Red Hat build of Quarkus 3.2

Service binding

Last Updated: 2024-04-04

Legal Notice

Abstract

Explore service binding and workload projection to understand their need for connection to other services for additional information retrieval.

Table of Contents

MAKING OPEN SOURCE MORE INCLUSIVE

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

CHAPTER 1. SERVICE BINDING

The following chapter provides information about service binding and workload projection that were added to Red Hat build of Quarkus in version 2.7.5 and are in the state of Technology Preview in version 3.2.

Generally, OpenShift applications and services also referred to as deployable workloads, need to be connected to other services for retrieving additional information, such as service URLs or credentials.

The Service Binding Operator facilitates retrieval of the necessary information, which is then made available to applications and service-binding tools like the **quarkus-kubernetes-service-binding** extension through environment variables without directly influencing or determining the use of the extension tool itself.

Quarkus supports the Service binding specification for Kubernetes to bind services to applications.

Specifically, Quarkus implements the workload projection part of the specification, enabling applications to bind to services like databases or brokers, requiring only minimal configuration.

To enable service binding for the available extensions, include the **quarkus-kubernetes-service-binding** extension to the application dependencies.

- You can use the following extensions for service binding and for workload projection:
  - **quarkus-jdbc-mariadb**
  - **quarkus-jdbc-mssql**
  - **quarkus-jdbc-mysql**
  - **quarkus-jdbc-postgresql**
  - **quarkus-mongo-client** - Technology Preview
  - **quarkus-kafka-client**
  - **quarkus-smallrye-reactive-messaging-kafka**
  - **quarkus-reactive-mssql-client** - Technology Preview
  - **quarkus-reactive-mysql-client**
  - **quarkus-reactive-pg-client**

1.1. WORKLOAD PROJECTION

Workload projection is a process of obtaining the configuration for services from the Kubernetes cluster. This configuration takes the form of directory structures that follow certain conventions and are attached to an application or a service as a mounted volume.

The **kubernetes-service-binding** extension uses this directory structure to create configuration sources, which allows you to configure additional modules, such as databases or message brokers.

You can use workload projection during application development to connect your application to a development database or other locally run services without changing the application code or configuration.

For an example of a workload projection where the directory structure is included in the test resources and passed to an integration test, see the Kubernetes Service Binding datasource GitHub repository.

NOTE

- The **k8s-sb** directory is the root of all service bindings.
  In this example, only one database called **fruit-db** is intended to be bound. This binding database has the **type** file, which specifies **postgresql** as the database type, while the other files in the directory provide the necessary information to establish the connection.
- When your Red Hat build of Quarkus project obtains information from **SERVICE_BINDING_ROOT** environment variables that are set by OpenShift Container Platform, you can locate generated configuration files that are present in the file system and use them to map the configuration-file values to properties of certain extensions.

1.2. INTRODUCTION TO SERVICE BINDING OPERATOR

The Service Binding Operator is an Operator that implements the Service Binding Specification for Kubernetes and is meant to simplify the binding of services to an application.

Containerized applications that support workload projection obtain service binding information in the form of volume mounts. The Service Binding Operator reads binding service information and mounts it to the application containers that need it.

The correlation between application and bound services is expressed through the **ServiceBinding** resources, which declares the intent of what services are meant to be bound to what application.

The Service Binding Operator watches for **ServiceBinding** resources, which inform the Operator what applications are meant to be bound with what services. When a listed application is deployed, the Service Binding Operator collects all the binding information that must be passed to the application and then upgrades the application container by attaching a volume mount with the binding information.

The Service Binding Operator completes the following actions:

- Observes **ServiceBinding** resources for workloads bound to a particular service.
- Applies the binding information to the workload using volume mounts.

The following chapter describes the automatic and semi-automatic service binding approaches and their use cases. The **kubernetes-service-binding** extension generates a **ServiceBinding** resource with either approach. With the semi-automatic approach, users must manually provide a configuration for target services. With the automatic approach, no additional configuration is needed for a limited set of services generating the **ServiceBinding** resource.

Additional resources

- Workload projection

1.3. SEMI-AUTOMATIC SERVICE BINDING

A service binding process starts with a user specification of required services that will be bound to a certain application. This expression is summarized in the **ServiceBinding** resource generated by the **kubernetes-service-binding** extension. The use of the **kubernetes-service-binding** extensions helps

users to generate **ServiceBinding** resources with minimal configuration, therefore simplifying the process overall.

The Service Binding Operator responsible for the binding process then reads the information from the **ServiceBinding** resource and mounts the required files to a container accordingly.

- An example of the **ServiceBinding** resource:

```
apiVersion: binding.operators.coreos.com/v1beta1
kind: ServiceBinding
metadata:
 name: binding-request
 namespace: service-binding-demo
spec:
 application:
   name: java-app
   group: apps
   version: v1
   resource: deployments
 services:
 - group: postgres-operator.crunchydata.com
   version: v1beta1
   kind: Database
   name: db-demo
   id: postgresDB
```

NOTE

- The **quarkus-kubernetes-service-binding** extension provides a more compact way of expressing the same information. For example:

```
quarkus.kubernetes-service-binding.services.db-demo.api-
version=postgres-operator.crunchydata.com/v1beta1
quarkus.kubernetes-service-binding.services.db-demo.kind=Database
```

After adding the earlier configuration properties inside your **application.properties**, the **quarkus-kubernetes**, in combination with the **quarkus-kubernetes-service-binding** extension, automatically generates the **ServiceBinding** resource.

The earlier mentioned **db-demo** property-configuration identifier now has a double role and also completes the following actions:

- Correlates and groups **api-version** and **kind** properties together.
- Defines the **name** property for the custom resource, which you can edit later if needed. For example:

```
quarkus.kubernetes-service-binding.services.db-demo.api-version=postgres-
operator.crunchydata.com/v1beta1
quarkus.kubernetes-service-binding.services.db-demo.kind=Database
quarkus.kubernetes-service-binding.services.db-demo.name=my-db
```

Additional resources

- How to use Quarkus with the Service Binding Operator
- List of bindable Operators

1.4. GENERATING A SERVICEBINDING CUSTOM RESOURCE BY USING THE SEMI-AUTOMATIC METHOD

You can generate a **ServiceBinding** resource semi-automatically. The following procedure shows the OpenShift Container Platform deployment process, including the installation of operators for configuring and deploying an application.

In this procedure, you install the Service Binding Operator and the PostgreSQL Operator from Crunchy Data.

IMPORTANT

PostgreSQL Operator is a third-party component. For PostgreSQL Operator support policies and terms of use, contact the software vendor Crunchy Data.

Then, the procedure involves creating a PostgreSQL cluster, setting up a straightforward application, and subsequently deploying and binding it to the provisioned cluster.

Prerequisites

- You have created an OpenShift Container Platform 4.11 cluster.
- You have administrator access to OperatorHub and OpenShift Container Platform to install cluster-wide operators from OperatorHub.
- You have installed:
  - The OpenShift, **oc**, orchestration tool
  - Maven and Java

Procedure

The steps in the following procedure use the HOME (~) directory as a saving and installation destination.

1. Install the Service Binding Operator version 1.3.3 and higher using the Installing the Service Binding Operator from the OpenShift Container Platform web UI procedure.
   a. Verify the installation:

   ```
   oc get csv -w
   ```

   Proceed to the next step when the **phase** of the Service Binding Operator is set to **Succeeded**.
2. Install the Crunchy PostgreSQL Operator from OperatorHub by using either the web console or CLI.
   a. Verify the installation:

   ```
   oc get csv -w
   ```

Proceed to the next step when the operator's **phase** is set to **Succeeded**.

3. Create a PostgreSQL cluster:

   a. Create a new OpenShift Container Platform namespace, which will be used for creating a cluster and deploying your application later. This namespace will be referred to as **demo** throughout the procedure.

```
oc new-project demo
```

   b. Create the following custom resource and save it as **pg-cluster.yml**:

```
apiVersion: postgres-operator.crunchydata.com/v1beta1
kind: PostgresCluster
metadata:
  name: hippo
spec:
  openshift: true
  image: registry.developers.crunchydata.com/crunchydata/crunchy-postgres:ubi8-14.2-1
  postgresVersion: 14
  instances:
    - name: instance1
      dataVolumeClaimSpec:
        accessModes:
        - "ReadWriteOnce"
        resources:
          requests:
            storage: 1Gi
  backups:
    pgbackrest:
      image: registry.developers.crunchydata.com/crunchydata/crunchy-pgbackrest:ubi8-
2.38-0
      repos:
      - name: repo1
        volume:
          volumeClaimSpec:
            accessModes:
            - "ReadWriteOnce"
            resources:
              requests:
                storage: 1Gi
```

NOTE

This YAML has been reused from Service Binding Operator Quickstart.

   c. Apply the created custom resource:

```
oc apply -f ~/pg-cluster.yml
```

NOTE

This command assumes that you saved the **pg-cluster.yml** file in the HOME directory.

d. Check the pods to verify the installation:

```
oc get pods -n demo
```

- Wait for the Pods to enter the **READY** state, indicating the installation is complete.

4. Create a Quarkus application that binds to the PostgreSQL database.
The application you are creating is a basic **todo** application that connects to PostgreSQL using Hibernate and Panache.

a. Generate the application:

```
mvn com.redhat.quarkus.platform:quarkus-maven-plugin:3.2.11.Final-redhat-
00001:create \
  -DplatformGroupId=com.redhat.quarkus.platform \
  -DplatformVersion=3.2.11.Final-redhat-00001 \
  -DprojectGroupId=org.acme \
  -DprojectArtifactId=todo-example \
  -DclassName="org.acme.TodoResource" \
  -Dpath="/todo"
```

b. Add all required extensions for connecting to PostgreSQL, generating all required resources, and building a container image for our application:

```
./mvnw quarkus:add-extension -Dextensions="resteasy-reactive-jackson,jdbc-
postgresql,hibernate-orm-panache,openshift,kubernetes-service-binding"
```

c. Create a simple entity, as outlined in the following example:

```
package org.acme;

import jakarta.persistence.Column;
import jakarta.persistence.Entity;

import io.quarkus.hibernate.orm.panache.PanacheEntity;

@Entity
public class Todo extends PanacheEntity {

    @Column(length = 40, unique = true)
    public String title;

    public boolean completed;

    public Todo() {
    }

    public Todo(String title, Boolean completed) {
        this.title = title;
    }

}
```

d. Expose the entity:

```
package org.acme;

import jakarta.transaction.Transactional;
import jakarta.ws.rs.*;
import jakarta.ws.rs.core.Response;
import jakarta.ws.rs.core.Response.Status;
import java.util.List;

  @Path("/todo")
  public class TodoResource {

    @GET
    @Path("/")
    public List<Todo> getAll() {
      return Todo.listAll();
    }

    @GET
    @Path("/{id}")
    public Todo get(@PathParam("id") Long id) {
        Todo entity = Todo.findById(id);
        if (entity == null) {
            throw new WebApplicationException("Todo with id of " + id + " does not exist.",
Status.NOT_FOUND);
        }
        return entity;
    }

    @POST
    @Path("/")
    @Transactional
    public Response create(Todo item) {
        item.persist();
        return Response.status(Status.CREATED).entity(item).build();
    }

    @GET
    @Path("/{id}/complete")
    @Transactional
    public Response complete(@PathParam("id") Long id) {
        Todo entity = Todo.findById(id);
        entity.id = id;
        entity.completed = true;
        return Response.ok(entity).build();
    }


    @DELETE
    @Transactional
    @Path("/{id}")
    public Response delete(@PathParam("id") Long id) {
        Todo entity = Todo.findById(id);
        if (entity == null) {
            throw new WebApplicationException("Todo with id of " + id + " does not exist.",
Status.NOT_FOUND);
        }
```

```
            entity.delete();
            return Response.noContent().build();
        }

    }
```

5. Bind to the target PostgreSQL cluster by generating a **ServiceBinding** resource.

    a. Provide the service coordinates to generate the binding and configure the data source:

    - apiVersion: **postgres-operator.crunchydata.com/v1beta1**
    - kind: **PostgresCluster**
    - name: **pg-cluster**
      This is accomplished by setting a **quarkus.kubernetes-service-binding.services.<id>.** prefix, as demonstrated in the example below. The **id** is used to group properties together and can be assigned any value.

```
quarkus.kubernetes-service-binding.services.my-db.api-version=postgres-operator.crunchydata.com/v1beta1
quarkus.kubernetes-service-binding.services.my-db.kind=PostgresCluster
quarkus.kubernetes-service-binding.services.my-db.name=hippo

quarkus.datasource.db-kind=postgresql
quarkus.hibernate-orm.database.generation=drop-and-create
quarkus.hibernate-orm.sql-load-script=import.sql
```

    b. Create an **import.sql** script with some initial data:

```
INSERT INTO todo(id, title, completed) VALUES (nextval('hibernate_sequence'), 'Finish the blog post', false);
```

6. Deploy the application, including **ServiceBinding**, and apply it to the cluster:

```
mvn clean install -Dquarkus.kubernetes.deploy=true -DskipTests
```

   Wait for the deployment to finish.

Verification

1. Verify the deployment:

```
oc get pods -n demo -w
```

2. Verify the installation:

    a. Port forward to the HTTP port locally, and then access the **/todo** endpoint.

```
oc port-forward service/todo-example 8080:80
```

    b. Open the following URL in a web browser:

```
http://localhost:8080/todo
```

Additional resources

- For more information, see the Service Binding Operator section of the Quick Start guide.

1.5. AUTOMATIC SERVICE BINDING

The **quarkus-kubernetes-service-binding** extension can automatically generate the **ServiceBinding** resource when it detects an application needing access to external services provided by compatible bindable operators.

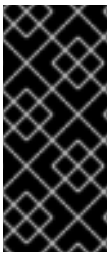
NOTE

Automatic service binding can only be generated for a limited set of service types.

In alignment with the established Kubernetes and Quarkus service terminology, this chapter uses the term "kinds" to refer to these service types.

Table 1.1. Operators that support automatic service binding

Service binding type	Operator	API version	Kind
postgresql	CrunchyData Postgres	postgres-operator.crunchydata.com/v1beta1	PostgresCluster
mysql	Percona XtraDB Cluster	pxc.percona.com/v1-9-0	PerconaXtraDBCluster
mongo	Percona MongoDB	psmdb.percona.com/v1-9-0	PerconaServerMongoDB



IMPORTANT

- Red Hat build of Quarkus 3.2 support for MongoDB Operator is provided as a Technology Preview and applies to the client only.
- See the Quarkus application configurator page for a list of supported Panache extensions in Red Hat build of Quarkus 3.2.

1.5.1. Automatic datasource binding

For traditional databases, automatic binding is initiated whenever a datasource is configured as follows:

```
quarkus.datasource.db-kind=postgresql
```

The configuration mentioned earlier, in conjunction with the presence of extensions such as **quarkus-datasource**, **quarkus-jdbc-postgresql**, **quarkus-kubernetes**, and **quarkus-kubernetes-service-binding** in the application, leads to the creation of the **ServiceBinding** resource for the **postgresql** database type.

By using the **apiVersion** and **kind** properties of the Operator resource, which matches the used **postgresql** Operator, the generated **ServiceBinding** resource binds the service or resource to the application.

When you do not specify a name for your database service, the value of the **db-kind** property is used as the default name.

```
services:
- apiVersion: postgres-operator.crunchydata.com/v1beta1
  kind: PostgresCluster
  name: postgresql
```

Specified the name of the datasource as follows:

```
quarkus.datasource.fruits-db.db-kind=postgresql
```

The **service** in the generated **ServiceBinding** then displays as follows:

```
services:
- apiVersion: postgres-operator.crunchydata.com/v1beta1
  kind: PostgresCluster
  name: fruits-db
```

Similarly, if you use **mysql**, the name of the datasource can be specified as follows:

```
quarkus.datasource.fruits-db.db-kind=mysql
```

The generated **service** contains the following:

```
services:
- apiVersion: pxc.percona.com/v1-9-0
  kind: PerconaXtraDBCluster
  name: fruits-db
```

1.5.1.1. Customizing automatic service binding

While the automatic service binding feature was developed to eliminate as much of the manual configuration as possible, there are scenarios where you might need to modify the generated **ServiceBinding** resource manually.

The generation process exclusively relies on information extracted from the application and the knowledge of the supported Operators, which might not reflect what is deployed in the cluster.

The generated resource is based purely on the knowledge of the supported bindable operators for popular service kinds and a set of conventions that were developed to prevent possible mismatches, such as:

- The target resource name does not match the datasource name.
- A specific Operator needs to be used rather than the default Operator for that service kind.
- Version conflicts occur when a user needs to use a version other than the default or the latest.

Conventions:

- Target resource coordinates are established according to the Operator type and service kind.
- By default, the target resource name aligns with the service kind, such as **postgresql**, **mysql**, or **mongo**.
- In the case of named datasources, the datasource name is used.
- The client's name is used for named **mongo** clients.

Example 1: Name mismatch

For cases where you need to modify the generated **ServiceBinding** to fix a name mismatch, use the **quarkus.kubernetes-service-binding.services** properties and specify the service's name as the service key.

The **service key** is usually the name of the service, for example, the name of the datasource or the name of the **mongo** client. When this value is unavailable, the datasource type, such as **postgresql**, **mysql**, or **mongo**, is used instead.

To avoid naming conflicts between different types of services, prefix the **service key** with a specific datasource type, such as **postgresql-*<person>***.

The following example shows how to customize the **apiVersion** property of the **PostgresCluster** resource:

```
quarkus.datasource.db-kind=postgresql
quarkus.kubernetes-service-binding.services.postgresql.api-version=postgres-
operator.crunchydata.com/v1beta2
```

Example 2: Application of a custom name for a datasource

In Example 1, the service key **db-kind** (**postgresql**) was used. In this instance, following the convention, the datasource name (**fruits-db**) is used because the datasource is named.

The following example shows that for a named datasource, the datasource name is used as the name of the target resource:

```
quarkus.datasource.fruits-db.db-kind=postgresql
```

This has the same effect as the following configuration:

```
quarkus.kubernetes-service-binding.services.fruits-db.api-version=postgres-
operator.crunchydata.com/v1beta1
quarkus.kubernetes-service-binding.services.fruits-db.kind=PostgresCluster
quarkus.kubernetes-service-binding.services.fruits-db.name=fruits-db
```

Additional resources

- For additional information about the available properties, see the workload projection part of the Kubernetes service binding specification.

Revised on 2024-04-04 11:42:46 UTC