Notatki - Logowanie do konsoli

Spis treści

SLF4J i Logback

Chciałbym żeby dalsze zgłębianie tematu dotyczącego logowania odbywało się przy wykorzystaniu frameworka `logback`. Może pojawić się pytanie - dlaczego? Osobiście jego widziałem w praktyce najczęściej.

Uspokoję jednocześnie, że z jednej strony wspomnieliśmy już o standaryzacji, czyli `slf4j`, więc wszystko czego się tutaj nauczymy jest uniwersalne. Z drugiej strony jeżeli nauczymy się jak działa `logback` i jak go skonfigurować, to każdy inny framework do logowania będzie dla nas prosty bo działają one w analogiczny sposób. Nie ma to zatem aż takiego znaczenia jaki framework do logowania zostanie tutaj przedstawiony. Z powodów wspomnianych wcześniej padło na `logback`.

Mam nadzieję, że `logback` nie pomyli Ci się z `lombok`. To są 2 różne rzeczy, tak jakby co.

Gradle

Zmieńmy konfigurację `build.gradle` na poniższą:

```
dependencies {
    implementation group: 'ch.qos.logback', name: 'logback-classic', version: '1.2.6'
}
```

Czyli pozbywamy się zależności `slf4j-api` oraz `slf4j-simple`. Sprawdźmy teraz jakie zależności zostaną dodane do naszego projektu przy wykorzystaniu komendy:

```
gradlew dependencies
```

Zobaczymy wtedy, że dzięki dodaniu tej jednej zależności, w naszym projekcie będziemy używać:

```
\--- ch.qos.logback:logback-classic:1.2.6
     +--- ch.qos.logback:logback-core:1.2.6
     \--- org.slf4j:slf4j-api:1.7.32
```

Czyli możemy teraz wykorzystać `logback` z wykorzystaniem **warstwy abstrakcji**, czyli `slf4j`. Zwróć uwagę, że stało się to przy wykorzystaniu tylko jednej zależności `logback-classic`.

Maven

To samo dla Maven:

```
<!-- reszta pliku-->
    <dependencies>
        <!-- https://mvnrepository.com/artifact/ch.qos.logback/logback-classic -->
        <dependency>
            <groupId>ch.qos.logback</groupId>
            <artifactId>logback-classic</artifactId>
            <version>1.2.6</version>
        </dependency>
    </dependencies>
<!-- reszta pliku-->
```

Sprawdźmy teraz jakie zależności zostaną dodane do naszego projektu przy wykorzystaniu komendy:

```
mvn dependency:tree
```

Zobaczymy wtedy, że dzięki dodaniu tej jednej zależności, w naszym projekcie będziemy używać:

```
[INFO] pl.zajavka:example-project-maven:jar:1.0.0
[INFO] \- ch.qos.logback:logback-classic:jar:1.2.6:compile
[INFO]    +- ch.qos.logback:logback-core:jar:1.2.6:compile
[INFO]    \- org.slf4j:slf4j-api:jar:1.7.32:compile
```

Czyli możemy teraz wykorzystać `logback` z wykorzystaniem **warstwy abstrakcji**, czyli `slf4j`. Zwróć uwagę, że stało się to przy wykorzystaniu tylko jednej zależności `logback-classic`.

Logowanie do konsoli

Stwórzmy teraz plik `logback.xml` i umieśćmy go w katalogu `src/main/resources`. Dzięki temu zostanie on umieszczony na `classpath` i zostaną użyte ustawienia z tego pliku. Jeżeli nie wskażemy lokalizacji tego pliku na `classpath`, zostaną wybrane ustawienia domyślne.

logback.xml

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

```
    <appender name="CONSOLE" class="ch.qos.logback.core.ConsoleAppender">
        <encoder>
            <pattern>%d{yyyy-MM-dd HH:mm:ss.SSS} [%t] %-5level %logger{36} - %msg%n</pattern>
        </encoder>
    </appender>

    <root level="error">
        <appender-ref ref="CONSOLE"/>
    </root>

</configuration>
```

Przejdźmy do wyjaśnienia zagadnień związanych z konfiguracją.

Configuration

Jest to tag `xml`, czyli fragment pliku `xml`, w którym określamy konfigurację logowania. Link do dokumentacji.

Appender

`Logback` deleguje zapisywanie logów do komponentów, które noszą nazwę `Appender`. Możemy w nim określić miejsce docelowe zapisywania logów i stąd wyróżniamy takie appendery jak `ConsoleAppender`, `FileAppender` lub `SMTPAppender` (który pozwala na wysyłkę logów mailem). Link do dokumentacji.

- **encoder** - Encoder jest obiektem, który jest odpowiedzialny za formatowanie logów przy ich zapisie. Jeżeli chcemy określić swój format drukowania logów, użyjemy w tym celu pola `pattern`. Format logów może zawierać informacje wypisane poniżej (Link do dokumentacji):
  - **%d{yyyy-MM-dd HH:mm:ss.SSS}** - określa format wydruku daty i czasu,
  - **%t** - nazwa wątku, który akurat wykonuje nasz fragment kodu. Dotychczas rozmawiamy tylko o aplikacjach jednowątkowych, więc nie wyjaśniam, o co chodzi z tymi wątkami.
  - **%level** - poziom logowania `TRACE`, `INFO` itp. `-5` oznacza w tym przypadku, że jeżeli poziom nazwa poziomu logowania zawiera `4` znaki, to uzupełnienie do `5` znaków ma być wypełnione spacjami, minus natomiast mówi, że ma się to stać od prawej strony. Parametr numeryczny pozwala nam zachować stałą szerokość kolumny, aby wydruk był bardziej czytelny, Spróbuj na próbę pozbyć się `-5`.
  - **%logger** - nazwa loggera, który loguje, czyli w praktyce powinna być to nazwa klasy, która loguje dany fragment tekstu. Parametr `{36}` określa długość nazwy loggera. Jeżeli faktyczna nazwy loggera (razem z paczką) będzie dłuższa niż określony parametr, to nazwa ta zacznie być zwijana. Dodaję tabelkę z dokumentacji opisującą jak to będzie wyglądało.
  - **%msg** - umieszcza wiadomość, która faktycznie jest przekazana do zalogowania.
  - **%n** - oznacza przejście do nowej linijki.

Gdy mamy już zdefiniowaliśmy appender, należy jeszcze określić jak będziemy go używać.

Jedna uwaga do wzoru zapisu logów, czyli `pattern`. Pattern jest ustawiany jeden dla całej aplikacji. Czyli to nie jest tak, że jeden developer loguje informacje zgodnie z

> jednym wzorem, a drugi z innym. Wzorzec taki jest plikiem konfiguracyjnym aplikacji i jest zastosowany do całej aplikacji. To zespół decyduje, w jakim wzorze cała aplikacja ma logować.

Root

Root określa główny logger. Definiując `root logger` podajemy poziom logowania, który nas interesuje i który ma być zastosowany do appenderów, które są określone w tagu `root`. Wartość atrybutu `level` może przyjąć wartości: `TRACE`, `DEBUG`, `INFO`, `WARN`, `ERROR`, `ALL` lub `OFF` i **nie jest** case-sensitive. Możemy tutaj dołączyć `0` lub więcej appenderów. Dzięki temu możemy przykładowo określić, że mamy 2 główne appendery, jeden który loguje do konsoli i drugi, który loguje do pliku. Link do dokumentacji. Tag `root` nie wymaga podania atrybutu `name` oraz `additivity`, o których rozpiszemy się przy tagu **logger**.

Zanim przejdziemy dalej, przyjmijmy, że odnosimy się do struktury plików Java jak poniżej:

Klasa Logger1

```
package pl.zajavka.logger1;

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

public class Logger1 {

    private static final Logger LOGGER = LoggerFactory.getLogger(Logger1.class);

    public static void log1() {
        LOGGER.trace("logging in Logger 1");
        LOGGER.debug("logging in Logger 1");
        LOGGER.info("logging in Logger 1");
    }
}
```

Klasa Logger2

```
package pl.zajavka.logger2;

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

public class Logger2 {

    private static final Logger LOGGER = LoggerFactory.getLogger(Logger2.class);

    public static void log2() {
        LOGGER.trace("logging in Logger 2");
        LOGGER.debug("logging in Logger 2");
        LOGGER.info("logging in Logger 2");
    }
}
```

Klasa SLF4JLogging

```
package pl.zajavka;
```

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import pl.zajavka.logger1.Logger1;
import pl.zajavka.logger2.Logger2;

public class SLF4JLogging {

    private static final Logger LOGGER = LoggerFactory.getLogger(SLF4JLogging.class);

    public static void main(String[] args) {
        LOGGER.trace("Hello zajavka!, parametr: {}", "trace");
        LOGGER.debug("Hello zajavka!, parametr: {}", "debug");
        LOGGER.info("Hello zajavka!, parametr: {}", "info");
        LOGGER.warn("Hello zajavka!, parametr: {}", "warn");
        LOGGER.error("Hello zajavka!, parametr: {}", "error");

        Logger1.log1();
        Logger2.log2();
    }
}
```

Logger

Rozszerzmy konfigurację logowania do tej widocznej poniżej:

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

    <appender name="CONSOLE" class="ch.qos.logback.core.ConsoleAppender">
        <encoder>
            <pattern>%d{yyyy-MM-dd HH:mm:ss.SSS} [%t] %-5level %logger{36} - %msg%n</pattern>
        </encoder>
    </appender>

    <logger name="pl.zajavka.logger1" level="debug" additivity="true">
        <appender-ref ref="CONSOLE"/>
    </logger>

    <logger name="pl.zajavka.logger2" level="debug" additivity="false">
        <appender-ref ref="CONSOLE"/>
    </logger>

    <root level="error">
        <appender-ref ref="CONSOLE"/>
    </root>

</configuration>
```

W praktyce używa się taga `root` i ewentualnie dodatkowych tagów `logger`. Nawet w dokumentacji jest taki obrazek (w tej sekcji), który pokazuje, że w pliku konfiguracyjnym możemy mieć od 0 do wielu appenderów, od 0 do wielu loggerów, ale `root` musi być ☺. Piszę o tym, bo jak usuniemy tag root, to i tak będzie to działało, ale tylko do paczek określonych jawnie w nazwie appendera. Dlatego w praktyce podaje się `root`, który ma zastosowanie do wszystkich paczek.

W `logger` możemy określić atrybuty:

Pole name

W name podaje się nazwę paczki, do której ma zostać zastosowany dany `logger`.

Pole level

Poziom logowania, który ma być zastosowany do podanego `loggera`.

Flaga additivity

Flaga, która jest przyjemnie opisana w dokumentacji:

> Appender additivity is not intended as a trap for new users. It is quite a convenient logback feature. For instance, you can configure logging such that log messages appear on the console (for all loggers in the system) while messages only from some specific set of loggers flow into a specific appender.

Flaga ta często powoduje powstanie błędu podwójnego logowania (taki sam błąd wywołałem w konfiguracji określając tę flagę na `true`). Wystarczy wtedy ustawić ją na `false` i problem znika.

Additivity w praktyce jest stosowana np. w przypadku gdy chcemy aby `root` logował zawsze do konsoli i nie wydzielamy oddzielnych loggerów do konsoli (tak jak dodaliśmy loggery dla konkretnych paczek w przykładzie wyżej), a jednocześnie chcemy dodać `logger` zapisujący do pliku. Jeżeli nie ustawilibyśmy additivity na `true` w loggerze zapisującym do pliku, to niektóre wiadomości byłyby logowane tylko do pliku, ale nigdy do konsoli. Logowanie do pliku będzie niedługo.

Additivity można w takiej sytuacji rozumieć analogicznie do dziedziczenia, gdzie `root` jest rodzicem, który również loguje wiadomości wszystkich innych loggerów, pod warunkiem, że nie ustawimy **additivity="false"**. Jeżeli ustawimy **additivity="true"** to daną wiadomość będzie logował i konkretny `logger` i `root`. Stąd wynika dodawanie zduplikowanych wiadomości, jeżeli dodamy `logger` do konsoli z **additivity="true"**.

Na podstawie przykładu wyżej można sprawdzić, że jeżeli nie dodamy loggerów na poszczególne paczki określających poziom `DEBUG`, to loguje się tylko `ERROR` określony w `root`.