ONAP Application Logging Specification v1.2 (Casablanca)

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.
    - MDC = Mapped Diagnostic Context
- Human and machine-readable output format(s).
- Files, locations and other conventions.

Original AT&T ONAP Logging guidelines (pre amsterdam release) - for historical reference only: https://wiki.onap.org/download/attachments/1015849/ONAP%20application%20logging%20guidelines.pdf?api=v2

The Acumos logging specification follows this document at https://wiki.acumos.org/display/OAM/Log+Standards

Logback reference: Logging Developer Guide#Logback.xml based on https://gerrit.onap.org/r/#/c/62405

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 is being deployed using ONAP Operations Manager since the amsterdam release - see usage in Logging Analytics Dashboards (Kibana)

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

Supported Languages

Java (including Scala and Clojure - because they compile to the same bytecode) and Python are supported, but conventions may be implemented by other technologies like GO.

Java

The library and aop wrapper are written in Java and will work for Clojure (thanks to a discussion with Shwetank Dave that reminded me of languages that compile to bytecode) and Scala as well. see: https://git.onap.org/logging-analytics/tree/reference

Clojure

See the java libraries

Scala

see the java libraries

Python

https://lists.onap.org/g/onap-discuss/topic/logging_python_logging/25307286?p=,,,20,0,0,0::recentpostdate%2Fsticky,,,20,2,0,25307286

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.

Logging Specification Compliance

For the casablanca release the logging specification has been updated and finalized as of June 2018. Implementation of this specification is required but the method of implementation is optional based on each team's level of possible engagement.

High Level the changes are introduction of MDC's (key/value pairs like requestID=...) and Markers (labels like ENTRY/EXIT)

The ELK stack (indexes and dashboards) is focused on the new specification in Casablanca - if time permits we will have a migration path and/or support for both the older Amsterdam/Beijing release and Casablanca - **LOG-584** - Getting issue details... STATUS

Level 0: Log specification unchanged

Just keep your code as-is and commit to migrating to the the MDC and Marker focused specification for Dublin - logs will be the older format for now

Level 1: Log specification compliance via local project changes, library or wrapper

Continue to use your own library/wrapper if you have one like in SDC and AAI, change individual source files.

Level 2: Log specification compliance via SLF4J supplied logging-analytics library

Use Luke's SLF4J library directly as wrapper for MDCs and Markers by calling the library from within each function.

Level 3: Log specification compliance via AOP library over SLF4J supplied logging-analytics library

Use a spring based Aspect library that emits Markers automatically around function calls and retrofit your code to log via Luke's SLF4J for internal log messages.

ONAP Application Logging Specification v1.2 (Casablanca)#LoggingWithAOP

Logging Library Location and Use

see https://git.onap.org/logging-analytics/tree/reference/logging-slf4j

and usage ONAP Development#DeveloperUseoftheLoggingLibrary ONAP Development#KubernetesDevOps and Logging User Guide#LoggingDevOps

see Spring AOP example (minimal changes to existing code base) WIP in **LOG-135** - Getting issue details... STATUS documented at ONAP Development#LoggingWithAOP

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, metrics, ~~security~~ error 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 ONAP Application Logging Specification v1.2 (Casablanca) #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 as the default provider.

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 ONAP Application Logging Specification v1.2 (Casablanca)#Configuration.

See HELM template https://git.onap.org/logging-analytics/tree/reference/provider/helm/logback

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 ONAP Application Logging Specification v1.2 (Casablanca)#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 Casablanca 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 and scalable system such as ONAP this is critical to understanding behavior and performance.

General

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

- Use a logging facade such as SLF4J or EELF.
- Write log messages in English.
- Write meaningful messages. Consider what will be useful to consumers of logger output.
- Log at the appropriate level. Be aware of the volume of logs that will be produced.
- Safeguard the information in exceptions, and ensure it is never lost.
- Use error codes to characterize exceptions.
- 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.

Standard Attributes

These are attributes common to all log messages. They are either:

- Explicitly required by logging APIs:
  - Logger
  - Level
  - Message
  - Exception (note that exception is the only standard attribute that may routinely be empty).
- Implicitly derived by the logging provider:
  - Timestamp
  - Thread.

Which means you normally can't help but report them.

See https://www.slf4j.org/api/org/slf4j/Logger.html and https://logback.qos.ch/manual/layouts.html#ClassicPatternLayout for their origins and use.

Logger Name

This indicates the name of the logger that logged the message.

In Java it is convention to name the logger after the class or package using that logger.

- In Java, report the class or package name.
- In Python, the class or source filename.

Most other languages will fit one of those patterns.

Level

Severity, typically drawn from the enumeration **{TRACE, DEBUG, INFO, WARN, ERROR}**.

Think carefully about the information you report at each log level. The default log level is INFO.

Some loggers define non-standard levels, like FINE, FINER, WARNING, SEVERE, FATAL or CRITICAL. Use these judiciously, or avoid them.

Message

The free text payload of a log event.

This is the most important item of information in most log messages. See ONAP Application Logging Specification v1.2 (Casablanca)#General guidelines.

Internationalization

Diagnostic log messages generally do not need to be internationalized.

Parameterization

Parameterized messages allow serialization to be deferred until AFTER level threshold checks. This means the cost is never incurred for messages that won't be written.

- Favor parameterized messages, especially for INFO and DEBUG logging.
- Perform expensive serialization in the **#toString** method of wrapper classes.

For example:

```
logger.debug("Relax - this won't hurt: {}", new ToStringWrapper(costlyToSerialize));
```

Exception

The error stacktrace, where applicable.

Log unabridged stacktraces upon error.

When rethrowing, ensure that frame information is not lost:

- By logging the original exception at the point where it was caught.
- By setting the original exception as the cause when rethrowing.

Timestamp

Logged as an ISO8601 UTC datetime. Millisecond (or greater) precision is preferable.

For example:

```
2018-07-05T20:21:34.794Z
```

Offset timestamps are OK provided the offset is included. (In the above example, the "Z" is a shorthand indicating an offset of zero – UTC).

Thread

The name of the thread from which the log message was emitted.

Thread names don't necessarily convey useful information, and their reliability depends on the thread model implemented by different runtimes, but they are sometimes used in heuristic analysis.

Efficiency

There is tension between utility and efficiency. IO bandwidth is finite, and the cost of serialization can be significant, especially at higher diagnostic levels.

Methods and Line Numbers

Many loggers can use reflection to emit the originating (Java) method, and even individual line numbers.

This information is certainly useful, but very expensive. Most logging implementations recommend that this not be enabled in production.

Level Thresholds

Level indicates severity.

Logger output is typically filtered by logger and level. The default logging level is INFO, so particular consideration should be given to the efficiency of INFO-level logging.

When DEBUG-level logging is configured, it's probably for good reason, and a greater overhead is expected. Be aware that it's not unusual for DEBUG logging to be left enabled inadvertently, however.

WARN and ERROR-level messages are of higher value, and comparatively rare, so their cost is less of a concern.

Conditionals

A common pattern is to place conditionals around (expensive) serialization.

For example:

```
if (logger.isDebugEnabled()) {
    logger.debug("But this WILL hurt: " + costlyToSerialize);
}
```

Parameterized logging is preferable.

Context

MDCs

A Mapped Diagnostic Context (MDC) allows an arbitrary string-valued attribute to be attached to a Java thread via a ThreadLocal variable. 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 ONAP Application Logging Specification v1.2 (Casablanca)#Text Output).

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

Example

From Luke Parker's call graph work in https://git.onap.org/logging-analytics/tree/reference/logging-slf4j-demo

```
LogEntry(markers=ENTRY, logger=ComponentAlpha, requestID=eb3e0dc2-6c3c-4bb7-8ed6-e5cc4ec7aad2,
invocationID=06c815ef-5969-45cc-b319-d0dbcde89329, timestamp=Tue May 08 04:23:27 AEST 2018)
```

Mapped Diagnostic Context Table

- Must be set as early in invocation as possible.
- Must be unset on exit.
- keep in sync with https://wiki.acumos.org/display/OAM/Log+Standards

Pipe Order	Name	Type	Group	Description	Applicable (per log file)	Marker Associations	Moved MDC to standard attribute	Removed (was in older spec)	Required? Y/N/C (C= context dependent) N = not required L=Library provided	Derived	Historical	Acumos ref	Use Cases
1	LogTimestamp	log system		use %d field - see %d{"yyyy-MM-dd'T'HH:mm:ss.SSSXXX",UTC}					L				
2	EntryTimestamp	MDC		if part of an ENTRY marker log					C				
3	InvokeTimestamp	MDC		if part of an INVOKE marker log					C				
4	RequestID	MDC		UUID to track the processing of each client request across all the ONAP components involved in its processing					Y				
5	InvocationID	MDC		UUID correlates log entries relating to a single invocation of a single component In the case of an asynchronous request, the InvocationID should come from the original request					Y				
6	InstanceID	MDC		UUID to differentiate between multiple instances of the same (named) log writing service /application					Y		was InstanceUUID		
7	ServiceInstanceID	MDC							C				
8	thread	log system		use %thread field					L				
9	ServiceName			The service inside the partner doing the call - includes API name					Y				

10	PartnerName			unauthenticated = The part of the URI specifying the agent that the caller used to make the call to the component that is logging the message. authenticated = userid 1. If an authenticated API, then log the userid 2. Otherwise, if the HTTP header "X-ONAP-PartnerName" was provided, then log that (note: this was a direction that we seemed to be going but never completed) 3. Otherwise, if the HTTP header "User-Agent" was provided, then log that 4. Otherwise, log "UNKNOWN" (since the field is currently required, something must be in it)					Y			user	
11	StatusCode			This field indicates the high level status of the request - one of (COMPLETE, ERROR, INPROGRESS)					Y				
12	ResponseCode			This field contains application-specific error codes.					Y				
13	ResponseDesc			This field contains a human readable description of the ResponseCode					Y				
14	level			%level					L				
15	Severity			Logging level by default aligned with the reported log level - one of INFO/TRACE/DEBUG/WARN/ERROR/FATAL					Y			level (but numbers)	
16	ServerIPAddress								C				
17	ElapsedTime								C				
18	ServerFQDN			The VM FQDN if the server is virtualized. Otherwise the host name of the logging component.					Y				

19	ClientIPAddress			This field contains the requesting remote client application's IP address if known. Otherwise empty.					Y				
20	VirtualServerName								C				
21	ContextName								C				
22	TargetEntity			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.					C				
23	TargetServiceName			The name of the API or operation activities invoked (name on the remote /target application) at the TargetEntity.					C				
24	TargetElement			VNF/PNF context dependent - on CRUD operations of VNF/PNFs The IDs that need to be covered with the above Attributes are - VNF_ID OR VNFC_ID : (Unique identifier for a VNF asset that is being instantiated or that would generate an alarms) - VSERVER_ID OR VM_ID (or vmid): (Unique identified for a virtual server or virtual machine on which a Control Loop action is usually taken on, or that is installed as part of instantiation flow) - PNF : (What is the Unique identifier used within ONAP)					C				
25	User	MDC		User - used for %X{user}					C				
26	p_logger	log system		The name of the class doing the logging (in my case the ApplicationController – close to the targetservicename but at the class granular level - this field is %logger					L				

27	p_mdc	log system		allows forward compatability with ELK indexers that read all MDCs in a single field - while maintaining separate MDCs above. The key/value pairs all in one pipe field (will have some duplications currently with MDC's that are in their own pipe – but allows us to expand the MDC list – replaces customvalue1-3 older fields - this field is %mdc					L				
28	p_message	log system		The marker labels INVOKE, ENTRY, EXIT – and later will also include DEBUG, AUDIT, METRICS, ERROR when we go to 1 log file - this field is %marker					L				
	~~RootException~~	~~log system~~		~~%rootException - Dublin spec only~~					~~L~~				
29	p_marker	log system		The marker labels INVOKE, ENTRY, EXIT – and later will also include DEBUG, AUDIT, METRICS, ERROR when we go to 1 log file - this field is %marker					L				

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-TransactionID**, **X-ECOMP-TransactionID**, **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. RequestID be propagated across all interfaces, not just REST Interfaces.

This value:

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

Receiving the **X-ONAP-RequestID** 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-ONAP-RequestID");
if (StringUtils.isBlank(txId)) {
        txId = UUID.randomUUID().toString();
}
MDC.put("RequestID", txID);
```

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

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

Note that it's been suggested that for the duration of Casablanca we report the request ID using all **three** headers:

1. **X-ONAP-RequestID** (canonical)
2. **X-RequestID** (deprecated)
3. **X-TransactionID** (deprecated)

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 METRICS 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.

MDC - InstanceID

(formerly InstanceUUID)

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.

Handles parallel threads or running across a load balanced set of microservices - for identification.

MDC - PartnerName

This field should contain the name of the client application user agent or user invoking the API. The identification of the entity that made the request being served. For a serving API that is authenticating the request, this should be the authenticated username or equivalent (e.g. a userid or a mechid).

For example SDC-BE instead of just SDC for the overall pods

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**. (try to incorporate both levels - the container name and the pod the container is in within the kubernetes deployment)
- 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

The URI that the caller used to make the call to the component that is logging the message.

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., **Layer3Se rviceActivateRequest**).

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.

MDC - 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. And INPROGRESS for states between the two.

Discussion: status/response/severity relationship

status = global, response below is app specific

Ability to render severity-like line in a non-debug log

MDC - ResponseCode

This field contains application-specific error codes. For consistency, common error categorizations should be used.

MDC - Severity

OPS specific

Use/Map existing https://www.slf4j.org/api/org/apache/commons/logging/Log.html

ENUM is INFO/TRACE/DEBUG/WARN/ERROR/FATAL

By default - align this severity with the reported log level

(optionally a way to map actual level from reported level if required)

MDC - ServerFQDN

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.

Best effort (ip, fqdn)

(previously covered by removed "Server" field)

redundancy between clientIP, server, virtualServer name is OK - and helpfull for runtime OPS/Hybrid envs

supercedes virtualServerName

Report what is in the http header

Discussion: roll all 3 fqdn, hostname or ip into one field - do we ever need two of the 3 fields concurrently?

TODO: Verify what is also available from a filebeat agent when it exists

MDC - ClientIPAddress

This field contains the requesting remote client application's IP address if known. Otherwise this field can be empty.

We don't differentiate between inside/outside ONAP for the IP - this supports hybrid environments

Derived from the system

redundancy between clientIP, server, virtualServer name is OK - and helpfull for runtime OPS/Hybrid envs

Discussion: do we need both ip and fqdn fields?

Report what is in the http header

MDC - EntryTimestamp

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".

Context dependent on whether part of an ENTRY marker

Audit requires this field

MDC - InvokeTimestamp

Timestamp on invocation start.

Context dependent on whether part of an INVOKE marker

metrics needs this field.

MDC - 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.

Example: SDC-BE

MDC - TargetServiceName

It contains the name of the API or operation activities invoked (name on the remote/target application) at the TargetEntity.

Example: Class name of rest endpoint

Discussion: on building call graph vs human readable single line - keep for human readable

Used as valuable URI - to annnote invoke marker

Review in terms of ONAP Application Logging Specification v1.2 (Casablanca)#Marker-INVOKE - possiblly add INVOKE-return - to filter reporting

TBD: Coverage by log file type (debug, trace, ...)

TBD: cover off discussion on reducing log files to two (DEBUG/rest) for C* release

MDC - TargetElement

VNF/PNF context dependent - on CRUD operations of VNF/PNFs

The IDs that need to be covered with the above Attributes are

- VNF_ID OR VNFC_ID : (Unique identifier for a VNF asset that is being instantiated or that would generate an alarms)
- VSERVER_ID OR VM_ID (or vmid): (Unique identified for a virtual server or virtual machine on which a Control Loop action is usually taken on, or that is installed as part of instantiation flow)
- PNF : (What is the Unique identifier used within ONAP)

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.

Deprecation

Indexing makes many of the remaining attributes redundant. So for example:

- There is considerable duplication:
  - **BeginTimestamp**, **EndTimestamp**, **ElapsedTime**. These are all captured elsewhere (and **ElapsedTime** is even redundant within that triplet).
  - **Server**, **ServerIPAddress**, **ServerFQDN**, **VirtualServiceName**. Overkill. Should be one, plus optionally **ClientIPAddress** (or some variant thereof).
  - **TargetEntity**, **TargetServiceName**, not obviously different to similar attributes.
- There is junk:
  - **Severity**? Nagios codes?
  - **ProcessKey**?
  - All the stuff that's already grayed out in the table above.
  - People may defend these individually, maybe vigorously, but they're domain-specific:
    - That absolutely doesn't mean they can't be used.
    - Beats configuration allows ad hoc contexts to be indexed.
    - But perhaps they don't belong in this kind of spec.
- Redundant attributes *do* matter, because:
  - Populating and propagating everything prescribed by the guide approaches being prohibitive. People won't do it, and people *don't* do it.
  - If something might be in one of several attributes then that's worse than it being in just one.
- That means:
  - We're left with only two MANDATORY attributes, necessary to build invocations graphs:
    - **RequestID** - top-level transactions.
    - **InvocationID** - inter-component invocations.
  - And a minimal number of OPTIONAL descriptive attributes: **ServiceInstanceID**, **InstanceID**, **Server, StatusCode**, **ResponseCode**, **ResponseDescription**.
  - Those are the ones we need to document clearly, support in APIs, etc.
  - That's <=10, a manageable number.
  - And again, that matters because if the number isn't manageable, people won't (and don't) comply.

Some of that is contentious, but it's just talking points at this stage. We've tiptoed around the issue of extant conventions, and the ongoing result is a lot of attributes that nobody's really sure how to use, and which don't result in better logs. In Casablanca it's time to be less conservative.

Examples

Markers

Markers unambiguously assign semantics to individual log messages. They allow messages that have a specific *meaning* to be cheaply and easily identified in logger output, without inherently unreliable (and more costly, and less easily enforced) schemes like scanning for magic strings in the text of each log message.

ONAP logging requires the emission of Markers reporting **entry**, **exit** and **invocation** as the execution of requests pass between ONAP components. This information is used to generate a **call graph**.

ONAP components are also free to use Markers for their own purposes. Any Markers that are logged will be automatically indexed by Logstash.

Markers differ from MDCs in two important ways:

1. They have a name, but no value. They are a tag - like a label.
2. They are specified explicitly on invocation. They are not ThreadLocal, and they do not propagate.

EELF's implementation can be modified to emit Markers, but its public APIs do not allow them to be passed in by callers.

Log4J docs, SLF4J docs

see code on reference folder in git clone ssh://michaelobrien@gerrit.onap.org:29418/logging-analytics

Examples

Marker formatting is using tabs - **LOG-553** - Getting issue details... STATUS - see https://git.onap.org/logging-analytics/tree/reference/logging-slf4j/src/test/resources/logback.xml#n9

```
Note there are 3 tabs (see p_mak in logback.xml) delimiting the MARKERS (ENTRY and EXIT) at the end of each line
```

```
<property name="p_mak" value="%replace(%replace(%marker){'\t', '\\\\t'}){'\n','\\\\n'}"/>
```

```
2018-07-05T20:21:34.794Z	http-nio-8080-exec-2	INFO	org.onap.demo.logging.
ApplicationService	InstanceID=ede7dd52-91e8-45ce-9406-fbafd17a7d4c, RequestID=f9d8bb0f-4b4b-4700-9853-
d3b79d861c5b, ServiceName=/logging-demo/rest/health/health, InvocationID=8f4c1f1d-5b32-4981-b658-e5992f28e6c8,
InvokeTimestamp=2018-07-05T20:21:26.617Z, PartnerName=, ClientIPAddress=0:0:0:0:0:0:0:1,
ServerFQDN=localhost	ENTRY
2018-07-05T20:22:09.268Z	http-nio-8080-exec-2	INFO	org.onap.demo.logging.
ApplicationService	ResponseCode=, InstanceID=ede7dd52-91e8-45ce-9406-fbafd17a7d4c, RequestID=f9d8bb0f-
4b4b-4700-9853-d3b79d861c5b, ServiceName=/logging-demo/rest/health/health, ResponseDescription=,
InvocationID=8f4c1f1d-5b32-4981-b658-e5992f28e6c8, Severity=, InvokeTimestamp=2018-07-05T20:21:26.617Z,
PartnerName=, ClientIPAddress=0:0:0:0:0:0:0:1, ServerFQDN=localhost, StatusCode=
EXIT
```

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

TODO: add table detailing which log files each marker is a part of - from a use case perspective

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. Candidate for framework

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 METRICS 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, ... );
```

EELF examples of INVOCATION_ID reporting, without changing published APIs.

Marker - INVOKE-RETURN

This should be reported by the caller of another ONAP component via REST on return.

InvokeTimestamp context dependent MDC will be reported here.

SLF4J:

SLF4J

```
TBD
```

Marker - INVOKE-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, ... );
```

EELF example of SYNCHRONOUS reporting, without changing published APIs.

Error Codes

Error codes are reported as MDCs.

TODO: add to table

Exceptions should be accompanied by an error code. Typically this is achieved by incorporating error codes 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 error codes 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

TODO: 20190115 - do not take the example in this section until I reverify it in terms of the reworked spec example in

https://gerrit.onap.org/r/#/c/62405/20/reference/logging-kubernetes/logdemonode/charts/logdemonode/resources/config/logback.xml

LOG-630 - Getting issue details... STATUS

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, we will expand on JSON support

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.

Expand on out-of-container locations off /dockerdata-nfs

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**:

expand on Casablanca differences, and adding as a config setting in OOM

```
/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 https://git.onap.org/logging-analytics/tree/. 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. metrics
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.

This includes previous logs that went to application.log

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.

Application Log (deprecated)

see example in https://git.onap.org/oom/tree/kubernetes/portal/charts/portal-sdk/resources/config/deliveries/properties/ONAPPORTALSDK/logback.xml

We no longer support this 5th log file - see error.log

Log File Locations

There are several locations where logs are available on the host, on the nfs share and in each application and filebeat container

Logs on each Kubernetes host VM - docker empty.dir shares

Logs on the /dockerdata-nfs share across hosts

Logs on each microservice docker container - /var/log/onap

Logs on each filebeat docker container sidecar - /var/log/onap

New ONAP Component Checklist

Add this procedure to the Project Proposal Template

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 ONAP Application Logging Specification v1.2 (Casablanca)#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 ONAP Application Logging Specification v1.2 (Casablanca)#Output Location.
5. Report transaction state:
    a. Retrieve, default and propagate **RequestID**. See ONAP Application Logging Specification v1.2 (Casablanca)#MDC - RequestID.
    b. At each invocation of one ONAP component by another:
        i. Initialize and propagate **InvocationID**. See ONAP Application Logging Specification v1.2 (Casablanca)#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?)

Pending Specification Work

id	date	item	details	status
	20180614	MDC ClientIPAddress	Ask question of OPS to remove this field - 20180419	todo
	20180614	MDC ResponseCode / ResponseDescription	expand/find note 1*	todo

Developer Guide

see separate page (cross releases) in Logging Developer Guide