify the other properties it needs as manifest entries, principally a Start-Class.

Example:

<target name="build" depends="compile">

<jar destfile="target/${ant.project.name}-${spring-boot.version}.jar" compress="false">

<mappedresources>

<fileset dir="target/classes" />

<globmapper from="\*" to="BOOT-INF/classes/\*"/>

</mappedresources>

<mappedresources>

<fileset dir="src/main/resources" erroronmissingdir="false"/>

<globmapper from="\*" to="BOOT-INF/classes/\*"/>

</mappedresources>

<mappedresources>

<fileset dir="${lib.dir}/runtime" />

<globmapper from="\*" to="BOOT-INF/lib/\*"/>

</mappedresources>

<zipfileset src="${lib.dir}/loader/spring-boot-loader-jar-${spring-boot.version}.jar" />

<manifest>

<attribute name="Main-Class" value="org.springframework.boot.loader.JarLauncher" />

<attribute name="Start-Class" value="${start-class}" />

</manifest>

</jar>

</target>

The [Ant Sample](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot-samples/spring-boot-sample-ant) has a build.xml with a manual task that should work if you run it with

$ ant -lib <folder containing ivy-2.2.jar> clean manual

after which you can run the application with

$ java -jar target/\*.jar

## 84.11 How to use Java 6

If you want to use Spring Boot with Java 6 there are a small number of configuration changes that you will have to make. The exact changes depend on your application’s functionality.

### 84.11.1 Embedded servlet container compatibility

If you are using one of Boot’s embedded Servlet containers you will have to use a Java 6-compatible container. Both Tomcat 7 and Jetty 8 are Java 6 compatible. See[Section 73.16, “Use Tomcat 7.x or 8.0”](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto-use-tomcat-7) and [Section 73.18, “Use Jetty 8”](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto-use-jetty-8) for details.

### 84.11.2 Jackson

Jackson 2.7 and later requires Java 7. If you want to use Jackson with Java 6 you will have to downgrade to Jackson 2.6.

Spring Boot uses the Jackson BOM that was introduced as of Jackson 2.7 so you can’t just override the jackson.version property. In order to use Jackson 2.6, you will have to define the individual modules in the dependencyManagement section of your build, check [this example](https://github.com/spring-projects/spring-boot/blob/0ffc7dc13f6de82c199a6d503354a88c7aaec2d9/spring-boot-dependencies/pom.xml#L523-L597) for more details.

### 84.11.3 JTA API compatibility

While the Java Transaction API itself doesn’t require Java 7 the official API jar contains classes that have been built to require Java 7. If you are using JTA then you will need to replace the official JTA 1.2 API jar with one that has been built to work on Java 6. To do so, exclude any transitive dependencies onjavax.transaction:javax.transaction-api and replace them with a dependency onorg.jboss.spec.javax.transaction:jboss-transaction-api\_1.2\_spec:1.0.0.Final

## 85. Traditional deployment

## 85.1 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 using spring-boot-starter-parent (which configures Maven’s war plugin for you) all you need to do is to 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'

*// …*

}

|  |
| --- |
| [Note] |
| If you are using a version of Gradle that supports compile only dependencies (2.12 or later), you should continue to use providedRuntime. Among other limitations, compileOnly dependencies are not on the test classpath so any web-based integration tests will fail. |

If you’re using the [Spring Boot build tools](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#build-tool-plugins), 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 using java -jar on the command line.

|  |
| --- |
| [Tip] |
| Take a look at Spring Boot’s sample applications for a [Maven-based example](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot-samples/spring-boot-sample-traditional/pom.xml) of the above-described configuration. |

## 85.2 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.

## 85.3 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](https://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 @SpringBootApplicationannotation. Example:

*@SpringBootApplication*

**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 (via @SpringBootApplication)*

*// 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/> in web.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 through an @ImportResource 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);

}

|  |
| --- |
| [Note] |
| If you intend to start your application as a war or as an executable application, you need to share the customizations of the builder in a method that is both available to the SpringBootServletInitializer callback and the main method, something like:  *@SpringBootApplication*  **public** **class** Application **extends** SpringBootServletInitializer {  *@Override*  **protected** SpringApplicationBuilder configure(SpringApplicationBuilder builder) {  **return** configureApplication(builder);  }  **public** **static** **void** main(String[] args) {  configureApplication(**new** SpringApplicationBuilder()).run(args);  }  **private** **static** SpringApplicationBuilder configureApplication(SpringApplicationBuilder builder) {  **return** builder.sources(Application.**class**).bannerMode(Banner.Mode.OFF);  }  } |

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](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#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.

## 85.4 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>

## 85.5 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 include org.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 1.0.2.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>metricsFilter</filter-name>

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

</filter>

<filter-mapping>

<filter-name>metricsFilter</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).

# Part X. Appendices

## Appendix A. Common application properties

Various properties can be specified inside your application.properties/application.yml file or as command line switches. This section provides a list of common Spring Boot properties and references to the underlying classes that consume them.

|  |
| --- |
| [Note] |
| Property contributions can come from additional jar files on your classpath so you should not consider this an exhaustive list. It is also perfectly legit to define your own properties. |
| [Warning] |
| This sample file is meant as a guide only. Do **not** copy/paste the entire content into your application; rather pick only the properties that you need. |

*# ===================================================================*

*# COMMON SPRING BOOT PROPERTIES*

*#*

*# This sample file is provided as a guideline. Do NOT copy it in its*

*# entirety to your own application. ^^^*

*# ===================================================================*

*# ----------------------------------------*

*# CORE PROPERTIES*

*# ----------------------------------------*

*# BANNER*

banner.charset=UTF-8 *# Banner file encoding.*

banner.location=classpath:banner.txt *# Banner file location.*

banner.image.location=classpath:banner.gif *# Banner image file location (jpg/png can also be used).*

banner.image.width= *# Width of the banner image in chars (default 76)*

banner.image.height= *# Height of the banner image in chars (default based on image height)*

banner.image.margin= *# Left hand image margin in chars (default 2)*

banner.image.invert= *# If images should be inverted for dark terminal themes (default false)*

*# LOGGING*

logging.config= *# Location of the logging configuration file. For instance `classpath:logback.xml` for Logback*

logging.exception-conversion-word=%wEx *# Conversion word used when logging exceptions.*

logging.file= *# Log file name. For instance `myapp.log`*

logging.level.\*= *# Log levels severity mapping. For instance `logging.level.org.springframework=DEBUG`*

logging.path= *# Location of the log file. For instance `/var/log`*

logging.pattern.console= *# Appender pattern for output to the console. Only supported with the default logback setup.*

logging.pattern.file= *# Appender pattern for output to the file. Only supported with the default logback setup.*

logging.pattern.level= *# Appender pattern for log level (default %5p). Only supported with the default logback setup.*

logging.register-shutdown-hook=false *# Register a shutdown hook for the logging system when it is initialized.*

*# AOP*

spring.aop.auto=true *# Add @EnableAspectJAutoProxy.*

spring.aop.proxy-target-class=false *# Whether subclass-based (CGLIB) proxies are to be created (true) as opposed to standard Java interface-based proxies (false).*

*# IDENTITY (*[ContextIdApplicationContextInitializer](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot/src/main/java/org/springframework/boot/context/ContextIdApplicationContextInitializer.java))

spring.application.index= *# Application index.*

spring.application.name= *# Application name.*

*# ADMIN (*[SpringApplicationAdminJmxAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/admin/SpringApplicationAdminJmxAutoConfiguration.java))

spring.application.admin.enabled=false *# Enable admin features for the application.*

spring.application.admin.jmx-name=org.springframework.boot:type=Admin,name=SpringApplication *# JMX name of the application admin MBean.*

*# AUTO-CONFIGURATION*

spring.autoconfigure.exclude= *# Auto-configuration classes to exclude.*

*# SPRING CORE*

spring.beaninfo.ignore=true *# Skip search of BeanInfo classes.*

*# SPRING CACHE (*[CacheProperties](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/cache/CacheProperties.java))

spring.cache.cache-names= *# Comma-separated list of cache names to create if supported by the underlying cache manager.*

spring.cache.caffeine.spec= *# The spec to use to create caches. Check CaffeineSpec for more details on the spec format.*

spring.cache.couchbase.expiration=0 *# Entry expiration in milliseconds. By default the entries never expire.*

spring.cache.ehcache.config= *# The location of the configuration file to use to initialize EhCache.*

spring.cache.guava.spec= *# The spec to use to create caches. Check CacheBuilderSpec for more details on the spec format.*

spring.cache.infinispan.config= *# The location of the configuration file to use to initialize Infinispan.*

spring.cache.jcache.config= *# The location of the configuration file to use to initialize the cache manager.*

spring.cache.jcache.provider= *# Fully qualified name of the CachingProvider implementation to use to retrieve the JSR-107 compliant cache manager. Only needed if more than one JSR-107 implementation is available on the classpath.*

spring.cache.type= *# Cache type, auto-detected according to the environment by default.*

*# SPRING CONFIG - using environment property only (*[ConfigFileApplicationListener](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot/src/main/java/org/springframework/boot/context/config/ConfigFileApplicationListener.java))

spring.config.location= *# Config file locations.*

spring.config.name=application *# Config file name.*

*# HAZELCAST (*[HazelcastProperties](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/hazelcast/HazelcastProperties.java))

spring.hazelcast.config= *# The location of the configuration file to use to initialize Hazelcast.*

*# PROJECT INFORMATION (*[ProjectInfoProperties](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/info/ProjectInfoProperties.java))

spring.info.build.location=classpath:META-INF/build-info.properties *# Location of the generated build-info.properties file.*

spring.info.git.location=classpath:git.properties *# Location of the generated git.properties file.*

*# JMX*

spring.jmx.default-domain= *# JMX domain name.*

spring.jmx.enabled=true *# Expose management beans to the JMX domain.*

spring.jmx.server=mbeanServer *# MBeanServer bean name.*

*# Email (*[MailProperties](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/mail/MailProperties.java))

spring.mail.default-encoding=UTF-8 *# Default MimeMessage encoding.*

spring.mail.host= *# SMTP server host. For instance `smtp.example.com`*

spring.mail.jndi-name= *# Session JNDI name. When set, takes precedence to others mail settings.*

spring.mail.password= *# Login password of the SMTP server.*

spring.mail.port= *# SMTP server port.*

spring.mail.properties.\*= *# Additional JavaMail session properties.*

spring.mail.protocol=smtp *# Protocol used by the SMTP server.*

spring.mail.test-connection=false *# Test that the mail server is available on startup.*

spring.mail.username= *# Login user of the SMTP server.*

*# APPLICATION SETTINGS (*[SpringApplication](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot/src/main/java/org/springframework/boot/SpringApplication.java))

spring.main.banner-mode=console *# Mode used to display the banner when the application runs.*

spring.main.sources= *# Sources (class name, package name or XML resource location) to include in the ApplicationContext.*

spring.main.web-environment= *# Run the application in a web environment (auto-detected by default).*

*# FILE ENCODING (*[FileEncodingApplicationListener](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot/src/main/java/org/springframework/boot/context/FileEncodingApplicationListener.java))

spring.mandatory-file-encoding= *# Expected character encoding the application must use.*

*# INTERNATIONALIZATION (*[MessageSourceAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/context/MessageSourceAutoConfiguration.java))

spring.messages.always-use-message-format=false *# Set whether to always apply the MessageFormat rules, parsing even messages without arguments.*

spring.messages.basename=messages *# Comma-separated list of basenames, each following the ResourceBundle convention.*

spring.messages.cache-seconds=-1 *# Loaded resource bundle files cache expiration, in seconds. When set to -1, bundles are cached forever.*

spring.messages.encoding=UTF-8 *# Message bundles encoding.*

spring.messages.fallback-to-system-locale=true *# Set whether to fall back to the system Locale if no files for a specific Locale have been found.*

*# OUTPUT*

spring.output.ansi.enabled=detect *# Configure the ANSI output.*

*# PID FILE (*[ApplicationPidFileWriter](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot/src/main/java/org/springframework/boot/system/ApplicationPidFileWriter.java))

spring.pid.fail-on-write-error= *# Fail if ApplicationPidFileWriter is used but it cannot write the PID file.*

spring.pid.file= *# Location of the PID file to write (if ApplicationPidFileWriter is used).*

*# PROFILES*

spring.profiles.active= *# Comma-separated list (or list if using YAML) of* [active profiles](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#howto-set-active-spring-profiles).

spring.profiles.include= *# Unconditionally activate the specified comma separated profiles (or list of profiles if using YAML).*

*# SENDGRID (*[SendGridAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/sendgrid/SendGridAutoConfiguration.java))

spring.sendgrid.api-key= *# SendGrid api key (alternative to username/password).*

spring.sendgrid.username= *# SendGrid account username.*

spring.sendgrid.password= *# SendGrid account password.*

spring.sendgrid.proxy.host= *# SendGrid proxy host.*

spring.sendgrid.proxy.port= *# SendGrid proxy port.*

*# ----------------------------------------*

*# WEB PROPERTIES*

*# ----------------------------------------*

*# EMBEDDED SERVER CONFIGURATION (*[ServerProperties](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/web/ServerProperties.java))

server.address= *# Network address to which the server should bind to.*

server.compression.enabled=false *# If response compression is enabled.*

server.compression.excluded-user-agents= *# List of user-agents to exclude from compression.*

server.compression.mime-types=text/html,text/xml,text/plain,text/css,text/javascript,application/javascript *# Comma-separated list of MIME types that should be compressed.*

server.compression.min-response-size=2048 *# Minimum response size that is required for compression to be performed.*

server.connection-timeout= *# Time in milliseconds that connectors will wait for another HTTP request before closing the connection. When not set, the connector's container-specific default will be used. Use a value of -1 to indicate no (i.e. infinite) timeout.*

server.context-parameters.\*= *# Servlet context init parameters. For instance `server.context-parameters.a=alpha`*

server.context-path= *# Context path of the application.*

server.display-name=application *# Display name of the application.*

server.max-http-header-size=0 *# Maximum size in bytes of the HTTP message header.*

server.error.include-stacktrace=never *# When to include a "stacktrace" attribute.*

server.error.path=/error *# Path of the error controller.*

server.error.whitelabel.enabled=true *# Enable the default error page displayed in browsers in case of a server error.*

server.jetty.acceptors= *# Number of acceptor threads to use.*

server.jetty.max-http-post-size=0 *# Maximum size in bytes of the HTTP post or put content.*

server.jetty.selectors= *# Number of selector threads to use.*

server.jsp-servlet.class-name=org.apache.jasper.servlet.JspServlet *# The class name of the JSP servlet.*

server.jsp-servlet.init-parameters.\*= *# Init parameters used to configure the JSP servlet*

server.jsp-servlet.registered=true *# Whether or not the JSP servlet is registered*

server.port=8080 *# Server HTTP port.*

server.server-header= *# Value to use for the Server response header (no header is sent if empty)*

server.servlet-path=/ *# Path of the main dispatcher servlet.*

server.use-forward-headers= *# If X-Forwarded-\* headers should be applied to the HttpRequest.*

server.session.cookie.comment= *# Comment for the session cookie.*

server.session.cookie.domain= *# Domain for the session cookie.*

server.session.cookie.http-only= *# "HttpOnly" flag for the session cookie.*

server.session.cookie.max-age= *# Maximum age of the session cookie in seconds.*

server.session.cookie.name= *# Session cookie name.*

server.session.cookie.path= *# Path of the session cookie.*

server.session.cookie.secure= *# "Secure" flag for the session cookie.*

server.session.persistent=false *# Persist session data between restarts.*

server.session.store-dir= *# Directory used to store session data.*

server.session.timeout= *# Session timeout in seconds.*

server.session.tracking-modes= *# Session tracking modes (one or more of the following: "cookie", "url", "ssl").*

server.ssl.ciphers= *# Supported SSL ciphers.*

server.ssl.client-auth= *# Whether client authentication is wanted ("want") or needed ("need"). Requires a trust store.*

server.ssl.enabled= *# Enable SSL support.*

server.ssl.enabled-protocols= *# Enabled SSL protocols.*

server.ssl.key-alias= *# Alias that identifies the key in the key store.*

server.ssl.key-password= *# Password used to access the key in the key store.*

server.ssl.key-store= *# Path to the key store that holds the SSL certificate (typically a jks file).*

server.ssl.key-store-password= *# Password used to access the key store.*

server.ssl.key-store-provider= *# Provider for the key store.*

server.ssl.key-store-type= *# Type of the key store.*

server.ssl.protocol=TLS *# SSL protocol to use.*

server.ssl.trust-store= *# Trust store that holds SSL certificates.*

server.ssl.trust-store-password= *# Password used to access the trust store.*

server.ssl.trust-store-provider= *# Provider for the trust store.*

server.ssl.trust-store-type= *# Type of the trust store.*

server.tomcat.accept-count= *# Maximum queue length for incoming connection requests when all possible request processing threads are in use.*

server.tomcat.accesslog.buffered=true *# Buffer output such that it is only flushed periodically.*

server.tomcat.accesslog.directory=logs *# Directory in which log files are created. Can be relative to the tomcat base dir or absolute.*

server.tomcat.accesslog.enabled=false *# Enable access log.*

server.tomcat.accesslog.file-date-format=.yyyy-MM-dd *# Date format to place in log file name.*

server.tomcat.accesslog.pattern=common *# Format pattern for access logs.*

server.tomcat.accesslog.prefix=access\_log *# Log file name prefix.*

server.tomcat.accesslog.rename-on-rotate=false *# Defer inclusion of the date stamp in the file name until rotate time.*

server.tomcat.accesslog.request-attributes-enabled=false *# Set request attributes for IP address, Hostname, protocol and port used for the request.*

server.tomcat.accesslog.rotate=true *# Enable access log rotation.*

server.tomcat.accesslog.suffix=.log *# Log file name suffix.*

server.tomcat.additional-tld-skip-patterns= *# Comma-separated list of additional patterns that match jars to ignore for TLD scanning.*

server.tomcat.background-processor-delay=30 *# Delay in seconds between the invocation of backgroundProcess methods.*

server.tomcat.basedir= *# Tomcat base directory. If not specified a temporary directory will be used.*

server.tomcat.internal-proxies=10\\.\\d{1,3}\\.\\d{1,3}\\.\\d{1,3}|\\

192\\.168\\.\\d{1,3}\\.\\d{1,3}|\\

169\\.254\\.\\d{1,3}\\.\\d{1,3}|\\

127\\.\\d{1,3}\\.\\d{1,3}\\.\\d{1,3}|\\

172\\.1[6-9]{1}\\.\\d{1,3}\\.\\d{1,3}|\\

172\\.2[0-9]{1}\\.\\d{1,3}\\.\\d{1,3}|\\

172\\.3[0-1]{1}\\.\\d{1,3}\\.\\d{1,3} *# regular expression matching trusted IP addresses.*

server.tomcat.max-connections= *# Maximum number of connections that the server will accept and process at any given time.*

server.tomcat.max-http-post-size=0 *# Maximum size in bytes of the HTTP post content.*

server.tomcat.max-threads=0 *# Maximum amount of worker threads.*

server.tomcat.min-spare-threads=0 *# Minimum amount of worker threads.*

server.tomcat.port-header=X-Forwarded-Port *# Name of the HTTP header used to override the original port value.*

server.tomcat.protocol-header= *# Header that holds the incoming protocol, usually named "X-Forwarded-Proto".*

server.tomcat.protocol-header-https-value=https *# Value of the protocol header that indicates that the incoming request uses SSL.*

server.tomcat.redirect-context-root= *# Whether requests to the context root should be redirected by appending a / to the path.*

server.tomcat.remote-ip-header= *# Name of the http header from which the remote ip is extracted. For instance `X-FORWARDED-FOR`*

server.tomcat.uri-encoding=UTF-8 *# Character encoding to use to decode the URI.*

server.undertow.accesslog.dir= *# Undertow access log directory.*

server.undertow.accesslog.enabled=false *# Enable access log.*

server.undertow.accesslog.pattern=common *# Format pattern for access logs.*

server.undertow.accesslog.prefix=access\_log. *# Log file name prefix.*

server.undertow.accesslog.rotate=true *# Enable access log rotation.*

server.undertow.accesslog.suffix=log *# Log file name suffix.*

server.undertow.buffer-size= *# Size of each buffer in bytes.*

server.undertow.direct-buffers= *# Allocate buffers outside the Java heap.*

server.undertow.io-threads= *# Number of I/O threads to create for the worker.*

server.undertow.max-http-post-size=0 *# Maximum size in bytes of the HTTP post content.*

server.undertow.worker-threads= *# Number of worker threads.*

*# FREEMARKER (*[FreeMarkerAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/freemarker/FreeMarkerAutoConfiguration.java))

spring.freemarker.allow-request-override=false *# Set whether HttpServletRequest attributes are allowed to override (hide) controller generated model attributes of the same name.*

spring.freemarker.allow-session-override=false *# Set whether HttpSession attributes are allowed to override (hide) controller generated model attributes of the same name.*

spring.freemarker.cache=false *# Enable template caching.*

spring.freemarker.charset=UTF-8 *# Template encoding.*

spring.freemarker.check-template-location=true *# Check that the templates location exists.*

spring.freemarker.content-type=text/html *# Content-Type value.*

spring.freemarker.enabled=true *# Enable MVC view resolution for this technology.*

spring.freemarker.expose-request-attributes=false *# Set whether all request attributes should be added to the model prior to merging with the template.*

spring.freemarker.expose-session-attributes=false *# Set whether all HttpSession attributes should be added to the model prior to merging with the template.*

spring.freemarker.expose-spring-macro-helpers=true *# Set whether to expose a RequestContext for use by Spring's macro library, under the name "springMacroRequestContext".*

spring.freemarker.prefer-file-system-access=true *# Prefer file system access for template loading. File system access enables hot detection of template changes.*

spring.freemarker.prefix= *# Prefix that gets prepended to view names when building a URL.*

spring.freemarker.request-context-attribute= *# Name of the RequestContext attribute for all views.*

spring.freemarker.settings.\*= *# Well-known FreeMarker keys which will be passed to FreeMarker's Configuration.*

spring.freemarker.suffix= *# Suffix that gets appended to view names when building a URL.*

spring.freemarker.template-loader-path=classpath:/templates/ *# Comma-separated list of template paths.*

spring.freemarker.view-names= *# White list of view names that can be resolved.*

*# GROOVY TEMPLATES (*[GroovyTemplateAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/groovy/template/GroovyTemplateAutoConfiguration.java))

spring.groovy.template.allow-request-override=false *# Set whether HttpServletRequest attributes are allowed to override (hide) controller generated model attributes of the same name.*

spring.groovy.template.allow-session-override=false *# Set whether HttpSession attributes are allowed to override (hide) controller generated model attributes of the same name.*

spring.groovy.template.cache= *# Enable template caching.*

spring.groovy.template.charset=UTF-8 *# Template encoding.*

spring.groovy.template.check-template-location=true *# Check that the templates location exists.*

spring.groovy.template.configuration.\*= *# See GroovyMarkupConfigurer*

spring.groovy.template.content-type=test/html *# Content-Type value.*

spring.groovy.template.enabled=true *# Enable MVC view resolution for this technology.*

spring.groovy.template.expose-request-attributes=false *# Set whether all request attributes should be added to the model prior to merging with the template.*

spring.groovy.template.expose-session-attributes=false *# Set whether all HttpSession attributes should be added to the model prior to merging with the template.*

spring.groovy.template.expose-spring-macro-helpers=true *# Set whether to expose a RequestContext for use by Spring's macro library, under the name "springMacroRequestContext".*

spring.groovy.template.prefix= *# Prefix that gets prepended to view names when building a URL.*

spring.groovy.template.request-context-attribute= *# Name of the RequestContext attribute for all views.*

spring.groovy.template.resource-loader-path=classpath:/templates/ *# Template path.*

spring.groovy.template.suffix=.tpl *# Suffix that gets appended to view names when building a URL.*

spring.groovy.template.view-names= *# White list of view names that can be resolved.*

*# SPRING HATEOAS (*[HateoasProperties](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/hateoas/HateoasProperties.java))

spring.hateoas.use-hal-as-default-json-media-type=true *# Specify if application/hal+json responses should be sent to requests that accept application/json.*

*# HTTP message conversion*

spring.http.converters.preferred-json-mapper=jackson *# Preferred JSON mapper to use for HTTP message conversion. Set to "gson" to force the use of Gson when both it and Jackson are on the classpath.*

*# HTTP encoding (*[HttpEncodingProperties](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/web/HttpEncodingProperties.java))

spring.http.encoding.charset=UTF-8 *# Charset of HTTP requests and responses. Added to the "Content-Type" header if not set explicitly.*

spring.http.encoding.enabled=true *# Enable http encoding support.*

spring.http.encoding.force= *# Force the encoding to the configured charset on HTTP requests and responses.*

spring.http.encoding.force-request= *# Force the encoding to the configured charset on HTTP requests. Defaults to true when "force" has not been specified.*

spring.http.encoding.force-response= *# Force the encoding to the configured charset on HTTP responses.*

spring.http.encoding.mapping= *# Locale to Encoding mapping.*

*# MULTIPART (*[MultipartProperties](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/web/MultipartProperties.java))

spring.http.multipart.enabled=true *# Enable support of multi-part uploads.*

spring.http.multipart.file-size-threshold=0 *# Threshold after which files will be written to disk. Values can use the suffixed "MB" or "KB" to indicate a Megabyte or Kilobyte size.*

spring.http.multipart.location= *# Intermediate location of uploaded files.*

spring.http.multipart.max-file-size=1MB *# Max file size. Values can use the suffixed "MB" or "KB" to indicate a Megabyte or Kilobyte size.*

spring.http.multipart.max-request-size=10MB *# Max request size. Values can use the suffixed "MB" or "KB" to indicate a Megabyte or Kilobyte size.*

spring.http.multipart.resolve-lazily=false *# Whether to resolve the multipart request lazily at the time of file or parameter access.*

*# JACKSON (*[JacksonProperties](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/jackson/JacksonProperties.java))

spring.jackson.date-format= *# Date format string or a fully-qualified date format class name. For instance `yyyy-MM-dd HH:mm:ss`.*

spring.jackson.default-property-inclusion= *# Controls the inclusion of properties during serialization.*

spring.jackson.deserialization.\*= *# Jackson on/off features that affect the way Java objects are deserialized.*

spring.jackson.generator.\*= *# Jackson on/off features for generators.*

spring.jackson.joda-date-time-format= *# Joda date time format string. If not configured, "date-format" will be used as a fallback if it is configured with a format string.*

spring.jackson.locale= *# Locale used for formatting.*

spring.jackson.mapper.\*= *# Jackson general purpose on/off features.*

spring.jackson.parser.\*= *# Jackson on/off features for parsers.*

spring.jackson.property-naming-strategy= *# One of the constants on Jackson's PropertyNamingStrategy. Can also be a fully-qualified class name of a PropertyNamingStrategy subclass.*

spring.jackson.serialization.\*= *# Jackson on/off features that affect the way Java objects are serialized.*

spring.jackson.time-zone= *# Time zone used when formatting dates. For instance `America/Los\_Angeles`*

*# JERSEY (*[JerseyProperties](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/jersey/JerseyProperties.java))

spring.jersey.application-path= *# Path that serves as the base URI for the application. Overrides the value of "@ApplicationPath" if specified.*

spring.jersey.filter.order=0 *# Jersey filter chain order.*

spring.jersey.init.\*= *# Init parameters to pass to Jersey via the servlet or filter.*

spring.jersey.servlet.load-on-startup=-1 *# Load on startup priority of the Jersey servlet.*

spring.jersey.type=servlet *# Jersey integration type.*

*# SPRING LDAP (*[LdapProperties](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/ldap/LdapProperties.java))

spring.ldap.urls= *# LDAP URLs of the server.*

spring.ldap.base= *# Base suffix from which all operations should originate.*

spring.ldap.username= *# Login user of the server.*

spring.ldap.password= *# Login password of the server.*

spring.ldap.base-environment.\*= *# LDAP specification settings.*

*# EMBEDDED LDAP (*[EmbeddedLdapProperties](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/ldap/embedded/EmbeddedLdapProperties.java))

spring.ldap.embedded.base-dn= *# The base DN*

spring.ldap.embedded.credential.username= *# Embedded LDAP username.*

spring.ldap.embedded.credential.password= *# Embedded LDAP password.*

spring.ldap.embedded.ldif=classpath:schema.ldif *# Schema (LDIF) script resource reference.*

spring.ldap.embedded.port= *# Embedded LDAP port.*

spring.ldap.embedded.validation.enabled=true *# Enable LDAP schema validation.*

spring.ldap.embedded.validation.schema= *# Path to the custom schema.*

*# SPRING MOBILE DEVICE VIEWS (*[DeviceDelegatingViewResolverAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/mobile/DeviceDelegatingViewResolverAutoConfiguration.java))

spring.mobile.devicedelegatingviewresolver.enable-fallback=false *# Enable support for fallback resolution.*

spring.mobile.devicedelegatingviewresolver.enabled=false *# Enable device view resolver.*

spring.mobile.devicedelegatingviewresolver.mobile-prefix=mobile/ *# Prefix that gets prepended to view names for mobile devices.*

spring.mobile.devicedelegatingviewresolver.mobile-suffix= *# Suffix that gets appended to view names for mobile devices.*

spring.mobile.devicedelegatingviewresolver.normal-prefix= *# Prefix that gets prepended to view names for normal devices.*

spring.mobile.devicedelegatingviewresolver.normal-suffix= *# Suffix that gets appended to view names for normal devices.*

spring.mobile.devicedelegatingviewresolver.tablet-prefix=tablet/ *# Prefix that gets prepended to view names for tablet devices.*

spring.mobile.devicedelegatingviewresolver.tablet-suffix= *# Suffix that gets appended to view names for tablet devices.*

*# SPRING MOBILE SITE PREFERENCE (*[SitePreferenceAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/mobile/SitePreferenceAutoConfiguration.java))

spring.mobile.sitepreference.enabled=true *# Enable SitePreferenceHandler.*

*# MUSTACHE TEMPLATES (*[MustacheAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/mustache/MustacheAutoConfiguration.java))

spring.mustache.allow-request-override= *# Set whether HttpServletRequest attributes are allowed to override (hide) controller generated model attributes of the same name.*

spring.mustache.allow-session-override= *# Set whether HttpSession attributes are allowed to override (hide) controller generated model attributes of the same name.*

spring.mustache.cache= *# Enable template caching.*

spring.mustache.charset= *# Template encoding.*

spring.mustache.check-template-location= *# Check that the templates location exists.*

spring.mustache.content-type= *# Content-Type value.*

spring.mustache.enabled= *# Enable MVC view resolution for this technology.*

spring.mustache.expose-request-attributes= *# Set whether all request attributes should be added to the model prior to merging with the template.*

spring.mustache.expose-session-attributes= *# Set whether all HttpSession attributes should be added to the model prior to merging with the template.*

spring.mustache.expose-spring-macro-helpers= *# Set whether to expose a RequestContext for use by Spring's macro library, under the name "springMacroRequestContext".*

spring.mustache.prefix=classpath:/templates/ *# Prefix to apply to template names.*

spring.mustache.request-context-attribute= *# Name of the RequestContext attribute for all views.*

spring.mustache.suffix=.html *# Suffix to apply to template names.*

spring.mustache.view-names= *# White list of view names that can be resolved.*

*# SPRING MVC (*[WebMvcProperties](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/web/WebMvcProperties.java))

spring.mvc.async.request-timeout= *# Amount of time (in milliseconds) before asynchronous request handling times out.*

spring.mvc.date-format= *# Date format to use. For instance `dd/MM/yyyy`.*

spring.mvc.dispatch-trace-request=false *# Dispatch TRACE requests to the FrameworkServlet doService method.*

spring.mvc.dispatch-options-request=true *# Dispatch OPTIONS requests to the FrameworkServlet doService method.*

spring.mvc.favicon.enabled=true *# Enable resolution of favicon.ico.*

spring.mvc.formcontent.putfilter.enabled=true *# Enable Spring's HttpPutFormContentFilter.*

spring.mvc.ignore-default-model-on-redirect=true *# If the content of the "default" model should be ignored during redirect scenarios.*

spring.mvc.locale= *# Locale to use. By default, this locale is overridden by the "Accept-Language" header.*

spring.mvc.locale-resolver=accept-header *# Define how the locale should be resolved.*

spring.mvc.log-resolved-exception=false *# Enable warn logging of exceptions resolved by a "HandlerExceptionResolver".*

spring.mvc.media-types.\*= *# Maps file extensions to media types for content negotiation.*

spring.mvc.message-codes-resolver-format= *# Formatting strategy for message codes. For instance `PREFIX\_ERROR\_CODE`.*

spring.mvc.servlet.load-on-startup=-1 *# Load on startup priority of the Spring Web Services servlet.*

spring.mvc.static-path-pattern=/\*\* *# Path pattern used for static resources.*

spring.mvc.throw-exception-if-no-handler-found=false *# If a "NoHandlerFoundException" should be thrown if no Handler was found to process a request.*

spring.mvc.view.prefix= *# Spring MVC view prefix.*

spring.mvc.view.suffix= *# Spring MVC view suffix.*

*# SPRING RESOURCES HANDLING (*[ResourceProperties](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/web/ResourceProperties.java))

spring.resources.add-mappings=true *# Enable default resource handling.*

spring.resources.cache-period= *# Cache period for the resources served by the resource handler, in seconds.*

spring.resources.chain.cache=true *# Enable caching in the Resource chain.*

spring.resources.chain.enabled= *# Enable the Spring Resource Handling chain. Disabled by default unless at least one strategy has been enabled.*

spring.resources.chain.gzipped=false *# Enable resolution of already gzipped resources.*

spring.resources.chain.html-application-cache=false *# Enable HTML5 application cache manifest rewriting.*

spring.resources.chain.strategy.content.enabled=false *# Enable the content Version Strategy.*

spring.resources.chain.strategy.content.paths=/\*\* *# Comma-separated list of patterns to apply to the Version Strategy.*

spring.resources.chain.strategy.fixed.enabled=false *# Enable the fixed Version Strategy.*

spring.resources.chain.strategy.fixed.paths=/\*\* *# Comma-separated list of patterns to apply to the Version Strategy.*

spring.resources.chain.strategy.fixed.version= *# Version string to use for the Version Strategy.*

spring.resources.static-locations=classpath:/META-INF/resources/,classpath:/resources/,classpath:/static/,classpath:/public/ *# Locations of static resources.*

*# SPRING SESSION (*[SessionProperties](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/session/SessionProperties.java))

spring.session.hazelcast.flush-mode=on-save *# Sessions flush mode.*

spring.session.hazelcast.map-name=spring:session:sessions *# Name of the map used to store sessions.*

spring.session.jdbc.initializer.enabled= *# Create the required session tables on startup if necessary. Enabled automatically if the default table name is set or a custom schema is configured.*

spring.session.jdbc.schema=classpath:org/springframework/session/jdbc/schema-@@platform@@.sql *# Path to the SQL file to use to initialize the database schema.*

spring.session.jdbc.table-name=SPRING\_SESSION *# Name of database table used to store sessions.*

spring.session.mongo.collection-name=sessions *# Collection name used to store sessions.*

spring.session.redis.flush-mode=on-save *# Sessions flush mode.*

spring.session.redis.namespace= *# Namespace for keys used to store sessions.*

spring.session.store-type= *# Session store type.*

*# SPRING SOCIAL (*[SocialWebAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/social/SocialWebAutoConfiguration.java))

spring.social.auto-connection-views=false *# Enable the connection status view for supported providers.*

*# SPRING SOCIAL FACEBOOK (*[FacebookAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/social/FacebookAutoConfiguration.java))

spring.social.facebook.app-id= *# your application's Facebook App ID*

spring.social.facebook.app-secret= *# your application's Facebook App Secret*

*# SPRING SOCIAL LINKEDIN (*[LinkedInAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/social/LinkedInAutoConfiguration.java))

spring.social.linkedin.app-id= *# your application's LinkedIn App ID*

spring.social.linkedin.app-secret= *# your application's LinkedIn App Secret*

*# SPRING SOCIAL TWITTER (*[TwitterAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/social/TwitterAutoConfiguration.java))

spring.social.twitter.app-id= *# your application's Twitter App ID*

spring.social.twitter.app-secret= *# your application's Twitter App Secret*

*# THYMELEAF (*[ThymeleafAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/thymeleaf/ThymeleafAutoConfiguration.java))

spring.thymeleaf.cache=true *# Enable template caching.*

spring.thymeleaf.check-template=true *# Check that the template exists before rendering it.*

spring.thymeleaf.check-template-location=true *# Check that the templates location exists.*

spring.thymeleaf.content-type=text/html *# Content-Type value.*

spring.thymeleaf.enabled=true *# Enable MVC Thymeleaf view resolution.*

spring.thymeleaf.encoding=UTF-8 *# Template encoding.*

spring.thymeleaf.excluded-view-names= *# Comma-separated list of view names that should be excluded from resolution.*

spring.thymeleaf.mode=HTML5 *# Template mode to be applied to templates. See also StandardTemplateModeHandlers.*

spring.thymeleaf.prefix=classpath:/templates/ *# Prefix that gets prepended to view names when building a URL.*

spring.thymeleaf.suffix=.html *# Suffix that gets appended to view names when building a URL.*

spring.thymeleaf.template-resolver-order= *# Order of the template resolver in the chain.*

spring.thymeleaf.view-names= *# Comma-separated list of view names that can be resolved.*

*# SPRING WEB SERVICES (*[WebServicesProperties](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/webservices/WebServicesProperties.java))

spring.webservices.path=/services *# Path that serves as the base URI for the services.*

spring.webservices.servlet.init= *# Servlet init parameters to pass to Spring Web Services.*

spring.webservices.servlet.load-on-startup=-1 *# Load on startup priority of the Spring Web Services servlet.*

*# ----------------------------------------*

*# SECURITY PROPERTIES*

*# ----------------------------------------*

*# SECURITY (*[SecurityProperties](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/security/SecurityProperties.java))

security.basic.authorize-mode=role *# Security authorize mode to apply.*

security.basic.enabled=true *# Enable basic authentication.*

security.basic.path=/\*\* *# Comma-separated list of paths to secure.*

security.basic.realm=Spring *# HTTP basic realm name.*

security.enable-csrf=false *# Enable Cross Site Request Forgery support.*

security.filter-order=0 *# Security filter chain order.*

security.filter-dispatcher-types=ASYNC, FORWARD, INCLUDE, REQUEST *# Security filter chain dispatcher types.*

security.headers.cache=true *# Enable cache control HTTP headers.*

security.headers.content-security-policy= *# Value for content security policy header.*

security.headers.content-security-policy-mode=default *# Content security policy mode.*

security.headers.content-type=true *# Enable "X-Content-Type-Options" header.*

security.headers.frame=true *# Enable "X-Frame-Options" header.*

security.headers.hsts=all *# HTTP Strict Transport Security (HSTS) mode (none, domain, all).*

security.headers.xss=true *# Enable cross site scripting (XSS) protection.*

security.ignored= *# Comma-separated list of paths to exclude from the default secured paths.*

security.require-ssl=false *# Enable secure channel for all requests.*

security.sessions=stateless *# Session creation policy (always, never, if\_required, stateless).*

security.user.name=user *# Default user name.*

security.user.password= *# Password for the default user name. A random password is logged on startup by default.*

security.user.role=USER *# Granted roles for the default user name.*

*# SECURITY OAUTH2 CLIENT (*[OAuth2ClientProperties](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/security/oauth2/OAuth2ClientProperties.java))

security.oauth2.client.client-id= *# OAuth2 client id.*

security.oauth2.client.client-secret= *# OAuth2 client secret. A random secret is generated by default*

*# SECURITY OAUTH2 RESOURCES (*[ResourceServerProperties](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/security/oauth2/resource/ResourceServerProperties.java))

security.oauth2.resource.filter-order= *# The order of the filter chain used to authenticate tokens.*

security.oauth2.resource.id= *# Identifier of the resource.*

security.oauth2.resource.jwt.key-uri= *# The URI of the JWT token. Can be set if the value is not available and the key is public.*

security.oauth2.resource.jwt.key-value= *# The verification key of the JWT token. Can either be a symmetric secret or PEM-encoded RSA public key.*

security.oauth2.resource.jwk.key-set-uri= *# The URI for getting the set of keys that can be used to validate the token.*

security.oauth2.resource.prefer-token-info=true *# Use the token info, can be set to false to use the user info.*

security.oauth2.resource.service-id=resource *#*

security.oauth2.resource.token-info-uri= *# URI of the token decoding endpoint.*

security.oauth2.resource.token-type= *# The token type to send when using the userInfoUri.*

security.oauth2.resource.user-info-uri= *# URI of the user endpoint.*

*# SECURITY OAUTH2 SSO (*[OAuth2SsoProperties](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/security/oauth2/client/OAuth2SsoProperties.java))

security.oauth2.sso.filter-order= *# Filter order to apply if not providing an explicit WebSecurityConfigurerAdapter*

security.oauth2.sso.login-path=/login *# Path to the login page, i.e. the one that triggers the redirect to the OAuth2 Authorization Server*

*# ----------------------------------------*

*# DATA PROPERTIES*

*# ----------------------------------------*

*# FLYWAY (*[FlywayProperties](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/flyway/FlywayProperties.java))

flyway.baseline-description= *#*

flyway.baseline-version=1 *# version to start migration*

flyway.baseline-on-migrate= *#*

flyway.check-location=false *# Check that migration scripts location exists.*

flyway.clean-on-validation-error= *#*

flyway.enabled=true *# Enable flyway.*

flyway.encoding= *#*

flyway.ignore-failed-future-migration= *#*

flyway.init-sqls= *# SQL statements to execute to initialize a connection immediately after obtaining it.*

flyway.locations=classpath:db/migration *# locations of migrations scripts*

flyway.out-of-order= *#*

flyway.password= *# JDBC password if you want Flyway to create its own DataSource*

flyway.placeholder-prefix= *#*

flyway.placeholder-replacement= *#*

flyway.placeholder-suffix= *#*

flyway.placeholders.\*= *#*

flyway.schemas= *# schemas to update*

flyway.sql-migration-prefix=V *#*

flyway.sql-migration-separator= *#*

flyway.sql-migration-suffix=.sql *#*

flyway.table= *#*

flyway.url= *# JDBC url of the database to migrate. If not set, the primary configured data source is used.*

flyway.user= *# Login user of the database to migrate.*

flyway.validate-on-migrate= *#*

*# LIQUIBASE (*[LiquibaseProperties](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/liquibase/LiquibaseProperties.java))

liquibase.change-log=classpath:/db/changelog/db.changelog-master.yaml *# Change log configuration path.*

liquibase.check-change-log-location=true *# Check the change log location exists.*

liquibase.contexts= *# Comma-separated list of runtime contexts to use.*

liquibase.default-schema= *# Default database schema.*

liquibase.drop-first=false *# Drop the database schema first.*

liquibase.enabled=true *# Enable liquibase support.*

liquibase.labels= *# Comma-separated list of runtime labels to use.*

liquibase.parameters.\*= *# Change log parameters.*

liquibase.password= *# Login password of the database to migrate.*

liquibase.rollback-file= *# File to which rollback SQL will be written when an update is performed.*

liquibase.url= *# JDBC url of the database to migrate. If not set, the primary configured data source is used.*

liquibase.user= *# Login user of the database to migrate.*

*# COUCHBASE (*[CouchbaseProperties](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/couchbase/CouchbaseProperties.java))

spring.couchbase.bootstrap-hosts= *# Couchbase nodes (host or IP address) to bootstrap from.*

spring.couchbase.bucket.name=default *# Name of the bucket to connect to.*

spring.couchbase.bucket.password= *# Password of the bucket.*

spring.couchbase.env.endpoints.key-value=1 *# Number of sockets per node against the Key/value service.*

spring.couchbase.env.endpoints.query=1 *# Number of sockets per node against the Query (N1QL) service.*

spring.couchbase.env.endpoints.view=1 *# Number of sockets per node against the view service.*

spring.couchbase.env.ssl.enabled= *# Enable SSL support. Enabled automatically if a "keyStore" is provided unless specified otherwise.*

spring.couchbase.env.ssl.key-store= *# Path to the JVM key store that holds the certificates.*

spring.couchbase.env.ssl.key-store-password= *# Password used to access the key store.*

spring.couchbase.env.timeouts.connect=5000 *# Bucket connections timeout in milliseconds.*

spring.couchbase.env.timeouts.key-value=2500 *# Blocking operations performed on a specific key timeout in milliseconds.*

spring.couchbase.env.timeouts.query=7500 *# N1QL query operations timeout in milliseconds.*

spring.couchbase.env.timeouts.socket-connect=1000 *# Socket connect connections timeout in milliseconds.*

spring.couchbase.env.timeouts.view=7500 *# Regular and geospatial view operations timeout in milliseconds.*

*# DAO (*[PersistenceExceptionTranslationAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/dao/PersistenceExceptionTranslationAutoConfiguration.java))

spring.dao.exceptiontranslation.enabled=true *# Enable the PersistenceExceptionTranslationPostProcessor.*

*# CASSANDRA (*[CassandraProperties](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/cassandra/CassandraProperties.java))

spring.data.cassandra.cluster-name= *# Name of the Cassandra cluster.*

spring.data.cassandra.compression=none *# Compression supported by the Cassandra binary protocol.*

spring.data.cassandra.connect-timeout-millis= *# Socket option: connection time out.*

spring.data.cassandra.consistency-level= *# Queries consistency level.*

spring.data.cassandra.contact-points=localhost *# Comma-separated list of cluster node addresses.*

spring.data.cassandra.fetch-size= *# Queries default fetch size.*

spring.data.cassandra.keyspace-name= *# Keyspace name to use.*

spring.data.cassandra.load-balancing-policy= *# Class name of the load balancing policy.*

spring.data.cassandra.port= *# Port of the Cassandra server.*

spring.data.cassandra.password= *# Login password of the server.*

spring.data.cassandra.read-timeout-millis= *# Socket option: read time out.*

spring.data.cassandra.reconnection-policy= *# Reconnection policy class.*

spring.data.cassandra.repositories.enabled= *# Enable Cassandra repositories.*

spring.data.cassandra.retry-policy= *# Class name of the retry policy.*

spring.data.cassandra.serial-consistency-level= *# Queries serial consistency level.*

spring.data.cassandra.schema-action=none *# Schema action to take at startup.*

spring.data.cassandra.ssl=false *# Enable SSL support.*

spring.data.cassandra.username= *# Login user of the server.*

*# DATA COUCHBASE (*[CouchbaseDataProperties](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/data/couchbase/CouchbaseDataProperties.java))

spring.data.couchbase.auto-index=false *# Automatically create views and indexes.*

spring.data.couchbase.consistency=read-your-own-writes *# Consistency to apply by default on generated queries.*

spring.data.couchbase.repositories.enabled=true *# Enable Couchbase repositories.*

*# ELASTICSEARCH (*[ElasticsearchProperties](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/data/elasticsearch/ElasticsearchProperties.java))

spring.data.elasticsearch.cluster-name=elasticsearch *# Elasticsearch cluster name.*

spring.data.elasticsearch.cluster-nodes= *# Comma-separated list of cluster node addresses. If not specified, starts a client node.*

spring.data.elasticsearch.properties.\*= *# Additional properties used to configure the client.*

spring.data.elasticsearch.repositories.enabled=true *# Enable Elasticsearch repositories.*

*# DATA LDAP*

spring.data.ldap.repositories.enabled=true *# Enable LDAP repositories.*

*# MONGODB (*[MongoProperties](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/mongo/MongoProperties.java))

spring.data.mongodb.authentication-database= *# Authentication database name.*

spring.data.mongodb.database=test *# Database name.*

spring.data.mongodb.field-naming-strategy= *# Fully qualified name of the FieldNamingStrategy to use.*

spring.data.mongodb.grid-fs-database= *# GridFS database name.*

spring.data.mongodb.host=localhost *# Mongo server host. Cannot be set with uri.*

spring.data.mongodb.password= *# Login password of the mongo server. Cannot be set with uri.*

spring.data.mongodb.port=27017 *# Mongo server port. Cannot be set with uri.*

spring.data.mongodb.repositories.enabled=true *# Enable Mongo repositories.*

spring.data.mongodb.uri=mongodb://localhost/test *# Mongo database URI. Cannot be set with host, port and credentials.*

spring.data.mongodb.username= *# Login user of the mongo server. Cannot be set with uri.*

*# DATA REDIS*

spring.data.redis.repositories.enabled=true *# Enable Redis repositories.*

*# NEO4J (*[Neo4jProperties](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/data/neo4j/Neo4jProperties.java))

spring.data.neo4j.compiler= *# Compiler to use.*

spring.data.neo4j.embedded.enabled=true *# Enable embedded mode if the embedded driver is available.*

spring.data.neo4j.open-in-view=false *# Register OpenSessionInViewInterceptor. Binds a Neo4j Session to the thread for the entire processing of the request.*

spring.data.neo4j.password= *# Login password of the server.*

spring.data.neo4j.repositories.enabled=true *# Enable Neo4j repositories.*

spring.data.neo4j.uri= *# URI used by the driver. Auto-detected by default.*

spring.data.neo4j.username= *# Login user of the server.*

*# DATA REST (*[RepositoryRestProperties](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/data/rest/RepositoryRestProperties.java))

spring.data.rest.base-path= *# Base path to be used by Spring Data REST to expose repository resources.*

spring.data.rest.default-page-size= *# Default size of pages.*

spring.data.rest.detection-strategy=default *# Strategy to use to determine which repositories get exposed.*

spring.data.rest.enable-enum-translation= *# Enable enum value translation via the Spring Data REST default resource bundle.*

spring.data.rest.limit-param-name= *# Name of the URL query string parameter that indicates how many results to return at once.*

spring.data.rest.max-page-size= *# Maximum size of pages.*

spring.data.rest.page-param-name= *# Name of the URL query string parameter that indicates what page to return.*

spring.data.rest.return-body-on-create= *# Return a response body after creating an entity.*

spring.data.rest.return-body-on-update= *# Return a response body after updating an entity.*

spring.data.rest.sort-param-name= *# Name of the URL query string parameter that indicates what direction to sort results.*

*# SOLR (*[SolrProperties](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/solr/SolrProperties.java))

spring.data.solr.host=http://127.0.0.1:8983/solr *# Solr host. Ignored if "zk-host" is set.*

spring.data.solr.repositories.enabled=true *# Enable Solr repositories.*

spring.data.solr.zk-host= *# ZooKeeper host address in the form HOST:PORT.*

*# DATASOURCE (*[DataSourceAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/jdbc/DataSourceAutoConfiguration.java) & [DataSourceProperties](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/jdbc/DataSourceProperties.java))

spring.datasource.continue-on-error=false *# Do not stop if an error occurs while initializing the database.*

spring.datasource.data= *# Data (DML) script resource references.*

spring.datasource.data-username= *# User of the database to execute DML scripts (if different).*

spring.datasource.data-password= *# Password of the database to execute DML scripts (if different).*

spring.datasource.dbcp2.\*= *# Commons DBCP2 specific settings*

spring.datasource.driver-class-name= *# Fully qualified name of the JDBC driver. Auto-detected based on the URL by default.*

spring.datasource.generate-unique-name=false *# Generate a random datasource name.*

spring.datasource.hikari.\*= *# Hikari specific settings*

spring.datasource.initialize=true *# Populate the database using 'data.sql'.*

spring.datasource.jmx-enabled=false *# Enable JMX support (if provided by the underlying pool).*

spring.datasource.jndi-name= *# JNDI location of the datasource. Class, url, username & password are ignored when set.*

spring.datasource.name=testdb *# Name of the datasource.*

spring.datasource.password= *# Login password of the database.*

spring.datasource.platform=all *# Platform to use in the schema resource (schema-${platform}.sql).*

spring.datasource.schema= *# Schema (DDL) script resource references.*

spring.datasource.schema-username= *# User of the database to execute DDL scripts (if different).*

spring.datasource.schema-password= *# Password of the database to execute DDL scripts (if different).*

spring.datasource.separator=; *# Statement separator in SQL initialization scripts.*

spring.datasource.sql-script-encoding= *# SQL scripts encoding.*

spring.datasource.tomcat.\*= *# Tomcat datasource specific settings*

spring.datasource.type= *# Fully qualified name of the connection pool implementation to use. By default, it is auto-detected from the classpath.*

spring.datasource.url= *# JDBC url of the database.*

spring.datasource.username= *# Login user of the database.*

spring.datasource.xa.data-source-class-name= *# XA datasource fully qualified name.*

spring.datasource.xa.properties= *# Properties to pass to the XA data source.*

*# JEST (Elasticsearch HTTP client) (*[JestProperties](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/elasticsearch/jest/JestProperties.java))

spring.elasticsearch.jest.connection-timeout=3000 *# Connection timeout in milliseconds.*

spring.elasticsearch.jest.multi-threaded=true *# Enable connection requests from multiple execution threads.*

spring.elasticsearch.jest.password= *# Login password.*

spring.elasticsearch.jest.proxy.host= *# Proxy host the HTTP client should use.*

spring.elasticsearch.jest.proxy.port= *# Proxy port the HTTP client should use.*

spring.elasticsearch.jest.read-timeout=3000 *# Read timeout in milliseconds.*

spring.elasticsearch.jest.uris=http://localhost:9200 *# Comma-separated list of the Elasticsearch instances to use.*

spring.elasticsearch.jest.username= *# Login user.*

*# H2 Web Console (*[H2ConsoleProperties](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/h2/H2ConsoleProperties.java))

spring.h2.console.enabled=false *# Enable the console.*

spring.h2.console.path=/h2-console *# Path at which the console will be available.*

spring.h2.console.settings.trace=false *# Enable trace output.*

spring.h2.console.settings.web-allow-others=false *# Enable remote access.*

*# JOOQ (*[JooqAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/jooq/JooqAutoConfiguration.java))

spring.jooq.sql-dialect= *# SQLDialect JOOQ used when communicating with the configured datasource. For instance `POSTGRES`*

*# JPA (*[JpaBaseConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/orm/jpa/JpaBaseConfiguration.java), [HibernateJpaAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/orm/jpa/HibernateJpaAutoConfiguration.java))

spring.data.jpa.repositories.enabled=true *# Enable JPA repositories.*

spring.jpa.database= *# Target database to operate on, auto-detected by default. Can be alternatively set using the "databasePlatform" property.*

spring.jpa.database-platform= *# Name of the target database to operate on, auto-detected by default. Can be alternatively set using the "Database" enum.*

spring.jpa.generate-ddl=false *# Initialize the schema on startup.*

spring.jpa.hibernate.ddl-auto= *# DDL mode. This is actually a shortcut for the "hibernate.hbm2ddl.auto" property. Default to "create-drop" when using an embedded database, "none" otherwise.*

spring.jpa.hibernate.naming.implicit-strategy= *# Hibernate 5 implicit naming strategy fully qualified name.*

spring.jpa.hibernate.naming.physical-strategy= *# Hibernate 5 physical naming strategy fully qualified name.*

spring.jpa.hibernate.naming.strategy= *# Hibernate 4 naming strategy fully qualified name. Not supported with Hibernate 5.*

spring.jpa.hibernate.use-new-id-generator-mappings= *# Use Hibernate's newer IdentifierGenerator for AUTO, TABLE and SEQUENCE.*

spring.jpa.open-in-view=true *# Register OpenEntityManagerInViewInterceptor. Binds a JPA EntityManager to the thread for the entire processing of the request.*

spring.jpa.properties.\*= *# Additional native properties to set on the JPA provider.*

spring.jpa.show-sql=false *# Enable logging of SQL statements.*

*# JTA (*[JtaAutoConfiguration](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/transaction/jta/JtaAutoConfiguration.java))

spring.jta.enabled=true *# Enable JTA support.*

spring.jta.log-dir= *# Transaction logs directory.*

spring.jta.transaction-manager-id= *# Transaction manager unique identifier.*

*# ATOMIKOS (*[AtomikosProperties](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot/src/main/java/org/springframework/boot/jta/atomikos/AtomikosProperties.java))

spring.jta.atomikos.connectionfactory.borrow-connection-timeout=30 *# Timeout, in seconds, for borrowing connections from the pool.*

spring.jta.atomikos.connectionfactory.ignore-session-transacted-flag=true *# Whether or not to ignore the transacted flag when creating session.*

spring.jta.atomikos.connectionfactory.local-transaction-mode=false *# Whether or not local transactions are desired.*

spring.jta.atomikos.connectionfactory.maintenance-interval=60 *# The time, in seconds, between runs of the pool's maintenance thread.*

spring.jta.atomikos.connectionfactory.max-idle-time=60 *# The time, in seconds, after which connections are cleaned up from the pool.*

spring.jta.atomikos.connectionfactory.max-lifetime=0 *# The time, in seconds, that a connection can be pooled for before being destroyed. 0 denotes no limit.*

spring.jta.atomikos.connectionfactory.max-pool-size=1 *# The maximum size of the pool.*

spring.jta.atomikos.connectionfactory.min-pool-size=1 *# The minimum size of the pool.*

spring.jta.atomikos.connectionfactory.reap-timeout=0 *# The reap timeout, in seconds, for borrowed connections. 0 denotes no limit.*

spring.jta.atomikos.connectionfactory.unique-resource-name=jmsConnectionFactory *# The unique name used to identify the resource during recovery.*

spring.jta.atomikos.datasource.borrow-connection-timeout=30 *# Timeout, in seconds, for borrowing connections from the pool.*

spring.jta.atomikos.datasource.default-isolation-level= *# Default isolation level of connections provided by the pool.*

spring.jta.atomikos.datasource.login-timeout= *# Timeout, in seconds, for establishing a database connection.*

spring.jta.atomikos.datasource.maintenance-interval=60 *# The time, in seconds, between runs of the pool's maintenance thread.*

spring.jta.atomikos.datasource.max-idle-time=60 *# The time, in seconds, after which connections are cleaned up from the pool.*

spring.jta.atomikos.datasource.max-lifetime=0 *# The time, in seconds, that a connection can be pooled for before being destroyed. 0 denotes no limit.*

spring.jta.atomikos.datasource.max-pool-size=1 *# The maximum size of the pool.*

spring.jta.atomikos.datasource.min-pool-size=1 *# The minimum size of the pool.*

spring.jta.atomikos.datasource.reap-timeout=0 *# The reap timeout, in seconds, for borrowed connections. 0 denotes no limit.*

spring.jta.atomikos.datasource.test-query= *# SQL query or statement used to validate a connection before returning it.*

spring.jta.atomikos.datasource.unique-resource-name=dataSource *# The unique name used to identify the resource during recovery.*

spring.jta.atomikos.properties.checkpoint-interval=500 *# Interval between checkpoints.*

spring.jta.atomikos.properties.default-jta-timeout=10000 *# Default timeout for JTA transactions.*

spring.jta.atomikos.properties.enable-logging=true *# Enable disk logging.*

spring.jta.atomikos.properties.force-shutdown-on-vm-exit=false *# Specify if a VM shutdown should trigger forced shutdown of the transaction core.*

spring.jta.atomikos.properties.log-base-dir= *# Directory in which the log files should be stored.*

spring.jta.atomikos.properties.log-base-name=tmlog *# Transactions log file base name.*

spring.jta.atomikos.properties.max-actives=50 *# Maximum number of active transactions.*

spring.jta.atomikos.properties.max-timeout=300000 *# Maximum timeout (in milliseconds) that can be allowed for transactions.*

spring.jta.atomikos.properties.serial-jta-transactions=true *# Specify if sub-transactions should be joined when possible.*

spring.jta.atomikos.properties.service= *# Transaction manager implementation that should be started.*

spring.jta.atomikos.properties.threaded-two-phase-commit=false *# Use different (and concurrent) threads for two-phase commit on the participating resources.*

spring.jta.atomikos.properties.transaction-manager-unique-name= *# Transaction manager's unique name.*

*# BITRONIX*

spring.jta.bitronix.connectionfactory.acquire-increment=1 *# Number of connections to create when growing the pool.*

spring.jta.bitronix.connectionfactory.acquisition-interval=1 *# Time, in seconds, to wait before trying to acquire a connection again after an invalid connection was acquired.*

spring.jta.bitronix.connectionfactory.acquisition-timeout=30 *# Timeout, in seconds, for acquiring connections from the pool.*

spring.jta.bitronix.connectionfactory.allow-local-transactions=true *# Whether or not the transaction manager should allow mixing XA and non-XA transactions.*

spring.jta.bitronix.connectionfactory.apply-transaction-timeout=false *# Whether or not the transaction timeout should be set on the XAResource when it is enlisted.*

spring.jta.bitronix.connectionfactory.automatic-enlisting-enabled=true *# Whether or not resources should be enlisted and delisted automatically.*

spring.jta.bitronix.connectionfactory.cache-producers-consumers=true *# Whether or not produces and consumers should be cached.*

spring.jta.bitronix.connectionfactory.defer-connection-release=true *# Whether or not the provider can run many transactions on the same connection and supports transaction interleaving.*

spring.jta.bitronix.connectionfactory.ignore-recovery-failures=false *# Whether or not recovery failures should be ignored.*

spring.jta.bitronix.connectionfactory.max-idle-time=60 *# The time, in seconds, after which connections are cleaned up from the pool.*

spring.jta.bitronix.connectionfactory.max-pool-size=10 *# The maximum size of the pool. 0 denotes no limit.*

spring.jta.bitronix.connectionfactory.min-pool-size=0 *# The minimum size of the pool.*

spring.jta.bitronix.connectionfactory.password= *# The password to use to connect to the JMS provider.*

spring.jta.bitronix.connectionfactory.share-transaction-connections=false *# Whether or not connections in the ACCESSIBLE state can be shared within the context of a transaction.*

spring.jta.bitronix.connectionfactory.test-connections=true *# Whether or not connections should be tested when acquired from the pool.*

spring.jta.bitronix.connectionfactory.two-pc-ordering-position=1 *# The position that this resource should take during two-phase commit (always first is Integer.MIN\_VALUE, always last is Integer.MAX\_VALUE).*

spring.jta.bitronix.connectionfactory.unique-name=jmsConnectionFactory *# The unique name used to identify the resource during recovery.*

spring.jta.bitronix.connectionfactory.use-tm-join=true Whether or not TMJOIN should be used when starting XAResources.

spring.jta.bitronix.connectionfactory.user= *# The user to use to connect to the JMS provider.*

spring.jta.bitronix.datasource.acquire-increment=1 *# Number of connections to create when growing the pool.*

spring.jta.bitronix.datasource.acquisition-interval=1 *# Time, in seconds, to wait before trying to acquire a connection again after an invalid connection was acquired.*

spring.jta.bitronix.datasource.acquisition-timeout=30 *# Timeout, in seconds, for acquiring connections from the pool.*

spring.jta.bitronix.datasource.allow-local-transactions=true *# Whether or not the transaction manager should allow mixing XA and non-XA transactions.*

spring.jta.bitronix.datasource.apply-transaction-timeout=false *# Whether or not the transaction timeout should be set on the XAResource when it is enlisted.*

spring.jta.bitronix.datasource.automatic-enlisting-enabled=true *# Whether or not resources should be enlisted and delisted automatically.*

spring.jta.bitronix.datasource.cursor-holdability= *# The default cursor holdability for connections.*

spring.jta.bitronix.datasource.defer-connection-release=true *# Whether or not the database can run many transactions on the same connection and supports transaction interleaving.*

spring.jta.bitronix.datasource.enable-jdbc4-connection-test= *# Whether or not Connection.isValid() is called when acquiring a connection from the pool.*

spring.jta.bitronix.datasource.ignore-recovery-failures=false *# Whether or not recovery failures should be ignored.*

spring.jta.bitronix.datasource.isolation-level= *# The default isolation level for connections.*

spring.jta.bitronix.datasource.local-auto-commit= *# The default auto-commit mode for local transactions.*

spring.jta.bitronix.datasource.login-timeout= *# Timeout, in seconds, for establishing a database connection.*

spring.jta.bitronix.datasource.max-idle-time=60 *# The time, in seconds, after which connections are cleaned up from the pool.*

spring.jta.bitronix.datasource.max-pool-size=10 *# The maximum size of the pool. 0 denotes no limit.*

spring.jta.bitronix.datasource.min-pool-size=0 *# The minimum size of the pool.*

spring.jta.bitronix.datasource.prepared-statement-cache-size=0 *# The target size of the prepared statement cache. 0 disables the cache.*

spring.jta.bitronix.datasource.share-transaction-connections=false *# Whether or not connections in the ACCESSIBLE state can be shared within the context of a transaction.*

spring.jta.bitronix.datasource.test-query= *# SQL query or statement used to validate a connection before returning it.*

spring.jta.bitronix.datasource.two-pc-ordering-position=1 *# The position that this resource should take during two-phase commit (always first is Integer.MIN\_VALUE, always last is Integer.MAX\_VALUE).*

spring.jta.bitronix.datasource.unique-name=dataSource *# The unique name used to identify the resource during recovery.*

spring.jta.bitronix.datasource.use-tm-join=true Whether or not TMJOIN should be used when starting XAResources.

spring.jta.bitronix.properties.allow-multiple-lrc=false *# Allow multiple LRC resources to be enlisted into the same transaction.*

spring.jta.bitronix.properties.asynchronous2-pc=false *# Enable asynchronously execution of two phase commit.*

spring.jta.bitronix.properties.background-recovery-interval-seconds=60 *# Interval in seconds at which to run the recovery process in the background.*

spring.jta.bitronix.properties.current-node-only-recovery=true *# Recover only the current node.*

spring.jta.bitronix.properties.debug-zero-resource-transaction=false *# Log the creation and commit call stacks of transactions executed without a single enlisted resource.*

spring.jta.bitronix.properties.default-transaction-timeout=60 *# Default transaction timeout in seconds.*

spring.jta.bitronix.properties.disable-jmx=false *# Enable JMX support.*

spring.jta.bitronix.properties.exception-analyzer= *# Set the fully qualified name of the exception analyzer implementation to use.*

spring.jta.bitronix.properties.filter-log-status=false *# Enable filtering of logs so that only mandatory logs are written.*

spring.jta.bitronix.properties.force-batching-enabled=true *# Set if disk forces are batched.*

spring.jta.bitronix.properties.forced-write-enabled=true *# Set if logs are forced to disk.*

spring.jta.bitronix.properties.graceful-shutdown-interval=60 *# Maximum amount of seconds the TM will wait for transactions to get done before aborting them at shutdown time.*

spring.jta.bitronix.properties.jndi-transaction-synchronization-registry-name= *# JNDI name of the TransactionSynchronizationRegistry.*

spring.jta.bitronix.properties.jndi-user-transaction-name= *# JNDI name of the UserTransaction.*

spring.jta.bitronix.properties.journal=disk *# Name of the journal. Can be 'disk', 'null' or a class name.*

spring.jta.bitronix.properties.log-part1-filename=btm1.tlog *# Name of the first fragment of the journal.*

spring.jta.bitronix.properties.log-part2-filename=btm2.tlog *# Name of the second fragment of the journal.*

spring.jta.bitronix.properties.max-log-size-in-mb=2 *# Maximum size in megabytes of the journal fragments.*

spring.jta.bitronix.properties.resource-configuration-filename= *# ResourceLoader configuration file name.*

spring.jta.bitronix.properties.server-id= *# ASCII ID that must uniquely identify this TM instance. Default to the machine's IP address.*

spring.jta.bitronix.properties.skip-corrupted-logs=false *# Skip corrupted transactions log entries.*

spring.jta.bitronix.properties.warn-about-zero-resource-transaction=true *# Log a warning for transactions executed without a single enlisted resource.*

*# NARAYANA (*[NarayanaProperties](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot/src/main/java/org/springframework/boot/jta/narayana/NarayanaProperties.java))

spring.jta.narayana.default-timeout=60 *# Transaction timeout in seconds.*

spring.jta.narayana.expiry-scanners=com.arjuna.ats.internal.arjuna.recovery.ExpiredTransactionStatusManagerScanner *# Comma-separated list of expiry scanners.*

spring.jta.narayana.log-dir= *# Transaction object store directory.*

spring.jta.narayana.one-phase-commit=true *# Enable one phase commit optimisation.*

spring.jta.narayana.periodic-recovery-period=120 *# Interval in which periodic recovery scans are performed in seconds.*

spring.jta.narayana.recovery-backoff-period=10 *# Back off period between first and second phases of the recovery scan in seconds.*

spring.jta.narayana.recovery-db-pass= *# Database password to be used by recovery manager.*

spring.jta.narayana.recovery-db-user= *# Database username to be used by recovery manager.*

spring.jta.narayana.recovery-jms-pass= *# JMS password to be used by recovery manager.*

spring.jta.narayana.recovery-jms-user= *# JMS username to be used by recovery manager.*

spring.jta.narayana.recovery-modules= *# Comma-separated list of recovery modules.*

spring.jta.narayana.transaction-manager-id=1 *# Unique transaction manager id.*

spring.jta.narayana.xa-resource-orphan-filters= *# Comma-separated list of orphan filters.*

*# EMBEDDED MONGODB (*[EmbeddedMongoProperties](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/mongo/embedded/EmbeddedMongoProperties.java))

spring.mongodb.embedded.features=SYNC\_DELAY *# Comma-separated list of features to enable.*

spring.mongodb.embedded.storage.database-dir= *# Directory used for data storage.*

spring.mongodb.embedded.storage.oplog-size= *# Maximum size of the oplog in megabytes.*

spring.mongodb.embedded.storage.repl-set-name= *# Name of the replica set.*

spring.mongodb.embedded.version=2.6.10 *# Version of Mongo to use.*

*# REDIS (*[RedisProperties](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/data/redis/RedisProperties.java))

spring.redis.cluster.max-redirects= *# Maximum number of redirects to follow when executing commands across the cluster.*

spring.redis.cluster.nodes= *# Comma-separated list of "host:port" pairs to bootstrap from.*

spring.redis.database=0 *# Database index used by the connection factory.*

spring.redis.url= *# Connection URL, will override host, port and password (user will be ignored), e.g. redis://user:password@example.com:6379*

spring.redis.host=localhost *# Redis server host.*

spring.redis.password= *# Login password of the redis server.*

spring.redis.ssl=false *# Enable SSL support.*

spring.redis.pool.max-active=8 *# Max number of connections that can be allocated by the pool at a given time. Use a negative value for no limit.*

spring.redis.pool.max-idle=8 *# Max number of "idle" connections in the pool. Use a negative value to indicate an unlimited number of idle connections.*

spring.redis.pool.max-wait=-1 *# Maximum amount of time (in milliseconds) a connection allocation should block before throwing an exception when the pool is exhausted. Use a negative value to block indefinitely.*

spring.redis.pool.min-idle=0 *# Target for the minimum number of idle connections to maintain in the pool. This setting only has an effect if it is positive.*

spring.redis.port=6379 *# Redis server port.*

spring.redis.sentinel.master= *# Name of Redis server.*

spring.redis.sentinel.nodes= *# Comma-separated list of host:port pairs.*

spring.redis.timeout=0 *# Connection timeout in milliseconds.*

*# TRANSACTION (*[TransactionProperties](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/transaction/TransactionProperties.java))

spring.transaction.default-timeout= *# Default transaction timeout in seconds.*

spring.transaction.rollback-on-commit-failure= *# Perform the rollback on commit failures.*

*# ----------------------------------------*

*# INTEGRATION PROPERTIES*

*# ----------------------------------------*

*# ACTIVEMQ (*[ActiveMQProperties](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/jms/activemq/ActiveMQProperties.java))

spring.activemq.broker-url= *# URL of the ActiveMQ broker. Auto-generated by default.*

spring.activemq.close-timeout=15000 *# Time to wait, in milliseconds, before considering a close complete.*

spring.activemq.in-memory=true *# Specify if the default broker URL should be in memory. Ignored if an explicit broker has been specified.*

spring.activemq.non-blocking-redelivery=false *# Do not stop message delivery before re-delivering messages from a rolled back transaction. This implies that message order will not be preserved when this is enabled.*

spring.activemq.password= *# Login password of the broker.*

spring.activemq.send-timeout=0 *# Time to wait, in milliseconds, on Message sends for a response. Set it to 0 to indicate to wait forever.*

spring.activemq.user= *# Login user of the broker.*

spring.activemq.packages.trust-all= *# Trust all packages.*

spring.activemq.packages.trusted= *# Comma-separated list of specific packages to trust (when not trusting all packages).*

spring.activemq.pool.block-if-full=true *# Block when a connection is requested and the pool is full. Set it to false to throw a "JMSException" instead.*

spring.activemq.pool.block-if-full-timeout=-1 *# Blocking period, in milliseconds, before throwing an exception if the pool is still full.*

spring.activemq.pool.create-connection-on-startup=true *# Create a connection on startup. Can be used to warm-up the pool on startup.*

spring.activemq.pool.enabled=false *# Whether a PooledConnectionFactory should be created instead of a regular ConnectionFactory.*

spring.activemq.pool.expiry-timeout=0 *# Connection expiration timeout in milliseconds.*

spring.activemq.pool.idle-timeout=30000 *# Connection idle timeout in milliseconds.*

spring.activemq.pool.max-connections=1 *# Maximum number of pooled connections.*

spring.activemq.pool.maximum-active-session-per-connection=500 *# Maximum number of active sessions per connection.*

spring.activemq.pool.reconnect-on-exception=true *# Reset the connection when a "JMXException" occurs.*

spring.activemq.pool.time-between-expiration-check=-1 *# Time to sleep, in milliseconds, between runs of the idle connection eviction thread. When negative, no idle connection eviction thread runs.*

spring.activemq.pool.use-anonymous-producers=true *# Use only one anonymous "MessageProducer" instance. Set it to false to create one "MessageProducer" every time one is required.*

*# ARTEMIS (*[ArtemisProperties](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/jms/artemis/ArtemisProperties.java))

spring.artemis.embedded.cluster-password= *# Cluster password. Randomly generated on startup by default.*

spring.artemis.embedded.data-directory= *# Journal file directory. Not necessary if persistence is turned off.*

spring.artemis.embedded.enabled=true *# Enable embedded mode if the Artemis server APIs are available.*

spring.artemis.embedded.persistent=false *# Enable persistent store.*

spring.artemis.embedded.queues= *# Comma-separated list of queues to create on startup.*

spring.artemis.embedded.server-id= *# Server id. By default, an auto-incremented counter is used.*

spring.artemis.embedded.topics= *# Comma-separated list of topics to create on startup.*

spring.artemis.host=localhost *# Artemis broker host.*

spring.artemis.mode= *# Artemis deployment mode, auto-detected by default.*

spring.artemis.password= *# Login password of the broker.*

spring.artemis.port=61616 *# Artemis broker port.*

spring.artemis.user= *# Login user of the broker.*

*# SPRING BATCH (*[BatchProperties](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/batch/BatchProperties.java))

spring.batch.initializer.enabled= *# Create the required batch tables on startup if necessary. Enabled automatically if no custom table prefix is set or if a custom schema is configured.*

spring.batch.job.enabled=true *# Execute all Spring Batch jobs in the context on startup.*

spring.batch.job.names= *# Comma-separated list of job names to execute on startup (For instance `job1,job2`). By default, all Jobs found in the context are executed.*

spring.batch.schema=classpath:org/springframework/batch/core/schema-@@platform@@.sql *# Path to the SQL file to use to initialize the database schema.*

spring.batch.table-prefix= *# Table prefix for all the batch meta-data tables.*

*# JMS (*[JmsProperties](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/jms/JmsProperties.java))

spring.jms.jndi-name= *# Connection factory JNDI name. When set, takes precedence to others connection factory auto-configurations.*

spring.jms.listener.acknowledge-mode= *# Acknowledge mode of the container. By default, the listener is transacted with automatic acknowledgment.*

spring.jms.listener.auto-startup=true *# Start the container automatically on startup.*

spring.jms.listener.concurrency= *# Minimum number of concurrent consumers.*

spring.jms.listener.max-concurrency= *# Maximum number of concurrent consumers.*

spring.jms.pub-sub-domain=false *# Specify if the default destination type is topic.*

spring.jms.template.default-destination= *# Default destination to use on send/receive operations that do not have a destination parameter.*

spring.jms.template.delivery-delay= *# Delivery delay to use for send calls in milliseconds.*

spring.jms.template.delivery-mode= *# Delivery mode. Enable QoS when set.*

spring.jms.template.priority= *# Priority of a message when sending. Enable QoS when set.*

spring.jms.template.qos-enabled= *# Enable explicit QoS when sending a message.*

spring.jms.template.receive-timeout= *# Timeout to use for receive calls in milliseconds.*

spring.jms.template.time-to-live= *# Time-to-live of a message when sending in milliseconds. Enable QoS when set.*

*# APACHE KAFKA (*[KafkaProperties](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/kafka/KafkaProperties.java))

spring.kafka.bootstrap-servers= *# Comma-delimited list of host:port pairs to use for establishing the initial connection to the Kafka cluster.*

spring.kafka.client-id= *# Id to pass to the server when making requests; used for server-side logging.*

spring.kafka.consumer.auto-commit-interval= *# Frequency in milliseconds that the consumer offsets are auto-committed to Kafka if 'enable.auto.commit' true.*

spring.kafka.consumer.auto-offset-reset= *# What to do when there is no initial offset in Kafka or if the current offset does not exist any more on the server.*

spring.kafka.consumer.bootstrap-servers= *# Comma-delimited list of host:port pairs to use for establishing the initial connection to the Kafka cluster.*

spring.kafka.consumer.client-id= *# Id to pass to the server when making requests; used for server-side logging.*

spring.kafka.consumer.enable-auto-commit= *# If true the consumer's offset will be periodically committed in the background.*

spring.kafka.consumer.fetch-max-wait= *# Maximum amount of time in milliseconds the server will block before answering the fetch request if there isn't sufficient data to immediately satisfy the requirement given by "fetch.min.bytes".*

spring.kafka.consumer.fetch-min-size= *# Minimum amount of data the server should return for a fetch request in bytes.*

spring.kafka.consumer.group-id= *# Unique string that identifies the consumer group this consumer belongs to.*

spring.kafka.consumer.heartbeat-interval= *# Expected time in milliseconds between heartbeats to the consumer coordinator.*

spring.kafka.consumer.key-deserializer= *# Deserializer class for keys.*

spring.kafka.consumer.max-poll-records= *# Maximum number of records returned in a single call to poll().*

spring.kafka.consumer.value-deserializer= *# Deserializer class for values.*

spring.kafka.listener.ack-count= *# Number of records between offset commits when ackMode is "COUNT" or "COUNT\_TIME".*

spring.kafka.listener.ack-mode= *# Listener AckMode; see the spring-kafka documentation.*

spring.kafka.listener.ack-time= *# Time in milliseconds between offset commits when ackMode is "TIME" or "COUNT\_TIME".*

spring.kafka.listener.concurrency= *# Number of threads to run in the listener containers.*

spring.kafka.listener.poll-timeout= *# Timeout in milliseconds to use when polling the consumer.*

spring.kafka.producer.acks= *# Number of acknowledgments the producer requires the leader to have received before considering a request complete.*

spring.kafka.producer.batch-size= *# Number of records to batch before sending.*

spring.kafka.producer.bootstrap-servers= *# Comma-delimited list of host:port pairs to use for establishing the initial connection to the Kafka cluster.*

spring.kafka.producer.buffer-memory= *# Total bytes of memory the producer can use to buffer records waiting to be sent to the server.*

spring.kafka.producer.client-id= *# Id to pass to the server when making requests; used for server-side logging.*

spring.kafka.producer.compression-type= *# Compression type for all data generated by the producer.*

spring.kafka.producer.key-serializer= *# Serializer class for keys.*

spring.kafka.producer.retries= *# When greater than zero, enables retrying of failed sends.*

spring.kafka.producer.value-serializer= *# Serializer class for values.*

spring.kafka.properties.\*= *# Additional properties used to configure the client.*

spring.kafka.ssl.key-password= *# Password of the private key in the key store file.*

spring.kafka.ssl.keystore-location= *# Location of the key store file.*

spring.kafka.ssl.keystore-password= *# Store password for the key store file.*

spring.kafka.ssl.truststore-location= *# Location of the trust store file.*

spring.kafka.ssl.truststore-password= *# Store password for the trust store file.*

spring.kafka.template.default-topic= *# Default topic to which messages will be sent.*

*# RABBIT (*[RabbitProperties](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot-autoconfigure/src/main/java/org/springframework/boot/autoconfigure/amqp/RabbitProperties.java))

spring.rabbitmq.addresses= *# Comma-separated list of addresses to which the client should connect.*

spring.rabbitmq.cache.channel.checkout-timeout= *# Number of milliseconds to wait to obtain a channel if the cache size has been reached.*

spring.rabbitmq.cache.channel.size= *# Number of channels to retain in the cache.*

spring.rabbitmq.cache.connection.mode=channel *# Connection factory cache mode.*

spring.rabbitmq.cache.connection.size= *# Number of connections to cache.*

spring.rabbitmq.connection-timeout= *# Connection timeout, in milliseconds; zero for infinite.*

spring.rabbitmq.dynamic=true *# Create an AmqpAdmin bean.*

spring.rabbitmq.host=localhost *# RabbitMQ host.*

spring.rabbitmq.listener.simple.acknowledge-mode= *# Acknowledge mode of container.*

spring.rabbitmq.listener.simple.auto-startup=true *# Start the container automatically on startup.*

spring.rabbitmq.listener.simple.concurrency= *# Minimum number of consumers.*

spring.rabbitmq.listener.simple.default-requeue-rejected= *# Whether or not to requeue delivery failures; default `true`.*

spring.rabbitmq.listener.simple.idle-event-interval= *# How often idle container events should be published in milliseconds.*

spring.rabbitmq.listener.simple.max-concurrency= *# Maximum number of consumers.*

spring.rabbitmq.listener.simple.prefetch= *# Number of messages to be handled in a single request. It should be greater than or equal to the transaction size (if used).*

spring.rabbitmq.listener.simple.retry.enabled=false *# Whether or not publishing retries are enabled.*

spring.rabbitmq.listener.simple.retry.initial-interval=1000 *# Interval between the first and second attempt to deliver a message.*

spring.rabbitmq.listener.simple.retry.max-attempts=3 *# Maximum number of attempts to deliver a message.*

spring.rabbitmq.listener.simple.retry.max-interval=10000 *# Maximum interval between attempts.*

spring.rabbitmq.listener.simple.retry.multiplier=1.0 *# A multiplier to apply to the previous delivery retry interval.*

spring.rabbitmq.listener.simple.retry.stateless=true *# Whether or not retry is stateless or stateful.*

spring.rabbitmq.listener.simple.transaction-size= *# Number of messages to be processed in a transaction. For best results it should be less than or equal to the prefetch count.*

spring.rabbitmq.password= *# Login to authenticate against the broker.*

spring.rabbitmq.port=5672 *# RabbitMQ port.*

spring.rabbitmq.publisher-confirms=false *# Enable publisher confirms.*

spring.rabbitmq.publisher-returns=false *# Enable publisher returns.*

spring.rabbitmq.requested-heartbeat= *# Requested heartbeat timeout, in seconds; zero for none.*

spring.rabbitmq.ssl.enabled=false *# Enable SSL support.*

spring.rabbitmq.ssl.key-store= *# Path to the key store that holds the SSL certificate.*

spring.rabbitmq.ssl.key-store-password= *# Password used to access the key store.*

spring.rabbitmq.ssl.trust-store= *# Trust store that holds SSL certificates.*

spring.rabbitmq.ssl.trust-store-password= *# Password used to access the trust store.*

spring.rabbitmq.ssl.algorithm= *# SSL algorithm to use. By default configure by the rabbit client library.*

spring.rabbitmq.template.mandatory=false *# Enable mandatory messages.*

spring.rabbitmq.template.receive-timeout=0 *# Timeout for `receive()` methods.*

spring.rabbitmq.template.reply-timeout=5000 *# Timeout for `sendAndReceive()` methods.*

spring.rabbitmq.template.retry.enabled=false *# Set to true to enable retries in the `RabbitTemplate`.*

spring.rabbitmq.template.retry.initial-interval=1000 *# Interval between the first and second attempt to publish a message.*

spring.rabbitmq.template.retry.max-attempts=3 *# Maximum number of attempts to publish a message.*

spring.rabbitmq.template.retry.max-interval=10000 *# Maximum number of attempts to publish a message.*

spring.rabbitmq.template.retry.multiplier=1.0 *# A multiplier to apply to the previous publishing retry interval.*

spring.rabbitmq.username= *# Login user to authenticate to the broker.*

spring.rabbitmq.virtual-host= *# Virtual host to use when connecting to the broker.*

*# ----------------------------------------*

*# ACTUATOR PROPERTIES*

*# ----------------------------------------*

*# ENDPOINTS (*[AbstractEndpoint](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot-actuator/src/main/java/org/springframework/boot/actuate/endpoint/AbstractEndpoint.java) subclasses)

endpoints.enabled=true *# Enable endpoints.*

endpoints.sensitive= *# Default endpoint sensitive setting.*

endpoints.actuator.enabled=true *# Enable the endpoint.*

endpoints.actuator.path= *# Endpoint URL path.*

endpoints.actuator.sensitive=false *# Enable security on the endpoint.*

endpoints.auditevents.enabled= *# Enable the endpoint.*

endpoints.auditevents.path= *# Endpoint path.*

endpoints.auditevents.sensitive=false *# Enable security on the endpoint.*

endpoints.autoconfig.enabled= *# Enable the endpoint.*

endpoints.autoconfig.id= *# Endpoint identifier.*

endpoints.autoconfig.path= *# Endpoint path.*

endpoints.autoconfig.sensitive= *# Mark if the endpoint exposes sensitive information.*

endpoints.beans.enabled= *# Enable the endpoint.*

endpoints.beans.id= *# Endpoint identifier.*

endpoints.beans.path= *# Endpoint path.*

endpoints.beans.sensitive= *# Mark if the endpoint exposes sensitive information.*

endpoints.configprops.enabled= *# Enable the endpoint.*

endpoints.configprops.id= *# Endpoint identifier.*

endpoints.configprops.keys-to-sanitize=password,secret,key,token,.\*credentials.\*,vcap\_services *# Keys that should be sanitized. Keys can be simple strings that the property ends with or regex expressions.*

endpoints.configprops.path= *# Endpoint path.*

endpoints.configprops.sensitive= *# Mark if the endpoint exposes sensitive information.*

endpoints.docs.curies.enabled=false *# Enable the curie generation.*

endpoints.docs.enabled=true *# Enable actuator docs endpoint.*

endpoints.docs.path=/docs *#*

endpoints.docs.sensitive=false *#*

endpoints.dump.enabled= *# Enable the endpoint.*

endpoints.dump.id= *# Endpoint identifier.*

endpoints.dump.path= *# Endpoint path.*

endpoints.dump.sensitive= *# Mark if the endpoint exposes sensitive information.*

endpoints.env.enabled= *# Enable the endpoint.*

endpoints.env.id= *# Endpoint identifier.*

endpoints.env.keys-to-sanitize=password,secret,key,token,.\*credentials.\*,vcap\_services *# Keys that should be sanitized. Keys can be simple strings that the property ends with or regex expressions.*

endpoints.env.path= *# Endpoint path.*

endpoints.env.sensitive= *# Mark if the endpoint exposes sensitive information.*

endpoints.flyway.enabled= *# Enable the endpoint.*

endpoints.flyway.id= *# Endpoint identifier.*

endpoints.flyway.sensitive= *# Mark if the endpoint exposes sensitive information.*

endpoints.health.enabled= *# Enable the endpoint.*

endpoints.health.id= *# Endpoint identifier.*

endpoints.health.mapping.\*= *# Mapping of health statuses to HttpStatus codes. By default, registered health statuses map to sensible defaults (i.e. UP maps to 200).*

endpoints.health.path= *# Endpoint path.*

endpoints.health.sensitive= *# Mark if the endpoint exposes sensitive information.*

endpoints.health.time-to-live=1000 *# Time to live for cached result, in milliseconds.*

endpoints.heapdump.enabled= *# Enable the endpoint.*

endpoints.heapdump.path= *# Endpoint path.*

endpoints.heapdump.sensitive= *# Mark if the endpoint exposes sensitive information.*

endpoints.hypermedia.enabled=false *# Enable hypermedia support for endpoints.*

endpoints.info.enabled= *# Enable the endpoint.*

endpoints.info.id= *# Endpoint identifier.*

endpoints.info.path= *# Endpoint path.*

endpoints.info.sensitive= *# Mark if the endpoint exposes sensitive information.*

endpoints.jolokia.enabled=true *# Enable Jolokia endpoint.*

endpoints.jolokia.path=/jolokia *# Endpoint URL path.*

endpoints.jolokia.sensitive=true *# Enable security on the endpoint.*

endpoints.liquibase.enabled= *# Enable the endpoint.*

endpoints.liquibase.id= *# Endpoint identifier.*

endpoints.liquibase.sensitive= *# Mark if the endpoint exposes sensitive information.*

endpoints.logfile.enabled=true *# Enable the endpoint.*

endpoints.logfile.external-file= *# External Logfile to be accessed.*

endpoints.logfile.path=/logfile *# Endpoint URL path.*

endpoints.logfile.sensitive=true *# Enable security on the endpoint.*

endpoints.loggers.enabled=true *# Enable the endpoint.*

endpoints.loggers.id= *# Endpoint identifier.*

endpoints.loggers.path=/logfile *# Endpoint path.*

endpoints.loggers.sensitive=true *# Mark if the endpoint exposes sensitive information.*

endpoints.mappings.enabled= *# Enable the endpoint.*

endpoints.mappings.id= *# Endpoint identifier.*

endpoints.mappings.path= *# Endpoint path.*

endpoints.mappings.sensitive= *# Mark if the endpoint exposes sensitive information.*

endpoints.metrics.enabled= *# Enable the endpoint.*

endpoints.metrics.filter.enabled=true *# Enable the metrics servlet filter.*

endpoints.metrics.filter.gauge-submissions=merged *# Http filter gauge submissions (merged, per-http-method)*

endpoints.metrics.filter.counter-submissions=merged *# Http filter counter submissions (merged, per-http-method)*

endpoints.metrics.id= *# Endpoint identifier.*

endpoints.metrics.path= *# Endpoint path.*

endpoints.metrics.sensitive= *# Mark if the endpoint exposes sensitive information.*

endpoints.shutdown.enabled= *# Enable the endpoint.*

endpoints.shutdown.id= *# Endpoint identifier.*

endpoints.shutdown.path= *# Endpoint path.*

endpoints.shutdown.sensitive= *# Mark if the endpoint exposes sensitive information.*

endpoints.trace.enabled= *# Enable the endpoint.*

endpoints.trace.filter.enabled=true *# Enable the trace servlet filter.*

endpoints.trace.id= *# Endpoint identifier.*

endpoints.trace.path= *# Endpoint path.*

endpoints.trace.sensitive= *# Mark if the endpoint exposes sensitive information.*

*# ENDPOINTS CORS CONFIGURATION (*[EndpointCorsProperties](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot-actuator/src/main/java/org/springframework/boot/actuate/autoconfigure/EndpointCorsProperties.java))

endpoints.cors.allow-credentials= *# Set whether credentials are supported. When not set, credentials are not supported.*

endpoints.cors.allowed-headers= *# Comma-separated list of headers to allow in a request. '\*' allows all headers.*

endpoints.cors.allowed-methods=GET *# Comma-separated list of methods to allow. '\*' allows all methods.*

endpoints.cors.allowed-origins= *# Comma-separated list of origins to allow. '\*' allows all origins. When not set, CORS support is disabled.*

endpoints.cors.exposed-headers= *# Comma-separated list of headers to include in a response.*

endpoints.cors.max-age=1800 *# How long, in seconds, the response from a pre-flight request can be cached by clients.*

*# JMX ENDPOINT (*[EndpointMBeanExportProperties](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot-actuator/src/main/java/org/springframework/boot/actuate/autoconfigure/EndpointMBeanExportProperties.java))

endpoints.jmx.domain= *# JMX domain name. Initialized with the value of 'spring.jmx.default-domain' if set.*

endpoints.jmx.enabled=true *# Enable JMX export of all endpoints.*

endpoints.jmx.static-names= *# Additional static properties to append to all ObjectNames of MBeans representing Endpoints.*

endpoints.jmx.unique-names=false *# Ensure that ObjectNames are modified in case of conflict.*

*# JOLOKIA (*[JolokiaProperties](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot-actuator/src/main/java/org/springframework/boot/actuate/autoconfigure/JolokiaProperties.java))

jolokia.config.\*= *# See Jolokia manual*

*# MANAGEMENT HTTP SERVER (*[ManagementServerProperties](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot-actuator/src/main/java/org/springframework/boot/actuate/autoconfigure/ManagementServerProperties.java))

management.add-application-context-header=true *# Add the "X-Application-Context" HTTP header in each response.*

management.address= *# Network address that the management endpoints should bind to.*

management.context-path= *# Management endpoint context-path. For instance `/actuator`*

management.cloudfoundry.enabled= *# Enable extended Cloud Foundry actuator endpoints*

management.cloudfoundry.skip-ssl-validation= *# Skip SSL verification for Cloud Foundry actuator endpoint security calls*

management.port= *# Management endpoint HTTP port. Uses the same port as the application by default. Configure a different port to use management-specific SSL.*

management.security.enabled=true *# Enable security.*

management.security.roles=ACTUATOR *# Comma-separated list of roles that can access the management endpoint.*

management.security.sessions=stateless *# Session creating policy to use (always, never, if\_required, stateless).*

management.ssl.ciphers= *# Supported SSL ciphers. Requires a custom management.port.*

management.ssl.client-auth= *# Whether client authentication is wanted ("want") or needed ("need"). Requires a trust store. Requires a custom management.port.*

management.ssl.enabled= *# Enable SSL support. Requires a custom management.port.*

management.ssl.enabled-protocols= *# Enabled SSL protocols. Requires a custom management.port.*

management.ssl.key-alias= *# Alias that identifies the key in the key store. Requires a custom management.port.*

management.ssl.key-password= *# Password used to access the key in the key store. Requires a custom management.port.*

management.ssl.key-store= *# Path to the key store that holds the SSL certificate (typically a jks file). Requires a custom management.port.*

management.ssl.key-store-password= *# Password used to access the key store. Requires a custom management.port.*

management.ssl.key-store-provider= *# Provider for the key store. Requires a custom management.port.*

management.ssl.key-store-type= *# Type of the key store. Requires a custom management.port.*

management.ssl.protocol=TLS *# SSL protocol to use. Requires a custom management.port.*

management.ssl.trust-store= *# Trust store that holds SSL certificates. Requires a custom management.port.*

management.ssl.trust-store-password= *# Password used to access the trust store. Requires a custom management.port.*

management.ssl.trust-store-provider= *# Provider for the trust store. Requires a custom management.port.*

management.ssl.trust-store-type= *# Type of the trust store. Requires a custom management.port.*

*# HEALTH INDICATORS*

management.health.db.enabled=true *# Enable database health check.*

management.health.cassandra.enabled=true *# Enable cassandra health check.*

management.health.couchbase.enabled=true *# Enable couchbase health check.*

management.health.defaults.enabled=true *# Enable default health indicators.*

management.health.diskspace.enabled=true *# Enable disk space health check.*

management.health.diskspace.path= *# Path used to compute the available disk space.*

management.health.diskspace.threshold=0 *# Minimum disk space that should be available, in bytes.*

management.health.elasticsearch.enabled=true *# Enable elasticsearch health check.*

management.health.elasticsearch.indices= *# Comma-separated index names.*

management.health.elasticsearch.response-timeout=100 *# The time, in milliseconds, to wait for a response from the cluster.*

management.health.jms.enabled=true *# Enable JMS health check.*

management.health.ldap.enabled=true *# Enable LDAP health check.*

management.health.mail.enabled=true *# Enable Mail health check.*

management.health.mongo.enabled=true *# Enable MongoDB health check.*

management.health.rabbit.enabled=true *# Enable RabbitMQ health check.*

management.health.redis.enabled=true *# Enable Redis health check.*

management.health.solr.enabled=true *# Enable Solr health check.*

management.health.status.order=DOWN, OUT\_OF\_SERVICE, UP, UNKNOWN *# Comma-separated list of health statuses in order of severity.*

*# INFO CONTRIBUTORS (*[InfoContributorProperties](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot-actuator/src/main/java/org/springframework/boot/actuate/autoconfigure/InfoContributorProperties.java))

management.info.build.enabled=true *# Enable build info.*

management.info.defaults.enabled=true *# Enable default info contributors.*

management.info.env.enabled=true *# Enable environment info.*

management.info.git.enabled=true *# Enable git info.*

management.info.git.mode=simple *# Mode to use to expose git information.*

*# REMOTE SHELL (*[ShellProperties](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot-actuator/src/main/java/org/springframework/boot/actuate/autoconfigure/ShellProperties.java))

management.shell.auth.type=simple *# Authentication type. Auto-detected according to the environment.*

management.shell.auth.jaas.domain=my-domain *# JAAS domain.*

management.shell.auth.key.path= *# Path to the authentication key. This should point to a valid ".pem" file.*

management.shell.auth.simple.user.name=user *# Login user.*

management.shell.auth.simple.user.password= *# Login password.*

management.shell.auth.spring.roles=ACTUATOR *# Comma-separated list of required roles to login to the CRaSH console.*

management.shell.command-path-patterns=classpath\*:/commands/\*\*,classpath\*:/crash/commands/\*\* *# Patterns to use to look for commands.*

management.shell.command-refresh-interval=-1 *# Scan for changes and update the command if necessary (in seconds).*

management.shell.config-path-patterns=classpath\*:/crash/\* *# Patterns to use to look for configurations.*

management.shell.disabled-commands=jpa\*,jdbc\*,jndi\* *# Comma-separated list of commands to disable.*

management.shell.disabled-plugins= *# Comma-separated list of plugins to disable. Certain plugins are disabled by default based on the environment.*

management.shell.ssh.auth-timeout = *# Number of milliseconds after user will be prompted to login again.*

management.shell.ssh.enabled=true *# Enable CRaSH SSH support.*

management.shell.ssh.idle-timeout = *# Number of milliseconds after which unused connections are closed.*

management.shell.ssh.key-path= *# Path to the SSH server key.*

management.shell.ssh.port=2000 *# SSH port.*

management.shell.telnet.enabled=false *# Enable CRaSH telnet support. Enabled by default if the TelnetPlugin is available.*

management.shell.telnet.port=5000 *# Telnet port.*

*# TRACING (*[TraceProperties](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot-actuator/src/main/java/org/springframework/boot/actuate/trace/TraceProperties.java))

management.trace.include=request-headers,response-headers,cookies,errors *# Items to be included in the trace.*

*# METRICS EXPORT (*[MetricExportProperties](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot-actuator/src/main/java/org/springframework/boot/actuate/metrics/export/MetricExportProperties.java))

spring.metrics.export.aggregate.key-pattern= *# Pattern that tells the aggregator what to do with the keys from the source repository.*

spring.metrics.export.aggregate.prefix= *# Prefix for global repository if active.*

spring.metrics.export.delay-millis=5000 *# Delay in milliseconds between export ticks. Metrics are exported to external sources on a schedule with this delay.*

spring.metrics.export.enabled=true *# Flag to enable metric export (assuming a MetricWriter is available).*

spring.metrics.export.excludes= *# List of patterns for metric names to exclude. Applied after the includes.*

spring.metrics.export.includes= *# List of patterns for metric names to include.*

spring.metrics.export.redis.key=keys.spring.metrics *# Key for redis repository export (if active).*

spring.metrics.export.redis.prefix=spring.metrics *# Prefix for redis repository if active.*

spring.metrics.export.send-latest= *# Flag to switch off any available optimizations based on not exporting unchanged metric values.*

spring.metrics.export.statsd.host= *# Host of a statsd server to receive exported metrics.*

spring.metrics.export.statsd.port=8125 *# Port of a statsd server to receive exported metrics.*

spring.metrics.export.statsd.prefix= *# Prefix for statsd exported metrics.*

spring.metrics.export.triggers.\*= *# Specific trigger properties per MetricWriter bean name.*

*# ----------------------------------------*

*# DEVTOOLS PROPERTIES*

*# ----------------------------------------*

*# DEVTOOLS (*[DevToolsProperties](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot-devtools/src/main/java/org/springframework/boot/devtools/autoconfigure/DevToolsProperties.java))

spring.devtools.livereload.enabled=true *# Enable a livereload.com compatible server.*

spring.devtools.livereload.port=35729 *# Server port.*

spring.devtools.restart.additional-exclude= *# Additional patterns that should be excluded from triggering a full restart.*

spring.devtools.restart.additional-paths= *# Additional paths to watch for changes.*

spring.devtools.restart.enabled=true *# Enable automatic restart.*

spring.devtools.restart.exclude=META-INF/maven/\*\*,META-INF/resources/\*\*,resources/\*\*,static/\*\*,public/\*\*,templates/\*\*,\*\*/\*Test.class,\*\*/\*Tests.class,git.properties *# Patterns that should be excluded from triggering a full restart.*

spring.devtools.restart.poll-interval=1000 *# Amount of time (in milliseconds) to wait between polling for classpath changes.*

spring.devtools.restart.quiet-period=400 *# Amount of quiet time (in milliseconds) required without any classpath changes before a restart is triggered.*

spring.devtools.restart.trigger-file= *# Name of a specific file that when changed will trigger the restart check. If not specified any classpath file change will trigger the restart.*

*# REMOTE DEVTOOLS (*[RemoteDevToolsProperties](https://github.com/spring-projects/spring-boot/tree/v1.5.7.RELEASE/spring-boot-devtools/src/main/java/org/springframework/boot/devtools/autoconfigure/RemoteDevToolsProperties.java))

spring.devtools.remote.context-path=/.~~spring-boot!~ *# Context path used to handle the remote connection.*

spring.devtools.remote.debug.enabled=true *# Enable remote debug support.*

spring.devtools.remote.debug.local-port=8000 *# Local remote debug server port.*

spring.devtools.remote.proxy.host= *# The host of the proxy to use to connect to the remote application.*

spring.devtools.remote.proxy.port= *# The port of the proxy to use to connect to the remote application.*

spring.devtools.remote.restart.enabled=true *# Enable remote restart.*

spring.devtools.remote.secret= *# A shared secret required to establish a connection (required to enable remote support).*

spring.devtools.remote.secret-header-name=X-AUTH-TOKEN *# HTTP header used to transfer the shared secret.*

*# ----------------------------------------*

*# TESTING PROPERTIES*

*# ----------------------------------------*

spring.test.database.replace=any *# Type of existing DataSource to replace.*

spring.test.mockmvc.print=default *# MVC Print option.*

## Appendix B. Configuration meta-data

Spring Boot jars are shipped with meta-data files that provide details of all supported configuration properties. The files are designed to allow IDE developers to offer contextual help and “code completion” as users are working with application.properties or application.yml files.

The majority of the meta-data file is generated automatically at compile time by processing all items annotated with @ConfigurationProperties. However, it is possible to [write part of the meta-data manually](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#configuration-metadata-additional-metadata) for corner cases or more advanced use cases.

## B.1 Meta-data format

Configuration meta-data files are located inside jars under META-INF/spring-configuration-metadata.json They use a simple JSON format with items categorized under either “groups” or “properties” and additional values hint categorized under "hints":

{"groups": **[**

**{**

"name": "server"**,**

"type": "org.springframework.boot.autoconfigure.web.ServerProperties"**,**

"sourceType": "org.springframework.boot.autoconfigure.web.ServerProperties"

**},**

**{**

"name": "spring.jpa.hibernate"**,**

"type": "org.springframework.boot.autoconfigure.orm.jpa.JpaProperties$Hibernate"**,**

"sourceType": "org.springframework.boot.autoconfigure.orm.jpa.JpaProperties"**,**

"sourceMethod": "getHibernate()"

**}**

...

],"properties": **[**

**{**

"name": "server.port"**,**

"type": "java.lang.Integer"**,**

"sourceType": "org.springframework.boot.autoconfigure.web.ServerProperties"

**},**

**{**

"name": "server.servlet-path"**,**

"type": "java.lang.String"**,**

"sourceType": "org.springframework.boot.autoconfigure.web.ServerProperties"**,**

"defaultValue": "/"

**},**

**{**

"name": "spring.jpa.hibernate.ddl-auto"**,**

"type": "java.lang.String"**,**

"description": "DDL mode. This is actually a shortcut for the \"hibernate.hbm2ddl.auto\" property."**,**

"sourceType": "org.springframework.boot.autoconfigure.orm.jpa.JpaProperties$Hibernate"

**}**

...

],"hints": **[**

**{**

"name": "spring.jpa.hibernate.ddl-auto"**,**

"values": **[**

**{**

"value": "none"**,**

"description": "Disable DDL handling."

**},**

**{**

"value": "validate"**,**

"description": "Validate the schema, make no changes to the database."

**},**

**{**

"value": "update"**,**

"description": "Update the schema if necessary."

**},**

**{**

"value": "create"**,**

"description": "Create the schema and destroy previous data."

**},**

**{**

"value": "create-drop"**,**

"description": "Create and then destroy the schema at the end of the session."

**}**

**]**

**}**

]**}**

Each “property” is a configuration item that the user specifies with a given value. For example server.port and server.servlet-path might be specified inapplication.properties as follows:

server.port=9090

server.servlet-path=/home

The “groups” are higher level items that don’t themselves specify a value, but instead provide a contextual grouping for properties. For example the server.port andserver.servlet-path properties are part of the server group.

|  |
| --- |
| [Note] |
| It is not required that every “property” has a “group”, some properties might just exist in their own right. |

Finally, “hints” are additional information used to assist the user in configuring a given property. When configuring the spring.jpa.hibernate.ddl-auto property, a tool can use it to offer some auto-completion help for the none, validate, update, create and create-drop values.

### B.1.1 Group Attributes

The JSON object contained in the groups array can contain the following attributes:

| **Name** | **Type** | **Purpose** |
| --- | --- | --- |
| name | String | The full name of the group. This attribute is mandatory. |
| type | String | The class name of the data type of the group. For example, if the group was based on a class annotated with @ConfigurationProperties the attribute would contain the fully qualified name of that class. If it was based on a @Bean method, it would be the return type of that method. The attribute may be omitted if the type is not known. |
| description | String | A short description of the group that can be displayed to users. May be omitted if no description is available. It is recommended that descriptions are a short paragraphs, with the first line providing a concise summary. The last line in the description should end with a period (.). |
| sourceType | String | The class name of the source that contributed this group. For example, if the group was based on a @Bean method annotated with @ConfigurationProperties this attribute would contain the fully qualified name of the @Configuration class containing the method. The attribute may be omitted if the source type is not known. |
| sourceMethod | String | The full name of the method (include parenthesis and argument types) that contributed this group. For example, the name of a @ConfigurationProperties annotated @Bean method. May be omitted if the source method is not known. |

### B.1.2 Property Attributes

The JSON object contained in the properties array can contain the following attributes:

| **Name** | **Type** | **Purpose** |
| --- | --- | --- |
| name | String | The full name of the property. Names are in lowercase dashed form (e.g. server.servlet-path). This attribute is mandatory. |
| type | String | The full signature of the data type of the property. For example, java.lang.String but also a full generic type such as java.util.Map<java.util.String,acme.MyEnum>. This attribute can be used to guide the user as to the types of values that they can enter. For consistency, the type of a primitive is specified using its wrapper counterpart, i.e. boolean becomes java.lang.Boolean. Note that this class may be a complex type that gets converted from a String as values are bound. May be omitted if the type is not known. |
| description | String | A short description of the group that can be displayed to users. May be omitted if no description is available. It is recommended that descriptions are a short paragraphs, with the first line providing a concise summary. The last line in the description should end with a period (.). |
| sourceType | String | The class name of the source that contributed this property. For example, if the property was from a class annotated with @ConfigurationProperties this attribute would contain the fully qualified name of that class. May be omitted if the source type is not known. |
| defaultValue | Object | The default value which will be used if the property is not specified. Can also be an array of value(s) if the type of the property is an array. May be omitted if the default value is not known. |
| deprecation | Deprecation | Specify if the property is deprecated. May be omitted if the field is not deprecated or if that information is not known. See below for more details. |

The JSON object contained in the deprecation attribute of each properties element can contain the following attributes:

| **Name** | **Type** | **Purpose** |
| --- | --- | --- |
| level | String | The level of deprecation, can be either warning (default) or error. When a property has a warning deprecation level it should still be bound in the environment. When it has an error deprecation level however, the property is no longer managed and will not be bound. |
| reason | String | A short description of the reason why the property was deprecated. May be omitted if no reason is available. It is recommended that descriptions are a short paragraphs, with the first line providing a concise summary. The last line in the description should end with a period (.). |
| replacement | String | The full name of the property that is replacing this deprecated property. May be omitted if there is no replacement for this property. |
| [Note] |
| Prior to Spring Boot 1.3, a single deprecated boolean attribute can be used instead of the deprecation element. This is still supported in a deprecated fashion and should no longer be used. If no reason and replacement are available, an empty deprecation object should be set. |

Deprecation can also be specified declaratively in code by adding the @DeprecatedConfigurationProperty annotation to the getter exposing the deprecated property. For instance, let’s assume the app.foo.target property was confusing and was renamed to app.foo.name

*@ConfigurationProperties("app.foo")*

**public** **class** FooProperties {

**private** String name;

**public** String getName() { ... }

**public** **void** setName(String name) { ... }

*@DeprecatedConfigurationProperty(replacement = "app.foo.name")*

*@Deprecated*

**public** String getTarget() {

**return** getName();

}

*@Deprecated*

**public** **void** setTarget(String target) {

setName(target);

}

}

|  |
| --- |
| [Note] |
| There is no way to set a level as warning is always assumed since code is still handling the property. |

The code above makes sure that the deprecated property still works (delegating to the name property behind the scenes). Once the getTarget and setTargetmethods can be removed from your public API, the automatic deprecation hint in the meta-data will go away as well. If you want to keep a hint, adding manual meta-data with an error deprecation level ensures that users are still informed about that property and is particularly useful when a replacement is provided.

### B.1.3 Hint Attributes

The JSON object contained in the hints array can contain the following attributes:

| **Name** | **Type** | **Purpose** |
| --- | --- | --- |
| name | String | The full name of the property that this hint refers to. Names are in lowercase dashed form (e.g. server.servlet-path). If the property refers to a map (e.g. system.contexts) the hint either applies to the keys of the map (system.context.keys) or the values (system.context.values). This attribute is mandatory. |
| values | ValueHint[] | A list of valid values as defined by the ValueHint object (see below). Each entry defines the value and may have a description |
| providers | ValueProvider[] | A list of providers as defined by the ValueProvider object (see below). Each entry defines the name of the provider and its parameters, if any. |

The JSON object contained in the values attribute of each hint element can contain the following attributes:

| **Name** | **Type** | **Purpose** |
| --- | --- | --- |
| value | Object | A valid value for the element to which the hint refers to. Can also be an array of value(s) if the type of the property is an array. This attribute is mandatory. |
| description | String | A short description of the value that can be displayed to users. May be omitted if no description is available. It is recommended that descriptions are a short paragraphs, with the first line providing a concise summary. The last line in the description should end with a period (.). |

The JSON object contained in the providers attribute of each hint element can contain the following attributes:

| **Name** | **Type** | **Purpose** |
| --- | --- | --- |
| name | String | The name of the provider to use to offer additional content assistance for the element to which the hint refers to. |
| parameters | JSON object | Any additional parameter that the provider supports (check the documentation of the provider for more details). |

### B.1.4 Repeated meta-data items

It is perfectly acceptable for “property” and “group” objects with the same name to appear multiple times within a meta-data file. For example, you could bind two separate classes to the same prefix, with each potentially offering overlap of property names. While this is not supposed to be a frequent scenario, consumers of meta-data should take care to ensure that they support such scenarios.

## B.2 Providing manual hints

To improve the user experience and further assist the user in configuring a given property, you can provide additional meta-data that:

1. Describes the list of potential values for a property.
2. Associates a provider to attach a well-defined semantic to a property so that a tool can discover the list of potential values based on the project’s context.

### B.2.1 Value hint

The name attribute of each hint refers to the name of a property. In the initial example above, we provide 5 values for the spring.jpa.hibernate.ddl-auto property:none, validate, update, create and create-drop. Each value may have a description as well.

If your property is of type Map, you can provide hints for both the keys and the values (but not for the map itself). The special .keys and .values suffixes must be used to refer to the keys and the values respectively.

Let’s assume a foo.contexts that maps magic String values to an integer:

*@ConfigurationProperties("foo")*

**public** **class** FooProperties {

**private** Map<String,Integer> contexts;

*// getters and setters*

}

The magic values are foo and bar for instance. In order to offer additional content assistance for the keys, you could add the following to [the manual meta-data of the module](https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#configuration-metadata-additional-metadata):

{"hints": **[**

**{**

"name": "foo.contexts.keys"**,**

"values": **[**

**{**

"value": "foo"

**},**

**{**

"value": "bar"

**}**

**]**

**}**

]**}**

|  |
| --- |
| [Note] |
| Of course, you should have an Enum for those two values instead. This is by far the most effective approach to auto-completion if your IDE supports it. |

### B.2.2 Value provider

Providers are a powerful way of attaching semantics to a property. We define in the section below the official providers that you can use for your own hints. Bare in mind however that your favorite IDE may implement some of these or none of them. It could eventually provide its own as well.

|  |
| --- |
| [Note] |
| As this is a new feature, IDE vendors will have to catch up with this new feature. |

The table below summarizes the list of supported providers:

| **Name** | **Description** |
| --- | --- |
| any | Permit any additional value to be provided. |
| class-reference | Auto-complete the classes available in the project. Usually constrained by a base class that is specified via the target parameter. |
| handle-as | Handle the property as if it was defined by the type defined via the mandatory target parameter. |
| logger-name | Auto-complete valid logger names. Typically, package and class names available in the current project can be auto-completed. |
| spring-bean-reference | Auto-complete the available bean names in the current project. Usually constrained by a base class that is specified via the targetparameter. |
| spring-profile-name | Auto-complete the available Spring profile names in the project. |
| [Tip] |
| No more than one provider can be active for a given property but you can specify several providers if they can all manage the property in some ways. Make sure to place the most powerful provider first as the IDE must use the first one in the JSON section it can handle. If no provider for a given property is supported, no special content assistance is provided either. |

#### Any

The **any** provider permits any additional values to be provided. Regular value validation based on the property type should be applied if this is supported.

This provider will be typically used if you have a list of values and any extra values are still to be considered as valid.

The example below offers on and off as auto-completion values for system.state; any other value is also allowed:

{"hints": **[**

**{**

"name": "system.state"**,**

"values": **[**

**{**

"value": "on"

**},**

**{**

"value": "off"

**}**

]**,**

"providers": **[**

**{**

"name": "any"

**}**

**]**

**}**

]**}**

#### Class reference

The **class-reference** provider auto-completes classes available in the project. This provider supports these parameters:

| **Parameter** | **Type** | **Default value** | **Description** |
| --- | --- | --- | --- |
| target | String(Class) | none | The fully qualified name of the class that should be assignable to the chosen value. Typically used to filter out non candidate classes. Note that this information can be provided by the type itself by exposing a class with the appropriate upper bound. |
| concrete | boolean | true | Specify if only concrete classes are to be considered as valid candidates. |

The meta-data snippet below corresponds to the standard server.jsp-servlet.class-name property that defines the JspServlet class name to use:

{"hints": **[**

**{**

"name": "server.jsp-servlet.class-name"**,**

"providers": **[**

**{**

"name": "class-reference"**,**

"parameters": **{**

"target": "javax.servlet.http.HttpServlet"

**}**

**}**

**]**

**}**

]**}**

#### Handle As

The **handle-as** provider allows you to substitute the type of the property to a more high-level type. This typically happens when the property has a java.lang.Stringtype because you don’t want your configuration classes to rely on classes that may not be on the classpath. This provider supports these parameters:

| **Parameter** | **Type** | **Default value** | **Description** |
| --- | --- | --- | --- |
| **target** | String (Class) | none | The fully qualified name of the type to consider for the property. This parameter is mandatory. |

The following types can be used:

* Any java.lang.Enum that lists the possible values for the property (By all means, try to define the property with the Enum type instead as no further hint should be required for the IDE to auto-complete the values).
* java.nio.charset.Charset: auto-completion of charset/encoding values (e.g. UTF-8)
* java.util.Locale: auto-completion of locales (e.g. en\_US)
* org.springframework.util.MimeType: auto-completion of content type values (e.g. text/plain)
* org.springframework.core.io.Resource: auto-completion of Spring’s Resource abstraction to refer to a file on the filesystem or on the classpath. (e.g. classpath:/foo.properties)

|  |
| --- |
| [Note] |
| If multiple values can be provided, use a Collection or Array type to teach the IDE about it. |

The meta-data snippet below corresponds to the standard liquibase.change-log property that defines the path to the changelog to use. It is actually used internally as a org.springframework.core.io.Resource but cannot be exposed as such as we need to keep the original String value to pass it to the Liquibase API.

{"hints": **[**

**{**

"name": "liquibase.change-log"**,**

"providers": **[**

**{**

"name": "handle-as"**,**

"parameters": **{**

"target": "org.springframework.core.io.Resource"

**}**

**}**

**]**

**}**

]**}**

#### Logger name

The **logger-name** provider auto-completes valid logger names. Typically, package and class names available in the current project can be auto-completed. Specific frameworks may have extra magic logger names that could be supported as well.

Since a logger name can be any arbitrary name, really, this provider should allow any value but could highlight valid packages and class names that are not available in the project’s classpath.

The meta-data snippet below corresponds to the standard logging.level property, keys are logger names and values correspond to the standard log levels or any custom level:

{"hints": **[**

**{**

"name": "logging.level.keys"**,**

"values": **[**

**{**

"value": "root"**,**

"description": "Root logger used to assign the default logging level."

**}**

]**,**

"providers": **[**

**{**

"name": "logger-name"

**}**

**]**

**},**

**{**

"name": "logging.level.values"**,**

"values": **[**

**{**

"value": "trace"

**},**

**{**

"value": "debug"

**},**

**{**

"value": "info"

**},**

**{**

"value": "warn"

**},**

**{**

"value": "error"

**},**

**{**

"value": "fatal"

**},**

**{**

"value": "off"

**}**

]**,**

"providers": **[**

**{**

"name": "any"

**}**

**]**

**}**

]**}**

#### Spring bean reference

The **spring-bean-reference** provider auto-completes the beans that are defined in the configuration of the current project. This provider supports these parameters:

| **Parameter** | **Type** | **Default value** | **Description** |
| --- | --- | --- | --- |
| target | String(Class) | none | The fully qualified name of the bean class that should be assignable to the candidate. Typically used to filter out non candidate beans. |

The meta-data snippet below corresponds to the standard spring.jmx.server property that defines the name of the MBeanServer bean to use:

{"hints": **[**

**{**

"name": "spring.jmx.server"**,**

"providers": **[**

**{**

"name": "spring-bean-reference"**,**

"parameters": **{**

"target": "javax.management.MBeanServer"

**}**

**}**

**]**

**}**

]**}**

|  |
| --- |
| [Note] |
| The binder is not aware of the meta-data so if you provide that hint, you will still need to transform the bean name into an actual Bean reference using the ApplicationContext. |

#### Spring profile name

The **spring-profile-name** provider auto-completes the Spring profiles that are defined in the configuration of the current project.

The meta-data snippet below corresponds to the standard spring.profiles.active property that defines the name of the Spring profile(s) to enable:

{"hints": **[**

**{**

"name": "spring.profiles.active"**,**

"providers": **[**

**{**

"name": "spring-profile-name"

**}**

**]**

**}**

]**}**

## B.3 Generating your own meta-data using the annotation processor

You can easily generate your own configuration meta-data file from items annotated with @ConfigurationProperties by using the spring-boot-configuration-processor jar. The jar includes a Java annotation processor which is invoked as your project is compiled. To use the processor, simply include spring-boot-configuration-processor as an optional dependency, for example with Maven you would add:

<dependency>

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

<artifactId>spring-boot-configuration-processor</artifactId>

<optional>true</optional>

</dependency>

With Gradle, you can use the [propdeps-plugin](https://github.com/spring-gradle-plugins/propdeps-plugin) and specify:

dependencies {

optional "org.springframework.boot:spring-boot-configuration-processor"

}

compileJava.dependsOn(processResources)

|  |
| --- |
| [Note] |
| You need to add compileJava.dependsOn(processResources) to your build to ensure that resources are processed before code is compiled. Without this directive any additional-spring-configuration-metadata.json files will not be processed. |

The processor will pick up both classes and methods that are annotated with @ConfigurationProperties. The Javadoc for field values within configuration classes will be used to populate the description attribute.

|  |
| --- |
| [Note] |
| You should only use simple text with @ConfigurationProperties field Javadoc since they are not processed before being added to the JSON. |

Properties are discovered via the presence of standard getters and setters with special handling for collection types (that will be detected even if only a getter is present). The annotation processor also supports the use of the @Data, @Getter and @Setter lombok annotations.

|  |
| --- |
| [Note] |
| If you are using AspectJ in your project, you need to make sure that the annotation processor only runs once. There are several ways to do this: with Maven, you can configure the maven-apt-plugin explicitly and add the dependency to the annotation processor only there. You could also let the AspectJ plugin run all the processing and disable annotation processing in the maven-compiler-plugin configuration:  <plugin>  <groupId>org.apache.maven.plugins</groupId>  <artifactId>maven-compiler-plugin</artifactId>  <configuration>  <proc>none</proc>  </configuration>  </plugin> |

### B.3.1 Nested properties

The annotation processor will automatically consider inner classes as nested properties. For example, the following class:

*@ConfigurationProperties(prefix="server")*

**public** **class** ServerProperties {

**private** String name;

**private** Host host;

*// ... getter and setters*

**private** **static** **class** Host {

**private** String ip;

**private** **int** port;

*// ... getter and setters*

}

}

Will produce meta-data information for server.name, server.host.ip and server.host.port properties. You can use the @NestedConfigurationPropertyannotation on a field to indicate that a regular (non-inner) class should be treated as if it were nested.

|  |
| --- |
| [Tip] |
| This has no effect on collections and maps as those types are automatically identified and a single meta-data property is generated for each of them. |

### B.3.2 Adding additional meta-data

Spring Boot’s configuration file handling is quite flexible; and it is often the case that properties may exist that are not bound to a @ConfigurationProperties bean. You may also need to tune some attributes of an existing key. To support such cases and allow you to provide custom "hints", the annotation processor will automatically merge items from META-INF/additional-spring-configuration-metadata.json into the main meta-data file.

If you refer to a property that has been detected automatically, the description, default value and deprecation information are overridden if specified. If the manual property declaration is not identified in the current module, it is added as a brand new property.

The format of the additional-spring-configuration-metadata.json file is exactly the same as the regular spring-configuration-metadata.json. The additional properties file is optional, if you don’t have any additional properties, simply don’t add it.