OpenShift Container Platform 4.11

Red Hat build of OpenTelemetry

Red Hat build of OpenTelemetry

Last Updated: 2024-02-28

OpenShift Container Platform 4.11 Red Hat build of OpenTelemetry

Red Hat build of OpenTelemetry

法律上の通知

Copyright © 2024 Red Hat, Inc.

The text of and illustrations in this document are licensed by Red Hat under a Creative Commons Attribution–Share Alike 3.0 Unported license ("CC-BY-SA"). An explanation of CC-BY-SA is available at
http://creativecommons.org/licenses/by-sa/3.0/
. In accordance with CC-BY-SA, if you distribute this document or an adaptation of it, you must provide the URL for the original version.

Red Hat, as the licensor of this document, waives the right to enforce, and agrees not to assert, Section 4d of CC-BY-SA to the fullest extent permitted by applicable law.

Red Hat, Red Hat Enterprise Linux, the Shadowman logo, the Red Hat logo, JBoss, OpenShift, Fedora, the Infinity logo, and RHCE are trademarks of Red Hat, Inc., registered in the United States and other countries.

Linux ® is the registered trademark of Linus Torvalds in the United States and other countries.

Java ® is a registered trademark of Oracle and/or its affiliates.

XFS ® is a trademark of Silicon Graphics International Corp. or its subsidiaries in the United States and/or other countries.

MySQL ® is a registered trademark of MySQL AB in the United States, the European Union and other countries.

Node.js ® is an official trademark of Joyent. Red Hat is not formally related to or endorsed by the official Joyent Node.js open source or commercial project.

The OpenStack ® Word Mark and OpenStack logo are either registered trademarks/service marks or trademarks/service marks of the OpenStack Foundation, in the United States and other countries and are used with the OpenStack Foundation's permission. We are not affiliated with, endorsed or sponsored by the OpenStack Foundation, or the OpenStack community.

All other trademarks are the property of their respective owners.

概要

オープンソース の Red Hat build of OpenTelemetry プロジェクトを使用して、クラウドネイティブソフトウェア用に統合かつ標準化された、ベンダー中立のテレメトリーデータを収集します。

目次

第1章 RED HAT BUILD OF OPENTELEMETRY のリリースノート ... 3
1.1. RED HAT BUILD OF OPENTELEMETRY の概要 3
1.2. RED HAT BUILD OF OPENTELEMETRY 3.0 3
1.3. サポート 5
1.4. 多様性を受け入れるオープンソースの強化 6

第2章 RED HAT BUILD OF OPENTELEMETRY のインストール ... 7
2.1. WEB コンソールからの RED HAT BUILD OF OPENTELEMETRY のインストール 7
2.2. CLI を使用した RED HAT BUILD OF OPENTELEMETRY のインストール 9
2.3. 関連情報 11

第3章 RED HAT BUILD OF OPENTELEMETRY の設定とデプロイ ... 12
3.1. OPENTELEMETRY COLLECTOR 設定オプション 12
3.2. OPENTELEMETRY COLLECTOR を使用したさまざまなクラスターからの可観測性データの収集 39
3.3. 監視スタックにメトリクスを送信するための設定 44
3.4. RED HAT BUILD OF OPENTELEMETRY のモニタリングのセットアップ 45
3.5. 関連情報 46

第4章 OPENTELEMETRY インストルメンテーション注入の設定とデプロイ ... 47
4.1. OPENTELEMETRY インストルメンテーション設定オプション 47

第5章 RED HAT BUILD OF OPENTELEMETRY の使用 ... 54
5.1. OPENTELEMETRY COLLECTOR を使用してトレースを TEMPOSTACK に転送する 54
5.2. トレースとメトリクスを OPENTELEMETRY COLLECTOR に送信する 56

第6章 RED HAT BUILD OF OPENTELEMETRY のトラブルシューティング ... 62
6.1. OPENTELEMETRY COLLECTOR ログの取得 62
6.2. メトリクスの公開 62
6.3. デバッグエクスポーター 63

第7章 DISTRIBUTED TRACING PLATFORM (JAEGER) から RED HAT BUILD OF OPENTELEMETRY への移行 64
7.1. DISTRIBUTED TRACING PLATFORM (JAEGER) から、サイドカーのある RED HAT BUILD OF OPENTELEMETRY への移行 64
7.2. DISTRIBUTED TRACING PLATFORM (JAEGER) からサイドカーなしの RED HAT BUILD OF OPENTELEMETRY への移行 66

第8章 RED HAT BUILD OF OPENTELEMETRY の更新 ... 69
8.1. 関連情報 69

第9章 RED HAT BUILD OF OPENTELEMETRY の削除 ... 70
9.1. WEB コンソールを使用した OPENTELEMETRY COLLECTOR インスタンスの削除 70
9.2. CLI を使用した OPENTELEMETRY COLLECTOR インスタンスの削除 70
9.3. 関連情報 71

第1章 RED HAT BUILD OF OPENTELEMETRY のリリースノート

1.1. RED HAT BUILD OF OPENTELEMETRY の概要

Red Hat build of OpenTelemetry は、オープンソースの OpenTelemetry プロジェクト に基づいており、クラウドネイティブソフトウェア用に統合かつ標準化された、ベンダー中立のテレメトリーデータ収集を提供することを目的としています。Red Hat build of OpenTelemetry 製品は、OpenTelemetry Collector のデプロイと管理、およびワークロードインストルメンテーションの簡素化に対するサポートを提供します。

OpenTelemetry Collector は、テレメトリーデータを複数の形式で受信、処理、転送できるため、テレメトリー処理とテレメトリーシステム間の相互運用性にとって理想的なコンポーネントとなります。Collector は、メトリクス、トレース、ログを収集および処理するための統合ソリューションを提供します。

OpenTelemetry Collector には、次のような多くの機能があります。

データ収集および処理ハブ

これは、さまざまなソースからメトリクスやトレースなどのテレメトリーデータを収集する中心的なコンポーネントとして機能します。このデータは、インストルメントされたアプリケーションとインフラストラクチャーから作成できます。

カスタマイズ可能なテレメトリーデータパイプライン

OpenTelemetry Collector はカスタマイズできるように設計されています。さまざまなプロセッサー、エクスポーター、レシーバーをサポートします。

自動インストルメンテーション機能

自動インストルメンテーションにより、アプリケーションに可観測性を追加するプロセスが簡素化されます。開発者は、基本的なテレメトリーデータのコードを手動でインストルメントする必要はありません。

OpenTelemetry Collector の使用例の一部を次に示します。

一元的なデータ収集

マイクロサービスアーキテクチャーでは、Collector をデプロイして、複数のサービスからデータを集約できます。

データの補完と処理

データを分析ツールに転送する前に、Collector はこのデータを強化、フィルタリング、および処理できます。

マルチバックエンドの受信とエクスポート

Collector は、複数の監視および分析プラットフォームに同時にデータを送受信できます。

1.2. RED HAT BUILD OF OPENTELEMETRY 3.0

Red Hat build of OpenTelemetry 3.0 は OpenTelemetry 0.89.0 に基づいています。

1.2.1. 新機能および機能拡張

この更新では、次の機能拡張が導入されています。

- **OpenShift distributed tracing data collection Operator** は、**Red Hat build of OpenTelemetry Operator** という名前に変更されました。

- ARM アーキテクチャーのサポート。
- メトリクス収集用の Prometheus レシーバーのサポート。
- トレースとメトリクスを Kafka に送信するための Kafka レシーバーおよびエクスポーターのサポート。
- クラスター全体のプロキシー環境のサポート。
- Prometheus エクスポーターが有効になっている場合、Red Hat build of OpenTelemetry Operatorは Prometheus **ServiceMonitor** カスタムリソースを作成します。
- Operator は、アップストリームの OpenTelemetry 自動インストルメンテーションライブラリーを注入できるようにする **Instrumentation** カスタムリソースを有効にします。

1.2.2. 削除通知

- Red Hat build of OpenTelemetry 3.0 では、Jaeger エクスポーターが削除されました。バグ修正とサポートは、2.9 ライフサイクルの終了までのみ提供されます。Jaeger Collector にデータを送信するための Jaeger エクスポーターの代わりに、OTLP エクスポーターを使用できます。

1.2.3. バグ修正

この更新では、次のバグ修正が導入されています。

- **oc adm category Mirror** CLI コマンドを使用する場合の、非接続環境のサポートが修正されました。

1.2.4. 既知の問題

現在、Red Hat build of OpenTelemetry Operatorのクラスター監視はバグ (TRACING-3761) により無効になっています。このバグにより、クラスター監視およびサービスモニターオブジェクトに必要なラベル **openshift.io/cluster-monitoring=true** が欠落しているため、クラスター監視が Red Hat build of OpenTelemetry Operatorからメトリクスをスクレイピングできなくなっています。

回避策

次のようにクラスター監視を有効にできます。

1. Operator namespace に次のラベルを追加します: **oc label namespace openshift-opentelemetry-operator openshift.io/cluster-monitoring=true**
2. サービスモニター、ロール、およびロールバインディングを作成します。

```
apiVersion: monitoring.coreos.com/v1
kind: ServiceMonitor
metadata:
  name: opentelemetry-operator-controller-manager-metrics-service
  namespace: openshift-opentelemetry-operator
spec:
  endpoints:
  - bearerTokenFile: /var/run/secrets/kubernetes.io/serviceaccount/token
    path: /metrics
    port: https
    scheme: https
    tlsConfig:
```

```
      insecureSkipVerify: true
    selector:
      matchLabels:
        app.kubernetes.io/name: opentelemetry-operator
        control-plane: controller-manager
---
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: otel-operator-prometheus
  namespace: openshift-opentelemetry-operator
  annotations:
    include.release.openshift.io/self-managed-high-availability: "true"
    include.release.openshift.io/single-node-developer: "true"
rules:
- apiGroups:
  - ""
  resources:
  - services
  - endpoints
  - pods
  verbs:
  - get
  - list
  - watch
---
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: otel-operator-prometheus
  namespace: openshift-opentelemetry-operator
  annotations:
    include.release.openshift.io/self-managed-high-availability: "true"
    include.release.openshift.io/single-node-developer: "true"
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: Role
  name: otel-operator-prometheus
subjects:
- kind: ServiceAccount
  name: prometheus-k8s
  namespace: openshift-monitoring
```

1.3. サポート

本書で説明されている手順、または OpenShift Container Platform で問題が発生した場合は、Red Hat カスタマーポータル にアクセスしてください。カスタマーポータルでは、次のことができます。

- Red Hat 製品に関するアーティクルおよびソリューションを対象とした Red Hat ナレッジベースの検索またはブラウズ。
- Red Hat サポートに対するサポートケースの送信。
- その他の製品ドキュメントへのアクセス。

クラスターの問題を特定するには、OpenShift Cluster Manager Hybrid Cloud Console で Insights を使用できます。Insights により、問題の詳細と、利用可能な場合は問題の解決方法に関する情報が提供されます。

本書の改善への提案がある場合、またはエラーを見つけた場合は、最も関連性の高いドキュメントコンポーネントの Jira Issue を送信してください。セクション名や OpenShift Container Platform バージョンなどの具体的な情報を提供してください。

1.4. 多様性を受け入れるオープンソースの強化

Red Hat では、コード、ドキュメント、Web プロパティーにおける配慮に欠ける用語の置き換えに取り組んでいます。まずは、マスター (master)、スレーブ (slave)、ブラックリスト (blacklist)、ホワイトリスト (whitelist) の 4 つの用語の置き換えから始めます。この取り組みは膨大な作業を要するため、今後の複数のリリースで段階的に用語の置き換えを実施して参ります。詳細は、Red Hat CTO である Chris Wright のメッセージ をご覧ください。

第2章 RED HAT BUILD OF OPENTELEMETRY のインストール

Red Hat build of OpenTelemetry をインストールするには、次の手順を実行します。

1. Red Hat build of OpenTelemetry Operatorをインストールします。
2. OpenTelemetry Collector インスタンスの namespace を作成します。
3. **OpenTelemetryCollector** カスタムリソースを作成して、OpenTelemetry Collector インスタンスをデプロイします。

2.1. WEB コンソールからの RED HAT BUILD OF OPENTELEMETRY のインストール

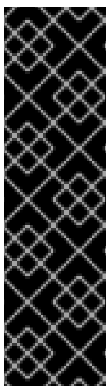
Red Hat build of OpenTelemetry は、Web コンソールの **Administrator** ビューからインストールできます。

前提条件

- **cluster-admin** ロールを持つクラスター管理者として Web コンソールにログインしている。
- Red Hat OpenShift Dedicated の場合、**dedicated-admin** ロールを持つアカウントを使用してログインしている。

手順

1. Red Hat build of OpenTelemetry Operatorをインストールします。
   a. **Operators** → **OperatorHub** に移動し、**Red Hat build of OpenTelemetry Operator** を検索します。
   b. **Red Hat** が提供する **Red Hat build of OpenTelemetry Operator**を選択し、**Install** → **Install** → **View Operator** と進みます。



重要

デフォルトのプリセットで Operator がインストールされます。

- **Update channel** → **stable**
- **Installation mode** → **All namespaces on the cluster**
- **Installed Namespace** → **openshift-operators**
- **Update approval** → **Automatic**

   c. インストール済み Operator ページの **Details** タブの **ClusterServiceVersion details** で、インストールの **Status** が **Succeeded** であることを確認します。

2. **Home** → **Projects** → **Create Project** に移動して、次の手順で作成する **OpenTelemetry Collector** インスタンスの任意のプロジェクトを作成します。
3. **OpenTelemetry Collector** インスタンスを作成します。
   a. **Operators** → **Installed Operators** に移動します。

b. **OpenTelemetry Collector** → **Create OpenTelemetry Collector** → **YAML view** を選択します。

c. **YAML view** で、OTLP、Jaeger、Zipkin レシーバー、およびデバッグエクスポーターを使用して **OpenTelemetryCollector** カスタムリソース (CR) をカスタマイズします。

```yaml
apiVersion: opentelemetry.io/v1alpha1
kind: OpenTelemetryCollector
metadata:
  name: otel
  namespace: <project_of_opentelemetry_collector_instance>
spec:
  mode: deployment
  config: |
    receivers:
      otlp:
        protocols:
          grpc:
          http:
      jaeger:
        protocols:
          grpc:
          thrift_binary:
          thrift_compact:
          thrift_http:
      zipkin:
    processors:
      batch:
      memory_limiter:
        check_interval: 1s
        limit_percentage: 50
        spike_limit_percentage: 30
    exporters:
      debug:
    service:
      pipelines:
        traces:
          receivers: [otlp,jaeger,zipkin]
          processors: [memory_limiter,batch]
          exporters: [debug]
```

d. **Create** を選択します。

検証

1. **Project:** ドロップダウンリストを使用して、**OpenTelemetry Collector** インスタンスのプロジェクトを選択します。

2. **Operators** → **Installed Operators** に移動して、**OpenTelemetry Collector** インスタンスの **ステータス** が **Condition: Ready** であることを確認します。

3. **Workloads** → **Pods** に移動して、**OpenTelemetry Collector** インスタンスのすべてのコンポーネント Pod が実行されていることを確認します。

2.2. CLI を使用した RED HAT BUILD OF OPENTELEMETRY のインストール

Red Hat build of OpenTelemetry はコマンドラインからインストールできます。

前提条件

- **cluster-admin** ロールを持つクラスター管理者によるアクティブな OpenShift CLI (**oc**) セッション。

  ヒント

  - OpenShift CLI (**oc**)のバージョンが最新であり、OpenShift Container Platform バージョンと一致していることを確認してください。
  - **oc login** を実行します。

    ```
    $ oc login --username=<your_username>
    ```

手順

1. Red Hat build of OpenTelemetry Operatorをインストールします。

   a. 次のコマンドを実行して、Red Hat build of OpenTelemetry Operatorのプロジェクトを作成します。

   ```
   $ oc apply -f - << EOF
   apiVersion: project.openshift.io/v1
   kind: Project
   metadata:
     labels:
       kubernetes.io/metadata.name: openshift-opentelemetry-operator
       openshift.io/cluster-monitoring: "true"
     name: openshift-opentelemetry-operator
   EOF
   ```

   b. 以下のコマンドを実行して、Operator グループを作成します。

   ```
   $ oc apply -f - << EOF
   apiVersion: operators.coreos.com/v1
   kind: OperatorGroup
   metadata:
     name: openshift-opentelemetry-operator
     namespace: openshift-opentelemetry-operator
   spec:
     upgradeStrategy: Default
   EOF
   ```

   c. 以下のコマンドを実行して、サブスクリプションを作成します。

   ```
   $ oc apply -f - << EOF
   apiVersion: operators.coreos.com/v1alpha1
   kind: Subscription
   metadata:
   ```

```
  name: opentelemetry-product
  namespace: openshift-opentelemetry-operator
spec:
  channel: stable
  installPlanApproval: Automatic
  name: opentelemetry-product
  source: redhat-operators
  sourceNamespace: openshift-marketplace
EOF
```

d. 次のコマンドを実行して、Operator のステータスを確認します。

```
$ oc get csv -n openshift-opentelemetry-operator
```

2. 後続の手順で作成する OpenTelemetry Collector インスタンス用に選択したプロジェクトを作成します。

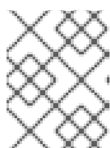
   - メタデータなしでプロジェクトを作成するには、次のコマンドを実行します。

```
$ oc new-project <project_of_opentelemetry_collector_instance>
```

   - メタデータを含むプロジェクトを作成するには、次のコマンドを実行します。

```
$ oc apply -f - << EOF
apiVersion: project.openshift.io/v1
kind: Project
metadata:
  name: <project_of_opentelemetry_collector_instance>
EOF
```

3. OpenTelemetry Collector 用に作成したプロジェクトに OpenTelemetry Collector インスタンスを作成します。



注記

同じクラスター上の別々のプロジェクトに複数の OpenTelemetry Collector インスタンスを作成できます。

a. OTLP、Jaeger、Zipkin レシーバーとデバッグエクスポーターを使用して、**OpenTelemetry Collector** カスタムリソース (CR) をカスタマイズします。

```
apiVersion: opentelemetry.io/v1alpha1
kind: OpenTelemetryCollector
metadata:
  name: otel
  namespace: <project_of_opentelemetry_collector_instance>
spec:
  mode: deployment
  config: |
    receivers:
      otlp:
        protocols:
          grpc:
          http:
```

```
      jaeger:
        protocols:
          grpc:
          thrift_binary:
          thrift_compact:
          thrift_http:
      zipkin:
    processors:
      batch:
      memory_limiter:
        check_interval: 1s
        limit_percentage: 50
        spike_limit_percentage: 30
    exporters:
      debug:
    service:
      pipelines:
        traces:
          receivers: [otlp,jaeger,zipkin]
          processors: [memory_limiter,batch]
          exporters: [debug]
```

b. 次のコマンドを実行して、カスタマイズされた CR を適用します。

```
$ oc apply -f - << EOF
<OpenTelemetryCollector_custom_resource>
EOF
```

検証

1. 次のコマンドを実行して、OpenTelemetry Collector Pod の **status.phase** が **Running** で、**conditions** が **type: Ready** であることを確認します。

```
$ oc get pod -l app.kubernetes.io/managed-by=opentelemetry-operator,app.kubernetes.io/instance=<namespace>.<instance_name> -o yaml
```

2. 次のコマンドを実行して、OpenTelemetry Collector サービスを取得します。

```
$ oc get service -l app.kubernetes.io/managed-by=opentelemetry-operator,app.kubernetes.io/instance=<namespace>.<instance_name>
```

2.3. 関連情報

- クラスター管理者の作成
- OperatorHub.io
- Web コンソールへのアクセス
- Web コンソールを使用した OperatorHub からのインストール
- インストールされた Operator からのアプリケーションの作成
- OpenShift CLI の使用を開始

第3章 RED HAT BUILD OF OPENTELEMETRY の設定とデプロイ

Red Hat build of OpenTelemetry Operatorは、Red Hat build of OpenTelemetry リソースを作成およびデプロイするときに使用されるアーキテクチャーと設定を定義するカスタムリソース定義 (CRD) ファイルを使用します。デフォルト設定をインストールするか、ファイルを変更できます。

3.1. OPENTELEMETRY COLLECTOR 設定オプション

OpenTelemetry Collector は、テレメトリーデータにアクセスする 5 種類のコンポーネントで構成されます。

レシーバー

レシーバー (プッシュまたはプルベース) は、データが Collector に到達する方法です。通常、レシーバーは指定された形式のデータを受け入れて内部形式に変換し、それを適用可能なパイプラインで定義されるプロセッサーおよびエクスポーターに渡します。デフォルトでは、レシーバーは設定されていません。1 つまたは複数のレシーバーを設定する必要があります。レシーバーは 1 つまたは複数のデータソースをサポートする場合があります。

プロセッサー

オプション: プロセッサーは、データを受信してからエクスポートするまでにデータを処理します。デフォルトでは、プロセッサーは有効になっていません。プロセッサーは、すべてのデータソースに対して有効にする必要があります。すべてのプロセッサーがすべてのデータソースをサポートするわけではありません。データソースによっては、複数のプロセッサーが有効になっている可能性があります。プロセッサーの順序が重要であることに注意してください。

エクスポーター

エクスポーター (プッシュまたはプルベース) は、データを 1 つ以上のバックエンドまたは宛先に送信する方法です。デフォルトでは、エクスポーターは設定されていません。1 つまたは複数のエクスポーターを設定する必要があります。エクスポーターは 1 つ以上のデータソースをサポートできます。エクスポーターはデフォルト設定で使用できますが、多くの場合、少なくとも宛先およびセキュリティー設定を指定するための設定が必要です。

コネクター

コネクターは 2 つのパイプラインを接続します。1 つのパイプラインの終了時にエクスポーターとしてデータを消費し、別のパイプラインの開始時にレシーバーとしてデータを出力します。同じまたは異なるデータ型のデータを消費および出力できます。データを生成および出力して、消費されたデータを要約することも、単にデータを複製またはルーティングすることもできます。

エクステンション

エクステンションにより、Collector に機能が追加されます。たとえば、認証をレシーバーとエクスポーターに自動的に追加できます。

カスタムリソース YAML ファイルで、コンポーネントのインスタンスを複数定義できます。コンポーネントは、設定した後に YAML ファイルの **spec.config.service** セクションで定義されたパイプラインで有効にする必要があります。ベストプラクティスとしては、必要なコンポーネントのみを有効にします。

OpenTelemetry Collector カスタムリソースファイルの例

```
apiVersion: opentelemetry.io/v1alpha1
kind: OpenTelemetryCollector
metadata:
  name: cluster-collector
  namespace: tracing-system
spec:
  mode: deployment
```

```
observability:
  metrics:
    enableMetrics: true
config: |
  receivers:
    otlp:
      protocols:
        grpc:
        http:
  processors:
  exporters:
    otlp:
      endpoint: jaeger-production-collector-headless.tracing-system.svc:4317
      tls:
        ca_file: "/var/run/secrets/kubernetes.io/serviceaccount/service-ca.crt"
    prometheus:
      endpoint: 0.0.0.0:8889
      resource_to_telemetry_conversion:
        enabled: true # by default resource attributes are dropped
  service: (1)
    pipelines:
      traces:
        receivers: [otlp]
        processors: []
        exporters: [jaeger]
      metrics:
        receivers: [otlp]
        processors: []
        exporters: [prometheus]
```

1. コンポーネントが設定されていても、**service** セクションで定義されていない場合、そのコンポーネントは有効になりません。

表3.1 Operator が OpenTelemetry Collector を定義するために使用するパラメーター

パラメーター	説明	値	デフォルト
`receivers:`	レシーバーは、データが Collector に到達する方法です。デフォルトでは、レシーバーは設定されていません。設定が有効とみなされるためには、少なくとも 1 つの有効なレシーバーが必要です。レシーバーは、パイプラインに追加して有効にされます。	**otlp**、**jaeger**、**prometheus**、**zipkin**、**kafka**、**opencensus**	None

パラメーター	説明	値	デフォルト
processors:	プロセッサーは、データの受信とエクスポートの間にデータを処理します。デフォルトでは、プロセッサーは有効になっていません。	**batch、memory_limiter、resourcedetection、attributes、span、k8sattributes、filter、routing**	None
exporters:	エクスポーターは、1つ以上のバックエンドまたは宛先にデータを送信します。デフォルトでは、エクスポーターは設定されていません。設定が有効とみなされるためには、少なくとも1つの有効なエクスポーターが必要です。エクスポーターは、パイプラインに追加して有効にされます。エクスポーターはデフォルト設定で使用できますが、多くの場合、少なくとも宛先およびセキュリティー設定を指定するための設定が必要です。	**otlp、otlphttp、debug、prometheus、kafka**	None
connectors:	コネクターはパイプラインのペアを結合します。つまり、パイプラインの終わりのエクスポーターとしてデータを消費し、パイプラインの開始レシーバーとしてデータを出力します。これを使用して、消費されたデータを要約、複製、またはルーティングすることができます。	**spanmetrics**	None
extensions:	テレメトリーデータの処理を含まないタスク用のオプションのコンポーネント。	**bearertokenauth、oauth2client、jaegerremotesamplin、pprof、health_check、memory_ballast、zpages**	None

パラメーター	説明	値	デフォルト
service: pipelines:	コンポーネントは、それらを **services.pipeline** セクションのパイプラインに追加して有効にされます。		

service:
  pipelines:
    traces:
      receivers:
	レシーバーは、それらを **service.pipelines.traces** セクションに追加してトレース用に有効にします。		None

service:
  pipelines:
    traces:
      processors:
	プロセッサーは、それらを **service.pipelines.traces** セクションに追加してトレース用に有効にします。		None

service:
  pipelines:
    traces:
      exporters:
	エクスポーターは、それらを **service.pipelines.traces** セクションに追加してトレース用に有効にします。		None

service:
  pipelines:
    metrics:
      receivers:
	メトリクスのレシーバーを有効にするには、**service.pipelines.metrics** の下に追加します。		None

service:
  pipelines:
    metrics:
      processors:
	メトリクスのプロセッサーを有効にするには、**service.pipelines.metrics** の下に追加します。		None

service:
  pipelines:
    metrics:
      exporters:
	メトリクスのエクスポーターを有効にするには、**service.pipelines.metrics** の下に追加します。		None

3.1.1. OpenTelemetry Collector コンポーネント

3.1.1.1. レシーバー

レシーバーはデータをCollector に入れます。

3.1.1.1.1. OTLP レシーバー

OTLP レシーバーは、OpenTelemetry Protocol (OTLP) を使用してトレースとメトリクスを取り込みます。

OTLP レシーバーが有効になっている OpenTelemetry Collector カスタムリソース

```
config: |
  receivers:
    otlp:
      protocols:
        grpc:
          endpoint: 0.0.0.0:4317 (1)
          tls: (2)
            ca_file: ca.pem
            cert_file: cert.pem
            key_file: key.pem
            client_ca_file: client.pem (3)
            reload_interval: 1h (4)
        http:
          endpoint: 0.0.0.0:4318 (5)
          tls: (6)

  service:
    pipelines:
      traces:
        receivers: [otlp]
      metrics:
        receivers: [otlp]
```

1. OTLP gRPC エンドポイント。省略した場合、デフォルトの **0.0.0.0:4317** が使用されます。
2. サーバー側の TLS 設定。TLS 証明書へのパスを定義します。省略した場合、TLS は無効になります。
3. サーバーがクライアント証明書を検証する TLS 証明書へのパス。これにより、**TLSConfig** で **ClientCAs** および **ClientAuth** の値が **RequireAndVerifyClientCert** に設定されます。詳細は、**Config** of the Golang TLS package を参照してください。
4. 証明書をリロードする間隔を指定します。この値が設定されていない場合、証明書はリロードされません。**reload_interval** は、**ns**、**us** (または **µs**)、**ms**、**s**、**m**、**h** などの有効な時間単位を含む文字列を受け入れます。
5. OTLP HTTP エンドポイント。デフォルト値は **0.0.0.0:4318** です。
6. サーバー側の TLS 設定。詳細は、**grpc** プロトコル設定セクションを参照してください。

3.1.1.1.2. Jaeger レシーバー

Jaeger レシーバーは、Jaeger 形式でトレースを取り込みます。

Jaeger レシーバーが有効になっている OpenTelemetry Collector カスタムリソース

```
config: |
  receivers:
    jaeger:
      protocols:
        grpc:
          endpoint: 0.0.0.0:14250 (1)
        thrift_http:
          endpoint: 0.0.0.0:14268 (2)
        thrift_compact:
          endpoint: 0.0.0.0:6831 (3)
        thrift_binary:
          endpoint: 0.0.0.0:6832 (4)
        tls: (5)

  service:
    pipelines:
      traces:
        receivers: [jaeger]
```

1. Jaeger gRPC エンドポイント。省略した場合、デフォルトの **0.0.0.0:14250** が使用されます。
2. Jaeger Thrift HTTP エンドポイント。省略した場合、デフォルトの **0.0.0.0:14268** が使用されます。
3. Jaeger Thrift Compact エンドポイント。省略した場合、デフォルトの **0.0.0.0:6831** が使用されます。
4. Jaeger Thrift Binary エンドポイント。省略した場合、デフォルトの **0.0.0.0:6832** が使用されます。
5. サーバー側の TLS 設定。詳細は、OTLP レシーバー設定セクションを参照してください。

3.1.1.1.3. Prometheus レシーバー

Prometheus レシーバーは現在、テクノロジープレビュー 機能のみとなっています。

Prometheus レシーバーはメトリクスエンドポイントをスクレイプします。

Prometheus レシーバーが有効になっている OpenTelemetry Collector カスタムリソース

```
config: |
  receivers:
      prometheus:
        config:
          scrape_configs: (1)
            - job_name: 'my-app' (2)
              scrape_interval: 5s (3)
              static_configs:
                - targets: ['my-app.example.svc.cluster.local:8888'] (4)
  service:
```

```
    pipelines:
      metrics:
        receivers: [prometheus]
```

1. Prometheus 形式を使用して設定をスクレイプします。
2. Prometheus のジョブ名。
3. メトリクスデータをスクレイプする間隔。時間単位を受け入れます。デフォルト値は **1m** です。
4. メトリクスが公開されるターゲット。この例では、**example** プロジェクトの **my-app** アプリケーションからメトリクスをスクレイプします。

3.1.1.1.4. Zipkin レシーバー

Zipkin レシーバーは、Zipkin v1 および v2 形式でトレースを取り込みます。

Zipkin レシーバーが有効になっている OpenTelemetry Collector カスタムリソース

```
config: |
  receivers:
    zipkin:
      endpoint: 0.0.0.0:9411 # (1)
      tls: # (2)

  service:
    pipelines:
      traces:
        receivers: [zipkin]
```

1. Zipkin HTTP エンドポイント。省略した場合、デフォルトの **0.0.0.0:9411** が使用されます。
2. サーバー側の TLS 設定。詳細は、OTLP レシーバー設定セクションを参照してください。

3.1.1.1.5. Kafka レシーバー

Kafka レシーバーは現在、テクノロジープレビュー 機能のみとなっています。

Kafka レシーバーは、Kafka からトレース、メトリクス、ログを OTLP 形式で受信します。

Kafka レシーバーが有効になっている OpenTelemetry Collector カスタムリソース

```
config: |
  receivers:
    kafka:
      brokers: ["localhost:9092"] # (1)
      protocol_version: 2.0.0 # (2)
      topic: otlp_spans # (3)
      auth:
        plain_text: # (4)
          username: example
          password: example
```

```
        tls: (5)
          ca_file: ca.pem
          cert_file: cert.pem
          key_file: key.pem
          insecure: false (6)
          server_name_override: kafka.example.corp (7)
  service:
    pipelines:
      traces:
        receivers: [kafka]
```

1. Kafka ブローカーのリスト。デフォルトは **localhost:9092** です。
2. Kafka プロトコルのバージョン。たとえば、**2.0.0** などです。これは必須フィールドです。
3. 読み取り元の Kafka トピックの名前。デフォルトは **otlp_spans** です。
4. 平文認証設定。省略した場合、平文認証は無効になります。
5. クライアント側の TLS 設定。TLS 証明書へのパスを定義します。省略した場合、TLS 認証は無効になります。
6. サーバーの証明書チェーンとホスト名の検証を無効にします。デフォルトは **false** です。
7. ServerName は、仮想ホスティングをサポートするためにクライアントによって要求されたサーバーの名前を示します。

3.1.1.1.6. OpenCensus レシーバー

OpenCensus レシーバーは、OpenCensus プロジェクトとの下位互換性を提供し、インストルメント化されたコードベースの移行を容易にします。gRPC または HTTP および Json を介して OpenCensus 形式でメトリクスとトレースを受信します。

OpenCensus レシーバーが有効になっている OpenTelemetry Collector カスタムリソース

```
config: |
  receivers:
    opencensus:
      endpoint: 0.0.0.0:9411 (1)
      tls: (2)
      cors_allowed_origins: (3)
        - https://*.<example>.com
  service:
    pipelines:
      traces:
        receivers: [opencensus]
        ...
```

1. OpenCensus エンドポイント。省略した場合、デフォルトは **0.0.0.0:55678** です。
2. サーバー側の TLS 設定。詳細は、OTLP レシーバー設定セクションを参照してください。
3. HTTP JSON エンドポイントを使用して、オプションで CORS を設定することもできます。これ

3.1.1.2. プロセッサー

プロセッサーは、データの受信とエクスポートの間にデータを処理します。

3.1.1.2.1. Batch プロセッサー

Batch プロセッサーは、トレースとメトリクスをバッチ処理して、テレメトリー情報の転送に必要な発信接続の数を減らします。

Batch プロセッサーを使用する場合の OpenTelemetry Collector カスタムリソースの例

```
config: |
  processor:
    batch:
      timeout: 5s
      send_batch_max_size: 10000
  service:
    pipelines:
      traces:
        processors: [batch]
      metrics:
        processors: [batch]
```

表3.2 Batch プロセッサーで使用されるパラメーター

パラメーター	説明	デフォルト
timeout	バッチサイズに関係なく、特定の期間後にバッチを送信します。	**200ms**
send_batch_size	指定された数のスパンまたはメトリクスの後に、Telemetry データのバッチを送信します。	**8192**
send_batch_max_size	バッチの最大許容サイズ。**send_batch_size** 以上である必要があります。	**0**
metadata_keys	アクティブにすると、**client.Metadata** で見つかった一意の値セットごとにバッチャーインスタンスが作成されます。	**[]**
metadata_cardinality_limit	**metadata_keys** を設定すると、プロセス中に処理されるメタデータのキーと値の組み合わせの数が制限されます。	**1000**

3.1.1.2.2. Memory Limiter プロセッサー

Memory Limiter プロセッサーは、Collector のメモリー使用量を定期的にチェックし、ソフトメモリー

リミットに達するとデータ処理を一時停止します。このプロセッサーは、トレース、メトリクス、およびログをサポートします。先行コンポーネント (通常はレシーバー) は、同じデータの送信を再試行することが想定されており、受信データにバックプレッシャーを適用する場合があります。メモリー使用量がハードリミットを超えると、Memory Limiter プロセッサーによってガベージコレクションが強制的に実行されます。

Memory Limiter プロセッサーを使用する場合の OpenTelemetry Collector カスタムリソースの例

```
config: |
  processor:
    memory_limiter:
      check_interval: 1s
      limit_mib: 4000
      spike_limit_mib: 800
  service:
    pipelines:
      traces:
        processors: [batch]
      metrics:
        processors: [batch]
```

表3.3 Memory Limiter プロセッサーによって使用されるパラメーター

パラメーター	説明	デフォルト
check_interval	メモリー使用量の測定間の時間。最適な値は **1s** です。トラフィックが急増するパターンの場合は、**check_interval** を減らすか、**spike_limit_mib** を増やすことができます。	**0s**
limit_mib	ハードリミット。ヒープに割り当てられるメモリーの最大量 (MiB 単位)。通常、OpenTelemetry Collector の合計メモリー使用量は、この値より約 50 MiB 大きくなります。	**0**
spike_limit_mib	スパイクリミット。これは、予想されるメモリー使用量の最大スパイク (MiB 単位) です。最適な値は、**limit_mib** の約 20% です。ソフトリミットを計算するには、**limit_mib** から **spike_limit_mib** を減算します。	**limit_mib** の 20%

パラメーター	説明	デフォルト
limit_percentage	**limit_mib** と同じですが、使用可能な合計メモリーのパーセンテージとして表されます。**limit_mib** 設定は、この設定よりも優先されます。	**0**
spike_limit_percentage	**spike_limit_mib** と同じですが、使用可能な合計メモリーのパーセンテージとして表されます。**limit_percentage** 設定と併用することを目的としています。	**0**

3.1.1.2.3. Resource Detection プロセッサー

Resource Detection プロセッサーは現在、テクノロジープレビュー 機能のみとなっています。

Resource Detection プロセッサーは、OpenTelemetry のリソースセマンティック標準に合わせて、ホストリソースの詳細を識別します。検出された情報を使用して、テレメトリーデータ内のリソース値を追加または置換できます。このプロセッサーはトレース、メトリクスをサポートしており、Docket メタデータディテクターや **OTEL_RESOURCE_ATTRIBUTES** 環境変数ディテクターなどの複数のディテクターと併用できます。

Resource Detection プロセッサーに必要な OpenShift Container Platform の権限

```
kind: ClusterRole
metadata:
  name: otel-collector
rules:
- apiGroups: ["config.openshift.io"]
  resources: ["infrastructures", "infrastructures/status"]
  verbs: ["get", "watch", "list"]
```

Resource Detection プロセッサーを使用する OpenTelemetry Collector

```
  config: |
    processor:
      resourcedetection:
        detectors: [openshift]
        override: true
    service:
      pipelines:
        traces:
          processors: [resourcedetection]
        metrics:
          processors: [resourcedetection]
```

環境変数ディテクターを備えた Resource Detection Processor を使用する OpenTelemetry Collector

```
config: |
  processors:
    resourcedetection/env:
      detectors: [env] (1)
      timeout: 2s
      override: false
```

1. 使用するディテクターを指定します。この例では、環境ディテクターが指定されています。

3.1.1.2.4. Attributes プロセッサー

Attributes プロセッサーは現在、テクノロジープレビュー 機能のみとなっています。

Attributes プロセッサーは、スパン、ログ、またはメトリクスの属性を変更できます。入力データをフィルタリングして照合し、特定のアクションに対してそのようなデータを含めたり除外したりするようにこのプロセッサーを設定できます。

プロセッサーはアクションのリストを操作し、設定で指定された順序でアクションを実行します。次のアクションがサポートされています。

Insert
: 指定されたキーがまだ存在しない場合は、入力データに新しい属性を挿入します。

Update
: キーがすでに存在する場合は、入力データの属性を更新します。

Upsert
: 挿入アクションと更新アクションを組み合わせます。キーがまだ存在しない場合は、新しい属性を挿入します。キーがすでに存在する場合は属性を更新します。

Delete
: 入力データから属性を削除します。

Hash
: 既存の属性値を SHA1 としてハッシュします。

Extract
: 正規表現ルールを使用して、ルールで定義された入力キーからターゲットキーまでの値を抽出します。ターゲットキーがすでに存在する場合は、Span プロセッサーの **to_attributes** 設定と同様に、既存の属性をソースとしてオーバーライドされます。

Convert
: 既存の属性を指定された型に変換します。

Attributes プロセッサーを使用した OpenTelemetry Collector

```
config: |
  processors:
    attributes/example:
      actions:
        - key: db.table
          action: delete
        - key: redacted_span
          value: true
          action: upsert
        - key: copy_key
```

```
        from_attribute: key_original
        action: update
      - key: account_id
        value: 2245
        action: insert
      - key: account_password
        action: delete
      - key: account_email
        action: hash
      - key: http.status_code
        action: convert
        converted_type: int
```

3.1.1.2.5. Resource プロセッサー

Resource プロセッサーは現在、テクノロジープレビュー 機能のみとなっています。

Resource プロセッサーは、リソース属性に変更を適用します。このプロセッサーは、トレース、メトリクス、およびログをサポートします。

Resource Detection プロセッサーを使用する OpenTelemetry Collector

```
config: |
  processor:
    attributes:
    - key: cloud.availability_zone
      value: "zone-1"
      action: upsert
    - key: k8s.cluster.name
      from_attribute: k8s-cluster
      action: insert
    - key: redundant-attribute
      action: delete
```

属性は、属性の削除、属性の挿入、または属性のアップサートなど、リソース属性に適用されるアクションを表します。

3.1.1.2.6. Span プロセッサー

Span プロセッサーは現在、テクノロジープレビュー 機能のみとなっています。

Span プロセッサーは、その属性に基づいてスパン名を変更するか、スパン名からスパン属性を抽出します。スパンステータスを変更することもできます。スパンを含めたり除外したりすることもできます。このプロセッサーはトレースをサポートしています。

スパンの名前変更には、**from_attributes** 設定を使用して、新しい名前の属性を指定する必要があります。

Span プロセッサーを使用してスパンの名前を変更する OpenTelemetry Collector

```
config: |
  processor:
    span:
```

```
      name:
        from_attributes: [<key1>, <key2>, ...] (1)
        separator: <value> (2)
```

1. 新しいスパン名を形成するキーを定義します。
2. オプションの区切り文字。

プロセッサーを使用して、スパン名から属性を抽出できます。

Span プロセッサーを使用してスパン名から属性を抽出する OpenTelemetry Collector

```
config: |
  processor:
    span/to_attributes:
      name:
        to_attributes:
          rules:
            - ^\/api\/v1\/document\/(?P<documentId>.*)\/update$ (1)
```

1. このルールは、抽出の実行方法を定義します。さらにルールを定義できます。たとえば、この場合、正規表現が名前と一致すると、**documentID** 属性が作成されます。この例では、入力スパン名が **/api/v1/document/12345678/update** の場合、出力スパン名は **/api/v1/document/{documentId}/update** となり、新しい **"documentId"="12345678"** 属性がスパンに追加されます。

スパンステータスを変更できます。

ステータス変更に Span プロセッサーを使用する OpenTelemetry Collector

```
config: |
  processor:
    span/set_status:
      status:
        code: Error
        description: "<error_description>"
```

3.1.1.2.7. Kubernetes Attributes プロセッサー

Kubernetes Attributes プロセッサーは現在、テクノロジープレビュー 機能のみとなっています。

Kubernetes Attributes プロセッサーでは、Kubernetes メタデータを使用してスパン、メトリクス、およびログリソース属性を自動設定できます。このプロセッサーは、トレース、メトリクス、およびログをサポートします。このプロセッサーは、Kubernetes リソースを自動的に識別し、そこからメタデータを抽出して、この抽出されたメタデータをリソース属性として関連するスパン、メトリクス、ログに組み込みます。Kubernetes API を利用してクラスター内で動作しているすべての Pod を検出し、IP アドレス、Pod UID、およびその他の関連メタデータの記録を維持します。

Kubernetes Attributes プロセッサーに必要な最小限の OpenShift Container Platform 権限

```
kind: ClusterRole
metadata:
```

```
  name: otel-collector
rules:
- apiGroups: ['']
  resources: ['pods', 'namespaces']
  verbs: ['get', 'watch', 'list']
```

Kubernetes Attributes プロセッサーを使用した OpenTelemetry Collector

```
config: |
  processors:
    k8sattributes:
      filter:
        node_from_env_var: KUBE_NODE_NAME
```

3.1.1.3. Filter プロセッサー

Filter プロセッサーは現在、テクノロジープレビュー 機能のみとなっています。

Filter プロセッサーは、OpenTelemetry Transformation Language を活用して、テレメトリーデータを破棄する基準を確立します。これらの条件のいずれかが満たされると、テレメトリーデータは破棄されます。論理 OR 演算子を使用して条件を組み合わせることができます。このプロセッサーは、トレース、メトリクス、およびログをサポートします。

OTLP エクスポーターが有効になっている OpenTelemetry Collector カスタムリソース

```
config: |
  processors:
    filter/ottl:
      error_mode: ignore (1)
      traces:
        span:
          - 'attributes["container.name"] == "app_container_1"' (2)
          - 'resource.attributes["host.name"] == "localhost"' (3)
```

1. エラーモードを定義します。**ignore** に設定すると、条件によって返されたエラーが無視されます。**propagate** に設定すると、エラーがパイプラインに返されます。エラーが発生すると、ペイロードが Collector から削除されます。
2. **container.name == app_container_1** 属性を持つスパンをフィルタリングします。
3. **host.name == localhost** リソース属性を持つスパンをフィルタリングします。

3.1.1.4. Routing プロセッサー

Routing プロセッサーは現在、テクノロジープレビュー 機能のみとなっています。

Routing プロセッサーは、ログ、メトリクス、またはトレースを特定のエクスポーターにルーティングします。このプロセッサーは、受信 HTTP 要求 (gRPC またはプレーン HTTP) からヘッダーを読み取るか、リソース属性を読み取ることができ、読み取った値に従って、関連するエクスポーターにトレース情報を送信します。

OTLP エクスポーターが有効になっている OpenTelemetry Collector カスタムリソース

```
config: |
  processors:
    routing:
      from_attribute: X-Tenant (1)
      default_exporters: (2)
      - jaeger
      table: (3)
      - value: acme
        exporters: [jaeger/acme]
  exporters:
    jaeger:
      endpoint: localhost:14250
    jaeger/acme:
      endpoint: localhost:24250
```

1. ルートを実行するときのルックアップ値の HTTP ヘッダー名。
2. 属性値が次のセクションの表に存在しない場合のデフォルトのエクスポーター。
3. どの値をどのエクスポーターにルーティングするかを定義するテーブル。

オプションで、**from_attribute** 内での属性の検索場所を定義する **attribute_source** 設定を作成できます。許可される値は、HTTP ヘッダーを含むコンテキストを検索する場合は **context**、またはリソース属性を検索する場合は **resource** です。

3.1.1.5. エクスポーター

エクスポーターは、1 つ以上のバックエンドまたは宛先にデータを送信します。

3.1.1.5.1. OTLP エクスポーター

OTLP gRPC エクスポーターは、OpenTelemetry Protocol (OTLP) を使用してトレースとメトリクスをエクスポートします。

OTLP エクスポーターが有効になっている OpenTelemetry Collector カスタムリソース

```
config: |
  exporters:
    otlp:
      endpoint: tempo-ingester:4317 (1)
      tls: (2)
        ca_file: ca.pem
        cert_file: cert.pem
        key_file: key.pem
        insecure: false (3)
        insecure_skip_verify: false # (4)
        reload_interval: 1h (5)
        server_name_override: <name> (6)
      headers: (7)
        X-Scope-OrgID: "dev"
  service:
    pipelines:
      traces:
```

```
      exporters: [otlp]
    metrics:
      exporters: [otlp]
```

1. OTLP gRPC エンドポイント。**https://** スキームが使用される場合、クライアントトランスポートセキュリティーが有効になり、**tls** の **insecure** 設定をオーバーライドします。
2. クライアント側の TLS 設定。TLS 証明書へのパスを定義します。
3. **true** に設定すると、クライアントトランスポートセキュリティーは無効になります。デフォルト値は **false** です。
4. **true** に設定されている場合、証明書の検証は省略します。デフォルト値は **false** です。
5. 証明書をリロードする間隔を指定します。この値が設定されていない場合、証明書はリロードされません。**reload_interval** は、**ns**、**us** (または **µs**)、**ms**、**s**、**m**、**h** などの有効な時間単位を含む文字列を受け入れます。
6. 要求の authority ヘッダーフィールドなど、認証局の仮想ホスト名をオーバーライドします。これをテストに使用できます。
7. ヘッダーは、接続が確立されている間に実行されるすべての要求に対して送信されます。

3.1.1.5.2. OTLP HTTP エクスポーター

OTLP HTTP エクスポーターは、OpenTelemetry Protocol (OTLP) を使用してトレースとメトリクスをエクスポートします。

OTLP エクスポーターが有効になっている OpenTelemetry Collector カスタムリソース

```
config: |
  exporters:
    otlphttp:
      endpoint: http://tempo-ingester:4318 (1)
      tls: (2)
      headers: (3)
        X-Scope-OrgID: "dev"
      disable_keep_alives: false (4)

  service:
    pipelines:
      traces:
        exporters: [otlphttp]
      metrics:
        exporters: [otlphttp]
```

1. OTLP HTTP エンドポイント。**https://** スキームが使用される場合、クライアントトランスポートセキュリティーが有効になり、**tls** の **insecure** 設定をオーバーライドします。
2. クライアント側の TLS 設定。TLS 証明書へのパスを定義します。
3. ヘッダーは、すべての HTTP 要求で送信されます。
4. true の場合、HTTP keep-alives を無効にします。単一の HTTP リクエストに対してのみ、サー

3.1.1.5.3. デバッグエクスポーター

デバッグエクスポーターは、トレースとメトリクスを標準出力に出力します。

デバッグエクスポーターが有効になっている OpenTelemetry Collector カスタムリソース

```
config: |
  exporters:
    debug:
      verbosity: detailed (1)
  service:
    pipelines:
      traces:
        exporters: [logging]
      metrics:
        exporters: [logging]
```

(1) デバッグエクスポートの詳細度: **detailed** または **normal** または **basic**。**detailed** に設定すると、パイプラインデータの詳細がログに記録されます。デフォルトは **normal** です。

3.1.1.5.4. Prometheus エクスポーター

Prometheus エクスポーターは現在、テクノロジープレビュー 機能のみとなっています。

Prometheus エクスポーターは、Prometheus または OpenMetrics 形式でメトリクスをエクスポートします。

Prometheus エクスポーターが有効になっている OpenTelemetry Collector カスタムリソース

```
ports:
- name: promexporter (1)
  port: 8889
  protocol: TCP
config: |
  exporters:
    prometheus:
      endpoint: 0.0.0.0:8889 (2)
      tls: (3)
        ca_file: ca.pem
        cert_file: cert.pem
        key_file: key.pem
      namespace: prefix (4)
      const_labels: (5)
        label1: value1
      enable_open_metrics: true (6)
      resource_to_telemetry_conversion: (7)
        enabled: true
      metric_expiration: 180m (8)
      add_metric_suffixes: false (9)
  service:
    pipelines:
      metrics:
        exporters: [prometheus]
```

1. Collector Pod およびサービスから Prometheus ポートを公開します。**ServiceMonitor** または **PodMonitor** カスタムリソースのポート名を使用して、Prometheus によるメトリクスのスクレイピングを有効にできます。
2. メトリクスが公開されるネットワークエンドポイント。
3. サーバー側の TLS 設定。TLS 証明書へのパスを定義します。
4. 設定されている場合は、提供された値でメトリクスをエクスポートします。デフォルトはありません。
5. エクスポートされたすべてのメトリクスに適用されるキーと値のペアのラベル。デフォルトはありません。
6. **true** の場合、メトリクスは OpenMetrics 形式を使用してエクスポートされます。手本 (exemplar) は、OpenMetrics 形式で、ヒストグラムおよびモノトニックサムメトリクス (**counter** など) に対してのみエクスポートできます。デフォルトでは無効になっています。
7. **enabled** が **true** の場合、すべてのリソース属性はデフォルトでメトリクスラベルに変換されます。デフォルトでは無効になっています。
8. 更新なしでメトリクスが公開される期間を定義します。デフォルトは **5m** です。
9. メトリクスの型と単位の接尾辞を追加します。Jaeger コンソールの監視タブが有効になっている場合は、無効にする必要があります。デフォルトは **true** です。

3.1.1.5.5. Kafka エクスポーター

Kafka エクスポータは現在、テクノロジープレビュー 機能のみとなっています。

Kafka エクスポータは、ログ、メトリクス、およびトレースを Kafka にエクスポートします。このエクスポーターは、メッセージをブロックしてバッチ処理しない同期プロデューサーを使用します。スループットと回復力を高めるには、バッチおよびキュー再試行プロセッサーと併用する必要があります。

Kafka エクスポーターが有効になっている OpenTelemetry Collector カスタムリソース

```
config: |
  exporters:
    kafka:
      brokers: ["localhost:9092"] (1)
      protocol_version: 2.0.0 (2)
      topic: otlp_spans (3)
      auth:
        plain_text: (4)
          username: example
          password: example
        tls: (5)
          ca_file: ca.pem
          cert_file: cert.pem
          key_file: key.pem
          insecure: false (6)
          server_name_override: kafka.example.corp (7)
  service:
```

```
    pipelines:
      traces:
        exporters: [kafka]
```

1. Kafka ブローカーのリスト。デフォルトは **localhost:9092** です。
2. Kafka プロトコルのバージョン。たとえば、**2.0.0** などです。これは必須フィールドです。
3. 読み取り元の Kafka トピックの名前。デフォルトは次のとおりです。トレースの場合は **otlp_spans**、メトリクスの場合は **otlp_metrics**、ログの場合は **otlp_logs** です。
4. 平文認証設定。省略した場合、平文認証は無効になります。
5. クライアント側の TLS 設定。TLS 証明書へのパスを定義します。省略した場合、TLS 認証は無効になります。
6. サーバーの証明書チェーンとホスト名の検証を無効にします。デフォルトは **false** です。
7. ServerName は、仮想ホスティングをサポートするためにクライアントによって要求されたサーバーの名前を示します。

3.1.1.6. コネクター

コネクターは 2 つのパイプラインを接続します。

3.1.1.6.1. Spanmetrics コネクター

Spanmetrics コネクターは現在、テクノロジープレビュー 機能のみとなっています。

Spanmetrics コネクターは、スパンデータからの Request, Error, Duration (R.E.D) OpenTelemetry メトリクスを集計します。

Spanmetrics コネクターが有効になっている OpenTelemetry Collector カスタムリソース

```
  config: |
    connectors:
      spanmetrics:
        metrics_flush_interval: 15s # (1)
    service:
      pipelines:
        traces:
          exporters: [spanmetrics]
        metrics:
          receivers: [spanmetrics]
```

1. 生成されたメトリクスのフラッシュ間隔を定義します。デフォルトは **15s** です。

3.1.1.7. エクステンション

エクステンションにより、Collector に機能が追加されます。

3.1.1.7.1. BearerTokenAuth エクステンション

BearerTokenAuth エクステンションは現在、テクノロジープレビュー 機能のみとなっています。

BearerTokenAuth エクステンションは、HTTP および gRPC プロトコルに基づくレシーバーとエクスポーター用のオーセンティケーターです。OpenTelemetry Collector カスタムリソースを使用して、レシーバーおよびエクスポーター側で BearerTokenAuth エクステンションのクライアント認証とサーバー認証を設定できます。このエクステンションは、トレース、メトリクス、およびログをサポートします。

BearerTokenAuth エクステンション用に設定されたクライアントおよびサーバー認証を含む OpenTelemetry Collector カスタムリソース

```
config: |
  extensions:
    bearertokenauth:
      scheme: "Bearer" (1)
      token: "<token>" (2)
      filename: "<token_file>" (3)

  receivers:
    otlp:
      protocols:
        http:
          auth:
            authenticator: bearertokenauth (4)
  exporters:
    otlp:
      auth:
        authenticator: bearertokenauth (5)

  service:
    extensions: [bearertokenauth]
    pipelines:
      traces:
        receivers: [otlp]
        exporters: [otlp]
```

1. BearerTokenAuth エクステンションを設定して、カスタム **scheme** を送信できます。デフォルトは **Bearer** です。
2. BearerTokenAuth エクステンショントークンをメタデータとして追加して、メッセージを識別できます。
3. すべてのメッセージとともに送信される認証トークンを含むファイルへのパス。
4. オーセンティケーター設定を OTLP レシーバーに割り当てることができます。
5. オーセンティケーター設定を OTLP エクスポーターに割り当てることができます。

3.1.1.7.2. OAuth2Client エクステンション

OAuth2Client エクステンションは現在、テクノロジープレビュー 機能のみとなっています。

OAuth2Client エクステンションは、HTTP および gRPC プロトコルに基づくエクスポーター用のオーセンティケーターです。OAuth2Client エクステンションのクライアント認証は、OpenTelemetry

Collector カスタムリソースの別のセクションで設定されます。このエクステンションは、トレース、メトリクス、およびログをサポートします。

OAuth2Client エクステンション用にクライアント認証が設定された OpenTelemetry Collector カスタムリソース

```
config: |
  extensions:
    oauth2client:
      client_id: <client_id> (1)
      client_secret: <client_secret> (2)
      endpoint_params: (3)
        audience: <audience>
      token_url: https://example.com/oauth2/default/v1/token (4)
      scopes: ["api.metrics"] (5)
      # tls settings for the token client
      tls: (6)
        insecure: true (7)
        ca_file: /var/lib/mycert.pem (8)
        cert_file: <cert_file> (9)
        key_file: <key_file> (10)
      timeout: 2s (11)

  receivers:
    otlp:
      protocols:
        http:

  exporters:
    otlp:
      auth:
        authenticator: oauth2client (12)

  service:
    extensions: [oauth2client]
    pipelines:
      traces:
        receivers: [otlp]
        exporters: [otlp]
```

1. ID プロバイダーによって提供されるクライアント ID。
2. ID プロバイダーに対してクライアントを認証するために使用される機密キー。
3. キーと値のペア形式の追加のメタデータ。認証中に転送されます。たとえば **audience** は、アクセストークンの対象を指定し、トークンの受信者を示します。
4. Collector がアクセストークンを要求する OAuth2 トークンエンドポイントの URL。
5. スコープは、クライアントによって要求された特定の権限またはアクセスレベルを定義します。
6. トークンクライアントの Transport Layer Security (TLS) 設定。トークンを要求するときに安全な接続を確立するために使用されます。

7 **true** に設定すると、安全でないまたは検証されていない TLS 接続を使用して、設定されたトークンエンドポイントを呼び出すようにCollector が設定されます。

8 TLS ハンドシェイク中にサーバーの証明書を検証するために使用される認証局 (CA) ファイルへのパス。

9 クライアントが必要に応じて OAuth2 サーバーに対して自身を認証するために使用する必要があるクライアント証明書ファイルへのパス。

10 認証に必要な場合にクライアント証明書と併用されるクライアントの秘密キーファイルへのパス。

11 トークンクライアントのリクエストのタイムアウトを設定します。

12 オーセンティケーター設定を OTLP エクスポーターに割り当てることができます。

3.1.1.7.3. Jaeger Remote Sampling エクステンション

Jaeger Remote Sampling エクステンションは現在、テクノロジープレビュー 機能のみとなっています。

Jaeger Remote Sampling エクステンションを使用すると、Jaeger のリモートサンプリング API の後にサンプリングストラテジーを提供できるようになります。このエクステンションを設定して、パイプラインの Jaeger Collector などのバッキングリモートサンプリングサーバーに、またはローカルファイルシステムから静的 JSON ファイルにリクエストをプロキシーできます。

設定された Jaeger Remote Sampling エクステンションを備えた OpenTelemetry Collector カスタムリソース

```
config: |
  extensions:
    jaegerremotesampling:
      source:
        reload_interval: 30s (1)
        remote:
          endpoint: jaeger-collector:14250 (2)
        file: /etc/otelcol/sampling_strategies.json (3)

  receivers:
    otlp:
      protocols:
        http:

  exporters:
    otlp:

  service:
    extensions: [jaegerremotesampling]
    pipelines:
      traces:
        receivers: [otlp]
        exporters: [otlp]
```

1 サンプリング設定が更新される時間間隔。

2 Jaeger Remote Sampling ストラテジープロバイダーに到達するためのエンドポイント。

3 JSON 形式のサンプリングストラテジー設定を含むローカルファイルへのパス。

Jaeger Remote Sampling ストラテジーファイルの例

```
{
  "service_strategies": [
    {
      "service": "foo",
      "type": "probabilistic",
      "param": 0.8,
      "operation_strategies": [
        {
          "operation": "op1",
          "type": "probabilistic",
          "param": 0.2
        },
        {
          "operation": "op2",
          "type": "probabilistic",
          "param": 0.4
        }
      ]
    },
    {
      "service": "bar",
      "type": "ratelimiting",
      "param": 5
    }
  ],
  "default_strategy": {
    "type": "probabilistic",
    "param": 0.5,
    "operation_strategies": [
      {
        "operation": "/health",
        "type": "probabilistic",
        "param": 0.0
      },
      {
        "operation": "/metrics",
        "type": "probabilistic",
        "param": 0.0
      }
    ]
  }
}
```

3.1.1.7.4. パフォーマンスプロファイラーエクステンション

Performance Profiler エクステンションは現在、テクノロジープレビュー 機能のみとなっています。

Performance Profiler エクステンションにより、Go **net/http/pprof** エンドポイントが有効になります。これは通常、開発者がパフォーマンスプロファイルを収集し、サービスの問題を調査するために使用されます。

Performance Profiler エクステンションが設定された OpenTelemetry Collector カスタムリソース

```
config: |
  extensions:
    pprof:
      endpoint: localhost:1777 (1)
      block_profile_fraction: 0 (2)
      mutex_profile_fraction: 0 (3)
      save_to_file: test.pprof (4)

  receivers:
    otlp:
      protocols:
        http:

  exporters:
    otlp:

  service:
    extensions: [pprof]
    pipelines:
      traces:
        receivers: [otlp]
        exporters: [otlp]
```

1. このエクステンションがリッスンするエンドポイント。**localhost:** を使用してローカルでのみ使用できるようにするか、**":"** を使用してすべてのネットワークインターフェイスで使用できるようにします。デフォルト値は **localhost:1777** です。
2. ブロッキングイベントの一部がプロファイリングされるように設定します。プロファイリングを無効にするには、これを **0** または負の整数に設定します。**runtime** パッケージについては、ドキュメント を参照してください。デフォルト値は **0** です。
3. プロファイリングされるミューテックス競合イベントの一部を設定します。プロファイリングを無効にするには、これを **0** または負の整数に設定します。**runtime** パッケージについては、ドキュメント を参照してください。デフォルト値は **0** です。
4. CPU プロファイルを保存するファイルの名前。Collector が起動すると、プロファイリングが開始されます。プロファイリングは、Collector の終了時にファイルに保存されます。

3.1.1.7.5. ヘルスチェックエクステンション

Health Check エクステンションは現在、テクノロジープレビュー 機能のみとなっています。

Health Check エクステンションは、OpenTelemetry Collector のステータスをチェックするための HTTP URL を提供します。このエクステンションは、OpenShift の liveness および readiness プローブとして使用できます。

ヘルスチェックエクステンションが設定された OpenTelemetry Collector カスタムリソース

```
config: |
  extensions:
    health_check:
      endpoint: "0.0.0.0:13133" (1)
      tls: (2)
        ca_file: "/path/to/ca.crt"
        cert_file: "/path/to/cert.crt"
        key_file: "/path/to/key.key"
      path: "/health/status" (3)
      check_collector_pipeline: (4)
        enabled: true (5)
        interval: "5m" (6)
        exporter_failure_threshold: 5 (7)

  receivers:
    otlp:
      protocols:
        http:

  exporters:
    otlp:

  service:
    extensions: [health_check]
    pipelines:
      traces:
        receivers: [otlp]
        exporters: [otlp]
```

1. ヘルスチェックステータスを公開するためのターゲット IP アドレス。デフォルトは **0.0.0.0:13133** です。
2. TLS サーバー側の設定。TLS 証明書へのパスを定義します。省略した場合、TLS は無効になります。
3. ヘルスチェックサーバーのパス。デフォルトは / です。
4. Collector パイプラインのヘルスチェック用の設定。
5. Collector パイプラインのヘルスチェックを有効にします。デフォルトは **false** です。
6. 失敗数を確認する時間間隔。デフォルトは **5m** です。
7. コンテナーが依然として正常であるとマークされる失敗の数のしきい値。デフォルトは **5** です。

3.1.1.7.6. Memory Ballast エクステンション

Memory Ballast エクステンションは現在、テクノロジープレビュー 機能のみとなっています。

Memory Ballast エクステンションを使用すると、アプリケーションはプロセスのメモリーバラストを設定できます。

Memory Ballast エクステンションが設定された OpenTelemetry Collector カスタムリソース

```
config: |
  extensions:
    memory_ballast:
      size_mib: 64 (1)
      size_in_percentage: 20 (2)

  receivers:
    otlp:
      protocols:
        http:

  exporters:
    otlp:

  service:
    extensions: [memory_ballast]
    pipelines:
      traces:
        receivers: [otlp]
        exporters: [otlp]
```

1. メモリーバラストのサイズを MiB 単位で設定します。両方が指定されている場合は、**size_in_percentage** のよりも優先されます。

2. メモリーバラストを合計メモリーに対するパーセンテージ (**1** - **100**) として設定します。コンテナー化された物理ホスト環境をサポートします。

3.1.1.7.7. zPages エクステンション

zPages エクステンションは現在、テクノロジープレビュー 機能のみとなっています。

zPages エクステンションは、zPages を提供するエクステンションに HTTP エンドポイントを提供します。エンドポイントでは、このエクステンションは、インストルメントされたコンポーネントをデバッグするためのライブデータを提供します。すべてのコアエクスポーターとレシーバーは、一部の zPages インストルメンテーションを提供します。

zPages は、トレースやメトリクスを調べるためにバックエンドに依存する必要がなく、プロセス内診断に役立ちます。

zPages エクステンションが設定された OpenTelemetry Collector カスタムリソース

```
config: |
  extensions:
    zpages:
      endpoint: "localhost:55679" (1)

  receivers:
    otlp:
      protocols:
        http:
  exporters:
    otlp:

  service:
```

```
      extensions: [zpages]
      pipelines:
        traces:
          receivers: [otlp]
          exporters: [otlp]
```

1. zPages を提供する HTTP エンドポイントを指定します。**localhost:** を使用してローカルでのみ使用できるようにするか、**":"** を使用してすべてのネットワークインターフェイスで使用できるようにします。デフォルトは **localhost:55679** です。

3.2. OPENTELEMETRY COLLECTOR を使用したさまざまなクラスターからの可観測性データの収集

マルチクラスター設定の場合、リモートクラスターごとに 1 つの OpenTelemetry Collector インスタンスを作成してから、すべてのテレメトリーデータを 1 つの OpenTelemetry Collector インスタンスに転送できます。

前提条件

- Red Hat build of OpenTelemetry Operatorがインストールされている。
- Tempo Operator がインストールされている。
- TempoStack インスタンスがクラスターにデプロイされている。
- 証明書 (Issuer、自己署名証明書、CA issuer、クライアントとサーバーの証明書) がマウントされている。これらの証明書のいずれかを作成するには、手順 1 を参照してください。

手順

1. OpenTelemetry Collector インスタンスに次の証明書をマウントし、すでにマウントされている証明書を省略します。

   a. Red Hat OpenShift の cert-manager Operator を使用して証明書を生成する Issuer

   ```
   apiVersion: cert-manager.io/v1
   kind: Issuer
   metadata:
     name: selfsigned-issuer
   spec:
     selfSigned: {}
   ```

   b. 自己署名証明書

   ```
   apiVersion: cert-manager.io/v1
   kind: Certificate
   metadata:
     name: ca
   spec:
     isCA: true
     commonName: ca
     subject:
       organizations:
   ```

```
      - Organization # <your_organization_name>
    organizationalUnits:
      - Widgets
  secretName: ca-secret
  privateKey:
    algorithm: ECDSA
    size: 256
  issuerRef:
    name: selfsigned-issuer
    kind: Issuer
    group: cert-manager.io
```

c. CA issuer

```
apiVersion: cert-manager.io/v1
kind: Issuer
metadata:
  name: test-ca-issuer
spec:
  ca:
    secretName: ca-secret
```

d. クライアントとサーバーの証明書

```
apiVersion: cert-manager.io/v1
kind: Certificate
metadata:
  name: server
spec:
  secretName: server-tls
  isCA: false
  usages:
    - server auth
    - client auth
  dnsNames:
  - "otel.observability.svc.cluster.local" (1)
  issuerRef:
    name: ca-issuer
---
apiVersion: cert-manager.io/v1
kind: Certificate
metadata:
  name: client
spec:
  secretName: client-tls
  isCA: false
  usages:
    - server auth
    - client auth
  dnsNames:
  - "otel.observability.svc.cluster.local" (2)
  issuerRef:
    name: ca-issuer
```

1. サーバー OpenTelemetry Collector インスタンスのソルバーにマップされる正確な DNS 名のリスト。
2. クライアント OpenTelemetry Collector インスタンスのソルバーにマップされる正確な DNS 名のリスト。

2. OpenTelemetry Collector インスタンスのサービスアカウントを作成します。

ServiceAccount の例

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: otel-collector-deployment
```

3. サービスアカウントのクラスターロールを作成します。

ClusterRole の例

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: otel-collector
rules:
  1
  2
- apiGroups: ["", "config.openshift.io"]
  resources: ["pods", "namespaces", "infrastructures", "infrastructures/status"]
  verbs: ["get", "watch", "list"]
```

1. **k8sattributesprocessor** には、Pod と namespace リソースに対する権限が必要です。
2. **resourcedetectionprocessor** には、インフラストラクチャーとステータスに対する権限が必要です。

4. クラスターロールをサービスアカウントにバインドします。

ClusterRoleBinding の例

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: otel-collector
subjects:
- kind: ServiceAccount
  name: otel-collector-deployment
  namespace: otel-collector-<example>
roleRef:
  kind: ClusterRole
  name: otel-collector
  apiGroup: rbac.authorization.k8s.io
```

5. YAML ファイルを作成して、エッジクラスターで **OpenTelemetryCollector** カスタムリソース (CR) を定義します。

エッジクラスター用の OpenTelemetryCollector カスタムリソースの例

```
apiVersion: opentelemetry.io/v1alpha1
kind: OpenTelemetryCollector
metadata:
  name: otel
  namespace: otel-collector-<example>
spec:
  mode: daemonset
  serviceAccount: otel-collector-deployment
  config: |
    receivers:
      jaeger:
        protocols:
          grpc:
          thrift_binary:
          thrift_compact:
          thrift_http:
      opencensus:
      otlp:
        protocols:
          grpc:
          http:
      zipkin:
    processors:
      batch:
      k8sattributes:
      memory_limiter:
        check_interval: 1s
        limit_percentage: 50
        spike_limit_percentage: 30
      resourcedetection:
        detectors: [openshift]
    exporters:
      otlphttp:
        endpoint: https://observability-cluster.com:443 (1)
        tls:
          insecure: false
          cert_file: /certs/server.crt
          key_file: /certs/server.key
          ca_file: /certs/ca.crt
    service:
      pipelines:
        traces:
          receivers: [jaeger, opencensus, otlp, zipkin]
          processors: [memory_limiter, k8sattributes, resourcedetection, batch]
          exporters: [otlp]
  volumes:
    - name: otel-certs
      secret:
        name: otel-certs
```

```
volumeMounts:
  - name: otel-certs
    mountPath: /certs
```

1. Collector エクスポーターは、OTLP HTTP をエクスポートするように設定されており、中央クラスターから OpenTelemetry Collector を指します。

6. YAML ファイルを作成して、中央クラスターに **OpenTelemetryCollector** カスタムリソース (CR) を定義します。

中央クラスターの OpenTelemetryCollector カスタムリソースの例

```
apiVersion: opentelemetry.io/v1alpha1
kind: OpenTelemetryCollector
metadata:
  name: otlp-receiver
  namespace: observability
spec:
  mode: "deployment"
  ingress:
    type: route
    route:
      termination: "passthrough"
  config: |
    receivers:
      otlp:
        protocols:
          http:
            tls: # (1)
              cert_file: /certs/server.crt
              key_file: /certs/server.key
              client_ca_file: /certs/ca.crt
    exporters:
      logging:
      otlp:
        endpoint: "tempo-<simplest>-distributor:4317" # (2)
        tls:
          insecure: true
    service:
      pipelines:
        traces:
          receivers: [otlp]
          processors: []
          exporters: [otlp]
  volumes:
    - name: otel-certs
      secret:
        name: otel-certs
  volumeMounts:
    - name: otel-certs
      mountPath: /certs
```

1. Collector レシーバーには、最初の手順にリストされている証明書が必要です。

2. Collector エクスポーターは、OTLP をエクスポートするように設定され、Tempo ディストリビュータエンドポイントを指します。この例では、これは **"tempo-simplest-**

3.3. 監視スタックにメトリクスを送信するための設定

OpenTelemetry Collector カスタムリソース (CR) は、Collector のパイプラインメトリクスと有効な Prometheus エクスポーターをスクレイプするための Prometheus **ServiceMonitor** CR を作成するように設定できます。

Prometheus エクスポーターを使用した OpenTelemetry Collector カスタムリソースの例

```
spec:
  mode: deployment
  observability:
    metrics:
      enableMetrics: true (1)
  config: |
    exporters:
      prometheus:
        endpoint: 0.0.0.0:8889
        resource_to_telemetry_conversion:
          enabled: true # by default resource attributes are dropped
    service:
      telemetry:
        metrics:
          address: ":8888"
      pipelines:
        metrics:
          receivers: [otlp]
          exporters: [prometheus]
```

1. コレクターの内部メトリクスエンドポイントと Prometheus エクスポーターメトリクスエンドポイントをスクレイプする Prometheus **ServiceMonitor** CR を作成するように Operator を設定します。メトリクスは OpenShift モニタリングスタックに保存されます。

あるいは、手動で作成した Prometheus **PodMonitor** を使用すると、Prometheus のスクレイピング中に追加された重複したラベルを削除するなど、細かい制御を行うことができます。

Collector メトリクスをスクレイプするように監視スタックを設定する PodMonitor カスタムリソースの例

```
apiVersion: monitoring.coreos.com/v1
kind: PodMonitor
metadata:
  name: otel-collector
spec:
  selector:
    matchLabels:
      app.kubernetes.io/name: `<cr_name>-collector` (1)
  podMetricsEndpoints:
  - port: metrics (2)
  - port: promexporter (3)
```

```
relabelings:
- action: labeldrop
  regex: pod
- action: labeldrop
  regex: container
- action: labeldrop
  regex: endpoint
metricRelabelings:
- action: labeldrop
  regex: instance
- action: labeldrop
  regex: job
```

1. OpenTelemetry Collector カスタムリソースの名前。
2. OpenTelemetry Collector の内部メトリクスポートの名前。このポート名は、必ず **metrics** になります。
3. OpenTelemetry Collector の Prometheus エクスポーターポートの名前。

3.4. RED HAT BUILD OF OPENTELEMETRY のモニタリングのセットアップ

Red Hat build of OpenTelemetry Operatorは、各 OpenTelemetry Collector インスタンスの監視とアラートをサポートし、Operator 自体に関するアップグレードと運用メトリクスを公開します。

3.4.1. OpenTelemetry Collector メトリクスの設定

OpenTelemetry Collector インスタンスのメトリクスとアラートを有効化できます。

前提条件

- ユーザー定義プロジェクトのモニタリングがクラスターで有効にされている。

手順

- OpenTelemetry Collector インスタンスのメトリクスを有効にするには、**spec.observability.metrics.enableMetrics** フィールドを **true** に設定します。

```
apiVersion: opentelemetry.io/v1alpha1
kind: OpenTelemetryCollector
metadata:
  name: <name>
spec:
  observability:
    metrics:
      enableMetrics: true
```

検証

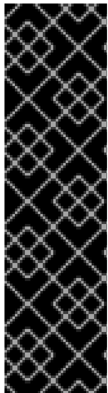
Web コンソールの **Administrator** ビューを使用して、正常に設定されたことを確認できます。

- Observe → Targets に移動し、Source: User でフィルタリングして、**opentelemetry-collector-<instance_name>** 形式の ServiceMonitors のステータスが Up であることを確認します。

3.5. 関連情報

- ユーザー定義プロジェクトのモニタリングの有効化

第4章 OPENTELEMETRY インストルメンテーション注入の設定とデプロイ



重要

OpenTelemetry インストルメンテーション注入はテクノロジープレビューのみの機能です。テクノロジープレビュー機能は、Red Hat 製品サポートのサービスレベルアグリーメント (SLA) の対象外であり、機能的に完全ではない場合があります。Red Hat は、実稼働環境でこれらを使用することを推奨していません。テクノロジープレビュー機能は、最新の製品機能をいち早く提供して、開発段階で機能のテストを行いフィードバックを提供していただくことを目的としています。

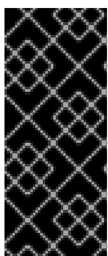
Red Hat のテクノロジープレビュー機能のサポート範囲に関する詳細は、テクノロジープレビュー機能のサポート範囲 を参照してください。

Red Hat build of OpenTelemetry Operatorは、インストルメンテーションの設定を定義するカスタムリソース定義 (CRD) ファイルを使用します。

4.1. OPENTELEMETRY インストルメンテーション設定オプション

Red Hat build of OpenTelemetry では、OpenTelemetry 自動インストルメンテーションライブラリーをワークロードに注入して設定できます。現在、プロジェクトは、Go、Java、Node.js、Python、.NET、および Apache HTTP Server (**httpd**) からのインストルメンテーションライブラリーの注入をサポートしています。

OpenTelemetry の自動インストルメンテーションとは、コードを手動で変更することなく、フレームワークがアプリケーションを自動的にインストルメンテーションする機能を指します。これにより、開発者と管理者は、最小限の労力と既存のコードベースへの変更で、アプリケーションに可観測性を導入できるようになります。



重要

Red Hat build of OpenTelemetry Operatorは、インストルメンテーションライブラリーの注入メカニズムのみをサポートしますが、インストルメンテーションライブラリーやアップストリームイメージはサポートしません。お客様は独自のインストルメンテーションイメージをビルドすることも、コミュニティーイメージを使用することもできます。

4.1.1. インストルメンテーションオプション

インストルメンテーションオプションは、**OpenTelemetryCollector** カスタムリソースで指定されます。

OpenTelemetryCollector カスタムリソースファイルのサンプル

```
apiVersion: opentelemetry.io/v1alpha1
kind: Instrumentation
metadata:
  name: java-instrumentation
spec:
  env:
    - name: OTEL_EXPORTER_OTLP_TIMEOUT
      value: "20"
```

```
exporter:
  endpoint: http://production-collector.observability.svc.cluster.local:4317
propagators:
  - w3c
sampler:
  type: parentbased_traceidratio
  argument: "0.25"
java:
  env:
  - name: OTEL_JAVAAGENT_DEBUG
    value: "true"
```

表4.1 Operator がインストルメンテーションを定義するために使用するパラメーター

パラメーター	説明	値
env	すべてのインストルメンテーションにわたって定義する共通の環境変数。	
exporter	エクスポーターの設定。	
propagators	プロパゲーターは、プロセス間のコンテキスト伝播設定を定義します。	**tracecontext**、**baggage**、**b3**、**b3multi**、**jaeger**、**ottrace**、**none**
resource	リソース属性の設定。	
sampler	サンプリング設定。	
apacheHttpd	Apache HTTP Server インストルメンテーションの設定。	
dotnet	.NET インストルメンテーションの設定。	
go	Go インストルメンテーションの設定。	
java	Java インストルメンテーションの設定。	
nodejs	Node.js インストルメンテーションの設定。	

パラメーター	説明	値
python	Python インストルメンテーションの設定。	

4.1.2. Service Mesh でのインストルメンテーション CR の使用

Red Hat OpenShift Service Mesh でインストルメンテーションカスタムリソース (CR) を使用する場合は、**b3multi** プロパゲーターを使用する必要があります。

4.1.2.1. Apache HTTP Server の自動インストルメンテーションの設定

表4.2 .spec.apacheHttpd フィールドのパラメーター

名前	説明	デフォルト
attrs	Apache HTTP Server に固有の属性。	
configPath	Apache HTTP Server 設定の場所。	/usr/local/apache2/conf
env	Apache HTTP Server に固有の環境変数。	
image	Apache SDK と自動インストルメンテーションを備えたコンテナーイメージ。	
resourceRequirements	コンピュートリソースの要件。	
version	Apache HTTP Server のバージョン。	2.4

注入を有効化するための PodSpec アノテーション

```
instrumentation.opentelemetry.io/inject-apache-httpd: "true"
```

4.1.2.2. .NET 自動インストルメンテーションの設定

名前	説明
env	.NET に固有の環境変数。
image	.NET SDK と自動インストルメンテーションを備えたコンテナーイメージ。
resourceRequirements	コンピュートリソースの要件。

.NET 自動インストルメンテーションの場合、エクスポータのエンドポイントが **4317** に設定されている場合は、必須の **OTEL_EXPORTER_OTLP_ENDPOINT** 環境変数を設定する必要があります。.NET 自動インストルメンテーションはデフォルトで **http/proto** を使用し、テレメトリーデータは **4318** ポートに設定する必要があります。

注入を有効化するための PodSpec アノテーション

```
instrumentation.opentelemetry.io/inject-dotnet: "true"
```

4.1.2.3. Go 自動インストルメンテーションの設定

名前	説明
env	Go に固有の環境変数。
image	Go SDK と自動インストルメンテーションを備えたコンテナーイメージ。
resourceRequirements	コンピュートリソースの要件。

注入を有効化するための PodSpec アノテーション

```
instrumentation.opentelemetry.io/inject-go: "true"
```

OpenShift クラスターの Go 自動インストルメンテーションに必要な追加の権限

```
apiVersion: security.openshift.io/v1
kind: SecurityContextConstraints
metadata:
  name: otel-go-instrumentation-scc
allowHostDirVolumePlugin: true
allowPrivilegeEscalation: true
allowPrivilegedContainer: true
```

```
allowedCapabilities:
- "SYS_PTRACE"
fsGroup:
  type: RunAsAny
runAsUser:
  type: RunAsAny
seLinuxContext:
  type: RunAsAny
seccompProfiles:
- '*'
supplementalGroups:
  type: RunAsAny
```

ヒント

OpenShift クラスターで Go 自動インストルメンテーションの権限を適用するための CLI コマンドは次のとおりです。

```
$ oc adm policy add-scc-to-user otel-go-instrumentation-scc -z <service_account>
```

4.1.2.4. Java 自動インストルメンテーションの設定

名前	説明
env	Java に固有の環境変数。
image	Java SDK と自動インストルメンテーションを備えたコンテナーイメージ。
resourceRequirements	コンピュートリソースの要件。

注入を有効化するための PodSpec アノテーション

```
instrumentation.opentelemetry.io/inject-java: "true"
```

4.1.2.5. Node.js 自動インストルメンテーションの設定

名前	説明
env	Node.js に固有の環境変数。
image	Node.js SDK と自動インストルメンテーションを備えたコンテナーイメージ。

名前	説明
resourceRequirements	コンピュートリソースの要件。

注入を有効化するための PodSpec アノテーション

```
instrumentation.opentelemetry.io/inject-nodejs: "true"
instrumentation.opentelemetry.io/otel-go-auto-target-exe: "/path/to/container/executable"
```

instrumentation.opentelemetry.io/otel-go-auto-target-exe アノテーションは、必要な **OTEL_GO_AUTO_TARGET_EXE** 環境変数の値を設定します。

4.1.2.6. Python 自動インストルメンテーションの設定

名前	説明
env	Python に固有の環境変数。
image	Python SDK と自動インストルメンテーションを備えたコンテナーイメージ。
resourceRequirements	コンピュートリソースの要件。

Python 自動インストルメンテーションの場合、エクスポーターのエンドポイントが **4317** に設定されている場合は、**OTEL_EXPORTER_OTLP_ENDPOINT** 環境変数を設定する必要があります。Python 自動インストルメンテーションはデフォルトで **http/proto** を使用し、テレメトリーデータは **4318** ポートに設定する必要があります。

注入を有効化するための PodSpec アノテーション

```
instrumentation.opentelemetry.io/inject-python: "true"
```

4.1.2.7. OpenTelemetry SDK 変数の設定

Pod 内の OpenTelemetry SDK 変数は、次のアノテーションを使用して設定できます。

```
instrumentation.opentelemetry.io/inject-sdk: "true"
```

すべてのアノテーションは、以下の値を受け入れることに注意してください。

true
: namespace からインストル **Instrumentation** リソースを注入します。

false

インストルメンテーションはいっさい注入しません。

instrumentation-name

現在の namespace から注入するインストルメンテーションリソースの名前。

other-namespace/instrumentation-name

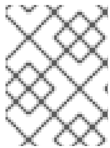
別の namespace から注入するインストルメンテーションリソースの名前。

4.1.2.8. マルチコンテナー Pod

インストルメンテーションは、Pod の仕様に従ってデフォルトで利用可能な最初のコンテナー上で実行されます。場合によっては、注入のターゲットコンテナーを指定することもできます。

Pod のアノテーション

```
instrumentation.opentelemetry.io/container-names: "<container_1>,<container_2>"
```



注記

Go 自動インストルメンテーションは、複数コンテナーの自動インストルメンテーション注入をサポートしていません。

第5章 RED HAT BUILD OF OPENTELEMETRY の使用

Red Hat build of OpenTelemetry をセットアップして使用し、トレースを OpenTelemetry Collector または TempoStack に送信できます。

5.1. OPENTELEMETRY COLLECTOR を使用してトレースを TEMPOSTACK に転送する

TempoStack への転送トレースを設定するには、OpenTelemetry Collector をデプロイして設定します。指定されたプロセッサー、レシーバー、エクスポーターを使用して、OpenTelemetry Collector をデプロイメントモードでデプロイできます。その他のモードについては、**関連情報** に記載されたリンクを使用して、OpenTelemetry Collector ドキュメントを参照してください。

前提条件

- Red Hat build of OpenTelemetry Operatorがインストールされている。
- Tempo Operator がインストールされている。
- TempoStack がクラスターにデプロイされている。

手順

1. OpenTelemetry Collector のサービスアカウントを作成します。

   ServiceAccount の例

   ```
   apiVersion: v1
   kind: ServiceAccount
   metadata:
     name: otel-collector-deployment
   ```

2. サービスアカウントのクラスターロールを作成します。

   ClusterRole の例

   ```
   apiVersion: rbac.authorization.k8s.io/v1
   kind: ClusterRole
   metadata:
     name: otel-collector
   rules:
     1
     2
   - apiGroups: ["", "config.openshift.io"]
     resources: ["pods", "namespaces", "infrastructures", "infrastructures/status"]
     verbs: ["get", "watch", "list"]
   ```

   1 **k8sattributesprocessor** には、Pod および namespace リソースに対する権限が必要です。

   2 **resourcedetectionprocessor** には、インフラストラクチャーとステータスに対する権限が必要です。

3. クラスターロールをサービスアカウントにバインドします。

ClusterRoleBinding の例

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: otel-collector
subjects:
- kind: ServiceAccount
  name: otel-collector-deployment
  namespace: otel-collector-example
roleRef:
  kind: ClusterRole
  name: otel-collector
  apiGroup: rbac.authorization.k8s.io
```

4. YAML ファイルを作成して、**OpenTelemetryCollector** カスタムリソース (CR) を定義します。

OpenTelemetryCollector の例

```
apiVersion: opentelemetry.io/v1alpha1
kind: OpenTelemetryCollector
metadata:
  name: otel
spec:
  mode: deployment
  serviceAccount: otel-collector-deployment
  config: |
    receivers:
      jaeger:
        protocols:
          grpc:
          thrift_binary:
          thrift_compact:
          thrift_http:
      opencensus:
      otlp:
        protocols:
          grpc:
          http:
      zipkin:
    processors:
      batch:
      k8sattributes:
      memory_limiter:
        check_interval: 1s
        limit_percentage: 50
        spike_limit_percentage: 30
      resourcedetection:
        detectors: [openshift]
    exporters:
      otlp:
        endpoint: "tempo-simplest-distributor:4317" (1)
        tls:
```

```
        insecure: true
    service:
      pipelines:
        traces:
          receivers: [jaeger, opencensus, otlp, zipkin] (2)
          processors: [memory_limiter, k8sattributes, resourcedetection, batch]
          exporters: [otlp]
```

1. Collector エクスポーターは、OTLP をエクスポートするように設定され、作成済みの Tempo ディストリビューターエンドポイント (この例では **"tempo-simplest-distributor:4317"**) を指します。

2. Collector は、Jaeger トレース、OpenCensus プロトコル経由の OpenCensus トレース、Zipkin プロトコル経由の Zipkin トレース、および GRPC プロトコル経由の OTLP トレースのレシーバーを使用して設定されます。

ヒント

tracegen をテストとしてデプロイできます。

```
apiVersion: batch/v1
kind: Job
metadata:
  name: tracegen
spec:
  template:
    spec:
      containers:
        - name: tracegen
          image: ghcr.io/open-telemetry/opentelemetry-collector-contrib/tracegen:latest
          command:
            - "./tracegen"
          args:
            - -otlp-endpoint=otel-collector:4317
            - -otlp-insecure
            - -duration=30s
            - -workers=1
      restartPolicy: Never
  backoffLimit: 4
```

関連情報

- OpenTelemetry Collector ドキュメント
- GitHub 上でのデプロイメント例

5.2. トレースとメトリクスを OPENTELEMETRY COLLECTOR に送信する

OpenTelemetry Collector へのトレースとメトリクスの送信は、サイドカー注入の有無にかかわらず可能です。

5.2.1. サイドカー注入を使用してトレースとメトリクスを OpenTelemetry Collector に送信する

サイドカー注入を使用して、OpenTelemetry Collector インスタンスへのテレメトリーデータの送信をセットアップできます。

Red Hat build of OpenTelemetry Operatorでは、デプロイメントワークロードへのサイドカー注入と、OpenTelemetry Collector にテレメトリーデータを送信するためのインストルメンテーションの自動設定が可能です。

前提条件

- Red Hat OpenShift distributed tracing platform (Tempo) がインストールされ、TempoStack インスタンスがデプロイされている。
- Web コンソールまたは OpenShift CLI (**oc**) を使用してクラスターにアクセスできる。
  - **cluster-admin** ロールを持つクラスター管理者として Web コンソールにログインしている。
  - **cluster-admin** ロールを持つクラスター管理者によるアクティブな OpenShift CLI (**oc**) セッション。
  - Red Hat OpenShift Dedicated を使用する場合は **dedicated-admin** ロールを持つアカウント。

手順

1. OpenTelemetry Collector インスタンスのプロジェクトを作成します。

```
apiVersion: project.openshift.io/v1
kind: Project
metadata:
  name: observability
```

2. サービスアカウントを作成します。

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: otel-collector-sidecar
  namespace: observability
```

3. **k8sattributes** および **resourcedetection** プロセッサーの権限をサービスアカウントに付与します。

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: otel-collector
rules:
- apiGroups: ["", "config.openshift.io"]
  resources: ["pods", "namespaces", "infrastructures", "infrastructures/status"]
  verbs: ["get", "watch", "list"]
---
```

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: otel-collector
subjects:
- kind: ServiceAccount
  name: otel-collector-sidecar
  namespace: observability
roleRef:
  kind: ClusterRole
  name: otel-collector
  apiGroup: rbac.authorization.k8s.io
```

4. OpenTelemetry Collector をサイドカーとしてデプロイします。

```
apiVersion: opentelemetry.io/v1alpha1
kind: OpenTelemetryCollector
metadata:
  name: otel
  namespace: observability
spec:
  serviceAccount: otel-collector-sidecar
  mode: sidecar
  config: |
    serviceAccount: otel-collector-sidecar
    receivers:
      otlp:
        protocols:
          grpc:
          http:
    processors:
      batch:
      memory_limiter:
        check_interval: 1s
        limit_percentage: 50
        spike_limit_percentage: 30
      resourcedetection:
        detectors: [openshift]
        timeout: 2s
    exporters:
      otlp:
        endpoint: "tempo-<example>-gateway:8090" (1)
        tls:
          insecure: true
    service:
      pipelines:
        traces:
          receivers: [jaeger]
          processors: [memory_limiter, resourcedetection, batch]
          exporters: [otlp]
```

(1) これは、**<example>** Tempo Operator を使用してデプロイされた TempoStack インスタンスのゲートウェイを指します。

5. **otel-collector-sidecar** サービスアカウントを使用してデプロイメントを作成します。

6. **sidecar.opentelemetry.io/inject: "true"** アノテーションを **Deployment** オブジェクトに追加します。これにより、ワークロードから OpenTelemetry Collector インスタンスにデータを送信するために必要なすべての環境変数が注入されます。

5.2.2. サイドカー注入を使用せずにトレースとメトリクスを OpenTelemetry Collector に送信する

サイドカー注入を使用せずに、テレメトリーデータを OpenTelemetry Collector インスタンスに送信するようにセットアップできます。これには、いくつかの環境変数を手動で設定する必要があります。

前提条件

- Red Hat OpenShift distributed tracing platform (Tempo) がインストールされ、TempoStack インスタンスがデプロイされている。
- Web コンソールまたは OpenShift CLI (**oc**) を使用してクラスターにアクセスできる。
    - **cluster-admin** ロールを持つクラスター管理者として Web コンソールにログインしている。
    - **cluster-admin** ロールを持つクラスター管理者によるアクティブな OpenShift CLI (**oc**) セッション。
    - Red Hat OpenShift Dedicated を使用する場合は **dedicated-admin** ロールを持つアカウント。

手順

1. OpenTelemetry Collector インスタンスのプロジェクトを作成します。

```
apiVersion: project.openshift.io/v1
kind: Project
metadata:
  name: observability
```

2. サービスアカウントを作成します。

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: otel-collector-deployment
  namespace: observability
```

3. **k8sattributes** および **resourcedetection** プロセッサーの権限をサービスアカウントに付与します。

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: otel-collector
rules:
- apiGroups: ["", "config.openshift.io"]
  resources: ["pods", "namespaces", "infrastructures", "infrastructures/status"]
  verbs: ["get", "watch", "list"]
---
```

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: otel-collector
subjects:
- kind: ServiceAccount
  name: otel-collector
  namespace: observability
roleRef:
  kind: ClusterRole
  name: otel-collector
  apiGroup: rbac.authorization.k8s.io
```

4. **OpenTelemetryCollector** カスタムリソースを使用して OpenTelemetry Collector インスタンスをデプロイします。

```
apiVersion: opentelemetry.io/v1alpha1
kind: OpenTelemetryCollector
metadata:
  name: otel
  namespace: observability
spec:
  mode: deployment
  serviceAccount: otel-collector-deployment
  config: |
    receivers:
      jaeger:
        protocols:
          grpc:
          thrift_binary:
          thrift_compact:
          thrift_http:
      opencensus:
      otlp:
        protocols:
          grpc:
          http:
      zipkin:
    processors:
      batch:
      k8sattributes:
      memory_limiter:
        check_interval: 1s
        limit_percentage: 50
        spike_limit_percentage: 30
      resourcedetection:
        detectors: [openshift]
    exporters:
      otlp:
        endpoint: "tempo-<example>-distributor:4317" (1)
        tls:
          insecure: true
    service:
      pipelines:
        traces:
```

```
      receivers: [jaeger, opencensus, otlp, zipkin]
      processors: [memory_limiter, k8sattributes, resourcedetection, batch]
      exporters: [otlp]
```

1. これは、**<example>** Tempo Operator を使用してデプロイされた TempoStack インスタンスのゲートウェイを指します。

5. インストルメント化されたアプリケーションを使用してコンテナーに環境変数を設定します。

名前	説明	デフォルト値
OTEL_SERVICE_NAME	**service.name** リソース属性の値を設定します。	**""**
OTEL_EXPORTER_OTLP_ENDPOINT	オプションで指定したポート番号を持つシグナル型のベースエンドポイント URL。	**https://localhost:4317**
OTEL_EXPORTER_OTLP_CERTIFICATE	gRPC クライアントの TLS 認証情報の証明書ファイルへのパス。	**https://localhost:4317**
OTEL_TRACES_SAMPLER	トレースに使用されるサンプラー。	**parentbased_always_on**
OTEL_EXPORTER_OTLP_PROTOCOL	OTLP エクスポーターのトランスポートプロトコル。	**grpc**
OTEL_EXPORTER_OTLP_TIMEOUT	OTLP エクスポーターが各バッチエクスポートを待機する最大時間間隔。	**10s**
OTEL_EXPORTER_OTLP_INSECURE	gRPC リクエストのクライアントトランスポートセキュリティーを無効にします。HTTPS スキーマはこれをオーバーライドします。	**False**

第6章 RED HAT BUILD OF OPENTELEMETRY のトラブルシューティング

OpenTelemetry Collector のヘルスを測定し、データの取り込みに関する問題を調査する方法は複数あります。

6.1. OPENTELEMETRY COLLECTOR ログの取得

OpenTelemetry Collector のログを取得するには、以下の手順を実行します。

手順

1. **OpenTelemetryCollector** カスタムリソース (CR) で関連するログレベルを設定します。

```
config: |
  service:
    telemetry:
      logs:
        level: debug (1)
```

(1) Collector のログレベル。サポートされている値には、**info**、**warn**、**error**、または **debug** が含まれます。デフォルトは **info** です。

2. **oc logs** コマンドまたは Web コンソールを使用してログを取得します。

6.2. メトリクスの公開

OpenTelemetry Collector は、処理したデータボリュームに関するメトリクスを公開します。同様のメトリクスがメトリクスおよびログシグナル用にされていますが、以下はスパン用のメトリクスです。

otelcol_receiver_accepted_spans
: パイプラインに正常にプッシュされたスパンの数。

otelcol_receiver_refused_spans
: パイプラインにプッシュできなかったスパンの数。

otelcol_exporter_sent_spans
: 宛先に正常に送信されたスパンの数。

otelcol_exporter_enqueue_failed_spans
: 送信キューに追加できなかったスパンの数。

Operator は、メトリクスエンドポイントのスクレイプに使用できる **<cr_name>-collector-monitoring** テレメトリーサービスを作成します。

手順

1. **OpenTelemetryCollector** カスタムリソースに次の行を追加して、テレメトリーサービスを有効にします。

```
config: |
  service:
    telemetry:
```

```
        metrics:
          address: ":8888" (1)
```

(1) 内部 Collector のメトリクスが公開されるアドレス。デフォルトは **:8888** です。

1. ポート転送Collector Pod を使用する次のコマンドを実行して、メトリクスを取得します。

```
$ oc port-forward <collector_pod>
```

2. **http://localhost:8888/metrics** でメトリクスエンドポイントにアクセスします。

6.3. デバッグエクスポーター

収集されたデータを標準出力にエクスポートするようにデバッグエクスポーターを設定できます。

手順

1. **OpenTelemetryCollector** カスタムリソースを次のように設定します。

```
config: |
  exporters:
    debug:
      verbosity: detailed
  service:
    pipelines:
      traces:
        exporters: [debug]
      metrics:
        exporters: [debug]
      logs:
        exporters: [debug]
```

2. **oc logs** コマンドまたは Web コンソールを使用して、ログを標準出力にエクスポートします。

第7章 DISTRIBUTED TRACING PLATFORM (JAEGER) から RED HAT BUILD OF OPENTELEMETRY への移行

アプリケーションに Red Hat OpenShift distributed tracing platform (Jaeger) をすでに使用している場合は、OpenTelemetry オープンソースプロジェクトに基づく Red Hat build of OpenTelemetry に移行できます。

Red Hat build of OpenTelemetry は、分散システムでの可観測性を促進するための API、ライブラリー、エージェント、およびインストルメンテーションのセットを提供します。Red Hat build of OpenTelemetry に含まれる OpenTelemetry Collector は、Jaeger プロトコルを取り込めるため、アプリケーションの SDK を変更する必要はありません。

distributed tracing platform (Jaeger) から Red Hat build of OpenTelemetry に移行するには、トレースをシームレスにレポートするように OpenTelemetry Collector とアプリケーションを設定する必要があります。サイドカーおよびサイドカーレスデプロイメントを移行できます。

7.1. DISTRIBUTED TRACING PLATFORM (JAEGER) から、サイドカーのある RED HAT BUILD OF OPENTELEMETRY への移行

Red Hat build of OpenTelemetry Operatorは、デプロイメントワークロードへのサイドカー注入をサポートしているため、distributed tracing platform (Jaeger) サイドカーから Red Hat build of OpenTelemetry サイドカーに移行できます。

前提条件

- Red Hat OpenShift distributed tracing platform (Jaeger) がクラスターで使用されている。
- Red Hat build of OpenTelemetry がインストールされている。

手順

1. OpenTelemetry Collector をサイドカーとして設定します。

```
apiVersion: opentelemetry.io/v1alpha1
kind: OpenTelemetryCollector
metadata:
  name: otel
  namespace: <otel-collector-namespace>
spec:
  mode: sidecar
  config: |
    receivers:
      jaeger:
        protocols:
          grpc:
          thrift_binary:
          thrift_compact:
          thrift_http:
    processors:
      batch:
      memory_limiter:
        check_interval: 1s
        limit_percentage: 50
        spike_limit_percentage: 30
```

```
    resourcedetection:
      detectors: [openshift]
      timeout: 2s
  exporters:
    otlp:
      endpoint: "tempo-<example>-gateway:8090" (1)
      tls:
        insecure: true
  service:
    pipelines:
      traces:
        receivers: [jaeger]
        processors: [memory_limiter, resourcedetection, batch]
        exporters: [otlp]
```

1. このエンドポイントは、**<example>** Tempo Operator を使用してデプロイされた TempoStack インスタンスのゲートウェイを指します。

2. アプリケーションを実行するためのサービスアカウントを作成します。

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: otel-collector-sidecar
```

3. 一部のプロセッサーで必要な権限のためのクラスターロールを作成します。

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: otel-collector-sidecar
rules:
  (1)
- apiGroups: ["config.openshift.io"]
  resources: ["infrastructures", "infrastructures/status"]
  verbs: ["get", "watch", "list"]
```

1. **resourcedetectionprocessor** には、インフラストラクチャーとインフラストラクチャー/ステータスに対する権限が必要です。

4. **ClusterRoleBinding** を作成して、サービスアカウントの権限を設定します。

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: otel-collector-sidecar
subjects:
- kind: ServiceAccount
  name: otel-collector-deployment
  namespace: otel-collector-example
roleRef:
```

```
  kind: ClusterRole
  name: otel-collector
  apiGroup: rbac.authorization.k8s.io
```

5. OpenTelemetry Collector をサイドカーとしてデプロイします。

6. **Deployment** オブジェクトから **"sidecar.jaegertracing.io/inject": "true"** アノテーションを削除することで、注入された Jaeger Agent をアプリケーションから削除します。

7. **sidecar.opentelemetry.io/inject: "true"** アノテーションを **Deployment** オブジェクトの **.spec.template.metadata.annotations** フィールドに追加して、OpenTelemetry サイドカーの自動注入を有効にします。

8. 作成したサービスアカウントをアプリケーションのデプロイメントに使用します。そうすることで、プロセッサーは正しい情報を取得してトレースに追加できます。

7.2. DISTRIBUTED TRACING PLATFORM (JAEGER) からサイドカーなしの RED HAT BUILD OF OPENTELEMETRY への移行

サイドカーをデプロイせずに、distributed tracing platform (Jaeger) から Red Hat build of OpenTelemetry に移行できます。

前提条件

- Red Hat OpenShift distributed tracing platform (Jaeger) がクラスターで使用されている。
- Red Hat build of OpenTelemetry がインストールされている。

手順

1. OpenTelemetry Collector デプロイメントを設定します。

2. OpenTelemetry Collector のデプロイ先となるプロジェクトを作成します。

```
apiVersion: project.openshift.io/v1
kind: Project
metadata:
  name: observability
```

3. OpenTelemetry Collector インスタンスを実行するためのサービスアカウントを作成します。

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: otel-collector-deployment
  namespace: observability
```

4. プロセッサーに必要な権限を設定するためのクラスターロールを作成します。

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: otel-collector
rules:
```

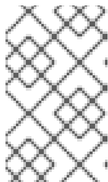
```
1
2
- apiGroups: ["", "config.openshift.io"]
  resources: ["pods", "namespaces", "infrastructures", "infrastructures/status"]
  verbs: ["get", "watch", "list"]
```

1. **k8sattributesprocessor** には、**pods** および **namespace** リソースに対する権限が必要です。
2. **resourcedetectionprocessor** には、**infrastructures** および **infrastructures/status** に対する権限が必要です。

5. ClusterRoleBinding を作成して、サービスアカウントの権限を設定します。

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: otel-collector
subjects:
- kind: ServiceAccount
  name: otel-collector-deployment
  namespace: observability
roleRef:
  kind: ClusterRole
  name: otel-collector
  apiGroup: rbac.authorization.k8s.io
```

6. OpenTelemetry Collector インスタンスを作成します。



注記

このCollector は、トレースを TempoStack インスタンスにエクスポートします。Red Hat Tempo Operator を使用して TempoStack インスタンスを作成し、正しいエンドポイントを配置する必要があります。

```
apiVersion: opentelemetry.io/v1alpha1
kind: OpenTelemetryCollector
metadata:
  name: otel
  namespace: observability
spec:
  mode: deployment
  serviceAccount: otel-collector-deployment
  config: |
    receivers:
      jaeger:
        protocols:
          grpc:
          thrift_binary:
          thrift_compact:
          thrift_http:
    processors:
      batch:
```

```
      k8sattributes:
      memory_limiter:
        check_interval: 1s
        limit_percentage: 50
        spike_limit_percentage: 30
      resourcedetection:
        detectors: [openshift]
    exporters:
      otlp:
        endpoint: "tempo-example-gateway:8090"
        tls:
          insecure: true
    service:
      pipelines:
        traces:
          receivers: [jaeger]
          processors: [memory_limiter, k8sattributes, resourcedetection, batch]
          exporters: [otlp]
```

7. トレースエンドポイントを OpenTelemetry Operator に指定します。

8. トレースをアプリケーションから Jaeger に直接エクスポートする場合は、API エンドポイントを Jaeger エンドポイントから OpenTelemetry Collector エンドポイントに変更します。

Golang を使用する jaegerexporter でトレースをエクスポートする場合の例

```
exp, err := jaeger.New(jaeger.WithCollectorEndpoint(jaeger.WithEndpoint(url))) (1)
```

(1) URL は OpenTelemetry Collector API エンドポイントを指します。

第8章 RED HAT BUILD OF OPENTELEMETRY の更新

バージョンのアップグレードの場合、Red Hat build of OpenTelemetry Operatorは Operator Lifecycle Manager (OLM) を使用します。これは、クラスター内の Operator のインストール、アップグレード、およびロールベースのアクセス制御 (RBAC) を制御します。

OLM は、デフォルトで OpenShift Container Platform で実行されます。OLM は利用可能な Operator のクエリーやインストールされた Operator のアップグレードを実行します。

Red Hat build of OpenTelemetry Operatorが新しいバージョンにアップグレードされると、管理する実行中の OpenTelemetry Collector インスタンスがスキャンされ、Operator の新しいバージョンに対応するバージョンにアップグレードされます。

8.1. 関連情報

- Operator Lifecycle Manager の概念およびリソース
- インストール済み Operator の更新

第9章 RED HAT BUILD OF OPENTELEMETRY の削除

OpenShift Container Platform クラスターから Red Hat build of OpenTelemetry を削除する手順は次のとおりです。

1. Red Hat build of OpenTelemetry Pod をすべてシャットダウンします。
2. OpenTelemetryCollector インスタンスを削除します。
3. Red Hat build of OpenTelemetry Operator を削除します。

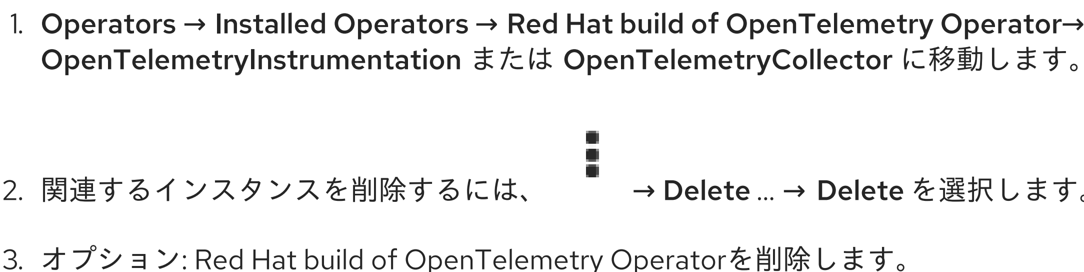
9.1. WEB コンソールを使用した OPENTELEMETRY COLLECTOR インスタンスの削除

Web コンソールの **Administrator** ビューで OpenTelemetry Collector インスタンスを削除できます。

前提条件

- **cluster-admin** ロールを持つクラスター管理者として Web コンソールにログインしている。
- Red Hat OpenShift Dedicated の場合、**dedicated-admin** ロールを持つアカウントを使用してログインしている。

手順

1. **Operators** → **Installed Operators** → **Red Hat build of OpenTelemetry Operator**→ **OpenTelemetryInstrumentation** または **OpenTelemetryCollector** に移動します。
2. 関連するインスタンスを削除するには、 → **Delete** … → **Delete** を選択します。
3. オプション: Red Hat build of OpenTelemetry Operatorを削除します。

9.2. CLI を使用した OPENTELEMETRY COLLECTOR インスタンスの削除

コマンドラインで OpenTelemetry Collector インスタンスを削除できます。

前提条件

- **cluster-admin** ロールを持つクラスター管理者によるアクティブな OpenShift CLI (**oc**) セッション。

  ヒント

  - OpenShift CLI (**oc**)のバージョンが最新であり、OpenShift Container Platform バージョンと一致していることを確認してください。
  - **oc login** を実行します。

    ```
    $ oc login --username=<your_username>
    ```

手順

1. 次のコマンドを実行して、OpenTelemetry Collector インスタンスの名前を取得します。

```
$ oc get deployments -n <project_of_opentelemetry_instance>
```

2. 次のコマンドを実行して、OpenTelemetry Collector インスタンスを削除します。

```
$ oc delete opentelemetrycollectors <opentelemetry_instance_name> -n
<project_of_opentelemetry_instance>
```

3. オプション: Red Hat build of OpenTelemetry Operatorを削除します。

検証

- OpenTelemetry Collector インスタンスが正常に削除されたことを確認するには、**oc get deployments** を再度実行します。

```
$ oc get deployments -n <project_of_opentelemetry_instance>
```

9.3. 関連情報

- クラスターからの Operator の削除
- OpenShift CLI の使用を開始