Logging

jim stafford

Fall 2024 v2024-08-09: Built: 2024-11-19 21:33 EST

Table of Contents

Chapter 1. Introduction

1.1. Why log?

Logging has many uses within an application — spanning:

- auditing actions
- reporting errors
- providing debug information to assist in locating a problem

With much of our code located in libraries — logging is not just for our application code. We will want to know audit, error, and debug information in our library calls as well:

- did that timer fire?
- which calls failed?
- what HTTP headers were input or returned from a REST call?

1.2. Why use a Logger over System.out?

Use of Loggers allow statements to exist within the code that will either:

- be disabled
- log output uninhibited
- log output with additional properties (e.g., timestamp, thread, caller, etc.)

Logs commonly are written to the console and/or files by default — but that is not always the case. Logs can also be exported into centralized servers or database(s) so they can form an integrated picture of a distributed system and provide search and alarm capabilities.

However simple or robust your end logs become, logging starts with the code and is a very important thing to include from the beginning (even if we waited a few modules to cover it).

1.3. Goals

The student will learn:

- to understand the value in using logging over simple System.out.println calls
- to understand the interface and implementation separation of a modern logging framework
- the relationship between the different logger interfaces and implementations
- to use log levels and verbosity to properly monitor the application under different circumstances

* to express valuable context information in logged messages
* to manage logging verbosity
* to configure the output of logs to provide useful information

1.4. Objectives

At the conclusion of this lecture and related exercises, the student will be able to:

1. obtain access to an SLF4J Logger
2. issue log events at different severity levels
3. filter log events based on source and severity thresholds
4. efficiently bypass log statements that do not meet criteria
5. format log events for regular and exception parameters
6. customize log patterns
7. customize appenders
8. add contextual information to log events using Mapped Diagnostic Context
9. trigger additional logging events using Markers
10. use Spring Profiles to conditionally configure logging

Chapter 2. Starting References

There are many resources on the Internet that cover logging, the individual logging implementations, and the Spring Boot opinionated support for logging. You may want to keep a browser window open to one or more of the following starting links while we cover this material. You will not need to go thru all of them, but know there is a starting point to where detailed examples and explanations can be found if not covered in this lesson.

1. Spring Boot Logging Feature provides documentation from a top-down perspective of how it supplies a common logging abstraction over potentially different logging implementations.
2. SLF4J Web Site provides documentation, articles, and presentations on SLF4J — the chosen logging interface for Spring Boot and much of the Java community.
3. Logback Web Site provides a wealth of documentation, articles, and presentations on Logback — the default logging implementation for Spring Boot.
4. Log4J2 Web Site provides core documentation on Log4J2 — a directly supported Spring Boot alternative logging implementation.
5. Java Util Logging (JUL) Documentation Web Site provides an overview of JUL — a lesser supported Spring Boot alternative implementation for logging.

Chapter 3. Logging Dependencies

Most of what we need to perform logging is supplied through our dependency on the `spring-boot-starter` and its dependency on `spring-boot-starter-logging`. The only time we need to supply additional dependencies is when we want to change the default logging implementation or make use of optional, specialized extensions provided by that logging implementation.

Take a look at the transitive dependencies brought in by a straight forward dependency on `spring-boot-starter`.

Spring Boot Starter Logging Dependencies

```
$ mvn dependency:tree
...
[INFO] info.ejava.examples.app:appconfig-logging-example:jar:6.1.0-SNAPSHOT
[INFO] \- org.springframework.boot:spring-boot-starter:jar:3.3.2:compile
...
[INFO]    +- org.springframework.boot:spring-boot-starter-logging:jar:3.3.2:compile ①
[INFO]    |  +- ch.qos.logback:logback-classic:jar:1.5.6:compile
[INFO]    |  |  +- ch.qos.logback:logback-core:jar:1.5.6:compile
[INFO]    |  |  \- org.slf4j:slf4j-api:jar:2.0.13:compile
[INFO]    |  +- org.apache.logging.log4j:log4j-to-slf4j:jar:2.23.1:compile
[INFO]    |  |  \- org.apache.logging.log4j:log4j-api:jar:2.23.1:compile
[INFO]    |  \- org.slf4j:jul-to-slf4j:jar:2.0.13:compile
...
[INFO]    +- org.springframework:spring-core:jar:3.3.2:compile
[INFO]    |  \- org.springframework:spring-jcl:jar:3.3.2:compile
```

① dependency on `spring-boot-starter` brings in `spring-boot-starter-logging`

3.1. Logging Libraries

Notice that:

- `spring-core` dependency brings in its own repackaging and optimizations of Commons Logging within `spring-jcl`
  - `spring-jcl` provides a thin wrapper that looks for logging APIs and self-bootstraps itself to use them — with a preference for the SLF4J interface, then Log4J2 directly, and then JUL as a fallback
  - `spring-jcl` looks to have replaced the need for jcl-over-slf4j
- `spring-boot-starter-logging` provides dependencies for the SLF4J API, adapters and three optional implementations
  - implementations — these will perform the work behind the SLF4J interface calls
    - Logback (the default)
    - Log4J2
    - Java Util Logging

  - adapters — these will bridge the SLF4J calls to the implementations
    - `Logback` implements SLF4J natively - no adapter necessary
    - `log4j-to-slf4j` bridges Log4j to SLF4J
    - `jul-to-slf4j` - bridges Java Util Logging (JUL) to SLF4J

> If we use Spring Boot with `spring-boot-starter` right out of the box, we will be using the SLF4J API and Logback implementation configured to work correctly for most cases.

3.2. Spring and Spring Boot Internal Logging

Spring and Spring Boot use an internal version of the Apache Commons Logging API (Git Repo) (that was previously known as the Jakarta Commons Logging or JCL (Ref: Wikipedia, Apache Commons Logging)) that is rehosted within the `spring-jcl` module to serve as a bridge to different logging implementations (Ref: Spring Boot Logging).

Chapter 4. Getting Started

OK. We get the libraries we need to implement logging right out of the box with the basic `spring-boot-starter`. How do we get started generating log messages? Lets begin with a comparison with `System.out` so we can see how they are similar and different.

4.1. System.out

System.out was built into Java from day 1

- no extra imports are required
- no extra libraries are required

System.out writes to wherever `System.out` references. The default is stdout. You have seen many earlier examples just like the following.

Example System.out Call

```
@Component
@Profile("system-out") ①
public class SystemOutCommand implements CommandLineRunner {
    public void run(String... args) throws Exception {
        System.out.println("System.out message");
    }
}
```

① restricting component to profile to allow us to turn off unwanted output after this demo

4.2. System.out Output

The example `SystemOutCommand` component above outputs the following statement when called with the `system-out` profile active (using `spring.profiles.active` property).

Example System.out Output

```
$ java -jar target/appconfig-logging-example-*-SNAPSHOT-bootexec.jar \
--spring.profiles.active=system-out ①

System.out message ②
```

① activating profile that turns on our component and turns off all logging

② System.out is not impacted by logging configuration and printed to stdout

4.3. Turning Off Spring Boot Logging

Where did all the built-in logging (e.g., Spring Boot banner, startup messages, etc.) go in the last example?

The `system-out` profile specified a `logging.level.root` property that effectively turned off all logging.

application-system-out.properties

```
spring.main.banner-mode=off ①
logging.level.root=OFF ②
```

① turns off printing of verbose Spring Boot startup banner

② turns off all logging (inheriting from the root configuration)

Technically the logging was only turned off for loggers inheriting the root configuration — but we will ignore that detail for right now and just say "all logging".

4.4. Getting a Logger

Logging frameworks make use of the fundamental design idiom — separate interface from implementation. We want our calling code to have simple access to a simple interface to express information to be logged and the severity of that information. We want the implementation to have limitless capability to produce and manage the logs, but want to only pay for what we likely will use. Logging frameworks allow that to occur and provide primary access thru a logging interface and a means to create an instance of that logger. The following diagram shows the basic stereotype roles played by the factory and logger.

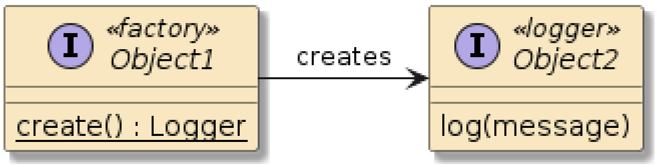


Figure 1. Logging Framework Primary Stereotypes

- Factory creates Logger

Lets take a look at several ways to obtain a Logger using different APIs and techniques.

4.5. Java Util Logger Interface Example

The Java Util Logger (JUL) has been built into Java since 1.4. The primary interface is the `Logger` class. It is used as both the factory and interface for the `Logger` to issue log messages.

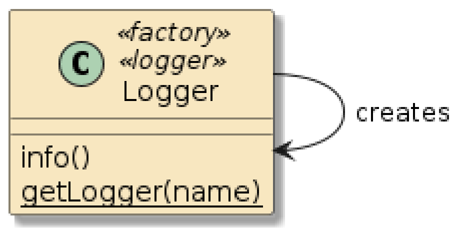


Figure 2. Java Util Logger (JUL) Logger

The following snippet shows an example JUL call.

Example Java Util Logging (JUL) Call

```
package info.ejava.examples.app.config.logging.factory;
...
import java.util.logging.Logger; ①

@Component
public class JULLogger implements CommandLineRunner {
    private static final Logger log = Logger.getLogger(JULLogger.class.getName()); ②

    @Override
    public void run(String... args) throws Exception {
        log.info("Java Util logger message"); ③
    }
}
```

① import the JUL `Logger` class

② get a reference to the JUL `Logger` instance by String name

③ issue a log event

The JUL `Logger` class is used for both the factory and logging interface.

4.6. JUL Example Output

The following output shows that even code using the JUL interface will be integrated into our standard Spring Boot logs.

Example Java Util Logging (JUL) Output

```
java -jar target/appconfig-logging-example-*-SNAPSHOT-bootexec.jar \
--spring.profiles.active=factory
...
20:40:54,136 INFO  info.ejava.examples.app.config.logging.factory.JULLogger - Java
Util logger message
...
```

However, JUL is not widely used as an API or implementation. I won't detail it here, but it has been reported to be much slower and missing robust features of modern alternatives. That does not mean JUL cannot be used as an API for your code (and the libraries your code relies on) and an implementation for your packaged application. It just means using it as an implementation is uncommon and won't be the default in Spring Boot and other frameworks.

4.7. SLF4J Logger Interface Example

Spring Boot provides first class support for the SLF4J logging interface. The following example shows a sequence similar to the JUL sequence except using `Logger` interface and `LoggerFactory` class from the SLF4J library.

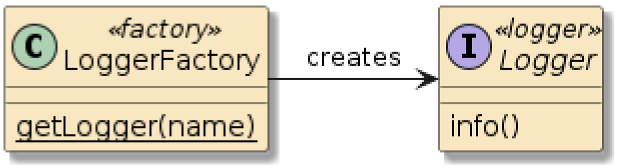


Figure 3. SLF4J LoggerFactory and Logger

SLF4J Example Call

```
package info.ejava.examples.app.config.logging.factory;

import org.slf4j.Logger; ①
import org.slf4j.LoggerFactory;
...
@Component
public class DeclaredLogger implements CommandLineRunner {
    private static final Logger log = LoggerFactory.getLogger(DeclaredLogger.class);
②

    @Override
    public void run(String... args) throws Exception {
        log.info("declared SLF4J logger message"); ③
    }
}
```

① import the SLF4J `Logger` interface and `LoggerFactory` class

② get a reference to the SLF4J `Logger` instance using the `LoggerFactory` class

③ issue a log event

One immediate improvement SLF4J has over JUL interface is the convenience `getLogger()` method that accepts a class. Loggers are structured in a tree hierarchy and it is common best practice to name them after the fully qualified class that they are called from. The `String` form is also available but the `Class` form helps encourage and simplify following a common best practice.

4.8. SLF4J Example Output

SLF4J Example Output

```
$ java -jar target/appconfig-logging-example-*-SNAPSHOT-bootexec.jar \
--spring.profiles.active=factory ①
...
20:40:55,156 INFO  info.ejava.examples.app.config.logging.factory.DeclaredLogger -
declared SLF4J logger message
...
```

① supplying custom profile to filter output to include only the factory examples

4.9. Lombok SLF4J Declaration Example

Naming loggers after the fully qualified class name is so common that the Lombok library was able to successfully take advantage of that fact to automate the tasks for adding the imports and declaring the Logger during Java compilation.

Lombok Example Call

```
package info.ejava.examples.app.config.logging.factory;

import lombok.extern.slf4j.Slf4j;
...
@Component
@Slf4j ①
public class LombokDeclaredLogger implements CommandLineRunner {
②
    @Override
    public void run(String... args) throws Exception {
        log.info("lombok declared SLF4J logger"); ③
    }
}
```

① `@Slf4j` annotation automates the import statements and Logger declaration

② Lombok will declare a static `log` property using `LoggerFactory` during compilation

③ normal log statement provided by calling class — no different from earlier example

4.10. Lombok Example Output

Since Lombok primarily automates code generation at compile time, the produced output is identical to the previous manual declaration example.

Lombok Example Output

```
$ java -jar target/appconfig-logging-example-*-SNAPSHOT-bootexec.jar \
--spring.profiles.active=factory
```

```
...
20:40:55,155 INFO  info.ejava.examples.app.config.logging.factory.LombokDeclaredLogger
- lombok declared SLF4J logger message
...
```

4.11. Lombok Dependency

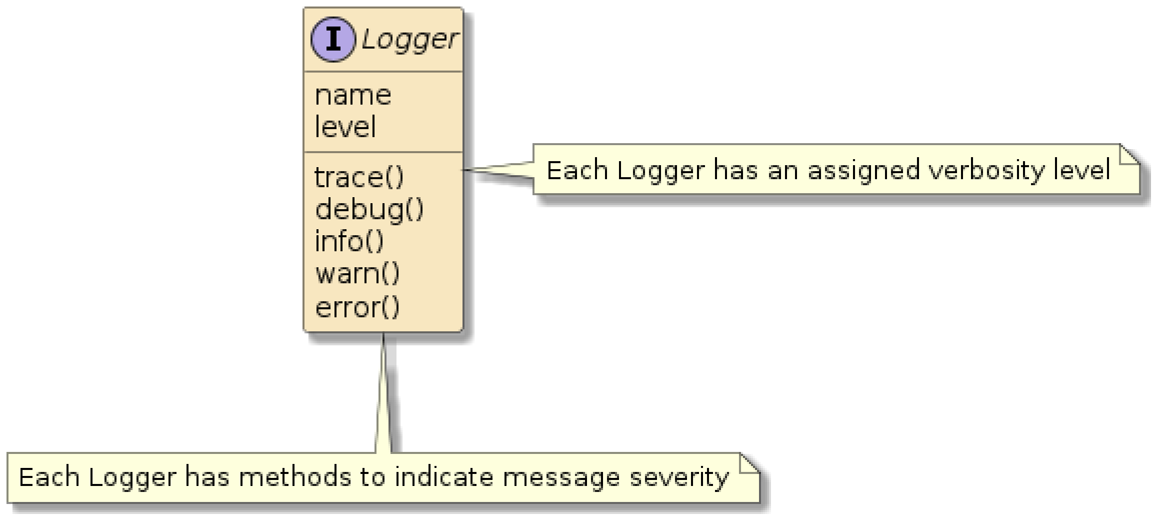
Of course, we need to add the following dependency to the project `pom.xml` to enable Lombok annotation processing.

Lombok Dependency

```
<!-- used to declare logger -->
<dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
    <scope>provided</scope>
</dependency>
```

Chapter 5. Logging Levels

The `Logger` returned from the `LoggerFactory` will have an associated `level` assigned elsewhere — that represents its verbosity threshold. We issue messages to the `Logger` using separate methods that indicate their severity.



Logger
name
level
trace()
debug()
info()
warn()
error()
Each Logger has an assigned verbosity level
Each Logger has methods to indicate message severity

Figure 4. Logging Level

The logging severity level calls supported by SLF4J span from `trace()` to `error()`. The logging threshold levels supported by Spring Boot and Logback also span from `TRACE` to `ERROR` and include an `OFF` (all levels are case insensitive). When we compare levels and thresholds — treat `TRACE` being less severe than `ERROR`.

Severity levels supported by other APIs (JUL Levels, Log4J2 Levels) are mapped to levels supported by SLF4J.

5.1. Common Level Use

Although there cannot be enforcement of when to use which level — there are common conventions. The following is a set of conventions I live by:

TRACE

Detailed audits events and verbose state of processing

- example: Detailed state at a point in the code. SQL, params, and results of a query.

DEBUG

Audit events and state giving insight of actions performed

- example: Beginning/ending a decision branch or a key return value

INFO

Notable audit events and state giving some indication of overall activity performed

- example: Started/completed transaction for purchase

WARN

Something unusual to highlight but the application was able to recover

- example: Read timeout for remote source

ERROR

Significant or unrecoverable error occurred and an important action failed. These should be extremely limited in their use.

- example: Cannot connect to database

5.2. Log Level Adjustments

Obviously, there are no perfect guidelines. Adjustments need to be made on a case by case basis.

When forming your logging approach — ask yourself "are the logs telling me what I need to know when I look under the hood?", "what can they tell me with different verbosity settings?", and "what will it cost in terms of performance and storage?".

The last thing you want to do is to be called in for a problem and the logs tell you nothing or too much of the wrong information. Even worse — changing the verbosity of the logs will not help for when the issue occurs the next time.

5.3. Logging Level Example Calls

The following is an example of making very simple calls to the logger at different severity levels.

Logging Level Example Calls

```
package info.ejava.examples.app.config.logging.levels;
...
@Slf4j
public class LoggerLevels implements CommandLineRunner {
    @Override
    public void run(String... args) throws Exception {
        log.trace("trace message"); ①
        log.debug("debug message");
        log.info("info message");
        log.warn("warn message");
        log.error("error message"); ①
    }
}
```

① example issues one log message at each of the available LSF4J severity levels

5.4. Logging Level Output: INFO

This example references a simple profile that configures loggers for our package to report at the

INFO severity level to simulate the default.

Logging Level INFO Example Output

```
$ java -jar target/appconfig-logging-example-*-SNAPSHOT-bootexec.jar \
--spring.profiles.active=levels ①

06:36:15,910 INFO  info.ejava.examples.app.config.logging.levels.LoggerLevels - info
message ②
06:36:15,912 WARN  info.ejava.examples.app.config.logging.levels.LoggerLevels - warn
message
06:36:15,912 ERROR info.ejava.examples.app.config.logging.levels.LoggerLevels - error
message
```

① profile sets `info.ejava.examples.app.config.logging.levels` threshold level to INFO

② messages logged for `INFO`, `WARN`, and `ERROR` severity because they are >= `INFO`

The referenced profile turns off all logging except for the `info.ejava⋯levels` package being demonstrated and customizes the pattern of the logs. We will look at that more soon.

application-levels.properties

```
#application-levels.properties
logging.pattern.console=%date{HH:mm:ss.SSS} %-5level %logger - %msg%n ③

logging.level.info.ejava.examples.app.config.logging.levels=info ②
logging.level.root=OFF ①
```

① all loggers are turned off by default

② example package logger threshold level produces log events with severity >= `INFO`

③ customized console log messages to contain pertinent example info

5.5. Logging Level Output: DEBUG

Using the command line to express a `logging.level` property, we lower the threshold for our logger to `DEBUG` and get one additional severity level in the output.

Logging Level DEBUG Example Output

```
$ java -jar target/appconfig-logging-example-*-SNAPSHOT-bootexec.jar \
--spring.profiles.active=levels \
--logging.level.info.ejava.examples.app.config.logging.levels=DEBUG ①

06:37:04,292 DEBUG info.ejava.examples.app.config.logging.levels.LoggerLevels - debug
message ②
06:37:04,293 INFO  info.ejava.examples.app.config.logging.levels.LoggerLevels - info
message
06:37:04,294 WARN  info.ejava.examples.app.config.logging.levels.LoggerLevels - warn
message
```

```
06:37:04,294 ERROR info.ejava.examples.app.config.logging.levels.LoggerLevels - error
message
```

① logging.level sets `info.ejava.examples.app.config.logging.levels` threshold level to DEBUG

② messages logged for `DEBUG`, `INFO`, `WARN`, and `ERROR` severity because they are >= `DEBUG`

5.6. Logging Level Output: TRACE

Using the command line to express a `logging.level` property, we lower the threshold for our logger to `TRACE` and get two additional severity levels in the output over what we produced with `INFO`.

Logging Level TRACE Example Output

```
$ java -jar target/appconfig-logging-example-*-SNAPSHOT-bootexec.jar \
--spring.profiles.active=levels \
--logging.level.info.ejava.examples.app.config.logging.levels=TRACE ①

06:37:19,968 TRACE info.ejava.examples.app.config.logging.levels.LoggerLevels - trace
message ②
06:37:19,970 DEBUG info.ejava.examples.app.config.logging.levels.LoggerLevels - debug
message
06:37:19,970 INFO  info.ejava.examples.app.config.logging.levels.LoggerLevels - info
message
06:37:19,970 WARN  info.ejava.examples.app.config.logging.levels.LoggerLevels - warn
message
06:37:19,970 ERROR info.ejava.examples.app.config.logging.levels.LoggerLevels - error
message
```

① logging.level sets `info.ejava.examples.app.config.logging.levels` threshold level to TRACE

② messages logged for all severity levels because they are >= `TRACE`

5.7. Logging Level Output: WARN

Using the command line to express a `logging.level` property, we raise the threshold for our logger to `WARN` and get one less severity level in the output over what we produced with `INFO`.

Logging Level WARN Example Output

```
$ java -jar target/appconfig-logging-example-*-SNAPSHOT-bootexec.jar \
--spring.profiles.active=levels \
--logging.level.info.ejava.examples.app.config.logging.levels=WARN ①

06:37:32,753 WARN  info.ejava.examples.app.config.logging.levels.LoggerLevels - warn
message ②
06:37:32,755 ERROR info.ejava.examples.app.config.logging.levels.LoggerLevels - error
message
```

① logging.level sets `info.ejava.examples.app.config.logging.levels` threshold level to WARN

② messages logged for `WARN`, and `ERROR` severity because they are >= `WARN`

5.8. Logging Level Output: OFF

Using the command line to express a `logging.level` property, we set the threshold for our logger to `OFF` and get no output produced.

Logging Level OFF Example Output

```
$ java -jar target/appconfig-logging-example-*-SNAPSHOT-bootexec.jar \
--spring.profiles.active=levels \
--logging.level.info.ejava.examples.app.config.logging.levels=OFF ①

②
```

① logging.level sets `info.ejava.examples.app.config.logging.levels` threshold level to OFF

② no messages logged because logger turned off

Chapter 6. Discarded Message Expense

The designers of logger frameworks are well aware that excess logging — even statements that are disabled — can increase the execution time of a library call or overall application. We have already seen how severity level thresholds can turn off output and that gives us substantial savings within the logging framework itself. However, we must be aware that building a message to be logged can carry its own expense and be aware of the tools to mitigate the problem.

Assume we have a class that is relatively expensive to obtain a String representation.

Example Expensive toString()

```
class ExpensiveToLog {
    public String toString() { ①
        try { Thread.sleep(1000); } catch (Exception ignored) {}
        return "hello";
    }
}
```

① calling `toString()` on instances of this class will incur noticeable delay

6.1. Blind String Concatenation

Now lets say we create a message to log through straight, eager String concatenation. What is wong here?

Blind String Concatenation Example

```
ExpensiveToLog obj=new ExpensiveToLog();
//...
log.debug("debug for expensiveToLog: " + obj + "!");
```

1. The log message will get formed by eagerly concatenating several Strings together
2. One of those Strings is produced by a relatively expensive `toString()` method
3. **Problem**: The work of eagerly forming the String is wasted if `DEBUG` is not enabled

6.2. Verbosity Check

Assuming the information from the `toString()` call is valuable and needed when we have `DEBUG` enabled — a verbosity check is one common solution we can use to determine if the end result is worth the work. There are two very similar ways we can do this.

The first way is to dynamically check the current threshold level of the logger within the code and only execute if the requested severity level is enabled. We are still going to build the relatively expensive String when `DEBUG` is enabled but we are going to save all that processing time when it is not enabled. This overall approach of using a code block works best when creating the message requires multiple lines of code. This specific technique of dynamically checking is suitable when there are very few checks within a class.

6.2.1. Dynamic Verbosity Check

The first way is to dynamically check the current threshold level of the logger within the code and only execute if the requested severity level is enabled. We are still going to build the relatively expensive String when `DEBUG` is enabled but we are going to save all that processing time when it is not enabled. This overall approach of using a code block works best when creating the message requires multiple lines of code. This specific technique of dynamically checking is suitable when there are very few checks within a class.

Dynamic Verbosity Check Example

```
if (log.isDebugEnabled()) { ①
    log.debug("debug for expensiveToLog: " + obj +"!");
}
```

① code block with expensive `toString()` call is bypassed when `DEBUG` disabled

6.2.2. Static Final Verbosity Check

A variant of the first approach is to define a `static final boolean` variable at the start of the class, equal to the result of the enabled test. This variant allows the JVM to know that the value of the `if` predicate will never change allowing the code block and further checks to be eliminated when disabled. This alternative is better when there are multiple blocks of code that you want to make conditional on the threshold level of the logger. This solution assumes the logger threshold will never be changed or that the JVM will be restarted to use the changed value. I have seen this technique commonly used in libraries where they anticipate many calls and they are commonly judged on their method throughput performance.

Static Verbosity Check Example

```
private static final boolean DEBUG_ENABLED = log.isDebugEnabled(); ①
...
    if (DEBUG_ENABLED) { ②
        log.debug("debug for expensiveToLog: " + obj + "!");
    }
...
```

① logger's verbosity level tested when class loaded and stored in `static final` variable

② code block with expensive `toString()`

6.3. SLF4J Parameterized Logging

SLF4J API offers another solution that removes the need for the `if` clause — thus cleaning your code of those extra conditional blocks. The SLF4J `Logger` interface has a `format` and `args` variant for each verbosity level call that permits the threshold to be consulted prior to converting any of the parameters to a String.

The format specification uses a set of curly braces ("{}") to express an insertion point for an ordered set of arguments. There are no format options. It is strictly a way to lazily call `toString()` on each argument and insert the result.

SLF4J Parameterized Logging Example

```
log.debug("debug for expensiveToLog: {}!", obj); ① ②
```

① `{}` is a placeholder for the result of `obj.toString()` if called

② `obj.toString()` only called and overall message concatenated if logger threshold set to ⇐ `DEBUG`

6.4. Simple Performance Results: Disabled

Not scientific by any means, but the following results try to highlight the major cost differences between blind concatenation and the other methods. The basic results also show the parameterized logging technique to be on par with the threshold level techniques with far less code complexity.

The test code warms up the logger with a few calls and then issues the debug statements shown above in succession with time hacks taken in between each.

The first set of results are from logging threshold set to `INFO`. The blind concatenation shows that it eagerly calls the `obj.toString()` method just to have its resulting message discarded. The other methods do not pay a measurable penalty.

- test code
  - warms up logger with few calls
  - issues the debug statements shown above in succession
  - time hacks taken in between each
- first set of results are from logging threshold set to `INFO`
  - blind concatenation shows it eagerly calls the `obj.toString()` method just to have its resulting message discarded
  - other methods do not pay a measurable penalty

Disabled Logger Relative Results

```
$ java -jar target/appconfig-logging-example-*-SNAPSHOT-bootexec.jar \
--spring.profiles.active=expense \
--logging.level.info.ejava.examples.app.config.logging.expense=INFO

11:44:25.462 INFO  info.ejava.examples.app.config.logging.expense.DisabledOptimization
- warmup logger
11:44:26.476 INFO  info.ejava.examples.app.config.logging.expense.DisabledOptimization
- \

concat: 1012, ifDebug=0, DEBUG_ENABLED=0, param=0 ① ②
```

① eager blind concatenation pays `toString()` cost even when not needed (1012ms)

② verbosity check and lazy parameterized logging equally efficient (0ms)

6.5. Simple Performance Results: Enabled

The second set of results are from logging threshold set to `DEBUG`. You can see that causes the relatively expensive `toString()` to be called for each of the four techniques shown with somewhat equal results. I would not put too much weight on a few milliseconds difference between the calls here except to know that neither provide a noticeable processing delay over the other when the logging threshold has been met.

Enabled Logger Relative Results

```
$ java -jar target/appconfig-logging-example-*-SNAPSHOT-bootexec.jar \
--spring.profiles.active=expense \
--logging.level.info.ejava.examples.app.config.logging.expense=DEBUG

11:44:43.560 INFO  info.ejava.examples.app.config.logging.expense.DisabledOptimization
- warmup logger
11:44:43.561 DEBUG info.ejava.examples.app.config.logging.expense.DisabledOptimization
- warmup logger
11:44:44.572 DEBUG info.ejava.examples.app.config.logging.expense.DisabledOptimization
- debug for expensiveToLog: hello!
11:44:45.575 DEBUG info.ejava.examples.app.config.logging.expense.DisabledOptimization
- debug for expensiveToLog: hello!
11:44:46.579 DEBUG info.ejava.examples.app.config.logging.expense.DisabledOptimization
- debug for expensiveToLog: hello!
11:44:46.579 DEBUG info.ejava.examples.app.config.logging.expense.DisabledOptimization
- debug for expensiveToLog: hello!
11:44:47.582 INFO  info.ejava.examples.app.config.logging.expense.DisabledOptimization
- \

concat: 1010, ifDebug=1003, DEBUG_ENABLED=1004, param=1003 ①
```

① all four methods paying the cost of the relatively expensive `obj.toString()` call

Chapter 7. Exception Logging

SLF4J interface and parameterized logging goes one step further to also support `Exceptions`. If you pass an `Exception` object as the last parameter in the list — it is treated special and will not have its `toString()` called with the rest of the parameters. Depending on the configuration in place, the stack trace for the `Exception` is logged instead. The following snippet shows an example of an `Exception` being thrown, caught, and then logged.

Example Exception Logging

```
public void run(String... args) throws Exception {
    try {
        log.info("calling iThrowException");
        iThrowException();
    } catch (Exception ex) {
        log.warn("caught exception", ex); ①
    }
}

private void iThrowException() throws Exception {
    throw new Exception("example exception");
}
```

① `Exception` passed to logger with message

7.1. Exception Example Output

When we run the example, note that the message is printed in its normal location and a stack trace is added for the supplied `Exception` parameter.

Example Exception Logging Output

```
$ java -jar target/appconfig-logging-example-*-SNAPSHOT-bootexec.jar \
--spring.profiles.active=exceptions

13:41:17.119 INFO  info.ejava.examples.app.config.logging.exceptions.ExceptionExample
- calling iThrowException
13:41:17.121 WARN  info.ejava.examples.app.config.logging.exceptions.ExceptionExample
- caught exception ①
java.lang.Exception: example exception ②
    at
info.ejava.examples.app.config.logging.exceptions.ExceptionExample.iThrowException(Exc
eptionExample.java:23)
    at
info.ejava.examples.app.config.logging.exceptions.ExceptionExample.run(ExceptionExampl
e.java:15)
    at
org.springframework.boot.SpringApplication.callRunner(SpringApplication.java:784)
...
```

```
    at org.springframework.boot.loader.Launcher.launch(Launcher.java:51)
    at org.springframework.boot.loader.JarLauncher.main(JarLauncher.java:52)
```

① normal message logged

② stack trace for last `Exception` parameter logged

7.2. Exception Logging and Formatting

Note that you can continue to use parameterized logging with Exceptions. The message passed in above was actually a format with no parameters. The snippet below shows a format with two parameters and an `Exception`.

Example Exception Logging with Parameters

```
log.warn("caught exception {} {}", "p1","p2", ex);
```

The first two parameters are used in the formatting of the core message. The last Exception parameters is printed as a regular exception.

Example Exception Logging with Parameters Output

```
13:41:17.119 INFO  info.ejava.examples.app.config.logging.exceptions.ExceptionExample
- calling iThrowException
13:41:17.122 WARN  info.ejava.examples.app.config.logging.exceptions.ExceptionExample
- caught exception p1 p2 ①
java.lang.Exception: example exception ②
    at
info.ejava.examples.app.config.logging.exceptions.ExceptionExample.iThrowException(Exc
eptionExample.java:23)
    at
info.ejava.examples.app.config.logging.exceptions.ExceptionExample.run(ExceptionExampl
e.java:15)
    at
org.springframework.boot.SpringApplication.callRunner(SpringApplication.java:784)
...
    at org.springframework.boot.loader.Launcher.launch(Launcher.java:51)
    at org.springframework.boot.loader.JarLauncher.main(JarLauncher.java:52)
```

① two early parameters ("p1" and "p2") where used to complete the message template

② `Exception` passed as the last parameter had its stack trace logged

Chapter 8. Logging Pattern

Each of the previous examples showed logging output using a particular pattern. The pattern was expressed using a `logging.pattern.console` property. The Logback Conversion Documentation provides details about how the logging pattern is defined.

Sample Custom Pattern

```
logging.pattern.console=%date{HH:mm:ss.SSS} %-5level %logger - %msg%n
```

The pattern consisted of:

- %date (or %d)- time of day down to millisecs
- %level (or %p, %le)- severity level left justified and padded to 5 characters
- %logger (or %c, %lo)- full name of logger
- %msg (or %m, %message) - full logged message
- %n - operating system-specific new line

If you remember, that produced the following output.

Review: LoggerLevels Example Pattern Output

```
java -jar target/appconfig-logging-example-*-SNAPSHOT-bootexec.jar \
--spring.profiles.active=levels

06:00:38.891 INFO  info.ejava.examples.app.config.logging.levels.LoggerLevels - info
message
06:00:38.892 WARN  info.ejava.examples.app.config.logging.levels.LoggerLevels - warn
message
06:00:38.892 ERROR info.ejava.examples.app.config.logging.levels.LoggerLevels - error
message
```

8.1. Default Console Pattern

Spring Boot comes out of the box with a slightly more verbose default pattern expressed with the CONSOLE_LOG_PATTERN property. The following snippet depicts the information found within the Logback property definition — with some new lines added in to help read it.

Gist of `CONSOLE_LOG_PATTERN` from GitHub

```
%clr(%d{${LOG_DATEFORMAT_PATTERN:-yyyy-MM-dd HH:mm:ss.SSS}}){faint}
%clr(${LOG_LEVEL_PATTERN:-%5p})
%clr(${PID:- }){magenta}
%clr(---){faint}
%clr([%15.15t]){faint}
%clr(%-40.40logger{39}){cyan}
%clr(:){faint}
```

```
%m%n${LOG_EXCEPTION_CONVERSION_WORD:-%wEx}}
```

You should see some familiar conversion words from my earlier pattern example. However, there are some additional conversion words used as well. Again, keep the Logback Conversion Documentation close by to lookup any additional details.

- %d - timestamp defaulting to full format
- %p - severity level right justified and padded to 5 characters
- $PID - system property containing the process ID
- %t (or %thread) - thread name right justified and padded to 15 characters and chopped at 15 characters
- %logger - logger name optimized to fit within 39 characters , left justified and padded to 40 characters, chopped at 40 characters
- %m - fully logged message
- %n - operating system-specific new line
- %wEx - Spring Boot-defined exception formatting

8.2. Default Console Pattern Output

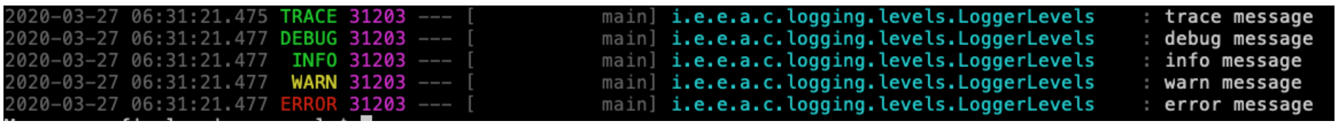
We will take a look at conditional variable substitution in a moment. This next example reverts to the default `CONSOLE_LOG_PATTERN`.

LoggerLevels Output with Default Spring Boot Console Log Pattern

```
java -jar target/appconfig-logging-example-*-SNAPSHOT-bootexec.jar \
--logging.level.root=OFF \
--logging.level.info.ejava.examples.app.config.logging.levels.LoggerLevels=TRACE

2020-03-27 06:31:21.475 TRACE 31203 --- [           main]
i.e.e.a.c.logging.levels.LoggerLevels    : trace message
2020-03-27 06:31:21.477 DEBUG 31203 --- [           main]
i.e.e.a.c.logging.levels.LoggerLevels    : debug message
2020-03-27 06:31:21.477  INFO 31203 --- [           main]
i.e.e.a.c.logging.levels.LoggerLevels    : info message
2020-03-27 06:31:21.477  WARN 31203 --- [           main]
i.e.e.a.c.logging.levels.LoggerLevels    : warn message
2020-03-27 06:31:21.477 ERROR 31203 --- [           main]
i.e.e.a.c.logging.levels.LoggerLevels    : error message
```

Spring Boot defines color coding for the console that is not visible in the text of this document. The color for severity level is triggered by the level — red for `ERROR`, yellow for `WARN`, and green for the other three levels.



```
2020-03-27 06:31:21.475 TRACE 31203 --- [           main] i.e.e.a.c.logging.levels.LoggerLevels    : trace message
2020-03-27 06:31:21.477 DEBUG 31203 --- [           main] i.e.e.a.c.logging.levels.LoggerLevels    : debug message
2020-03-27 06:31:21.477  INFO 31203 --- [           main] i.e.e.a.c.logging.levels.LoggerLevels    : info message
2020-03-27 06:31:21.477  WARN 31203 --- [           main] i.e.e.a.c.logging.levels.LoggerLevels    : warn message
2020-03-27 06:31:21.477 ERROR 31203 --- [           main] i.e.e.a.c.logging.levels.LoggerLevels    : error message
```

Figure 5. Default Spring Boot Console Log Pattern Coloring

8.3. Variable Substitution

Logging configurations within Spring Boot make use of variable substitution. The value of `LOG_DATEFORMAT_PATTERN` will be applied wherever the expression `${LOG_DATEFORMAT_PATTERN}` appears. The `"${}"` characters are part of the variable expression and will not be part of the result.

8.4. Conditional Variable Substitution

Variables can be defined with default values in the event they are not defined. In the following expression `${LOG_DATEFORMAT_PATTERN:-yyyy-MM-dd HH:mm:ss.SSS}`:

- the value of LOG_DATEFORMAT_PATTERN will be used if defined
- the value of "yyyy-MM-dd HH:mm:ss.SSS" will be used if not defined

The `"${}"` and embedded `":-"` characters following the variable name are part of the expression when appearing within an XML configuration file and will not be part of the result. The dash (`-`) character should be removed if using within a property definition.

8.5. Date Format Pattern

As we saw from a peek at the Spring Boot `CONSOLE_LOG_PATTERN` default definition, we can change the format of the timestamp using the `LOG_DATEFORMAT_PATTERN` system property. That system property can flexibly be set using the `logging.pattern.dateformat` property. See the Spring Boot Documentation for information on this and other properties. The following example shows setting that property using a command line argument.

Setting Date Format

```
$ java -jar target/appconfig-logging-example-*-SNAPSHOT-bootexec.jar \
--logging.level.root=OFF \
--logging.level.info.ejava.examples.app.config.logging.levels.LoggerLevels=INFO \
--logging.pattern.dateformat="HH:mm:ss.SSS" ①

08:20:42.939  INFO 39013 --- [           main] i.e.e.a.c.logging.levels.LoggerLevels
: info message
08:20:42.942  WARN 39013 --- [           main] i.e.e.a.c.logging.levels.LoggerLevels
: warn message
08:20:42.942 ERROR 39013 --- [           main] i.e.e.a.c.logging.levels.LoggerLevels
: error message
```

① setting `LOG_DATEFORMAT_PATTERN` using `logging.pattern.dateformat` property

8.6. Log Level Pattern

We also saw from the default definition of `CONSOLE_LOG_PATTERN` that the severity level of the output can be changed using the `LOG_LEVEL_PATTERN` system property. That system property can be flexibly set with the `logging.pattern.level` property. The following example shows setting the format to a single character, left justified. Therefore, we can map `INFO`⇒ `I`, `WARN`⇒ `W`, and `ERROR`⇒ `E`.

```
$ java -jar target/appconfig-logging-example-*-SNAPSHOT-bootexec.jar \
--logging.level.root=OFF \
--logging.level.info.ejava.examples.app.config.logging.levels.LoggerLevels=INFO \
--logging.pattern.dateformat="HH:mm:ss.SSS" \
--logging.pattern.level="%.-1p" ①
②
08:59:17.376 I 44756 --- [           main] i.e.e.a.c.logging.levels.LoggerLevels    :
info message
08:59:17.379 W 44756 --- [           main] i.e.e.a.c.logging.levels.LoggerLevels    :
warn message
08:59:17.379 E 44756 --- [           main] i.e.e.a.c.logging.levels.LoggerLevels    :
error message
```

① `logging.level.pattern` expressed to be 1 character, left justified

② single character produced in console log output

8.7. Conversion Pattern Specifiers

Spring Boot Features Web Page documents some formatting rules. However, more details on the parts within the conversion pattern are located on the Logback Pattern Layout Web Page. The overall end-to-end pattern definition I have shown you is called a "Conversion Pattern". Conversion Patterns are made up of:

- Literal Text (e.g., `---`, whitespace, `:`) — hard-coded strings providing decoration and spacing for conversion specifiers
- Conversion Specifiers - (e.g., `%-40.40logger{39}`) — an expression that will contribute a formatted property of the current logging context
  - starts with `%`
  - followed by format modifiers — (e.g., `-40.40`) — addresses min/max spacing and right/left justification
    - optionally provide minimum number of spaces
      - use a negative number (`-#`) to make it left justified and a positive number (`#`) to make it right justified
    - optionally provide maximum number of spaces using a decimal place and number (`.#`). Extra characters will be cut off
      - use a negative number (`.-#`) to start from the left and positive number (`.#`) to start from the right

- followed by a conversion word (e.g., `logger`, `msg`) — keyword name for the property
- optional parameters (e.g., `{39}`) — see individual conversion words for details on each

8.8. Format Modifier Impact Example

The following example demonstrates how the different format modifier expressions can impact the `level` property.

Table 1. %level Format Modifier Impact Example

logging.pattern.loglevel	output	comment

`[%level]`	`[INFO]`
	`[WARN]`
	`[ERROR]`
		value takes whatever space necessary

`[%6level]`	`[  INFO]`
	`[  WARN]`
	`[ ERROR]`
		value takes at least 6 characters, right justified

`[%-6level]`	`[INFO  ]`
	`[WARN  ]`
	`[ERROR ]`
		value takes at least 6 characters, left justified

`[%.-2level]`	`[IN]`
	`[WA]`
	`[ER]`
		value takes no more than 2 characters, starting from the left

`[%.2level]`	`[FO]`
	`[RN]`
	`[OR]`
		value takes no more than 2 characters, starting from the right

8.9. Example Override

Earlier you saw how we could control the console pattern for the %date and %level properties. To go any further, we are going to have to override the entire `CONSOLE_LOG_PATTERN` system property and can define it using the `logging.pattern.console` property.

That is too much to define on the command line, so lets move our definition to a profile-based property file (`application-layout.properties`)

application-layout.properties

```
#application-layout.properties ①

#default to time-of-day for the date
logging.pattern.dateformat=HH:mm:ss.SSS
#supply custom console layout
logging.pattern.console=%clr(%d{${LOG_DATEFORMAT_PATTERN:HH:mm:ss.SSS}}){faint} \
%clr(${LOG_LEVEL_PATTERN:%5p}) \
%clr(-){faint} \
%clr(%.27logger{40}){cyan}\
%clr(#){faint}\
%clr(%method){cyan}\ ②
%clr(:){faint}\
%clr(%line){cyan} \ ②
%m%n\
${LOG_EXCEPTION_CONVERSION_WORD:%wEx}}

logging.level.info.ejava.examples.app.config.logging.levels.LoggerLevels=INFO
logging.level.root=OFF
```

① property file used when `layout` profile active

② customization added `method` and `line` of caller — at a processing expense

8.10. Expensive Conversion Words

I added two new helpful properties that could be considered controversial because they require extra overhead to obtain that information from the JVM. The technique has commonly involved throwing and catching an exception internally to determine the calling location from the self-generated stack trace:

- %method (or %M) - name of method calling logger
- %line (or %L) - line number of the file where logger call was made

The additional "expensive" fields are being used for console output for demonstrations using a demonstration profile. Consider your log information needs on a case-by-case basis and learn from this lesson what and how you can modify the logs for your specific needs. For example — to debug an error, you can switch to a more detailed and verbose profile without changing code.

8.11. Example Override Output

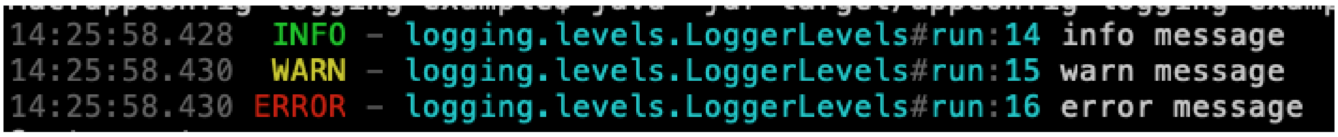
We can activate the profile and demonstrate the modified format using the following command.

Example Console Pattern Override Output

```
$ java -jar target/appconfig-logging-example-*-SNAPSHOT-bootexec.jar \
--spring.profiles.active=layout

14:25:58.428  INFO - logging.levels.LoggerLevels#run:14 info message
14:25:58.430  WARN - logging.levels.LoggerLevels#run:15 warn message
14:25:58.430 ERROR - logging.levels.LoggerLevels#run:16 error message
```

The coloring does not show up above so the image below provides a perspective of what that looks like.



```
14:25:58.428  INFO - logging.levels.LoggerLevels#run:14 info message
14:25:58.430  WARN - logging.levels.LoggerLevels#run:15 warn message
14:25:58.430 ERROR - logging.levels.LoggerLevels#run:16 error message
```

Figure 6. Example Console Pattern Override Coloring

8.12. Layout Fields

Please see the Logback Layouts Documentation for a detailed list of conversion words and how to optionally format them.

Chapter 9. Loggers

We have demonstrated a fair amount capability thus far without having to know much about the internals of the logger framework. However, we need to take a small dive into the logging framework in order to explain some further concepts.

- Logger Ancestry
- Logger Inheritance
- Appenders
- Logger Additivity

9.1. Logger Tree

Loggers are organized in a hierarchy starting with a root logger called "root". As you would expect, higher in the tree are considered ancestors and lower in the tree are called descendants.

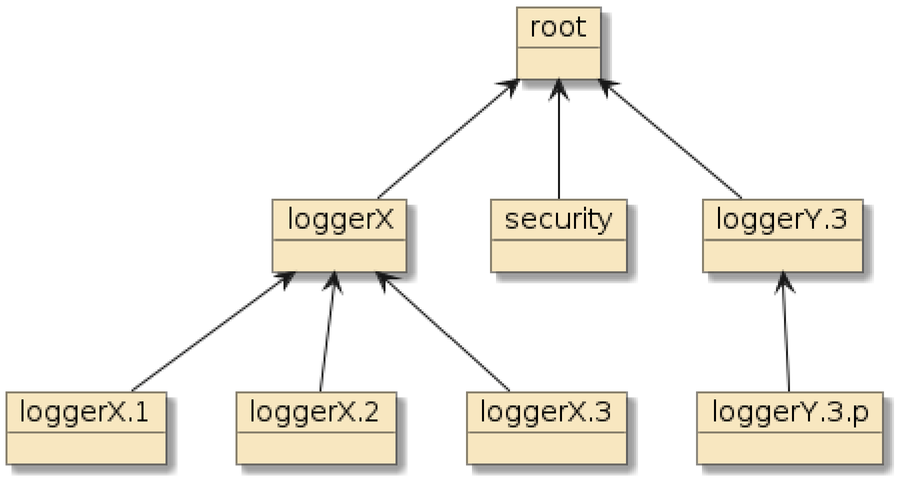


Figure 7. Example Logger Tree

Except for root, the ancestor/descendant structure of loggers depends on the hierarchical name of each logger. Based on the loggers in the diagram

- X, Y.3, and security are descendants and direct children of root
- Y.3 is example of logger lacking an explicitly defined parent in hierarchy before reaching root. We can skip many levels between child and root and still retain same hierarchical name
- X.1, X.2, and X.3 are descendants of X and root and direct children of X
- Y.3.p is descendant of Y.3 and root and direct child of Y.3

9.2. Logger Inheritance

Each logger has a set of allowed properties. Each logger may define its own value for those properties, inherit the value of its parent, or be assigned a default (as in the case for root).

9.3. Logger Threshold Level Inheritance

The first inheritance property we will look at is a familiar one to you — severity threshold level. As the diagram shows

- root, loggerX, security, loggerY.3, loggerX.1 and loggerX.3 set an explicit value for their threshold
- loggerX.2 and loggerY.3.p inherit the threshold from their parent

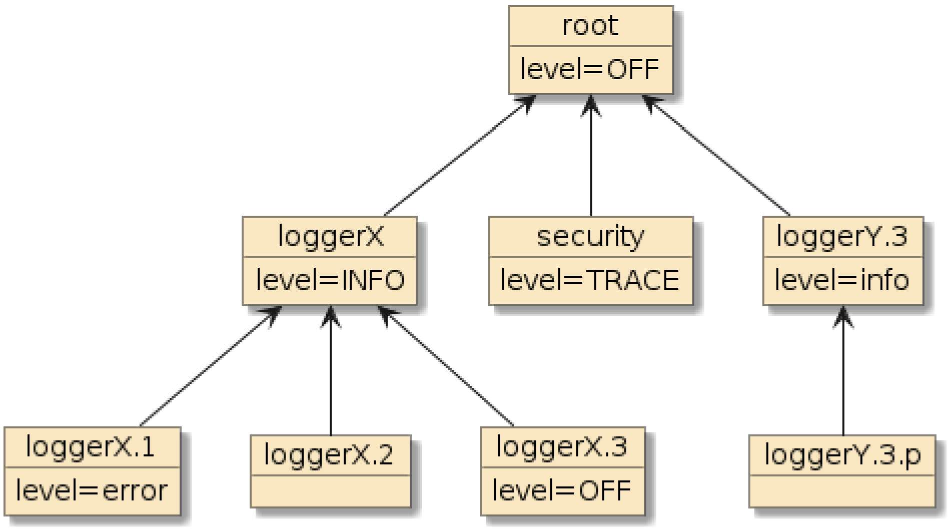


Figure 8. Logger Level Inheritance

9.4. Logger Effective Threshold Level Inheritance

The following table shows the specified and effective values applied to each logger for their threshold.

Table 2. Effective Logger Threshold Level

logger name	specified threshold	effective threshold
`root`	`OFF`	`OFF`
X	`INFO`	`INFO`
X.1	`ERROR`	`ERROR`
X.2		`INFO`
X.3	`OFF`	`OFF`
Y.3	`WARN`	`WARN`
Y.3.p		`WARN`
security	`TRACE`	`TRACE`

9.5. Example Logger Threshold Level Properties

These thresholds can be expressed in a property file.

application-tree.properties Snippet

```
logging.level.X=info
logging.level.X.1=error
logging.level.X.3=OFF
logging.level.security=trace
logging.level.Y.3=warn
logging.level.root=OFF
```

9.6. Example Logger Threshold Level Output

The output below demonstrates the impact of logging level inheritance from ancestors to descendants.

Effective Logger Theshold Level Output

```
$ java -jar target/appconfig-logging-example-*-SNAPSHOT-bootexec.jar \
--spring.profiles.active=tree

CONSOLE  05:58:14.956  INFO - X#run:25 X info
CONSOLE  05:58:14.959  WARN - X#run:26 X warn
CONSOLE  05:58:14.959 ERROR - X#run:27 X error
CONSOLE  05:58:14.960 ERROR - X.1#run:27 X.1 error ②
CONSOLE  05:58:14.960  INFO - X.2#run:25 X.2 info ①
CONSOLE  05:58:14.960  WARN - X.2#run:26 X.2 warn
CONSOLE  05:58:14.960 ERROR - X.2#run:27 X.2 error
CONSOLE  05:58:14.960  WARN - Y.3#run:26 Y.3 warn
CONSOLE  05:58:14.960 ERROR - Y.3#run:27 Y.3 error
CONSOLE  05:58:14.960  WARN - Y.3.p#run:26 Y.3.p warn ①
CONSOLE  05:58:14.961 ERROR - Y.3.p#run:27 Y.3.p error
CONSOLE  05:58:14.961 TRACE - security#run:23 security trace ③
CONSOLE  05:58:14.961 DEBUG - security#run:24 security debug
CONSOLE  05:58:14.962  INFO - security#run:25 security info
CONSOLE  05:58:14.962  WARN - security#run:26 security warn
CONSOLE  05:58:14.962 ERROR - security#run:27 security error
```

① X.2 and Y.3.p exhibit the same threshold level as their parents X (`INFO`) and Y.3 (`WARN`)

② X.1 (`ERROR`) and X.3 (`OFF`) override their parent threshold levels

③ security is writing all levels >= TRACE

Chapter 10. Appenders

Loggers generate `LoggerEvents` but do not directly log anything. Appenders are responsible for taking a `LoggerEvent` and producing a message to a log. There are many types of appenders. We have been working exclusively with a `ConsoleAppender` thus far but will work with some others before we are done. At this point — just know that a `ConsoleLogger` uses:

- an encoder to determine when to write messages to the log
- a layout to determine how to transform an individual `LoggerEvent` to a String
- a pattern when using a `PatternLayout` to define the transformation

10.1. Logger has N Appenders

Each of the loggers in our tree has the chance to have 0..N appenders.

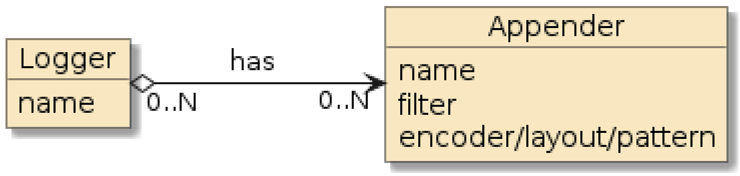


Figure 9. Logger / Appender Relationship

10.2. Logger Configuration Files

To date we have been able to work mostly with Spring Boot properties when using loggers. However, we will need to know a few things about the Logger Configuration File in order to define an appender and assign it to logger(s). We will start with how the logger configuration is found.

Logback and Log4J2 both use XML as their primary definition language. Spring Boot will automatically locate a well-known named configuration file in the root of the classpath:

- `logback.xml` or `logback-spring.xml` for Logback
- `log4j2.xml` or `log4j2-spring.xml` for Log4J2

Spring Boot documentation recommends using the `-spring.xml` suffixed files over the provider default named files in order for Spring Boot to assure that all documented features can be enabled. Alternately, we can explicitly specify the location using the `logging.config` property to reference anywhere in the classpath or file system.

application-tree.properties Reference

```
...
logging.config=classpath:/logging-configs/tree/logback-spring.xml ①
...
```

① an explicit property reference to the logging configuration file to use

10.3. Logback Root Configuration Element

The XML file has a root `configuration` element which contains details of the appender(s) and logger(s). See the Spring Boot Configuration Documentation and the Logback Configuration Documentation for details on how to configure.

logging-configs/tree/logback-spring.xml Configuration

```
<configuration debug="false"> ①
   ...
</configuration>
```

① `debug` attribute triggers logback debug

10.4. Retain Spring Boot Defaults

We will lose most/all of the default Spring Boot customizations for logging when we define our own custom logging configuration file. We can restore them by adding an include. This is that same file that we looked at earlier for the definition of `CONSOLE_LOG_PATTERN`.

logging-configs/tree/logback-spring.xml Retain Spring Boot Defaults

```
<configuration>
    <!-- bring in Spring Boot defaults for Logback -->
    <include resource="org/springframework/boot/logging/logback/defaults.xml"/>
   ...
</configuration>
```

10.5. Appender Configuration

Our example tree has three (3) appenders total. Each adds a literal string prefix so we know which appender is being called.

logging-configs/tree/logback-spring.xml Appenders

```
<!-- leverages what Spring Boot would have given us for console -->
<appender name="console" class="ch.qos.logback.core.ConsoleAppender">
    <encoder class="ch.qos.logback.classic.encoder.PatternLayoutEncoder"> ①
        <pattern>CONSOLE  ${CONSOLE_LOG_PATTERN}</pattern> ②
        <charset>utf8</charset>
    </encoder>
</appender>
<appender name="X-appender" class="ch.qos.logback.core.ConsoleAppender">
    <encoder>
        <pattern>X        ${CONSOLE_LOG_PATTERN}</pattern>
    </encoder>
</appender>
<appender name="security-appender" class="ch.qos.logback.core.ConsoleAppender">
```

```
    <encoder>
        <pattern>SECURITY ${CONSOLE_LOG_PATTERN}</pattern>
    </encoder>
</appender>
```

① PatternLayoutEncoder is the default encoder

② `CONSOLE_PATTERN_LAYOUT` is defined in included defaults.xml

This example forms the basis for demonstrating logger/appender assignment and appender additivity. `ConsoleAppender` is used in each case for ease of demonstration and not meant to depict a realistic configuration.

10.6. Appenders Attached to Loggers

The appenders are each attached to a single logger using the `appender-ref` element.

- console is attached to the root logger
- X-appender is attached to loggerX logger
- security-appender is attached to security logger

I am latching the two child appender assignments within an `appenders` profile to:

1. keep them separate from the earlier log level demo
2. demonstrate how to leverage Spring Boot extensions to build profile-based conditional logging configurations.

logging-configs/tree/logback-spring.xml Loggers

```
<springProfile name="appenders"> ①
    <logger name="X">
        <appender-ref ref="X-appender"/> ②
    </logger>

    <!-- this logger starts a new tree of appenders, nothing gets written to root
logger -->
    <logger name="security" additivity="false">
        <appender-ref ref="security-appender"/>
    </logger>
</springProfile>

<root>
    <appender-ref ref="console"/>
</root>
```

① using Spring Boot Logback extension to only enable appenders when profile active

② appenders associated with logger using `appender-ref`

10.7. Appender Tree Inheritance

These appenders, in addition to level, are inherited from ancestor to descendant unless there is an override defined by the property `additivity=false`.

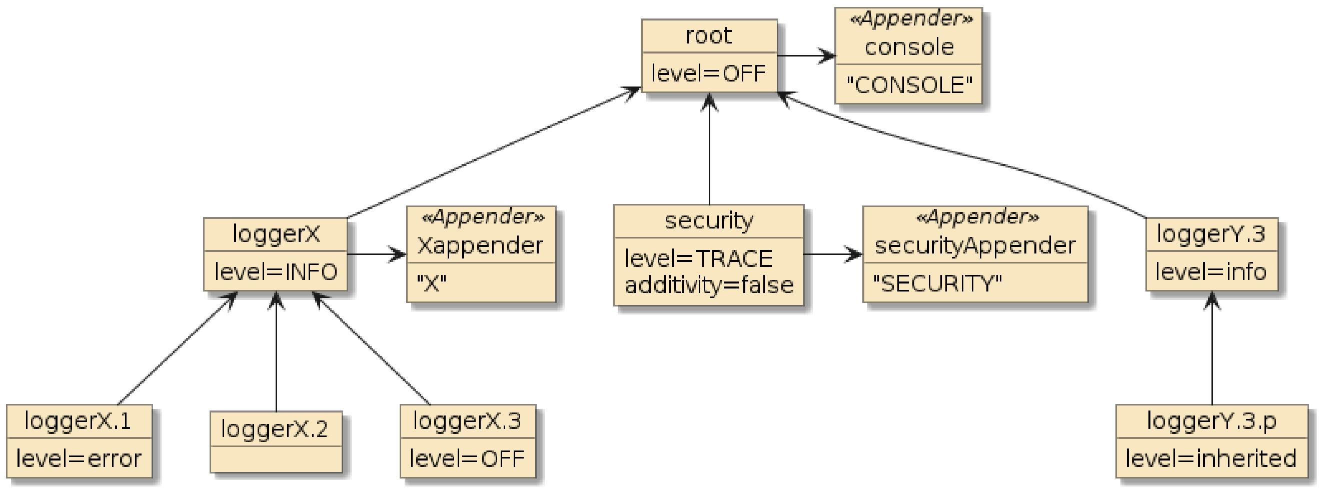


Figure 10. Example Logger Tree with Appenders

10.8. Appender Additivity Result

logger name	assigned threshold	assigned appender	effective threshold	effective appender
`root`	`OFF`	`console`	`OFF`	`console`
`X`	`INFO`	`X-appender`	`INFO`	`console, X-appender`
`X.1`	`ERROR`		`ERROR`	`console, X-appender`
`X.2`			`INFO`	`console, X-appender`
`X.3`	`OFF`		`OFF`	`console, X-appender`
`Y.3`	`WARN`		`WARN`	`console`
`Y.3.p`			`WARN`	`console`
`security *additivity=false`	`TRACE`	`security-appender`	`TRACE`	`security-appender`

10.9. Logger Inheritance Tree Output

```
$ java -jar target/appconfig-logging-example-*-SNAPSHOT-bootexec.jar \
--spring.profiles.active=tree,appenders

X        19:12:07.220  INFO - X#run:25 X info ①
CONSOLE  19:12:07.220  INFO - X#run:25 X info ①
X        19:12:07.224  WARN - X#run:26 X warn
```

```
CONSOLE  19:12:07.224  WARN - X#run:26 X warn
X        19:12:07.225 ERROR - X#run:27 X error
CONSOLE  19:12:07.225 ERROR - X#run:27 X error
X        19:12:07.225 ERROR - X.1#run:27 X.1 error
CONSOLE  19:12:07.225 ERROR - X.1#run:27 X.1 error
X        19:12:07.225  INFO - X.2#run:25 X.2 info
CONSOLE  19:12:07.225  INFO - X.2#run:25 X.2 info
X        19:12:07.225  WARN - X.2#run:26 X.2 warn
CONSOLE  19:12:07.225  WARN - X.2#run:26 X.2 warn
X        19:12:07.226 ERROR - X.2#run:27 X.2 error
CONSOLE  19:12:07.226 ERROR - X.2#run:27 X.2 error
CONSOLE  19:12:07.226  WARN - Y.3#run:26 Y.3 warn ②
CONSOLE  19:12:07.227 ERROR - Y.3#run:27 Y.3 error ②
CONSOLE  19:12:07.227  WARN - Y.3.p#run:26 Y.3.p warn
CONSOLE  19:12:07.227 ERROR - Y.3.p#run:27 Y.3.p error
SECURITY 19:12:07.227 TRACE - security#run:23 security trace ③
SECURITY 19:12:07.227 DEBUG - security#run:24 security debug ③
SECURITY 19:12:07.227  INFO - security#run:25 security info ③
SECURITY 19:12:07.228  WARN - security#run:26 security warn ③
SECURITY 19:12:07.228 ERROR - security#run:27 security error ③
```

① log messages written to logger X and descendants are written to console and X-appender appenders

② log messages written to logger Y.3 and descendants are written only to console appender

③ log messages written to security logger are written only to security appender because of `additivity=false`

Chapter 11. Mapped Diagnostic Context

Thus far, we have been focusing on calls made within the code without much concern about the overall context in which they were made. In a multi-threaded, multi-user environment there is additional context information related to the code making the calls that we may want to keep track of — like userId and transactionId.

SLF4J and the logging implementations support the need for call context information through the use of Mapped Diagnostic Context (MDC). The MDC class is a essentially a `ThreadLocal` map of strings that are assigned for the current thread. The values of the MDC are commonly set and cleared in container filters that fire before and after client calls are executed.

11.1. MDC Example

The following is an example where the `run()` method is playing the role of the container filter — setting and clearing the MDC. For this MDC map — I am setting a "user" and "requestId" key with the current user identity and a value that represents the request. The `doWork()` method is oblivious of the MDC and simply logs the start and end of work.

MDC Example

```
import org.slf4j.MDC;
...
public class MDCLogger implements CommandLineRunner {
    private static final String[] USERS = new String[]{"jim", "joe", "mary"};
    private static final SecureRandom r = new SecureRandom();

    @Override
    public void run(String... args) throws Exception {
        for (int i=0; i<5; i++) {
            String user = USERS[r.nextInt(USERS.length)];
            MDC.put("user", user); ①
            MDC.put("requestId", Integer.toString(r.nextInt(99999)));
            doWork();
            MDC.clear(); ②
            doWork();
        }
    }

    public void doWork() {
        log.info("starting work");
        log.info("finished work");
    }
}
```

① `run()` method simulates container filter setting context properties before call executed

② context properties removed after all calls for the context complete

11.2. MDC Example Pattern

To make use of the new "user" and "requestId" properties of the thread, we can add the `%mdc` (or %X) conversion word to the appender pattern as follows.

Adding MDC Properties to Pattern

```
#application-mdc.properties
logging.pattern.console=%date{HH:mm:ss.SSS} %-5level [%-9mdc{user:-
anonymous}][%5mdc{requestId}] %logger{0} - %msg%n
```

- %mdc{user:-anonymous} - the identity of the user making the call or "anonymous" if not supplied
- %mdc{requestId} - the specific request made or blank if not supplied

11.3. MDC Example Output

The following is an example of running the MDC example. Users are randomly selected and work is performed for both identified and anonymous users. This allows us to track who made the work request and sort out the results of each work request.

MDC Example Output

```
$ java -jar target/appconfig-logging-example-*-SNAPSHOT-bootexec.jar
--spring.profiles.active=mdc

17:11:59.100 INFO  [jim      ][61165] MDCLogger - starting work
17:11:59.101 INFO  [jim      ][61165] MDCLogger - finished work
17:11:59.101 INFO  [anonymous][     ] MDCLogger - starting work
17:11:59.101 INFO  [anonymous][     ] MDCLogger - finished work
17:11:59.101 INFO  [mary     ][ 8802] MDCLogger - starting work
17:11:59.101 INFO  [mary     ][ 8802] MDCLogger - finished work
17:11:59.101 INFO  [anonymous][     ] MDCLogger - starting work
17:11:59.101 INFO  [anonymous][     ] MDCLogger - finished work
17:11:59.101 INFO  [mary     ][86993] MDCLogger - starting work
17:11:59.101 INFO  [mary     ][86993] MDCLogger - finished work
17:11:59.101 INFO  [anonymous][     ] MDCLogger - starting work
17:11:59.101 INFO  [anonymous][     ] MDCLogger - finished work
17:11:59.102 INFO  [mary     ][67677] MDCLogger - starting work
17:11:59.102 INFO  [mary     ][67677] MDCLogger - finished work
17:11:59.102 INFO  [anonymous][     ] MDCLogger - starting work
17:11:59.102 INFO  [anonymous][     ] MDCLogger - finished work
17:11:59.102 INFO  [jim      ][25693] MDCLogger - starting work
17:11:59.102 INFO  [jim      ][25693] MDCLogger - finished work
17:11:59.102 INFO  [anonymous][     ] MDCLogger - starting work
17:11:59.102 INFO  [anonymous][     ] MDCLogger - finished work
```

Like standard `ThreadLocal` variables, child threads do not inherit values of parent thread. Each thread will maintain its own MDC properties.

11.4. Clearing MDC Context

We are responsible for setting the MDC context variables as well as clearing them when the work is complete.

- One way to do that is using a finally block and manually calling MDC.clear()

```
try {
    MDC.put("user", user); ①
    MDC.put("requestId", requestId);
    doWork();
} finally {
    MDC.clear();
}
```

- Another is by using a try-with-closable and have the properties automatically cleared when the try block finishes.

```
try (MDC.MDCCloseable mdc = MDC.putCloseable("user", user);
     MDC.MDCCloseable mdc = MDC.putCloseable("requestId", requestId) {
    doWork();
}
```

Chapter 12. Markers

SLF4J and the logging implementations support markers. Unlike MDC data — which quietly sit in the background — markers are optionally supplied on a per-call basis. Markers have two primary uses

- trigger reporting events to appenders — e.g., flush log, send the e-mail
- implement additional severity levels — e.g., `log.warn(FLESH_WOUND,"come back here!")` versus `log.warn(FATAL,"ouch!!!")` [1]

> The additional functionality commonly is implemented through the use of filters assigned to appenders looking for these `Markers`.

> To me having triggers initiated by the logging statements does not sound appropriate (but still could be useful). However, when the thought of filtering comes up — I think of cases where we may want to better classify the subject(s) of the statement so that we have more to filter on when configuring appenders. More than once I have been in a situation where adjusting the verbosity of a single logger was not granular enough to provide an ideal result.

12.1. Marker Class

`Markers` have a single property called name and an optional collection of child `Markers`. The name and collection properties allow the parent marker to represent one or more values. Appender filters test `Markers` using the `contains()` method to determine if the parent or any children are the targeted value.

`Markers` are obtained through the `MarkerFactory` — which caches the `Markers` it creates unless requested to make them detached so they can be uniquely added to separate parents.

12.2. Marker Example

The following simple example issues two log events. The first is without a `Marker` and the second with a `Marker` that represents the value `ALARM`.

Marker Example

```
import org.slf4j.Marker;
import org.slf4j.MarkerFactory;
...
public class MarkerLogger implements CommandLineRunner {
    private static final Marker ALARM = MarkerFactory.getMarker("ALARM"); ①

    @Override
    public void run(String... args) throws Exception {
        log.warn("non-alarming warning statement"); ②
        log.warn(ALARM,"alarming statement"); ③
    }
}
```

① created single managed marker

② no marker added to logging call

③ marker added to logging call to trigger something special about this call

12.3. Marker Appender Filter Example

The Logback configuration has two appenders. The first appender — `alarms` — is meant to log only log events with an ALARM marker. I have applied the Logback-supplied `EvaluatorFilter` and `OnMarkerEvaluator` to eliminate any log events that do not meet that criteria.

Alarm Appender

```
    <appender name="alarms" class="ch.qos.logback.core.ConsoleAppender">
        <filter class="ch.qos.logback.core.filter.EvaluatorFilter">
            <evaluator name="ALARM" class=
"ch.qos.logback.classic.boolex.OnMarkerEvaluator">
                <marker>ALARM</marker>
            </evaluator>
            <onMatch>ACCEPT</onMatch>
            <onMismatch>DENY</onMismatch>
        </filter>
        <encoder>
            <pattern>%red(ALARM&gt;&gt;&gt; ${CONSOLE_LOG_PATTERN})</pattern>
        </encoder>
    </appender>
```

The second appender — console — accepts all log events.

All Event Appender

```
    <appender name="console" class="ch.qos.logback.core.ConsoleAppender">
        <encoder>
            <pattern>${CONSOLE_LOG_PATTERN}</pattern>
        </encoder>
    </appender>
```

Both appenders are attached to the same root logger — which means that anything logged to the alarm appender will also be logged to the console appender.

Both Appenders added to root Logger

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

```
        <appender-ref ref="console"/>
        <appender-ref ref="alarms"/>
    </root>
</configuration>
```

12.4. Marker Example Result

The following shows the results of running the marker example — where both events are written to the console appender and only the log event with the `ALARM` Marker is written to the alarm appender.

```
$ java -jar target/appconfig-logging-example-*-SNAPSHOT-bootexec.jar \
--spring.profiles.active=markers

18:06:52.135 WARN  [] MarkerLogger - non-alarming warning statement ①
18:06:52.136 WARN  [ALARM] MarkerLogger - alarming statement ①
ALARM>>> 18:06:52.136 WARN  [ALARM] MarkerLogger - alarming statement ②
```

① non-ALARM and ALARM events are written to the console appender

② ALARM event is also written to alarm appender

[1] *"what are markers in java logging frameworks and what is a reason to use them"*, Stack Overflow, 2019

Chapter 13. File Logging

Each topic and example so far has been demonstrated using the console because it is simple to demonstrate and to try out for yourself. However, once we get into more significant use of our application we are going to need to write this information somewhere to analyze later when necessary.

For that purpose, Spring Boot has a built-in appender ready to go for file logging. It is not active by default but all we have to do is specify the file name or path to trigger its activation.

Trigger FILE Appender

```
$ java -jar target/appconfig-logging-example-*-SNAPSHOT-bootexec.jar \
--spring.profiles.active=levels \
--logging.file.name="mylog.log" ①

$ ls ./mylog.log ②
./mylog.log
```

① adding this property adds file logging to default configuration

② this expressed logfile will be written to `mylog.log` in current directory

13.1. root Logger Appenders

As we saw earlier with appender additivity, multiple appenders can be associated with the same logger (root logger in this case). With the trigger property supplied, a file-based appender is added to the root logger to produce a log file in addition to our console output.

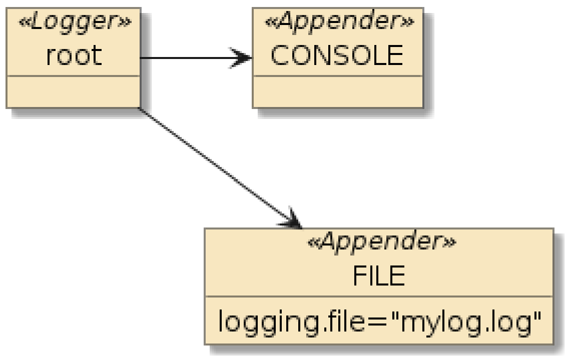


Figure 11. root Logger Appenders

13.2. FILE Appender Output

Under these simple conditions, a file is produced in the current directory with the specified `mylog.log` filename and the following contents.

FILE Appender File Output

```
$ cat mylog.log ①
2020-03-29 07:14:33.533  INFO 90958 --- [main] i.e.e.a.c.logging.levels.LoggerLevels
```

```
: info message
2020-03-29 07:14:33.542  WARN 90958 --- [main] i.e.e.a.c.logging.levels.LoggerLevels
: warn message
2020-03-29 07:14:33.542 ERROR 90958 --- [main] i.e.e.a.c.logging.levels.LoggerLevels
: error message
```

① written to file specified by `logging.file` property

The file and parent directories will be created if they do not exist. The default definition of the appender will append to an existing file if it already exists. Therefore — if we run the example a second time we get a second set of messages in the file.

FILE Appender Defaults to Append Mode

```
$ cat mylog.log
2020-03-29 07:14:33.533  INFO 90958 --- [main] i.e.e.a.c.logging.levels.LoggerLevels
: info message
2020-03-29 07:14:33.542  WARN 90958 --- [main] i.e.e.a.c.logging.levels.LoggerLevels
: warn message
2020-03-29 07:14:33.542 ERROR 90958 --- [main] i.e.e.a.c.logging.levels.LoggerLevels
: error message
2020-03-29 07:15:00.338  INFO 91090 --- [main] i.e.e.a.c.logging.levels.LoggerLevels
: info message ①
2020-03-29 07:15:00.342  WARN 91090 --- [main] i.e.e.a.c.logging.levels.LoggerLevels
: warn message
2020-03-29 07:15:00.342 ERROR 91090 --- [main] i.e.e.a.c.logging.levels.LoggerLevels
: error message
```

① messages from second execution appended to same log

13.3. Spring Boot FILE Appender Definition

If we take a look at the definition for Spring Boot's Logback FILE Appender, we can see that it is a Logback `RollingFileAppender` with a Logback `SizeAndTimeBasedRollingPolicy`.

Spring Boot Default FILE Appender Definition

```
<appender name="FILE"
          class="ch.qos.logback.core.rolling.RollingFileAppender"> ①
    <filter class="ch.qos.logback.classic.filter.ThresholdFilter">
        <level>${FILE_LOG_THRESHOLD}</level>
    </filter>
    <encoder>
        <pattern>${FILE_LOG_PATTERN}</pattern>
        <charset>${FILE_LOG_CHARSET}</charset>
    </encoder>
    <file>${LOG_FILE}</file>
    <rollingPolicy class="ch.qos.logback.core.rolling.SizeAndTimeBasedRollingPolicy">
②
        <fileNamePattern>${LOGBACK_ROLLINGPOLICY_FILE_NAME_PATTERN:-
${LOG_FILE}.%d{yyyy-MM-dd}.%i.gz}</fileNamePattern>
        <cleanHistoryOnStart>${LOGBACK_ROLLINGPOLICY_CLEAN_HISTORY_ON_START:-
false}</cleanHistoryOnStart>
        <maxFileSize>${LOGBACK_ROLLINGPOLICY_MAX_FILE_SIZE:-10MB}</maxFileSize>
        <totalSizeCap>${LOGBACK_ROLLINGPOLICY_TOTAL_SIZE_CAP:-0}</totalSizeCap>
        <maxHistory>${LOGBACK_ROLLINGPOLICY_MAX_HISTORY:-7}</maxHistory>
    </rollingPolicy>
</appender>
```

① performs file rollover functionality based on configured policy

② specifies policy and policy configuration to use

13.4. RollingFileAppender

The Logback RollingFileAppender will:

- write log messages to a specified file — and at some point, switch to writing to a different file
- use a triggering policy to determine the point in which to switch files (i.e., "when it will occur")
- use a rolling policy to determine how the file switchover will occur (i.e., "what will occur")
- use a single policy for both if the rolling policy implements both policy interfaces
- use file append mode by default

The rollover settings/state is evaluated no sooner than once a minute. If you set the maximum sizes to small amounts and log quickly for test/demonstration purposes, you will exceed your defined size limits until the recheck timeout has expired.

13.5. SizeAndTimeBasedRollingPolicy

The Logback SizeAndTimeBasedRollingPolicy will:

- trigger a file switch when the current file reaches a maximum size
- trigger a file switch when the granularity of the primary date (%d) pattern in the file path/name would rollover to a new value
- supply a name for the old/historical file using a mandatory date (%d) pattern and index (%i)
- define a maximum number of historical files to retain
- define a total size to allocate to current and historical files

13.6. FILE Appender Properties

name	description	default
logging.file.path	full or relative path of directory written to — ignored when `logging.file.name` provided	.

name	description	default
logging.file.name	full or relative path of filename written to — may be manually built using `${logging.file.path}`	${logging.file.path}/spring.log
logging.pattern.rolling-file-name	pattern expression for historical file — must include a date and index — may express compression	${logging.file.name}.%d{yyyy-MM-dd}.%i.gz
logging.logback.rollingpolicy.max-file-size	maximum size of log before changeover — must be less than `total-size-cap`	10MB
logging.logback.rollingpolicy.max-history	maximum number of historical files to retain when changing over because of date criteria	7
logging.logback.rollingpolicy.total-size-cap	maximum amount of total space to consume — must be greater than `max-size`	(no limit)

If file logger property value is invalid, the application will run without the FILE appender activated.

13.7. logging.file.path

If we specify only the `logging.file.path`, the filename will default to `spring.log` and will be written to the directory path we supply.

logging.file.path Example

```
$ java -jar target/appconfig-logging-example-*-SNAPSHOT-bootexec.jar \
--logging.file.path=target/logs ①
...
$ ls target/logs ②
spring.log
```

① specifying `logging.file.path` as `target/logs`

② produces a `spring.log` in that directory

13.8. logging.file.name

If we specify only the `logging.file.name`, the file will be written to the filename and directory we explicitly supply.

logging.file.name Example

```
$ java -jar target/appconfig-logging-example-*-SNAPSHOT-bootexec.jar \
--logging.file.name=target/logs/mylog.log ①
...
$ ls target/logs ②
mylog.log
```

① specifying a `logging.file.name`

② produces a logfile with that path and name

13.9. logging.logback.rollingpolicy.max-file-size Trigger

One trigger for changing over to the next file is `logging.logback.rollingpolicy.max-file-size`. Note this property is Logback-specific. The condition is satisfied when the current logfile reaches this value. The default is 10MB.

The following example changes that to 9400 Bytes. Once each instance of `logging.file.name` reached the `logging.logback.rollingpolicy.max-file-size`, it is compressed and moved to a filename with the pattern from `logging.pattern.rolling-file-name`. I picked 9400 Bytes based on the fact the application wrote 4800 Bytes each minute. The file size would be evaluated each minute and exceed the limit every 2 minutes. Notice the uncompressed, archived files are at least 9400 Bytes and 2 minutes apart.

Evaluation is no more often than 1 minute.

logging.logback.rollingpolicy.max-file-size Example

```
java -jar target/appconfig-logging-example-*-SNAPSHOT-bootexec.jar \
--spring.profiles.active=rollover \
--logging.file.name=target/logs/mylog.log \
--logging.pattern.rolling-file-name='${logging.file.name}.%d{yyyy-MM-dd}.%i.log' \
--logging.logback.rollingpolicy.max-file-size=9400B ①
...
ls -ltrh target/logs/

-rw-r--r--  1 jim  staff   9.4K Aug  9 11:29 mylog.log.2024-08-09.0.log ③
-rw-r--r--  1 jim  staff   9.5K Aug  9 11:31 mylog.log.2024-08-09.1.log ②
-rw-r--r--  1 jim  staff   1.6K Aug  9 11:31 mylog.log ①
```

① `logging.logback.rollingpolicy.max-file-size` limits the size of the current logfile

② historical logfiles renamed according to `logging.pattern.rolling-file-name` pattern

③ file size is evaluated each minute and archived when satisfied

13.10. logging.pattern.rolling-file-name

There are several aspects of `logging.pattern.rolling-file-name` to be aware of

- `%d` timestamp pattern and `%i` index are required. The FILE appender will either be disabled (for `%i`) or the application startup will fail (for `%d`) if not specified
- the timestamp pattern directly impacts changeover when there is a value change in the result of applying the timestamp pattern. Many of my examples here use a pattern that includes `HH:mm:ss` just for demonstration purposes. A more common pattern would be by date only.
- the index is used when the `logging.logback.rollingpolicy.max-file-size` triggers the changeover and we already have a historical name with the same timestamp.
- the number of historical files is throttled using `logging.logback.rollingpolicy.max-history` only when index is used and not when file changeover is due to `logging.logback.rollingpolicy.max-file-size`
- the historical file will be compressed if `gz` is specified as the suffix

13.11. Timestamp Rollover Example

The following example shows the file changeover occurring because the evaluation of the `%d` template expression within `logging.pattern.rolling-file-name` changing. The historical file is left uncompressed because the `logging.pattern.rolling-file-name` does not end in `gz`.

Timestamp Rollover Example

```
$ java -jar target/appconfig-logging-example-*-SNAPSHOT-bootexec.jar \
--spring.profiles.active=rollover \
--logging.file.name=target/logs/mylog.log \
--logging.pattern.rolling-file-name='${logging.file.name}.%d{yyyy-MM-dd
-HH:mm:ss}.%i'.log ①
...
$ ls -ltrh target/logs

total 64
-rw-r--r--  1 jim  staff    79B Aug  9 12:04 mylog.log.2024-08-09-12:04:54.0.log
-rw-r--r--  1 jim  staff    79B Aug  9 12:04 mylog.log.2024-08-09-12:04:55.0.log
-rw-r--r--  1 jim  staff    79B Aug  9 12:04 mylog.log.2024-08-09-12:04:56.0.log
-rw-r--r--  1 jim  staff    79B Aug  9 12:04 mylog.log.2024-08-09-12:04:57.0.log
-rw-r--r--  1 jim  staff    79B Aug  9 12:04 mylog.log.2024-08-09-12:04:58.0.log
-rw-r--r--  1 jim  staff    79B Aug  9 12:04 mylog.log.2024-08-09-12:04:59.0.log
-rw-r--r--  1 jim  staff    79B Aug  9 12:05 mylog.log.2024-08-09-12:05:00.0.log
-rw-r--r--  1 jim  staff    79B Aug  9 12:05 mylog.log


$ file target/logs/mylog.log.2024-08-09-12\:04\:54.0.log ②
target/logs/mylog.log.2024-08-09-12:04:54.0.log: ASCII text
```

① `logging.pattern.rolling-file-name` pattern triggers changeover at the seconds boundary

② historical logfiles are left uncompressed because of `.log` name suffix specified

> Using a date pattern to include minutes and seconds is just for demonstration and learning purposes. Most patterns would be daily.

13.12. History Compression Example

The following example is similar to the previous one with the exception that the `logging.pattern.rolling-file-name` ends in `gz` — triggering the historical file to be compressed.

History Compression Example

```
$ java -jar target/appconfig-logging-example-*-SNAPSHOT-bootexec.jar \
--spring.profiles.active=rollover \
--logging.file.name=target/logs/mylog.log \
--logging.pattern.rolling-file-name='${logging.file.name}.%d{yyyy-MM-dd-HH:mm}.%i'.gz
①
...
$ ls -ltrh target/logs

total 48
-rw-r--r--  1 jim  staff   193B Aug  9 13:39 mylog.log.2024-08-09-13:38.0.gz ①
-rw-r--r--  1 jim  staff   534B Aug  9 13:40 mylog.log.2024-08-09-13:39.0.gz
-rw-r--r--  1 jim  staff   540B Aug  9 13:41 mylog.log.2024-08-09-13:40.0.gz
-rw-r--r--  1 jim  staff   528B Aug  9 13:42 mylog.log.2024-08-09-13:41.0.gz
-rw-r--r--  1 jim  staff   539B Aug  9 13:43 mylog.log.2024-08-09-13:42.0.gz
-rw-r--r--  1 jim  staff   1.7K Aug  9 13:43 mylog.log

$ file target/logs/mylog.log.2024-08-09-13\:38.0.gz
target/logs/mylog.log.2024-08-09-13:38.0.gz: gzip compressed data, original size
modulo 2^32 1030
```

① historical logfiles are compressed when pattern uses a `.gz` suffix

13.13. logging.logback.rollingpolicy.max-history Example

`logging.logback.rollingpolicy.max-history` will constrain the number of files created for independent timestamps. In the example below, I constrained the limit to 2. Note that the `logging.logback.rollingpolicy.max-history` property does not seem to apply to files terminated because of size. For that, we can use `logging.file.total-size-cap`.

Max History Example

```
$ java -jar target/appconfig-logging-example-*-SNAPSHOT-bootexec.jar \
--spring.profiles.active=rollover \
--logging.file.name=target/logs/mylog.log \
--logging.pattern.rolling-file-name='${logging.file.name}.%d{yyyy-MM-dd
-HH:mm:ss}.%i'.log \ --logging.logback.rollingpolicy.max-history=2
...
$ ls -ltrh target/logs

total 24
-rw-r--r--  1 jim  staff    80B Aug  9 12:15 mylog.log.2024-08-09-12:15:31.0.log ①
-rw-r--r--  1 jim  staff    80B Aug  9 12:15 mylog.log.2024-08-09-12:15:32.0.log ①
-rw-r--r--  1 jim  staff    80B Aug  9 12:15 mylog.log
```

① specifying `logging.logback.rollingpolicy.max-history` limited number of historical logfiles. Oldest files exceeding the criteria are deleted.

13.14. logging.logback.rollingpolicy.total-size-cap Index Example

The following example triggers file changeover every 1000 Bytes and makes use of the index because we encounter multiple changes per timestamp pattern. The files are aged-off at the point where total size for all logs reaches `logging.logback.rollingpolicy.total-size-cap`. Thus historical files with indexes 1 thru 9 have been deleted at this point in time in order to stay below the file size limit.

Total Size Limit (with Index) Example

```
$ java -jar target/appconfig-logging-example-*-SNAPSHOT-bootexec.jar \
--spring.profiles.active=rollover \
--logging.file.name=target/logs/mylog.log \
--logging.logback.rollingpolicy.max-file-size=1000 \
--logging.pattern.rolling-file-name='${logging.file.name}.%d{yyyy-MM-dd}.%i'.log \
--logging.logback.rollingpolicy.total-size-cap=10000 ①
...
$ ls -ltr target/logs

total 40                                                ②
-rw-r--r--  1 jim  staff   4.7K Aug  9 12:37 mylog.log.2024-08-09.10.log ①
-rw-r--r--  1 jim  staff   4.7K Aug  9 12:38 mylog.log.2024-08-09.11.log ①
-rw-r--r--  1 jim  staff   2.7K Aug  9 12:39 mylog.log ①
```

① `logging.logback.rollingpolicy.total-size-cap` constrains current plus historical files retained

② historical files with indexes 1 thru 9 were deleted to stay below file size limit

13.15. logging.logback.rollingpolicy.total-size-cap no Index Example

The following example triggers file changeover every second and makes no use of the index because the timestamp pattern is so granular that `max-size` is not reached before the timestamp changes the base. As with the previous example, the files are also aged-off when the total byte count reaches `logging.logback.rollingpolicy.total-size-cap`.

Total Size Limit (without Index) Example

```
$ java -jar target/appconfig-logging-example-*-SNAPSHOT-bootexec.jar \
--spring.profiles.active=rollover \
--logging.file.name=target/logs/mylog.log \
--logging.logback.rollingpolicy.max-file-size=100 \
--logging.pattern.rolling-file-name='${logging.file.name}.%d{yyyy-MM-dd
-HH:mm:ss}.%i'.log \
--logging.logback.rollingpolicy.max-history=200 \
--logging.logback.rollingpolicy.total-size-cap=500 ①
...
$ ls -ltrh target/logs

total 56
-rw-r--r--  1 jim  staff    79B Aug  9 12:44 mylog.log.2024-08-09-12:44:33.0.log ①
-rw-r--r--  1 jim  staff    79B Aug  9 12:44 mylog.log.2024-08-09-12:44:34.0.log ①
-rw-r--r--  1 jim  staff    79B Aug  9 12:44 mylog.log.2024-08-09-12:44:35.0.log ①
-rw-r--r--  1 jim  staff    79B Aug  9 12:44 mylog.log.2024-08-09-12:44:36.0.log ①
-rw-r--r--  1 jim  staff    79B Aug  9 12:44 mylog.log.2024-08-09-12:44:37.0.log ①
-rw-r--r--  1 jim  staff    79B Aug  9 12:44 mylog.log.2024-08-09-12:44:38.0.log ①
-rw-r--r--  1 jim  staff    80B Aug  9 12:44 mylog.log ①
```

① `logging.logback.rollingpolicy.total-size-cap` constrains current plus historical files retained

The `logging.logback.rollingpolicy.total-size-cap` value — if specified — must be larger than the `logging.logback.rollingpolicy.max-file-size` constraint. Otherwise the file appender will not be activated.

Chapter 14. Custom Configurations

At this point, you should have a good foundation in logging and how to get started with a decent logging capability and understand how the default configuration can be modified for your immediate and profile-based circumstances. For cases when this is not enough, know that:

- detailed XML Logback and Log4J2 configurations can be specified — which allows the definition of loggers, appenders, filters, etc. of nearly unlimited power
- Spring Boot provides include files that can be used as a starting point for defining the custom configurations without giving up most of what Spring Boot defines for the default configuration

14.1. Logback Configuration Customization

Although Spring Boot actually performs much of the configuration manually through code, a set of XML includes are supplied that simulate most of what that setup code performs. We can perform the following steps to create a custom Logback configuration.

- create a `logback-spring.xml` file with a parent `configuration` element
  - place in root of application archive (i.e., `src/main/resources` of source tree)
- include one or more of the provided XML includes

14.2. Provided Logback Includes

- defaults.xml - defines the logging configuration defaults we have been working with
- base.xml - defines root logger with CONSOLE and FILE appenders we have discussed
  - puts you at the point of the out-of-the-box configuration
- console-appender.xml - defines the `CONSOLE` appender we have been working with
  - uses the `CONSOLE_LOG_PATTERN`
- file-appender.xml - defines the `FILE` appender we have been working with
  - uses the `RollingFileAppender` with `FILE_LOG_PATTERN` and `SizeAndTimeBasedRollingPolicy`

These files provide an XML representation of what Spring Boot configures with straight Java code. There are minor differences (e.g., enable/disable FILE Appender) between using the supplied XML files and using the out-of-the-box defaults.

14.3. Customization Example: Turn off Console Logging

The following is an example custom configuration where we wish to turn off console logging and only rely on the logfiles. This result is essentially a copy/edit of the supplied base.xml.

logging-configs/no-console/logback-spring.xml

```
<!-- logging-configs/no-console/logback-spring.xml ①
    Example Logback configuration file to turn off CONSOLE Appender and retain all
other
    FILE Appender default behavior.
-->
<configuration>
    <include resource="org/springframework/boot/logging/logback/defaults.xml"/> ②
    <property name="LOG_FILE" value="${LOG_FILE:-${LOG_PATH:-${LOG_TEMP:-
${java.io.tmpdir:-/tmp}}}/spring.log}"/> ③
    <include resource="org/springframework/boot/logging/logback/file-appender.xml"/>
④

    <root>
        <appender-ref ref="FILE"/> ⑤
    </root>
</configuration>
```

① a logback-spring.xml file has been created to host the custom configuration

② the standard Spring Boot defaults are included

③ `LOG_FILE` defined using the original expression from Spring Boot `base.xml`

④ the standard Spring Boot FILE appender is included

⑤ only the FILE appender is assigned to our logger(s)

14.4. LOG_FILE Property Definition

The only complicated part is what I copy/pasted from base.xml to express the `LOG_FILE` property used by the included FILE appender:

LOG_FILE Property Definition

```
<property name="LOG_FILE"
value="${LOG_FILE:-${LOG_PATH:-${LOG_TEMP:-${java.io.tmpdir:-/tmp}}}/spring.log}"/>
```

- use the value of `${LOG_FILE}` if that is defined
- otherwise use the filename `spring.log` and for the path
  - use `${LOG_PATH}` if that is defined
  - otherwise use `${LOG_TEMP}` if that is defined
  - otherwise use `${java.io.tmpdir}` if that is defined
  - otherwise use `/tmp`

14.5. Customization Example: Leverage Restored Defaults

Our first execution uses all defaults and is written to `${java.io.tmpdir}/spring.log`

Example with Default Logfile

```
java -jar target/appconfig-logging-example-*-SNAPSHOT-bootexec.jar  \
--logging.config=src/main/resources/logging-configs/no-console/logback-spring.xml
(no console output)

$ ls -ltr $TMPDIR/spring.log  ①
-rw-r--r--  1 jim  staff  67238 Apr  2 06:42
/var/folders/zm/cskr47zn0yjd0zwkn870y5sc0000gn/T//spring.log
```

① logfile written to restored default `${java.io.tmpdir}/spring.log`

14.6. Customization Example: Provide Override

Our second execution specified an override for the logfile to use. This is expressed exactly as we did earlier with the default configuration.

Example with Specified Logfile

```
java -jar target/appconfig-logging-example-*-SNAPSHOT-bootexec.jar \
--logging.config=src/main/resources/logging-configs/no-console/logback-spring.xml \
--logging.file.name="target/logs/mylog.log" ②
(no console output)

$ ls -ltr target/logs ①

total 136
-rw-r--r--  1 jim  staff  67236 Apr  2 06:46 mylog.log ①
```

① logfile written to `target/logs/mylog.log`

② defined using `logging.file.name`

Chapter 15. Spring Profiles

Spring Boot extends the logback.xml capabilities to allow us to easily take advantage of profiles. Any of the elements within the configuration file can be wrapped in a `springProfile` element to make their activation depend on the profile value.

Example Profile Use

```
<springProfile name="appenders"> ①
    <logger name="X">
        <appender-ref ref="X-appender"/>
    </logger>

    <!-- this logger starts a new tree of appenders, nothing gets written to root
logger -->
    <logger name="security" additivity="false">
        <appender-ref ref="security-appender"/>
    </logger>
</springProfile>
```

① elements are activated when `appenders` profile is activated

See Profile-Specific Configuration for more examples involving multiple profile names and boolean operations.

Chapter 16. Summary

In this module we:

- made a case for the value of logging
- demonstrated how logging frameworks are much better than `System.out` logging techniques
- discussed the different interface, adapter, and implementation libraries involved with Spring Boot logging
- learned how the interface of the logging framework is separate from the implementation
- learned to log information at different severity levels using loggers
- learned how to write logging statements that can be efficiently executed when disabled
- learned how to establish a hierarchy of loggers
- learned how to configure appenders and associate with loggers
- learned how to configure pattern layouts
- learned how to configure the FILE Appender
- looked at additional topics like Mapped Data Context (MDC) and Markers that can augment standard logging events

We covered the basics in great detail so that you understood the logging framework, what kinds of things are available to you, how it was doing its job, and how it could be configured. However, we still did not cover everything. For example, we left topics like accessing and viewing logs within a distributed environment, structured appender formatters (e.g., JSON), etc.. It is important for you to know that this lesson placed you at a point where those logging extensions can be implemented by you in a straight forward manner.