Instrumentation for Spring Boot Applications

Document Level Classification

200

Introduction

This guide provides a detailed overview of the Micrometer OTLP, offering insights on how to efficiently integrate and utilize it in Java applications to gather telemetry data, including metrics.

Getting Started

Prerequisites

- Java 8+ is required.
- If you are using Spring Boot 3, you need Java JDK 17+.
- Gradle.

Gotchas

These are learnings from migrating existing services from NewRelic to OTEL and should be taken into consideration when migrating from NewRelic to OTEL.

NewRelic Micrometer Registry

In some cases we have seen the use of a custom implementation of a NewRelic metric registry so these metrics are sent to NewRelic. The solution is to switch the registry to use epic-common-metrics-micrometer which OTEL will collect micrometer metrics from.

Example Solution: Launcher-Service

NewRelic

We have seen hard-coded NewRelic counters in code:

```
NewRelic.incrementCounter("Custom/Throttling/Count");
```

The solution is to switch these to use Micrometer:

```
import io.micrometer.core.instrument.Metrics;...Metrics.counter("thrott
```

Example Solution: Launcher-Service

Automatic Instrumentation

If your service is built on Epic's shared uberjar-service-corretto image, you will need to at least build **360135c7566d3e1cbbbd48e5469dfa26ee4938e7.b177** (or an image built after June 14, 2023) to enable auto-instrumentation.

Automatic instrumentation simplifies telemetry integration. When using `Epic App` you can enable auto-instrumentation following these steps:

- Add the following annotation to your Helm chart to enable auto-instrumentation:

```
podAnnotations:
  instrumentation.opentelemetry.io/inject-java: "observability/java-v2"
```

If you have multiple containers (for example a sidecar), the first container listed in your container spec will be instrumented. If this is not ideal, you can clarify this by adding an additional annotation:

```
    instrumentation.opentelemetry.io/container-names: "myapp,myapp2"
```

Installation of Micrometer OTLP

You **do not need** additional configuration as it works out of the box using environment variables. The OpenTelemetry auto-instrumentation agent automatically registers an OTLP micrometer registry. You **do not need to initialize your own.**

From the standard we have at Epic for Java OpenTelemetry Metrics, this are some of the decisions on the standard:

- Metrics MUST be exported using the provided Java Agent. This will automatically install the micrometer bridge for you.
  - You MUST NOT install the micrometer bridge explicitly.
  - You MUST NOT install any other library from `io.opentelemetry` or `io.micrometer`. Both of these will result in multiple instrumentations and duplicate metrics.
  - Services MUST NOT enable any other registry configuration (e.g. Prometheus).
  - Services SHOULD NOT override the default export configuration defined by observability team (in env vars via epic-app).

Configuration

You do not need to configure anything manually, as the OTEL operator running in your k8s cluster will decorate your service pods with the necessary configuration via environment variables, such as `OTEL_EXPORTER_OTLP_METRICS_ENDPOINT`, `OTEL_EXPORTER_OTLP_HEADERS`, etc.

Manual Instrumentation

We have prepared a demo, csk-apm-demo, to help you learn about manual instrumentation using the epic-app Helm chart and Micrometer OTLP. This demo also covers methods for instrumenting your code. You can also read more about how to configure custom metrics in the documentation.

Enable Auto-Instrumentation

Even if you are doing manual instrumentation, enabling auto-instrumentation is encouraged, because it facilitates the configuration of everything required. Auto-instrumentation sets up the OTLP exporters, configures the collectors, and adds standard metrics, allowing you to focus on adding custom telemetry without dealing with low-level details.

To enable auto-instrumentation in your epic-app Helm chart follow the instructions above in **Automatic Instrumentation**.

Adding code for Instrumentation

This basic setup has no effect on your app yet. You need to add code for creating custom instrumentation of your app.

Traces

Epic does not yet have a generalized tracing storage system, as this is a very expensive problem, but we have a few tools we are experimenting with to provide visibility into tracing data. By default, OpenTelemetry auto-instrumentation is configured to send traces with a 1% sampling rate. You are encouraged to instrument your applications as though we have a full tracing today - even with 1% sampling, you can use the available traces to gain insight to your applications. When we do implement a complete tracing solution, we can enable more sophisticated sampling strategies. For more information, see the State of Tracing and Guidance for 2024.

The following tools are available to analyze trace data:

- Trace Metrics - these are generated by the OTEL collectors by aggregating span counts and their duration and producing metrics. While not very granular, they can give you a better idea of where you are spending time in your application. Since the metrics are generated based on span name, please avoid putting high cardinality data in your span names. You can see a working example of trace metrics in our OTEL unified dashboard.
- Tempo - accessed via a Grafana datasource, this will let you query and view complete traces. It will also render a service map for your service which can help you see relationships to other applications, if they are also instrumented with the OTEL agent. You can see an example of this in use in the OTEL unified dashboard.

Metrics

OpenTelemetry has support for metrics. A **metric** is a **measurement** of a service captured at runtime. The moment of capturing a measurements is known as a **metric event**, which consists not only of the measurement itself, but also the time at which it was captured and associated metadata.

Metrics combine individual measurements into aggregations, and produce data which is constant as a function of system load.

JMX Metrics

If your application exports metrics via JMX, you can configure the OTEL agent to collect those. See the jmxagent documentation here. By default, the OTEL agent looks for a JMX config file at /epicgames/otel/jmx.yaml in your container.

Converting JMX Metrics from an existing NR JMX Config file

If your app is currently using the JMX Metrics Extension for New Relic, use the handy NewRelic Extention Configuration to OTEL JMX Configuration utility to convert your existing configuration to an OTEL-compatible config, then simply drop it in your container at /epicgames/otel/jmx.yaml.

Supported Metrics

A **Meter** is the interface for collecting a set of measurements (which we individually call metrics) about your application. Micrometer supports a set of `Meter` primitives, but the Micrometer OTLP library supports exporting the following data points in OTLP format:

1. Sums
2. Gauge
3. Histogram
4. Summary

The following table maps OTLP data points and the Micrometer meters:

OTLP data point	Micrometer meter type
Sums	Counter, FunctionCounter
Gauge	Gauge, TimeGauge, MultiGauge
Histogram	Timer, DistributionSummary, LongTaskTimer, FunctionTimer (only sum and count are set)
Summary	Timer, DistributionSummary, LongTaskTimer

In the official documentation you can explore the full list of supported Micrometer types and the equivalent OTLP data point.

Acquiring a Meter Registry

You can acquire a `MeterRegistry` instance anywhere in your application where you need to collect and manage metrics.

This is how we did it in our example `csk-apm-demo`.

Typically, you want to inject the `MeterRegistry` instance.

```
private final MeterRegistry meterRegistry;
```

To initialize the `MeterRegistry` you need to inject it into your class.

```
this.meterRegistry = registry;
```

The `MeterRegistry` allows you to centralize and manage all your application's custom metrics you might add.

For example, to create and increment a custom counter metric, use the following code:

```
Counter.builder("http.requests").tags(tags).register(meterRegistry).inc
```

⚠️⚠️ We recommend you follow naming meters convention that separates words with a dot (e.g., "http.requests"), to help guarantee the portability of metric names across multiple monitoring systems. In Grafana you will find this metric as "http_requests".

Tags

Now that we have a counter, or any other meter primitive for that matter, we want to give it attributes. An identifier of a Meter consists of a name and tags.

Metrics represent quantitative data to be measured, while **tags or dimensions** provide context to those metrics. Micrometer uses the terms, “dimensions” and “tags,” interchangeably, and the Micrometer interface is `Tag` simply because it is shorter

- **Metrics** in OpenTelemetry are used to capture measurements and are usually quantitative data points, like counters or gauges.

- **Tags** are key-value pairs given to metrics to provide additional context, helping to reason about the data, and organize and filter the data.

When you generate metrics, adding attributes creates unique metric series, based on each distinct set of attributes that receive measurements.

This leads to the concept of **cardinality**, which is the total number of unique series. Cardinality directly affects the size of the metric payloads that are exported. Therefore, it's important to **carefully** select the dimensions included in these attributes to prevent a surge in cardinality, often referred to as a 'cardinality explosion'.

Tag Definition

A Tag is a key-value pair with the following properties:

- **Tag Key**:
    - MUST be a non-null and non-empty string.
    - Case sensitivity is preserved. Keys that differ in casing are treated as distinct keys.
- **Attribute Value** is either:
    - A primitive type: string, boolean, double precision floating point (IEEE 754)**.**

Be cautious about what data you store in a tag, as each distinct tag value produces a unique metric. A service with a hundred pods each emitting a thousand unique label values will produce 100,000 metrics. Please store your high cardinality values as log or trace data.

Tag Naming

We recommend that you follow the same lowercase dot notation convention for meter names when naming tags. Using this consistent naming convention for tags allows for better translation into the respective monitoring system's idiomatic naming schemes.

Adding Tags to Metrics

Tags are a powerful feature in Micrometer that allows you to add dimensions to your metrics. This can help you analyze and segment your data more effectively.

Below, we explain how to create and use tags in your metrics. We started by defining a set of base tags that will be common across for our get metric. In this example, tags for the `uri` and `method` are created.

```
var baseTags = Tags.of("uri", "/api/v1/storedvalue/", "method", "GET");
```

⚠️⚠️Note we generalize the uri here and **omit** the actual value for **storedvalue** as this would generate a high cardinality label.

Then we created specific counters which extend the `baseTags` with additional information, such as HTTP status, based on the results.

We also increment our counter for `"http.requests"`

```
    var successTags = baseTags.and("status", "200");
    Counter.builder("http.requests")
            .tags(successTags)
            .register(meterRegistry)
            .increment();
```

```
    var notFoundTags = baseTags.and("status", "404");
    Counter.builder("http.requests")
            .tags(notFoundTags)
            .register(meterRegistry)
            .increment();
```

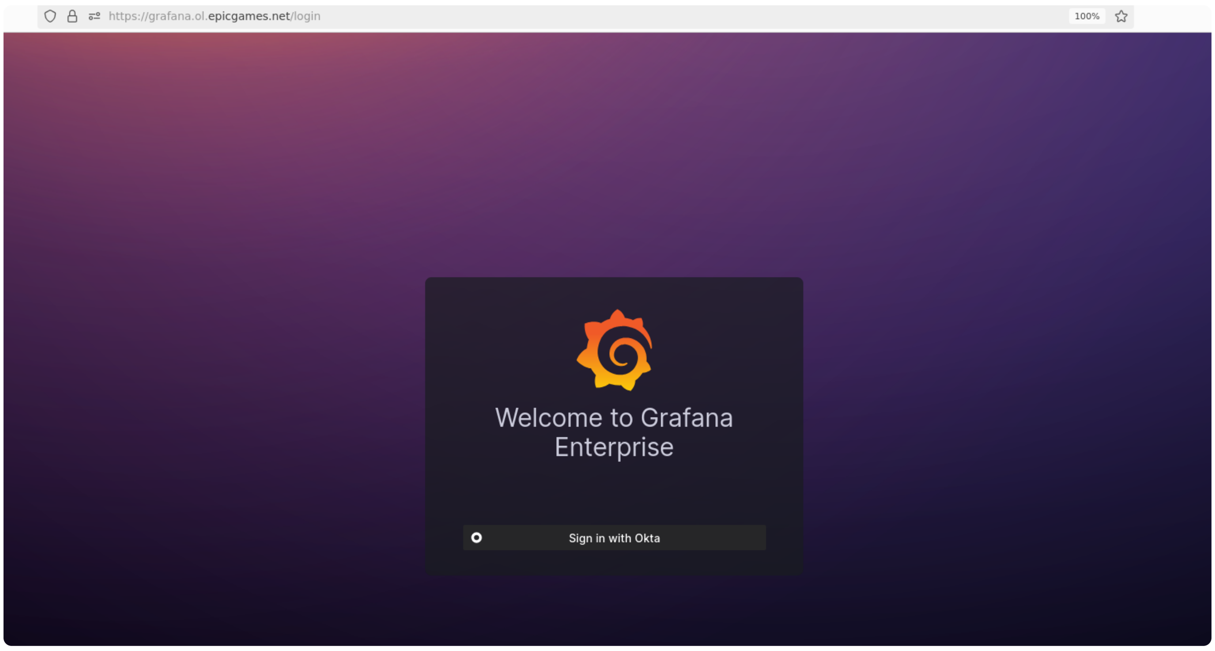
Visualizing Metrics with Grafana

After creating Micrometer OTLP metrics for your application with the appropriate attributes, you'll want to visualize these metrics. We'll use Grafana for this purpose.

Connecting to Grafana

To connect to Grafana, navigate to Grafana Core at this link. Ensure you have the "Epic Grafana (Prod)" tile in Okta. If you don't, request it in SailPoint by searching for "Grafana Epic Live" and requesting either the "Grafana Epic editor" or "Grafana Epic viewer" role.

Logging into Grafana


https://grafana.ol.epicgames.net/login
100%
Welcome to Grafana Enterprise
Sign in with Okta

1. Visit Grafana.
2. Click "Sign in with Okta."

OTEL Unified Dashboards

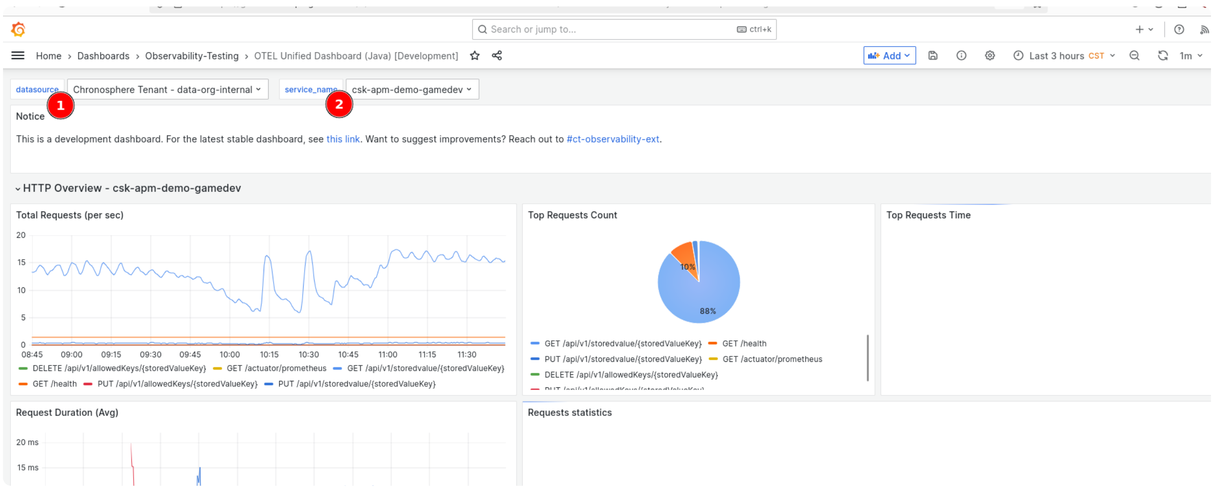
Once you have instrumented your application, you can use our pre-made Grafana dashboards to view general metrics about your application's performance.

First, you may want to use our OTEL Unified Dashboard (Java) - Development.

This dashboard is useful during the development phase of your application, before it has been deployed to production.

Make sure to select the proper:

1. Datasource
2. Service name.



datasource: Chronosphere Tenant - data-org-internal
service_name: csk-apm-demo-gamedev

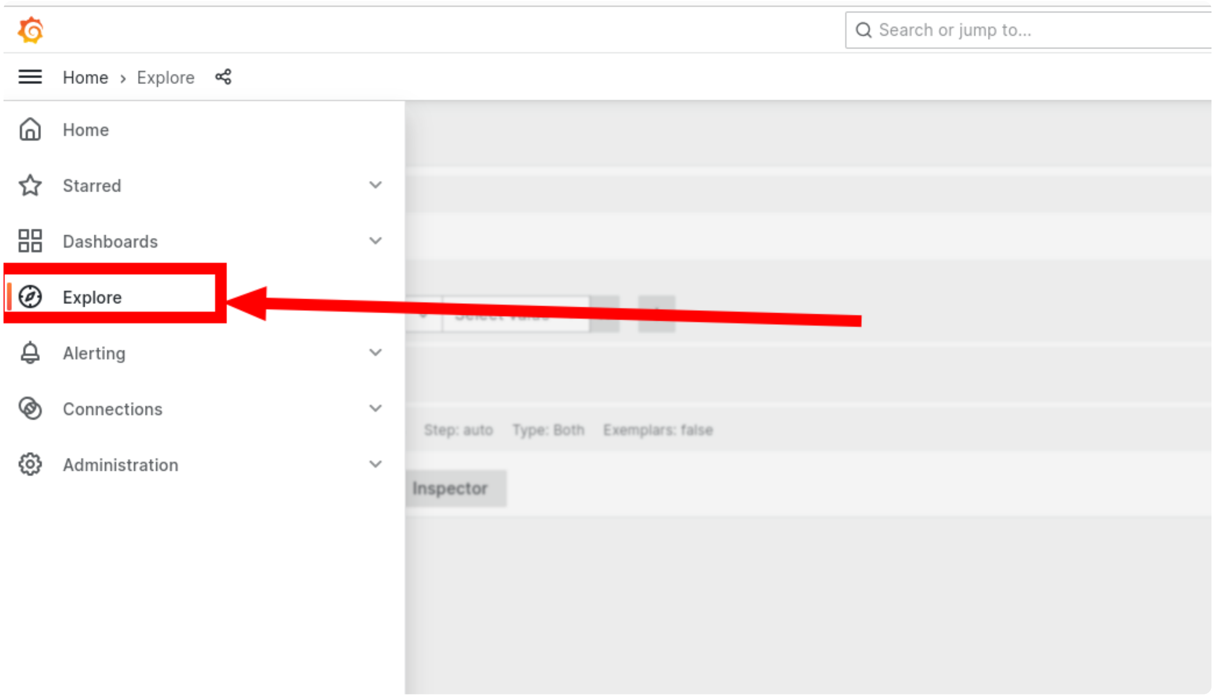
Notice

This is a development dashboard. For the latest stable dashboard, see this link. Want to suggest improvements? Reach out to #ct-observability-ext.

Once your application is in production, you can also use the OTEL Unified Dashboard (Java) to gain insights about your application, this also requires you to select the proper data source and service name.

Find Grafana Explore Feature

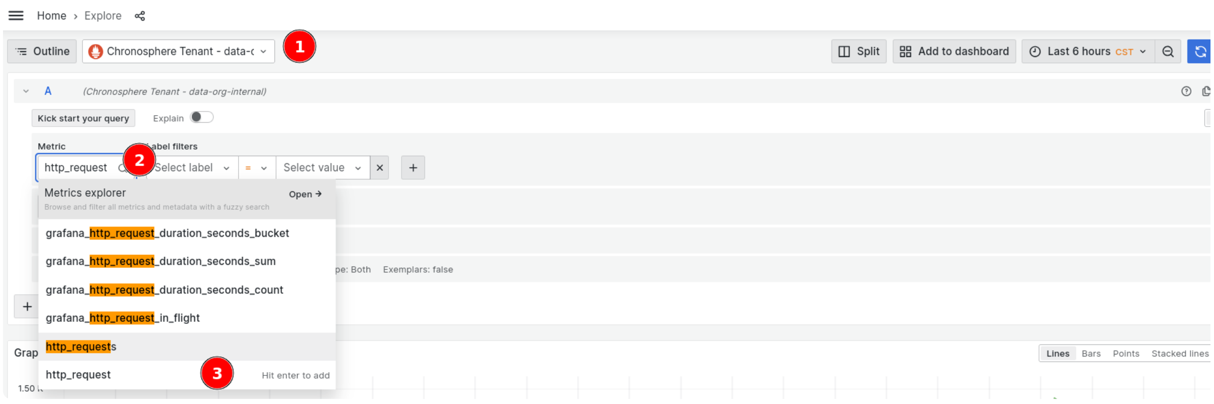
Before creating a dashboard, explore your data to verify it and construct your queries. Use Grafana's Explore feature, which simplifies the interface and lets you focus solely on querying your data.



1. Navigate to "Home > Explore."
2. Learn more about the Explore feature [in the official docs](#).

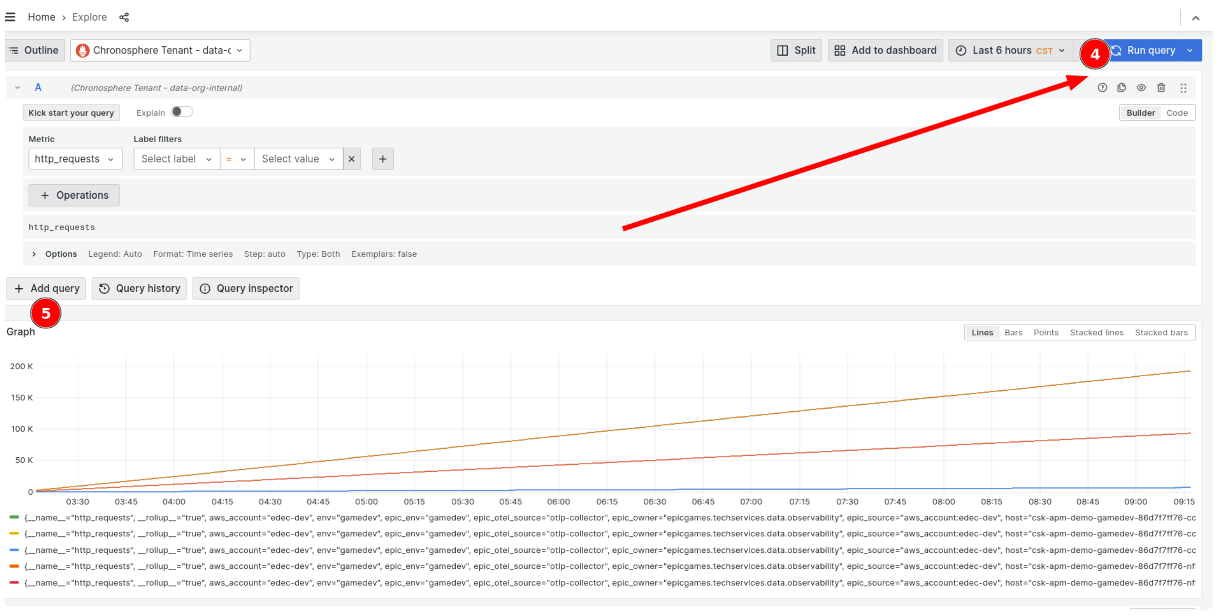
Finding and Exploring Your Data

Once you are inside the Explore feature, you can start by selecting the appropriate Data Source. For this tutorial, the appropriate data source is "Chronosphere - Global," the Prometheus backend that we are using at Epic Games.

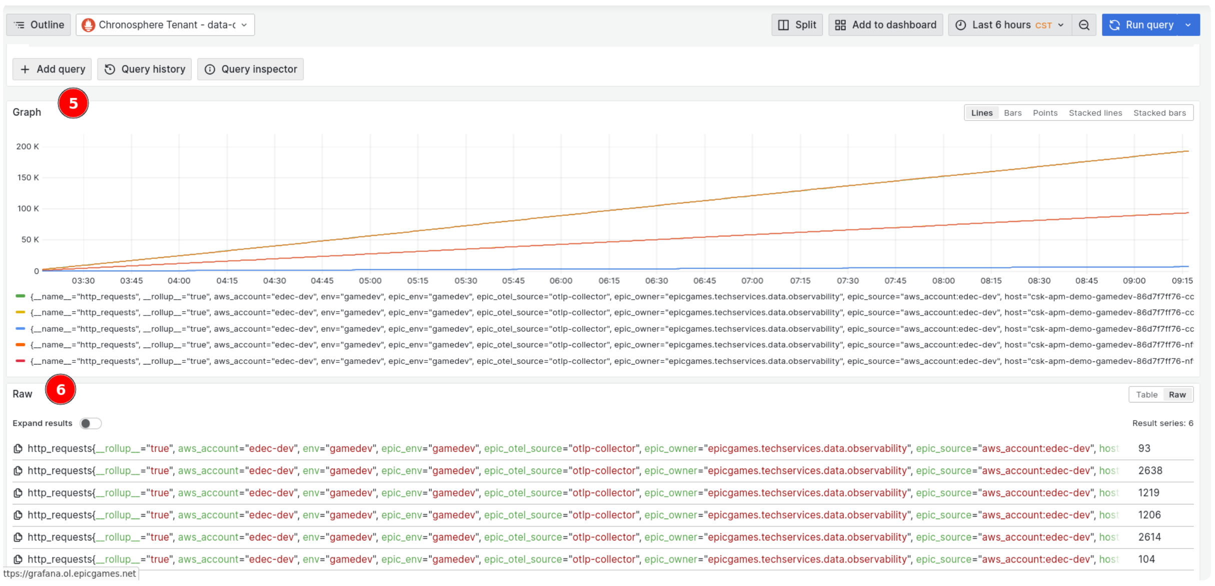


Home › Explore
Outline
Chronosphere Tenant - data-c
1
Split
Add to dashboard
Last 6 hours CST
A
(Chronosphere Tenant - data-org-internal)
Kick start your query
Explain
Metric
Label filters
http_request
2
Select label
=
Select value
Metrics explorer
Browse and filter all metrics and metadata with a fuzzy search
Open →
grafana_http_request_duration_seconds_bucket
grafana_http_request_duration_seconds_sum
grafana_http_request_duration_seconds_count
grafana_http_request_in_flight
http_requests
http_request
3
Hit enter to add
Lines
Bars
Points
Stacked lines

1. Click on the data sources option and select the correct data source for your data, in our case "Chronosphere - data-org-internal."
2. Click on the "Metric" section and start typing the name of the metric you are looking for.
3. In this tutorial, we created the following metrics: `http_requests`. So that is the want we selected to explore.
4. Adjust the filters and options as necessary, then click on "Run Query."
5. Visualize your data in the "Graph" section.
6. If you scroll down, you can also see the "Raw" section with your data.



```
{__name__="http_requests", __rollup__="true", aws_account="edec-dev", env="gamedev", epic_env="gamedev", epic_otel_source="otlp-collector", epic_owner="epicgames.techservices.data.observability", epic_source="aws_account:edec-dev", host="csk-apm-demo-gamedev-86d7f7ff76-cc
{__name__="http_requests", __rollup__="true", aws_account="edec-dev", env="gamedev", epic_env="gamedev", epic_otel_source="otlp-collector", epic_owner="epicgames.techservices.data.observability", epic_source="aws_account:edec-dev", host="csk-apm-demo-gamedev-86d7f7ff76-cc
{__name__="http_requests", __rollup__="true", aws_account="edec-dev", env="gamedev", epic_env="gamedev", epic_otel_source="otlp-collector", epic_owner="epicgames.techservices.data.observability", epic_source="aws_account:edec-dev", host="csk-apm-demo-gamedev-86d7f7ff76-cc
{__name__="http_requests", __rollup__="true", aws_account="edec-dev", env="gamedev", epic_env="gamedev", epic_otel_source="otlp-collector", epic_owner="epicgames.techservices.data.observability", epic_source="aws_account:edec-dev", host="csk-apm-demo-gamedev-86d7f7ff76-nf
{__name__="http_requests", __rollup__="true", aws_account="edec-dev", env="gamedev", epic_env="gamedev", epic_otel_source="otlp-collector", epic_owner="epicgames.techservices.data.observability", epic_source="aws_account:edec-dev", host="csk-apm-demo-gamedev-86d7f7ff76-nf
```



```
{__name__="http_requests", __rollup__="true", aws_account="edec-dev", env="gamedev", epic_env="gamedev", epic_otel_source="otlp-collector", epic_owner="epicgames.techservices.data.observability", epic_source="aws_account:edec-dev", host="csk-apm-demo-gamedev-86d7f7ff76-cc
{__name__="http_requests", __rollup__="true", aws_account="edec-dev", env="gamedev", epic_env="gamedev", epic_otel_source="otlp-collector", epic_owner="epicgames.techservices.data.observability", epic_source="aws_account:edec-dev", host="csk-apm-demo-gamedev-86d7f7ff76-cc
{__name__="http_requests", __rollup__="true", aws_account="edec-dev", env="gamedev", epic_env="gamedev", epic_otel_source="otlp-collector", epic_owner="epicgames.techservices.data.observability", epic_source="aws_account:edec-dev", host="csk-apm-demo-gamedev-86d7f7ff76-cc
{__name__="http_requests", __rollup__="true", aws_account="edec-dev", env="gamedev", epic_env="gamedev", epic_otel_source="otlp-collector", epic_owner="epicgames.techservices.data.observability", epic_source="aws_account:edec-dev", host="csk-apm-demo-gamedev-86d7f7ff76-nf
{__name__="http_requests", __rollup__="true", aws_account="edec-dev", env="gamedev", epic_env="gamedev", epic_otel_source="otlp-collector", epic_owner="epicgames.techservices.data.observability", epic_source="aws_account:edec-dev", host="csk-apm-demo-gamedev-86d7f7ff76-nf
```

Result series: 6

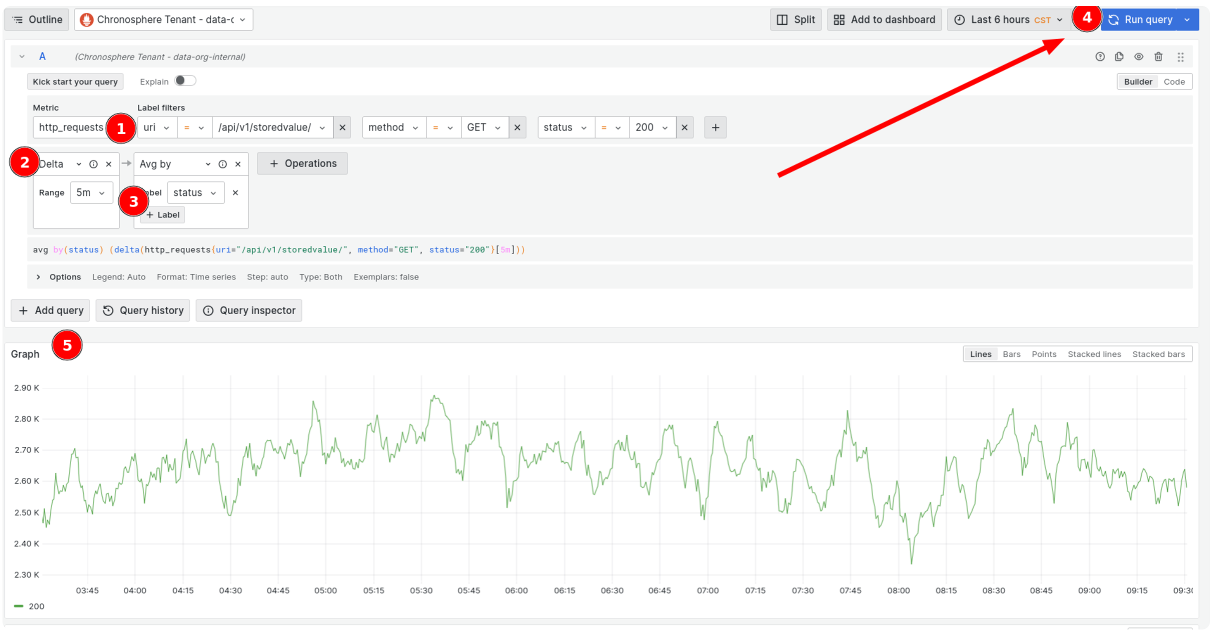
```
http_requests{__rollup__="true", aws_account="edec-dev", env="gamedev", epic_env="gamedev", epic_otel_source="otlp-collector", epic_owner="epicgames.techservices.data.observability", epic_source="aws_account:edec-dev", host   93
http_requests{__rollup__="true", aws_account="edec-dev", env="gamedev", epic_env="gamedev", epic_otel_source="otlp-collector", epic_owner="epicgames.techservices.data.observability", epic_source="aws_account:edec-dev", host   2638
http_requests{__rollup__="true", aws_account="edec-dev", env="gamedev", epic_env="gamedev", epic_otel_source="otlp-collector", epic_owner="epicgames.techservices.data.observability", epic_source="aws_account:edec-dev", host   1219
http_requests{__rollup__="true", aws_account="edec-dev", env="gamedev", epic_env="gamedev", epic_otel_source="otlp-collector", epic_owner="epicgames.techservices.data.observability", epic_source="aws_account:edec-dev", host   1206
http_requests{__rollup__="true", aws_account="edec-dev", env="gamedev", epic_env="gamedev", epic_otel_source="otlp-collector", epic_owner="epicgames.techservices.data.observability", epic_source="aws_account:edec-dev", host   2614
http_requests{__rollup__="true", aws_account="edec-dev", env="gamedev", epic_env="gamedev", epic_otel_source="otlp-collector", epic_owner="epicgames.techservices.data.observability", epic_source="aws_account:edec-dev", host   104
```

Transforming your data

The above instructions show you how to visualize the data, but it doesn't look good. As you can see, the `http_requests` metric produces cumulative metrics. In our case, `http_requests` will only increase as more requests are served. These metrics provide the total count since the last reset (e.g., since the application start).

To transform the data into a useful graphic, we did the following.

1. Included labels to limit the data we want to see. Specifically, we used the "uri", "method", and "status" labels to focus our visualization on GET requests with a 200 status code.
2. Then, since this is a cumulative metric, we applied a `delta` operation with a range of "5m".
3. We also averaged the data by the "status" label, using the "avg by" operation, to smooth out the visualization.
4. We hit "Run query" again
5. The Graph section now looks better.



Metric: http_requests

Label filters: uri = /api/v1/storedvalue/ · method = GET · status = 200

Delta — Range: 5m → Avg by — Label: status

```
avg by(status) (delta(http_requests{uri="/api/v1/storedvalue/", method="GET", status="200"}[5m]))
```

Options Legend: Auto Format: Time series Step: auto Type: Both Exemplars: false

This is just a simple example of how to visualize your metrics with Grafana, the correct Labels and Operations that you need to use will depend on your metrics,

Learning to Create Dashboards and Alerts in Grafana

For comprehensive instructions on creating dashboards and alerts for your data, refer to our Grafana User Manual. This manual covers topics such as creating dashboards, setting up alerts, notification policies, best practices, how to get support, and more. You can always reach out to us on #ct-obs-support-ext in slack for help and guidance.

Page Information:
Page ID: 81068818
Space: Cloud Developer Platform
Downloaded: 2025-07-12 04:08:46