Notatki - Logowanie - Proste przykłady

Spis treści

Logowanie w Javie

Poziomy logowania

Poziomy logowania wprowadzają możliwość ustawiania progów, które określają czy dany poziom (priorytet) wiadomości ma się logować, czy też nie. Innymi słowy, poziom logowania mówi, jak ważna jest dana wiadomość, albo patrząc na to z innej strony - jak szczegółowe informacje zawiera.

Poniżej wymieniamy poziomy logowania od najmniej szczegółowego do najbardziej szczegółowego.

Obraz 1. Poziomy logowania

Poziom logowania możemy zatem rozumieć jak poziom szczegółowości zalogowanej informacji. Czyli **ERROR** zawiera najmniej szczegółowe, ale zarazem najbardziej istotne informacje (błędy są najbardziej istotne, bo je w pierwszej kolejności należy naprawić). **TRACE** natomiast zawiera najbardziej szczegółowe informacje, ale ten rodzaj informacji nie jest istotny przy normalnym przebiegu działania aplikacji, czyli normalnie w logach nie włączamy **TRACE**, bo ten poziom zawiera za dużo szczegółów, które nie są nam potrzebne w codziennym życiu.

Trace

`Trace` pokazuje bardzo szczegółowe informacje odnośnie przebiegu działania aplikacji. Używamy tego dosyć rzadko, w sytuacjach, gdy potrzebujemy pełnej widoczności wartości jakie są wyliczane w procesie. Uruchomienie logowania w poziomie `Trace` pokazuje nam bardzo dużo szczegółowych informacji o aplikacji. Najczęściej opisuje kroki tak dokładne, że nie jest to istotne w codziennym użytkowaniu.

```
logger.trace("Logujemy wiadomość z poziomem TRACE");
```

Debug

Mniej dokładny poziom niż `Trace`, natomiast nie jest to ciągle poziom, który powinien być używany w normalnym przebiegu działania aplikacji. Dlatego sama nazwa nazywa się `Debug`. `Debug` powinien być używany do logowania informacji, które są potrzebne przy szukaniu błędów w aplikacji, a nie do przeglądania normalnego jej stanu.

```
logger.debug("Logujemy wiadomość z poziomem DEBUG");
```

Info

Standardowy poziom logowania informacji, który obrazuje normalne działanie aplikacji. Z tym poziomem logujemy informacje o normalnym procesie, który zakończył się sukcesem, czyli np. użytkownik dokonał zamówienia i zakończyło się to sukcesem. Jeżeli natomiast nie interesuje nas zapisywanie normalnego przebiegu działania aplikacji, użyjemy następnego poziomu.

```
logger.info("Logujemy wiadomość z poziomem INFO");
```

Warn

Poziom logowania, który wskazuje stan, w którym wystąpił jakiś problem, ale nie jest to "błąd" i aplikacja nie przestaje działać. Możemy na to patrzeć analogicznie do warninga, który jest pokazywany oprócz błędów kompilacji. Informuje nas to o potencjalnych problemach. Jeżeli interesuje nas faktyczny błąd, użyjemy następnego poziomu.

```
logger.warn("Logujemy wiadomość z poziomem WARN");
```

Error

Poziom logowania wskazujący na błąd w systemie, który uniemożliwia działanie funkcji systemu. Np. bramka do płatności uniemożliwia dokonanie płatności. Albo użytkownik nie jest wstanie zobaczyć ofert sprzedaży produktów, które chcemy w systemie sprzedawać.

```
logger.error("Logujemy wiadomość z poziomem ERROR");
```

Fatal

Oprócz poziomów wymienionych wcześniej, istnieje jeszcze poziom Fatal. Oznacza on błąd krytyczny systemu, np. baza danych nie jest dostępna. W praktyce natomiast ten poziom też jest raczej rzadko stosowany. Co ciekawe, nie wspiera go `Slf4j`, o którym powiemy sobie już niedługo.

All

Ten poziom logowania loguje wszystko co jest tylko zdefiniowane.

Off

Ten poziom logowania nie loguje nic. Innymi słowy ten poziom służy do kompletnego wyłączenia logowania.

Zależności między poziomami

Zależność między tymi poziomami jest taka, że jeżeli ustawimy np. poziom `Debug`, to będą logowały się wszystkie wiadomości z poziomem `Debug` i bardziej istotnym (wyższym), czyli `Debug`, `Info`, `Warn` i `Error`. Jeżeli ustawimy poziom `Warn`, to będą logowały się tylko wiadomości z poziomami `Warn` i `Error`. A jak ustawimy `Error` to tylko `Error`.

Formatowanie logów

Jakie informacje są nam potrzebne, aby dobrze odczytać informacje o przebiegu działania aplikacji:

- Czas wykonania zdarzenia (to na pewno, często z dokładnością do milisekund),
- Miejsce w kodzie, gdzie takie zdarzenie wystąpiło,
- Który wątek wykonał operację (Wiem, że na razie nie wiemy czym jest wątek. Cały czas rozmawiamy o aplikacjach, które operują na jednym wątku, ale w praktyce loguje się taką informację),
- Poziom istotności zalogowanej informacji (`DEBUG`, `INFO`, `WARN`, `ERROR`)
- Możemy ewentualnie nadać kolor konkretnym fragmentom informacji jeżeli wyświetlamy logi w konsoli
- Stacktrace, który często jest również bardzo istotny, aby móc dokładnie prześledzić, które metody były wykonywane do momentu wyrzucenia błędu przez nasz program

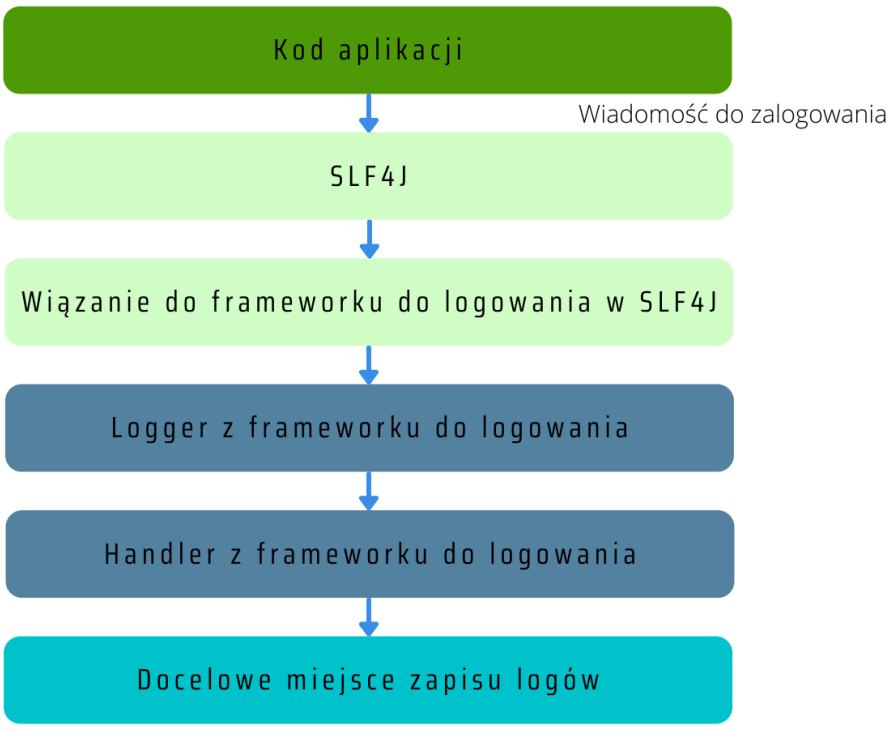
W praktyce format logów można albo przyjąć domyślny oferowany przez framework, albo ustawić

własny w konfiguracji

Wizualizacja poziomu abstrakcji slf4j

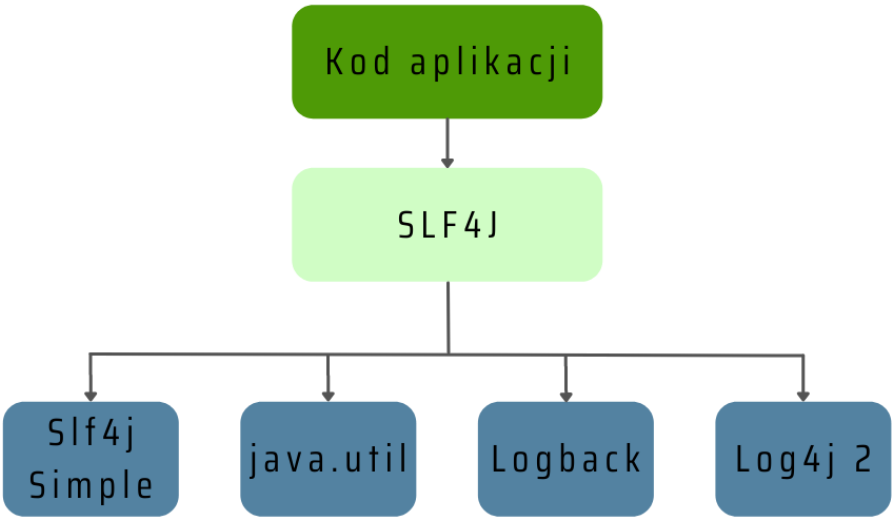
Chcę tu jeszcze raz podkreślić czemu mówię o `slf4j` zamiast zwyczajnie użyć `java.util.logging` i załatwione. Oczywiście nie ma problemu żebyśmy używali w aplikacji `java.util.logging`. To jest filozofia na podobnym poziomie, czy stosować `lombok`, czy pisać wszystko samemu. Równie dobrze moglibyśmy się zastanawiać, czy pisać kod w Intellij czy w notatniku. Natomiast narzędzia powstają w tym celu, żeby nam ułatwiać życie, a `slf4j` ma za zadanie ustandaryzować nam podejście do logowania.

Biblioteki dedykowane do logowania posiadają wbudowane ustawienia, np. prekonfigurowane formattery oraz stawiają na poprawienie wydajności w stosunku do np. `java.util.logging`. Jeżeli zdecydujemy się wybór którejś z bibliotek, w przyszłości możemy mieć problem z tym, że będziemy chcieli taką bibliotekę zmienić. Najlepiej jest wtedy operować na nakładce/fasadzie nad takimi bibliotekami, żeby nie martwić się później, że ewentualna wymiana biblioteki będzie od nas wymagała wielu zmian w kodzie. `Lombok` jest przykładem takiej biblioteki, że jeżeli w przyszłości chcielibyśmy z niego zrezygnować, to musimy przepisać pół projektu ☺. W przypadku frameworków do logowania wygląda to inaczej. W przypadku frameworków do logowania, fasadą jest właśnie **`slf4j`**. Abstrakcyjna wizualizacja działania przy wykorzystaniu `slf4j` wygląda w ten sposób:



Obraz 2. Abstrakcyjnie wyrażony sposób działania SLF4J

Jeżeli natomiast chcielibyśmy sobie zwizualizować to, że `slf4j` jest warstwą abstrakcji, natomiast potrzebujemy konkretnych frameworków, żeby faktycznie to logowanie zrealizować, to możemy spojrzeć na poniższą grafikę.



Obraz 3. SLF4J i frameworki

Przejdźmy w końcu do przykładów z kodem

java.util.logging

Zacznijmy od najprostszego przykładu i wykorzystania `java.util.logging`, spójrzmy na fragment kodu poniżej:

```
package pl.zajavka;

import java.util.logging.Level;
import java.util.logging.Logger;

public class LoggingMain {

    private static Logger LOGGER = Logger.getLogger(LoggingMain.class.getName());

    public static void main(String[] args) {
        LOGGER.log(Level.WARNING, "Hello zajavka!");
        LOGGER.log(Level.INFO, "Hello zajavka!");
    }
}
```

Pamiętasz, jak wspomniałem o drukowaniu przy wykorzystaniu `System.err`? Zwróć uwagę, że wynik jest na czerwono. Zobacz też możliwe wartości dostępne w klasie `Level`. Nie ma tu poziomów takich jak `DEBUG` albo `TRACE`. Ten przykład pokazuje dlaczego warto jest mieć standaryzację w tym obszarze. Jednocześnie też zwróć uwagę, że nie musieliśmy dodawać żadnych bibliotek aby użyć `java.util.logging`.

`Logger` jest inicjalizowany przy wykorzystaniu nazwy klasy. Jest to bardzo popularna praktyka i przydatna. Dzięki temu wiemy, która klasa loguje jakieś informacje. Jednocześnie możemy używać tego loggera w zakresie całej klasy, bo właśnie o to nam chodzi, żebyśmy mieli logger per klasa.

Slf4j

Jeżeli natomiast to samo chcielibyśmy zrobić przy wykorzystaniu `slf4j`, zapis taki wyglądałby w ten

sposób:

```
package pl.zajavka;

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

public class SLF4JLogging {

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

    public static void main(String[] args) {
        LOGGER.trace("Hello zajavka!, parametr: {}", 123);
        LOGGER.debug("Hello zajavka!, parametr: {}", 123);
        LOGGER.info("Hello zajavka!, parametr: {}", 123);
        LOGGER.warn("Hello zajavka!, parametr: {}", 123);
        LOGGER.error("Hello zajavka!, parametr: {}", 123);
    }
}
```

Żeby kod się kompilował, musimy dodać dependencje do projektu.

Gradle

Przykładowo do pliku `build.gradle`:

```
dependencies {
    implementation group: 'org.slf4j', name: 'slf4j-api', version: '1.7.36' ①
    implementation group: 'org.slf4j', name: 'slf4j-simple', version: '1.7.36' ②
}
```

① Ta zależność może być rozumiana jako klasy i metody warstwy abstrakcji jaką jest `slf4j`. Dzieki niej możemy stosować samą w sobie warstwę abstrakcji.

② Ta zależność może być rozumiana jako prosta implementacja frameworku do logowania, który faktycznie loguje informacje na ekranie. Dokumentacja wspomina, że przy wykorzystaniu tej zależności, wszystkie wiadomości będą logowanie z poziomem `INFO` na `System.err`. Później jeszcze do tego wrócimy.

Maven

Ta sama konfiguracja dla Maven:

```
<!-- reszta pliku-->
    <dependencies>
        <!-- https://mvnrepository.com/artifact/org.slf4j/slf4j-api -->
        <dependency>
            <groupId>org.slf4j</groupId>
            <artifactId>slf4j-api</artifactId>
            <version>1.7.36</version>
        </dependency>
        <!-- https://mvnrepository.com/artifact/org.slf4j/slf4j-simple -->
        <dependency>
            <groupId>org.slf4j</groupId>
```

```
            <artifactId>slf4j-simple</artifactId>
            <version>1.7.36</version>
        </dependency>
    </dependencies>
<!-- reszta pliku-->
```

Na ekranie wydrukuje się w tym momencie coś takiego:

```
[main] INFO pl.zajavka.SLF4JLogging - Hello zajavka!, parametr: 123
[main] WARN pl.zajavka.SLF4JLogging - Hello zajavka!, parametr: 123
[main] ERROR pl.zajavka.SLF4JLogging - Hello zajavka!, parametr: 123
```

Zobacz, że w przykładzie kodu pojawiają się już poziomy logowania `TRACE`, `DEBUG`, `INFO`, `WARN` i `ERROR`. Natomiast zapis `{}` określa nam parametr w tekście, który możemy uzupełnić tak jak w przykładzie podawana jest wartość `123`. Zobacz też, że nie wydrukowały nam się na ekranie zapisy z poziomem logowania `TRACE` i `DEBUG`, jest to kwestia konfiguracji. Nie wydrukowały nam się też timestampy wykonania danego fragmentu kodu.

Stacktrace

W jaki sposób zalogować stacktrace? Jest to bardzo proste, wystarczy przekazać złapany wyjątek na końcu parametrów wywołania np. metody `error()`:

```
package pl.zajavka;

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

public class SLF4JLogging {

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

    public static void main(String[] args) {
        try {
            method1();
        } catch (Exception ex) {
            LOGGER.error("Exception was thrown", ex);
        }
    }

    private static void method1() {
        method2();
    }

    private static void method2() {
        method3();
    }

    private static void method3() {
        method4();
    }

    private static void method4() {
        method5();
    }
```

```
    private static void method5() {
        throw new RuntimeException("Throwing some exception!");
    }
}
```

Po uruchomieniu powyższego fragmentu kodu, na ekranie zostanie wydrukowane:

```
19:59:42.370 [main] ERROR pl.zajavka.SLF4JLogging - Exception was thrown
java.lang.RuntimeException: Throwing some exception!
    at pl.zajavka.SLF4JLogging.method5(SLF4JLogging.java:36)
    at pl.zajavka.SLF4JLogging.method4(SLF4JLogging.java:32)
    at pl.zajavka.SLF4JLogging.method3(SLF4JLogging.java:28)
    at pl.zajavka.SLF4JLogging.method2(SLF4JLogging.java:24)
    at pl.zajavka.SLF4JLogging.method1(SLF4JLogging.java:20)
    at pl.zajavka.SLF4JLogging.main(SLF4JLogging.java:12)
```

Znowu o abstrakcji

Dzięki temu, że stosujemy warstwę abstrakcji w postaci `slf4j` zyskujemy:

- jedno API jeżeli pracujemy na wielu projektach - jest zwyczajnie wygodniej jeżeli w kilku projektach możemy używać tych samych klas i metod,
- prosty sposób na używanie frameworka pod spodem, który faktycznie chcemy zastosować,
- prosty sposób na wymianę takiego frameworka jeżeli będziemy mieli taką potrzebę.

SimpleLogger

Jeżeli napisalibyśmy teraz kod pokazany poniżej:

```
package pl.zajavka;

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

public class Main {

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

    public static void main(String[] args) {
        System.out.println(LOGGER.getClass());
    }
}
```

A następnie go uruchomili, to na ekranie wydrukowane zostanie:

```
class org.slf4j.impl.SimpleLogger
```

Tutaj umieszczam link do dokumentacji klasy `SimpleLogger`. Jest to logger, który będzie używany gdy dołączymy zależność `slf4j-simple`. Czyli wracamy do wcześniejszego fragmentu wyjaśniającego

znaczenie zależności `slf4j-simple`. Chciałbym tutaj wspomnieć o kilku kwestiach.

1. **org.slf4j.impl.SimpleLogger** to nie **java.util.logging**. `slf4j` jest warstwą abstrakcji, która standaryzuje nam podejście do logowania, natomiast `java.util.logging` jest frameworkiem do logowania w Javie, który jest dostępny "out of the box". Nie musimy dodawać żadnych zależności.
2. Skoro `slf4j` to warstwa abstrakcji, musimy również zapewnić jakieś powiązanie `slf4j` z frameworkiem, który będzie faktycznie używany do logowania. Określa się to słowem **`binding`**. Pod tym linkiem umieszczam fragment dokumentacji pokazujący przykładowe bindingi do różnych frameworków. Zwróć uwagę, że jest tutaj wymieniony binding `slf4j-simple`. Na potrzeby dalszych przykładów skupimy się natomiast na **logback**.
3. Jeżeli stosujemy `slf4j-simple`, możemy zmieniać ustawienia logowania poprzez zapewnienie pliku **simplelogger.properties**, którego możliwe ustawienia są opisane tutaj. Plik taki musi się znaleźć na `classpath` podczas uruchamiania programu. Nie chcę natomiast przechodzić przez przykłady wykorzystania tego pliku z ustawieniami, bo wolę poświęcić ten czas na omówienie `logback`, który będzie przedstawiony już niedługo.
4. I ostatnia kwestia. Jeżeli nie dodamy zależności `slf4j-simple`, czyli zostawimy tylko zależność `slf4j-api`, na ekranie zobaczymy tekst, który umieszczam poniżej. Opisane jest to również tutaj.

```
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further details.
```