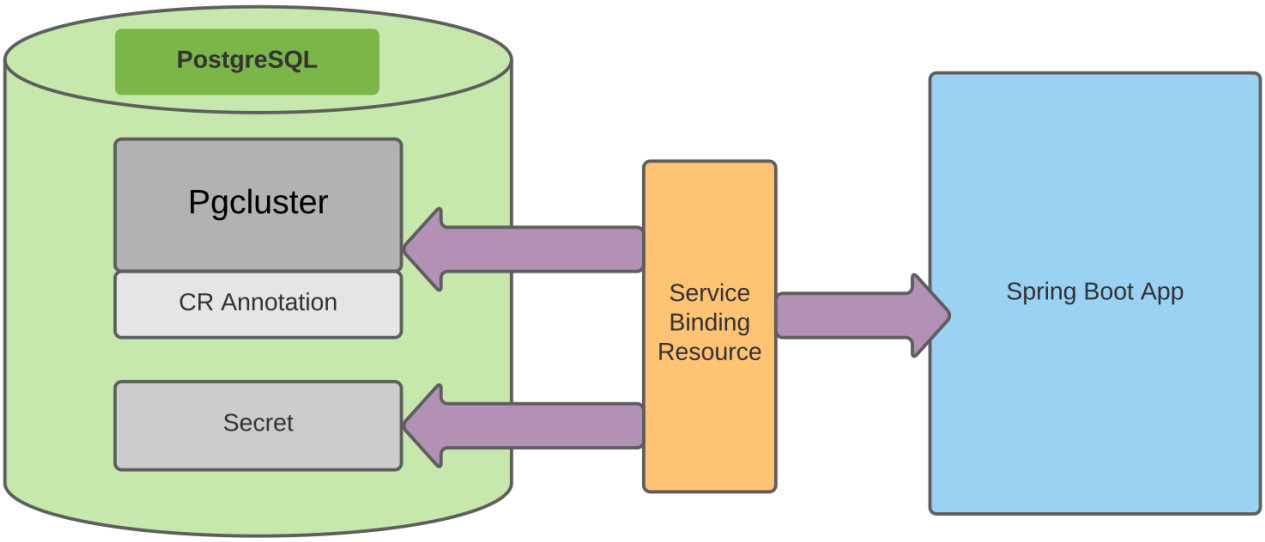
Quick start

In this quick start, you will see a sample application that you can deploy and use to play around. This will allow you to understand how Service Binding Operator can be used to simplify the connection between a service, like a database and the application.

For this sample, we will be using a PostgreSQL database and a simple application which will be the Spring Boot REST API server sample.

To illustrate what we are going to do, here is a visual representation of the application we are going to setup.



In this configuration, we will leverage the service binding operator, to collect the binding metadatas from the PostgreSQL database and inject them into the sample application.

Before starting, we invit you to refer to the Prerequisites section to make sure you have all the needed components configured on your K8s cluster.

The quick start will then consist into three main steps:

1. Create a PostgreSQL database instance
2. Deploy the application

3. Connect the application to the database with Service Binding

Prerequisites

In order to follow the quick start, you'll need the following tools installed and configured:

- Kubernetes cluster (**Note:** You can use minikube or kind, locally)
- kubectl
- Service Binding Operator

Create a PostgreSQL database instance

The application is going to use to a PostgreSQL database backend which can be setup using the Crunchy PostgreSQL operator from OperatorHub.io.

The installation of the operator doesn't create a database instance itself, so we need to create one.

1. To create a database instance, you need to create custom resource `Pgcluster` and that will trigger the operator reconciliation. For convenience, run this command to create `Pgcluster` custom resource:

```
kubectl apply -f https://servicebinding.dev/resourecs/pgcluster.yaml
```

In this `Pgcluster` custom resource, you might notice some annotations that we added, those will be necessary to enable binding later into the quick start guide:

```
service.binding/database: path={.spec.name}
service.binding/host: path={.spec.name}
service.binding/port: path={.spec.port}
```

The annotions points to the values of `database`, `host`, and `post` in resource attributes. For more details, refer Exposing binding data section.

2. Once the database is created, we need to ensure all the pods in `my-postgresql` namespace are running (it will take few minutes):

```
kubectl get pod -n my-postgresql
```

You should see output something like this:

```
NAME                                        READY   STATUS    RESTARTS
AGE
backrest-backup-hippo-9gtqf                 1/1     Running   0
13s
hippo-597dd64d66-4ztww                      1/1     Running   0
3m33s
hippo-backrest-shared-repo-66ddc6cf77-sjgqp   1/1     Running   0
4m27s
```

The database has been created and is empty at this stage. We now need to set its schema and we will also inject a sample data set, so we can play around with the application.

3. You can initialize the database with the schema and sample data using this command:

```
bash <(curl -s https://servicebinding.dev/resourecs/init-database.sh)>
```

We have now finished to configured the database for the application. We are ready to deploy the sample application and connect it to the database.

Deploying an application

In this section, we are going to deploy the application on our kubernetes cluster. For that, we will use a deployment configuration and do the configuration of our local environment to be able to test the application.

1. Deploy the `spring-petclinic-rest` app with this `Deployment` configuration:

```
kubectl apply -f https://servicebinding.dev/resourecs/app-
deployment.yaml
```

2. Let's now setup the port forwarding from the application port so we can access it from our local environment

```
kubectl port-forward --address 0.0.0.0 svc/spring-petclinic-rest 9966:80
-n my-postgresql
```

3. You should be able to open http://localhost:9966/petclinic and see a Swagger UI where you can play with the API.

At this stage, the application is not yet connected to the database. So, if you try to play around the APIs, you'll see errors returned by the application.

For example, if you try to access the list of all pets, you can see an error like this:

```
curl -X GET "http://localhost:9966/petclinic/api/pets" -H "accept: applica

{"className":"org.springframework.transaction.CannotCreateTransactionExcep
not open JPA EntityManager for transaction; nested exception is
org.hibernate.exception.JDBCConnectionException: Unable to acquire JDBC
Connection"}
```

Now, we are going to see how you can use Service Binding to easily connect the application to the database.

Connecting the application to the database

Suppose the Service Binding operator is not present. In that case, the application's admin needs to extract all the configuration details and create a Secret resource and expose it to the application through volume mount in Kubernetes. The steps would be something like this:

1. Identify the required values for connecting the application to the database
2. Locate the resources where the values are present
3. Take the values from different resources and create a Secret resource

4. Mount the Secret resource into the application
5. Depending on the application requirement the values should be exposed as env var or file.

In this quick start, we are going to leverage Service Binding as a way to easily and safely connect the application to the database service. In order to do that, we'll need to create a Service Binding ressource which will trigger the Service Binding Operator to inject the binding metadatas into the application.

1. Create the ServiceBinding custom resource to inject the bindings:

The `.spec` has two sections, the first one is a list of service resources (`.spec.services`) and the second one is the `.spec.application` . The services resources points to the database's service resources. How the values are exposed from service resources are explained Exposing binding data section. In the below ServiceBinding resource, there are two service resources, one custom resource and another Secret resource. This is required as the values required for database connectivity are living in these two resources. The application points to a `Deployment` or any similar resource with an embedded `PodSpec` . The mappings has extra mappings required for connectity.

```
apiVersion: binding.operators.coreos.com/v1alpha1
kind: ServiceBinding
metadata:
    name: spring-petclinic-rest
    namespace: my-postgresql
spec:
    services:
    - group: "crunchydata.com"
      version: v1
      kind: Pgcluster
      name: hippo
    - group: ""
      version: v1
      kind: Secret
      name: hippo-hippo-secret
    application:
      name: spring-petclinic-rest
      group: apps
      version: v1
      resource: deployments
    mappings:
    - name: type
```

```
      value: "postgresql"
```

For simplicity, you can copy/paste the following command to create the resource:

```
kubectl apply -f https://servicebinding.dev/resourecs/service-
binding.yaml
```

To learn more about creating service bindings, you can find more information on the following document..

By creating this `Service Binding` resource, we now have values from the database's binding metadata injected into the application container as files (that's the default behavior, but you can also inject environment variables if you prefer). If you check under `/bindings/spring-petclinic-rest` directory you'll see all the values from the secret resource injected there. In the above example, you'll find `username` and `password`. And the values pointed out through the annotation are also injected which includes `database`, `host`, and `port`. Finally, from the mappings, `type` is also injected. The application looks for `SERVICE_BINDING_ROOT` env var to find the location of `/bindings` directory. See the using injected bindings section about how the values can be used from the application.

2. Let's now check how the application is behaving and setup the port forwarding of the application port to access it from our local environment

```
kubectl port-forward --address 0.0.0.0 svc/spring-petclinic-rest 9966:80
-n my-postgresql
```

3. Open http://localhost:9966/petclinic, you should see a Swagger UI where you can play with the API.

If you try to access list of all pets, you can see the application is now connected to the database and see the sample data initially configured:

```
$ curl -X GET "http://localhost:9966/petclinic/api/pets" -H "accept:
application/json"
[{"id":1,"name":"Leo","birthDate":"2000/09/07","type":
{"id":1,"name":"cat"},
```

```
"owner":
{"id":1,"firstName":"George","lastName":"Franklin","address":"110...
```

Next Steps

In this sample, we setup a database and connected it to an application using the Service Binding operator to collect the connection metadata and expose them to the application.

By using service bindings, developers are able to more easily leverage the services available to them on a Kubernetes cluster. This method provides consistency accross different services and is repeatable for the developers. By remove the usual manual and error prone configuration, they benefit from a unified way to do application-to-service linkage.

You can continue to learn more about Service Binding by:

- Creating Service Bindings
- Using Injected Bindings
- Exposing binding data