

---

# Pengenalan Logging



# Pengenalan Logging

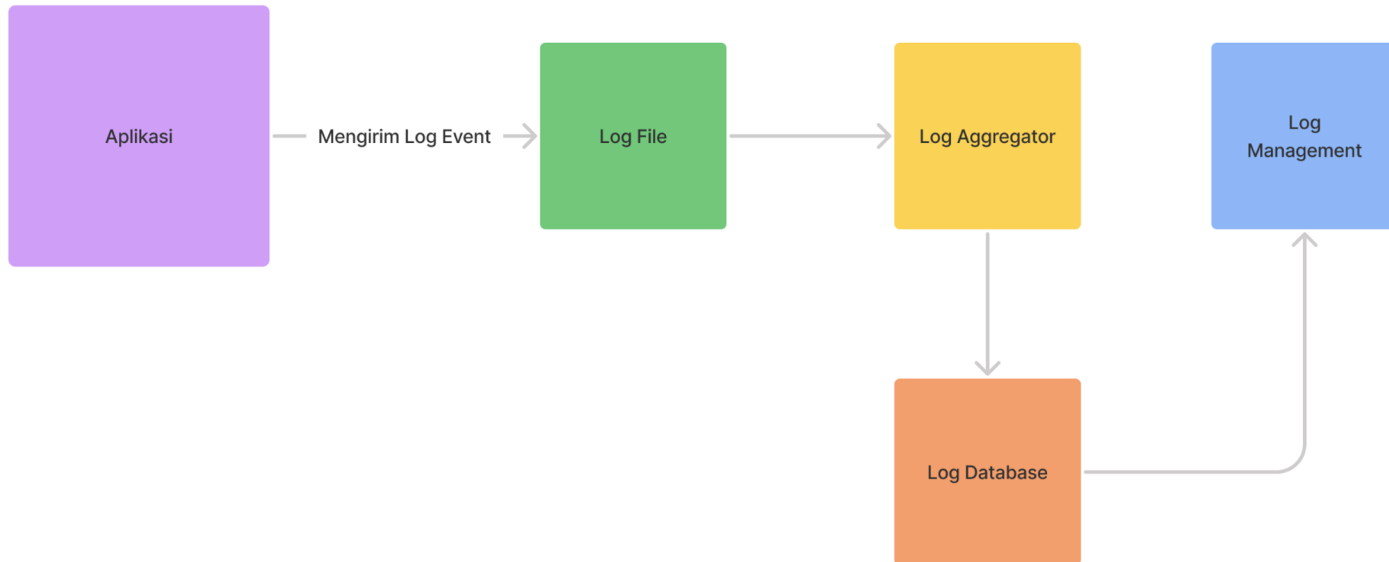
- Log file adalah file yang berisikan informasi kejadian dari sebuah sistem
- Biasanya dalam log file, terdapat informasi waktu kejadian dan pesan kejadian
- Logging adalah aksi menambah informasi log ke log file
- Logging sudah menjadi standard industri untuk menampilkan informasi yang terjadi di aplikasi yang kita buat
- Logging bukan hanya untuk menampilkan informasi, kadang digunakan untuk proses debugging ketika terjadi masalah di aplikasi kita



# Diagram Logging



# Ekosistem Logging



---

# Logging Library



# Java Logging

- Java sendiri sebenarnya memilih package yang dikhususkan untuk logging
- Namun saat ini, kebanyakan programmer tidak menggunakannya
- Hal ini dikarenakan penggunaannya yang kurang flexible
- <https://docs.oracle.com/javase/8/docs/api/java/util/logging/package-summary.html>



# Logging Library

Diluar Java Logging, banyak sekali library yang bisa kita gunakan untuk logging, seperti :

- Apache Common Log
- Apache Log4J
- Logback
- dan lain-lain



# SLF4J

- Pada kelas ini kita akan menggunakan SLF4J
- SLF4J merupakan framework logging yang digunakan seperti API, dimana kita bisa berganti-ganti implementasi logging framework nya
- SLF4J banyak sekali digunakan oleh programmer karena sangat flexible untuk berganti-ganti library logging, tanpa harus melakukan perubahan di kode program nya
- Kita hanya perlu memilih library yang akan digunakan, dan secara otomatis SLF4J akan menggunakan implementasi library tersebut
- <http://www.slf4j.org/>





## Diagram SLF4J





# Logback

- Untuk implementasi library logging nya, kita akan menggunakan Logback
- Logback merupakan salah satu library logging yang populer di Java
- Terutama Logback digunakan secara default di framework Spring Boot, salah satu framework paling populer di Java
- <http://logback.qos.ch/>

---

# Membuat Project



# Membuat Project

- <https://start.spring.io/>

—

Logger



# Logger

- Logger adalah class utama untuk melakukan logging
- Saat kita membuat Logger, biasanya kita akan menyebutkan nama Logger nya
- Biasanya nama logger menggunakan nama class lokasi Logger tersebut
- Hal ini agar mudah ketika melihat hasil log, dari mana hasil log tersebut
- <http://www.slf4j.org/apidocs/org/slf4j/Logger.html>



# Membuat Logger

- Untuk membuat Logger, kita tidak perlu membuat objectnya manual menggunakan new
- Kita bisa memanfaatkan factory class LoggerFactory
- <http://www.slf4j.org/apidocs/org/slf4j/LoggerFactory.html>



## Kode : Hello Log

```
public class MainApp {  
  
    private final static Logger log = LoggerFactory.getLogger(MainApp.class);  
  
    public static void main(String[] args) {  
        log.info("Hello Log");  
        log.info("Selamat Belajar Logging");  
    }  
}
```



—

Level



# Level

- Log memiliki banyak level
- Level disini merupakan level jenis informasi log
- Level itu bertingkat, semakin tinggi artinya informasinya semakin penting
- Level juga biasanya disesuaikan dengan jenis errornya
- Setiap level memiliki method di logger, sehingga kita bisa gunakan level langsung di method nya



## Level (dari terendah ke tertinggi)

Level	Keterangan
trace	Biasanya untuk menambahkan informasi tracing
debug	Biasanya untuk menambahkan informasi debug
info	Biasanya untuk menambahkan informasi
warn	Biasanya untuk menambahkan peringatan
error	Biasanya untuk menambahkan error



## Kode : Level

```
private final Logger log = LoggerFactory.getLogger(LevelTest.class);
```

```
@Test
```

```
void testLevel() {  
    log.trace("Trace");  
    log.debug("Debug");  
    log.info("Info");  
    log.warn("Warn");  
    log.error("Error");  
}
```

---

# Log Format



# Log Format

- Kadang kita ingin menggunakan parameter saat melakukan logging
- Biasanya kita akan membuat string concat untuk membuat pesan logging nya
- Namun SLF4J sudah menyediakan log format
- Kita bisa menggunakan beberapa method overloading nya



## Log Method

Method	Keterangan
<code>level(String)</code>	Melakukan logging berisi string
<code>level(String, Object...)</code>	Melakukan logging dengan parameter, gunakan <code>{}</code> sebagai parameter nya
<code>level(String, Throwable)</code>	Melakukan logging dengan menambah stack trace error



## Kode : Log Format

```
private final Logger log = LoggerFactory.getLogger(LogFormatTest.class);

@Test
void testLogFormat() {

    log.info("Without Parameter");
    log.info("{} + {} = {}", 10, 10, (10 + 10));
    log.error("Ups", new NullPointerException());
}
```



---

# Configuration



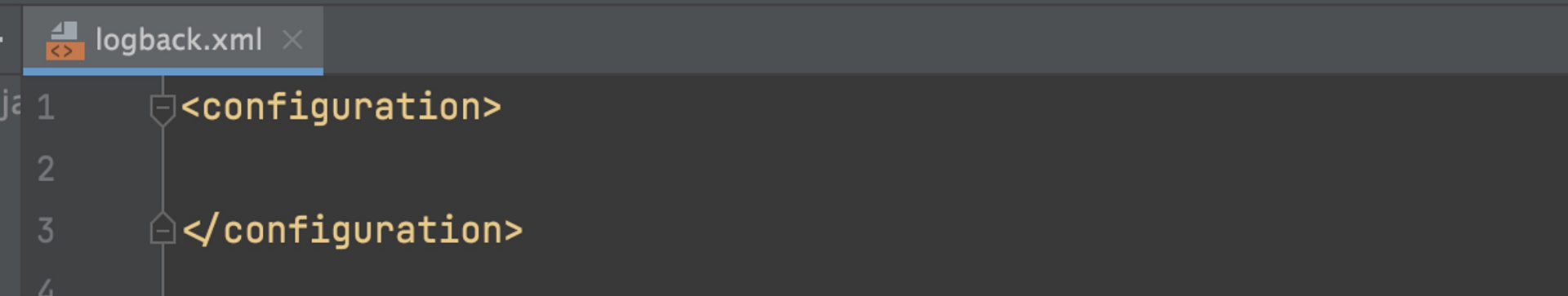
# Configuration

- Secara default, saat kita menggunakan logback, kita tidak butuh configuration
- Namun kadang-kadang kita ingin menggunakan configuration
- Misal, ketika di laptop kita ingin menjalankan logging sampai ke level trace, namun ketika production misal kita hanya butuh di level warning agar tidak terlalu banyak log
- Hal tersebut, perlu kita lakukan dengan membuat file konfigurasi



# logback.xml

- Logback akan membaca konfigurasi dari file logback.xml yang terdapat di default package
- Artinya kita butuh membuat file logback.xml pada default package



```
1 <configuration>
2
3 </configuration>
```

The screenshot shows a code editor with a tab labeled 'logback.xml'. The code content is as follows:



# Level Configuration

- Secara default, saat kita membuat file konfigurasi, logback akan membaca level yang harus dikeluarkan dari file logback
- Jika tidak ada, maka otomatis tidak akan keluar
- Oleh karena itu, hal pertama yang perlu kita lakukan adalah, menambahkan logger level, untuk memberitahu level mana yang ingin kita keluarkan di log file
- <https://gist.github.com/khannedy/e2ec8fcb20344523f3d9e2a8fcc108d7>

## Kode : Root Logger

logback.xml

```
1 <configuration>
2   <root level="debug">
3     <appender class="ch.qos.logback.core.ConsoleAppender">
4       <encoder>
5         <pattern>%d{HH:mm:ss.SSS} [%thread] %-5level %logger{36} - %msg%n</pattern>
6       </encoder>
7     </appender>
8   </root>
9 </configuration>
```



# Logger Package

- Kadang kita ingin membuat logging level berbeda-beda untuk package
- Misal untuk package framework, kita ingin gunakan warn, tapi untuk package aplikasi kita, kita ingin gunakan info
- Secara default, sebelumnya kita sudah buat root, root adalah default fallback semua package level ketika tidak dibuatkan konfigurasi secara spesifik
- Namun jika kita ingin, kita juga bisa membuat logger level lebih dari satu
- Tinggal gunakan prefix package nya saja, artinya semua package di dalamnya akan ikut logger level tersebut
- <https://gist.github.com/khannedy/63ac69c2be341c02d4b0a28e19490639>



## Kode : Package Logger

```
<logger name="programmerzamannow" level="trace"/>

<root level="warn">
  <appender class="ch.qos.logback.core.ConsoleAppender">
    <encoder>
      <pattern>%d{HH:mm:ss.SSS} [%thread] %-5level %logger{36} - %msg%n</pattern>
    </encoder>
  </appender>
</root>
```

---

# Appender





# Appender

- Saat kita melakukan logging, kita bisa menentukan destinasi log file yang akan dibuat, atau dinamakan Appender
- Logback sudah menyediakan banyak sekali appender, jadi sebenarnya kita tidak perlu membuat appender secara manual
- <http://logback.qos.ch/apidocs/ch/qos/logback/core/Appender.html>



# Console Appender

- Appender yang paling sederhana adalah Console
- Dimana appender ini hanya meneruskan log event yang kita kirim menggunakan logger ke dalam console atau System.out
- ConsoleAppender sangat cocok ketika aplikasi yang kita buat di bungkus dalam docker atau kubernetes misalnya, karena kita cukup menampilkannya di console, dan secara otomatis log bisa diambil oleh docker dan kubernetes



# File Appender

- FileAppender merupakan appender yang mengirim log event ke file
- FileAppender sangat cocok ketika kita masih menggunakan VM untuk deploy aplikasi kita
- Jadi semua log event akan disimpan di file
- <https://gist.github.com/khannedy/6afed2b90e55fd2b66b7d7d0a6d6a0ad>



## Kode : File Appender

```
<appender name="FILE" class="ch.qos.logback.core.FileAppender">
  <file>application.log</file>
  <encoder>
    <pattern>%d{HH:mm:ss.SSS} [%thread] %-5level %logger{36} - %msg%n</pattern>
  </encoder>
</appender>

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



# RollingFileAppender

- Kadang saat menyimpan semua log event di file, lama-lama file tersebut akan terlalu besar
- Logback menyediakan RollingFileAppender, yaitu appender yang menyimpan data nya di file, namun kita bisa lakukan rolling, artinya per waktu tertentu akan dibuatkan file baru
- Selain itu kita bisa juga set maksimal ukuran file nya, sehingga ketika sudah mencapai batas maksimal, akan dibuatkan file baru lagi
- Ini lebih direkomendasikan untuk digunakan dibanding menggunakan FileAppender
- <https://gist.github.com/khannedy/f128a692bc00102503eae6a44838720>



## Kode : Rolling File Appender

```
<appender name="FILE" class="ch.qos.logback.core.rolling.RollingFileAppender">
  <file>application.log</file>
  <rollingPolicy class="ch.qos.logback.core.rolling.SizeAndTimeBasedRollingPolicy">
    <!-- rollover daily -->
    <fileNamePattern>application-%d{yyyy-MM-dd}.%i.log</fileNamePattern>
    <!-- each file should be at most 100MB, keep 60 days worth of history, but at most 20GB -->
    <maxFileSize>100MB</maxFileSize>
    <maxHistory>60</maxHistory>
    <totalSizeCap>20GB</totalSizeCap>
  </rollingPolicy>
  <encoder>
```



## Dan Masih Banyak Appender Lainnya

- DBAppender <http://logback.qos.ch/apidocs/ch/qos/logback/classic/db/DBAppender.html>
- OutputStreamAppender  
<http://logback.qos.ch/apidocs/ch/qos/logback/core/OutputStreamAppender.html>
- SMTPAppender <http://logback.qos.ch/apidocs/ch/qos/logback/classic/net/SMTPAppender.html>
- SockerAppender  
<http://logback.qos.ch/apidocs/ch/qos/logback/classic/net/SocketAppender.html>
- SyslogAppender  
<http://logback.qos.ch/apidocs/ch/qos/logback/classic/net/SyslogAppender.html>



# Layout





# Layout

- Layout adalah komponen dalam logback yang digunakan untuk melakukan transformasi dari LogEvent menjadi String
- <http://logback.qos.ch/apidocs/ch/qos/logback/core/Layout.html>



# PatternLayout

- Secara default, layout di logback menggunakan PatternLayout
- PatternLayout merupakan layout yang memiliki banyak pattern yang bisa kita gunakan untuk menampilkan representasi String dari log event
- <http://logback.qos.ch/apidocs/ch/qos/logback/access/PatternLayout.html>



# Conversion Word

- Untuk melihat lebih detail dari apa saja yang bisa kita gunakan dalam pattern layout, kita bisa liat dokumentasinya
- <http://logback.qos.ch/manual/layouts.html#conversionWord>

---

# Mapped Diagnostic Context



# Mapped Diagnostic Context

- Saat kita membuat aplikasi, biasanya aplikasi kita akan diakses oleh banyak sekali user, dan artinya mungkin bisa diakses oleh banyak thread
- MDC merupakan fitur seperti thread local, dimana kita bisa memberi informasi tambahan kepada logger, tanpa harus kita kirim data tersebut secara manual ke class atau method
- Agar lebih dapat gambaran besarnya, kita akan coba sebuah kasus



## Contoh Aplikasi

- Saat membuat aplikasi, kadang kita ingin melakukan log seperti request-id
- Dan di semua log, kita akan tambahkan request-id, agar tahu, log ini muncul dari request siapa
- Biasanya yang kita lakukan adalah, kita akan mengirim request-id dari class pertama sampai class terakhir, biasanya via parameter



# Tugas

- Buat class Controller, Service dan Repository
- Controller akan memanggil Service, dan Service akan memanggil Repository
- Di tiap method tambahkan parameter requestId, dan di tiap method, log informasi requestId tersebut



## Kode : Tanpa MDC

✓ Tests passed: 1 of 1 test - 220 ms

```
/Users/khannedy/Tools/jdk-11.0.2.jdk/Contents/Home/bin/java ...
```

```
15:34:21.915 [main] INFO p.logging.MyController - 12345 : controller save
```

```
15:34:21.918 [main] INFO programmerzamannow.logging.MyService - 12345 : service save
```

```
15:34:21.919 [main] INFO p.logging.MyRepository - 12345 : repository save
```





# Menggunakan MDC

- MDC mirip dengan Map, dimana kita bisa menambahkan data dengan key, dan juga menghapusnya
- Saat kita menggunakan MDC, secara otomatis data di MDC bisa kita akses di logger
- <http://www.slf4j.org/apidocs/org/slf4j/MDC.html>



## Kode : Konfigurasi MDC

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



## Kode : Menggunakan MDC

```
MDC.put( key: "requestId", val: "12345");

MyController controller = new MyController(new MyService(new MyRepository()));
controller.save();

MDC.remove( key: "requestId");
```



# Multi Thread

- Data MDC disimpan dalam ThreadLocal, artinya selama didalam thread yang sama, kita bisa mengakses data di MDC
- Oleh karena itu jika kita membuat aplikasi berbasis multithread, selama satu user mendapat satu thread, kita bisa menggunakan MDC
- Namun jika aplikasi kita sudah reactive, yang tidak jelas thread mana yang memproses method mana, kita tidak bisa menggunakan fitur MDC lagi



## Kode : MDC di Multi Thread

```
MyController controller = new MyController(new MyService(new MyRepository()));

for (int i = 0; i < 10; i++) {
    new Thread(() → {
        MDC.put( key: "requestId", UUID.randomUUID().toString());

        controller.save();

        MDC.remove( key: "requestId");
    }).start();
}
```