Notatki - Logowanie - Intro

Spis treści

Czym jest logowanie?

Logowanie (**Logging**) jest procesem zapisywania przebiegu wykonania aplikacji w jakimś miejscu. Przebieg działania aplikacji może być drukowany na ekranie w trakcie działania aplikacji lub zapisywany do pliku - mówimy wtedy o logach zapisanych w pliku. Zapisywanie przebiegu na ekranie nie jest w żaden sposób trwałe, bo jak aplikacja przestanie działać i wyłączymy komputer, to tracimy te zapisy. Ewentualnie jeżeli uruchomimy aplikację ponownie to również stracimy zapis z poprzedniego wykonania, który został wydrukowany na ekranie. Jeżeli chodzi o zapis do plików - jest on trwały, przynajmniej do momentu usunięcia takiego pliku 😁.

Logowanie w rozumieniu opisanym powyżej jest czym innym niż logowanie na Facebooka, czyli jest to co innego niż podanie użytkownika i hasła. Niefortunna zbieżność nazw...

Po co się to robi?

Wyobraź sobie, że uruchamiasz program, który liczy coś przez bardzo długi czas. Zostawiasz taki program na noc. W jaki sposób można później prześledzić jego wykonanie w celu ewentualnej diagnozy jakiś błędów? Najlepiej jest zapisać przebieg takiego programu w jakiś trwały sposób, np. do pliku tekstowego.

W praktyce sytuacja staje się o wiele bardziej skomplikowana, gdyż takie pliki z logami muszą być przetrzymywane w jakimś centralnym miejscu (aby móc łatwo znaleźć zapis przebiegu aplikacji z danego dnia i godziny), do tego również powinniśmy mieć możliwość znalezienia w łatwy sposób interesującej nas akcji. To prowadzi do wielu zagadnień, których nie będziemy tutaj poruszać. Skupimy się na tym, co trzeba zrobić, aby nasza aplikacja mogła logować, czyli zapisywać przebieg swojego

działania.

Dostępne rozwiązania

W praktyce spotkasz się z bibliotekami, które zapewniają nam obiekty i metody, pozwalające w prosty sposób określić konfigurację logowania. Biblioteki takie będą również realizowały samą kwestię logowania, czyli zapisywania przebiegu działania aplikacji. Z racji dostępności wielu bibliotek, powstały również narzędzia będące warstwą abstrakcji nad tymi bibliotekami, aby umożliwić prostą wymianę takiej biblioteki pod spodem, jeżeli okaże się, że wynikła w naszym projekcie taka potrzeba.

W tym miejscu należy dodać, że teoretycznie możliwe jest aby takie logowanie realizować w sposób "manualny", "ręcznie", "po swojemu". Wrócimy do tego tematu na jak już poznamy dostępne możliwości.

Framework

Zacznijmy od frameworków i ich nazw, które możesz spotkać na swojej drodze. Zanim jednak to nastąpi wyjaśnijmy różnicę.

Czym różni się library od framework? Klik

- **library** - zestaw funkcji/metod, które można wywołać, które są zorganizowane w postaci klas. Po każdym wywołaniu funkcji z biblioteki to my przejmujemy kontrolę nad dalszym wywołaniem kodu.
- **framework** - jest abstrakcyjnym projektem, który posiada wbudowane zachowania. Aby go używać, wstawiamy swoje własne zachowania/metody w różne miejsca we frameworku np. przez tworzenie własnych klas i wstawianie ich we framework. Następnie kod frameworka wykonuje nasz kod w miejscach, które uzupełniliśmy.

Takie są definicje, natomiast w praktyce jest to używane wymiennie, albo framework mówi się na biblioteki, które są "duże" ☺.

Tutaj znajdziesz link do Wikipedii, gdzie zostały wymienione frameworki do logowania w Java. Służą one do ułatwiania i standaryzacji procesu logowania. Przejdźmy zatem do najważniejszych i najbardziej popularnych z nich.

java.util.logging

Paczka `java.util.logging` zapewnia podstawowe klasy i interfejsy dające możliwość logowania w aplikacji. Są dostępne razem z JDK od Javy w wersji 1.4.

Log4j

Log4j jest bardzo znanym frameworkiem, który można często spotkać w przeróżnych projektach. Zostało jednak ogłoszone, że 5 Sierpnia 2015 `log4j` doszedł do końca swojej przygody i zaleca się update do jego następcy **Log4j 2**. Link

Logback

Początkowo Logback miał być zastępcą `Log4j`. Jest szybszy niż `Log4j`. Zapewnia również wsparcie dla

kontenerów (servlet containers) takich jak `Tomcat` lub `Jetty` (niżej uwaga do tego). Daje również możliwość kompresowania logów (jeżeli mamy pliki z logami, możemy je potem spakować do plików, np. `.zip`). Zapewnia również wbudowaną integrację z **SLF4J**, o którym powiemy zaraz. Składa się z 3 modułów:

- **logback-core** – moduł zawiera podstawowe funkcje logowania,
- **logback-classic** – moduł zawiera m.in. wsparcie `slf4j`,
- **logback-access** – moduł zapewnia integrację z servlet containers (na razie się tym nie martw).

Wiem, że jeszcze nie wiadomo co to Tomcat albo Jetty. Wiem, że brzmi to jak abstrakcja. Spokojnie, będzie w swoim czasie. W skrócie powiem, że będzie nam to potrzebne jak przejdziemy do pisania aplikacji, które dają możliwość komunikacji przez internet.

Pod tym linkiem znajdziesz stronę projektu i dokumentację.

Log4j 2

Log4j 2 jest nowszą wersją swojego poprzednika `Log4j`. Zapewnia ulepszenia, które również zapewnia `logback`. Naprawia również niektóre problemy, które występują w `logback`. Strona projektu

SLF4J

To jest bardzo ważny skrót od słów **Simple Logging Facade**. `SLF4J` jest warstwą abstrakcji, która może zostać "nałożona" na rozwiązania wspomniane wcześniej. Czyli inaczej mówiąc `SLF4J` może być naszym pośrednikiem w użyciu frameworków do logowania. Pozwala to nam jako użytkownikom określić w konfiguracji, którego z frameworków do logowania chcemy używać, natomiast w kodzie stosować warstwę abstrakcji, którą daje nam `slf4j`. Jeżeli w pewnym momencie zdecydujemy, że chcemy zmienić framework do logowania, który jest "pod spodem", nie musimy nic zmieniać w kodzie. Wystarczy, że zmienimy konfigurację projektu i załatwione. Pod tym linkiem znajdziesz stronę projektu i dokumentację.

Jak żyć?

Jak zdecydować co wybrać? To zależy. Od tego, czy trafiamy na nowy projekt, czy może na taki, gdzie ta decyzja została już podjęta.

Jeżeli trafimy do projektu, w którym stosowane są starsze rozwiązania do logowania, warto zastanowić się nad wprowadzeniem warstwy abstrakcji, którą daje nam `slf4j`, a następnie po ujednoliceniu tego w kodzie, nad przepięciem frameworka do logowania pod spodem.

Jeżeli pracujemy nad nowym projektem i musimy wybrać rozwiązanie do logowania, możemy wybrać `logback` lub `log4j 2` i nałożyć na to warstwę abstrakcji w postaci `slf4j`.

Logowanie w Javie

Przejdźmy do omówienia kluczowych klas, które są używane w procesie logowania.

Logger

`Logger` jest obiektem używanym w kodzie do wywoływania metod logujących. Obiekt ten jest z reguły definiowany dla każdej z klas, w których chcemy logować. Dzięki temu mamy zachowany kontekst, która klasa w projekcie dokonała danego zapisu, czyli wyprodukowała daną linijkę w logach. Konfiguracja logowania w aplikacji polega na dodaniu odpowiedniego frameworka do projektu i zdefiniowaniu loggera w klasach, z których chcemy zapisać informację w logach. Definiujemy taki `Logger` jak poniżej, w każdej klasie, z której chcemy logować. Później poznamy sposób jak to skrócić.

```
package pl.zajavka;

import java.util.logging.Logger;

public class LoggingExample {

    private static final Logger logger = Logger.getLogger(LoggingExample.class.getName());
}
```

Handler

`Handler` jest używany do eksportowania logów do interesującego nas miejsca docelowego. Takim miejscem docelowym może być konsola aplikacji, plik na dysku lub jakaś lokalizacja w internecie, z którą musimy się połączyć zdalnie. Możemy wyróżnić dwa standardowe handlery:

- **ConsoleHandler** - drukuje logi na konsolę przy wykorzystaniu `System.err`. Inaczej mówi się o tym, że logi są wyrzucane na wyjście standardowe,
- **FileHandler** - umożliwia zapis logów do pliku.

Handlers możemy włączać lub wyłączać za pomocą konfiguracji. Czyli możemy logować tylko w konsoli, tylko w pliku, albo w obu itp. Oczywiście robimy to przy wykorzystaniu konfiguracji danego frameworka do logowania. Domyślną implementację `ConsoleHandler` umieszczam poniżej. Możemy również zdefiniować własny `Handler` jeżeli będzie nam to potrzebne. Najczęściej stosowane są te standardowe.

Kod źródłowy klasy ConsoleHandler

```
public class ConsoleHandler extends StreamHandler {

    public ConsoleHandler() {
        // configure with specific defaults for ConsoleHandler
        super(Level.INFO, new SimpleFormatter(), null); ①

        setOutputStreamPrivileged(System.err); ②
    }

    @Override
    public void publish(LogRecord record) {
        super.publish(record);
        flush();
    }

    @Override
```

```
    public void close() {
        flush();
    }
}
```

① Ustawiany jest jakiś poziom `Level.INFO`, o tym będziemy niedługo rozmawiać.

② Zwróć uwagę na wykorzystanie `System.err`.

A dlaczego nie zostać przy System.out.println()?

Możesz zadać sobie teraz pytanie, po co to wszystko. Przecież stosowaliśmy wcześniej `System.out.println()`, informacje drukowały się w konsoli i wszystko było w porządku. Dowiedzieliśmy się też jak zapisywać informacje do plików, to po co teraz wymyślamy jakieś frameworki do logowania. Dlaczego nie robimy tego samodzielnie? Powodów jest kilka:

- Musielibyśmy pisać kod do realizacji procesu logowania sami, a po co to robić jeżeli mamy już gotowe biblioteki do tego,
- Jeżeli mamy gotową bibliotekę do robienia czegoś, to warto jest ją zastosować zamiast wymyślać koło na nowo, chociażby z tego powodu, że biblioteka taka rozwiązała już problemy, na które i tak byśmy się natknęli sami. Po zo zatem tracić na to czas,
- Logowanie potrafi znacząco spowolnić aplikacje, bo przecież zapis do pliku zajmuje czas. Jest to dodatkowe wywołanie kodu i wykonanie jakiejś czynności. Dlatego kolejne rozwiązania starają się być szybsze niż poprzednie,
- W pewnym momencie pojawi się taki problem, że nie wszystkie wiadomości chcemy mieć zapisane w pliku, wiadomości w pewnym momencie zaczęłyby mieć priorytet, np. błędy chcemy logować bardzo często (w praktyce zawsze), ale informacje dotyczące normalnego przebiegu programu mogą nas interesować tylko wybiórczo. Pojawi nam się zatem temat priorytetów logowanych wiadomości. W tym właśnie celu stosowane są poziomy logowania, o których niżej.
- Jeżeli będziemy chcieli po prostu wyłączyć zapisywanie informacji na ekranie podczas działania aplikacji, to stosując `System.out.println()` trzeba się trochę pobawić żeby zrobić to w prosty sposób.

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

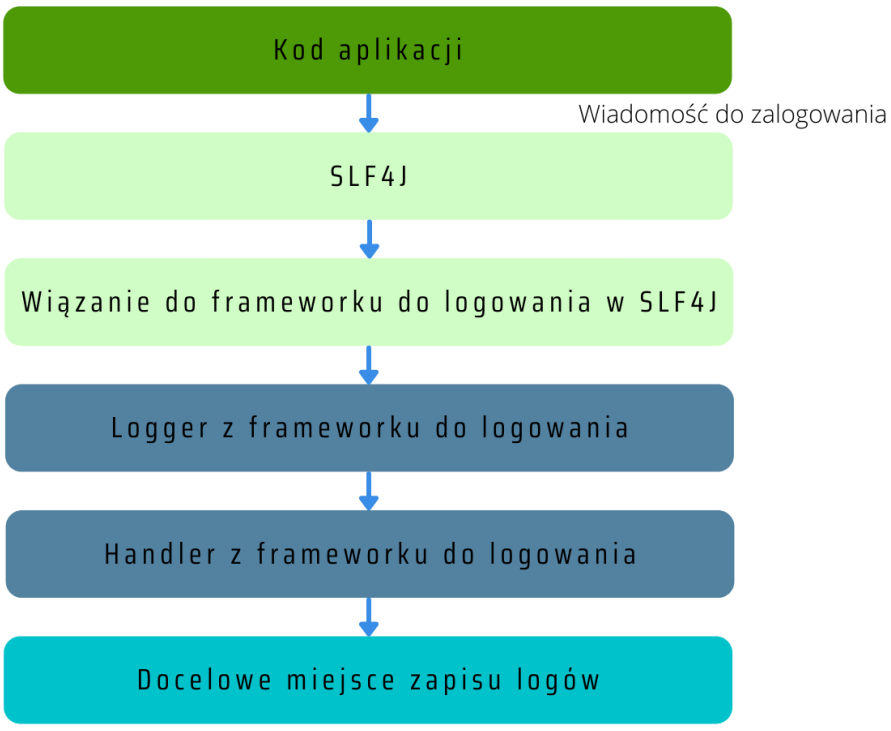
W praktyce format logów można albo przyjąć domyślny oferowany przez framework, albo ustawić

własny w konfiguracji

Wizualizacja poziomu abstrakcji slf4j

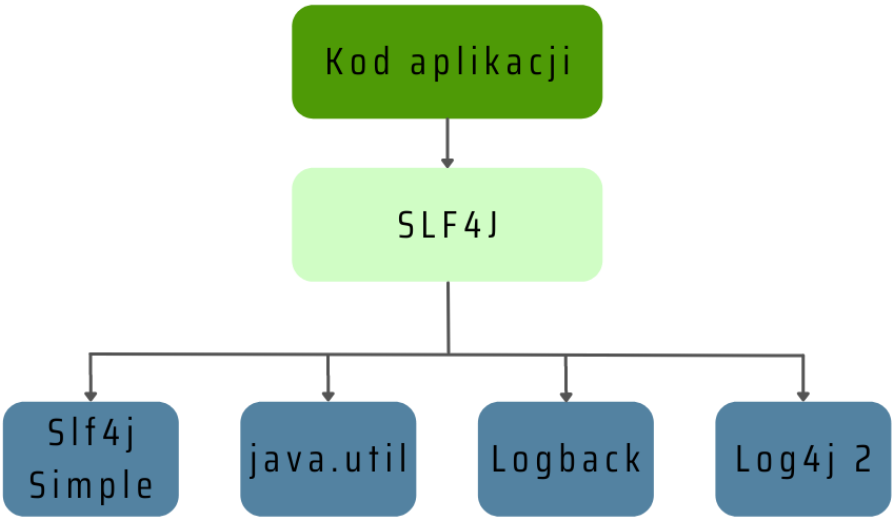
Chcę tu jeszcze raz podkreślić czemu mówię o `slf4j` zamiast zwyczajnie użyć `java.util.logging` i załatwione. Oczywiście nie ma problemu żebyśmy używali w aplikacji `java.util.logging`. To jest filozofia na podobnym poziomie, czy stosować `lombok`, czy pisać wszystko samemu. Równie dobrze moglibyśmy 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.
```

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`.

Notatki - Logowanie do pliku

Spis treści

Logowanie do pliku

Zmieńmy konfigurację `logback.xml` na następującą:

logback.xml

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

    <property name="HOME_LOG" value="${user.dir}/logs/pl.zajavka.log"/>

    <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>

    <appender name="FILE" class="ch.qos.logback.core.FileAppender">
        <file>${HOME_LOG}</file>
        <append>true</append>
        <immediateFlush>true</immediateFlush>
        <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="false">
        <appender-ref ref="FILE"/>
    </logger>

    <root level="warn">
        <appender-ref ref="CONSOLE"/>
```

```
    </root>

</configuration>
```

W przykładzie powyżej dochodzi nam `appender`, który loguje informacje do pliku `pl.zajavka.log` w folderze `logs`, który zostanie utworzony w working directory (`user.dir`). Ustawienia określają, że jeżeli plik już istnieje to mamy do niego dodać logi, a nie nadpisywać (`append=true`). Określa również własny format zapisu logów, co daje nam możliwość innego formatu zapisu logów w konsoli i do pliku.

Ważne jest tutaj, w jaki sposób jest dodany logger `pl.zajavka.logger1`. Logi generowane przez klasy znajdujące się w paczce `pl.zajavka.logger1` są zapisywane tylko do pliku, ale nie będą logowane do konsoli. Wynika to z ustawień parametru **additivity**. Pojawił się również tag `property`, który pozwala na określenie zmiennych, które będziemy używać w pliku z konfiguracją.

Rozszerzenie pliku `.log` jest rozszerzeniem umownym, aczkolwiek Intellij proponuje instalację pluginu, który pomoże nam ten plik przeglądać. W praktyce możemy normalnie otwierać go w notatniku, chociaż polecam w Intellij.

Flaga additivity

Przypomnę jak fajnie `additivity` jest opisane 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.

Additivity w praktyce jest stosowana np. w przypadku gdy chcemy, aby `root` logował zawsze do konsoli i nie wydzielamy oddzielnych loggerów do konsoli, a jednocześnie chcemy dodać `logger` zapisujący do pliku. Jeżeli ustawiliśmy `additivity` na `false` w loggerze zapisującym do pliku, to wiadomości zapisywane w pliku nie pojawiają się w konsoli.

`Logger` z flagą **additivity** można w takiej sytuacji rozumieć jak nadpisanie ustawień określonych w `root` dla konkretnej paczki. W powyższym przykładzie wszystko loguje się w konsoli pod warunkiem ustawienia **additivity="true"**. Jeżeli ustawimy **additivity="false"**, wtedy wiadomości logowane w pliku nie pojawiają się w konsoli. Można to rozumieć w ten sposób, `logger` nadpisuje wtedy informację o logowaniu w konkretnej paczce, przez co dana paczka nie jest logowana przez `root`.

Additivity można również zrozumieć analogicznie do dziedziczenia, gdzie `root` jest rodzicem, który również loguje wiadomości wszystkich innych loggerów, pod warunkiem, że loggery mają ustawione **additivity="true"**. Jeżeli ustawimy **additivity="false"** wtedy taka sytuacja nie ma już miejsca.

Oznacza to, że gdy ustawimy **additivity="true"** to daną wiadomość będzie logował i konkretny `logger` i `root`. Stąd wynika dodawanie zduplikowanych wiadomości, jeżeli dodamy `logger` logujący do konsoli z **additivity="true"**.

Logowanie dużej ilości informacji do plików

Wyobraźmy sobie teraz, że nasz program działa bardzo długo.

W praktyce nie piszemy aplikacji, które są uruchamiane, działają 20 sekund i kończą swoje działanie. Takie oczywiście też mogą być potrzebne, ale najczęściej pisane są aplikacje, które są uruchamiane na serwerze i działają tam 24 godziny na dobę oczekując, aż ktoś je o coś poprosi. Pisząc o coś poprosi mam na myśli inną aplikację np. frontend do naszego backendu, który requestuje o stworzenie jakiegoś zasobu (np. zamówienia dla użytkownika), pobranie jakiegoś zasobu, usunięcie, albo obliczenie czegoś. Aplikacja taka kończy swoje działanie dopiero jak wystąpi błąd krytyczny, albo gdy jest wdrażana jej nowsza wersja. Wtedy uruchamiana jest nowsza wersja i ponownie, działa ona 24h na dobę przez 7 dni w tygodniu. Jak myślisz, dlaczego możesz oglądać filmy na YouTube niezależnie od dnia i godziny? Bo aplikacja YouTube działa 24/h i oczekuje aż ktoś poprosi ją o wyświetlenie filmu ☺.

Wiedząc już, co to oznacza, że program może działać bardzo długo, łatwo się domyślić, że nie możemy cały czas logować zdarzeń do jednego pliku. Wynika to z tego, że taki plik tekstowy będzie miał w końcu zbyt duży rozmiar aby w jakiś sensowny sposób móc go otworzyć.

Zróbmy w tym celu eksperyment. Wywołaj metodę `LoggerLoop.log()`, którą umieszczam poniżej, z poniższymi ustawieniami w pliku `logback.xml`.

Klasa LoggerLoop

```
package pl.zajavka.loggerloop;

// importy

public class LoggerLoop {

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

    private static final Map<Integer, Consumer<Integer>> ACTIONS = Map.of(
        0, i -> LOGGER.debug("some debug message, counter: {}", i),
        1, i -> LOGGER.info("some info message, counter: {}", i),
        2, i -> LOGGER.warn("some warn message, counter: {}", i),
        3, i -> LOGGER.error("some error message, counter: {}", i)
    );

    public static void log() {
        IntStream.rangeClosed(0, 100_000_000)
            .map(i -> i % 4)
            .forEach(key -> Optional.ofNullable(ACTIONS.get(key))
                .orElseThrow(() -> new RuntimeException("Case not handled"))
                .accept(key));
    }
}
```

Wspomniana konfiguracja - plik logback.xml

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

    <property name="HOME_LOG" value="${user.dir}/logs/pl.zajavka.log"/>

    <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>

    <appender name="FILE" class="ch.qos.logback.core.FileAppender">
        <file>${HOME_LOG}</file>
        <append>true</append>
        <immediateFlush>true</immediateFlush>
        <encoder>
            <pattern>%d{yyyy-MM-dd HH:mm:ss.SSS} [%t] %-5level %logger{36} - %msg%n</pattern>
        </encoder>
    </appender>

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

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

</configuration>
```

Możesz teraz zobaczyć, że wywołanie metody `LoggerLoop.log()` wygenerowało plik z logami, którego rozmiar wynosi około 1GB. Oczywiście można bawić się w przetrzymywanie takich rozmiarów plików, ale w praktyce prowadzi to do tego, że bardzo ciężko jest taki plik przeglądać. Warto zatem spróbować taki plik podzielić na części. W tym celu powstał **RollingFileAppender**.

RollingFileAppender

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

    <property name="HOME_LOG" value="${user.dir}/logs/pl.zajavka.log"/>
    <property name="HOME_LOG_ROLLING" value="${user.dir}/logs/pl.zajavka-rolling.log"/>

    <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>

    <appender name="FILE" class="ch.qos.logback.core.FileAppender">
        <file>${HOME_LOG}</file>
        <append>true</append>
        <immediateFlush>true</immediateFlush>
        <encoder>
            <pattern>%d{yyyy-MM-dd HH:mm:ss.SSS} [%t] %-5level %logger{36} - %msg%n</pattern>
        </encoder>
    </appender>

    <appender name="FILE-ROLLING" class="ch.qos.logback.core.rolling.RollingFileAppender">
        <file>${HOME_LOG_ROLLING}</file>

        <rollingPolicy class="ch.qos.logback.core.rolling.SizeAndTimeBasedRollingPolicy">
            <fileNamePattern>logs/archived/pl.zajavka-rolling.%d{yyyy-MM-dd}.%i.log</fileNamePattern>
            <maxFileSize>10MB</maxFileSize>
            <totalSizeCap>1GB</totalSizeCap>
```

```
            <maxHistory>10</maxHistory>
        </rollingPolicy>

        <encoder>
            <pattern>%d{yyyy-MM-dd HH:mm:ss.SSS} [%t] %-5level %logger{36} - %msg%n</pattern>
        </encoder>
    </appender>

    <logger name="pl.zajavka.loggerloop" level="debug" additivity="false">
        <appender-ref ref="FILE-ROLLING"/>
    </logger>

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

</configuration>
```

Podejście 'rolling' polega na tym, że będą tworzone nowe pliki z logami albo gdy obecny plik, do którego loguje aplikacja osiągnie rozmiar określony w parametrach, albo gdy spełnione np. będą warunki daty, czyli np. nastanie nowy dzień. W obu przypadkach, powstanie kolejny plik z logami, a aplikacja będzie logowała do pliku 'bieżącego'. Inaczej mówiąc, aplikacja cały czas loguje do jednego pliku, do momentu gdy zostaną spełnione jakieś warunki. Wtedy zapisane już logi są zapisywane do pliku zarchiwizowanego, a aplikacja loguje dalej do pustego pliku aż do osiągnięcia wspomnianych warunków brzegowych lub limitów.

Klasa wykorzystana w tagu **rollingPolicy**, czyli `SizeAndTimeBasedRollingPolicy` określa jak ma się zachowywać 'rolowanie' plików z logami. Użycie tej klasy oznacza, że warunkami brzegowymi będzie albo data albo rozmiar plików. Czyli pliki z logami będą 'rolowane' albo gdy skończy się np. dany dzień, albo gdy zostanie osiągnięty maksymalny dopuszczalny rozmiar pliku z logami.

Pamiętać należy o tym, że określić inną nazwę pliku dla `FILE-ROLLING` appender, dlatego też wprowadziłem zmienną `HOME_LOG_ROLLING`. Jeżeli tego nie zrobimy, to dostaniemy błąd:

```
'File' option has the same value "C:\Users\krogowski\.../pl.zajavka.log" as that given for appender
[FILE] defined earlier.
```

W tagu **rollingPolicy** określamy teraz warunki, przy których mają być tworzone nowe pliki z datą i numerem pliku danego dnia. Nie są to oczywiście wszystkie ustawienia, ale bardziej przydatne jakie możemy wyróżnić to:

- **%i** - indeks kolejnego pliku, który jest tworzony przy wykorzystywaniu rolowania. Parametry `%d` oraz `%i` są wymagane.
- **fileNamePattern** - określamy w tym wzorze, że nazwa pliku, który jest **archiwizowany** ma zawierać w sobie datę i numer pliku danego dnia,
- **maxFileSize** - każdy zarchiwizowany plik może mieć maksymalnie rozmiar określony w konfiguracji,
- **totalSizeCap** - ustawienie określające, że jeżeli sumaryczna wielkość wszystkich zarchiwizowanych plików przekroczy rozmiar określony w konfiguracji mają zostać kasowane te stare,
- **maxHistory** - pliki mają być przechowywane maksymalnie przez określoną ilość dni, oczywiście w

tym czasie musi działać nasza aplikacja, bo jak nie będzie działać to nikt tego nie usunie.

Encoder i **pattern** już znamy. Wiemy też już, że aby ten appender został użyty, należy wymienić go w którymś z loggerów, albo w root, podając:

```
<appender-ref ref="FILE-ROLLING"/>
```

Możemy też narzucić, że chcemy aby `logback` automatycznie kompresował nam pliki archiwalne np. do formatu `.gz`. W tym celu zmieniamy wpis **fileNamePattern** na poniższy (dołożyliśmy `.gz`):

```
<fileNamePattern>logs/archived/pl.zajavka-rolling.%d{yyyy-MM-dd}.%i.log.gz</fileNamePattern>
```

Jeżeli natomiast nie chcesz czekać całego dnia żeby zobaczyć, że logi faktycznie rolują się przy zakończonym okresie czasu, możesz użyć sekund. W tym celu wykorzystaj poniższe ustawienie:

```
<fileNamePattern>logs/archived/pl.zajavka-rolling.%d{s}.%i.log</fileNamePattern>
```

I wracamy do Lomboka

Co? Po co?

Wracamy do `Lomboka`, bo daje nam on pewne uproszczenie.

Jeżeli dodamy zależności do **lombok** i do **logback-classic**, możemy wtedy zastosować taki zapis:

```
package pl.zajavka;

import lombok.extern.slf4j.Slf4j;
import pl.zajavka.logger1.Logger1;
import pl.zajavka.logger2.Logger2;

@Slf4j
public class SLF4JLogging {

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

Czyli stosując adnotację `@Slf4j` otrzymujemy zachowanie identyczne jak gdybyśmy mieli kod napisany jak poniżej.

```
public class SLF4JLogging {
```

```
    private static final Logger log = org.slf4j.LoggerFactory.getLogger(SLF4JLogging.class);

    // reszta kodu
}
```

Znowu filozoficznie

Tak jak cały czas wspominam, to developer decyduje co i ile loguje. A czemu filozoficznie?

Czy jak dużo logujemy, to kod jest mniej czytelny?

I tak i nie. Z jednej strony logowanie jest kodem, który musisz napisać ręcznie, więc "nie wnosi" on żadnych kroków do logiki biznesowej działania programu, a zajmuje miejsce. Na tej podstawie można sądzić, że kod jest przez to mniej czytelny. Z drugiej strony po jakimś czasie przyzwyczajasz się już do tego stopnia, że nie przeszkadza Ci "ta gorsza czytelność" i traktujesz logi jak coś normalnego.

Czy pisać kod do logowania od razu, czy na końcu?

W sumie to każdy robi to inaczej. Z jednej strony jak tworzysz jakiś proces w kodzie, to jesteś na bieżąco i możesz wtedy łatwo wywnioskować, które informacje będą później najbardziej przydatne w diagnozie z logów.

Z drugiej strony, życie już niejednokrotnie pokazało, że te informacje, które Ci się wydaje, że będą przydatne na etapie diagnozy błędu, wcale Ci nie pomagają, czyli zalogowałeś/-aś za mało potrzebnych i za dużo niepotrzebnych rzeczy. W związku z tym logowane informacje często się poprawia/dostosowuje, gdy wystąpi jakiś błąd. Wtedy deweloper, który diagnozuje błąd, może jednoznacznie stwierdzić, jakie informacje o danym procesie są potrzebne do znalezienia błędu, a jakie były kompletnie nieprzydatne. Wtedy taki logujący kod się refactoruje.

Musisz pamiętać, że logi są dla "Ciebie z przyszłości" i to sobie będziesz dziękować albo to siebie będziesz mieć ochotę udusić, jak w aplikacji wystąpi jakiś błąd, a Ty będziesz mieć tylko szczątkowe informacje, żeby móc przeprowadzić diagnozę.

Co logować?

Wszystko, co pomoże Ci w późniejszej diagnozie. Jak zalogujesz za mało, to możesz nie znaleźć przyczyny błędu. Jeżeli z kolei zalogujesz za dużo, to będziesz mieć problem ze znalezieniem przyczyny, bo będzie to jak szukanie igły w stogu siana. Niestety to jest jeden z tych elementów wiedzy programistycznej, który musisz wyczuć w praktyce.

Każdy proces jest inny i do każdego procesu będziesz podchodzić indywidualnie.

Przykład

Spójrzmy natomiast na przykład, co i jak moglibyśmy zalogować. Wyobraź sobie, że tworzysz funkcjonalność, w której firma *Krzak* w sklepie internetowym sprzedaje produkty dedykowane dla konkretnego klienta na podstawie jego wcześniejszych zachowań. Czyli firma *Krzak* ma stworzony

algorytm, który na podstawie zachowań klienta w sklepie, jest w stanie zaproponować mu produkty, co do których jest największe prawdopodobieństwo, że klient je kupi. Czyli np. klient *Adam Kowalski, clientId = 1454412* zobaczy propozycję zakupu produktów: *rower*, *monitor* oraz *piłka*, natomiast drugi klient *Robert Biernacki, clientId = 9922123* zobaczy propozycję zakupu produktów: *teczka*, *chusteczki* oraz *kabel USB*. Jak w takiej sytuacji moglibyśmy logować?

```
import java.util.ArrayList;@Slf4j
@AllArgsConstructor
class ProductOfferingService {

    private final PredictionService predictionService;

    public List<Product> getProductsList(final Long clientId) {
        // kod, który umożliwia pobranie proponowanych produktów
        try {
            var products = predictionService.predictProducts(clientId);
            log.info(
                "Offering: [{}] products for clientId: [{}]",
                products.size(), clientId); ①
            log.debug(
                "Offering: [{}] products for clientId: [{}]",
                products.stream().map(Product::getId).collect(toList()), clientId); ②
            log.trace(
                "Offering: [{}] products for clientId: [{}]",
                products, clientId); ③
            return products;
        } catch (Exception ex) {
            log.warn("Unable to retrieve products for clientId: [{}]", clientId, ex); ④
            return new ArrayList<>(); // albo 'throw ex';
        }
    }

}
```

① Z poziomem `info` logujemy tylko ilość produktów proponowanych klientowi oraz id klienta, dla którego te produkty są proponowane.

② Z poziomem `debug` logujemy listę id produktów, które zostały zaproponowane klientowi o konkretnym id klienta.

③ Z poziomem `trace` logujemy wszystkie szczegóły dotyczące produktów, które zostały zaproponowane klientowi o konkretnym id klienta.

④ W momencie, gdy wystąpi błąd pobrania przykładowych produktów dla klienta, możemy zdecydować (na podstawie wymagań biznesowych) jak krytyczny błąd to jest. Jeżeli błąd **umożliwia** klientowi dalsze działanie z aplikacją (czyli nie udało się pobrać proponowanych produktów, ale klient jest w stanie skończyć to, po co wszedł do naszego sklepu), to możemy wtedy zalogować taki błąd jako `warn`. Jeżeli natomiast błąd **uniemożliwia** klientowi dalsze działanie z aplikacją (czyli nie udało się pobrać proponowanych produktów i w związku z tym klient nie może skończyć tego po co wszedł do naszego sklepu), to wtedy logujemy taki błąd jako `error` i robimy *rethrow* wyjątku.

A co z wydajnością aplikacji?

Czy duża ilość logów ma wpływ na wydajność aplikacji? Zapoznaj się z tym wątkiem.

Podsumowanie

Mam nadzieję, że udało mi się przedstawić w pigułce, ile problemów może nieść ze sobą poprawne skonfigurowanie zapisywania przebiegu działania aplikacji. Oczywiście przedstawione przypadki są proste i całe szczęście dosyć często spotykane, natomiast w praktyce mogą pojawić się inne - bardziej złożone.

Wiedząc już jakie problemy staraliśmy się rozwiązać, możesz jeszcze raz spróbować odpowiedzieć na sobie pytanie, dlaczego w praktyce nie używa się `System.out.println()` 😀.