Red Hat Integration 2020.Q1

Getting Started with Service Registry

Getting Started with Service Registry

Last Updated: 2021-02-19

Red Hat Integration 2020.Q1 Getting Started with Service Registry

Getting Started with Service Registry

Legal Notice

Abstract

This guide introduces Service Registry, explains how to install in your storage environment, and how to manage artifacts stored in Service Registry.

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:

- Section 1.1, “Service Registry overview”
- Section 1.2, “Supported artifact types”

IMPORTANT

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

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

1.1. SERVICE REGISTRY OVERVIEW

Service Registry is a datastore for standard event schemas and API designs. You can use Service Registry to decouple the structure of your data from your applications and to share and manage your data structures 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 the registry at runtime without needing to redeploy. Developer teams can query the registry for existing schemas required by services deployed in production and can register new schemas required for new services.

Service Registry provides the following capabilities:

- Support for multiple payload formats for standard event schemas and API specifications.
- Apache Kafka-based storage in Red Hat AMQ Streams.
- Manage registry content using a REST API, a Maven plug-in, or a 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.
- Compatibility with existing Confluent schema registry client applications.

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

1.1.1. Service Registry artifacts

The items stored in Service Registry, such as event schemas and API specifications, are known as *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 Section 1.2, “Supported artifact types”.

1.1.2. Registry REST API

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. 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. You can also manage the lifecycle state of a version: enabled, disabled, or deprecated.

Artifact metadata

Manage details such as when the artifact was created, last updated, and so on.

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.

For detailed information, see the Apicurio Registry REST API documentation .

Compatibility with other schema registries

The Service Registry REST API is compatible with the Confluent schema registry REST API. This means that applications using Confluent client libraries can use Service Registry instead as a drop-in replacement. For more details, see Replacing Confluent Schema Registry with Red Hat Integration Service Registry.

1.1.3. Storage options

Service Registry supports the following underlying storage implementations for artifacts:

- Red Hat AMQ Streams 1.4
- Red Hat AMQ Streams 1.3

1.1.4. Available distributions

Table 1.1. Service Registry distributions

Distribution	Location
Container image	Red Hat Container Catalog
Maven repository	Software Downloads for Red Hat Fuse
Full Maven repository (with all dependencies)	Software Downloads for Red Hat Fuse
Source code	Software Downloads for Red Hat Fuse

NOTE

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

1.1.5. Client serializers/deserializers

Event-based producer applications can use serializers to encode messages that conform to a specific event schema. Consumer applications can then use deserializers to validate that messages have been serialized using the correct schema, based on a specific schema ID. Service Registry provides client serializers/deserializers 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 serializer/deserializer implementations for these message types, which 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

For instructions on how to use the Service Registry client serializer/deserializer for Apache Avro in AMQ Streams producer and consumer applications, see Using AMQ Streams on OpenShift .

1.1.6. Registry demonstration

Service Registry provides an open source demonstration example 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.

This demonstration also provides simple examples of both Avro and JSON Schema serialization/deserialization with storage in Apache Kafka: https://github.com/Apicurio/apicurio-registry-demo/tree/master/src/main/java/io/apicurio/registry/demo/simple

For another demonstration example with detailed instructions on Avro serialization/deserialization with storage in Apache Kafka, see the Red Hat Developer article on Getting Started with Red Hat Integration Service Registry.

1.2. SUPPORTED ARTIFACT TYPES

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

Table 1.2. 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

CHAPTER 2. INTRODUCTION TO SERVICE REGISTRY RULES

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

2.1. RULES FOR REGISTRY CONTENT

To govern content evolution in the registry, you can configure optional rules for artifacts added to the registry, as a post-installation step. Any rules configured for an artifact must pass before a new artifact version can be uploaded to the registry. 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 company standards
- New content includes breaking changes to the current artifact version

2.1.1. When rules are applied

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

- Creating an artifact
- Updating an artifact
- Creating 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 any currently configured global rules are applied.

2.1.2. 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. For example, a validation rule might use a **Map<String,String>**, or a compatibility rule might use a single property of **BACKWARD** for compatibility with existing versions.

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.

2.2. SUPPORTED RULE TYPES

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

Table 2.1. Service Registry rule types

Type	Description
VALIDITY	Validates 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.
COMPATIBILITY	Ensures 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.

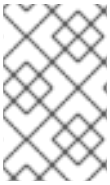
CHAPTER 3. INSTALLING SERVICE REGISTRY

This chapter explains how to set up storage in AMQ Streams and how to install and run Service Registry:

- Section 3.1, “Setting up AMQ Streams storage on OpenShift”
- Section 3.2, “Installing Service Registry with AMQ Streams storage on OpenShift”

Prerequisites

- Section 1.1, “Service Registry overview”



NOTE

You can install more than one instance of Service Registry depending on your environment. The number of instances depends on your storage, for example, Kafka cluster configuration, and on the number and type of artifacts stored in the registry.

3.1. SETTING UP AMQ STREAMS STORAGE ON OPENSHIFT

This topic explains how to install and configure Red Hat AMQ Streams storage for Service Registry on OpenShift. The following versions are supported:

- AMQ Streams 1.4 or 1.3
- OpenShift 4.3, 4.2, or 3.11

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.
- You must have installed AMQ Streams using the instructions in Using AMQ Streams on OpenShift.
  Alternatively, to install using the simple demonstration example shown in this section, you must have:
  - Downloaded AMQ Streams from the Red Hat customer portal
  - OpenShift cluster adminstrator access

Procedure

1. If you do not already have AMQ Streams installed, install AMQ Streams on your OpenShift cluster. For example, enter the following command from your AMQ Streams download directory:

   ```
   oc apply -f install/cluster-operator/
   ```

2. If you do not already have a Kafka cluster set up, create a new Kafka cluster with AMQ Streams. For example:

   ```
   $ cat << EOF | oc create -f -
   ```

```
apiVersion: kafka.strimzi.io/v1beta1
kind: Kafka
metadata:
 name: my-cluster
spec:
 kafka:
   replicas: 3
   listeners:
     external:
       type: route
   storage:
     type: ephemeral
 zookeeper:
   replicas: 3
   storage:
     type: ephemeral
 entityOperator:
   topicOperator: {}
EOF
```

This simple example creates a cluster with 3 Zookeeper nodes and 3 Kafka nodes using ephemeral storage. All data is lost when the Pods are no longer running on OpenShift.

3. Create the required **storage-topic** to store Service Registry artifacts in AMQ Streams. For example:

```
$ cat << EOF | oc create -f -
apiVersion: kafka.strimzi.io/v1beta1
kind: KafkaTopic
metadata:
  name: storage-topic
  labels:
    strimzi.io/cluster: my-cluster
spec:
  partitions: 3
  replicas: 3
  config:
    cleanup.policy: compact
EOF
```

4. Create the required **global-id-topic** to store Service Registry global IDs in AMQ Streams. For example:

```
$ cat << EOF | oc create -f -
apiVersion: kafka.strimzi.io/v1beta1
kind: KafkaTopic
metadata:
  name: global-id-topic
  labels:
    strimzi.io/cluster: my-cluster
spec:
  partitions: 3
  replicas: 3
  config:
    cleanup.policy: compact
EOF
```

Additional resources

For more details on installing AMQ Streams and on creating Kafka clusters and topics:

- Using AMQ Streams on OpenShift
- How to run AMQ Streams on Minishift

3.2. INSTALLING SERVICE REGISTRY WITH AMQ STREAMS STORAGE ON OPENSHIFT

This topic explains how to install and run Service Registry with storage in Red Hat AMQ Streams using an OpenShift template.

The following versions are supported:

- AMQ Streams 1.4 or 1.3
- OpenShift 4.3, 4.2, or 3.11

Prerequisites

- You must have an OpenShift cluster with cluster administrator access.
- You must have already installed AMQ Streams and configured your Kafka cluster on OpenShift. See Section 3.1, "Setting up AMQ Streams storage on OpenShift" .
- Ensure that you can access the Service Registry image in the Red Hat Container Catalog :
  - Create a service account and pull secret for the image. For details, see Container Service Accounts.
  - Download the pull secret and submit it to your OpenShift cluster. For example:

```
$ oc create -f 11223344_service-registry-secret.yaml --namespace=myproject
```

Procedure

1. Get the Service Registry OpenShift template.
2. Enter the following command to get the name of the Kafka bootstrap service running in AMQ Streams on your OpenShift cluster:

```
$ oc get services | grep .*kafka-bootstrap
```

3. Create a new OpenShift application using the **oc new-app** command. For example:

```
$ oc new-app service-registry-template.yml \
  -p KAFKA_BOOTSTRAP_SERVERS=my-cluster-kafka-bootstrap:9092 \
  -p REGISTRY_ROUTE=my-cluster-service-registry-myproject.example.com \
  -p APPLICATION_ID=my-kafka-streams-app
```

You must specify the following arguments:

- **service-registry-template.yml**: The OpenShift template file for Service Registry.
- **KAFKA_BOOTSTRAP_SERVERS**: The name of the Kafka bootstrap service on your OpenShift cluster, followed by the Kafka broker port. For example: **my-cluster-kafka-bootstrap:9092**.
- **REGISTRY_ROUTE**: The name of the OpenShift route to expose Service Registry, which is based on your OpenShift cluster environment. For example: **my-cluster-service-registry-myproject.example.com**.
- **APPLICATION_ID**: The name of your AMQ Streams application. For example: **my-kafka-streams-app**.
  You can also specify the following environment variables using the **-e** option:
- **APPLICATION_SERVER_HOST**: The IP address of your Kafka Streams application server host, which is required in a multi-node Kafka configuration. Defaults to **$(POD_IP)**.
- **APPLICATION_SERVER_PORT**: The port number of your Kafka Streams application server, which is required in a multi-node Kafka configuration. Defaults to **9000**.

4. Verify the command output when complete. For example:

```
Deploying template "myproject/service-registry" for "service-registry-template.yml" to project myproject

 service-registry
 ---------
 Congratulations on deploying Service Registry into OpenShift!

 All components have been deployed and configured.

  * With parameters:
    * Registry Route Name=my-cluster-service-registry-myproject.example.com
    * Registry Max Memory Limit=1300Mi
    * Registry Memory Requests=600Mi
    * Registry Max CPU Limit=1
    * Registry CPU Requests=100m
    * Kafka Bootstrap Servers=my-cluster-kafka-bootstrap:9092
    * Kafka Application ID=my-kafka-streams-app

--> Creating resources ...
    imagestream.image.openshift.io "registry" created
    service "service-registry" created
    deploymentconfig.apps.openshift.io "service-registry" created
    route.route.openshift.io "service-registry" created
--> Success
    Access your application via route 'my-cluster-service-registry-myproject.example.com'
```

5. Enter **oc status** to view your Service Registry installation on OpenShift.

Additional resources

- For sample REST API requests, see the Registry REST API documentation .
- For details on example client applications:

- https://github.com/Apicurio/apicurio-registry-demo
- Getting Started with Red Hat Integration Service Registry

CHAPTER 4. MANAGING ARTIFACTS IN SERVICE REGISTRY

This chapter provides details on different approaches to managing artifacts in Service Registry:

- Section 4.1, "Managing artifacts using the Registry REST API"
- Section 4.2, "Managing artifacts using the Service Registry Maven plug-in"
- Section 4.3, "Managing artifacts in a Java client application"

4.1. MANAGING ARTIFACTS USING THE REGISTRY REST API

The Registry REST API enables client applications to manage artifacts in the registry, for example, in a CI/CD pipeline deployed in production. This REST API provides create, read, update, and delete operations for artifacts, versions, metadata, and rules stored in the registry.

When creating 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.

This chapter shows a simple curl-based example of using the Registry REST API to create and retrieve a Apache Avro schema artifact in the registry.

Prerequisites

- See Section 1.1.2, "Registry REST API".
- Service Registry must be installed and running in your environment. For details, see Chapter 3, *Installing Service Registry*.

Procedure

1. Create an artifact in the registry using the **/artifacts** operation. The following example **curl** command creates 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/artifacts
```

   This example shows creating 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 created. 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 https://MY-REGISTRY-URL/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 Registry REST API documentation .

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

Service Registry provides a Maven plug-in to enable you 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.

Prerequisites

- Service Registry must be installed and running in your environment
- Maven must be installed and configured in your environment

Procedure

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

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

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

2. You can also update your Maven **pom.xml** file to download a previously registered artifact:

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

(1) Specify **download** as the execution goal.

(2) You must specify the Service Registry URL.

(3) You can download multiple artifacts to a specified directory using the artifact ID.

Additional resources

- For more details on the Maven plug-in, see https://github.com/Apicurio/apicurio-registry-demo.

4.3. MANAGING ARTIFACTS IN A JAVA CLIENT APPLICATION

You can also manage artifacts in the registry using a Java client application. The Service Registry Java client classes enable you to create, read, update, or delete artifacts in the registry.

Prerequisites

- See Section 1.1.5, “Client serializers/deserializers”
- You must have implemented a client application in Java that imports the Service Registry client classes: **io.apicurio.registry.client.RegistryClient**
- Service Registry must be installed and running in your environment.

Procedure

- Update your client application to create a new artifact in the registry. The following example shows creating an Apache Avro schema artifact from a Kafka producer client application:

```
String registryUrl_node1 = PropertiesUtil.property(clientProperties, "registry.url.node1",
 "https://my-cluster-service-registry-myproject.example.com"); (1)
  try (RegistryService service = RegistryClient.create(registryUrl_node1))
  {
   String artifactId = ApplicationImpl.INPUT_TOPIC + "-value";
   try {
    service.getArtifactMetaData(artifactId); (2)
   }
   catch (WebApplicationException e) {
    CompletionStage < ArtifactMetaData > csa = service.createArtifact( (3)
       ArtifactType.AVRO,
       artifactId,
        new ByteArrayInputStream(LogInput.SCHEMA$.toString().getBytes())
       );
   csa.toCompletableFuture().get();
   }
}
```

(1) Configure the client application with the Service Registry URL in the client properties. You can create properties for more than one registry node.

(2) Check to see if the schema artifact already exists in the registry based on the artifact ID.

(3) Create the new schema artifact in the registry.

Additional resources

- For an example Java client application, see https://github.com/Apicurio/apicurio-registry-demo.
- For more details on Kafka client applications, see: Using AMQ Streams on OpenShift .

APPENDIX A. USING YOUR SUBSCRIPTION

Fuse 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 Fuse** entries in the **Integration and Automation** category.
3. Select the desired Fuse 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 .