Data Lineage SDK Documentation

Data Lineage SDK Documentation

This document provides comprehensive documentation for the Data Lineage SDK. It includes integration instructions, usage examples.

1. Introduction

Data lineage provides insights into how data flows through systems, from its sources to its ultimate use in features and products. It tracks how data is ingested, stored, transformed, and consumed, creating a view of its journey.

Value to users:

- **Increased Visibility:** Data lineage removes the "black box" nature of data processing, showing how data is used and transformed within systems.
- **Improved Understanding**: It helps users understand the dependencies and relationships between different datasets and fields (in ML terminology).
- **Enhanced Troubleshooting**: Data lineage aids in diagnosing data quality issues and identifying the root causes of problems.
- **Better Decision-Making**: By understanding data origins and transformations, users can make more informed decisions based on reliable data.
- **Operational Insights**: It provides insights into feature usage, data dependencies, and system performance.
- **Answering Critical Questions**: It helps answer questions like "which features are used for Matching?" or "how is data transformed from one system to another?"
- **End-to-End Tracking:** It enables tracking of data lineage across various systems, providing a complete picture of the data flow.

Terminology

- **OLE:** Operational Lineage Event, a "data processing receipt" that represents data flow, data usage or data maintenance, for or between data assets.
- **Data asset**: A data asset is an expansion of the concept of a dataset to include both less-structured data sources as well as data query result sets which are presented to users through dashboards, workbooks and query workspaces
- **Processor**: An application, process, ETL, or script that is creating or manipulating data
- **Indeed Resource Name (IRN)**: A canonical identifier used within Indeed to uniquely identify resources across different systems. The primary purpose of an IRN is to provide a stable,

easy-to-construct, and extensible identifier that can be used to address resources without needing to access a database. In the context of Data Lineage, this can identify datasets, processors, schemas. Ref: IRN RFC

2. Setup and Installation

To build or extend the lineage graph, you need to instrument the systems that transform or serve the data you are interested in. Instrumentation can easily be done by integrating the oplin-publisher library into your system and emitting operational lineage events (OLEs).

There are three transport channels available to emit OLEs.

- **Kafka**: This is the recommended transport as it is direct to Oplin processing.
- **Logrepo**: This is available for applications that cannot connect to Kafka. A logrepo processor will read the emitted logentries and forward them to the Kafka transport.
- **HTTP Proxy**: For systems that prefer to talk to HTTP servers, Oplin has set up a proxy that can receive the same data on a REST server and forward OLE data to Kafka.

To get started, add one of the following lines to your build.gradle file.

```
// Include your choice of the available concrete publishing implentations

// kafka-based
compile 'com.indeed:oplin-publisher-kafka'

// logrepo-based
compile 'com.indeed:oplin-publisher-logrepo'

// HTTP-based
compile 'com.indeed:oplin-publisher-http-proxy'
```

Note: Be sure to include only one in your application.

3. Basic Configuration

In java, we typically use Spring Beans for configuration. To use Oplin Data Lineage we will configure the OplinClient and Processor so that we can use them later in instrumentation locations.

3.1 OperationalLineagePublisher

Depending on the publisher you want to use, the configuration can look slightly different.

3.1.1 OperationalLineageAggregatePublisherConfig

The common publisher framework supports two modes of operation.

- **Sampling**: A percentage of the events will actually be published.
- **Aggregation**: All events of the same shape will be combined and counted. Publishing will happen on the next instance of a message after a configured delay.

Property Name	Type	Description
aggregatePublishDelaySec	integer	The delay between publishing unique events, in seconds. For example, if it was set to 60 then a unique event will be recorded and counted for a minute before it publishes the event + count. Default value: 0s
maxAggregatedEventCount	integer	The maximum count to aggregate before publishing regardless of time. For example, if it was set to 100 and the unique count hit 100 before the aggregatePublishDelaySec, it would publish immediately. Default value: 1000
samplingRate	double	A value from 0.0 ⇒ 1.0 representing 0% to 100% of events. Default value: 0.0

For aggregation, you can configure the two related parameters to control how often OLE's are emitted, as described above.

```
@Bean
OperationalLineageAggregatePublisherConfig aggregatePublisherConfig(
    @Value("${data.lineage.publishDelaySec:0}") int publishDelaySec,
    @Value("${data.lineage.maxCount:1000}") double maxAggregatedEventCount
) {
    return OperationalLineageAggregatePublisherConfig.newBuilder()
        .maxAggregatedEventCount(maxAggregatedEventCount)
        .aggregatePublishDelaySec(publishDelaySec)
        .build();
}
```

For sampling, you only need to specify the sampling rate.

```
@Bean
OperationalLineageAggregatePublisherConfig aggregatePublisherConfig(
    @Value("${data.lineage.samplingRate:0.0}") double samplingRate
) {
    return OperationalLineageAggregatePublisherConfig.newBuilder()
        .samplingRate(samplingRate)
        .build();
}
```

3.1.2 Kafka Publisher

ℹ️ The application also need get the permission to write into common-kafka. An example JIRA to request such permission is CD-1372

The kafka-specific configuration that is required depends on the common Properties object found in the Spring. It assumes there exists the common kafka properties below.

Property Name	Type	Description
properties	Properties	This is the producer KafkaProperties. Assuming you are using workloadID, there are multiple ways to configure this, but the most straightforward will be Configure a Java Client for Confluent Cloud | Event Bus Client Library and using the eventBusKafkaProducerProps Bean.

aggregatePublisherConfig	OperationalLineageAggregatePublisherConfig	See 3.1.1.1

Example Java

```java
@Bean
OperationalLineageKafkaPublisherConfig oplinKafkaPublisherConfig(
        @Qualifier("eventbusKafkaProducerProps") Properties props,
        OperationalLineageAggregatePublisherConfig aggregatePublisherConfig
) {
    return OperationalLineageKafkaPublisherConfig.newBuilder(props)
        .aggregatePublisherConfig(aggregatePublisherConfig)
        .build();
}

@Bean
public OperationalLineagePublisher publisher(
    OperationalLineageKafkaPublisherConfig config
) {
    return new OperationalLineageKafkaPublisher(config);
}
```

3.1.3 Logrepo Publisher

The logrepo-specific configuration that is required depends on

Property Name	Type	Description
logEntryFactory	LogEntryFactory	Typically: LogEntryFactory.getDefault()
aggregatePublisherConfig	OperationalLineageAggregatePublisherConfig	See 3.1.1.1

```java
@Bean
LogEntryFactory logEntryFactory() {
    return LogEntryFactory.getDefault();
}

@Bean
OperationalLineageLogRepoPublisherConfig oplinLogRepoPublisherConfig(
    LogEntryFactory factory,
    OperationalLineageAggregatePublisherConfig aggregatePublisherConfig
) {
    return OperationalLineageLogRepoPublisherConfig
        .newBuilder(samplingRate, factory)
        .aggregatePublisherConfig(aggregatePublisherConfig)
        .build();
}
```

3.1.4 HTTP Proxy Publisher

The HTTP-specific configuration that is required depends on ... this one looks weird.

```java
@Bean
RestClient restClient(
    @Value("${data.lineage.proxy.url}") String url,
    @Value("${data.lineage.proxy.username}") String username,
    @Value("${data.lineage.proxy.password}") String password
) {
    return new RestClientImpl(url, username, password);
}

@Bean
OperationalLineageProxyPublisherConfig operationalLineageProxyPublisherConfig(
    RestClient restClient,
    OperationalLineageAggregatePublisherConfig aggregatePublisherConfig
) {
    return OperationalLineageProxyPublisherConfig.newBuilder(restClient)
        .aggregatePublisherConfig(aggregatePublisherConfig)
        .build();
}
```

NOTE: Please reach out to the oplin team for username and password details.

3.2 Processor

The processor defines the application transforming, using or serving the instrumented data. It encompasses enough metadata to uniquely identify where within the Indeed ecosystem this application is running.

Property Name	Type	Description
datacenter	String	Typically: IndeedSystemProperty.DC.value() NOTE: This is the datacenter in which the Processor performs the operation.
environment	String	Typically: IndeedStagingLevel.get().getId()
name	String	The unique name of the processor
id	String	The IRN of the processor. As a convention, we use DaemonClassIrnUtils to generate the IRN for the daemon class consistent

		with backstage IRNs.
type	String	A string describing the kind of processing. This is mostly a note for the implementing team. For example, in MDP we use { Collect, Integrate, Compute, Present } but in other systems, there might be different types.
version	String	A string representing the unique version. In MDP, we have a utility class called CurrentVersionUtils that can determine this in Java applications.

```java
@Bean
public Processor oplinProcessor() {
    return Processor.builder()
        .datacenter(IndeedSystemProperty.DC.value())
        .environment(IndeedStagingLevel.get().getId())
        .name(HortonDaemon.class.getName())
        .id(irnForDaemonClass(HortonDaemon.class))
        .type("MDP Present")
        .version("1.2.3")
        .build();
}
```

3.3 Indeed Resource Namespace (IRNs)

IRNs are an integral part of the data lineage. They are used to uniquely identify parts of the data lineage graph, including the data assets that are used to stitch together a complete lineage picture.

3.3.1 IRNs in MDP

For generating IRNs within MDP, we have several utilities. found below in the Data Lineage SDK Documentation | 5.2.2 IRN Utilities section.

3.3.2 References

Introducing IRNs for Data Assets

Indeed Resource Name for Datasets

https://docs.google.com/document/d/1U0RnAhOZ2rh5c8m9jMsPIU5zH_9AsAkBXx1x9hpC3lk/edit?tab=t.0#heading=h.6ubzbv7m1i8o Connect your Google account

https://docs.google.com/document/d/1o0GPwfUW690_Ka0NGsznJ6Hz_U88uu4H9Xy7S24YpzM/edit?tab=t.0 Connect your Google account

4. Instrumentation

Once you have a properly configured Publisher and Processor, you can begin to think about instrumentation. Instrumentation of data as it moves through a system consists of OLE's being emitted from various processors as the data is transferred and transformed.

A Transform OLE consists of three main components

- The source data asset schemas
- The target data asset schema
- The field mappings in between

4.1 Data Assets

A data asset Schema is, essentially, a list of fields and a unique identifier (its IRN). For fields, the names are mandatory, but type and tag metadata can also be included.

The IRN is used for matching data assets in the lineage graph. Depending on whether an asset is a source or a target will determine where it falls in the graph.

4.1.1 Example

A simple example: we might have a data asset that has two integer fields, say x and y.

```java
try(var ole = new TransformDtoBuilder(publisher, processor, downstream, timestamp)) {
    var irnForSrc = oplinClient.irnFromInMemoryDataAsset(SomeClass.class);
    ole.addSourceDataAsset(irnForSrc);
    ole.addFieldToSource(irnForSrc, "x");
    ole.addFieldToSource(irnForSrc, "y");
    ole.setTargetDataAssetIrn(oplinClient.irnFromInMemoryDataAsset(TargetClass.class));
    ole.addTargetFieldNamed("sum");
    ole.addTargetFieldNamed("product");
}
```

In the above example, we generate the IRN for the single source. In this case, we use the helper method to generate an IRN for an in-memory data asset. Then we use that IRN to register fields with the given names (x and y).

We do something similar for the target data asset. Note the subtle different between them. Since each OLE is for a single target, we **set** the IRN for the target vs **adding** an IRN for a source. The source:target is an N:1 relationship.

4.1.2 Example with oplinClient

For completion, the following example shows how this would be written with the oplinClient, a helper to simplify this kind of code. More details can be found in the linked documentation.

```java
try (final var ole = oplinClient.builderFor()) {
    var irnForSrc = oplinClient.irnFromInMemoryDataAsset(SomeClass.class);
    ole.addSourceDataAsset(irnForSrc);
    ole.addFieldToSource(irnForSrc, "x");
    ole.addFieldToSource(irnForSrc, "y");

    ole.setTargetDataAssetIrn(oplinClient.irnFromInMemoryDataAsset(TargetClass.class));
    ole.addTargetFieldNamed("sum");
    ole.addTargetFieldNamed("product");
}
```

4.2 Field Mappings

Given the source data assets and the target data asset, we can piece together a lineage between those assets as data moves through different processors. However, being able to understand the connection of field data between those data assets provides a much more in-depth understanding of a data lineage graph.

In order to build out such knowledge, the OLE is able to also log the field-level lineage data. This is in the field mappings section, which will contain a set of independent field mappings. Each field mapping can be an N:N level of field mapping.

4.2.1 Complete Example

A simple example: if we have a transform from a source data asset with two integers. We might map them to a target data asset with two fields called sum and product. We would expect to see that both x and y map to both sum and product, but in two different mappings.

```java
try(final var ole = oplinClient.builderFor()) {
    final var irnForSrc = oplinClient.irnFromInMemoryDataAsset(SourceClass.class);
    ole.addSourceDataAsset(irnForSrc);
    ole.addFieldToSource(irnForSrc, "x");
    ole.addFieldToSource(irnForSrc, "y");

    ole.setTargetDataAssetIrn(oplinClient.irnFromInMemoryDataAsset(TargetClass.class));
    ole.addTargetFieldNamed("sum");
    ole.addTargetFieldNamed("product");

    ole.addMapping(irnForSrc, "x", "sum");
    ole.addMapping(irnForSrc, "y", "sum");
    ole.addMapping(irnForSrc, "x", "product");
    ole.addMapping(irnForSrc, "y", "product");
}
```

As with the previous data asset example, we register the two schemas and then we add mappings between them.

Note 1: This example uses the oplinClient method, but could be written with the full TransformDtoBuilder constructor as well.

Note 2: Because source:target is N:1, we need to specify which source the from field comes from but not the target.

4.2.2 Example (simplified)

The OLE builder has some short-cuts built in. Since the schema can be inferred from the mapping, we can remove the addition of fields code and jump straight to mappings, as seen below:

```java
try (final var ole = oplinClient.builderFor()) {
    final var irnForSrc = oplinClient.irnFromInMemoryDataAsset(SourceClass.class);
    ole.addSourceDataAsset(irnForSrc);
    ole.setTargetDataAssetIrn(oplinClient.irnFromInMemoryDataAsset(TargetClass.class));

    ole.addMapping(irnForSrc, "x", "sum");
    ole.addMapping(irnForSrc, "y", "sum");
    ole.addMapping(irnForSrc, "x", "product");
    ole.addMapping(irnForSrc, "y", "product");
}
```

The emitted OLE will be the same as the original example but with less code to maintain.

4.5 OperationalLineageMetadataDtoV2

The OLE metadata class is where the asset and mapping data are contained.

In addition, there are several other fields to be aware of.

Name	Type	Description
eventTime	Instant	The time the event is captured. If using the recommended OplinClient, this will be automatically generated and set to Instant.now()
processor	Processor	The processor, as defined previously. If using the recommended OplinClient, this will be automatically set to the configured Processor.
aggregateCount	int	Useful when the publisher is operating with client-side aggregation enabled.

		This is set by the Publisher, and dependent on the Publisher settings.
downstreamConsumer	String	Useful in server/client situations, for distinguishing which clients are requesting which data. If using the recommended OplinClient, this can be set using the builderFor(<downstream>) variant of constructing a new builder.
tags	Map	A list of tags in the form of key-value pairs that provide additional context and metadata to resources or data points. They will be used for indexing and filtering.
attributes	Map	Attributes are additional information in the form of key-value pairs. This information won’t be used for filtering but to give more context about the execution.

4.5 OperationalLineageTransformDto

The OperationalLineageTransformDto is the OLE holding the data lineage metadata used in building the processed lineage. As such, this documentation will focus on only this OLE.

As of this writing, this OLE has no data specific to the event and only contains the aforementioned fields in its OperationalLineageMetadataDtoV2.

4.6 TransformDtoBuilder

The TransformDtoBuilder is a helper class for building and emitting a Transform OLE.

This is done by creating a new OLE via the TransformDtoBuilder interface, setting the proper fields and then closing the OLE to publish it.

Essentially, you create a builder and you populate it with the source and target datasets, along with the field mappings in between. When the builder is closed, it will use the configured publisher to send the built OLE to Oplin. For convenience, the builder is AutoClosable, meaning we can wrap it in a try-catch block to ensure the publish is called.

An example of how to use this:

```java
try(var ole = new TransformDtoBuilder(publisher, processor, downstream, timestamp)) {
    // source schema
    ole.addSourceDataAsset("irn-for-source");
    ole.addFieldToSource("irn-for-source", "source-field-name");

    // target schema
    ole.setTargetDataAssetIrn("irn-for-target");
    ole.addTargetFieldNamed("target-field-name");

    // field mappings
    ole.addMapping("irn-for-source", "source-field-name", "target-field-name");
}
```

In the above example, you can clearly see that we create

- a source schema with an IRN and a single field.
- a target schema with its own IRN and its own field.
- A mapping between the two fields.

This is a complete OLE with minimal effort. This kind of code can be added to any place a dataset has transformed.

Full documentation of this class can be found in the Java API documentation here.

5. Oplin Publisher Core Utilities

Within the core library, there are some handy utilities that can help in some of the common use-cases identified during development. As more instrumentation is done, more utilities are welcome as experiences are collected.

If there is something you think can or should be added, please reach out in #help-data-lineage to discuss. As of now, the documentation below will describe the current utility suite.

5.1 OplinClient

The OplinClient is a container object for storing the Publisher and Processor. It is intended to make the generation of new TransformDtoBuilder much easier, as we can re-use these variables and even populate the current timestamp.

5.1.1 Configuration

Once we have the publisher and processor configured, we define a new OplinClient:

```java
@Bean
public OplinClient oplinClient(
    OperationalLineagePublisher publisher,
    Processor processor
) {
    return new OplinClient(publisher, processor);
}
```

5.1.2 oplinClient.builderFor()

For convenience, the OplinClient is used to keep track of the publisher and processor and can generate a new TransformDtoBuilder with the current time as its timestamp.

There are two variants of this method, one allowing you to set the downstream client. This is typically used for tracking usage. For example, in a grpc service, the client might send the application name and you could assign that as the downstream.

```java
try(var ole = oplinClient.builderFor(clientName)) {
    // source schema
    ole.addSourceDataAsset("irn-for-source");
    ole.addFieldToSource("irn-for-source", "source-field-name");

    // target schema
    ole.setTargetDataAssetIrn("irn-for-target");
    ole.addTargetFieldNamed("target-field-name");

    // field mappings
    ole.addMapping("irn-for-source", "source-field-name", "target-field-name");

    // set aggregateCount
    ole.lineageBuilder.setAggregateCount(size)
}
```

Full OplinClient API documentation can be found here.

5.2 MDP Lineage Utilities

While we strive to include general utilities into the core library, there are some MDP-specific utilities that may, at some point, be promoted into the common library. This section aims to describe these utilities and why they are here.

5.2.1 CurrentVersionUtils

One of the processor fields is the version, which is a pretty common thing to need. In researching, we found a few different iterations on a common approach so we settled on one and put that approach here.

It is a static call that is built on the common ***current_version.txt*** found in Indeed java applications.

To use:

```java
var version = CurrentVersionUtils.getCurrentRevision();
```

5.2.2 IRN Utilities

- DaemonClassIrnUtils builds IRNs for daemons, useful in processor ids.
- GraphqlIrnUtils builds IRNs related to GraphQL schemas, used in our collection API.

Because these IRNs are not standardized (yet), we keep these as MDP-specific implementations. If/when they become available, we will migrate these to the latest version.

https://code.corp.indeed.com/company-data/mdp-lineage-utils

5.2.3 Full javadoc Documentation

We have incorporated building javadoc from the library and the docs are available here:

https://company-data.pages.corp.indeed.com/mdp-lineage-utils/javadoc/

If there’s any question or suggestion, feel free to reach out to #help-data-lineage and we can explain and update the docs as needed.

6. Error Handling

Errors in sending data are caught and suppressed, eliminating the need to worry about unintentional errors. While we strive to not interfere or affect instrumented systems, mistakes can happen. If there's any disruption to service, please don't hesitate to reach out to the #help-data-lineage channel and raise the issue with the team.

7. Java Instrumentation Agent

See wiki here:

Page: Data Lineage Proto Tracing Agent

See README in the agent repo here:

https://code.corp.indeed.com/oplin/proto-trace-agent

8. FAQ

8.1. My component has 1 input and creates N outputs, do I emit 1 lineage event or N?

The events are target-based, so in this case you would create N events.

NOTE: A single target can have multiple sources. The relationship is N:1 between sources and targets.

8.2. My component has 1 input and fans out N of the same outputs, do I emit 1 lineage event or N?

The lineage event has an aggregation count that, when using client-side aggregation, will automatically accumulate and send only 1 event (bucketed by time and count).

Alternatively, if you know the count already, rather than emitting multiple known duplicate events, this aggregation count will be exposed in the API and can be set directly (see CD-1590).

8.3. My component is an API, should a failed lineage call fail the API call?

No. The lineage is meant to be best-effort and if there's any issue with the lineage system, it should not affect the instrumented system.

8.4. Can we pre-emptively send a lineage request?

Yes. The idea of lineage is to collect how data is transformed through the system. Theoretically, this could be emitted from code analysis tools which never depend on actual successful transformations, so the question of *when* an OLE is sent is less important than the contents of the OLE itself.

Another aspect to consider is lineage can also be used to track usage, so if a downstream consumer is requesting a field, even if that request were to fail, it's important to know that the client exists. In this case, it makes sense to emit the lineage even in service failure scenarios.

8.5 Where's the javadocs?

For oplin-specific code:

Core Libraries

Kafka Publisher

Log Repo Publisher

HTTP Proxy Publisher

For mdp-specific utilities:

MDP Lineage Utilities Library

9. Currently Instrumented Systems

- mdp-compute (python using logrepo) - niffler/mdp-compute-connector
- generic-online-syncer (java kafka) - niffler/mdp-generic-online-syncer
- horton (java kafka) - squall/horton
- Match Providers (java agent using http proxy)

10. tl;dr Just Give Me The Code

10.1 DataLineageConfig.java (Kafka)

```java
@Bean
OperationalLineageKafkaPublisherConfig oplinKafkaPublisherConfig(
        @Qualifier("eventbusKafkaProducerProps") Properties props,
        OperationalLineageAggregatePublisherConfig aggPublisherConfig
) {
    return OperationalLineageKafkaPublisherConfig.newBuilder(props)
        .aggregatePublisherConfig(aggPublisherConfig)
        .build();
}

@Bean
public OperationalLineagePublisher publisher(
    OperationalLineageKafkaPublisherConfig config
) {
    return new OperationalLineageKafkaPublisher(config);
}

@Bean
public Processor oplinProcessor() {
    return Processor.builder()
        .datacenter(IndeedSystemProperty.DC.value())
        .environment(IndeedStagingLevel.get().getId())
        .name(MyProcessor.class.getName())
        .id(irnForDaemonClass(MyProcessor.class))
        .type("MDP Present")
        .version(CurrentVersionUtils.getCurrentRevision())
        .build();
}

@Bean
public OplinClient oplinClient(
    OperationalLineagePublisher publisher,
    Processor processor
) {
    return new OplinClient(publisher, processor);
}
```

10.2 Instrumentation in code

```java
try (final var ole = oplinClient.builderFor()) {
    var irnSrc = oplinClient.irnFromInMemoryDataAsset(SourceClass.class);
    ole.addSourceDataAsset(irnSrc);
    ole.setTargetDataAssetIrn(
        oplinClient.irnFromInMemoryDataAsset(TargetClass.class)
    );

    ole.addMapping(irnSrc, "x", "sum");
    ole.addMapping(irnSrc, "y", "sum");
    ole.addMapping(irnSrc, "x", "product");
    ole.addMapping(irnSrc, "y", "product");
}
```

11. I Still Need Help.

If there are still questions after reading through this document, feel free to reach out to help dedicated help slack channel: #help-data-lineage