RFC 219 LogService Update

Draft

50 Pages

Abstract

Logging is a crucial component to discover software bugs in a software system. The OSGi Log Service was the first compendium service and the Java eco-system gained over time many different log solutions: Log4j 2, Logback, Java Util Logging, etc. Since the OSGi Log Service was not further developed, the API does not take advantages of any of the new features in Java and looks very simplistic in comparison to mainstream Java. This document seeks to improve the Log Service API and add additional roles to upgrade it to Java 8.

0 Document Information

0.1 License

DISTRIBUTION AND FEEDBACK LICENSE, Version 2.0

The OSGi Alliance hereby grants you a limited copyright license to copy and display this document (the “Distribution”) in any medium without fee or royalty. This Distribution license is exclusively for the purpose of reviewing and providing feedback to the OSGi Alliance. You agree not to modify the Distribution in any way and further agree to not participate in any way in the making of derivative works thereof, other than as a necessary result of reviewing and providing feedback to the Distribution. You also agree to cause this notice, along with the accompanying consent, to be included on all copies (or portions thereof) of the Distribution. The OSGi Alliance also grants you a perpetual, non-exclusive, worldwide, fully paid-up, royalty free, limited license (without the right to sublicense) under any applicable copyrights, to create and/or distribute an implementation of the Distribution that: (i) fully implements the Distribution including all its required interfaces and functionality; (ii) does not modify, subset, superset or otherwise extend the OSGi Name Space, or include any public or protected packages, classes, Java interfaces, fields or methods within the OSGi Name Space other than those required and authorized by the Distribution. An implementation that does not satisfy limitations (i)-(ii) is not considered an implementation of the Distribution, does not receive the benefits of this license, and must not be described as an implementation of the Distribution. "OSGi Name Space" shall mean the public class or interface declarations whose names begin with "org.osgi" or any recognized successors or replacements thereof. The OSGi Alliance expressly reserves all rights not granted pursuant to these limited copyright licenses including termination of the license at will at any time.

EXCEPT FOR THE LIMITED COPYRIGHT LICENSES GRANTED ABOVE, THE OSGi ALLIANCE DOES NOT GRANT, EITHER EXPRESSLY OR IMPLIEDLY, A LICENSE TO ANY INTELLECTUAL PROPERTY IT, OR ANY THIRD PARTIES, OWN OR CONTROL. Title to the copyright in the Distribution will at all times remain with the OSGi Alliance. The example companies, organizations, products, domain names, e-mail addresses, logos, people, places, and events depicted therein are fictitious. No association with any real company, organization, product, domain name, email address, logo, person, place, or event is intended or should be inferred.

THE DISTRIBUTION IS PROVIDED "AS IS," AND THE OSGi ALLIANCE (INCLUDING ANY THIRD PARTIES THAT HAVE CONTRIBUTED TO THE DISTRIBUTION) MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, NON-INFRINGEMENT, OR TITLE; THAT THE CONTENTS OF THE DISTRIBUTION ARE SUITABLE FOR ANY PURPOSE; NOR THAT THE IMPLEMENTATION OF SUCH CONTENTS WILL NOT INFRINGE ANY THIRD PARTY PATENTS, COPYRIGHTS, TRADEMARKS OR OTHER RIGHTS.
NEITHER THE OSGi ALLIANCE NOR ANY THIRD PARTY WILL BE LIABLE FOR ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF OR RELATING TO ANY USE OR DISTRIBUTION OF THE DISTRIBUTION.

Implementation of certain elements of this Distribution may be subject to third party intellectual property rights, including without limitation, patent rights (such a third party may or may not be a member of the OSGi Alliance). The OSGi Alliance is not responsible and shall not be held responsible in any manner for identifying or failing to identify any or all such third party intellectual property rights.

The Distribution is a draft. As a result, the final product may change substantially by the time of final publication, and you are cautioned against relying on the content of this Distribution. You are encouraged to update any implementation of the Distribution if and when such Distribution becomes a final specification.

The OSGi Alliance is willing to receive input, suggestions and other feedback (“Feedback”) on the Distribution. By providing such Feedback to the OSGi Alliance, you grant to the OSGi Alliance and all its Members a non-exclusive, non-transferable,

worldwide, perpetual, irrevocable, royalty-free copyright license to copy, publish, license, modify, sublicense or otherwise distribute and exploit your Feedback for any purpose. Likewise, if incorporation of your Feedback would cause an implementation of the Distribution, including as it may be modified, amended, or published at any point in the future ("Future Specification"), to necessarily infringe a patent or patent application that you own or control, you hereby commit to grant to all implementers of such Distribution or Future Specification an irrevocable, worldwide, sublicenseable, royalty free license under such patent or patent application to make, have made, use, sell, offer for sale, import and export products or services that implement such Distribution or Future Specification. You warrant that (a) to the best of your knowledge you have the right to provide this Feedback, and if you are providing Feedback on behalf of a company, you have the rights to provide Feedback on behalf of your company; (b) the Feedback is not confidential to you and does not violate the copyright or trade secret interests of another; and (c) to the best of your knowledge, use of the Feedback would not cause an implementation of the Distribution or a Future Specification to necessarily infringe any third-party patent or patent application known to you. You also acknowledge that the OSGi Alliance is not required to incorporate your Feedback into any version of the Distribution or a Future Specification.

I HEREBY ACKNOWLEDGE AND AGREE TO THE TERMS AND CONDITIONS DELINEATED ABOVE.

0.2 Trademarks

OSGi™ is a trademark, registered trademark, or service mark of the OSGi Alliance in the US and other countries. Java is a trademark, registered trademark, or service mark of Oracle Corporation in the US and other countries. All other trademarks, registered trademarks, or service marks used in this document are the property of their respective owners and are hereby recognized.

0.3 Feedback

This document can be downloaded from the OSGi Alliance design repository at https://github.com/osgi/design The public can provide feedback about this document by opening a bug at https://www.osgi.org/bugzilla/.

0.4 Table of Contents

0 Document Information **2**
0.1 License 2
0.2 Trademarks 3
0.3 Feedback 3
0.4 Table of Contents 3
0.5 Terminology and Document Conventions 4
0.6 Revision History 4

1 Introduction **5**

2 Application Domain **6**
2.1 OSGi Log Service 6
2.2 Open Source 6
2.3 OSGi enRoute 7
2.4 Apache Sling Log Service 8
2.5 Terminology + Abbreviations 9

3 Problem Description **9**

4 Requirements **10**
4.1 SLF4J Loggers 10
4.2 Log Service 10
4.3 Log Admin 10
4.4 Log Reader 11
4.5 Log Entry 11

5 Technical Solution **11**
5.1 Logger 11
5.1.1 Logger names 11
5.1.2 DS Support for Logger 12
5.2 Obtaining LogEntries 12
5.3 LogEntry 13
5.4 Logger Admin 13
5.4.1 Logger Context 13
5.4.2 Logger Configuration 14
5.4.3 Configuration Admin Integration 14
5.4.4 Effective Log Level 14
5.5 LogService Legacy 15
5.6 Mapping of Events 15
5.7 Outstanding work 16

6 Javadoc **16**

7 Considered Alternatives **49**

8 Security Considerations **50**

9 Document Support **50**
9.1 References 50
9.2 Author's Address 50
9.3 Acronyms and Abbreviations 50
9.4 End of Document 50

0.5 Terminology and Document Conventions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY" and "OPTIONAL" in this document are to be interpreted as described in 1.

```
Source code is shown in this typeface.
```

0.6 Revision History

The last named individual in this history is currently responsible for this document.

Revision	Date	Comments
Initial	2016-01-05	Initial draft. BJ Hargrave
2nd draft	2016-01-14	Updates from CPEG meeting in Madrid. Add FormatterLogger, LogService extends LoggerFactory, DS support for Logger injection. BJ Hargrave

Revision	Date	Comments
3rd draft	2016-01-21	Updates from CPEG call. Moved LogStream into a separate package. BJ Hargrave
4th draft	2016-02-16	Updated additional arguments description for {} place holders to match SLF4J behavior. BJ Hargrave
5th draft	2016-02-26	Added org.osgi.service.log.admin package design to support inspecting and configuring log levels for named Loggers by bundle. BJ Hargrave
6th draft	2016-03-01	Updated handling of legacy log methods given changes draft 5. Also clarified some aspects of Configuration Admin integration. BJ Hargrave
7th draft	2016-03-02	Added information on updating 101.6 Mapping of Events. BJ Hargrave
8th draft	2016-03-03	Updates from CPEG call. Configuration changes via LoggerContext are not persisted. To persistently configure log levels, you must do so via the Configuration Admin integration. BJ Hargrave

1 Introduction

This RFC originates from a general desire in the OSGi community to upgrade the OSGi Log service and provide a more mainstream solution to make OSGi logging look more modern. The RFC is influenced by the OSGi enRoute project.

2 Application Domain

2.1 OSGi Log Service

The Log Service API has 4 methods. Each method takes a log level and a *message* string. In the OSGi Log Service this is an int. The ERROR level is 1, the TRACE level is 4, additional levels are accepted and stored. The 4 variations are used to pass a Service Reference and a Throwable.

Since the Log Service is aware of the bundle logging it can automatically provide this bundle in the entries. The OSGi Log Service is a dispatcher, it brokers between a *log client* and zero or more *log appenders*. The clients get the OSGi Log Service and the appenders get the OSGi Log Reader Service and register a listener with the Log Reader Service. The listener is then updated of any logging entries submitted by any client. The OSGi Log Reader has an optional history of recent events. The purpose of the history is to capture the log *entries* before the appender had been able to register itself.

A log entry consists of the bundle, a message, and an optional Service Reference and/or Throwable.

Since the OSGi Log Service uses services there can be multiple implementations and there is no guarantee that there is a log service is present. However, in general, there is only one Log Service and Log Reader service registered. In Declarative Services (DS) terms, the Log service should in general be a static dependency of a log client, which implies that the highest ranking log service is used. Though theoretically possible, few clients log to all registered service. Multiple Log Services is deemed an anomaly because it is a broker model and multiple brokers forfeit the purpose a bit.

Since the Log Service is a service it is possible that there is a need to log before the Log Service is available. Best practices in this case is to record the events until the Log Service becomes available, print to standard out, or ignore events. In Declarative Services, the bind methods that are called before the component is activated (and thus can be called before the Log Service is bound) can throw exceptions that are then logged by the Service Component Runtime.

2.2 Open Source

In the Open Source world a frenzy took place in developing log APIs. The current situation is quite complex because there are so many choices which created their own problems requiring facades that could log to many different logging subsystems. About ten years ago Java introduced java.util.logging but received a lot of flack from the industry because they had not followed best practices. Logging seems to be a quite sensitive product in our industry.

Today it seems that the Simple Logging Facade for Java (SLF4J) is the most mainstream API for clients. The reason of its success is partly its design. It provides an API that is identical to some of the other APIs and it can easily forward the logged entries to other log subsystems. This makes SLF4J attractive from the point of view of the log client.

SLF4J has the concept of a named *logger*. A logger is generally created in a static variable and is obtained from the `LoggerFactory` class. The name is generally the class name (there is an overloaded method on the Logger Factory to give a class object). When the first logger is created, the SLF4J code does some very heavy handed dynamic class loading magic to find a *provider*. The factory classes of the provider are generally implemented in a standard package in the SLF4J namespace. The provider then creates an implementation of the Logger class that is returned to the client. Since this is all static, it happens lazily on the first creation. However, it does require all

classes to be visible from the API classes. In OSGi it is therefore necessary to provide the implementation in the same bundle as the API bundle, or use a fragment on the API bundle.

The name of the logger is then used to establish on what level should actually be logged. Since Java class names are hierarchical, wildcards can be used to set the levels for related loggers. In SLF4J, the configuration is set with a properties file/resource that is searched for on well known places. In OSGi, fragments on the API bundle are often used to provide these properties. If a different configuration is needed then the application must be restarted. A logger is set to be *active* for a given level when log messages are passed to the appender.

The SLF4J API is a hodgepodge of log methods that come from different other log APIs and improvements over time. In general, the level is encoded in the method name. i.e. there are error, warn, trace, debug and info.

An important aspect of logging is the performance. Enterprise code is heavily instrumented and logging can take a significant portion of the code and CPU time. It is crucial to minimize the overhead of logging. This is the reason why often the actual log method is not called when the level is not active:

```
if (logger.isDebugEnabled())
  logger.debug("Hello " + name );
```

The reason of this pattern that this way the concatenation of the strings only takes place when the level is active. With the advent of Java 5 we got varargs. Varargs made it easy to defer the cost of computations of the parameters to when it is actually necessary. This made printf like loggers popular:

```
logger.debug("Hello {}", name );
```

This reduced the clutter of log messages significantly. The SLF4J Logger provides printf like methods for all supported levels but does not use the familiar % syntax of the Java String Formatters. It uses the message based format with curly braces.

SLF4J also provides capture of the current threads and *markers*. Markers allow the introduction of variables in the log that can bind different parts in an execution.

List of currently used logging frameworks:

- Log4j and the next version Log4j 2
- SLF4J and the update Logback
- Java 2 Logging API

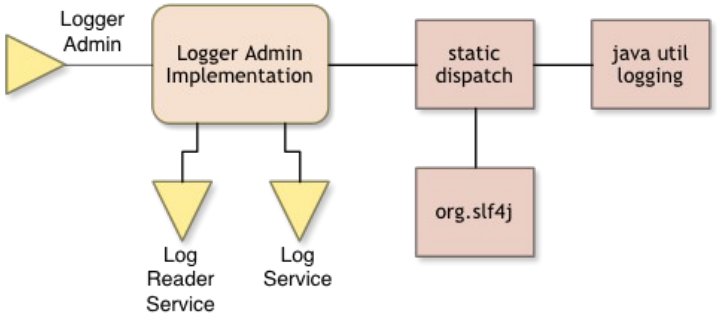
2.3 OSGi enRoute

In the OSGi enRoute [7]. project it was clear from the beginning that SLF4J was so popular in open source that it needed to be supported. However, though SLF4J and its appenders are delivered as bundles, it was not seen as a good idea to bypass the OSGi service model since the factory model is heavily based on dynamic class loading, causing all kinds of visibility problems. Therefore the approach was taken to provide a special enRoute appender that captured the log entries and forwarded them to the Log Service. Since this appender can be created before any OSGi framework is available it cannot rely on any of the OSGi mechanisms. It therefore buffers the log entry in memory as well as any loggers that access it.

Once the enRoute logger bundle becomes available it then accesses the static history and registers itself as the master. This logger bundle is configured with rules for the active levels for each logger. Based on these rules it forwards the entries to the OSGi Log Service. It is assumed to any appenders to other log systems would then use a Log Reader service. Mapping SLF4J entries to OSGi entries was a tad painful since the OSGi entries missed concepts like threads, sequence numbers, and markers.

Since dynamics are so important in OSGi, it also defined a Log Service Admin that could be used to manage the active levels of the loggers as well as provide the names of loggers that have logged. With this service (and corresponding Gogo commands) it is possible to dynamically change the active level of loggers.

The service diagram of the enRoute solution looked as depicted here:



OSGi enRoute registers an SLF4J Logger service and uses a Service Factory to capture the bundle. The name of this logger is then the symbolic name and version of the captured bundle. Though a logging service has the slight disadvantage that it is not available during initialization, it has the huge benefit of the service model. Since Declarative Services takes care of logging for errors in binding methods, the practical disadvantage is actually quite small.

Additionally, the OSGi enRoute also provided a utility to log based on a mechanism pioneered in bnd. The utility took an interface and a Logger and returned a proxy. Each method on the interface was a log message. The proxy handler would take the method name and turn it into a message, interleaving it with the arguments of the method. The level was defined by the return type.

Annotations were added to override the automatic message generation and to allow reorder and format arguments with the Java String Formatter API. By using Java types, the IDE helps finding log messages and refactoring log messages. The overhead is quite minimal since dynamic proxies have become quite fast today.

Last, and maybe least, OSGi enRoute added an additional level: AUDIT. This is a non-maskable level. Many financial institutions use log messages to audit, using a special level can provide more guarantees.

2.4 Apache Sling Log Service

The Apache Sling Log Service follows similar patterns but uses a slightly different route. In addition to consider SLF4J the most used log library, it also acknowledges that logback (http://logback.qos.ch/) is the most common backend for processing log entries.

Instead of funneling all log entries through the OSGi LogService, everything is passed on to SLF4J and from there to logback. Therefore the OSGi LogService logs to SLF4J and all the bridges are in place to pass log entries done through JUL or LOG4J to SLF4J.

Loggers and log levels can be configured through OSGi configurations. Config changes are of course processed dynamically and directly applied. On the other hand instead of using the OSGi LogService as an extension model, logback concepts are used through the whiteboard pattern. Logback TurboFilters, Filters, Appenders etc. can be registered dynamically as well as some other extensions.

This approach uses the most common parts today used for logging: SLF4J for clients and logback for processing.

[1] http://sling.apache.org/documentation/development/logging.html

2.5 Terminology + Abbreviations

- SLF4J - Simple Logging Facade for Java

3 Problem Description

The current OSGi Log Service suffers from the following problems:

- There is no way to set log levels so that not all messages are logged or a certain client.
- The OSGi Log Service API with its separate level is awkward to use in the code, the current practice is to use methods with the level name.
- The log client must construct the full message before the call is executed. This costs time and screen space.
- The current API does not capture threads, nor provides markers, or maintains sequence numbers
- The Log Service is not always available when a entry must be logged, especially during initialization. Then bundles need to buffer the log entries or print them on the console.
- The Log Service specification was developed before the whiteboard model was popular. This means that Log Listeners must first get the Log Reader Service and then register themselves. This is awkward, the whiteboard is much more convenient.
- Static loggers like SLF4J do not capture the bundle information
- Static loggers like SLF4J require class loading hacks to link them to an appender that are very non-OSGi like.

4 Requirements

4.1 SLF4J Loggers

- S0010 – It must be possible to use SLF4J API to log
- S0020 – It must be possible to create a static SLF4J logger (the normal way)
- S0030 – It must be possible to use a SFL4J Logger service that is automatically named according to the bundle's symbolic name and version.
- S0040 – Describe how static loggers are cleaned up when a bundle is uninstalled
- S0050 – Define how the the SLF4J go to the OSGi Log Service, taking into account that the bundle implementing the Log Service might be active when the SLF4J log entry is made and no entries may be lost.
- S0060 - Provide SLF4J support as an open source impl (via github for example) rather than bake SLF4J into a specification.

4.2 Log Service

- L0010 – The Log Service must provide a new level for AUDIT that cannot be ignored
- L0020 – The Log Service must provide a new level for TRACE to match SLF4J.
- L0030 – The Log Service API should be extended with the methods from the SLF4J Logger API. A varargs method should be added as will as the common 1, 2, … n arg versions for performance (avoid compiler always creating array to hold varargs).
- L0050 – Generally ensure that all SLF4J concepts map to a Log Service concept.
- L0060 – Provide visibility to the current log level so it can be interrogated.

4.3 Log Admin

- A0010 – The active levels of the loggers must be dynamically changeable
- A0020 – It must be possible to get a list of active loggers with their active level.
- A0030 – It must be possible to set the active level of a bundle for all logger from that bundle, assuming that static loggers come from the bundle they were loaded from.
- A0040 – It must be possible to get some Key Performance Indicators (KPIs) of the Logging subsystem like log entries per second, total entries, black listed readers, etc.
- A0050 – Provide a means to take action if a KPI reaches a threshold.

4.4 Log Reader

- R0010 – The Log Reader Listener must become whiteboard

4.5 Log Entry

- E0010 – It must be possible to capture thread information (thread id/thread name) in a Log Entry
- E0030 – The solution must provide the log entry with a sequence number.
- E0040 – The solution must provide the log entry with a logger name.
- E0050 – The solution should provide an option to include location info like the class and the method and line number if available.

5 Technical Solution

5.1 Logger

A new type is added to hold the various logging methods: Logger. The old log methods on LogService are now deprecated and a super interface, LoggerFactory, is added having methods to obtain a named Logger object. In SLF4J parlance, the LoggerFactory serves the function of an ILoggerFactory.

Like SLJ4J, logging method names are based upon the log levels and will only log if the level is in effect. Methods are present to test if a log level is in effect to enable work avoidance.

Like SLJ4J, the Logger log methods support formatted messages with “{}” place holders to avoid object-to-string conversion and string concatenation if the log level is not in effect.

As an option, the LoggerFactory also supports obtaining FormatterLogger objects which use printf-style formatting.

The implementation of the LogService specification must register the logging service under both the LogService name and the LoggerFactory name since they represent the same log and since LogService extends LoggerFactory.

5.1.1 Logger names

Logger names should be in the form of a fully qualified Java class names with segments separated by full stop (`'.'` \u002E). For example:

```
com.foo.Bar
```

Logger names form a hierarchy. A logger name is said to be an ancestor of another logger name if the logger name followed by a full stop ('.' \u002E) is a prefix of the descendant logger name. The root logger name ("ROOT") is the top ancestor of the logger name hierarchy. For example:

```
com.foo.Bar

com.foo

com

ROOT
```

5.1.2 DS Support for Logger

Since logging is both important and needed early in code execution, DS will add special support for injecting Logger and FormatterLogger objects even though they themselves are not services. When a component references the Logger or FormatterLogger types, SCR must get first get the LoggerFactory service matching the reference and then call the getLogger(String, Class) method passing the component implementation class name as the first argument and the Logger type as the second argument. The returned Logger object is then injected for the reference, rather than the LoggerFactory service used to create the Logger.

A DS example using Logger:

```
@Component
public class MyComponent {
  @Reference
  private Logger logger;
  @Activate
  void activate(ComponentContext context) {
    logger.trace("activating component id {}",
      context.getProperties().get("component.id"));
  }
}
```

5.2 Obtaining LogEntries

A new LogStream service is defined to replace the LogReaderService. The LogStream service is a prototype scope service which must be obtained as a prototype scope service using ServiceObjects or `@Reference(scope = ReferenceScope.PROTOTYPE_REQUIRED)` in a Declarative Service component.

Since the log is basically an ongoing stream of LogEntries having asynchronous arrival, a LogStream object is a PushStream<LogEntry> and can be used receive the LogEntries. The following code snippet show how one could get the log entries and print them.

```
@Reference(scope = ReferenceScope.PROTOTYPE_REQUIRED)
public void handleLog(LogStream logs) {
    logs.forEach(l -> System.out.println(l))
        .onResolve(() -> System.out.println("LogStream closed"));
}
```

An existing LogListener implementation can also be used with LogStream.

```
private LogListener ll;
@Reference(scope = ReferenceScope.PROTOTYPE_REQUIRED)
public void handleLog(LogStream logs) {
```

```
    logs.forEach(ll::logged)
        .onResolve(() -> System.out.println("LogStream closed"));
}
```

This change depends upon RFC 216 Push Streams.

The LogStream interface is in a separate package, `org.osgi.service.log.stream`, so that the existing `org.osgi.service.log` package does not use the `org.osgi.util.pushstream` package. The `org.osgi.util.pushstream` package requires `org.osgi.util.promise` and Java 8. Having LogStream in a separate package allows the `org.osgi.service.log` package to be implemented in a framework without dragging in the dependencies on `org.osgi.util.pushstream`, `org.osgi.util.promise` and Java 8. In this case, LogStream can be implemented by an external bundle which sources LogEntries from the LogReaderService.

5.3 LogEntry

LogEntry is extended to include:

- Thread information on the log entry creator.
- A sequence number which increases for each created log entry.
- The name of the Logger used to create the log entry.
- A StackTraceElement element of the caller that created the log entry.

5.4 Logger Admin

A Logger Admin service is defined which allows for the configuration of Loggers. The Logger Admin service can be used to obtain the Logger Context for a bundle. LoggerContexts are named similar to targeted PIDs in Configuration Admin. Each bundle may have its own named Logger Context based upon its bundle symbolic name, bundle version, and bundle location. There is also a root Logger Context from which each named Logger Context inherits. The root Logger Context has no name.

The Logger Admin service is associated with the Logger Factory service it administrates via the osgi.log.service.id service property whose value is a Long containing the service.id of the Logger Factory service.

5.4.1 Logger Context

The logger implementation must locate the Logger Context for the bundle to determine the effective log level of the logger when a bundle logs. The *best matching name* for the Logger Context is the longest name, which has a non-empty Logger Context, according to this syntax:

```
name ::= symbolic-name ( '|' version ( '|' location )? )?
```

The version must be formatted canonically, that is, according to the `toString()` method of the `Version` class. So the Logger Context for a bundle is searched for using the following names in the given order:

```
<symbolic-name>|<version>|<location>

<symbolic-name>|<version>
```

<symbolic-name>

If a non-empty Logger Context is not found, the Logger Context with the name <symbolic-name> is used for the bundle.

This allows a bundle to have no Logger Context configured. In this case it will use the root Logger Context's configuration. It also allows a bundle to be configured based upon bundle symbolic name, bundle symbolic name and version or even bundle symbolic name, version, and location. The latter forms may be of interest if there are multiple versions of a bundle installed.

5.4.2 Logger Configuration

Logger Contexts can be configured using the getLogLevels and setLogLevels methods of the Logger Context. Logger names, including the root logger name ("ROOT"), can be configured to a specific log level. The configured log level can also be null which means inherit the log level of the ancestor logger name.

Any change to the configuration of a Logger Context must be effective immediately for all loggers that would rely upon the configuration of the Logger Context. Changes to the configuration of a Logger Context via the setLogLevels method are not persisted.

5.4.3 Configuration Admin Integration

If Configuration Admin is present, ~~Logger Context~~ configuration information in Configuration Admin must be ~~mapped~~set into ~~Configuration Admin~~the mapped Logger Context. This allows external Logger Context configuration such as via RFC 218 Configurer. The name of the Logger Context is mapped ~~to~~from a Configuration Admin targeted PID as follows:

- ~~For the root Logger Context, which has no name, t~~The PID "org.osgi.service.log.admin" is ~~used~~mapped to the root Logger Context, which has no name.
- ~~For named Logger Contexts, the~~ PIDs starting with ~~is~~ "org.osgi.service.log.admin|" are mapped to named Logger Contexts~~suffixed by the Logger Context name~~ by removing the above prefix. For example, ~~if the Logger Context is named "com.foo.bar", then~~ the PID ~~b e will~~ "org.osgi.service.log.admin|com.foo.bar" is mapped to the Logger Context named "com.foo.bar".

In a Configuration Admin Configuration ~~for~~mapped to a Logger Context, the dictionary keys are logger names (key type is String) and the values are the names of the LogLevel enums (value types is String) with the addition of the special value "NULL" which signifies a null log level value for the logger name. If the Configuration contains any key/value pairs whose value is not the name of a LogLevel enum or "NULL", that key/value pair must be ignored~~by the Logger Context~~ when setting the configuration into the Logger Context.

Any change to the Configuration ~~for~~mapped to a Logger Context must be ~~effective~~set into the Logger Context as soon as possible ~~for all loggers that would rely upon the configuration of the Logger Context~~. Since notification of Configuration changes happen asynchronously, it may take a brief period of time before Configuration changes can be made effective.

This section is not meant to require that a logging implementation must require Configuration Admin ~~or must store its primary logger configuration information in Configuration Admin~~. But if Configuration Admin is present, the ~~current logger configuration information must be visible and modifiable as~~ Configuration Admin Configurations must be used to set the log levels in the mapped Logger Contexts.

5.4.4 Effective Log Level

Once the Logger Context for the logging bundle is determined, the effective log level for the logger's name is found by the following steps:

1. If the logger name is configured with a non-null log level in the bundle's Logger Context, use the configured log level.

2. For each ancestor logger name of the logger name, if the ancestor logger name is configured with a non-null log level in the bundle's Logger Context, use the configured log level.

3. If the logger name is configured with a non-null log level in the root Logger Context, use the configured log level.

4. For each ancestor logger name of the logger name, if the ancestor logger name is configured with a non-null log level in the root Logger Context, use the configured log level.

5. Use LogLevel.WARN because no non-null log level has been found for the logger name or any of its ancestor logger names.

5.5 LogService Legacy

The LogService interface has its original members deprecated. But the log methods can still be used by bundles. These log methods are now specified to log to the logger name “LogService” which allows legacy logging to be configured as specified above. Furthermore, the integeral log level values used with the log methods are mapped to the new LogLevels ~~where possible.~~as follows:

- LOG_ERROR - LogLevel.ERROR
- LOG_WARNING - LogLevel.WARN
- LOG_INFO - LogLevel.INFO
- LOG_DEBUG - LogLevel.DEBUG
- Any other value - LogLevel.TRACE

~~If an~~ The integeral log level value is ~~used which does not map, this value~~ is stored in the generated LogEntry to be returned by getLevel~~() and LogLevel.TRACE is used for the log level~~.

5.6 Mapping of Events

The log implementation must map framework events, ServiceEvent, BundleEvent, and FrameworkEvent, into log entries. This section of the spec must be updated to state the logger names used. This will allow the effective log level for this mapping to be configured. FrameworkEvents are logged under the logger name “Events.Framework”, ServiceEvents under the logger name “Events.Service”, and BundleEvents under the logger name “Events.Bundle”. These logger names all share the ancestor logger name “Events” to allow for shared configuration.

When mapping log entries to Event Admin, the list of logging levels must be expanded to include the new LogLevel names. The properties of the event are expanded to add:

- log.loggername – (String) The name of the Logger.
- log.threadinfo – (String) The thread information for the thread creating the log entry.

5.7 Outstanding work

- Static access to Loggers. This may be done outside of this specification as an open source project on the OSGi GitHub account.

6 Javadoc

OSGi Javadoc

3/3/16 5:20 PM

Package Summary		Page
org.osgi.service.log	Log Service Package Version 1.4.	18
org.osgi.service.log.admin	Log Admin Package Version 1.0.	43
org.osgi.service.log.stream	Log Stream Package Version 1.0.	48

Package org.osgi.service.log

```
@org.osgi.annotation.versioning.Version(value="1.4")
```

Log Service Package Version 1.4.

See:
[Description](#)

Interface Summary		Page
FormatterLogger	Provides methods for bundles to write messages to the log using printf-style format strings.	*19*
LogEntry	Provides methods to access the information contained in an individual Log Service log entry.	*20*
Logger	Provides methods for bundles to write messages to the log using SLF4J-style format strings.	*24*
LoggerFactory	Logger Factory service for logging information.	*32*
LogListener	Subscribes to `LogEntry` objects from the `LogReaderService`.	*36*
LogReaderService	LogReaderService for obtaining logging information.	*37*
LogService	LogService for logging information.	*39*

Enum Summary		Page
LogLevel	Log Levels.	*34*

Package org.osgi.service.log Description

Log Service Package Version 1.4.

Bundles wishing to use this package must list the package in the Import-Package header of the bundle's manifest. This package has two types of users: the consumers that use the API in this package and the providers that implement the API in this package.

Example import for consumers using the API in this package:

```
Import-Package: org.osgi.service.log; version="[1.4,2.0)"
```

Example import for providers implementing the API in this package:

```
Import-Package: org.osgi.service.log; version="[1.4,1.5)"
```

Interface FormatterLogger

org.osgi.service.log

All Superinterfaces:
Logger

```
@org.osgi.annotation.versioning.ProviderType
public interface FormatterLogger
extends Logger
```

Provides methods for bundles to write messages to the log using printf-style format strings.

Messages can be formatted by the Logger once the Logger determines the log level is enabled. Uses printf-style format strings as described in `Formatter`.

You can also add a `Throwable` and/or `ServiceReference` to the generated `LogEntry` by passing them to the logging methods as additional arguments. If the last argument is a `Throwable` or `ServiceReference`, it is added to the generated `LogEntry` and then if the next to last argument is a `ServiceReference` or `Throwable` and not the same type as the last argument, it is added to the generated `LogEntry`. For example:

```
logger.info("Found service %s.", serviceReference, serviceReference);
logger.warn("Something named %s happened.", name, serviceReference,
  throwable);
logger.error("Failed.", exception);
```

If an exception occurs formatting the message, the logged message will indicate the formatting failure including the format string and the arguments.

Since:
1.4
ThreadSafe

Fields inherited from interface org.osgi.service.log.Logger
ROOT_LOGGER_NAME

Methods inherited from interface org.osgi.service.log.Logger
audit, audit, audit, audit, debug, debug, debug, debug, error, error, error, error, getName, info, info, info, info, isDebugEnabled, isErrorEnabled, isInfoEnabled, isTraceEnabled, isWarnEnabled, trace, trace, trace, trace, warn, warn, warn, warn

Interface LogEntry

org.osgi.service.log

```
@org.osgi.annotation.versioning.ProviderType
public interface LogEntry
```

Provides methods to access the information contained in an individual Log Service log entry.

A `LogEntry` object may be acquired from the `LogReaderService.getLog` method or by registering a `LogListener` object.

ThreadSafe

Method Summary		Page
org.osgi.framework.Bundle	**getBundle**() Returns the bundle that created this `LogEntry` object.	20
Throwable	**getException**() Returns the exception object associated with this `LogEntry` object.	21
int	**getLevel**() **Deprecated.** *Since 1.4.*	21
StackTraceElement	**getLocation**() Returns the location information of the creation of this `LogEntry` object.	23
String	**getLoggerName**() Returns the name of the `Logger` object used to create this `LogEntry` object.	22
LogLevel	**getLogLevel**() Returns the level of this `LogEntry` object.	22
String	**getMessage**() Returns the human readable message associated with this `LogEntry` object.	21
long	**getSequence**() Returns the sequence number for this `LogEntry` object.	22
org.osgi.framework.ServiceReference<?>	**getServiceReference**() Returns the `ServiceReference` object for the service associated with this `LogEntry` object.	21
String	**getThreadInfo**() Returns a string representing the thread which created this `LogEntry` object.	22
long	**getTime**() Returns the value of `currentTimeMillis()` at the time this `LogEntry` object was created.	21

Method Detail

getBundle

```
org.osgi.framework.Bundle getBundle()
```

Returns the bundle that created this `LogEntry` object.

Returns:

The bundle that created this `LogEntry` object; `null` if no bundle is associated with this `LogEntry` object.

---

getServiceReference

```
org.osgi.framework.ServiceReference<?> getServiceReference()
```

Returns the `ServiceReference` object for the service associated with this `LogEntry` object.

Returns:
`ServiceReference` object for the service associated with this `LogEntry` object; `null` if no `ServiceReference` object was provided.

---

getLevel

```
@Deprecated
int getLevel()
```

Deprecated. *Since 1.4. Replaced by `getLogLevel()`.*

Returns the integer level of this `LogEntry` object.

If one of the `log` methods of `LogService` was used, this is the specified integer level. Otherwise, this is the `ordinal` value of the `log level`.

Returns:
Integer level of this `LogEntry` object.

---

getMessage

```
String getMessage()
```

Returns the human readable message associated with this `LogEntry` object.

Returns:
`String` containing the message associated with this `LogEntry` object.

---

getException

```
Throwable getException()
```

Returns the exception object associated with this `LogEntry` object.

In some implementations, the returned exception may not be the original exception. To avoid references to a bundle defined exception class, thus preventing an uninstalled bundle from being garbage collected, the Log Service may return an exception object of an implementation defined Throwable subclass. The returned object will attempt to provide as much information as possible from the original exception object such as the message and stack trace.

Returns:
`Throwable` object of the exception associated with this `LogEntry`;`null` if no exception is associated with this `LogEntry` object.

---

getTime

```
long getTime()
```

Returns the value of `currentTimeMillis()` at the time this `LogEntry` object was created.

Returns:
The system time in milliseconds when this `LogEntry` object was created.
See Also:
"System.currentTimeMillis()"

---

getLogLevel

`LogLevel` **`getLogLevel`**`()`

Returns the level of this `LogEntry` object.

Returns:
The level of this `LogEntry` object.
Since:
1.4

---

getLoggerName

`String` **`getLoggerName`**`()`

Returns the name of the `Logger` object used to create this `LogEntry` object.

Returns:
The name of the `Logger` object used to create this `LogEntry` object.
Since:
1.4

---

getSequence

`long` **`getSequence`**`()`

Returns the sequence number for this `LogEntry` object.

A unique, non-negative value that is larger than all previously assigned values since the log implementation was started. These values are transient and are reused upon restart of the log implementation.

Returns:
The sequence number for this `LogEntry` object.
Since:
1.4

---

getThreadInfo

`String` **`getThreadInfo`**`()`

Returns a string representing the thread which created this `LogEntry` object.

This string must contain the name of the thread and may contain other information about the thread.

Returns:
A string representing the thread which created this `LogEntry` object.
Since:
1.4

---

getLocation

```
StackTraceElement getLocation()
```

Returns the location information of the creation of this `LogEntry` object.

Returns:
The location information of the creation of this `LogEntry` object.

Since:
1.4

Interface Logger

org.osgi.service.log

All Known Subinterfaces:
FormatterLogger

```
@org.osgi.annotation.versioning.ProviderType
public interface Logger
```

Provides methods for bundles to write messages to the log using SLF4J-style format strings.

Messages can be formatted by the Logger once the Logger determines the log level is enabled. Use `"{}"` as a place holder for an argument. If you need to use the literal `"{}"` in the formatted message, precede the place holder with a backslash: `"\\{}"`. If you need to place a backslash before the place holder, precede the backslash with a backslash: `"\\\\{}"`.

You can also add a `Throwable` and/or `ServiceReference` to the generated `LogEntry` by passing them to the logging methods as additional arguments. If the last argument is a `Throwable` or `ServiceReference`, it is added to the generated `LogEntry` and then if the next to last argument is a `ServiceReference` or `Throwable` and not the same type as the last argument, it is added to the generated `LogEntry`. These arguments will not be used for place holders. For example:

```
logger.info("Found service {}.", serviceReference, serviceReference);
logger.warn("Something named {} happened.", name, serviceReference,
  throwable);
logger.error("Failed.", exception);
```

Since:
1.4
ThreadSafe

Field Summary		Page
String	**ROOT_LOGGER_NAME** Root Logger Name.	25

Method Summary		Page
void	**audit**(String message) Log a message at the LogLevel.AUDIT level.	31
void	**audit**(String format, Object arg) Log a formatted message at the LogLevel.AUDIT level.	31
void	**audit**(String format, Object arg1, Object arg2) Log a formatted message at the LogLevel.AUDIT level.	31
void	**audit**(String format, Object... arguments) Log a formatted message at the LogLevel.AUDIT level.	31
void	**debug**(String message) Log a message at the LogLevel.DEBUG level.	27
void	**debug**(String format, Object arg) Log a formatted message at the LogLevel.DEBUG level.	27
void	**debug**(String format, Object arg1, Object arg2) Log a formatted message at the LogLevel.DEBUG level.	27
void	**debug**(String format, Object... arguments) Log a formatted message at the LogLevel.DEBUG level.	28
void	**error**(String message) Log a message at the LogLevel.ERROR level.	30

void	**error**(String format, Object arg) Log a formatted message at the LogLevel.ERROR level.	30
void	**error**(String format, Object arg1, Object arg2) Log a formatted message at the LogLevel.ERROR level.	30
void	**error**(String format, Object... arguments) Log a formatted message at the LogLevel.ERROR level.	31
String	**getName**() Return the name of this Logger.	26
void	**info**(String message) Log a message at the LogLevel.INFO level.	28
void	**info**(String format, Object arg) Log a formatted message at the LogLevel.INFO level.	28
void	**info**(String format, Object arg1, Object arg2) Log a formatted message at the LogLevel.INFO level.	28
void	**info**(String format, Object... arguments) Log a formatted message at the LogLevel.INFO level.	29
boolean	**isDebugEnabled**() Is logging enabled for the LogLevel.DEBUG level?	27
boolean	**isErrorEnabled**() Is logging enabled for the LogLevel.ERROR level?	30
boolean	**isInfoEnabled**() Is logging enabled for the LogLevel.INFO level?	28
boolean	**isTraceEnabled**() Is logging enabled for the LogLevel.TRACE level?	26
boolean	**isWarnEnabled**() Is logging enabled for the LogLevel.WARN level?	29
void	**trace**(String message) Log a message at the LogLevel.TRACE level.	26
void	**trace**(String format, Object arg) Log a formatted message at the LogLevel.TRACE level.	26
void	**trace**(String format, Object arg1, Object arg2) Log a formatted message at the LogLevel.TRACE level.	26
void	**trace**(String format, Object... arguments) Log a formatted message at the LogLevel.TRACE level.	27
void	**warn**(String message) Log a message at the LogLevel.WARN level.	29
void	**warn**(String format, Object arg) Log a formatted message at the LogLevel.WARN level.	29
void	**warn**(String format, Object arg1, Object arg2) Log a formatted message at the LogLevel.WARN level.	29
void	**warn**(String format, Object... arguments) Log a formatted message at the LogLevel.WARN level.	30

Field Detail

ROOT_LOGGER_NAME

```
public static final String ROOT_LOGGER_NAME = "ROOT"
```

Root Logger Name.

Method Detail

getName

```
String getName()
```

Return the name of this Logger.

Returns:
The name of this Logger.

---

isTraceEnabled

```
boolean isTraceEnabled()
```

Is logging enabled for the `LogLevel.TRACE` level?

Returns:
`true` if logging is enabled for the `LogLevel.TRACE` level.

---

trace

```
void trace(String message)
```

Log a message at the `LogLevel.TRACE` level.

Parameters:
`message` - The message to log.

---

trace

```
void trace(String format,
           Object arg)
```

Log a formatted message at the `LogLevel.TRACE` level.

Parameters:
`format` - The format of the message to log.
`arg` - The argument to format into the message.

---

trace

```
void trace(String format,
           Object arg1,
           Object arg2)
```

Log a formatted message at the `LogLevel.TRACE` level.

Parameters:
`format` - The format of the message to log.
`arg1` - The first argument to format into the message.
`arg2` - The second argument to format into the message.

---

trace

```
void trace(String format,
           Object... arguments)
```

Log a formatted message at the LogLevel.TRACE level.

Parameters:
format - The format of the message to log.
arguments - The arguments to format into the message.

---

isDebugEnabled

```
boolean isDebugEnabled()
```

Is logging enabled for the LogLevel.DEBUG level?

Returns:
true if logging is enabled for the trace level.

---

debug

```
void debug(String message)
```

Log a message at the LogLevel.DEBUG level.

Parameters:
message - The message to log.

---

debug

```
void debug(String format,
           Object arg)
```

Log a formatted message at the LogLevel.DEBUG level.

Parameters:
format - The format of the message to log.
arg - The argument to format into the message.

---

debug

```
void debug(String format,
           Object arg1,
           Object arg2)
```

Log a formatted message at the LogLevel.DEBUG level.

Parameters:
format - The format of the message to log.
arg1 - The first argument to format into the message.
arg2 - The second argument to format into the message.

---

debug

```
void debug(String format,
           Object... arguments)
```

Log a formatted message at the `LogLevel.DEBUG` level.

Parameters:
`format` - The format of the message to log.
`arguments` - The arguments to format into the message.

---

isInfoEnabled

```
boolean isInfoEnabled()
```

Is logging enabled for the `LogLevel.INFO` level?

Returns:
`true` if logging is enabled for the `trace` level.

---

info

```
void info(String message)
```

Log a message at the `LogLevel.INFO` level.

Parameters:
`message` - The message to log.

---

info

```
void info(String format,
          Object arg)
```

Log a formatted message at the `LogLevel.INFO` level.

Parameters:
`format` - The format of the message to log.
`arg` - The argument to format into the message.

---

info

```
void info(String format,
          Object arg1,
          Object arg2)
```

Log a formatted message at the `LogLevel.INFO` level.

Parameters:
`format` - The format of the message to log.
`arg1` - The first argument to format into the message.
`arg2` - The second argument to format into the message.

---

info

```
void info(String format,
          Object... arguments)
```

Log a formatted message at the LogLevel.INFO level.

Parameters:
format - The format of the message to log.
arguments - The arguments to format into the message.

---

isWarnEnabled

```
boolean isWarnEnabled()
```

Is logging enabled for the LogLevel.WARN level?

Returns:
true if logging is enabled for the trace level.

---

warn

```
void warn(String message)
```

Log a message at the LogLevel.WARN level.

Parameters:
message - The message to log.

---

warn

```
void warn(String format,
          Object arg)
```

Log a formatted message at the LogLevel.WARN level.

Parameters:
format - The format of the message to log.
arg - The argument to format into the message.

---

warn

```
void warn(String format,
          Object arg1,
          Object arg2)
```

Log a formatted message at the LogLevel.WARN level.

Parameters:
format - The format of the message to log.
arg1 - The first argument to format into the message.
arg2 - The second argument to format into the message.

---

warn

```
void warn(String format,
          Object... arguments)
```

Log a formatted message at the LogLevel.WARN level.

Parameters:
format - The format of the message to log.
arguments - The arguments to format into the message.

---

isErrorEnabled

```
boolean isErrorEnabled()
```

Is logging enabled for the LogLevel.ERROR level?

Returns:
true if logging is enabled for the trace level.

---

error

```
void error(String message)
```

Log a message at the LogLevel.ERROR level.

Parameters:
message - The message to log.

---

error

```
void error(String format,
           Object arg)
```

Log a formatted message at the LogLevel.ERROR level.

Parameters:
format - The format of the message to log.
arg - The argument to format into the message.

---

error

```
void error(String format,
           Object arg1,
           Object arg2)
```

Log a formatted message at the LogLevel.ERROR level.

Parameters:
format - The format of the message to log.
arg1 - The first argument to format into the message.
arg2 - The second argument to format into the message.

---

error

```
void error(String format,
           Object... arguments)
```

Log a formatted message at the LogLevel.ERROR level.

Parameters:
format - The format of the message to log.
arguments - The arguments to format into the message.

---

audit

```
void audit(String message)
```

Log a message at the LogLevel.AUDIT level.

Parameters:
message - The message to log.

---

audit

```
void audit(String format,
           Object arg)
```

Log a formatted message at the LogLevel.AUDIT level.

Parameters:
format - The format of the message to log.
arg - The argument to format into the message.

---

audit

```
void audit(String format,
           Object arg1,
           Object arg2)
```

Log a formatted message at the LogLevel.AUDIT level.

Parameters:
format - The format of the message to log.
arg1 - The first argument to format into the message.
arg2 - The second argument to format into the message.

---

audit

```
void audit(String format,
           Object... arguments)
```

Log a formatted message at the LogLevel.AUDIT level.

Parameters:
format - The format of the message to log.
arguments - The arguments to format into the message.

Interface LoggerFactory

org.osgi.service.log

All Known Subinterfaces:
LogService

```
@org.osgi.annotation.versioning.ProviderType
public interface LoggerFactory
```

Logger Factory service for logging information.

Provides methods for bundles to obtain named Loggers that can be used to write messages to the log.

Logger names should be in the form of a fully qualified Java class names with segments separated by full stop ('.' \u002E). For example:

```
com.foo.Bar
```

Logger names exist in a hierarchy. A logger name is said to be an ancestor of another logger name if the logger name followed by a full stop ('.' \u002E) is a prefix of the descendant logger name. The root logger name is the top ancestor of the logger name hierarchy. For example:

```
com.foo.Bar
com.foo
com
ROOT
```

Since:
1.4
ThreadSafe

Method Summary		Page
Logger	**getLogger**(Class<?> clazz) Return the Logger named with the specified class.	33
L	**getLogger**(Class<?> clazz, Class<L> loggerType) Return the Logger of the specified type named with the specified class.	33
Logger	**getLogger**(String name) Return the Logger named with the specified name.	32
L	**getLogger**(String name, Class<L> loggerType) Return the Logger of the specified type named with the specified name.	33

Method Detail

getLogger

Logger **getLogger**(String name)

Return the Logger named with the specified name.

Parameters:
name - The name to use for the logger name.
Returns:
The Logger named with the specified name. If the name parameter is equal to Logger.ROOT_LOGGER_NAME, then the root logger is returned.

---

getLogger

```
Logger getLogger(Class<?> clazz)
```

Return the Logger named with the specified class.

Parameters:
clazz - The class to use for the logger name.
Returns:
The Logger named with the name of the specified class.

---

getLogger

```
L getLogger(String name,
            Class<L> loggerType)
```

Return the Logger of the specified type named with the specified name.

Type Parameters:
L - The Logger type.
Parameters:
name - The name to use for the logger name.
loggerType - The type of Logger. Can be Logger or FormatterLogger.
Returns:
The Logger or FormatterLogger named with the specified name. If the name parameter is equal to Logger.ROOT_LOGGER_NAME, then the root logger is returned.
Throws:
IllegalArgumentException - If the specified type is not a supported Logger type.

---

getLogger

```
L getLogger(Class<?> clazz,
            Class<L> loggerType)
```

Return the Logger of the specified type named with the specified class.

Type Parameters:
L - A Logger type.
Parameters:
clazz - The class to use for the logger name.
loggerType - The type of Logger. Can be Logger or FormatterLogger.
Returns:
The Logger or FormatterLogger named with the name of the specified class.
Throws:
IllegalArgumentException - If the specified type is not a supported Logger type.

Enum LogLevel

org.osgi.service.log

```
java.lang.Object
  └ java.lang.Enum<LogLevel>
      └ org.osgi.service.log.LogLevel
```

All Implemented Interfaces:
Comparable<LogLevel>, Serializable

```
public enum LogLevel
extends Enum<LogLevel>
```

Log Levels.

Since:
1.4

Enum Constant Summary	Page
AUDIT Audit – Information that must always be logged.	34
DEBUG Debug – Detailed output for debugging operations.	35
ERROR Error – Information about an error situation.	34
INFO Info – Information about normal operation.	35
TRACE Trace level – Large volume of output for tracing operations.	35
WARN Warning – Information about a failure or unwanted situation that is not blocking.	35

Method Summary		Page
boolean	**implies**(LogLevel other) Returns whether this log level implies the specified log level.	35
static LogLevel	**valueOf**(String name)	35
static LogLevel[]	**values**()	35

Enum Constant Detail

AUDIT

```
public static final LogLevel AUDIT
```

Audit – Information that must always be logged.

ERROR

```
public static final LogLevel ERROR
```

Error – Information about an error situation.

WARN

```
public static final LogLevel WARN
```

Warning – Information about a failure or unwanted situation that is not blocking.

INFO

```
public static final LogLevel INFO
```

Info – Information about normal operation.

DEBUG

```
public static final LogLevel DEBUG
```

Debug – Detailed output for debugging operations.

TRACE

```
public static final LogLevel TRACE
```

Trace level – Large volume of output for tracing operations.

Method Detail

values

```
public static LogLevel[] values()
```

---

valueOf

```
public static LogLevel valueOf(String name)
```

---

implies

```
public boolean implies(LogLevel other)
```

Returns whether this log level implies the specified log level.

Parameters:
`other` - The other log level.
Returns:
`true` If this log level implies the specified log level; `false` otherwise.

Interface LogListener

org.osgi.service.log

All Superinterfaces:
EventListener

```
@org.osgi.annotation.versioning.ConsumerType
@FunctionalInterface
public interface LogListener
extends EventListener
```

Subscribes to `LogEntry` objects from the `LogReaderService`.

A `LogListener` object may be registered with the Log Reader Service using the `LogReaderService.addLogListener` method. After the listener is registered, the `logged` method will be called for each `LogEntry` object created. The `LogListener` object may be unregistered by calling the `LogReaderService.removeLogListener` method.

Since 1.4, `LogStream` is the preferred way to obtain `LogEntry` objects.

ThreadSafe

Method Summary		Page
void	**logged**(LogEntry entry) Listener method called for each LogEntry object created.	36

Method Detail

logged

```
void logged(LogEntry entry)
```

Listener method called for each LogEntry object created.

Parameters:
`entry` - A `LogEntry` object containing log information.

Interface LogReaderService

org.osgi.service.log

```
@org.osgi.annotation.versioning.ProviderType
public interface LogReaderService
```

LogReaderService for obtaining logging information.

There are two ways to obtain LogEntry objects:

- The primary way to obtain LogEntry objects is to get a LogStream object from the service registry. This replaces adding a LogListener object.
- To obtain past LogEntry objects, the getLog() method can be called which will return an Enumeration of the LogEntry objects in the log.

ThreadSafe

Method Summary		Page
void	**addLogListener**(LogListener listener) Subscribes to LogEntry objects.	37
Enumeration<LogEntry>	**getLog**() Returns an Enumeration of the LogEntry objects in the log.	38
void	**removeLogListener**(LogListener listener) Unsubscribes to LogEntry objects.	37

Method Detail

addLogListener

```
void addLogListener(LogListener listener)
```

Subscribes to LogEntry objects.

This method registers a LogListener object with the Log Reader Service. The LogListener.logged(LogEntry) method will be called for each LogEntry object placed into the log.

When a bundle which registers a LogListener object is stopped or otherwise releases the Log Reader Service, the Log Reader Service must remove all of the bundle's listeners.

If this Log Reader Service's list of listeners already contains a listener l such that (l==listener), this method does nothing.

Since 1.4, LogStream is the preferred way to obtain LogEntry objects.

Parameters:

listener - A LogListener object to register; the LogListener object is used to receive LogEntry objects.

---

removeLogListener

```
void removeLogListener(LogListener listener)
```

Unsubscribes to LogEntry objects.

This method unregisters a LogListener object from the Log Reader Service.

If `listener` is not contained in this Log Reader Service's list of listeners, this method does nothing.

Since 1.4, `LogStream` is the preferred way to obtain `LogEntry` objects.

Parameters:
`listener` - A `LogListener` object to unregister.

---

getLog

`Enumeration<LogEntry> getLog()`

Returns an `Enumeration` of the `LogEntry` objects in the log.

Each element of the enumeration is a `LogEntry` object, ordered with the most recent entry first. Whether the enumeration is of all `LogEntry` objects since the Log Service was started or some recent past is implementation-specific. Also implementation-specific is which level `LogEntry` objects are included in the enumeration.

Returns:
An `Enumeration` of the `LogEntry` objects in the log.

Interface LogService

org.osgi.service.log

All Superinterfaces:
LoggerFactory

```
@org.osgi.annotation.versioning.ProviderType
public interface LogService
extends LoggerFactory
```

LogService for logging information.

Replaced by LoggerFactory.

ThreadSafe

Field Summary		Page
int	**LOG_DEBUG** **Deprecated.** *Since 1.4.*	40
int	**LOG_ERROR** **Deprecated.** *Since 1.4.*	39
int	**LOG_INFO** **Deprecated.** *Since 1.4.*	40
int	**LOG_WARNING** **Deprecated.** *Since 1.4.*	40

Method Summary		Page
void	**log**(int level, String message) **Deprecated.** *Since 1.4.*	40
void	**log**(int level, String message, Throwable exception) **Deprecated.** *Since 1.4.*	41
void	**log**(org.osgi.framework.ServiceReference<?> sr, int level, String message) **Deprecated.** *Since 1.4.*	41
void	**log**(org.osgi.framework.ServiceReference<?> sr, int level, String message, Throwable exception) **Deprecated.** *Since 1.4.*	42

Methods inherited from interface org.osgi.service.log.LoggerFactory
getLogger, getLogger, getLogger, getLogger

Field Detail

LOG_ERROR

```
@Deprecated
public static final int LOG_ERROR = 1
```

Deprecated.

An error message (Value 1).

This log entry indicates the bundle or service may not be functional.

LOG_WARNING

```
@Deprecated
public static final int LOG_WARNING = 2
```

Deprecated.

A warning message (Value 2).

This log entry indicates a bundle or service is still functioning but may experience problems in the future because of the warning condition.

LOG_INFO

```
@Deprecated
public static final int LOG_INFO = 3
```

Deprecated.

An informational message (Value 3).

This log entry may be the result of any change in the bundle or service and does not indicate a problem.

LOG_DEBUG

```
@Deprecated
public static final int LOG_DEBUG = 4
```

Deprecated.

A debugging message (Value 4).

This log entry is used for problem determination and may be irrelevant to anyone but the bundle developer.

Method Detail

log

```
@Deprecated
void log(int level,
         String message)
```

Deprecated. *Since 1.4. Replaced by Logger. See LoggerFactory.*

Logs a message.

The `ServiceReference` field and the `Throwable` field of the `LogEntry` object will be set to `null`.

This method will log to the `Logger` named `"LogService"` for the bundle. The specified level is mapped to a `LogLevel` as follows:

1. `LOG_ERROR` - `LogLevel.ERROR`
2. `LOG_WARNING` - `LogLevel.WARN`
3. `LOG_INFO` - `LogLevel.INFO`
4. `LOG_DEBUG` - `LogLevel.DEBUG`
5. Any other value - `LogLevel.TRACE`

In the generated log entry, `LogEntry.getLevel()` must return the specified level.

Parameters:

`level` - The severity of the message. This should be one of the defined log levels but may be any integer that is interpreted in a user defined way.
`message` - Human readable string describing the condition or `null`.

---

log

```
@Deprecated
void log(int level,
         String message,
         Throwable exception)
```

Deprecated. *Since 1.4. Replaced by Logger. See LoggerFactory.*

Logs a message with an exception.

The `ServiceReference` field of the `LogEntry` object will be set to `null`.

This method will log to the `Logger` named `"LogService"` for the bundle. The specified level is mapped to a `LogLevel` as follows:

- `LOG_ERROR` - `LogLevel.ERROR`
- `LOG_WARNING` - `LogLevel.WARN`
- `LOG_INFO` - `LogLevel.INFO`
- `LOG_DEBUG` - `LogLevel.DEBUG`
- Any other value - `LogLevel.TRACE`

In the generated log entry, `LogEntry.getLevel()` must return the specified level.

Parameters:

`level` - The severity of the message. This should be one of the defined log levels but may be any integer that is interpreted in a user defined way.
`message` - The human readable string describing the condition or `null`.
`exception` - The exception that reflects the condition or `null`.

---

log

```
@Deprecated
void log(org.osgi.framework.ServiceReference<?> sr,
         int level,
         String message)
```

Deprecated. *Since 1.4. Replaced by Logger. See LoggerFactory.*

Logs a message associated with a specific `ServiceReference` object.

The `Throwable` field of the `LogEntry` will be set to `null`.

This method will log to the `Logger` named `"LogService"` for the bundle. The specified level is mapped to a `LogLevel` as follows:

- `LOG_ERROR` - `LogLevel.ERROR`
- `LOG_WARNING` - `LogLevel.WARN`
- `LOG_INFO` - `LogLevel.INFO`
- `LOG_DEBUG` - `LogLevel.DEBUG`
- Any other value - `LogLevel.TRACE`

In the generated log entry, `LogEntry.getLevel()` must return the specified level.

Parameters:

`sr` - The `ServiceReference` object of the service that this message is associated with or `null`.

`level` - The severity of the message. This should be one of the defined log levels but may be any integer that is interpreted in a user defined way.
`message` - Human readable string describing the condition or `null`.

---

log

```
@Deprecated
void log(org.osgi.framework.ServiceReference<?> sr,
         int level,
         String message,
         Throwable exception)
```

Deprecated. *Since 1.4. Replaced by `Logger`. See `LoggerFactory`.*

Logs a message with an exception associated and a `ServiceReference` object.

This method will log to the `Logger` named `"LogService"` for the bundle. The specified level is mapped to a `LogLevel` as follows:

- `LOG_ERROR` - `LogLevel.ERROR`
- `LOG_WARNING` - `LogLevel.WARN`
- `LOG_INFO` - `LogLevel.INFO`
- `LOG_DEBUG` - `LogLevel.DEBUG`
- Any other value - `LogLevel.TRACE`

In the generated log entry, `LogEntry.getLevel()` must return the specified level.

Parameters:

`sr` - The `ServiceReference` object of the service that this message is associated with.
`level` - The severity of the message. This should be one of the defined log levels but may be any integer that is interpreted in a user defined way.
`message` - Human readable string describing the condition or `null`.
`exception` - The exception that reflects the condition or `null`.

Package org.osgi.service.log.admin

```
@org.osgi.annotation.versioning.Version(value="1.0")
```

Log Admin Package Version 1.0.

See:
Description

Interface Summary		Page
LoggerAdmin	LoggerAdmin service for configuring loggers.	44
LoggerContext	Logger Context for a bundle.	46

Package org.osgi.service.log.admin Description

Log Admin Package Version 1.0.

Bundles wishing to use this package must list the package in the Import-Package header of the bundle's manifest. This package has two types of users: the consumers that use the API in this package and the providers that implement the API in this package.

Example import for consumers using the API in this package:

```
Import-Package: org.osgi.service.log.admin; version="[1.0,2.0)"
```

Example import for providers implementing the API in this package:

```
Import-Package: org.osgi.service.log.admin; version="[1.0,1.1)"
```

Interface LoggerAdmin

org.osgi.service.log.admin

```
@org.osgi.annotation.versioning.ProviderType
public interface LoggerAdmin
```

LoggerAdmin service for configuring loggers.

Each bundle may have its own named LoggerContext based upon its bundle symbolic name, bundle version, and bundle location. There is also a root Logger Context from which each named Logger Context inherits. The root Logger Context has no name.

When a bundle logs, the logger implementation must locate the Logger Context for the bundle to determine the effective log level of the logger name. The *best matching name* for the Logger Context is the longest name, which has a non-empty Logger Context, according to this syntax:

```
name ::= symbolic-name ( '|' version ( '|' location )? )?
```

The version must be formatted canonically, that is, according to the `toString()` method of the `Version` class. So the Logger Context for a bundle is searched for using the following names in the given order:

```
<symbolic-name>|<version>|<location>
<symbolic-name>|<version>
<symbolic-name>
```

If a non-empty Logger Context is not found, the Logger Context with the name `<symbolic-name>` is used for the bundle.

ThreadSafe

Field Summary		Page
String	**LOG_SERVICE_ID** Logger Admin service property to associate the Logger Admin service with a LoggerFactory service.	44

Method Summary		Page
LoggerContext	**getLoggerContext**(String name) Get the Logger Context for the specified name.	45

Field Detail

LOG_SERVICE_ID

```
public static final String LOG_SERVICE_ID = "osgi.log.service.id"
```

Logger Admin service property to associate the Logger Admin service with a LoggerFactory service.

This service property is set to the `service.id` for the LoggerFactory service administered by this Logger Admin.

The value of this service property must be of type `Long`.

Method Detail

getLoggerContext

`LoggerContext` **`getLoggerContext`**`(String name)`

Get the Logger Context for the specified name.

Parameters:
`name` - The name of the Logger Context. Can be `null` to specify the root Logger Context.
Returns:
The Logger Context for the specified name.

Interface LoggerContext

org.osgi.service.log.admin

```
@org.osgi.annotation.versioning.ProviderType
public interface LoggerContext
```

Logger Context for a bundle.

Any change to the configuration of this Logger Context must be effective immediately for all loggers that would rely upon the configuration of this Logger Context.

ThreadSafe

Method Summary		Page
void	**clear**() Clear the configuration of this Logger Context.	47
LogLevel	**getEffectiveLogLevel**(String name) Returns the effective log level of the logger name in this Logger Context.	46
Map<String,LogLevel>	**getLogLevels**() Returns the configured log levels for this Logger Context.	47
String	**getName**() Returns the name for this Logger Context.	46
boolean	**isEmpty**() Returns whether the configuration of this Logger Context is empty.	47
void	**setLogLevels**(Map<String,LogLevel> logLevels) Configure the log levels for this Logger Context.	47

Method Detail

getName

```
String getName()
```

Returns the name for this Logger Context.

Returns:
The name for this Logger Context. The root Logger Context has no name and returns `null`.

getEffectiveLogLevel

```
LogLevel getEffectiveLogLevel(String name)
```

Returns the effective log level of the logger name in this Logger Context.

The effective log level for a logger name is found by the following steps:

1. If the specified logger name is configured with a non-null log level in this Logger Context, return the configured log level.
2. For each ancestor logger name of the specified logger name, if the ancestor logger name is configured with a non-null log level in this Logger Context, return the configured log level.
3. If the specified logger name is configured with a non-null log level in the root Logger Context, return the configured log level.
4. For each ancestor logger name of the specified logger name, if the ancestor logger name is configured with a non-null log level in the root Logger Context, return the configured log level.

5. Return LogLevel.WARN because no non-null log level has been found for the specified logger name or any of its ancestor logger names.

Parameters:
`name` - The logger name.
Returns:
The effective log level of the logger name in this Logger Context.

---

getLogLevels

```
Map<String,LogLevel> getLogLevels()
```

Returns the configured log levels for this Logger Context.

Returns:
The configured log levels for this Logger Context. The keys are the logger names and the values are the log levels. The returned map may be empty if no logger names are configured for this Logger Context. The log level value can be `null`. The returned map is the property of the caller who can modify the map and use it as input to setLogLevels(Map). The returned map must support all optional Map operations.

---

setLogLevels

```
void setLogLevels(Map<String,LogLevel> logLevels)
```

Configure the log levels for this Logger Context.

All previous log levels configured for this Logger Context are cleared and then the log levels in the specified map are configured.

Parameters:
`logLevels` - The log levels to configure for this Logger Context. The keys are the logger names and the values are the log levels. The log level value can be `null`. The specified map is the property of the caller and this method must not modify or retain the specified map.

---

clear

```
void clear()
```

Clear the configuration of this Logger Context.

The configured log levels will be cleared.

---

isEmpty

```
boolean isEmpty()
```

Returns whether the configuration of this Logger Context is empty.

Returns:
`true` if this Logger Context has no configuration. That is, the configured log levels are empty. Otherwise `false` is returned.

Package org.osgi.service.log.stream

```
@org.osgi.annotation.versioning.Version(value="1.0")
```

Log Stream Package Version 1.0.

See:
Description

Interface Summary		Page
LogStream	LogStream service for receiving log information.	*49*

Package org.osgi.service.log.stream Description

Log Stream Package Version 1.0.

Bundles wishing to use this package must list the package in the Import-Package header of the bundle's manifest. This package has two types of users: the consumers that use the API in this package and the providers that implement the API in this package.

Example import for consumers using the API in this package:

```
Import-Package: org.osgi.service.log.stream; version="[1.0,2.0)"
```

Example import for providers implementing the API in this package:

```
Import-Package: org.osgi.service.log.stream; version="[1.0,1.1)"
```

Interface LogStream

org.osgi.service.log.stream

All Superinterfaces:
AutoCloseable, Closeable, org.osgi.util.pushstream.PushStream<LogEntry>

```
@org.osgi.annotation.versioning.ProviderType
public interface LogStream
extends org.osgi.util.pushstream.PushStream<LogEntry>
```

LogStream service for receiving log information.

This service must be registered as a `prototype` service so that each user can receive a new, distinct `org.osgi.util.pushstream.PushStream` of `LogEntry` objects. Users of this service must obtain this service using `org.osgi.framework.ServiceObjects` to get a new, distinct `org.osgi.util.pushstream.PushStream`. For example, to inject a `LogStream` service into a Declarative Services component:

```
@Reference(scope = ReferenceScope.PROTOTYPE_REQUIRED)
private LogStream logs;
```

When a `LogStream` service object is released and `org.osgi.framework.PrototypeServiceFactory.ungetService(org.osgi.framework.Bundle, org.osgi.framework.ServiceRegistration, Object)` is called, `Closeable.close()` must be called on the `LogStream` to ensure the stream is closed.

ThreadSafe

Methods inherited from interface org.osgi.util.pushstream.PushStream
allMatch, anyMatch, buffer, buildBuffer, coalesce, coalesce, coalesce, collect, count, distinct, filter, findAny, findFirst, flatMap, forEach, forEachEvent, fork, limit, map, max, merge, min, noneMatch, onClose, onError, reduce, reduce, reduce, sequential, skip, sorted, sorted, split, toArray, toArray, window, window, window, window

7 Considered Alternatives

None at this time.

8 Security Considerations

Bundles using the LogListener as a service will now need ServicePermission to register it as a service. The implementation will need ServicePermission to get it as a service.

To configure log levels via the LoggerAdmin service, bundles will need ServicePermission to get the LoggerAdmin service. To create configurations in Config Admin that configure log levels, bundles will need ConfigurationPermission to configure the desired PID names.

9 Document Support

9.1 References

[1]. Bradner, S., Key words for use in RFCs to Indicate Requirement Levels, RFC2119, March 1997.

[2]. Software Requirements & Specifications. Michael Jackson. ISBN 0-201-87712-0

[3]. https://issues.apache.org/jira/browse/FELIX-536

[4]. SLF4J, http://www.slf4j.org

[5]. Apache Log4j 2, http://logging.apache.org/log4j/2.x/

[6]. Equinox Log Service (org.eclipse.equinox.log)

[7]. http://enroute.osgi.org/services/osgi.enroute.logger.api.html

[8]. Logback Architecture. http://logback.qos.ch/manual/architecture.html

[9]. Logback Configuration. http://logback.qos.ch/manual/configuration.html

9.2 Author's Address

Name	BJ Hargrave
Company	IBM

9.3 Acronyms and Abbreviations

9.4 End of Document