Logging with SLF4J and logback

Introduction

In this article from my free Java 8 course, I will talk about logging. The term *logging* originates from ancient times when a ship's captain used to write down everything that was happening with the ship. (For example, the speed the ship was going and what course the ship was on.) The book he wrote this information into was called a logbook. The *log* part of the word came from a special tool the sailors measured the ship's speed with, which consisted of a reel of string with a log tied onto one end.

As a programmer, you are the "captain" of your code. You want to "write down" – or "log" – events that occur while your program is running. For example, you could print what is going on to the console, to a file or by email. You want to monitor things such as if the system has been hacked, what methods your code is calling, and how efficiently it's working. You could then later go through your logging file and analyze the relevant information to help you improve and debug your code.

Logback

For many years, the most prominent logging framework was Log4j. This has changed. At the moment, LOGBack is used because it's more powerful. LOGBack was developed as an improvement on Log4j by the same developer who created both, Ceki Gülcü.

How do we start logging? First of all, we configure the dependencies using Maven. The configurations requires dependencies org.SLF4J, logback-classic and logback-classic. This is the implementation of SLF4J and is the core code of LOGBack, therefore, we need these three dependencies to get logging into our code.

```
<dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-api</artifactId>
    <version>1.7.5</version>
</dependency>
<dependency>
    <groupId>ch.qos.logback</groupId>
    <artifactId>logback-core</artifactId>
    <version>1.0.13</version>
</dependency>
<dependency>
    <groupId>ch.qos.logback</groupId>
    <artifactId>logback-classic</artifactId>
    <version>1.0.13</version>
</dependency>
```
Example 1

Next, under the resources directory (`src/main/resources`), there should be a file called `logback.xml`, in which we can define certain things, such as how and where logging should be done (see Example 2). In this case, giving the class `ConsoleAppender` will allow the code to be logged directly to the console. There are also other implementations for logging that allow us to log into a file, into a database or into an email (to name just a few). We can also provide our own implementation for new logging sources that don't exist.

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration debug="true">
    <appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">
        <encoder>
            <pattern>%d{HH:mm:ss.SSS} [%thread] %-5level %logger{35} - %msg %n
            </pattern>
        </encoder>
    </appender>

    <logger name="com.marcusbiel.java8course" level="debug"/>

    <root level="INFO">
        <appender-ref ref="STDOUT" />
    </root>
</configuration>
```
Example 2

In this case, we are using the `ConsoleAppender`, so we know the log is being written to the console. But how and when is a message written to the console? You should read the documentation for details, but, briefly, the printed pattern shown in Example 2 gives the time when the event occurred, the thread, and the logging level. The logging level is a sort of "grouping mechanism". We have different levels of logging, which, to some extent, indicate a problem's severity. Logging level `DEBUG`, as the name implies, is used for debugging purposes. Debugging is the process of finding and resolving bugs in your code. `DEBUG` should be used infrequently, since for debugging we could use the debugger included in an IDE or we could write tests. There are also `INFO` and `ERROR` logging levels. Logging level `ERROR` is used if there is an error in the code, and `INFO` is used for general information like for a server or application starting or stopping.

In the `logback.xml` file (under `src/main/resources`), we specify that we want to log on level `DEBUG`, and we provide a package name, which will specify that we are enabling logging for classes under this particular package. We have mentioned the com.marcusbiel.java8course package, but we can extend that and log only for specific classes, for example com.marcusbiel.java8course.CarService. For the level, we could put ERROR instead of DEBUG which would mean "Log it only if it's an error." We could also type INFO, which would mean "Only log it if it's information." INFO includes the info level errors, and DEBUG includes all three levels - debug, info and error. Different levels make the logging mechanism more flexible. Different loggers for different packages could be used as well. For example, we could have a different package like com.marcusbiel.java8course2, and provide a different logging configuration for it. We could then have one package on DEBUG and the other one on ERROR. We must also set a tag for the root level, which is generally set to INFO level.

First of all, let's go to our CarService class and add the logging details, as shown in Figure 3. We need to import org.slf4j.Logger. SLF4J knows to access LOGBack and it will find the dependency of LOGBack classic in the Maven configuration file. We also import org.slf4j.LoggerFactory (A factory is another type of design pattern, by the way. In short, factories allow you to centralize the place of object creation, allowing for cleaner code and preventing code duplication).

```
package com.marcusbiel.java8course;

import org.junit.Test;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class CarService {

    private final Logger log = LoggerFactory.getLogger(CarService.class);

    public void process(String input) {
        if (log.isDebugEnabled()) {
            log.debug("processing car:" + input);
        }
    }
}
```

Example 3

In the CarService class we create a private final Logger and call it log. We then say LoggerFactory.getLogger(..). This is generally how you define a logger. For now, we don't need to worry about how the object is created, as the factory is smart and does all the necessary work on its own. Inside the constructor we pass the Logger constructor the name of the same class in which it was defined, i.e. CarService.class. We used the if-condition check to make sure that our logger was in debug mode before we logged, which is good practice, because otherwise your debugging could result in a decrease in performance. We would be creating strings without actually logging them, which would be using unnecessary hardware resources.

Example 4 shows the test that will use the logging service. Now the logging can be used in the `process()` function of the class `CarService`.

```
package com.marcusbiel.java8course;

import org.junit.Test;

public class CarServiceTest {

     @Test
     public void shouldDemonstrateLogging(){
          CarService carService = new CarService();
          carService.process("BMW");
     }
}
```

Example 4

The input to the `process()` method is given as "`BMW`". This will create a `String` that will be concatenated with "`processing car:`", and will be logged.

In this scenario, it's not so bad to use logging without the check, as all that would happen is that the two strings' arguments would be concatenated. But consider that the input string is of infinite length. This concatenation of two strings, as silly as this may sound, can take a lot of time. Here we just have a very simple, small program with just one debug line. Imagine a thousand users concurrently calling this method, and it taking 200 milliseconds per method call. 200 milliseconds might not seem like a lot of time, but if the method is called thousands of times, this could use a lot of processing power on a server or a PC, leading to a drop in performance. We only want to concatenate the strings and prepare them for logging *if* logging is enabled. If we call a logging method and logging is not enabled, without checking first, the strings will be concatenated, *but not written to the log*. As a result, the concatenation would have been done unnecessarily. This is something we would like to prevent, and is the reason why we use `isDebugEnabled`, `isInfoEnabled`, or `isErrorEnabled`, and so on.

Even though this `String` concatenation logging is already fairly simple with Log4j and SLF4J, there is a smarter, simpler way, as shown below in Figure 5. We can put in curly braces in place of the variables, replace the "+" sign with a comma and then write the input String.

The framework will check if the user wants to log on debug level. If so, it will take this input, convert it to a string, if it's not a string already, and concatenate both strings. Like in Example 3, the strings won't be concatenated unless the system is in debug mode, which helps to keep your program as efficient as possible. Here, however, we don't have to use a conditional to check this since the debug function internally concatenates the `String`. This is stylistically better because it takes up fewer lines, while still providing the same functionality and protection from inefficiencies.

```
public void process(String input) {
    log.debug("processing car: {}", input);
}
```
Example 5

Here we have the input, given from the process method, and we want to print ”`processing car:`” concatenated with the input `String`. In our test we put `BMW` as the input `String`. In our configuration file, `logback.xml`, we set the `com.marcusbiel.java8course` package's log level as `DEBUG`, and the root level as `INFO`. Let's run the test. On level `DEBUG`, the timestamp, the package, the class `CarService` and the `String` "`processing car: BMW`" were logged on the console.

Now, let's try deactivating the debugging logger. Let's say we only want to log on `ERROR` level for the package `com.marcusbiel.javac8ourse`. We make this adjustment in `logback.xml`. Let's execute it again. There will be no `BMW` logged. Now let's change the root level to be `DEBUG`, and change the package, for example `com.marcusbiel.java8course2`. We're not in `com.marcusbiel.java8course2`, we are in the package `com.marcusbiel.java8course`, which means that we should log on the root level which is `DEBUG`. We would expect the test to log again. Let's try it. It logs: "DEBUG `com.marcusbiel.java8course.CarService` – processing car: BMW". Even though we aren't in the package `com.marcusbiel.java8course2`, the root level is defined as `DEBUG`, which is why the `DEBUG` message is shown.

In the `logback.xml` file, there is a `ConsoleAppender` attribute in the appender tag (see Example 1). `ConsoleAppender` is a type of appender that logs to the console which we can also see in our IDE. If we can run this from a terminal, we will see it directly within the terminal.

Logger Documentation

If you look in the Logger documentation, in the section "typical usage pattern", the example code they show (see Example 6 below) will be very similar to what we did in our example. You can log with `logger.isDebugEnabled()`, as shown before, or use the better alternative of the curly braces.

They are very similar, but the first way uses three lines of code that clutter your code, and the other way uses only one line, which makes your code cleaner and simpler. Logging is important, but it should not get in the way of what you are trying to achieve with your code. Be sure to read through the documentation for a better understanding.

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class Wombat {
     final Logger logger = LoggerFactory.getLogger(Wombat.class);
     Integer t;
     Integer oldT;
     public void setTemperature(Integer temperature) {
          oldT = t;
          t = temperature;
          logger.debug("Temperature set to {}. Old temperature was {}.", t, 
oldT);
          if(temperature.intValue() > 50) {
               logger.info("Temperature has risen above 50 degrees.");
          }
     }
}
```

Example 6

Logging on Different Levels

Now we will look at logging on different levels, for example `ERROR`. We're still on `DEBUG` level, and `ERROR` is included within `DEBUG` because `DEBUG` is the most specific logging level. In the traces, we would expect to see `ERROR`, not `DEBUG`, as a prefix for the printed log line, in order to differentiate how important some logging events are. We might log, for example, "Some error occurred", in the case of an error. In our example, "`processing car:`" is not an error, therefore, we should not log that on `ERROR` level. We can also log on `WARN` level, in case of a warning. We would use *warning* for something very close to an error, but where it is not critical to the operation of the system and no one has to physically intervene in any way. `INFO`, for example, could be used when a server or a program starts up, so that the system admin's team knows that everything is going well.

Now, let's set the root level to `ERROR`. The package that we're not using `com.marcusbiel.java8course2`, as well as the root level are now on the `ERROR` level. They are printed to STDOUT, which is the name I gave to the appender. When running that, we expect not to see any logging. We will see the info from LOGBack, but the logging itself has gone. If we change the package back to `com.marcusbiel.java8course` we should see the log message again.

Appenders

Before I end off, I would also like to quickly go over Appenders. If you're interested in reading more, you can check out the appender documentation.
There are different types of Appenders. A few examples are: `ConsoleAppender`, `FileAppender`, `RollingFileAppender`, `DBAppender` and `SMTPAppender`. We have already used `ConsoleAppender` to write to the console. `FileAppender` is used to write into a file. `RollingFileAppender` means that new files are created when the file has a certain size or, for example, a new log file is created at the beginning of each day. This is important for servers, because if we're always writing into the same file, the file size could reach tons of gigabytes. Therefore, `RollingFileAppender` is used on servers very often. `DBAppender` allows us to directly log into a database, and is very easily configured. `SMTPAppender` is used for logging and sending an email.

There are a lot of Appenders that you can use out of the box. As previously mentioned, you can always extend it and write your own Appender, which would allow you to log to any other source.

Thanks for reading!