Red Hat Integration 2020-Q4

『Getting Started with Service Registry』

Service Registry 1.1

Last Updated: 2022-07-30

Red Hat Integration 2020-Q4 『Getting Started with Service Registry』

Service Registry 1.1

Enter your first name here. Enter your surname here.
Enter your organisation's name here. Enter your organisational division here.
Enter your email address here.

法律上の通知

Copyright © 2022 | You need to change the HOLDER entity in the en-US/Getting_Started_with_Service_Registry.ent file |.

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

このガイドでは、Service Registry を紹介し、選択したレジストリーストレージでのインストール方法について説明します。Service Registry Web コンソール、REST API、Maven プラグイン、または Java クライアントを使用してイベントスキーマと API 設計を管理する方法を説明します。このガイドでは、コンシューマーおよびプロデューサーアプリケーションで Kafka クライアントシリアライザーとデシリアライザーを使用する方法についても説明します。また、OpenShift の Service Registry コンテンツタイプ、ルール設定、および環境変数も説明します。

目次

第1章 SERVICE REGISTRY の概要 5
1.1. SERVICE REGISTRY の概要 5
Service Registry の機能 5
1.2. SERVICE REGISTRY のスキーマおよび API アーティファクト 6
1.3. SERVICE REGISTRY ストレージのオプション 6
1.4. SERVICE REGISTRY WEB コンソールを使用したコンテンツの管理 7
1.5. KAFKA クライアントシリアライザー/デシリアライザーでのスキーマの検証 8
1.6. KAFKA CONNECT コンバーターを使用した外部システムへのデータのストリーミング 9
1.7. SERVICE REGISTRY デモ例 10
1.8. SERVICE REGISTRY で利用可能なディストリビューション 11

第2章 SERVICE REGISTRY のコンテンツルール 13
2.1. ルールを使用したレジストリーコンテンツの管理 13
2.2. ルールの適用時 13
2.3. ルールの仕組み 13
2.4. コンテンツルールの設定 14
アーティファクトルールの設定 14
グローバルルールの設定 14

第3章 SERVICE REGISTRY クイックスタート 16
3.1. QUICKSTART SERVICE REGISTRY OPERATOR のインストール 16
3.2. QUICKSTART SERVICE REGISTRY デプロイメント 16

第4章 OPENSHIFT での SERVICE REGISTRY のインストール 18
4.1. OPENSHIFT OPERATORHUB からの SERVICE REGISTRY のインストール 18

第5章 AMQ STREAMS での SERVICE REGISTRY ストレージのデプロイ 20
5.1. OPENSHIFT OPERATORHUB からの AMQ STREAMS のインストール: 20
5.2. OPENSHIFT での AMQ STREAMS ストレージを使用した SERVICE REGISTRY の設定 21
5.3. AMQ STREAMS での SERVICE REGISTRY ストレージを使用した TLS セキュリティーの設定 23
5.4. AMQ STREAMS での SERVICE REGISTRY ストレージでの SCRAM セキュリティーの設定 26

第6章 POSTGRESQL データベースでの SERVICE REGISTRY ストレージのデプロイ 29
6.1. OPENSHIFT OPERATORHUB からの POSTGRESQL データベースのインストール 29
6.2. OPENSHIFT での POSTGRESQL データベースストレージを使用した SERVICE REGISTRY の設定 30
6.3. SERVICE REGISTRY POSTGRESQL ストレージのバックアップ 32
6.4. SERVICE REGISTRY POSTGRESQL ストレージの復元 32

第7章 INFINISPAN での組み込み SERVICE REGISTRY ストレージのデプロイ 34
7.1. OPENSHIFT での組み込み INFINISPAN ストレージでの SERVICE REGISTRY の設定 34

第8章 SERVICE REGISTRY デプロイメントの設定および管理 36
8.1. OPENSHIFT での SERVICE REGISTRY ヘルスチェックの設定 36
8.2. SERVICE REGISTRY ヘルスチェックの環境変数 37
liveness 環境変数 37
readiness 環境変数 38
8.3. OPENSHIFT クラスター内から SERVICE REGISTRY への HTTPS 接続の設定 39
8.4. OPENSHIFT クラスター外から SERVICE REGISTRY への HTTPS 接続の設定 41

第9章 WEB コンソールを使用した SERVICE REGISTRY コンテンツの管理 43
9.1. SERVICE REGISTRY WEB コンソールの設定 43
Web コンソールのデプロイメント環境の設定 43
読み取り専用モードでのコンソールの設定 43

9.2. SERVICE REGISTRY WEB コンソールを使用したアーティファクトの追加 44
9.3. SERVICE REGISTRY WEB コンソールを使用したアーティファクトの表示 45
9.4. SERVICE REGISTRY WEB コンソールを使用したコンテンツルールの設定 47

第10章 REST API を使用した SERVICE REGISTRY コンテンツの管理 49
10.1. REGISTRY REST API の概要 49
10.2. REGISTRY REST API コマンドを使用したアーティファクトの管理 50

第11章 MAVEN プラグインを使用した SERVICE REGISTRY コンテンツの管理 52
11.1. SERVICE REGISTRY MAVEN プラグインを使用したアーティファクトの管理 52
Maven プラグインを使用したアーティファクトの登録 52
Maven プラグインを使用したアーティファクトのダウンロード 53
Maven プラグインを使用したアーティファクトのテスト 53

第12章 JAVA クライアントを使用した SERVICE REGISTRY コンテンツの管理 55
12.1. SERVICE REGISTRY JAVA クライアント 55
12.2. SERVICE REGISTRY クライアントアプリケーションの作成 55
12.3. SERVICE REGISTRY JAVA CLIENT CONFIGURATION 56
カスタムヘッダー設定 56
TLS 設定 56

第13章 KAFKA クライアントシリアライザー/デシリアライザーを使用したスキーマの検証 58
13.1. KAFKA クライアントアプリケーションおよび SERVICE REGISTRY 58
プロデューサースキーマの設定 59
コンシューマースキーマの設定 59
13.2. スキーマを検索するストラテジー 60
アーティファクト ID ストラテジー 61
アーティファクト ID を返すストラテジー 61
グローバル ID ストラテジー 61
グローバル ID を返すストラテジー 61
グローバル ID ストラテジーの設定 62
13.3. SERVICE REGISTRY シリアライザー/デシリアライザー定数 62
シリアライザー/デシリアライザーサービスの定数 62
検索ストラテジーの定数 62
コンバーターの定数 63
Avro データプロバイダーの定数 63
13.4. 異なるクライアントのシリアライザー/デシリアライザータイプの使用 63
シリアライザー/デシリアライザーの Kafka アプリケーション設定 64
13.4.1. Service Registry を使用した Avro SerDe の設定 65
13.4.2. Service Registry を使用した JSON スキーマ SerDe の設定 67
13.4.3. Service Registry を使用した Protobuf SerDe の設定 68
13.5. スキーマの SERVICE REGISTRY への登録 69
Service Registry Web コンソール 69
curl コマンドの例 70
Maven プラグインの例 70
プロデューサークライアントを使用した設定例 70
13.6. KAFKA コンシューマークライアントからのスキーマの使用 71
13.7. KAFKA プロデューサークライアントからのスキーマの使用 71
13.8. KAFKA STREAMS アプリケーションからのスキーマの使用 72

第14章 SERVICE REGISTRY アーティファクトの参照 74
14.1. SERVICE REGISTRY アーティファクトタイプ 74
14.2. SERVICE REGISTRY アーティファクトの状態 74
14.3. SERVICE REGISTRY アーティファクトのメタデータ 75

14.4. SERVICE REGISTRY コンテンツルールタイプ 76
14.5. SERVICE REGISTRY コンテンツルールの成熟度 77

第15章 SERVICE REGISTRY OPERATOR の設定リファレンス 79
15.1. SERVICE REGISTRY カスタムリソース 79
15.2. SERVICE REGISTRY CR 仕様 80
15.3. SERVICE REGISTRY CR のステータス 82
15.4. SERVICE REGISTRY 管理リソース 83
15.5. SERVICE REGISTRY OPERATOR ラベル 84

付録A サブスクリプションの使用 85
アカウントへのアクセス 85
サブスクリプションのアクティベート 85
ZIP および TAR ファイルのダウンロード 85
パッケージ用のシステムの登録 85

第1章 SERVICE REGISTRY の概要

本章では、Service Registry の概念および機能を紹介し、レジストリーに保存されるサポート対象のアーティファクトタイプの詳細を提供します。

- 「Service Registry の概要」
- 「Service Registry のスキーマおよび API アーティファクト」
- 「Service Registry ストレージのオプション」
- 「Service Registry Web コンソールを使用したコンテンツの管理」
- 「Kafka クライアントシリアライザー/デシリアライザーでのスキーマの検証」
- 「Kafka Connect コンバーターを使用した外部システムへのデータのストリーミング」
- 「Service Registry デモ例」
- 「Service Registry で利用可能なディストリビューション」

1.1. SERVICE REGISTRY の概要

Service Registry は、API およびイベント駆動型アーキテクチャー全体で標準的なイベントスキーマおよび API 設計を共有するためのデータストアです。Service Registry を使用して、クライアントアプリケーションからデータの構造を切り離し、REST インターフェースを使用して実行時にデータ型と API の記述を共有および管理できます。

たとえば、クライアントアプリケーションは、再デプロイせずに最新のスキーマ更新を実行時に Service Registry との間で動的にプッシュまたはプルできます。開発者チームは、すでに実稼働でデプロイされているサービスに必要な既存のスキーマのレジストリーをクエリーでき、新規サービスを開発する際に新しいスキーマを登録できます。

クライアントアプリケーションが、クライアントアプリケーションでレジストリー URL を指定することで、Service Registry に保存されているスキーマおよび API 設計を使用できるようにすることができます。たとえば、レジストリーにはメッセージをシリアライズおよびデシリアライズするために使用されるスキーマを保存できます。その後、クライアントアプリケーションからスキーマを参照して、送受信されるメッセージとこれらのスキーマの互換性を維持することができます。

Service Registry を使用して、アプリケーションからデータ構造を切り離し、メッセージ全体のサイズを減らすことでコストを削減し、組織内のスキーマおよび API 設計の一貫性を高めて効率化します。Service Registry は、開発者および管理者がレジストリーコンテンツの管理を簡単に行えるように Web コンソールを提供します。

さらに、オプションのルールを構成して、レジストリーコンテンツの展開を管理できます。たとえば、これらには、アップロードされたコンテンツが構文的および意味的に有効であること、または他のバージョンとの上位互換性と下位互換性があることを確認するためのルールが含まれます。設定済みのルールは新規バージョンをレジストリーにアップロードする前に渡す必要があります。これにより、無効または互換性のないスキーマや API 設計に無駄な時間を費やさないようにします。

Service Registry は、Apicurio Registry オープンソースコミュニティープロジェクトに基づいています。詳細は https://github.com/apicurio/apicurio-registry を参照してください。

Service Registry の機能

- 標準イベントスキーマと API 仕様の複数のペイロード形式をサポート

- AMQ Streams、組み込み Infinispan、または PostgreSQL データベースを含むプラグ可能なストレージオプション
- Web コンソール、REST API コマンド、Maven プラグイン、または Java クライアントを使用したレジストリーコンテンツ管理
- レジストリーコンテンツが時間とともにどのように進化するかを管理するためのコンテンツ検証とバージョン互換性のルール
- 外部システム用の Kafka Connect との統合を含む、Apache Kafka スキーマレジストリの完全なサポート
- 実行時に Kafka およびその他のメッセージタイプを検証するクライアントシリアライザー/デシリアライザー(Serdes)
- メモリーフットプリントが低く、デプロイメントの時間が高速化されるクラウドネイティブ Quarkus Java ランタイム
- 既存の Confluent スキーマレジストリークライアントアプリケーションとの互換性
- OpenShift での Service Registry の Operator ベースのインストール

1.2. SERVICE REGISTRY のスキーマおよび API アーティファクト

イベントスキーマや API 仕様などの Service Registry に保存される項目は、レジストリー **アーティファクト** と呼ばれます。以下は、単純な株価アプリケーションの JSON 形式の Apache Avro スキーマアーティファクトの例を示しています。

```
{
   "type": "record",
   "name": "price",
   "namespace": "com.example",
   "fields": [
       {
           "name": "symbol",
           "type": "string"
       },
       {
           "name": "price",
           "type": "string"
       }
   ]
}
```

スキーマまたは API 契約がレジストリーのアーティファクトとして追加されると、クライアントアプリケーションはそのスキーマまたは API 契約を使用して、実行時にクライアントメッセージが正しいデータ構造に準拠することを確認できます。

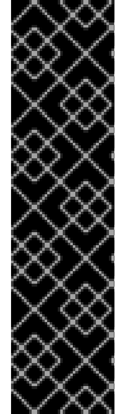
Service Registry は、標準のイベントスキーマおよび API 仕様の幅広いメッセージペイロード形式をサポートしています。たとえば、サポートされている形式には、Apache Avro、Google プロトコルバッファー、GraphQL、AsyncAPI、OpenAPI などがあります。詳細は 14章*Service Registry アーティファクトの参照* を参照してください。

1.3. SERVICE REGISTRY ストレージのオプション

Service Registry は、レジストリーアーティファクトの以下の基礎となるストレージ実装を提供します。

表1.1 Service Registry ストレージのオプション

ストレージオプション	Release
AMQ Streams 1.5 の Kafka Streams ベースのストレージ	一般公開
組み込み Infinispan 10 のキャッシュベースのストレージ	テクノロジープレビューとしてのみ提供
PostgreSQL 12 データベースの Java Persistence API ベースのストレージ	テクノロジープレビューとしてのみ提供



重要

Infinispan または PostgreSQL の Service Registry ストレージはテクノロジープレビュー機能です。テクノロジープレビュー機能は Red Hat の実稼働環境でのサービスレベルアグリーメント (SLA) ではサポートされていないため、Red Hat では実稼働環境での使用を推奨していません。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。

これらの機能は、近々発表予定の製品機能をリリースに先駆けてご提供することにより、お客様は機能性をテストし、開発プロセス中にフィードバックをお寄せいただくことができます。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、「テクノロジプレビュー機能のサポート範囲」を参照してください。

関連情報

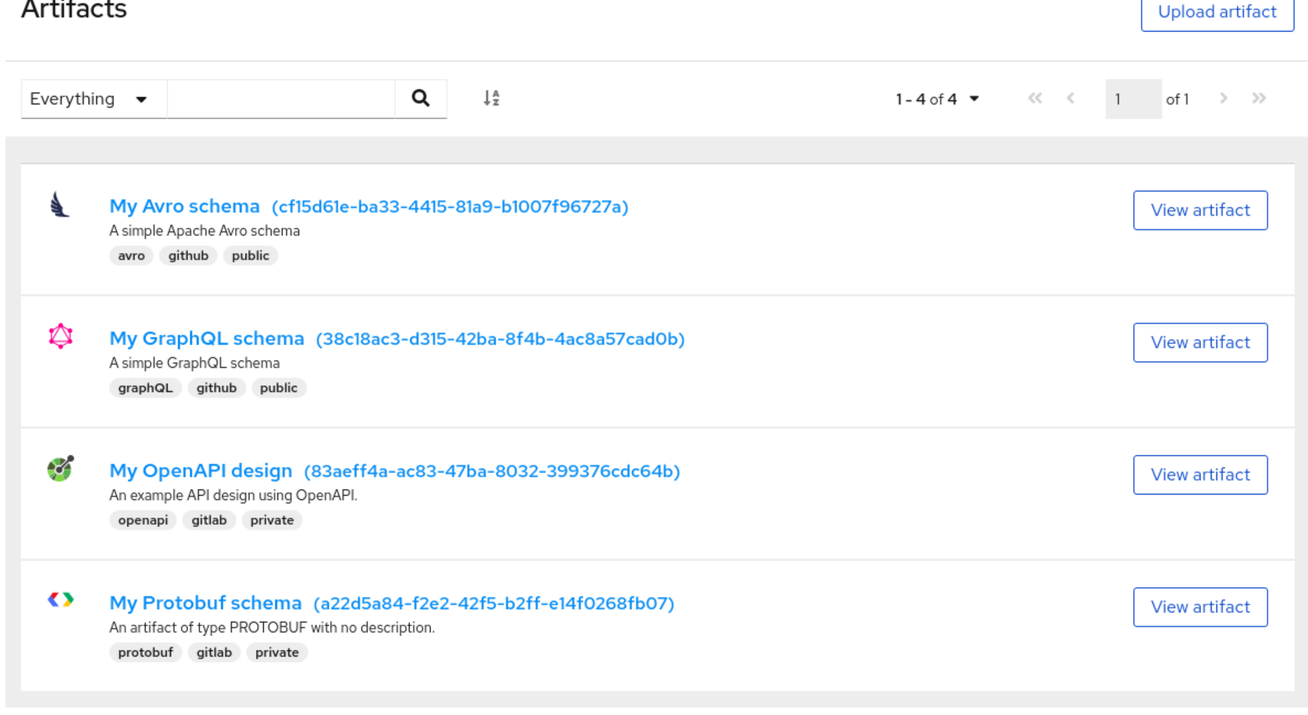
- *4章OpenShift での Service Registry のインストール*
- *5章AMQ Streams での Service Registry ストレージのデプロイ*

1.4. SERVICE REGISTRY WEB コンソールを使用したコンテンツの管理

Service Registry Web コンソールを使用して、レジストリーに保存されているアーティファクトを閲覧および検索し、新しいアーティファクトおよびアーティファクトバージョンをアップロードできます。ラベル、名前、および説明でアーティファクトを検索できます。アーティファクトのコンテンツの表示、利用可能なバージョンすべての表示、アーティファクトファイルをローカルでダウンロードすることもできます。

Service Registry Web コンソールを使用して、グローバルに、アーティファクトごとにレジストリーコンテンツの任意のルールを設定することもできます。コンテンツの検証および互換性に関するこれらの任意のルールは、新しいアーティファクトまたはアーティファクトバージョンがレジストリーにアップロードする場合に適用されます。詳細は *14章Service Registry アーティファクトの参照* を参照してください。

図1.1 Service Registry Web コンソール



Service Registry Web コンソールは、Service Registry デプロイメントのメインエンドポイント (たとえば、**http://MY-REGISTRY-URL/ui**) から利用できます。

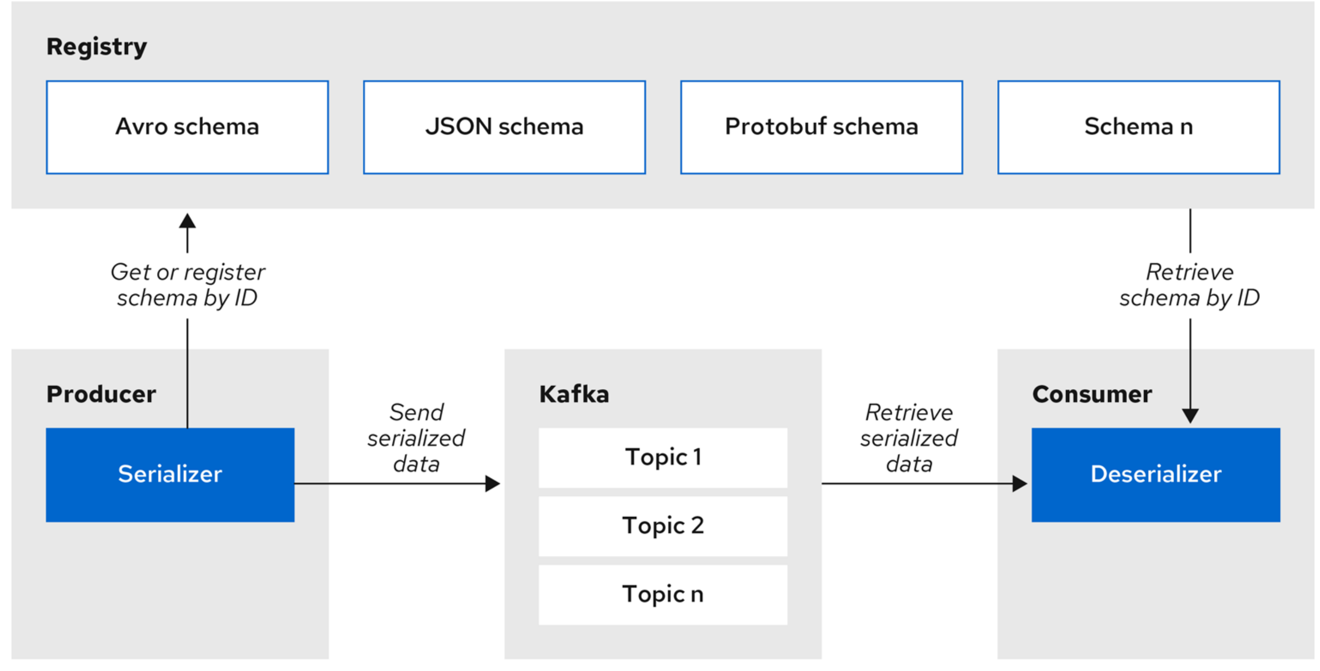
関連情報

- *9章Web コンソールを使用した Service Registry コンテンツの管理*

1.5. KAFKA クライアントシリアライザー/デシリアライザーでのスキーマの検証

Kafka プロデューサーアプリケーションは、シリアライザーを使用して、特定のイベントスキーマに準拠するメッセージをエンコードできます。Kafka コンシューマーアプリケーションはデシリアライザーを使用して、特定のスキーマ ID に基づいてメッセージが適切なスキーマを使用してシリアライズされたことを検証できます。

図1.2 Service Registry および Kafka クライアントシリアライザー/デシリアライザーアーキテクチャー



Service Registry は、実行時に以下のメッセージタイプを検証するために Kafka クライアントシリアライザー/デシリアライザー(Serdes)を提供します。

- Apache Avro
- Google プロトコルバッファー
- JSON スキーマ

Service Registry Maven リポジトリーおよびソースコードディストリビューションには、これらのメッセージタイプの Kafka シリアライザー/デシリアライザー実装が含まれており、Kafka クライアント開発者がレジストリーと統合するために使用できます。これらの実装には、サポートされる各メッセージタイプにカスタム **io.apicurio.registry.utils.serde** Java クラスが含まれ、クライアントアプリケーションが検証用に実行時にレジストリーからスキーマをプルするために使用できます。

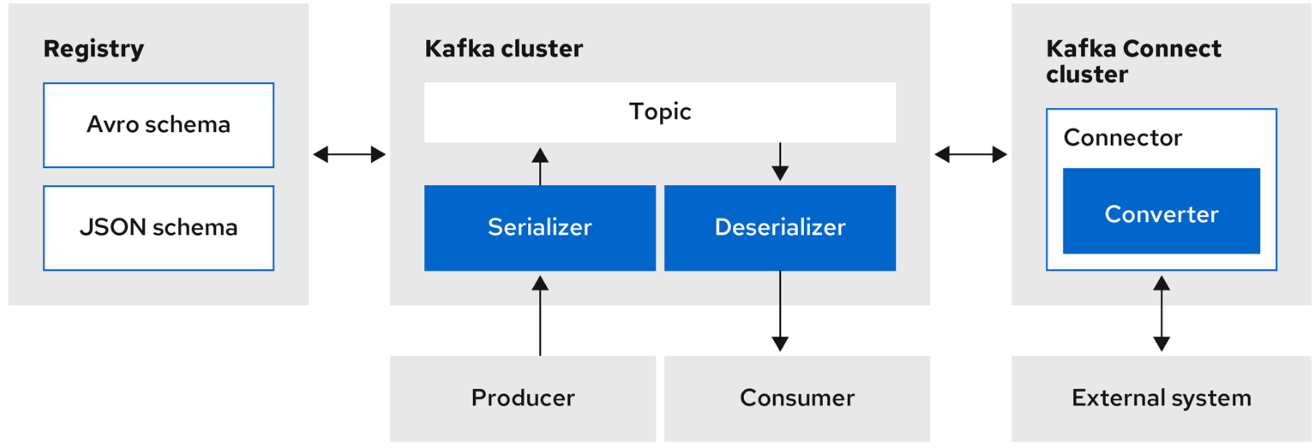
関連情報

- 13章*Kafka クライアントシリアライザー/デシリアライザーを使用したスキーマの検証*

1.6. KAFKA CONNECT コンバーターを使用した外部システムへのデータのストリーミング

Apache Kafka Connect と Service Registry を使用して、Kafka と外部システム間でデータをストリーミングできます。Kafka Connect を使用すると、異なるシステムのコネクターを定義して、大量のデータを Kafka ベースのシステムに出し入れできます。

図1.3 Service Registry および Kafka Connect アーキテクチャー



Service Registry は、Kafka Connect に次の機能を提供します。

- Kafka Connect スキーマのストレージ
- Apache Avro および JSON スキーマの Kafka Connect コンバーター
- スキーマを管理するレジストリー REST API

Avro および JSON スキーマコンバーターを使用して、Kafka Connect スキーマを Avro または JSON スキーマにマッピングすることができます。これらのスキーマは、メッセージのキーと値をコンパクトな Avro バイナリー形式または人間が判読できる JSON 形式にシリアライズすることができます。メッセージにはスキーマ情報が含まれず、スキーマ ID のみが含まれるため、変換された JSON も冗長性が低くなります。

Service Registry は、Kafka トピックで使用される Avro および JSON スキーマを管理および追跡できます。スキーマは Service Registry に保存され、メッセージコンテンツから切り離されるため、各メッセージには小さなスキーマ識別子だけを含める必要があります。Kafka など I/O 律速のシステムの場合、これはプロデューサーおよびコンシューマーのトータルスループットが向上することを意味します。

Service Registry によって提供される Avro および JSON スキーマシリアライザーおよびデシリアライザー(Serdes)は、このユースケースの Kafka プロデューサーおよびコンシューマーによっても使用されます。変更イベントを使用するために作成する Kafka コンシューマーアプリケーションは、Avro または JSON Serdes を使用して変更イベントをデシリアライズすることができます。これらの Serde は Kafka ベースのシステムにインストールし、Kafka Connect や Debezium および Camel Kafka Connector などの Kafka Connect ベースのシステムと共に使用できます。

その他のリソース

- Apache Kafka Connect のドキュメント
- Avro serialization in Debezium User Guide
- Getting Started with Camel Kafka Connector
- Demonstration of using Kafka Connect with Debezium and Apicurio Registry

1.7. SERVICE REGISTRY デモ例

Service Registry は、Apache Kafka Streams のストレージを使用した Apache Avro のシリアライズ/デシリアライズのソースデモを提供します。以下の例は、シリアライザー/デシリアライザーが実行時にレジストリーから Avro スキーマを取得し、これを使用して Kafka メッセージをシリアライズおよびデシリアライズする方法を示しています。詳細は、https://github.com/Apicurio/apicurio-registry-demo を参照してください。

Service Registry は、以下のサンプルアプリケーションも提供します。

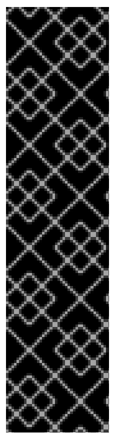
- シンプルな Avro の例
- 単純な JSON スキーマの例
- Confluent Serdes の統合
- Avro Bean の例
- カスタム ID ストラテジーの例
- Simple Avro Maven の例
- REST クライアントの例

詳細は、 https://github.com/Apicurio/apicurio-registry-examples を参照してください

1.8. SERVICE REGISTRY で利用可能なディストリビューション

表1.2 Service Registry Operator およびイメージ

ディストリビューション	場所	Release
Service Registry Operator	**Operators → OperatorHub** の OpenShift Web コンソール	一般公開 (GA)
Service Registry Operator のコンテナーイメージ	Red Hat Ecosystem Catalog	一般公開 (GA)
AMQ Streams での Kafka ストレージのコンテナーイメージ	Red Hat Ecosystem Catalog	一般公開
組み込み Infinispan ストレージのコンテナーイメージ	Red Hat エコシステムカタログ	テクノロジープレビューとしてのみ提供
PostgreSQL での JPA ストレージのコンテナーイメージ	Red Hat エコシステムカタログ	テクノロジープレビューとしてのみ提供



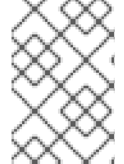
重要

Infinispan または PostgreSQL の Service Registry ストレージはテクノロジープレビュー機能です。テクノロジープレビュー機能は Red Hat の実稼働環境でのサービスレベルアグリーメント (SLA) ではサポートされていないため、Red Hat では実稼働環境での使用を推奨していません。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。

これらの機能は、近々発表予定の製品機能をリリースに先駆けてご提供することにより、お客様は機能性をテストし、開発プロセス中にフィードバックをお寄せいただくことができます。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、「テクノロジプレビュー機能のサポート範囲」を参照してください。

表1.3 Service Registry zip ダウンロード

ディストリビューション	場所	Release
Example custom resource definitions for installation	Red Hat Integration のソフトウェアダウンロード	GA（一般提供）およびテクニカルプレビュー
Kafka Connect converters	Red Hat Integration のソフトウェアダウンロード	一般公開 (GA)
Maven repository	Red Hat Integration のソフトウェアダウンロード	一般公開 (GA)
Source code	Red Hat Integration のソフトウェアダウンロード	一般公開 (GA)



注記

利用可能な Service Registry ディストリビューションにアクセスするには、Red Hat Integration のサブスクリプションが必要で、Red Hat カスタマーポータルにログインする必要があります。

第2章 SERVICE REGISTRY のコンテンツルール

本章では、レジストリーコンテンツを管理するために使用されるオプションのルールを紹介し、利用可能なルール設定の詳細を説明します。

- 「ルールを使用したレジストリーコンテンツの管理」
- 「ルールの適用時」
- 「ルールの仕組み」
- 「コンテンツルールの設定」

2.1. ルールを使用したレジストリーコンテンツの管理

レジストリーコンテンツの展開を管理するために、レジストリーに追加されるアーティファクトコンテンツの任意のルールを設定できます。設定されたグローバルルールまたはアーティファクトルールはすべて、新しいアーティファクトバージョンをレジストリーにアップロードする前に渡す必要があります。設定されたアーティファクトルールは、設定されたグローバルルールを上書きします。

これらのルールの目的は、無効なコンテンツがレジストリーに追加されないようにすることです。たとえば、次の理由でコンテンツが無効になる可能性があります。

- 特定のアーティファクトタイプ (**AVRO** や **PROTOBUF など**) の構文が無効です
- 有効な構文で、セマンティクスが仕様に違反している
- 新しいコンテンツに現在のアーティファクトバージョンに関連する変更の違反が含まれる場合の非互換性

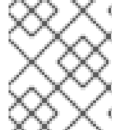
Service Registry Web コンソール、REST API コマンド、または Java クライアントアプリケーションを使用して、これらのコンテンツルールを追加できます。

2.2. ルールの適用時

ルールは、コンテンツがレジストリーに追加される場合にのみ適用されます。これには、以下の REST 操作が含まれます。

- アーティファクトの追加
- アーティファクトの更新
- アーティファクトバージョンの追加

ルールに違反した場合、Service Registry は HTTP エラーを返します。応答本文には、違反したルールと、何が問題だったのかを示すメッセージが含まれます。



注記

アーティファクトにルールが設定されていない場合、現在設定されているグローバルルールのセットがあればそれが適用されます。

2.3. ルールの仕組み

各ルールには、名前と任意の設定情報があります。レジストリーストレージは、各アーティファクトのルール一覧とグローバルルールの一覧を維持します。リスト内の各ルールは、ルール実装に固有の名前と設定プロパティーのセットで構成されます。

アーティファクトの現在のバージョン (存在する場合) および追加されるアーティファクトの新しいバージョンのコンテンツを含むルールが提供されます。ルール実装は、アーティファクトがルールを渡すかどうかに応じて true または false を返します。そうでない場合、レジストリはその理由を HTTP エラー応答で報告します。一部のルールは、コンテンツの以前のバージョンを使用しない場合があります。たとえば、互換性ルールは以前のバージョンを使用しますが、構文またはセマンティック妥当性ルールは使用しません。

その他のリソース

詳細は 14章*Service Registry アーティファクトの参照* を参照してください。

2.4. コンテンツルールの設定

アーティファクトごとに個別にルールを設定することも、グローバルに設定することもできます。Service Registry は、特定のアーティファクトに設定したルールを適用します。そのレベルでルールが設定されていない場合、Service Registry はグローバルに設定されたルールを適用します。グローバルルールが設定されていない場合は、ルールが適用されません。

アーティファクトルールの設定

Service Registry Web コンソールまたは REST API を使用してアーティファクトルールを設定できます。詳細は以下を参照してください。

- 9章*Web コンソールを使用した Service Registry コンテンツの管理*
- Apicurio Registry REST API ドキュメント

グローバルルールの設定

グローバルルールは、複数の方法で設定できます。

- REST API で **/rules** 操作を使用します
- Service Registry Web コンソールの使用
- Service Registry アプリケーションプロパティーを使用したデフォルトのグローバルルールの設定

デフォルトのグローバルルールの設定

アプリケーションレベルで Service Registry を設定して、グローバルなルールを有効または無効にすることができます。以下のアプリケーションプロパティー形式を使用して、インストール後の設定を行わずに、インストール時にデフォルトのグローバルルールを設定できます。

```
registry.rules.global.<ruleName>
```

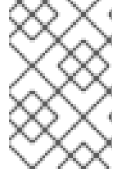
現在、以下のルール名がサポートされています。

- **compatibility**
- **validity**

application プロパティーの値は、設定されたルールに固有の有効な設定オプションである必要があります。以下の表は、各ルールの有効な値を示しています。

表2.1 Service Registry のコンテンツルール

ルール	値
Validity	**FULL**
	SYNTAX_ONLY
	NONE
互換性	**BACKWARD**
	BACKWARD_TRANSITIVE
	FORWARD
	FORWARD_TRANSITIVE
	FULL
	FULL_TRANSITIVE
	NONE



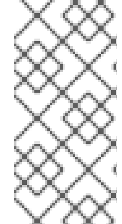
注記

これらのアプリケーションプロパティーは、Java システムプロパティーとして設定することも、Quarkus **application.properties** ファイルに含めることもできます。詳細は、Quarkus のドキュメント を参照してください。

第3章 SERVICE REGISTRY クイックスタート

本章では、OpenShift コマンドラインを使用して Service Registry Operator を迅速にインストールする方法について説明します。このクイックスタートの例では、組み込み Infinispan ストレージオプションを使用して Service Registry をデプロイします。

- 「Quickstart Service Registry Operator のインストール」
- 「Quickstart Service Registry デプロイメント」



注記

実稼働環境で推奨されるインストールオプションは 「OpenShift OperatorHub からの Service Registry のインストール」 です。

実稼働環境で推奨されるストレージオプションは AMQ Streams です。詳細は、5章*AMQ Streams での Service Registry ストレージのデプロイ* を参照してください。

3.1. QUICKSTART SERVICE REGISTRY OPERATOR のインストール

Service Registry Operator は、ダウンロードしたインストールファイルとサンプルを使用して、Operator Lifecycle Manager を使用せずにコマンドラインで迅速に導入することができます。

前提条件

- Red Hat Integration Downloads に移動し、製品バージョンを選択し、Service Registry CRD **.zip** ファイルをダウンロードする必要があります。

手順

1. インストール用のプロジェクトを作成します。たとえば、**service-registry**:

```
oc new-project service-registry
```

2. **{NAMESPACE}** を **service-registry** に置き換えて、**install/cluster_role_binding.yaml** に namespace を設定します。

3. **install/** フォルダーにあるファイルを適用します。

```
oc apply -f install/ -n service-registry
```

3.2. QUICKSTART SERVICE REGISTRY デプロイメント

新しい Service Registry デプロイメントを迅速に作成するには、埋め込み Infinispan ストレージオプションを使用します。この場合、外部ストレージを前提条件として設定する必要はありません。

前提条件

- Service Registry Operator がすでにインストールされていることを確認する。

手順

1. Operator がデプロイされている namespace に **ApicurioRegistry** カスタムリソース(CR)を作成します。

```
oc apply -f ./examples/apicurioregistry_infinispan_cr.yaml -n service-registry
```

Infinispan ストレージの CR の例

```
apiVersion: apicur.io/v1alpha1
kind: ApicurioRegistry
metadata:
  name: example-apicurioregistry
spec:
  configuration:
    persistence: "infinispan"
    infinispan:
      clusterName: "example-apicurioregistry"
      # ^ Optional
```

2. Service Registry Web コンソール用に自動作成されたルートにアクセスします。以下に例を示します。

```
http://example-apicurioregistry.my-project.my-domain-name.com/
```

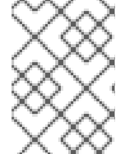
第4章 OPENSHIFT での SERVICE REGISTRY のインストール

本章では、Service Registry のインストール方法を説明します。

- 「OpenShift OperatorHub からの Service Registry のインストール」

前提条件

- 1章*Service Registry の概要*



注記

環境に応じて、複数の Service Registry インスタンスをインストールできます。インスタンス数は、Service Registry に保存されているアーティファクトの数および種類と、選択したストレージオプションによって異なります。

4.1. OPENSHIFT OPERATORHUB からの SERVICE REGISTRY のインストール

OperatorHub から OpenShift クラスターに Service Registry Operator をインストールできます。OperatorHub は OpenShift Container Platform Web コンソールから使用でき、クラスター管理者が Operator を検出およびインストールするためのインターフェースを提供します。詳細は、OpenShift のドキュメント を参照してください。

前提条件

- クラスター管理者として OpenShift クラスターにアクセスできる。

手順

1. OpenShift Container Platform Web コンソールで、クラスター管理者権限を持つアカウントを使用してログインします。
2. 新しい OpenShift プロジェクトを作成します。
   a. 左側のナビゲーションメニューで、**Home** > **Project** > **Create Project** とクリックします。
   b. プロジェクト名 (**my-project** など) を入力し、**Create** をクリックします。
3. 左側のナビゲーションメニューで、**Operators** > **OperatorHub** とクリックします。
4. **Filter by keyword** テキストボックスに **registry** を入力し、**Red Hat Integration - Service Registry Operator** を見つけます。
5. Operator に関する情報を読み、**Install** をクリックして Operator サブスクリプションページを表示します。
6. サブスクリプション設定を選択します。以下に例を示します。
   - **Update Channel** > 以下のチャンネルから 1 つ選択します。
     - **ServiceRegistry-1:** 1.1.0 および 1.0.1 などのマイナーおよびパッチの更新すべて。1.0.x へのインストールは自動的に 1.1.x にアップグレードされます。

    - ServiceRegistry **-1.0**: 1.0.1 および 1.0.2 などのパッチ更新のみ。1.0.x へのインストールは自動的に 1.1.x を無視します。
    - ServiceRegistry **-1.1**: 1.1.1 や 1.1.2 などのパッチ更新のみ。1.1.x へのインストールは自動的に 1.0.x を無視します。
  - **Installation Mode** > **A specific namespace on the cluster**> **my-project**
  - **Approval Strategy** > **Manual**
7. **Install** をクリックし、Operator が使用できるようになるまでしばらく待ちます。

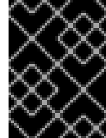
その他のリソース

- Operator の OpenShift クラスターへの追加
- GitHub の Apicurio Registry Operator コミュニティー

第5章 AMQ STREAMS での SERVICE REGISTRY ストレージのデプロイ

本章では、AMQ Streams で Service Registry ストレージをインストールし、設定する方法を説明します。

- 「OpenShift OperatorHub からの AMQ Streams のインストール:」
- 「OpenShift での AMQ Streams ストレージを使用した Service Registry の設定」
- 「AMQ Streams での Service Registry ストレージを使用した TLS セキュリティーの設定」
- 「AMQ Streams での Service Registry ストレージでの SCRAM セキュリティーの設定」



重要

AMQ Streams の Service Registry ストレージは、実稼働環境で推奨されるストレージオプションです。

前提条件

- *4章OpenShift での Service Registry のインストール*

5.1. OPENSHIFT OPERATORHUB からの AMQ STREAMS のインストール:

AMQ Streams がインストールされていない場合は、OperatorHub から OpenShift クラスターに AMQ Streams Operator をインストールできます。OperatorHub は OpenShift Container Platform Web コンソールから使用でき、クラスター管理者が Operator を検出およびインストールするためのインターフェースを提供します。詳細は、OpenShift のドキュメント を参照してください。

前提条件

- クラスター管理者として OpenShift クラスターにアクセスできる必要があります。
- AMQ Streams のインストールに関する詳細は、「AMQ Streams on OpenShift の使用 」を参照してください。ここでは、OpenShift OperatorHub を使用したインストールの簡単な例を示します。

手順

1. OpenShift Container Platform Web コンソールで、クラスター管理者権限を持つアカウントを使用してログインします。
2. Service Registry がインストールされている OpenShift プロジェクトに切り替えます。たとえば、**Project** ドロップダウンから、**my-project** を選択します。
3. 左側のナビゲーションメニューで、**Operators** > **OperatorHub** とクリックします。
4. **Filter by keyword** テキストボックスに **AMQ Streams** を入力し、**Red Hat Integration - AMQ Streams** を見つけます。
5. Operator に関する情報を読み、**Install** をクリックして Operator サブスクリプションページを表示します。

6. サブスクリプション設定を選択します。以下に例を示します。
    - **Update Channel** > **amq-streams-1.5.x**
    - **Installation Mode** > **A specific namespace on the cluster**> **my-project**
    - **Approval Strategy** > **Manual**
7. **Install** をクリックし、Operator が使用できるようになるまでしばらく待ちます。

その他のリソース

- Operator の OpenShift クラスターへの追加
- OpenShift での AMQ Streams の使用

5.2. OPENSHIFT での AMQ STREAMS ストレージを使用した SERVICE REGISTRY の設定

ここでは、AMQ Streams on OpenShift を使用して Service Registry に Kafka ベースのストレージを設定する方法を説明します。このストレージオプションは、OpenShift の Kafka クラスターに **persistent** ストレージが設定されている実稼働環境に適しています。既存の Kafka クラスターに Service Registry をインストールするか、環境に応じて新しい Kafka クラスターを作成できます。

前提条件

- クラスター管理者として OpenShift クラスターにアクセスできる。
- Service Registry がすでにインストールされている。*4章OpenShift での Service Registry のインストール* を参照してください。
- AMQ Streams がすでにインストールされている。「OpenShift OperatorHub からの AMQ Streams のインストール:」 を参照してください。

手順

1. OpenShift Container Platform Web コンソールで、クラスター管理者権限を持つアカウントを使用してログインします。
2. Kafka クラスターがまだ設定されていない場合は、AMQ Streams を使用して新しい Kafka クラスターを作成します。たとえば、OpenShift OperatorHub では以下を実行します。
    a. **Installed Operators** > **Red Hat Integration - AMQ Streams**の順にクリックします。
    b. **Provided APIs** > **Kafka** で **Create Instance** をクリックし、新しい Kafka クラスターを作成します。
    c. 適切にカスタムリソース定義を編集し、**Create** をクリックします。

警告

デフォルトの例では、3 つの Zookeeper ノード、および、**ephemeral** ストレージを持つ 3 つの Kafka ノードを持つクラスターが作成されます。この一時ストレージは開発およびテストにのみ適しており、実稼働には適していません。詳細は、『 Using AMQ Streams on OpenShift 』を参照してください。

3. クラスターの準備ができたら、**Provided APIs** > **Kafka** > **my-cluster** > **YAML** をクリックします。

4. **status** ブロックで、**bootstrapServers** 値のコピーを作成します。これは、後で ServiceRegistry をデプロイするために使用します。以下に例を示します。

```
status:
  conditions:
  ...
  listeners:
    - addresses:
        - host: my-cluster-kafka-bootstrap.my-project.svc
          port: 9092
      bootstrapServers: 'my-cluster-kafka-bootstrap.my-project.svc:9092'
      type: plain
  ...
```

5. Kafka トピックを作成し、Service Registry アーティファクトを保存します。
   a. **Provided APIs** > **Kafka Topic** で、**Create topic** をクリックします。
   b. デフォルトのトピック名を **my-topic** から必要な **storage-topic** に変更します。

6. Kafka トピックを作成し、Service Registry のグローバル ID を保存します。
   a. **Provided APIs** > **Kafka Topic** で、**Create topic** をクリックします。
   b. デフォルトのトピック名を **my-topic** から必要な **global-id-topic** に変更します。

7. **Installed Operators** > **Red Hat Integration - Service Registry**> **ApicurioRegistry** > **Create ApicurioRegistry** とクリックします。

8. 次のカスタムリソース定義を貼り付けますが、前にコピーした **bootstrapServers** 値を使用します。

```
apiVersion: apicur.io/v1alpha1
kind: ApicurioRegistry
metadata:
  name: example-apicurioregistry
spec:
  configuration:
    persistence: "streams"
    streams:
      bootstrapServers: "my-cluster-kafka-bootstrap.my-project.svc:9092"
```

9. **Create** をクリックし、OpenShift で Service Registry ルートが作成されるまで待機します。

10. **Networking** > **Route** をクリックして、Service Registry Web コンソールの新規ルートにアクセスします。以下に例を示します。

```
http://example-apicurioregistry.my-project.my-domain-name.com/
```

関連情報

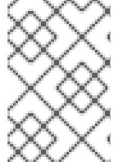
- AMQ Streams を使用した Kafka クラスターおよびトピックの作成に関する詳細は、『Using AMQ Streams on OpenShift』を参照してください。

5.3. AMQ STREAMS での SERVICE REGISTRY ストレージを使用した TLS セキュリティーの設定

AMQ Streams Operator および Service Registry Operator を、暗号化された Transport Layer Security (TLS) 接続を使用するように設定できます。

前提条件

- OperatorHub またはコマンドラインを使用して Service Registry Operator をインストールする。
- AMQ Streams Operator をインストールする、または Kafka が OpenShift クラスターからアクセスできる。



注記

ここでは、AMQ Streams Operator が利用可能であることを前提としていますが、任意の Kafka デプロイメントを使用できます。この場合、Service Registry Operator が想定する Openshift シークレットを手動で作成する必要があります。

手順

1. OpenShift Web コンソールで **Installed Operators** をクリックし、**AMQ Streams** Operator の詳細を選択してから、**Kafka** タブをクリックします。

2. **Create Kafka** をクリックし、Service Registry ストレージの新しい Kafka クラスターをプロビジョニングします。

3. Kafka クラスターに TLS 認証を使用するように、 **authorization** フィールドと **tls** フィールドを設定します。次に例を示します。

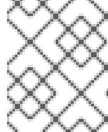
```
apiVersion: kafka.strimzi.io/v1beta1
kind: Kafka
metadata:
  name: my-cluster
  namespace: registry-example-streams-tls
spec:
  kafka:
    authorization:
      type: simple
    version: 2.5.0
```

```
    replicas: 3
    listeners:
      plain: {}
      tls:
        authentication:
          type: tls
    config:
      offsets.topic.replication.factor: 3
      transaction.state.log.replication.factor: 3
      transaction.state.log.min.isr: 2
      log.message.format.version: '2.5'
    storage:
      type: ephemeral
  zookeeper:
    replicas: 3
    storage:
      type: ephemeral
  entityOperator:
    topicOperator: {}
    userOperator: {}
```

4. Kafka トピックを作成し、Service Registry アーティファクトを保存します。
   a. **Provided APIs** > **Kafka Topic** で、**Create topic** をクリックします。
   b. デフォルトのトピック名を **my-topic** から必要な **storage-topic** に変更します。
5. Kafka トピックを作成し、Service Registry のグローバル ID を保存します。
   a. **Provided APIs** > **Kafka Topic** で、**Create topic** をクリックします。
   b. デフォルトのトピック名を **my-topic** から必要な **global-id-topic** に変更します。
6. **Kafka User** リソースを作成し、Service Registry ユーザーの認証および承認を設定します。たとえば、**spec** ブロックでは **metadata** セクションにユーザー名を指定するか、デフォルトの **my-user** を使用できます。

```
spec:
  authentication:
    type: tls
  authorization:
    acls:
      - operation: All
        resource:
          name: '*'
          patternType: literal
          type: topic
      - operation: All
        resource:
          name: '*'
          patternType: literal
          type: cluster
      - operation: All
        resource:
          name: '*'
          patternType: literal
```

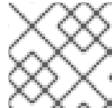
```
      type: transactionalId
    - operation: All
      resource:
        name: '*'
        patternType: literal
        type: group
  type: simple
```



注記

Service Registry が必要とするトピックおよびリソースに合わせて承認を設定する必要があります。これは簡単な例です。

7. **Workloads** をクリックしてから **Secrets** をクリックし、Service Registry が Kafka クラスターに接続するために AMQ Stremas によって作成される 2 つのシークレットを見つけます。
   - **my-cluster-cluster-ca-cert** - Kafka クラスターの PKCS12 トラストストアが含まれています
   - **my-user** - ユーザーのキーストアが含まれます

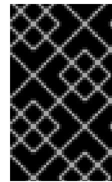


注記

シークレットの名前は、クラスターまたはユーザー名によって異なります。

8. シークレットを手動で作成する場合は、以下のキーと値のペアを含める必要があります。
   - my-cluster-ca-cert
     - **ca.p12** - PKCS12 形式のトラストストア
     - **ca.password** - truststore password
   - my-user
     - **user.p12** - PKCS12 形式のキーストア
     - **user.password** - keystore password
9. Service Registry をデプロイするように、以下の設定例を設定します。

```
apiVersion: apicur.io/v1alpha1
kind: ApicurioRegistry
metadata:
  name: example-apicurioregistry
spec:
  configuration:
    persistence: "streams"
    streams:
      bootstrapServers: "my-cluster-kafka-bootstrap.registry-example-streams-tls.svc:9093"
      security:
        tls:
          keystoreSecretName: my-user
          truststoreSecretName: my-cluster-cluster-ca-cert
```



重要

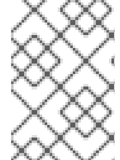
プレーンでセキュアでないユースケースとは別の **bootstrapServers** アドレスを使用する必要があります。アドレスは TLS 接続をサポートする必要があり、指定された **Kafka** リソースの **type:tls** フィールドにあります。

5.4. AMQ STREAMS での SERVICE REGISTRY ストレージでの SCRAM セキュリティーの設定

Kafka クラスターの Salted Challenge Response Authentication Mechanism (SCRAM-SHA-512) を使用するように AMQ Streams Operator および Service Registry Operator を設定できます。

前提条件

- OperatorHub またはコマンドラインを使用して Service Registry Operator をインストールする。
- AMQ Streams Operator をインストールする、または Kafka が OpenShift クラスターからアクセスできる。



注記

ここでは、AMQ Streams Operator が利用可能であることを前提としていますが、任意の Kafka デプロイメントを使用できます。この場合、Service Registry Operator が想定する Openshift シークレットを手動で作成する必要があります。

手順

1. OpenShift Web コンソールで **Installed Operators** をクリックし、**AMQ Streams** Operator の詳細を選択してから、**Kafka** タブをクリックします。
2. **Create Kafka** をクリックし、Service Registry ストレージの新しい Kafka クラスターをプロビジョニングします。
3. Kafka クラスターに SCRAM-SHA-512 認証を使用するように、 **authorization** フィールドと **tls** フィールドを設定します。次に例を示します。

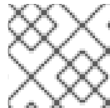
```
apiVersion: kafka.strimzi.io/v1beta1
kind: Kafka
metadata:
  name: my-cluster
  namespace: registry-example-streams-tls
spec:
  kafka:
    authorization:
      type: simple
    version: 2.5.0
    replicas: 3
    listeners:
      plain: {}
      tls:
        authentication:
          type: scram-sha-512
    config:
      offsets.topic.replication.factor: 3
```

```
      transaction.state.log.replication.factor: 3
      transaction.state.log.min.isr: 2
      log.message.format.version: '2.5'
    storage:
      type: ephemeral
  zookeeper:
    replicas: 3
    storage:
      type: ephemeral
  entityOperator:
    topicOperator: {}
    userOperator: {}
```

4. Kafka トピックを作成し、Service Registry アーティファクトを保存します。
   a. **Provided APIs** > **Kafka Topic** で、**Create topic** をクリックします。
   b. デフォルトのトピック名を **my-topic** から必要な **storage-topic** に変更します。
5. Kafka トピックを作成し、Service Registry のグローバル ID を保存します。
   a. **Provided APIs** > **Kafka Topic** で、**Create topic** をクリックします。
   b. デフォルトのトピック名を **my-topic** から必要な **global-id-topic** に変更します。
6. **Kafka User** リソースを作成し、Service Registry ユーザーの SCRAM 認証および承認を設定します。たとえば、**spec** ブロックは **authentication** セクションを参照してください。

```
spec:
  authentication:
    type: scram-sha-512
  authorization:
    acls:
      - operation: All
        resource:
          name: '*'
          patternType: literal
          type: topic
      - operation: All
        resource:
          name: '*'
          patternType: literal
          type: cluster
      - operation: All
        resource:
          name: '*'
          patternType: literal
          type: transactionalId
      - operation: All
        resource:
          name: '*'
          patternType: literal
          type: group
    type: simple
```

7. **Workloads** をクリックしてから **Secrets** をクリックし、Service Registry が Kafka クラスターに接続するために AMQ Stremas によって作成される 2 つのシークレットを見つけます。
   - **my-cluster-cluster-ca-cert** - Kafka クラスターの PKCS12 トラストストアが含まれています
   - **my-user** - ユーザーのキーストアが含まれます

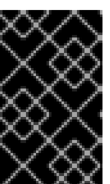


注記

シークレットの名前は、クラスターまたはユーザー名によって異なります。

8. シークレットを手動で作成する場合は、以下のキーと値のペアを含める必要があります。
   - my-cluster-ca-cert
     - **ca.p12** -PKCS12 形式のトラストストア
     - **ca.password** - truststore password
   - my-user
     - **パスワード** - ユーザーパスワード
9. Service Registry をデプロイするように、以下の設定例を設定します。

```
apiVersion: apicur.io/v1alpha1
kind: ApicurioRegistry
metadata:
  name: example-apicurioregistry
spec:
  configuration:
    persistence: "streams"
    streams:
      bootstrapServers: "my-cluster-kafka-bootstrap.registry-example-streams-scram.svc:9093"
      security:
        scram:
          truststoreSecretName: my-cluster-cluster-ca-cert
          user: my-user
          passwordSecretName: my-user
```



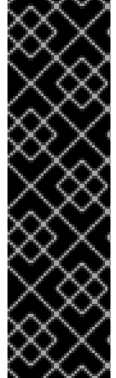
重要

プレーンでセキュアでないユースケースとは別の **bootstrapServers** アドレスを使用する必要があります。アドレスは TLS 接続をサポートする必要があり、指定された **Kafka** リソースの **type:tls** フィールドにあります。

第6章 POSTGRESQL データベースでの SERVICE REGISTRY ストレージのデプロイ

本章では、PostgreSQL データベースで Service Registry ストレージをインストール、設定、および管理する方法を説明します。

- 「OpenShift OperatorHub からの PostgreSQL データベースのインストール」
- 「OpenShift での PostgreSQL データベースストレージを使用した Service Registry の設定」
- 「Service Registry PostgreSQL ストレージのバックアップ」
- 「Service Registry PostgreSQL ストレージの復元」



重要

PostgreSQL の Service Registry ストレージはテクノロジープレビュー機能です。テクノロジープレビュー機能は Red Hat の実稼働環境でのサービスレベルアグリーメント (SLA) ではサポートされていないため、Red Hat では実稼働環境での使用を推奨していません。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。

これらの機能は、近々発表予定の製品機能をリリースに先駆けてご提供することにより、お客様は機能性をテストし、開発プロセス中にフィードバックをお寄せいただくことができます。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、https://access.redhat.com/ja/support/offerings/techpreview を参照してください。

前提条件

- *4章OpenShift での Service Registry のインストール*

6.1. OPENSHIFT OPERATORHUB からの POSTGRESQL データベースのインストール

PostgreSQL データベース Operator がインストールされていない場合は、OperatorHub から OpenShift クラスターに PostgreSQL Operator をインストールできます。OperatorHub は OpenShift Container Platform Web コンソールから使用でき、クラスター管理者が Operator を検出およびインストールするためのインターフェースを提供します。詳細は、OpenShift のドキュメント を参照してください。

前提条件

- クラスター管理者として OpenShift クラスターにアクセスできる。

手順

1. OpenShift Container Platform Web コンソールで、クラスター管理者権限を持つアカウントを使用してログインします。
2. Service Registry がインストールされている OpenShift プロジェクトに切り替えます。たとえば、**Project** ドロップダウンから、**my-project** を選択します。
3. 左側のナビゲーションメニューで、**Operators** > **OperatorHub** とクリックします。

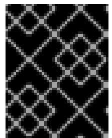
4. **Filter by keyword** テキストボックスに **PostgreSQL** と入力して、ご使用の環境に適した Operator を検索します。たとえば、**Crunchy PostgreSQL for OpenShift** や **PostgreSQL Operator by Dev4Ddevs.com** です。

5. Operator に関する情報を読み、**Install** をクリックして Operator サブスクリプションページを表示します。

6. サブスクリプション設定を選択します。以下に例を示します。

   - **Update Channel** > **stable**
   - **Installation Mode** > **A specific namespace on the cluster**> **my-project**
   - **Approval Strategy** > **Manual**

7. **Install** をクリックし、Operator が使用できるようになるまでしばらく待ちます。



重要

データベースの作成と管理方法の詳細については、選択した**PostgreSQL** Operator のドキュメントを読む必要があります。

その他のリソース

- Operator の OpenShift クラスターへの追加
- Crunchy PostgreSQL Operator QuickStart

6.2. OPENSHIFT での POSTGRESQL データベースストレージを使用した SERVICE REGISTRY の設定

本セクションでは、PostgreSQL データベース Operator を使用して、OpenShift 上の Service Registry の Java Persistence API ベースのストレージを設定する方法を説明します。既存のデータベースに Service Registry をインストールするか、環境に応じて新規データベースを作成することができます。本セクションでは、Dev4Ddevs.com による PostgreSQL Operator を使用する簡単な例を紹介します。

前提条件

- クラスター管理者として OpenShift クラスターにアクセスできる。
- Service Registry がすでにインストールされている。*4章OpenShift での Service Registry のインストール* を参照してください。
- OpenShift に PostgreSQL Operator がすでにインストールされている。(例:「OpenShift OperatorHub からの PostgreSQL データベースのインストール」)。

手順

1. OpenShift Container Platform Web コンソールで、クラスター管理者権限を持つアカウントを使用してログインします。

2. Service Registry および PostgreSQL Operator がインストールされている OpenShift プロジェクトに切り替えます。たとえば、**Project** ドロップダウンから、**my-project** を選択します。

3. Service Registry ストレージの PostgreSQL データベースを作成します。たとえば、**Installed Operators** > **PostgreSQL Operator by Dev4Ddevs.com** > **Create database** > **YAML** とクリックします。

4. 以下のようにデータベース設定を編集します。

   - **name**: 値を **registry** に変更します
   - **image**: 値を **centos/postgresql-10-centos7**に変更します。

5. 実際の環境に応じて、必要に応じてその他のデータベース設定を編集します。以下に例を示します。

```
apiVersion: postgresql.dev4devs.com/v1alpha1
kind: Database
metadata:
  name: registry
  namespace: my-project
spec:
  databaseCpu: 30m
  databaseCpuLimit: 60m
  databaseMemoryLimit: 512Mi
  databaseMemoryRequest: 128Mi
  databaseName: example
  databaseNameKeyEnvVar: POSTGRESQL_DATABASE
  databasePassword: postgres
  databasePasswordKeyEnvVar: POSTGRESQL_PASSWORD
  databaseStorageRequest: 1Gi
  databaseUser: postgres
  databaseUserKeyEnvVar: POSTGRESQL_USER
  image: centos/postgresql-10-centos7
  size: 1
```

6. **Create Database** を クリックし、データベースが作成されるまで待ちます。

7. **Installed Operators** > **Red Hat Integration - Service Registry**> **ApicurioRegistry** > **Create ApicurioRegistry** とクリックします。

8. 以下のカスタムリソース定義に貼り付け、データベース **url** およびクレデンシャルの値を編集して環境と一致するようにします。

```
apiVersion: apicur.io/v1alpha1
kind: ApicurioRegistry
metadata:
  name: example-apicurioregistry
spec:
  configuration:
    persistence: "jpa"
    dataSource:
      url: "jdbc:postgresql://SERVICE_NAME.NAMESPACE.svc:5432/"
      # e.g. url: "jdbc:postgresql://acid-minimal-cluster.my-project.svc:5432/"
      userName: "postgres"
      password: "PASSWORD"
      # ^ Optional
```

9. **Create** をクリックし、OpenShift で Service Registry ルートが作成されるまで待機します。

10. **Networking** > **Route** をクリックして、Service Registry Web コンソールの新規ルートにアクセスします。以下に例を示します。

```
http://example-apicurioregistry.my-project.my-domain-name.com/
```

関連情報

- Crunchy PostgreSQL Operator QuickStart
- Apicurio Registry Operator QuickStart

6.3. SERVICE REGISTRY POSTGRESQL ストレージのバックアップ

PostgreSQL データベースで Java Persistence API ストレージを使用する場合は、Service Registry に保存されているデータを定期的にバックアップする必要があります。

SQL Dumpは、どのような PostgreSQL インストールでも動作するシンプルな手順です。これは pg_dump ユーティリティーを使用して、ダンプ時と同じ状態でデータベースを再作成するために使用できる SQL コマンドでファイルを生成します。

pg_dump は通常の PostgreSQL クライアントアプリケーションで、データベースにアクセスできる任意のリモートホストから実行することができます。他のクライアントと同様に、実行できる操作はユーザーの権限によって制限されます。

手順

- **pg_dump** コマンドを使用して、出力をファイルにリダイレクトします。

  ```
  $ pg_dump dbname > dumpfile
  ```

  -hhost および **-pport** オプションを使用して、 **pg_dump** が接続するデータベースサーバーを指定できます。

- gzip などの圧縮ツールを使用して大きなダンプファイルを減らすことができます。以下に例を示します。

  ```
  $ pg_dump dbname | gzip > filename.gz
  ```

関連情報

クライアント認証の詳細は、PostgreSQL のドキュメント を参照してください。

6.4. SERVICE REGISTRY POSTGRESQL ストレージの復元

psql ユーティリティーを使用して、 **pg_dump** によって作成された SQL Dump ファイルを復元できます。

前提条件

- **pg_dump** を使用して、PostgreSQL データベースをすでにバックアップしている。「Service Registry PostgreSQL ストレージのバックアップ」を参照してください。
- オブジェクトを所有するユーザー、またはダンプされたデータベースのオブジェクトに対する権限があるユーザーがすべて存在している。

手順

1. 以下のコマンドを入力して、データベースを作成します。

```
$ createdb -T template0 dbname
```

2. 以下のコマンドを入力して SQL ダンプを復元します

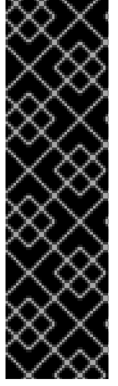
```
$ psql dbname < dumpfile
```

3. クエリーオプティマイザーが便利な統計を持つように、各データベースで ANALYZE を実行します。

第7章 INFINISPAN での組み込み SERVICE REGISTRY ストレージのデプロイ

本章では、埋め込み Infinispan キャッシュで Service Registry ストレージを設定する方法を説明します。

- 「OpenShift での組み込み Infinispan ストレージでの Service Registry の設定」



重要

Infinispan の Service Registry ストレージはテクノロジープレビュー機能です。テクノロジープレビュー機能は Red Hat の実稼働環境でのサービスレベルアグリーメント (SLA) ではサポートされていないため、Red Hat では実稼働環境での使用を推奨していません。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。

これらの機能は、近々発表予定の製品機能をリリースに先駆けてご提供することにより、お客様は機能性をテストし、開発プロセス中にフィードバックをお寄せいただくことができます。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、https://access.redhat.com/ja/support/offerings/techpreview を参照してください。

前提条件

- *4章OpenShift での Service Registry のインストール*

7.1. OPENSHIFT での組み込み INFINISPAN ストレージでの SERVICE REGISTRY の設定

ここでは、OpenShift 上の Service Registry に Infinispan キャッシュベースのストレージを設定する方法を説明します。このストレージオプションは、Quarkus ベースの Service Registry サーバーに埋め込まれた Infinispan コミュニティー Java ライブラリーに基づいています。このストレージオプションを使用して個別の Infinispan サーバーをインストールする必要はありません。このオプションは開発またはデモのみに適していますが、実稼働環境には適していません。

前提条件

- クラスター管理者として OpenShift クラスターにアクセスできる。
- Service Registry がすでにインストールされている。*4章OpenShift での Service Registry のインストール*を参照してください。

手順

1. OpenShift Container Platform Web コンソールで、クラスター管理者権限を持つアカウントを使用してログインします。
2. **Installed Operators** > **Red Hat Integration - Service Registry**> **ApicurioRegistry** > **Create ApicurioRegistry** とクリックします。
3. 以下のカスタムリソース定義に貼り付けます。

```
apiVersion: apicur.io/v1alpha1
kind: ApicurioRegistry
metadata:
```

```
  name: example-apicurioregistry
spec:
  configuration:
    persistence: "infinispan"
    infinispan: # Currently uses embedded version of Infinispan
      clusterName: "example-apicurioregistry"
      # ^ Optional
```

4. **Create** をクリックし、OpenShift で Service Registry ルートが作成されるまで待機します。

5. **Networking** > **Route** をクリックして、Service Registry Web コンソールの新規ルートにアクセスします。以下に例を示します。

```
http://example-apicurioregistry.my-project.my-domain-name.com/
```

関連情報

- Infinispan クラスターの設定に関する詳細は、Apicurio レジストリーのデモ で利用可能なカスタムリソースの例を参照してください。

- Infinispan の詳細は、https://infinispan.org/を参照してください。

第8章 SERVICE REGISTRY デプロイメントの設定および管理

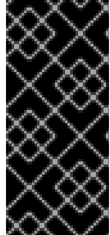
本章では、OpenShift での Service Registry デプロイメントのオプションの設定および管理方法について説明します。

- 「OpenShift での Service Registry ヘルスチェックの設定」
- 「Service Registry ヘルスチェックの環境変数」
- 「OpenShift クラスター内から Service Registry への HTTPS 接続の設定」
- 「OpenShift クラスター外から Service Registry への HTTPS 接続の設定」

8.1. OPENSHIFT での SERVICE REGISTRY ヘルスチェックの設定

liveness および readiness プローブのオプションの環境変数を設定して、OpenShift の Service Registry サーバーの健全性を監視できます。

- アプリケーションが進行可能な場合は **liveness プローブ** のテスト。アプリケーションが進行不可能な場合、OpenShift は障害のある Pod を自動的に再起動します。
- アプリケーションが要求を処理する準備ができている場合は**readiness プローブ** のテスト。アプリケーションが準備できていない場合、リクエストに圧倒されてしまい、プローブが失敗した期間は OpenShiftがリクエストの送信を停止します。他の Pod が OK の場合は、引き続き要求を受け取ります。



重要

liveness および readiness 環境変数のデフォルト値はほとんどの場合を想定して設計されており、環境で必要とされる場合にのみ変更する必要があります。デフォルトへの変更は、ハードウェア、ネットワーク、および保存されたデータ量によって異なります。これらの値は、不要なオーバーヘッドを回避するために、できるだけ低く抑える必要があります。

前提条件

- クラスター管理者として OpenShift クラスターにアクセスできる。
- 任意のストレージオプションを使用して、OpenShift に Service Registry がすでにインストールされている。*4章OpenShift での Service Registry のインストール* を参照してください。
- AMQ Streams、埋め込み Infinispan、または PostgreSQL に選択した Service Registry ストレージがインストールされ、設定されている必要があります。

手順

1. OpenShift Container Platform Web コンソールで、クラスター管理者権限を持つアカウントを使用してログインします。
2. **Installed Operators** > **Red Hat Integration - Service Registry**をクリックします。
3. **ApicurioRegistry** タブで、**example-apicurioregistry** などのデプロイメントの Operator カスタムリソースをクリックします。

4. メインの概要ページで、**Deployment Name** セクションと Service Registry デプロイメントの対応する **DeploymentConfig** 名を見つけます (例: **example-apicurioregistry**)。

5. 左側のナビゲーションメニューで**Workloads** > **Deployment Configs** をクリックし、**DeploymentConfig** 名を選択します。

6. **Environment** タブをクリックして、**Single values env** セクションに環境変数を入力します。以下に例を示します。

   - NAME: **LIVENESS_STATUS_RESET**
   - VALUE: **350**

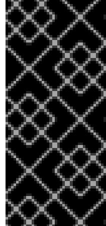
7. 下部にある **Save** をクリックします。
   代わりに、OpenShift **oc** コマンドを使用して、これらの手順を実行することもできます。詳細は、OpenShift CLI のドキュメント を参照してください。

その他のリソース

- 「Service Registry ヘルスチェックの環境変数」
- Monitoring application health

8.2. SERVICE REGISTRY ヘルスチェックの環境変数

このセクションでは、OpenShift の Service Registry ヘルスチェックに使用できる環境変数について説明します。これには、OpenShift 上の Service Registry サーバーの健全性を監視する liveness および readiness プローブが含まれます。手順の例は、「OpenShift での Service Registry ヘルスチェックの設定」 を参照してください。



重要

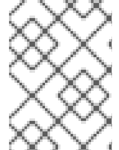
以下の環境変数は参考としてのみ提供されます。デフォルト値はほとんどの場合を想定して設計されており、環境に必要な場合のみ変更する必要があります。デフォルトへの変更は、ハードウェア、ネットワーク、および保存されたデータ量によって異なります。これらの値は、不要なオーバーヘッドを回避するために、できるだけ低く抑える必要があります。

liveness 環境変数

表8.1 Service Registry liveness プローブの環境変数

名前	詳細	型	デフォルト
LIVENESS_ERROR_THRESHOLD	liveness プローブが失敗するまでに発生する可能性のある liveness の問題またはエラーの数。	Integer	**1**

名前	詳細	型	デフォルト
LIVENESS_COUNTER_RESET	しきい値となる数のエラーが発生する期間。たとえば、この値が 60 でしきい値が 1 の場合、1 分間に 2 件のエラーが発生するとチェックが失敗します。	秒	**60**
LIVENESS_STATUS_RESET	liveness プローブが OK ステータスにリセットされるために、エラーなしで経過する必要のある秒数。	秒	**300**
LIVENESS_ERRORS_IGNORED	無視された liveness 例外のコンマ区切りリスト。	文字列	**io.grpc.StatusRuntimeException,org.apache.kafka.streams.errors.InvalidStateStoreException**



注記

OpenShift は liveness チェックに失敗した Pod を自動的に再起動するため、liveness 設定は readiness 設定とは異なり、OpenShift 上の Service Registry の動作に直接影響を与えません。

readiness 環境変数

表8.2 Service Registry readiness プローブの環境変数

名前	詳細	型	デフォルト
READINESS_ERROR_THRESHOLD	readiness プローブが失敗するまでに発生する可能性のある readiness の問題またはエラーの数。	Integer	**1**
READINESS_COUNTER_RESET	しきい値となる数のエラーが発生する期間。たとえば、この値が 60 でしきい値が 1 の場合、1 分間に 2 件のエラーが発生するとチェックが失敗します。	秒	**60**
READINESS_STATUS_RESET	liveness プローブが OK ステータスにリセットされるために、エラーなしで経過する必要のある秒数。ここでは、Pod が通常の動作に戻るまでの準備ができていない状態の期間を意味します。	秒	**300**

名前	詳細	型	デフォルト
READINESS_TIMEOUT	readiness は 2 つの操作のタイムアウトを追跡します。 • ストレージリクエストが完了するまでの時間 • HTTP REST API リクエストが応答を返すまでの時間 これらの操作に設定されたタイムアウトよりも時間がかかった場合、これは readiness 問題またはエラーとしてカウントされます。この値は、両方の操作のタイムアウトを制御します。	秒	**5**

関連情報

- 「OpenShift での Service Registry ヘルスチェックの設定」
- Monitoring application health

8.3. OPENSHIFT クラスター内から SERVICE REGISTRY への HTTPS 接続の設定

以下の手順では、OpenShift クラスター内から HTTPS 接続のポートを公開するように Service Registry デプロイメントを設定する方法を説明します。

警告

このような接続は、クラスター外部で直接利用できません。ルーティングはホスト名に基づいており、HTTPS 接続の場合はエンコードされます。そのため、エッジターミネーションまたはその他の設定は必要です。「OpenShift クラスター外から Service Registry への HTTPS 接続の設定」 を参照してください。

前提条件

- Service Registry Operator がインストールされている。

手順

1. 自己署名証明書を使用して **keystore** を生成します。独自の証明書を使用している場合は、この手順を省略できます。

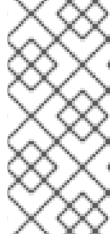
```
keytool -genkey -trustcacerts -keyalg RSA -keystore registry-keystore.jks -storepass
password
```

2. キーストアおよびキーストアのパスワードを保持する新しいシークレットを作成します。

   a. OpenShift Web コンソールの左側のナビゲーションメニューで、**Workloads** > **Secrets** > **Create Key/Value Secret** とクリックします。

   b. 次の値を使用します。

      - 名前： **registry-keystore**
      - キー 1: **keystore.jks**
      - 値 1: **registry-keystore.jks** （アップロードしたファイル）
      - キー 2: **パスワード**
      - 値 2: **パスワード**



注記

java.io.IOException: Invalid keystore format が発生した場合、バイナリーファイルのアップロードは正しく機能しませんでした。別の方法として、**cat registry-keystore.jks | base64 -w0 > data.txt** を使用してファイルを base64 文字列にエンコードし、yaml として **Secret** リソースを編集してエンコードされたファイルを手動で追加してください。

3. Service Registry インスタンスの **DeploymentConfig** リソースを編集します。Service Registry Operator の status フィールドで正しい名前を見つけることができます。

   a. キーストアシークレットをボリュームとして追加します。

```
template:
  spec:
    volumes:
    - name: registry-keystore-secret-volume
      secret:
      secretName: registry-keystore
```

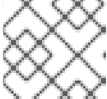
   b. ボリュームマウントを追加します。

```
volumeMounts:
  - name: registry-keystore-secret-volume
    mountPath: /etc/registry-keystore
    readOnly: true
```

   c. **JAVA_OPTIONS** および **KEYSTORE_PASSWORD** 環境変数を追加します。

```
- name: KEYSTORE_PASSWORD
  valueFrom:
    secretKeyRef:
      name: registry-keystore
      key: password
- name: JAVA_OPTIONS
    value: >-
```

```
-Dquarkus.http.ssl.certificate.key-store-file=/etc/registry-keystore/keystore.jks
-Dquarkus.http.ssl.certificate.key-store-file-type=jks
-Dquarkus.http.ssl.certificate.key-store-password=$(KEYSTORE_PASSWORD)
```



注記

順序は、文字列の補間を使用する場合に重要です。

d. HTTPS ポートを有効にします。

```
ports:
  - containerPort: 8080
    protocol: TCP
  - containerPort: 8443
    protocol: TCP
```

4. Service Registry インスタンスの **Service** リソースを編集します。Service Registry Operator の status フィールドで正しい名前を見つけることができます。

```
ports:
  - name: http
    protocol: TCP
    port: 8080
    targetPort: 8080
  - name: https
    protocol: TCP
    port: 8443
    targetPort: 8443
```

5. 接続が機能していることを確認します。

a. SSH を使用してクラスターの Pod に接続します (Service Registry Pod を使用できます)。

```
oc rsh -n default example-apicurioregistry-deployment-vx28s-4-lmtqb
```

b. **Service**リソースから Service Registry Pod のクラスター IP を見つけます (Web コンソールの **Location** 列を参照)。

c. その後、テスト要求を実行します (自己署名証明書を使用するので、セキュアでないフラグが必要になります)。

```
curl -k https://172.30.209.198:8443/health
[...]
```

8.4. OPENSHIFT クラスター外から SERVICE REGISTRY への HTTPS 接続の設定

以下の手順では、OpenShift クラスター外からの接続に対して HTTPS エッジターミネーションを使用したルートを公開するために Service Registry デプロイメントを設定する方法を説明します。

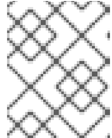
前提条件

- Service Registry Operator がインストールされている。
- セキュアなルートを作成するための OpenShift ドキュメント を読む。

手順

- Service Registry Operator によって作成される HTTP ルートの他に、2 つ目の **Route** を追加します。以下の例を参照してください。

```
kind: Route
apiVersion: route.openshift.io/v1
metadata:
  [...]
  labels:
    app: example-apicurioregistry
    [...]
spec:
  host: example-apicurioregistry-default.apps.example.com
  to:
    kind: Service
    name: example-apicurioregistry-service-9whd7
    weight: 100
  port:
    targetPort: 8080
  tls:
    termination: edge
    insecureEdgeTerminationPolicy: Redirect
  wildcardPolicy: None
```



注記

insecureEdgeTerminationPolicy: Redirect 設定プロパティーが設定されていることを確認してください。

- 証明書を指定しない場合、OpenShift はデフォルトを使用します。または、以下のコマンドを使用してカスタムの自己署名証明書を生成することもできます。

```
openssl genrsa 2048 > host.key &&
openssl req -new -x509 -nodes -sha256 -days 365 -key host.key -out host.cert
```

次に、OpenShift CLI を使用してルートを作成します。

```
oc create route edge \
  --service=example-apicurioregistry-service-9whd7 \
  --cert=host.cert --key=host.key \
  --hostname=example-apicurioregistry-default.apps.example.com \
  --insecure-policy=Redirect \
  -n default
```

第9章 WEB コンソールを使用した SERVICE REGISTRY コンテンツの管理

本章では、Service Registry Web コンソールを使用して、レジストリーに保存されているアーティファクトを管理する方法を説明します。これには、レジストリーコンテンツのアップロードと参照、およびオプションのルールの設定が含まれます。

- 「Service Registry Web コンソールの設定」
- 「Service Registry Web コンソールを使用したアーティファクトの追加」
- 「Service Registry Web コンソールを使用したアーティファクトの表示」
- 「Service Registry Web コンソールを使用したコンテンツルールの設定」

9.1. SERVICE REGISTRY WEB コンソールの設定

デプロイメント環境専用の Service Registry Web コンソールを設定したり、その動作をカスタマイズしたりすることができます。本セクションでは、Service Registry Web コンソールにオプションの環境変数を設定する方法を説明します。

前提条件

- Service Registry がすでにインストールされている。

Web コンソールのデプロイメント環境の設定

ユーザーがブラウザーで Service Registry Web コンソールに移動すると、一部の初期設定が読み込まれます。以下の 2 つの重要な設定プロパティーがあります。

- バックエンド Service Registry REST API の URL
- フロントエンド Service Registry Web コンソールの URL

通常、Service Registry はこれらの設定を自動的に検出して生成しますが、、一部のデプロイメント環境ではこの自動検出が失敗する場合があります。その場合には、環境のこれらの URL を明示的に設定するように環境変数を設定できます。

手順

以下の環境変数を設定し、デフォルトの URL を上書きします。

- **REGISTRY_UI_CONFIG_APIURL**: バックエンド Service Registry REST API の URL を設定します。例：**https://registry.my-domain.com/api**
- **REGISTRY_UI_CONFIG_UIURL**: フロントエンド Service Registry Web コンソールの URL を設定します。たとえば、**https://registry.my-domain.com/ui**

読み取り専用モードでのコンソールの設定

オプション機能として、Service Registry の Web コンソールを読み取り専用モードに設定することができます。このモードでは、Service Registry Webコンソールでユーザーが登録されたアーティファクトを変更できる機能がすべて無効になります。たとえば、これには以下が含まれます。

- アーティファクトの作成
- アーティファクトの新規バージョンのアップロード

- アーティファクトのメタデータの更新
- アーティファクトの削除

手順

以下の環境変数を設定して、Service Registry の Web コンソールを読み取り専用モードにします。

- **REGISTRY_UI_FEATURES_READONLY**: **true** に設定すると、読み取り専用モードが有効になります。デフォルトは **false** です。

9.2. SERVICE REGISTRY WEB コンソールを使用したアーティファクトの追加

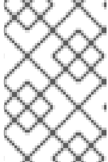
Service Registry Web コンソールを使用して、イベントスキーマと API デザインアーティファクトをレジストリーにアップロードできます。アップロード可能なアーティファクトタイプに関する詳細は、14 章*Service Registry アーティファクトの参照* を参照してください。本セクションでは、Service Registry アーティファクトのアップロード、アーティファクトルールの適用、および新しいアーティファクトバージョンの追加の簡単な例を紹介します。

前提条件

- Service Registry が環境にインストールされ、実行されている。

手順

1. Service Registry Web コンソールに接続します。
   http://MY_REGISTRY_URL/ui
2. **Upload Artifact** をクリックし、以下を指定します。
   - id: デフォルトの空の設定を使用して **ID** を自動的に生成するか、特定のアーティファクト ID を入力します。
   - **Type**: デフォルトの **Auto-Detect** 設定を使用してアーティファクトタイプを自動的に検出し、ドロップダウンからアーティファクトタイプを選択します (例: **Avro Schema** または **OpenAPI**)。

     注記

     Service Registry サーバーは、**Kafka Connect** スキーマアーティファクトタイプを自動的に検出できません。このアーティファクトタイプを手動で選択する必要があります。

   - **Artifact**: ドラッグアンドドロップまたは **Browse** をクリックして、 **my-schema.json** や **my-openapi.json** などのファイルをアップロードします。
3. **Upload** をクリックし、**Artifact Details** を表示します。

図9.1 Service Registry Web コンソールのアーティファクトの詳細

Artifacts > 76088d93-6010-4b80-a7f9-180912239106

Artifact Details

Version: latest ▾ Upload new version

Info | Content

FullName

An artifact of type AVRO with no description.

Status ENABLED
Created an hour ago
Modified an hour ago

Download

Content Rules

Validity Rule | Ensure that content is *valid* when updating this artifact. | Full ▾

Compatibility Rule | Enforce a compatibility level when updating this artifact (e.g. Backwards Compatibility). | Enable

- **info**: アーティファクト名、説明、ライフサイクルのステータス、作成時、および最終更新時を表示します。**Edit Artifact Metadata** 鉛筆アイコンをクリックしてアーティファクト名と説明を編集するか、またはラベルを追加し、**Download** をクリックしてアーティファクトファイルをローカルにダウンロードできます。また、有効化および設定できるアーティファクト**コンテンツルール**も表示します。

- **ドキュメント** (OpenAPI のみ): 自動生成される REST API ドキュメントを表示します。

- **Content**: 全アーティファクトコンテンツの読み取り専用ビューを表示します。

4. **Content Rules** で **Enable** をクリックして **Validity Rule** または **Compatibility Rule** を設定し、ドロップダウンから適切なルール設定を選択します。詳細は 14章*Service Registry アーティファクトの参照* を参照してください。

5. **Upload new version** をクリックして新しいアーティファクトバージョンを追加し、ドラッグアンドドロップまたは **Browse** をクリックしてファイル (**my-schema.json** や **my-openapi.json** など) をアップロードします。

6. アーティファクトを削除するには、**Upload new versio** の横にあるゴミ箱アイコンをクリックします。

警告

アーティファクトを削除すると、アーティファクトとそのバージョンがすべて削除され、元に戻すことはできません。アーティファクトバージョンはイミュータブルで、個別に削除できません。

その他のリソース

- 「Service Registry Web コンソールを使用したアーティファクトの表示」
- 「Service Registry Web コンソールを使用したコンテンツルールの設定」

9.3. SERVICE REGISTRY WEB コンソールを使用したアーティファクトの表示

Service Registry Web コンソールを使用して、レジストリーに保存されているイベントスキーマおよび

API デザインアーティファクトを参照できます。本セクションでは、Service Registry アーティファクト、バージョン、およびアーティファクトルールを表示する簡単な例を紹介します。レジストリーに保存されているアーティファクトタイプの詳細は、14章*Service Registry アーティファクトの参照* を参照してください。

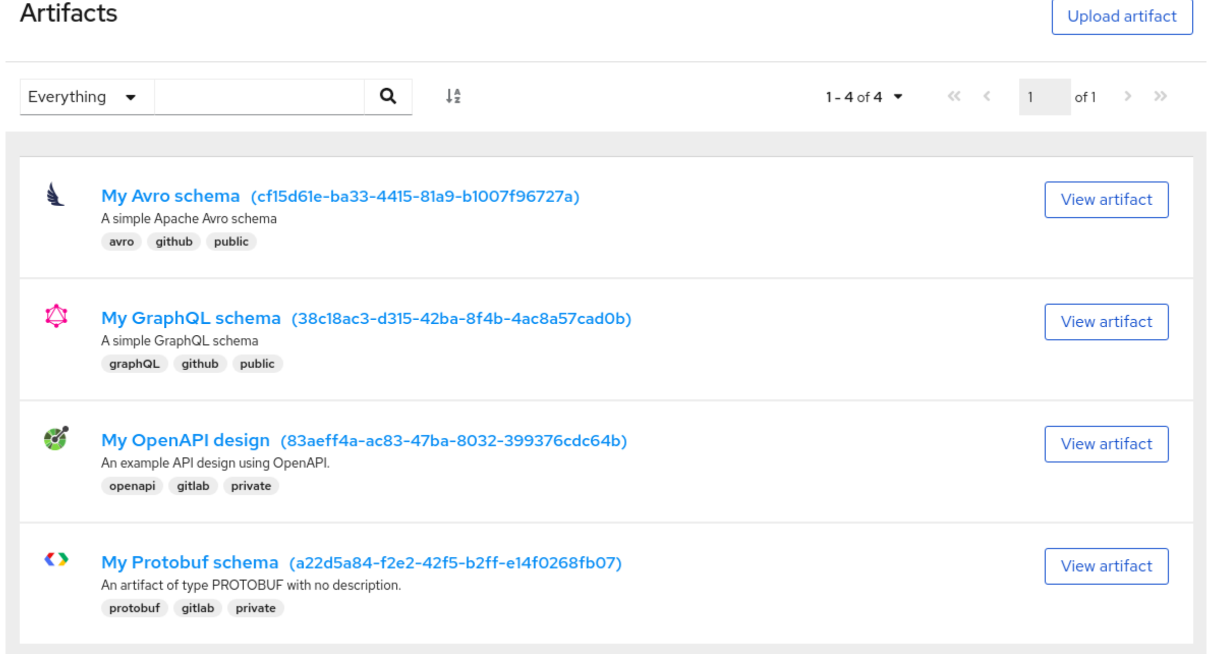
前提条件

- Service Registry が環境にインストールされ、実行されている。
- Service Registry Web コンソール、REST API コマンド、Maven プラグイン、または Java クライアントアプリケーションを使用して、アーティファクトがレジストリーに追加されている必要があります。

手順

1. Service Registry Web コンソールに接続します。
   http://MY_REGISTRY_URL/ui
2. レジストリーに保存されているアーティファクト一覧を参照するか、検索文字列を入力してアーティファクトを検索します。特定の **Name**、**Description**、**Label**、または **Everything** で検索できます。

   図9.2 Service Registry Web コンソールでのアーティファクトの閲覧

3. **View artifact** をクリックして **Artifact Details** を表示します。
   - **info**: アーティファクト名、説明、ライフサイクルのステータス、作成時、および最終更新時を表示します。**Edit Artifact Metadata** 鉛筆アイコンをクリックしてアーティファクト名と説明を編集するか、またはラベルを追加し、**Download** をクリックしてアーティファクトファイルをローカルにダウンロードできます。また、有効化および設定できるアーティファクト**コンテンツルール**も表示します。
   - **ドキュメント** (OpenAPI のみ): 自動生成される REST API ドキュメントを表示します。
   - **Content**: 全アーティファクトコンテンツの読み取り専用ビューを表示します。

4. 追加バージョンが追加されている場合は、ドロップダウンから異なるアーティファクト**Version** を表示します。

その他のリソース

- 「Service Registry Web コンソールを使用したアーティファクトの追加」
- 「Service Registry Web コンソールを使用したコンテンツルールの設定」

9.4. SERVICE REGISTRY WEB コンソールを使用したコンテンツルールの設定

Service Registry Web コンソールを使用して、無効なコンテンツがレジストリーに追加されないようにオプションのルールを設定できます。設定されたアーティファクトルールまたはグローバルルールはすべて、新しいアーティファクトバージョンをレジストリーにアップロードする前に渡す必要があります。設定されたアーティファクトルールは、設定されたグローバルルールを上書きします。詳細は 2章*Service Registry のコンテンツルール* を参照してください。

本セクションでは、グローバルルールとアーティファクトルールを設定する簡単な例を紹介します。選択可能なさまざまなルールタイプおよび関連する設定の詳細は、14章*Service Registry アーティファクトの参照* を参照してください。

前提条件

- Service Registry が環境にインストールされ、実行されている。
- アーティファクトルールの場合、Service Registry Web コンソール、REST API コマンド、Maven プラグイン、または Java クライアントアプリケーションを使用して、アーティファクトがレジストリーに追加されている必要があります。

手順

1. Service Registry Web コンソールに接続します。
   http://MY_REGISTRY_URL/ui
2. アーティファクトルールの場合、レジストリーに保存されているアーティファクト一覧を参照するか、検索文字列を入力してアーティファクトを検索します。特定のアーティファクト **Name**、**Description**、**Label**、または **Everything** で検索できます。
3. **View artifact** をクリックして **Artifact Details** を表示します。
4. **Content Rules** で **Enable** をクリックしてアーティファクトの **有効性ルール**または **互換性ルール**を設定し、ドロップダウンから適切なルール設定を選択します。詳細は 14章*Service Registry アーティファクトの参照* を参照してください。

図9.3 Service Registry Web コンソールでのコンテンツルールの設定

Content Rules			
Validity Rule	Ensure that content is *valid* when updating this artifact.	Full	
Compatibility Rule	Enforce a compatibility level when updating this artifact (e.g. Backwards Compatibility).	Enable	

5. グローバルルールの場合は、ツールバーの右上の **Settings** cog アイコンをクリックし、**Enable** をクリックしてグローバル **Validity Rule** または **Compatibility Rule** を設定し、ドロップダウンから適切なルール設定を選択します。詳細は 14章*Service Registry アーティファクトの参照* を参照してください。

6. アーティファクトルールまたはグローバルルールを無効にするには、ルールの横にあるゴミ箱アイコンをクリックします。

その他のリソース

- 「Service Registry Web コンソールを使用したアーティファクトの追加」

第10章 REST API を使用した SERVICE REGISTRY コンテンツの管理

本章では、Registry REST API について説明し、レジストリーに保存されているアーティファクトを管理する方法を説明します。

- 「Registry REST API の概要」
- 「Registry REST API コマンドを使用したアーティファクトの管理」

その他のリソース

- Apicurio Registry REST API ドキュメント

10.1. REGISTRY REST API の概要

Registry REST API を使用すると、クライアントアプリケーションは Service Registry のアーティファクトを管理できます。この API では、以下を行うために作成、読み取り、更新、および削除の操作が提供されます。

アーティファクト

レジストリーに保存されているスキーマおよび API デザインアーティファクトを管理します。これには、名前、ID、説明、ラベルなどでアーティファクトの参照または検索も含まれます。アーティファクトのライフサイクル状態 (enabled、disabled、または deprecated) を管理することもできます。

アーティファクトのバージョン

アーティファクトコンテンツの更新時に作成されるバージョンを管理します。これには、名前、ID、説明、ラベルなど、バージョンの閲覧または検索も含まれます。バージョンのライフサイクル状態（enabled、disabled、または deprecated）を管理することもできます。

アーティファクトのメタデータ

アーティファクトの作成または変更時、その現在の状態などのアーティファクトの詳細を管理します。一部のメタデータを編集できますが、一部は読み取り専用です。たとえば、編集可能なメタデータにはアーティファクト名、説明、またはラベルが含まれますが、アーティファクトが作成され、変更された場合は読み取り専用になります。

グローバルルール

すべてのアーティファクトのコンテンツ展開を管理するルールを設定し、無効または互換性のないコンテンツがレジストリーに追加されないようにします。グローバルルールは、アーティファクトに独自の特定のアーティファクトルールが設定されていない場合にのみ適用されます。

アーティファクトルール

特定のアーティファクトのコンテンツ展開を管理するルールを設定し、無効または互換性のないコンテンツがレジストリーに追加されないようにします。アーティファクトルールは、設定されたグローバルルールを上書きします。

他のスキーマレジストリーとの互換性

Registry REST API は、Apache Avro、Google Protocol バッファー、および JSON スキーマアーティファクトタイプのサポートが含まれる Confluent スキーマレジストリー REST API と互換性があります。Confluent クライアントライブラリーを使用するアプリケーションは Service Registry をドロップイン置換として使用できます。詳細は、「Replacing Confluent Schema Registry with Red Hat Integration Service Registry」を参照してください。

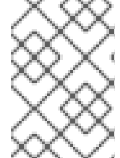
その他のリソース

- 詳細は、Apicurio Registry REST API のドキュメント を参照してください。
- Registry REST API ドキュメントは、Service Registry デプロイメントのメインエンドポイントからも利用できます（例： **http://MY-REGISTRY-URL/api** ）。

10.2. REGISTRY REST API コマンドを使用したアーティファクトの管理

クライアントアプリケーションは、Registry REST API コマンドを使用して、Service Registry のアーティファクトを管理できます（たとえば、実稼働環境にデプロイされた CI/CD パイプラインなど）。Registry REST API は、レジストリーに保存されるアーティファクト、バージョン、メタデータ、およびルールの作成、読み取り、更新、および削除操作を提供します。詳細は、Apicurio Registry REST API のドキュメント を参照してください。

本セクションでは、レジストリー REST API を使用してレジストリーに Apache Avro スキーマアーティファクトを追加および取得する簡単な curl ベースの例を紹介します。



注記

REST API を使用して Service Registry にアーティファクトを追加する場合、一意のアーティファクト ID を指定しない場合、Service Registry は UUID として自動的に生成します。

前提条件

- 1章*Service Registry の概要* を参照してください。
- Service Registry が環境にインストールされ、実行されている。

手順

1. **/artifacts** 操作を使用して、レジストリーにアーティファクトを追加します。以下の **curl** コマンドの例は、株価アプリケーションの単純なアーティファクトを追加します。

   ```
   $ curl -X POST -H "Content-type: application/json; artifactType=AVRO" -H "X-Registry-ArtifactId: share-price" --data
   '{"type":"record","name":"price","namespace":"com.example","fields":
   [{"name":"symbol","type":"string"},{"name":"price","type":"string"}]}' http://MY-REGISTRY-HOST/api/artifacts
   ```

   以下の例では、**共有価格** のアーティファクト ID を持つ Avro スキーマアーティファクトを追加する方法を示します。

   MY-REGISTRY-HOST は、Service Registry がデプロイされているホスト名です。例: **my-cluster-service-registry-myproject.example.com**。

2. 応答に、アーティファクトが追加されたことを確認するために、想定される JSON ボディーが含まれていることを確認します。以下に例を示します。

   ```
   {"createdOn":1578310374517,"modifiedOn":1578310374517,"id":"share-price","version":1,"type":"AVRO","globalId":8}
   ```

3. アーティファクト ID を使用してレジストリーからアーティファクトを取得します。たとえば、この場合は、指定の ID が **share-price** になります。

```
$ curl http://MY-REGISTRY-URL/api/artifacts/share-price
'{"type":"record","name":"price","namespace":"com.example","fields":
[{"name":"symbol","type":"string"},{"name":"price","type":"string"}]}
```

関連情報

- REST API のサンプル要求の詳細は、Apicurio Registry REST API のドキュメント を参照してください。

第11章 MAVEN プラグインを使用した SERVICE REGISTRY コンテンツの管理

本章では、Service Registry Maven プラグインを使用してレジストリーに保存されているアーティファクトを管理する方法を説明します。

- 「Service Registry Maven プラグインを使用したアーティファクトの管理」

前提条件

- 1章*Service Registry の概要* を参照してください。
- Service Registry が環境にインストールされ、実行されている。
- Maven が使用している環境にインストールおよび設定されている。

11.1. SERVICE REGISTRY MAVEN プラグインを使用したアーティファクトの管理

Service Registry Maven プラグインを使用して、開発ビルドの一部としてレジストリーアーティファクトをアップロードまたはダウンロードできます。たとえば、このプラグインは、スキーマの更新がクライアントアプリケーションと互換性があることをテストおよび検証するのに便利です。

Maven プラグインを使用したアーティファクトの登録

おそらく、Maven プラグインの最も一般的なユースケースは、ビルド中にアーティファクトを登録することです。これは、**register** 実行目標を使用して実現できます。

手順

- Maven **pom.xml** ファイルを更新して、 **apicurio-registry-maven-plugin** を使用してアーティファクトを登録します。以下の例は、Apache Avro スキーマの登録を示しています。

```
<plugin>
  <groupId>io.apicurio</groupId>
  <artifactId>apicurio-registry-maven-plugin</artifactId>
  <version>${registry.version}</version>
  <executions>
    <execution>
      <phase>generate-sources</phase>
      <goals>
        <goal>register</goal> (1)
      </goals>
      <configuration>
        <registryUrl>http://my-cluster-service-registry-myproject.example.com/api</registryUrl>
(2)
        <artifactType>AVRO</artifactType>
        <artifacts>
          <schema1>${project.basedir}/schemas/schema1.avsc</schema1> (3)
        </artifacts>
      </configuration>
    </execution>
  </executions>
</plugin>
```

1. スキーマアーティファクトをレジストリーにアップロードするための実行目標として **register** を指定します。
2. **/api** エンドポイントで Service Registry URL を指定する必要があります。
3. アーティファクト ID と場所を使用して複数のアーティファクトをアップロードできます。

Maven プラグインを使用したアーティファクトのダウンロード

Maven プラグインを使用して Service Registry からアーティファクトをダウンロードすることもできます。これは、たとえば、登録されたスキーマからコードを生成する場合などに便利です。

手順

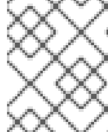
- Maven **pom.xml** ファイルを更新して、**apicurio-registry-maven-plugin** を使用してアーティファクトをダウンロードします。以下の例は、アーティファクト ID による単一のスキーマのダウンロードを示しています。

```
<plugin>
<groupId>io.apicurio</groupId>
<artifactId>apicurio-registry-maven-plugin</artifactId>
<version>${registry.version}</version>
<executions>
  <execution>
    <phase>generate-sources</phase>
    <goals>
      <goal>download</goal> (1)
    </goals>
    <configuration>
      <registryUrl>http://my-cluster-service-registry-myproject.example.com/api</registryUrl>
(2)
      <ids>
        <param1>schema1</param1> (3)
      </ids>
      <artifactExtension>.avsc</artifactExtension> (4)
      <outputDirectory>${project.build.directory}</outputDirectory>
   </configuration>
 </execution>
</executions>
</plugin>
```

1. 実行目標として **download** を指定します。
2. **/api** エンドポイントで Service Registry URL を指定する必要があります。
3. アーティファクト ID を使用すると、複数のアーティファクトを指定したディレクトリーにダウンロードできます。
4. プラグインは適切なファイル拡張子の選択を自動的に試みますが、**< artifactExtension>** を使用して上書きできます。

Maven プラグインを使用したアーティファクトのテスト

実際にアーティファクトを変更せずに、アーティファクトをが登録できることを確認する場合があります。これは、ルールが Service Registry に設定されている場合に最も便利です。アーティファクトのコンテンツが設定済みのルールのいずれかに違反する場合、アーティファクトのテストに失敗します。



注記

アーティファクトがテストに合格した場合でも、Service Registry にはコンテンツが追加されません。

手順

- Maven **pom.xml** ファイルを更新して、 **apicurio-registry-maven-plugin** を使用してアーティファクトをテストします。Apache Avro スキーマのテストの例を以下に示します。

```
<plugin>
  <groupId>io.apicurio</groupId>
  <artifactId>apicurio-registry-maven-plugin</artifactId>
  <version>${registry.version}</version>
  <executions>
    <execution>
      <phase>generate-sources</phase>
      <goals>
        <goal>test-update</goal> (1)
      </goals>
      <configuration>
        <registryUrl>http://my-cluster-service-registry-myproject.example.com/api</registryUrl>
(2)
        <artifactType>AVRO</artifactType>
        <artifacts>
          <schema1>${project.basedir}/schemas/schema1.avsc</schema1> (3)
        </artifacts>
      </configuration>
    </execution>
  </executions>
</plugin>
```

1. スキーマアーティファクトをテストするための実行目標として **test-update** を指定します。
2. **/api** エンドポイントで Service Registry URL を指定する必要があります。
3. アーティファクト ID と場所を使用して複数のアーティファクトをテストできます。

関連情報

- Service Registry Maven プラグインの詳細は、「Registry demonstration example」を参照してください。

第12章 JAVA クライアントを使用した SERVICE REGISTRY コンテンツの管理

本章では、Service Registry Java クライアントを使用する方法を説明します。

- 「Service Registry Java クライアント」
- 「Service Registry クライアントアプリケーションの作成」
- 「Service Registry Java client configuration」

12.1. SERVICE REGISTRY JAVA クライアント

Java クライアントアプリケーションを使用して、Service Registry に保存されているアーティファクトを管理できます。Service Registry Java クライアントクラスを使用して、レジストリーに保存されているアーティファクトを作成、読み取り、更新、または削除できます。

正しい依存関係をプロジェクトに追加すると、Service Registry Java クライアントにアクセスできます。「Service Registry クライアントアプリケーションの作成」 を参照してください。

Service Registry クライアントは自動的に閉じられ、Retrofit および OkHttp をベースライブラリーとして使用して実装されます。これにより、カスタムヘッダーの追加や TLS(Transport Layer Security)認証の有効化など、使用をカスタマイズすることができます。詳細は 「Service Registry Java client configuration」 を参照してください。

12.2. SERVICE REGISTRY クライアントアプリケーションの作成

本セクションでは、Java クライアントアプリケーションを使用して Service Registry に保存されているアーティファクトを管理する方法を説明します。Service Registry Java クライアントは **Autocloseable** インターフェースを拡張します。

前提条件

- *1章Service Registry の概要* を参照してください。
- Service Registry が環境にインストールされ、実行されている。

手順

1. 以下の依存関係を Maven プロジェクトに追加します。

```
<dependency>
    <groupId>io.apicurio</groupId>
    <artifactId>apicurio-registry-rest-client</artifactId>
    <version>${apicurio-registry.version}</version>
</dependency>
```

2. 以下のようにレジストリークライアントを作成します。

```
public class ClientExample {

    private static final RegistryRestClient client;

```

```
    public static void main(String[] args) throws Exception {
      // Create a registry client
      String registryUrl = "https://registry.my-domain.com/api"; (1)
      RegistryRestClient client = RegistryRestClientFactory.create(registryUrl); (2)
    }
}
```

1. **/api** エンドポイントで Service Registry URL を指定する必要があります。
2. Service Registry クライアント作成時の他のオプションは、次のセクションの Java クライアント設定を参照してください。

3. クライアントが作成されると、クライアントを介して Service Registry REST API からのすべての操作を使用することができます。詳細は、Apicurio Registry REST API のドキュメント を参照してください。

関連情報

- Service Registry クライアントを使用およびカスタマイズする例は、「 Registry client demonstration example 」を参照してください。
- AMQ Streams プロデューサーおよびコンシューマーアプリケーションで Apache Avro に Service Registry Kafka クライアントシリアライザー/デシリアライザーを使用する方法は、「 Using AMQ Streams on Openshift 」を参照してください。

12.3. SERVICE REGISTRY JAVA CLIENT CONFIGURATION

Service Registry Java クライアントには、クライアントファクトリーを基にした以下の設定オプションが含まれます。

表12.1 Service Registry Java クライアント設定オプション

オプション	説明	引数
プレーンクライアント	実行中のレジストリーと対話するために使用される基本的な REST クライアント。	**baseUrl**
カスタム HTTP クライアント	ユーザーによって提供される OkHttpClient を使用するレジストリークライアント。	**baseUrl, okhttpClient**
カスタム設定	カスタム設定を含むマップを受け入れるレジストリークライアント。これは、呼び出しにカスタムヘッダーを追加する場合に便利です。	**baseUrl、Map<String Object> configs**

カスタムヘッダー設定
カスタムヘッダーを設定するには、**configs** マップキーに **apicurio.registry.request.headers** プレフィックスを追加する必要があります。たとえば、値が **Basic:xxxxx** の **apicurio.registry.request.headers.Authorization** のキーは、値が **Basic: xxxxx** の **Authorization** のヘッダーになります。

TLS 設定

以下のプロパティーを使用して、Service Registry Java クライアントの Transport Layer Security (TLS) 認証を設定できます。

- **apicurio.registry.request.ssl.truststore.location**
- **apicurio.registry.request.ssl.truststore.password**
- **apicurio.registry.request.ssl.truststore.type**
- **apicurio.registry.request.ssl.keystore.location**
- **apicurio.registry.request.ssl.keystore.password**
- **apicurio.registry.request.ssl.keystore.type**
- **apicurio.registry.request.ssl.key.password**

第13章 KAFKA クライアントシリアライザー/デシリアライザーを使用したスキーマの検証

Service Registry は、プロデューサーおよびコンシューマーアプリケーションの Kafka クライアントシリアライザー/デシリアライザーを提供します。Kafka プロデューサーアプリケーションは、シリアライザーを使用して、特定のイベントスキーマに準拠するメッセージをエンコードします。Kafka コンシューマーアプリケーションはデシリアライザーを使用して、特定のスキーマ ID に基づいてメッセージが適切なスキーマを使用してシリアライズされたことを検証します。これにより、スキーマが一貫して使用されるようにし、実行時にデータエラーが発生しないようにします。

本章では、Kafka プロデューサーおよびコンシューマークライアントアプリケーションで、Apache Avro、JSON スキーマ、および Google Protobuf の Kafka クライアントシリアライザーとデシリアライザーを使用する方法を説明します。

- 「Kafka クライアントアプリケーションおよび Service Registry」
- 「スキーマを検索するストラテジー」
- 「Service Registry シリアライザー/デシリアライザー定数」
- 「異なるクライアントのシリアライザー/デシリアライザータイプの使用」
- 「Service Registry を使用した Avro SerDe の設定」
- 「Service Registry を使用した JSON スキーマ SerDe の設定」
- 「Service Registry を使用した Protobuf SerDe の設定」
- 「スキーマの Service Registry への登録」
- 「Kafka コンシューマークライアントからのスキーマの使用」
- 「Kafka プロデューサークライアントからのスキーマの使用」
- 「Kafka Streams アプリケーションからのスキーマの使用」

前提条件

- 読む必要があります。 1章*Service Registry の概要*
- Service Registry がインストールされている必要があります。
- Kafka プロデューサーおよびコンシューマークライアントアプリケーションが作成済みである必要があります。
  Kafka クライアントアプリケーションの詳細は、『 Using AMQ Streams on Openshift 』を参照してください。

13.1. KAFKA クライアントアプリケーションおよび SERVICE REGISTRY

Service Registry を使用すると、クライアントアプリケーション設定からスキーマ管理が分離されます。クライアントコードに URL を指定して、アプリケーションがレジストリーからスキーマを使用できるようにします。

たとえば、スキーマを保存して、レジストリーにメッセージをシリアライズおよびデシリアライズすることができます。次に、レジストリーを使用するアプリケーションから参照され、送受信されるメッ

セージとこれらのスキーマの互換性を維持するようにします。Kafka クライアントアプリケーションは、実行時にスキーマを Service Registry からプッシュまたはプルできます。

スキーマは進化するので、Service Registry でルールを定義できます。たとえば、スキーマへの変更が有効で、アプリケーションによって使用される以前のバージョンとの互換性を維持するようにします。Service Registry は、変更済みのスキーマと以前のスキーマバージョンを比較することで、互換性をチェックします。

Service Registry は、以下のような複数のスキーマ技術のスキーマレジストリーサポートを提供します。

- Avro
- Protobuf
- JSON スキーマ

これらのスキーマ技術は、Service Registry によって提供される Kafka クライアントのシリアライザー/デシリアライザー(SerDe)サービスを介してクライアントアプリケーションで使用できます。Service Registry によって提供される SerDe クラスの成熟度および使用法は異なる場合があります。それぞれのタイプ固有のセクションを参照してください。

プロデューサースキーマの設定

プロデューサークライアントアプリケーションは、シリアライザーを使用して、特定のブローカートピックに送信するメッセージを正しいデータ形式にします。

プロデューサーが Service Registry を使用してシリアライズできるようにするには、以下を行います。

- スキーマを Service Registry に定義、登録します （任意）。
- プロデューサークライアントコードを設定し ます。
  - Service Registry の URL
  - メッセージで使用する Service Registry シリアライザー
  - Kafka メッセージを Service Registry のアーティファクト ID にマップするストラテジー
  - Service Registry でのシリアライズに使用するスキーマを検索または登録するストラテジー

スキーマを登録したら、Kafka および Service Registry を開始するときに、スキーマにアクセスして、プロデューサーにより Kafka ブローカートピックに送信されるメッセージをフォーマットできます。または（設定により）、プロデューサーは初回使用時にスキーマを自動的に登録できます。

スキーマがすでに存在する場合、Service Registry に定義される互換性ルールに基づいて REST API を使用して新バージョンのスキーマを作成できます。バージョンは、スキーマの進化にともなう互換性チェックに使用します。アーティファクト ID およびスキーマバージョンは、スキーマを識別する一意のタプルを表します。

コンシューマースキーマの設定

コンシューマークライアントアプリケーションは、デシリアライザーを使用することで、そのアプリケーションが消費するメッセージを特定のブローカートピックから正しいデータ形式にします。

コンシューマーがデシリアライズに Service Registry を使用できるようにするには、以下を実行します。

- スキーマを Service Registry に定義、登録します。

- コンシューマークライアントコードを設定し ます。
  - Service Registry の URL
  - メッセージで使用する Service Registry デシリアライザー
  - デシリアライズの入力データストリーム

次に、消費されるメッセージに書き込まれたグローバル ID を使用して、デシリアライザーによってスキーマが取得されます。スキーマグローバル ID は、プロデューサーアプリケーションの設定に応じて、メッセージヘッダーまたはメッセージペイロード自体に置くことができます。

メッセージペイロードでグローバル ID を見つけると、データの形式は（コンシューマーへのシグナルとして）マジックバイトで始まり、その後にグローバル ID の後に、通常通りメッセージデータが続きます。

以下に例を示します。

```
# ...
[MAGIC_BYTE]
[GLOBAL_ID]
[MESSAGE DATA]
```

これで、Kafka および Service Registry を起動すると、スキーマにアクセスして、Kafka ブローカートピックから受け取ったメッセージをフォーマットできるようになりました。

13.2. スキーマを検索するストラテジー

Kafka クライアントシリアライザーは **検索ストラテジーを** 使用して、メッセージスキーマが Service Registry に登録されるアーティファクト ID およびグローバル ID を決定します。

特定のトピックおよびメッセージで、以下の Java インターフェースの実装を使用できます。

- **ArtifactIdStrategy**、アーティファクト ID を返す。
- **GlobalIdStrategy**、グローバル ID を返す。

各ストラテジーのクラスは、**io.apicurio.registry.utils.serde.strategy** パッケージに編成されます。デフォルトのストラテジーは、アーティファクト ID **TopicIdStrategy** で、メッセージを受信する Kafka トピックと同じ名前の Service Registry アーティファクトを検索します。

例

```
public String artifactId(String topic, boolean isKey, T schema) {
    return String.format("%s-%s", topic, isKey ? "key" : "value");
}
```

- **topic** パラメーターは、メッセージを受信する Kafka トピックの名前です。
- **isKey** パラメーターは、メッセージキーがシリアライズされる場合は **true**、メッセージ値がシリアライズされる場合は **false** です。
- **schema** パラメーターは、シリアライズまたはデシリアライズされるメッセージのスキーマです。
- 返される **artifactID** は、スキーマが Service Registry に登録されるアーティファクト ID です。

使用する検索アップストラテジーは、スキーマを保存する方法と場所によって異なります。たとえば、同じ Avro メッセージタイプを持つ Kafka トピックが複数ある場合、**レコード ID** を使用するストラテジーを使用することがあります。

アーティファクト ID ストラテジー

アーティファクト ID ストラテジーは、Kafka トピックおよびメッセージ情報を Service Registry のアーティファクト ID にマップする方法を提供します。マッピングの共通規則は、Kafka メッセージのキーと値のどちらにシリアライザーを使用するかによって、Kafka トピック名と **key** または **value** を結合することです。

ただし、Service Registry によって提供されるストラテジーを使用するか、**io.apicurio.registry.utils.serde.strategy.ArtifactIdStrategy** を実装するカスタム Java クラスを作成して、マッピングの代替規則を使用できます。

アーティファクト ID を返すストラテジー

Service Registry は、**ArtifactIdStrategy** の実装に基づいてアーティファクト ID を返す以下のストラテジーを提供します。

RecordIdStrategy

スキーマのフルネームを使用する Avro 固有のストラテジー。

TopicRecordIdStrategy

トピック名およびスキーマのフルネームを使用する Avro 固有のストラテジー。

TopicIdStrategy

トピック名と、**key** または **value** サフィックスを使用するデフォルトストラテジー。

SimpleTopicIdStrategy

トピック名のみを使用する単純なストラテジー。

グローバル ID ストラテジー

グローバル ID ストラテジーは、アーティファクト ID ストラテジーによって提供されるアーティファクト ID の下に登録されたスキーマの特定バージョンを見つけて識別します。すべてのアーティファクトのすべてのバージョンには、グローバルで一意の識別子が 1 つだけあり、それを使用してそのアーティファクトの内容を取得できます。このグローバル ID はすべての Kafka メッセージに含まれ、デシリアライザーは Service Registry からスキーマを適切にフェッチできます。

グローバル ID ストラテジーは、既存のアーティファクトバージョンを検索できます。見つからない場合は、使用するストラテジーに応じて登録できます。**io.apicurio.registry.utils.serde.strategy.GlobalIdStrategy** を実装するカスタム Java クラスを作成して、独自のストラテジーを指定することもできます。

グローバル ID を返すストラテジー

Service Registry は、**GlobalIdStrategy** の実装に基づいてグローバル ID を返す以下のストラテジーを提供します。

FindLatestIdStrategy

アーティファクト ID に基づいて最新のスキーマバージョンのグローバル ID を返すストラテジー。

FindBySchemaIdStrategy

アーティファクト ID に基づいてスキーマコンテンツと一致する、グローバル ID を返すストラテジー。

CachedSchemaIdStrategy

スキーマをキャッシュし、キャッシュされたスキーマのグローバル ID を使用するストラテジー。

GetOrCreateIdStrategy

アーティファクト ID に基づいて最新のスキーマを取得しようとし、存在しない場合は新しいスキーマを作成するストラテジー。

AutoRegisterIdStrategy

スキーマを更新し、更新されたスキーマのグローバル ID を使用するストラテジー。

グローバル ID ストラテジーの設定

グローバル ID ストラテジーに以下のアプリケーションプロパティーを設定できます。

- Apicurio **.registry.check-period-ms**: リモートスキーマ検索期間をミリ秒単位で設定します。

アプリケーションプロパティーを Java システムプロパティーとして設定することも、Quarkus **application.properties** ファイルに含めることもできます。詳細は、Quarkus のドキュメント を参照してください。

13.3. SERVICE REGISTRY シリアライザー/デシリアライザー定数

本セクションの定数を使用して、特定のクライアントシリアライザー/デシリアライザー(SerDe)サービスおよびスキーマ検索ストラテジーを直接クライアントに設定できます。

または、プロパティーファイルまたはプロパティーインスタンスで定数を指定することもできます。

シリアライザー/デシリアライザーサービスの定数

```
public abstract class AbstractKafkaSerDe<T extends AbstractKafkaSerDe<T>> implements
AutoCloseable {
  protected final Logger log = LoggerFactory.getLogger(getClass());

  public static final String REGISTRY_URL_CONFIG_PARAM = "apicurio.registry.url"; (1)
  public static final String REGISTRY_CACHED_CONFIG_PARAM = "apicurio.registry.cached"; (2)
  public static final String REGISTRY_ID_HANDLER_CONFIG_PARAM = "apicurio.registry.id-
handler"; (3)
  public static final String REGISTRY_CONFLUENT_ID_HANDLER_CONFIG_PARAM =
"apicurio.registry.as-confluent"; (4)
```

1. (必須) Service Registry の URL。
2. クライアントがリクエストを実行し、以前の結果のキャッシュから情報を検索して処理時間を短縮できるようにします。キャッシュが空の場合、検索は Service Registry から実行されます。
3. ID 処理を拡張することで、他の ID 形式をサポートし、その形式に Service Registry SerDe サービスとの互換性を持たせます。たとえば、ID 形式を **Long** から **Integer** に変更すると Confluent ID 形式がサポートされます。
4. Confluent ID の処理を簡素化するフラグ。**true** に設定すると、**Integer** がグローバル ID の検索に使用されます。

検索ストラテジーの定数

```
public abstract class AbstractKafkaStrategyAwareSerDe<T, S extends
AbstractKafkaStrategyAwareSerDe<T, S>> extends AbstractKafkaSerDe<S> {
  public static final String REGISTRY_ARTIFACT_ID_STRATEGY_CONFIG_PARAM =
```

```
"apicurio.registry.artifact-id"; (1)
  public static final String REGISTRY_GLOBAL_ID_STRATEGY_CONFIG_PARAM =
"apicurio.registry.global-id"; (2)
```

1. artifactId ストラテジー。
2. グローバル ID ストラテジー

コンバーターの定数

```
public class SchemalessConverter<T> extends AbstractKafkaSerDe<SchemalessConverter<T>>
implements Converter {
  public static final String REGISTRY_CONVERTER_SERIALIZER_PARAM =
"apicurio.registry.converter.serializer"; (1)
  public static final String REGISTRY_CONVERTER_DESERIALIZER_PARAM =
"apicurio.registry.converter.deserializer"; (2)
```

1. (必須) コンバーターと使用するシリアライザー。
2. (必須) コンバーターと使用するデシリアライザー。

Avro データプロバイダーの定数

```
public interface AvroDatumProvider<T> {
  String REGISTRY_AVRO_DATUM_PROVIDER_CONFIG_PARAM = "apicurio.registry.avro-datum-
provider"; (1)
  String REGISTRY_USE_SPECIFIC_AVRO_READER_CONFIG_PARAM = "apicurio.registry.use-
specific-avro-reader"; (2)
```

1. スキーマにデータを書き込む Avro データプロバイダー。リフレクションを使用する場合としない場合があります。
2. Avro 固有のデータリーダーの使用を設定するフラグ。

```
DefaultAvroDatumProvider (io.apicurio.registry.utils.serde.avro) (1)
ReflectAvroDatumProvider (io.apicurio.registry.utils.serde.avro) (2)
```

1. デフォルトのデータリーダー。
2. リフレクションを使用するデータリーダー。

13.4. 異なるクライアントのシリアライザー/デシリアライザータイプの使用

Kafka アプリケーションでスキーマ技術を使用する場合は、使用する特定のスキーマタイプを選択する必要があります。一般的なオプションは以下のとおりです。

- Apache Avro
- JSON スキーマ
- Google Protobuf

選択するスキーマ技術は、ユースケースと設定に依存します。当然ながら、Kafka を使用してカスタムシリアライザーおよびデシリアライザークラスを実装することができるので、Service Registry REST Java クライアントを使用した Service Registry 機能の利用など、いつでも独自のクラスを作成することができます。

便宜上、Service Registry は Avro、JSON スキーマ、および Protobuf スキーマテクノロジーに追加設定なしで SerDe クラスを提供します。以下のセクションでは、各タイプを使用するように Kafka アプリケーションを設定する方法を説明します。

シリアライザー/デシリアライザーの Kafka アプリケーション設定
Kafka アプリケーションで Service Registry によって提供されるシリアライザーまたはデシリアライザークラスの 1 つを使用するには、正しい設定プロパティーを設定する必要があります。以下の例は、Kafka プロデューサーアプリケーションでシリアライザーを設定する方法と、Kafka コンシューマーアプリケーションでデシリアライザーを設定する方法を示しています。

Kafka プロデューサーのシリアライザー設定の例

```
public Producer<Object,Object> createKafkaProducer(String kafkaBootstrapServers, String
topicName) {
    Properties props = new Properties();

    // Configure standard Kafka settings
    props.putIfAbsent(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, kafkaBootstrapServers);
    props.putIfAbsent(ProducerConfig.CLIENT_ID_CONFIG, "Producer-" + topicName);
    props.putIfAbsent(ProducerConfig.ACKS_CONFIG, "all");

    // Use a Service Registry-provided Kafka serializer
    props.putIfAbsent(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
        io.apicurio.registry.utils.serde.AvroKafkaSerializer.class.getName());
    props.putIfAbsent(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
        io.apicurio.registry.utils.serde.AvroKafkaSerializer.class.getName());

    // Configure Service Registry location
    props.putIfAbsent(AbstractKafkaSerDe.REGISTRY_URL_CONFIG_PARAM, REGISTRY_URL);

    // Map the topic name (plus -key/value) to the artifactId in the registry

props.putIfAbsent(AbstractKafkaSerializer.REGISTRY_ARTIFACT_ID_STRATEGY_CONFIG_PARAM
,
        io.apicurio.registry.utils.serde.strategy.TopicIdStrategy.class.getName());

    // Get an existing schema or auto-register if not found

props.putIfAbsent(AbstractKafkaSerializer.REGISTRY_GLOBAL_ID_STRATEGY_CONFIG_PARAM,
        io.apicurio.registry.utils.serde.strategy.GetOrCreateIdStrategy.class.getName());

    // Create the Kafka producer
    Producer<Object, Object> producer = new KafkaProducer<>(props);
    return producer;
}
```

Kafka コンシューマーのデシリアライザー設定の例

```
public Consumer<Object,Object> createKafkaConsumer(String kafkaBootstrapServers, String
topicName) {
```

```
    Properties props = new Properties();

    // Configure standard Kafka settings
    props.putIfAbsent(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, kafkaBootstrapServers);
    props.putIfAbsent(ConsumerConfig.GROUP_ID_CONFIG, "Consumer-" + topicName);
    props.putIfAbsent(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG, "true");
    props.putIfAbsent(ConsumerConfig.AUTO_COMMIT_INTERVAL_MS_CONFIG, "1000");
    props.putIfAbsent(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "earliest");

    // Use a Service Registry-provided Kafka deserializer
    props.putIfAbsent(ProducerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
        io.apicurio.registry.utils.serde.AvroKafkaDeserializer.class.getName());
    props.putIfAbsent(ProducerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
        io.apicurio.registry.utils.serde.AvroKafkaDeserializer.class.getName());

    // Configure Service Registry location
    props.putIfAbsent(AbstractKafkaSerDe.REGISTRY_URL_CONFIG_PARAM, REGISTRY_URL);

    // No other configuration needed for deserializer because  globalId of the schema
    // the deserializer uses is sent as part of the message. The deserializer simply
    // extracts that globalId and uses it to look up the schema from the registry.

    // Create the Kafka consumer
    KafkaConsumer<Long, GenericRecord> consumer = new KafkaConsumer<>(props);
    return consumer;
}
```

13.4.1. Service Registry を使用した Avro SerDe の設定

Service Registry は、Avro を可能な限り簡単に使用できるように、Apache Avro 用の Kafka クライアントシリアライザーおよびデシリアライザークラスを提供します。

- **io.apicurio.registry.utils.serde.AvroKafkaSerializer**
- **io.apicurio.registry.utils.serde.AvroKafkaDeserializer**

Avro シリアライザーの設定

Avro シリアライザークラスは以下の方法で設定できます。

- URL としての Service Registry の場所
- アーティファクト ID ストラテジー
- グローバル ID ストラテジー
- グローバル ID の場所
- グローバル ID ハンドラー
- Avro datum プロバイダー
- Avro エンコーディング

グローバル ID の場所

シリアライザーは、スキーマの一意のグローバル ID を Kafka メッセージの一部として渡し、コン

シューマーがデシリアライズに適切なスキーマを使用できるようにします。グローバル ID の場所はメッセージのペイロードまたはメッセージヘッダーになります。デフォルトの方法では、メッセージペイロードでグローバル ID を渡します。代わりに、メッセージヘッダーで送信される ID が必要な場合は、以下の設定プロパティーを設定できます。

```
props.putIfAbsent(AbstractKafkaSerDe.USE_HEADERS, "true")
```

プロパティー name は **apicurio.registry.use.headers** です。

グローバル ID ハンドラー

Kafka メッセージボディーに渡すときにグローバル ID をエンコードする方法を正確にカスタマイズできます。設定プロパティー **apicurio.registry.id-handler** を、**io.apicurio.registry.utils.serde.strategy.IdHandler** インターフェースを実装するクラスに設定します。Service Registry は、そのインターフェースの実装を 2 つ提供します。

- **io.apicurio.registry.utils.serde.strategy.DefaultIdHandler** - ID を 8 バイト長として格納します。
- **io.apicurio.registry.utils.serde.strategy.Legacy4ByteIdHandler** - ID を 4 バイト int として保存します。

Service Registry は、アーティファクトのグローバル ID を long として表しますが、従来の理由（または他のレジストリーとの互換性、または他のレジストリーとの互換性、または ID を送信する場合）では、4 バイトを使用したい場合があります。

Avro datum プロバイダー

Avro は、データを読み書きするためのさまざまなデータライターとリーダーを提供します。Service Registry は、3 つの異なるタイプをサポートします。

- Generic
- Specific
- Reflect

Service Registry **AvroDatumProvider** は、実際に使用するタイプを抽象化したものです。デフォルトでは **DefaultAvroDatumProvider** が使用されます。

設定可能な設定オプションは 2 つあります。

- Apicurio. **registry.avro-datum-provider**: **AvroDatumProvider** 実装の完全修飾 Java クラス名を指定します（例： **io.apicurio.registry.utils.serde.avro.ReflectAvroDatumProvider**）。
- Apicurio **.registry.use-specific-avro-reader -** true または false。**DefaultAvroDatumProvider**を使用する際に特定のタイプを使用します。

Avro エンコーディング

Apache Avro を使用してデータをシリアライザーする場合、Avro バイナリーエンコーディング形式を使用するのが一般的です。これにより、データは可能な限り効率的な形式でエンコードされます。ただし、Avro は JSON としてデータのエンコードもサポートします。JSON のエンコーディングは、各メッセージのペイロードを検査することが非常に容易であるため、多くの場合、ロギング、デバッグなどのユースケースに使用されます。Service Registry Avro シリアライザーは、エンコーディングをデフォルト（バイナリー）から JSON に変更するように設定することができます。

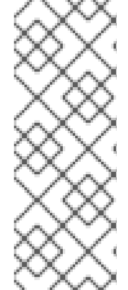
apicurio.avro.encoding プロパティーを設定して、使用する Avro エンコーディングを設定します。値は **JSON** または **BINARY** のいずれかである必要があります。

Avro デシリアライザーの設定

シリアライザーの設定と一致するように、Avro デシリアライザークラスを設定する必要があります。これにより、以下の方法で Avro デシリアライザークラスを設定することができます。

- URL としての Service Registry の場所
- グローバル ID ハンドラー
- Avro datum プロバイダー
- Avro エンコーディング

これらの設定オプションについては、シリアライザーセクションを参照してください。プロパティー名と値は同じです。



注記

デシリアライザーの設定時には、以下のオプションは必要ありません。

- アーティファクト ID ストラテジー
- グローバル ID ストラテジー
- グローバル ID の場所

これらのオプションは必要ない理由は、デシリアライザークラスはこの情報をメッセージ自体から把握できることです。2 つのストラテジーの場合、シリアライザーはメッセージの一部としてスキーマのグローバル ID を送信するため、それらは必要ありません。

グローバル ID の場所は、メッセージペイロードの開始時にマジックバイトを確認するだけで、デシリアライザーによって決定されます。そのバイトが見つかった場合、グローバル ID は設定済みのハンドラーを使用してメッセージペイロードから読み込まれます。マジックバイトが見つからない場合、グローバル ID はメッセージヘッダーから読み取られます。

13.4.2. Service Registry を使用した JSON スキーマ SerDe の設定

Service Registry は、JSON スキーマを可能な限り簡単に使用できるように、JSON スキーマの Kafka クライアントシリアライザーおよびデシリアライザークラスを提供します。

- **io.apicurio.registry.utils.serde.JsonSchemaKafkaSerializer**
- **io.apicurio.registry.utils.serde.JsonSchemaKafkaDeserializer**

Apache Avro とは異なり、JSON スキーマは実際にはシリアライズテクノロジーではありません。検証テクノロジーです。その結果、JSON スキーマの設定オプションは大きく異なります。たとえば、データは常に JSON としてエンコードされるため、エンコーディングオプションはありません。

JSON スキーマシリアライザーの設定

JSON スキーマシリアライザークラスを以下の方法で設定できます。

- URL としての Service Registry の場所

- アーティファクト ID ストラテジー
- グローバル ID ストラテジー
- 検証 enabled/無効

標準以外の設定プロパティーは、JSON スキーマ検証が有効または無効であるかです。検証機能はデフォルトで無効になっていますが、**apicurio.registry.serdes.json-schema.validation-enabled** を **"true"** に設定して有効にできます。以下に例を示します。

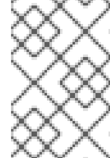
```
props.putIfAbsent(JsonSchemaSerDeConstants.REGISTRY_JSON_SCHEMA_VALIDATION_ENABLED, "true")`
```

JSON スキーマデシリアライザーの設定

JSON スキーマデシリアライザークラスを以下の方法で設定できます。

- URL としての Service Registry の場所
- 検証 enabled/無効

デシリアライザーは、設定も簡単です。スキーマをロードできるように、Service Registry の場所を指定する必要があります。他の設定は、検証を実行するかどうかだけです。これらの設定プロパティーはシリアライザーの場合と同じです。



注記

デシリアライザーの検証は、シリアライザーが Kafka メッセージでグローバル ID を渡す場合にのみ機能します。これは、シリアライザーで検証が有効になっている場合にのみ発生します。

13.4.3. Service Registry を使用した Protobuf SerDe の設定

Service Registry は、Google Protobuf の Kafka クライアントシリアライザーおよびデシリアライザークラスを提供し、Protobuf を可能な限り簡単に使用できるようにします。

- **io.apicurio.registry.utils.serde.ProtobufKafkaSerializer**
- **io.apicurio.registry.utils.serde.ProtobufKafkaDeserializer**

Protobuf シリアライザーの設定

Protobuf シリアライザークラスを以下の方法で設定できます。

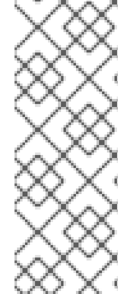
- URL としての Service Registry の場所
- アーティファクト ID ストラテジー
- グローバル ID ストラテジー
- グローバル ID の場所
- グローバル ID ハンドラー

Protobuf デシリアライザーの設定

シリアライザーの設定と一致するように、Protobuf デシリアライザークラスを設定する必要があります。これにより、Protobuf デシリアライザークラスを以下の方法で設定できます。

- URL としての Service Registry の場所
- グローバル ID ハンドラー

これらの設定オプションについては、シリアライザーセクションを参照してください。プロパティー名と値は同じです。



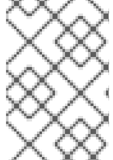
注記

デシリアライザーの設定時には、以下のオプションは必要ありません。

- アーティファクト ID ストラテジー
- グローバル ID ストラテジー
- グローバル ID の場所

これらのオプションは必要ない理由は、デシリアライザークラスはこの情報をメッセージ自体から把握できることです。2 つのストラテジーの場合、シリアライザーはメッセージの一部としてスキーマのグローバル ID を送信するため、それらは必要ありません。

グローバル ID の場所は、メッセージペイロードの開始時にマジックバイトを確認するだけで、（デシリアライザーによって）決定されます。そのバイトが見つかった場合、グローバル ID はメッセージペイロードから読み取られます（設定されたハンドラーを使用）。マジックバイトが見つからない場合、グローバル ID はメッセージヘッダーから読み取られます。



注記

Protobuf デシリアライザーは、正確な Protobuf Message 実装へデシリアライズしませんが、**DynamicMessage** インスタンスにはデシリアライズされません（そうでない場合には適切な API がないため）。

13.5. スキーマの SERVICE REGISTRY への登録

スキーマを Apache Avro などの適切な形式で定義したら、スキーマを Service Registry に追加できます。

以下を使用してスキーマを追加できます。

- Service Registry Web コンソール
- Service Registry API を使用する curl コマンド
- Service Registry に付属する Maven プラグイン
- クライアントコードに加えられたスキーマ設定

スキーマを登録するまでは、クライアントアプリケーションは Service Registry を使用できません。

Service Registry Web コンソール

Service Registry をインストールしたら、**ui** エンドポイントから Web コンソールに接続します。

http://MY-REGISTRY-URL/ui

コンソールから、スキーマを追加、表示、および設定できます。また、レジストリーに無効なコンテンツが追加されないようにするルールを作成することもできます。

curl コマンドの例

```
curl -X POST -H "Content-type: application/json; artifactType=AVRO" \
  -H "X-Registry-ArtifactId: prices-value" \
  --data '{ (1)
      "type":"record",
      "name":"price",
      "namespace":"com.redhat",
      "fields":[{"name":"symbol","type":"string"},
      {"name":"price","type":"string"}]
    }'
  https://my-cluster-service-registry-myproject.example.com/api/artifacts -s (2)
```

1. Avro スキーマアーティファクト。
2. Service Registry を公開する OpenShift ルート名。

Maven プラグインの例

```
<plugin>
<groupId>io.apicurio</groupId>
<artifactId>apicurio-registry-maven-plugin</artifactId>
<version>${registry.version}</version>
<executions>
  <execution>
    <phase>generate-sources</phase>
    <goals>
      <goal>register</goal>
    </goals>
    <configuration>
      <registryUrl>https://my-cluster-service-registry-myproject.example.com/api</registryUrl>
      <artifactType>AVRO</artifactType>
      <artifacts>
        <schema1>${project.basedir}/schemas/schema1.avsc</schema1>
      </artifacts>
    </configuration>
  </execution>
</executions>
</plugin>
```

プロデューサークライアントを使用した設定例

```
String registryUrl_node1 = PropertiesUtil.property(clientProperties, "registry.url.node1", (1)
    "https://my-cluster-service-registry-myproject.example.com/api");
try (RegistryService service = RegistryClient.create(registryUrl_node1)) {
    String artifactId = ApplicationImpl.INPUT_TOPIC + "-value";
    try {
        service.getArtifactMetaData(artifactId); (2)
    } catch (WebApplicationException e) {
        CompletionStage <ArtifactMetaData> csa = service.createArtifact(
            ArtifactType.AVRO,
```

```
            artifactId,
            new ByteArrayInputStream(LogInput.SCHEMA$.toString().getBytes())
        );
        csa.toCompletableFuture().get();
    }
}
```

1. プロパティーが登録されています。複数のノードに対してプロパティーを登録できます。
2. アーティファクト ID に基づいてスキーマがすでに存在しているかを確認します。

13.6. KAFKA コンシューマークライアントからのスキーマの使用

この手順では、Service Registry からのスキーマを使用するように Java で書かれた Kafka コンシューマークライアントを設定する方法について説明します。

前提条件

- Service Registry がインストールされている必要があります。
- スキーマが Service Registry に登録されている必要があります。

手順

1. Service Registry の URL でクライアントを設定します。以下に例を示します。

```
String registryUrl = "https://registry.example.com/api";
Properties props = new Properties();
props.putIfAbsent(AbstractKafkaSerDe.REGISTRY_URL_CONFIG_PARAM, registryUrl);
```

2. Service Registry デシリアライザーでクライアントを設定します。以下に例を示します。

```
// Configure Kafka
props.putIfAbsent(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, SERVERS);
props.putIfAbsent(ConsumerConfig.GROUP_ID_CONFIG, "Consumer-" + TOPIC_NAME);
props.putIfAbsent(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG, "true");
props.putIfAbsent(ConsumerConfig.AUTO_COMMIT_INTERVAL_MS_CONFIG, "1000");
props.putIfAbsent(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "earliest");
props.putIfAbsent(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
    AvroKafkaDeserializer.class.getName()); (1)
props.putIfAbsent(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
    AvroKafkaDeserializer.class.getName()); (2)
```

1. Service Registry によって提供されるデシリアライザー。
2. デシリアライズは Apache Avro JSON 形式です。

13.7. KAFKA プロデューサークライアントからのスキーマの使用

この手順では、Service Registry からのスキーマを使用するように Java で書かれた Kafka プロデューサークライアントを設定する方法について説明します。

前提条件

- Service Registry がインストールされている必要があります。
- スキーマが Service Registry に登録されている必要があります。

手順

1. Service Registry の URL でクライアントを設定します。以下に例を示します。

```
String registryUrl = "https://registry.example.com/api";
Properties props = new Properties();
props.putIfAbsent(AbstractKafkaSerDe.REGISTRY_URL_CONFIG_PARAM, registryUrl);
```

2. クライアントをシリアライザーで設定し、Service Registry でスキーマを検索するようにストラテジーを設定します。以下に例を示します。

```
props.put(CommonClientConfigs.BOOTSTRAP_SERVERS_CONFIG, "my-cluster-kafka-bootstrap:9092");
props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
AvroKafkaSerializer.class.getName()); (1)
props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
AvroKafkaSerializer.class.getName()); (2)
props.put(AbstractKafkaSerializer.REGISTRY_GLOBAL_ID_STRATEGY_CONFIG_PARAM,
FindLatestIdStrategy.class.getName()); (3)
```

(1) Service Registry により提供されるメッセージキーのシリアライザー。

(2) Service Registry により提供されるメッセージ値のシリアライザー。

(3) スキーマのグローバル ID を検索する検索ストラテジー。

13.8. KAFKA STREAMS アプリケーションからのスキーマの使用

この手順では、Service Registry からのスキーマを使用するように Java で書かれた Kafka Streams クライアントを設定する方法を説明します。

前提条件

- Service Registry がインストールされている必要があります。
- スキーマが Service Registry に登録されている必要があります。

手順

1. Service Registry で REST クライアントを作成および設定します。以下に例を示します。

```
String registryUrl = "https://registry.example.com/api";
RegistryService client = RegistryClient.cached(registryUrl);
```

2. シリアライザー、デシリアライザーの設定、および Kafka Streams クライアントを作成します。以下に例を示します。

```
Serializer<LogInput> serializer = new AvroKafkaSerializer<>( (1)
    client,
    new DefaultAvroDatumProvider<LogInput>().setUseSpecificAvroReader(true)
);
Deserializer<LogInput> deserializer = new AvroKafkaDeserializer <> ( (2)
    client,
    new DefaultAvroDatumProvider<LogInput>().setUseSpecificAvroReader(true)
);
Serde<LogInput> logSerde = Serdes.serdeFrom( (3)
    serializer,
    deserializer
);
KStream<String, LogInput> input = builder.stream( (4)
    INPUT_TOPIC,
    Consumed.with(Serdes.String(), logSerde)
);
```

(1) Service Registry によって提供されるシリアライザー。

(2) Service Registry によって提供されるデシリアライザー。

(3) デシリアライズは Apache Avro 形式です。

(4) Kafka Streams クライアントアプリケーション。

第14章 SERVICE REGISTRY アーティファクトの参照

本章では、Service Registry に保存されるサポート対象のアーティファクトタイプ、状態、メタデータ、およびコンテンツルールの詳細を説明します。

- 「Service Registry アーティファクトタイプ」
- 「Service Registry アーティファクトの状態」
- 「Service Registry アーティファクトのメタデータ」
- 「Service Registry コンテンツルールタイプ」
- 「Service Registry コンテンツルールの成熟度」

その他のリソース

- 詳細は、Apicurio Registry REST API のドキュメント を参照してください。

14.1. SERVICE REGISTRY アーティファクトタイプ

以下のアーティファクトタイプは Service Registry に保存および管理できます。

表14.1 Service Registry アーティファクトタイプ

型	説明
ASYNCAPI	AsyncAPI 仕様
AVRO	Apache Avro スキーマ
GRAPHQL	GraphQL スキーマ
JSON	JSON スキーマ
KCONNECT	Apache Kafka Connect スキーマ
OPENAPI	OpenAPI 仕様
PROTOBUF	Google プロトコルバッファースキーマ
PROTOBUF_FD	Google プロトコルバッファーファイル記述子
WSDL	Web Services Definition Language
XSD	XML Schema Definition

14.2. SERVICE REGISTRY アーティファクトの状態

以下は、Service Registry の有効なアーティファクト状態です。

表14.2 Service Registry アーティファクトの状態

状態	説明
ENABLED	基本状態、全ての操作が可能です。
DISABLED	アーティファクトとそのメタデータは、Service Registry Web コンソールを使用して表示および検索できますが、そのコンテンツはどのクライアントでもフェッチできません。
非推奨	アーティファクトは完全に使用可能ですが、アーティファクトのコンテンツがフェッチされるたびに、ヘッダーが REST API 応答に追加されます。Service Registry Rest Client は、非推奨となったコンテンツが見つかったときにも警告をログに記録します。

14.3. SERVICE REGISTRY アーティファクトのメタデータ

アーティファクトが Service Registry に追加されると、メタデータプロパティーのセットがアーティファクトの内容と共に保存されます。このメタデータは、設定可能な一部のプロパティーと、生成された読み取り専用プロパティーのセットで構成されます。

表14.3 Service Registry メタデータプロパティー

プロパティー	型	編集可能
id	string	false
type	ArtifactType	false
state	ArtifactState	true
version	integer	false
createdBy	string	false
createdOn	date	false
modifiedBy	string	false
modifiedOn	date	false
name	string	true
description	string	true
labels	文字列の配列	true

プロパティー	型	編集可能
properties	map	true

アーティファクトメタデータの更新

- Service Registry REST API を使用して、メタデータエンドポイントを使用して編集可能なプロパティーのセットを更新できます。
- **state** プロパティーは、状態遷移 API を使用することでのみ編集できます。たとえば、アーティファクトを **deprecated** または **disabled** としてマークできます。

関連情報

詳細は、Apicurio Registry REST API documentation の **/artifacts/{artifactId}/meta** セクションを参照してください。

14.4. SERVICE REGISTRY コンテンツルールタイプ

Service Registry のコンテンツ展開を管理するには、以下のルールタイプを指定できます。

表14.4 Service Registry コンテンツルールタイプ

型	説明
VALIDITY	レジストリーに追加する前にデータを検証します。このルールに使用できる設定値は以下のとおりです。 ● **FULL**: 検証はシンタックスとセマンティックの両方で行われます。 ● **SYNTAX_ONLY**: 検証は構文のみです。

型	説明
COMPATIBILITY	新たに追加されたアーティファクトが以前に追加したバージョンと互換性があることを確認します。このルールに使用できる設定値は以下のとおりです。 ● **FULL**: 新しいアーティファクトは、最後に追加されたアーティファクトと上位および下位互換性があります。 ● **FULL_TRANSITIVE**: 新しいアーティファクトは、以前に追加されたすべてのアーティファクトと上位互換性および下位互換性があります。 ● **BACKWARD**: 新しいアーティファクトを使用するクライアントは、最後に追加されたアーティファクトを使用して書き込まれたデータを読み取りできます。 ● **BACKWARD_TRANSITIVE**: 新しいアーティファクトを使用しているクライアントは、以前に追加されたすべてのアーティファクトを使用して書き込まれたデータを読み取ることができます。 ● **FORWARD**: 最後に追加されたアーティファクトを使用するクライアントは、新しいアーティファクトを使用して書き込まれたデータを読み取りできます。 ● **FORWARD_TRANSITIVE**: 以前に追加されたすべてのアーティファクトを使用しているクライアントは、新しいアーティファクトを使用して書き込まれたデータを読み取ることができます。 ● **NONE**: 下位互換性および上位互換性チェックはすべて無効になります。

14.5. SERVICE REGISTRY コンテンツルールの成熟度

すべてのコンテンツルールは、Service Registry でサポートされるすべてのアーティファクトタイプに対して完全に実装されるわけではありません。以下の表は、各ルールおよびアーティファクトタイプの現在の成熟度レベルを示しています。

表14.5 Service Registry コンテンツルールの成熟度マトリックス

アーティファクトタイプ	検証ルール	互換性ルール
Avro	Full	Full
Protobuf	Full	なし

アーティファクトタイプ	検証ルール	互換性ルール
JSON スキーマ	Full	Full
OpenAPI	Full	なし
AsyncAPI	Syntax Only	なし
GraphQL	Syntax Only	なし
Kafka Connect	Syntax Only	なし
WSDL	Syntax Only	なし
XSD	Syntax Only	None

第15章 SERVICE REGISTRY OPERATOR の設定リファレンス

本章では、Service Registry Operator をデプロイするように Service Registry を設定するために使用されるカスタムリソースの詳細情報を提供します。

- 「Service Registry カスタムリソース」
- 「Service Registry CR 仕様」
- 「Service Registry CR のステータス」
- 「Service Registry Operator ラベル」
- 「Service Registry 管理リソース」

15.1. SERVICE REGISTRY カスタムリソース

Service Registry Operator は、OpenShift 上の Service Registry の単一デプロイメントを表す **ApicurioRegistry** custom resource (CR) を定義します。

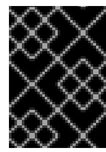
これらのリソースオブジェクトはユーザーによって作成および維持され、Service Registry のデプロイおよび設定方法を Service Registry Operator に指示します。

ApicurioRegistry CR の例

次のコマンドは、**ApicurioRegistry** リソースを表示します。

```
oc edit apicurioregistry example-apicurioregistry
```

```
apiVersion: apicur.io/v1alpha1
kind: ApicurioRegistry
metadata:
  name: example-apicurioregistry
  namespace: demo-streams
  # ...
spec:
  configuration:
    persistence: streams
    streams:
      bootstrapServers: 'my-cluster-kafka-bootstrap.demo-streams.svc:9092'
  deployment:
    host: >-
      example-apicurioregistry.demo-streams.example.com
status:
  deploymentName: example-apicurioregistry-deployment-qsdb7
  host: >-
    example-apicurioregistry.demo-streams.example.com
  image: >-
    registry.redhat.io/integration/service-registry-streams-
rhel8@sha256:4b56da802333d2115cb3a0acc8d97445bd0dab67b639c361816df27b7f1aa296
  ingressName: example-apicurioregistry-ingress-7mlnw
  replicaCount: 1
  serviceName: example-apicurioregistry-service-xvnmz
```



重要

現時点で、Service Registry Operator は独自のプロジェクト namespace のみを監視します。したがって、**ApicurioRegistry** CR を同じ namespace に作成する必要があります。

関連情報

- Extending the Kubernetes API with Custom Resource Definitions

15.2. SERVICE REGISTRY CR 仕様

spec は、オペレーターがアーカイブするための望ましい状態または設定を提供するために使用される **ApicurioRegistry** CR の一部です。

ApicurioRegistry CR 仕様コンテンツ

以下のブロック例には、可能な **spec** 設定オプションの完全なツリーが含まれます。フィールドによっては、必須ではないものや、同時に定義してはいけないものもあります。

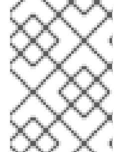
```
spec:
  configuration:
    persistence: <string>
    dataSource:
      url: <string>
      userName: <string>
      password: <string>
    kafka:
      bootstrapServers: <string>
    streams:
      bootstrapServers: <string>
      applicationId: <string>
      applicationServerPort: <string>
      security:
        tls:
          truststoreSecretName: <string>
          keystoreSecretName: <string>
        scram:
          mechanism: <string>
          truststoreSecretName: <string>
          user: <string>
          passwordSecretName: <string>
    infinispan:
      clusterName: <string>
    ui:
      readOnly: <string>
    logLevel: <string>
  deployment:
    replicas: <int32>
    host: <string>
```

以下の表は、各設定オプションについて説明しています。

表15.1 ApicurioRegistry CR 仕様設定オプション

設定オプション	型	デフォルト値	説明
設定	-	-	Service Registry アプリケーションの設定セクション
configuration/persistence	string	**mem**	ストレージバックエンド。**jpa**、**streams**、**infinispan**のいずれか
configuration/dataSource	-	-	JPA ストレージバックエンドのデータベース接続設定
configuration/dataSource/url	string	**必須**	データベース接続 URL 文字列
configuration/dataSource/userName	string	**必須**	データベースコネクションユーザー
configuration/dataSource/password	string	**空**	データベース接続パスワード
configuration/streams	-	-	Kafka Streams ストレージバックエンドの設定
configuration/streams/bootstrapServers	string	**必須**	Streams ストレージバックエンドの Kafka ブートストラップサーバー URL。
configuration/streams/applicationId	string	**ApicurioRegistry CR 名**	Kafka Streams アプリケーション ID
configuration/streams/applicationServerPort	string	**9000**	-
configuration/streams/security/tls	-	-	Kafka Streams ストレージバックエンドの TLS 認証を設定するセクション。
configuration/streams/security/tls/truststoreSecretName	string	**必須**	Kafka の TLS トラストストアが含まれるシークレットの名前
configuration/streams/security/tls/keystoreSecretName	string	**必須**	ユーザー TLS キーストアを含むシークレットの名前
configuration/streams/security/scram/truststoreSecretName	string	**必須**	Kafka の TLS トラストストアが含まれるシークレットの名前

設定オプション	型	デフォルト値	説明
configuration/streams/security/scram/user	string	**必須**	SCRAM ユーザー名
configuration/streams/security/scram/passwordSecretName	string	**必須**	SCRAM ユーザーパスワードが含まれるシークレットの名前
configuration/streams/security/scram/mechanism	string	**SCRAM-SHA-512**	SASL メカニズム
configuration/infinispan	-	-	Infinispan 永続性設定セクション
configuration/infinispan/clusterName	string	**ApicurioRegistry CR 名**	Infinispan クラスター名
configuration/ui	-	-	Service Registry Web コンソール設定
configuration/ui/readOnly	string	**false**	Service Registry Web コンソールを読み取り専用モードに設定します。
configuration/logLevel	string	**INFO**	Service Registry オペランドログレベル。**INFO** の 1 つ、**DEBUG**
deployment	-	-	オペランドデプロイメント設定のセクション
deployment/replicas	正の整数	**1**	デプロイする Service Registry Pod 数
deployment/host	string	**ApicurioRegistry CR 名および namespace から自動生成**	Service Registry コンソールおよび API が利用できるホスト/URL



注記

オプションが **必須** とされている場合は、有効になっている他の設定オプションの条件である可能性があります。空の値は受け入れられる可能性がありますが、Operator は指定されたアクションを実行しません。

15.3. SERVICE REGISTRY CR のステータス

status は、Service Registry Operator によって管理される CR のセクションであり、現在のデプロイメントとアプリケーションの状態の説明が含まれています。

ApicurioRegistry CR ステータスのコンテンツ

status セクションには、次のフィールドが含まれています。

```
status:
  image: <string>
  deploymentName: <string>
  serviceName: <string>
  ingressName: <string>
  replicaCount: <int32>
  host: <string>
```

表15.2 ApicurioRegistry CR status フィールド

status フィールド	型	Description
image	string	Operator がデプロイする Service Registry オペランドイメージ。設定で選択したストレージオプションに基づいて変更される可能性があります。
deploymentName	string	Service Registry をデプロイするために使用される Operator によって管理される **Deployment** または **DeploymentConfig** の名前。
serviceName	string	**Service** Registry オペランドをサービスとして公開する、Operator によって管理される Service の名前。
ingressName	string	Service Registry を HTTP 経由でアクセスできるようにするために Operator によって管理される **Ingress** の名前。**Route** は OCP にも作成されます。
replicaCount	int32	デプロイされる Service Registry オペランド Pod の数。
host	string	Service Registry UI および REST API にアクセスできる URL。

15.4. SERVICE REGISTRY 管理リソース

Service Registry のデプロイ時に Service Registry Operator によって管理されるリソースは以下のとおりです。

- **DeploymentConfig**
- **サービス**
- **Ingress**
- **ルート**
- **PodDisruptionBudget**

15.5. SERVICE REGISTRY OPERATOR ラベル

通常 Service Registry Operator によって管理されるリソースには、以下のようにラベルが付けられます。

表15.3 管理されたリソースの Service Registry Operator ラベル

ラベル	説明
app	指定された **ApicurioRegistry** CR の名前に基づく、リソースが属する Service Registry デプロイメントの名前。
apicur.io/type	デプロイメントのタイプ: **apicurio-registry** または **operator**
apicur.io/name	デプロイの名前: **app** または **apicurio-registry-operator** と同じ値
apicur.io/version	Service Registry または Service Registry Operator のバージョン
app.kubernetes.io/*	アプリケーションのデプロイメントに推奨される Kubernetes ラベルのセット。
com.company および **rht.*`**	Red Hat 製品のメータリングラベル。

その他のリソース

- アプリケーションデプロイメントに推奨される Kubernetes ラベル

付録A サブスクリプションの使用

Service Registry は、ソフトウェアサブスクリプションから提供されます。サブスクリプションを管理するには、Red Hat カスタマーポータルでアカウントにアクセスします。

アカウントへのアクセス

1. access.redhat.com に移動します。
2. アカウントがない場合は、作成します。
3. アカウントにログインします。

サブスクリプションのアクティベート

1. access.redhat.com に移動します。
2. **サブスクリプション** に移動します。
3. **Activate a subscription** に移動し、16 桁のアクティベーション番号を入力します。

ZIP および TAR ファイルのダウンロード

ZIP または TAR ファイルにアクセスするには、カスタマーポータルを使用して、ダウンロードする関連ファイルを検索します。RPM パッケージを使用している場合は、この手順は必要ありません。

1. ブラウザーを開き、access.redhat.com/downloads で Red Hat カスタマーポータルの **Product Downloads** ページにログインします。
2. **Integration and Automation** カテゴリーで **Red Hat Integration** エントリーを見つけます。
3. 必要な Service Registry 製品を選択します。**Software Downloads** ページが開きます。
4. コンポーネントの **Download** リンクをクリックします。

パッケージ用のシステムの登録

Red Hat Enterprise Linux に RPM パッケージをインストールするには、システムを登録する必要があります。ZIP または TAR ファイルを使用している場合、この手順は必要ありません。

1. access.redhat.com に移動します。
2. **Registration Assistant** に移動します。
3. ご使用の OS バージョンを選択し、次のページに進みます。
4. システムターミナルで listed コマンドを使用して、登録を完了します。

詳細は 「How to Register and Subscribe a System to the Red Hat Customer Portal 」を参照してください。