



# Red Hat Integration 2021.Q2

## Service Registry User Guide

Service Registry 2.0 - Technology Preview



# Red Hat Integration 2021.Q2 Service Registry User Guide

---

Service Registry 2.0 - Technology Preview

## Legal Notice

Copyright © 2021 Red Hat, Inc.

The text of and illustrations in this document are licensed by Red Hat under a Creative Commons Attribution–Share Alike 3.0 Unported license ("CC-BY-SA"). An explanation of CC-BY-SA is available at

<http://creativecommons.org/licenses/by-sa/3.0/>

. In accordance with CC-BY-SA, if you distribute this document or an adaptation of it, you must provide the URL for the original version.

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

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

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

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

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

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

Node.js<sup>®</sup> 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<sup>®</sup> 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.

## Abstract

This guide introduces Service Registry and explains how to manage event schemas and API designs using the Service Registry web console, REST API, Maven plug-in, or Java client. This guide also explains how to use Kafka client serializers and deserializers in your Java consumer and producer applications. It also describes the supported Service Registry content types, and optional rule configuration.

## Table of Contents

<b>PREFACE</b> .....	<b>4</b>
MAKING OPEN SOURCE MORE INCLUSIVE	4
<b>CHAPTER 1. INTRODUCTION TO SERVICE REGISTRY</b> .....	<b>5</b>
1.1. SERVICE REGISTRY OVERVIEW	5
Service Registry capabilities	6
1.2. SCHEMA AND API ARTIFACTS AND GROUPS IN SERVICE REGISTRY	6
Schema and API groups	7
1.3. MANAGE CONTENT USING SERVICE REGISTRY WEB CONSOLE	7
1.4. REGISTRY CORE REST API OVERVIEW	8
Compatibility with other schema registry REST APIs	9
1.5. SERVICE REGISTRY STORAGE OPTIONS	9
1.6. VALIDATE SCHEMAS WITH KAFKA CLIENT SERIALIZERS/DESERIALIZERS	9
1.7. STREAM DATA TO EXTERNAL SYSTEMS WITH KAFKA CONNECT CONVERTERS	10
1.8. SERVICE REGISTRY DEMONSTRATION EXAMPLES	11
1.9. SERVICE REGISTRY AVAILABLE DISTRIBUTIONS	12
<b>CHAPTER 2. SERVICE REGISTRY CONTENT RULES</b> .....	<b>14</b>
2.1. GOVERN REGISTRY CONTENT USING RULES	14
2.2. WHEN RULES ARE APPLIED	14
2.3. HOW RULES WORK	14
2.4. CONTENT RULE CONFIGURATION	15
Configure artifact rules	15
Configure global rules	15
<b>CHAPTER 3. MANAGING SERVICE REGISTRY CONTENT USING THE WEB CONSOLE</b> .....	<b>17</b>
3.1. ADDING ARTIFACTS USING THE SERVICE REGISTRY WEB CONSOLE	17
3.2. VIEWING ARTIFACTS USING THE SERVICE REGISTRY WEB CONSOLE	18
3.3. CONFIGURING CONTENT RULES USING THE SERVICE REGISTRY WEB CONSOLE	20
<b>CHAPTER 4. MANAGING SERVICE REGISTRY CONTENT USING THE REST API</b> .....	<b>22</b>
4.1. MANAGING SCHEMA AND API ARTIFACTS USING REGISTRY REST API COMMANDS	22
4.2. MANAGING SCHEMA AND API ARTIFACT VERSIONS USING REGISTRY REST API COMMANDS	23
4.3. EXPORTING AND IMPORTING REGISTRY CONTENT USING REGISTRY REST API COMMANDS	24
<b>CHAPTER 5. MANAGING SERVICE REGISTRY CONTENT USING THE MAVEN PLUG-IN</b> .....	<b>26</b>
5.1. ADDING SCHEMA AND API ARTIFACTS USING THE MAVEN PLUG-IN	26
5.2. DOWNLOADING SCHEMA AND API ARTIFACTS USING THE MAVEN PLUG-IN	27
5.3. TESTING SCHEMA AND API ARTIFACTS USING THE MAVEN PLUG-IN	28
<b>CHAPTER 6. MANAGING SERVICE REGISTRY CONTENT USING A JAVA CLIENT</b> .....	<b>30</b>
6.1. SERVICE REGISTRY JAVA CLIENT	30
6.2. WRITING SERVICE REGISTRY CLIENT APPLICATIONS	30
6.3. SERVICE REGISTRY JAVA CLIENT CONFIGURATION	31
Custom header configuration	31
TLS configuration options	31
<b>CHAPTER 7. VALIDATING SCHEMAS USING KAFKA CLIENT SERIALIZERS/DESERIALIZERS IN JAVA</b> ...	<b>33</b>
7.1. KAFKA CLIENT APPLICATIONS AND SERVICE REGISTRY	33
Service Registry schema technologies	34
Producer schema configuration	34
Consumer schema configuration	34
7.2. STRATEGIES TO LOOK UP A SCHEMA IN SERVICE REGISTRY	35

ArtifaceResolverStrategy interface	36
Strategies to return an artifact reference	36
DefaultSchemaResolver interface	36
Strategies to return a global ID	36
7.3. SERVICE REGISTRY SERIALIZER/DESERIALIZER CONFIGURATION	37
Configuration for SerDe services	37
Configuration for SerDe lookup strategies	38
Configuration for Kafka converters	38
Configuration for different schema types	38
7.4. USING DIFFERENT CLIENT SERIALIZER/DESERIALIZER TYPES	38
Kafka application configuration for serializers/deserializers	38
7.4.1. Configure Avro SerDe with Service Registry	40
7.4.2. Configure JSON Schema SerDe with Service Registry	42
7.4.3. Configure Protobuf SerDe with Service Registry	43
7.5. REGISTERING A SCHEMA IN SERVICE REGISTRY	44
Service Registry web console	44
Curl command example	44
Maven plug-in example	45
Configuration using a producer client example	46
7.6. USING A SCHEMA FROM A KAFKA CONSUMER CLIENT	46
7.7. USING A SCHEMA FROM A KAFKA PRODUCER CLIENT	47
7.8. USING A SCHEMA FROM A KAFKA STREAMS APPLICATION	47
<b>CHAPTER 8. SERVICE REGISTRY ARTIFACT REFERENCE</b> .....	<b>49</b>
8.1. SERVICE REGISTRY ARTIFACT TYPES	49
8.2. SERVICE REGISTRY ARTIFACT STATES	49
8.3. SERVICE REGISTRY ARTIFACT METADATA	50
8.4. SERVICE REGISTRY CONTENT RULE TYPES	51
8.5. SERVICE REGISTRY CONTENT RULE MATURITY	52
<b>APPENDIX A. USING YOUR SUBSCRIPTION</b> .....	<b>54</b>
Accessing your account	54
Activating a subscription	54
Downloading ZIP and TAR files	54
Registering your system for packages	54



## PREFACE

### MAKING OPEN SOURCE MORE INCLUSIVE

Red Hat is committed to replacing problematic language in our code, documentation, and web properties. We are beginning with these four terms: master, slave, blacklist, and whitelist. Because of the enormity of this endeavor, these changes will be implemented gradually over several upcoming releases. For more details, see [our CTO Chris Wright's message](#).



# CHAPTER 1. INTRODUCTION TO SERVICE REGISTRY

This chapter introduces Service Registry concepts and features and provides details on the supported artifact types that are stored in the registry:

- [Section 1.1, "Service Registry overview"](#)
- [Section 1.2, "Schema and API artifacts and groups in Service Registry"](#)
- [Section 1.3, "Manage content using Service Registry web console"](#)
- [Section 1.4, "Registry core REST API overview"](#)
- [Section 1.5, "Service Registry storage options"](#)
- [Section 1.6, "Validate schemas with Kafka client serializers/deserializers"](#)
- [Section 1.7, "Stream data to external systems with Kafka Connect converters"](#)
- [Section 1.8, "Service Registry demonstration examples"](#)
- [Section 1.9, "Service Registry available distributions"](#)



## IMPORTANT

Service Registry is a Technology Preview feature only. Technology Preview features are not supported with Red Hat production service level agreements (SLAs) and might not be functionally complete. Red Hat does not recommend using them in production.

These features provide early access to upcoming product features, enabling customers to test functionality and provide feedback during the development process. For more information about the support scope of Red Hat Technology Preview features, see <https://access.redhat.com/support/offerings/techpreview>.

## 1.1. SERVICE REGISTRY OVERVIEW

Service Registry is a datastore for sharing standard event schemas and API designs across API and event-driven architectures. You can use Service Registry to decouple the structure of your data from your client applications, and to share and manage your data types and API descriptions at runtime using a REST interface.

For example, client applications can dynamically push or pull the latest schema updates to or from Service Registry at runtime without needing to redeploy. Developer teams can query the registry for existing schemas required for services already deployed in production, and can register new schemas required for new services in development.

You can enable client applications to use schemas and API designs stored in Service Registry by specifying the registry URL in your client application code. For example, the registry can store schemas used to serialize and deserialize messages, which can then be referenced from your client applications to ensure that the messages that they send and receive are compatible with those schemas.

Using Service Registry to decouple your data structure from your applications reduces costs by decreasing overall message size, and creates efficiencies by increasing consistent reuse of schemas and API designs across your organization. Service Registry provides a web console to make it easy for developers and administrators to manage registry content.

In addition, you can configure optional rules to govern the evolution of your registry content. For example, these include rules to ensure that uploaded content is syntactically and semantically valid, or is backwards and forwards compatible with other versions. Any configured rules must pass before new versions can be uploaded to the registry, which ensures that time is not wasted on invalid or incompatible schemas or API designs.

Service Registry is based on the Apicurio Registry open source community project. For details, see <https://github.com/apicurio/apicurio-registry>.

### Service Registry capabilities

- Multiple payload formats for standard event schemas and API specifications
- Pluggable registry storage options in AMQ Streams or PostgreSQL database
- Registry content management using a web console, REST API command, Maven plug-in, or Java client
- Rules for content validation and version compatibility to govern how registry content evolves over time
- Full Apache Kafka schema registry support, including integration with Kafka Connect for external systems
- Kafka client serializers/deserializers (Serdes) to validate message types at runtime
- Cloud-native Quarkus Java runtime for low memory footprint and fast deployment times
- Compatibility with existing Confluent or IBM schema registry client applications
- Operator-based installation of Service Registry on OpenShift
- OpenID Connect (OIDC) authentication using Red Hat Single Sign-On

## 1.2. SCHEMA AND API ARTIFACTS AND GROUPS IN SERVICE REGISTRY

The items stored in Service Registry, such as event schemas and API designs, are known as registry *artifacts*. The following shows an example of an Apache Avro schema artifact in JSON format for a simple share price application:

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

When a schema or API design is added as an artifact in the registry, client applications can then use that schema or API design to validate that the client messages conform to the correct data structure at runtime.

Service Registry supports a wide range of message payload formats for standard event schemas and API specifications. For example, supported formats include Apache Avro, Google Protobuf, GraphQL, AsyncAPI, OpenAPI, and others. For more details, see [Chapter 8, Service Registry artifact reference](#).

### Schema and API groups

An *artifact group* is an optional named collection of schema or API artifacts. Each group contains a logically related set of schemas or API designs, typically managed by a single entity, belonging to a particular application or organization.

You can create optional artifact groups when adding your schemas and API designs to organize them in Service Registry. For example, you could create groups to match your **development** and **production** application environments, or your **sales** and **engineering** organizations.

Schema and API groups can contain multiple artifact types. For example, you could have Protobuf, Avro, JSON Schema, OpenAPI, and AsyncAPI schema and API artifacts all in the same group.

You can create schema and API artifacts and optional groups using the Service Registry web console, core REST API, Maven plug-in, or Java client application. The following simple example shows using the 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
```

This example adds an Avro schema with an artifact ID of **share-price** in an artifact group named **my-group**.



#### NOTE

Specifying a group is optional when using the Service Registry web console, where a **default** group is automatically created. When using the v2 REST API or Maven plug-in, you can specify the **default** group in the API path if you do not want to create a unique group.

#### Additional resources

- For more details on schemas and groups, see the [Cloud Native Computing Foundation \(CNCF\) Schema Registry API Version 0.1](#).

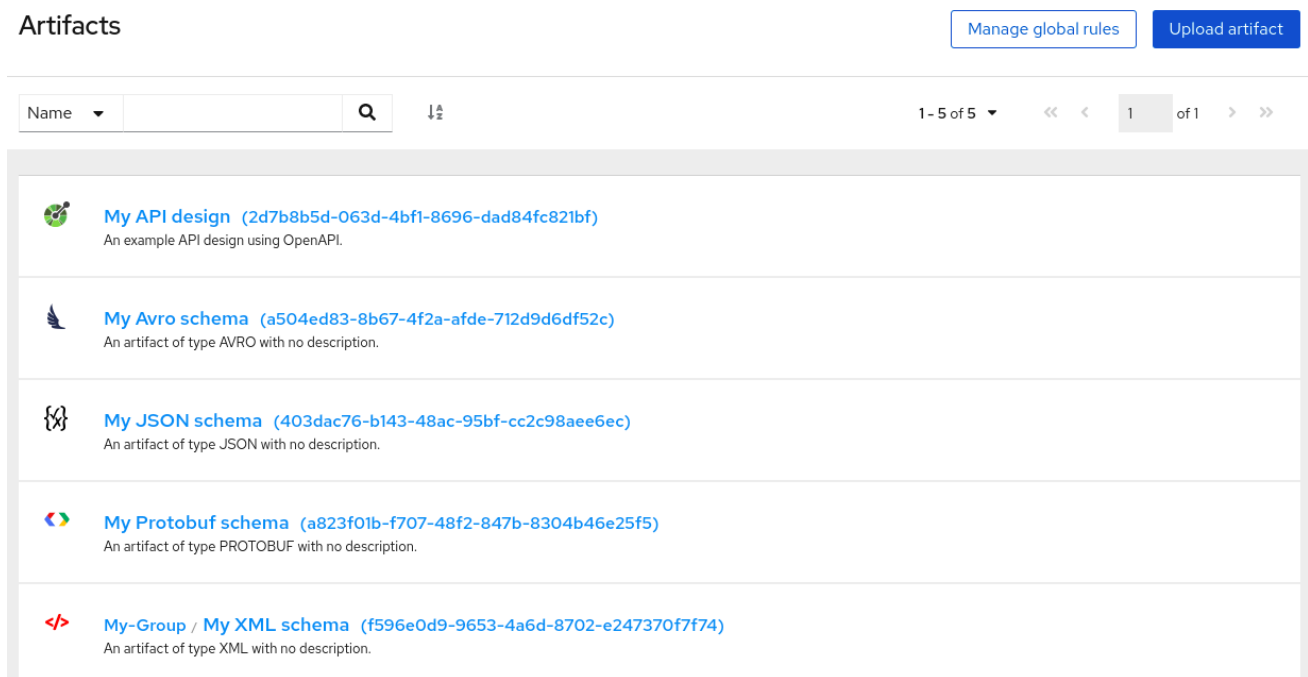
## 1.3. MANAGE CONTENT USING SERVICE REGISTRY WEB CONSOLE

You can use the Service Registry web console to browse and search the schema and API artifacts and optional groups stored in the registry, and to add new schema and API artifacts, groups, and versions. You can search for artifacts by label, name, group, and description. You can view an artifact's content or its available versions, or download an artifact file locally.

You can also use the web console to configure optional rules for registry content, both globally and for each schema and API artifact. These optional rules for content validation and compatibility are applied when new schema and API artifacts or versions are uploaded to the registry. For more details, see

## Chapter 8, *Service Registry artifact reference* .

Figure 1.1. Service Registry web console



The Service Registry web console is available from the main endpoint of your Service Registry deployment, for example, on **http://MY-REGISTRY-URL/ui**.

### Additional resources

- [Chapter 3, \*Managing Service Registry content using the web console\*](#)

## 1.4. REGISTRY CORE REST API OVERVIEW

Using the Service Registry core REST API, client applications can manage the schema and API artifacts in Service Registry. This API provides create, read, update, and delete operations for:

### Artifacts

Manage schema and API artifacts stored in the registry. You can also manage the lifecycle state of an artifact: enabled, disabled, or deprecated.

### Artifact versions

Manage versions that are created when a schema or API artifact is updated. You can also manage the lifecycle state of an artifact version: enabled, disabled, or deprecated.

### Artifact metadata

Manage details about a schema or API artifact, such as when it was created or modified, and its current state. You can edit the artifact name, description, or label. The artifact group and when the artifact was created or modified are read-only.

### Artifact rules

Configure rules to govern the content evolution of a specific schema or API artifact to prevent invalid or incompatible content from being added to the registry. Artifact rules override any global rules configured.

### Global rules

Configure rules to govern the content evolution of all schema and API artifacts artifacts to prevent invalid or incompatible content from being added to the registry. Global rules are applied only if an artifact does not have its own specific artifact rules configured.

### Search

Browse or search for schema and API artifacts and versions, for example, by name, group, description, or label.

### Admin

Export or import registry content in a **.zip** file, and manage logging levels for the registry server instance at runtime.

### Compatibility with other schema registry REST APIs

Service Registry version 2 provides API compatibility with the following schema registries by including implementations of their respective REST APIs:

- Service Registry version 1
- Confluent schema registry version 6
- IBM schema registry version 1
- Cloud Native Computing Foundation schema registry version 0

Applications using Confluent client libraries can use Service Registry as a drop-in replacement. For more details, see [Replacing Confluent Schema Registry with Red Hat Integration Service Registry](#) .

### Additional resources

- For detailed information, see the [Apicurio Registry REST API documentation](#) .
- API documentation for the core Service Registry REST API and for all compatible APIs is available from the main endpoint of your Service Registry deployment, for example, on **<http://MY-REGISTRY-URL/apis>**.

## 1.5. SERVICE REGISTRY STORAGE OPTIONS

Service Registry provides the following options for the underlying storage of registry data:

- PostgreSQL 12 database
- AMQ Streams 1.7

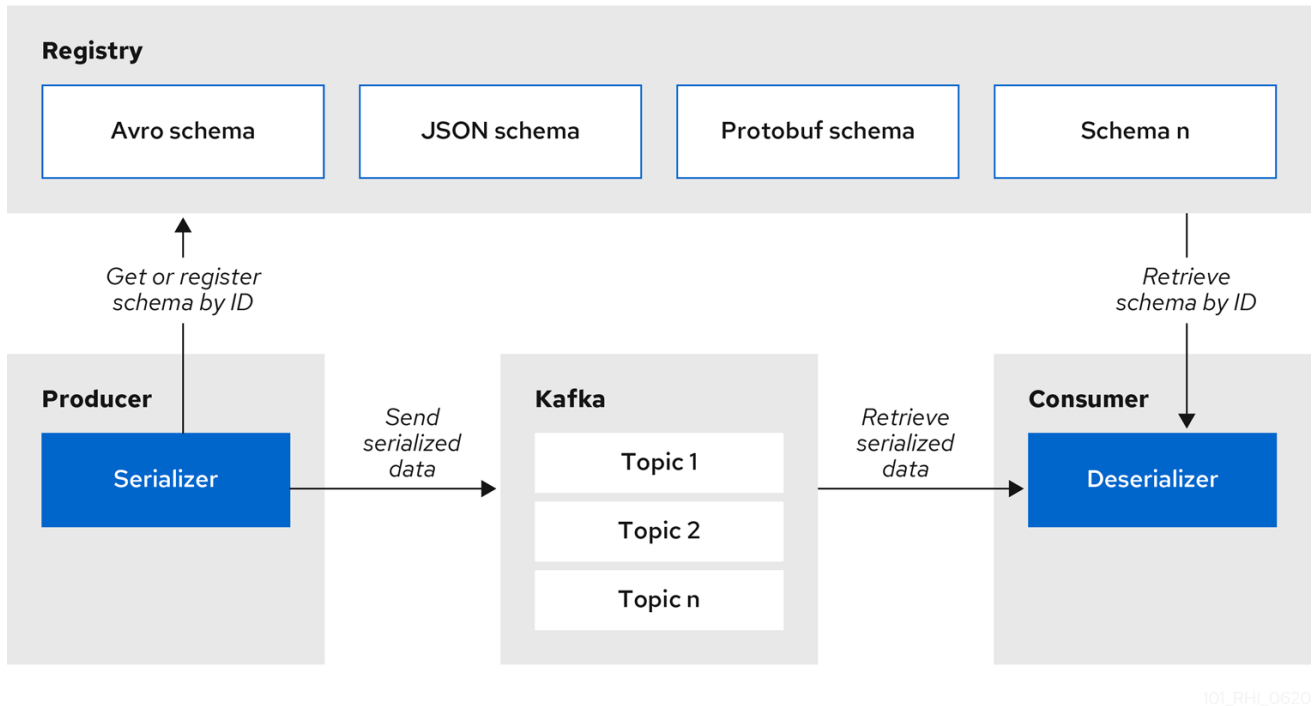
### Additional resources

- For more details on storage options, see [Installing and deploying Service Registry on OpenShift](#)

## 1.6. VALIDATE SCHEMAS WITH KAFKA CLIENT SERIALIZERS/DESERIALIZERS

Kafka producer applications can use serializers to encode messages that conform to a specific event schema. Kafka consumer applications can then use deserializers to validate that messages have been serialized using the correct schema, based on a specific schema ID.

Figure 1.2. Service Registry and Kafka client SerDe architecture



Service Registry provides Kafka client serializers/deserializers (SerDes) to validate the following message types at runtime:

- Apache Avro
- Google protocol buffers
- JSON Schema

The Service Registry Maven repository and source code distributions include the Kafka SerDe implementations for these message types, which Kafka client developers can use to integrate with the registry. These implementations include custom Java classes for each supported message type, for example, `io.apicurio.registry.serde.avro`, which client applications can use to pull schemas from the registry at runtime for validation.

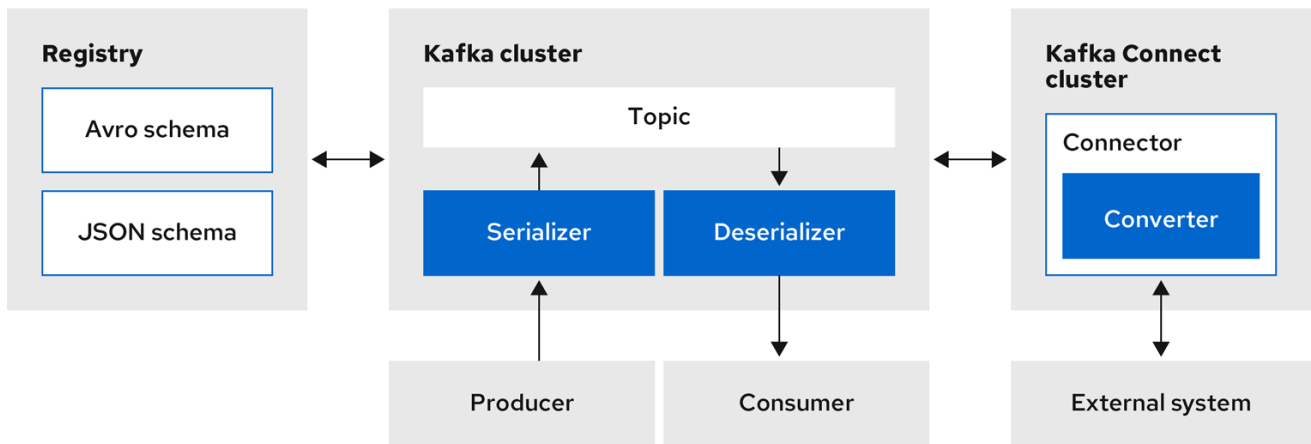
#### Additional resources

- [Chapter 7, Validating schemas using Kafka client serializers/deserializers in Java](#)

## 1.7. STREAM DATA TO EXTERNAL SYSTEMS WITH KAFKA CONNECT CONVERTERS

You can use Service Registry with Apache Kafka Connect to stream data between Kafka and external systems. Using Kafka Connect, you can define connectors for different systems to move large volumes of data into and out of Kafka-based systems.

Figure 1.3. Service Registry and Kafka Connect architecture



ID1\_RHL\_0620

Service Registry provides the following features for Kafka Connect:

- Storage for Kafka Connect schemas
- Kafka Connect converters for Apache Avro and JSON Schema
- Registry REST API to manage schemas

You can use the Avro and JSON Schema converters to map Kafka Connect schemas into Avro or JSON schemas. Those schemas can then serialize message keys and values into the compact Avro binary format or human-readable JSON format. The converted JSON is also less verbose because the messages do not contain the schema information, only the schema ID.

Service Registry can manage and track the Avro and JSON schemas used in the Kafka topics. Because the schemas are stored in Service Registry and decoupled from the message content, each message must only include a tiny schema identifier. For an I/O bound system like Kafka, this means more total throughput for producers and consumers.

The Avro and JSON Schema serializers and deserializers (SerDes) provided by Service Registry are also used by Kafka producers and consumers in this use case. Kafka consumer applications that you write to consume change events can use the Avro or JSON Serdes to deserialize these change events. You can install these Serdes into any Kafka-based system and use them along with Kafka Connect, or with Kafka Connect-based systems such as Debezium and Camel Kafka Connector.

#### Additional resources

- [Apache Kafka Connect documentation](#)
- [Avro serialization in Debezium User Guide](#)
- [Getting Started with Camel Kafka Connector](#)
- [Demonstration of using Kafka Connect with Debezium and Apicurio Registry](#)

## 1.8. SERVICE REGISTRY DEMONSTRATION EXAMPLES

Service Registry provides open source example applications that demonstrate how to use the registry in different use case scenarios. For example, these include storing schemas used by Kafka serializer and

deserializer (SerDe) classes. These Java classes fetch the schema from the registry for use when producing or consuming operations to serialize, deserialize, or validate the Kafka message payload.

These example applications include the following:

- Simple Avro
- Simple JSON Schema
- Confluent SerDes integration
- Avro bean
- Custom ID strategy
- Simple Avro Maven
- REST client
- Mix Avro schemas
- Cloud Events

For more details, see <https://github.com/Apicurio/apicurio-registry-examples>

## 1.9. SERVICE REGISTRY AVAILABLE DISTRIBUTIONS

Table 1.1. Service Registry Operator and images

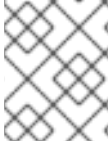
Distribution	Location	Release category
Service Registry Operator	OpenShift web console under <b>Operators → OperatorHub</b>	Technology Preview
Container image for Service Registry Operator	<a href="#">Red Hat Ecosystem Catalog</a>	Technology Preview
Container image for Kafka storage in AMQ Streams	<a href="#">Red Hat Ecosystem Catalog</a>	Technology Preview
Container image for database storage in PostgreSQL	<a href="#">Red Hat Ecosystem Catalog</a>	Technology Preview

Table 1.2. Service Registry zip downloads

Distribution	Location	Release category
Example custom resource definitions for installation	<a href="#">Software Downloads for Red Hat Integration</a>	Technology Preview
Kafka Connect converters	<a href="#">Software Downloads for Red Hat Integration</a>	Technology Preview



Distribution	Location	Release category
Maven repository	<a href="#">Software Downloads for Red Hat Integration</a>	Technology Preview
Source code	<a href="#">Software Downloads for Red Hat Integration</a>	Technology Preview

**NOTE**

You must have a subscription for Red Hat Integration and be logged into the Red Hat Customer Portal to access the available Service Registry distributions.

## CHAPTER 2. SERVICE REGISTRY CONTENT RULES

This chapter introduces the optional rules used to govern registry content and provides details on the available rule configuration:

- [Section 2.1, "Govern registry content using rules"](#)
- [Section 2.2, "When rules are applied"](#)
- [Section 2.3, "How rules work"](#)
- [Section 2.4, "Content rule configuration"](#)

### 2.1. GOVERN REGISTRY CONTENT USING RULES

To govern the evolution of registry content, you can configure optional rules for artifact content added to the registry. All configured global rules or artifact rules must pass before a new artifact version can be uploaded to the registry. Configured artifact rules override any configured global rules.

The goal of these rules is to prevent invalid content from being added to the registry. For example, content can be invalid for the following reasons:

- Invalid syntax for a given artifact type (for example, **AVRO** or **PROTOBUF**)
- Valid syntax, but semantics violate a specification
- Incompatibility, when new content includes breaking changes relative to the current artifact version

You can add these optional content rules using the Service Registry web console, REST API commands, or a Java client application.

### 2.2. WHEN RULES ARE APPLIED

Rules are applied only when content is added to the registry. This includes the following REST operations:

- Adding an artifact
- Updating an artifact
- Adding an artifact version

If a rule is violated, Service Registry returns an HTTP error. The response body includes the violated rule and a message showing what went wrong.



#### NOTE

If no rules are configured for an artifact, the set of currently configured global rules are applied, if any.

### 2.3. HOW RULES WORK

Each rule has a name and optional configuration information. The registry storage maintains the list of rules for each artifact and the list of global rules. Each rule in the list consists of a name and a set of configuration properties, which are specific to the rule implementation.

A rule is provided with the content of the current version of the artifact (if one exists) and the new version of the artifact being added. The rule implementation returns true or false depending on whether the artifact passes the rule. If not, the registry reports the reason why in an HTTP error response. Some rules might not use the previous version of the content. For example, compatibility rules use previous versions, but syntax or semantic validity rules do not.

### Additional resources

For more details, see [Chapter 8, Service Registry artifact reference](#).

## 2.4. CONTENT RULE CONFIGURATION

You can configure rules individually for each artifact, as well as globally. Service Registry applies the rules configured for the specific artifact. If no rules are configured at that level, Service Registry applies the globally configured rules. If no global rules are configured, no rules are applied.

### Configure artifact rules

You can configure artifact rules using the Service Registry web console or REST API. For details, see the following:

- [Chapter 3, Managing Service Registry content using the web console](#)
- [Apicurio Registry REST API documentation](#)

### Configure global rules

You can configure global rules in several ways:

- Use the **/rules** operations in the REST API
- Use the Service Registry web console
- Set default global rules using Service Registry application properties

### Configure default global rules

You can configure Service Registry at the application level to enable or disable global rules. You can configure default global rules at installation time without post-install configuration using the following application property format:

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

The following rule names are currently supported:

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

The value of the application property must be a valid configuration option that is specific to the rule being configured. The following table shows the valid values for each rule:

**Table 2.1. Service Registry content rules**

Rule	Value
Validity	FULL
	SYNTAX_ONLY
	NONE
Compatibility	BACKWARD
	BACKWARD_TRANSITIVE
	FORWARD
	FORWARD_TRANSITIVE
	FULL
	FULL_TRANSITIVE
	NONE



#### NOTE

You can configure these application properties as Java system properties or include them in the Quarkus **application.properties** file. For more details, see the [Quarkus documentation](#).

## CHAPTER 3. MANAGING SERVICE REGISTRY CONTENT USING THE WEB CONSOLE

This chapter explains how to manage schema and API artifacts stored in the registry using the Service Registry web console. This includes uploading and browsing registry content, and configuring optional rules:

- [Section 3.1, “Adding artifacts using the Service Registry web console”](#)
- [Section 3.2, “Viewing artifacts using the Service Registry web console”](#)
- [Section 3.3, “Configuring content rules using the Service Registry web console”](#)

### 3.1. ADDING ARTIFACTS USING THE SERVICE REGISTRY WEB CONSOLE

You can use the Service Registry web console to upload event schema and API design artifacts to the registry. For more details on the artifact types that you can upload, see [Chapter 8, Service Registry artifact reference](#). This section shows simple examples of uploading Service Registry artifacts, applying artifact rules, and adding new artifact versions.

#### Prerequisites

- Service Registry must be installed and running in your environment.

#### Procedure

1. Connect to the Service Registry web console on:  
**http://MY\_REGISTRY\_URL/ui**
2. Click **Upload artifact**, and specify the following:
  - **Group & ID:** Use the default empty settings to automatically generate an ID and **default** group, or enter an optional artifact group or ID.
  - **Type:** Use the default **Auto-Detect** setting to automatically detect the artifact type, or select the artifact type from the drop-down, for example, **Avro Schema** or **OpenAPI**.



#### NOTE


The Service Registry server cannot automatically detect the **Kafka Connect Schema** artifact type. You must manually select this artifact type.

- **Artifact:** Drag and drop or click **Browse** to upload a file, for example, **my-schema.json** or **my-openapi.json**.
3. Click **Upload** and view the **Artifact Details**:

Figure 3.1. Artifact Details in Service Registry web console


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** Full  

Ensure that content is *valid* when updating this artifact.

**Compatibility Rule** Enable

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

- **Info:** Displays the artifact name and optional group, description, lifecycle status, when created, and last modified. Click the **Edit Artifact Metadata** pencil icon to edit the artifact name and description or add labels, and click **Download** to download the artifact file locally. Also displays artifact **Content Rules** that you can enable and configure.
  - **Documentation** (OpenAPI only): Displays automatically-generated REST API documentation.
  - **Content:** Displays a read-only view of the full artifact content.
4. In **Content Rules**, click **Enable** to configure a **Validity Rule** or **Compatibility Rule**, and select the appropriate rule configuration from the drop-down. For more details, see [Chapter 8, Service Registry artifact reference](#).
  5. Click **Upload new version** to add a new artifact version, and drag and drop or click **Browse** to upload the file, for example, **my-schema.json** or **my-openapi.json**.
  6. To delete an artifact, click the trash icon next to **Upload new version**.



#### WARNING

Deleting an artifact deletes the artifact and all of its versions, and cannot be undone. Artifact versions are immutable and cannot be deleted individually.

#### Additional resources

- [Section 3.2, "Viewing artifacts using the Service Registry web console"](#)
- [Section 3.3, "Configuring content rules using the Service Registry web console"](#)

## 3.2. VIEWING ARTIFACTS USING THE SERVICE REGISTRY WEB CONSOLE

You can use the Service Registry web console to browse the event schema and API design artifacts stored in the registry. This section shows simple examples of viewing Service Registry artifacts, groups, versions, and artifact rules. For more details on the artifact types stored in the registry, see [Chapter 8, Service Registry artifact reference](#).

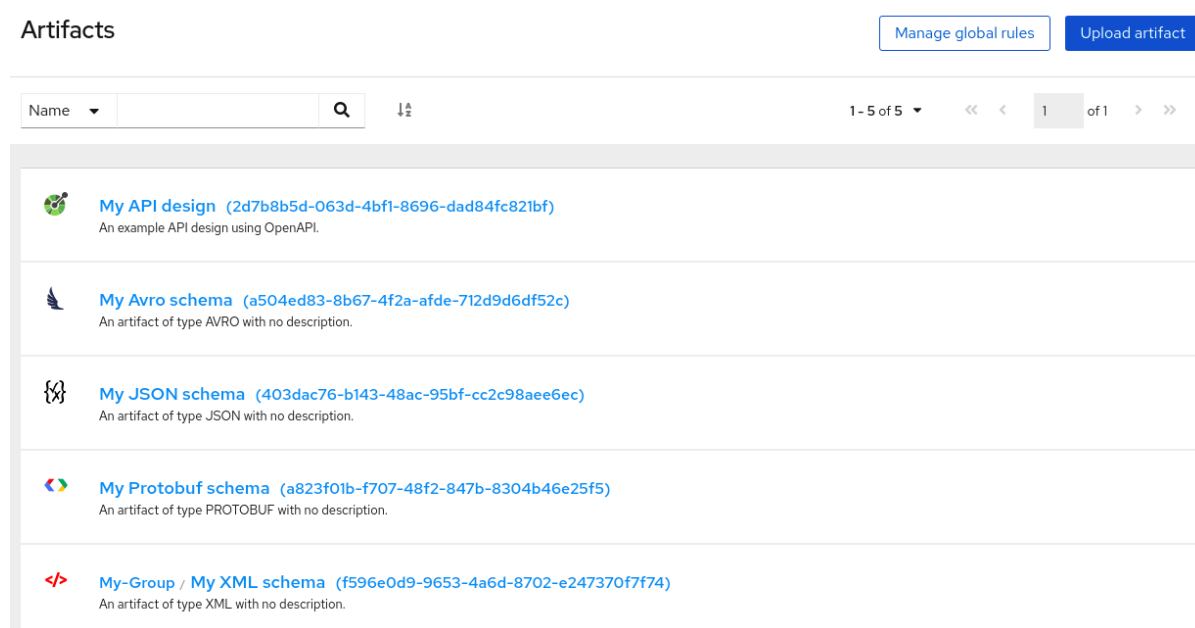
## Prerequisites

- Service Registry must be installed and running in your environment.
- Artifacts must have been added to the registry using the Service Registry web console, REST API commands, Maven plug-in, or a Java client application.

## Procedure

1. Connect to the Service Registry web console on:  
**http://MY\_REGISTRY\_URL/ui**
2. Browse the list of artifacts stored in the registry, or enter a search string to find an artifact. You can select to search by a specific **Name**, **Group**, **Description**, or **Labels**.

**Figure 3.2. Browse artifacts in Service Registry web console**



3. Click **View artifact** to view the **Artifact Details**:
  - **Info**: Displays the artifact name and optional group, description, lifecycle status, when created, and last modified. Click the **Edit Artifact Metadata** pencil icon to edit the artifact name and description or add labels, and click **Download** to download the artifact file locally. Also displays artifact **Content Rules** that you can enable and configure.
  - **Documentation** (OpenAPI only): Displays automatically-generated REST API documentation.
  - **Content**: Displays a read-only view of the full artifact content.
4. Select to view a different artifact **Version** from the drop-down, if additional versions have been added.

## Additional resources

- [Section 3.1, "Adding artifacts using the Service Registry web console"](#)
- [Section 3.3, "Configuring content rules using the Service Registry web console"](#)

### 3.3. CONFIGURING CONTENT RULES USING THE SERVICE REGISTRY WEB CONSOLE

You can use the Service Registry web console to configure optional rules to prevent invalid content from being added to the registry. All configured artifact rules or global rules must pass before a new artifact version can be uploaded to the registry. Configured artifact rules override any configured global rules. For more details, see [Chapter 2, Service Registry content rules](#).

This section shows a simple example of configuring global and artifact rules. For details on the different rule types and associated configuration settings that you can select, see [Chapter 8, Service Registry artifact reference](#).

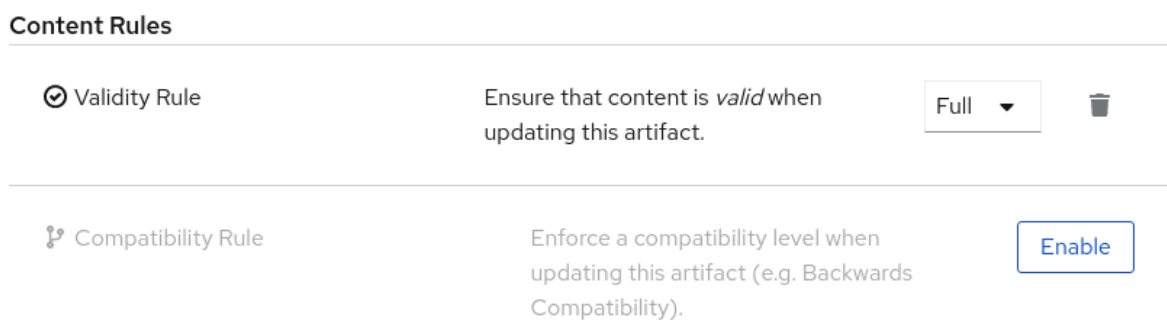
#### Prerequisites

- Service Registry must be installed and running in your environment.
- For artifact rules, artifacts must have been added to the registry using the Service Registry web console, REST API commands, Maven plug-in, or a Java client application.

#### Procedure

1. Connect to the Service Registry web console on:  
**http://MY\_REGISTRY\_URL/ui**
2. For artifact rules, browse the list of artifacts stored in the registry, or enter a search string to find an artifact. You can select to search by a specific artifact **Name**, **Group**, **Description**, or **Labels**.
3. Click **View artifact** to view the **Artifact Details**.
4. In **Content Rules**, click **Enable** to configure an artifact **Validity Rule** or **Compatibility Rule**, and select the appropriate rule configuration from the drop-down. For more details, see [Chapter 8, Service Registry artifact reference](#).

**Figure 3.3. Configure content rules in Service Registry web console**



5. For global rules, click **Manage global rules** at the top right of the toolbar, and click **Enable** to configure a global **Validity Rule** or **Compatibility Rule**, and select the appropriate rule configuration from the drop-down. For more details, see [Chapter 8, Service Registry artifact reference](#).
6. To disable an artifact rule or global rule, click the trash icon next to the rule.

#### Additional resources



- [Section 3.1, "Adding artifacts using the Service Registry web console"](#)

## CHAPTER 4. MANAGING SERVICE REGISTRY CONTENT USING THE REST API

Client applications can use Registry REST API operations to manage schema and API artifacts in Service Registry, for example, in a CI/CD pipeline deployed in production. The Registry REST API provides create, read, update, and delete operations for artifacts, versions, metadata, and rules stored in the registry. For detailed information, see the [Apicurio Registry REST API documentation](#).

This chapter describes the Service Registry core REST API and shows how to use it to manage schema and API artifacts stored in the registry:

- [Section 4.1, “Managing schema and API artifacts using Registry REST API commands”](#)
- [Section 4.2, “Managing schema and API artifact versions using Registry REST API commands”](#)
- [Section 4.3, “Exporting and importing registry content using Registry REST API commands”](#)

### Prerequisites

- [Section 1.4, “Registry core REST API overview”](#)

### Additional resources

- [Apicurio Registry REST API documentation](#)

## 4.1. MANAGING SCHEMA AND API ARTIFACTS USING REGISTRY REST API COMMANDS

This section shows a simple curl-based example of using the registry v2 core REST API to add and retrieve an Apache Avro schema artifact in the registry.

### Prerequisites

- Service Registry must be installed and running in your environment.

### Procedure

1. Add an artifact to the registry using the `/groups/{group}/artifacts` operation. The following example **curl** command adds a simple artifact for a share price application:

```
$ 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"}]}' \ 2
http://MY-REGISTRY-HOST/apis/registry/v2/groups/my-group/artifacts 3
```

- 1** This example adds an Avro schema artifact with an artifact ID of **share-price**. If you do not specify a unique artifact ID, Service Registry generates one automatically as a UUID.

- 2** **MY-REGISTRY-HOST** is the host name on which Service Registry is deployed. For example: **my-cluster-service-registry-myproject.example.com**.

- 3**

This example specifies a group ID of **my-group** in the API path. If you do not specify a unique group ID, you must specify **../groups/default** in the API path.

2. Verify that the response includes the expected JSON body to confirm that the artifact was added. For example:

```
{
  "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
}
```

- 1 No version was specified when adding the artifact, so the default version **1** is created automatically.
- 2 This was the second artifact added to the registry, so the global ID and content ID have a value of **2**.

3. Retrieve the artifact content from the registry using its artifact ID in the API path. In this example, the specified ID is **share-price**:

```
$ curl http://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"}]}
```

#### Additional resources

- For more REST API sample requests, see the [Apicurio Registry REST API documentation](#).

## 4.2. MANAGING SCHEMA AND API ARTIFACT VERSIONS USING REGISTRY REST API COMMANDS

If you do not specify an artifact version when adding schema and API artifacts to Service Registry using the v2 REST API, Service Registry generates one automatically. The default version when creating a new artifact is **1**.

Service Registry also supports custom versioning where you can specify a version using the **X-Registry-Version** HTTP request header as a string. Specifying a custom version value overrides the default version normally assigned when creating or updating an artifact. You can then use this version value when executing REST API operations that require a version.

This section shows a simple curl-based example of using the registry v2 core REST API to add and retrieve a custom Apache Avro schema version in the registry. You can specify custom versions when using the REST API to add or update artifacts or to add artifact versions.

#### Prerequisites

- Service Registry must be installed and running in your environment.

#### Procedure

1. Add an artifact version in the registry using the **/groups/{group}/artifacts** operation. The following example **curl** command adds a simple artifact for a share price application:

```
$ curl -X POST -H "Content-type: application/json; artifactType=AVRO" \
-H "X-Registry-ArtifactId: my-share-price" -H "X-Registry-Version: 1.1.1" \ 1
--data '{"type":"record","name":"p","namespace":"com.example", \
"fields":[{"name":"symbol","type":"string"}, {"name":"price","type":"string"}]}' \ 2
http://MY-REGISTRY-HOST/apis/registry/v2/groups/my-group/artifacts 3
```

- 1 This example adds an Avro schema artifact with an artifact ID of **my-share-price** and version of **1.1.1**. If you do not specify a version, Service Registry automatically generates a default version of **1**.
  - 2 **MY-REGISTRY-HOST** is the host name on which Service Registry is deployed. For example: **my-cluster-service-registry-myproject.example.com**.
  - 3 This example specifies a group ID of **my-group** in the API path. If you do not specify a unique group ID, you must specify **./groups/default** in the API path.
2. Verify that the response includes the expected JSON body to confirm that the custom artifact version was added. For example:

```
{"createdBy":"","createdOn":"2021-04-16T10:51:43+0000","modifiedBy":"","
"modifiedOn":"2021-04-16T10:51:43+0000","id":"my-share-price","version":"1.1.1", 1
"type":"AVRO","globalId":3,"state":"ENABLED","groupId":"my-group","contentId":3} 2
```

- 1 A custom version of **1.1.1** was specified when adding the artifact.
  - 2 This was the third artifact added to the registry, so the global ID and content ID have a value of **3**.
3. Retrieve the artifact content from the registry using its artifact ID and version in the API path. In this example, the specified ID is **my-share-price** and the version is **1.1.1**:

```
$ curl http://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"}]}
```

#### Additional resources

- For more REST API sample requests, see the [Apicurio Registry REST API documentation](#).

## 4.3. EXPORTING AND IMPORTING REGISTRY CONTENT USING REGISTRY REST API COMMANDS

This section shows a simple curl-based example of using the registry v2 core REST API to export and import existing registry data in **.zip** format from one Service Registry instance to another. For example, this is useful when migrating or upgrading from one Service Registry v2.x instance to another.

#### Prerequisites

- Service Registry must be installed and running in your environment.

#### Procedure

## Procedure

1. Export the registry data from your existing source Service Registry instance:

```
$ curl http://MY-REGISTRY-HOST/apis/registry/v2/admin/export \
--output my-registry-data.zip
```

**MY-REGISTRY-HOST** is the host name on which the source Service Registry is deployed. For example: **my-cluster-source-registry-myproject.example.com**.

2. Import the registry data into your target Service Registry instance:

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

**MY-REGISTRY-HOST** is the host name on which the target Service Registry is deployed. For example: **my-cluster-target-registry-myproject.example.com**.

## Additional resources

- For more details, see the **admin** endpoint in the [Apicurio Registry REST API documentation](#).
- For details on export tools for migrating from Service Registry version 1.x to 2.x, see [Apicurio Registry export utility for 1.x versions](#)

## CHAPTER 5. MANAGING SERVICE REGISTRY CONTENT USING THE MAVEN PLUG-IN

This chapter explains how to manage schema and API artifacts stored in the registry using the Service Registry Maven plug-in:

- [Section 5.1, “Adding schema and API artifacts using the Maven plug-in”](#)
- [Section 5.2, “Downloading schema and API artifacts using the Maven plug-in”](#)
- [Section 5.3, “Testing schema and API artifacts using the Maven plug-in”](#)

### Prerequisites

- See [Chapter 1, Introduction to Service Registry](#)
- Service Registry must be installed and running in your environment
- Maven must be installed and configured in your environment

### 5.1. ADDING SCHEMA AND API ARTIFACTS USING THE MAVEN PLUG-IN

The most common use case for the Maven plug-in is adding artifacts during a build. You can accomplish this by using the **register** execution goal.

#### Procedure

- Update your Maven **pom.xml** file to use the **apicurio-registry-maven-plugin** to register an artifact. The following example shows registering Apache Avro and GraphQL schemas:

```
<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
          </artifact>
        </artifacts>
      </configuration>
    </execution>
  </executions>
</plugin>
```

```

        <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 Specify **register** as the execution goal to upload the schema artifact to the registry.
- 2 Specify the Service Registry URL with the **../apis/registry/v2** endpoint.
- 3 Specify the Service Registry artifact group ID. You can specify the **default** group if you do not want to use a unique group.
- 4 You can upload multiple artifacts using the specified group ID, artifact ID, and location.

### Additional resources

- For more details on the Service Registry Maven plug-in, see the [Registry demonstration example](#)

## 5.2. DOWNLOADING SCHEMA AND API ARTIFACTS USING THE MAVEN PLUG-IN

You can use the Maven plug-in to download artifacts from Service Registry. This is often useful, for example, when generating code from a registered schema.

### Procedure

- Update your Maven **pom.xml** file to use the **apicurio-registry-maven-plugin** to download an artifact. The following example shows downloading Apache Avro and GraphQL schemas.

```

<plugin>
  <groupId>io.apicurio</groupId>
  <artifactId>apicurio-registry-maven-plugin</artifactId>
  <version>${apicurio.version}</version>
  <executions>
    <execution>
      <phase>generate-sources</phase>
      <goals>
        <goal>download</goal> 1
      </goals>
    </execution>
  </executions>
  <configuration>
    <registryUrl>http://REGISTRY-URL/apis/registry/v2</registryUrl> 2
    <artifacts>
      <artifact>
        <groupId>TestGroup</groupId> 3
        <artifactId>FullNameRecord</artifactId> 4
        <file>${project.build.directory}/classes/record.avsc</file>
      </artifact>
    </artifacts>
  </configuration>
</plugin>

```

```

        <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 Specify **download** as the execution goal.
- 2 Specify the Service Registry URL with the `../apis/registry/v2` endpoint.
- 3 Specify the Service Registry artifact group ID. You can specify the **default** group if you do not want to use a unique group.
- 4 You can download multiple artifacts to a specified directory using the artifact ID.

#### Additional resources

- For more details on the Service Registry Maven plug-in, see the [Registry demonstration example](#)

## 5.3. TESTING SCHEMA AND API ARTIFACTS USING THE MAVEN PLUG-IN

You might want to verify that an artifact can be registered without actually making any changes. This is often useful when rules are configured in Service Registry. Testing the artifact results in a failure if the artifact content violates any of the configured rules.



#### NOTE

Even if the artifact passes the test, no content is added to Service Registry.

#### Procedure

- Update your Maven **pom.xml** file to use the **apicurio-registry-maven-plugin** to test an artifact. The following example shows testing an Apache Avro schema:

```

<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> ❶
  </goals>
  <configuration>
    <registryUrl>http://REGISTRY-URL/apis/registry/v2</registryUrl> ❷
    <artifacts>
      <artifact>
        <groupId>TestGroup</groupId> ❸
        <artifactId>FullNameRecord</artifactId>
        <file>${project.basedir}/src/main/resources/schemas/record.avsc</file> ❹
      </artifact>
      <artifact>
        <groupId>TestGroup</groupId>
        <artifactId>ExampleAPI</artifactId>
        <type>GRAPHQL</type>
        <file>${project.basedir}/src/main/resources/apis/example.graphql</file>
      </artifact>
    </artifacts>
  </configuration>
</execution>
</executions>
</plugin>

```

- ❶ Specify **test-update** as the execution goal to test the schema artifact.
- ❷ Specify the Service Registry URL with the **../apis/registry/v2** endpoint.
- ❸ Specify the Service Registry artifact group ID. You can specify the **default** group if you do not want to use a unique group.
- ❹ You can test multiple artifacts from specified directory using the artifact ID.

### Additional resources

- For more details on the Service Registry Maven plug-in, see the [Registry demonstration example](#)

## CHAPTER 6. MANAGING SERVICE REGISTRY CONTENT USING A JAVA CLIENT

This chapter explains how to use the Service Registry Java client:

- [Section 6.1, "Service Registry Java client"](#)
- [Section 6.2, "Writing Service Registry client applications"](#)
- [Section 6.3, "Service Registry Java client configuration"](#)

### 6.1. SERVICE REGISTRY JAVA CLIENT

You can manage artifacts stored in Service Registry using a Java client application. You can create, read, update, or delete artifacts stored in the registry using the Service Registry Java client classes. You can also perform admin functions using the client, such as managing global rules or importing and exporting registry data.

You can access the Service Registry Java client by adding the correct dependency to your project. For more details, see [Section 6.2, "Writing Service Registry client applications"](#).

The Service Registry client is implemented using the HTTP client provided by the JDK. This gives you the ability to customize its use, for example, by adding custom headers or enabling options for Transport Layer Security (TLS) authentication. For more details, see [Section 6.3, "Service Registry Java client configuration"](#).

### 6.2. WRITING SERVICE REGISTRY CLIENT APPLICATIONS

This section explains how to manage artifacts stored in Service Registry using a Java client application.

#### Prerequisites

- See [Chapter 1, Introduction to Service Registry](#)
- Service Registry must be installed and running in your environment

#### Procedure

1. Add the following dependency to your Maven project:

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

2. Create a registry client as follows:

```
public class ClientExample {

  private static final RegistryRestClient client;

  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** If you specify an example registry URL of <https://my-registry.my-domain.com>, the client will automatically append `/apis/registry/v2`.
  - 2** For more options when creating a Service Registry client, see the Java client configuration in the next section.
3. When the client is created, you can use all the operations from the Service Registry REST API through the client. For more details, see the [Apicurio Registry REST API documentation](#).

### Additional resources

- For an open source example of how to use and customize the Service Registry client, see the [Registry REST client demonstration example](#).
- For details on how to use the Service Registry Kafka client serializers/deserializers (SerDes) in producer and consumer applications, see [Chapter 7, Validating schemas using Kafka client serializers/deserializers in Java](#)

## 6.3. SERVICE REGISTRY JAVA CLIENT CONFIGURATION

The Service Registry Java client includes the following configuration options, based on the client factory:

Table 6.1. Service Registry Java client configuration options

Option	Description	Arguments
Plain client	Basic REST client used to interact with a running registry.	<b>baseUrl</b>
Client with custom configuration	Registry client using the configuration provided by the user.	<b>baseUrl, Map&lt;String Object&gt; configs</b>
Client with custom configuration and authentication	Registry client that accepts a map containing custom configuration. This is useful, for example, to add custom headers to the calls. This also requires providing an <b>auth</b> instance used to authenticate requests.	<b>baseUrl, Map&lt;String Object&gt; configs, Auth auth</b>

### Custom header configuration

To configure custom headers, you must add the **apicurio.registry.request.headers** prefix to the **configs** map key. For example, a key of **apicurio.registry.request.headers.Authorization** with a value of **Basic: xxxxx** results in a header of **Authorization** with value of **Basic: xxxxx**.

### TLS configuration options

You can configure Transport Layer Security (TLS) authentication for the Service Registry Java client using the following properties:

- **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**

# CHAPTER 7. VALIDATING SCHEMAS USING KAFKA CLIENT SERIALIZERS/DESERIALIZERS IN JAVA

Service Registry provides client serializers/deserializers (SerDes) for Kafka producer and consumer applications written in Java. Kafka producer applications use serializers to encode messages that conform to a specific event schema. Kafka consumer applications use deserializers to validate that messages have been serialized using the correct schema, based on a specific schema ID. This ensures consistent schema use and helps to prevent data errors at runtime.

This chapter explains how to use Kafka client SerDe in your producer and consumer client applications:

- [Section 7.1, “Kafka client applications and Service Registry”](#)
- [Section 7.2, “Strategies to look up a schema in Service Registry”](#)
- [Section 7.3, “Service Registry serializer/deserializer configuration”](#)
- [Section 7.4, “Using different client serializer/deserializer types”](#)
- [Section 7.4.1, “Configure Avro SerDe with Service Registry”](#)
- [Section 7.4.2, “Configure JSON Schema SerDe with Service Registry”](#)
- [Section 7.4.3, “Configure Protobuf SerDe with Service Registry”](#)
- [Section 7.5, “Registering a schema in Service Registry”](#)
- [Section 7.6, “Using a schema from a Kafka consumer client”](#)
- [Section 7.7, “Using a schema from a Kafka producer client”](#)
- [Section 7.8, “Using a schema from a Kafka Streams application”](#)

## Prerequisites

- You have read [Chapter 1, \*Introduction to Service Registry\*](#)
- You have installed Service Registry
- You have created Kafka producer and consumer client applications  
For more details on Kafka client applications, see [Using AMQ Streams on Openshift](#).

## 7.1. KAFKA CLIENT APPLICATIONS AND SERVICE REGISTRY

Service Registry decouples schema management from client application configuration. You can enable a Java client application to use a schema from Service Registry by specifying its URL in your client code.

You can store the schemas in the registry to serialize and deserialize messages, which are referenced from your client applications to ensure that the messages that they send and receive are compatible with those schemas. Kafka client applications can push or pull their schemas from Service Registry at runtime.

Schemas can evolve, so you can define rules in Service Registry, for example, to ensure that schema changes are valid and do not break previous versions used by applications. Service Registry checks for compatibility by comparing a modified schema with previous schema versions.

## Service Registry schema technologies

Service Registry provides schema registry support for schema technologies such as:

- Avro
- Protobuf
- JSON Schema

These schema technologies can be used by client applications through the Kafka client serializer/deserializer (SerDe) services provided by Service Registry. The maturity and usage of the SerDe classes provided by Service Registry might vary. The sections that follow provide more details about each schema type.

## Producer schema configuration

A producer client application uses a serializer to put the messages that it sends to a specific broker topic into the correct data format.

To enable a producer to use Service Registry for serialization:

- [Define and register your schema with Service Registry](#) (if it does not already exist).
- [Configure your producer client code](#) with the following:
  - URL of Service Registry
  - Service Registry serializer to use with messages
  - Strategy to map the Kafka message to a schema artifact in Service Registry
  - Strategy to look up or register the schema used for serialization in Service Registry

After registering your schema, when you start Kafka and Service Registry, you can access the schema to format messages sent to the Kafka broker topic by the producer. Alternatively, depending on configuration, the producer can automatically register the schema on first use.

If a schema already exists, you can create a new version using the registry REST API based on compatibility rules defined in Service Registry. Versions are used for compatibility checking as a schema evolves. A group ID, artifact ID, and version represents a unique tuple that identifies a schema.

## Consumer schema configuration

A consumer client application uses a deserializer to get the messages that it consumes from a specific broker topic into the correct data format.

To enable a consumer to use Service Registry for deserialization:

- [Define and register your schema with Service Registry](#) (if it does not already exist)
- [Configure the consumer client code](#) with the following:
  - URL of Service Registry
  - Service Registry deserializer to use with the messages
  - Input data stream for deserialization

## Retrieve schemas using a global ID

By default, the schema is retrieved from Service Registry by the deserializer using a global ID, which is specified in the message being consumed. The schema global ID can be located in the message headers or in the message payload, depending on the configuration of the producer application.

When locating the global ID in the message payload, the format of the data begins with a magic byte, used as a signal to consumers, followed by the global ID, and the message data as normal. For example:

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

Then when you start Kafka and Service Registry, you can access the schema to format messages received from the Kafka broker topic.

### Retrieve schemas using a content ID

Alternatively, you can configure to retrieve schemas from Service Registry based on the content ID, which is the unique ID of the artifact content. While the global ID is the unique ID of an artifact version.

The content ID does not uniquely identify a version, but uniquely identifies the version content only. If multiple versions share the exact same content, they have a different global ID but the same content ID. The Confluent schema registry uses content ID by default.

## 7.2. STRATEGIES TO LOOK UP A SCHEMA IN SERVICE REGISTRY

The Kafka client serializer uses *lookup strategies* to determine the artifact ID and global ID under which the message schema is registered in Service Registry. For a given topic and message, you can use different implementations of the **ArtifactResolverStrategy** Java interface to return a reference to an artifact in the registry.

The classes for each strategy are in the **io.apicurio.registry.serde.strategy** package. Specific strategy classes for Avro SerDe are in the **io.apicurio.registry.serde.avro.strategy package**. The default strategy is the **TopicIdStrategy**, which looks for Service Registry artifacts with the same name as the Kafka topic receiving messages.

### Example

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

- The **topic** parameter is the name of the Kafka topic receiving the message.
- The **isKey** parameter is **true** when the message key is serialized, and **false** when the message value is serialized.
- The **schema** parameter is the schema of the message serialized or deserialized.
- The **ArtifactReference** returned contains the artifact ID under which the schema is registered.

Which lookup strategy you use depends on how and where you store your schema. For example, you might use a strategy that uses a *record ID* if you have different Kafka topics with the same Avro message type.

### ArtifactResolverStrategy interface

The artifact resolver strategy provides a way to map the Kafka topic and message information to an artifact in Service Registry. The common convention for the mapping is to combine the Kafka topic name with the **key** or **value**, depending on whether the serializer is used for the Kafka message key or value.

However, you can use alternative conventions for the mapping by using a strategy provided by Service Registry, or by creating a custom Java class that implements **io.apicurio.registry.serde.strategy.ArtifactResolverStrategy**.

### Strategies to return an artifact reference

Service Registry provides the following strategies to return an artifact reference based on an implementation of **ArtifactResolverStrategy**:

#### RecordIdStrategy

Avro-specific strategy that uses the full name of the schema.

#### TopicRecordIdStrategy

Avro-specific strategy that uses the topic name and the full name of the schema.

#### TopicIdStrategy

Default strategy that uses the topic name and **key** or **value** suffix.

#### SimpleTopicIdStrategy

Simple strategy that only uses the topic name.

### DefaultSchemaResolver interface

The default schema resolver locates and identifies the specific version of the schema registered under the artifact reference provided by the artifact resolver strategy. Every version of every artifact has a single globally unique identifier that can be used to retrieve the content of that artifact. This global ID is included in every Kafka message so that a deserializer can properly fetch the schema from Apicurio Registry.

The default schema resolver can look up an existing artifact version, or it can register one if not found, depending on which strategy is used. You can also provide your own strategy by creating a custom Java class that implements **io.apicurio.registry.serde.SchemaResolver**. However, it is recommended to use the **DefaultSchemaResolver** and specify configuration properties instead.

### Strategies to return a global ID

When using the **DefaultSchemaResolver**, you can configure its behavior using application properties. The following table shows some commonly used examples:

Table 7.1. Service Registry global ID configuration options

Property	Type	Description	Default
<b>apicurio.registry.find-latest</b>	<b>boolean</b>	Specify whether the serializer tries to find the latest artifact in the registry for the corresponding group ID and artifact ID.	<b>false</b>



Property	Type	Description	Default
<b>apicurio.registry.use-id</b>	<b>String</b>	Instructs the serializer to write the specified ID to Kafka and instructs the deserializer to use this ID to find the schema.	-
<b>apicurio.registry.auto-register</b>	<b>boolean</b>	Specify whether the serializer tries to create an artifact in the registry. The JSON Schema serializer does not support this.	<b>false</b>
<b>apicurio.registry.check-period-ms</b>	<b>String</b>	Specify how long to cache the global ID in milliseconds. If not configured, the global ID is fetched every time.	-

## TIP

You can configure application properties as Java system properties or include them in the Quarkus **application.properties** file. For more details, see the [Quarkus documentation](#).

## Additional resources

- For more details, see the [SerdeConfig Java class](#)

## 7.3. SERVICE REGISTRY SERIALIZER/DESERIALIZER CONFIGURATION

You can configure specific client serializer/deserializer (SerDe) services and schema lookup strategies directly in a client application using the example constants shown in this section. Alternatively, you can configure the corresponding Service Registry application properties in a file or an instance.

The following sections show examples of SerDe constants and configuration options.

### Configuration for SerDe services

```
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** The required URL of Service Registry.
- 2** Extends ID handling to support other ID formats and make them compatible with Service Registry SerDe services. For example, changing the default ID format from **Long** to **Integer** supports the Confluent ID format.
- 3** Simplifies the handling of Confluent IDs. If set to **true**, an **Integer** is used for the global ID lookup. The setting should not be used with the **ID\_HANDLER** option.

### Additional resources

- For more details, see the [SerdeConfig Java class](#)

### Configuration for SerDe lookup strategies

```
public class SerdeConfig {

    public static final String ARTIFACT_RESOLVER_STRATEGY = "apicurio.registry.artifact-resolver-
strategy";
    ...
}
```

### Additional resources

- For more details, see [Section 7.2, “Strategies to look up a schema in Service Registry”](#)

### Configuration for Kafka converters

```
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** The required serializer to use with the Service Registry Kafka converter.

**2** The required deserializer to use with the Service Registry Kafka converter.

### Additional resources

- For more details, see the [SerdeBasedConverter Java class](#)

### Configuration for different schema types

For details on how to configure SerDe for different schema technologies, see the following:

- [Section 7.4.1, “Configure Avro SerDe with Service Registry”](#)
- [Section 7.4.2, “Configure JSON Schema SerDe with Service Registry”](#)
- [Section 7.4.3, “Configure Protobuf SerDe with Service Registry”](#)

## 7.4. USING DIFFERENT CLIENT SERIALIZER/DESERIALIZER TYPES

When using schemas in your Kafka client applications, you must choose which specific schema type to use, depending on your use case. Service Registry provides SerDe Java classes for Apache Avro, JSON Schema, and Google Protobuf. The following sections explain how to configure Kafka applications to use each type.

You can also use Kafka to implement custom serializer and deserializer classes, and leverage Service Registry functionality using the Service Registry REST Java client.

### Kafka application configuration for serializers/deserializers

Using the SerDe classes provided by Service Registry in your Kafka application involves setting the correct configuration properties. The following simple Avro examples show how to configure a serializer in a Kafka producer application and how to configure a deserializer in a Kafka consumer application.

### Example serializer configuration in a Kafka producer

```
// 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;
}
```

### Example deserializer configuration in a Kafka consumer

```
// 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;
}

```

### Additional resources

- For an example application, see the [Simple Avro example](#)

## 7.4.1. Configure Avro SerDe with Service Registry

Service Registry provides the following Kafka client serializer and deserializer classes for Apache Avro:

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

### Configure the Avro serializer

You can configure the Avro serializer class with the following:

- Service Registry URL
- Artifact resolver strategy
- ID location
- ID encoding
- Avro datum provider
- Avro encoding

#### ID location

The serializer passes the unique ID of the schema as part of the Kafka message so that consumers can use the correct schema for deserialization. The ID can be in the message payload or in the message headers. The default location is the message payload. To send the ID in the message headers, set the following configuration property:

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

The property name is **apicurio.registry.headers.enabled**.

#### ID encoding

You can customize how the schema ID is encoded when passing it in the Kafka message body. Set the **apicurio.registry.id-handler** configuration property to a class that implements the **io.apicurio.registry.serde.IdHandler** interface. Service Registry provides the following implementations:

- **io.apicurio.registry.serde.DefaultIdHandler**: Stores the ID as an 8-byte long
- **io.apicurio.registry.serde.Legacy4ByteIdHandler**: Stores the ID as an 4-byte integer

Service Registry represents the schema ID as a long, but for legacy reasons, or for compatibility with other registries or SerDe classes, you might want to use 4 bytes when sending the ID.

### Avro datum provider

Avro provides different datum writers and readers to write and read data. Service Registry supports three different types:

- Generic
- Specific
- Reflect

The Service Registry **AvroDatumProvider** is the abstraction of which type is used, where **DefaultAvroDatumProvider** is used by default.

You can set the following configuration options:

- **apicurio.registry.avro-datum-provider**: Specifies a fully-qualified Java class name of the **AvroDatumProvider** implementation, for example **io.apicurio.registry.serde.avro.ReflectAvroDatumProvider**
- **apicurio.registry.use-specific-avro-reader**: Set to **true** to use a specific type when using **DefaultAvroDatumProvider**

### Avro encoding

When using Avro to serialize data, you can use the Avro binary encoding format to ensure the data is encoded in as efficient a format as possible. Avro also supports encoding the data as JSON, which makes it easier to inspect the payload of each message, for example, for logging or debugging.

You can set the Avro encoding by configuring the **apicurio.registry.avro.encoding** property with a value of **JSON** or **BINARY**. The default is **BINARY**.

### Configure the Avro deserializer

You must configure the Avro deserializer class to match the following configuration settings of the serializer:

- Service Registry URL
- ID encoding
- Avro datum provider
- Avro encoding

See the serializer section for these configuration options. The property names and values are the same.



#### NOTE

The following options are not required when configuring the deserializer:

- Artifact resolver strategy
- ID location

The deserializer class can determine the values for these options from the message. The strategy is not required because the serializer is responsible for sending the ID as part of the message.

The ID location is determined by checking for the magic byte at the start of the message payload. If that byte is found, the ID is read from the message payload using the configured handler. If the magic byte is not found, the ID is read from the message headers.

### Additional resources

- For more details on Avro configuration, see the [AvroKafkaSerdeConfig Java class](#)
- For an example application, see the [Simple Avro example](#)

## 7.4.2. Configure JSON Schema SerDe with Service Registry

Service Registry provides the following Kafka client serializer and deserializer classes for JSON Schema:

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

Unlike Apache Avro, JSON Schema is not a serialization technology, but is instead a validation technology. As a result, configuration options for JSON Schema are quite different. For example, there is no encoding option, because data is always encoded as JSON.

### Configure the JSON Schema serializer

You can configure the JSON Schema serializer class as follows:

- Service Registry URL
- Artifact resolver strategy
- Schema validation

The only non-standard configuration property is JSON Schema validation, which is enabled by default. You can disable this by setting **apicurio.registry.serde.validation-enabled** to **"false"**. For example:

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

### Configure the JSON Schema deserializer

You can configure the JSON Schema deserializer class as follows:

- Service Registry URL
- Schema validation
- Class for deserializing data

You must provide the location of Service Registry so that the schema can be loaded. The other configuration is optional.

**NOTE**

Deserializer validation only works if the serializer passes the global ID in the Kafka message, which will only happen when validation is enabled in the serializer.

**Additional resources**

- For more details, see the [JsonSchemaKafkaDeserializerConfig Java class](#)
- For an example application, see the [Simple JSON Schema example](#)

**7.4.3. Configure Protobuf SerDe with Service Registry**

Service Registry provides the following Kafka client serializer and deserializer classes for Google Protobuf:

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

**Configure the Protobuf serializer**

You can configure the Protobuf serializer class as follows:

- Service Registry URL
- Artifact resolver strategy
- ID location
- ID encoding
- Schema validation

For details on these configuration options, see the following sections:

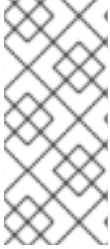
- [Section 7.3, “Service Registry serializer/deserializer configuration”](#)
- [Section 7.4.1, “Configure Avro SerDe with Service Registry”](#)

**Configure the Protobuf deserializer**

You must configure the Protobuf deserializer class to match the following configuration settings in the serializer:

- Service Registry URL
- ID encoding

The configuration property names and values are the same as for the serializer.

**NOTE**

The following options are not required when configuring the deserializer:

- Artifact resolver strategy
- ID location

The deserializer class can determine the values for these options from the message. The strategy is not required because the serializer is responsible for sending the ID as part of the message.

The ID location is determined by checking for the magic byte at the start of the message payload. If that byte is found, the ID is read from the message payload using the configured handler. If the magic byte is not found, the ID is read from the message headers.

**NOTE**

The Protobuf deserializer does not deserialize to your exact Protobuf Message implementation, but rather to a **DynamicMessage** instance. There is no appropriate API to do otherwise.

**Additional resources**

- For example applications, see the [Protobuf Bean and Protobuf Find Latest examples](#)

## 7.5. REGISTERING A SCHEMA IN SERVICE REGISTRY

After you have defined a schema in the appropriate format, such as Apache Avro, you can add the schema to Service Registry.

You can add the schema using the following approaches:

- Service Registry web console
- curl command using the Service Registry REST API
- Maven plug-in supplied with Service Registry
- Schema configuration added to your client code

Client applications cannot use Service Registry until you have registered your schemas.

**Service Registry web console**

When Service Registry is installed, you can connect to the web console from the **ui** endpoint:

**`http://MY-REGISTRY-URL/ui`**

From the console, you can add, view and configure schemas. You can also create the rules that prevent invalid content being added to the registry.

**Curl command example**

```
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 Simple Avro schema artifact.
- 2 OpenShift route name that exposes Service Registry.

## Maven plug-in example

```
<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 Specify **register** as the execution goal to upload the schema artifact to the registry.
- 2 Specify the Service Registry URL with the **../apis/registry/v2** endpoint.
- 3 Specify the Service Registry artifact group ID.
- 4 You can upload multiple artifacts using the specified group ID, artifact ID, and location.

## Configuration using a producer client example

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

**1** You can register properties against more than one URL node.

**2** Check to see if the schema already exists based on the artifact ID.

## 7.6. USING A SCHEMA FROM A KAFKA CONSUMER CLIENT

This procedure describes how to configure a Kafka consumer client written in Java to use a schema from Service Registry.

### Prerequisites

- Service Registry is installed
- The schema is registered with Service Registry

### Procedure

1. Configure the client with the URL of Service Registry. For example:

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

2. Configure the client with the Service Registry deserializer. For example:

```
// 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** The deserializer provided by Service Registry.
- 2** The deserialization is in Apache Avro JSON format.

## 7.7. USING A SCHEMA FROM A KAFKA PRODUCER CLIENT

This procedure describes how to configure a Kafka producer client written in Java to use a schema from Service Registry.

### Prerequisites

- Service Registry is installed
- The schema is registered with Service Registry

### Procedure

1. Configure the client with the URL of Service Registry. For example:

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

2. Configure the client with the serializer, and the strategy to look up the schema in Service Registry. For example:

```
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, FindLatestIdStrategy.class.getName());
3
```

- 1** The serializer for the message key provided by Service Registry.
- 2** The serializer for the message value provided by Service Registry.
- 3** The lookup strategy to find the global ID for the schema.

## 7.8. USING A SCHEMA FROM A KAFKA STREAMS APPLICATION

This procedure describes how to configure a Kafka Streams client written in Java to use an Apache Avro schema from Service Registry.

### Prerequisites

- Service Registry is installed
- The schema is registered with Service Registry

## Procedure

1. Create and configure a Java client with the Service Registry URL:

```
String registryUrl = "https://registry.example.com/apis/registry/v2";  
RegistryService client = RegistryClient.cached(registryUrl);
```

2. Configure the serializer and 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** The Avro serializer provided by Service Registry.
- 2** The Avro deserializer provided by Service Registry.
- 3** Configures the Service Registry URL and the Avro reader for deserialization in Avro format.

3. Create the Kafka Streams client:

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

## CHAPTER 8. SERVICE REGISTRY ARTIFACT REFERENCE

This chapter provides details on the supported artifact types, states, metadata, and content rules that are stored in Service Registry.

- [Section 8.1, "Service Registry artifact types"](#)
- [Section 8.2, "Service Registry artifact states"](#)
- [Section 8.3, "Service Registry artifact metadata"](#)
- [Section 8.4, "Service Registry content rule types"](#)
- [Section 8.5, "Service Registry content rule maturity"](#)

### Additional resources

- For more detailed information, see the [Apicurio Registry REST API documentation](#)

## 8.1. SERVICE REGISTRY ARTIFACT TYPES

You can store and manage the following artifact types in Service Registry:

Table 8.1. Service Registry artifact types

Type	Description
<b>ASYNCAPI</b>	AsyncAPI specification
<b>AVRO</b>	Apache Avro schema
<b>GRAPHQL</b>	GraphQL schema
<b>JSON</b>	JSON Schema
<b>KCONNECT</b>	Apache Kafka Connect schema
<b>OPENAPI</b>	OpenAPI specification
<b>PROTOBUF</b>	Google protocol buffers schema
<b>WSDL</b>	Web Services Definition Language
<b>XSD</b>	XML Schema Definition

## 8.2. SERVICE REGISTRY ARTIFACT STATES

These are the valid artifact states in Service Registry:

Table 8.2. Service Registry artifact states

State	Description
<b>ENABLED</b>	Basic state, all the operations are available.
<b>DISABLED</b>	The artifact and its metadata is viewable and searchable using the Service Registry web console, but its content cannot be fetched by any client.
<b>DEPRECATED</b>	The artifact is fully usable but a header is added to the REST API response whenever the artifact content is fetched. The Service Registry Rest Client will also log a warning whenever it sees deprecated content.

### 8.3. SERVICE REGISTRY ARTIFACT METADATA

When an artifact is added to Service Registry, a set of metadata properties is stored along with the artifact content. This metadata consists of a set of generated read-only properties, along with some properties that you can set.

Table 8.3. Service Registry metadata properties

Property	Type	Editable
<b>id</b>	string	false
<b>type</b>	ArtifactType	false
<b>state</b>	ArtifactState	true
<b>version</b>	integer	false
<b>createdBy</b>	string	false
<b>createdOn</b>	date	false
<b>modifiedBy</b>	string	false
<b>modifiedOn</b>	date	false
<b>name</b>	string	true
<b>description</b>	string	true
<b>labels</b>	array of string	true
<b>properties</b>	map	true

### Updating artifact metadata

- You can use the Service Registry REST API to update the set of editable properties using the metadata endpoints.
- You can edit the **state** property only by using the state transition API. For example, you can mark an artifact as **deprecated** or **disabled**.

### Additional resources

For more details, see the `/artifacts/{artifactId}/meta` sections in the [Apicurio Registry REST API documentation](#).

## 8.4. SERVICE REGISTRY CONTENT RULE TYPES

You can specify the following rule types to govern content evolution in Service Registry:

Table 8.4. Service Registry content rule types

Type	Description
<b>VALIDITY</b>	<p>Validate data before adding it to the registry. The possible configuration values for this rule are:</p> <ul style="list-style-type: none"> <li>• <b>FULL</b>: The validation is both syntax and semantic.</li> <li>• <b>SYNTAX_ONLY</b>: The validation is syntax only.</li> </ul>

Type	Description
<b>COMPATIBILITY</b>	<p>Ensure that newly added artifacts are compatible with previously added versions. The possible configuration values for this rule are:</p> <ul style="list-style-type: none"> <li>● <b>FULL</b>: The new artifact is forward and backward compatible with the most recently added artifact.</li> <li>● <b>FULL_TRANSITIVE</b>: The new artifact is forward and backward compatible with all previously added artifacts.</li> <li>● <b>BACKWARD</b>: Clients using the new artifact can read data written using the most recently added artifact.</li> <li>● <b>BACKWARD_TRANSITIVE</b>: Clients using the new artifact can read data written using all previously added artifacts.</li> <li>● <b>FORWARD</b>: Clients using the most recently added artifact can read data written using the new artifact.</li> <li>● <b>FORWARD_TRANSITIVE</b>: Clients using all previously added artifacts can read data written using the new artifact.</li> <li>● <b>NONE</b>: All backward and forward compatibility checks are disabled.</li> </ul>

## 8.5. SERVICE REGISTRY CONTENT RULE MATURITY

Not all content rules are fully implemented for every artifact type supported by Service Registry. The following table shows the current maturity level for each rule and artifact type.

Table 8.5. Service Registry content rule maturity matrix

Artifact type	Validity rule	Compatibility rule
Avro	Full	Full
Protobuf	Full	None
JSON Schema	Full	Full
OpenAPI	Full	None
AsyncAPI	Syntax Only	None
GraphQL	Syntax Only	None



Artifact type	Validity rule	Compatibility rule
Kafka Connect	Syntax Only	None
WSDL	Syntax Only	None
XSD	Syntax Only	None

## APPENDIX A. USING YOUR SUBSCRIPTION

Service Registry is provided through a software subscription. To manage your subscriptions, access your account at the Red Hat Customer Portal.

### Accessing your account

1. Go to [access.redhat.com](https://access.redhat.com).
2. If you do not already have an account, create one.
3. Log in to your account.

### Activating a subscription

1. Go to [access.redhat.com](https://access.redhat.com).
2. Navigate to **My Subscriptions**.
3. Navigate to **Activate a subscription** and enter your 16-digit activation number.

### Downloading ZIP and TAR files

To access ZIP or TAR files, use the customer portal to find the relevant files for download. If you are using RPM packages, this step is not required.

1. Open a browser and log in to the Red Hat Customer Portal **Product Downloads** page at [access.redhat.com/downloads](https://access.redhat.com/downloads).
2. Locate the **Red Hat Integration** entries in the **Integration and Automation** category.
3. Select the desired Service Registry product. The **Software Downloads** page opens.
4. Click the **Download** link for your component.

### Registering your system for packages

To install RPM packages on Red Hat Enterprise Linux, your system must be registered. If you are using ZIP or TAR files, this step is not required.

1. Go to [access.redhat.com](https://access.redhat.com).
2. Navigate to **Registration Assistant**.
3. Select your OS version and continue to the next page.
4. Use the listed command in your system terminal to complete the registration.

To learn more see [How to Register and Subscribe a System to the Red Hat Customer Portal](#) .