SLF4J user manual

The Simple Logging Facade for Java (SLF4J) serves as a simple facade or abstraction for various logging frameworks, such as java.util.logging, logback and log4j. SLF4J allows the end-user to plug in the desired logging framework at *deployment* time. Note that SLF4J-enabling your library/application implies the addition of only a single mandatory dependency, namely *slf4j-api-1.7.12.jar*.

SINCE 1.6.0 If no binding is found on the class path, then SLF4J will default to a no-operation implementation.

SINCE 1.7.0 Printing methods in the Logger interface now offer variants accepting varargs instead of `Object[]`. This change implies that SLF4J requires JDK 1.5 or later. Under the hood the Java compiler transforms the varargs part in methods into `Object[]`. Thus, the Logger interface generated by the compiler is indistinguishable in 1.7.x from its 1.6.x counterpart. It follows that SLF4J version 1.7.x is totally 100% no-ifs-or-buts compatible with SLF4J version 1.6.x.

SINCE 1.7.5 Significant improvement in logger retrieval times. Given the extent of the improvement, users are highly encouraged to migrate to SLF4J 1.7.5 or later.

SINCE 1.7.9 By setting the `slf4j.detectLoggerNameMismatch` system property to true, SLF4J can automatically spot incorrectly named loggers.

Hello World

As customary in programming tradition, here is an example illustrating the simplest way to output "Hello world" using SLF4J. It begins by getting a logger with the name "HelloWorld". This logger is in turn used to log the message "Hello World".

```
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");
  }
}
```

To run this example, you first need to download the slf4j distribution, and then to unpack it. Once that is done, add the file *slf4j-api-1.7.12.jar* to your class path.

Compiling and running *HelloWorld* will result in the following output being printed on the console.

```
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further details.
```

This warning is printed because no slf4j binding could be found on your class path.

The warning will disappear as soon as you add a binding to your class path. Assuming you add *slf4j-simple-1.7.12.jar* so that your class path contains:

- slf4j-api-1.7.12.jar
- slf4j-simple-1.7.12.jar

Compiling and running *HelloWorld* will now result in the following output on the console.

```
0 [main] INFO HelloWorld - Hello World
```

Typical usage pattern 典型的使用模式

The sample code below illustrates the typical usage pattern for SLF4J. Note the use of {}-placeholders on line 15. See the question "What is the fastest way of logging?" in the FAQ for more details.

```
 1: import org.slf4j.Logger;
 2: import org.slf4j.LoggerFactory;
 3:
 4: public class Wombat {
 5:
 6:   final Logger logger = LoggerFactory.getLogger(Wombat.class);
 7:   Integer t;
 8:   Integer oldT;
 9:
10:   public void setTemperature(Integer temperature) {

    oldT = t;
    t = temperature;

    logger.debug("Temperature set to {}. Old temperature was {}.", t, oldT);

    if(temperature.intValue() > 50) {
      logger.info("Temperature has risen above 50 degrees.");
    }
  }
}
```

Binding with a logging framework at deployment time 绑定日志记录框架

As mentioned previously, SLF4J supports various logging frameworks. The SLF4J distribution ships with several jar files referred to as "SLF4J bindings", with each binding corresponding to a supported framework.

slf4j-log4j12-1.7.12.jar

Binding for log4j version 1.2, a widely used logging framework. You also need to place *log4j.jar* on your class path.

slf4j-jdk14-1.7.12.jar

Binding for java.util.logging, also referred to as JDK 1.4 logging

slf4j-nop-1.7.12.jar

Binding for NOP, silently discarding all logging.

slf4j-simple-1.7.12.jar

Binding for Simple implementation, which outputs all events to System.err. Only messages of level INFO and higher are printed. This binding may be useful in the context of small applications.

slf4j-jcl-1.7.12.jar

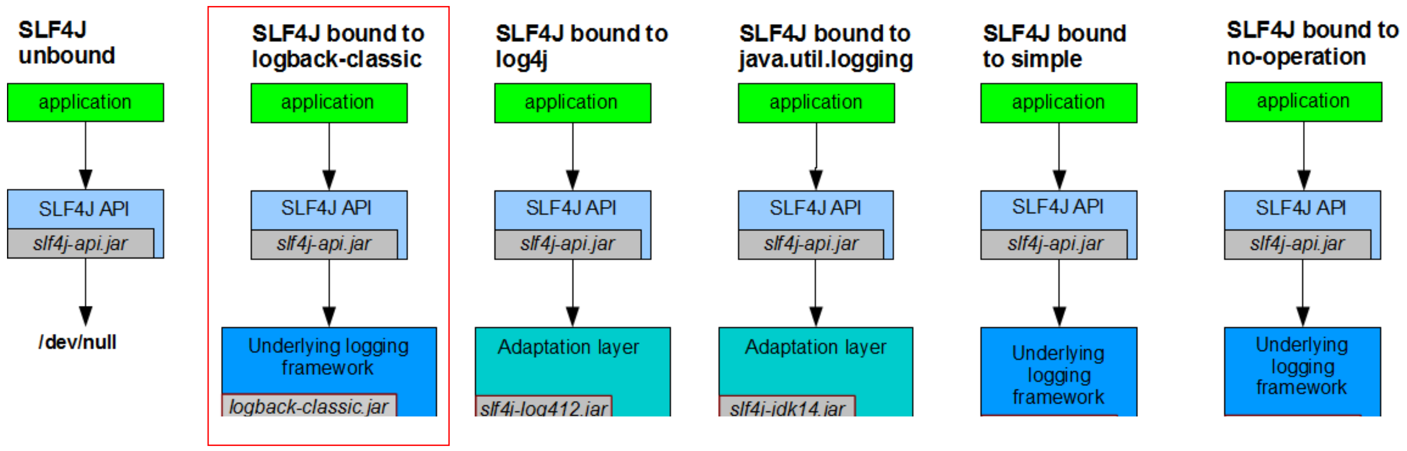
Binding for Jakarta Commons Logging. This binding will delegate all SLF4J logging to JCL.

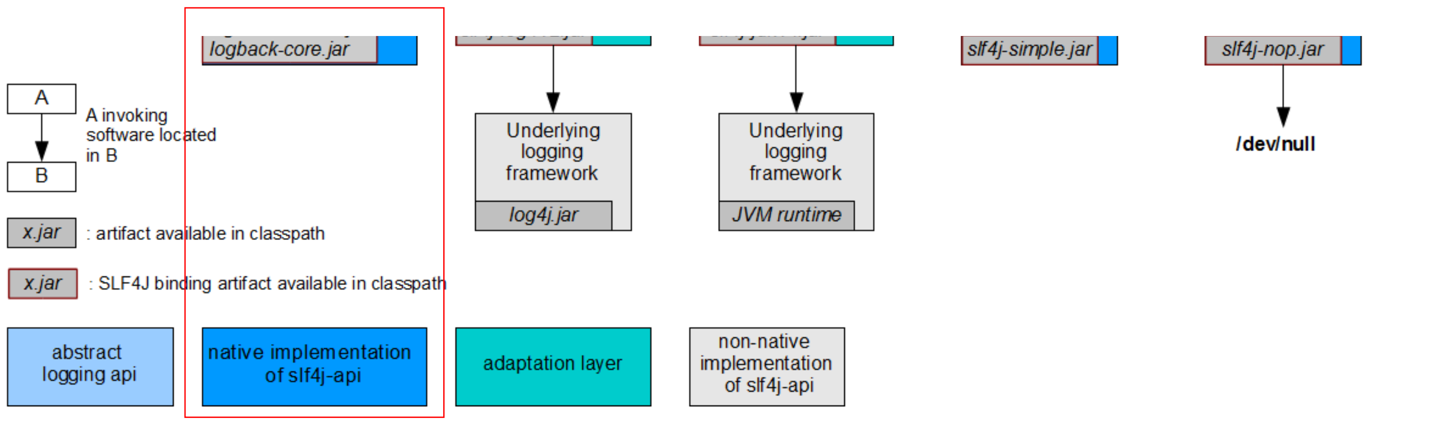
logback-classic-1.0.13.jar (requires logback-core-1.0.13.jar)

NATIVE IMPLEMENTATION There are also SLF4J bindings external to the SLF4J project, e.g. logback which implements SLF4J natively. Logback's `ch.qos.logback.classic.Logger` class is a direct implementation of SLF4J's `org.slf4j.Logger` interface. Thus, using SLF4J in conjunction with logback involves strictly zero memory and computational overhead.

To switch logging frameworks, just replace slf4j bindings on your class path. For example, to switch from java.util.logging to log4j, just replace slf4j-jdk14-1.7.12.jar with slf4j-log4j12-1.7.12.jar.

SLF4J does not rely on any special class loader machinery. In fact, each SLF4J binding is hardwired *at compile time* to use one and only one specific logging framework. For example, the slf4j-log4j12-1.7.12.jar binding is bound at compile time to use log4j. In your code, in addition to *slf4j-api-1.7.12.jar*, you simply drop **one and only one** binding of your choice onto the appropriate class path location. Do not place more than one binding on your class path. Here is a graphical illustration of the general idea.




logback-core.jar
slf4j-simple.jar
slf4j-nop.jar
A
B
A invoking software located in B
Underlying logging framework
log4j.jar
Underlying logging framework
JVM runtime
/dev/null
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

The SLF4J interfaces and their various adapters are extremely simple. Most developers familiar with the Java language should be able to read and fully understand the code in less than one hour. No knowledge of class loaders is necessary as SLF4J does not make use nor does it directly access any class loaders. As a consequence, SLF4J suffers from none of the class loader problems or memory leaks observed with Jakarta Commons Logging (JCL).

Given the simplicity of the SLF4J interfaces and its deployment model, developers of new logging frameworks should find it very easy to write SLF4J bindings.

Libraries

Authors of widely-distributed components and libraries may code against the SLF4J interface in order to avoid imposing an logging framework on their end-user. Thus, the end-user may choose the desired logging framework at deployment time by inserting the corresponding slf4j binding on the classpath, which may be changed later by replacing an existing binding with another on the class path and restarting the application. This approach has proven to be simple and very robust.

As of SLF4J version 1.6.0, if no binding is found on the class path, then slf4j-api will default to a no-operation implementation discarding all log requests. Thus, instead of throwing a `NoClassDefFoundError` because the `org.slf4j.impl.StaticLoggerBinder` class is missing, SLF4J version 1.6.0 and later will emit a single warning message about the absence of a binding and proceed to discard all log requests without further protest. For example, let Wombat be some biology-related framework depending on SLF4J for logging. In order to avoid imposing a logging framework on the end-user, Wombat's distribution includes *slf4j-api.jar* but no binding. Even in the absence of any SLF4J binding on the class path, Wombat's distribution will still work out-of-the-box, and without requiring the end-user to download a binding from SLF4J's web-site. Only when the end-user decides to enable logging will she need to install the SLF4J binding corresponding to the logging framework chosen by her.

BASIC RULE **Embedded components such as libraries or frameworks should not declare a dependency on any SLF4J binding but only depend on slf4j-api**. When a library declares a transitive dependency on a specific binding, that binding is imposed on the end-user negating the purpose of SLF4J. Note that declaring a non-transitive dependency on a binding, for example for testing, does not affect the end-user.

SLF4J usage in embedded components is also discussed in the FAQ in relation with logging configuration, dependency reduction and testing.

Declaring project dependencies for logging　　声明日志记录的项目依赖

Given Maven's transitive dependency rules, for "regular" projects (not libraries or frameworks) declaring logging dependencies can be accomplished with a single dependency declaration.

LOGBACK-CLASSIC If you wish to use logback-classic as the underlying logging framework, all you need to do is to declare "ch.qos.logback:logback-classic" as a dependency in your *pom.xml* file as shown below. In addition to *logback-classic-1.0.13.jar*, this will pull *slf4j-api-1.7.12.jar* as well as *logback-core-1.0.13.jar* into your project. Note that explicitly declaring a dependency on *logback-core-1.0.13* or *slf4j-api-1.7.12.jar* is not wrong and may be necessary to impose the correct version of said artifacts by virtue of Maven's "nearest definition" dependency mediation rule.

```
<dependency>
  <groupId>ch.qos.logback</groupId>
  <artifactId>logback-classic</artifactId>
  <version>1.0.13</version>
</dependency>
```

LOG4J If you wish to use log4j as the underlying logging framework, all you need to do is to declare "org.slf4j:slf4j-log4j12" as a dependency in your *pom.xml* file as shown below. In addition to *slf4j-log4j12-1.7.12.jar*, this will pull *slf4j-api-1.7.12.jar* as well as *log4j-1.2.17.jar* into your project. Note that explicitly declaring a dependency on *log4j-1.2.17.jar* or *slf4j-api-1.7.12.jar* is not wrong and may be necessary to impose the correct version of said artifacts by virtue of Maven's "nearest definition" dependency mediation rule.

```
<dependency>
  <groupId>org.slf4j</groupId>
  <artifactId>slf4j-log4j12</artifactId>
  <version>1.7.12</version>
</dependency>
```

JAVA.UTIL.LOGGING If you wish to use java.util.logging as the underlying logging framework, all you need to do is to declare "org.slf4j:slf4j-jdk14" as a dependency in your *pom.xml* file as shown below. In addition to *slf4j-jdk14-1.7.12.jar*, this will pull *slf4j-api-1.7.12.jar* into your project. Note that explicitly declaring a dependency on *slf4j-api-1.7.12.jar* is not wrong and may be necessary to impose the correct version of said artifact by virtue of Maven's "nearest definition" dependency mediation rule.

```
<dependency>
  <groupId>org.slf4j</groupId>
  <artifactId>slf4j-jdk14</artifactId>
  <version>1.7.12</version>
</dependency>
```

Binary compatibility

An SLF4J binding designates an artifact such as *slf4j-jdk14.jar* or *slf4j-log4j12.jar* used to *bind* slf4j to an underlying logging framework, say, java.util.logging and respectively log4j.

Mixing different versions of *slf4j-api.jar* and SLF4J binding can cause problems. For example, if you are using slf4j-api-1.7.12.jar, then you should also use slf4j-simple-1.7.12.jar, using slf4j-simple-1.5.5.jar will not work.

However, from the client's perspective all versions of slf4j-api are compatible. Client code compiled with *slf4j-api-N.jar* will run perfectly fine with *slf4j-api-M.jar* for any N and M. You only need to ensure that the version of your binding matches that of the slf4j-api.jar. You do not have to worry about the version of slf4j-api.jar used by a given dependency in your project. You can always use any version of *slf4j-api.jar*, and as long as the version of *slf4j-api.jar* and its binding match, you should be fine.

At initialization time, if SLF4J suspects that there may be an slf4j-api vs. binding version mismatch problem, it will emit a warning about the suspected mismatch.

From the client's perspective all versions of slf4j-api are compatible. Client code compiled with slf4j-api-N.jar will run perfectly fine with slf4j-api-M.jar for any N and M. You only need to ensure that the version of your binding matches that of the slf4j-api.jar. You do not have to worry about the version of slf4j-api.jar used by a given dependency in your project.

Consolidate logging via SLF4J

Often times, a given project will depend on various components which rely on logging APIs other than SLF4J. It is common to find projects depending on a combination of JCL, java.util.logging, log4j and SLF4J. It then becomes desirable to consolidate logging through a single channel. SLF4J caters for this common use-case by providing bridging modules for JCL, java.util.logging and log4j. For more details, please refer to the page on **Bridging legacy APIs**.

Mapped Diagnostic Context (MDC) support

"Mapped Diagnostic Context" is essentially a map maintained by the logging framework where the application code provides key-value pairs which can then be inserted by the logging framework in log messages. MDC data can also be highly helpful in filtering messages or triggering certain actions.

SLF4J supports MDC, or mapped diagnostic context. If the underlying logging framework offers MDC functionality, then SLF4J will delegate to the underlying framework's MDC. Note that at this time, only log4j and logback offer MDC functionality. If the underlying framework does not offer MDC, for example java.util.logging, then SLF4J will still store MDC data but the information therein will need to be retrieved by custom user code.

Thus, as a SLF4J user, you can take advantage of MDC information in the presence of log4j or logback, but without forcing these logging frameworks upon your users as dependencies.

For more information on MDC please see the chapter on MDC in the logback manual.

Executive summary

Advantage	Description

Select your logging framework at deployment time	The desired logging framework can be plugged in at deployment time by inserting the appropriate jar file (binding) on your class path.
Fail-fast operation	Due to the way that classes are loaded by the JVM, the framework binding will be verified automatically very early on. If SLF4J cannot find a binding on the class path it will emit a single warning message and default to no-operation implementation.
Bindings for popular logging frameworks	SLF4J supports popular logging frameworks, namely log4j, java.util.logging, Simple logging and NOP. The logback project supports SLF4J natively.
Bridging legacy logging APIs	The implementation of JCL over SLF4J, i.e *jcl-over-slf4j.jar*, will allow your project to migrate to SLF4J piecemeal, without breaking compatibility with existing software using JCL. Similarly, log4j-over-slf4j.jar and jul-to-slf4j modules will allow you to redirect log4j and respectively java.util.logging calls to SLF4J. See the page on Bridging legacy APIs for more details.
Migrate your source code	The slf4j-migrator utility can help you migrate your source to use SLF4J.
Support for parameterized log messages	All SLF4J bindings support parameterized log messages with significantly improved performance results.