Red Hat build of Quarkus 3.8

サービスバインディング

Last Updated: 2024-09-17

法律上の通知

概要

サービスバインディングとワークロードプロジェクションの詳細を説明し、追加情報を取得するには他のサービスに接続する必要がある点について理解を深めます。

目次

RED HAT BUILD OF QUARKUS ドキュメントへのフィードバックの提供

エラーを報告したり、ドキュメントを改善したりするには、Red Hat Jira アカウントにログインし、課題を送信してください。Red Hat Jira アカウントをお持ちでない場合は、アカウントを作成するように求められます。

手順

1. 次のリンクをクリックして **チケットを作成します**。
2. **Summary** に課題の簡単な説明を入力します。
3. **Description** に課題や機能拡張の詳細な説明を入力します。問題があるドキュメントのセクションへの URL も記載してください。
4. **Submit** をクリックすると、課題が作成され、適切なドキュメントチームに転送されます。

多様性を受け入れるオープンソースの強化

Red Hat では、コード、ドキュメント、Web プロパティーにおける配慮に欠ける用語の置き換えに取り組んでいます。まずは、マスター (master)、スレーブ (slave)、ブラックリスト (blacklist)、ホワイトリスト (whitelist) の 4 つの用語の置き換えから始めます。この取り組みは膨大な作業を要するため、今後の複数のリリースで段階的に用語の置き換えを実施して参ります。詳細は、Red Hat CTO である Chris Wright のメッセージ をご覧ください。

第1章 サービスバインディング

次の章では、Red Hat build of Quarkus バージョン 2.7.5 バージョン 2.7.5 で追加され、バージョン 3.8 では テクノロジープレビュー になっているサービスバインディングとワークロードプロジェクションを説明します。

一般的に、OpenShift のアプリケーションとサービスはデプロイ可能なワークロードとも呼ばれ、サービスの URL や認証情報などの追加情報を取得するために、他のサービスに接続する必要があります。

Service Binding Operator は、簡単に必要な情報を取得し、拡張ツール自体の使用に直接影響を与えたり決定したりすることなく、環境変数を通じてアプリケーションや **quarkus-kubernetes-service-binding** 拡張などのサービスバインディングツールで利用できるようにします。

Quarkus は、サービスをアプリケーションにバインドするための Service binding specification for Kubernetes をサポートします。

具体的には、Quarkus は仕様の ワークロードプロジェクション 部分を実装しているため、アプリケーションはデータベースやブローカーなどのサービスを最小限の設定でバインドできます。

利用可能なエクステンションのサービスバインディングを有効にするには、アプリケーションの依存関係に **quarkus-kubernetes-service-binding** エクステンションを含めます。

- サービスバインディングとワークロードプロジェクションには、次のエクステンションを使用できます。
  - **quarkus-jdbc-mariadb**
  - **quarkus-jdbc-mssql**
  - **quarkus-jdbc-mysql**
  - **quarkus-jdbc-postgresql**
  - **quarkus-mongo-client** - テクノロジープレビュー
  - **quarkus-kafka-client**
  - **quarkus-smallrye-reactive-messaging-kafka**
  - **quarkus-reactive-mssql-client** - テクノロジープレビュー
  - **quarkus-reactive-mysql-client**
  - **quarkus-reactive-pg-client**

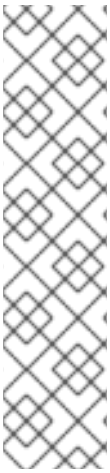
1.1. ワークロードプロジェクション

ワークロードプロジェクションは、Kubernetes クラスターからサービスの設定を取得するプロセスです。この設定は、特定の規則に従うディレクトリー構造の形式をとり、マウントされたボリュームとしてアプリケーションまたはサービスに接続されます。

kubernetes-service-binding エクステンションは、このディレクトリー構造を使用して設定ソースを作成します。これにより、データベースやメッセージブローカーなどの追加モジュールを設定できるようになります。

アプリケーション開発中にワークロードプロジェクションを使用して、アプリケーションのコードや設定を変更せずに、アプリケーションを開発データベースやローカルで実行される他のサービスに接続できます。

ディレクトリー構造がテストリソースに含まれ、結合テストに渡されるワークロードプロジェクションの例は、Kubernetes Service Binding datasource GitHub リポジトリーを参照してください。



注記

- **k8s-sb** ディレクトリーは、すべてのサービスバインディングのルートです。この例では、**fruit-db** という名前データベース 1 つのみをバインドします。このバインディングデータベースには、データベースタイプとして **postgresql** を指定する **type** ファイルがあります。ディレクトリー内の他のファイルは、接続を確立するために必要な情報を提供します。
- Red Hat build of Quarkus プロジェクトが、OpenShift Container Platform によって設定された **SERVICE_BINDING_ROOT** 環境変数から情報を取得すると、ファイルシステム内に存在する生成された設定ファイルを見つけ、それを使用して設定ファイルの値を特定のエクステンションのプロパティーにマップできます。

1.2. SERVICE BINDING OPERATOR の概要

Service Binding Operator は、Service Binding Specification for Kubernetes を実装する Operator であり、アプリケーションへのサービスのバインドを単純化するために使用できます。

ワークロードプロジェクション をサポートするコンテナー化されたアプリケーションは、ボリュームマウントのフォーマットでサービスバインディング情報を取得します。Service Binding Operator は、バインディングサービス情報を読み取り、それを必要とするアプリケーションコンテナーにマウントします。

アプリケーションとバインドされたサービスの相関関係は、どのサービスがどのアプリケーションにバインドされるかを宣言する **ServiceBinding** リソースを通して表現されます。

Service Binding Operator は **ServiceBinding** リソースを監視します。このリソースは、どのアプリケーションがどのサービスにバインドされるかを Operator に通知します。リストされたアプリケーションがデプロイされると、Service Binding Operator はアプリケーションに渡す必要があるすべてのバインディング情報を収集し、バインディング情報を含むボリュームマウントを接続することでアプリケーションコンテナーをアップグレードします。

Service Binding Operator は、次のアクションを実行します。

- 特定のサービスにバインドされたワークロードの **ServiceBinding** リソースを監視します。
- ボリュームマウントを使用して、バインディング情報をワークロードに適用します。

次の章では、自動および半自動のサービスバインディングアプローチとそのユースケースを説明します。**kubernetes-service-binding** エクステンションは、どちらのアプローチでも **ServiceBinding** リソースを生成します。半自動アプローチでは、ユーザーはターゲットサービスの設定を手動で指定する必要があります。自動アプローチでは、**ServiceBinding** リソースを生成する限定的なサービスセットに対する追加の設定は必要ありません。

関連情報

- ワークロードプロジェクション

1.3. 半自動サービスバインディング

サービスバインディングプロセスでは、まずユーザーが、特定のアプリケーションにバインドされる必要なサービスを指定します。この表現は、**kubernetes-service-binding** エクステンションが生成する **ServiceBinding** リソースに要約されます。**kubernetes-service-binding** エクステンションを使用すると、ユーザーは最小限の設定で **ServiceBinding** リソースを生成できるため、プロセス全体が簡略化されます。

次に、バインディングプロセスを実行する Service Binding Operator が **ServiceBinding** リソースから情報を読み取り、必要なファイルをコンテナーにマウントします。

- **ServiceBinding** リソースの例:

```
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
```

注記

- **quarkus-kubernetes-service-binding** エクステンションを使用すると、同じ情報をよりコンパクトに表現できます。以下に例を示します。

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

application.properties 内に以前の設定プロパティーを追加した後、**quarkus-kubernetes** と **quarkus-kubernetes-service-binding** エクステンションの組み合わせにより、**ServiceBinding** リソースが自動的に生成されます。

前述の **db-demo** プロパティー設定識別子には 2 つのロールがあり、次のアクションも実行します。

- **api-version** プロパティーと **api-version** プロパティーを相互に関連付けてグループ化します。
- カスタムリソースの **name** プロパティーを定義します。これは、必要に応じて後で編集できます。以下に例を示します。

```
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
```

関連情報

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

1.4. 半自動メソッドを使用した SERVICEBINDING カスタムリソースの生成

ServiceBinding リソースを半自動的に生成できます。以下の手順は、アプリケーションの設定とデプロイに使用する Operator のインストールを含む、OpenShift Container Platform のデプロイメントプロセスを示しています。

この手順では、Service Binding Operator と Crunchy Data の PostgreSQL Operator をインストールします。

重要

PostgreSQL Operator は、サードパーティーのコンポーネントです。PostgreSQL Operator のサポートポリシーと使用条件は、ソフトウェアベンダーである Crunchy Data にお問い合わせください。

次の手順には、PostgreSQL クラスターの作成、単純なアプリケーションのセットアップ、その後のプロビジョニングされたクラスターへのデプロイとバインドが含まれます。

前提条件

- OpenShift Container Platform 4.12 クラスターを作成している。
- OperatorHub からクラスター全体に Operator をインストールするために必要な、OperatorHub および OpenShift Container Platform への管理者権限を持っている。
- 以下がインストールされている。
  - OpenShift、**oc**、オーケストレーションツール
  - Maven と Java

手順

以下の手順では、HOME (~) ディレクトリーを保存先およびインストール先として使用します。

1. OpenShift Container Platform Web UI から Service Binding Operator をインストールする に記載された手順を使用して、Service Binding Operator バージョン 1.3.3 以降をインストールします。
   a. インストールを確認します。

   ```
   oc get csv -w
   ```

   Service Binding Operator の **phase** が **Succeeded** に設定されたことを確認し、次のステップに進みます。

2. Web コンソールまたは CLI を使用して、OperatorHub から Crunchy PostgreSQL Operator をインストールします。

   a. インストールを確認します。

   ```
   oc get csv -w
   ```

   Operator の **phase** が **Succeeded** に設定されたことを確認し、次のステップに進みます。

3. PostgreSQL クラスターを作成します。

   a. 新しい OpenShift Container Platform namespace を作成します。これは、後でクラスターを作成し、アプリケーションをデプロイするために使用します。ここで説明する手順では、この namespace を **demo** と呼びます。

   ```
   oc new-project demo
   ```

   b. 次のカスタムリソースを作成し、**pg-cluster.yml** として保存します。

   ```
   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
   ```

   注記

   この YAML は、Service Binding Operator Quickstart から再利用しています。

   c. 作成したカスタムリソースを適用します。

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

注記

このコマンドは、**pg-cluster.yml** ファイルが HOME ディレクトリーに保存されていることを前提としています。

d. Pod でインストールを確認します。

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

- インストールが完了して Pod が **READY** 状態になるまで待ちます。

4. PostgreSQL データベースにバインドする Quarkus アプリケーションを作成します。
ここで作成するアプリケーションは、Hibernate と Panache を使用して PostgreSQL に接続する基本的な **todo** アプリケーションです。

a. アプリケーションを生成します。

```
mvn com.redhat.quarkus.platform:quarkus-maven-plugin:3.8.5.SP1-redhat-00001:create
\
  -DplatformGroupId=com.redhat.quarkus.platform \
  -DplatformVersion=3.8.5.SP1-redhat-00001 \
  -DprojectGroupId=org.acme \
  -DprojectArtifactId=todo-example \
  -DclassName="org.acme.TodoResource" \
  -Dpath="/todo"
```

b. PostgreSQL への接続、すべての必要リソースの生成、およびアプリケーション用コンテナーイメージの構築に必要なエクステンションをすべて追加します。

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

c. 次の例に示すように、単純なエンティティーを作成します。

```
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. エンティティーを公開します。

```
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. **ServiceBinding** リソースを生成して、ターゲット PostgreSQL クラスターにバインドします。
   a. サービス座標を指定してバインディングを生成し、データソースを設定します。
      - apiVersion: **postgres-operator.crunchydata.com/v1beta1**
      - kind: **PostgresCluster**
      - name: **pg-cluster**
        次の例に示すとおり、これは接頭辞 **quarkus.kubernetes-service-binding.services.<id>.** を設定して行います。**id** は、プロパティーをグループ化するために使用され、任意の値を割り当てることができます。

```
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. いくつかの初期データを指定して **import.sql** スクリプトを作成します。

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

6. アプリケーション (**ServiceBinding** を含む) をデプロイし、クラスターに適用します。

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

   デプロイメントが完了するまで待ちます。

検証

1. デプロイメントを確認します。

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

2. インストールを確認します。

   a. ローカルで HTTP ポートにポート転送し、**/todo** エンドポイントにアクセスします。

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

   b. Web ブラウザーで、以下の URL を開きます。

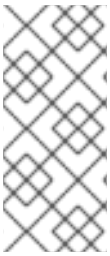
   ```
   http://localhost:8080/todo
   ```

関連情報

- 詳細は、Quick Start ガイドの Service Binding Operator セクションを参照してください。

1.5. 自動サービスバインディング

quarkus-kubernetes-service-binding エクステンションは、互換性のあるバインド可能な Operator によって提供される外部サービスへのアクセスを必要とするアプリケーションを検出した場合に、**ServiceBinding** リソースを自動的に生成できます。



注記

自動サービスバインディングは、限られた一連のサービスタイプセットに対してのみ生成できます。

定着した Kubernetes および Quarkus サービス用語に合わせて、この章ではこれらのサービスタイプを "kind" と呼びます。

表1.1 自動サービスバインディングをサポートする Operator

サービスバインディングのタイプ	Operator	API バージョン	種類
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

重要

- Red Hat build of Quarkus 3.8 の MongoDB Operator に対するサポートは、テクノロジープレビューとして提供され、クライアントのみに適用されます。
- Red Hat build of Quarkus 3.8 でサポートされている Panache エクステンションのリストは、Quarkus application configurator ページを参照してください。

1.5.1. 自動データソースバインディング

従来のデータベースでは、データソースが次のように設定されている場合に自動バインドが開始します。

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

上記が設定されていると、アプリケーション内に **quarkus-datasource**、**quarkus-jdbc-postgresql**、**quarkus-kubernetes**、**quarkus-kubernetes-service-binding** などのエクステンションが存在すれば、**postgresql** データベースタイプの **ServiceBinding** リソースが作成されます。

生成された **ServiceBinding** リソースは、使用している **postgresql** Operator と一致する Operator リソースの **apiVersion** プロパティーと **kind** プロパティーを使用して、サービスまたはリソースをアプリケーションにバインドします。

データベースサービスの名前を指定しない場合は、**db-kind** プロパティーの値がデフォルトの名前として使用されます。

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

データソースの名前を次のように指定します。

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

生成された **ServiceBinding** 内の **service** が、次のように表示されます。

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

同様に、**mysql** を使用する場合はデータソースの名前を次のように指定できます。

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

生成された **service** には以下が含まれます。

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

1.5.1.1. 自動サービスバインディングのカスタマイズ

自動サービスバインディング機能は、できる限り手動設定を排除するために開発されましたが、生成された **ServiceBinding** リソースを手動で変更することが必要になる場合もあります。

生成プロセスは、アプリケーションから展開した情報とサポートされている Operator の知識のみに依存し、クラスターにデプロイされたものが反映されない可能性があります。

生成されたリソースは、一般的なサービス kind でサポートされているバインド可能な Operator の知識と、潜在的な不一致を防ぐために開発された一連の規則のみに基づきます。

- ターゲットリソース名はデータソース名と一致しません。
- そのサービス kind のデフォルト Operator ではなく、特定の Operator を使用する必要があります。
- ユーザーがデフォルトまたは最新以外のバージョンを使用する必要がある場合は、バージョンの競合が発生します。

命名規則:

- ターゲットリソースの座標は、Operator タイプとサービス kind に応じて確立されます。
- デフォルトでは、ターゲットリソース名は **postgresql**、**mysql**、**mongo** などのサービス kind と一致します。
- 名前付きデータソースの場合は、そのデータソース名が使用されます。
- クライアントの名前は、名前付き **mongo** クライアントに使用されます。

例 1: 名前の不一致

名前の不一致を修正するために生成された **ServiceBinding** を変更する必要がある場合は、**quarkus.kubernetes-service-binding.services** プロパティーを使用してサービス名をサービスキーとして指定します。

通常、**service key** はサービスの名前 (例: データソースの名前、 **mongo** クライアントの名前など) です。この値が使用できない場合は、**postgresql**、**mysql**、**mongo** などのデータソースタイプが代わりに使用されます。

サービスタイプ間の名前の競合を避けるには、接頭辞として **service key** の前に特定のデータソースタイプ (**postgresql-<person>** など) を付けます。

次の例は、**PostgresCluster** リソースの **apiVersion** プロパティーをカスタマイズする方法を示しています。

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

例 2: データソースのカスタム名の適用

例 1 では、サービスキー **db-kind** (**postgresql**) が使用されました。この例では、データソースに名前が付けられているため、命名規則に従ってデータソース名 (**fruits-db**) が使用されます。

次の例は、名前付きデータソースの場合はデータソース名がターゲットリソースの名前として使用されることを示しています。

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

これは、次の設定と同じ効果があります。

```
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
```

関連情報

- 使用可能なプロパティーの詳細は、Kubernetes サービスバインディング仕様の workload projection を参照してください。

改訂日時: 2024-07-24