Configuring Logback with Spring Boot

Logback is provided out of the box with Spring Boot when you use one of the Spring Boot starter dependencies as they include spring-boot-starter-logging providing logging without any configuration and can be altered to work differently if required. There are two ways of providing your own configuration, if you only need simpler alterations they can be added to a properties file such as application.properties or for more complex needs you can use XML or Groovy to specify your settings. In this tutorial we will focus on using XML to define custom logging configuration and look at some of the basics of doing so, as well as a brief look at using property files to specify simple alterations to the standard setup provided by Spring Boot.

In this post I have used the dependency spring-boot-starter to pull in spring-boot-starter-logging which can be found below.

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter</artifactId>
</dependency>
```

This will make use of spring-boot-starter-logging which in turn has dependencies on.

```
<dependencies>
  <dependency>
    <groupId>ch.qos.logback</groupId>
    <artifactId>logback-classic</artifactId>
  </dependency>
  <dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>jul-to-slf4j</artifactId>
  </dependency>
  <dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>log4j-over-slf4j</artifactId>
  </dependency>
</dependencies>
```

logback-classic contains the logback-core dependency and between them they contain everything we need to get started. Spring Boot logging guide mentions that a dependency on jcl-over-slf4j is required but this is missing when using spring-boot-starter-parent at version 2.0.0.M3 so I assume some magic has been done somewhere to remove this dependency. Although it does exist if you are using (the current) version 1.5.6.RELEASE.

Before we start looking at configuring Logback its worth having a quick look through how to send a message to the log from within a class.

```
@Service
public class MyServiceImpl implements MyService {

  private static final Logger LOGGER =
LoggerFactory.getLogger(MyServiceImpl.class);

  @Override
  public void doStuff(final String value) {
    LOGGER.trace("doStuff needed more information - {}", value);
    LOGGER.debug("doStuff needed to debug - {}", value);
    LOGGER.info("doStuff took input - {}", value);
    LOGGER.warn("doStuff needed to warn - {}", value);
    LOGGER.error("doStuff encountered an error with value - {}", value);
  }
}
```

The LOGGER allows messages to be written to the log using the methods which represent each logging level, trace, debug, info, warn, error followed be the message. The braces / curly brackets will be replaced by the value passed in as a method parameter.

Now we can start looking at configuring Logback itself by starting with a relatively simple example. Below is the logback.xml file that is one of the files that Logback will search for to configure it's settings.

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

  <appender name="STDOUT"
class="ch.qos.logback.core.ConsoleAppender">
    <encoder>
      <pattern>
        %d{dd-MM-yyyy HH:mm:ss.SSS} %magenta([%thread])
%highlight(%-5level) %logger{36}.%M - %msg%n
```

```
      </pattern>
    </encoder>
  </appender>

  <root level="info">
    <appender-ref ref="STDOUT"/>
  </root>

</configuration>
```
It creates an appender of class ConsoleAppender which will output log messages to the console like System.out.print normally would. A pattern is set that the log messages will adhere to which come provided with some notations that are replaced with generated values depending on message that has been sent to the logger. Some notations have been included in the example and below are explanations of what each do.

- %d - outputs the time which the log message occurred in formats that SimpleDateFormat allows.
- %thread - outputs the name of the thread that the log message occurred in.
- $-5level - outputs the logging level of the log message.
- %logger{36} - outputs the package + class name the log message occurred in. The number inside the brackets represents the maximum length of the package + class name. If the output is longer than the specified length it will take a substring of the first character of each individual package starting from the root package until the output is below the maximum length. The class name will never be reduced. A nice diagram of this can be found in the Conversion word docs.
- %M - outputs the name of the method that the log message occurred in (apparently this is quite slow to use and not recommended unless your not worried about performance or the method name is particularly important to you).
- %msg - outputs the actual log message.
- %n - line break
- %magenta() - sets the colour of the output contained in the brackets to magenta (other colours are available).
- highlight() - sets the colour of the output contained in the brackets to the depending on the logging level (for example ERROR = red).

The appender that was created is then referenced in the root logger. In the above example the logging level has been set to INFO (lowercase or uppercase can be used). Causing it to only output messages that are defined at log level INFO or above (INFO, WARN, ERROR).

The available logging levels in Logback are:

- OFF (output no logs)
- ERROR
- WARN
- INFO
- DEBUG
- TRACE

Returning to the snippet shown above with the logging level of INFO only messages of level INFO or above (WARN and ERROR) are output to the log. So if we called MyService.doStuff("value") it would generate the following (spring related logs have been removed from this and all following output examples).

```
28-08-2017 13:32:18.549 [main] INFO
com.lankydan.service.MyServiceImpl.doStuff - doStuff took input - value
28-08-2017 13:32:18.549 [main] WARN
com.lankydan.service.MyServiceImpl.doStuff - doStuff needed to warn -
value
28-08-2017 13:32:18.549 [main] ERROR
com.lankydan.service.MyServiceImpl.doStuff - doStuff encountered an
error with value - value
```

Notice how even though TRACE and DEBUG level messages were sent to the logger they were not displayed as they are below INFO's level.

If you wanted to write the equivalent of previous code example from within application.properties you could do so as follows.

```
logging.level.root=info
logging.pattern.console=%d{dd-MM-yyyy HH:mm:ss.SSS}
%magenta([%thread]) %highlight(%-5level) %logger.%M - %msg%n
```

When done in this form a logback.xml file is not required and as you can see the configuration is quite a bit shorter and useful for simpler setups.

When using Spring Boot, a default configuration for Logback is provided which is overridden when you add your own logback.xml. If you wish to include Spring Boot's configuration you can add the below inside the <configuration> tags.

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

See [Spring Boot docs - Configure Logback for logging]() for more information on this.

If you want to log messages of class at a different level to the root level then you can define your own logger for the class. This will allow you to set the logging level for that particular class as well as specify other properties that are unique to that class. Below is how you would define a logger for a single class.

```
<logger name="com.lankydan.service.MyServiceImpl" level="debug">
  <appender-ref ref="STDOUT" />
</logger>
```

If you then went on to run this piece of code, with the root logger still defined it will generate the output of.

```
27-08-2017 17:02:10.248 [main] DEBUG
com.lankydan.service.MyServiceImpl.doStuff - doStuff needed to debug
- value
27-08-2017 17:02:10.248 [main] DEBUG
com.lankydan.service.MyServiceImpl.doStuff - doStuff needed to debug
- value
27-08-2017 17:02:10.248 [main] INFO
com.lankydan.service.MyServiceImpl.doStuff - doStuff took input - value
27-08-2017 17:02:10.248 [main] INFO
com.lankydan.service.MyServiceImpl.doStuff - doStuff took input - value
27-08-2017 17:02:10.248 [main] WARN
com.lankydan.service.MyServiceImpl.doStuff - doStuff needed to warn -
value
27-08-2017 17:02:10.248 [main] WARN
com.lankydan.service.MyServiceImpl.doStuff - doStuff needed to warn -
value
27-08-2017 17:02:10.248 [main] ERROR
com.lankydan.service.MyServiceImpl.doStuff - doStuff encountered an
error with value - value
27-08-2017 17:02:10.248 [main] ERROR
com.lankydan.service.MyServiceImpl.doStuff - doStuff encountered an
error with value - value
```

As you can see each log message has been generated twice, which is probably not what you want. To fix this additivity="false needs to be used. Not using additivity="false will cause the message to be printed out twice due to the root log appender and the class level appender both writing to the log. Even if the root level is ERROR by setting the class level to DEBUG it overwrites it globally and will cause the root appender to also write to DEBUG level for the MyServiceImpl class. Below is what the code should look like with this property included.

```
<logger name="com.lankydan.service.MyServiceImpl" additivity="false"
level="debug">
  <appender-ref ref="STDOUT" />
</logger>
```

Another possible solution is to only set the log level for the class without writing to the log (due to no appender defined), this is equivalent to the version above but makes the assumption that another log appender (in this case the root appender) is writing to the log for it to work

```
<logger name="com.lankydan.service.MyServiceImpl" level="debug"/>
```

If either of these solutions are used the output returns to what is expected.

```
27-08-2017 16:30:47.818 [main] DEBUG
com.lankydan.service.MyServiceImpl.doStuff - doStuff needed to debug
- value
27-08-2017 16:30:47.834 [main] INFO
com.lankydan.service.MyServiceImpl.doStuff - doStuff took input - value
27-08-2017 16:30:47.834 [main] WARN
com.lankydan.service.MyServiceImpl.doStuff - doStuff needed to warn -
value
27-08-2017 16:30:47.834 [main] ERROR
com.lankydan.service.MyServiceImpl.doStuff - doStuff encountered an
error with value - value
```

Class level logging can be written in application.properties by adding the following.

```
logging.level.com.lankydan.service.MyServiceImpl=debug
```

Package level logging can also be defined by simply using the package name instead of the class name in the logger tag.

```
<logger name="com.lankydan.service" additivity="false" level="debug">
  <appender-ref ref="STDOUT" />
</logger>
```

More proof can be found by adding logging to one of the springframework packages and then moving onto one of the classes instead. For example

```
<logger name="org.springframework.boot" level="debug">
  <appender-ref ref="STDOUT" />
```

```
</logger>
```
compared to

```
<logger name="org.springframework.boot.SpringApplication"
level="debug">
  <appender-ref ref="STDOUT" />
</logger>
```
Prints out a completely different amount of log lines. Maybe hundreds vs one or two lines, with the SpringApplication logs being contained inside the org.springframework.boot logs.

Package level logging in application.properties follows the same format of using the package instead of the class name.

```
logging.level.com.lankydan.service=debug
```
Properties can be defined allowing them to be reused through the configuration file, which is handy when you need to mark an output folder for the logs to go to.

```
<property name="LOG_PATH" value="logs"/>
```
This property named LOG_PATH is used in further examples and will use the directory DEV_HOME/logs where DEV_HOME is the root directory of your project (at least this was the case for mine...). This probably isn't the best place to save the logs to in reality but for the needs of this tutorial it is suitable. LOG_PATH is a property that has importance to the default Spring Boot logging setup but a property of any name can be created. The value of LOG_PATH can then be accessed throughout the rest of the configuration by adding ${LOG_PATH}.

This configuration can be achieved through application.properties as LOG_PATH has importance within Spring Boot.

```
logging.path=logs
```
This also works when you define your own property / variable, allowing you to reference it from within the rest of your code. For example.

```
propertyA=value
propertyB=${propertyA} # extra configuration if required
```
${propertyA} will be replaced by the value of propertyA allowing propertyB to make use of it.

To save to the logs to file FileAppender can be used. This is a simple file appender and will save all the logs to a singular file which could become very large so you are more likely to use the RollingFileAppender that we will take a look at later on.

```
<appender name="SAVE-TO-FILE"
class="ch.qos.logback.core.FileAppender">

  <file>${LOG_PATH}/log.log</file>

  <encoder
class="ch.qos.logback.classic.encoder.PatternLayoutEncoder">
    <Pattern>
      %d{dd-MM-yyyy HH:mm:ss.SSS} [%thread] %-5level %logger{36}.
%M - %msg%n
    </Pattern>
  </encoder>

</appender>
```

There isn't much to it, it follows the same sort of structure to the ConsoleAppender with the addition to naming a file that the log messages are saved to. It is worth noting that I have removed the colours that were added to the encoder pattern when saving to file as it will include characters that are not meant to be displayed and will clutter the log file. This appender can then be referenced in the same way as the STDOUT appender shown earlier allowing it to be actually be used. This will be shown below and following code snippets will use the same code.

```
<logger name="com.lankydan.service.MyServiceImpl" additivity="false"
level="debug">
  <appender-ref ref="SAVE-TO-FILE"/>
</logger>
```

Following on from the previous application.properties snippet where the logging.path was set, which actually causes the logs to be output to file (as well as the console) if other settings haven't been played around with to much. Therefore you could stop there, but the pattern written to the file and the name of the file are not under your control if done this way. The example below will demonstrate a similar configuration as the SAVE-TO-FILE appender shown above.

```
logging.pattern.console=
logging.path=logs
logging.file=${logging.path}/log.log
logging.pattern.file=%d{dd-MM-yyyy HH:mm:ss.SSS} [%thread] %-5level
```

%logger{36}.%M - %msg%n
Where this varies from the XML configuration is that the example shows the appender being referenced in the logger for MyServiceImpl but the above application.properties snippet will also include the root logger and therefore output all log messages to file. The logging.pattern.console has been added to stop it from outputting to console to keep it in line with the XML code above (this doesn't seem to be a nice way to do it but I have not seen another solution).

RollingFileAppender will save the logs to different files depending on their rolling policy. This is handy as it allows the log output to be split out into various forms that you have control over. For example you could separate the log files based on date so you can look at errors that have occurred in the past on particular dates, separate on file size so you don't need to go searching through a massive never ending file or do both and separate by date and size. Below are some code snippets that demonstrate the policies that we just talked about.

TimeBasedRollingPolicy will create a new file based on date. The code below will create a new file each day and append the date to the name of the log file by using the %d notation. The format of the %d notation is important as the rollover time period is inferred from it. The example below will rollover each day, but to rollover monthly instead a different pattern of %d{MM-yyyy} could be used which excludes the day part of the date. Different roll over periods can be used not just daily or monthly due to the period being inferred, as long as the format inside the %d notation coheres to what SimpleDateFormat allows.

```
<appender name="SAVE-TO-FILE"
class="ch.qos.logback.core.rolling.RollingFileAppender">

  <file>${LOG_PATH}/log.log</file>

  <encoder
class="ch.qos.logback.classic.encoder.PatternLayoutEncoder">
    <Pattern>
      %d{dd-MM-yyyy HH:mm:ss.SSS} [%thread] %-5level %logger{36}.
%M - %msg%n
    </Pattern>
  </encoder>

  <rollingPolicy
```

```
class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">
    <fileNamePattern>
      ${LOG_PATH}/archived/log_%d{dd-MM-yyyy}.log
    </fileNamePattern>
    <maxHistory>10</maxHistory>
    <totalSizeCap>100MB</totalSizeCap>
  </rollingPolicy>

</appender>
```

I have included some of the properties that are available to the TimeBasedRollingPolicy in the above example. maxHistory specifies how long the archived log files will be kept before they are automatically deleted. The time they are kept for depends on the rollover time period specified in the file name, so in the above example the rollover period is daily allowing a maximum of 10 days worth of archived logs to be stored before they are deleted. totalSizeCap limits the maximum size of all archived log files, it requires the maxHistory property to be set with maxHistory taking precedence over totalSizeCap when removing archived files.

This configuration is out of the scope of what can be done inside the application.properties file, the same can also be said for the following examples. Although the default configuration will allow the log file to rollover when it reaches 10MB and allows up to 7 archived log files.

To rollover only on file size a rolling policy of FixedWindowRollingPolicy and a triggering policy of SizeBasedTriggeringPolicy need to be used. In the previous example the logs were saved to an archive folder when rolled over, but for this policy I have not saved them as such as the separation of logs is mainly to help make them easier to traverse due to the smaller file sizes.

```
<appender name="SAVE-TO-FILE"
class="ch.qos.logback.core.rolling.RollingFileAppender">

  <file>${LOG_PATH}/log.log</file>

  <encoder
class="ch.qos.logback.classic.encoder.PatternLayoutEncoder">
    <Pattern>
      %d{dd-MM-yyyy HH:mm:ss.SSS} [%thread] %-5level %logger{36}.
%M - %msg%n
```

```
    </Pattern>
  </encoder>

  <rollingPolicy
class="ch.qos.logback.core.rolling.FixedWindowRollingPolicy">
    <fileNamePattern>
      ${LOG_PATH}/log_%i.log
    </fileNamePattern>
    <minIndex>2</minIndex>
    <maxIndex>3</maxIndex>
  </rollingPolicy>

  <triggeringPolicy
class="ch.qos.logback.core.rolling.SizeBasedTriggeringPolicy">
    <maxFileSize>1KB</maxFileSize>
  </triggeringPolicy>

</appender>
```

The optional properties of minIndex and maxIndex found in the FixedWindowRollingPolicy specify minimum and maximum value that %i can take in the log file names. Therefore in the above example when the logs are rolled over they can take the name log_2.log and log_3.log (although starting for 2 is weird and only included for clarity, normally it would start from 1). The process of generating the log files is as follows (using the above code snippet as an example); the log.log file will take all new log inputs and when the maxFileSize is reached log.log is renamed to the archived file log_2.log and a new log.log file is created, when log_2.log has also reached the max size all log files are renamed and shifted along one with a new log.log file being created again. This process will continue if the maxIndex is not set, but when it is the log file with the specified maximum index is deleted (it contains the oldest messages) at the point when another archive file should be created. Following the same example from above this means when log_4.log should be created log_3.log is deleted instead and all the other logs are renamed accordingly.

If you are confused about what I have written above regarding how the files are rolled over, don't worry as even I think after writing that explanation it could be done better. So below I have made a second attempt to illustrate how it works (which hopefully is easier to understand).

log.log maximum file size reached -> log.log renamed to log_2.log and new log.log is created
log_2.log maximum file size reached -> log_2.log renamed to log_3.log, log.log named to log_2.log and new log.log is created log_3.log maximum file size reached -> log_3.log is deleted, log_2.log renamed to log_3.log, log.log renamed to log_2.log and new log.log is created

If I have still done a bad job explaining this process to you then see the FixedWindowRollingPolicy docs which will hopefully get you there if I have failed...

SizeAndTimeBasedRollingPolicy takes parts of both the examples above allowing it to rollover on size and time. Note that it uses both the %d and %i notation for including the date and log number respectively in the file name.

```
<appender name="SAVE-TO-FILE"
class="ch.qos.logback.core.rolling.RollingFileAppender">

    <file>${LOG_PATH}/log.log</file>

    <encoder
class="ch.qos.logback.classic.encoder.PatternLayoutEncoder">
      <Pattern>
        %d{dd-MM-yyyy HH:mm:ss.SSS} [%thread] %-5level %logger{36}.
%M - %msg%n
      </Pattern>
    </encoder>

    <rollingPolicy
class="ch.qos.logback.core.rolling.SizeAndTimeBasedRollingPolicy">
      <fileNamePattern>
        ${LOG_PATH}/archived/log_%d{dd-MM-yyyy}_%i.log
      </fileNamePattern>
      <maxFileSize>10MB</maxFileSize>
      <maxHistory>10</maxHistory>
      <totalSizeCap>100MB</totalSizeCap>
    </rollingPolicy>

</appender>
```

As you can see it contains the maxFileSize, maxHistory and totalSizeCap providing it control over the size of individual files as well as the collection of files. Therefore the above example will keep 10 days worth of history split into files of 10MB and when the total size of all files reaches 100MB the oldest files will be removed.

Now that we have looked at how to define multiple appenders that can output to the console or to file we can combine them to output to both forms at once. Simply by referencing multiple appenders within the logger.

```
<logger name="com.lankydan.service.MyServiceImpl" additivity="false"
level="debug">
  <appender-ref ref="STDOUT"/>
  <appender-ref ref="SAVE-TO-FILE"/>
</logger>
```

So now this logger will output to the console thanks to STDOUT as well as to file using the SAVE-TO-FILE appender.

A similar configuration can be achieved via application.properties. If you go back up the page you might be able to figure out how to do it yourself as a previous example had one extra line added to prevent it from printing to console and to file. Again this will contain log messages from the root logger and not just MyServiceImpl as the snippet above would.

```
logging.path=logs
logging.file=${logging.path}/log.log
logging.pattern.file=%d{dd-MM-yyyy HH:mm:ss.SSS} [%thread] %-5level
%logger{36}.%M - %msg%n
```

A useful feature that Spring Boot provides when using Logback is the ability to separate configuration between environments. So if you wanted to save to file and print to console in your development environment but only print to file in production then this can be achieved with ease.

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

  <!-- config for STDOUT and SAVE-TO-FILE -->

  <springProfile name="dev">
    <root level="info">
      <appender-ref ref="STDOUT"/>
      <appender-ref ref="SAVE-TO-FILE"/>
    </root>
    <logger name="com.lankydan.service.MyServiceImpl"
```

```
additivity="false" level="debug">
      <appender-ref ref="STDOUT"/>
      <appender-ref ref="SAVE-TO-FILE"/>
    </logger>
  </springProfile>

  <springProfile name="prod">
    <root level="info">
      <appender-ref ref="SAVE-TO-FILE"/>
    </root>
    <logger name="com.lankydan.service.MyServiceImpl"
additivity="false" level="error">
      <appender-ref ref="SAVE-TO-FILE"/>
    </logger>
  </springProfile>

</configuration>
```

The first step to get this to work is to rename the logback.xml file to logback-spring.xml allowing the springProfile tag to be used. In this tag a name can be provided which can be set via properties, environment variables or VM options. Below is how you can set the springProfile name to dev which has been used to represent a development environment.

To set in application.properties or as an environment variable

```
spring.profiles.active=dev
```

Or as a VM option

```
-Dspring.profiles.active=dev
```

Now when the application is ran the springProfile for dev will be used causing the logs to be output to the console and to file. If this was then being pushed to production the property needs to be set to prod which will alter the configuration to what is deemed suitable, such as only writing logs to file and possibly changing the logging level of all or certain classes/packages.

A similar configuration can also be provided via application.properties. Well, not actually application.properties but instead from application-dev.properties and application-prod.properties which are separate property files for each environment. Following the naming convention of application-{environment}.properties where {environment} is replaced with the environment name. Depending on your VM options or environment variables one of these can be chosen just like when done through springProfile in logback-spring.xml. Below are the equivalent configurations for the above code snippet.

application-dev.properties

```
logging.level.root=info
logging.level.com.lankydan.service=debug
logging.path=logs
logging.file=${logging.path}/log.log
logging.pattern.file=%d{dd-MM-yyyy HH:mm:ss.SSS} [%thread] %-5level
%logger{36}.%M - %msg%n
logging.pattern.console=%d{dd-MM-yyyy HH:mm:ss.SSS}
%magenta([%thread]) %highlight(%-5level) %logger.%M - %msg%n
```

application-prod.properties

```
logging.level.root=info
logging.level.com.lankydan.service=error
logging.path=logs
logging.file=${logging.path}/log.log
logging.pattern.file=%d{dd-MM-yyyy HH:mm:ss.SSS} [%thread] %-5level
%logger{36}.%M - %msg%n
logging.pattern.console=
```

I think that I should wrap up this post at this point as it was a lot longer than I was originally expecting. That being said there is a lot more that can be done with Logback and Spring Boot that I have not covered here. In conclusion from this tutorial you should have grasped a understanding on how to use Logback with Spring Boot, including how to use property files to alter the default settings provided by Spring Boot and how to go even further and create your own custom made configurations using Logback via logback.xml and logback-spring.xml. As well as having an idea of the limits that configuration inside property files can provide so that you know when it is time to switch over to using Logback directly to get you to the finish line.