Red Hat OpenShift Serverless 1.32

可観測性

管理者および開発者メトリクス、クラスターロギング、トレースなどの可観測性機能

Last Updated: 2024-05-20

Red Hat OpenShift Serverless 1.32 可観測性

管理者および開発者メトリクス、クラスターロギング、トレースなどの可観測性機能

法律上の通知

概要

このドキュメントでは、Knative サービスのパフォーマンスを監視する方法を説明します。また、OpenShift Serverless で OpenShift Logging および OpenShift 分散トレーシングを使用する方法についても詳しく説明します。

目次

第1章 管理者メトリック

1.1. サーバーレス管理者のメトリクス

メトリクスにより、クラスター管理者は OpenShift Serverless クラスターコンポーネントおよびワークロードのパフォーマンスを監視できます。

Web コンソールの **Administrator** パースペクティブで **Dashboards** に移動すると、OpenShift サーバーレスのさまざまなメトリックを表示できます。

1.1.1. 前提条件

- クラスターのメトリクスの有効化に関する詳細は、OpenShift Container Platform ドキュメントの メトリクスの管理 を参照する。
- クラスター管理者アクセス権 (または OpenShift Dedicated または Red Hat OpenShift Service on AWS の専用管理者アクセス権) が割り当てられたアカウントにアクセスできる。
- Web コンソールの **Administrator** パースペクティブにアクセスできる。

警告

サービスメッシュが mTLS で有効にされている場合は、サービスメッシュが Prometheus のメトリクスの収集を阻止するため、Knative Serving のメトリクスがデフォルトで無効にされます。

この問題の解決については、Enabling Knative Serving metrics when using Service Mesh with mTLS の有効化を参照してください。

メトリクスの収集は、Knative サービスの自動スケーリングには影響しません。これは、収集要求がアクティベーターを通過しないためです。その結果、Pod が実行していない場合に収集が行われることはありません。

1.2. SERVERLESS コントローラーメトリック

以下のメトリックは、コントローラーロジックを実装するコンポーネントによって出力されます。これらのメトリックは、調整要求がワークキューに追加される調整操作とワークキューの動作に関する詳細を示します。

メトリック名	説明	タイプ	タグ	単位
work_queue_depth	ワークキューの深さ。	ゲージ	**reconciler**	整数 (単位なし)
reconcile_count	調整操作の数。	カウンター	**reconciler**、**success**	整数 (単位なし)

メトリック名	説明	タイプ	タグ	単位
reconcile_latency	調整操作のレイテンシー。	ヒストグラム	**reconciler**、**success**	ミリ秒
workqueue_adds_total	ワークキューによって処理される追加アクションの合計数。	カウンター	**name**	整数 (単位なし)
workqueue_queue_latency_seconds	アイテムが要求される前にワークキューにとどまる時間の長さ。	ヒストグラム	**name**	秒
workqueue_retries_total	ワークキューによって処理された再試行回数。	カウンター	**name**	整数 (単位なし)
workqueue_work_duration_seconds	ワークキューからの項目の処理にかかる時間の長さ。	ヒストグラム	**name**	秒
workqueue_unfinished_work_seconds	未処理のワークキュー項目が進行中であった時間の長さ。	ヒストグラム	**name**	秒
workqueue_longest_running_processor_seconds	最も長い間未処理のワークキュー項目が進行中であった時間の長さ。	ヒストグラム	**name**	秒

1.3. WEBHOOK メトリック

Webhook メトリックは操作に関する有用な情報を表示します。たとえば、多数の操作が失敗する場合は、これはユーザーが作成したリソースに問題があることを示している可能性があります。

メトリック名	説明	タイプ	タグ	単位

メトリック名	説明	タイプ	タグ	単位
request_count	Webhook にルーティングされる要求の数。	カウンター	**admission_allowed**、**kind_group**、**kind_kind**、**kind_version**、**request_operation**、**resource_group**、**resource_namespace**、**resource_resource**、**resource_version**	整数 (単位なし)
request_latencies	Webhook 要求の応答時間。	ヒストグラム	**admission_allowed**、**kind_group**、**kind_kind**、**kind_version**、**request_operation**、**resource_group**、**resource_namespace**、**resource_resource**、**resource_version**	ミリ秒

1.4. KNATIVE EVENTING メトリック

クラスター管理者は、Knative Eventing コンポーネントの以下のメトリックを表示できます。

HTTP コードからメトリックを集計することで、イベントは正常なイベント (2xx) および失敗したイベント (5xx) の 2 つのカテゴリーに分類できます。

1.4.1. ブローカー Ingress メトリック

以下のメトリックを使用してブローカー Ingress をデバッグし、どのように実行されているかを確認し、どのイベントが Ingress コンポーネントによってディスパッチされているかを確認できます。

メトリック名	説明	タイプ	タグ	単位
event_count	ブローカーによって受信されるイベントの数。	カウンター	**broker_name**、**event_type**、**namespace_name**、**response_code**、**response_code_class**、**unique_name**	整数 (単位なし)

メトリック名	説明	タイプ	タグ	単位
event_dispatch_latencies	イベントのチャネルへのディスパッチにかかる時間。	ヒストグラム	**broker_name**、**event_type**、**namespace_name**、**response_code**、**response_code_class**、**unique_name**	ミリ秒

1.4.2. ブローカーフィルターメトリック

以下のメトリックを使用してブローカーフィルターをデバッグし、それらがどのように実行されているかを確認し、どのイベントがフィルターによってディスパッチされているかを確認できます。イベントでフィルタリングアクションのレイテンシーを測定することもできます。

メトリック名	説明	タイプ	タグ	単位
event_count	ブローカーによって受信されるイベントの数。	カウンター	**broker_name**、**container_name**、**filter_type**、**namespace_name**、**response_code**、**response_code_class**、**trigger_name**、**unique_name**	整数 (単位なし)
event_dispatch_latencies	イベントのチャネルへのディスパッチにかかる時間。	ヒストグラム	**broker_name**、**container_name**、**filter_type**、**namespace_name**、**response_code**、**response_code_class**、**trigger_name**、**unique_name**	ミリ秒
event_processing_latencies	トリガーサブスクライバーにディスパッチされる前にイベントの処理にかかる時間。	ヒストグラム	**broker_name**、**container_name**、**filter_type**、**namespace_name**、**trigger_name**、**unique_name**	ミリ秒

1.4.3. InMemoryChannel dispatcher メトリック

以下のメトリックを使用して **InMemoryChannel** チャネルをデバッグし、それらがどのように実行されているかを確認し、どのイベントがチャネルによってディスパッチされているかを確認できます。

メトリック名	説明	タイプ	タグ	単位
event_count	**InMemoryChannel** チャネルでディスパッチされるイベントの数。	カウンター	**broker_name**、**container_name**、**filter_type**、**namespace_name**、**response_code**、**response_code_class**、**trigger_name**、**unique_name**	整数 (単位なし)
event_dispatch_latencies	**InMemoryChannel** チャネルからのイベントのディスパッチにかかる時間。	ヒストグラム	**broker_name**、**container_name**、**filter_type**、**namespace_name**、**response_code**、**response_code_class**、**trigger_name**、**unique_name**	ミリ秒

1.4.4. イベントソースメトリック

以下のメトリックを使用して、イベントがイベントソースから接続されたイベントシンクに配信されていることを確認できます。

メトリック名	説明	タイプ	タグ	単位
event_count	イベントソースによって送信されるイベントの数。	カウンター	**broker_name**、**container_name**、**filter_type**、**namespace_name**、**response_code**、**response_code_class**、**trigger_name**、**unique_name**	整数 (単位なし)
retry_event_count	最初に配信に失敗した後にイベントソースによって送信される再試行イベントの数。	カウンター	**event_source**、**event_type**、**name**、**namespace_name**、**resource_group**、**response_code**、**response_code_class**、**response_error**、**response_timeout**	整数 (単位なし)

1.5. KNATIVE SERVING メトリック

クラスター管理者は、Knative Serving コンポーネントの以下のメトリックを表示できます。

1.5.1. activator メトリック

以下のメトリックを使用して、トラフィックが activator 経由で渡されるときにアプリケーションがどのように応答するかを理解することができます。

メトリック名	説明	タイプ	タグ	単位
request_concurrency	activator にルーティングされる同時要求の数、またはレポート期間における平均同時実行数。	ゲージ	**configuration_name、container_name、namespace_name、pod_name、revision_name、service_name**	整数 (単位なし)
request_count	activator にルーティングされる要求の数。これらは、activator ハンドラーから実行された要求です。	カウンター	**configuration_name、container_name、namespace_name、pod_name、response_code、response_code_class、revision_name、service_name**	整数 (単位なし)
request_latencies	実行され、ルーティングされた要求の応答時間 (ミリ秒単位)。	ヒストグラム	**configuration_name、container_name、namespace_name、pod_name、response_code、response_code_class、revision_name、service_name**	ミリ秒

1.5.2. Autoscaler メトリック

Autoscaler コンポーネントは、それぞれのリビジョンの Autoscaler の動作に関連する多数のメトリックを公開します。たとえば、任意の時点で、Autoscaler がサービスに割り当てようとする Pod のターゲット数、安定期間中の 1 秒あたりの要求の平均数、または Knative Pod Autoscaler (KPA) を使用している場合に Autoscaler がパニックモードであるかどうかなどを監視できます。

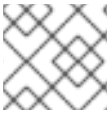
メトリック名	説明	タイプ	タグ	単位

メトリック名	説明	タイプ	タグ	単位
desired_pods	Autoscaler がサービスへの割り当てを試みる Pod 数。	ゲージ	**configuration_name**、**namespace_name**、**revision_name**、**service_name**	整数 (単位なし)
excess_burst_capacity	stable ウインドウで提供される追加のバースト容量。	ゲージ	**configuration_name**、**namespace_name**、**revision_name**、**service_name**	整数 (単位なし)
stable_request_concurrency	stable ウィンドウで監視される各 Pod の要求数の平均。	ゲージ	**configuration_name**、**namespace_name**、**revision_name**、**service_name**	整数 (単位なし)
panic_request_concurrency	panic ウィンドウで監視される各 Pod の要求数の平均。	ゲージ	**configuration_name**、**namespace_name**、**revision_name**、**service_name**	整数 (単位なし)
target_concurrency_per_pod	Autoscaler が各 Pod への送信を試みる同時要求の数。	ゲージ	**configuration_name**、**namespace_name**、**revision_name**、**service_name**	整数 (単位なし)
stable_requests_per_second	stable ウィンドウで監視される各 Pod の 1 秒当たりの要求数の平均。	ゲージ	**configuration_name**、**namespace_name**、**revision_name**、**service_name**	整数 (単位なし)
panic_requests_per_second	panic ウィンドウで監視される各 Pod の 1 秒当たりの要求数の平均。	ゲージ	**configuration_name**、**namespace_name**、**revision_name**、**service_name**	整数 (単位なし)
target_requests_per_second	Autoscaler が各 Pod をターゲットとする 1 秒あたりの要求の数。	ゲージ	**configuration_name**、**namespace_name**、**revision_name**、**service_name**	整数 (単位なし)

メトリック名	説明	タイプ	タグ	単位
panic_mode	この値は、Autoscaler がパニックモードの場合は **1** になります。Autoscaler がパニックモードではない場合は **0** になります。	ゲージ	**configuration_name**、**namespace_name**、**revision_name**、**service_name**	整数 (単位なし)
requested_pods	Autoscaler が Kubernetes クラスターから要求した Pod 数。	ゲージ	**configuration_name**、**namespace_name**、**revision_name**、**service_name**	整数 (単位なし)
actual_pods	割り当てられ、現在準備完了状態にある Pod 数。	ゲージ	**configuration_name**、**namespace_name**、**revision_name**、**service_name**	整数 (単位なし)
not_ready_pods	準備未完了状態の Pod 数。	ゲージ	**configuration_name**、**namespace_name**、**revision_name**、**service_name**	整数 (単位なし)
pending_pods	現在保留中の Pod 数。	ゲージ	**configuration_name**、**namespace_name**、**revision_name**、**service_name**	整数 (単位なし)
terminating_pods	現在終了中の Pod 数。	ゲージ	**configuration_name**、**namespace_name**、**revision_name**、**service_name**	整数 (単位なし)

1.5.3. Go ランタイムメトリック

各 Knative Serving コントロールプレーンプロセスは、Go ランタイムメモリーの統計を多数出力します (MemStats) 。



注記

各メトリックの **name** タグは空のタグです。

メトリック名	説明	タイプ	タグ	単位
go_alloc	割り当てられたヒープオブジェクトのバイト数。このメトリックは **heap_alloc** と同じです。	ゲージ	**name**	整数 (単位なし)
go_total_alloc	ヒープオブジェクトに割り当てられる累積バイト数。	ゲージ	**name**	整数 (単位なし)
go_sys	オペレーティングシステムから取得したメモリーの合計バイト数。	ゲージ	**name**	整数 (単位なし)
go_lookups	ランタイムが実行したポインター検索の数。	ゲージ	**name**	整数 (単位なし)
go_mallocs	割り当てられるヒープオブジェクトの累積数。	ゲージ	**name**	整数 (単位なし)
go_frees	解放されているヒープオブジェクトの累積数。	ゲージ	**name**	整数 (単位なし)
go_heap_alloc	割り当てられたヒープオブジェクトのバイト数。	ゲージ	**name**	整数 (単位なし)
go_heap_sys	オペレーティングシステムから取得したヒープメモリーのバイト数。	ゲージ	**name**	整数 (単位なし)
go_heap_idle	アイドル状態の未使用スパンのバイト数。	ゲージ	**name**	整数 (単位なし)
go_heap_in_use	現在使用中のスパンのバイト数。	ゲージ	**name**	整数 (単位なし)
go_heap_released	オペレーティングシステムに返された物理メモリーのバイト数。	ゲージ	**name**	整数 (単位なし)

メトリック名	説明	タイプ	タグ	単位
go_heap_objects	割り当てられるヒープオブジェクトの数。	ゲージ	**name**	整数 (単位なし)
go_stack_in_use	現在使用中のスタックスパンのバイト数。	ゲージ	**name**	整数 (単位なし)
go_stack_sys	オペレーティングシステムから取得したスタックメモリーのバイト数。	ゲージ	**name**	整数 (単位なし)
go_mspan_in_use	割り当てられた **mspan** 構造のバイト数。	ゲージ	**name**	整数 (単位なし)
go_mspan_sys	**mspan** 構造のオペレーティングシステムから取得したメモリーのバイト数。	ゲージ	**name**	整数 (単位なし)
go_mcache_in_use	割り当てられた **mcache** 構造のバイト数。	ゲージ	**name**	整数 (単位なし)
go_mcache_sys	**mcache** 構造のためにオペレーティングシステムから取得したメモリーのバイト数。	ゲージ	**name**	整数 (単位なし)
go_bucket_hash_sys	バケットハッシュテーブルのプロファイリングにおけるメモリーのバイト数。	ゲージ	**name**	整数 (単位なし)
go_gc_sys	ガベージコレクションメタデータのメモリーのバイト数。	ゲージ	**name**	整数 (単位なし)

メトリック名	説明	タイプ	タグ	単位
go_other_sys	その他のオフヒープランタイム割り当てのメモリーのバイト数。	ゲージ	**name**	整数 (単位なし)
go_next_gc	次のガベージコレクションサイクルのターゲットヒープサイズ。	ゲージ	**name**	整数 (単位なし)
go_last_gc	最後のガベージコレクションが完了した時間 (Epoch または Unix 時間)。	ゲージ	**name**	ナノ秒
go_total_gc_pause_ns	プログラム開始以降のガベージコレクションの stop-the-world 停止の累積時間。	ゲージ	**name**	ナノ秒
go_num_gc	完了したガベージコレクションサイクルの数。	ゲージ	**name**	整数 (単位なし)
go_num_forced_gc	ガベージコレクションの機能を呼び出すアプリケーションが原因で強制されたガベージコレクションサイクルの数。	ゲージ	**name**	整数 (単位なし)
go_gc_cpu_fraction	プログラムの開始以降にガベージコレクターによって使用されたプログラムの使用可能な CPU 時間の一部。	ゲージ	**name**	整数 (単位なし)

第2章 開発者メトリック

2.1. SERVERLESS 開発者メトリックの概要

メトリクスを使用すると、開発者は Knative サービスのパフォーマンスを監視できます。OpenShift Container Platform モニタリングスタックを使用して、Knative サービスのヘルスチェックおよびメトリクスを記録し、表示できます。

Web コンソールの **Developer** パースペクティブで **Dashboards** に移動して、OpenShift Serverless のさまざまなメトリクスを表示できます。

警告

サービスメッシュが mTLS で有効にされている場合は、サービスメッシュが Prometheus のメトリクスの収集を阻止するため、Knative Serving のメトリクスがデフォルトで無効にされます。

この問題の解決については、Enabling Knative Serving metrics when using Service Mesh with mTLS の有効化を参照してください。

メトリクスの収集は、Knative サービスの自動スケーリングには影響しません。これは、収集要求がアクティベーターを通過しないためです。その結果、Pod が実行していない場合に収集が行われることはありません。

2.1.1. OpenShift Container Platform の関連情報

- モニタリングの概要
- ユーザー定義プロジェクトのモニタリングの有効化
- サービスのモニター方法の指定

2.2. デフォルトで公開される KNATIVE サービスメトリクス

表2.1 ポート 9090 の各 Knative サービスについてデフォルトで公開されるメトリック

メトリック名、単位、およびタイプ	説明	メトリックのタグ
queue_requests_per_second メトリックの単位: 無次元 メトリックのタイプ: ゲージ	キュープロキシーに到達する、1 秒あたりのリクエスト数。 式: **stats.RequestCount / r.reportingPeriodSeconds** **stats.RequestCount** は、指定のレポート期間のネットワーク **pkg** 統計から直接計算されます。	destination_configuration="event-display", destination_namespace="pingsource1", destination_pod="event-display-00001-deployment-6b455479cb-75p6w", destination_revision="event-display-00001"

メトリック名、単位、およびタイプ	説明	メトリックのタグ
queue_proxied_operations_per_second メトリックの単位: 無次元 メトリックのタイプ: ゲージ	1秒あたりのプロキシー化された要求の数。 式: **stats.ProxiedRequestCount / r.reportingPeriodSeconds** **stats.ProxiedRequestCount** は指定されたレポート期間のネットワーク **pkg** 統計から直接計算されます。	
queue_average_concurrent_requests メトリックの単位: 無次元 メトリックのタイプ: ゲージ	この Pod で現在処理されている要求の数。 平均同時実行性は、ネットワークの **pkg** 側で次のように計算されます。 ● **req** の変更が行われると、変更間の時間デルタが計算されます。この結果に基づいて、デルタ上の現在の同時実行数が計算され、現在計算されている同時実行数に追加されます。また、デルタの合計が保持されます。デルタでの現在の同時実行処理は、以下のように計算されます。 **global_concurrency** × デルタ ● レポートが実行されるたびに、合計および現在の計算された同時実行性がリセットされます。 ● 平均同時実行値を報告すると、現在の計算処理はデルタの合計で除算されます。 ● 新しいリクエストが出されると、グローバル同時実行カウンターが増えます。リクエストが完了すると、カウンターが減少します。	destination_configuration="event-display", destination_namespace="pingsource1", destination_pod="event-display-00001-deployment-6b455479cb-75p6w", destination_revision="event-display-00001"

メトリック名、単位、およびタイプ	説明	メトリックのタグ
queue_average_proxied_concurrent_requests メトリックの単位: 無次元 メトリックのタイプ: ゲージ	この Pod で現在処理されているプロキシー要求の数。 **stats.AverageProxiedConcurrency**	destination_configuration="event-display", destination_namespace="pingsource1", destination_pod="event-display-00001-deployment-6b455479cb-75p6w", destination_revision="event-display-00001"
process_uptime メトリック単位: 秒 メトリックのタイプ: ゲージ	プロセスが起動している秒数。	destination_configuration="event-display", destination_namespace="pingsource1", destination_pod="event-display-00001-deployment-6b455479cb-75p6w", destination_revision="event-display-00001"

表2.2 ポート 9091 の各 Knative サービスについてデフォルトで公開されるメトリック

メトリック名、単位、およびタイプ	説明	メトリックのタグ
request_count メトリックの単位: 無次元 メトリックのタイプ: カウンター	**queue-proxy** にルーティングされる要求の数。	configuration_name="event-display", container_name="queue-proxy", namespace_name="apiserversource1", pod_name="event-display-00001-deployment-658fd4f9cf-qcnr5", response_code="200", response_code_class="2xx", revision_name="event-display-00001", service_name="event-display"
request_latencies メトリックの単位: ミリ秒 メトリックのタイプ: ヒストグラム	応答時間 (ミリ秒単位)。	configuration_name="event-display", container_name="queue-proxy", namespace_name="apiserversource1", pod_name="event-display-00001-deployment-658fd4f9cf-qcnr5", response_code="200", response_code_class="2xx", revision_name="event-display-00001", service_name="event-display"

メトリック名、単位、およびタイプ	説明	メトリックのタグ
app_request_count メトリックの単位: 無次元 メトリックのタイプ: カウンター	**user-container** にルーティングされる要求の数。	configuration_name="event-display", container_name="queue-proxy", namespace_name="apiserversource1", pod_name="event-display-00001-deployment-658fd4f9cf-qcnr5", response_code="200", response_code_class="2xx", revision_name="event-display-00001", service_name="event-display"
app_request_latencies メトリックの単位: ミリ秒 メトリックのタイプ: ヒストグラム	応答時間 (ミリ秒単位)。	configuration_name="event-display", container_name="queue-proxy", namespace_name="apiserversource1", pod_name="event-display-00001-deployment-658fd4f9cf-qcnr5", response_code="200", response_code_class="2xx", revision_name="event-display-00001", service_name="event-display"
queue_depth メトリックの単位: 無次元 メトリックのタイプ: ゲージ	提供および待機キューの現在の項目数。無制限の同時実行の場合は報告されません。**breaker.inFlight** が使用されます。	configuration_name="event-display", container_name="queue-proxy", namespace_name="apiserversource1", pod_name="event-display-00001-deployment-658fd4f9cf-qcnr5", response_code="200", response_code_class="2xx", revision_name="event-display-00001", service_name="event-display"

2.3. カスタムアプリケーションメトリックを含む KNATIVE サービス

Knative サービスによってエクスポートされるメトリックのセットを拡張できます。正確な実装は、使用するアプリケーションと言語によって異なります。

以下のリストは、処理されたイベントカスタムメトリックの数をエクスポートするサンプル Go アプリケーションを実装します。

```
package main

import (
  "fmt"
  "log"
  "net/http"
```

```
  "os"

  "github.com/prometheus/client_golang/prometheus" 1
  "github.com/prometheus/client_golang/prometheus/promauto"
  "github.com/prometheus/client_golang/prometheus/promhttp"
)

var (
  opsProcessed = promauto.NewCounter(prometheus.CounterOpts{ 2
    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!\n", target)
  opsProcessed.Inc() 3
}

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() { 4
    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.Handle("/metrics", promhttp.Handler()) 5
  log.Printf("helloworld: listening on port %s", port)
  log.Fatal(http.ListenAndServe(fmt.Sprintf(":%s", port), nil))
}
```

1 Prometheus パッケージの追加。

2 **opsProcessed** メトリックの定義。

3 **opsProcessed** メトリックのインクリメント。

4 メトリック要求に別のサーバーを使用するように設定。

5 メトリックおよび **metrics** サブパスの通常の要求と同じポートを使用するように設定。

2.4. カスタムメトリックの収集の設定

カスタムメトリクスの収集は、ユーザーワークロードのモニタリング用に設計された Prometheus のインスタンスで実行されます。ユーザーのワークロードのモニタリングを有効にしてアプリケーションを作成した後に、モニタリングスタックがメトリクスを収集する方法を定義する設定が必要になります。

以下のサンプル設定は、アプリケーションの **ksvc** を定義し、サービスモニターを設定します。正確な設定は、アプリケーションおよびメトリックのエクスポート方法によって異なります。

```
apiVersion: serving.knative.dev/v1 1
kind: Service
metadata:
  name: helloworld-go
spec:
  template:
    metadata:
      labels:
        app: helloworld-go
      annotations:
    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 2
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
---
apiVersion: v1 3
```

```
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
```

1. アプリケーション仕様。
2. アプリケーションのメトリックが収集される設定。
3. メトリクスの収集方法の設定。

2.5. サービスのメトリックの検証

メトリクスとモニタリングスタックをエクスポートするようにアプリケーションを設定したら、Web コンソールでメトリクスを検査できます。

前提条件

- OpenShift Container Platform Web コンソールにログインしている。
- OpenShift Serverless Operator および Knative Serving がインストールされている。

手順

1. オプション: メトリックに表示できるアプリケーションに対する要求を実行します。

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

出力例

```
Hello Go Sample v1!
```

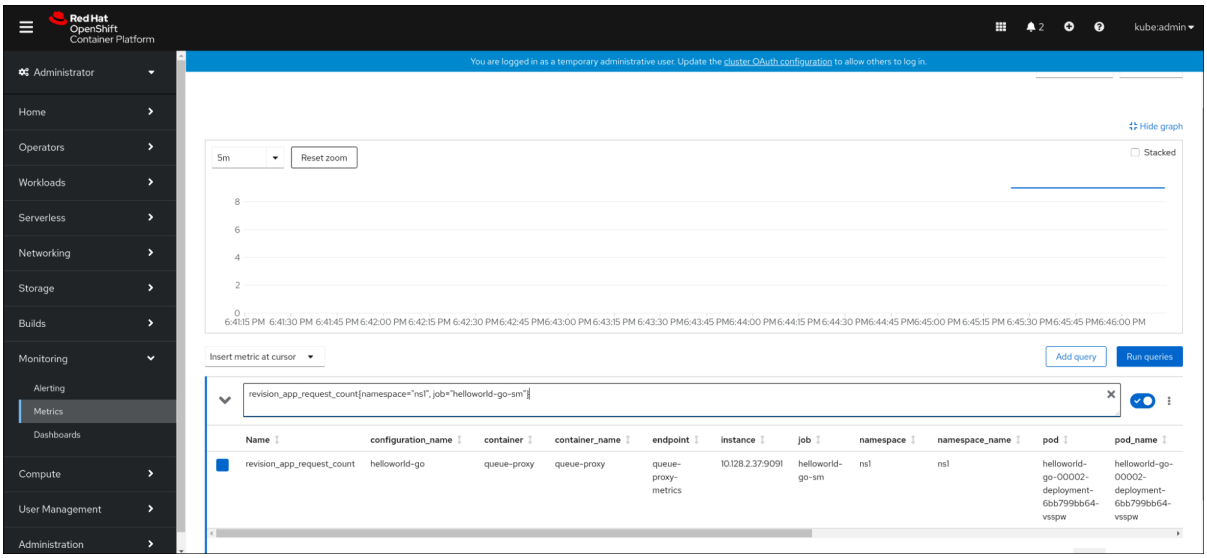
2. Web コンソールで、**Observe → Metrics** インターフェイスに移動します。
3. 入力フィールドに、監視するメトリクスのクエリーを入力します。以下に例を示します。

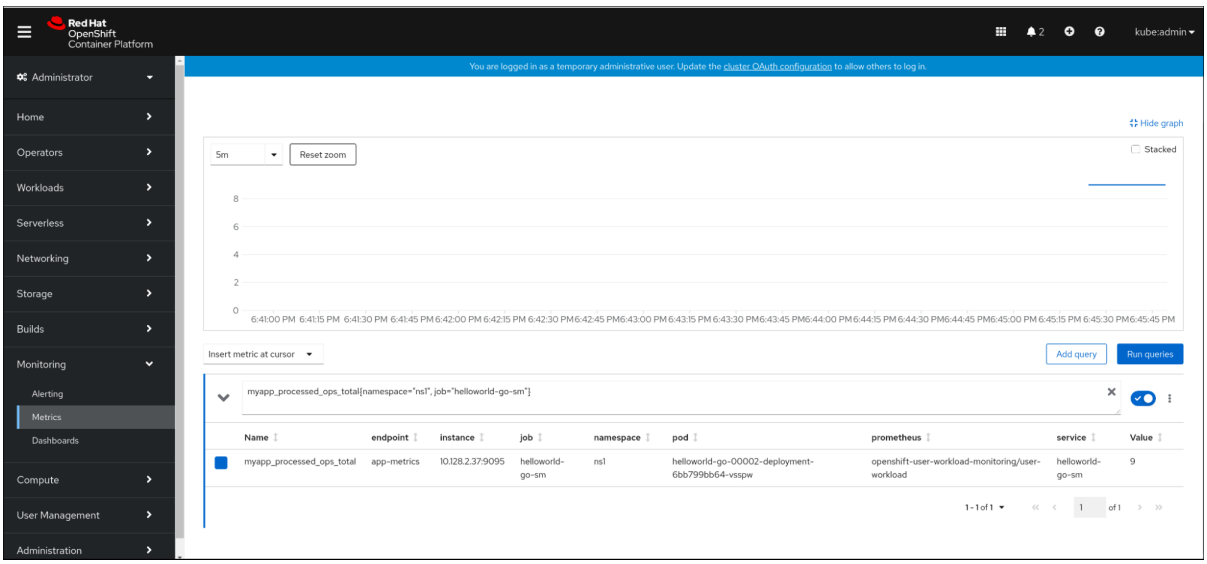
```
revision_app_request_count{namespace="ns1", job="helloworld-go-sm"}
```

別の例:

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

4. 可視化されたメトリクスを確認します。





2.5.1. キュープロキシーメトリクス

各 Knative サービスには、アプリケーションコンテナーへの接続をプロキシーするプロキシーコンテナーがあります。キュープロキシーのパフォーマンスについて多くのメトリックが報告されます。

以下のメトリックを使用して、要求がプロキシー側でキューに入れられているかどうか、およびアプリケーション側で要求を処理する際の実際の遅延を測定できます。

メトリック名	説明	タイプ	タグ	単位

メトリック名	説明	タイプ	タグ	単位
revision_request_count	**queue-proxy** Pod にルーティングされる要求の数。	カウンター	**configuration_name**、**container_name**、**namespace_name**、**pod_name**、**response_code**、**response_code_class**、**revision_name**、**service_name**	整数 (単位なし)
revision_request_latencies	リビジョン要求の応答時間。	ヒストグラム	**configuration_name**、**container_name**、**namespace_name**、**pod_name**、**response_code**、**response_code_class**、**revision_name**、**service_name**	ミリ秒
revision_app_request_count	**user-container** Pod にルーティングされる要求の数。	カウンター	**configuration_name**、**container_name**、**namespace_name**、**pod_name**、**response_code**、**response_code_class**、**revision_name**、**service_name**	整数 (単位なし)
revision_app_request_latencies	リビジョンアプリケーション要求の応答時間。	ヒストグラム	**configuration_name**、**namespace_name**、**pod_name**、**response_code**、**response_code_class**、**revision_name**、**service_name**	ミリ秒

メトリック名	説明	タイプ	タグ	単位
revision_queue_depth	**serving** および **waiting** キューの現在の項目数。無制限の同時実行が設定されている場合には、このメトリックは報告されません。	ゲージ	**configuration_name**、**event-display**、**container_name**、**namespace_name**、**pod_name**、**response_code_class**、**revision_name**、**service_name**	整数 (単位なし)

2.6. サービスメトリックのダッシュボード

namespace でキュープロキシーメトリクスを集約する専用のダッシュボードを使用してメトリクスを検査できます。

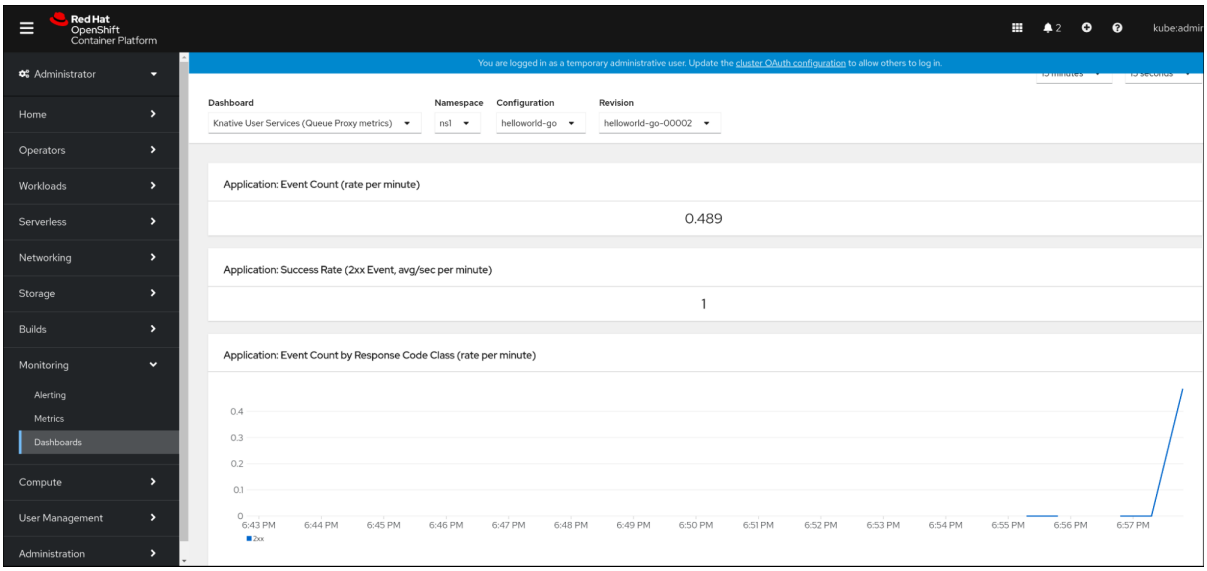
2.6.1. ダッシュボードでのサービスのメトリクスの検証

前提条件

- OpenShift Container Platform Web コンソールにログインしている。
- OpenShift Serverless Operator および Knative Serving がインストールされている。

手順

1. Web コンソールで、**Observe → Metrics** インターフェイスに移動します。
2. **Knative User Services (Queue Proxy metrics)** ダッシュボードを選択します。
3. アプリケーションに対応する **Namespace**、**Configuration**、および **Revision** を選択します。
4. 可視化されたメトリクスを確認します。



第3章 クラスターロギング

3.1. OPENSHIFT SERVERLESS での OPENSHIFT LOGGING の使用

3.1.1. Red Hat OpenShift の logging サブシステムのデプロイについて

OpenShift Container Platform クラスター管理者は、OpenShift Container Platform Web コンソールまたは CLI コマンドを使用してロギングシステムをデプロイし、OpenShift Elasticsearch Operator および Red Hat OpenShift Logging Operator をインストールできます。Operator がインストールされたら、**ClusterLogging** カスタムリソース (CR) を作成して、ロギングサブシステム pod およびロギングサブシステムをサポートするために必要なその他のリソースをスケジュールします。Operator は、ロギングサブシステムのデプロイ、アップグレード、および保守を担当します。

ClusterLogging CR は、ログを収集し、保存し、視覚化するために必要なロギングスタックのすべてのコンポーネントを含む完全なロギングシステム環境を定義します。Red Hat OpenShift Logging Operator はロギングシステム CR を監視し、ロギングデプロイメントを適宜調整します。

管理者およびアプリケーション開発者は、表示アクセスのあるプロジェクトのログを表示できます。

3.1.2. Red Hat OpenShift のロギングサブシステムのデプロイおよび設定

Logging システムは、小規模および中規模の OpenShift Container Platform クラスター用に調整されたデフォルト設定で使用されるように設計されています。

以下のインストール方法には、サンプルの **ClusterLogging** カスタムリソース (CR) が含まれます。これを使用して、ロギングサブシステムインスタンスを作成し、ロギングサブシステムの環境を設定できます。

デフォルトのロギングサビシステムのインストールを使用する必要がある場合は、サンプル CR を直接使用できます。

デプロイメントをカスタマイズする必要がある場合は、必要に応じてサンプル CR に変更を加えます。以下では、OpenShift Logging インスタンスのインストール時に実行し、インストール後に変更する設定を説明します。**ClusterLogging** カスタムリソース外で加える変更を含む、各コンポーネントの使用方法は、設定に関するセクションを参照してください。

3.1.2.1. ロギングサブシステムの設定および調整

ロギングサブシステムは、**openshift-logging** プロジェクトにデプロイされる **ClusterLogging** カスタムリソースを変更することによって設定できます。

インストール時またはインストール後に、以下のコンポーネントのいずれかを変更できます。

メモリーおよび CPU

resources ブロックを有効なメモリーおよび CPU 値で変更して、各コンポーネントの CPU およびメモリーの両方の制限を調整できます。

```
spec:
  logStore:
    elasticsearch:
      resources:
        limits:
          cpu:
          memory: 16Gi
```

```
        requests:
          cpu: 500m
          memory: 16Gi
      type: "elasticsearch"
  collection:
    logs:
      fluentd:
        resources:
          limits:
            cpu:
            memory:
          requests:
            cpu:
            memory:
        type: "fluentd"
  visualization:
    kibana:
      resources:
        limits:
          cpu:
          memory:
        requests:
          cpu:
          memory:
      type: kibana
```

Elasticsearch ストレージ

storageClass name および **size** パラメーターを使用し、Elasticsearch クラスターの永続ストレージのクラスおよびサイズを設定できます。Red Hat OpenShift Logging Operator は、これらのパラメーターに基づいて、Elasticsearch クラスターの各データノードについて永続ボリューム要求 (PVC) を作成します。

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

この例では、クラスターの各データノードが gp2 ストレージの 200G を要求する PVC にバインドされるように指定します。それぞれのプライマリーシャードは単一のレプリカによってサポートされます。

注記

storage ブロックを省略すると、一時ストレージのみを含むデプロイメントになります。

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

Elasticsearch レプリケーションポリシー

Elasticsearch シャードをクラスター内のデータノードにレプリケートする方法を定義するポリシーを設定できます。

- **FullRedundancy**: 各インデックスのシャードはすべてのデータノードに完全にレプリケートされます。
- **MultipleRedundancy**: 各インデックスのシャードはデータノードの半分に分散します。
- **SingleRedundancy**: 各シャードの単一コピー。2 つ以上のデータノードが存在する限り、ログは常に利用可能かつ回復可能です。
- **ZeroRedundancy**: シャードのコピーはありません。ログは、ノードの停止または失敗時に利用不可になる (または失われる) 可能性があります。

3.1.2.2. 変更された ClusterLogging カスタムリソースのサンプル

以下は、前述のオプションを使用して変更された **ClusterLogging** カスタムリソースの例です。

変更された ClusterLogging リソースのサンプル

```
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
      requests:
        cpu: 200m
        memory: 16Gi
      storage:
        storageClassName: "gp2"
        size: "200G"
    redundancyPolicy: "SingleRedundancy"
visualization:
  type: "kibana"
  kibana:
    resources:
      limits:
        memory: 1Gi
      requests:
        cpu: 500m
        memory: 1Gi
    replicas: 1
collection:
  logs:
    type: "fluentd"
    fluentd:
      resources:
        limits:
          memory: 1Gi
        requests:
          cpu: 200m
          memory: 1Gi
```

3.2. KNATIVE SERVING コンポーネントのログの検索

以下の手順を使用して、Knative Serving コンポーネントのログを検索できます。

3.2.1. OpenShift Logging の使用による Knative Serving コンポーネントのログの検索

前提条件

- OpenShift CLI (**oc**) がインストールされている。

手順

1. Kibana ルートを取得します。

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

2. ルートの URL を使用して Kibana ダッシュボードに移動し、ログインします。
3. インデックスが **.all** に設定されていることを確認します。インデックスが **.all** に設定されていない場合は、OpenShift Container Platform システムログのみが一覧表示されます。
4. **knative-serving** namespace を使用してログをフィルタリングします。**kubernetes.namespace_name:knative-serving** を検索ボックスに入力して結果をフィルタリングします。

注記

Knative Serving はデフォルトで構造化ロギングを使用します。OpenShift Logging Fluentd 設定をカスタマイズしてこれらのログの解析を有効にできます。これにより、ログの検索がより容易になり、ログレベルでのフィルターにより問題を迅速に特定できるようになります。

3.3. KNATIVE SERVING サービスのログの検索

以下の手順を使用して、Knative Serving サービスのログを検索できます。

3.3.1. OpenShift Logging を使用した Knative Serving でデプロイされたサービスのログの検索

OpenShift Logging により、アプリケーションがコンソールに書き込むログは Elasticsearch で収集されます。以下の手順で、Knative Serving を使用してデプロイされたアプリケーションにこれらの機能を適用する方法の概要を示します。

前提条件

- OpenShift CLI (**oc**) がインストールされている。

手順

1. Kibana ルートを取得します。

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

2. ルートの URL を使用して Kibana ダッシュボードに移動し、ログインします。
3. インデックスが **.all** に設定されていることを確認します。インデックスが **.all** に設定されていない場合は、OpenShift システムログのみが一覧表示されます。
4. **knative-serving** namespace を使用してログをフィルタリングします。検索ボックスにサービスのフィルターを入力して、結果をフィルターします。

   フィルターの例

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

   /configuration または **/revision** を使用してフィルタリングすることもできます。

5. **kubernetes.container_name:<user_container>** を使用して検索を絞り込み、ご使用のアプリケーションで生成されるログのみを表示することができます。それ以外の場合は、queue-proxy からのログが表示されます。

注記

アプリケーションで JSON ベースの構造化ロギングを使用することで、実稼働環境でのこれらのログの迅速なフィルターを実行できます。

第4章 トレーシング

4.1. リクエストのトレース

分散トレースは、アプリケーションを設定する各種のサービスを使用した要求のパスを記録します。これは、各種の異なる作業単位についての情報を連携させ、分散トランザクションでのイベントチェーン全体を把握できるようにするために使用されます。作業単位は、異なるプロセスまたはホストで実行される場合があります。

4.1.1. 分散トレースの概要

サービスの所有者は、分散トレースを使用してサービスをインストルメント化し、サービスアーキテクチャーに関する洞察を得ることができます。分散トレースを使用して、現代的なクラウドネイティブのマイクロサービスベースのアプリケーションにおける、コンポーネント間の対話の監視、ネットワークプロファイリング、およびトラブルシューティングを行うことができます。

分散トレースを使用すると、以下の機能を実行できます。

- 分散トランザクションの監視
- パフォーマンスとレイテンシーの最適化
- 根本原因分析の実行

Red Hat OpenShift の分散トレースは、2 つの主要コンポーネントで設定されています。

- **Red Hat OpenShift 分散トレースプラットフォーム**: このコンポーネントは、オープンソースの Jaeger プロジェクト に基づいています。
- **Red Hat OpenShift 分散トレースデータ収集**: このコンポーネントは、オープンソースの OpenTelemetry プロジェクト に基づいています。

これらのコンポーネントは共に、特定のベンダーに依存しない OpenTracing API およびインストルメンテーションに基づいています。

4.1.2. OpenShift Container Platform の関連情報

- Red Hat OpenShift 分散トレースのアーキテクチャー
- 分散トレースのインストール

4.2. RED HAT OPENSHIFT 分散トレースの使用

OpenShift Serverless で Red Hat 分散トレースを使用して、サーバーレスアプリケーションを監視およびトラブルシューティングできます。

4.2.1. Red Hat 分散トレースを使用して分散トレースを有効にする

Red Hat OpenShift 分散トレースは、複数のコンポーネントで設定されており、トレースデータを収集し、保存し、表示するためにそれらが連携します。

前提条件

- クラスター管理者のアクセスを持つ OpenShift Container Platform アカウントを使用できる。

- OpenShift Serverless Operator、Knative Serving、および Knative Eventing をインストールしていない。これらは Red Hat OpenShift 分散トレースのインストール後にインストールする必要があります。
- OpenShift Container Platform の分散トレーシングのインストールのドキュメントに従って、Red Hat OpenShift の分散トレーシングをインストールしている。
- OpenShift CLI (**oc**) がインストールされている。
- OpenShift Container Platform でアプリケーションおよび他のワークロードを作成するために、プロジェクトを作成しているか、適切なロールおよびパーミッションを持つプロジェクトにアクセスできる。

手順

1. **OpenTelemetryCollector** カスタムリソース (CR) を作成します。

   OpenTelemetryCollector CR の例

```
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. Red Hat 分散トレースがインストールされているネームスペースで 2 つの Pod が実行されていることを確認します。

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

   出力例

```
NAME                                         READY   STATUS    RESTARTS   AGE
cluster-collector-collector-85c766b5c-b5g99   1/1     Running   0          5m56s
jaeger-all-in-one-inmemory-ccbc9df4b-ndkl5    2/2     Running   0          15m
```

3. 次のヘッドレスサービスが作成されていることを確認します。

```
$ oc get svc -n <namespace> | grep headless
```

出力例

```
cluster-collector-collector-headless          ClusterIP   None           <none>        9411/TCP
7m28s
jaeger-all-in-one-inmemory-collector-headless   ClusterIP   None           <none>
9411/TCP,14250/TCP,14267/TCP,14268/TCP   16m
```

これらのサービスは、Jaeger、Knative Serving、および Knative Eventing を設定するのに使用されます。Jaeger サービスの名前は異なる場合があります。

4. OpenShift Serverless Operator のインストールのドキュメントに従って、OpenShift Serverless Operator をインストールします。

5. 以下の **KnativeServing** CR を作成して Knative Serving をインストールします。

KnativeServing CR の例

```
apiVersion: operator.knative.dev/v1beta1
kind: KnativeServing
metadata:
    name: knative-serving
    namespace: knative-serving
spec:
  config:
    tracing:
      backend: "zipkin"
      zipkin-endpoint: "http://cluster-collector-collector-headless.tracing-
system.svc:9411/api/v2/spans"
      debug: "false"
      sample-rate: "0.1" 1
```

1 **sample-rate** はサンプリングの可能性を定義します。**sample-rate: "0.1"** を使用すると、10 トレースの 1 つがサンプリングされます。

6. 次の **KnativeEventing** CR を作成して、Knative Eventing をインストールします。

KnativeEventing CR の例

```
apiVersion: operator.knative.dev/v1beta1
kind: KnativeEventing
metadata:
    name: knative-eventing
    namespace: knative-eventing
spec:
  config:
    tracing:
      backend: "zipkin"
      zipkin-endpoint: "http://cluster-collector-collector-headless.tracing-
```

```
system.svc:9411/api/v2/spans"
      debug: "false"
      sample-rate: "0.1" (1)
```

(1) **sample-rate** はサンプリングの可能性を定義します。**sample-rate: "0.1"** を使用すると、10 トレースの 1 つがサンプリングされます。

7. Knative サービスを作成します。

サービスの例

```
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  name: helloworld-go
spec:
  template:
    metadata:
      labels:
        app: helloworld-go
      annotations:
        autoscaling.knative.dev/minScale: "1"
        autoscaling.knative.dev/target: "1"
    spec:
      containers:
      - image: quay.io/openshift-knative/helloworld:v1.2
        imagePullPolicy: Always
        resources:
          requests:
            cpu: "200m"
        env:
        - name: TARGET
          value: "Go Sample v1"
```

8. サービスにいくつかのリクエストを行います。

HTTPS 要求の例

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

9. Jaeger Web コンソールの URL を取得します。

コマンドの例

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

Jaeger コンソールを使用してトレースを検証できるようになりました。

4.3. JAEGER 分散トレースの使用

Red Hat OpenShift 分散トレースのすべてのコンポーネントをインストールしたくない場合でも、OpenShift Serverless を使用する OpenShift Container Platform で分散トレースを使用できます。

4.3.1. 分散トレースを有効にするための Jaeger の設定

Jaeger を使用して分散トレースを有効にするには、Jaeger をスタンドアロン統合としてインストールおよび設定する必要があります。

前提条件

- OpenShift Container Platform に対するクラスター管理者権限があるか、Red Hat OpenShift Service on AWS または OpenShift Dedicated に対するクラスターまたは専用管理者権限がある。
- OpenShift Serverless Operator、Knative Serving、および Knative Eventing がインストールされている。
- Red Hat 分散トレースプラットフォーム Operator がインストールされている。
- OpenShift CLI (**oc**) がインストールされている。
- アプリケーションおよび他のワークロードを作成するために、プロジェクトを作成しているか、適切なロールおよびパーミッションが割り当てられたプロジェクトにアクセスできる。

手順

1. 以下を含む **Jaeger** カスタムリソース YAML ファイルを作成し、これを適用します。

   Jaeger CR

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

2. **KnativeServing** CR を編集し、トレース用に YAML 設定を追加して、Knative Serving のトレースを有効にします。

   Serving の YAML のトレース例

   ```
   apiVersion: operator.knative.dev/v1beta1
   kind: KnativeServing
   metadata:
     name: knative-serving
     namespace: knative-serving
   spec:
     config:
       tracing:
         sample-rate: "0.1" 1
         backend: zipkin 2
         zipkin-endpoint: "http://jaeger-collector.default.svc.cluster.local:9411/api/v2/spans" 3
         debug: "false" 4
   ```

   1 **sample-rate** はサンプリングの可能性を定義します。**sample-rate: "0.1"** を使用すると、10 トレースの 1 つがサンプリングされます。

2 **backend** は **zipkin** に設定される必要があります。

3 **zipkin-endpoint** は **jaeger-collector** サービスエンドポイントを参照する必要があります。このエンドポイントを取得するには、Jaeger CR が適用される namespace を置き換えます。

4 デバッグは **false** に設定する必要があります。**debug: "true"** を設定してデバッグモードを有効にして、サンプリングをバイパスしてすべてのスパンがサーバーに送信されるようにします。

3. **KnativeEventing** CR を編集して、Knative Eventing のトレースを有効にします。

Eventing の YAML のトレース例

```
apiVersion: operator.knative.dev/v1beta1
kind: KnativeEventing
metadata:
  name: knative-eventing
  namespace: knative-eventing
spec:
  config:
    tracing:
      sample-rate: "0.1" 1
      backend: zipkin 2
      zipkin-endpoint: "http://jaeger-collector.default.svc.cluster.local:9411/api/v2/spans" 3
      debug: "false" 4
```

1 **sample-rate** はサンプリングの可能性を定義します。**sample-rate: "0.1"** を使用すると、10 トレースの 1 つがサンプリングされます。

2 **backend** を **zipkin** に設定します。

3 **zipkin-endpoint** を **jaeger-collector** サービスエンドポイントに指定する必要があります。このエンドポイントを取得するには、Jaeger CR が適用される namespace を置き換えます。

4 デバッグは **false** に設定する必要があります。**debug: "true"** を設定してデバッグモードを有効にして、サンプリングをバイパスしてすべてのスパンがサーバーに送信されるようにします。

検証

jaeger ルートを使用して Jaeger Web コンソールにアクセスし、追跡データを表示できます。

1. 以下のコマンドを入力して **jaeger** ルートのホスト名を取得します。

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

出力例

```
NAME     HOST/PORT                         PATH   SERVICES       PORT    TERMINATION
WILDCARD
jaeger   jaeger-default.apps.example.com          jaeger-query   <all>   reencrypt     None
```

2. ブラウザーでエンドポイントアドレスを開き、コンソールを表示します。