ONAP Application Logging Guidelines v1.2 (Beijing)

About This Document

Official R1 documentation snapshot in https://onap.readthedocs.io/en/latest/submodules/logging-analytics.git/docs/

This document specifies logging conventions to be followed by ONAP component applications.

ONAP logging is intended to support operability, debugging and reporting on ONAP. These guidelines address:

- Events that are written by ONAP components.
- Propagation of transaction and invocation information between components.
- MDCs, Markers and other information that should be attached to log messages.
- Human- and machine-readable output format(s).
- Files, locations and other conventions.

Java is assumed, but conventions may be implemented by non-Java components.

Original ONAP Logging guidelines: https://wiki.onap.org/download/attachments/1015849/ONAP%20application%20logging%20guidelines.pdf?api=v2

Introduction

The purpose of ONAP logging is to capture information needed to operate, troubleshoot and report on the performance of the ONAP platform and its constituent components. Log records may be viewed and consumed directly by users and systems, indexed and loaded into a datastore, and used to compute metrics and generate reports.

The processing of a single client request will often involve multiple ONAP components and/or subcomponents (interchangeably referred to as 'application' in this document). The ability to track flows across components is critical to understanding ONAP's behavior and performance. ONAP logging uses a universally unique RequestID value in log records to track the processing of every client request through all the ONAP components involved in its processing.

A reference configuration of Elastic Stack can be deployed using ONAP Operations Manager.

This document gives conventions you can follow to generate conformant, indexable logging output from your component.

How to Log

ONAP prescribes conventions. The use of certain APIs and providers is recommended, but they are not mandatory. Most components log via EELF or SLF 4J to a provider like Logback or Log4j.

EELF

EELF is the **Event and Error Logging Framework**, described at https://github.com/att/EELF.

EELF abstracts your choice of logging provider, and decorates the familiar Logger contracts with features like:

- Localization.
- Error codes.
- Generated wiki documentation.
- Separate audit, metric, security and debug logs.

EELF is a facade, so logging output is configured in two ways:

1. By selection of a logging provider such as Logback or Log4j, typically via the classpath.
2. By way of a provider configuration document, typically **logback.xml** or **log4j.xml**. See Providers.

SLF4J

SLF4J is a logging facade, and a humble masterpiece. It combines what's common to all major, modern Java logging providers into a single interface. This decouples the caller from the provider, and encourages the use of what's universal, familiar and proven.

EELF also logs via SLF4J's abstractions.

Providers

Logging providers are normally enabled by their presence in the classpath. This means the decision may have been made for you, in some cases implicitly by dependencies. If you have a strong preference then you can change providers, but since the implementation is typically abstracted behind EELF or SLF4J, it may not be worth the effort.

Logback

Logback is the most commonly used provider. It is generally configured by an XML document named **logback.xml**. See Configuration.

Log4j 2.X

Log4j 2.X is somewhat less common than Logback, but equivalent. It is generally configured by an XML document named **log4j.xml**. See Configuration.

Log4j 1.X

Strongly discouraged from Beijing onwards, since 1.X is EOL, and since it does not support escaping, so its output may not be machine-readable. See https://logging.apache.org/log4j/1.2/.

This affects OpenDaylight-based components like SDNC and APPC, since ODL releases prior to Carbon bundled Log4j 1.X, and make it difficult to replace. The Common Controller SDK Project project targets ODL Carbon, so remaining instances of Log4j 1.X should disappear by the time of the Beijing release.

What to Log

The purpose of logging is to capture diagnostic information.

An important aspect of this is analytics, which requires tracing of requests between components. In a large, distributed system such as ONAP this is critical to understanding behavior and performance.

Messages, Levels, Components and Categories

It isn't the aim of this document to reiterate the basics, so advice here is general:

- Use a logger. Consider using EELF.
- Write log messages in English.
- Write meaningful messages. Consider what will be useful to consumers of logger output.
- Use errorcodes to characterise exceptions.
- Log at the appropriate level. Be aware of the volume of logs that will be produced.
- Log in a machine-readable format. See Conventions.
- Log for analytics as well as troubleshooting.

Others have written extensively on this:

- http://www.masterzen.fr/2013/01/13/the-10-commandments-of-logging/
- https://www.loggly.com/blog/how-to-write-effective-logs-for-remote-logging/
- And so on.

Context

TODO: more on the importance of transaction ID propagation.

MDCs

A Mapped Diagnostic Context (MDC) allows an arbitrary string-valued attribute to be attached to a Java thread. The MDC's value is then emitted with each message logged by that thread. The set of MDCs associated with a log message is serialized as unordered name-value pairs (see Text Output).

A good discussion of MDCs can be found at https://logback.qos.ch/manual/mdc.html.

MDCs:

- Must be set as early in invocation as possible.
- Must be unset on exit.

Logging

Via SLF4J:

```
import java.util.UUID;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.slf4j.MDC;
// ...
final Logger logger = LoggerFactory.getLogger(this.getClass());
MDC.put("SomeUUID", UUID.randomUUID().toString());
try {
    logger.info("This message will have a UUID-valued 'SomeUUID' MDC attached.");
    // ...
}
finally {
    MDC.clear();
}
```

EELF doesn't directly support MDCs, but its default provider (where **com.att.eelf.configuration.SLF4jWrapper** is the configured EELF provider)normally logs via SLF4J, and SLF4J will receive any MDC that is set:

```
import java.util.UUID;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.slf4j.MDC;
import com.att.eelf.configuration.EELFLogger;
import com.att.eelf.configuration.EELFManager;
// ...
final EELFLogger logger = EELFManager.getInstance().getLogger(this.getClass());
MDC.put("SomeUUID", UUID.randomUUID().toString());
try {
    logger.info("This message will have a UUID-valued 'SomeUUID' MDC attached.");
    // ...
}
finally {
    MDC.clear();
}
```

Serializing

Output of MDCs must ensure that:

- All reported MDCs are logged with both name AND value. Logging output should not treat any MDCs as special.
- All MDC names and values are escaped.

Escaping in Logback configuration can be achieved with:

```
%replace(%replace(%mdc){'\t','\\\\t'}){'\n','\\\\n'}
```

MDC - RequestID

This is often referred to by other names, including "Transaction ID", and one of several (pre-standardization) REST header names including **X-ECOMP-RequestID** and **X-ONAP-RequestID**.

ONAP logging uses a universally unique "**RequestID"** value in log records to track the processing of each client request across all the ONAP components involved in its processing.

This value:

- Is logged as a **RequestID** MDC.
- Is propagated between components in REST calls as an **X-TransactionID** HTTP header.

Receiving the **X-TransactionID** will vary by component according to APIs and frameworks. In general:

```
import javax.ws.rs.core.HttpHeaders;
// ...
final HttpHeaders headers = ...;
// ...
String txId = headers.getRequestHeaders().getFirst("X-TransactionID");
if (StringUtils.isBlank(txId)) {
        txId = UUID.randomUUID().toString();
}
MDC.put("RequestID", txID);
```

Setting the **X-TransactionID** likewise will vary. For example:

```
final String txID = MDC.get("RequestID");
HttpURLConnection cx = ...;
// ...
cx.setRequestProperty("X-TransactionID", txID);
```

MDC - InvocationID

InvocationID is similar to **RequestID**, but where **RequestID** correlates records relating a single, top-level invocation of ONAP as it traverses many systems, **InvocationID** correlates log entries relating to a single invocation of a single component. Typically this means via REST, but in certain cases an **I nvocationID** may be allocated without a new invocation, e.g. when a request is retried.

RequestID and **InvocationID** allow an execution graph to be derived. This requires that:

- The relationship between **RequestID** and **InvocationID** is reported.
- The relationship between caller and recipient is reported for each invocation.

The proposed approach is that:

- Callers:
  - Issue a new, unique **InvocationID** UUID for each downstream call they make.
  - Log the new **InvocationID**, indicating the intent to invoke:
    - With Markers **INVOKE**, and **SYNCHRONOUS** if the invocation is synchronous.
    - With their own **InvocationID** still set as an MDC.
  - Pass the **InvocationID** as an **X-InvocationID** REST header.
- Invoked components:
  - Retrieve the **InvocationID** from REST headers upon invocation, or generate a UUID default.
  - Set the **InvocationID** MDC.
  - Write a log entry with the Marker **ENTRY**. (In EELF this will be to the AUDIT log).
  - Act as per Callers in all downstream requests.
  - Write a log entry with the Marker **EXIT** upon return. (In EELF this will be to the METRIC log).
  - Unset all MDCs on exit.

That seems onerous, but:

- It's only a few calls.
- It can be largely abstracted in the case of EELF logging.

TODO: code.

MDC - PartnerName

This field should contain the name of the client application user agent or user invoking the API.

This is often used for heuristic analysis to identify invocations between ONAP individual ONAP components. Its value has never been clearly stipulated, so a common problem has been a lack of consistency.

There is no clear consensus, but:

- Use the short name of your component, e.g. **xyzdriver**.
- Values should be human-readable.
- Values should be fine-grained enough to disambiguate subcomponents where it's likely to matter. This is subjective.
- Be consistent: your component should ALWAYS report same value.

Real-life examples include **MSO**, **bpmnclient**, **BPELClient**, (all of which are reported by SO), **openECOMP** (SDNC), **vid** (VID!) etc. (See the problem?)

Usage overlaps with **InvocationID**, which doesn't mean **PartnerName** gets retired, but which might mean it serves a more descriptive purpose. (Since it hasn't proven to be a great way of generating a call graph).

MDC - ServiceName

For EELF Audit log records that capture API requests, this field contains the name of the API invoked at the component creating the record (e.g., **Layer3ServiceActivateRequest**).

For EELF Audit log records that capture processing as a result of receipt of a message, this field should contain the name of the module that processes the message.

Usage is the same for indexable logs.

MDCs - the Rest

Other MDCs are logged in a wide range of contexts.

Certain MDCs and their semantics may be specific to EELF log types.

TODO: cross-reference EELF output to v1 doc.

ID	MDC	Description	Required	EELF Audit	EELF Metric	EELF Error	EELF Debug
	Request ID	See above.	Y				
	InvocationID	See above.	Y				
	Service Name	See above.	Y				
	Partner Name	See above.	Y				

1	BeginTimestamp	Date-time that processing activities being logged begins. The value should be represented in UTC and formatted per ISO 8601, such as “2015-06-03T13:21:58+00:00”. The time should be shown with the maximum resolution available to the logging component (e.g., milliseconds, microseconds) by including the appropriate number of decimal digits. For example, when millisecond precision is available, the date-time value would be presented as, as “2015-06-03T13:21:58.340+00:00”.	Y				
2	EndTimestamp	Date-time that processing for the request or event being logged ends. Formatting rules are the same as for the BeginTimestamp field above. In the case of a request that merely logs an event and has not subsequent processing, the EndTimestamp value may equal the BeginTimestamp value.	Y				
3	Elapsed Time	This field contains the elapsed time to complete processing of an API call or transaction request (e.g., processing of a message that was received). This value should be the difference between. EndTimestamp and BeginTimestamp fields and must be expressed in milliseconds.	Y				
4	ServiceInstanceID	This field is optional and should only be included if the information is readily available to the logging component. Transaction requests that create or operate on a particular instance of a service/resource can identify/reference it via a unique “serviceInstanceID” value. This value can be used as a primary key for obtaining or updating additional detailed data about that specific service instance from the inventory (e.g., AAI). In other words: • In the case of processing/logging a transaction request for creating a new service instance, the serviceInstanceID value is determined by either a) the MSO client and passed to MSO or b) by MSO itself upon receipt of a such a request. • In other cases, the serviceInstanceID value can be used to reference a specific instance of a service as would happen in a “MACD”-type request. • ServiceInstanceID is associated with a requestID in log records to facilitate tracing its processing over multiple requests and for a specific service instance. Its value may be left “empty” in subsequent record to the 1 st record where a requestID value is associated with the serviceInstanceID value. NOTE: AAI won’t have a serviceInstanceUUID for every service instance. For example, no serviceInstanceUUID is available when the request is coming from an application that may import inventory data.					
5	VirtualServerName	Physical/virtual server name. Optional: empty if determined that its value can be added by the agent that collects the log files collecting.					
6	StatusCode	This field indicates the high level status of the request. It must have the value COMPLETE when the request is successful and ERROR when there is a failure.	Y				
7	ResponseCode	This field contains application-specific error codes. For consistency, common error categorizations should be used.					
8	ResponseDescription	This field contains a human readable description of the **ResponseCode**.					11
9	Instance UUID	If known, this field contains a universally unique identifier used to differentiate between multiple instances of the same (named) log writing service/application. Its value is set at instance creation time (and read by it, e.g., at start/initialization time from the environment). This value should be picked up by the component instance from its configuration file and subsequently used to enable differentiation of log records created by multiple, locally load balanced ONAP component or subcomponent instances that are otherwise identically configured.					
10	Severity	Optional: 0, 1, 2, 3 see Nagios monitoring/alerting for specifics/details.					
11	TargetEntity	It contains the name of the ONAP component or sub-component, or external entity, at which the operation activities captured in this metrics log record is invoked.	Y				
12	TargetServiceName	It contains the name of the API or operation activities invoked at the TargetEntity.	Y				
13	Server	This field contains the Virtual Machine (VM) Fully Qualified Domain Name (FQDN) if the server is virtualized. Otherwise, it contains the host name of the logging component.	Y				
14	ServerIP Address	This field contains the logging component host server’s IP address if known (e.g. Jetty container’s listening IP address). Otherwise it is empty.					
15	ServerFQDN	Unclear, but possibly duplicating one or both of **Server** and **ServerIPAddress**.					
16	ClientIP Address	This field contains the requesting remote client application’s IP address if known. Otherwise this field can be empty.					
17	Process Key	This field can be used to capture the flow of a transaction through the system by indicating the components and operations involved in processing. If present, it can be denoted by a comma separated list of components and applications.					
18	Remote Host	Unknown.					
19	AlertSeverity	Unknown.					
20	TargetVirtualEntity	Unknown					
21	ClassName	Defunct. Doesn’t require an MDC.					
22	ThreadID	Defunct. Doesn’t require an MDC.					
23	Custom Field1	(Defunct now that MDCs are serialized as NVPs.)					

24	Custom Field2	(Defunct now that MDCs are serialized as NVPs.)					
25	Custom Field3	(Defunct now that MDCs are serialized as NVPs.)					
26	Custom Field4	(Defunct now that MDCs are serialized as NVPs.)					

Examples

SDC-BE

20170907: audit.log

```
root@ip-172-31-93-160:/dockerdata-nfs/onap/sdc/logs/SDC/SDC-BE# tail -f audit.log
2017-09-07T18:04:03.679Z|||||qtp1013423070-72297||ASDC|SDC-BE|||||||N/A|INFO||||10.42.88.30||o.o.s.v.r.s.
VendorLicenseModelsImpl||ActivityType=<audit>, Desc=< --Audit-- Create VLM. VLM Name: lm4>
```

TODO: this is the earlier output format. Let's find an example which matches the latest line format.

Markers

Markers differ from MDCs in two important ways:

1. They have a name, but no value. They are a tag.
2. Their scope is limited to logger calls which specifically reference them; they are not ThreadLocal.

Logging

Via SLF4J:

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.slf4j.Marker;
import org.slf4j.MarkerFactory;
// ...
final Logger logger = LoggerFactory.getLogger(this.getClass());
final Marker marker = MarkerFactory.getMarker("MY_MARKER");
logger.warn(marker, "This warning has a 'MY_MARKER' annotation.");
```

EELF does not allow Markers to be set directly. See notes on the **InvocationID** MDC.

Serializing

Marker names also need to be escaped, though they're much less likely to contain problematic characters than MDC values.

Escaping in Logback configuration can be achieved with:

```
%replace(%replace(%marker){'\t','\\\\t'}){'\n','\\\\n'}
```

Marker - ENTRY

This should be reported as early in invocation as possible, immediately after setting the **RequestID** and **InvocationID** MDCs.

It can be automatically set by EELF, and written to the AUDIT log.

It must be manually set otherwise.

EELF:

EELF

```
final EELFLogger logger = EELFManager.getAuditLogger();
logger.auditEvent("Entering.");
```

SLF4J:

SLF4J

```
public static final Marker ENTRY = MarkerFactory.getMarker("ENTRY");
// ...
final Logger logger = LoggerFactory.getLogger(this.getClass());
logger.debug(ENTRY, "Entering.");
```

Marker - EXIT

This should be reported as late in invocation as possible, immediately before unsetting the **RequestID** and **InvocationID** MDCs.

It can be automatically reported by EELF, and written to the METRIC log.

It must be manually set otherwise.

EELF:

EELF

```
final EELFLogger logger = EELFManager.getMetricsLogger();
logger.metricsEvent("Exiting.");
```

SLF4J:

SLF4J

```
public static final Marker EXIT = MarkerFactory.getMarker("EXIT");
// ...
final Logger logger = LoggerFactory.getLogger(this.getClass());
logger.debug(EXIT, "Exiting.");
```

Marker - INVOKE

This should be reported by the caller of another ONAP component via REST, including a newly allocated **InvocationID**, which will be passed to the caller.

SLF4J:

SLF4J

```
public static final Marker INVOKE = MarkerFactory.getMarker("INVOKE");
// ...

// Generate and report invocation ID.

final String invocationID = UUID.randomUUID().toString();
MDC.put(MDC_INVOCATION_ID, invocationID);
try {
    logger.debug(INVOKE_SYNCHRONOUS, "Invoking synchronously ... ");
}
finally {
    MDC.remove(MDC_INVOCATION_ID);
}

// Pass invocationID as HTTP X-InvocationID header.

callDownstreamSystem(invocationID, ... );
```

TODO: EELF examples of INVOCATION_ID reporting, without changing published APIs.

Marker - SYNCHRONOUS

This should accompany **INVOKE** when the invocation is synchronous.

SLF4J:

SLF4J

```
public static final Marker INVOKE_SYNCHRONOUS;
static {
    INVOKE_SYNCHRONOUS = MarkerFactory.getMarker("INVOKE");
    INVOKE_SYNCHRONOUS.add(MarkerFactory.getMarker("SYNCHRONOUS"));
}
// ...

// Generate and report invocation ID.

final String invocationID = UUID.randomUUID().toString();
MDC.put(MDC_INVOCATION_ID, invocationID);
try {
    logger.debug(INVOKE_SYNCHRONOUS, "Invoking synchronously ... ");
}
finally {
    MDC.remove(MDC_INVOCATION_ID);
}

// Pass invocationID as HTTP X-InvocationID header.

callDownstreamSystem(invocationID, ... );
```

TODO: EELF example of SYNCHRONOUS reporting, without changing published APIs.

Errorcodes

Errorcodes are reported as MDCs.

Exceptions should be accompanied by an errrorcode. Typically this is achieved by incorporating errorcodes into your exception hierarchy and error handling. ONAP components generally do not share this kind of code, though EELF defines a marker interface (meaning it has no methods) **EELFResolvableErrorEnum**.

A common convention is for errorcodes to have two components:

1. A **prefix**, which identifies the origin of the error.
2. A **suffix**, which identifies the kind of error.

Suffixes may be numeric or text. They may also be common to more than one component.

For example:

```
COMPONENT_X.STORAGE_ERROR
```

Output Format

Several considerations:

1. Logs should be human-readable (within reason).
2. Shipper and indexing performance and durability depends on logs that can be parsed quickly and reliably.
3. Consistency means fewer shipping and indexing rules are required.

Text Output

ONAP needs to strike a balance between human-readable and machine-readable logs. This means:

- The use of PIPE (|) as a delimiter. (Previously tab, and before that ... pipe).
- Escaping all messages, exceptions, MDC values, Markers, etc. to replace tabs and pipes in their content.
- Escaping all newlines with **\n** so that each entry is on **one line**.

In logback, this looks like:

```
<property name="defaultPattern" value="%nopexception%logger
|%date{yyyy-MM-dd'T'HH:mm:ss.SSSXXX,UTC}
|%level
|%replace(%replace(%replace(%message){'\t','\\\\t'}){'\n','\\\\n'}){'|','\\\\|'}
|%replace(%replace(%replace(%mdc){'\t','\\\\t'}){'\n','\\\\n'}){'|','\\\\|'}
|%replace(%replace(%replace(%rootException){'\t','\\\\t'}){'\n','\\\\n'}){'|','\\\\|'}
|%replace(%replace(%replace(%marker){'\t','\\\\t'}){'\n','\\\\n'}){'|','\\\\|'}
|%thread
|%n"/>
```

The output of which, with MDCs, a Marker and a nested exception, **with newlines added for readability**, looks like:

```
org.onap.example.component1.subcomponent1.LogbackTest
|2017-08-06T16:09:03.594Z
|ERROR
|Here's an error, that's usually bad
|key1=value1, key2=value2 with space, key5=value5"with"quotes, key3=value3\nwith\nnewlines,
key4=value4\twith\ttabs
|java.lang.RuntimeException: Here's Johnny
\n\tat org.onap.example.component1.subcomponent1.LogbackTest.main(LogbackTest.java:24)
\nWrapped by: java.lang.RuntimeException: Little pigs, little pigs, let me come in
\n\tat org.onap.example.component1.subcomponent1.LogbackTest.main(LogbackTest.java:27)
|AMarker1
|main
```

Default Logstash indexing rules understand output in this format.

XML Output

For Log4j 1.X output, since escaping is not supported, the best alternative is to emit logs in XML format.

There may be other instances where XML (or JSON) output may be desirable. Default indexing rules support

Default Logstash indexing rules understand the XML output of Log4J's XMLLayout.

Note that we're hoping that support for indexing of XML output can be deprecated during Beijing. This relies on the adoption of ODL Carbon, which should eliminate any remnant of Log4J1.X.

Output Location

Standardization of output locations makes logs easier to locate and ship for indexing.

Logfiles should default to beneath **/var/log**, and beneath **/var/log/ONAP** in the case of core ONAP components:

```
/var/log/ONAP/<component>[/<subcomponent>]/*.log
```

For the duration of Beijing, logs will be written to a separate directory, **/var/log/ONAP_EELF**:

```
/var/log/ONAP_EELF/<component>[/<subcomponent>]/*.log
```

Configuration

Logging providers should be configured by file. Files should be at a predictable, static location, so that they can be written by deployment automation. Ideally this should be under **/etc/ONAP**, but compliance is low.

Locations

All logger provider configuration document locations namespaced by component and (if applicable) subcomponent by default:

```
/etc/ONAP/<component>[/<subcomponent>]/<provider>.xml
```

Where **<provider>.xml**, will typically be one of:

1. logback.xml
2. log4j.xml
3. log4j.properties

Reconfiguration

Logger providers should reconfigure themselves automatically when their configuration file is rewritten. All major providers should support this.

The default interval is 10s.

Overrides

The location of the configuration file MAY be overrideable, for example by an environment variable, but this is left for individual components to decide.

Archetypes

Configuration archetypes can be found in the ONAP codebase. Choose according to your provider, and whether you're logging via EELF. Efforts to standardize them are underway, so the ones you should be looking for are where pipe (|) is used as a separator. (Previously it was "|").

Retention

Logfiles are often large. Logging providers allow retention policies to be configured.

Retention has to balance:

- The need to index logs before they're removed.
- The need to retain logs for other (including regulatory) purposes.

Defaults are subject to change. Currently they are:

1. Files <= 50MB before rollover.
2. Files retain for 30 days.
3. Total files capped at 10GB.

In Logback configuration XML:

```
<appender name="file" class="ch.qos.logback.core.rolling.RollingFileAppender">
    <file>${outputDirectory}/${outputFilename}.log</file>
    <rollingPolicy class="ch.qos.logback.core.rolling.SizeAndTimeBasedRollingPolicy">
        <fileNamePattern>${outputDirectory}/${outputFilename}.%d{yyyy-MM-dd}.%i.log.zip</fileNamePattern>
        <maxFileSize>50MB</maxFileSize>
        <maxHistory>30</maxHistory>
        <totalSizeCap>10GB</totalSizeCap>
    </rollingPolicy>
    <encoder>
        <!-- ... -->
    </encoder>
</appender>
```

Types of EELF Logs

EELF guidelines stipulate that an application should output log records to four separate files:

1. audit
2. metric
3. error
4. debug

This applies only to EELF logging. Components which log directly to a provider may choose to emit the same set of logs, but most do not.

Audit Log

An audit log is required for EELF-enabled components, and provides a summary view of the processing of a (e.g., transaction) request within an application. It captures activity requests that are received by an ONAP component, and includes such information as the time the activity is initiated, then it finishes, and the API that is invoked at the component.

Audit log records are intended to capture the high level view of activity within an ONAP component. Specifically, an API request handled by an ONAP component is reflected in a single Audit log record that captures the time the request was received, the time that processing was completed, as well as other information about the API request (e.g., API name, on whose behalf it was invoked, etc).

Metrics Log

A metrics log is required for EELF-enabled components, and provides a more detailed view into the processing of a transaction within an application. It captures the beginning and ending of activities needed to complete it. These can include calls to or interactions with other ONAP or non-ONAP entities.

Suboperations invoked as part of the processing of the API request are logged in the Metrics log. For example, when a call is made to another ONAP component or external (i.e., non-ONAP) entity, a Metrics log record captures that call. In such a case, the Metrics log record indicates (among other things) the time the call is made, when it returns, the entity that is called, and the API invoked on that entity. The Metrics log record contain the same RequestID as the Audit log record so the two can be correlated.

Note that a single request may result in multiple Audit log records at an ONAP component and may result in multiple Metrics log records generated by the component when multiple suboperations are required to satisfy the API request captured in the Audit log record.

Error Log

An error log is required for EELF-enabled components, and is intended to capture info, warn, error and fatal conditions sensed (“exception handled”) by the software components.

Debug Log

A debug log is optional for EELF-enabled components, and is intended to capture whatever data may be needed to debug and correct abnormal conditions of the application.

Engine.out

Console logging may also be present, and is intended to capture “system/infrastructure” records. That is stdout and stderr assigned to a single “engine.out” file in a directory configurable (e.g. as an environment/shell variable) by operations personnel.

New ONAP Component Checklist

By following a few simple rules:

- Your component's output will be indexed automatically.
- Analytics will be able to trace invocation through your component.

Obligations fall into two categories:

1. Conventions regarding configuration, line format and output.
2. Ensuring the propagation of contextual information.

You must:

1. Choose a Logging provider and/or EELF. Decisions, decisions.
2. Create a configuration file based on an existing archetype. See Configuration.
3. Read your configuration file when your components initialize logging.
4. Write logs to a standard location so that they can be shipped by Filebeat for indexing. See Output Location.
5. Report transaction state:
    a. Retrieve, default and propagate **RequestID**. See MDC - RequestID.
    b. At each invocation of one ONAP component by another:
        i. Initialize and propagate **InvocationID**. See MDC - Invocation ID.
        ii. Report **INVOKE** and **SYNCHRONOUS** markers in caller.
        iii. Report **ENTRY** and **EXIT** markers in recipient.
6. Write useful logs!

They are unordered.

What's New

(Including what **WILL** be new in v1.2 / R2).

1. Field separator reverted to pipe.
2. Dual appenders in Logback and Log4j reference configurations:
    a. Indexable, for shipping and indexing.
    b. EELF, for backward compatibility.
    c. Minor changes to path conventions.
3. XML output deprecated (required only for Log4j1.2, which is also expected to go).
4. Improved documentation of semantics and usage (including initialization and propagation via ThreadLocal and HTTP headers) for existing MDCs and attributes.
5. Add MDCs/Markers + usage for invocation IDs, allowing call graphs to be built without reliance on heuristics.
6. Revisiting persistence (a clear requirement) and rollover settings, based on feedback from operations.
7. More discussion of How to Log. (Where previously guidelines were largely concerned with architecture and mechanics).

8. Locking in other changes proposed in R1, including MDC serialization, escaping, etc. These can be treated as accepted. (Note that they only affect indexable output).

In addition, we expect to provide (as a Beijing deliverable) a minimal, synthetic component as an example of best-practices, and this will provide all code examples for this guide. (Does that mean the example will log via EELF, or will we end up with two variants?)