Red Hat OpenShift Serverless 1.30

Observability

Observability features including administrator and developer metrics, cluster logging, and tracing.
Observability features including administrator and developer metrics, cluster logging, and tracing.
Abstract

This document provides details on how to monitor the performance of Knative services. It also details how to use OpenShift Logging and OpenShift distributed tracing with OpenShift Serverless.
CHAPTER 1. ADMINISTRATOR METRICS

1.1. SERVERLESS ADMINISTRATOR METRICS

Metrics enable cluster administrators to monitor how OpenShift Serverless cluster components and workloads are performing.

You can view different metrics for OpenShift Serverless by navigating to Dashboards in the web console Administrator perspective.

1.1.1. Prerequisites

- See the OpenShift Container Platform documentation on Managing metrics for information about enabling metrics for your cluster.

- You have access to an account with cluster administrator access (or dedicated administrator access for OpenShift Dedicated or Red Hat OpenShift Service on AWS).

- You have access to the Administrator perspective in the web console.

![WARNING]

If Service Mesh is enabled with mTLS, metrics for Knative Serving are disabled by default because Service Mesh prevents Prometheus from scraping metrics.

For information about resolving this issue, see Enabling Knative Serving metrics when using Service Mesh with mTLS.

Scraping the metrics does not affect autoscaling of a Knative service, because scraping requests do not go through the activator. Consequently, no scraping takes place if no pods are running.

1.2. SERVERLESS CONTROLLER METRICS

The following metrics are emitted by any component that implements a controller logic. These metrics show details about reconciliation operations and the work queue behavior upon which reconciliation requests are added to the work queue.

<table>
<thead>
<tr>
<th>Metric name</th>
<th>Description</th>
<th>Type</th>
<th>Tags</th>
<th>Unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>work_queue_depth</td>
<td>The depth of the work queue.</td>
<td>Gauge</td>
<td>reconciler</td>
<td>Integer (no units)</td>
</tr>
<tr>
<td>reconcile_count</td>
<td>The number of reconcile operations.</td>
<td>Counter</td>
<td>reconciler, success</td>
<td>Integer (no units)</td>
</tr>
</tbody>
</table>
### 1.3. WEBHOOK METRICS

Webhook metrics report useful information about operations. For example, if a large number of operations fail, this might indicate an issue with a user-created resource.

<table>
<thead>
<tr>
<th>Metric name</th>
<th>Description</th>
<th>Type</th>
<th>Tags</th>
<th>Unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>reconcile_latency</td>
<td>The latency of reconcile operations.</td>
<td>Histogram</td>
<td>reconciler,</td>
<td>Milliseconds</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>success</td>
<td></td>
</tr>
<tr>
<td>workqueue_adds_total</td>
<td>The total number of add actions handled by the work queue.</td>
<td>Counter</td>
<td>name</td>
<td>Integer (no units)</td>
</tr>
<tr>
<td>workqueue_queue_latency_seconds</td>
<td>The length of time an item stays in the work queue before being requested.</td>
<td>Histogram</td>
<td>name</td>
<td>Seconds</td>
</tr>
<tr>
<td>workqueue_retries_total</td>
<td>The total number of retries that have been handled by the work queue.</td>
<td>Counter</td>
<td>name</td>
<td>Integer (no units)</td>
</tr>
<tr>
<td>workqueue_work_duration_seconds</td>
<td>The length of time it takes to process and item from the work queue.</td>
<td>Histogram</td>
<td>name</td>
<td>Seconds</td>
</tr>
<tr>
<td>workqueue_unfinished_work_seconds</td>
<td>The length of time that outstanding work queue items have been in progress.</td>
<td>Histogram</td>
<td>name</td>
<td>Seconds</td>
</tr>
<tr>
<td>workqueue_longest_running_processor_seconds</td>
<td>The length of time that the longest outstanding work queue items has been in progress.</td>
<td>Histogram</td>
<td>name</td>
<td>Seconds</td>
</tr>
</tbody>
</table>
### 1.4. KNATIVE EVENTING METRICS

Cluster administrators can view the following metrics for Knative Eventing components.

By aggregating the metrics from HTTP code, events can be separated into two categories; successful events (2xx) and failed events (5xx).

#### 1.4.1. Broker ingress metrics

You can use the following metrics to debug the broker ingress, see how it is performing, and see which events are being dispatched by the ingress component.

<table>
<thead>
<tr>
<th>Metric name</th>
<th>Description</th>
<th>Type</th>
<th>Tags</th>
<th>Unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>request_count</td>
<td>The number of requests that are routed to the webhook.</td>
<td>Counter</td>
<td>admission_allowed, kind_group, kind_kind, kind_version, request_operation, resource_group, resource_namespace, resource_resource, resource_version</td>
<td>Integer (no units)</td>
</tr>
<tr>
<td>request_latencies</td>
<td>The response time for a webhook request.</td>
<td>Histogram</td>
<td>admission_allowed, kind_group, kind_kind, kind_version, request_operation, resource_group, resource_namespace, resource_resource, resource_version</td>
<td>Milliseconds</td>
</tr>
</tbody>
</table>

---

CHAPTER 1. ADMINISTRATOR METRICS
### 1.4.2. Broker filter metrics

You can use the following metrics to debug broker filters, see how they are performing, and see which events are being dispatched by the filters. You can also measure the latency of the filtering action on an event.

<table>
<thead>
<tr>
<th>Metric name</th>
<th>Description</th>
<th>Type</th>
<th>Tags</th>
<th>Unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>event_count</td>
<td>Number of events received by a broker.</td>
<td>Counter</td>
<td>broker_name, event_type, namespace_name, response_code, response_code_class, unique_name</td>
<td>Integer (no units)</td>
</tr>
<tr>
<td>event_dispatch_latencies</td>
<td>The time taken to dispatch an event to a channel.</td>
<td>Histogram</td>
<td>broker_name, event_type, namespace_name, response_code, response_code_class, unique_name</td>
<td>Milliseconds</td>
</tr>
</tbody>
</table>

- **event_count**: Number of events received by a broker.
  - Type: Counter
  - Tags: broker_name, event_type, namespace_name, response_code, response_code_class, unique_name
  - Unit: Integer (no units)

- **event_dispatch_latencies**: The time taken to dispatch an event to a channel.
  - Type: Histogram
  - Tags: broker_name, event_type, namespace_name, response_code, response_code_class, unique_name
  - Unit: Milliseconds
### 1.4.3. InMemoryChannel dispatcher metrics

You can use the following metrics to debug InMemoryChannel channels, see how they are performing, and see which events are being dispatched by the channels.

<table>
<thead>
<tr>
<th>Metric name</th>
<th>Description</th>
<th>Type</th>
<th>Tags</th>
<th>Unit</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>event_processing_latencies</strong></td>
<td>The time it takes to process an event before it is dispatched to a trigger subscriber.</td>
<td>Histogram</td>
<td>broker_name, container_name, filter_type, namespace_name, trigger_name, unique_name</td>
<td>Milliseconds</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Metric name</th>
<th>Description</th>
<th>Type</th>
<th>Tags</th>
<th>Unit</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>event_count</strong></td>
<td>Number of events dispatched by InMemoryChannel channels.</td>
<td>Counter</td>
<td>broker_name, container_name, filter_type, namespace_name, response_code, response_code_class, trigger_name, unique_name</td>
<td>Integer (no units)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Metric name</th>
<th>Description</th>
<th>Type</th>
<th>Tags</th>
<th>Unit</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>event_dispatch_latencies</strong></td>
<td>The time taken to dispatch an event from an InMemoryChannel channel.</td>
<td>Histogram</td>
<td>broker_name, container_name, filter_type, namespace_name, response_code, response_code_class, trigger_name, unique_name</td>
<td>Milliseconds</td>
</tr>
</tbody>
</table>

### 1.4.4. Event source metrics

You can use the following metrics to verify that events have been delivered from the event source to the connected event sink.

<table>
<thead>
<tr>
<th>Metric name</th>
<th>Description</th>
<th>Type</th>
<th>Tags</th>
<th>Unit</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>event_count</strong></td>
<td>Number of events dispatched by InMemoryChannel channels.</td>
<td>Counter</td>
<td>broker_name, container_name, filter_type, namespace_name, response_code, response_code_class, trigger_name, unique_name</td>
<td>Integer (no units)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Metric name</th>
<th>Description</th>
<th>Type</th>
<th>Tags</th>
<th>Unit</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>event_dispatch_latencies</strong></td>
<td>The time taken to dispatch an event from an InMemoryChannel channel.</td>
<td>Histogram</td>
<td>broker_name, container_name, filter_type, namespace_name, response_code, response_code_class, trigger_name, unique_name</td>
<td>Milliseconds</td>
</tr>
<tr>
<td>Metric name</td>
<td>Description</td>
<td>Type</td>
<td>Tags</td>
<td>Unit</td>
</tr>
<tr>
<td>------------------</td>
<td>------------------------------------------------------------------------------</td>
<td>-----------</td>
<td>----------------------------------------------------------------------</td>
<td>-----------------------</td>
</tr>
<tr>
<td>event_count</td>
<td>Number of events sent by the event source.</td>
<td>Counter</td>
<td>broker_name, container_name, filter_type, namespace_name, response_code, response_code_class, trigger_name, unique_name</td>
<td>Integer (no units)</td>
</tr>
<tr>
<td>retry_event_count</td>
<td>Number of retried events sent by the event source after initially failing to be delivered.</td>
<td>Counter</td>
<td>event_source, event_type, name, namespace_name, resource_group, response_code, response_code_class, response_error, response_timeout</td>
<td>Integer (no units)</td>
</tr>
</tbody>
</table>

### 1.5. KNATIVE SERVING METRICS

Cluster administrators can view the following metrics for Knative Serving components.

#### 1.5.1. Activator metrics

You can use the following metrics to understand how applications respond when traffic passes through the activator.

<table>
<thead>
<tr>
<th>Metric name</th>
<th>Description</th>
<th>Type</th>
<th>Tags</th>
<th>Unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>request_concurrency</td>
<td>The number of concurrent requests that are routed to the activator, or average concurrency over a reporting period.</td>
<td>Gauge</td>
<td>configuration_name, container_name, namespace_name, pod_name, revision_name, service_name</td>
<td>Integer (no units)</td>
</tr>
</tbody>
</table>
### 1.5.2. Autoscaler metrics

The autoscaler component exposes a number of metrics related to autoscaler behavior for each revision. For example, at any given time, you can monitor the targeted number of pods the autoscaler tries to allocate for a service, the average number of requests per second during the stable window, or whether the autoscaler is in panic mode if you are using the Knative pod autoscaler (KPA).

<table>
<thead>
<tr>
<th>Metric name</th>
<th>Description</th>
<th>Type</th>
<th>Tags</th>
<th>Unit</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>desired_pods</strong></td>
<td>The number of pods the autoscaler tries to allocate for a service.</td>
<td>Gauge</td>
<td><strong>configuration_name</strong>, <strong>namespace_name</strong>, <strong>revision_name</strong>, <strong>service_name</strong></td>
<td>Integer (no units)</td>
</tr>
<tr>
<td><strong>excess_burst_capacity</strong></td>
<td>The excess burst capacity served over the stable window.</td>
<td>Gauge</td>
<td><strong>configuration_name</strong>, <strong>namespace_name</strong>, <strong>revision_name</strong>, <strong>service_name</strong></td>
<td>Integer (no units)</td>
</tr>
<tr>
<td>Metric name</td>
<td>Description</td>
<td>Type</td>
<td>Tags</td>
<td>Unit</td>
</tr>
<tr>
<td>--------------------------------</td>
<td>------------------------------------------------------------------------------</td>
<td>----------</td>
<td>---------------------------------------------------</td>
<td>--------------------------</td>
</tr>
<tr>
<td><strong>stable_request_concurrency</strong></td>
<td>The average number of requests for each observed pod over the stable window.</td>
<td>Gauge</td>
<td>configuration_name, namespace_name, revision_name, service_name</td>
<td>Integer (no units)</td>
</tr>
<tr>
<td><strong>panic_request_concurrency</strong></td>
<td>The average number of requests for each observed pod over the panic window.</td>
<td>Gauge</td>
<td>configuration_name, namespace_name, revision_name, service_name</td>
<td>Integer (no units)</td>
</tr>
<tr>
<td><strong>target_concurrency_per_pod</strong></td>
<td>The number of concurrent requests that the autoscaler tries to send to each pod.</td>
<td>Gauge</td>
<td>configuration_name, namespace_name, revision_name, service_name</td>
<td>Integer (no units)</td>
</tr>
<tr>
<td><strong>stable_requests_per_second</strong></td>
<td>The average number of requests-per-second for each observed pod over the stable window.</td>
<td>Gauge</td>
<td>configuration_name, namespace_name, revision_name, service_name</td>
<td>Integer (no units)</td>
</tr>
<tr>
<td><strong>panic_requests_per_second</strong></td>
<td>The average number of requests-per-second for each observed pod over the panic window.</td>
<td>Gauge</td>
<td>configuration_name, namespace_name, revision_name, service_name</td>
<td>Integer (no units)</td>
</tr>
<tr>
<td><strong>target_requests_per_second</strong></td>
<td>The number of requests-per-second that the autoscaler targets for each pod.</td>
<td>Gauge</td>
<td>configuration_name, namespace_name, revision_name, service_name</td>
<td>Integer (no units)</td>
</tr>
<tr>
<td><strong>panic_mode</strong></td>
<td>This value is 1 if the autoscaler is in panic mode, or 0 if the autoscaler is not in panic mode.</td>
<td>Gauge</td>
<td>configuration_name, namespace_name, revision_name, service_name</td>
<td>Integer (no units)</td>
</tr>
<tr>
<td>Metric name</td>
<td>Description</td>
<td>Type</td>
<td>Tags</td>
<td>Unit</td>
</tr>
<tr>
<td>---------------------</td>
<td>--------------------------------------------------</td>
<td>-------</td>
<td>----------------------------------------------------------------------</td>
<td>-----------------------</td>
</tr>
<tr>
<td>requested_pods</td>
<td>The number of pods that the autoscaler has requested from the Kubernetes cluster.</td>
<td>Gauge</td>
<td>configuration_name, namespace_name, revision_name, service_name</td>
<td>Integer (no units)</td>
</tr>
<tr>
<td>actual_pods</td>
<td>The number of pods that are allocated and currently have a ready state.</td>
<td>Gauge</td>
<td>configuration_name, namespace_name, revision_name, service_name</td>
<td>Integer (no units)</td>
</tr>
<tr>
<td>not_ready_pods</td>
<td>The number of pods that have a not ready state.</td>
<td>Gauge</td>
<td>configuration_name, namespace_name, revision_name, service_name</td>
<td>Integer (no units)</td>
</tr>
<tr>
<td>pending_pods</td>
<td>The number of pods that are currently pending.</td>
<td>Gauge</td>
<td>configuration_name, namespace_name, revision_name, service_name</td>
<td>Integer (no units)</td>
</tr>
<tr>
<td>terminating_pods</td>
<td>The number of pods that are currently terminating.</td>
<td>Gauge</td>
<td>configuration_name, namespace_name, revision_name, service_name</td>
<td>Integer (no units)</td>
</tr>
</tbody>
</table>

1.5.3. Go runtime metrics

Each Knative Serving control plane process emits a number of Go runtime memory statistics (MemStats).

**NOTE**

The name tag for each metric is an empty tag.
<table>
<thead>
<tr>
<th>Metric name</th>
<th>Description</th>
<th>Type</th>
<th>Tags</th>
<th>Unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>go_alloc</td>
<td>The number of bytes of allocated heap objects. This metric is the same as heap_alloc.</td>
<td>Gauge</td>
<td>name</td>
<td>Integer (no units)</td>
</tr>
<tr>
<td>go_total_alloc</td>
<td>The cumulative bytes allocated for heap objects.</td>
<td>Gauge</td>
<td>name</td>
<td>Integer (no units)</td>
</tr>
<tr>
<td>go_sys</td>
<td>The total bytes of memory obtained from the operating system.</td>
<td>Gauge</td>
<td>name</td>
<td>Integer (no units)</td>
</tr>
<tr>
<td>go_lookups</td>
<td>The number of pointer lookups performed by the runtime.</td>
<td>Gauge</td>
<td>name</td>
<td>Integer (no units)</td>
</tr>
<tr>
<td>go_mallocs</td>
<td>The cumulative count of heap objects allocated.</td>
<td>Gauge</td>
<td>name</td>
<td>Integer (no units)</td>
</tr>
<tr>
<td>go_frees</td>
<td>The cumulative count of heap objects that have been freed.</td>
<td>Gauge</td>
<td>name</td>
<td>Integer (no units)</td>
</tr>
<tr>
<td>go_heap_alloc</td>
<td>The number of bytes of allocated heap objects.</td>
<td>Gauge</td>
<td>name</td>
<td>Integer (no units)</td>
</tr>
<tr>
<td>go_heap_sys</td>
<td>The number of bytes of heap memory obtained from the operating system.</td>
<td>Gauge</td>
<td>name</td>
<td>Integer (no units)</td>
</tr>
<tr>
<td>go_heap_idle</td>
<td>The number of bytes in idle, unused spans.</td>
<td>Gauge</td>
<td>name</td>
<td>Integer (no units)</td>
</tr>
<tr>
<td>go_heap_in_use</td>
<td>The number of bytes in spans that are currently in use.</td>
<td>Gauge</td>
<td>name</td>
<td>Integer (no units)</td>
</tr>
<tr>
<td>Metric name</td>
<td>Description</td>
<td>Type</td>
<td>Tags</td>
<td>Unit</td>
</tr>
<tr>
<td>-----------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>--------</td>
<td>------</td>
<td>------------------</td>
</tr>
<tr>
<td>go_heap_released</td>
<td>The number of bytes of physical memory returned to the operating system.</td>
<td>Gauge</td>
<td>name</td>
<td>Integer (no units)</td>
</tr>
<tr>
<td>go_heap_objects</td>
<td>The number of allocated heap objects.</td>
<td>Gauge</td>
<td>name</td>
<td>Integer (no units)</td>
</tr>
<tr>
<td>go_stack_in_use</td>
<td>The number of bytes in stack spans that are currently in use.</td>
<td>Gauge</td>
<td>name</td>
<td>Integer (no units)</td>
</tr>
<tr>
<td>go_stack_sys</td>
<td>The number of bytes of stack memory obtained from the operating system.</td>
<td>Gauge</td>
<td>name</td>
<td>Integer (no units)</td>
</tr>
<tr>
<td>go_mspan_in_use</td>
<td>The number of bytes of allocated mspan structures.</td>
<td>Gauge</td>
<td>name</td>
<td>Integer (no units)</td>
</tr>
<tr>
<td>go_mspan_sys</td>
<td>The number of bytes of memory obtained from the operating system for mspan structures.</td>
<td>Gauge</td>
<td>name</td>
<td>Integer (no units)</td>
</tr>
<tr>
<td>go_mcache_in_use</td>
<td>The number of bytes of allocated mcache structures.</td>
<td>Gauge</td>
<td>name</td>
<td>Integer (no units)</td>
</tr>
<tr>
<td>go_mcache_sys</td>
<td>The number of bytes of memory obtained from the operating system for mcache structures.</td>
<td>Gauge</td>
<td>name</td>
<td>Integer (no units)</td>
</tr>
<tr>
<td>go_bucket_hash_sys</td>
<td>The number of bytes of memory in profiling bucket hash tables.</td>
<td>Gauge</td>
<td>name</td>
<td>Integer (no units)</td>
</tr>
<tr>
<td>Metric name</td>
<td>Description</td>
<td>Type</td>
<td>Tags</td>
<td>Unit</td>
</tr>
<tr>
<td>---------------------</td>
<td>------------------------------------------------------------------------------</td>
<td>---------</td>
<td>------</td>
<td>----------------</td>
</tr>
<tr>
<td>go_gc_sys</td>
<td>The number of bytes of memory in garbage collection metadata.</td>
<td>Gauge</td>
<td>name</td>
<td>Integer (no units)</td>
</tr>
<tr>
<td>go_other_sys</td>
<td>The number of bytes of memory in miscellaneous, off-heap runtime allocations.</td>
<td>Gauge</td>
<td>name</td>
<td>Integer (no units)</td>
</tr>
<tr>
<td>go_next_gc</td>
<td>The target heap size of the next garbage collection cycle.</td>
<td>Gauge</td>
<td>name</td>
<td>Integer (no units)</td>
</tr>
<tr>
<td>go_last_gc</td>
<td>The time that the last garbage collection was completed in Epoch or Unix time.</td>
<td>Gauge</td>
<td>name</td>
<td>Nanoseconds</td>
</tr>
<tr>
<td>go_total_gc_pause_ns</td>
<td>The cumulative time in garbage collection stop-the-world pauses since the program started.</td>
<td>Gauge</td>
<td>name</td>
<td>Nanoseconds</td>
</tr>
<tr>
<td>go_num_gc</td>
<td>The number of completed garbage collection cycles.</td>
<td>Gauge</td>
<td>name</td>
<td>Integer (no units)</td>
</tr>
<tr>
<td>go_num_forced_gc</td>
<td>The number of garbage collection cycles that were forced due to an application calling the garbage collection function.</td>
<td>Gauge</td>
<td>name</td>
<td>Integer (no units)</td>
</tr>
</tbody>
</table>
The fraction of the available CPU time of the program that has been used by the garbage collector since the program started.

<table>
<thead>
<tr>
<th>Metric name</th>
<th>Description</th>
<th>Type</th>
<th>Tags</th>
<th>Unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>go_gc_cpu_fraction</td>
<td>The fraction of the available CPU time of the program that has been used by the garbage collector since the program started.</td>
<td>Gauge</td>
<td>name</td>
<td>Integer (no units)</td>
</tr>
</tbody>
</table>
CHAPTER 2. DEVELOPER METRICS

2.1. SERVERLESS DEVELOPER METRICS OVERVIEW

Metrics enable developers to monitor how Knative services are performing. You can use the OpenShift Container Platform monitoring stack to record and view health checks and metrics for your Knative services.

You can view different metrics for OpenShift Serverless by navigating to Dashboards in the web console Developer perspective.

WARNING

If Service Mesh is enabled with mTLS, metrics for Knative Serving are disabled by default because Service Mesh prevents Prometheus from scraping metrics.

For information about resolving this issue, see Enabling Knative Serving metrics when using Service Mesh with mTLS.

Scraping the metrics does not affect autoscaling of a Knative service, because scraping requests do not go through the activator. Consequently, no scraping takes place if no pods are running.

2.1.1. Additional resources for OpenShift Container Platform

- Monitoring overview
- Enabling monitoring for user-defined projects
- Specifying how a service is monitored

2.2. KNATIVE SERVICE METRICS EXPOSED BY DEFAULT

Table 2.1. Metrics exposed by default for each Knative service on port 9090

<table>
<thead>
<tr>
<th>Metric name, unit, and type</th>
<th>Description</th>
<th>Metric tags</th>
</tr>
</thead>
<tbody>
<tr>
<td>queue_requests_per_second</td>
<td>Number of requests per second that hit the queue proxy.</td>
<td>destination_configuration=&quot;event-display&quot;, destination_namespace=&quot;pingsource1&quot;, destination_pod=&quot;event-display-00001-deployment-6b455479cb-75p6w&quot;, destination_revision=&quot;event-display-00001&quot;</td>
</tr>
<tr>
<td>Metric unit: dimensionless</td>
<td>Formula: (\text{stats}\text{.RequestCount} / \text{r.reportingPeriodSeconds})</td>
<td></td>
</tr>
<tr>
<td>Metric type: gauge</td>
<td>stats.RequestCount is calculated directly from the networking pkg stats for the given reporting duration.</td>
<td></td>
</tr>
<tr>
<td>Metric name, unit, and type</td>
<td>Description</td>
<td>Metric tags</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>-------------</td>
<td>-------------</td>
</tr>
<tr>
<td>queue_proxied_operations_per_second</td>
<td>Number of proxied requests per second.</td>
<td>destination_configuration=&quot;event-display&quot;, destination_namespace=&quot;pingsource1&quot;, destination_pod=&quot;event-display-00001-deployment-6b455479cb-75p6w&quot;, destination_revision=&quot;event-display-00001&quot;</td>
</tr>
<tr>
<td>Metric unit: dimensionless</td>
<td>Formula: ( \frac{\text{stats.ProxiedRequestCount}}{\text{r.reportingPeriodSeconds}} )</td>
<td></td>
</tr>
<tr>
<td>Metric type: gauge</td>
<td>stats.ProxiedRequestCount is calculated directly from the networking pkg stats for the given reporting duration.</td>
<td></td>
</tr>
<tr>
<td>queue_average_concurrent_requests</td>
<td>Number of requests currently being handled by this pod.</td>
<td></td>
</tr>
<tr>
<td>Metric unit: dimensionless</td>
<td>Average concurrency is calculated at the networking pkg side as follows:</td>
<td></td>
</tr>
<tr>
<td>Metric type: gauge</td>
<td>- When a req change happens, the time delta between changes is calculated. Based on the result, the current concurrency number over delta is computed and added to the current computed concurrency. Additionally, a sum of the deltas is kept. Current concurrency over delta is computed as follows:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>[ \text{global_concurrency} \times \delta ]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- Each time a reporting is done, the sum and current computed concurrency are reset.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- When reporting the average concurrency the current computed concurrency is divided by the sum of deltas.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- When a new request comes in, the global concurrency counter is increased. When a request is completed, the counter is decreased.</td>
<td></td>
</tr>
<tr>
<td>Metric name, unit, and type</td>
<td>Description</td>
<td>Metric tags</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>-------------</td>
<td>-------------</td>
</tr>
<tr>
<td><code>queue_average_proxied_concurrent_requests</code></td>
<td>Number of proxied requests currently being handled by this pod: <code>stats.AverageProxiedConcurrency</code></td>
<td>destination_configuration=&quot;event-display&quot;, destination_namespace=&quot;pingsource1&quot;, destination_pod=&quot;event-display-00001-deployment-6b455479cb-75p6w&quot;, destination_revision=&quot;event-display-00001&quot;</td>
</tr>
<tr>
<td>process_uptime</td>
<td>The number of seconds that the process has been up.</td>
<td>destination_configuration=&quot;event-display&quot;, destination_namespace=&quot;pingsource1&quot;, destination_pod=&quot;event-display-00001-deployment-6b455479cb-75p6w&quot;, destination_revision=&quot;event-display-00001&quot;</td>
</tr>
</tbody>
</table>

Table 2.2. Metrics exposed by default for each Knative service on port 9091

<table>
<thead>
<tr>
<th>Metric name, unit, and type</th>
<th>Description</th>
<th>Metric tags</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>request_count</code></td>
<td>The number of requests that are routed to <code>queue-proxy</code>.</td>
<td>configuration_name=&quot;event-display&quot;, container_name=&quot;queue-proxy&quot;, namespace_name=&quot;apiserversource1&quot;, pod_name=&quot;event-display-00001-deployment-658fd4f9cf-qcnr5&quot;, response_code=&quot;200&quot;, response_code_class=&quot;2xx&quot;, revision_name=&quot;event-display-00001&quot;, service_name=&quot;event-display&quot;</td>
</tr>
<tr>
<td><code>request_latencies</code></td>
<td>The response time in milliseconds.</td>
<td>configuration_name=&quot;event-display&quot;, container_name=&quot;queue-proxy&quot;, namespace_name=&quot;apiserversource1&quot;, pod_name=&quot;event-display-00001-deployment-658fd4f9cf-qcnr5&quot;, response_code=&quot;200&quot;, response_code_class=&quot;2xx&quot;, revision_name=&quot;event-display-00001&quot;, service_name=&quot;event-display&quot;</td>
</tr>
<tr>
<td>Metric name, unit, and type</td>
<td>Description</td>
<td>Metric tags</td>
</tr>
<tr>
<td>----------------------------</td>
<td>-------------</td>
<td>-------------</td>
</tr>
<tr>
<td>app_request_count</td>
<td>The number of requests that are routed to user-container.</td>
<td>configuration_name=&quot;event-display&quot;, container_name=&quot;queue-proxy&quot;, namespace_name=&quot;apiserversource1&quot;, pod_name=&quot;event-display-00001-deployment-658fd4f9cf-qcnr5&quot;, response_code=&quot;200&quot;, response_code_class=&quot;2xx&quot;, revision_name=&quot;event-display-00001&quot;, service_name=&quot;event-display&quot;</td>
</tr>
<tr>
<td>app_request_latencies</td>
<td>The response time in milliseconds.</td>
<td>configuration_name=&quot;event-display&quot;, container_name=&quot;queue-proxy&quot;, namespace_name=&quot;apiserversource1&quot;, pod_name=&quot;event-display-00001-deployment-658fd4f9cf-qcnr5&quot;, response_code=&quot;200&quot;, response_code_class=&quot;2xx&quot;, revision_name=&quot;event-display-00001&quot;, service_name=&quot;event-display&quot;</td>
</tr>
<tr>
<td>queue_depth</td>
<td>The current number of items in the serving and waiting queue, or not reported if unlimited concurrency. breaker.inFlight is used.</td>
<td>configuration_name=&quot;event-display&quot;, container_name=&quot;queue-proxy&quot;, namespace_name=&quot;apiserversource1&quot;, pod_name=&quot;event-display-00001-deployment-658fd4f9cf-qcnr5&quot;, response_code=&quot;200&quot;, response_code_class=&quot;2xx&quot;, revision_name=&quot;event-display-00001&quot;, service_name=&quot;event-display&quot;</td>
</tr>
</tbody>
</table>

### 2.3. KNATIVE SERVICE WITH CUSTOM APPLICATION METRICS

You can extend the set of metrics exported by a Knative service. The exact implementation depends on your application and the language used.

The following listing implements a sample Go application that exports the count of processed events custom metric.

```go
package main

import (  
    "fmt"
)
```
```go
var 
    opsProcessed = promauto.NewCounter(prometheus.CounterOpts{
        Name: "myapp_processed_ops_total",
        Help: "The total number of processed events",
    })

func handler(w http.ResponseWriter, r *http.Request) {
    log.Print("helloworld: received a request")
    target := os.Getenv("TARGET")
    if target == "" {
        target = "World"
    }
    fmt.Fprintf(w, "Hello %s!
", target)
    opsProcessed.Inc()  
}

func main() {
    log.Print("helloworld: starting server...")

    port := os.Getenv("PORT")
    if port == "" {
        port = "8080"
    }

    http.HandleFunc("/", handler)

    // Separate server for metrics requests
    go func() {
        mux := http.NewServeMux()
        server := &http.Server{
            Addr: fmt.Sprintf(":%s", "9095"),
            Handler: mux,
        }
        mux.Handle("/metrics", promhttp.Handler())
        log.Printf("prometheus: listening on port %s", 9095)
        log.Fatal(server.ListenAndServe())
    }()

    // Use same port as normal requests for metrics
    //http.HandleFunc("/metrics", promhttp.Handler())
    log.Printf("helloworld: listening on port %s", port)
    log.Fatal(http.ListenAndServe(fmt.Sprintf(":%s", port), nil))
}
```
Including the Prometheus packages.

Defining the opsProcessed metric.

Incrementing the opsProcessed metric.

Configuring to use a separate server for metrics requests.

Configuring to use the same port as normal requests for metrics and the metrics subpath.

2.4. CONFIGURATION FOR SCRAPING CUSTOM METRICS

Custom metrics scraping is performed by an instance of Prometheus purposed for user workload monitoring. After you enable user workload monitoring and create the application, you need a configuration that defines how the monitoring stack will scrape the metrics.

The following sample configuration defines the ksvc for your application and configures the service monitor. The exact configuration depends on your application and how it exports the metrics.

```yaml
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  name: helloworld-go
spec:
  template:
    metadata:
      labels:
        app: helloworld-go
    spec:
      containers:
      - image: docker.io/skonto/helloworld-go:metrics
        resources:
          requests:
            cpu: "200m"
        env:
          - name: TARGET
            value: "Go Sample v1"
---
apiVersion: monitoring.coreos.com/v1
kind: ServiceMonitor
metadata:
  labels:
  name: helloworld-go-sm
spec:
  endpoints:
  - port: queue-proxy-metrics
    scheme: http
  - port: app-metrics
    scheme: http
  namespaceSelector: {}
  selector:
    matchLabels:
      name: helloworld-go-sm
```
2.5. EXAMINING METRICS OF A SERVICE

After you have configured the application to export the metrics and the monitoring stack to scrape them, you can examine the metrics in the web console.

Prerequisites

- You have logged in to the OpenShift Container Platform web console.
- You have installed the OpenShift Serverless Operator and Knative Serving.

Procedure

1. Optional: Run requests against your application that you will be able to see in the metrics:

   ```
   $ hello_route=$(oc get ksvc helloworld-go -n ns1 -o jsonpath='{.status.url}') && \
   curl $hello_route
   ```

   **Example output**

   ```
   Hello Go Sample v1!
   ```

2. In the web console, navigate to the **Observe → Metrics** interface.

3. In the input field, enter the query for the metric you want to observe, for example:

```yaml
apiVersion: v1
kind: Service
metadata:
  labels:
    name: helloworld-go-sm
  name: helloworld-go-sm
spec:
  ports:
  - name: queue-proxy-metrics
    port: 9091
    protocol: TCP
    targetPort: 9091
  - name: app-metrics
    port: 9095
    protocol: TCP
    targetPort: 9095
selector:
  serving.knative.dev/service: helloworld-go
type: ClusterIP
```
revision_app_request_count{namespace="ns1", job="helloworld-go-sm"}

Another example:

myapp_processed_ops_total{namespace="ns1", job="helloworld-go-sm"}

4. Observe the visualized metrics:

2.5.1. Queue proxy metrics

Each Knative service has a proxy container that proxies the connections to the application container. A number of metrics are reported for the queue proxy performance.

You can use the following metrics to measure if requests are queued at the proxy side and the actual delay in serving requests at the application side.

<table>
<thead>
<tr>
<th>Metric name</th>
<th>Description</th>
<th>Type</th>
<th>Tags</th>
<th>Unit</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Metric name</td>
<td>Description</td>
<td>Type</td>
<td>Tags</td>
<td>Unit</td>
</tr>
<tr>
<td>-------------------------</td>
<td>------------------------------------------------------------------------------</td>
<td>--------------</td>
<td>----------------------------------------------------------------------</td>
<td>--------------------</td>
</tr>
<tr>
<td>revision_request_count</td>
<td>The number of requests that are routed to queue-proxy pod.</td>
<td>Counter</td>
<td>configuration_name, container_name, namespace_name, pod_name, response_code, response_code_class, revision_name, service_name</td>
<td>Integer (no units)</td>
</tr>
<tr>
<td>revision_request_latencies</td>
<td>The response time of revision requests.</td>
<td>Histogram</td>
<td>configuration_name, container_name, namespace_name, pod_name, response_code, response_code_class, revision_name, service_name</td>
<td>Milliseconds</td>
</tr>
<tr>
<td>revision_app_request_count</td>
<td>The number of requests that are routed to the user-container pod.</td>
<td>Counter</td>
<td>configuration_name, container_name, namespace_name, pod_name, response_code, response_code_class, revision_name, service_name</td>
<td>Integer (no units)</td>
</tr>
<tr>
<td>revision_app_request_latencies</td>
<td>The response time of revision app requests.</td>
<td>Histogram</td>
<td>configuration_name, namespace_name, pod_name, response_code, response_code_class, revision_name, service_name</td>
<td>Milliseconds</td>
</tr>
</tbody>
</table>
### 2.6. DASHBOARD FOR SERVICE METRICS

You can examine the metrics using a dedicated dashboard that aggregates queue proxy metrics by namespace.

### 2.6.1. Examining metrics of a service in the dashboard

#### Prerequisites

- You have logged in to the OpenShift Container Platform web console.
- You have installed the OpenShift Serverless Operator and Knative Serving.

#### Procedure

1. In the web console, navigate to the **Observe → Metrics** interface.
2. Select the **Knative User Services (Queue Proxy metrics)** dashboard.
3. Select the **Namespace**, **Configuration**, and **Revision** that correspond to your application.
4. Observe the visualized metrics:

#### Metric Table

<table>
<thead>
<tr>
<th>Metric name</th>
<th>Description</th>
<th>Type</th>
<th>Tags</th>
<th>Unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>revision_queue_depth</td>
<td>The current number of items in the serving and waiting queues. This metric is not reported if unlimited concurrency is configured.</td>
<td>Gauge</td>
<td>configuration_name, event_display, container_name, namespace_name, pod_name, response_code_class, revision_name, service_name</td>
<td>Integer (no units)</td>
</tr>
</tbody>
</table>

---

**CHAPTER 2. DEVELOPER METRICS**
CHAPTER 3. CLUSTER LOGGING

3.1. USING OPENSIFT LOGGING WITH OPENSIFT SERVERLESS

3.1.1. About deploying the logging subsystem for Red Hat OpenShift

OpenShift Container Platform cluster administrators can deploy the logging subsystem using the OpenShift Container Platform web console or CLI to install the OpenShift Elasticsearch Operator and Red Hat OpenShift Logging Operator. When the Operators are installed, you create a `ClusterLogging` custom resource (CR) to schedule logging subsystem pods and other resources necessary to support the logging subsystem. The Operators are responsible for deploying, upgrading, and maintaining the logging subsystem.

The `ClusterLogging` CR defines a complete logging subsystem environment that includes all the components of the logging stack to collect, store and visualize logs. The Red Hat OpenShift Logging Operator watches the logging subsystem CR and adjusts the logging deployment accordingly.

Administrators and application developers can view the logs of the projects for which they have view access.

3.1.2. About deploying and configuring the logging subsystem for Red Hat OpenShift

The logging subsystem is designed to be used with the default configuration, which is tuned for small to medium sized OpenShift Container Platform clusters.

The installation instructions that follow include a sample `ClusterLogging` custom resource (CR), which you can use to create a logging subsystem instance and configure your logging subsystem environment.

If you want to use the default logging subsystem install, you can use the sample CR directly.

If you want to customize your deployment, make changes to the sample CR as needed. The following describes the configurations you can make when installing your OpenShift Logging instance or modify after installation. See the Configuring sections for more information on working with each component, including modifications you can make outside of the `ClusterLogging` custom resource.

3.1.2.1. Configuring and Tuning the logging subsystem

You can configure your logging subsystem by modifying the `ClusterLogging` custom resource deployed in the `openshift-logging` project.

You can modify any of the following components upon install or after install:

**Memory and CPU**

You can adjust both the CPU and memory limits for each component by modifying the `resources` block with valid memory and CPU values:

```yaml
spec:
  logStore:
    elasticsearch:
      resources:
        limits:
          cpu:
```
Elasticsearch storage

You can configure a persistent storage class and size for the Elasticsearch cluster using the `storageClass name` and `size` parameters. The Red Hat OpenShift Logging Operator creates a persistent volume claim (PVC) for each data node in the Elasticsearch cluster based on these parameters.

```yaml
spec:
  logStore:
    type: "elasticsearch"
    elasticsearch:
      nodeCount: 3
      storage:
        storageClassName: "gp2"
        size: "200G"
```

This example specifies each data node in the cluster will be bound to a PVC that requests "200G" of "gp2" storage. Each primary shard will be backed by a single replica.
NOTE

Omitting the `storage` block results in a deployment that includes ephemeral storage only.

```yaml
spec:
  logStore:
    type: "elasticsearch"
  elasticsearch:
    nodeCount: 3
    storage: {}
```

Elasticsearch replication policy

You can set the policy that defines how Elasticsearch shards are replicated across data nodes in the cluster:

- **FullRedundancy.** The shards for each index are fully replicated to every data node.
- **MultipleRedundancy.** The shards for each index are spread over half of the data nodes.
- **SingleRedundancy.** A single copy of each shard. Logs are always available and recoverable as long as at least two data nodes exist.
- **ZeroRedundancy.** No copies of any shards. Logs may be unavailable (or lost) in the event a node is down or fails.

3.1.2.2. Sample modified `ClusterLogging` custom resource

The following is an example of a `ClusterLogging` custom resource modified using the options previously described.

Sample modified `ClusterLogging` custom resource

```yaml
apiVersion: "logging.openshift.io/v1"
kind: "ClusterLogging"
metadata:
  name: "instance"
  namespace: "openshift-logging"
spec:
  managementState: "Managed"
  logStore:
    type: "elasticsearch"
  retentionPolicy:
    application:
      maxAge: 1d
    infra:
      maxAge: 7d
    audit:
      maxAge: 7d
  elasticsearch:
    nodeCount: 3
  resources:
    limits:
      cpu: 200m
      memory: 16Gi
```
3.2. FINDING LOGS FOR KNATIVE SERVING COMPONENTS

You can find the logs for Knative Serving components using the following procedure.

3.2.1. Using OpenShift Logging to find logs for Knative Serving components

Prerequisites

- Install the OpenShift CLI (`oc`).

Procedure

1. Get the Kibana route:

   ```
   $ oc -n openshift-logging get route kibana
   ```

2. Use the route’s URL to navigate to the Kibana dashboard and log in.

3. Check that the index is set to `.all`. If the index is not set to `.all`, only the OpenShift Container Platform system logs will be listed.

4. Filter the logs by using the `knative-serving` namespace. Enter `kubernetes.namespace_name:knative-serving` in the search box to filter results.
3.3. FINDING LOGS FOR KNATIVE SERVING SERVICES

You can find the logs for Knative Serving services using the following procedure.

3.3.1. Using OpenShift Logging to find logs for services deployed with Knative Serving

With OpenShift Logging, the logs that your applications write to the console are collected in Elasticsearch. The following procedure outlines how to apply these capabilities to applications deployed by using Knative Serving.

Prerequisites

- Install the OpenShift CLI (`oc`).

Procedure

1. Get the Kibana route:

   ```bash
   $ oc -n openshift-logging get route kibana
   ```

2. Use the route’s URL to navigate to the Kibana dashboard and log in.

3. Check that the index is set to `.all`. If the index is not set to `.all`, only the OpenShift system logs will be listed.

4. Filter the logs by using the `knative-serving` namespace. Enter a filter for the service in the search box to filter results.

   **Example filter**

   ```bash
   kubernetes.namespace_name=default AND kubernetes.labels.serving_knative_dev/service: {service_name}
   ```

   You can also filter by using `/configuration` or `/revision`.

5. Narrow your search by using `kubernetes.container_name:<user_container>` to only display the logs generated by your application. Otherwise, you will see logs from the queue-proxy.

**NOTE**

Use JSON-based structured logging in your application to allow for the quick filtering of these logs in production environments.
CHAPTER 4. TRACING

4.1. TRACING REQUESTS

Distributed tracing records the path of a request through the various services that make up an application. It is used to tie information about different units of work together, to understand a whole chain of events in a distributed transaction. The units of work might be executed in different processes or hosts.

4.1.1. Distributed tracing overview

As a service owner, you can use distributed tracing to instrument your services to gather insights into your service architecture. You can use distributed tracing for monitoring, network profiling, and troubleshooting the interaction between components in modern, cloud-native, microservices-based applications.

With distributed tracing you can perform the following functions:

- Monitor distributed transactions
- Optimize performance and latency
- Perform root cause analysis

Red Hat OpenShift distributed tracing consists of two main components:

- Red Hat OpenShift distributed tracing platform - This component is based on the open source Jaeger project.
- Red Hat OpenShift distributed tracing data collection - This component is based on the open source OpenTelemetry project.

Both of these components are based on the vendor-neutral OpenTracing APIs and instrumentation.

4.1.2. Additional resources for OpenShift Container Platform

- Red Hat OpenShift distributed tracing architecture
- Installing distributed tracing

4.2. USING RED HAT OPENSerceHIT DISTRIBUTED TRACING

You can use Red Hat OpenShift distributed tracing with OpenShift Serverless to monitor and troubleshoot serverless applications.

4.2.1. Using Red Hat OpenShift distributed tracing to enable distributed tracing

Red Hat OpenShift distributed tracing is made up of several components that work together to collect, store, and display tracing data.

Prerequisites

- You have access to an OpenShift Container Platform account with cluster administrator access.
You have not yet installed the OpenShift Serverless Operator, Knative Serving, and Knative Eventing. These must be installed after the Red Hat OpenShift distributed tracing installation.

You have installed Red Hat OpenShift distributed tracing by following the OpenShift Container Platform "Installing distributed tracing" documentation.

You have installed the OpenShift CLI (oc).

You have created a project or have access to a project with the appropriate roles and permissions to create applications and other workloads in OpenShift Container Platform.

Procedure

1. Create an OpenTelemetryCollector custom resource (CR):

Example OpenTelemetryCollector CR

```yaml
apiVersion: opentelemetry.io/v1alpha1
kind: OpenTelemetryCollector
metadata:
  name: cluster-collector
  namespace: <namespace>
spec:
  mode: deployment
  config:
    receivers:
      zipkin:
    processors:
    exporters:
      jaeger:
        endpoint: jaeger-all-in-one-inmemory-collector-headless.tracing-system.svc:14250
        tls:
          ca_file: /var/run/secrets/kubernetes.io/serviceaccount/service-ca.crt
    logging:
    service:
      pipelines:
        traces:
          receivers: [zipkin]
          processors: []
          exporters: [jaeger, logging]
```

2. Verify that you have two pods running in the namespace where Red Hat OpenShift distributed tracing is installed:

   ```bash
   $ oc get pods -n <namespace>
   ```

   Example output

<table>
<thead>
<tr>
<th>NAME</th>
<th>READY</th>
<th>STATUS</th>
<th>RESTARTS</th>
<th>AGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>cluster-collector-collector-85c766b5c-b5g99</td>
<td>1/1</td>
<td>Running</td>
<td>0</td>
<td>5m56s</td>
</tr>
<tr>
<td>jaeger-all-in-one-inmemory-ccbc9df4b-ndkl5</td>
<td>2/2</td>
<td>Running</td>
<td>0</td>
<td>15m</td>
</tr>
</tbody>
</table>

3. Verify that the following headless services have been created:
These services are used to configure Jaeger, Knative Serving, and Knative Eventing. The name of the Jaeger service may vary.

4. Install the OpenShift Serverless Operator by following the "Installing the OpenShift Serverless Operator" documentation.

5. Install Knative Serving by creating the following KnativeServing CR:

Example KnativeServing CR

```yaml
apiVersion: operator.knative.dev/v1beta1
kind: KnativeServing
metadata:
  name: knative-serving
  namespace: knative-serving
spec:
  config:
    tracing:
      backend: "zipkin"
    debug: "false"
    sample-rate: "0.1"  
```

The sample-rate defines sampling probability. Using sample-rate: "0.1" means that 1 in 10 traces are sampled.

6. Install Knative Eventing by creating the following KnativeEventing CR:

Example KnativeEventing CR

```yaml
apiVersion: operator.knative.dev/v1beta1
kind: KnativeEventing
metadata:
  name: knative-eventing
  namespace: knative-eventing
spec:
  config:
    tracing:
      backend: "zipkin"
    debug: "false"
    sample-rate: "0.1"  
```
The **sample-rate** defines sampling probability. Using **sample-rate: "0.1"** means that 1 in 10 traces are sampled.

7. Create a Knative service:

**Example service**

```yaml
custom-resource
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  name: helloworld-go
spec:
template:
  metadata:
    labels:
      app: helloworld-go
  spec:
    containers:
      - image: quay.io/openshift-knative/helloworld:v1.2
        imagePullPolicy: Always
        resources:
          requests:
            cpu: "200m"
        env:
          - name: TARGET
            value: "Go Sample v1"
```

8. Make some requests to the service:

**Example HTTPS request**

```
$ curl https://helloworld-go.example.com
```

9. Get the URL for the Jaeger web console:

**Example command**

```
$ oc get route jaeger-all-in-one-inmemory -o jsonpath='{.spec.host}' -n <namespace>
```

You can now examine traces by using the Jaeger console.

### 4.3. USING JAEGER DISTRIBUTED TRACING

If you do not want to install all of the components of Red Hat OpenShift distributed tracing, you can still use distributed tracing on OpenShift Container Platform with OpenShift Serverless.

#### 4.3.1. Configuring Jaeger to enable distributed tracing

To enable distributed tracing using Jaeger, you must install and configure Jaeger as a standalone integration.
Prerequisites

- You have cluster administrator permissions on OpenShift Container Platform, or you have cluster or dedicated administrator permissions on Red Hat OpenShift Service on AWS or OpenShift Dedicated.

- You have installed the OpenShift Serverless Operator, Knative Serving, and Knative Eventing.

- You have installed the Red Hat OpenShift distributed tracing platform Operator.

- You have installed the OpenShift CLI (oc).

- You have created a project or have access to a project with the appropriate roles and permissions to create applications and other workloads.

Procedure

1. Create and apply a Jaeger custom resource (CR) that contains the following:

   Jaeger CR

   ```yaml
   apiVersion: jaegertracing.io/v1
   kind: Jaeger
   metadata:
     name: jaeger
     namespace: default
   ```

2. Enable tracing for Knative Serving, by editing the KnativeServing CR and adding a YAML configuration for tracing:

   Tracing YAML example for Serving

   ```yaml
   apiVersion: operator.knative.dev/v1beta1
   kind: KnativeServing
   metadata:
     name: knative-serving
     namespace: knative-serving
   spec:
     config:
       tracing:
         sample-rate: "0.1"
         backend: zipkin
         zipkin-endpoint: "http://jaeger-collector.default.svc.cluster.local:9411/api/v2/spans"
         debug: "false"
   ```

   1 The sample-rate defines sampling probability. Using sample-rate: "0.1" means that 1 in 10 traces are sampled.

   2 backend must be set to zipkin.

   3 The zipkin-endpoint must point to your jaeger-collector service endpoint. To get this endpoint, substitute the namespace where the Jaeger CR is applied.

   4 Debugging should be set to false. Enabling debug mode by setting debug: "true" allows all spans to be sent to the server, bypassing sampling.
3. Enable tracing for Knative Eventing by editing the `KnativeEventing` CR:

**Tracing YAML example for Eventing**

```yaml
apiVersion: operator.knative.dev/v1beta1
class: KnativeEventing
metadata:
  name: knative-eventing
  namespace: knative-eventing
spec:
  config:
    tracing:
      sample-rate: "0.1" ➊
      backend: zipkin ➋
      debug: "false" ➍
```

➊ The `sample-rate` defines sampling probability. Using `sample-rate: "0.1"` means that 1 in 10 traces are sampled.

➋ Set `backend` to `zipkin`.

➌ Point the `zipkin-endpoint` to your `jaeger-collector` service endpoint. To get this endpoint, substitute the namespace where the Jaeger CR is applied.

➍ Debugging should be set to `false`. Enabling debug mode by setting `debug: "true"` allows all spans to be sent to the server, bypassing sampling.

**Verification**

You can access the Jaeger web console to see tracing data, by using the `jaeger` route.

1. Get the `jaeger` route’s hostname by entering the following command:

   ```bash
   $ oc get route jaeger -n default
   ```

   **Example output**

<table>
<thead>
<tr>
<th>NAME</th>
<th>HOST/PORT</th>
<th>PATH</th>
<th>SERVICES</th>
<th>PORT</th>
<th>TERMINATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>WILDCARD</td>
<td>jaeger-default.apps.example.com</td>
<td>jaeger-query</td>
<td>&lt;all&gt;</td>
<td>reencrypt</td>
<td>None</td>
</tr>
</tbody>
</table>

2. Open the endpoint address in your browser to view the console.