Logging with SF4L and Logback

J.Serrat

102759 Software Design

November 3, 2015

Index

Sources

1. `http://www.slf4j.org/manual.html`
2. `http://logback.qos.ch/manual/`

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 SF4L+Logback classic, $\sim$ standard for Java
- review its key concepts: loggers, appenders and filters (pending: layouts)

Logging versus debugger

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.

Brian W. Kernighan and Rob Pike, *The Practice of Programming* (1999)

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 . . .

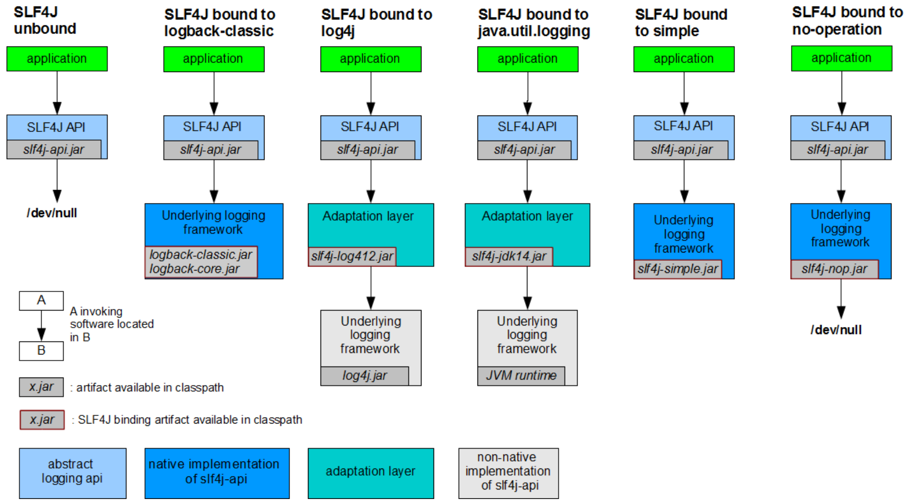
`https://en.wikipedia.org/wiki/Java_logging_framework`

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

SLF4J


SLF4J unbound
application
SLF4J API
slf4j-api.jar
/dev/null
SLF4J bound to logback-classic
application
SLF4J API
slf4j-api.jar
Underlying logging framework
logback-classic.jar
logback-core.jar
SLF4J bound to log4j
application
SLF4J API
slf4j-api.jar
Adaptation layer
slf4j-log412.jar
Underlying logging framework
log4j.jar
SLF4J bound to java.util.logging
application
SLF4J API
slf4j-api.jar
Adaptation layer
slf4j-jdk14.jar
Underlying logging framework
JVM runtime
SLF4J bound to simple
application
SLF4J API
slf4j-api.jar
Underlying logging framework
slf4j-simple.jar
SLF4J bound to no-operation
application
SLF4J API
slf4j-api.jar
Underlying logging framework
slf4j-nop.jar
/dev/null
A
B
A invoking software located in B
x.jar
: artifact available in classpath
x.jar
: SLF4J binding artifact available in classpath
abstract logging api
native implementation of slf4j-api
adaptation layer
non-native implementation of slf4j-api

SLF4J

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

SLF4J

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");
  }
}
```

Logback

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`

Logback

```
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

```
package myPackage;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class Heater {
static Logger logger = LoggerFactory.getLogger("myPackage.Heater");
    //...
}
public class Boiler extends Heater {
static Logger logger =
    LoggerFactory.getLogger("myPackage.Heater.Boiler");
    //...
}
```

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` :

```
<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>
```

class	level
Heater	warn
Boiler	warn
others	info

Logback

```
16:49:31 [main] INFO  myPackage.Main - Entering main()
16:49:31 [main] WARN  myPackage.Heater - Temperature set above 70
    degrees, to 83 degrees.
16:49:31 [main] ERROR myPackage.Heater - Temperature set above 100
    degrees, to 113 degrees.
16:49:31 [main] WARN  myPackage.Heater - Temperature set above 70
    degrees, to 86 degrees.
16:49:31 [main] ERROR myPackage.Heater - Temperature set above 100
    degrees, to 116 degrees.
```

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.

Appenders

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`

```
<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>
```

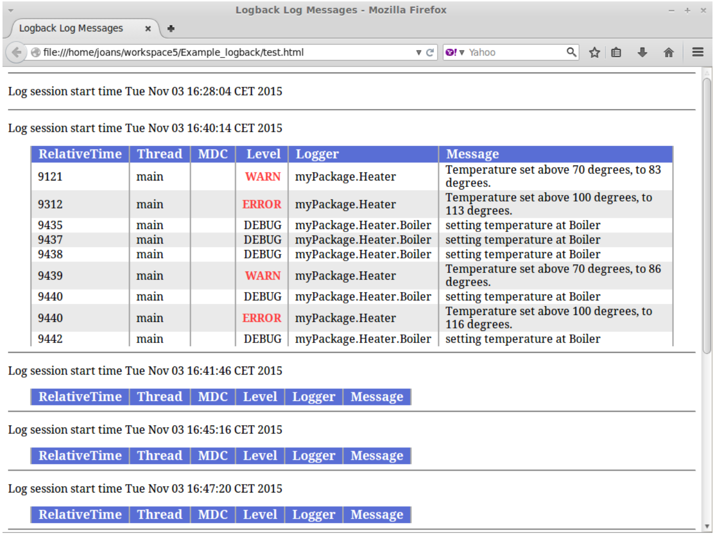
All the output goes to console *and* file `test.dat`. Chan choose whether to accumulate or overwrite output of each run.

Appenders

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

```
<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


Logback Log Messages - Mozilla Firefox
Logback Log Messages
file:///home/joans/workspace5/Example_logback/test.html
Yahoo

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

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

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

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

RelativeTime	Thread	MDC	Level	Logger	Message

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

RelativeTime	Thread	MDC	Level	Logger	Message

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

RelativeTime	Thread	MDC	Level	Logger	Message

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.

```
<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]`

. . .

Appenders

```
<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 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>
```