Building Resilient Microservices with Istio and Red Hat OpenShift Service Mesh

Tell us what you think!

Congratulations on completing at least 50% of the course. We would appreciate feedback on your learning experience. Click below to fill out a brief survey.

TAKE ME TO THE SURVEY (HTTPS://SURVEY.OLE.REDHAT.COM/SURVEY?COURSE=DO328&COURSE_VER=2.0&MODALITY=COURSE&OFFERING=328&UTOKEN=CHJHYNUUBXBVBM51C2FTEQ%3D%3D)

NOT RIGHT NOW

☐ Don't ask again

(/rol/app/courses/do328-2.0/pages/ch03) Previous (/rol/app/cou Next

VIDEO - Building Resilient Microservices with Istio and Red Hat OpenShift Ser ... | Guided Exercise: Tracing Services with Jaeger

Play on TV

Guided Exercise: Tracing Services with Jaeger

- Deploy two microservices on OpenShift Service Mesh, and trace the service communication using Jaeger.
- `servicea`: written in JavaScript using the Node.js runtime.
- `serviceb`: written in Java using the Quarkus framework.

Traffic enters the service mesh through `servicea`. `servicea` calls `serviceb` and returns a response.

You will do the following in this exercise:

1. Enable distributed tracing in both microservices using Jaeger.
2. Build container images locally for both microservices using `podman`.
3. Push the built container images to the `Quay.io` public container registry.
4. Deploy both microservices to the service mesh, and trace the service calls between the two microservices.

Outcomes

You should be able to deploy applications to OpenShift Service Mesh, and trace the path of the service calls for requests entering the service mesh.

To perform this exercise, ensure you have access to:

- A configured and running OpenShift cluster.
- An installed and running OpenShift Service Mesh in the OpenShift cluster.
- The OpenShift CLI (`/usr/local/bin/oc`).
- An account with the `Quay.io` container registry, and `podman` installed locally on your workstation.

As the `student` user on the `workstation` machine, use the `lab` command to validate the prerequisites for this exercise:

```
[student@workstation ~]$ lab observe-jaeger start
```

Procedure 3.1. Instructions

1. Clone the source code for the microservices from GitHub. Inspect the source code for both microservices.

   Use a text editor like `VSCodium`, which supports syntax highlighting for editing JavaScript and Java source files.

   1.1. Open a new terminal window on your workstation. From the home directory, clone the source code for the microservices from GitHub.

   ```
   [student@workstation ~]$ git clone  https://github.com/RedHatTraining/DO328-apps
   ...output omitted...
   Cloning into 'DO328-apps'...
   ...output omitted...
   ```

   1.2. Copy the contents of the `/home/student/DO328-apps/tracing-ge` folder from your local Git repository to the `/home/student/DO328/labs/observe-jaeger` folder.

   ```
   [student@workstation ~]$ cp -Rv ~/DO328-apps/tracing-ge/* \
   ~/DO328/labs/observe-jaeger/
   ```

   You should now see two folders called `servicea` and `serviceb` in the `/home/student/DO328/labs/observe-jaeger/` folder.

   You should also see shell scripts and service mesh related YAML files in the same folder. These were created by the lab start script.

   1.3. Review the source code for `servicea` in the `/home/student/DO328/labs/observe-jaeger/servicea/index.js` file.

   This microservice exposes a single HTTP GET endpoint, which calls `serviceb` and returns a response to the client.

1

```
...output omitted...
server.get("/", async (request) => {
    const { rootSpan } = request;
    const msg = await serviceb.callServiceB(rootSpan);

    return 'Hello from ServiceA!.\nResponse from ServiceB => ' + msg + '\n';
});
...output omitted...
```

1.4. Review the source code for `serviceb` in the `/home/student/DO328/labs/observe-jaeger/serviceb/src/main/java/com/redhat/training/serviceb/ServiceB.java` file.

This microservice contains a single HTTP GET endpoint that returns a string.

```
...output omitted...
String message = "Hello from ServiceB!";

@GET
@Produces(MediaType.TEXT_PLAIN)
public String sayHello() {
  return message;
}
...output omitted...
```

2. Enable tracing for `servicea` using the `Jaeger` and `OpenTracing` libraries.

2.1. Navigate to the `/home/student/DO328/labs/observe-jaeger/servicea` folder on the command line terminal. All references to file paths in this step are relative to this folder.

```
[student@workstation ~]$ cd ~/DO328/labs/observe-jaeger/servicea
```

2.2. Inspect the `package.json` file, which declares all the NPM packages required for running this microservice.

Install the packages declared in `package.json`, followed by installing the `opentracing` and `jaeger-client` dependencies.

```
[student@workstation servicea]$ npm install
[student@workstation servicea]$ npm install --save \
 jaeger-client@3.17.2 opentracing@0.14.4
...output omitted...
+ opentracing@0.14.4
+ jaeger-client@3.17.2
...output omitted...
```

2.3. Edit the `index.js` file.

Add the default value for the `TRACE_COLLECTOR_URL` variable. This should be the URL of the Jaeger collector that is running in the service mesh.

```
const TRACE_COLLECTOR_URL = ...output omitted... || "http://jaeger-collector.istio-system.svc:14268/api/traces";
```

Note how the collector URL is used to initialize the tracer.

```
const tracer = Tracer.create("servicea", TRACE_COLLECTOR_URL, logger);
```

The service name `servicea` is also passed to the `Tracer.create()` method, which indicates the starting point of the trace.

2.4. Edit the `Tracer.js` file, which configures the tracer properties for this microservice.

Add the parameters for the `config` and `reporter` properties in the `create` function as follows:

```
  const config = {
    serviceName: serviceName,
    sampler: {
      type: "const",
      param: 1,
    },
    reporter: {
      logSpans: true,
      collectorEndpoint
    }
  };
```

You can also copy the code from the solution file `/home/student/DO328/solutions/observe-jaeger/servicea/Tracer.js`.

Note the use of HTTP headers to propagate the trace context using the `tracer.registerInjector()` and `tracer.registerExtractor()` methods.

```
tracer.registerInjector(FORMAT_HTTP_HEADERS, codec);                          1
tracer.registerExtractor(FORMAT_HTTP_HEADERS, codec);
```

2.5. Edit the `HttpServer.js` file. Note the callback methods, `traceRequest` and `traceResponse` registered as hooks to the server. These methods are called after every HTTP request to the microservice, and before sending the response to the client respectively.

Edit the `traceRequest` method and add code to create a new root span with a unique string id. Add Opentracing tags to the span to identify the original URL of the request and the HTTP method (GET, POST, PUT, DELETE and more).

```
const span = tracer.startSpan(`${method}:servicea`);
span.setTag(Opentracing.Tags.HTTP_URL, originalUrl);
span.setTag(Opentracing.Tags.HTTP_METHOD, method);
```

You can also copy the code from the solution file `/home/student/DO328/solutions/observe-jaeger/servicea/HttpServer.js`.

2.6. Briefly review the `Services/ServiceB.js` file. Do not make any changes to this file. The `get()` method is invoked on every request to the root URL of the microservice ("/").

The implementation is inherited from the parent `RestClient` class. Edit the `Services/RestClient.js` file.

Create a new span for the `get()` method, which is a child span of the root span. Add Tags to indicate which endpoint is being called, as well as the caller URL, the HTTP method, and the type of span.

```
async get(url, rootspan) {
  ...output omitted...
  const span = this.tracer.startSpan(spanName, { childOf: rootSpan.context() });
  span.setTag(Tags.PEER_HOSTNAME, this.baseURL);
  span.setTag(Tags.HTTP_URL, url);
  span.setTag(Tags.HTTP_METHOD, "GET");
  span.setTag(Tags.SPAN_KIND, Tags.SPAN_KIND_RPC_CLIENT);
  ...output omitted...
}
```

Inject the span data as HTTP headers to propagate the span context. Add the following code to the `_buildAxiosRequestConfig()` method.

```
_buildAxiosRequestConfig(span) {
  const headers = {};
  this.tracer.inject(span, FORMAT_HTTP_HEADERS, headers);
  return { headers };
}
```

You can also copy the code from the solution file `/home/student/DO328/solutions/observe-jaeger/servicea/Services/RestClient.js`.

2.7. Save your changes. To ensure that there are no syntax errors, run `npm start` and ensure that the microservice starts without any errors.

If there are errors, then compare your changes with the solution files in the `/home/student/DO328/solutions/observe-jaeger/servicea` folder.

```
[student@workstation servicea]$ npm start

 servicea@1.0.0 start ...output omitted...
 node index.js

...output omitted...: "Initializing Jaeger Tracer ...output omitted...
...output omitted...: "Server listening at http://0.0.0.0:8080 "}
```

Press **Ctrl**+**C** to stop the server.

3. Enable tracing for `serviceb` using the `quarkus-smallrye-opentracing` library.

3.1. Change to the `/home/student/DO328/labs/observe-jaeger/serviceb` folder. All references to file paths in this step are relative to this folder.

```
[student@workstation ~]$ cd ~/DO328/labs/observe-jaeger/serviceb
```

3.2. Inspect the maven `pom.xml` file, which declares the dependencies for this microservice.

Include the `quarkus-smallrye-opentracing` dependency for enabling tracing.

```
<dependency>
  <groupId>io.quarkus</groupId>
  <artifactId>quarkus-smallrye-opentracing</artifactId>
</dependency>
```

You can also copy the code from the solution file `/home/student/DO328/solutions/observe-jaeger/serviceb/pom.xml`.

3.3. Briefly review the code in the `src/main/java/com/redhat/training/serviceb/ServiceB.java` file. Do not make any changes to this file.

Tracing is automatically enabled by adding the `quarkus-smallrye-opentracing` maven dependency, and then enabling some jaeger related environment variables in the Quarkus `application.properties` file. 1

3.4. Edit the `src/main/resources/application.properties` file and add the jaeger related properties.

```
quarkus.jaeger.service-name=serviceb
quarkus.jaeger.sampler-type=const
quarkus.jaeger.sampler-param=1
quarkus.log.console.format=%d{HH:mm:ss} %-5p traceId=%X{traceId}, spanId=%X{spanId}, sampled=%X{sampled} [%c{2.}] (%t) %s%e%
n
quarkus.jaeger.endpoint=http://jaeger-collector.istio-system.svc:14268/api/traces
quarkus.jaeger.propagation=b3
quarkus.jaeger.reporter-log-spans=true
```

You can also copy the code from the solution file `/home/student/DO328/solutions/observe-jaeger/serviceb/src/main/resources/application.properties`.

3.5. Save your changes. To ensure that there are no syntax errors, run `mvn clean package` and ensure that a fat JAR is created in the `target` folder.

```
[student@workstation serviceb]$ mvn clean package
...output omitted...
[INFO] Building serviceb 1.0.0
...output omitted...
...output omitted... Building fat jar: observe-jaeger/serviceb/target/serviceb-1.0.0-runner.jar
[INFO] [io.quarkus.deployment.QuarkusAugmentor] Quarkus augmentation completed in 2440ms
...output omitted...
[INFO] BUILD SUCCESS
...output omitted...
```

3.6. Start the microservice and verify that there are no errors.

```
[student@workstation serviceb]$ java -jar target/serviceb-1.0.0-runner.jar
...output omitted... (powered by Quarkus 1.3.2.Final) started in 0.747s.
Listening on: http://0.0.0.0:8080
...output omitted...
```

Verify that there are no errors. If there are errors, then compare your changes with the solution files in the `/home/student/DO328/solutions/observe-jaeger/serviceb` folder. Press **Ctrl**+**C** to stop the server.

4. Build container images for both microservices using `podman`.

4.1. Load your classroom environment configuration.

Run the following command to load the environment variables:

```
[student@workstation serviceb]$ source /usr/local/etc/ocp4.config
```

4.2. Review the `Dockerfile` for `servicea`. Use Red Hat Universal Base Images (UBI) as the base for building your container image. Do not make any changes to the Dockerfile.

4.3. Build the container image for `servicea` using `podman`.

```
[student@workstation serviceb]$ cd ~/DO328/labs/observe-jaeger/servicea
[student@workstation servicea]$ podman build -t \
 quay.io/${RHT_OCP4_QUAY_USER}/ossm-tracing-servicea:1.0 .
STEP 1: FROM registry.access.redhat.com/ubi8/nodejs-12
...output omitted...
STEP 13: COMMIT quay.io/youruser/ossm-tracing-servicea:1.0
```

4.4. Similarly, build the container for `serviceb` after reviewing the `Dockerfile`.

```
[student@workstation servicea]$ cd ~/DO328/labs/observe-jaeger/serviceb
[student@workstation serviceb]$ podman build -t \
 quay.io/${RHT_OCP4_QUAY_USER}/ossm-tracing-serviceb:1.0 .
STEP 1: FROM registry.access.redhat.com/ubi8:8.1
...output omitted...
STEP 15: COMMIT quay.io/youruser/ossm-tracing-serviceb:1.0
```

4.5. Verify that the container images for `servicea` and `serviceb` are built successfully.

```
[student@workstation serverb]$ podman images
REPOSITORY                                  TAG ...output omitted...
quay.io/youruser/ossm-tracing-serviceb      1.0 ...output omitted...
quay.io/youruser/ossm-tracing-servicea      1.0 ...output omitted...
...output omitted...
```

5. Create new public image repositories in Quay.io to store the newly built container images. Push the container images to `Quay.io`.

5.1. Create two new public container image repositories called `ossm-tracing-servicea`, and `ossm-tracing-serviceb` in Quay.io. Refer to the instructions in the section called " Creating a Quay Account " (/rol/app/courses/do328-2.0/pages/pr01s04) for creating public container image repositories. 1

> **WARNING**
>
> If you skip this step and push the container images without creating public repositories, then the `podman push` commands create private container image repositories by default.

5.2. Login to your Quay.io account using `podman`.

```
[student@workstation serverb]$ podman login -u ${RHT_OCP4_QUAY_USER} quay.io
```

You will be prompted for your Quay.io password.

5.3. Push the container image for `servicea` to Quay.io.

```
[student@workstation serverb]$ podman push \
 quay.io/${RHT_OCP4_QUAY_USER}/ossm-tracing-servicea:1.0
...output omitted...
Writing manifest to image destination
Storing signatures
```

5.4. Push the container image for `serviceb` to Quay.io.

```
[student@workstation serverb]$ podman push \
 quay.io/${RHT_OCP4_QUAY_USER}/ossm-tracing-serviceb:1.0
...output omitted...
Writing manifest to image destination
Storing signatures
```

6. Create the `tracing` project and then add it to the `ServiceMeshMemberRoll` resource.

6.1. Log in to OpenShift as the `developer` user.

```
[student@workstation serverb]$ oc login -u ${RHT_OCP4_DEV_USER} \
 -p ${RHT_OCP4_DEV_PASSWORD} ${RHT_OCP4_API}
Login successful.
...output omitted...
```

6.2. Create the `tracing` project.

```
[student@workstation serverb]$ oc new-project tracing
Now using project "tracing" on server
"https://api.cluster.domain.example.com:6443".
...output omitted...
```

6.3. Add the `tracing` project to the list of members in the `ServiceMeshMemberRoll` resource. Edit the `default` ServiceMeshMemberRoll resource in the OpenShift web console and add the `tracing` project to the member list.

```
apiVersion: maistra.io/v1
kind: ServiceMeshMemberRoll
metadata:
...output omitted...
spec:
  members:
  - tracing
...output omitted...
```

You can also run the `add-project-to-smmr.sh` script in the `/home/student/DO328/labs/observe-jaeger` folder to add the `metrics` project to the list of members in the `ServiceMeshMemberRoll` resource.

```
[student@workstation serverb]$ cd /home/student/DO328/labs/observe-jaeger
[student@workstation observe-jaeger]$ oc patch servicemeshmemberroll/default \
-n istio-system --type=merge \
-p '{"spec": {"members": ["tracing"]}}'
servicemeshmemberroll.maistra.io/default patched
```

> **NOTE**
>
> You can also use the `oc edit smmr default -n istio-system` command and add the `tracing` project to the member list.

7. Deploy the microservices to OpenShift service mesh.

7.1. Edit the `*-deploy.yaml` files in the `/home/student/DO328/labs/observe-jaeger` folder, which describes the necessary resources to deploy both applications.

The deployment files have the `sidecar.istio.io/inject: "true"` annotation included to inject the Envoy proxy after deployment. 1

7.2. Edit the `/home/student/DO328/labs/observe-jaeger/servicea-deploy.yaml` file. Edit the `spec.template.spec.containers.image` attribute and add the Quay.io URL of the newly built container image.

```
...output omitted...
spec:
  containers:
  - name: servicea
    image: quay.io/youruser/ossm-tracing-servicea:1.0
    imagePullPolicy: IfNotPresent
...output omitted...
```

7.3. Edit the `/home/student/DO328/labs/observe-jaeger/serviceb-deploy.yaml` file. Add the Quay.io URL of the container image for `serverb`.

```
...output omitted...
spec:
  containers:
  - name: serviceb
    image: quay.io/youruser/ossm-tracing-serviceb:1.0
    imagePullPolicy: IfNotPresent
...output omitted...
```

7.4. Run the `oc create` command to deploy the applications.

```
[student@workstation observe-jaeger]$ oc create -f servicea-deploy.yaml
deployment.apps/servicea created
service/servicea created
[student@workstation observe-jaeger]$ oc create -f serviceb-deploy.yaml
deployment.apps/serviceb created
service/serviceb created
```

7.5. Run the `oc get pods` command and verify that both microservices are deployed and in `Running` state.

```
[student@workstation observe-jaeger]$ oc get pods
NAME                        READY   STATUS    RESTARTS   AGE
servicea-6c9fffcc58-v699r   2/2     Running   0          10m
serviceb-78489f94bf-z4vdh   2/2     Running   0          10m
```

7.6. Inspect the `/home/student/DO328/labs/observe-jaeger/gateway.yaml` file, which describes the ingress gateway for traffic entering the mesh.

Use the `oc create` command to create the ingress gateway.

```
[student@workstation observe-jaeger]$ oc create -f gateway.yaml
gateway.networking.istio.io/observe-jaeger-gateway created
```

7.7. Create a `VirtualService` to redirect the ingress traffic to `servicea`, which acts as an entry point into the mesh.

Examine the `virtual-service.yaml` file, which routes the ingress traffic to `servicea`.

Use the `oc create` command to create the virtual service.

```
[student@workstation observe-jaeger]$ oc create -f virtual-service.yaml
virtualservice.networking.istio.io/observe-jaeger-vs created
```

8. Test the microservices.

8.1. Run the `oc get route` command to get the URL of the Istio gateway.

You can also cut and paste the full command from the `get-ingress-gateway-url.sh` file.

Export the ingress gateway URL to an environment variable called `GATEWAY_URL`.

```
[student@workstation observe-jaeger]$ GATEWAY_URL=$( \
  oc get route istio-ingressgateway -n istio-system \
 -o template --template '{{ "http://" }}{{ .spec.host }}')
```

8.2. Execute the `curl` command in combination with the `GATEWAY_URL` variable to access the application.

```
[student@workstation observe-jaeger]$ curl ${GATEWAY_URL}/trace
Hello from ServiceA!.
Response from ServiceB => Hello from ServiceB!
```

9. Visualize traces generated by the microservices using the Jaeger web console.

9.1. Run the `oc get route` command to gather the Jaeger web console URL. You can also copy the commands from the `get-jaeger-url.sh` file.

```
[student@workstation observe-jaeger]$ JAEGER_URL=$( \
  oc get route jaeger -n istio-system \                                          1
 -o template --template '{{ "https://" }}{{ .spec.host }}')
```

9.2. Use Firefox web browser to access the Jaeger web console.

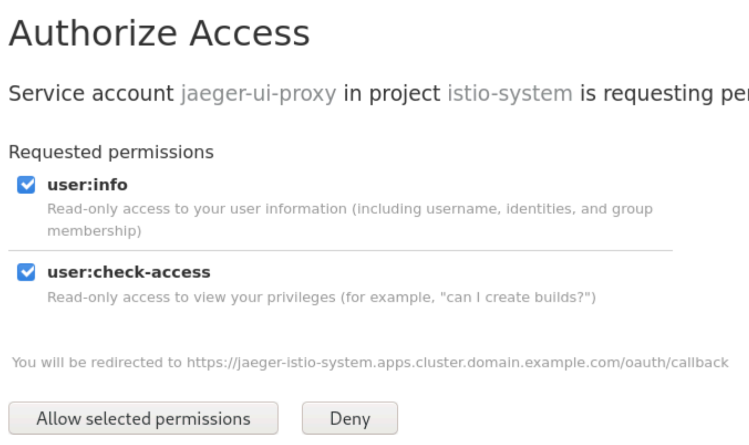
```
[student@workstation observe-jaeger]$ firefox ${JAEGER_URL} &
```

9.3. Click **Log in with OpenShift**.

Log in using your developer user account. Your user name is the `RHT_OCP4_DEV_USER` variable in the `/usr/local/etc/ocp4.config` classroom configuration file. Your password is the `RHT_OCP4_DEV_PASSWORD` variable in the same file.

If you are accessing the Jaeger web console for the first time, then you will be prompted with a page asking you to authorize service account access to your account.

Click **Allow selected permissions** to bring up the Jaeger web console.



Authorize Access

Service account jaeger-ui-proxy in project istio-system is requesting permission to access your account (developer)

Requested permissions

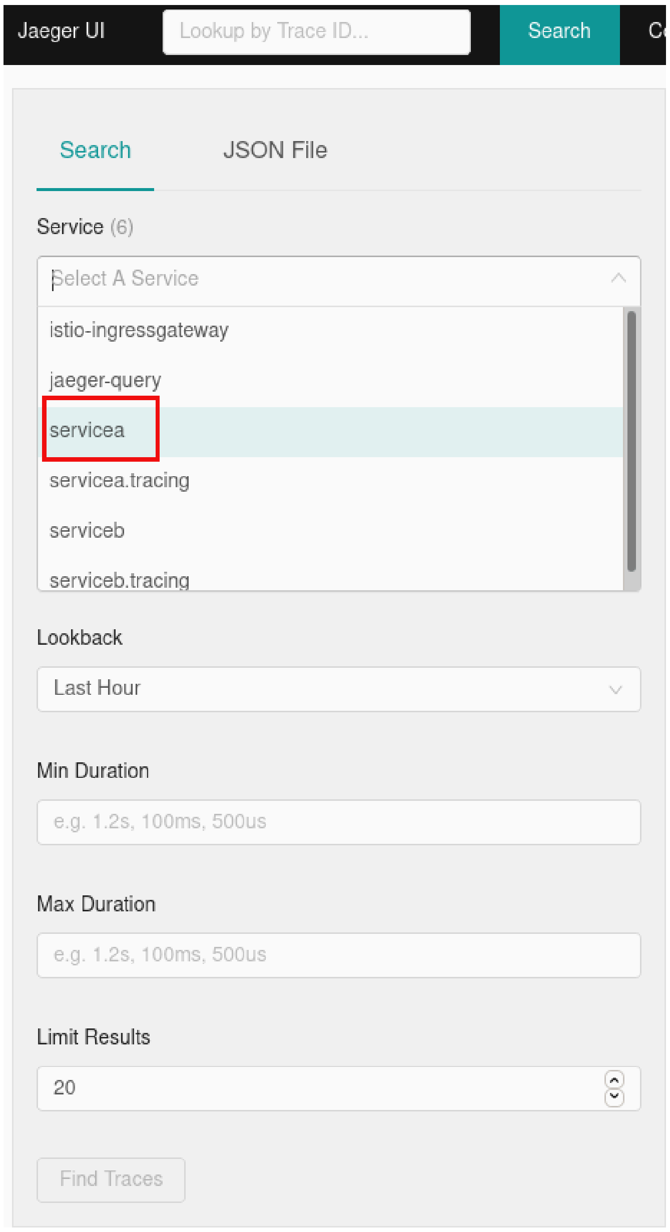
☑ **user:info**
Read-only access to your user information (including username, identities, and group membership)

☑ **user:check-access**
Read-only access to view your privileges (for example, "can I create builds?")

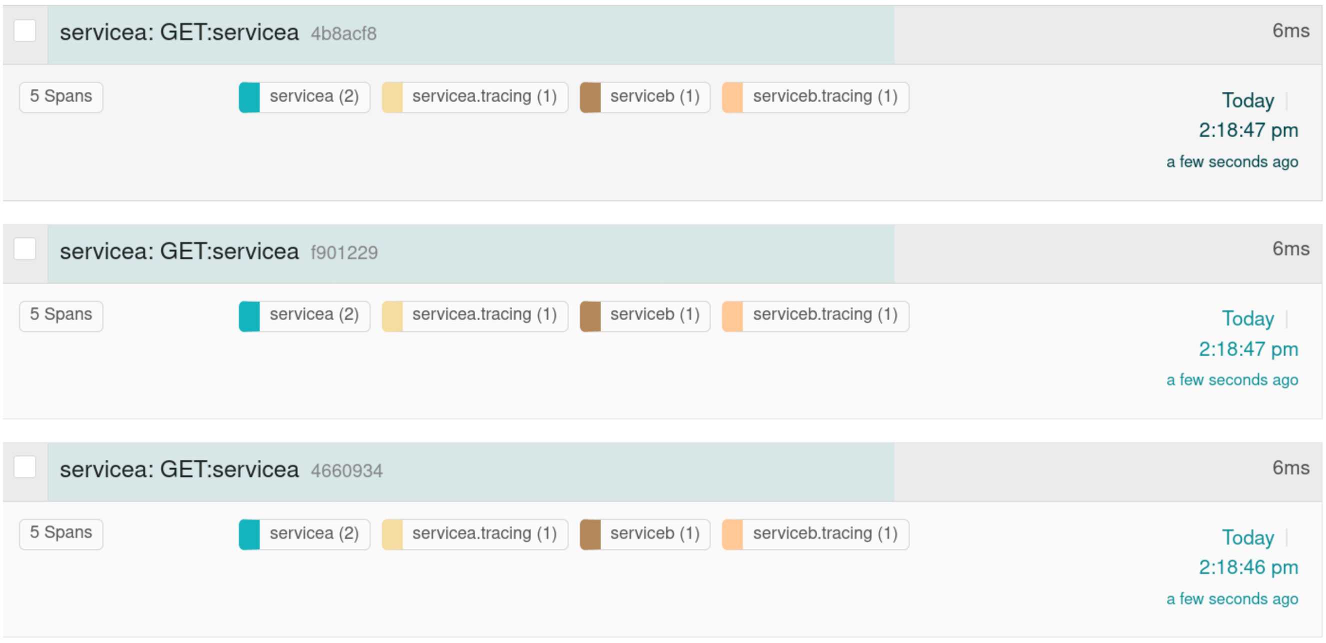
You will be redirected to https://jaeger-istio-system.apps.cluster.domain.example.com/oauth/callback

9.4. Execute the `curl` command against the URL `${GATEWAY_URL}/trace` multiple times to generate some load and allow the microservices to send traces and span information to Jaeger.

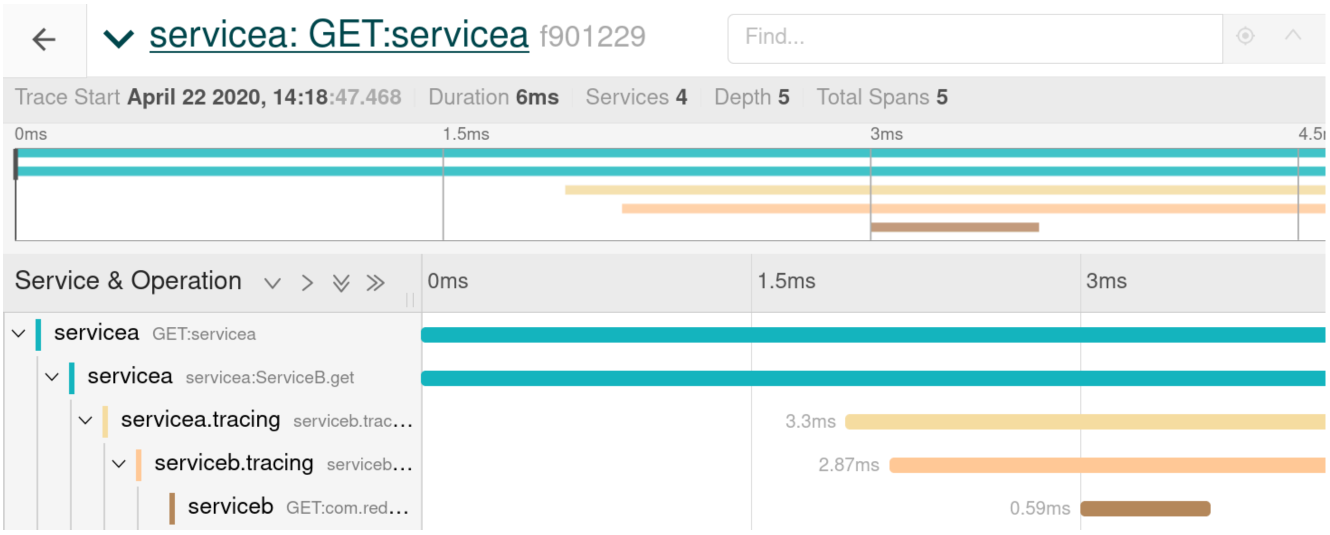
9.5. In the Jaeger web console, refresh the page and then select the `servicea` in the left panel. Click **Find Traces** to retrieve the traces.



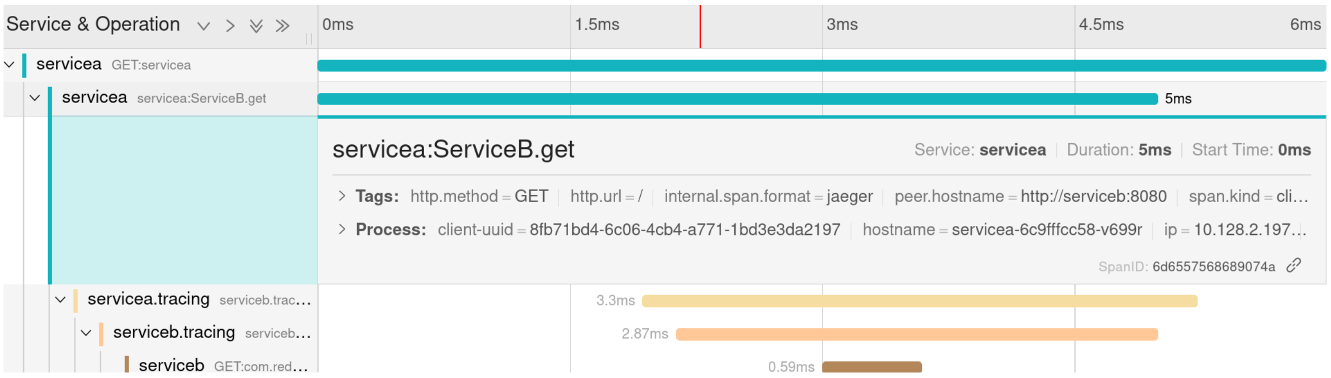
9.6. You should see a number of traces displayed, corresponding to the number of requests that you made to the service mesh.


```
servicea: GET:servicea 4b8acf8                                                    6ms
5 Spans   servicea (2)  servicea.tracing (1)  serviceb (1)  serviceb.tracing (1)  Today
                                                                                  2:18:47 pm
                                                                                  a few seconds ago

servicea: GET:servicea f901229                                                    6ms
5 Spans   servicea (2)  servicea.tracing (1)  serviceb (1)  serviceb.tracing (1)  Today
                                                                                  2:18:47 pm
                                                                                  a few seconds ago

servicea: GET:servicea 4660934                                                    6ms
5 Spans   servicea (2)  servicea.tracing (1)  serviceb (1)  serviceb.tracing (1)  Today
                                                                                  2:18:46 pm
                                                                                  a few seconds ago
```

Click any one trace and observe the spans reported by both microservices. Note how the spans from `serviceb` are shown as child spans of `servicea`.[1]


servicea: GET:servicea f901229
Find...
Trace Start April 22 2020, 14:18:47.468
Duration 6ms
Services 4
Depth 5
Total Spans 5
0ms
1.5ms
3ms
Service & Operation
0ms
1.5ms
3ms
servicea GET:servicea
servicea servicea:ServiceB.get
servicea.tracing serviceb.trac...
3.3ms
serviceb.tracing serviceb...
2.87ms
serviceb GET:com.red...
0.59ms

You can click each of the spans and note the runtime values, which have been propagated through HTTP headers as the traffic flows from `servicea` to `serviceb`.



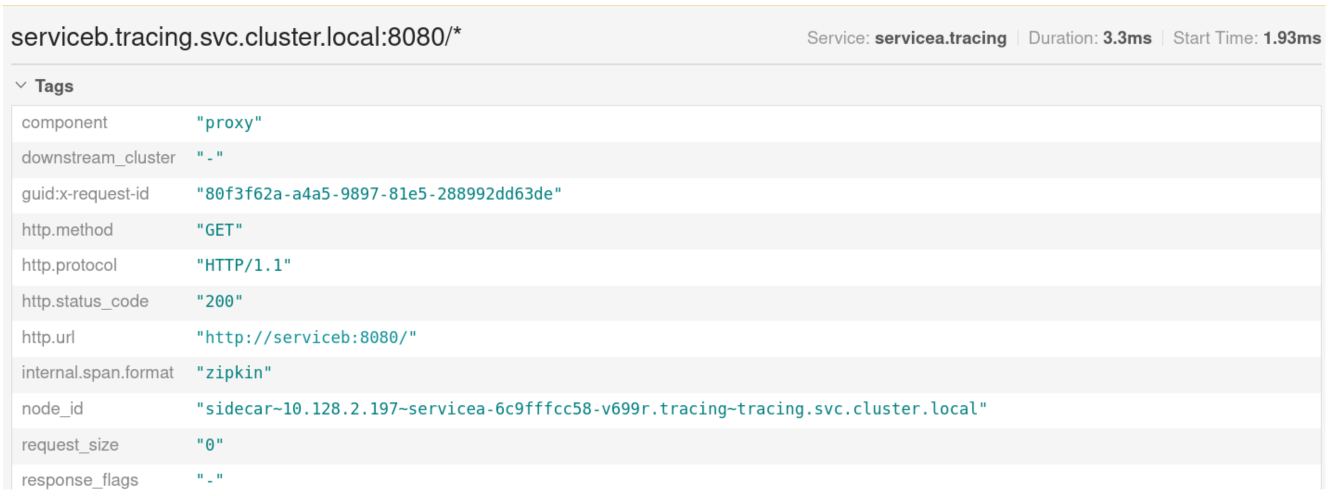
servicea:ServiceB.get Service: **servicea** Duration: **5ms** Start Time: **0ms**

Tags: http.method = GET http.url = / internal.span.format = jaeger peer.hostname = http://serviceb:8080 span.kind = cli...

Process: client-uuid = 8fb71bd4-6c06-4cb4-a771-1bd3e3da2197 hostname = servicea-6c9fffcc58-v699r ip = 10.128.2.197...

SpanID: 6d6557568689074a

Expand **Tags** for the `servicea.tracing` span to see contextual span information, which was propagated to services.



serviceb.tracing.svc.cluster.local:8080/* Service: **servicea.tracing** Duration: **3.3ms** Start Time: **1.93ms**

Tags

component	"proxy"
downstream_cluster	"-"
guid:x-request-id	"80f3f62a-a4a5-9897-81e5-288992dd63de"
http.method	"GET"
http.protocol	"HTTP/1.1"
http.status_code	"200"
http.url	"http://serviceb:8080/"
internal.span.format	"zipkin"
node_id	"sidecar~10.128.2.197~servicea-6c9fffcc58-v699r.tracing~tracing.svc.cluster.local"
request_size	"0"
response_flags	"-"

NOTE

You will see extra `*.tracing` spans in the Jaeger console. These are propagated by the Envoy proxy as it intercepts traffic bound for the services in the mesh.

.0. Return to the home directory.

```
[student@workstation observe-jaeger]$ cd
```

Finish

On the `workstation` machine, use the `lab` command to complete this exercise. This is important to ensure that resources from previous exercises do not impact upcoming exercises. 1

```
[student@workstation ~]$ lab observe-jaeger finish
```

This concludes the section.

< Previous (/rol/app/courses/do328-2.0/pages/ch03)

Next > (/rol/app/cou

Red Hat
Learning Community

Discuss Building Resilient Microservices with Istio and Red Hat OpenShift Service Mesh (https://learn.redhat.com/t5/DO328-Building-Resilient/gh-p/DO328BuildingResilientMicroserviceswithIstioandRedHatOpenShiftServiceMesh)

Welcome to the Building Resilient Microservices with Istio and OpenShift Service Mesh (DO328) group!
cschunke Aug 12, 2023

We are excited to launch a space dedicated to the Red Hat Training course, Building Resilient Microservices with Istio and Red Hat OpenShift Service Mesh! To gain the most value from this group - click the "Join Group" button in the upper right hand corner of the group home page.We encourage group members to...

3 1 432

Revision: do328-2.0-469a011

1