Observability 100: Using Elastic and OpenTelemetry to Observe Kubernetes

Installing OpenTelemetry in Kubernetes

We have our application stack running on Kubernetes. Now let's observe it using Elastic!

Install the OpenTelemetry Operator

With the advent of the OpenTelemetry Operator and related Helm chart, you can now easily deploy an entire observability signal collection package for Kubernetes, inclusive of: * application traces, metrics, and logs * infrastructure traces (nginx), metrics and logs * application and infrastructure metrics

1. button label="Elastic"
2. Click `Add Data` in bottom-left pane
3. Click `Kubernetes`
4. Click `OpenTelemetry (Full Observability)`
5. Click `Copy to clipboard` below `Add the OpenTelemetry repository to Helm`
6. button label="Terminal"
7. Paste and run helm chart command
8. button label="Elastic"
9. Click `Copy to clipboard` below `Install the OpenTelemetry Operator`
10. button label="Terminal"
11. Paste and run k8s commands to install OTel operator

If you get an error saying that the current version of the `values.yaml` file is unavailable, run this command in the terminal:

```
helm upgrade --install opentelemetry-kube-stack open-telemetry/opentelemetry-kube-stack   --namespace opentelemetry-operator-system   --value
```

Checking the Install

First, list out the available namespaces:

```
kubectl get namespaces
```

And get a list of pods running in the `opentelemetry-operator-system` namespace:

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

And let's look at the logs from the daemonset collector to see if it is exporting to Elasticsearch without error...

Checking Observability

Let's confirm what signals are coming into Elastic.

First, let's check for logs. Navigate to the button label="Elastic" tab and click on `Observability` > `Discover`.

Next, let's check for infrastructure metrics. Navigate to the button label="Elastic" tab and click on `Infrastructure` > `Hosts`.

Finally, let's check for application traces. Navigate to the button label="Elastic" tab and click on `Applications` > `Service Inventory`. Note that there is not yet any APM data flowing in.

Let's figure out what's up.

Debugging OpenTelemetry in Kubernetes

Debugging APM

Let's describe one of our application pods... Specifically, the monkey pod:

```
kubectl -n trading-1 describe pod monkey
```

Look at the environment variables section; note that there are not yet any OTel environment variables loaded into the pod.

It turns out that after you apply the OTel Operator to your Kubernetes cluster, you need to restart all of your application services:

```
kubectl -n trading-1 rollout restart deployment
```

Now let's wait for our monkey pod to restart:

```
kubectl -n trading-1 get pods
```

And once it has restarted, let's describe it again:

```
kubectl -n trading-1 describe pod monkey
```

And now we can see OTel ENV vars being injected into the monkey pod. Let's check if we have APM data flowing in. Navigate to the button label="Elastic" tab and click on `Applications` > `Service Inventory`. Ok cool, this is starting to look good.

Why is router not showing up?

Click on the `trader` app and look at the `POST /trade/request` transaction. Scroll down to the bottom (trace samples) and look at the waterfall graph. Notice the broken trace. It looks like perhaps one of our applications is not being instrumented. Click on the `POST` span and look at `attributes.service.target.name`. Note that this `POST` is intended to target the `router` service, yet we don't see the `router` service in our Service Map.

Let's look at our `router` pod and see if we can figure out what's up.

```
kubectl -n trading-1 describe pod router
```

Huh. no OTel ENVs, even though the pod was restarted. Let's have a look at that deployment yaml. Click on the button label="Code" button and examine the deployment yaml. Look at the deployment yaml for the `trader` service and compare it to the `router` service. Notice anything missing?

In order for the Operator to attach the correct APM agent, you need to apply an appropriate annotation to each pod. Note that the router pod is missing an annotation. Let's add it. Navigate to the button label="Source" tab and open `router.yaml`.

Uncomment the `instrumentation.opentelemetry.io/inject-nodejs` directive:

```
spec:
  template:
    metadata:
      annotations:
        instrumentation.opentelemetry.io/inject-nodejs: "opentelemetry-operator-system/elastic-instrumentation"
      ...
```

And then reapply the yaml:

```
./build.sh -d force -s router
```

Note that `router` was reconfigured:

```
deployment.apps/router configured
```

Wait for the `router` pod to restart:

```
kubectl -n trading-1 get pods
```

And once it has restarted, let's describe it again:

```
kubectl -n trading-1 describe pod router
```

And now it looks like our OTel ENVs are getting injected as expected. Let's check Elasticsearch.

Navigate to the button label="Elastic" tab and click on `Applications` > `Service Inventory`. Note that we can now see a full distributed trace, as expected!

Automatic Instrumentation

Automatic Instrumentation

Let's say you aren't using the Kubernetes OTel Operator, or Kubernetes at all. How do you attach the OTel SDK to your applications?

Java Attach

Navigate to the button label="Source" tab and open `src/recorder-java/Dockerfile`.

OTel Variables

How did the agent know where to send spans? We have instructed the Kubernetes OTel Operator inject these variables automatically. We could have also defined them manually.

Navigate to the button label="Source" tab and open `k8s/yaml/recorder-java.yaml`.

Inferred Spans

Let's add automatic inferred spans:

Navigate to the button label="Source" tab and open `src/recorder-java/Dockerfile`.

Modify the Java startup to look like this:

```
EXPOSE 9003
ENTRYPOINT ["java", \
"-javaagent:edot-javaagent.jar", \
"-Dotel.inferred.spans.enabled=true", "-Dotel.inferred.spans.sampling.interval=1ms", "-Dotel.inferred.spans.min.duration=0ms", "-Dotel.infer
"-jar", "/usr/src/app/recorder.jar"]
```

And rebuild and deploy the `recorder-java` service:

```
./build.sh -d force -b true -s recorder-java
```

Adding Annotations

Instrumentation through Annotations

For languages that support annotations (e.g., Python and Java), the OTel SDK lets you instrument with annotations.

Navigate to the button label="Source" tab and open `src/recorder-java/pom.xml`.

Navigate to the button label="Source" tab and open `src/recorder-java/src/main/java/com/example/recorder/TradeService.ja`

Adding annotations

Modify `src/recorder-java/src/main/java/com/example/recorder/TradeService.java` like

```
    @WithSpan
    public void auditCustomer(@SpanAttribute(Main.ATTRIBUTE_PREFIX + "trade") String customerId) {
        log.info("trading for " + customerId);
    }
```

And rebuild and deploy the `recorder-java` service:

```
./build.sh -d force -b true -s recorder-java
```

Adding Manual Instrumentation

Manual Instrumentation

Navigate to the button label="Source" tab and open `src/recorder-java/pom.xml`.

Navigate to the button label="Source" tab and open `src/recorder-java/src/main/java/com/example/recorder/Main.java`.
Navigate to the button label="Source" tab and open `src/recorder-java/src/main/java/com/example/recorder/TradeService.ja`

Adding instrumentation

Modify `src/recorder-java/src/main/java/com/example/recorder/TradeService.java` like

```
    public void auditSymbol(String symbol) {
        Span span = tracer.spanBuilder("auditSymbol").startSpan();
        try (Scope ignored = span.makeCurrent()) {
            span.setAttribute(Main.ATTRIBUTE_PREFIX + "symbol", symbol);
             log.info("trading symbol" + symbol);
        } finally {
            span.end();
        }
    }
```

And rebuild and deploy the `recorder-java` service:

```
./build.sh -d force -b true -s recorder-java
```

Review

Review
