Logging with SF4L and Logback

J.Serrat

102759 Software Design

November 3, 2015
## Sources

What’s logging

Logging means writing messages somewhere (console, files, a database...) to record the trace of an application execution. Normally for

- debugging, on/offline
- record user interaction (e.g. web server application)

Goals:

- learn why to log
- how to log with SLF4J+Logback classic, ~ standard for Java
- review its key concepts: loggers, appenders and filters (pending: layouts)
As personal choice, we tend not to use debuggers beyond getting a stack trace or the value of a variable or two. One reason is that (...) we find stepping through a program less productive than thinking harder and adding output statements and self-checking code at critical places.

Clicking over statements takes longer than scanning the output of judiciously-placed displays. It takes less time to decide where to put print statements than to single-step to the critical section of code, even assuming we know where that is.

More important, debugging statements stay with the program; debugging sessions are transient.

Logging versus debugger

Advantages

- logging provides **precise context** (where, when, sequence of events) about a run of the application
- once inserted into the code, generation of logging output requires no human intervention
- log output can be saved in persistent medium to be studied later
- logging frameworks are simpler and easier to learn and use than debuggers
Logging versus debugger

Advantages

- logging provides **precise context** (where, when, sequence of events) about a run of the application
- once inserted into the code, generation of logging output requires **no human intervention**
- log output can be **saved** in persistent medium to be studied later
- logging frameworks are **simpler and easier** to learn and use than debuggers

Drawbacks

- can slow down an application
- may be too verbose
- advanced uses need learn how to configure
Logging versus plain output

Why don’t simply generate output with `System.out.println()`?

We want more flexibility:

- first and foremost, output messages above some selectable priority level
- output messages for all or only certain modules or classes
- control how these messages are formatted
- decide where are they sent
Frameworks

Main frameworks in Java

- native `java.util.logging`, not much used

https://en.wikipedia.org/wiki/Java_logging_framework
Frameworks

Main frameworks in Java

- native `java.util.logging`, not much used
- Log4J: *de facto* standard until a few years

https://en.wikipedia.org/wiki/Java_logging_framework
Frameworks

Main frameworks in Java

- native `java.util.logging`, not much used
- **Log4J**: *de facto* standard until a few years
- **Logback**: successor of Log4J created by the same developer, used in many projects now

https://en.wikipedia.org/wiki/Java_logging_framework
Frameworks

Main frameworks in Java

- native `java.util.logging`, not much used
- **Log4J**: *de facto* standard until a few years
- **Logback**: successor of Log4J created by the same developer, used in many projects now
- **SLF4J** Simple Logging Façade for Java: façade pattern to some backend logger framework like Log4J or Logback

https://en.wikipedia.org/wiki/Java_logging_framework
Frameworks

Main frameworks in Java

- **native** `java.util.logging`, not much used
- **Log4J**: *de facto* standard until a few years
- **Logback**: successor of Log4J created by the same developer, used in many projects now
- **SLF4J** Simple Logging Façade for Java: façade pattern to some backend logger framework like Log4J or Logback
- **tinylog**: minimalist (75 KB Jar) logger for Java, optimized for ease of use. Output to console, file, JDBC, rolling files with many policies . . .

https://en.wikipedia.org/wiki/Java_logging_framework
SLF4J

Simple Logging Façade for Java

- simple façade or abstraction for various logging frameworks, such as java.util.logging, Logback or Log4j
- programmer plugs in the desired logging framework at deployment time
- they are exchangeable: you can readily switch back and forth between logging frameworks
- SLF4J-enabling your library/application implies the addition of a single mandatory dependency, slf4j-api-1.7.12.jar (as of 2015)
- if no binding is found on the class path, SLF4J will default to a no-operation
**Simplest usage:**

- include library slf4j-api-1.7.12.jar
- bind to Simple implementation slf4j-simple-1.7.12.jar
- outputs all events to System.err
- levels ERROR > WARN > INFO > DEBUG
- only messages of level INFO and higher are printed
Simplest usage:

- include library `slf4j-api-1.7.12.jar`
- bind to Simple implementation `slf4j-simple-1.7.12.jar`
- outputs all events to `System.err`
- levels `ERROR > WARN > INFO > DEBUG`
- only messages of level `INFO` and higher are printed

```java
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class HelloWorld {
    public static void main(String[] args) {
        Logger logger = LoggerFactory.getLogger(HelloWorld.class);
        logger.info("Hello\World");
        logger.debug("Not\printed");
    }
}
```
Better bind to Logback-classic: gain an amazing amount of functionality.

“Logback implements SLF4J natively”:

- Logback’s ch.qos.logback.classic.Logger class is a direct implementation of SLF4J’s org.slf4j.Logger interface
- using SLF4J in conjunction with Logback involves strictly zero memory and computational overhead
- simply replace former slf4j-simple-1.7.12.jar or any other binding libraries by logback-classic-1.0.13.jar and logback-core-1.0.13.jar
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class HelloWorld {
    public static void main(String[] args) {
        Logger logger = LoggerFactory.getLogger("HelloWorld");
        logger.debug("Hello world.");
        logger.trace("I’m in main method");
    }
}

Same as before:
- code does not reference any logback classes
- in most cases, you will only need SLF4J classes
- behavior configuration through XML file logback.xml
- new level TRACE, if switch back to Simple binding, trace() calls will be silently ignored
Logback

Loggers form a hierarchy, similar to Java packages. At the top is always the root logger.

root → myPackage.Heater → myPackage.Heater.Boiler
Logback

If not assigned a level in the XML configuration file, a logger inherits its parent level. At `logback.xml`:

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

<table>
<thead>
<tr>
<th>class</th>
<th>level</th>
</tr>
</thead>
<tbody>
<tr>
<td>Heater</td>
<td>warn</td>
</tr>
<tr>
<td>Boiler</td>
<td>warn</td>
</tr>
<tr>
<td>others</td>
<td>info</td>
</tr>
</tbody>
</table>
Logback

16:49:31 [main] INFO  myPackage.Main - Entering main()
Appenders

Logging requests can be printed into one or multiple destinations.

Each output destination is represented by an appender and can be:

- console
- files (plain text, HTML…)
- remote socket servers
- databases (MySQL, Oracle, PostgreSQL)

Appenders are added to a logger. Each enabled logging request to that logger will be forwarded to all of its appenders. And also requests to that loggers descendents in the hierarchy.
Adding a console appender to the root logger will make every logger to output at least to console.

```
<configuration>
  <appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">
    <encoder>
      <pattern>
        %d{HH:mm:ss} [%thread] %-5level %logger{36} - %msg%n
      </pattern>
    </encoder>
  </appender>
  <root level="info">
    <appender-ref ref="STDOUT"/>
  </root>
  <logger name="myPackage.Heater" level="warn"/>
</configuration>
```
Appenders: FileAppender

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

<root level="info">
    <appender-ref ref="STDOUT" />
    <appender-ref ref="FILE" />
</root>
```

All the output goes to console and file test.dat. Chan choose whether to accumulate or overwrite output of each run.
Appenders

Encoders represent output layout. Very easily we can output logs in HTML:

```xml
<appender name="FILE" class="ch.qos.logback.core.FileAppender">
  <encoder class="ch.qos.logback.core.encoder.LayoutWrappingEncoder">
    <layout class="ch.qos.logback.classic.html.HTMLLayout">
      <pattern>%relative%thread%mdc%level%logger%msg</pattern>
    </layout>
  </encoder>
  <file>test.html</file>
</appender>

<root level="info">
  <appender-ref ref="STDOUT" />
  <appender-ref ref="FILE" />
</root>
```
### Appenders

#### Log session start time Tue Nov 03 16:28:04 CET 2015

<table>
<thead>
<tr>
<th>RelativeTime</th>
<th>Thread</th>
<th>MDC</th>
<th>Level</th>
<th>Logger</th>
<th>Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>9121</td>
<td>main</td>
<td></td>
<td>WARN</td>
<td>myPackage.Heater</td>
<td>Temperature set above 70 degrees, to 83 degrees.</td>
</tr>
<tr>
<td>9312</td>
<td>main</td>
<td></td>
<td>ERROR</td>
<td>myPackage.Heater</td>
<td>Temperature set above 100 degrees, to 113 degrees.</td>
</tr>
<tr>
<td>9435</td>
<td>main</td>
<td></td>
<td>DEBUG</td>
<td>myPackage.Heater.Boiler</td>
<td>setting temperature at Boiler</td>
</tr>
<tr>
<td>9437</td>
<td>main</td>
<td></td>
<td>DEBUG</td>
<td>myPackage.Heater.Boiler</td>
<td>setting temperature at Boiler</td>
</tr>
<tr>
<td>9438</td>
<td>main</td>
<td></td>
<td>DEBUG</td>
<td>myPackage.Heater.Boiler</td>
<td>setting temperature at Boiler</td>
</tr>
<tr>
<td>9439</td>
<td>main</td>
<td></td>
<td>WARN</td>
<td>myPackage.Heater</td>
<td>Temperature set above 70 degrees, to 86 degrees.</td>
</tr>
<tr>
<td>9440</td>
<td>main</td>
<td></td>
<td>DEBUG</td>
<td>myPackage.Heater.Boiler</td>
<td>setting temperature at Boiler</td>
</tr>
<tr>
<td>9440</td>
<td>main</td>
<td></td>
<td>ERROR</td>
<td>myPackage.Heater</td>
<td>Temperature set above 100 degrees, to 116 degrees.</td>
</tr>
<tr>
<td>9442</td>
<td>main</td>
<td></td>
<td>DEBUG</td>
<td>myPackage.Heater.Boiler</td>
<td>setting temperature at Boiler</td>
</tr>
</tbody>
</table>

#### Log session start time Tue Nov 03 16:40:14 CET 2015

<table>
<thead>
<tr>
<th>RelativeTime</th>
<th>Thread</th>
<th>MDC</th>
<th>Level</th>
<th>Logger</th>
<th>Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>9121</td>
<td>main</td>
<td></td>
<td>WARN</td>
<td>myPackage.Heater</td>
<td>Temperature set above 70 degrees, to 83 degrees.</td>
</tr>
<tr>
<td>9312</td>
<td>main</td>
<td></td>
<td>ERROR</td>
<td>myPackage.Heater</td>
<td>Temperature set above 100 degrees, to 113 degrees.</td>
</tr>
<tr>
<td>9435</td>
<td>main</td>
<td></td>
<td>DEBUG</td>
<td>myPackage.Heater.Boiler</td>
<td>setting temperature at Boiler</td>
</tr>
<tr>
<td>9437</td>
<td>main</td>
<td></td>
<td>DEBUG</td>
<td>myPackage.Heater.Boiler</td>
<td>setting temperature at Boiler</td>
</tr>
<tr>
<td>9438</td>
<td>main</td>
<td></td>
<td>DEBUG</td>
<td>myPackage.Heater.Boiler</td>
<td>setting temperature at Boiler</td>
</tr>
<tr>
<td>9439</td>
<td>main</td>
<td></td>
<td>WARN</td>
<td>myPackage.Heater</td>
<td>Temperature set above 70 degrees, to 86 degrees.</td>
</tr>
<tr>
<td>9440</td>
<td>main</td>
<td></td>
<td>DEBUG</td>
<td>myPackage.Heater.Boiler</td>
<td>setting temperature at Boiler</td>
</tr>
<tr>
<td>9440</td>
<td>main</td>
<td></td>
<td>ERROR</td>
<td>myPackage.Heater</td>
<td>Temperature set above 100 degrees, to 116 degrees.</td>
</tr>
<tr>
<td>9442</td>
<td>main</td>
<td></td>
<td>DEBUG</td>
<td>myPackage.Heater.Boiler</td>
<td>setting temperature at Boiler</td>
</tr>
</tbody>
</table>

#### Log session start time Tue Nov 03 16:41:46 CET 2015

<table>
<thead>
<tr>
<th>RelativeTime</th>
<th>Thread</th>
<th>MDC</th>
<th>Level</th>
<th>Logger</th>
<th>Message</th>
</tr>
</thead>
</table>

#### Log session start time Tue Nov 03 16:45:16 CET 2015

<table>
<thead>
<tr>
<th>RelativeTime</th>
<th>Thread</th>
<th>MDC</th>
<th>Level</th>
<th>Logger</th>
<th>Message</th>
</tr>
</thead>
</table>

#### Log session start time Tue Nov 03 16:47:20 CET 2015

<table>
<thead>
<tr>
<th>RelativeTime</th>
<th>Thread</th>
<th>MDC</th>
<th>Level</th>
<th>Logger</th>
<th>Message</th>
</tr>
</thead>
</table>
Appenders: RollingFileAppender

Rollover files: log to a file and then, under a *certain condition*, change target to a new output file.

Conditions specified as *rolling policies*. Most popular is `TimeBasedRollingPolicy`: change to a new file each month, week, day or hour.

```xml
<appender name="log-file"
    class="ch.qos.logback.core.rolling.RollingFileAppender">
    <file>my-application.log</file>

    <rollingPolicy
        class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">
        <!-- rotate every day for log collection and archiving -->
        <fileNamePattern>my-application.%d{yyyyMMdd}.log</fileNamePattern>
    </rollingPolicy>

    ...
```
Filters

The basic rule for logging is level + hierarchy of loggers. Filters are an additional mechanism associated to appenders: an appender can select messages in several ways.

**LevelFilter** filters out logs that don’t match exactly the specified level.

**ThresholdFilter** filters out logs below some level . . .

**EvaluatorFilter** filters logs for which message string contains some regular expression like statement [13579]

. . .
<appender name="warnings"
    class="ch.qos.logback.core.rolling.RollingFileAppender">
    <file>warnings.log</file>
    <filter class="ch.qos.logback.classic.filter.LevelFilter">
        <!-- only log warnings -->
        <level>WARN</level>
    </filter>
</appender>

<appender name="problems"
    class="ch.qos.logback.core.rolling.RollingFileAppender">
    <file>problems.log</file>
    <filter class="ch.qos.logback.classic.filter.ThresholdFilter">
        <!-- only log problems, not debugging info -->
        <level>DEBUG</level>
    </filter>
</appender>