Using the OpenTelemetry Java SDK

Document Level Classification

200

Spring Boot Services

Refer to this documentation instead: Instrumentation for Spring Boot Applications (OATS Java Standard)

Introduction

This guide provides a detailed overview of the OpenTelemetry Java SDK, offering insights on how to efficiently integrate and utilize the SDK in Java applications to gather telemetry data, including metrics, traces, and logs.

Getting Started

Prerequisites

- Java 8+ is required for OpenTelemetry.
- Gradle

Installation of OTEL SDK

Refer here for installation instructions using Maven and Gradle

Automatic 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-v
```

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.

Add the necessary dependency to your service, since we are doing automatic instrumentation you only need to add `opentelemetry-api` because the previous annotation will handle everything else for you.

Here is an example of how to do it with Gradle using `libs.versions.toml`

```
otel-api = { group = "io.opentelemetry", name = "opentelemetry-api" }
```

Configure a bean for use in your service code, as outlined in the docs

```
@Configuration
public class OpenTelemetryConfig {
```

```
    @Bean
    public OpenTelemetry openTelemetry() {
        return AutoConfiguredOpenTelemetrySdk.initialize().getOpenTelem
    }
}
```

Manual Instrumentation Guide

We have prepared a demo, csk-apm-demo, to help you learn about manual instrumentation using the epic-app Helm chart and the OpenTelemetry SDK for Java. This demo also covers methods for instrumenting your code. You can also read more about how to configure custom metrics on 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, add the following annotation:

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

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.

Components

The OpenTelemetry Java SDK encompasses various functional components such as Metrics, Traces, and Logs.

Explore the full list of components and available artifacts for development here.

To include a specific component in your project, follow the instructions in Published Released to include the Bill of Material (BOM).

In general, you can specify the dependency as follows, replacing `{{artifact-id}}` with the value from the column "Artifact ID" in components

Maven:

```
<dependency>
  <groupId>io.opentelemetry</groupId>
  <artifactId>{{artifact-id}}</artifactId>
</dependency>
```

Gradle:

```
implementation('io.opentelemetry:{{artifact-id}}')
```

Add Necessary Dependencies

To ensure that the versions of dependencies (including transitive ones) are aligned, importing the `opentelemetry-bom` Bill of Material (BOM) is essential. This alignment is crucial for maintaining compatibility and preventing dependency conflicts across all OpenTelemetry dependencies.

A note about versions

OpenTelemetry Compatibility Notice

Ensure you use compatible versions of OpenTelemetry Java auto-instrumentation and the OpenTelemetry Java SDK with the current version of the OpenTelemetry Operator for optimal functionality.

We are currently using the OpenTelemetry Operator version 0.98.0 in Substrate.

This version of the operator is compatible with the OpenTelemetry Java auto-instrumentation version 1.32.1. This auto-instrumentation version is compatible with the OpenTelemetry Java SDK version 1.34.1.

The OpenTelemetry Java SDK is part of the OpenTelemetry Java release 1.35.1. The link includes the OpenTelemetry Java libraries available.

This is the list for supported libraries, frameworks, application servers, and JVMs for OpenTelemetry Java auto-instrumentation version 1.32.1.

Gradle Example

First, we imported the BOM in our `build.gradle` file in lines 6 to 10 of our `csk-apm-demo`

```
 dependencyManagement {
    imports {
        mavenBom("io.opentelemetry:opentelemetry-bom:1.34.1")
    }
}
```

Then, we included the required dependencies in our demo:

```
dependencies {
    implementation libraries.epic_common.oauth
    implementation libraries.epic_common.lettuce

    implementation("io.opentelemetry:opentelemetry-api")
    implementation("io.opentelemetry:opentelemetry-sdk")
    implementation("io.opentelemetry:opentelemetry-exporter-otlp:1.34.1
    implementation("io.opentelemetry:opentelemetry-exporter-logging")
    implementation("io.opentelemetry.semconv:opentelemetry-semconv:1.25
    implementation("io.opentelemetry:opentelemetry-sdk-extension-autoco
}
```

Note that we included the OpenTelemetry Protocol (OTLP) exporter. The OTLP exporter is crucial as it sends telemetry data to the OpenTelemetry backend:

```
 implementation("io.opentelemetry:opentelemetry-exporter-otlp:1.34.1")
```

Maven Example

If you are using Maven, here is an example of how you could add the OpenTelemetry BOM and some dependencies:

```
<project>
 <dependencyManagement>
     <dependencies>
         <dependency>
             <groupId>io.opentelemetry</groupId>
             <artifactId>opentelemetry-bom</artifactId>
             <version>1.38.0</version>
             <type>pom</type>
             <scope>import</scope>
```

```
            </dependency>
        </dependencies>
    </dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>io.opentelemetry</groupId>
            <artifactId>opentelemetry-api</artifactId>
        </dependency>
                    <dependency>
                            <groupId>io.opentelemetry</groupId>
                            <artifactId>opentelemetry-exporter-otlp</artifa
                    </dependency>
    </dependencies>
</project>
```

OpenTelemetry SDK Initialization

To set up the OpenTelemetry SDK for manual instrumentation, you'll need to import the necessary libraries and **initialize the SDK as early as possible** in your application lifecycle. Here's how you can do it:

Required Imports

First, import the required OpenTelemetry libraries. In our our `csk-apm-demo` we did this:

```
import io.opentelemetry.api.OpenTelemetry;
import io.opentelemetry.sdk.autoconfigure.AutoConfiguredOpenTelemetrySd
```

In this case we also imported Spring's `@Bean` like this:

```
import org.springframework.context.annotation.Bean;
```

SDK Initialization

To initialize the OpenTelemetry SDK, we defined a `@Bean` method in our Spring application class. This allows the SDK to be configured automatically using the environment variables or system properties set by the auto-instrumentation step we did earlier. Here is how we did it:

```
@Bean
public OpenTelemetry openTelemetry() {
    return AutoConfiguredOpenTelemetrySdk.initialize().getOpenTelemetry
}
```

By doing this, we ensure that the SDK is set up early in the application's lifecycle. The `AutoConfiguredOpenTelemetrySdk` class initializes the SDK based on the provided configuration, helping you avoid manual setup complexity.

Since we enabled auto-instrumentation in the first step, we leverage autoconfiguration, which simplifies the process by automatically discovering and configuring OpenTelemetry components.

To verify your code, build and run the app.

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.

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.

Initialize Metrics

If you followed the steps so far, you have a `MeterProvider` setup for you already. This lets you create a `Meter`.

A Meter creates metric instruments, capturing measurements about a service at runtime.

Import the Meter Library

To use the `Meter` in your application, you need to import the appropriate library. This is how we did it in our `csk-apm-demo` project:

```
import io.opentelemetry.api.metrics.Meter;
```

Acquiring a Meter

You can acquire a `Meter` instance anywhere in your application where you have manually instrumented code. To do this, use the `opentelemetry.meterBuilder(instrumentationScopeName)` method to create a new meter instance using the builder pattern.

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

```
        // OpenTelemetry Meter
        Meter meter = openTelemetry
                .meterBuilder("csk_apm_demo_controller")
                .setInstrumentationVersion("1.0.0")
                .build();
```

Alternatively, you can use the `opentelemetry.getMeter(instrumentationScopeName)` method to get or create a meter based on just the instrument scope name.

```
// Get or create a named meter instance by name only
Meter meter = openTelemetry.getMeter("csk_apm_demo_controller");
```

Now that you have meters initialized, you can create metric instruments.

Metric Instruments

In OpenTelemetry, measurements are captured by metric instruments, which are defined by:

- Name
- Kind
- Unit (optional)
- Description (optional)

The name, unit, and description are specified by the developer, or, defined via semantic conventions for common metrics like requests and processes.

Metric instruments can be one of the following kinds:

- **Counter**: A value that only increases over time. Think the odometer of a car, it will only go up.
- **Asynchronous Counter**: Similar to a Counter but collected once per export. Suitable for aggregated values rather than continuous increments.
- **UpDownCounter**: A value that can increase or decrease over time, such as a queue length.
- **Asynchronous UpDownCounter**: Similar to an UpDownCounter but collected once per export. Useful for aggregated values, like current queue size.
- **Gauge**: Measures a current value at the time it is read. Think the fuel gauge of a car. Gauges are asynchronous.

- **Histogram**: Aggregates values on the client side, useful for statistics like request latencies (e.g., how many requests take less than 1 second).

For more on synchronous and asynchronous instruments, and which kind is best suited for your use case, see Supplementary Guidelines.

Adding a Counter

Here is an example of how we added a LongCounter to our `csk-apm-demo` project: First, import the necessary library:

```
import io.opentelemetry.api.metrics.LongCounter;
```

Next, we declared our counters:

```
private final LongCounter getCounter;
private final LongCounter putCounter;
```

Then, we activated the counters using the meter already initialized in the previous section:

```
// Custom getCounter
this.getCounter = meter.counterBuilder("custom_getStoredValue_requests"
        .setDescription("Counts number of getStoredValue requests")
        .setUnit("requests")
        .build();

// Custom putCounter
this.putCounter = meter.counterBuilder("custom_putStoredValue_requests"
        .setDescription("Counts number of putStoredValue requests")
        .setUnit("requests")
        .build();
```

Increasing counter

It is very easy to increase our counters:

```
getCounter.add(1, attributes);

putCounter.add(1, attributes);
```

Notice that our counter is not just increasing its value, it also has **attributes** being passed.

Attributes

Now that we have a counter, or any other metric instrument for that matter, we want to give it attributes. **Metrics** represent quantitative data to be measured, while **attributes** provide context to those metrics, helping to organize and filter the results.

- **Metrics** in OpenTelemetry are used to capture measurements and are usually quantitative data points, like counters or gauges.
- **Attributes** are key-value pairs given to metrics to provide additional context, helping to 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’.

Attribute Definition

An Attribute is a key-value pair with the following properties:

- **Attribute 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 754-1985), or signed 64-bit integer.
    - An array of primitive type values. The array MUST be homogeneous and MUST NOT contain values of different types.

For more detailed information, see the Attribute documentation.

Adding Attributes and Increasing Counter

The OpenTelemetry SDK provides several methods to add attributes to counters, allowing you to capture and report additional context about the metrics.

First, import the necessary libraries. This is how we did it:

```
import io.opentelemetry.api.common.AttributeKey;
import io.opentelemetry.api.common.Attributes;
```

Example: Adding Attributes using `Attributes.of()`

To add attributes to a counter, we can use the `Attributes.of()` method. Here's how you can do it, as demonstrated in our example for the `putCounter`:

```
// OpenTelemetry: Increment putCounter
Attributes attributes = Attributes.of(
        AttributeKey.stringKey("Path"), "/api/v1/storedvalue/" + stored
        AttributeKey.stringKey("Method"), "PUT");
putCounter.add(1, attributes);
```

In this example, we set two keys for our attributes: `Path` and `Method`.

- `Path`: The URI path of the API call, dynamically including the `storedValueKey`.
- `Method`: The HTTP method used, in this case, "PUT".

help me improve the following Example section, to make it clear and well written, include the heading

Example: Adding Attributes using `Attributes.builder()`

Another handy method we can use is `Attributes.builder()` in our demo we first created a base set of attributes:

```
// OpenTelemetry: base attributes of getCounter
Attributes attributesBase = Attributes.builder()
        .put("Path", "/api/v1/storedvalue/" + storedValueKey)
        .put("Method", "GET")
        .build();
```

Next, we checked if the request was allowed and increased our `getCounter` with the appropriate attributes based on the return value:

```
return keyValueStore
        .getValue(storedValueKey)
        .map(value -> {
            Attributes successAttributes = Attributes.builder()
```

```
                    .putAll(attributesBase)
                    .put("HTTPStatus", "200")
                    .build();
            // Increment getCounter with successAttributes
            getCounter.add(1, successAttributes);
            return new StoredValueResponse(value); // If value was foun
        })
        .switchIfEmpty(Mono.defer(() -> {
            Attributes notFoundAttributes = Attributes.builder()
                    .putAll(attributesBase)
                    .put("HTTPStatus", "404")
                    .build();
            // Increment getCounter with notFoundAttributes
            getCounter.add(1, notFoundAttributes);
            return Mono.error(new ValueNotFoundException(storedValueKey
        }));
```

Notice that in this section of the code:

```
Attributes successAttributes = Attributes.builder()
        .putAll(attributesBase)
        .put("HTTPStatus", "200")
        .build();
// Increment getCounter with successAttributes
getCounter.add(1, successAttributes);
```

We use the `attributesBase` and add the attribute `HTTPStatus` with a value of `200` because the request was successful and valid.

While in this case:

```
Attributes notFoundAttributes = Attributes.builder()
        .putAll(attributesBase)
        .put("HTTPStatus", "404")
```

```
        .build();
// Increment getCounter with notFoundAttributes
getCounter.add(1, notFoundAttributes);
```

We return a `404` because the requested value was not found.

In both cases we increased our counter with the respective attributes:

```
getCounter.add(1, successAttributes);

getCounter.add(1, notFoundAttributes);
```

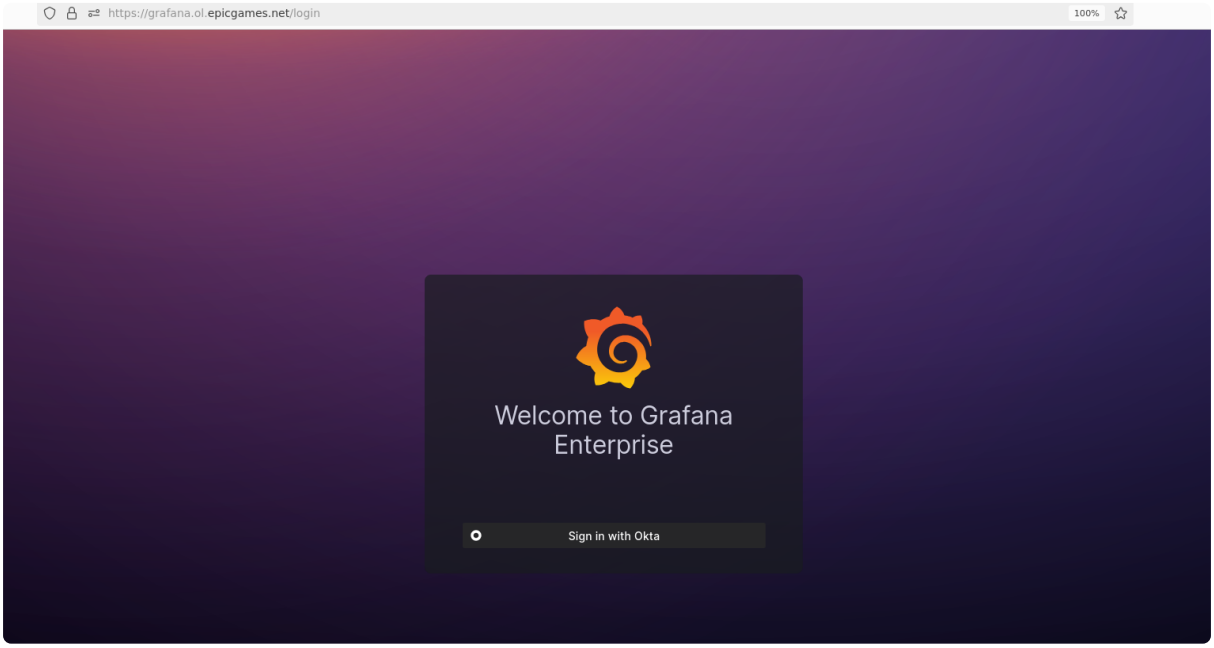
Visualizing Metrics with Grafana

After creating OpenTelemetry 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" and requesting either the "Grafana Epic editor" or "Grafana Epic viewer" role.

Logging into Grafana



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

OTEL Unified Dashboards

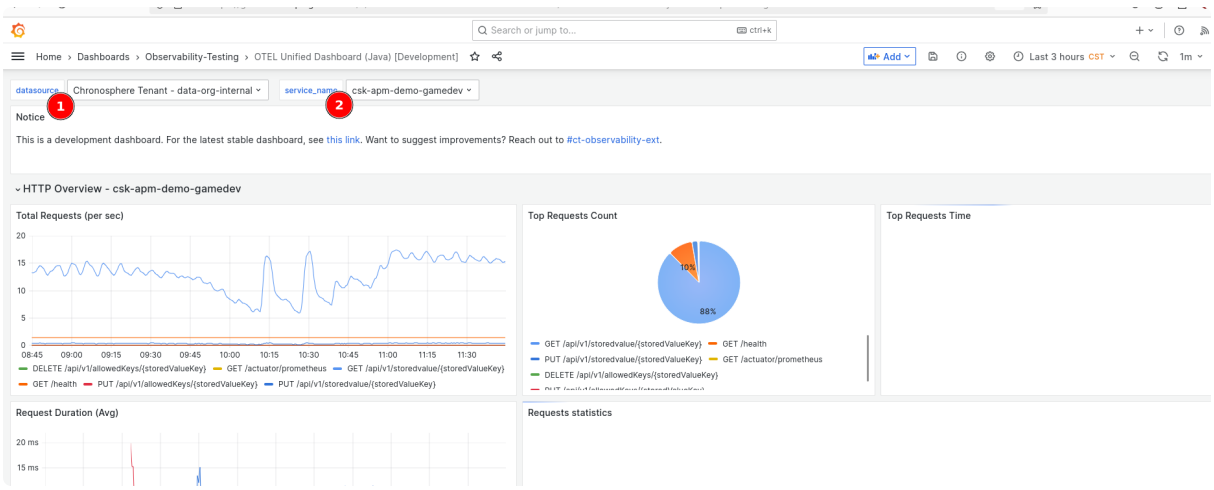
Once you have instrumented your application, you can use our pre-made Grafana dashboards to view general metrics about your application's performance.

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:

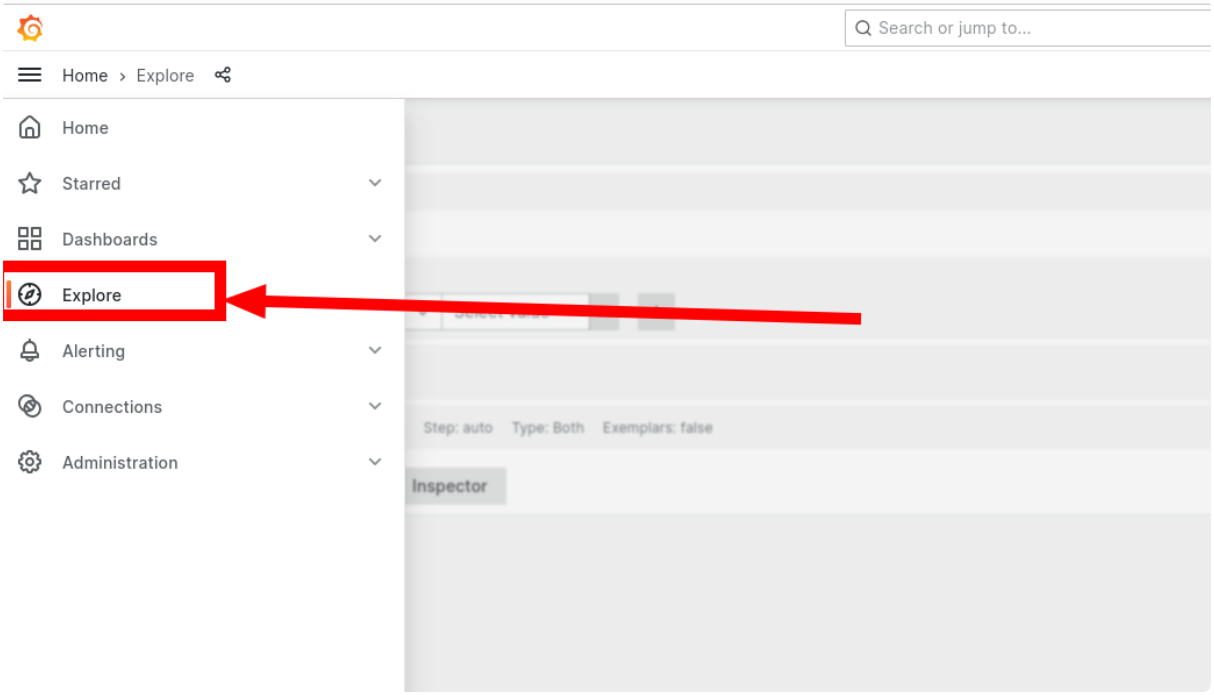
1. Datasource
2. Service name.



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

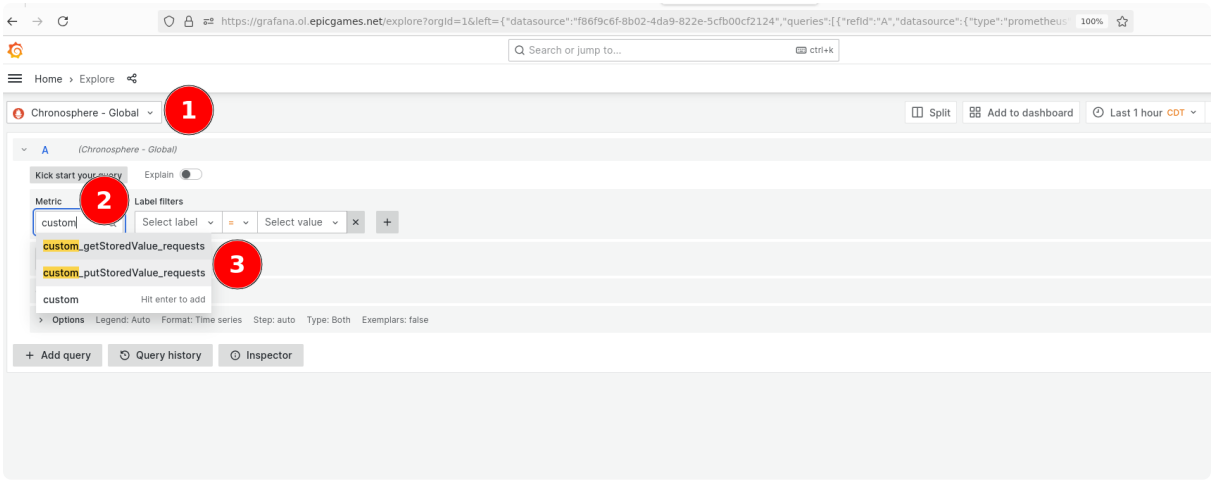
Before creating a dashboard, explore your data to verify it and construct your queries. Use Grafana's Explore feature, which simplifies the interface and lets you focus solely on querying your data.



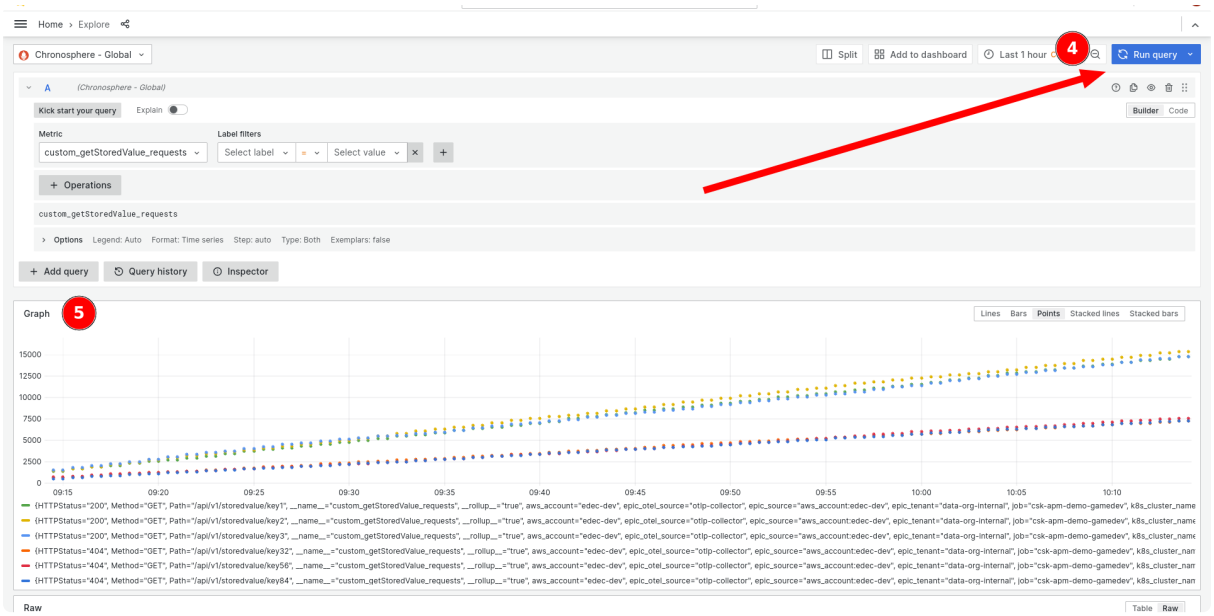
1. Navigate to "Home > Explore."
2. Learn more about the Explore feature [in the official docs].

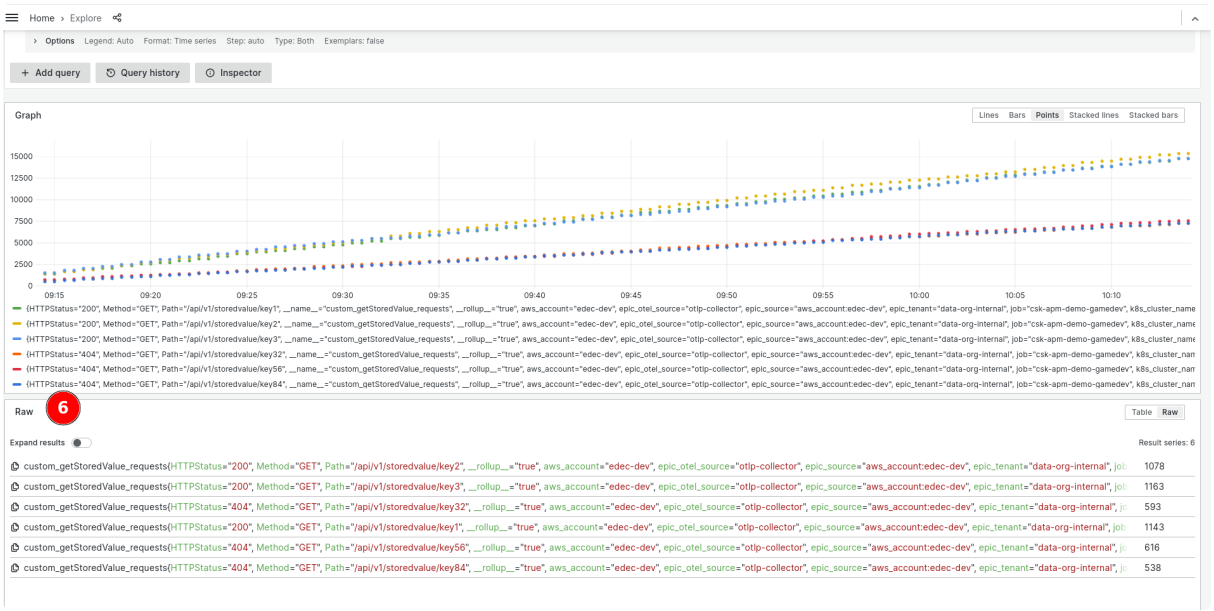
Finding and Exploring Your Data

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.



1. Click on the data sources option and select "Chronosphere - Global."
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: `custom_getStoredValue_requests`. Select the one you want 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.





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.