

JAVA LOGGERS



Hello!

Rafał Misiak

Java Developer dla Stibo Systems (DK) w Ciklum

slack: @rafalmisiak

rafalmisiak@gmail.com

1. Logowanie

Śledzimy działanie aplikacji

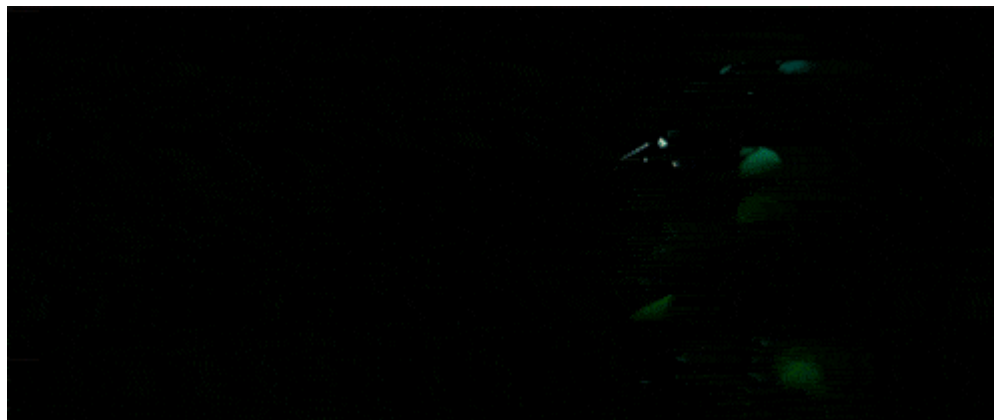
Logowanie

Dlaczego to takie ważne?

- Logi to jedna z ważniejszych rzeczy w czasie działania aplikacji, nie wpływająca bezpośrednio na funkcjonalność
- W chwili awarii lub błędnego zachowania aplikacji ułatwiają namierzenie bezpośredniej lub pośredniej przyczyny incydentu jak również pozwalają na śledzenie działania aplikacji
- Logowanie odbywa się za pośrednictwem Loggerów.

Dziennik logów

Kiedy zaglądam do dziennika po awarii



thecodinglove.com

- Logger przekazuje komunikat do wybranego miejsca docelowego (standardowe wyjście, plik, zdalny serwer logów)
- Każdy komunikat opatrzone jest poziomem, czasem wystąpienia zdarzenia oraz dodatkowymi informacjami opisującymi zdarzenie

- Najczęstszym przypadkiem jest logowanie do pliku co jest jedną z najdroższych operacji (zapis/odczyt)
- Własna próba tworzenia loggera może skończyć się spowolnieniem działania aplikacji lub utratą kontroli nad kolejnością logowanych zdarzeń (async)
- Biblioteki 3rd party radzą sobie z problemem kosztu obsługi pliku jak również asynchroniczności nieblokującej działania aplikacji

Grafana

Przykład wizualizacji logów

Logować możemy najróżniejsze zdarzenia: koniecznie błędy ale również każde logowanie użytkownika, informację o pełnym załadowaniu się serwisu, założonych kontach i rozkładzie tych operacji w poszczególnych godzinach, itp.

<http://play.grafana.org>

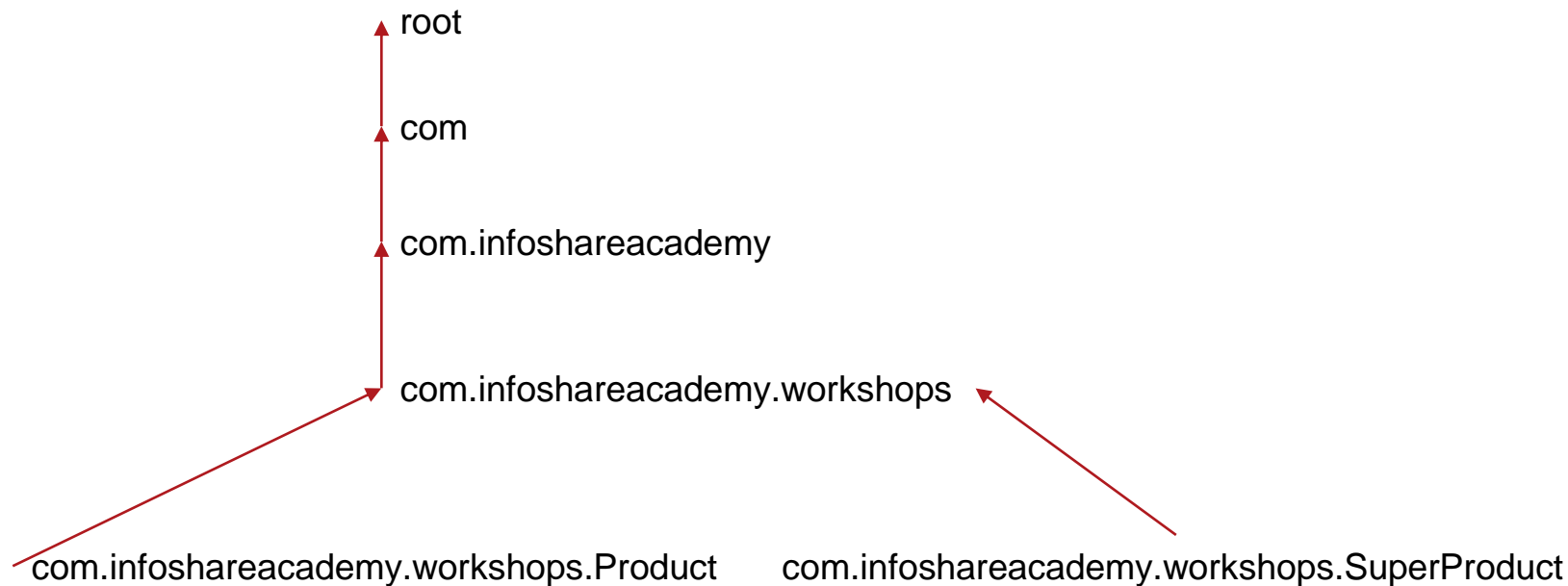
Logger

Nazewnictwo i hierarchia

- Każdy logger ma swoją nazwę – najczęściej jest to nazwa klasy (pełna z nazwą pakietu), w której jest on użyty
- Pełna nazwa klasy określa hierarchię loggerów
Logger o nazwie: *com.infoshareacademy.domain.Product* należy do hierarchii *com*, *com.infoshareacademy* ale nie należy do *org.springframework*
- Dla każdej hierarchii możemy stosować osobną konfigurację co uelastycznia podejście do tematu logów (na przykład w pliku mogą łądować tylko logi z danej hierarchii)

Logger

Nazewnictwo i hierarchia



Poziomy logowania

- Poziom logowania to nic innego jak „ważność” danej wiadomości zapisywanej w logach
- Każda biblioteka ma różne zestawy poziomów:
„FATAL, ERROR, WARNING, INFO, DEBUG, TRACE”
„SEVERE, WARNING, INFO, CONFIG, FINE, FINER, FINEST”

Kiedy zaglądam do dziennika logów na produkcji



W oparciu o poziomy logowania

- Poziomy mogą się przydać w celu:
 - Wysłania maila/zdarzenia na logstash jeśli zalogowany został poziom np. FATAL/SEVERE
 - Zapisywania w dzienniku logów o poziomach wyższych niż wybrany

Implementacja loggera

- Istnieje kilka bibliotek do obsługi logów. Bliżej przyjrzymy się tym najbardziej popularnym:
 - `java.util.logging`
 - `log4j 2`
 - `slf4j` (tylko/aż zbiór fasad)

Repozytorium GitHub

```
git clone  
https://github.com/infoshareacademy  
/jjdd2-materialy-loggery
```

2.

java.util.logging

Podstawowa biblioteka logowania

Słowem wstępu

java.util.logging

- Stanowi standard języka Java – nie są wymagane żadne dodatkowe zależności w celu użycia biblioteki
- Wyróżnia się poziomami logowania: CONFIG, FINE, FINER
- Każda wiadomość poprzedzona jest wartością enum dt. poziomu wpisu

Poziomy logowania

java.util.logging.Level

- **ALL** – wszystkie poziomy będą logowane
- **OFF** – logi zostaną wyłączone dla wszystkich poziomów
- **SEVERE** – najwyższy poziom logowania, przede wszystkim powinni być informowani administratorzy aplikacji oraz w rozsądny sposób końcowi użytkownicy
- **WARNING** – informowani powinni być końcowi użytkownicy jak również opiekunowie systemu, wiadomość wskazuje na potencjalny problem
- **INFO** – używany do logowania istotnych wiadomości dla użytkowników końcowych

Poziomy logowania

java.util.logging.Level

- **CONFIG** – informacja o wartościach konfiguracji, np. typ procesora, wybrany look and feel, poziom wykorzystywany szczególnie przy debugu
- **FINE** – poziom który powinien być wykorzystywany do logowania informacji skrajnie deweloperskich, na przykład potencjalne problemy wydajnościowe
- **FINER** – bardziej szczegółowe informacje dla deweloperów jak wywołania, zwracane wartości czy treści wyjątków
- **FINEST** – najbardziej szczegółowe informacje dla deweloperów

Poziomy logowania

logging.properties

Najniższy dopuszczalny poziom logowania opisany jest w pliku konfiguracyjnym:
JRE_HOME/lib

Należy zmienić równocześnie dwie pozycje:
.level= INFO
java.util.logging.ConsoleHandler.level = INFO

Logi z poziomem niższym niż wskazany nie będą respektowane.

Dlaczego domyślnie Level.INFO ?



thecodinglove.com

Inicjalizacja

java.util.logging.Logger

```
private final Logger logger =  
    Logger.getLogger(  
        getClass().getName()  
    );
```

Użycie

java.util.logging.Logger

```
logger.log(  
    Level.INFO,  
    „Processing {0} elements”,  
    list.size()  
);
```

Hierarchia loggerów

Każdy Logger śledzi „swojego” rodzica – Logger, który jest najbliższym w przestrzeni nazw Loggerów.

Uzależnienie loggera od nazwy pakietu:

```
protected final Logger logger1 =  
    Logger.getLogger(  
        getClass().getPackage().getName()  
    );
```


Hierarchia loggerów

Loggery są kojarzone w ramach namespace'ów:

```
Logger logger = Logger.getLogger("");
```

```
Logger logger1 = Logger.getLogger("com");
```

```
Logger logger2 =  
Logger.getLogger("com.infoshareacademy");
```

```
Logger logger3 = Logger.getLogger(  
"com.infoshareacademy.domain"  
);
```

Effective Level

```
logger.setLevel(Level)
```

Ustawia minimalny poziom z jakim można logować wiadomości wykorzystując tę instancję.

Póki jawnie nie zostanie ustawiony poziom, to jest on dziedziczony od rodzica.

Logger Localization

Istnieje możliwość zmiany lokalizacji dla tekstów serwowanych przez loggera w zależności od języka użytkownika.

Do tej operacji używamy ResourceBundle

```
private static Logger logger =  
Logger.getAnonymousLogger("messages");
```

Przy założeniu, że istnieje plik messages.properties

java.util.logging

ćwiczenie

1. Utwórz pakiet:
`com.infoshareacademy.workshops.logging`
2. Stwórz klasę `LoggingHierarchy` w nowym pakiecie
3. Stwórz dwa loggery dziedziczące po sobie w jednej klasie
4. W ramach nowej metody tej klasy zaloguj liczby parzyste loggerem podstawowym na poziomie `WARNING` a liczby nieparzyste parentem wcześniej wspomnianego loggera na poziomie `SEVERE`
5. Przedział logowanych liczb to `[1..10]`
6. Do zalogowania wiadomości parentem, wykorzystaj `getParent()`

java.util.logging

ćwiczenie cd

7. Do wiadomości doklej informację o nazwie użytego loggera
8. Uruchom aplikację, zapisz wynik
9. Zakomentuj linię odpowiedzialną za inicjalizację loggera parenta
10. Uruchom aplikację, porównaj wynik

3. log4j v2

Pierwsza propozycja 3rd party

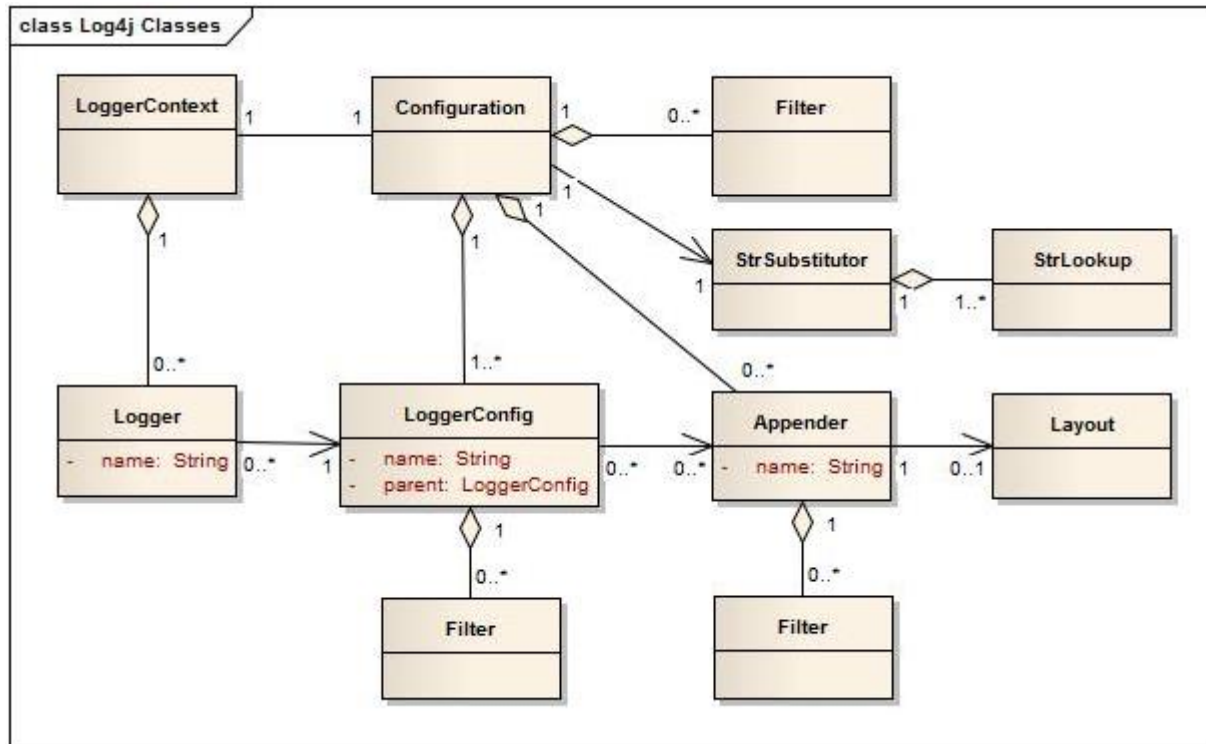
Maven

Zacznijmy od zależności

```
<dependency>  
  <groupId>org.apache.logging.log4j</groupId>  
  <artifactId>log4j-core</artifactId>  
  <version>2.9.0</version>  
</dependency>
```

```
<dependency>  
  <groupId>org.apache.logging.log4j</groupId>  
  <artifactId>log4j-api</artifactId>  
  <version>2.9.0</version>  
</dependency>
```

Architektura log4j2



log4j2

opis działania

- Aplikacja odwołuje się do **LogManager** po instancję **Logger** z konkretną nazwą
- **LogManager** zlokalizuje odpowiedni **LoggerContext**, a następnie przydzieli i zwróci instancję **Logger**
- Jeśli **Logger** jeszcze nie istnieje, zostanie utworzony i powiązany z **LoggerConfig**
- **LoggerConfig** tworzony jest z instancji **Logger** w pliku konfiguracyjnym. Służy również do obsługi **LogEvents** i delegowania ich do **Appenders**

- Logger „root” jest przypadkiem wyjątkowym, zawsze istnieje i jest na samej górze hierarchii loggerów
- Przykład inicjalizacji loggerów:

```
Logger logger =  
LogManager.getLogger(LogManager.ROOT_LOGGER_  
NAME);  
Logger logger = LogManager.getRootLogger();
```

log4j2

konfiguracja

Istnieje kilka sposobów konfiguracji loggera:

- Użycie konfiguracji opisanej jako XML, JSON, YAML
- Zaprogramowany, na przykład tworząc fabrykę konfiguracji i implementując konfigurację

log4j2

konfiguracja

Domyślnie, log4j2 sprawdza wartość właściwości systemowej: **log4j.configurationFile**. To właśnie w tym miejscu spodziewa się ścieżki do pliku konfiguracyjnego:

```
System.setProperty("log4j.configurationFile", "log4j.xml");
```

lub jako VM Options:

```
-Dlog4j.configurationFile=log4j.xml
```

log4j2

konfiguracja domyślna

Jeśli log4j2 nie znajdzie pliku konfiguracyjnego rozpocznie poszukiwania na classpath:

- log4j2-test.properties
- log4j2-test.yaml lub log4j2-test.yml
- log4j2-test.json lub log4j2-test.jsn
- log4j2-test.xml
- log4j2.properties
- log4j2.yml lub log4j2.yaml
- log4j2.json lub log4j2.jsn
- log4j2.xml

log4j2

konfiguracja domyślna

Jeśli nie zostanie znaleziony żaden z wcześniej wymienionych plików, zostaje zastosowana domyślna konfiguracja (DefaultConfiguration) z domyślnym zachowaniem:

- Logger: ROOT
- Level: ERROR
- Target: Console
- Format: %d{HH:mm:ss.SSS} [%t] %-5level %logger{36} - %msg%n

log4j2

konfiguracja zaprogramowana

Istnieje alternatywna metoda konfiguracji Loggerów log4j2. Można ją zrealizować za pomocą kodu.

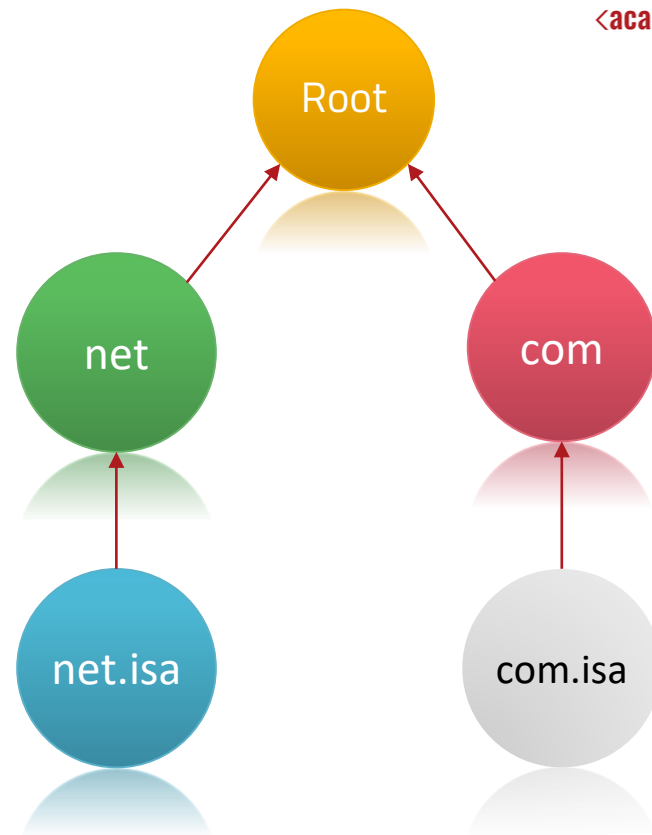
W tym procesie pomocne będą:

- *ConfigurationFactory*
- *LoggerContext*
- *Configuration*
- *ConsoleAppender*
- *LoggerConfig*

log4j2

Additive w hierarchii logów

Domyślnie, log4j2 jest addytywne. Oznacza to, że wszystkie nadrzędne loggery również zostaną użyte.



Levels

Wartości numeryczne

LEVEL	WEIGHT
OFF	0
FATAL	100
ERROR	200
WARN	300
INFO	400
DEBUG	500
TRACE	600
ALL	Integer.MAX_VALUE

Levels

Wartości niestandardowe

Istnieje możliwość stworzenia własnego, niestandardowego poziomu logów:

```
final Level VERBOSE = Level.forName("VERBOSE", 550);
```

```
logger.log(VERBOSE, "a verbose message");
```

log4j2

Appenders

Do tej pory używaliśmy tylko konsoli jako miejsca do którego wysyłane są wszystkie logi. Log4j2 posiada wiele innych możliwości, nazywanych appenderami:

ConsoleAppender, AsyncAppender, FailoverAppender, FileAppender, FlumeAppender, JDBCAppender, JMSAppender, JPAAppender, MemoryMappedFileAppender, NoSQLAppender, OutputStreamAppender, RandomAccessFileAppender, RewriteAppender, RollingFileAppender, RollingRandomAccessFileAppender, RoutingAppender, SMTPAppender, SocketAppender, SyslogAppender

Appenders

File

Appender plikowy <File> może być skonfigurowany do działania w trybie asynchronicznym. Dzięki takiej konfiguracji, w przypadku zablokowania działania loggera (brak dostępu do pliku, przepełniona kolejka itp.) aplikacja nie zatrzyma swojego działania.

```
<File name="MyFile" fileName="logs/app.log">
```

```
....
```

```
<Async name="Async">
```

```
    <AppenderRef ref="MyFile"/>
```

```
</Async>
```

log4j2

Filters

Nawet w przypadku istnienia kandydata LoggerConfig do obsłużenia danego zdarzenia logowania, istnieje możliwość skonfigurowania odrzucenia zdarzenia. Realizuje się takie zachowanie z pomocą filtrów.

Przykładowo: do konsoli chcemy logować każde zdarzenie jakie wystąpi w czasie działania aplikacji jednak w bazie danych/pliku wystarczy nam log, np. tylko z początku operacji oraz informujący o jej zakończeniu.

Filtrowanie liczby wystąpień

BurstFilter

level – poziom do odfiltrowania

rate – dozwolona liczba wystąpień zdarzeń na sekundę

maxBurst – maksymalna liczba wystąpień zanim filtrowanie przekroczenia średniego wystąpienia zostanie aktywowane

onMatch – akcja do wykonania kiedy nastąpiło dopasowanie (ACCEPT, DENY, def NEUTRAL)

onMismatch – akcja do wykonania kiedy nie nastąpiło dopasowanie (ACCEPT, DENY, def NEUTRAL)

Inne przykłady Filtrów

ThresholdFilter – filtr który akceptuje logi o zadanym poziomie lub bardziej szczegółowym. Jeśli ustawiony jest na ERROR, wówczas poziomy DEBUG będą traktowane jako mismatch

TimeFilter – filtr, który akceptuje logi z określonej pory dnia

Script – wykonuje skrypt, akceptacja logu jest w zależności od zwróconego true/false

RegexFilter – konfrontuje wiadomość z regexem, dopasowuje te wiadomości, które pasują do wzoru

log4j2

Layouts

Appender używa Layout'ów do sformatowania wiadomości loga w taki sposób aby pasował on do wyjściowego mechanizmu składania logów.

Layouts Pattern

Realizowane za pomocą: `PatternLayout`
Elastyczny layout konfigurowany za pomocą kombinacji znaków przypominających wyrażenia regularne.

Wrażenie zapisujemy w parametrze „pattern”.

Opis wszystkich dozwolonych opcji:
<https://logging.apache.org/log4j/2.0/manual/layouts.html>

Pattern

Przykład

```
<PatternLayout pattern="%d{HH:mm:ss.SSS} [%t] %-  
5level %logger{36} - %msg%n"/>
```

Doprowadzi do wyjścia:

```
logger.debug("one={}, two={}, three={}", 1, 2, 3);
```

```
20:10:54.488 [main] DEBUG pattern-layout - one=1,  
two=2, three=3
```

Layouts

CSV

Realizowane za pomocą: CsvParameterLayout
Tworzy log w formacie CSV.

Użycie:

```
logger.info("Ignored", value1, value2, value3);
```

```
logger.info(new ObjectArrayMessage("Ben", "Jack",  
"Philip"));
```

```
logger.debug("one={}, two={}, three={}", 1, 2, 3);
```

Layouts

JSON

Realizowane za pomocą: `JsonLayout`
Wyjście formatowane do struktury JSON.

Wybrane parametry:

`compact` = jeśli `true`, to appender nie użyje żadnych znaków nowej linii jak i wcięć

`includeStacktrace` – jeśli `true`, przy każdym logowaniu `Throwable` będzie zalogowany stack trace (`def true`)

log4j2

ćwiczenie

Przygotuj loggera, który będzie:

1. Logował wszystkie logi do konsoli
 - a) kolorując jednocześnie poziomy domyślnymi kolorami
 - b) skracając długość nazwy poziomu do 3 znaków
2. Logował te same logi o poziomie INFO do pliku tekstowego

4. slf4j

Simple Logging Facade

| slf4j dependency

```
<dependency>  
  <groupId>org.slf4j</groupId>  
  <artifactId>slf4j-api</artifactId>  
  <version>1.7.25</version>  
</dependency>
```

slf4j fasada

Slf4j nie jest w żadnym stopniu mechanizmem logowania w aplikacji. To nie jest kolejna biblioteka zawierająca klasy obsługujące logowanie.

Jest to zbiór interfejsów wystawionych do użycia. Slf4j wykrywa zdeployowany mechanizm logowania i „podłącza” się do niego.

slf4j

biding frameworks

- log4j
- log4j2
- java.util.logging
- NOP
- simple
- jcl
- Logback

Szczegóły dt zależności:

<https://www.slf4j.org/manual.html>

slf4j

na czym polega

Tworzona aplikacja jest całkowicie niezależna od systemu logowania. Wykorzystując slf4j, kod dostosowujemy do użycia interfejsów/fasad, a nie do właściwej implementacji konkretnej biblioteki logów. W takiej sytuacji, w dowolnej chwili, z relatywnie niskim kosztem jesteśmy w stanie wymienić jeden framework logowania na inny bez wprowadzania istotnych zmian w kodzie.

5. IOStreams

log4j

 **log4j**

OutputStream / PrintWriter

Istnieje również możliwość skonfigurowania loggera tak aby w czasie zapisu do streamów jednocześnie następowało logowanie z wykorzystaniem wybranego loggera.



Thanks!!