Red Hat Integration 2020-Q4

Getting Started with Service Registry

Service Registry 1.1

Last Updated: 2021-01-22

Red Hat Integration 2020-Q4 Getting Started with Service Registry

Service Registry 1.1

Abstract

This guide introduces Service Registry and explains how to install with your chosen registry storage. It shows 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 to use Kafka client serializers and deserializers in your consumer and producer applications. It also describes Service Registry content types, rule configuration, and environment variables on OpenShift.

Table of Contents

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:

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

- Support for multiple payload formats for standard event schemas and API specifications

- Pluggable storage options including AMQ Streams, embedded Infinispan, 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
- Client serializers/deserializers (Serdes) to validate Kafka and other message types at runtime
- Cloud-native Quarkus Java runtime for low memory footprint and fast deployment times
- Compatibility with existing Confluent schema registry client applications
- Operator-based installation of Service Registry on OpenShift

1.2. SCHEMA AND API ARTIFACTS IN SERVICE REGISTRY

The items stored in Service Registry, such as event schemas and API specifications, 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 contract is added as an artifact in the registry, client applications can then use that schema or API contract to validate that 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 protocol buffers, GraphQL, AsyncAPI, OpenAPI, and others. For more details, see Chapter 14, *Service Registry artifact reference*.

1.3. SERVICE REGISTRY STORAGE OPTIONS

Service Registry provides the following underlying storage implementations for registry artifacts:

Table 1.1. Service Registry storage options

Storage option	Release
Kafka Streams-based storage in AMQ Streams 1.5	General Availability
Cache-based storage in embedded Infinispan 10	Technical Preview only
Java Persistence API-based storage in PostgreSQL 12 database	Technical Preview only

IMPORTANT

Service Registry storage in Infinispan or PostgreSQL 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.

Additional resources

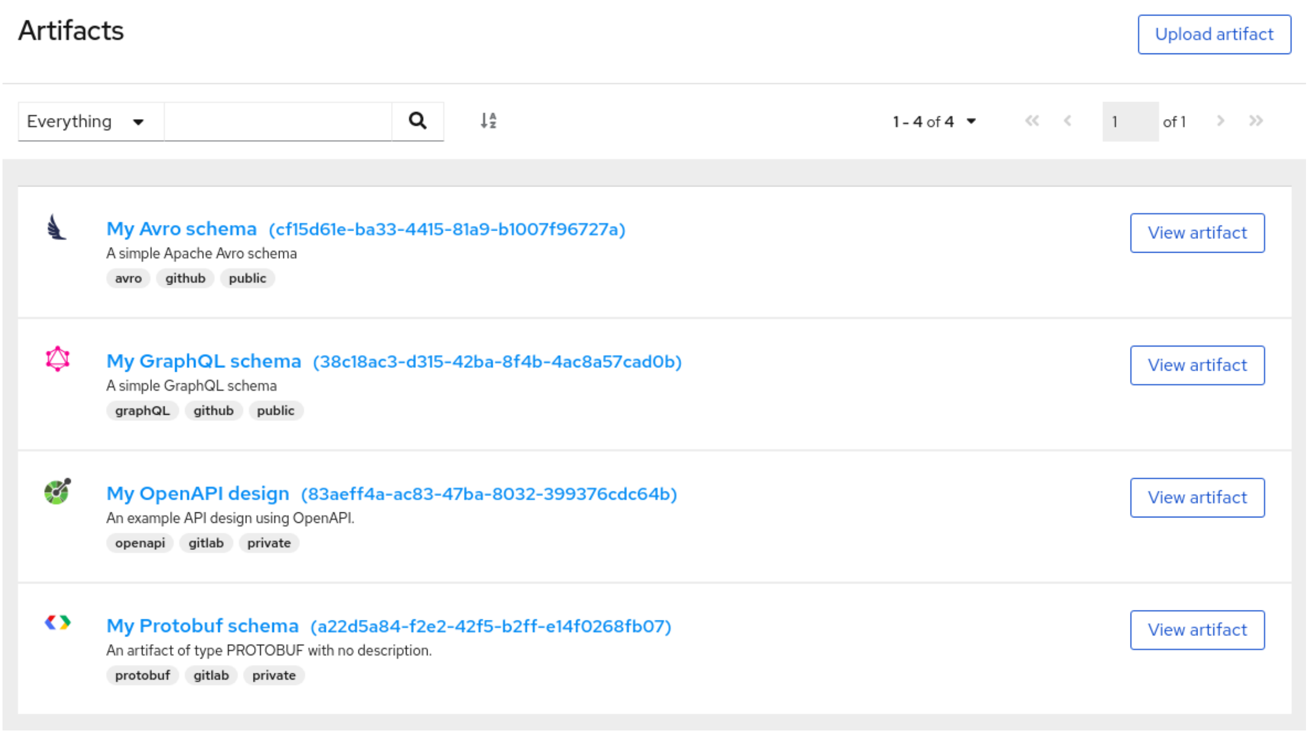
- Chapter 4, *Installing Service Registry on OpenShift*
- Chapter 5, *Deploying Service Registry storage in AMQ Streams*

1.4. MANAGE CONTENT USING SERVICE REGISTRY WEB CONSOLE

You can use the Service Registry web console to browse and search the artifacts stored in the registry, and to upload new artifacts and artifact versions. You can search for artifacts by label, name, and description. You can also view an artifact's content, view all of its available versions, or download an artifact file locally.

You can also use the Service Registry web console to configure optional rules for registry content, both globally and for each artifact. These optional rules for content validation and compatibility are applied when new artifacts or artifact versions are uploaded to the registry. For more details, see Chapter 14, *Service Registry artifact reference* .

Figure 1.1. Service Registry web console



My Avro schema (cf15d61e-ba33-4415-81a9-b1007f96727a)
A simple Apache Avro schema
avro github public

My GraphQL schema (38c18ac3-d315-42ba-8f4b-4ac8a57cad0b)
A simple GraphQL schema
graphQL github public

My OpenAPI design (83aeff4a-ac83-47ba-8032-399376cdc64b)
An example API design using OpenAPI.
openapi gitlab private

My Protobuf schema (a22d5a84-f2e2-42f5-b2ff-e14f0268fb07)
An artifact of type PROTOBUF with no description.
protobuf gitlab private

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

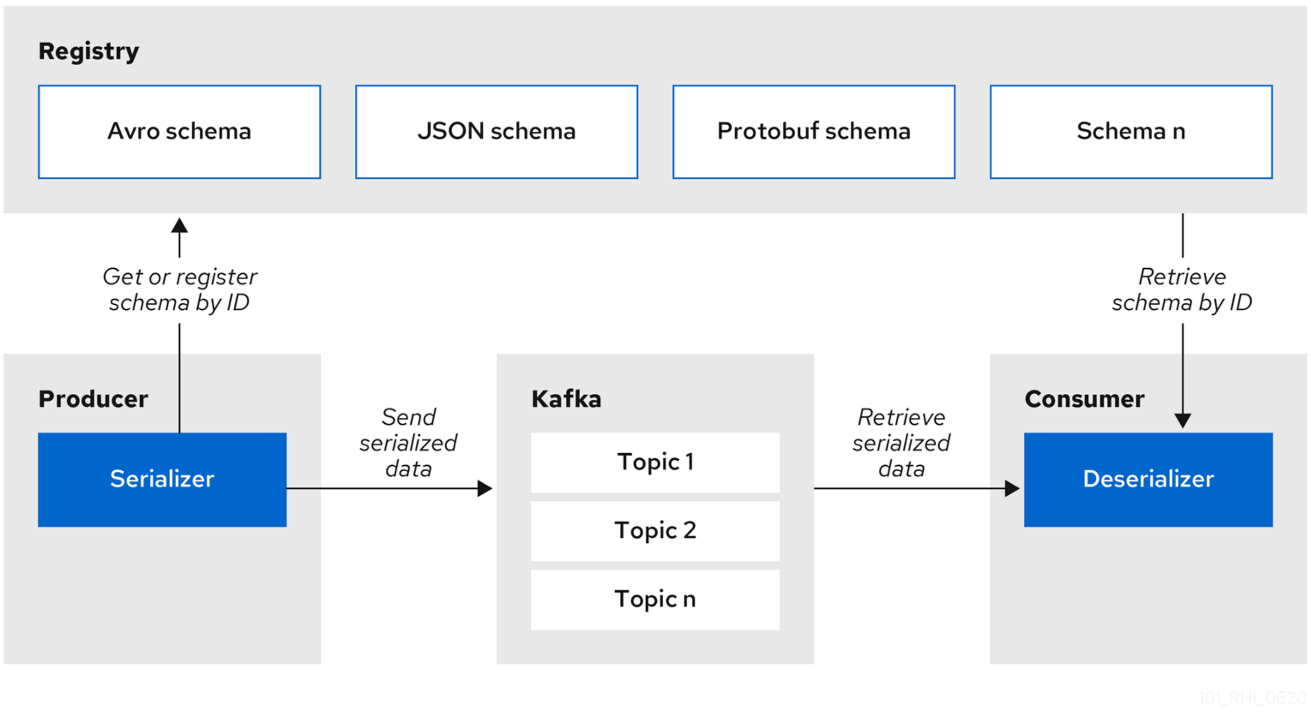
Additional resources

- Chapter 9, *Managing Service Registry content using the web console*

1.5. 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 serializer/deserializer 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 serializer/deserializer implementations for these message types, which Kafka client developers can use to integrate with the registry. These implementations include custom **io.apicurio.registry.utils.serde** Java classes for each supported message type, which client applications can use to pull schemas from the registry at runtime for validation.

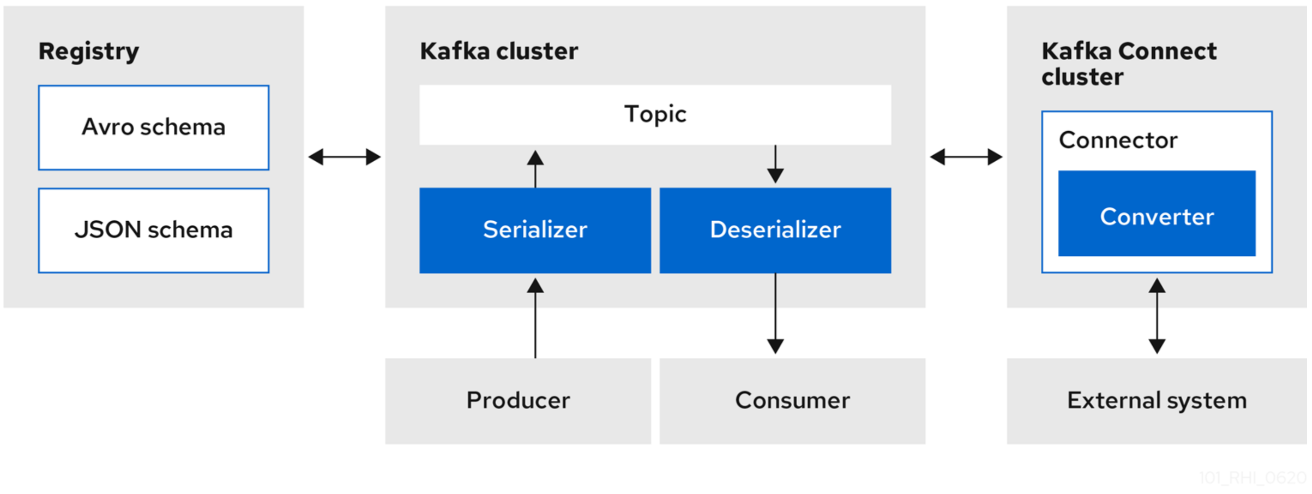
Additional resources

- Chapter 13, *Validating schemas using Kafka client serializers/deserializers*

1.6. 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



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

1.7. SERVICE REGISTRY DEMONSTRATION EXAMPLES

Service Registry provides an open source demonstration of Apache Avro serialization/deserialization with storage in Apache Kafka Streams. This example shows how the serializer/deserializer obtains the

Avro schema from the registry at runtime and uses it to serialize and deserialize Kafka messages. For more details, see https://github.com/Apicurio/apicurio-registry-demo.

Service Registry also provides the following example applications:

- Simple Avro example
- Simple JSON Schema example
- Confluent Serdes integration
- Avro bean example
- Custom ID strategy example
- Simple Avro Maven example
- REST client example

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

1.8. SERVICE REGISTRY AVAILABLE DISTRIBUTIONS

Table 1.2. Service Registry Operator and images

Distribution	Location	Release
Service Registry Operator	OpenShift web console under **Operators → OperatorHub**	General Availability
Container image for Service Registry Operator	Red Hat Ecosystem Catalog	General Availability
Container image for Kafka storage in AMQ Streams	Red Hat Ecosystem Catalog	General Availability
Container image for embedded Infinispan storage	Red Hat Ecosystem Catalog	Technical Preview only
Container image for JPA storage in PostgreSQL	Red Hat Ecosystem Catalog	Technical Preview only

IMPORTANT

Service Registry storage in Infinispan or PostgreSQL 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.

Table 1.3. Service Registry zip downloads

Distribution	Location	Release
Example custom resource definitions for installation	Software Downloads for Red Hat Integration	General Availability and Technical Preview
Kafka Connect converters	Software Downloads for Red Hat Integration	General Availability
Maven repository	Software Downloads for Red Hat Integration	General Availability
Source code	Software Downloads for Red Hat Integration	General Availability

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:

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 14, *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 9, *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. SERVICE REGISTRY QUICKSTART

This chapter explains how to quickly install Service Registry Operator using the OpenShift command line. This quickstart example deploys Service Registry using the embedded Infinispan storage option:

- Section 3.1, "Quickstart Service Registry Operator installation"
- Section 3.2, "Quickstart Service Registry deployment"

NOTE

The recommended installation option for production environments is Section 4.1, "Installing Service Registry from the OpenShift OperatorHub".

The recommended storage option for production environments is AMQ Streams. For details, see Chapter 5, *Deploying Service Registry storage in AMQ Streams* .

3.1. QUICKSTART SERVICE REGISTRY OPERATOR INSTALLATION

You can quickly deploy the Service Registry Operator on the command line, without the Operator Lifecycle Manager, by using a downloaded set of installation files and examples.

Prerequisites

- You must go to Red Hat Integration Downloads, select the product version, and download the Service Registry CRDs **.zip** file.

Procedure

1. Create a project for the installation, for example, **service-registry**:

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

2. Set the namespace in **install/cluster_role_binding.yaml** by replacing **{NAMESPACE}** with **service-registry**.

3. Apply the files located in the **install/** folder:

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

3.2. QUICKSTART SERVICE REGISTRY DEPLOYMENT

To quickly create a new Service Registry deployment, use the embedded Infinispan storage option, which does not require an external storage to be configured as a prerequisite.

Prerequisites

- Ensure that the Service Registry Operator is already installed.

Procedure

1. Create an **ApicurioRegistry** custom resource (CR) in the same namespace that the Operator is deployed:

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

Example CR for Infinispan storage

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

2. Access the automatically created route for the Service Registry web console. For example:

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

CHAPTER 4. INSTALLING SERVICE REGISTRY ON OPENSHIFT

This chapter explains how to install Service Registry:

- Section 4.1, “Installing Service Registry from the OpenShift OperatorHub”

Prerequisites

- Chapter 1, *Introduction to Service Registry*

NOTE

You can install more than one instance of Service Registry depending on your environment. The number of instances depends on the number and type of artifacts stored in Service Registry and on your chosen storage option.

4.1. INSTALLING SERVICE REGISTRY FROM THE OPENSHIFT OPERATORHUB

You can install the Service Registry Operator on your OpenShift cluster from the OperatorHub. The OperatorHub is available from the OpenShift Container Platform web console and provides an interface for cluster administrators to discover and install Operators. For more details, see the OpenShift documentation.

Prerequisites

- You must have cluster administrator access to an OpenShift cluster.

Procedure

1. In the OpenShift Container Platform web console, log in using an account with cluster administrator privileges.
2. Create a new OpenShift project:
   a. In the left navigation menu, click **Home** > **Project** > **Create Project**.
   b. Enter a project name, for example, **my-project**, and click **Create**.
3. In the left navigation menu, click **Operators** > **OperatorHub**.
4. In the **Filter by keyword** text box, enter **registry** to find the **Red Hat Integration - Service Registry Operator**.
5. Read the information about the Operator, and click **Install** to display the Operator subscription page.
6. Select your subscription settings, for example:
   - **Update Channel** > Select one of the following channels:
     - **serviceregistry-1**: All minor and patch updates, such as 1.1.0 and 1.0.1. An installation on 1.0.x automatically upgrades to 1.1.x.

    - **serviceregistry-1.0**: Patch updates only, such as 1.0.1 and 1.0.2. An installation on 1.0.x automatically ignores 1.1.x.
    - **serviceregistry-1.1**: Patch updates only, such as 1.1.1 and 1.1.2. An installation on 1.1.x automatically ignores 1.0.x.
  - **Installation Mode** > **A specific namespace on the cluster**> **my-project**
  - **Approval Strategy** > **Manual**

7. Click **Install**, and wait a few moments until the Operator is ready for use.

Additional resources

- Adding Operators to an OpenShift cluster
- Apicurio Registry Operator community in GitHub

CHAPTER 5. DEPLOYING SERVICE REGISTRY STORAGE IN AMQ STREAMS

This chapter explains how to install and configure Service Registry storage in AMQ Streams.

- Section 5.1, "Installing AMQ Streams from the OpenShift OperatorHub"
- Section 5.2, "Configuring Service Registry with AMQ Streams storage on OpenShift"
- Section 5.3, "Configuring TLS security with Service Registry storage in AMQ Streams"
- Section 5.4, "Configuring SCRAM security with Service Registry storage in AMQ Streams"

IMPORTANT

Service Registry storage in AMQ Streams is the recommended storage option for production environments.

Prerequisites

- Chapter 4, *Installing Service Registry on OpenShift*

5.1. INSTALLING AMQ STREAMS FROM THE OPENSHIFT OPERATORHUB

If you do not already have AMQ Streams installed, you can install the AMQ Streams Operator on your OpenShift cluster from the OperatorHub. The OperatorHub is available from the OpenShift Container Platform web console and provides an interface for cluster administrators to discover and install Operators. For more details, see the OpenShift documentation.

Prerequisites

- You must have cluster administrator access to an OpenShift cluster
- See Using AMQ Streams on OpenShift for detailed information on installing AMQ Streams. This section shows a simple example of installing using the OpenShift OperatorHub.

Procedure

1. In the OpenShift Container Platform web console, log in using an account with cluster administrator privileges.
2. Change to the OpenShift project in which Service Registry is installed. For example, from the **Project** drop-down, select **my-project**.
3. In the left navigation menu, click **Operators** > **OperatorHub**.
4. In the **Filter by keyword** text box, enter **AMQ Streams** to find the **Red Hat Integration - AMQ Streams** Operator.
5. Read the information about the Operator, and click **Install** to display the Operator subscription page.
6. Select your subscription settings, for example:

- **Update Channel** > **amq-streams-1.5.x**
- **Installation Mode** > **A specific namespace on the cluster**> **my-project**
- **Approval Strategy** > **Manual**

7. Click **Install**, and wait a few moments until the Operator is ready for use.

Additional resources

- Adding Operators to an OpenShift cluster
- Using AMQ Streams on OpenShift

5.2. CONFIGURING SERVICE REGISTRY WITH AMQ STREAMS STORAGE ON OPENSHIFT

This section explains how to configure Kafka-based storage for Service Registry using AMQ Streams on OpenShift. This storage option is suitable for production environments when **persistent** storage is configured for the Kafka cluster on OpenShift. You can install Service Registry in an existing Kafka cluster or create a new Kafka cluster, depending on your environment.

Prerequisites

- You must have an OpenShift cluster with cluster administrator access.
- You must have already installed Service Registry. See Chapter 4, *Installing Service Registry on OpenShift*.
- You must have already installed AMQ Streams. See Section 5.1, “Installing AMQ Streams from the OpenShift OperatorHub”.

Procedure

1. In the OpenShift Container Platform web console, log in using an account with cluster administrator privileges.
2. If you do not already have a Kafka cluster configured, create a new Kafka cluster using AMQ Streams. For example, in the OpenShift OperatorHub:
    a. Click **Installed Operators** > **Red Hat Integration - AMQ Streams**
    b. Under **Provided APIs** > **Kafka**, click **Create Instance** to create a new Kafka cluster.
    c. Edit the custom resource definition as appropriate, and click **Create**.

WARNING

The default example creates a cluster with 3 Zookeeper nodes and 3 Kafka nodes with **ephemeral** storage. This temporary storage is suitable for development and testing only, and not for production. For more details, see Using AMQ Streams on OpenShift .

3. After the cluster is ready, click **Provided APIs** > **Kafka** > **my-cluster** > **YAML**.

4. In the **status** block, make a copy of the **bootstrapServers** value, which you will use later to deploy Service Registry. For example:

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

5. Create a Kafka topic to store the Service Registry artifacts:

   a. Under **Provided APIs** > **Kafka Topic**, click **Create topic**.

   b. Change the default topic name from **my-topic** to the required **storage-topic**.

6. Create a Kafka topic to store the Service Registry global IDs:

   a. Under **Provided APIs** > **Kafka Topic**, click **Create topic**.

   b. Change the default topic name from **my-topic** to the required **global-id-topic**.

7. Click **Installed Operators** > **Red Hat Integration - Service Registry**> **ApicurioRegistry** > **Create ApicurioRegistry**.

8. Paste in the following custom resource definition, but use your **bootstrapServers** value that you copied earlier:

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

9. Click **Create** and wait for the Service Registry route to be created on OpenShift.

10. Click **Networking** > **Route** to access the new route for the Service Registry web console. For example:

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

Additional resources

- For more details on creating Kafka clusters and topics using AMQ Streams, see Using AMQ Streams on OpenShift.

5.3. CONFIGURING TLS SECURITY WITH SERVICE REGISTRY STORAGE IN AMQ STREAMS

You can configure the AMQ Streams Operator and Service Registry Operator to use an encrypted Transport Layer Security (TLS) connection.

Prerequisites

- You must install the Service Registry Operator using the OperatorHub or command line.

- You must install the AMQ Streams Operator or have Kafka accessible from your OpenShift cluster.

NOTE

This section assumes that the AMQ Streams Operator is available, however you can use any Kafka deployment. In that case, you must manually create the Openshift secrets that the Service Registry Operator expects.

Procedure

1. In the OpenShift web console, click **Installed Operators**, select the **AMQ Streams** Operator details, and then the **Kafka** tab.

2. Click **Create Kafka** to provision a new Kafka cluster for Service Registry storage.

3. Configure the **authorization** and **tls** fields to use TLS authentication for the Kafka cluster, for example:

    ```
    apiVersion: kafka.strimzi.io/v1beta1
    kind: Kafka
    metadata:
      name: my-cluster
      namespace: registry-example-streams-tls
    spec:
      kafka:
        authorization:
          type: simple
        version: 2.5.0
        replicas: 3
        listeners:
          plain: {}
    ```

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

4. Create a Kafka topic to store the Service Registry artifacts:

    a. Under **Provided APIs** > **Kafka Topic**, click **Create topic**.

    b. Change the default topic name from **my-topic** to the required **storage-topic**.

5. Create a Kafka topic to store the Service Registry global IDs:

    a. Under **Provided APIs** > **Kafka Topic**, click **Create topic**.

    b. Change the default topic name from **my-topic** to the required **global-id-topic**.

6. Create a **Kafka User** resource to configure authentication and authorization for the Service Registry user. For example, in the **spec** block, you can specify a user name in the **metadata** section or use the default **my-user**.

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

```
        name: '*'
        patternType: literal
        type: group
  type: simple
```

NOTE

You must configure the authorization specifically for the topics and resources that the Service Registry requires. This is a simple example.

7. Click **Workloads** and then **Secrets** to find two secrets that AMQ Streams creates for Service Registry to connect to the Kafka cluster:
   - **my-cluster-cluster-ca-cert** - contains the PKCS12 truststore for the Kafka cluster
   - **my-user** - contains the user’s keystore

   NOTE

   The name of the secret can vary based on your cluster or user name.

8. If you create the secrets manually, they must contain the following key-value pairs:
   - **my-cluster-ca-cert**
     - **ca.p12** - truststore in PKCS12 format
     - **ca.password** - truststore password
   - **my-user**
     - **user.p12** - keystore in PKCS12 format
     - **user.password** - keystore password

9. Configure the following example configuration to deploy the Service Registry.

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

IMPORTANT

You must use a different **bootstrapServers** address than in the plain insecure use case. The address must support TLS connections and is found in the specified **Kafka** resource under the **type: tls** field.

5.4. CONFIGURING SCRAM SECURITY WITH SERVICE REGISTRY STORAGE IN AMQ STREAMS

You can configure the AMQ Streams Operator and Service Registry Operator to use Salted Challenge Response Authentication Mechanism (SCRAM-SHA-512) for the Kafka cluster.

Prerequisites

- You must install the Service Registry Operator using the OperatorHub or command line.
- You must install the AMQ Streams Operator or have Kafka accessible from your OpenShift cluster.

NOTE

This section assumes that AMQ Streams Operator is available, however you can use any Kafka deployment. In that case, you must manually create the Openshift secrets that the Service Registry Operator expects.

Procedure

1. In the OpenShift web console, click **Installed Operators**, select the **AMQ Streams** Operator details, and then the **Kafka** tab.
2. Click **Create Kafka** to provision a new Kafka cluster for Service Registry storage.
3. Configure the **authorization** and **tls** fields to use SCRAM-SHA-512 authentication for the Kafka cluster, for example:

```
apiVersion: kafka.strimzi.io/v1beta1
kind: Kafka
metadata:
  name: my-cluster
  namespace: registry-example-streams-tls
spec:
  kafka:
    authorization:
      type: simple
    version: 2.5.0
    replicas: 3
    listeners:
      plain: {}
      tls:
        authentication:
          type: scram-sha-512
    config:
      offsets.topic.replication.factor: 3
      transaction.state.log.replication.factor: 3
      transaction.state.log.min.isr: 2
      log.message.format.version: '2.5'
    storage:
      type: ephemeral
  zookeeper:
    replicas: 3
    storage:
      type: ephemeral
  entityOperator:
    topicOperator: {}
    userOperator: {}
```

4. Create a Kafka topic to store the Service Registry artifacts:
    a. Under **Provided APIs** > **Kafka Topic**, click **Create topic**.
    b. Change the default topic name from **my-topic** to the required **storage-topic**.
5. Create a Kafka topic to store the Service Registry global IDs:
    a. Under **Provided APIs** > **Kafka Topic**, click **Create topic**.
    b. Change the default topic name from **my-topic** to the required **global-id-topic**.
6. Create a **Kafka User** resource to configure SCRAM authentication and authorization for the Service Registry user. For example, in the **spec** block, see the **authentication** section.

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

7. Click **Workloads** and then **Secrets** to find two secrets that AMQ Streams creates for Service Registry to connect to the Kafka cluster:
    - **my-cluster-cluster-ca-cert** - contains the PKCS12 truststore for the Kafka cluster

- **my-user** - contains the user's keystore

> NOTE
>
> The name of the secret can vary based on your cluster or user name.

8. If you create the secrets manually, they must contain the following key-value pairs:
   - my-cluster-ca-cert
     - **ca.p12** - the truststore in PKCS12 format
     - **ca.password** - truststore password
   - my-user
     - **password** - user password
9. Configure the following example settings to deploy the Service Registry:

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

> IMPORTANT
>
> You must use a different **bootstrapServers** address than in the plain insecure use case. The address must support TLS connections, and is found in the specified **Kafka** resource under the **type: tls** field.

CHAPTER 6. DEPLOYING SERVICE REGISTRY STORAGE IN A POSTGRESQL DATABASE

This chapter explains how to install, configure, and manage Service Registry storage in a PostgreSQL database.

- Section 6.1, "Installing a PostgreSQL database from the OpenShift OperatorHub"
- Section 6.2, "Configuring Service Registry with PostgreSQL database storage on OpenShift"
- Section 6.3, "Backing up Service Registry PostgreSQL storage"
- Section 6.4, "Restoring Service Registry PostgreSQL storage"

IMPORTANT

Service Registry storage in PostgreSQL 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.

Prerequisites

- Chapter 4, *Installing Service Registry on OpenShift*

6.1. INSTALLING A POSTGRESQL DATABASE FROM THE OPENSHIFT OPERATORHUB

If you do not already have a PostgreSQL database Operator installed, you can install a PostgreSQL Operator on your OpenShift cluster from the OperatorHub. The OperatorHub is available from the OpenShift Container Platform web console and provides an interface for cluster administrators to discover and install Operators. For more details, see the OpenShift documentation.

Prerequisites

- You must have cluster administrator access to an OpenShift cluster.

Procedure

1. In the OpenShift Container Platform web console, log in using an account with cluster administrator privileges.
2. Change to the OpenShift project in which Service Registry is installed. For example, from the **Project** drop-down, select **my-project**.
3. In the left navigation menu, click **Operators** > **OperatorHub**.

4. In the **Filter by keyword** text box, enter **PostgreSQL** to find an Operator suitable for your environment, for example, **Crunchy PostgreSQL for OpenShift** or **PostgreSQL Operator by Dev4Ddevs.com**.

5. Read the information about the Operator, and click **Install** to display the Operator subscription page.

6. Select your subscription settings, for example:

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

7. Click **Install**, and wait a few moments until the Operator is ready for use.

IMPORTANT

You must read the documentation from your chosen **PostgreSQL** Operator for details on how to create and manage your database.

Additional resources

- Adding Operators to an OpenShift cluster
- Crunchy PostgreSQL Operator QuickStart

6.2. CONFIGURING SERVICE REGISTRY WITH POSTGRESQL DATABASE STORAGE ON OPENSHIFT

This section explains how to configure Java Persistence API-based storage for Service Registry on OpenShift using a PostgreSQL database Operator. You can install Service Registry in an existing database or create a new database, depending on your environment. This section shows a simple example using the PostgreSQL Operator by Dev4Ddevs.com.

Prerequisites

- You must have an OpenShift cluster with cluster administrator access.
- You must have already installed Service Registry. See Chapter 4, *Installing Service Registry on OpenShift*.
- You must have already installed a PostgreSQL Operator on OpenShift. For example, see Section 6.1, “Installing a PostgreSQL database from the OpenShift OperatorHub” .

Procedure

1. In the OpenShift Container Platform web console, log in using an account with cluster administrator privileges.

2. Change to the OpenShift project in which Service Registry and your PostgreSQL Operator are installed. For example, from the **Project** drop-down, select **my-project**.

3. Create a PostgreSQL database for your Service Registry storage. For example, click **Installed Operators** > **PostgreSQL Operator by Dev4Ddevs.com** > **Create database** > **YAML**.

4. Edit the database settings as follows:

   - **name**: Change the value to **registry**
   - **image**: Change the value to **centos/postgresql-10-centos7**

5. Edit any other database settings as needed depending on your environment, for example:

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

6. Click **Create Database**, and wait until the database is created.

7. Click **Installed Operators** > **Red Hat Integration - Service Registry**> **ApicurioRegistry** > **Create ApicurioRegistry**.

8. Paste in the following custom resource definition, and edit the values for the database **url** and credentials to match your environment:

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

9. Click **Create** and wait for the Service Registry route to be created on OpenShift.

10. Click **Networking** > **Route** to access the new route for the Service Registry web console. For example:

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

Additional resources

- Crunchy PostgreSQL Operator QuickStart
- Apicurio Registry Operator QuickStart

6.3. BACKING UP SERVICE REGISTRY POSTGRESQL STORAGE

When using Java Persistence API storage in a PostgreSQL database, you must ensure that the data stored by Service Registry is backed up regularly.

SQL Dump is a simple procedure that works with any PostgreSQL installation. This uses the pg_dump utility to generate a file with SQL commands that you can use to recreate the database in the same state that it was in at the time of the dump.

pg_dump is a regular PostgreSQL client application, which you can execute from any remote host that has access to the database. Like any other client, the operations that can perform are limited to the user permissions.

Procedure

- Use the **pg_dump** command to redirect the output to a file:

  ```
  $ pg_dump dbname > dumpfile
  ```

  You can specify the database server that **pg_dump** connects to using the **-h host** and **-p port** options.

- You can reduce large dump files using a compression tool, such as gzip, for example:

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

Additional resources

For details on client authentication, see the PostgreSQL documentation.

6.4. RESTORING SERVICE REGISTRY POSTGRESQL STORAGE

You can restore SQL Dump files created by **pg_dump** using the **psql** utility.

Prerequisites

- You must have already backed up your PostgreSQL datbase using **pg_dump**. See Section 6.3, “Backing up Service Registry PostgreSQL storage”.
- All users who own objects or have permissions on objects in the dumped database must already exist.

Procedure

1. Enter the following command to create the database:

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

2. Enter the following command to restore the SQL dump

```
$ psql dbname < dumpfile
```

3. Run ANALYZE on each database so the query optimizer has useful statistics.

CHAPTER 7. DEPLOYING EMBEDDED SERVICE REGISTRY STORAGE IN INFINISPAN

This chapter explains how to configure Service Registry storage in an embedded Infinispan cache.

- Section 7.1, "Configuring Service Registry with embedded Infinispan storage on OpenShift"

IMPORTANT

Service Registry storage in Infinispan 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.

Prerequisites

- Chapter 4, *Installing Service Registry on OpenShift*

7.1. CONFIGURING SERVICE REGISTRY WITH EMBEDDED INFINISPAN STORAGE ON OPENSHIFT

This section explains how to configure Infinispan cache-based storage for Service Registry on OpenShift. This storage option is based on Infinispan community Java libraries embedded in the Quarkus-based Service Registry server. You do not need to install a separate Infinispan server using this storage option. This option is suitable for development or demonstration only, and is not suitable for production environments.

Prerequisites

- You must have an OpenShift cluster with cluster administrator access.
- You must have already installed Service Registry. See Chapter 4, *Installing Service Registry on OpenShift*.

Procedure

1. In the OpenShift Container Platform web console, log in using an account with cluster administrator privileges.
2. Click **Installed Operators** > **Red Hat Integration - Service Registry**> **ApicurioRegistry** > **Create ApicurioRegistry**.
3. Paste in the following custom resource definition:

```
apiVersion: apicur.io/v1alpha1
kind: ApicurioRegistry
metadata:
  name: example-apicurioregistry
```

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

4. Click **Create** and wait for the Service Registry route to be created on OpenShift.

5. Click **Networking** > **Route** to access the new route for the Service Registry web console. For example:

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

Additional resources

- For more details on configuring Infinispan clusters, see the example custom resources available from the Apicurio Registry demonstration.

- For more details on Infinispan, see https://infinispan.org/

CHAPTER 8. CONFIGURING AND MANAGING SERVICE REGISTRY DEPLOYMENT

This chapter explains how to configure and manage optional settings for your Service Registry deployment on OpenShift:

8.1. CONFIGURING SERVICE REGISTRY HEALTH CHECKS ON OPENSHIFT

You can configure optional environment variables for liveness and readiness probes to monitor the health of the Service Registry server on OpenShift:

- *Liveness probes* test if the application can make progress. If the application cannot make progress, OpenShift automatically restarts the failing Pod.
- *Readiness probes* test if the application is ready to process requests. If the application is not ready, it can become overwhelmed by requests, and OpenShift stops sending requests for the time that the probe fails. If other Pods are OK, they continue to receive requests.

IMPORTANT

The default values of the liveness and readiness environment variables are designed for most cases and should only be changed if required by your environment. Any changes to the defaults depend on your hardware, network, and amount of data stored. These values should be kept as low as possible to avoid unnecessary overhead.

Prerequisites

- You must have an OpenShift cluster with cluster administrator access.
- You must have already installed Service Registry on OpenShift with your preferred storage option. See Chapter 4, *Installing Service Registry on OpenShift* .
- You must have already installed and configured your chosen Service Registry storage in AMQ Streams, embedded Infinispan, or PostgreSQL.

Procedure

1. In the OpenShift Container Platform web console, log in using an account with cluster administrator privileges.
2. Click **Installed Operators** > **Red Hat Integration - Service Registry**.

3. On the **ApicurioRegistry** tab, click the Operator custom resource for your deployment, for example, **example-apicurioregistry**.

4. In the main overview page, find the **Deployment Name** section and the corresponding **DeploymentConfig** name for your Service Registry deployment, for example, **example-apicurioregistry**.

5. In the left navigation menu, click **Workloads** > **Deployment Configs**, and select your **DeploymentConfig** name.

6. Click the **Environment** tab, and enter your environment variables in the **Single values env** section, for example:
   - NAME: **LIVENESS_STATUS_RESET**
   - VALUE: **350**

7. Click **Save** at the bottom.
   Alternatively, you can perform these steps using the OpenShift **oc** command. For more details, see the OpenShift CLI documentation.

Additional resources

- Section 8.2, "Environment variables for Service Registry health checks"
- OpenShift documentation on monitoring application health

8.2. ENVIRONMENT VARIABLES FOR SERVICE REGISTRY HEALTH CHECKS

This section describes the available environment variables for Service Registry health checks on OpenShift. These include liveness and readiness probes to monitor the health of the Service Registry server on OpenShift. For an example procedure, see Section 8.1, "Configuring Service Registry health checks on OpenShift".

IMPORTANT

The following environment variables are provided for reference only. The default values are designed for most cases and should only be changed if required by your environment. Any changes to the defaults depend on your hardware, network, and amount of data stored. These values should be kept as low as possible to avoid unnecessary overhead.

Liveness environment variables

Table 8.1. Environment variables for Service Registry liveness probes

Name	Description	Type	Default
LIVENESS_ERROR_THRESHOLD	Number of liveness issues or errors that can occur before the liveness probe fails.	Integer	**1**

Name	Description	Type	Default
LIVENESS_COUNTER_RESET	Period in which the threshold number of errors must occur. For example, if this value is 60 and the threshold is 1, the check fails after two errors occur in 1 minute	Seconds	**60**
LIVENESS_STATUS_RESET	Number of seconds that must elapse without any more errors for the liveness probe to reset to OK status.	Seconds	**300**
LIVENESS_ERRORS_IGNORED	Comma-separated list of ignored liveness exceptions.	String	**io.grpc.StatusRuntimeException,org.apache.kafka.streams.errors.InvalidStateStoreException**

NOTE

Because OpenShift automatically restarts a Pod that fails a liveness check, the liveness settings, unlike readiness settings, do not directly affect behavior of Service Registry on OpenShift.

Readiness environment variables

Table 8.2. Environment variables for Service Registry readiness probes

Name	Description	Type	Default
READINESS_ERROR_THRESHOLD	Number of readiness issues or errors that can occur before the readiness probe fails.	Integer	**1**
READINESS_COUNTER_RESET	Period in which the threshold number of errors must occur. For example, if this value is 60 and the threshold is 1, the check fails after two errors occur in 1 minute.	Seconds	**60**
READINESS_STATUS_RESET	Number of seconds that must elapse without any more errors for the liveness probe to reset to OK status. In this case, this means how long the Pod stays not ready, until it returns to normal operation.	Seconds	**300**

Name	Description	Type	Default
READINESS_TIMEOUT	Readiness tracks the timeout of two operations: ● How long it takes for storage requests to complete ● How long it takes for HTTP REST API requests to return a response If these operations take more time than the configured timeout, this is counted as a readiness issue or error. This value controls the timeouts for both operations.	Seconds	**5**

Additional resources

- Section 8.1, "Configuring Service Registry health checks on OpenShift"
- OpenShift documentation on monitoring application health

8.3. CONFIGURING AN HTTPS CONNECTION TO SERVICE REGISTRY FROM INSIDE THE OPENSHIFT CLUSTER

The following procedure shows how to configure Service Registry deployment to expose a port for HTTPS connections from inside the OpenShift cluster.

WARNING

This kind of connection is not directly available outside of the cluster. Routing is based on hostname, which is encoded in the case of an HTTPS connection. Therefore, edge termination or other configuration is still needed. See Section 8.4, "Configuring an HTTPS connection to Service Registry from outside the OpenShift cluster".

Prerequisites

- You must have already installed the Service Registry Operator.

Procedure

1. Generate a **keystore** with a self-signed certificate. You can skip this step if you are using your own certificates.

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

2. Create a new secret to hold the keystore and keystore password.

   a. In the left navigation menu of the OpenShift web console, click **Workloads** > **Secrets** > **Create Key/Value Secret**

   b. Use the following values:

      - Name: **registry-keystore**
      - Key 1: **keystore.jks**
      - Value 1: *registry-keystore.jks* (uploaded file)
      - Key 2: **password**
      - Value 2: *password*

      NOTE

      If you encounter a **java.io.IOException: Invalid keystore format**, the upload of the binary file did not work properly. As an alternative, encode the file as a base64 string using **cat registry-keystore.jks | base64 -w0 > data.txt** and edit the **Secret** resource as yaml to manually add the encoded file.

3. Edit the **DeploymentConfig** resource of the Service Registry instance. You can find the correct name in a status field of the Service Registry Operator.

   a. Add the keystore secret as a volume:

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

   b. Add a volume mount:

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

   c. Add **JAVA_OPTIONS** and **KEYSTORE_PASSWORD** environment variables:

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

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

NOTE

Order is important when using string interpolation.

d. Enable the HTTPS port:

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

4. Edit the **Service** resource of the Service Registry instance. You can find the correct name in a status field of the Service Registry Operator.

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

5. Verify that the connection is working:

a. Connect into a pod on the cluster using SSH (you can use the Service Registry pod):

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

b. Find the cluster IP of the Service Registry pod from the **Service** resource (see the **Location** column in the web console).

c. Afterwards, execute a test request (we are using self-signed certificate, so an insecure flag is required):

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

8.4. CONFIGURING AN HTTPS CONNECTION TO SERVICE REGISTRY FROM OUTSIDE THE OPENSHIFT CLUSTER

The following procedure shows how to configure Service Registry deployment to expose an HTTPS edge-terminated route for connections from outside the OpenShift cluster.

Prerequisites

- You must have already installed the Service Registry Operator.
- Read the OpenShift documentation for creating secured routes .

Procedure

- Add a second **Route** in addition to the HTTP route created by the Service Registry Operator. See the following example:

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

NOTE

Make sure the **insecureEdgeTerminationPolicy: Redirect** configuration property is set.

- If you do not specify a certificate, OpenShift will use a default. You can alternatively generate a custom self-signed certificate using the following commands:

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

And then create a route using the OpenShift CLI:

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

CHAPTER 9. MANAGING SERVICE REGISTRY CONTENT USING THE WEB CONSOLE

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

- Section 9.1, "Configuring the Service Registry web console"
- Section 9.2, "Adding artifacts using the Service Registry web console"
- Section 9.3, "Viewing artifacts using the Service Registry web console"
- Section 9.4, "Configuring content rules using the Service Registry web console"

9.1. CONFIGURING THE SERVICE REGISTRY WEB CONSOLE

You can configure the Service Registry web console specifically for your deployment environment or to customize its behavior. This section provides details on how to configure optional environment variables for the Service Registry web console.

Prerequisites

- You must have already installed Service Registry.

Configuring the web console deployment environment

When a user navigates their browser to the Service Registry web console, some initial configuration settings are loaded. Two important configuration properties are:

- URL for backend Service Registry REST API
- URL for frontend Service Registry web console

Typically, Service Registry automatically detects and generates these settings, but there are some deployment environments where this automatic detection can fail. If this happens, you can configure environment variables to explicitly set these URLs for your environment.

Procedure

Configure the following environment variables to override the default URLs:

- **REGISTRY_UI_CONFIG_APIURL**: Set the URL for the backend Service Registry REST API. For example,**https://registry.my-domain.com/api**
- **REGISTRY_UI_CONFIG_UIURL**: Set the URL for the frontend Service Registry web console. For example, **https://registry.my-domain.com/ui**

Configuring the console in read-only mode

You can configure the Service Registry web console in read-only mode as an optional feature. This mode disables all features in the Service Registry web console that allow users to make changes to registered artifacts. For example, this includes the following:

- Creating an artifact
- Uploading a new version of an artifact
- Updating an artifact's metadata

- Deleting an artifact

Procedure

Configure the following environment variable to set the Service Registry web console in read-only mode:

- **REGISTRY_UI_FEATURES_READONLY**: Set to **true** to enable read-only mode. Defaults to **false**.

9.2. 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 14, *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:

   - **ID**: Use the default empty setting to automatically generate an ID, or enter a specific artifact 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**.

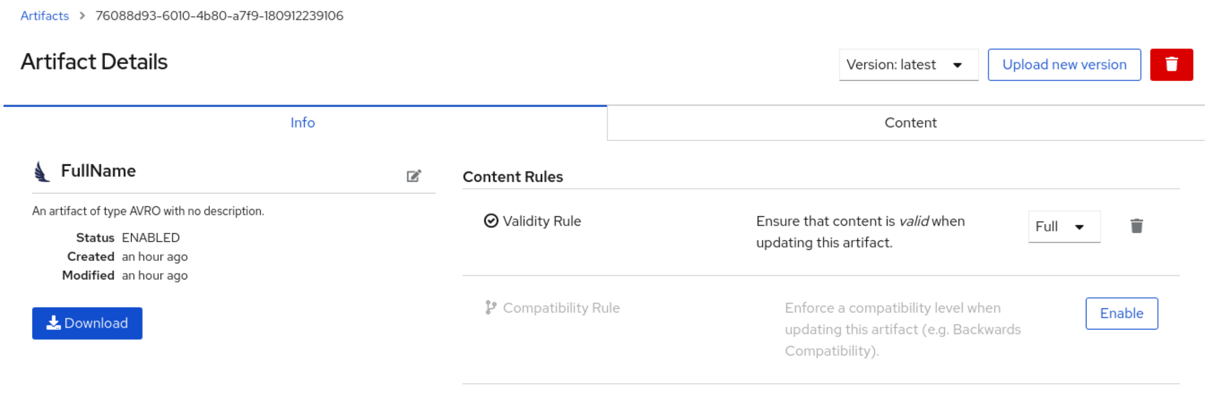
     NOTE

     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 9.1. 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 — Ensure that content is *valid* when updating this artifact. — Full

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

- **Info**: Displays the artifact name, description, lifecycle status, when created, and last modified. You can 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 14, *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 9.3, “Viewing artifacts using the Service Registry web console”

- Section 9.4, “Configuring content rules using the Service Registry web console”

9.3. 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, versions, and artifact rules. For more details on the artifact types stored in the registry, see Chapter 14, *Service Registry artifact reference*.

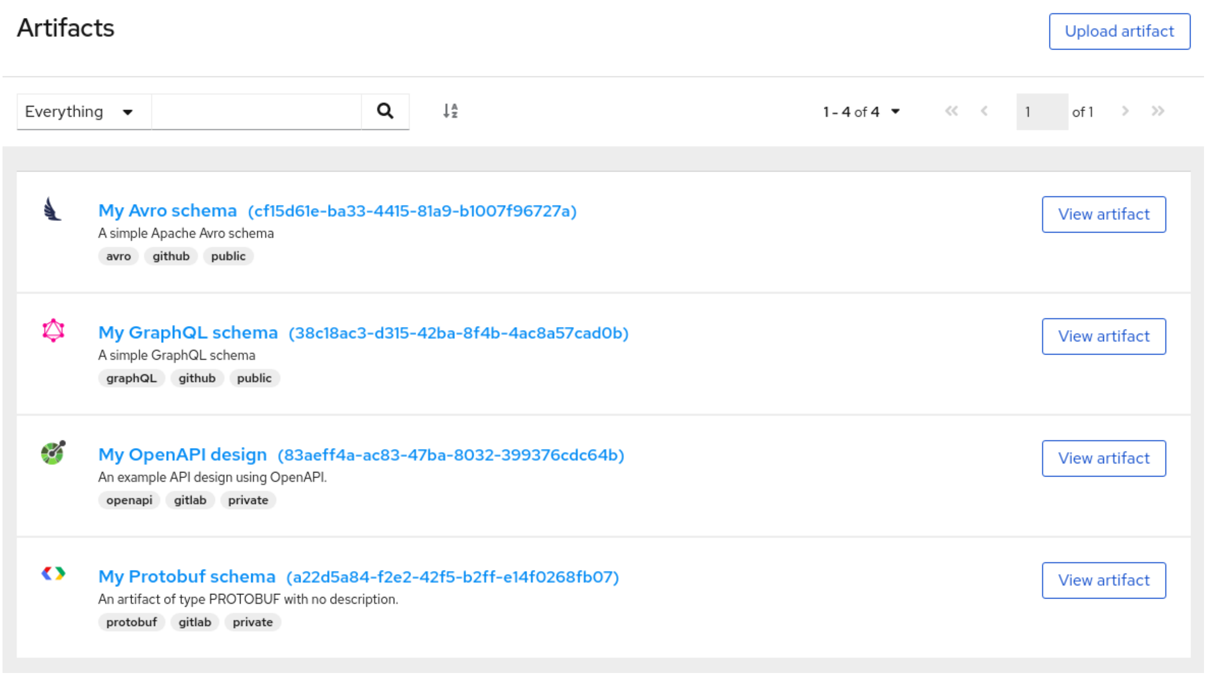
Prerequisites

- 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**, **Description**, **Label**, or **Everything**.

   Figure 9.2. Browse artifacts in Service Registry web console

   My Avro schema (cf15d61e-ba33-4415-81a9-b1007f96727a)
   A simple Apache Avro schema
   avro github public

   My GraphQL schema (38c18ac3-d315-42ba-8f4b-4ac8a57cad0b)
   A simple GraphQL schema
   graphQL github public

   My OpenAPI design (83aeff4a-ac83-47ba-8032-399376cdc64b)
   An example API design using OpenAPI.
   openapi gitlab private

   My Protobuf schema (a22d5a84-f2e2-42f5-b2ff-e14f0268fb07)
   An artifact of type PROTOBUF with no description.
   protobuf gitlab private

3. Click **View artifact** to view the **Artifact Details**:
   - **Info**: Displays the artifact name, description, lifecycle status, when created, and last modified. You can 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 9.2, “Adding artifacts using the Service Registry web console”
- Section 9.4, “Configuring content rules using the Service Registry web console”

9.4. 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 14, *Service Registry artifact reference*.

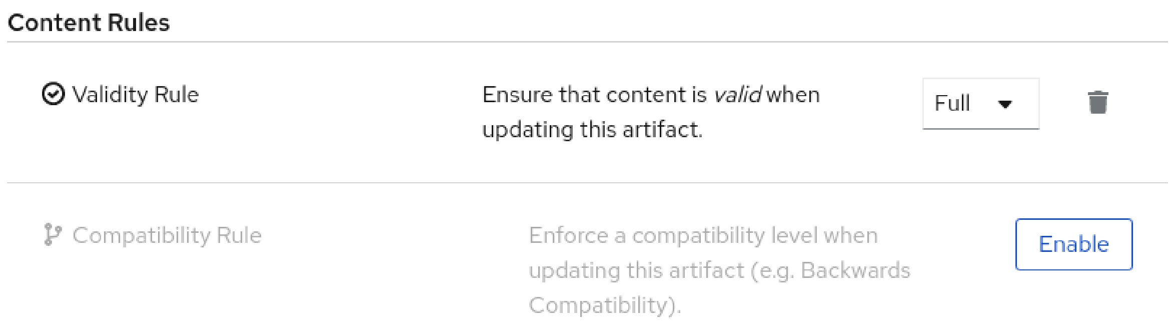
Prerequisites

- 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**, **Description**, **Label**, or **Everything**.
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 14, *Service Registry artifact reference* .

   Figure 9.3. Configure content rules in Service Registry web console


Content Rules

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

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

5. For global rules, click the **Settings** cog icon 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 14, *Service Registry artifact reference*.
6. To disable an artifact rule or global rule, click the trash icon next to the rule.

Additional resources

- Section 9.2, “Adding artifacts using the Service Registry web console”

CHAPTER 10. MANAGING SERVICE REGISTRY CONTENT USING THE REST API

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

- Section 10.1, "Registry REST API overview"
- Section 10.2, "Managing artifacts using Registry REST API commands"

Additional resources

- Apicurio Registry REST API documentation

10.1. REGISTRY REST API OVERVIEW

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

Artifacts

Manage the schema and API design artifacts stored in the registry. This also includes browse or search for artifacts, for example, by name, ID, description, or label. You can also manage the lifecycle state of an artifact: enabled, disabled, or deprecated.

Artifact versions

Manage the versions that are created when artifact content is updated. This also includes browse or search for versions, for example, by name, ID, description, or label. You can also manage the lifecycle state of a version: enabled, disabled, or deprecated.

Artifact metadata

Manage details about artifacts such as when an artifact was created or modified, its current state, and so on. Users can edit some metadata, and some is read-only. For example, editable metadata includes artifact name, description, or label, but when the artifact was created and modified are read-only.

Global rules

Configure rules to govern the content evolution of all 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.

Artifact rules

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

Compatibility with other schema registries

The Registry REST API is compatible with the Confluent schema registry REST API, which includes support for Apache Avro, Google Protocol buffers, and JSON Schema artifact types. Applications using Confluent client libraries can use Service Registry as a drop-in replacement instead. 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 .

- The Registry REST API documentation is also available from the main endpoint of your Service Registry deployment, for example, on **http://MY-REGISTRY-URL/api**.

10.2. MANAGING ARTIFACTS USING REGISTRY REST API COMMANDS

Client applications can use Registry REST API commands to manage 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 section shows a simple curl-based example of using the Registry REST API to add and retrieve an Apache Avro schema artifact in the registry.

NOTE

When adding artifacts in Service Registry using the REST API, if you do not specify a unique artifact ID, Service Registry generates one automatically as a UUID.

Prerequisites

- See Chapter 1, *Introduction to Service Registry*
- Service Registry must be installed and running in your environment.

Procedure

1. Add an artifact in the registry using the **/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" --data '{"type":"record","name":"price","namespace":"com.example","fields":[{"name":"symbol","type":"string"},{"name":"price","type":"string"}]}' http://MY-REGISTRY-HOST/api/artifacts
```

   This example shows adding an Avro schema artifact with an artifact ID of **share-price**.

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

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

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

3. Retrieve the artifact from the registry using its artifact ID. For example, in this case the specified ID is **share-price**:

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

Additional resources

- For more REST API sample requests, see the Apicurio Registry REST API documentation .

CHAPTER 11. MANAGING SERVICE REGISTRY CONTENT USING THE MAVEN PLUG-IN

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

- Section 11.1, "Managing artifacts using the Service Registry 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

11.1. MANAGING ARTIFACTS USING THE SERVICE REGISTRY MAVEN PLUG-IN

You can use the Service Registry Maven plug-in to upload or download registry artifacts as part of your development build. For example, this plug-in is useful for testing and validating that your schema updates are compatible with client applications.

Registering an artifact using the Maven plug-in

Probably the most common use case for the Maven plug-in is registering 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 an Apache Avro schema:

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

1. Specify **register** as the execution goal to upload the schema artifact to the registry.
2. You must specify the Service Registry URL with the **/api** endpoint.
3. You can upload multiple artifacts using the artifact ID and location.

Downloading an artifact using the Maven plug-in

You can also 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 a single schema by its artifact ID.

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

1. Specify **download** as the execution goal.
2. You must specify the Service Registry URL with the **/api** endpoint.
3. You can download multiple artifacts to a specified directory using the artifact ID.
4. The plug-in automatically tries to select an appropriate file extension, but you can override it using **<artifactExtension>**.

Testing an artifact using the Maven plug-in

You might want to verify that an artifact can be registered without actually making any changes. This is most 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>${registry.version}</version>
  <executions>
    <execution>
      <phase>generate-sources</phase>
      <goals>
        <goal>test-update</goal> (1)
      </goals>
      <configuration>
        <registryUrl>http://my-cluster-service-registry-myproject.example.com/api</registryUrl>
(2)
        <artifactType>AVRO</artifactType>
        <artifacts>
          <schema1>${project.basedir}/schemas/schema1.avsc</schema1> (3)
        </artifacts>
      </configuration>
    </execution>
  </executions>
</plugin>
```

1. Specify **test-update** as the execution goal to test the schema artifact.
2. You must specify the Service Registry URL with the **/api** endpoint.
3. You can test multiple artifacts using the artifact ID and location.

Additional resources

- For more details on the Service Registry Maven plug-in, see the Registry demonstration example

CHAPTER 12. MANAGING SERVICE REGISTRY CONTENT USING THE JAVA CLIENT

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

- Section 12.1, “Service Registry Java client”
- Section 12.2, “Writing Service Registry client applications”
- Section 12.3, “Service Registry Java client configuration”

12.1. 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 access the Service Registry Java client by adding the correct dependency to your project, see Section 12.2, “Writing Service Registry client applications” .

The Service Registry client is auto-closeable and is implemented using Retrofit and OkHttp as base libraries. This gives you the ability to customize its use, for example, by adding custom headers or enabling Transport Layer Security (TLS) authentication. For more details, see Section 12.3, “Service Registry Java client configuration”.

12.2. WRITING SERVICE REGISTRY CLIENT APPLICATIONS

This section explains how to manage artifacts stored in Service Registry using a Java client application. The Service Registry Java client extends the **Autocloseable** interface.

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-rest-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://registry.my-domain.com/api"; (1)
        RegistryRestClient client = RegistryRestClientFactory.create(registryUrl); (2)
    }
}
```

1. You must specify the Service Registry URL with the **/api** endpoint.
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 example of how to use and customize the Service Registry client, see the Registry client demonstration example.
- For details on how to use the Service Registry Kafka client serializer/deserializer for Apache Avro in AMQ Streams producer and consumer applications, see Using AMQ Streams on Openshift.

12.3. SERVICE REGISTRY JAVA CLIENT CONFIGURATION

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

Table 12.1. Service Registry Java client configuration options

Option	Description	Arguments
Plain client	Basic REST client used to interact with a running registry.	**baseUrl**
Custom HTTP client	Registry client using an OkHttpClient provided by the user.	**baseUrl**, **okhttpClient**
Custom configuration	Registry client that accepts a map containing custom configuration. This is useful, for example, to add custom headers to the calls.	**baseUrl**, **Map<String Object> configs**

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

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 13. VALIDATING SCHEMAS USING KAFKA CLIENT SERIALIZERS/DESERIALIZERS

Service Registry provides Kafka client serializers/deserializers for producer and consumer applications. Kafka producer applications use serializers to encode messages that conform to a specific event schema. Kafka consumer applications use deserializers to validate that the 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 provides instructions on how to use the Kafka client serializers and deserializers for Apache Avro, JSON Schema, and Google Protobuf in your Kafka producer and consumer client applications:

Prerequisites

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

13.1. KAFKA CLIENT APPLICATIONS AND SERVICE REGISTRY

Using Service Registry decouples schema management from client application configuration. You can enable an application to use a schema from the registry by specifying its URL in the client code.

For example, you can store the schemas to serialize and deserialize messages in the registry, which are then referenced from the applications that use them 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 changes to a schema 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 provides schema registry support for a number of schema technologies such as:

- Avro
- Protobuf
- JSON Schema

These schema technologies can be used by client applications through Kafka client serializer/deserializer (SerDe) services provided by Service Registry. The maturity and usage of the SerDe classes provided by Service Registry may vary. See the type-specific sections below for more details about each.

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 (optional)
- Configure the producer client code:
  - URL of Service Registry
  - Service Registry serializer to use with the messages
  - Strategy to map the Kafka message to an artifact ID 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 REST API based on compatibility rules defined in Service Registry. Versions are used for compatibility checking as a schema evolves. An artifact ID and schema 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
- Configure the consumer client code:
  - URL of Service Registry
  - Service Registry deserializer to use with the messages
  - Input data stream for deserialization

The schema is then retrieved by the deserializer using a global ID written into the message being consumed. The schema global ID can be located in the message headers or in the message payload itself, 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 (as a signal to consumers) followed by the global ID and then the message data as normal.

For example:

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

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

13.2. STRATEGIES TO LOOK UP A SCHEMA

The Kafka client serializer uses *lookup strategies* to determine the artifact ID and the global ID under which the message schema is registered in Service Registry.

For a given topic and message, you can use implementations of the following Java interfaces:

- **ArtifactIdStrategy** to return an artifact ID
- **GlobalIdStrategy** to return a global ID

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

Example

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

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

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.

Artifact ID strategy

The artifact ID strategy provides a way to map the Kafka topic and message information to an artifact ID 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.utils.serde.strategy.ArtifactIdStrategy**.

Strategies to return an artifact ID

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

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.

Global ID strategy

The global ID strategy locates and identifies the specific version of the schema registered under the artifact ID provided by the artifact ID 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 Service Registry.

The global ID strategy 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.utils.serde.strategy.GlobalIdStrategy**.

Strategies to return a global ID

Service Registry provides the following strategies to return a global ID based on an implementation of **GlobalIdStrategy**:

FindLatestIdStrategy

Strategy that returns the global ID of the latest schema version, based on an artifact ID.

FindBySchemaIdStrategy

Strategy that matches schema content, based on an artifact ID, to return a global ID.

CachedSchemaIdStrategy

Strategy that caches the schema, and uses the global ID of the cached schema.

GetOrCreateIdStrategy

Strategy that tries to get the latest schema, based on an artifact ID, and if it does not exist, creates a new schema.

AutoRegisterIdStrategy

Strategy that updates the schema, and uses the global ID of the updated schema.

Global ID strategy configuration

You can configure the following application property for the global ID strategy:

- **apicurio.registry.check-period-ms**: Configures the remote schema lookup period in milliseconds

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.

13.3. SERVICE REGISTRY SERIALIZER/DESERIALIZER CONSTANTS

You can configure specific client serializer/deserializer (SerDe) services and schema lookup strategies directly into a client using the constants outlined in this section.

Alternatively, you can specify the constants in a properties file, or a properties instance.

Constants for serializer/deserializer services

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

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

1. (Required) The URL of Service Registry.
2. Allows the client to make the request and look up the information from a cache of previous results, to improve processing time. If the cache is empty, the lookup is performed from Service Registry.
3. Extends ID handling to support other ID formats and make them compatible with Service Registry SerDe services. For example, changing the ID format from **Long** to **Integer** supports the Confluent ID format.
4. A flag to simplify the handling of Confluent IDs. If set to **true**, an **Integer** is used for the global ID lookup.

Constants for lookup strategies

```
public abstract class AbstractKafkaStrategyAwareSerDe<T, S extends
AbstractKafkaStrategyAwareSerDe<T, S>> extends AbstractKafkaSerDe<S> {
  public static final String REGISTRY_ARTIFACT_ID_STRATEGY_CONFIG_PARAM =
"apicurio.registry.artifact-id"; (1)
  public static final String REGISTRY_GLOBAL_ID_STRATEGY_CONFIG_PARAM =
"apicurio.registry.global-id"; (2)
```

1. ArtifactId strategy.
2. Global ID strategy.

Constants for converters

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

1 (Required) Serializer to use with the converter.

2 (Required) Deserializer to use with the converter.

Constants for Avro data providers

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

1 Avro Datum provider to write data to a schema, with or without reflection.

2 Flag to set to use an Avro-specific datum reader.

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

1 Default datum reader.

2 Datum reader using reflection.

13.4. USING DIFFERENT CLIENT SERIALIZER/DESERIALIZER TYPES

When using a schema technology in your Kafka applications, you must choose which specific schema type to use. Common options include:

- Apache Avro
- JSON Schema
- Google Protobuf

Which schema technology you choose is dependent on use case and preference. Of course you can use Kafka to implement custom serializer and deserializer classes, so you are always free to write your own classes, including leveraging Service Registry functionality using the Service Registry REST Java client.

For your convenience, Service Registry provides out-of-the box SerDe classes for Avro, JSON Schema, and Protobuf schema technologies. The following sections explains how to configure Kafka applications to use each type.

Kafka application configuration for serializers/deserializers

Using one of the serializer or deserializer classes provided by Service Registry in your Kafka application involves setting the correct configuration properties. The following simple 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

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

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

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

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

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

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

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

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

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

Example deserializer configuration in a Kafka consumer

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

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

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

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

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

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

13.4.1. Configure Avro SerDe with Service Registry

Service Registry provides Kafka client serializer and deserializer classes for Apache Avro to make using Avro as easy as possible:

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

Configure the Avro serializer

You can configure the Avro serializer class in the following ways:

- Service Registry location as a URL
- Artifact ID strategy
- Global ID strategy
- Global ID location
- Global ID handler
- Avro datum provider
- Avro encoding

Global ID location

The serializer passes the unique global ID of the schema as part of the Kafka message so that consumers can use the right schema for deserialization. The location of that global ID can be in the payload of the message or in the message headers. The default approach is to pass the global ID in the message payload. If you want the ID sent in the message headers instead, you can set the following configuration property:

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

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

Global ID handler

You can customize precisely how the global ID is encoded when passing it in the Kafka message body.

Set the configuration property **apicurio.registry.id-handler** to be a class that implements the **io.apicurio.registry.utils.serde.strategy.IdHandler** interface. Service Registry provides two implementations of that interface:

- **io.apicurio.registry.utils.serde.strategy.DefaultIdHandler** - stores the ID as an 8 byte long
- **io.apicurio.registry.utils.serde.strategy.Legacy4ByteIdHandler** - stores the ID as an 4 byte int

Service Registry represents the global ID of an artifact as a long, but for legacy reasons (or for compatibility with other registries or serde classes) you may 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 on which type is then actually used, where **DefaultAvroDatumProvider** is used by default.

There are two configuration options you can set:

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

Avro encoding

When using Apache Avro to serializer data, it is common to use the Avro binary encoding format. This is so that the data is encoded in as efficient a format as possible. However, Avro also supports encoding the data as JSON. Encoding as JSON is useful because it is much easier to inspect the payload of each message, often for logging, debugging, or other similar use cases. The Service Registry Avro serializer can be configured to change the encoding to JSON from the default (binary).

Set the Avro encoding to use by configuring the **apicurio.avro.encoding** property. The value must be either **JSON** or **BINARY**.

Configure the Avro deserializer

You must configure the Avro deserializer class to match the configuration settings of the serializer. As a result, you can configure the Avro deserializer class in the following ways:

- Service Registry location as a URL
- Global ID handler
- 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 needed when configuring the deserializer:

- Artifact ID strategy
- Global ID strategy
- Global ID location

The reason these options are not necessary is that the deserializer class can figure this information out from the message itself. In the case of the two strategies, they are not needed because the serializer is responsible for sending the global ID of the schema as part of the message.

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

13.4.2. Configure JSON Schema SerDe with Service Registry

Service Registry provides Kafka client serializer and deserializer classes for JSON Schema to make using JSON Schema as easy as possible:

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

Unlike Apache Avro, JSON Schema is not actually a serialization technology - it 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 in the following ways:

- Service Registry location as a URL
- Artifact ID strategy
- Global ID strategy
- Validation enabled/disabled

The only non-standard configuration property is whether JSON Schema validation is enabled or disabled. The validation feature is disabled by default but can be enabled by setting **apicurio.registry.serdes.json-schema.validation-enabled** to **"true"**. For example:

```
props.putIfAbsent(JsonSchemaSerDeConstants.REGISTRY_JSON_SCHEMA_VALIDATION_ENABLE
D, "true")`
```

Configure the JSON Schema deserializer

You can configure the JSON Schema deserializer class in the following ways:

- Service Registry location as a URL
- Validation enabled/disabled

The deserializer is simple to configure. You must provide the location of Service Registry so that the schema can be loaded. The only other configuration is whether or not to perform validation. These configuration properties are the same as for the serializer.

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.

13.4.3. Configure Protobuf SerDe with Service Registry

Service Registry provides Kafka client serializer and deserializer classes for Google Protobuf to make using Protobuf as easy as possible:

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

Configure the Protobuf serializer

You can configure the Protobuf serializer class in the following ways:

- Service Registry location as a URL
- Artifact ID strategy
- Global ID strategy
- Global ID location
- Global ID handler

Configure the Protobuf deserializer

You must configure the Protobuf deserializer class to match the configuration settings of the serializer. As a result, you can configure the Protobuf deserializer class in the following ways:

- Service Registry location as a URL
- Global ID handler

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

NOTE

The following options are not needed when configuring the deserializer:

- Artifact ID strategy
- Global ID strategy
- Global ID location

The reason these options are not necessary is that the deserializer class can figure this information out from the message itself. In the case of the two strategies, they are not needed because the serializer is responsible for sending the global ID of the schema as part of the message.

The location of that global ID is determined (by the deserializer) by simply checking for the magic byte at the start of the message payload. If that byte is found, the global ID is read from the message payload (using the configured handler). If the magic byte is not found, the global 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 (because there is no appropriate API to do otherwise).

13.5. 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 Service Registry web console
- A curl command using the Service Registry API
- A Maven plugin 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

Having installed Service Registry, you 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: prices-value" \
  --data '{ (1)
      "type":"record",
      "name":"price",
      "namespace":"com.redhat",
      "fields":[{"name":"symbol","type":"string"},
      {"name":"price","type":"string"}]
    }'
  https://my-cluster-service-registry-myproject.example.com/api/artifacts -s (2)
```

(1) Avro schema artifact.

2 OpenShift route name that exposes Service Registry.

Maven plugin example

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

Configuration using a producer client example

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

1 The properties are registered. You can register properties against more than one node.

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

13.6. 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/api";
Properties props = new Properties();
props.putIfAbsent(AbstractKafkaSerDe.REGISTRY_URL_CONFIG_PARAM, registryUrl);
```

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

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

1 The deserializer provided by Service Registry.

2 The deserialization is in Apache Avro JSON format.

13.7. 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/api";
Properties props = new Properties();
props.putIfAbsent(AbstractKafkaSerDe.REGISTRY_URL_CONFIG_PARAM, 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(AbstractKafkaSerializer.REGISTRY_GLOBAL_ID_STRATEGY_CONFIG_PARAM,
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. Lookup strategy to find the global ID for the schema.

13.8. USING A SCHEMA FROM A KAFKA STREAMS APPLICATION

This procedure describes how to configure a Kafka Streams 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. Create and configure a REST client with the Service Registry. For example:

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

2. Configure the serializer, deserializer, and create the Kafka Streams client. For example:

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

1. The serializer provided by Service Registry.

2 The deserializer provided by Service Registry.

3 The deserialization is in Apache Avro format.

4 The Kafka Streams client application.

CHAPTER 14. 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 14.1, “Service Registry artifact types”
- Section 14.2, “Service Registry artifact states”
- Section 14.3, “Service Registry artifact metadata”
- Section 14.4, “Service Registry content rule types”
- Section 14.5, “Service Registry content rule maturity”

Additional resources

- For more detailed information, see the Apicurio Registry REST API documentation

14.1. SERVICE REGISTRY ARTIFACT TYPES

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

Table 14.1. Service Registry artifact types

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

14.2. SERVICE REGISTRY ARTIFACT STATES

These are the valid artifact states in Service Registry:

Table 14.2. Service Registry artifact states

State	Description
ENABLED	Basic state, all the operations are available.
DISABLED	The artifact and its metadata is viewable and searchable using the Service Registry web console, but its content cannot be fetched by any client.
DEPRECATED	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.

14.3. 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 14.3. Service Registry metadata properties

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

Property	Type	Editable

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.

14.4. SERVICE REGISTRY CONTENT RULE TYPES

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

Table 14.4. Service Registry content rule types

Type	Description
VALIDITY	Validate data before adding it to the registry. The possible configuration values for this rule are: ● **FULL**: The validation is both syntax and semantic. ● **SYNTAX_ONLY**: The validation is syntax only.

Type	Description
COMPATIBILITY	Ensure that newly added artifacts are compatible with previously added versions. The possible configuration values for this rule are: • **FULL**: The new artifact is forward and backward compatible with the most recently added artifact. • **FULL_TRANSITIVE**: The new artifact is forward and backward compatible with all previously added artifacts. • **BACKWARD**: Clients using the new artifact can read data written using the most recently added artifact. • **BACKWARD_TRANSITIVE**: Clients using the new artifact can read data written using all previously added artifacts. • **FORWARD**: Clients using the most recently added artifact can read data written using the new artifact. • **FORWARD_TRANSITIVE**: Clients using all previously added artifacts can read data written using the new artifact. • **NONE**: All backward and forward compatibility checks are disabled.

14.5. 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 14.5. 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

CHAPTER 15. SERVICE REGISTRY OPERATOR CONFIGURATION REFERENCE

This chapter provides detailed information on the custom resource used to configure the Service Registry Operator to deploy Service Registry:

- Section 15.1, “Service Registry Custom Resource”
- Section 15.2, “Service Registry CR spec”
- Section 15.3, “Service Registry CR status”
- Section 15.5, “Service Registry Operator labels”
- Section 15.4, “Service Registry managed resources”

15.1. SERVICE REGISTRY CUSTOM RESOURCE

The Service Registry Operator defines an **ApicurioRegistry** custom resource (CR) that represents a single deployment of Service Registry on OpenShift.

These resource objects are created and maintained by users to instruct the Service Registry Operator how to deploy and configure Service Registry.

Example ApicurioRegistry CR

The following command displays the **ApicurioRegistry** resource:

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

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

IMPORTANT

The Service Registry Operator currently only watches its own project namespace. Therefore you must create the **ApicurioRegistry** CR in the same namespace.

Additional resources

- Extending the Kubernetes API with Custom Resource Definitions

15.2. SERVICE REGISTRY CR SPEC

The **spec** is the part of the **ApicurioRegistry** CR that is used to provide the desired state or configuration for the Operator to achieve.

ApicurioRegistry CR spec contents

The following example block contains the full tree of possible **spec** configuration options. Some fields may not be required or should not be defined at the same time.

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

The following table describes each configuration option:

Table 15.1. ApicurioRegistry CR spec configuration options

Configuration option	type	Default value	Description
configuration	-	-	Section for configuration of Service Registry application
configuration/persistence	string	**mem**	Storage backend. One of **jpa**, **streams**, **infinispan**
configuration/dataSource	-	-	Database connection configuration for JPA storage backend
configuration/dataSource/url	string	*required*	Database connection URL string
configuration/dataSource/userName	string	*required*	Database connection user
configuration/dataSource/password	string	*empty*	Database connection password
configuration/streams	-	-	Kafka Streams storage backend configuration
configuration/streams/bootstrapServers	string	*required*	Kafka bootstrap server URL, for Streams storage backend
configuration/streams/applicationId	string	*ApicurioRegistry CR name*	Kafka Streams application ID
configuration/streams/applicationServerPort	string	**9000**	-
configuration/streams/security/tls	-	-	Section to configure TLS authentication for Kafka Streams storage backend
configuration/streams/security/tls/truststoreSecretName	string	*required*	Name of a secret containing TLS truststore for Kafka
configuration/streams/security/tls/keystoreSecretName	string	*required*	Name of a secret containing user TLS keystore
configuration/streams/security/scram/truststoreSecretName	string	*required*	Name of a secret containing TLS truststore for Kafka

Configuration option	type	Default value	Description
configuration/streams/security/scram/user	string	*required*	SCRAM user name
configuration/streams/security/scram/passwordSecretName	string	*required*	Name of a secret containing SCRAM user password
configuration/streams/security/scram/mechanism	string	**SCRAM-SHA-512**	SASL mechanism
configuration/infinispan	-	-	Infinispan persistence configuration section
configuration/infinispan/clusterName	string	*ApicurioRegistry CR name*	Infinispan cluster name
configuration/ui	-	-	Service Registry web console settings
configuration/ui/readOnly	string	**false**	Set Service Registry web console to read-only mode
configuration/logLevel	string	**INFO**	Service Registry operand log level. One of **INFO**, **DEBUG**
deployment	-	-	Section for operand deployment settings
deployment/replicas	positive integer	**1**	Number of Service Registry pods to deploy
deployment/host	string	*auto-generated from ApicurioRegistry CR name and namespace*	Host/URL where the Service Registry console and API are available

NOTE

If an option is marked as *required*, it might be conditional on other configuration options being enabled. Empty values might be accepted, but the Operator does not perform the specified action.

15.3. SERVICE REGISTRY CR STATUS

The **status** is the section of the CR managed by the Service Registry Operator that contains a description of the current deployment and application state.

ApicurioRegistry CR status contents

The **status** section contains the following fields:

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

Table 15.2. ApicurioRegistry CR status fields

Status field	Type	Description
image	string	Service Registry operand image that the Operator deploys. Might change based on the storage option selected in configuration.
deploymentName	string	Name of the **Deployment** or **DeploymentConfig** managed by the Operator, used to deploy the Service Registry.
serviceName	string	Name of the **Service** managed by the Operator, to expose the Service Registry operand as a service.
ingressName	string	Name of the **Ingress** managed by the Operator, to make the Service Registry accessible via HTTP. A **Route** is also created on OCP.
replicaCount	int32	Number of Service Registry operand pods deployed.
host	string	URL where the Service Registry UI and REST API are accessible.

15.4. SERVICE REGISTRY MANAGED RESOURCES

The resources managed by the Service Registry Operator when deploying Service Registry are as follows:

- **DeploymentConfig**
- **Service**
- **Ingress**
- **Route**
- **PodDisruptionBudget**

15.5. SERVICE REGISTRY OPERATOR LABELS

Resources managed by the Service Registry Operator are usually labeled as follows:

Table 15.3. Service Registry Operator labels for managed resources

Label	Description
app	Name of the Service Registry deployment that the resource belongs to, based on the name of the specified **ApicurioRegistry** CR.
apicur.io/type	Type of the deployment: **apicurio-registry** or **operator**
apicur.io/name	Name of the deployment: same value as **app** or **apicurio-registry-operator**
apicur.io/version	Version of the Service Registry or the Service Registry Operator
app.kubernetes.io/*	A set of recommended Kubernetes labels for application deployments.
com.company and **rht.*`**	Metering labels for Red Hat products.

Additional resources

- Recommended Kubernetes labels for application deployments

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.
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.
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.
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.
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 .