# API Baseline Logging

## Logging Overview

This section details guidelines and components that facilitate the creation of an API logging philosophy, although non-API type applications can be implemented using these logging concepts as well.

## Logging Framework Used within Each API

This section will discuss logging framework capabilities for two of the more prevalent application development environments, namely Java (Enterprise as well Standard Editions) and Microsoft .NET

## Logging with Java supported API implementations

This section discusses the most popular java logging frameworks:

* **Simple Logging Facade for Java** (SLF4J) which is a logging facade that provides a common interface for different logging frameworks.
* **Log4j 2** and its predecessor Log4J and Logback.

### Simple Logging Facade for Java (SLF4J)

The Simple Logging Facade for Java (SLF4J) serves as a simple facade for various logging frameworks (e.g., log4j 2, java.util.logging (jul), Logback) allowing the end user to select the desired logging framework at deployment time.

#### Subset of Features

* The separation of the client API from the logging backend reduces the coupling between an application and any logging framework. This can make it easier to integrate with existing logging frameworks or to integrate functionality into application projects that have already made a choice of logging backend.

### Apache Log4j 2

Apache Log4j 2 is an upgrade to Log4j and provides significant improvements over its predecessor, in addition it provides many of the improvements available in Logback while fixing some inherent problems in Logback’s architecture.

#### Subset of Features

* API Separation

The API for Log4j is separate from the implementation making it clear for application developers which classes and methods they can use while ensuring forward compatibility. This allows the Log4j team to improve the implementation safely and in a compatible manner.

* Improved Performance

Log4j 2 contains next-generation Asynchronous Loggers based on the LMAX Disruptor library. In multi-threaded scenarios Asynchronous Loggers have higher throughput and orders of magnitude lower latency than Log4j and Logback.

### Logback

Logback is a successor to the popular log4j project, adding additional features from log4j and is conceptually similar to log4j. Subset of Features

* Faster implementation

Based on the previous work on log4j, Logback internals have been re-written to perform more efficiently on certain critical execution paths. Logback components are both faster, and they have a smaller memory footprint.

* Logback-classic “speaks” SLF4J natively

Since the ***Logger*** class in Logback-classic implements the SLF4J API natively, there is virtually no overhead when invoking a SLF4J logger with Logback classic as the underlying implementation. Additionally, since Logback-classic strongly encourages the use of SLF4J as its client API, if the need arises the switch to log4j or to java.util.logging.

### java.util.logging (jul)

The Java Runtime Java provides a default framework in the java.util.logging package although This most developers would agree that java.util.logging is being replaced with 3rd party logging frameworks.

#### Subset of Features

java.util.logging in included in the Java Runtime so third-party integration through jar files is not required.

## Logging with .NET supported API implementations

### Overview of .NET Framework and .NET Core

.NET Core is a cross platform, open source re-implementation of the .NET Framework. It is actively maintained by Microsoft and a huge community of developers over on GitHub. This section focuses on internal and external logging functionality supplied by .NET Core.

The following is a very high level over view of the two logging platforms:

* .NET Core

.NET Core is a cross-platform version of .NET for building websites, services, and console apps.

* .NET Framework

.NET Framework is a Windows-only version of .NET for building any type of app that runs on Windows.

#### .NET Core Logging Capabilities

ASP.NET Core supports a logging API that works with a variety of logging providers. Built-in providers facilitate the sending logs to one or more destinations plus plug in capability to a third-party logging framework.

.NET Core contains the following built-in logging providers:

* Console

The ***Microsoft.Extensions.Logging.Console*** provider package sends log output to the console.

* Debug

The ***Microsoft.Extensions.Logging.Debug*** provider package writes log output by using the ***System.Diagnostics.Debug*** class (Debug.WriteLine method calls).

* EventSource

For apps that target ASP.NET Core 1.1.0 or higher, the ***Microsoft.Extensions.Logging.EventSource*** provider package can implement event tracing.

* EventLog

The Microsoft.Extensions.Logging.EventLog provider package sends log output to the Windows Event Log.

* TraceSource

The ***Microsoft.Extensions.Logging.TraceSource*** provider package uses the ***System.Diagnostics.TraceSource*** libraries and providers.

* Azure App Service

The ***Microsoft.Extensions.Logging.AzureAppServices*** provider package writes logs to text files in an Azure App Service app's file system and to blob storage in an Azure Storage account. The provider is available only for apps that target ASP.NET Core 1.1 or later.

## JSON Format for Logged API Messages

This section presents a **proposed** standardized layout that will facilitate the creation of a JSON logging item. The following UML API Logging Class Diagram is used to define the data elements that will be logged, in a JSON format, by the appropriate logging infrastructure.

### API Logging Model

Figure 1 API Logging Model

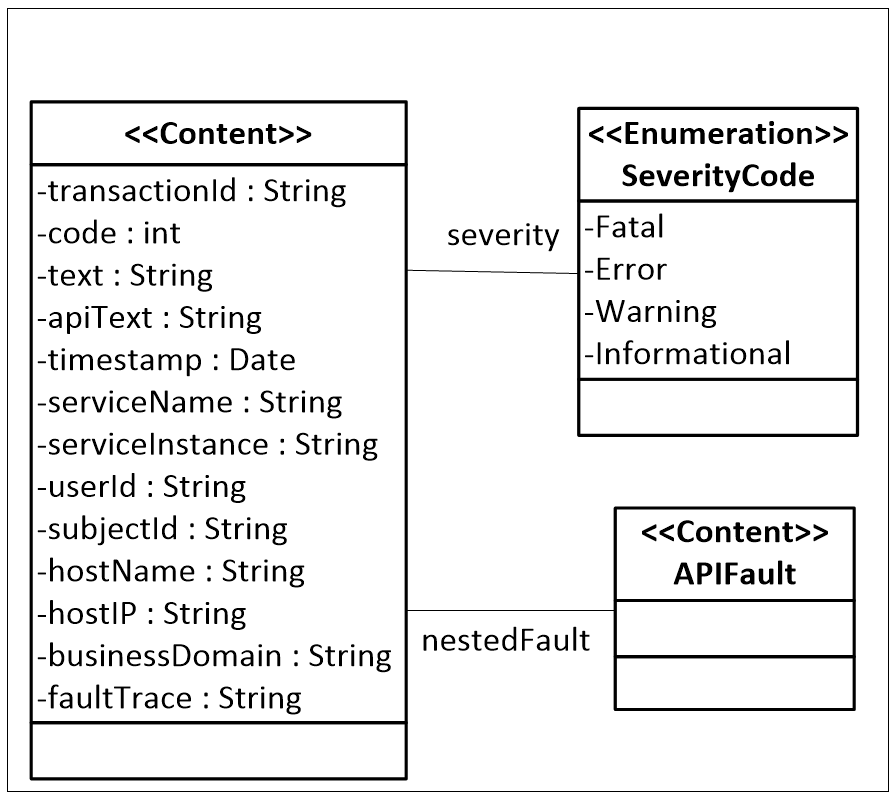


Table 1: API Logging Description Table

| **Element** | **Type** | **Required** | **Example** | **Description** |
| --- | --- | --- | --- | --- |
| transactionID | String | Yes | a20014d3e17ff4fcb578-b7a6-41f9-b974-cef684d5ec62 | A unique identifier passed to the service in the message header. |
| code | int | Yes | 404 | A descriptor of the fault condition, and generated by the service. It could be an application generated error code or it could map to the HTTP Response Code. |
| txt | String | Yes | Service Not Found | A technical textual description of the fault condition, and generated by the service. |
| apiText | String | No | Requested service not found. Validate invocation URL. | A user interface friendly text description of the fault condition. The service generating the fault may populate this field. |
| severity | String | Yes | Error | Enumeration:  **Fatal** – Severe error that causes premature termination of service execution. The service would not able to continue processing the request, and a response message would not be sent. An example would be an unrecoverable system level error.  **Error** – Significant error that causes the service to return a fault. An exception occurred that prevented the service from completing the request normally and a fault was returned. Examples might include a database constraint error, or disk full.  **Warning** – Significant error that does not result in the return of a fault, but is otherwise important to understanding service behavior. A recoverable exception that does not result in a fault message. Abnormal retries on a database connection for example. May also include interesting events not generated by exceptions, but useful in forensic analysis  **Information** – Not an error, but information that is otherwise important to understanding service behavior. Example might be shutting the service down for maintenance. |
| nestedFault | APIFault | No |  | Nested structure of faults bubbled up from lower in the chain of service calls.  Must be populated if fault is triggered by another fault. Is not populated if fault is triggered by an internal exception.  Refer to the ASG\_API Playbook\_09.01 Exception Handling\_Section document for a description of the API Fault. |
| timestamp | Time stamp | Yes | 201805-05T08:15:30-05:00 | Timestamp of the time the error occurred. |
| serviceName | String | Yes | Vet360 Veteran Address Lookup | Unique name of the service. The example is a hypothetical one. Refer to the Naming Standards document for guidelines. |
| serviceInstance | String | Yes | https://dev-exp-vet360.va.gov:8081/api/v1/veteranportal/address | This is the service endpoint. Refer to the Naming Standards document for guidelines. |
| userId | String | Yes | CN=Test User … | Fully qualified identifier of the user passed on the request. |
| subjectId | String | Yes | "subjectID": "Id-1234567890V654321, “IdType”: “NI” … | Fully qualified identifier of the subject of the request. This would typically be the veteran or beneficiary but may be any entity that is the primary subject of the request and/or important to understanding the error. |
| hostName | String | No | vaaacmhvapp12 | Name of the individual host within the cluster hosting the service instance. |
| hostIP | String | No | 101.34.28.134 | IP address of the individual host. |
| businessDomain | String | No | Vet360 Veteran Address Lookup | Name of the business domain |
| fault Trace  (or equivalent) | String | No | java.io.IOException: Unable to find file 'topicalIndex.csv' | Detailed information on the state of the execution of service code package when the error occurred |

The following example contains a minimized set of HTTP Response Message Header name-value pair parameters. Some elements in this figure are truncated for readability.

Figure 2 Logging Example

{

"APIError": {

"transaction ID": "a20014d3e17ff4fcb578-b7a6-41f9-b974-cef684d5ec62",

"code": "404",

"text": " Invalid URL Service Not Found ",

"apiText": " Requested service not found. Validate invocation URL ",

"severity": "Fatal",

"nestedFault": {

"code": "400",

"text": "NotFound"

"apiText": " Requested service not found. Validate invocation URL ",

}

"timestamp": "2018-05-05T08:15:30-05:00",

"serviceName": " Vet360 Veteran Address Lookup ",

"serviceInstance": "https://dev-exp-vet360.va.gov:8081/api/v1/veteranportal/addr...

"userID": "CN=Test …

"subjectID": "Id-1234567890V654321, “IdType”: “NI” …

"hostName": "vaaacmhvapp12",

"hostIP": "101.34.28.134",

"businessDomain": " Vet360 Veteran Address Lookup ",

"faultTrace": "java.io.IOException: Unable to find file 'topicalIndex.csv' ..."

}

}

## API Logging Policy Guidelines

The following table is a modified version of the Appendix C Extracted Policy Assertions table contained in the Enterprise Shared Services (ESS) Exception Handling Guidelines document. The content has been modified to be consistent with an API/Restful Service implementation

Refer to **Figure 1 API Logging Model** which defines the response logging model.

Table 2: Logging Policy Assertion Table

| **Logging Policy Assertions** | | |
| --- | --- | --- |
| **Specification / Standard / Guideline** | **Notes** | **Required / Recommended** |
| Any processing the results in an error message propogated to a service consumer must log information consistent with the API logical model |  | Required |
| Processing that does not result in an error but effects the behavior of service orchestrations must log information consistent with the API logical model | Examples might include partial data, and exceptions that trigger manual data quality workflow. | Required |
| Processing errors from legacy system and/or partner system components must log errors returned by the components called and be consistent with the API logical model | Legacy system components and partner system components do not comply with API error logging conventions, therefore the API services that integrate these components must log such errors. | Required |
| Faults received by an API service must be either wrapped as a nested APIFault, and/or handled and logged to the error log and be consistent with the API logical model | Refer to the ASG\_API Playbook\_09.01 Exception Handling Section document for a description of the API Fault. | Required |
| Errors in service orchestrations that result in incomplete result data sets must be indicated in the  API Logging Text element. | When a service or operation returns partial data, the status returned to the calling consumer shall flag the partial nature of the returned data. | Required |
| Information such shutdowns due to scheduled maintenance should be logged. |  | Recommended |

## API Audit Level Logging

Logging in general is a process of recording data about events that take place in an application. As such it focuses on things that are of interest at the development environment level.

Auditing Logging focuses on recording business domain-level events: a transaction is created, a user is performing an action, etc. In certain types of application, there is a legal obligation to record such events. The audit log retains events that are persisted to an audit logging infrastructure. Auditing is used to typically answer questions like who did what, why and when.

A system cannot audit without logging, but logging can exist without audit.

The drivers that require an audit log to be created will vary on the business rules of the application. Because an audit log is nothing more that a specialization of a generic log, the data requirements and formatting defined for generic logging should also be applied to audit logging.

The analysis that goes into the determination of business process that flow from business requirements will have to take into consideration and identify what are the auditing requirements.

## Integration with Third-Party Log Aggregators

Application logs can be difficult to deal with, but regardless logging is an important aspect of any production system. When faced with investigating a system issue, it’s much easier to use a log aggregation management solution then it is to parse a jumble of text-files spread throughout the system environment.

A log aggregation management tools will facilitate the pinpointing of the root cause of any application or software error, within a single query. Another factor is having a visual overview of how software is being used globally by the system’s user base, getting all this crucial data in one single component such as a dashboard is going to make the job of monitoring a system’s health easier.

There are many log management features in the market today and the following is but a small subset of these products although Splunk is being integrated in the VA’s cloud infrastructures so a high-level description of the product follows.

### Splunk

Splunk focuses its log management services around enterprise customers who need concise tools for searching, diagnosing and reporting any events surrounding data logs. Splunk’s software is built to support the process of indexing and deciphering logs of any type, whether structured, unstructured, or sophisticated application logs, based on a multi-line approach.

The VA has two Splunk products in the TRM; [Splunk Enterprise](https://www.oit.va.gov/Services/TRM/ToolPage.aspx?tid=6615) and [Splunk Enterprise Security](https://www.oit.va.gov/Services/TRM/ToolPage.aspx?tid=8352). Splunk Enterprise is used to manage logs and provide feedback on function and performance to developers and system owners. The Security product looks for vulnerabilities and is used in risk mitigation by VA Security. Both read the same log files from the same forwarders and indexers. While the ATO for a system may already have Splunk installed and in use, permissions for Splunk Enterprise for use by developers may be required separately. Licensing for Splunk is ‘log sized’ based. Thus developers use of Splunk may trigger larger log storage. It is important to contact the system owner for licensing information.

Key Features:

* Splunk understands machine-data of any type; servers, web servers, networks, exchanges, mainframes, security devices, etc.
* Flexible UI for searching and analyzing data in real-time.
* Drilling algorithm for finding anomalies and familiar patterns across log files.
* Monitoring and alert system for keeping an eye on important events and actions.
* Visual reporting using an automated dashboard output.

## Mule API Logging

Mulesoft provides built-in logging capabilities built on top of Apache Log4j 2.0. Any Mule API project created within Anypoint Studio will have log4j2.xml file available inside /src/main/resources and /src/test/resources for developers to configure and customize. Using standard Apache log4j 2.0 configurations inside those log configuration files, a developer can instruct where to log messages to (file, console, Windows Events, etc). A developer can also instruct what each logged message should contain, such as class name, date/time stamp, line number, etc.

In addition to using the default log4j 2.0, there is an option in Mule 3 to leverage an open-source Mule JSON Logger, developed by Mulesoft Professional Services and published online for any developer to use free of licensing. The details on the logger can be found here:

<https://github.com/mulesoft-consulting/json-logger>

The logger can be integrated with any Mule 3.x project as a third-party dependency. Recommended to first download and install the JSON Logger into a corporate dependency management platform (e.g., Artifactory or private Maven repository) and use it as a dependency inside Mule pom.xml.

However, the JSON Logger is no longer supported in Mule 4.x and will fail to be configured and integrated into Mule 4.x process flows. Instead, to implement a JSON logging solution, it is recommended to create a Dataweave reusable transformation to write log statements in JSON format.

Bottom line, with either solution there is an option of writing log statements from Mule process flows in a form of JSON. This is very useful especially when integrating Mule logs with Splunk or SumoLogic.