Red Hat Integration 2023.Q4

Service Registry 用户指南

管理 Service Registry 2.5 中的模式和 API

Last Updated: 2024-03-29

Red Hat Integration 2023.Q4 Service Registry 用户指南

管理 Service Registry 2.5 中的模式和 API

法律通告

摘要

本指南介绍了 Service Registry，并解释了如何使用 Service Registry Web 控制台、REST API、Maven 插件或 Java 客户端来管理事件模式和 API 设计。本指南还解释了如何在 Java 使用者和制作者应用程序中使用 Kafka 客户端序列化器和反序列化器。它还描述了支持的 Service Registry 内容类型和可选规则配置。

目录

前言

使开源包含更多

红帽致力于替换我们的代码、文档和 Web 属性中存在问题的语言。我们从这四个术语开始：master、slave、黑名单和白名单。由于此项工作十分艰巨，这些更改将在即将推出的几个发行版本中逐步实施。详情请查看 CTO Chris Wright 的信息。

对红帽文档提供反馈

我们感谢您对我们文档的反馈。

要改进，创建一个 JIRA 问题并描述您推荐的更改。提供尽可能多的详细信息，以便我们快速解决您的请求。

前提条件

- 您有一个红帽客户门户网站帐户。此帐户可让您登录到 Red Hat Jira Software 实例。
  如果您没有帐户，系统会提示您创建一个帐户。

流程

1. 单击以下链接： 创建问题。
2. 在 **Summary** 文本框中输入问题的简短描述。
3. 在 **Description** 文本框中提供以下信息：
   - 找到此问题的页面的 URL。
   - 有关此问题的详细描述。
     您可以将信息保留在任何其他字段中的默认值。
4. 点 **Create** 将 JIRA 问题提交到文档团队。

感谢您花时间来提供反馈。

第 1 章 SERVICE REGISTRY 简介

本章介绍了 Service Registry 概念和功能，并提供了有关存储在 registry 中支持的工件类型的详情：

1.1. 什么是 SERVICE REGISTRY？

Service Registry 是一个数据存储，用于在事件驱动的和 API 架构间共享标准事件模式和 API 设计。您可以使用 Service Registry 将数据的结构与客户端应用程序分离，并使用 REST 接口在运行时共享和管理您的数据类型和 API 描述。

客户端应用程序可以在运行时动态地从 Service Registry 推送或拉取最新的 schema 更新，而无需重新部署。开发人员团队可以查询 Service Registry 以获取生产环境中已部署服务所需的现有模式，并可注册开发中新服务所需的新模式。

您可以通过在客户端应用程序代码中指定 Service Registry URL，使客户端应用程序能够使用存储在 Service Registry 中的模式和 API 设计。Service Registry 可以存储用于序列化和反序列化消息的模式，这些消息从客户端应用程序引用，以确保它们发送和接收的消息与这些模式兼容。

使用 Service Registry 将数据结构与应用程序分离，通过降低整体消息大小来降低成本，并通过增加企业中的模式和 API 设计的一致性重复使用来提高效率。Service Registry 提供了一个 Web 控制台，方便开发人员和管理员管理 registry 内容。

您可以配置可选规则，以管理 Service Registry 内容的演进。这包括用于确保上传内容有效或者与其他版本兼容的规则。在将新版本上传到 Service Registry 之前，任何配置的规则都必须通过，这样可确保时间不会在无效或不兼容的模式或 API 设计中造成。

Service Registry 基于 Apicurio Registry 开源社区项目。详情请查看 https://github.com/apicurio/apicurio-registry。

Service Registry 功能

- 标准事件模式和 API 规格的多个有效负载格式，如 Apache Avro、JSON 架构、Google Protobuf、AsyncAPI、OpenAPI 和 OpenAPI 等。
- AMQ Streams 或 PostgreSQL 数据库中的可插拔 Service Registry 存储选项。
- 用于内容验证、兼容性和完整性的规则，以管理 Service Registry 内容随时间变化的方式。

- 使用 Web 控制台、REST API、命令行、Maven 插件或 Java 客户端进行 Service Registry 内容管理。
- 完整的 Apache Kafka 模式 registry 支持，包括与外部系统的 Kafka Connect 集成。
- Kafka 客户端序列化器/反序列化器(SerDes)在运行时验证消息类型。
- 与现有 Confluent 模式 registry 客户端应用程序兼容。
- Cloud-native Quarkus Java 运行时，用于低内存空间和快速部署时间。
- 在 OpenShift 上安装基于 Operator 的 Service Registry。
- 使用红帽单点登录的 OpenID Connect (OIDC)身份验证。

1.2. SERVICE REGISTRY 中的 SCHEMA 和 API 工件

存储在 Service Registry 中的项目（如事件模式和 API 设计）被称为 registry *工件*。下面显示了一个简单共享价格应用程序的 JSON 格式的 Apache Avro 模式工件示例：

Avro 模式示例

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

当将模式或 API 设计添加为 Service Registry 中的工件时，客户端应用程序可以使用该模式或 API 设计来验证客户端消息在运行时是否符合正确的数据结构。

模式和 API 组

工件组是一个可选的 命名集合，即 schema 或 API 工件的集合。每个组都包含一组逻辑相关的模式或 API 设计，通常由单个实体管理，属于一个特定的应用程序或机构。

您可以在添加模式和 API 设计时创建可选的工件组，以便在 Service Registry 中组织它们。例如，您可以创建组来匹配 **development** 和 **production** 的应用程序环境，或 **sales** 和 **engineering** 机构。

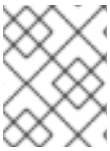
模式和 API 组可以包含多个工件类型。例如，您可以在同一个组中都有 Protobuf, Avro, JSON Schema, OpenAPI, 或 AsyncAPI 工件。

您可以使用 Service Registry web 控制台、REST API、命令行、Maven 插件或 Java 客户端应用程序创建模式和 API 工件和组。以下简单示例演示了使用 Core Registry REST API：

```
$ 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"}]}' \
https://my-registry.example.com/apis/registry/v2/groups/my-group/artifacts
```

本例创建一个名为 **my-group** 的工件组，并添加带有 **share-price** 的工件 ID 的 Avro 模式。



注意

在使用 Service Registry web 控制台时指定组是可选的，并自动创建一个 **default** 组。在使用 REST API 或 Maven 插件时，如果您不想创建唯一组，请在 API 路径中指定 **默认组**。

其他资源

- 有关支持的工件类型的信息，请参考 第 9 章 *Service Registry 工件参考*。
- 有关 Core Registry API 的详情，请查看 Apicurio Registry REST API 文档 。

对其他模式和 API 的引用

有些 Service Registry 工件类型可能会包括从一个 *工件文件到另一个工件引用*。您可以通过定义可重复使用的模式或 API 组件来创建效率，然后从多个位置引用它们。例如，您可以使用 **$ref** 语句在 JSON Schema 或 OpenAPI 中指定引用，或使用 **import** 语句在 Google Protobuf 中指定，或使用嵌套命名空间在 Apache Avro 中指定。

以下示例显示了一个名为 **TradeKey** 的简单 Avro 模式，其中包含使用嵌套命名空间对名为 **Exchange** 的另一个模式的引用：

带有嵌套交换模式的 Tradekey 模式

```
{
 "namespace": "com.kubetrade.schema.trade",
 "type": "record",
 "name": "TradeKey",
 "fields": [
  {
   "name": "exchange",
   "type": "com.kubetrade.schema.common.Exchange"
  },
  {
   "name": "key",
   "type": "string"
  }
 ]
}
```

Exchange 模式

```
{
 "namespace": "com.kubetrade.schema.common",
 "type": "enum",
 "name": "Exchange",
 "symbols" : ["GEMINI"]
}
```

工件引用存储在 Service Registry 中，作为从工件类型特定引用到内部 Service Registry 引用的工件元数据集合。Service Registry 中的每个工件引用都由以下内容组成：

- 组 ID
- 工件 ID
- 工件版本
- 工件引用名称

您可以使用 Service Registry 核心 REST API、Maven 插件和 Java serializers/反序列化器(SerDes)来管理工件引用。Service Registry 存储工件引用以及工件内容。Service Registry 还维护所有工件引用的集合，以便您可以搜索或列出特定工件的所有引用。

支持的工件类型
Service Registry 目前仅支持对以下工件类型的工件引用：

- Avro
- protobuf
- JSON 架构
- OpenAPI
- AsyncAPI

其他资源

- 有关管理工件引用的详情，请参考：
  - 第 4 章 *使用 REST API 管理 Service Registry 内容*.
  - 第 5 章 *使用 Maven 插件管理 Service Registry 内容*.
- 有关 Java 示例，请参阅 带有参考演示的 Apicurio Registry SerDes。

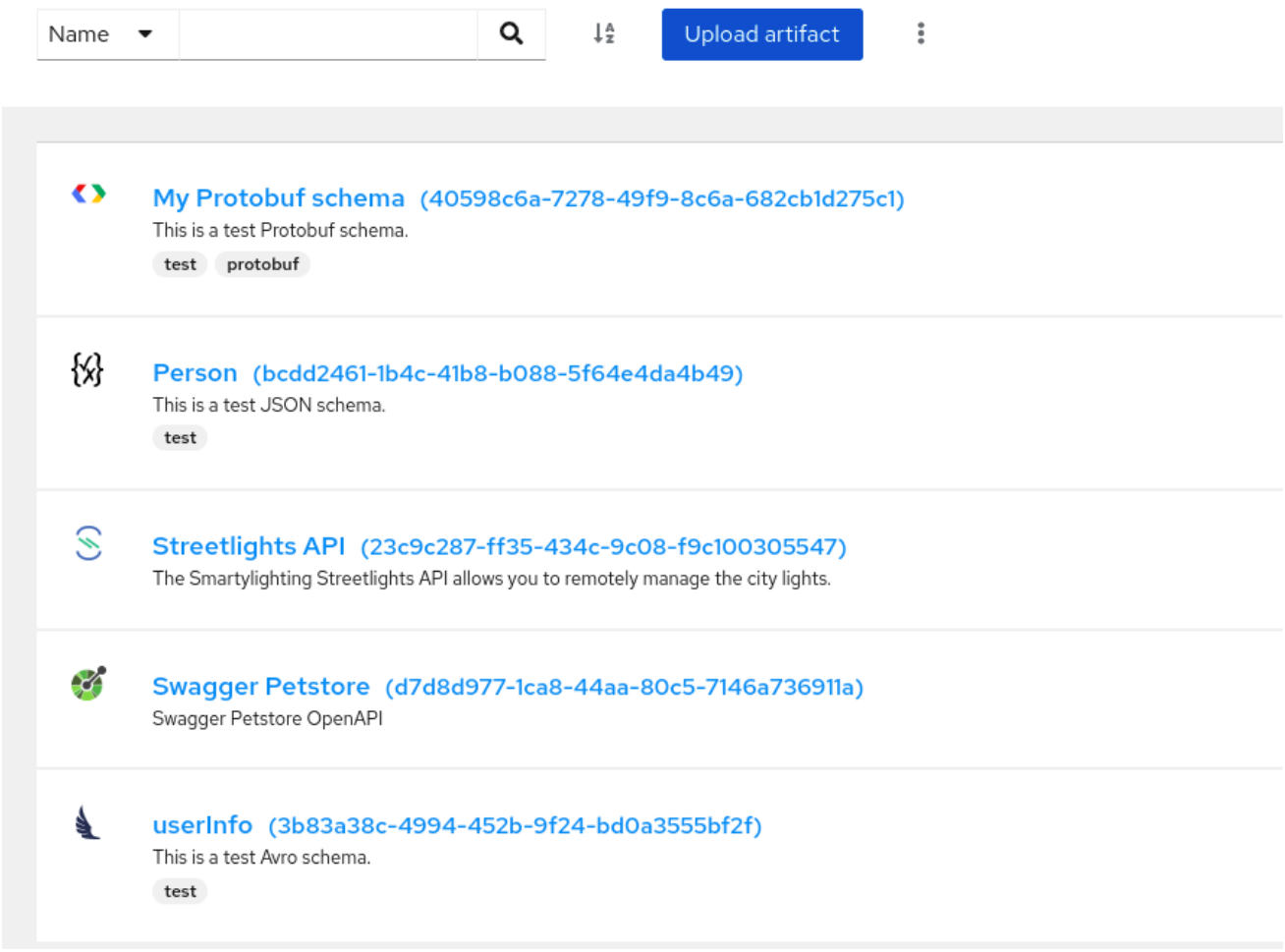
1.3. 使用 SERVICE REGISTRY WEB 控制台管理内容

您可以使用 Service Registry web 控制台浏览和搜索 registry 中存储的 schema 和 API 工件和可选组，并添加新的 schema 和 API 工件、组和版本。您可以根据标签、名称、组和描述搜索工件。您可以查看工件的内容或可用版本，或在本地下载工件文件。

您还可以为 registry 内容（全局和 API 工件）配置可选规则。当新的模式和 API 工件或版本上传到 registry 时，会应用这些可选内容验证和兼容性的规则。

如需了解更多详细信息，请参阅 第 10 章 *Service Registry 内容规则引用*。

图 1.1. Service Registry web 控制台



Service Registry web 控制台可从 **http://MY_REGISTRY_URL/ui** 获得。

其他资源

- *第 3 章 使用 Web 控制台管理 Service Registry 内容*

1.4. 客户端的 SERVICE REGISTRY REST API

客户端应用程序可以使用 Core Registry API v2 管理 Service Registry 中的 schema 和 API 工件。此 API 为以下功能提供操作：

Admin

在 **.zip** 文件中导出或导入 Service Registry 数据，并在运行时管理 Service Registry 实例的日志级别。

å·¥ä»¶

管理存储在 Service Registry 中的 schema 和 API 工件。您还可以管理工件的生命周期状态：enabled、disable 或 deprecated。

工件元数据

管理模式或 API 工件的详情。您可以编辑工件名称、描述或标签等详情。工件组等详情以及工件被创建或修改的时间为只读。

工件规则

配置规则以管理特定模式或 API 工件的内容演进，以防止将无效或不兼容的内容添加到 Service Registry 中。工件规则覆盖配置的任何全局规则。

工件版本

管理在 schema 或 API 工件被更新时创建的版本。您还可以管理工件版本的生命周期状态： enabled、disable 或 deprecated。

全局规则

配置规则以管理所有模式和 API 工件的内容演进，以防止将无效或不兼容的内容添加到 Service Registry 中。只有在工件没有配置自己的特定工件规则时，才会应用全局规则。

搜索

浏览或搜索 schema 和 API 工件和版本，例如，按名称、组、描述或标签。

System

获取 Service Registry 版本以及 Service Registry 实例的资源的限制。

用户

获取当前的 Service Registry 用户。

与其他架构 registry REST API 兼容

Service Registry 通过包括其各自 REST API 的实现来提供与以下架构 registry 的兼容性：

- Service Registry Core Registry API v1
- Confluent Schema Registry API v6
- Confluent Schema Registry API v7
- CNCF CloudEvents Schema Registry API v0

使用 Confluent 客户端库的应用程序可以使用 Service Registry 作为 drop-in 替换。如需了解更多详细信息，请参阅 替换 Confluent Schema Registry。

其他资源

- 有关 Core Registry API v2 的更多信息，请参阅 Apicurio Registry REST API 文档。
- 对于 Core Registry API v2 和所有兼容 API 的 API 文档，请浏览 Service Registry 实例的 **/apis** 端点，例如 **http://MY-REGISTRY-URL/apis**。

1.5. SERVICE REGISTRY 存储选项

Service Registry 为 registry 数据的底层存储提供以下选项：

表 1.1. Service Registry 数据存储选项

存储选项	描述
PostgreSQL 数据库	PostgreSQL 是在生产环境中性能、稳定性和数据管理（备份/恢复等）的建议数据存储选项。
AMQ Streams	为生产环境提供 Kafka 存储，其中数据库管理专业知识不可用，或者 Kafka 中的存储是特定的要求。

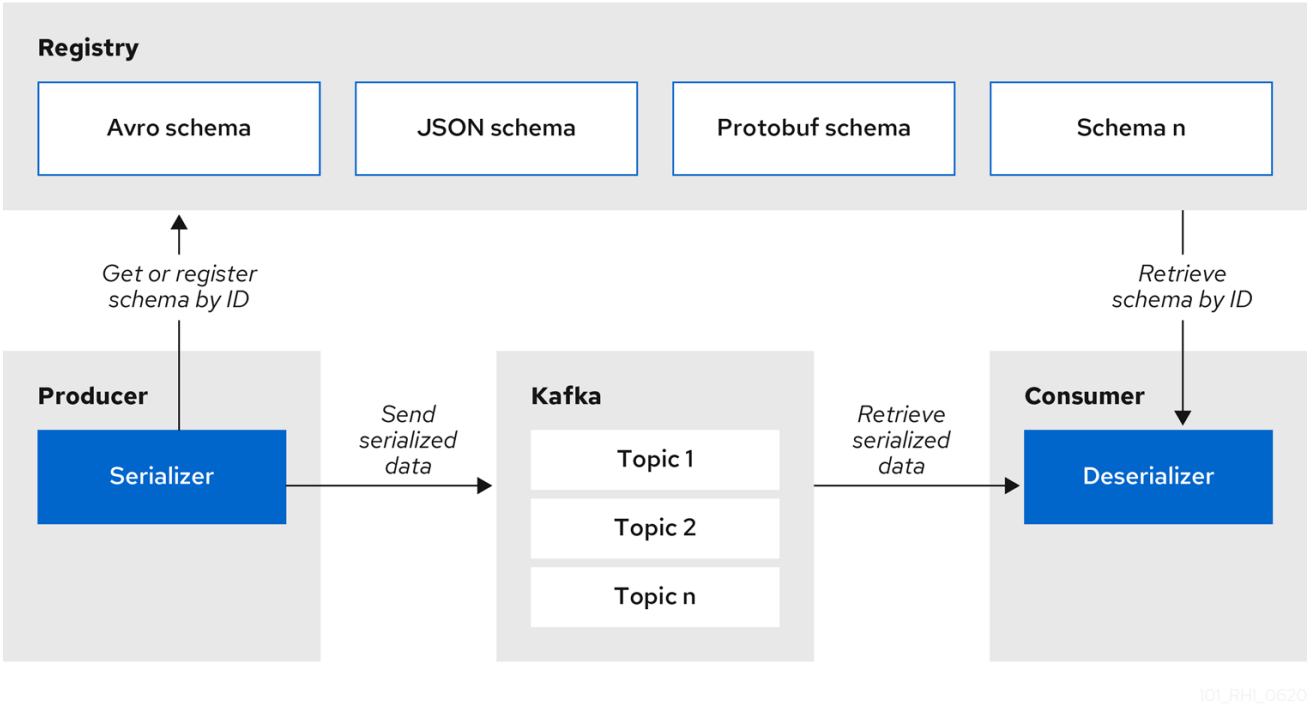
其他资源

- 如需有关存储选项的更多详细信息，请参阅在 OpenShift 上安装和部署 Service Registry。

1.6. 使用模式和 JAVA 客户端序列化器/反序列化器验证 KAFKA 消息

Kafka 制作者应用程序可以使用序列化器来对符合特定事件模式的信息进行编码。然后，Kafka 消费者应用程序可以使用反序列化器来验证消息是否使用正确的模式来序列化信息，具体取决于特定的模式 ID。

图 1.2. Service Registry 和 Kafka 客户端 SerDes 架构



Service Registry 提供 Kafka 客户端序列化器/反序列化器(SerDes)，以便在运行时验证以下消息类型：

- Apache Avro
- Google Protobuf
- JSON 架构

Service Registry Maven 存储库和源代码发行版本包括这些消息类型的 Kafka SerDes 实现，Kafka 客户端应用程序开发人员可用于与 Service Registry 集成。

这些实现包括每个支持的消息类型的自定义 Java 类，如 **io.apicurio.registry.serde.avro**，客户端应用程序在运行时可以从 Service Registry 中拉取 schema 进行验证。

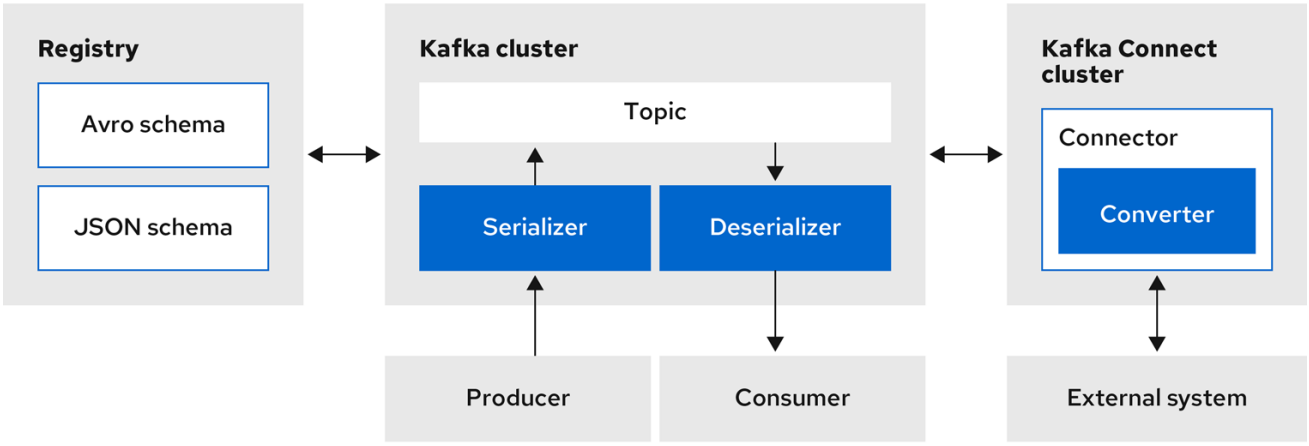
其他资源

- *第 7 章 在 Java 客户端中使用序列化器/反序列化器验证 Kafka 信息*

1.7. 使用 KAFKA CONNECT 转换器将数据流传输到外部系统

您可以在 Apache Kafka Connect 中使用 Service Registry 来流传输 Kafka 和外部系统之间的数据。使用 Kafka Connect，您可以为不同的系统定义连接器，将大量数据移动到和移出基于 Kafka 的系统。

图 1.3. Service Registry 和 Kafka Connect 架构



Service Registry 为 Kafka Connect 提供以下功能：

- Kafka Connect 模式的存储
- Kafka Connect converters for Apache Avro 和 JSON Schema
- 用于管理模式的核心 Registry API

您可以使用 Avro 和 JSON Schema 转换器将 Kafka Connect 模式映射到 Avro 或 JSON 模式。然后，这些模式可以将消息键和值序列化为紧凑的 Avro 二进制格式或人类可读的 JSON 格式。转换的 JSON 不太详细，因为消息不包含架构信息，而只有 schema ID。

Service Registry 可以管理和跟踪 Kafka 主题中使用的 Avro 和 JSON 模式。由于架构存储在 Service Registry 中，并与消息内容分离，所以每个消息必须只包含 tiny 模式标识符。对于 Kafka 等 I/O 绑定系统，这意味着生产者和消费者的总吞吐量。

Service Registry 提供的 Avro 和 JSON Schema serializers 和 deserializers (SerDes)用于 Kafka 生成者和消费者。您编写以消耗更改事件的 Kafka 消费者应用程序可以使用 Avro 或 JSON SerDes 来反序列化这些事件。您可以在任何基于 Kafka 的系统中安装 Service Registry SerDes，并将它们与 Kafka Connect 一起使用，或者与基于 Kafka Connect 的系统一起使用，如 Debezium。

其他资源

- 将 Debezium 配置为使用 Avro 序列化和 Service Registry
- 使用 Debezium 监控 Apicurio Registry 使用的 PostgreSQL 数据库示例
- Apache Kafka Connect 文档

1.8. SERVICE REGISTRY 演示示例

Service Registry 提供了开源示例应用程序，它演示了如何在不同用例中使用 Service Registry。例如，其中包括存储 Kafka serializer 和 deserializer (SerDes) Java 类使用的模式。这些类从 Service Registry 获取模式，以便在生成或消耗操作以序列化、反序列化或验证 Kafka 消息有效负载时使用。

这些应用程序演示了用例，如下例所示：

- Apache Avro Kafka SerDes

- Apache Avro Maven 插件
- Apache Camel Quarkus 和 Kafka
- CloudEvents
- Confluent Kafka SerDes
- 自定义 ID 策略
- 使用 Debezium 的事件驱动的架构
- Google Protobuf Kafka SerDes
- JSON Schema Kafka SerDes
- REST 客户端

其他资源

- 如需了解更多详细信息，请参阅 https://github.com/Apicurio/apicurio-registry-examples

1.9. SERVICE REGISTRY 可用发行版本

Service Registry 提供以下分发选项。

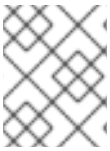
表 1.2. Service Registry Operator 和镜像

分发	位置	发行版本类别
Service Registry Operator	OpenShift Web 控制台在 **Operators → OperatorHub** 下	公开发行
Service Registry Operator 的容器镜像	Red Hat Ecosystem Catalog	公开发行
AMQ Streams 中 Kafka 存储的容器镜像	Red Hat Ecosystem Catalog	公开发行
PostgreSQL 中数据库存储的容器镜像	Red Hat Ecosystem Catalog	公开发行

表 1.3. Service Registry zip 下载

分发	位置	发行版本类别
安装自定义资源定义示例	Red Hat Software Downloads	公开发行
Service Registry v1 到 v2 迁移工具	Red Hat Software Downloads	公开发行
Maven 存储库	Red Hat Software Downloads	公开发行
源代码	Red Hat Software Downloads	公开发行

分发	位置	发行版本类别
Kafka Connect 转换器	Red Hat Software Downloads	公开发行



注意

您必须有一个 Red Hat Integration 订阅，并登录到红帽客户门户网站才能访问可用的 Service Registry 发行版本。

第 2 章 SERVICE REGISTRY 内容规则

本章介绍了用于管理 Service Registry 内容的可选规则，并提供了有关可用规则配置的详情：

2.1. 使用规则监管 SERVICE REGISTRY 内容

要管理添加到 Service Registry 的工件内容的演进，您可以配置可选规则。所有配置的全局规则或特定于工件的规则都必须通过，然后才能将新的工件版本上传到 Service Registry。配置的特定于工件的规则会覆盖任何配置的全局规则。

这些规则的目的是防止将无效的内容添加到 Service Registry 中。例如，内容可能会因为以下原因无效：

- 给定工件类型（例如 **AVRO** 或 **PROTOBUF** ）的无效语法。
- 有效的语法，但语义违反了规格。
- 不兼容，当新内容包含与当前工件版本相关的更改时。
- 工件参考完整性，例如：重复或不存在的工件引用映射。

您可以使用 Service Registry web 控制台、REST API 命令或 Java 客户端应用程序启用可选内容规则。

2.1.1. 应用规则时

只有在将内容添加到 Service Registry 时，才会应用规则。这包括以下 REST 操作：

- 添加工件
- 更新工件
- 添加工件版本

如果违反了规则，Service Registry 会返回 HTTP 错误。响应正文包含违反的规则以及显示错误的消息。

2.1.2. 规则优先级顺序

特定于工件和全局规则的优先顺序如下：

- 如果您启用特定于工件的规则，并且启用了等同的全局规则，工件规则会覆盖全局规则。
- 如果您禁用特定于工件的规则，并且启用了等同的全局规则，则应用全局规则。
- 如果您禁用特定于工件的规则，且禁用了等同的全局规则，则会为所有工件禁用该规则。

- 如果您在工件级别将规则值设置为 **NONE**，您可以覆盖启用的全局规则。在这种情况下，工件规则值为 **NONE** 优先于这个工件，但启用的全局规则将继续应用到在工件级别禁用规则的任何其他工件。

2.1.3. 规则的工作方式

每个规则都有一个名称和配置信息。Service Registry 维护每个工件和全局规则列表的规则列表。列表中的每一规则都由规则实施的名称和配置组成。

提供一个规则，其中包含工件当前版本的内容（如果存在），以及正在添加的工件的新版本。根据工件是否通过规则，规则实现会返回 true 或 false。如果没有，Service Registry 会报告 HTTP 错误响应中原因。有些规则可能没有使用之前版本的内容。例如，兼容性规则使用之前的版本，但语法或语义有效规则不使用。

其他资源

如需了解更多详细信息，请参阅 第 10 章 *Service Registry 内容规则引用*。

2.1.4. 内容规则配置

管理员可以配置 Service Registry 全局规则和特定于工件的规则。开发人员只能配置特定于工件的规则。

Service Registry 应用为特定工件配置的规则。如果在该级别上没有配置任何规则，Service Registry 将应用全局配置的规则。如果没有配置全局规则，则不会应用任何规则。

配置工件规则
您可以使用 Service Registry web 控制台或 REST API 配置工件规则。详情请查看以下内容：

- 第 3 章 *使用 Web 控制台管理 Service Registry 内容*
- Apicurio Registry REST API 文档

配置全局规则
管理员可以以多种方式配置全局规则：

- 在 REST API 中使用 **admin/rules** 操作
- 使用 Service Registry web 控制台
- 使用 Service Registry 应用程序属性设置默认全局规则

配置默认的全局规则

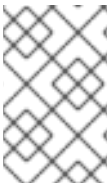
管理员可以在应用级别上配置 Service Registry 以启用或禁用全局规则。您可以在安装时配置默认全局规则，而无需使用以下应用程序属性格式进行安装后配置：

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

目前支持以下规则名称：

- **兼容性**
- **有效期**
- **完整性**

application 属性的值必须是特定于所配置规则的有效配置选项。



注意

您可以将这些应用程序属性配置为 Java 系统属性，或者在 Quarkus **application.properties** 文件中包括它们。如需了解更多详细信息，请参阅 Quarkus 文档。

第 3 章 使用 WEB 控制台管理 SERVICE REGISTRY 内容

您可以使用 Service Registry web 控制台管理 Service Registry 中存储的模式和 API 工件。这包括上传和浏览 Service Registry 内容、为内容配置可选规则，以及生成客户端 sdk 代码：

- 第 3.1 节 “使用 Service Registry web 控制台查看工件”
- 第 3.2 节 “使用 Service Registry web 控制台添加工件”
- 第 3.3 节 “使用 Service Registry web 控制台配置内容规则”
- 第 3.4 节 “使用 Service Registry web 控制台为 OpenAPI 工件生成客户端 SDK”
- 第 3.5 节 “使用 Service Registry web 控制台更改工件所有者”
- 第 3.6 节 “使用 Web 控制台配置 Service Registry 实例设置”
- 第 3.7 节 “使用 Service Registry web 控制台导出和导入数据”

3.1. 使用 SERVICE REGISTRY WEB 控制台查看工件

您可以使用 Service Registry web 控制台浏览存储在 Service Registry 中的 schema 和 API 工件。本节展示了查看 Service Registry 工件、组、版本和工件规则的简单示例。

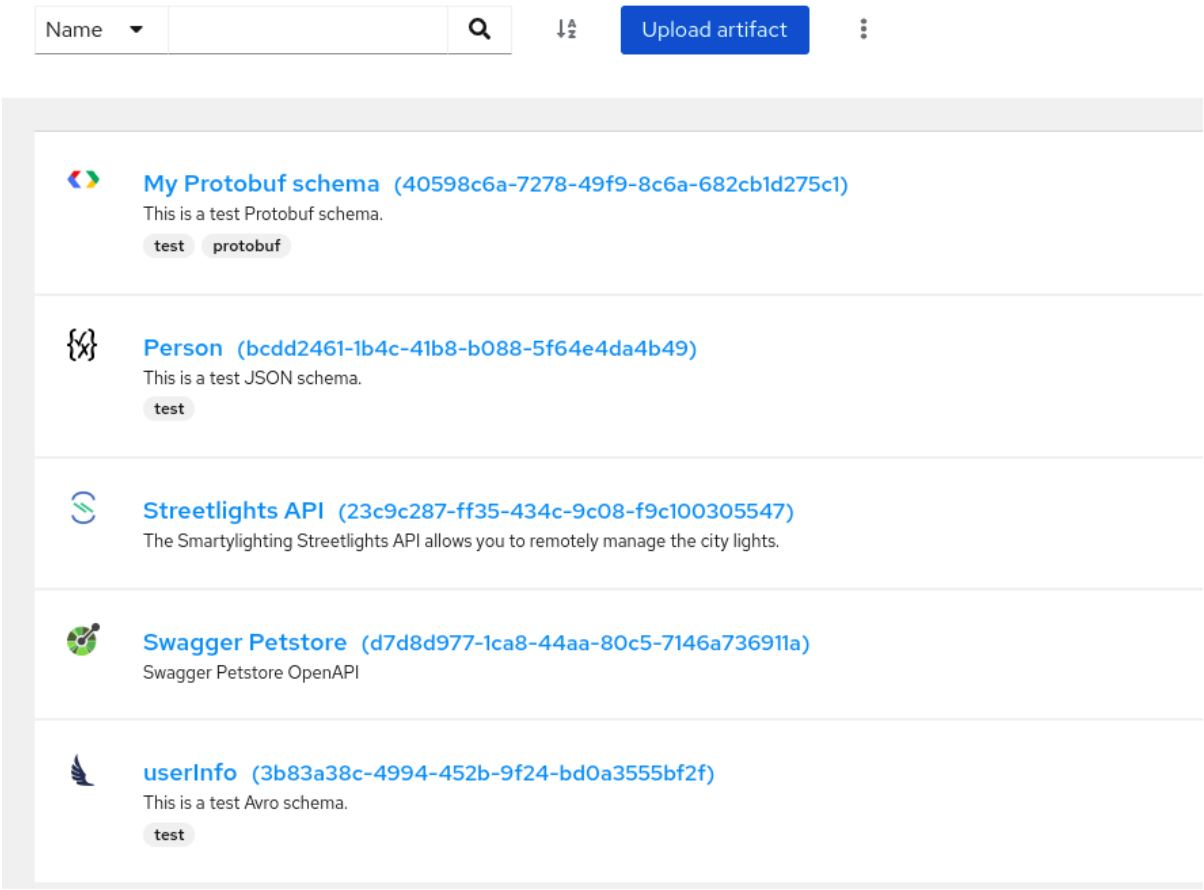
先决条件

- Service Registry 已安装并在您的环境中运行。
- 登录到 Service Registry web 控制台：
  http://MY_REGISTRY_URL/ui
- 使用 Web 控制台、命令行、Maven 插件或 Java 客户端应用程序将工件添加到 Service Registry 中。

流程

1. 在 **Artifacts** 选项卡中，浏览存储在 Service Registry 中的工件列表，或者输入搜索字符串以查找工件。您可以从列表中选择按特定条件搜索，如名称、组、标签或全局 ID。

图 3.1. Service Registry web 控制台中的工件



2. 点工件查看以下详情：
   - **概述** ：显示工件版本元数据，如工件名称、工件 ID、全局 ID、内容 ID、标签、属性等。另外，也显示可针对工件内容配置的有效性和兼容性的规则。
   - **文档** （仅限 OpenAPI 和 AsyncAPI）：显示自动生成的 REST API 文档。
   - **内容** ：显示完整工件内容的只读视图。对于 JSON 内容，您可以点 **JSON** 或 **YAML** 来显示您首选的格式。
   - **参考** ：显示此工件引用的所有工件的只读视图。您还可以单击 **引用此工件的 View 工件**。
3. 如果添加了此工件的额外版本，您可以从页面标头中的 **Version** 列表中选择它们。
4. 要将工件内容保存到本地文件，如 **my-openapi.json** 或 **my-protobuf-schema.proto**，然后点击页面末尾的 **Download**。

其他资源

- 第 3.2 节 “使用 Service Registry web 控制台添加工件”
- 第 3.3 节 “使用 Service Registry web 控制台配置内容规则”
- 第 10 章 *Service Registry 内容规则引用*

3.2. 使用 SERVICE REGISTRY WEB 控制台添加工件

您可以使用 Service Registry web 控制台将 schema 和 API 工件上传到 Service Registry。本节展示了上传 Service Registry 工件和添加新的工件版本的简单示例。

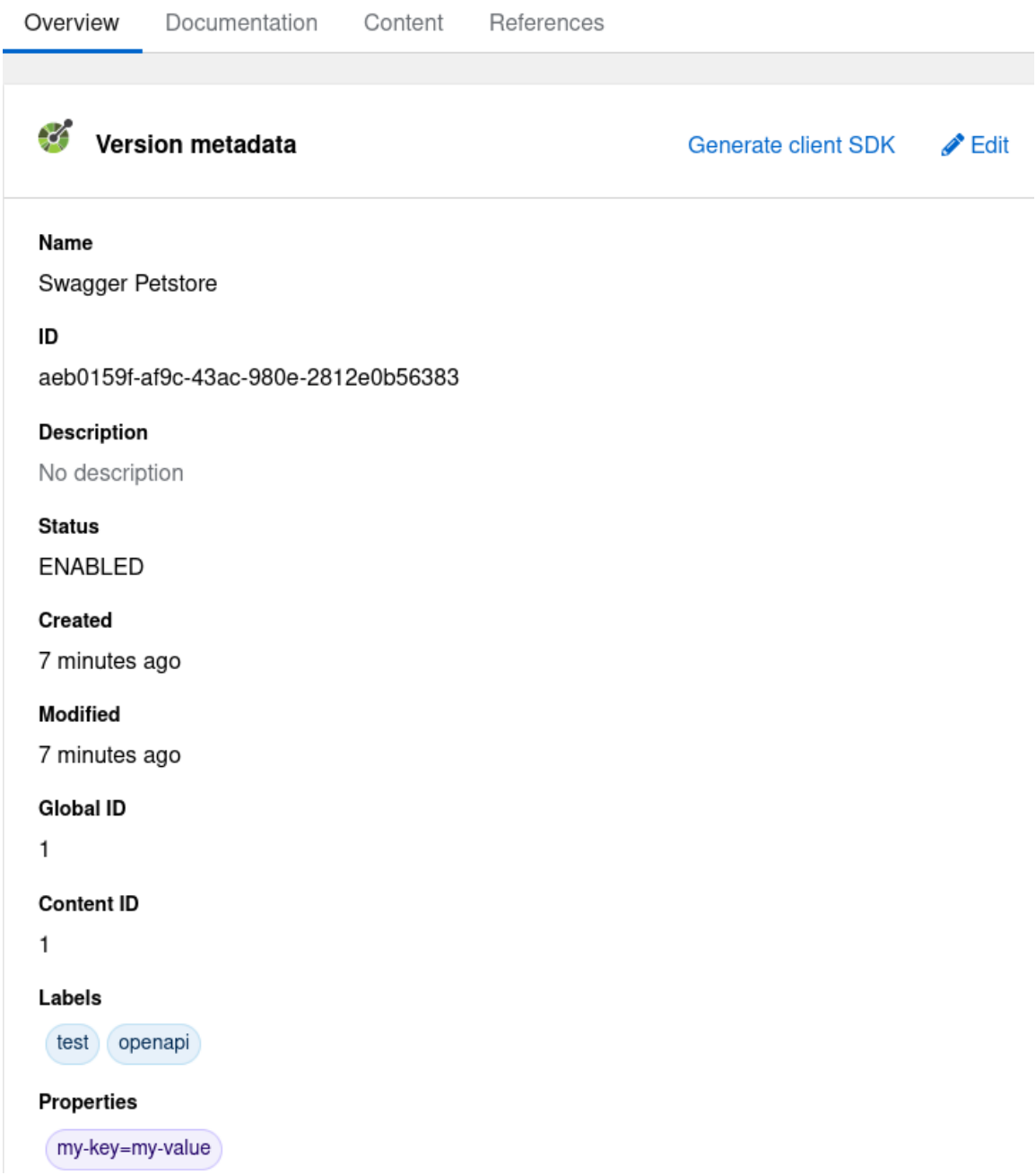
先决条件

- Service Registry 已安装并在您的环境中运行。
- 登录到 Service Registry web 控制台：
  http://MY_REGISTRY_URL/ui

流程

1. 在 **Artifacts** 选项卡中，点 **Upload artifact**，并指定以下详情：
   - **Group & ID** ：使用默认空设置自动生成工件 ID，并将工件 **添加到默认** 工件组中。或者，您可以输入可选的工件组名称或 ID。
   - **类型** ：使用默认 **Auto-Detect** 设置来自动检测工件类型，或者从列表中选择工件类型，如 **Avro Schema** 或 **OpenAPI**。您必须手动选择 **Kafka Connect Schema** 工件类型，无法自动检测到。
   - **工件** ：使用以下选项之一指定工件位置：
     - **从文件**: 点 **Browse**，然后选择一个文件，或者拖放文件。例如，**my-openapi.json** 或 **my-schema.proto**。或者，您可以在文本框中输入文件内容。
     - **从 URL** ：输入有效且可访问的 URL，然后单击 **Fetch**。例如：
       https://petstore3.swagger.io/api/v3/openapi.json。
2. 点 **Upload** 并查看工件详情：
   - **概述** ：显示工件版本元数据，如工件名称、工件 ID、全局 ID、内容 ID、标签、属性等。另外，也显示可针对工件内容配置的有效性和兼容性的规则。
   - **文档** （仅限 OpenAPI 和 AsyncAPI）：显示自动生成的 REST API 文档。
   - **内容** ：显示完整工件内容的只读视图。对于 JSON 内容，您可以点 **JSON** 或 **YAML** 来显示您首选的格式。
   - **参考** ：显示此工件引用的所有工件的只读视图。您还可以单击 **引用此工件的 View 工件**。您只能使用 Service Registry Maven 插件或 REST API 添加工件引用。
     以下示例显示了 OpenAPI 工件示例：

图 3.2. Service Registry web 控制台中的工件详情



3. 在 **Overview** 选项卡上，点 **Edit** 铅笔图标编辑工件元数据，如 name 或 description。
您还可以输入一个可选的、以逗号分隔的标签列表来搜索，或者添加与工件关联的任意属性的键值对。要添加属性，请执行以下步骤：

   a. 单击 **Add property**。

   b. 输入键名称和值。

   c. 重复前两个步骤来添加多个属性。

   d. 点击 **Save**。

4. 要将工件内容保存到本地文件，如 **my-protobuf-schema.proto** 或 **my-openapi.json**，请单击页面末尾的 **Download**。

5. 要添加新的工件版本，点页面标头中的 **Upload new version**，拖放或点 **Browse** 上传文件，如 **my-avro-schema.json** 或 **my-openapi.json**。

6. 要删除工件，请在页面标头中点 **Delete**。

警告

删除工件会删除工件及其所有版本，且无法撤销。

其他资源

- 第 3.1 节 “使用 Service Registry web 控制台查看工件”
- 第 3.3 节 “使用 Service Registry web 控制台配置内容规则”
- 第 10 章 *Service Registry 内容规则引用*

3.3. 使用 SERVICE REGISTRY WEB 控制台配置内容规则

您可以使用 Service Registry web 控制台配置可选规则，以防止将无效或不兼容的内容添加到 Service Registry 中。所有配置的、特定于工件的规则或全局规则都必须通过，然后才能将新的工件版本上传到 Service Registry。配置的特定于工件的规则会覆盖任何配置的全局规则。本节展示了配置全局和特定于工件的规则的简单示例。

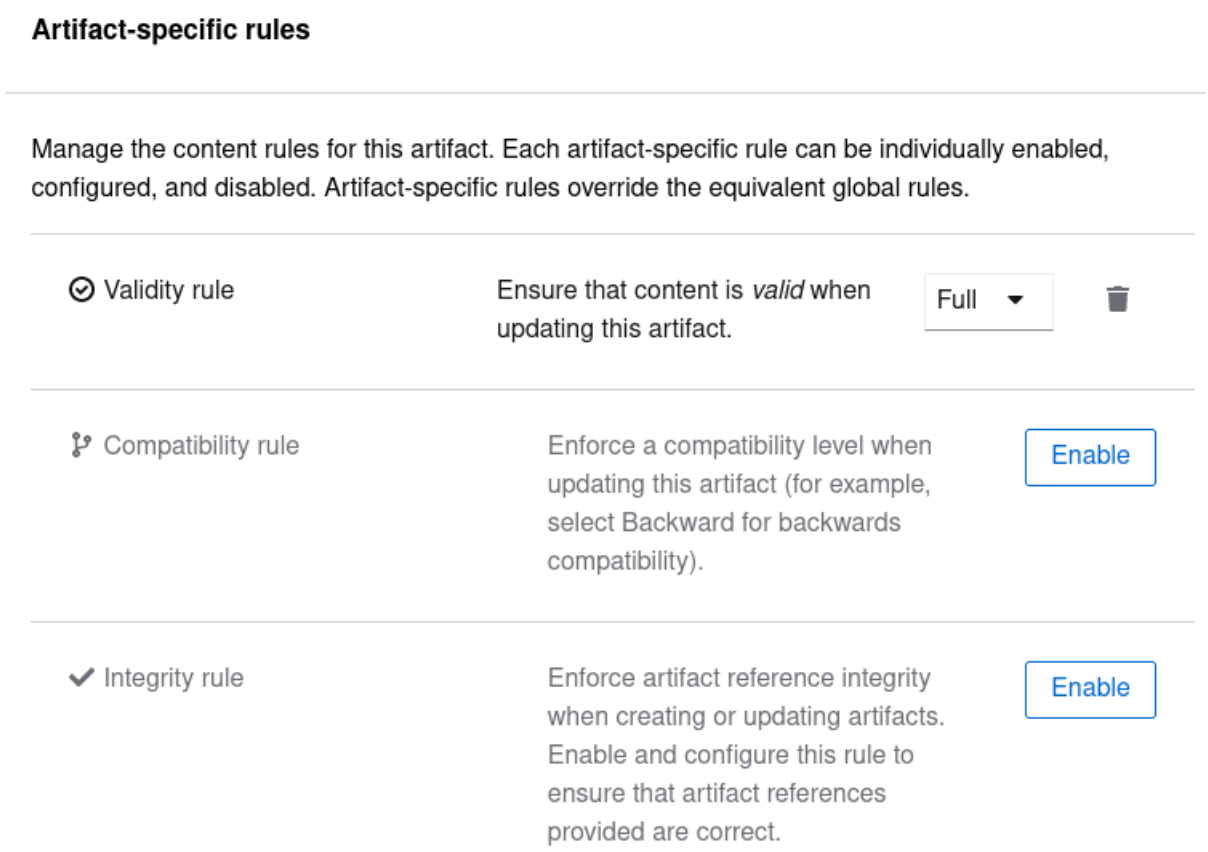
先决条件

- Service Registry 已安装并在您的环境中运行。
- 登录到 Service Registry web 控制台：
  http://MY_REGISTRY_URL/ui
- 使用 Web 控制台、命令行、Maven 插件或 Java 客户端应用程序将工件添加到 Service Registry 中。
- 启用基于角色的授权后，您对全局规则和特定于工件的规则具有管理员访问权限，或者只对特定于工件的规则进行开发人员访问权限。

流程

1. 在 **Artifacts** 选项卡中，浏览 Service Registry 中的工件列表，或者输入搜索字符串来查找工件。您可以从列表中选择按特定条件搜索，如工件名称、组、标签或全局 ID。
2. 点工件查看其版本详情和内容规则。
3. 在 **特定于** 工件的规则中，点 **Enable** 为工件内容配置有效性、兼容性或完整性规则，并从列表中选择适当的规则配置。例如，对于 **Validity 规则**，请选择 **Full**。

图 3.3. Service Registry web 控制台中的工件内容规则



4. 要访问全局规则，请点 **Global rules** 选项卡。点 **Enable** 为所有工件内容配置全局有效性、兼容性或完整性规则，然后从列表中选择适当的规则配置。
5. 要禁用工件规则或全局规则，请点规则旁边的垃圾箱图标。

其他资源

- 第 3.2 节 “使用 Service Registry web 控制台添加工件”
- 第 10 章 *Service Registry 内容规则引用*

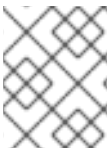
3.4. 使用 SERVICE REGISTRY WEB 控制台为 OPENAPI 工件生成客户端 SDK

您可以使用 Service Registry web 控制台为 OpenAPI 工件配置、生成和下载客户端软件开发套件(SDK)。然后，您可以使用生成的客户端 SDK 根据 OpenAPI 为特定平台构建客户端应用程序。

Service Registry 为以下编程语言生成客户端 SDK：

- C#
- Go
- Java
- PHP
- Python

- Ruby
- Swift
- TypeScript



注意

OpenAPI 工件的客户端 SDK 仅在浏览器中运行，无法使用 API 自动运行。每次在 Service Registry 中添加新工件版本时，您必须重新生成客户端 SDK。

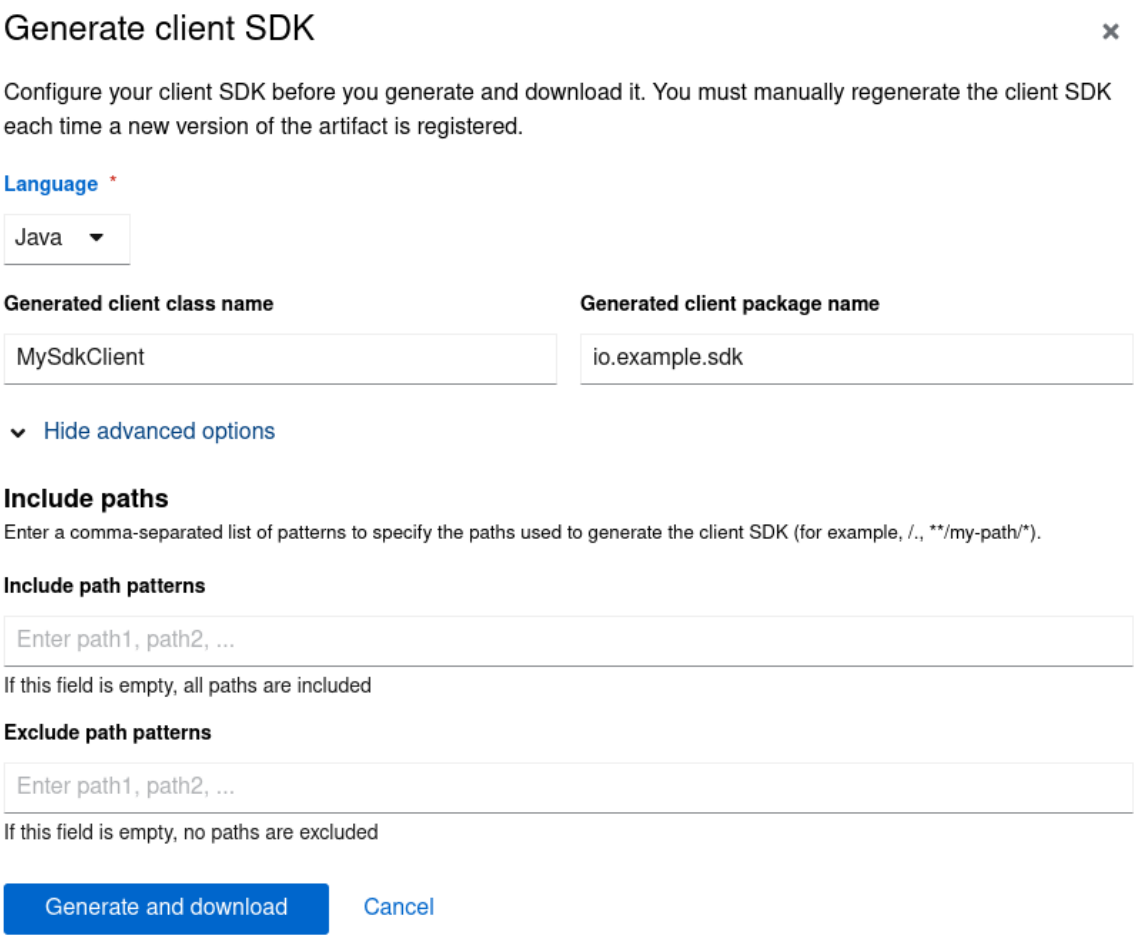
先决条件

- Service Registry 已安装并在您的环境中运行。
- 登录到 Service Registry web 控制台：
  http://MY_REGISTRY_URL/ui
- 使用 Web 控制台、命令行、Maven 插件或 Java 客户端应用程序将 OpenAPI 工件添加到 Service Registry 中。

流程

1. 在 **Artifacts** 选项卡中，浏览存储在 Service Registry 中的工件列表，或者输入搜索字符串来查找特定的 OpenAPI 工件。您可以从列表中选择按名称、组、标签或全局 ID 等条件进行搜索。
2. 点列表中的 OpenAPI 工件来查看其详情。
3. 在 **Version metadata** 部分中，点 **Generate client SDK**，并在对话框中配置以下设置：
   - **语言** ：选择生成客户端 SDK 的编程语言，例如 **Java**。
   - **生成的客户端类名称** ：输入客户端 SDK 的类名称，如 **MyJavaClientSDK。**
   - **生成的客户端软件包名称** ：输入客户端 SDK 的软件包名称，例如 **io.my.example.sdk**
4. 点 **Show advanced settings** 配置可选的以逗号分隔的路径模式列表来包含或排除：
   - **包括路径模式** ：输入生成客户端 SDK 时要包括的具体路径，例如： ****/ netobserv, **/my-path8:0:1::**。如果此字段为空，则会包括所有路径。
   - **排除路径模式** ：在生成客户端 SDK 时输入要排除的具体路径，例如： beyond **/my-other-pathAttr**。如果此字段为空，则不会排除任何路径。

图 3.4. 在 Service Registry web 控制台中生成 Java 客户端 SDK



5. 当您在对话框中配置了设置后，点 **Generate and download**。

6. 在对话框中输入 client SDK 的文件名，如 **my-client-java.zip**，然后单击 **Save** 以下载。

其他资源

- Service Registry 使用 Microsoft 中的 Kiota 来生成客户端 SDK。如需更多信息，请参阅 GitHub 中的 Kiota 项目。
- 有关使用生成的 SDK 来构建客户端应用程序的详情和示例，请参阅 Kiota 文档。

3.5. 使用 SERVICE REGISTRY WEB 控制台更改工件所有者

作为管理员，或作为 schema 或 API 工件的所有者，您可以使用 Service Registry Web 控制台将工件所有者更改为另一个用户帐户。

例如，如果在 **Settings** 选项卡中为 Service Registry 实例设置了 **Artifact owner-only** 授权选项，则此功能很有用，以便只有所有者或管理员可以修改工件。如果所有者用户离开机构或所有者帐户已被删除，您可能需要更改所有者。

注意

*只有*在 Service Registry 实例部署时启用了身份验证时，才会显示 **Artifact owner-only authorization** 设置和工件 **Owner** 字段。如需了解更多详细信息，请参阅在 OpenShift 上安装和部署 Service Registry。

先决条件

- 部署 Service Registry 实例并创建工件。
- 以工件的当前所有者或管理员身份登录到 Service Registry web 控制台：**http://MY_REGISTRY_URL/ui**

流程

1. 在 **Artifacts** 选项卡中，浏览存储在 Service Registry 中的工件列表，或者输入搜索字符串以查找工件。您可以从列表中选择按名称、组、标签或全局 ID 等条件进行搜索。
2. 点您要重新分配的工件。
3. 在 **Version metadata** 部分中，单击 **Owner** 字段旁边的铅笔图标。
4. 在 **New owner** 字段中，选择或输入帐户名称。
5. 单击 **更改所有者**。

其他资源

- 在 OpenShift 上安装和部署 Service Registry

3.6. 使用 WEB 控制台配置 SERVICE REGISTRY 实例设置

作为管理员，您可以使用 Service Registry web 控制台在运行时为 Service Registry 实例配置动态设置。您可以管理身份验证、授权和 API 兼容性等功能的配置选项。

注意

只有在部署了 Service Registry 实例时，才会在 web 控制台中显示身份验证和授权设置。如需了解更多详细信息，请参阅在 OpenShift 上安装和部署 Service Registry。

先决条件

- 已部署 Service Registry 实例。
- 使用管理员访问权限登录到 Service Registry web 控制台：**http://MY_REGISTRY_URL/ui**

流程

1. 在 Service Registry web 控制台中点 **Settings** 选项卡。
2. 选择您要为此 Service Registry 实例配置的设置：

表 3.1. 身份验证设置

设置	描述
HTTP 基本身份验证	仅在已经启用身份验证时显示。选择后，除了 OAuth 外，Service Registry 用户可以使用 HTTP 基本身份验证进行身份验证。默认不选择。

表 3.2. 授权设置

设置	描述
匿名读取访问	仅在已经选择身份验证时显示。选择后，Service Registry 会为没有任何凭证的匿名用户授予只读访问权限。如果要使用此实例在外部发布模式或 API，则此设置很有用。默认不选择。
工件仅所有者授权	仅在已经启用身份验证时显示。选择后，只有创建工件的用户才能修改该工件。默认不选择。
工件组所有者-仅限授权	仅在已经启用身份验证并选择了 **Artifact owner-only 授权** 时显示。选择后，只有创建工件组的用户具有对该工件组的写入权限，例如在该组中添加或删除工件。默认不选择。
经过身份验证的读访问权限	仅在已经启用身份验证时显示。选择后，无论其用户角色是什么，Service Registry 会至少授予来自任何经过身份验证的用户的请求的只读访问权限。默认不选择。

表 3.3. 兼容性设置

设置	描述
旧 ID 模式（兼容性 API）	选择后，Confluent Schema Registry 兼容性 API 使用 **globalId** 而不是 **contentId** 作为工件标识符。当从基于 v1 Core Registry API 的传统 Service Registry 实例迁移时，此设置很有用。默认不选择。

表 3.4. Web 控制台设置

设置	描述
下载链接到期	生成的到 **.zip** 下载文件的链接秒数在因为安全原因（例如，从实例导出工件数据时）之前处于活跃状态。默认值为 30 秒。
UI 只读模式	选择后，Service Registry web 控制台被设置为 read-only，从而导致创建、读取、更新或删除操作。使用 Core Registry API 所做的更改不受此设置的影响。默认不选择。

表 3.5. 其他属性

设置	描述
删除工件版本	选择后，用户可以使用 Core Registry API 删除此实例中的工件版本。默认不选择。

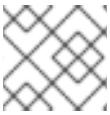
其他资源

- 在 OpenShift 上安装和部署 Service Registry

3.7. 使用 SERVICE REGISTRY WEB 控制台导出和导入数据

作为管理员，您可以使用 Service Registry web 控制台从一个 Service Registry 实例导出数据，并将此数据导入到另一个 Service Registry 实例中。您可以使用此功能在不同实例间轻松迁移数据。

以下示例演示了如何在一个 Service Registry 实例中导出和导入现有数据，并将其导入另一个实例。Service Registry 实例中包含的所有工件数据都会在 **.zip** 文件中导出。



注意

您只能导入从另一个 Service Registry 实例导出的 Service Registry 数据。

先决条件

- Service Registry 实例已创建，如下所示：
  - 从 中导出的源实例至少包含一个 schema 或 API 工件
  - 要导入到的目标实例为空，以保留唯一 ID
- 使用管理员访问权限登录到 Service Registry web 控制台：
  http://MY_REGISTRY_URL/ui

流程

1. 在源 Service Registry 实例的 web 控制台中，查看 **Artifacts** 选项卡。
2. 点 **Upload artifact** 旁的选项图标（三个垂直点），然后选择 **Download all artifacts (.zip file)**，将这个 Service Registry 实例的数据导出到 **.zip** 下载文件。
3. 在目标 Service Registry 实例的 web 控制台中，查看 **Artifacts** 选项卡。
4. 点 **Upload artifact** 旁的选项图标，然后选择 **Upload multiple artifacts**。
5. 拖放或浏览到您之前导出的 **.zip** 下载文件。
6. 点 **Upload**，并等待数据导入。

第 4 章 使用 REST API 管理 SERVICE REGISTRY 内容

客户端应用程序可以使用 Service Registry REST API 操作来管理 Service Registry 中的 schema 和 API 工件，例如在生产环境中部署的 CI/CD 管道中。Core Registry API v2 为存储在 Service Registry 中的工件、版本、元数据和规则提供操作。如需更多信息，请参阅 Apicurio Registry REST API 文档 。

本章介绍了如何使用核心 Registry API v2 执行以下任务的示例：

- 第 4.1 节 “使用 Service Registry REST API 命令管理 schema 和 API 工件”
- 第 4.2 节 “使用 Service Registry REST API 命令管理 schema 和 API 工件版本”
- 第 4.3 节 “使用 Service Registry REST API 命令管理 schema 和 API 工件引用”
- 第 4.4 节 “使用 Service Registry REST API 命令导出和导入 registry 数据”

先决条件

- 第 1 章 *Service Registry 简介*

其他资源

- Apicurio Registry REST API 文档

4.1. 使用 SERVICE REGISTRY REST API 命令管理 SCHEMA 和 API 工件

本节展示了一个简单的基于 curl 的示例，它使用 Core Registry API v2 在 Service Registry 中添加和检索简单的模式工件。

先决条件

- Service Registry 已安装并在您的环境中运行。

流程

1. 使用 **/groups/{group}/artifacts** 操作向 Service Registry 添加工件。以下示例 **curl** 命令为共享价格应用程序添加一个简单的模式工件：

```
$ curl -X POST -H "Content-Type: application/json; artifactType=AVRO" \
  -H "X-Registry-ArtifactId: share-price" \
  -H "Authorization: Bearer $ACCESS_TOKEN" \
  --data '{"type":"record","name":"price","namespace":"com.example", \
   "fields":[{"name":"symbol","type":"string"},{"name":"price","type":"string"}]}' \
  MY-REGISTRY-URL/apis/registry/v2/groups/my-group/artifacts
```

   - 这个示例添加了一个 Apache Avro 模式工件，工件 ID 为 **share-price**。如果没有指定唯一的工件 ID，Service Registry 会自动生成一个 UUID。
   - **MY-REGISTRY-URL** 是在其上部署 Service Registry 的主机名。例如： **my-cluster-service-registry-myproject.example.com**。
   - 本例在 API 路径中指定 **my-group** 的组 ID。如果没有指定唯一的组 ID，则必须在 API 路径中指定 **../groups/default**。

2. 验证响应是否包含预期的 JSON 正文，以确认是否已添加工件。例如：

```
{"createdBy":"","createdOn":"2021-04-16T09:07:51+0000","modifiedBy":"",
"modifiedOn":"2021-04-16T09:07:51+0000","id":"share-price","version":"1",
"type":"AVRO","globalId":2,"state":"ENABLED","groupId":"my-group","contentId":2}
```

- 添加工件时没有指定 version，因此会自动创建默认版本 **1**。
- 这是添加到 Service Registry 的第二个工件，因此全局 ID 和内容 ID 的值为 **2**。

3. 使用 API 路径中的工件 ID 从 Service Registry 检索工件内容。在本例中，指定的 ID 是 **share-price**:

```
$ curl -H "Authorization: Bearer $ACCESS_TOKEN" \
 MY-REGISTRY-URL/apis/registry/v2/groups/my-group/artifacts/share-price
 {"type":"record","name":"price","namespace":"com.example",
  "fields":[{"name":"symbol","type":"string"},{"name":"price","type":"string"}]}
```

其他资源

- 如需了解更多详细信息，请参阅 Apicurio Registry REST API 文档 。

4.2. 使用 SERVICE REGISTRY REST API 命令管理 SCHEMA 和 API 工件版本

如果您在使用 Core Registry API v2 添加 schema 和 API 工件时没有指定工件版本，则 Service Registry 会自动生成版本。创建新工件时的默认版本为 **1**。

Service Registry 还支持自定义版本控制，您可以在其中将 **X-Registry-Version** HTTP 请求标头用作字符串来指定版本。指定自定义版本值会覆盖在创建或更新工件时通常会分配的默认版本。然后，您可以在执行需要版本的 REST API 操作时使用此版本值。

本节展示了一个简单的基于 curl 的示例，它使用 Core Registry API v2 在 Service Registry 中添加并检索自定义 Apache Avro 模式版本。您可以指定自定义版本来添加或更新工件，或添加工件版本。

先决条件

- Service Registry 已安装并在您的环境中运行。

流程

1. 使用 **/groups/{group}/artifacts** 操作在 registry 中添加工件版本。以下示例 **curl** 命令为共享价格应用程序添加一个简单的工件：

```
$ curl -X POST -H "Content-Type: application/json; artifactType=AVRO" \
  -H "X-Registry-ArtifactId: my-share-price" -H "X-Registry-Version: 1.1.1" \
  -H "Authorization: Bearer $ACCESS_TOKEN" \
  --data '{"type":"record","name":" p","namespace":"com.example", \
   "fields":[{"name":"symbol","type":"string"},{"name":"price","type":"string"}]}' \
   MY-REGISTRY-URL/apis/registry/v2/groups/my-group/artifacts
```

- 这个示例添加了一个 Avro 模式工件，工件 ID 为 **my-share-price** 和 **1.1.1** 版本。如果没有指定版本，Service Registry 会自动生成默认版本 **1**。
- **MY-REGISTRY-URL** 是在其上部署 Service Registry 的主机名。例如： **my-cluster-service-registry-myproject.example.com**。

- 本例在 API 路径中指定 **my-group** 的组 ID。如果没有指定唯一的组 ID，则必须在 API 路径中指定 **../groups/default**。

2. 验证响应是否包含预期的 JSON 正文，以确认是否已添加了自定义工件版本。例如：

```
{"createdBy":"","createdOn":"2021-04-16T10:51:43+0000","modifiedBy":"",
"modifiedOn":"2021-04-16T10:51:43+0000","id":"my-share-price","version":"1.1.1",
"type":"AVRO","globalId":3,"state":"ENABLED","groupId":"my-group","contentId":3}
```

- 在添加工件时，指定了 **1.1.1** 的自定义版本。
- 这是添加到 registry 中的第三个工件，因此全局 ID 和内容 ID 的值为 **3**。

3. 使用 API 路径中的工件 ID 和版本，从 registry 检索工件内容。在本例中，指定的 ID 是 **my-share-price**，版本为 **1.1.1** ：

```
$ curl -H "Authorization: Bearer $ACCESS_TOKEN" \
MY-REGISTRY-URL/apis/registry/v2/groups/my-group/artifacts/my-share-
price/versions/1.1.1
{"type":"record","name":"price","namespace":"com.example",
 "fields":[{"name":"symbol","type":"string"},{"name":"price","type":"string"}]}
```

其他资源

- 如需了解更多详细信息，请参阅 Apicurio Registry REST API 文档 。

4.3. 使用 SERVICE REGISTRY REST API 命令管理 SCHEMA 和 API 工件引用

有些 Service Registry 工件类型可能会包括从一个 *工件文件到另一个工件引用*。您可以通过定义可重复使用的模式或 API 工件来创建效率，然后从工件引用中的多个位置引用它们。

以下工件类型支持工件引用：

- Apache Avro
- Google Protobuf
- JSON 架构
- OpenAPI
- AsyncAPI

本节展示了一个简单的基于 curl 的示例，它使用 Core Registry API v2 添加和检索对 Service Registry 中简单 Avro 模式工件引用的工件引用。

本例首先创建一个名为 **ItemId** 的模式工件：

ItemId 模式

```
{
  "namespace":"com.example.common",
  "name":"ItemId",
  "type":"record",
```

```
    "fields":[
      {
        "name":"id",
        "type":"int"
      }
    ]
}
```

然后，创建一个名为 **Item** 的 schema 工件，其中包含对嵌套 **ItemId** 工件的引用。

带有嵌套 ItemId 模式的项目模式

```
{
    "namespace":"com.example.common",
    "name":"Item",
    "type":"record",
    "fields":[
      {
        "name":"itemId",
        "type":"com.example.common.ItemId"
      },
    ]
}
```

先决条件

- Service Registry 已安装并在您的环境中运行。

流程

1. 添加 **ItemId** 模式工件，您要使用 **/groups/{group}/artifacts** 操作创建嵌套工件引用：

```
$ curl -X POST MY-REGISTRY-URL/apis/registry/v2/groups/my-group/artifacts \
  -H "Content-Type: application/json; artifactType=AVRO" \
  -H "X-Registry-ArtifactId: ItemId" \
  -H "Authorization: Bearer $ACCESS_TOKEN" \
  --data '{"namespace": "com.example.common", "type": "record", "name": "ItemId", "fields":
[{"name":"id", "type":"int"}]}'
```

   - 这个示例添加了一个 Avro schema 工件，工件 ID 为 **ItemId**。如果没有指定唯一的工件 ID，Service Registry 会自动生成一个 UUID。
   - **MY-REGISTRY-URL** 是在其上部署 Service Registry 的主机名。例如： **my-cluster-service-registry-myproject.example.com**。
   - 本例在 API 路径中指定 **my-group** 的组 ID。如果没有指定唯一的组 ID，则必须在 API 路径中指定 **../groups/default**。

2. 验证响应是否包含预期的 JSON 正文，以确认是否已添加工件。例如：

```
{"name":"ItemId","createdBy":"","createdOn":"2022-04-
14T10:50:09+0000","modifiedBy":"","modifiedOn":"2022-04-
14T10:50:09+0000","id":"ItemId","version":"1","type":"AVRO","globalId":1,"state":"ENABLED",
"groupId":"my-group","contentId":1,"references":[]}
```

3. 添加 **Item** 模式工件，它包括使用 **/groups/{group}/artifacts** 操作的到 **ItemId** 模式的工作引用：

```
$ curl -X POST MY-REGISTRY-URL/apis/registry/v2/groups/my-group/artifacts \
-H 'Content-Type: application/create.extended+json' \
-H "X-Registry-ArtifactId: Item" \
-H 'X-Registry-ArtifactType: AVRO' \
-H "Authorization: Bearer $ACCESS_TOKEN" \
--data-raw '{
    "content": "{\r\n \"namespace\":\"com.example.common\",\r\n  \"name\":\"Item\",\r\n
\"type\":\"record\",\r\n  \"fields\":[\r\n   {\r\n  \"name\":\"itemId\",\r\n
\"type\":\"com.example.common.ItemId\"\r\n        }\r\n    ]\r\n}",
    "references": [
        {
            "groupId": "my-group",
            "artifactId": "ItemId",
            "name": "com.example.common.ItemId",
            "version": "1"
        }
    ]
}'
```

- 对于工件引用，您必须指定 **application/create.extended+json** 的自定义内容类型，该类型扩展 **application/json** 内容类型。

4. 验证响应是否包含预期的 JSON 正文，以确认工件是通过引用创建的。例如：

```
{"name":"Item","createdBy":"","createdOn":"2022-04-
14T11:52:15+0000","modifiedBy":"","modifiedOn":"2022-04-
14T11:52:15+0000","id":"Item","version":"1","type":"AVRO","globalId":2,"state":"ENABLED","g
roupId":"my-group","contentId":2, "references":[{"artifactId":"ItemId","groupId":"my-
group","name":"ItemId","version":"1"}] }
```

5. 通过指定包含引用的工件的全局 ID，从 Service Registry 检索工件引用。在本例中，指定的全局 ID 为 **2**：

```
$ curl -H "Authorization: Bearer $ACCESS_TOKEN" MY-REGISTRY-
URL/apis/registry/v2/ids/globalIds/2/references
```

6. 验证响应是否包含此工件引用的预期 JSON 正文。例如：

```
[{"groupId":"my-
group","artifactId":"ItemId","version":"1","name":"com.example.common.ItemId"}]
```

其他资源

- 如需了解更多详细信息，请参阅 Apicurio Registry REST API 文档。
- 有关工件引用的更多示例，请参阅在 *第 8 章 在 Java 客户端中配置 Kafka serializers/deserializers* 中配置每个工件类型部分。

4.4. 使用 SERVICE REGISTRY REST API 命令导出和导入 REGISTRY 数据

作为管理员，您可以使用 Core Registry API v2 从一个 Service Registry 实例导出数据并导入到另一个 Service Registry 实例，以便您可以在不同实例之间迁移数据。

本节展示了一个简单的基于 curl 的示例，它使用 Core Registry API v2 将 **.zip** 格式的现有数据导出并导入到另一个 Service Registry 实例。Service Registry 实例中包含的所有工件数据都会在 **.zip** 文件中导出。

注意

您只能导入从另一个 Service Registry 实例导出的 Service Registry 数据。

先决条件

- Service Registry 已安装并在您的环境中运行。
- Service Registry 实例已创建：
  - 要从中导出数据的源实例至少包含一个 schema 或 API 工件。
  - 要导入数据的目标实例为空，以保留唯一的 ID。

流程

1. 从现有源 Service Registry 实例导出 Service Registry 数据：

```
$ curl MY-REGISTRY-URL/apis/registry/v2/admin/export \
  -H "Authorization: Bearer $ACCESS_TOKEN" \
  --output my-registry-data.zip
```

MY-REGISTRY-URL 是在其上部署源 Service Registry 的主机名。例如： **my-cluster-source-registry-myproject.example.com**。

2. 将 registry 数据导入到目标 Service Registry 实例中：

```
$ curl -X POST "MY-REGISTRY-URL/apis/registry/v2/admin/import" \
  -H "Content-Type: application/zip" -H "Authorization: Bearer $ACCESS_TOKEN" \
  --data-binary @my-registry-data.zip
```

MY-REGISTRY-URL 是在其上部署目标 Service Registry 的主机名。例如： **my-cluster-target-registry-myproject.example.com**。

其他资源

- 如需了解更多详细信息，请参阅 Apicurio Registry REST API 文档中的 **admin** 端点。
- 有关从 Service Registry 版本 1.x 迁移到 2.x 的导出工具的详情，请参阅 1.x 版本的 Apicurio Registry 导出工具。

第 5 章 使用 MAVEN 插件管理 SERVICE REGISTRY 内容

在开发客户端应用程序时，您可以使用 Service Registry Maven 插件来管理存储在 Service Registry 中的 schema 和 API 工件：

- 第 5.1 节 “使用 Maven 插件添加 schema 和 API 工件”
- 第 5.2 节 “使用 Maven 插件下载模式和 API 工件”
- 第 5.3 节 “使用 Maven 插件测试模式和 API 工件”
- 第 5.4 节 “使用 Service Registry Maven 插件手动添加工件引用”
- 第 5.5 节 “使用 Service Registry Maven 插件自动添加工件引用”

先决条件

- Service Registry 已安装并在您的环境中运行。
- 在您的环境中安装并配置 Apache Maven。

5.1. 使用 MAVEN 插件添加 SCHEMA 和 API 工件

Maven 插件的最常见用例是在构建客户端应用程序期间添加工件。您可以使用 **注册** 执行目标完成此操作。

先决条件

- 您已为您的客户端应用程序创建了一个 Maven 项目。如需了解更多详细信息，请参阅 Apache Maven 文档。

流程

1. 更新 Maven **pom.xml** 文件，以使用 **apicurio-registry-maven-plugin** 来注册工件。以下示例显示了注册 Apache Avro 和 GraphQL 模式：

```
<plugin>
  <groupId>io.apicurio</groupId>
  <artifactId>apicurio-registry-maven-plugin</artifactId>
  <version>${apicurio.version}</version>
  <executions>
    <execution>
      <phase>generate-sources</phase>
      <goals>
        <goal>register</goal> (1)
      </goals>
      <configuration>
        <registryUrl>MY-REGISTRY-URL/apis/registry/v2</registryUrl> (2)
        <authServerUrl>MY-AUTH-SERVER</authServerUrl>
        <clientId>MY-CLIENT-ID</clientId>
        <clientSecret>MY-CLIENT-SECRET</clientSecret> (3)
        <clientScope>MY-CLIENT-SCOPE</clientScope>
        <artifacts>
          <artifact>
```

```
            <groupId>TestGroup</groupId> 4
            <artifactId>FullNameRecord</artifactId>
            <file>${project.basedir}/src/main/resources/schemas/record.avsc</file>
            <ifExists>FAIL</ifExists>
          </artifact>
          <artifact>
            <groupId>TestGroup</groupId>
            <artifactId>ExampleAPI</artifactId> 5
            <type>GRAPHQL</type>
            <file>${project.basedir}/src/main/resources/apis/example.graphql</file>
            <ifExists>RETURN_OR_UPDATE</ifExists>
            <canonicalize>true</canonicalize>
          </artifact>
        </artifacts>
      </configuration>
    </execution>
  </executions>
</plugin>
```

1. 指定 **register** 作为执行目标，将 schema 工件上传到 Service Registry。
2. 使用 ./ **apis/registry/v2** 端点指定 Service Registry URL。
3. 如果需要身份验证，您可以指定身份验证服务器和客户端凭证。
4. 指定 Service Registry 工件组 ID。如果您不想使用唯一的组 ID，您可以指定 **默认组**。
5. 您可以使用指定的组 ID、工件 ID 和位置注册多个工件。

2. 构建您的 Maven 项目，例如使用 **mvn package** 命令。

其他资源

- 有关使用 Apache Maven 的详情，请查看 Apache Maven 文档。
- 有关使用 Service Registry Maven 插件的开源示例，请参阅 Apicurio Registry 演示示例。

5.2. 使用 MAVEN 插件下载模式和 API 工件

您可以使用 Maven 插件从 Service Registry 下载工件。例如，当从注册的模式生成代码时，这通常很有用。

先决条件

- 您已为您的客户端应用程序创建了一个 Maven 项目。如需了解更多详细信息，请参阅 Apache Maven 文档。

流程

1. 更新 Maven **pom.xml** 文件，以使用 **apicurio-registry-maven-plugin** 下载工件。以下示例显示了下载 Apache Avro 和 GraphQL 模式。

```
<plugin>
  <groupId>io.apicurio</groupId>
```

```
<artifactId>apicurio-registry-maven-plugin</artifactId>
<version>${apicurio.version}</version>
<executions>
 <execution>
  <phase>generate-sources</phase>
  <goals>
   <goal>download</goal> (1)
  </goals>
  <configuration>
    <registryUrl>MY-REGISTRY-URL/apis/registry/v2</registryUrl> (2)
    <authServerUrl>MY-AUTH-SERVER</authServerUrl>
    <clientId>MY-CLIENT-ID</clientId>
    <clientSecret>MY-CLIENT-SECRET</clientSecret> (3)
    <clientScope>MY-CLIENT-SCOPE</clientScope>
    <artifacts>
      <artifact>
        <groupId>TestGroup</groupId> (4)
        <artifactId>FullNameRecord</artifactId> (5)
        <file>${project.build.directory}/classes/record.avsc</file>
        <overwrite>true</overwrite>
      </artifact>
      <artifact>
        <groupId>TestGroup</groupId>
        <artifactId>ExampleAPI</artifactId>
        <version>1</version>
        <file>${project.build.directory}/classes/example.graphql</file>
        <overwrite>true</overwrite>
      </artifact>
    </artifacts>
  </configuration>
 </execution>
</executions>
</plugin>
```

(1) 指定 **download** 作为执行目标。

(2) 使用 ./ **apis/registry/v2** 端点指定 Service Registry URL。

(3) 如果需要身份验证，您可以指定身份验证服务器和客户端凭证。

(4) 指定 Service Registry 工件组 ID。如果您不想使用唯一组，您可以指定 **默认组**。

(5) 您可以使用工件 ID 将多个工件下载到指定的目录中。

2. 构建您的 Maven 项目，例如使用 **mvn package** 命令。

其他资源

- 有关使用 Apache Maven 的详情，请查看 Apache Maven 文档。
- 有关使用 Service Registry Maven 插件的开源示例，请参阅 Apicurio Registry 演示示例。

5.3. 使用 MAVEN 插件测试模式和 API 工件

您可能希望验证工件是否可以注册，而无需实际进行任何更改。当在 Service Registry 中配置规则时，这通常很有用。如果工件内容违反了任何配置的规则，则测试工件会导致失败。

注意

当使用 Maven 插件测试工件时，即使工件通过测试，也不会将内容添加到 Service Registry 中。

先决条件

- 您已为您的客户端应用程序创建了一个 Maven 项目。如需了解更多详细信息，请参阅 Apache Maven 文档。

流程

1. 更新 Maven **pom.xml** 文件，以使用 **apicurio-registry-maven-plugin** 测试工件。以下示例显示了测试 Apache Avro 模式：

```
<plugin>
 <groupId>io.apicurio</groupId>
 <artifactId>apicurio-registry-maven-plugin</artifactId>
 <version>${apicurio.version}</version>
 <executions>
    <execution>
     <phase>generate-sources</phase>
     <goals>
        <goal>test-update</goal> (1)
     </goals>
     <configuration>
        <registryUrl>MY-REGISTRY-URL/apis/registry/v2</registryUrl> (2)
        <authServerUrl>MY-AUTH-SERVER</authServerUrl>
        <clientId>MY-CLIENT-ID</clientId>
        <clientSecret>MY-CLIENT-SECRET</clientSecret> (3)
        <clientScope>MY-CLIENT-SCOPE</clientScope>
        <artifacts>
           <artifact>
              <groupId>TestGroup</groupId> (4)
              <artifactId>FullNameRecord</artifactId>
              <file>${project.basedir}/src/main/resources/schemas/record.avsc</file> (5)
           </artifact>
        </artifacts>
     </configuration>
  </execution>
 </executions>
</plugin>
```

(1) 指定 **test-update** 作为测试 schema 工件的执行目标。

(2) 使用 ./ **apis/registry/v2** 端点指定 Service Registry URL。

(3) 如果需要身份验证，您可以指定身份验证服务器和客户端凭证。

(4) 指定 Service Registry 工件组 ID。如果您不想使用唯一组，您可以指定 **默认组**。

5 您可以使用工件 ID 测试指定目录中的多个工件。

2. 构建您的 Maven 项目，例如使用 **mvn package** 命令。

其他资源

- 有关使用 Apache Maven 的详情，请查看 Apache Maven 文档。
- 有关使用 Service Registry Maven 插件的开源示例，请参阅 Apicurio Registry 演示示例。

5.4. 使用 SERVICE REGISTRY MAVEN 插件手动添加工件引用

有些 Service Registry 工件类型可能会包括从一个 *工件文件到另一个工件引用*。您可以通过定义可重复使用的模式或 API 工件来创建效率，然后从工件引用中的多个位置引用它们。

以下工件类型支持工件引用：

- Apache Avro
- Google Protobuf
- JSON 架构
- OpenAPI
- AsyncAPI

本节展示了一个简单的示例，它使用 Service Registry Maven 插件手动注册存储在 Service Registry 中的简单 Avro schema 工件引用。本例假定已在 Service Registry 中创建了以下 **Exchange** 模式工件：

Exchange 模式

```
{
  "namespace": "com.kubetrade.schema.common",
  "type": "enum",
  "name": "Exchange",
  "symbols" : ["GEMINI"]
}
```

然后，这个示例会创建一个 **TradeKey** schema 工件，其中包含对嵌套 **Exchange** schema 工件的引用：

TradeKey 模式，带有嵌套对 Exchange schema 的引用

```
{
  "namespace": "com.kubetrade.schema.trade",
  "type": "record",
  "name": "TradeKey",
  "fields": [
    {
      "name": "exchange",
      "type": "com.kubetrade.schema.common.Exchange"
    },
    {
      "name": "key",
```

```
      "type": "string"
    }
  ]
}
```

先决条件

- 您已为您的客户端应用程序创建了一个 Maven 项目。如需了解更多详细信息，请参阅 Apache Maven 文档。
- 引用的 **Exchange** 模式工件已在 Service Registry 中创建。

流程

1. 更新您的 Maven **pom.xml** 文件，以使用 **apicurio-registry-maven-plugin** 注册 **TradeKey** 模式，其中包含对 **Exchange** 模式的嵌套引用，如下所示：

```
<plugin>
  <groupId>io.apicurio</groupId>
  <artifactId>apicurio-registry-maven-plugin</artifactId>
  <version>${apicurio-registry.version}</version>
  <executions>
    <execution>
      <phase>generate-sources</phase>
      <goals>
        <goal>register</goal> (1)
      </goals>
      <configuration>
        <registryUrl>MY-REGISTRY-URL/apis/registry/v2</registryUrl> (2)
        <authServerUrl>MY-AUTH-SERVER</authServerUrl>
        <clientId>MY-CLIENT-ID</clientId>
        <clientSecret>MY-CLIENT-SECRET</clientSecret> (3)
        <clientScope>MY-CLIENT-SCOPE</clientScope>
        <artifacts>
          <artifact>
            <groupId>test-group</groupId> (4)
            <artifactId>TradeKey</artifactId>
            <version>2.0</version>
            <type>AVRO</type>
            <file>
              ${project.basedir}/src/main/resources/schemas/TradeKey.avsc
            </file>
            <ifExists>RETURN_OR_UPDATE</ifExists>
            <canonicalize>true</canonicalize>
            <references>
              <reference> (5)
                <name>com.kubetrade.schema.common.Exchange</name>
                <groupId>test-group</groupId>
                <artifactId>Exchange</artifactId>
                <version>2.0</version>
                <type>AVRO</type>
                <file>
                  ${project.basedir}/src/main/resources/schemas/Exchange.avsc
                </file>
                <ifExists>RETURN_OR_UPDATE</ifExists>
```

```
                        <canonicalize>true</canonicalize>
                    </reference>
                </references>
            </artifact>
        </artifacts>
    </configuration>
</execution>
</executions>
</plugin>
```

1. 指定 **register** 作为执行目标，将 schema 工件上传到 Service Registry。
2. 使用 ./ **apis/registry/v2** 端点指定 Service Registry URL。
3. 如果需要身份验证，您可以指定身份验证服务器和客户端凭证。
4. 指定 Service Registry 工件组 ID。如果您不想使用唯一的组 ID，您可以指定 **默认组**。
5. 使用其组 ID、工件 ID、版本、类型和位置指定 Service Registry 工件引用。您可以以这种方式注册多个工件引用。

2. 构建您的 Maven 项目，例如使用 **mvn package** 命令。

其他资源

- 有关使用 Apache Maven 的详情，请查看 Apache Maven 文档。
- 有关使用 Service Registry Maven 插件手动注册工件引用的开源示例，请参阅 avro-maven-with-references 演示示例。
- 有关工件引用的更多示例，请参阅在 第 8 章 *在 Java 客户端中配置 Kafka serializers/deserializers* 中配置每个工件类型部分。

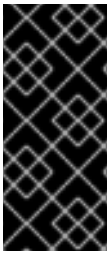
5.5. 使用 SERVICE REGISTRY MAVEN 插件自动添加工件引用

有些 Service Registry 工件类型可能会包括从一个 *工件文件到另一个工件引用*。您可以通过定义可重复使用的模式或 API 工件来创建效率，然后从工件引用中的多个位置引用它们。

以下工件类型支持工件引用：

- Apache Avro
- Google Protobuf
- JSON 架构
- OpenAPI
- AsyncAPI

您可以指定单个工件并配置 Service Registry Maven 插件，以自动检测同一目录中对工件的所有引用，并自动注册这些引用。这是一个技术预览功能。



重要

技术预览功能不被红帽产品服务级别协议(SLA)支持，且功能可能并不完善。红帽不推荐在生产环境中使用它们。这些技术预览功能可以使用户提早试用新的功能，并有机会在开发阶段提供反馈意见。

有关红帽技术预览功能支持范围的更多信息，请参阅技术预览功能支持范围。

本节展示了使用 Maven 插件注册 Avro 模式的简单示例，并自动检测并注册对简单 schema 工件引用。本例假定父 **TradeKey** 工件和嵌套 **Exchange** 模式工件在同一目录中都可用：

TradeKey 模式，带有嵌套对 Exchange schema 的引用

```
{
  "namespace": "com.kubetrade.schema.trade",
  "type": "record",
  "name": "TradeKey",
  "fields": [
    {
      "name": "exchange",
      "type": "com.kubetrade.schema.common.Exchange"
    },
    {
      "name": "key",
      "type": "string"
    }
  ]
}
```

Exchange 模式

```
{
  "namespace": "com.kubetrade.schema.common",
  "type": "enum",
  "name": "Exchange",
  "symbols" : ["GEMINI"]
}
```

先决条件

- 您已为您的客户端应用程序创建了一个 Maven 项目。如需了解更多详细信息，请参阅 Apache Maven 文档。
- **TradeKey** 模式工件和嵌套 **Exchange** 模式工件文件都位于同一目录中。

流程

1. 更新您的 Maven **pom.xml** 文件，以使用 **apicurio-registry-maven-plugin** 注册 **TradeKey** 模式，其中包含对 **Exchange** 模式的嵌套引用，如下所示：

```
<plugin>
  <groupId>io.apicurio</groupId>
  <artifactId>apicurio-registry-maven-plugin</artifactId>
  <version>${apicurio-registry.version}</version>
```

```
<executions>
  <execution>
    <phase>generate-sources</phase>
    <goals>
      <goal>register</goal> (1)
    </goals>
    <configuration>
      <registryUrl>MY-REGISTRY-URL/apis/registry/v2</registryUrl> (2)
      <authServerUrl>MY-AUTH-SERVER</authServerUrl>
      <clientId>MY-CLIENT-ID</clientId>
      <clientSecret>MY-CLIENT-SECRET</clientSecret> (3)
      <clientScope>MY-CLIENT-SCOPE</clientScope>
      <artifacts>
        <artifact>
          <groupId>test-group</groupId> (4)
          <artifactId>TradeKey</artifactId>
          <version>2.0</version>
          <type>AVRO</type>
          <file>
            ${project.basedir}/src/main/resources/schemas/TradeKey.avsc (5)
          </file>
          <ifExists>RETURN_OR_UPDATE</ifExists>
          <canonicalize>true</canonicalize>
          <autoRefs>true</autoRefs> (6)
        </artifact>
      </artifacts>
    </configuration>
  </execution>
</executions>
</plugin>
```

1. 指定 **register** 作为执行目标，将 schema 工件上传到 Service Registry。
2. 使用 ./ **apis/registry/v2** 端点指定 Service Registry URL。
3. 如果需要身份验证，您可以指定身份验证服务器和客户端凭证。
4. 指定包含引用的父工件组 ID。如果您不想使用唯一的组 ID，您可以指定 **默认组**。
5. 指定父工件文件的位置。所有引用的工件还必须位于同一目录中。
6. 将 **<autoRefs>** 选项设置为 true 来自动检测并注册同一目录中对工件的所有引用。您可以以这种方式注册多个工件引用。

2. 构建您的 Maven 项目，例如使用 **mvn package** 命令。

其他资源

- 有关使用 Apache Maven 的详情，请查看 Apache Maven 文档。
- 有关使用 Service Registry Maven 插件自动注册多个工件引用的开源示例，请参阅 avro-maven-with-references-auto 演示示例。
- 有关工件引用的更多示例，请参阅在 第 8 章 *在 Java 客户端中配置 Kafka serializers/deserializers* 中配置每个工件类型部分。

第 6 章 使用 JAVA 客户端管理 SERVICE REGISTRY 内容

您可以编写 Service Registry Java 客户端应用程序，并使用它来管理存储在 Service Registry 中的工件：

- 第 6.1 节 “Service Registry Java 客户端”
- 第 6.2 节 “编写 Service Registry Java 客户端应用程序”
- 第 6.3 节 “Service Registry Java 客户端配置”

6.1. SERVICE REGISTRY JAVA 客户端

您可以使用 Java 客户端应用程序管理 Service Registry 中存储的工件。您可以使用 Service Registry Java 客户端类创建、读取、更新或删除工件。您还可以使用 Service Registry Java 客户端来执行管理员功能，如管理全局规则或导入和导出 Service Registry 数据。

您可以通过在 Apache Maven 项目中添加正确的依赖项来访问 Service Registry Java 客户端。如需了解更多详细信息，请参阅 第 6.2 节 “编写 Service Registry Java 客户端应用程序”。

Service Registry 客户端使用 JDK 提供的 HTTP 客户端实现，您可以根据需要进行自定义。例如，您可以添加自定义标头或启用传输层安全(TLS)身份验证的配置选项。如需了解更多详细信息，请参阅 第 6.3 节 “Service Registry Java 客户端配置”。

6.2. 编写 SERVICE REGISTRY JAVA 客户端应用程序

您可以使用 Service Registry Java 客户端类编写 Java 客户端应用程序来管理 Service Registry 中存储的工件。

先决条件

- Service Registry 已安装并在您的环境中运行。
- 您已为您的 Java 客户端应用程序创建了 Maven 项目。如需了解更多详细信息，请参阅 Apache Maven。

流程

1. 在您的 Maven 项目中添加以下依赖项：

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

2. 创建 Service Registry 客户端，如下所示：

```
public class ClientExample {

    public static void main(String[] args) throws Exception {
        // Create a registry client
        String registryUrl = "https://my-registry.my-domain.com/apis/registry/v2"; (1)
```

```
        RegistryClient client = RegistryClientFactory.create(registryUrl); 2
    }
}
```

1 如果您指定了 **https://my-registry.my-domain.com** 的示例 Service Registry URL，客户端会自动附加 **/apis/registry/v2**。

2 有关创建 Service Registry 客户端时的更多选项，请参阅下一节中的 Java 客户端配置。

创建客户端时，您可以使用客户端中的 Service Registry REST API 中提供的所有操作。如需了解更多详细信息，请参阅 Apicurio Registry REST API 文档。

其他资源

- 有关如何使用和自定义 Service Registry 客户端的开源示例，请参阅 Apicurio Registry REST 客户端演示。
- 有关如何在生成者和消费者应用程序中使用 Service Registry Kafka 客户端序列化器(SerDes)的详情，请参考 *第 7 章 在 Java 客户端中使用序列化器/反序列化器验证 Kafka 信息*。

6.3. SERVICE REGISTRY JAVA 客户端配置

Service Registry Java 客户端包括以下配置选项，具体取决于客户端工厂：

表 6.1. Service Registry Java 客户端配置选项

选项	描述	参数
普通客户端	用于与正在运行的 Service Registry 实例交互的基本 REST 客户端。	**baseUrl**
带有自定义配置的客户端	使用用户提供的配置的服务 Registry 客户端。	**baseurl , Map<String Object> 配置**
带有自定义配置和身份验证的客户端	接受包含自定义配置的映射的 Service Registry 客户端。例如，这对于为调用添加自定义标头非常有用。您还必须提供身份验证服务器来验证请求。	**baseurl, Map<String Object> 配置, Auth auth**

自定义标头配置

要配置自定义标头，您必须将 **apicurio.registry.request.headers** 前缀添加到 **configs** map 键中。例如，**配置**映射键为 **apicurio.registry.request.headers.Authorization**，值为 **Basic: YWxhZGRpbjpvcGVuc2VzYW1**，将 **Authorization** 标头设置为相同的值。

TLS 配置选项

您可以使用以下属性为 Service Registry Java 客户端配置传输层安全(TLS)身份验证：

- **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**

其他资源

- 有关如何为 Service Registry Kafka 客户端序列化器/反序列化器(SerDes)配置身份验证的详情，请参考 *第 7 章 在 Java 客户端中使用序列化器/反序列化器验证 Kafka 信息*。

第 7 章 在 JAVA 客户端中使用序列化器/反序列化器验证 KAFKA 信息

Service Registry 为使用 Java 编写的 Kafka 生成者和消费者应用程序提供客户端序列化器/反序列化器(SerDes)。Kafka producer 应用程序使用序列化器来对符合特定事件模式的信息进行编码。Kafka 消费者应用程序使用 deserializers 来验证消息是否根据特定的模式 ID 使用正确的模式来序列化消息。这样可确保一致的架构使用，并有助于防止运行时出现数据错误。

本章解释了如何在生成者和消费者客户端应用程序中使用 Kafka 客户端 SerDe ：

- 第 7.1 节 “Kafka 客户端应用程序和 Service Registry”
- 第 7.2 节 “在 Service Registry 中查找模式的策略”
- 第 7.3 节 “在 Service Registry 中注册 schema”
- 第 7.4 节 “使用 Kafka 消费者客户端中的 schema”
- 第 7.5 节 “使用 Kafka producer 客户端中的 schema”
- 第 7.6 节 “使用 Kafka Streams 应用程序中的 schema”

先决条件

- 您已阅了读 第 1 章 *Service Registry 简介*。
- 已安装 Service Registry。
- 您已创建了 Kafka producer 和消费者客户端应用程序。
  有关 Kafka 客户端应用程序的详情，请参阅在 OpenShift 中部署和管理 AMQ Streams。

7.1. KAFKA 客户端应用程序和 SERVICE REGISTRY

Service Registry 将模式管理与客户端应用程序配置分离。您可以通过在客户端代码中指定 URL 来启用 Java 客户端应用程序使用来自 Service Registry 的 schema。

您可以在 Service Registry 中存储模式以序列化和反序列化消息，这些消息从客户端应用程序引用，以确保它们发送和接收的消息与这些模式兼容。Kafka 客户端应用程序可以在运行时从 Service Registry 中推送或拉取其模式。

模式可以演进，您可以在 Service Registry 中定义规则，例如，确保架构更改有效，且不会破坏应用程序使用的早期版本。Service Registry 通过将修改的模式与以前的模式进行比较来检查兼容性。

Service Registry 模式技术

Service Registry 为 schema 技术提供模式 registry 支持，例如：

- Avro
- protobuf
- JSON 架构

客户端应用程序可通过 Service Registry 提供的 Kafka 客户端序列化器(SerDes)服务使用这些模式技术。Service Registry 提供的 SerDes 类的成熟度和使用可能会有所不同。以下章节提供有关每个 schema 类型的更多详情。

producer 模式配置
制作者客户端应用程序使用序列化器将它发送到特定代理主题的消息放在正确的数据格式中。

启用制作者以使用 Service Registry 进行序列化：

- 使用 Service Registry 定义并注册您的模式 （如果还没有存在）。
- 使用以下命令配置制作者 客户端代码：
  - Service Registry 的 URL
  - 用于消息的 Service Registry serializer
  - 策略将 Kafka 信息映射到 Service Registry 中的 schema 工件
  - 在 Service Registry 中查找或注册用于序列化的模式的策略

注册 schema 后，当启动 Kafka 和 Service Registry 时，您可以访问 schema 来格式化由制作者发送到 Kafka 代理主题的信息。或者，根据配置，生成者可以在第一次使用时自动注册 schema。

如果 schema 已存在，您可以根据 Service Registry 中定义的兼容性规则，使用 registry REST API 创建新版本。版本用于兼容性检查，作为 schema 的演进。组 ID、工件 ID 和 version 代表用于标识 schema 的唯一元组。

consumer 模式配置
消费者客户端应用程序使用反序列化器将来自特定代理主题的消息获取到正确的数据格式。

启用消费者使用 Service Registry 进行反序列化：

- 使用 Service Registry 定义并注册您的模式 （如果尚不存在）
- 使用以下命令配置消费者客户端代码 ：
  - Service Registry 的 URL
  - 用于消息的 Service Registry deserializer
  - 为 deserialization 输入数据流

使用全局 ID 检索模式

默认情况下，架构通过使用全局 ID （在被消耗的消息中指定）从 Service Registry 检索。架构全局 ID 可以位于消息标头或消息有效负载中，具体取决于制作者应用的配置。

当在消息有效负载中查找全局 ID 时，数据格式以 magic 字节开头，用作消费者的信号，后跟全局 ID，以及消息数据正常。例如：

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

然后，当启动 Kafka 和 Service Registry 时，您可以访问模式来格式化从 Kafka 代理主题接收的消息。

使用内容 ID 检索模式

或者，您可以将 配置为根据内容 ID 从 Service Registry 检索模式，这是工件内容的唯一 ID。全局 ID 是工件版本的唯一 ID。

内容 ID 不唯一标识版本，而是仅唯一标识版本内容。如果多个版本共享相同的内容，它们具有不同的全局 ID，但具有相同的内容 ID。confluent Schema Registry 默认使用内容 ID。

7.2. 在 SERVICE REGISTRY 中查找模式的策略

Kafka 客户端序列化器使用 *lookup 策略来* 决定在 Service Registry 中注册消息模式的工件 ID 和全局 ID。对于给定主题和消息，您可以使用 **ArtifactReferenceResolverStrategy** Java 接口的不同实现来返回对 registry 中工件的引用。

每个策略的类位于 **io.apicurio.registry.serde.strategy** 软件包中。Avro SerDes 的特定策略类位于 **io.apicurio.registry.serde.avro.strategy 软件包中**。默认策略是 **TopicIdStrategy**，它查找名称与接收消息的 Kafka 主题相同的 Service Registry 工件。

示例

```
public ArtifactReference artifactReference(String topic, boolean isKey, T schema) {
    return ArtifactReference.builder()
            .groupId(null)
            .artifactId(String.format("%s-%s", topic, isKey ? "key" : "value"))
            .build();
```

- **topic** 参数是接收消息的 Kafka 主题的名称。
- 当 message 键被序列化时，**isKey** 参数为 **true**，当 message 值被序列化时为 **false**。
- **schema** 参数是消息序列化或反序列化的 schema。
- **ArtifactReference** 返回包含注册 schema 的工件 ID。

您使用的查找策略取决于您如何和存储模式。例如，如果您具有相同 Avro 消息类型的不同 Kafka 主题，您可以使用使用 *记录 ID* 的策略。

工件解析器策略

工件解析器策略提供了一种将 Kafka 主题和消息信息映射到 Service Registry 中的工件的方法。映射的常见约定是将 Kafka 主题名称与 **键或 值** 组合，具体取决于是否将序列化器用于 Kafka 消息键或值。

但是，您可以使用 Service Registry 提供的策略或创建一个实现 **io.apicurio.registry.serde.strategy.ArtifactReferenceResolverStrategy** 的自定义 Java 类来对映射使用替代惯例。

返回对工件的引用的策略

Service Registry 提供以下策略，根据 **ArtifactReferenceResolverStrategy** 的实现返回对工件的引用：

RecordIdStrategy

使用模式全名的 avro 特定策略。

TopicRecordIdStrategy

特定于 avro 的策略，它使用主题名称和模式的完整名称。

TopicIdStrategy

默认策略，它使用主题名和 **key** 或 **value** 后缀。

SimpleTopicIdStrategy

仅使用主题名称的简单策略。

DefaultSchemaResolver 接口

默认 schema 解析器会找到并标识在由工件解析器策略提供的工件引用下注册的 schema 的特定版本。每个工件的每个版本都有一个全局唯一标识符，可用于检索该工件的内容。这个全局 ID 都包含在每个 Kafka 信息中，以便反序列化器可以正确地从 Apicurio Registry 获取 schema。

默认模式解析器可以查找现有的工件版本，如果未找到，它可以根据使用哪个策略进行注册。您还可以通过创建一个实现 **io.apicurio.registry.resolver.SchemaResolver** 的自定义 Java 类来提供自己的策略。但是，建议您使用 **DefaultSchemaResolver** 并指定配置属性。

registry 查找选项的配置

使用 **DefaultSchemaResolver** 时，您可以使用应用属性配置其行为。下表显示了一些常用的示例：

表 7.1. Service Registry 查找配置选项

属性	类型	描述	默认
apicurio.registry.find-latest	**布尔值**	指定 serializer 是否尝试在 registry 中查找对应组 ID 和工件 ID 的最新工件。	**false**
apicurio.registry.use-id	**字符串**	指示序列化器将指定的 ID 写入 Kafka，并指示反序列化器使用此 ID 来查找 schema。	None
apicurio.registry.auto-register	**布尔值**	指定 serializer 是否尝试在 registry 中创建工件。JSON 架构序列化器不支持此功能。	**false**
apicurio.registry.check-period-ms	**字符串**	指定缓存全局 ID 以毫秒为单位的时间。如果没有配置，则会每次获取全局 ID。	None

7.3. 在 SERVICE REGISTRY 中注册 SCHEMA

在以适当的格式（如 Apache Avro）定义了模式后，您可以将模式添加到 Service Registry。

您可以使用以下方法添加模式：

- Service Registry web 控制台
- 使用 Service Registry REST API 的 curl 命令
- Service Registry 提供的 Maven 插件
- 模式配置添加到客户端代码中

在注册了 schema 前，客户端应用程序无法使用 Service Registry。

Service Registry web 控制台

安装 Service Registry 后，您可以从 **ui** 端点连接到 Web 控制台：

http://MY-REGISTRY-URL/ui

在控制台中，您可以添加、查看和配置模式。您还可以创建阻止将无效内容添加到 registry 的规则。

curl 命令示例

```
curl -X POST -H "Content-type: application/json; artifactType=AVRO" \
  -H "X-Registry-ArtifactId: share-price" \ 1
  --data '{
    "type":"record",
    "name":"price",
    "namespace":"com.example",
    "fields":[{"name":"symbol","type":"string"},
    {"name":"price","type":"string"}]}'
  https://my-cluster-my-registry-my-project.example.com/apis/registry/v2/groups/my-group/artifacts -s
2
```

1 简单的 Avro 模式工件。

2 公开 Service Registry 的 OpenShift 路由名称。

Maven 插件示例

```
<plugin>
  <groupId>io.apicurio</groupId>
  <artifactId>apicurio-registry-maven-plugin</artifactId>
  <version>${apicurio.version}</version>
  <executions>
     <execution>
       <phase>generate-sources</phase>
       <goals>
          <goal>register</goal> 1
       </goals>
       <configuration>
          <registryUrl>http://REGISTRY-URL/apis/registry/v2</registryUrl> 2
          <artifacts>
             <artifact>
                <groupId>TestGroup</groupId> 3
                <artifactId>FullNameRecord</artifactId>
                <file>${project.basedir}/src/main/resources/schemas/record.avsc</file>
                <ifExists>FAIL</ifExists>
             </artifact>
             <artifact>
                <groupId>TestGroup</groupId>
                <artifactId>ExampleAPI</artifactId> 4
                <type>GRAPHQL</type>
                <file>${project.basedir}/src/main/resources/apis/example.graphql</file>
                <ifExists>RETURN_OR_UPDATE</ifExists>
                <canonicalize>true</canonicalize>
             </artifact>
          </artifacts>
       </configuration>
    </execution>
  </executions>
</plugin>
```

1 指定 **register** 作为执行目标，将 schema 工件上传到 registry。

2 使用 ./ **apis/registry/v2** 端点指定 Service Registry URL。

3 指定 Service Registry 工件组 ID。

4 您可以使用指定的组 ID、工件 ID 和位置上传多个工件。

使用制作者客户端示例进行配置

```
String registryUrl_node1 = PropertiesUtil.property(clientProperties, "registry.url.node1",
    "https://my-cluster-service-registry-myproject.example.com/apis/registry/v2"); 1
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(
            "AVRO",
            artifactId,
            new ByteArrayInputStream(LogInput.SCHEMA$.toString().getBytes())
        );
        csa.toCompletableFuture().get();
    }
}
```

1 您可以对多个 URL 节点注册属性。

2 检查模式是否已基于工件 ID。

7.4. 使用 KAFKA 消费者客户端中的 SCHEMA

此流程描述了如何配置使用 Java 编写的 Kafka 消费者客户端，以使用 Service Registry 中的模式。

先决条件

- 已安装 Service Registry
- 模式使用 Service Registry 注册

流程

1. 使用 Service Registry 的 URL 配置客户端。例如：

```
String registryUrl = "https://registry.example.com/apis/registry/v2";
Properties props = new Properties();
props.putIfAbsent(SerdeConfig.REGISTRY_URL, registryUrl);
```

2. 使用 Service Registry deserializer 配置客户端。例如：

```
// Configure Kafka settings
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");
// Configure deserializer settings
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 提供的反序列化器。
2. 反序列化采用 Apache Avro JSON 格式。

7.5. 使用 KAFKA PRODUCER 客户端中的 SCHEMA

此流程描述了如何配置使用 Java 编写的 Kafka producer 客户端，以使用 Service Registry 中的 schema。

先决条件

- 已安装 Service Registry
- 模式使用 Service Registry 注册

流程

1. 使用 Service Registry 的 URL 配置客户端。例如：

```
String registryUrl = "https://registry.example.com/apis/registry/v2";
Properties props = new Properties();
props.putIfAbsent(SerdeConfig.REGISTRY_URL, registryUrl);
```

2. 使用 serializer 配置客户端，并在 Service Registry 中查找架构。例如：

```
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(SerdeConfig.FIND_LATEST_ARTIFACT, Boolean.TRUE); 3
```

1. Service Registry 提供的消息键的序列化器。
2. Service Registry 提供的 message 值的序列化器。
3. 查找策略，以查找该模式的全局 ID。

7.6. 使用 KAFKA STREAMS 应用程序中的 SCHEMA

此流程描述了如何配置使用 Java 编写的 Kafka Streams 客户端，以使用来自 Service Registry 的 Apache Avro 模式。

先决条件

- 已安装 Service Registry
- 模式使用 Service Registry 注册

流程

1. 使用 Service Registry URL 创建并配置 Java 客户端：

```
String registryUrl = "https://registry.example.com/apis/registry/v2";

RegistryService client = RegistryClient.cached(registryUrl);
```

2. 配置 serializer 和 deserializer：

```
Serializer<LogInput> serializer = new AvroKafkaSerializer<LogInput>(); (1)

Deserializer<LogInput> deserializer = new AvroKafkaDeserializer <LogInput>(); (2)

Serde<LogInput> logSerde = Serdes.serdeFrom(
    serializer,
    deserializer
);

Map<String, Object> config = new HashMap<>();
config.put(SerdeConfig.REGISTRY_URL, registryUrl);
config.put(AvroKafkaSerdeConfig.USE_SPECIFIC_AVRO_READER, true);
logSerde.configure(config, false); (3)
```

(1) Service Registry 提供的 Avro serializer。

(2) Service Registry 提供的 Avro deserializer。

(3) 配置 Service Registry URL 和 Avro reader，以 Avro 格式进行 deserialization。

3. 创建 Kafka Streams 客户端：

```
KStream<String, LogInput> input = builder.stream(
    INPUT_TOPIC,
    Consumed.with(Serdes.String(), logSerde)
);
```

第 8 章 在 JAVA 客户端中配置 KAFKA SERIALIZERS/DESERIALIZERS

本章详细介绍了如何在生成者和消费者 Java 客户端应用程序中配置 Kafka serializers/deserializers (SerDes)：

- 第 8.1 节 “在客户端应用程序中的 Service Registry serializer/deserializer 配置”
- 第 8.2 节 “Service Registry serializer/deserializer 配置属性”
- 第 8.3 节 “如何配置不同的客户端序列化器/反序列化器类型”
- 第 8.3.1 节 “使用 Service Registry 配置 Avro SerDe”
- 第 8.3.2 节 “使用 Service Registry 配置 JSON 架构 SerDe”
- 第 8.3.3 节 “使用 Service Registry 配置 Protobuf SerDes”

先决条件

- 您已阅读了 *第 7 章 在 Java 客户端中使用序列化器/反序列化器验证 Kafka 信息*。

8.1. 在客户端应用程序中的 SERVICE REGISTRY SERIALIZER/DESERIALIZER 配置

您可以使用本节中显示的示例常量直接在客户端应用中配置特定的客户端序列化器/反序列化器(SerDes)服务和模式查找策略。另外，您可以在文件或实例中配置对应的 Service Registry 应用程序属性。

以下小节展示了常用的 SerDes constants 和 configuration options 的示例。

配置 SerDes 服务

```
public class SerdeConfig {

  public static final String REGISTRY_URL = "apicurio.registry.url"; (1)
  public static final String ID_HANDLER = "apicurio.registry.id-handler"; (2)
  public static final String ENABLE_CONFLUENT_ID_HANDLER = "apicurio.registry.as-confluent";
(3)
```

1. Service Registry 所需的 URL。
2. 扩展 ID 处理以支持其他 ID 格式，并将它们与 Service Registry SerDes 服务兼容。例如，将默认 ID 格式从 **Long** 改为 **Integer** 支持 Confluent ID 格式。
3. 简化 Confluent ID 的处理。如果设置为 **true**，则使用 **Integer** 用于全局 ID 查找。设置不应与 **ID_HANDLER** 选项一起使用。

其他资源

- 有关配置选项的详情，请查看 第 8.2 节 “Service Registry serializer/deserializer 配置属性”

配置 SerDes 查找策略

```
public class SerdeConfig {
```

```
  public static final String ARTIFACT_RESOLVER_STRATEGY = "apicurio.registry.artifact-resolver-strategy"; 1
  public static final String SCHEMA_RESOLVER = "apicurio.registry.schema-resolver"; 2
...
```

1 1 实现工件解析器策略和映射的 Java 类，并在 Kafka SerDes 和工件 ID 之间映射。默认为主题 ID 策略。这仅由 serializer 类使用。

2 2 实施架构解析器的 Java 类。默认为 **DefaultSchemaResolver**。这供 serializer 和 deserializer 类使用。

其他资源

- 有关查找策略的详情，请查看 *第 7 章 在 Java 客户端中使用序列化器/反序列化器验证 Kafka 信息*
- 有关配置选项的详情，请查看 第 8.2 节 “Service Registry serializer/deserializer 配置属性”

配置 Kafka 转换器

```
public class SerdeBasedConverter<S, T> extends SchemaResolverConfigurer<S, T> implements Converter, Closeable {

  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. 用于 Service Registry Kafka 转换所需的序列化器。
2. 用于 Service Registry Kafka 转换所需的反序列化器。

其他资源

- 如需了解更多详细信息，请参阅 SerdeBasedConverter Java 类

配置不同模式类型

有关如何为不同的模式技术配置 SerDes 的详情，请参考：

- 第 8.3.1 节 “使用 Service Registry 配置 Avro SerDe”
- 第 8.3.2 节 “使用 Service Registry 配置 JSON 架构 SerDe”
- 第 8.3.3 节 “使用 Service Registry 配置 Protobuf SerDes”

8.2. SERVICE REGISTRY SERIALIZER/DESERIALIZER 配置属性

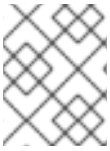
本节提供有关 Service Registry Kafka serializers/deserializers (SerDes)的 Java 配置属性的参考信息。

SchemaResolver 接口

Service Registry SerDes 基于 **SchemaResolver** 接口，它抽象了对 registry 的访问，并为所有支持的格式的 SerDes 类应用相同的查找逻辑。

表 8.1. SchemaResolver 接口的配置属性

常数	属性	描述	类型	默认
SCHEMA_RESOLVER	**apicurio.registry.schema-resolver**	由 serializers 和 deserializers 使用。实现 **SchemaResolver** 的完全限定 Java 类名称。	字符串	**io.apicurio.registry.resolver.DefaultSchemaResolver**



注意

建议使用 **DefaultSchemaResolver**，并为大多数用例提供有用的功能。对于某些高级用例，您可以使用 **SchemaResolver** 的自定义实现。

DefaultSchemaResolver 类

您可以使用 **DefaultSchemaResolver** 来配置功能，例如：

- 访问 registry API
- 如何在 registry 中查找工件
- 如何从 Kafka 写入和读取工件信息
- 反序列化器的回退选项

配置 registry API 访问选项

DefaultSchemaResolver 提供以下属性来配置对核心 registry API 的访问：

表 8.2. 访问 registry API 的配置属性

常数	属性	描述	类型	默认
REGISTRY_URL	**apicurio.registry.url**	由 serializers 和 deserializers 使用。用于访问 registry API 的 URL。	**字符串**	None
AUTH_SERVICE_URL	**apicurio.auth.service.url**	由 serializers 和 deserializers 使用。身份验证服务的 URL。使用 OAuth 客户端凭证流访问安全 registry 时需要此项。	**字符串**	None
AUTH_TOKEN_ENDPOINT	**apicurio.auth.service.token.endpoint**	由 serializers 和 deserializers 使用。令牌端点的 URL。没有指定安全 registry 和 **AUTH_SERVICE_URL** 时需要此项。	**字符串**	None

常数	属性	描述	类型	默认
AUTH_REALM	**apicurio.auth.realm**	由 serializers 和 deserializers 使用。realm 以访问身份验证服务。使用 OAuth 客户端凭证流访问安全 registry 时需要此项。	**字符串**	None
AUTH_CLIENT_ID	**apicurio.auth.client.id**	由 serializers 和 deserializers 使用。用于访问身份验证服务的客户端 ID。使用 OAuth 客户端凭证流访问安全 registry 时需要此项。	**字符串**	None
AUTH_CLIENT_SECRET	**apicurio.auth.client.secret**	由 serializers 和 deserializers 使用。用于访问身份验证服务的客户端机密。使用 OAuth 客户端凭证流访问安全 registry 时需要此项。	**字符串**	None
AUTH_USERNAME	**apicurio.auth.username**	由 serializers 和 deserializers 使用。访问 registry 的用户名。使用 HTTP 基本身份验证访问安全 registry 时需要此项。	**字符串**	None
AUTH_PASSWORD	**apicurio.auth.password**	由 serializers 和 deserializers 使用。访问 registry 的密码。使用 HTTP 基本身份验证访问安全 registry 时需要此项。	**字符串**	None

registry 查找选项的配置

DefaultSchemaResolver 使用以下属性来配置如何在 Service Registry 中查找工件。

表 8.3. registry 工件查找的配置属性

常数	属性	描述	类型	默认

常数	属性	描述	类型	默认
ARTIFACT_RESOLVER_STRATEGY	**apicurio.registry.artifact-resolver-strategy**	仅供 serializers 使用。实现 **ArtifactReferenceResolverStrategy** 的完全限定 Java 类名称，并将每个 Kafka 消息映射到 **ArtifactReference**(**groupId**,**artifactId**, 和 version)。例如，默认策略使用主题名称作为 schema **artifactId**。	**字符串**	**io.apicurio.registry.serde.strategy.TopicIdStrategy**
EXPLICIT_ARTIFACT_GROUP_ID	**apicurio.registry.artifact.group-id**	仅供 serializers 使用。设置用于查询或创建工件的 **groupId**。覆盖 **ArtifactResolverStrategy** 返回的 **groupId**。	**字符串**	None
EXPLICIT_ARTIFACT_ID	**apicurio.registry.artifact.artifact-id**	仅供 serializers 使用。设置用于查询或创建工件的 **artifactId**。覆盖 **ArtifactResolverStrategy** 返回的 **artifactId**。	**字符串**	None
EXPLICIT_ARTIFACT_VERSION	**apicurio.registry.artifact.version**	仅供 serializers 使用。设置用于查询或创建工件的工件版本。覆盖 **ArtifactResolverStrategy** 返回的版本。	**字符串**	None
FIND_LATEST_ARTIFACT	**apicurio.registry.find-latest**	仅供 serializers 使用。指定 serializer 是否尝试在 registry 中查找对应组 ID 和工件 ID 的最新工件。	**布尔值**	**false**
AUTO_REGISTER_ARTIFACT	**apicurio.registry.auto-register**	仅供 serializers 使用。指定 serializer 是否尝试在 registry 中创建工件。JSON 架构序列化器不支持此功能。	**布尔值**, **布尔值字符串**	**false**

常数	属性	描述	类型	默认
AUTO_REGISTER_ARTIFACT_IF_EXISTS	**apicurio.registry.auto-register.if-exists**	仅供 serializers 使用。当存在冲突创建工件时，配置客户端的行为，因为工件已存在。可用值为 **FAIL**、**UPDATE**、**RETURN** 或 **RETURN_OR_UPDATE**。	**字符串**	**RETURN_OR_UPDATE**
CHECK_PERIOD_MS	**apicurio.registry.check-period-ms**	由 serializers 和 deserializers 使用。指定在自动驱除前缓存工件的时长（毫秒）。如果设置为零，则每次获取工件。	**java.time.Duration**、非负号或整数字符串	**30000**
RETRY_BACKOFF_MS	**apicurio.registry.retry-backoff-ms**	由 serializers 和 deserializers 使用。如果无法从 Registry 检索模式，它可以重试多次。此配置选项控制重试尝试之间的延迟（毫秒）。	**java.time.Duration**、非负号或整数字符串	**300**
RETRY_COUNT	**apicurio.registry.retry-count**	由 serializers 和 deserializers 使用。如果无法从 Registry 检索模式，它可以重试多次。这个配置选项控制重试尝试的数量。	非负数或整数字符串	**3**
USE_ID	**apicurio.registry.use-id**	由 serializers 和 deserializers 使用。配置以使用指定的 **IdOption** 作为工件的标识符。选项为 **globalId** 和 **contentId**。指示序列化器将指定的 ID 写入 Kafka，并指示反序列化器使用此 ID 来查找 schema。	**字符串**	**globalId**

配置 Kafka 中的读/写 registry 工件
DefaultSchemaResolver 使用以下属性来配置工件信息如何写入并从 Kafka 读取。

表 8.4. Kafka 中读/写工件信息的配置属性

常数	属性	描述	类型	默认
ENABLE_HEADERS	**apicurio.registry.headers.enabled**	由 serializers 和 deserializers 使用。将工件标识符配置为 Kafka 消息标头，而不是在消息有效负载中。	**布尔值**	**true**
HEADERS_HANDLER	**apicurio.registry.headers.handler**	由 serializers 和 deserializers 使用。实现 **HeadersHandler** 以及到/从 Kafka 消息标头写入/读取工件标识的完全限定的 Java 类名称，。	**字符串**	**io.apicurio.registry.serde.headers.DefaultHeadersHandler**
ID_HANDLER	**apicurio.registry.id-handler**	由 serializers 和 deserializers 使用。实现 **IdHandler** 的类的完全限定域名，并将工件标识符写入/从消息有效负载写入/读取。仅在 **apicurio.registry.headers.enabled** 设为 **false** 时使用。	**字符串**	**io.apicurio.registry.serde.DefaultIdHandler**
ENABLE_CONFLUENT_ID_HANDLER	**apicurio.registry.as-confluent**	由 serializers 和 deserializers 使用。启用旧的、与 Confluent 兼容的 **IdHandler** 实施的快捷方式。仅在 **apicurio.registry.headers.enabled** 设为 **false** 时使用。	**布尔值**	**true**

配置反序列化器回退选项
DefaultSchemaResolver 使用下列属性来为所有反序列化器配置回退提供程序。

表 8.5. deserializer fall-back 供应商的配置属性

常数	属性	描述	类型	默认

常数	属性	描述	类型	默认
FALLBACK_ARTIFACT_PROVIDER	**apicurio.registry.fallback.provider**	仅供反序列化器使用。设置 **FallbackArtifactProvider** 的自定义实现，以解析用于 deserialization 的工件。**FallbackArtifactProvider** 配置回退工件，以便在查找失败时从 registry 获取。	**字符串**	**io.apicurio.registry.serde.fallback.DefaultFallbackArtifactProvider**

DefaultFallbackArtifactProvider 使用以下属性来配置反序列化器回退选项：

表 8.6. deserializer fall-back 选项**的配置属性**

常数	属性	描述	类型	默认
FALLBACK_ARTIFACT_ID	**apicurio.registry.fallback.artifact-id**	仅供反序列化器使用。设置用于解析用于反序列化的工件的 **artifactId**。	**字符串**	None
FALLBACK_ARTIFACT_GROUP_ID	**apicurio.registry.fallback.group-id**	仅供反序列化器使用。设置用作回退的 **groupId**，用于解析用于反序列化的组。	**字符串**	None
FALLBACK_ARTIFACT_VERSION	**apicurio.registry.fallback.version**	仅供反序列化器使用。设置用作回退版本，用于解析用于反序列化的工件。	**字符串**	None

其他资源

- 如需了解更多详细信息，请参阅 SerdeConfig Java 类。
- 您可以将应用程序属性配置为 Java 系统属性，或者在 Quarkus **application.properties** 文件中包括它们。如需了解更多详细信息，请参阅 Quarkus 文档。

8.3. 如何配置不同的客户端序列化器/反序列化器类型

当在 Kafka 客户端应用程序中使用 schema 时，您必须根据您的用例选择要使用的特定模式类型。Service Registry 为 Apache Avro、JSON Schema 和 Google Protobuf 提供 SerDe Java 类。以下小节解释了如何将 Kafka 应用程序配置为使用每种类型。

您还可以使用 Kafka 实现自定义序列化器和反序列化器类，并使用 Service Registry REST Java 客户端利用 Service Registry 功能。

serializers/deserializers 的 Kafka 应用程序配置

使用 Kafka 应用程序中的 Service Registry 提供的 SerDe 类涉及设置正确的配置属性。以下简单的 Avro 示例演示了如何在 Kafka producer 应用程序中配置序列化器以及如何在 Kafka 消费者应用程序中配置反序列化器。

Kafka producer 中的序列化器配置示例

```
// Create the Kafka producer
private static Producer<Object, Object> createKafkaProducer() {
    Properties props = new Properties();

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

    // Use Service Registry-provided Kafka serializer for Avro
    props.putIfAbsent(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
StringSerializer.class.getName());
    props.putIfAbsent(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
AvroKafkaSerializer.class.getName());

    // Configure the Service Registry location
    props.putIfAbsent(SerdeConfig.REGISTRY_URL, REGISTRY_URL);

    // Register the schema artifact if not found in the registry.
    props.putIfAbsent(SerdeConfig.AUTO_REGISTER_ARTIFACT, Boolean.TRUE);

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

Kafka consumer 中的 deserializer 配置示例

```
// Create the Kafka consumer
private static KafkaConsumer<Long, GenericRecord> createKafkaConsumer() {
    Properties props = new Properties();

    // Configure standard Kafka settings
    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");

    // Use Service Registry-provided Kafka deserializer for Avro
    props.putIfAbsent(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
StringDeserializer.class.getName());
    props.putIfAbsent(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
AvroKafkaDeserializer.class.getName());

    // Configure the Service Registry location
    props.putIfAbsent(SerdeConfig.REGISTRY_URL, REGISTRY_URL);

    // No other configuration needed because the schema globalId the deserializer uses is sent
```

```
    // in the payload. The deserializer extracts the 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;
}
```

其他资源

- 有关示例应用程序，请参阅 Simple Avro 示例

8.3.1. 使用 Service Registry 配置 Avro SerDe

本节解释了如何对 Apache Avro 使用 Kafka 客户端序列化器和反序列化器(SerDes)类。

Service Registry 为 Avro 提供以下 Kafka 客户端 SerDes 类：

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

配置 Avro serializer

您可以使用以下方法配置 Avro serializer 类：

- Service Registry URL
- 工件解析器策略
- ID 位置
- ID 编码
- avro datum 供应商
- avro 编码

ID 位置

serializer 将 schema 的唯一 ID 作为 Kafka 消息的一部分传递，以便消费者可以使用正确的模式进行反序列化。ID 可以在消息有效负载或消息标头中。默认位置是消息有效负载。要在消息标头中发送 ID，请设置以下配置属性：

```
props.putIfAbsent(SerdeConfig.ENABLE_HEADERS, "true")
```

属性名称为 **apicurio.registry.headers.enabled**。

ID 编码

您可以在 Kafka 消息正文中传递模式 ID 时对其进行编码。将 **apicurio.registry.id-handler** 配置属性设置为实施 **io.apicurio.registry.serde.IdHandler** 接口的类。Service Registry 提供以下实现：

- **io.apicurio.registry.serde.DefaultIdHandler**: 将 ID 存储为 8 字节长
- **io.apicurio.registry.serde.Legacy4ByteIdHandler**: 将 ID 存储为 4 字节整数

Service Registry 代表模式 ID，但由于旧原因，或者由于与其他 registry 或 SerDe 类兼容，您可能需要在发送 ID 时使用 4 字节。

avro datum 供应商

avro 提供不同的 datum writers 和 readers 来写入和读取数据。Service Registry 支持三种不同类型的：

- generic
- specific
- reflect

Service Registry **AvroDatumProvider** 是使用哪个类型的抽象，默认为使用 **DefaultAvroDatumProvider**。

您可以设置以下配置选项：

- **apicurio.registry.avro-datum-provider** ：指定 **AvroDatumProvider** 实现的完全限定 Java 类名称，如 **io.apicurio.registry.serde.avro.ReflectAvroDatumProvider**
- **apicurio.registry.use-specific-avro-reader**: 设置为 **true**，在使用 **DefaultAvroDatumProvider**时使用特定类型的

avro 编码

当使用 Avro 来序列化数据时，您可以使用 Avro 二进制编码格式来确保数据以尽可能有效的格式进行编码。avro 还支持将数据编码为 JSON，这有助于检查每个消息的有效负载，如日志记录或调试。

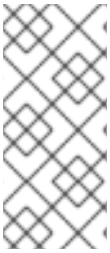
您可以通过将 **apicurio.registry.avro.encoding** 属性配置为 **JSON** 或 **BINARY**来设置 Avro 编码。默认值为 **BINARY**。

配置 Avro deserializer

您必须配置 Avro deserializer 类，以匹配序列化器的以下配置设置：

- Service Registry URL
- ID 编码
- avro datum 供应商
- avro 编码

有关这些配置选项，请参阅 serializer 部分。属性名称和值相同。



注意

配置反序列化器时不需要以下选项：

- 工件解析器策略
- ID 位置

deserializer 类可以从消息确定这些选项的值。该策略不是必需的，因为 serializer 负责发送 ID 作为消息的一部分。

ID 位置是通过在消息有效负载开始时检查 magic 字节来确定的。如果找到该字节，则使用配置的处理程序从消息有效负载中读取该 ID。如果没有找到 magic 字节，则从消息标头中读取 ID。

avro SerDes 和工件引用

当使用 Avro 消息和带有嵌套记录的模式时，会为每个嵌套记录注册一个新的工件。例如，以下 **TradeKey** 模式包含一个嵌套 **交换** 模式：

带有嵌套交换模式的 TradeKey 模式

```
{
  "namespace": "com.kubetrade.schema.trade",
  "type": "record",
  "name": "TradeKey",
  "fields": [
    {
      "name": "exchange",
      "type": "com.kubetrade.schema.common.Exchange"
    },
    {
      "name": "key",
      "type": "string"
    }
  ]
}
```

Exchange 模式

```
{
  "namespace": "com.kubetrade.schema.common",
  "type": "enum",
  "name": "Exchange",
  "symbols" : ["GEMINI"]
}
```

当将这些模式与 Avro SerDes 搭配使用时，会在 Service Registry 中创建两个工件，一个用于 **TradeKey** 模式，一个用于 **Exchange** schema。每当使用 **TradeKey** 模式的消息被序列化或反序列化时，都会检索这两个模式，允许您将定义分成不同的文件。

其他资源

- 有关 Avro 配置的详情，请参阅 AvroKafkaSerdeConfig Java 类
- 对于 Java 示例应用程序，请参阅：
  - 简单 Avro 示例
  - 带有参考示例的 SerDes

8.3.2. 使用 Service Registry 配置 JSON 架构 SerDe

本节解释了如何对 JSON Schema 使用 Kafka 客户端序列化器和反序列化器(SerDes)类。

Service Registry 为 JSON Schema 提供以下 Kafka 客户端 SerDes 类：

- **io.apicurio.registry.serde.jsonschema.JsonSchemaKafkaSerializer**

- **io.apicurio.registry.serde.jsonschema.JsonSchemaKafkaDeserializer**

与 Apache Avro 不同，JSON Schema 不是序列化技术，而是使用验证技术。因此，JSON 架构的配置选项非常不同。例如，没有编码选项，因为数据始终编码为 JSON。

配置 JSON 架构序列化器

您可以配置 JSON Schema serializer 类，如下所示：

- Service Registry URL
- 工件解析器策略
- 模式验证

唯一的非标准配置属性是 JSON Schema 验证，默认是启用的。您可以通过将 **apicurio.registry.serde.validation-enabled** 设置为 **"false"** 来禁用此功能。例如：

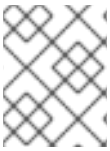
```
props.putIfAbsent(SerdeConfig.VALIDATION_ENABLED, Boolean.FALSE)
```

配置 JSON 架构反序列化器

您可以配置 JSON Schema 反序列化器类，如下所示：

- Service Registry URL
- 模式验证
- 用于取消序列化数据的类

您必须提供 Service Registry 的位置，以便可以加载 schema。其他配置是可选的。



注意

只有在 serializer 传递了 Kafka 消息中的全局 ID 时，反序列化器验证才能正常工作，这只有在 serializer 中启用了验证时才有效。

JSON 架构 SerDes 和工件引用

JSON Schema SerDes 无法从消息有效负载发现模式，因此必须先注册 schema 工件，这也应用工件引用。

根据架构的内容，如果 **$ref** 值是一个 URL，SerDes 会尝试使用该 URL 解析引用的模式，然后验证可以正常工作，针对主模式验证数据，并根据嵌套模式验证嵌套值。另外还实现了对在 Service Registry 中引用工件的支持。

例如，以下 Federal **.json** 模式引用 **city.json** 模式：

为 **city.json schema 的引用，并引用 city.json schema**

```
{
 "$id": "https://example.com/citizen.schema.json",
 "$schema": "http://json-schema.org/draft-07/schema#",
 "title": "Citizen",
 "type": "object",
 "properties": {
```

```
    "firstName": {
      "type": "string",
      "description": "The citizen's first name."
    },
    "lastName": {
      "type": "string",
      "description": "The citizen's last name."
    },
    "age": {
      "description": "Age in years which must be equal to or greater than zero.",
      "type": "integer",
      "minimum": 0
    },
    "city": {
      "$ref": "city.json"
    }
  }
}
```

city.json schema

```
{
  "$id": "https://example.com/city.schema.json",
  "$schema": "http://json-schema.org/draft-07/schema#",
  "title": "City",
  "type": "object",
  "properties": {
    "name": {
      "type": "string",
      "description": "The city's name."
    },
    "zipCode": {
      "type": "integer",
      "description": "The zip code.",
      "minimum": 0
    }
  }
}
```

在本例中，给定的公民具有城市。在 Service Registry 中，使用名称 **city.json** 创建对城市工件的引用。在 SerDes 中，当获取统计形模式时，也会获取城市模式，因为它被有意引用。序列化/解码数据时，引用名称用于解析嵌套模式，允许根据大众模式和嵌套城市模式进行验证。

其他资源

- 如需了解更多详细信息，请参阅 JsonSchemaKafkaDeserializerConfig Java 类
- 对于 Java 示例应用程序，请参阅：
  - 简单 JSON 架构示例
  - 带有参考示例的 SerDes

8.3.3. 使用 Service Registry 配置 Protobuf SerDes

本节解释了如何对 Google Protobuf 使用 Kafka 客户端序列化器和反序列化器(SerDes)类。

Service Registry 为 Protobuf 提供以下 Kafka 客户端 SerDes 类：

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

配置 Protobuf serializer

您可以配置 Protobuf serializer 类，如下所示：

- Service Registry URL
- 工件解析器策略
- ID 位置
- ID 编码
- 模式验证

有关这些配置选项的详情，请查看以下部分：

- 第 8.1 节 “在客户端应用程序中的 Service Registry serializer/deserializer 配置”
- 第 8.3.1 节 “使用 Service Registry 配置 Avro SerDe”

配置 Protobuf deserializer

您必须配置 Protobuf deserializer 类，以匹配序列化器中的以下配置设置：

- Service Registry URL
- ID 编码

配置属性名称和值与序列化器相同。

注意

配置反序列化器时不需要以下选项：

- 工件解析器策略
- ID 位置

deserializer 类可以从消息确定这些选项的值。该策略不是必需的，因为 serializer 负责发送 ID 作为消息的一部分。

ID 位置是通过在消息有效负载开始时检查 magic 字节来确定的。如果找到该字节，则使用配置的处理程序从消息有效负载中读取该 ID。如果没有找到 magic 字节，则从消息标头中读取 ID。

注意

Protobuf deserializer 不会反序列化到确切的 Protobuf 消息实现，而是切换到 **DynamicMessage** 实例。否则，没有适当的 API。

protobuf SerDes 和工件引用

当使用带有 **import** 语句的复杂 Protobuf 消息时，导入的 Protobuf 消息会作为单独的工件存储在 Service Registry 中。然后，当 Service Registry 获取检查 Protobuf 消息时，也会检索引用的方案，以便可以检查和序列化完整的消息模式。

例如，以下 **table_info.proto** 模式文件包含导入的 **mode.proto** 模式文件：

带有导入 mode.proto 文件的 table_info.proto 文件

```
syntax = "proto3";
package sample;
option java_package = "io.api.sample";
option java_multiple_files = true;

import "sample/mode.proto";

message TableInfo {

 int32 winIndex = 1;
 Mode mode = 2;
 int32 min = 3;
 int32 max = 4;
 string id = 5;
 string dataAdapter = 6;
 string schema = 7;
 string selector = 8;
 string subscription_id = 9;
}
```

mode.proto 文件

```
syntax = "proto3";
package sample;
option java_package = "io.api.sample";
option java_multiple_files = true;

enum Mode {

MODE_UNKNOWN = 0;
RAW = 1;
MERGE = 2;
DISTINCT = 3;
COMMAND = 4;
}
```

在本例中，两个 Protobuf 工件存储在 Service Registry 中，一个用于 **TableInfo**，一个用于 **Mode**。但是，由于 **Mode** 是 **TableInfo** 的一部分，因此当获取每个 **TableInfo** 以检查 SerDes 的消息时，mode 也返回为 **TableInfo** 引用的工件。

其他资源

- 对于 Java 示例应用程序，请参阅：
  - protobuf Bean 和 Protobuf Find Latest 示例

- 带有参考示例的 SerDes

第 9 章 SERVICE REGISTRY 工件参考

本章提供了存储在 Service Registry 中支持的工件类型、状态和元数据的参考信息。

- 第 9.1 节 “Service Registry 工件类型”
- 第 9.2 节 “Service Registry 工件状态”
- 第 9.3 节 “Service Registry 工件元数据”

其他资源

- 如需更多信息，请参阅 Apicurio Registry REST API 文档 。

9.1. SERVICE REGISTRY 工件类型

您可以在 Service Registry 中存储和管理广泛的模式和 API 工件类型。

表 9.1. Service Registry 工件类型

类型	描述
ASYNCAPI	AsyncAPI 规格
AVRO	Apache Avro 模式
GRAPHQL	graphql 模式
JSON	JSON 架构
KCONNECT	Apache Kafka Connect 模式
OPENAPI	OpenAPI 规格
PROTOBUF	Google 协议缓冲模式
WSDL	Web 服务定义语言
XML	可扩展标记语言
XSD	XML 架构定义

9.2. SERVICE REGISTRY 工件状态

Service Registry 中的有效工件状态为 **ENABLED**、**DISABLED** 和 **DEPRECATED**。

表 9.2. Service Registry 工件状态

状态	描述
ENABLED	基本状态，所有操作都可用。
DISABLED	工件及其元数据可以使用 Service Registry web 控制台查看并可搜索，但其内容无法被任何客户端获取。
弃用	工件完全可用，但在获取工件内容时，会添加一个标头到 REST API 响应中。Service Registry Rest 客户端也会在看到已弃用内容时记录警告。

9.3. SERVICE REGISTRY 工件元数据

当工件添加到 Service Registry 时，会创建一组元数据属性并与工件内容一起存储。这个元数据由系统生成的或用户生成的属性组成，它们是只读的，您可以在创建工件后更新的属性。

表 9.3. Service Registry 系统生成的元数据

属性	类型	描述
contentId	整数	Service Registry 中工件内容的唯一标识符。当工件版本有相同的内容时，同一内容 ID 可以被多个工件版本共享。例如，内容 ID **4** 可用于具有相同内容的多个工件版本。
createdBy	string	创建工件的用户的名称。
createdOn	date	创建工件的日期和时间，例如 **2023-10-11T14:15:28Z**。
globalId	整数	Service Registry 中工件版本的全局唯一标识符。例如，一个全局 ID **1** 被分配给 Service Registry 中创建的第一个工件版本。
modifiedBy	string	修改工件的用户的名称。
modifiedOn	date	修改工件的日期和时间，例如 **2023-10-11T14:15:28Z**。
type	ArtifactType	支持的工件类型，如 **AVRO**、**OPENAPI** 或 **PROTOBUF**。

表 9.4. Service Registry 用户提供的或系统生成的元数据

属性	类型	描述

属性	类型	描述
groupId	string	Service Registry 中工件组的唯一标识符，如 **development** 或 **production**。当使用 Service Registry web 控制台创建工件时，如果没有提供组 ID，则这 **设置为默认**。在使用 Apicurio Registry REST API、Java 客户端或 Maven 插件时，您必须提供组 ID。
id	string	Service Registry 中工件的唯一标识符。您可以提供工件 ID，或使用 Service Registry 生成的 UUID，例如 **8d168cad-1865-4e6c-bb7e-04e8be005bea**。工件的不同版本使用相同的工件 ID，但具有不同的全局 ID。
参考	ArtifactReference 数组	工件中包含的可选工件引用集合，您可以在创建工件时提供它们。以下简单示例显示了一个工件引用：**[{"groupId":"my-group","artifactId":"ItemId":"version":"1","name":"com.example.common.ItemId"}]**.
version	整数	工件的最新版本。您可以使用生成的版本，如 **3**，或使用 Service Registry REST API 或 Maven 插件提供版本，如 **2.1.6**。

表 9.5. Service Registry 可编辑的元数据

属性	类型	描述
description	string	工件的可选有意义的描述，例如，**这是测试 的简单 OpenAPI**。您可以提供描述，如果已提供，可以从 OpenAPI 和 AsyncAPI 工件的 **info** 部分自动发现它。
labels	字符串数组	可选以逗号分隔的标签列表，用于过滤和搜索工件，如 **test,protobuf**。由用户提供的。
name	string	工件的可选可读名称，如 **My first Avro schema**。您可以提供描述，如果 **title** 字段有一个值，则可以从 OpenAPI 和 AsyncAPI 工件的 **info** 部分自动发现它。
属性	map	与工件关联的用户定义的 name-value 对的可选列表。name 和 value 必须是字符串，如 **my-key** 和 **my-value**。
state	ArtifactState	工件的最新状态： **ENABLED**、**DISABLED** 或 **DEPRECATED**。默认为 **ENABLED**。

更新工件元数据

- 您可以使用 Service Registry REST API 或 Web 控制台更新一组可编辑的元数据属性。

- 您只能使用 Service Registry REST API 更新 **state** 属性。

其他资源

如需了解更多详细信息，请参阅 Apicurio Registry REST API 文档中的 **/artifacts/{artifactId}/meta** 端点。

第 10 章 SERVICE REGISTRY 内容规则引用

本章提供有关支持的内容规则类型的参考信息，它们对工件类型的支持级别，以及特定于工件和全局规则的优先级顺序。

- 第 10.1 节 “Service Registry 内容规则类型”
- 第 10.2 节 “Service Registry 内容规则成熟度”
- 第 10.3 节 “Service Registry 内容规则优先级”

其他资源

- 如需更多信息，请参阅 Apicurio Registry REST API 文档 。

10.1. SERVICE REGISTRY 内容规则类型

您可以指定 **VALIDITY**、**COMPATIBILITY** 和 **INTEGRITY** 规则类型，以管理 Service Registry 中的内容演进。这些规则类型适用于全局规则和特定于工件的规则。

表 10.1. Service Registry 内容规则类型

类型	描述
有效期	在将内容添加到 Service Registry 之前验证内容。此规则可能的配置值如下： • **FULL** ：验证是语法和语义。 • **SYNTAX_ONLY** ：验证只是语法。 • **NONE**: 禁用所有验证检查。

类型	描述
兼容性	在更新工件时强制实施兼容性级别（例如，选择 **BACKWARD** 以向后兼容）。确保新工件与之前添加的工件版本或客户端兼容。此规则可能的配置值如下： • **FULL** ：新工件正向前和向后兼容，与最近添加的工件兼容。 • **FULL_TRANSITIVE** ：新工件与所有以前添加的工件时向前和向后兼容的。 • **BACKWARD**: 使用新工件的客户端可以读取使用最近添加的工件写入的数据。 • **BACKWARD_TRANSITIVE** ：使用新工件的客户端可以读取使用所有之前添加的工件写入的数据。 • **FORWARD**: 使用最新添加的工件的客户端可以读取使用新工件写入的数据。 • **FORWARD_TRANSITIVE** ：使用所有以前添加的工件的客户端可以读取使用新工件写入的数据。 • **NONE** ：禁用所有向后和转发兼容性检查。
完整性	在创建或更新工件时强制实施工件引用完整性。启用并配置此规则，以确保任何提供的工件引用都正确。此规则可能的配置值如下： • **FULL**: 所有工件引用完整性检查都已启用。 • **NO_DUPLICATES**: 如果有任何重复的工件引用，则拒绝。 • **REFS_EXIST** ：如果存在对不存在的工件的引用，则忽略。 • **ALL_REFS_MAPPED**: 确保所有工件引用都已映射。 • **NONE** ：所有工件引用完整性检查都被禁用。

10.2. SERVICE REGISTRY 内容规则成熟度

不是所有内容规则都针对 Service Registry 支持的每个工件类型完全实现。下表显示了每个规则和工件类型的当前成熟度等级：

表 10.2. Service Registry 内容规则成熟度列表

工件类型	有效规则	兼容性规则	完整性规则
Avro	full	full	full
protobuf	full	full	full
JSON 架构	full	full	不支持映射检测
OpenAPI	full	None	full
AsyncAPI	仅语法	None	full
GraphQL	仅语法	None	不支持映射检测
Kafka Connect	仅语法	None	不支持映射检测
WSDL	full	None	不支持映射检测
XML	full	None	不支持映射检测
XSD	full	None	不支持映射检测

10.3. SERVICE REGISTRY 内容规则优先级

当您添加或更新工件时，Service Registry 应用规则来检查工件内容的有效性、兼容性或完整性。配置特定于工件的规则会覆盖对等配置的全局规则，如下表所示。

表 10.3. Service Registry 内容规则优先级

特定于工件的规则	全局规则	应用到此工件的规则	全局规则可用于其他工件？
Enabled	Enabled	特定于工件	是
Disabled	Enabled	全局	是
Disabled	Disabled	None	否
enabled，设置为 None	Enabled	None	是
Disabled	enabled，设置为 None	None	否

附录 A. 使用您的订阅

Service Registry 通过软件订阅提供。要管理您的订阅，请访问红帽客户门户中的帐户。

访问您的帐户

1. 转至 access.redhat.com。
2. 如果您还没有帐户，请创建一个帐户。
3. 登录到您的帐户。

激活订阅

1. 转至 access.redhat.com。
2. 导航到 **My Subscriptions**。
3. 导航到 **激活订阅** 并输入您的 16 位激活号。

下载 ZIP 和 TAR 文件

要访问 ZIP 或 TAR 文件，请使用客户门户网站查找下载的相关文件。如果您使用 RPM 软件包，则不需要这一步。

1. 打开浏览器并登录红帽客户门户网站 **产品下载页面**，网址为 access.redhat.com/downloads。
2. 在 Integration **和 Automation 类别中找到 Red Hat Integration** 条目。
3. 选择所需的 Service Registry 产品。此时会打开 **Software Downloads** 页面。
4. 单击组件的 **Download** 链接。

更新于 2024-01-26