Observability 200: Logging with OpenTelemetry

The advent of OpenTelemetry has forever changed how we capture observability signals. While OTel initially focused on delivering traces and metrics, support for collection of logs is now stable and gaining adoption, particularly in Kubernetes environments.

In this lab, we will explore several models for using OpenTelemetry to collect and parse logs.

Getting Our Bearings

In this lab, we will be working with an exemplary stock trading system, comprised of several services and their dependencies, all instrumented using OpenTelemetry.

Our trading system is composed of: * `proxy`: a nginx reverse proxy which proxies requests from the outside into the trading system * `trader`: a python application that trades stocks on orders from customers * `router`: a node.js application that routes committed trade records * `recorder-java`: a Java application that records trades to a PostgreSQL database

Finally, we have `monkey`, a python application we use for testing our system that makes periodic, automated trade requests on behalf of fictional customers.

Elasticsearch

We will be working with a live Elasticsearch instance, displayed in the browser tab to the left. We are currently looking at Elastic's dynamically generated Service Map. It shows all of the services that comprise our system, and how they interact with one another.

> [!NOTE] You are welcome to explore each service and our APM solution by clicking on each service icon in the Service Map and selecting `Service Details`

When you are ready, let's quick pivot to Discover where we can easily query and parse our logs. To do so, click `Discover` in the left-hand navigation pane

This workshop will leverage ES|QL, Elastic's query-time language, to query our logs. Please refer to the following diagram to familiarize yourself with the ES|QL interface in Discover. You can enter your queries in the pane at the top of the Elasticsearch Discover. You can change the time window of your search using the Time Filter. To execute a search, click the Play/Refresh icon.

When you are ready, click the `Next` button to continue.

In this model, we will be sending logs directly from a service to an OpenTelemetry Collector over the network using the OTLP protocol. This is the default mechanism most OpenTelemetry SDKs use for exporting logs from a service.

service

code

log(message)

span

Logging Framework

log API

OTel SDK

message

traceid,baggage

Appender: otlp

span / otlp

log / otlp

OpenTelemetry Collector

Receiver: otlp

Processor: enrichment

Processor: transform

Exporter: Elasticsearch

log / http

Elasticsearch
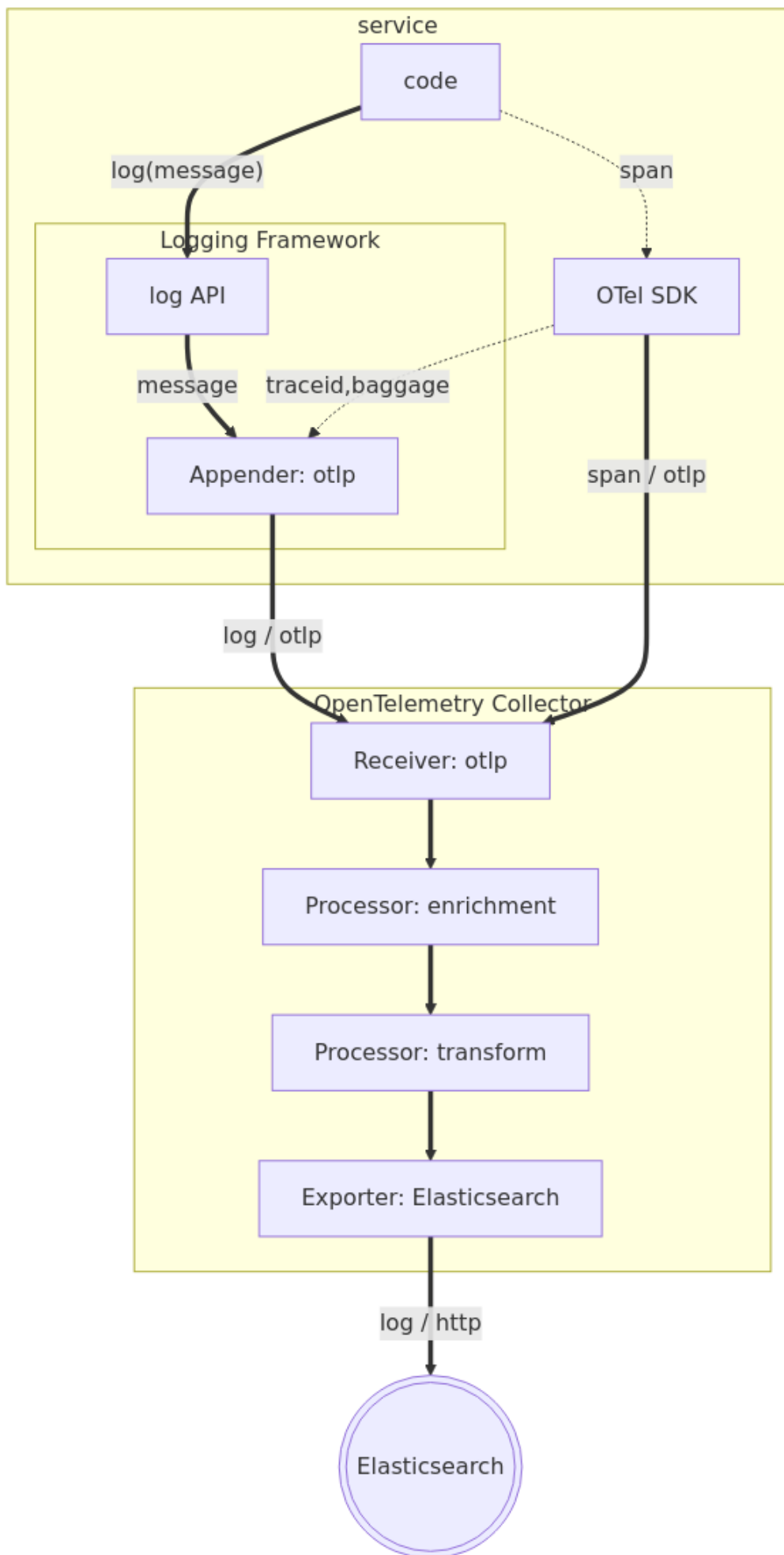
Looking at the diagram: 1) a service leverages an existing logging framework (e.g., logback in Java) to generate log statements 2) on service startup, the OTel SDK injects a new Appender module into the logging framework. This module formats the log metadata to appropriate OTel semantic conventions (e.g., log.level), adds appropriate contextual metadata (e.g., trace.id), and outputs the log lines via OTLP (typically buffered) to a configured OTel Collector 3) an OTel Collector (typically, but not necessarily) on the same node as the service receives the log lines via the `otlp` receiver 4) the Collector enriches the log line with additional metadata and optionally parses or otherwise transforms the message 5) the Collector then outputs the logs downstream (either directly to Elasticsearch, or more typically through a gateway Collector, and then to Elasticsearch)

Assumptions

While this model is relatively simple to implement, it assumes several things: 1) The service can be instrumented with Open-Telemetry (either through runtime zero-configuration instrumentation, or through explicit instrumentation). This essentially rules out use of this method for most opaque, third-party applications and services. 2) Your OTel pipelines are robust enough to forgo file-based logging. Traditional logging relied on services writing to files and agents reading or "tailing" those log files. File-based logging inherently adds a semi-reliable, FIFO, disk-based queue between services and the Collector. If there is a downstream failure in the telemetry pipeline (e.g., a failure in the Collector or downstream of the Collector) or back-pressure from Elasticsearch, the file will serve as a temporary, reasonably robust buffer. Notably, this concern can be mitigated with Collector-based disk queues and/or the use of a Kafka-like queue somewhere in-between the first Collector and Elasticsearch.

Advantages

There are, of course, many advantages to using OTLP as a logging protocol where possible: 1) you don't have to deal with file rotation or disk overflow due to logs 2) there is less io overhead (no file operations) on the node 3) the Collector need not be local to the node running the applications (though you would typically want a Collector per node for other reasons)

Additionally, exporting logs from a service using the OTel SDK offers the following general benefits: 1) logs are automatically formatted with OTel Semantic Conventions 2) key/values applied to log statements are automatically emitted as attributes 3) traceid and spanid are automatically added when appropriate 4) contextual metadata (e.g., service.name) are automatically emitted as attributes 5) custom metadata in baggage can be automatically applied as attributes to each log line

All of the above leads to logs with rich context and metadata with little to no additional work. It is worth noting that the Collector now supports a disk-based buffer between the receivers and the exporters.

Configuration

Most of the languages supported by OpenTelemetry are automatically instrumented for logging via OTLP by default. In the case of Java, for example, the OTel SDK, when in zero-code instrumentation, will automatically attach an OTLP appender to either Logback or Log4j.

In this example, we are leveraging the OpenTelemetry Operator for Kubernetes to automatically inject the OTel SDK into our services, including the `recorder-java` service. Our `recorder-java` service is using Logback as a logging framework with a slf4j facade.

Let's first validate we have logs coming in from our `recorder-java` service:

1. Open the button label="Elasticsearch" tab
2. Execute the following query:

```
FROM logs-*
| WHERE service.name == "recorder-java"
```

3. Open the first log record by clicking on the double arrow icon under `Actions`
4. Click on the `Attributes` tab

You note that the `log.file.path` attribute is empty, in this case indicating this log line was delivered via OTLP without having been written to a log file.

Checking the Source

Let's have a look at the configuration of our `recorder-java` service.

1. Open the button label="recorder-java Source" tab

2. Navigate to `src/main/resources/logback.xml`
3. Note that no appenders are specified in the logback configuration (they are automatically injected by the OTel SDK on startup)

Let's further validate that no logs are being written to stdout (which would be picked up and dumped to a log file by Kubernetes):

1. Open the button label="Terminal" tab
2. Execute the following to get a list of the active Kubernetes pods that comprise our trading system:

```
kubectl -n trading get pods
```

3. Find the active `recorder-java-...` pod in the list
4. Get stdout logs from the active `recorder-java` pod:

```
kubectl -n trading logs <recorder-java-...>
```

(replace ... with the pod instance id)

Note that there are no logs being written to stdout from `recorder-java` because we have not configured any appenders in the logback configuration.

This confirms that logs coming from the `recorder-java` application to our OTel Collector via OTLP, and not by way of a log file.

> [!NOTE] It is possible to leave a console appender in your logback configuration such that you can still view the logs locally (with `kubectl logs` or by tailing the log file itself). In this case, you would want to be sure you are explicitly excluding this log file from also being scrapped by your OTel Collector to avoid duplicative log input into Elasticsearch. We will show a straightforward way of doing this in a future challenge.

Correlation

One major advantage of using OTLP for logging is the ability to very easily append the active `trace.id` and `span.id` if the log is emitted during an active APM span.

1. Open the button label="Elasticsearch" tab
2. Click `Applications > Service Inventory` in the left-hand navigation pane
3. Click on the `recorder-java` service
4. Click on the `Transactions` tab
5. Click on the `POST /record` tab
6. Scroll down to `Trace sample`
7. Go to the second (2) `Trace sample`
8. Click on the `Logs` tab

These are all the logs associated with this particular trace sample.

Attributes via Structured Logging

Let's say that we think we might have a problem with the Garbage Collector in our Java Virtual Machine (JVM) running too often, possibly affecting database performance. As a developer, you might think to sample the amount of time spent in GC and then report that in a log file.

Say we wanted to graph GC time by region to see if perhaps the issue is localized. To do that, we need GC time as a metric value. While we could just encode it into the log message as text and parse it out, that's unneccessary with modern structured logging APIs and OpenTelemetry.

OTLP logging allows us to easily add attributes to our log lines by using key/value mechanisms present in your existing logging API. In this case, we can use the `addKeyValue()` API exposed by our logging facade, slf4j.

1. Open the button label="recorder-java Source" tab
2. Navigate to `src/main/java/com/example/recorder/TradeRecorder.java`
3. Find the following line:

```
log.atInfo().log("trade committed for " + trade.customerId);
```

and change it to:

```
log.atInfo().addKeyValue(Main.ATTRIBUTE_PREFIX + ".gc_time", utilities.getGarbageCollectorDeltaTime()).log("trade committed for " + trade.cus
```

Now let's recompile and redeploy our `recorder-java` service. 1. Open the button label="Terminal" tab 2. Execute the following:

```
./builddeploy.sh -s recorder-java
```

Now let's see what our logs look like in Elasticsearch. 1. Open the button label="Elasticsearch" tab 2. Click `Discover` in the left-hand navigation pane 3. Execute the following query:

```
FROM logs-*
| WHERE service.name == "recorder-java" and message LIKE "*trade committed*"
```

4. Open the first log record by clicking on the double arrow icon under `Actions`
5. Click on the `Attributes` tab

Note the added attribute `attributes.com.example.gc_time`!

Now let's graph `gc_time` to answer our question.

Execute the following query:

```
FROM logs-*
| WHERE service.name == "recorder-java"
| STATS count = MAX(attributes.com.example.gc_time)  BY attributes.com.example.region, BUCKET(@timestamp, 1 minute)
```

Indeed, it looks like only the `recorder-java` service deployed to the `NA` region is exhibiting this problem.

Attributes via Baggage

Note that the log record has other custom attributes like `attributes.com.example.customer_id`. We didn't add that in our logging statement in `recorder-java`. How did it get there?

1. Open the button label="Elasticsearch" tab
2. Click `Applications > Service Inventory` in the left-hand navigation pane
3. Click on the `Service Map` tab
4. Click on the `trader` service
5. Click on `Service Details`
6. Click on the `Transactions` tab
7. Scroll down and click on the `POST /trade/request` transaction under `Transactions`
8. Scroll down to the waterfall graph under `Trace sample`
9. Click on the first span `POST /trade/request` to open the flyout

Note that `attributes.com.example.customer_id` exists in this span too!

1. Close the `Transaction details` flyout
2. Click on the `Logs` tab under `Trace sample`
3. Click on the log line that looks like `trade committed for <customer.id>`

Note that `attributes.com.example.customer_id` exists here too!

This is a great example of the power of using OpenTelemetry Baggage. Baggage lets us inject attributes early on in our distributed service mesh and then automatically distribute and apply them downstream to every span and log message emitted in context!

Imagine how easy this will make the life of your SREs and analysts to easily search across all of your observability signals using the inputs they are accustomed to: namely, `customer_id`, for example.

Let's look at the code which initially stuck `customer_id` into OTel baggage:

1. Open the button label="Trader Source" tab
2. Navigate to `app.py`
3. Look for calls to `set_attribute_and_baggage()` inside the `decode_common_args()` function

Here, we are pushing attributes into OTel Baggage. OTel is propagating that baggage with every call to a distributed surface. The baggage follows the context of a given span through all dependent services. Within a given service, we can leverage BaggageProcessor extensions to automatically apply metadata in baggage as attributes to the active span (including logs).
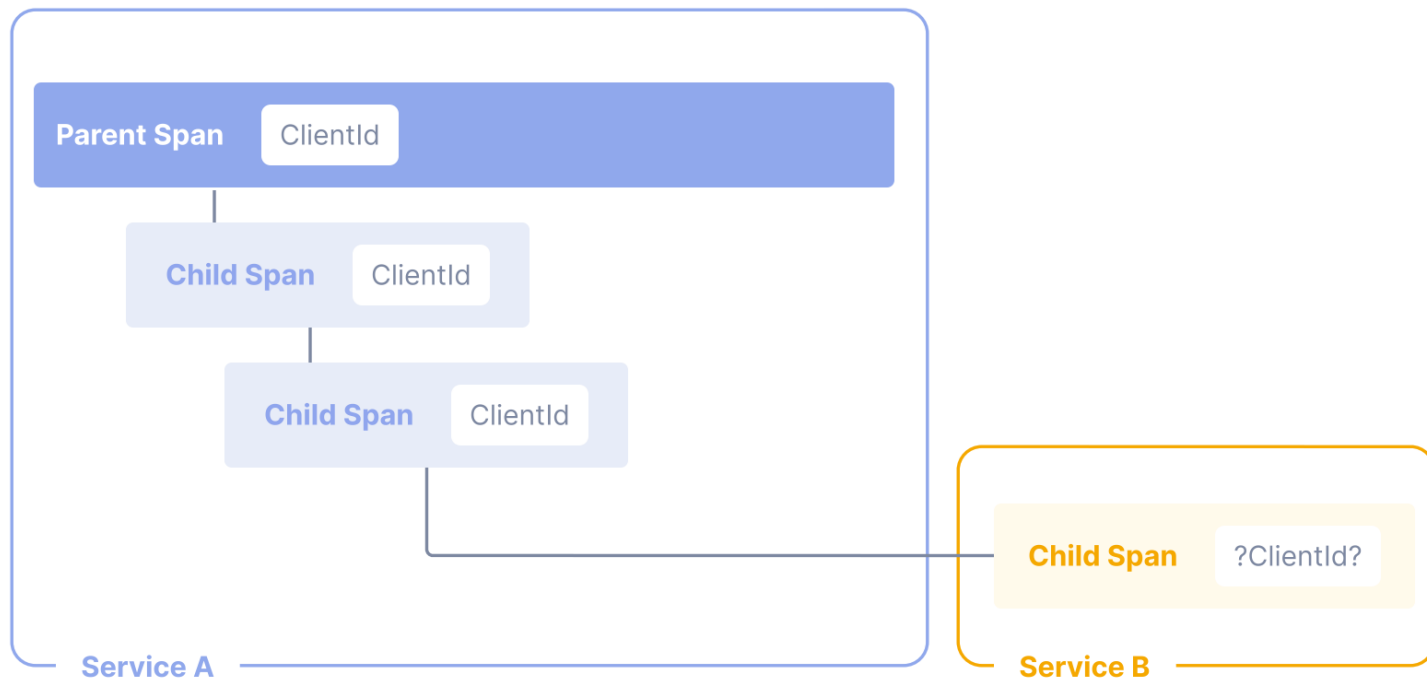
Let's add an additional attribute in our trader service.

1. Find the following line in the `decode_common_args()` function:

```
subscription = params.get('subscription', None)
```

2. Add the following to push `subscription` into baggage:

```python
if subscription is not None:
    set_attribute_and_baggage(f"{ATTRIBUTE_PREFIX}.subscription", subscription)
```

Now let's recompile and redeploy our `trader` service. 1. Open the button label="Terminal" tab 2. Execute the following:

```
./builddeploy.sh -s trader
```

And now let's check our work in Elasticsearch:

1. Open the button label="Elasticsearch" tab
2. Click `Applications > Service Inventory` in the left-hand navigation pane
3. Click on the `Service Map` tab
4. Click on the `trader` service
5. Click on `Service Details`
6. Click on the `Transactions` tab
7. Scroll down and click on the `POST /trade/request` transaction under `Transactions`
8. Scroll down to the waterfall graph under `Trace sample`
9. Click on the `Logs` tab
10. Click on the `trade committed for <customer_id>` log line emitted by the `recorder-java` service
11. Note the presence of the `subscription` attribute!

There are many reasons why use of OTLP-based logging may be impractical. Chief among them is accommodating services which cannot be instrumented with OpenTelemetry (e.g., third-party services). These services simply write their logs to disk directly, or more commonly to stdout, which is then written to disk by the Kubernetes or Docker logging provider, for example.

To accommodate such services, we can use the filelog receiver in the OTel Collector. In many regards, the `filelog` receiver is the OTel equivalent of Elastic's filebeat (often running as a module inside Elastic Agent).

```mermaid
flowchart TD
    service[service]
    service -->|log / text| File[File]
    File -->|log / text| Receiver

    subgraph OpenTelemetry Collector
        Receiver[Receiver: filelog]
        Receiver --> enrichment[Processor: enrichment]
        enrichment --> transform[Processor: transform]
        transform --> exporter[Exporter: Elasticsearch]
    end

    exporter -->|log / http| Elasticsearch((Elasticsearch))
```

service

log / text

File

log / text

OpenTelemetry Collector

Receiver: filelog

Processor: enrichment

Processor: transform

Exporter: Elasticsearch

log / http

Elasticsearch

Getting our bearings

In this example, we will be working with a service which outputs logs to stdout in a custom JSON format.

Let's first examine the raw JSON logs as they are currently being received by Elasticsearch: 1. Open the button label="Elasticsearch" tab 2. Execute the following query:

```
FROM logs-*
| WHERE service.name == "router"
```

3. Open the first log record by clicking on the double arrow icon under `Actions`
4. Click on the `Log overview` tab

Note that the body of the message is not particularly useable: * it has a "burned-in" JSON format * it contains both the message ("0") and associated metadata ("_meta") * the log level as presented by Elasticsearch will always be `INFO` regardless of the actual log level

Checking the Source

Now let's validate that these logs are indeed being emitted to stdout and written to disk:

1. Open the button label="Terminal" tab
2. Execute the following to get a list of the active Kubernetes pods that comprise our trading system:

```
kubectl -n trading get pods
```

3. Find the active `router-...` pod in the list
4. Get stdout logs from the active `router` pod:

```
kubectl -n trading logs <router-...>
```

(replace ... with the pod instance id)

Note that the logs are being written to stdout and are being captured by the Kubernetes logging provider. Let's validate that Kubernetes is writing this log stream to disk:

1. Open the button label="Terminal" tab
2. Get all log files associated with pods

```
cd /var/log/pods/
ls
```

3. Get logs for current instant of the `router` pod

```
cd trading_router*
ls
cd router
ls
```

4. Look at the router container logs

```
cat 0.log
```

Yup! Clearly these logs are being written to disk.

Parsing JSON logs

Many custom applications log to a JSON format to provide some structure to the log line. To fully appreciate this benefit in a logging backend, however, you need to parse that JSON (embedded in the log line) and extract fields of interest.

While you could do this with Elasticsearch using Streams (as we will see in the future challenge), with OpenTelemetry, this can also be done at the edge in the Collector using OTTL.

OTTL Playground

Crafting OTTL in a vacuum is tricky: the feedback loop of crafting OTTL, deploying it to the collector, validating it has the correct syntax, and validating it does what you expect can be long and painful.

Fortunately, there is a better way! Elastic has made available the OTTL Playground: a tool to interactively refine your OTTL before putting it in production.

1. Open the button label="OTTL Playground" tab
2. Paste into the `OTLP Payload` pane an example from our JSON formatted `router` logs:

```
{
  "resourceLogs": [
    {
      "resource": {},
      "scopeLogs": [
        {
          "scope": {},
          "logRecords": [
            {
              "timeUnixNano": "1544712660300000000",
              "observedTimeUnixNano": "1544712660300000000",
              "severityNumber": 10,
              "severityText": "Information",
              "traceId": "5b8efff798038103d269b633813fc60c",
              "spanId": "eee19b7ec3c1b174",
              "body": {
                "stringValue": "{\"0\": \"routing request to http://recorder-java:9003\", \"_meta\": {  \"runtime\": \"Nodejs\",      \"run
              }
            }
          ]
        }
      ]
    }
  ]
}
```

3. Paste into the `Configuration` pane the following:

```
log_statements:
  - context: log
    conditions:
      - body != nil and Substring(body, 0, 2) == "{\""
    statements:
      - set(cache, ParseJSON(body))
      - flatten(cache, "")
      - merge_maps(attributes, cache, "upsert")
```

Those initial set of log statements: 1. check if the message body is JSON formatted 2. if so, parses the body as json, flattens the key names (to prevent nesting), and merges all extracted keys to `attributes`

Click on the `Run >` button. In the `Result` pane, you can see the diff of what this OTTL would do, and it *kind of* matches what we expect.

It is far from ideal: * it does not conform to OTel semantic conventions (e.g., `_meta.logLevelName`, `_meta.date`) * the message body is stored as an attribute with key `0`

Let's clean that up with OTTL!

1. Paste the following into the `Configuration` pane:

```
log_statements:
  - context: log
    conditions:
      - body != nil and Substring(body, 0, 2) == "{\""
    statements:
      - set(cache, ParseJSON(body))
      - flatten(cache, "")
      - merge_maps(attributes, cache, "upsert")

      - set(time, Time(attributes["_meta.date"], "%Y-%m-%dT%H:%M:%SZ"))
      - set(severity_text, attributes["_meta.logLevelName"])
      - set(severity_number, attributes["_meta.logLevelId"])
      - delete_matching_keys(attributes, "_meta\\..*")

      - set(body, attributes["0"])
      - delete_key(attributes, "0")
```

2. Click on the `Run >` button

Ah, that looks much better! Here, we are: * converting the date from a string to an epoch timestamp and copying it into the proper semantic convention (semcon) field * copy the log level into the proper semcon fields * deleting the remaining `_meta.*` fields * copying the body from `attributes.0` to the proper semcon field * deleting the now defunct `attributes.0`

This looks great. Let's put this configuration into production!

Putting It Into Production

1. Open the button label="Collector Config" tab
2. Find the following block under `collectors/daemon/config/processors`:

```
transform/parse_json_body:
    error_mode: ignore
    # WORKSHOP CONTENT GOES HERE
```

3. Replace it with the OTTL we developed above:

```
transform/parse_json_body:
    error_mode: ignore
    log_statements:
      - context: log
        conditions:
          - body != nil and Substring(body, 0, 2) == "{\""
        statements:
          - set(cache, ParseJSON(body))
          - flatten(cache, "")
          - merge_maps(attributes, cache, "upsert")

          - set(time, Time(attributes["_meta.date"], "%Y-%m-%dT%H:%M:%SZ"))
          - set(severity_text, attributes["_meta.logLevelName"])
          - set(severity_number, attributes["_meta.logLevelId"])
          - delete_matching_keys(attributes, "_meta\\..*")

          - set(body, attributes["0"])
          - delete_key(attributes, "0")
```

Now let's redeploy the OTel Operator with our updated config:

1. Open the button label="Terminal" tab
2. Execute the following:

```
helm upgrade --install opentelemetry-kube-stack open-telemetry/opentelemetry-kube-stack --force \
  --namespace opentelemetry-operator-system \
  --values 'collector/values.yaml' \
  --version '0.9.1'
```

This will redeploy the OTelOperator, which in turn will restart the daemonset Collectors with their new config. We can check when the new configuration has taken affect by looking at the status of the daemonset Collectors.

1. Open the button label="Terminal" tab
2. Execute the following:

```
kubectl -n opentelemetry-operator-system get pods
```

When you see that the replacement daemonset Collectors have been up for at least 30 seconds, let's check the logs coming into Elastic:

1. Open the button label="Elasticsearch" tab
2. Click `Discover` in the left-hand navigation pane
3. Execute the following query:

```
FROM logs-*
| WHERE service.name == "router"
```

4. Open the first log record by clicking on the double arrow icon under `Actions`
5. Click on the `Log overview` tab

Yes! Note that cleanly parsed JSON logs: * the message body is now just the message * the log level and timestamp are set correctly

Handling Structured Logging

Let's take advantage of the support for structured logging in the tslog package.

1. Open the button label="Router Source" tab
2. Navigate to `app.ts`
3. Find the line in the function `customRouter()`

```
logger.info(`routing request to ${host}`);
```

4. Modify it to read:

```
logger.info(`routing request to ${host}`, {method: method});
```

This will add the routing method selected to the log line in a structured manner.

Now let's recompile and redeploy our `router` service: 1. Open the button label="Terminal" tab 2. Execute the following:

```
./builddeploy.sh -s router
```

Now let's see how this looks in Elasticsearch: 1. Open the button label="Elasticsearch" tab 2. Click `Discover` in the left-hand navigation pane 3. Execute the following query:

```
FROM logs-*
| WHERE service.name == "router"
```

4. Open the first log record by clicking on the double arrow icon under `Actions`
5. Click on the `Attributes` tab

Ok. We see `1.method`, but that's ugly. Let's fix it with OTTL!

1. Open the button label="OTTL Playground" tab
2. Paste into the `OTLP Payload` pane an example from our JSON formatted `router` logs with the new `method` key:

```
{
  "resourceLogs": [
    {
      "resource": {},
      "scopeLogs": [
        {
          "scope": {},
          "logRecords": [
            {
              "timeUnixNano": "1544712660300000000",
              "observedTimeUnixNano": "1544712660300000000",
              "severityNumber": 10,
              "severityText": "Information",
              "traceId": "5b8efff798038103d269b633813fc60c",
              "spanId": "eee19b7ec3c1b174",
              "body": {
                "stringValue": "{\"0\": \"routing request to http://recorder-java:9003\",  \"1.method\": \"random\",  \"_meta\": {     \"runt:
              }
            }
          ]
        }
      ]
    }
  ]
}
```

3. Press Run. Note the `1.method` attribute as expected.
4. Paste the following the `Configuration` pane:

```
log_statements:
  - context: log
    conditions:
      - body != nil and Substring(body, 0, 2) == "{\""
    statements:
      - set(cache, ParseJSON(body))
      - flatten(cache, "")
      - merge_maps(attributes, cache, "upsert")

      - set(time, Time(attributes["_meta.date"], "%Y-%m-%dT%H:%M:%SZ"))
      - set(severity_text, attributes["_meta.logLevelName"])
      - set(severity_number, attributes["_meta.logLevelId"])
      - delete_matching_keys(attributes, "_meta\\..*")

      - set(body, attributes["0"])
      - delete_key(attributes, "0")

      - replace_all_patterns(attributes, "key", "\\d+\\.", "")
```

5. Press Run

Much better. That last line removes the numeric prefix from any attributes and seems to work as expected.

Let's put it into production, the same as before: 1. Open the button label="Collector Config" tab 2. Replace your existing block under `collectors/daemon/config/processors`:

```
        transform/parse_json_body:
            error_mode: ignore
            ...
```

3. Replace with the OTTL we developed above:

```
        transform/parse_json_body:
            error_mode: ignore
            log_statements:
              - context: log
                conditions:
                  - body != nil and Substring(body, 0, 2) == "{\""
                statements:
                  - set(cache, ParseJSON(body))
                  - flatten(cache, "")
                  - merge_maps(attributes, cache, "upsert")

                  - set(time, Time(attributes["_meta.date"], "%Y-%m-%dT%H:%M:%SZ"))
                  - set(severity_text, attributes["_meta.logLevelName"])
                  - set(severity_number, attributes["_meta.logLevelId"])
                  - delete_matching_keys(attributes, "_meta\\..*")

                  - set(body, attributes["0"])
                  - delete_key(attributes, "0")

                  - replace_all_patterns(attributes, "key", "\\d+\\.", "")
```

Now let's redeploy the OTel Operator with our updated config:

1. Open the button label="Terminal" tab
2. Execute the following:

```
helm upgrade --install opentelemetry-kube-stack open-telemetry/opentelemetry-kube-stack --force \
  --namespace opentelemetry-operator-system \
  --values 'collector/values.yaml' \
  --version '0.9.1'
```

This will redeploy the OTelOperator, which in turn will restart the daemonset Collectors with their new config. We can check when the new configuration has taken affect by looking at the status of the daemonset Collectors.

1. Open the button label="Terminal" tab
2. Execute the following:

```
kubectl -n opentelemetry-operator-system get pods
```

When you see that the replacement daemonset Collectors have been up for at least 30 seconds, let's check the logs coming into Elastic:

1. Open the button label="Elasticsearch" tab
2. Click Discover in the left-hand navigation pane
3. Execute the following query:

```
FROM logs-*
| WHERE service.name == "router"
```
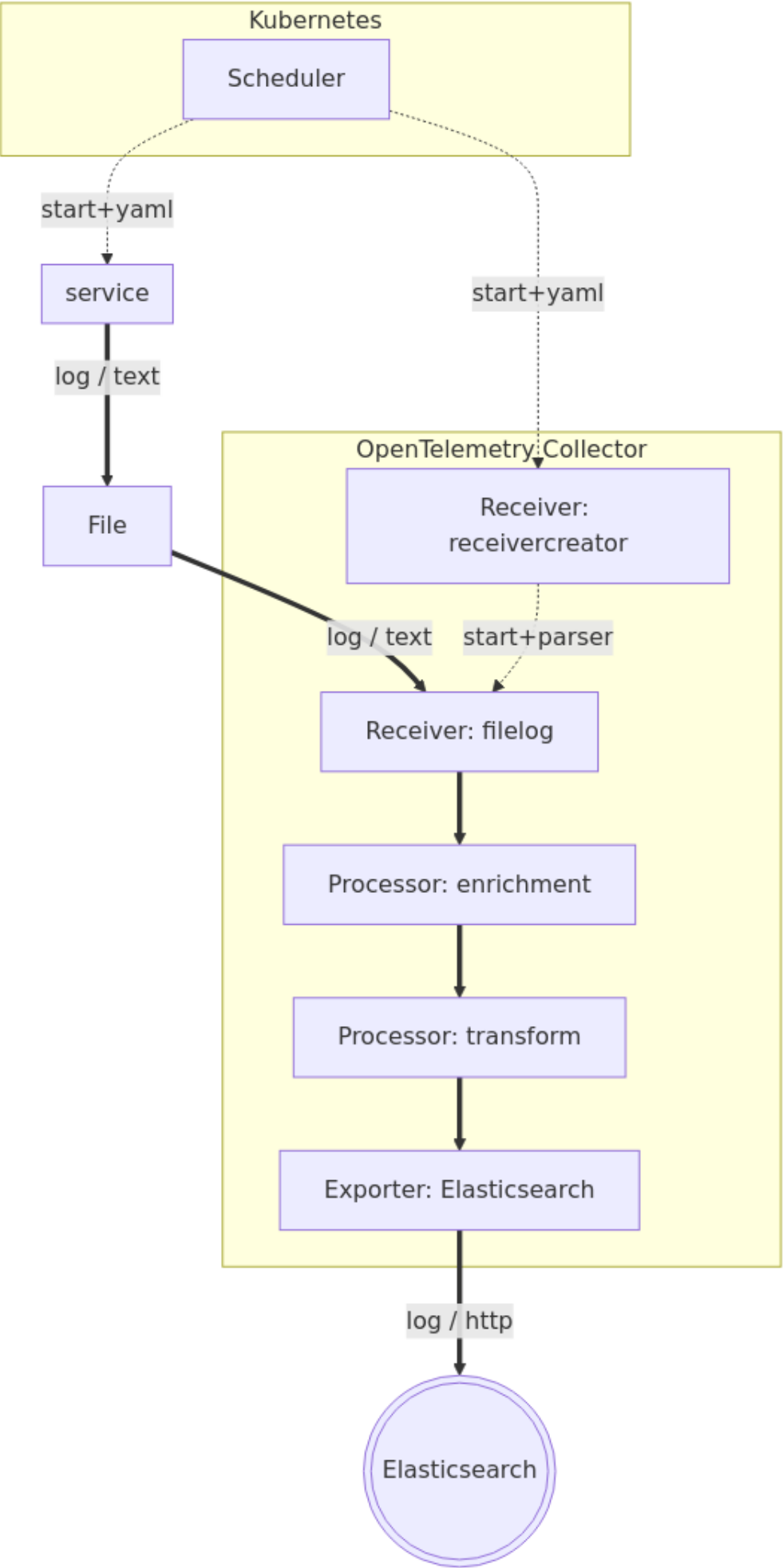
4. Open the first log record by clicking on the double arrow icon under Actions
5. Click on the Log overview tab

Nice and clean JSON logs in Elastic: perfect.

Well almost. You'll note that we had to modify the configuration of the Collector in the daemonset; the same Collector which handles logs from all of our services. Imagine we have multiple services, each which outputs a unique JSON schema. In that case, we would have to introduce routing in our Collector pipelines in order to selectively apply the right OTTL for a given log source... What if there was a way that the services themselves could specify their configuration?

Modifying the Collector config to parse specific logs feels awkward. Ideally, we could push bespoke parsing configurations to the deployment of the app or service itself. Realistically, it is the service which is in the best position to know the nuances of its custom logging pattern.

Fortunately, on Kubernetes, just such an option exists: the Receiver Creator can be used to dynamically instantiate `file` receivers with a custom configuration driven by the deployment yaml of each service.

Kubernetes

Scheduler

start+yaml

service

log / text

File

start+yaml

OpenTelemetry Collector

Receiver:
receivercreator

log / text

start+parser

Receiver: filelog

Processor: enrichment

Processor: transform

Exporter: Elasticsearch

log / http

Elasticsearch

Let's have a look at our postgresql logs.

1. Open the button label="Elasticsearch" tab
2. Click `Discover` in the left-hand navigation pane
3. Execute the following query:

```
FROM logs-*
| WHERE service.name == "postgresql"
```

4. Open the first log record by clicking on the double arrow icon under `Actions`
5. Click on the `Log overview` tab

Note the unstructured nature of these logs. One kind of neat aspect is the present of `traceparent` on some of the logs. Recall that these logs are generated from postgresql directly. Postgresql at present is not OpenTelemetry enabled, and thus has no native provisions for accepting a `traceparent` via distributed tracing. How did this line get there?

SQL Commentor. SQL comments is an extension (available for at least the Java language) which can append `traceparent` as a comment to SQL queries. Most databases (including postgresql) will output the comment as part of the audit log.

Let's parse these logs using the Receiver Creator.

Let's have a look at the Receiver Creator config 1. Open the button label="Collector Config" tab

Now we need to modify our `postgresql.yaml` to include our directives.

1. Open the button label="postgresql Config" tab
2. Find the following lines under `spec/template/metadata/annotations`:

```
    annotations:
      io.opentelemetry.discovery.logs/enabled: "true"
      # WORKSHOP CONTENT GOES HERE
```

3. Replace it with the following:

```
    annotations:
      io.opentelemetry.discovery.logs/enabled: "true"
      io.opentelemetry.discovery.logs/config: |
        operators:
        - type: container
        - type: regex_parser
          on_error: send_quiet
          parse_from: body
          regex: '^(?P<timestamp_field>\d{4}-\d{2}-\d{2}\s\d{2}:\d{2}:\d{2}.\d{3}\s[A-z]+)\s\[\d+\]\s(?P<severity_field>[A-Z]+):\s*(?<msg_
          timestamp:
            parse_from: attributes.timestamp_field
            on_error: send_quiet
            layout_type: strptime
            layout: '%Y-%m-%d %H:%M:%S.%L %Z'
          trace:
            trace_id:
              parse_from: attributes.trace_id
              on_error: send_quiet
            span_id:
              parse_from: attributes.span_id
              on_error: send_quiet
            trace_flags:
              parse_from: attributes.trace_flags
              on_error: send_quiet
          severity:
            parse_from: attributes.severity_field
            on_error: send_quiet
            mapping:
              warn:
                - WARNING
                - NOTICE
              error:
                - ERROR
              info:
                - LOG
                - INFO
                - STATEMENT
              debug1:
                - DEBUG1
              debug2:
                - DEBUG2
              debug3:
                - DEBUG3
              debug4:
                - DEBUG4
```

```
        debug5:
          - DEBUG5
        fatal:
          - FATAL
          - PANIC
      - type: move
        on_error: send_quiet
        from: attributes.msg_field
        to: body
      - type: remove
        on_error: send_quiet
        field: attributes.timestamp_field
      - type: remove
        on_error: send_quiet
        field: attributes.severity_field
      - type: remove
        on_error: send_quiet
        field: attributes.trace_version
      - type: remove
        on_error: send_quiet
        field: attributes.trace_id
      - type: remove
        on_error: send_quiet
        field: attributes.span_id
      - type: remove
        on_error: send_quiet
        field: attributes.trace_flags
```

This blog applies a regex.

> [!NOTE] You note that Receiver Creator unfortunately uses a different language than OTTL. Notably, Receiver Creator does its transforms within the receiver itself, unlike OTTL which is instrumented using a Processor.

Now apply it: 1. Open the button label="Terminal" tab 2. Execute the following:

```
./deploy.sh -s postgresql
./deploy.sh -s recorder-java
```

Check Elasticsearch: 1. Open the button label="Elasticsearch" tab 2. Click `Discover` in the left-hand navigation pane 3. Execute the following query:

```
FROM logs-* WHERE service.name == "postgresql"
```

4. Open the first log record by clicking on the double arrow icon under `Actions`
5. Click on the `Log overview` tab

Check logs:

1. Click `Applications` > `Service Inventory` in the left-hand navigation pane
2. Click on the `Service Map` tab
3. Click on the `trader` service
4. Click on `Service Details`
5. Click on the `Transactions` tab
6. Scroll down and click on the `POST /trade/request` transaction under `Transactions`
7. Scroll down to the waterfall graph under `Trace sample`
8. Click on the `Logs` tab
9. Click on the `execute <unnamed>: ...` log line emitted by the `postgresql` service
10. Click on the `Table` tab
11. Search for the attribute `trace.id`

Note how with SQL Commentor, OpenTelemetry, and Elastic, we can correlate our postgresql audit logs with our traces!

In many cases, you might be ok to parse your logs on-demand. As an example, i can use ES|QL to parse my nginx proxy logs as needed.

ES|QL

Let's first try query-time parsing using ES|QL:

1. Open the button label="Elasticsearch" tab
2. Execute the following query:

```
FROM logs-proxy.otel-default
| GROK body.text "%{IPORHOST:client_ip} %{USER:ident} %{USER:auth} \\[%{HTTPDATE:timestamp}\\] \"%{WORD:http_method} %{NOTSPACE:request_path}
| WHERE status_code IS NOT NULL
| EVAL @timestamp = DATE_PARSE("dd/MMM/yyyy:HH:mm:ss Z", timestamp) // use embedded timestamp as record timestamp
| KEEP @timestamp, client_ip, http_method, request_path, status_code, user_agent
```

So far, we've been using ES|QL to parse our proxy logs at query-time. While incredibly powerful for quick analysis, we can do even more with our logs if we parse them at ingest-time.

While you could use our OOTB integration to parse nginx logs, we've customized our nginx logs with a duration. Streams is really good at working with this.

Parsing with Streams

We will be working with Elastic Streams which makes it easy to setup log parsing pipelines.

1. Select `logs-proxy.otel-default` from the list of data streams (if you start typing, Elasticsearch will help you find it)
2. Select the `Processing` tab

Parsing the log message

We can parse our nginx log messages at ingest-time using the Elastic Grok processor.

1. Click `Add a processor`
2. Select the `Grok` Processor (if not already selected)
3. Set the `Field` to

`body.text`

4. Click `Generate pattern`. Elasticsearch will analyze your log lines and determine a suitable grok pattern.
5. To ensure a consistent lab experience, copy the following grok expression and paste it into the `Grok patterns` field (*do not* click the `Accept` button next to the generated pattern)

`%{IPV4:client.ip} - %{NOTSPACE:client.user} \[%{HTTPDATE:timestamp}\] "%{WORD:http.request.method} %{URIPATH:http.request.url.path} HTTP/%{NU`

6. Wait until the sample `body.text` on the right shows highlighting, then click `Add processor`

Parsing the timestamp

The nginx log line includes a timestamp; let's use that as our record timestamp.

1. Click `Add a processor`
2. Select `Date`
3. Set `Field` to `timestamp`
4. Elastic should auto-recognize the format: `dd/MMM/yyyy:HH:mm:ss XX`
5. Click `Add processor`

Now save the Processing by clicking `Save changes` in the bottom-right.

A faster way to query

Now let's jump back to Discover by clicking `Discover` in the left-hand navigation pane.

Execute the following query:

```
FROM logs-proxy.otel-default
| WHERE http.response.status_code IS NOT NULL
| KEEP @timestamp, client.ip, http.request.method, http.request.url.path, http.response.status_code, user_agent.original
```

In this lab, we presented several different ways of capturing and parsing logs using OpenTelemetry.

In reality, you will likely use a combination of these techniques, depending on your system:

- you might use OTLP for greenfield applications

- you might use Receiver Creator for k8s applications

- you might use file receiver for VMs
- you might rely on ES|QL parsing for on-demand parsing
- for permanent parsing, you might use edge parsing for well-known log formats (postgresql), but rely on Streams for more flexible formats (custom apps)