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()` 😀.