# Testing:

For this portion, I created an app.js file, following 2 other files, “tracing.js” for finding traces and “monitoring.js” for measuring the metrics. The app.js is listening on a port and consists of 2 different routes: 1. “/” and 2. “/date”. 1 displays text “Hello World” and 2 tells us the current date.

Tracing:

* For tracing, I ended up using Zipkin to display the traces, but since there was only 1 class (app.js) not much was visualized.

Metrics:

* For metrics, I used Prometheus, Grafana, and OtelCollecter in order to visualize them. Each metric consists of a MeterProvider – which consists of all metrics, MetricReader – which reads each metric from Provider every interval and sends them over to the MetricsExporter – which sends the metrics to the OtelCollecter.
* I also added a simple type of metric called, counter, and made it count every time a request was made to a specific route in the server.

So every time I visit a route on the server. Ex: “localhost/date”, a metric of that will then be recorded in my visualization tool, Prometheus and tracing of that call in Zipkin. In Prometheus, I can even see a graph of the metrics and in Zipkin I can see how the layout of the spans. More will be seen in the microservices application.

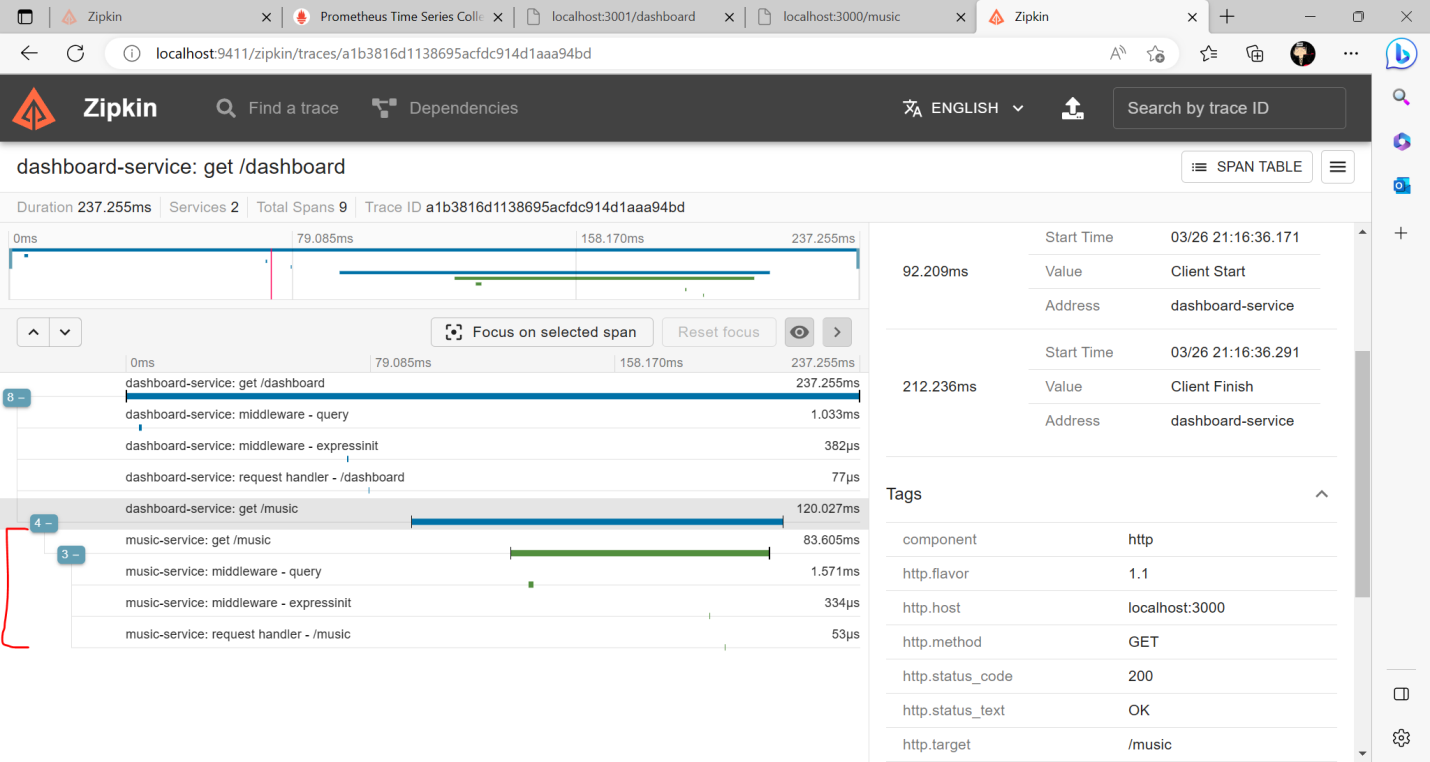
Package.json:

* Consists of all the dependencies used in all applications

# Microservices: (Received help from Youtube)

Most application are not mono, but they are comprised of miscroservices. Just for testing purposes, my application has 2 microservices linked together called, “music.js” and “dashboard.js”. ­ Both js files consist of zipkinExporters, tracerProviders, and ports to listen on. Dashboard.js relays the info from music.js, in that it just displays the contents. You can say that dashboard.js relies on music.js, since it is constantly checking if music.js’s server is active or not, through the use of a promise. This is why you absolutely have to run music.js first in order to see some results.

You can then view all the traces from these microservies through Zipkin. To work zipkin, I created a container of it in Docker and ran it before running the actual services I wanted to trace. In zipkin, you can tell what exactly happened when the dashboard.js’s server was ran.



The red line highlights where dashboard requests the data from music.js. The green span is the entirety of where music.js is accessed. The info on the right gives insight of the start time, value, and address, along with spanID and parentID of the particular span that was selected.

# Github Reporitory:

<https://github.com/Attested-paper51/OpenTelemetry>