Services Toolkit for VMware Tanzu Application Platform v0.7

Services Toolkit for VMware Tanzu Application Platform 0.7

You can find the most up-to-date technical documentation on the VMware website at:

https://docs.vmware.com/

VMware, Inc.
3401 Hillview Ave.
Palo Alto, CA 94304
www.vmware.com

Contents

About Services Toolkit

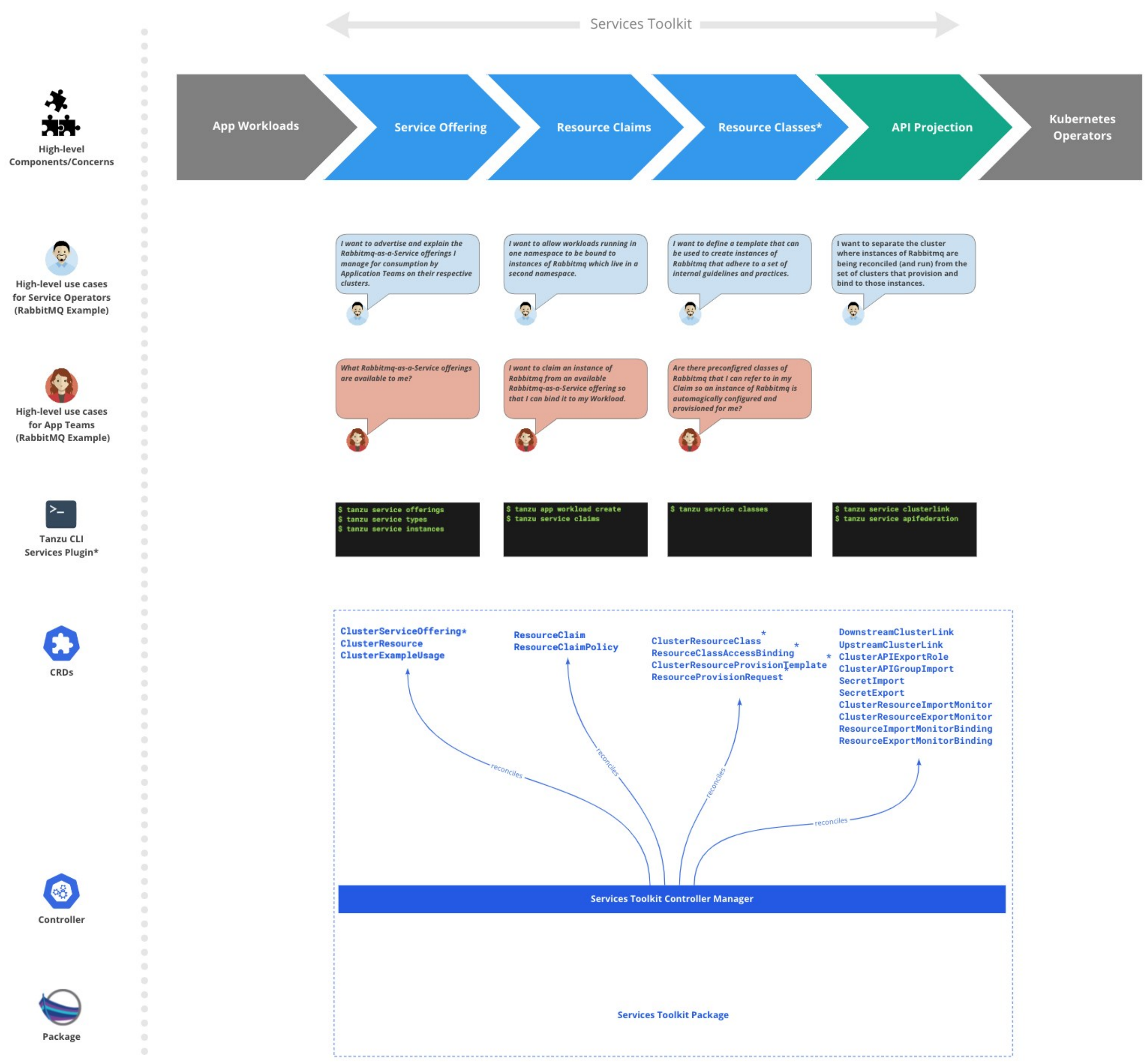
Services Toolkit is a collection of Kubernetes-native components supporting the discoverability, life-cycle management (CRUD), and connectivity of service resources (databases, message queues, DNS records, and so on) on Kubernetes.

The toolkit is currently comprised of the following components:

- Resource Claims
- Service Offering
- Service API Projection (experimental)
- Resource Replication (experimental)

Each component has value independent of the others, however the most powerful and valuable use cases can be unlocked by combining them together in unique and interesting ways. For a use case with examples of what can be done with the toolkit, see Getting Started.

For an example of how to consume AWS services with Services Toolkit, see either Consuming AWS RDS on Tanzu Application Platform (TAP) with AWS Controllers for Kubernetes (ACK) or Consuming AWS RDS on Tanzu Application Platform (TAP) with Crossplane.


Services Toolkit
High-level Components/Concerns
App Workloads
Service Offering
Resource Claims
Resource Classes*
API Projection
Kubernetes Operators
High-level use cases for Service Operators (RabbitMQ Example)
I want to advertise and explain the Rabbitmq-as-a-Service offerings I manage for consumption by Application Teams on their respective clusters.
I want to allow workloads running in one namespace to be bound to instances of Rabbitmq which live in a second namespace.
I want to define a template that can be used to create instances of Rabbitmq that adhere to a set of internal guidelines and practices.
I want to separate the cluster where instances of Rabbitmq are being reconciled (and run) from the set of clusters that provision and bind to those instances.
High-level use cases for App Teams (RabbitMQ Example)
What Rabbitmq-as-a-Service offerings are available to me?
I want to claim an instance of Rabbitmq from an available Rabbitmq-as-a-Service offering so that I can bind it to my Workload.
Are there preconfigred classes of Rabbitmq that I can refer to in my Claim so an instance of Rabbitmq is automagically configured and provisioned for me?
Tanzu CLI Services Plugin*
CRDs
ClusterServiceOffering*
ClusterResource
ClusterExampleUsage
ResourceClaim
ResourceClaimPolicy
ClusterResourceClass*
ResourceClassAccessBinding*
ClusterResourceProvisionTemplate*
ResourceProvisionRequest*
DownstreamClusterLink
UpstreamClusterLink
ClusterAPIExportRole
ClusterAPIGroupImport
SecretImport
SecretExport
ClusterResourceImportMonitor
ClusterResourceExportMonitor
ResourceImportMonitorBinding
ResourceExportMonitorBinding
reconciles
reconciles
reconciles
reconciles
Controller
Services Toolkit Controller Manager
Package
Services Toolkit Package

```
$ tanzu service offerings
$ tanzu service types
$ tanzu service instances
```

```
$ tanzu app workload create
$ tanzu service claims
```

```
$ tanzu service classes
```

```
$ tanzu service clusterlink
$ tanzu service apifederation
```

* indicates an item is on the roadmap and no concrete design is available yet. An early prototype or proposal might exist.

Motivation

Application teams need supporting service resources (databases, message queues, DNS records, and so on) to develop and run their applications. They do not want the burden of running these services themselves, so many organizations provide ticketing systems that allow application teams to manually make requests for new service resources to be created and managed for them. This process often takes weeks.

In the cloud, application teams have self-service access to create new managed resources that you can provision with API calls, such as RDS. Services Toolkit aims to provide a set of modular tools that you can use to provide a similar self-service experience to that of the cloud for service resources running on Tanzu.

Component Overview

Here is a brief overview of the components comprising Services Toolkit.

Resource claims

Resource claims enable application teams to express which service resources their applications require without having to know the intricacies of the service resource fulfilling the request. This replaces the traditional ticketing system previously mentioned with a model of application teams claiming resources and service operators providing resources to be claimed. This provides a self-service experience for the developer, but gives the service operators ultimate control of the service resources.

This also means application teams can request a service resource without having to know the exact name or namespace of the pre-provisioned service resource. Instead they express requirements using more meaningful metadata. For example, type, protocol, provider, and version. The claim is then fulfilled against an existing service resource using rules chosen by the service operator. This enables application teams to focus on their application and its dependencies.

To learn more about resource claims, see Resource claims.

Service Offering

To discover service resources and understand how to use them, application operators need access to a rich set of metadata that describes the semantics and management capabilities of the corresponding Service Resource Lifecycle APIs.

The fundamental building blocks of Service Resource Lifecycle APIs are aggregated APIs or CRDs, and these already define some metadata. However, this only consists of Kubernetes-level API descriptions, such as name and field.

Although this metadata is useful, application operators require more holistic information that covers details such as service-level management capabilities, QoS guarantees, and relationships between different resource types the API exposes. Application operators also require other information that aids discovery by application operators and higher-level tooling aimed at that role, such as keywords, icons, and so on.

Some metadata surfaced by service description and offering relate not only to the Service Resource Lifecycle API itself, but also to the specifics of the underlying infrastructure, such as the number and the topology of worker nodes in the Service Cluster, or the particular CSI and CNI implementations configured for the cluster.

For example, a service resource that is relevant to MySQL cannot claim high-availability for the provisioned databases if the service cluster in which the individual MySQL pods run consists of only a single worker node.

Because of this, the service operator is deemed responsible for ensuring that the correct level of accurate metadata is specified for a service resource. Service description and offering enables the association of metadata with service resources and surfacing it to application operators. The service operator can provide this metadata, and service authors can provide infrastructure-agnostic metadata, such as data that describes the relationships between different API resource types.

To learn more about service offering, see Service offering.

Service API Projection and Resource Replication (experimental)

VMware recommends that customers separate application and service infrastructure, which is done in their production environments. Benefits of this segmentation of infrastructure include:

- **Dedicated cluster requirements for workload or service clusters:** For example, service clusters might need access to SSDs.
- **Different cluster life cycle management:** Upgrades to service clusters can occur more cautiously.
- **Unique compliance requirements:** Data might have different compliance needs because it is stored on a service cluster.
- **Separation of permissions and access:** Application teams can only access the clusters where their applications are running.

One way to address these needs in a Kubernetes multicluster world is to split clusters into application workload clusters and service clusters, and then allow application teams to consume service resource APIs from their application workload cluster, with reconciliation of resources occurring on services clusters.

To learn more about service API projection and resource replication, see Service API projection and service resource replication.

Release notes

v0.7.1

Release Date: July 12, 2022

- Services Toolkit now integrates with Amazon RDS using the ACK Operator. See Consuming AWS RDS on Tanzu Application Platform with AWS Controllers for Kubernetes (ACK).
- Services Toolkit now integrates with Amazon RDS by using Crossplane. See Consuming AWS RDS on Tanzu Application Platform with Crossplane.
- New `ClusterInstanceClass` supports service instance abstraction. It is available using `tanzu service classes list` in `v0.3.0` of the Services plug-in for Tanzu CLI.
- You can now use the `InstanceQuery` API to discover claimable resources. It is available using `tanzu service claimable list --class CLASS` in `v0.3.0` of the Services plug-in for Tanzu CLI.
- ResourceClaims no longer mutate service resources with an annotation to mark a claimed resource. Instead it uses Kubernetes Leases.
- ResourceClaims no longer require the `update` permission when adding new service resources to Tanzu Application Platform.
- ResourceClaims now aggregate on ClusterRoles for service resources with the standard `servicebinding.io/controller: "true"` label from the Service Binding specification for Kubernetes This label is recommended over the existing `resourceclaims.services.apps.tanzu.vmware.com/controller: "true"` label, although the old label continues to work as expected.
- Performance enhancements to ResourceClaim controller tracker.
- All Services Toolkit components now conform to Tanzu Application Platform logging standards.
- Deprecation warning: `tanzu service types list` and `tanzu service instances list` commands are now deprecated. These commands are hidden from help text but remain functional if invoked. VMware intends to support these commands for either two additional minor releases (v0.6.0 of the CLI plug-in) or after one year (2023-07-12), whichever comes later. VMware recommends using `tanzu service class` and `tanzu service claimable` commands in place of `tanzu service type` and `tanzu service instance` from now on.

Bug Fixes

- ResourceClaims no longer overwrite existing secrets on cross namespace claims.
- Fix ResourceClaims incorrectly logging resource requests as part of tracking.

- ResourceClaims `.status.ClaimedResourceRef.Namespace` is now set for same namespace claims.

v0.6.0

Release Date: April 12, 2022

- Introduced default aggregating ClusterRoles for Tanzu Application Platform's App Editors, App Viewers, and App Operators.
- The `ResourceClaim` and `ResourceClaimPolicy` CRD category `resourceclaims` was removed to avoid clashes with the `ResourceClaim` resource plural.
- Fixed kubectl table output of `ResourceClaimPolicy`.
- All Services Toolkit pods now adhere to Restricted Pod Security Standards.
- Services plug-in for Tanzu CLI v0.2.0 includes the following changes:
  - Allows the management of `ResourceClaims` using `tanzu service claims <list/get/create/delete>`.
  - Alpha Warnings are now output to `stderr` instead of `stdout`.

v0.5.1

Release Date: March 3, 2022

- Fixed a race condition issue that might lead to a failure of the `services-toolkit` controller manager when a new `ResourceClaim` is being created whilst another is being deleted.
- Fixed an issue that caused `kapp-controller` to unnecessarily reconcile continuously.
- Services plug-in for Tanzu CLI at v0.1.2 now supports interactions with GCP clusters.

v0.5.0

Release Date: January 11, 2022

- Resource Claims now support cross namespace claiming by using `ResourceClaimPolicy` objects.
- Resource Claims are now exclusive. Multiple `ResourceClaim` objects can not claim a single service resource.
- Services Toolkit, specifically Resource Claims, now depends on at least `v0.5.0` of carvel-secretgen-controller in GitHub.
- Do not block claim deletion when it can not find GVR.

Breaking changes

- Rename `ClusterServiceResource` to `ClusterResource`
- Move `ClusterResource`, `ClusterExampleUsage` and `ResourceClaim` to `services.apps.tanzu.vmware.com` APIGroup

- Move `DownstreamClusterLink`, `UpstreamClusterLink`, `APIExportRoleBinding`, `APIResourceImport` and `ClusterAPIGroupImport` to `projection.apiresources.multicluster.x-tanzu.vmware.com` APIGroup
- Move `ClusterResourceExportMonitor`, `ClusterResourceImportMonitor`, `ResourceExportMonitorBinding`, `ResourceImportMonitorBinding`, `SecretExport` and `SecretImport` to `replication.apiresources.multicluster.x-tanzu.vmware.com` APIGroup
- Add the label prefix `replication.apiresources.multicluster.x-tanzu.vmware.com` for the `monitored-resource-*` labels of `ClusterResourceExportMonitor` and `ClusterResourceImportMonitor`
- Rename the Resource Claims finalizer from `claim.services.apps.tanzu.vmware.com/finalizer` to `resourceclaims.services.apps.tanzu.vmware.com/finalizer`. Existing `ResourceClaims` must be updated to remove the old finalizer to be deleted.
- Rename the Resource Claims aggregation `ClusterRole` label from `services.apps.tanzu.vmware.com/aggregate-to-resource-claims: "true"` to `resourceclaims.services.apps.tanzu.vmware.com/controller: "true"`. Existing aggregated roles must be updated to have the new label.
- Edit all deployment resources naming to use `services-toolkit` rather than the outdated `scp-toolkit`.

Getting started

The quickest and easiest way to get started with Services Toolkit is to experience it as part of Tanzu Application Platform. For more information about the main use cases, tools and APIs powered by the toolkit, see About consuming services on Tanzu Application Platform.

In addition, a number of additional use cases are available:

- Consuming AWS RDS on Tanzu Application Platform (TAP) with AWS Controllers for Kubernetes (ACK)
- Consuming AWS RDS on Tanzu Application Platform (TAP) with Crossplane
- Consuming Azure FlexibleServer PostgreSQL on Tanzu Application Platform (TAP) with Azure Server Operator v2
- Direct Secret References
- Dedicated Service Clusters

Install

Services Toolkit is packaged and distributed by using the carvel set of tools.

The Services Toolkit carvel package is currently published to the Tanzu Application Platform package repository.

There are two options for installation:

- To install it as part of a wider Tanzu Application Platform installation, see Installing Tanzu Application Platform.
- To install it as an individual package on its own, see Install Services Toolkit.

Consuming Services on Tanzu Application Platform

The best way to get started and to learn about Services Toolkit is to follow the getting started guides published for Tanzu Application Platform. Two guides are available, one pertaining to the roles of the Service Operator and Application Operator, and the other, complimentary guide pertaining to the role of the Application Developer. These guides are linked below.

- Set up services for consumption by developers
- Consume services on Tanzu Application Platform

Uninstall

To uninstall Services Toolkit run:

```
tanzu package installed delete services-toolkit
```

Use Cases and Walkthroughs

This section of the documentation covers common use cases and walkthroughs to help you learn about the capabilities and usage of Services Toolkit. Please refer to and select a use case of interest from the table of contents.

Direct Secret References

This use case leverages direct references to Kubernetes `Secret` resources to enable developers to connect their application workloads to almost any backing service, including backing services that:

- are running external to Tanzu Application Platform
- do not adhere to the ProvisionedService of the Service Binding Specification for Kubernetes in GitHub.

The following example demonstrates a procedure to bind a new application on Tanzu Application Platform to an existing PostgreSQL database that exists in Azure.

Depending on your Kubernetes distribution and the backing Service you are hoping to connect to your Tanzu Application Platform workloads, there could be extra work to set up networking between the workload and the service endpoint and to obtain the credentials for the backing service. This example assumes the credentials are available and networking has been set up.

1. Create a Kubernetes secret resource similar to the following example:

   ```
   # external-azure-db-binding-compatible.yaml
   ---
   apiVersion: v1
   kind: Secret
   metadata:
     name: external-azure-db-binding-compatible
   type: Opaque
   stringData:
     type: postgresql
     provider: azure
     host: EXAMPLE.DATABASE.AZURE.COM
     port: "5432"
     database: "EXAMPLE-DB-NAME"
     username: "USER@EXAMPLE"
     password: "PASSWORD"
   ```

   Kubernetes secret resources must abide by the Well-known Secret Entries specifications in GitHub. If you are planning to bind this secret to a Spring-based application workload and want to take advantage of the auto-wiring feature, this secret must also contain the properties required by Spring Cloud Bindings in GitHub.

2. Apply the YAML file by running:

```
kubectl apply -f external-azure-db-binding-compatible.yaml
```

3. Grant sufficient RBAC permissions to Services Toolkit to be able to read the secrets specified by the class:

```
# stk-secret-reader.yaml
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: stk-secret-reader
  labels:
    servicebinding.io/controller: "true"
rules:
- apiGroups:
  - ""
  resources:
  - secrets
  verbs:
  - get
  - list
  - watch
```

4. Apply your changes by running:

```
kubectl apply -f stk-secret-reader.yaml
```

5. Create a claim for the newly created secret by running:

```
tanzu service claim create external-azure-db-claim \
  --resource-name external-azure-db-binding-compatible \
  --resource-kind Secret \
  --resource-api-version v1
```

6. Obtain the claim reference of the claim by running:

```
tanzu service claim list -o wide
```

Expect to see the following output:

```
NAME                     READY  REASON  CLAIM REF
external-azure-db-claim  True           services.apps.tanzu.vmware.com/v1alpha1
:ResourceClaim:external-azure-db-claim
```

7. Create an application workload by running a command similar to the following example:

Example:

```
tanzu apps workload create WORKLOAD-NAME \
  --git-repo https://github.com/sample-accelerators/spring-petclinic \
  --git-branch main \
  --git-tag tap-1.2 \
  --type web \
  --label app.kubernetes.io/part-of=spring-petclinic \
  --annotation autoscaling.knative.dev/minScale=1 \
  --env SPRING_PROFILES_ACTIVE=postgres \
  --service-ref db=REFERENCE
```

Where:

- `WORKLOAD-NAME` is the name of the Application Workload. For example, `pet-clinic`.
- `REFERENCE` is the value of the `CLAIM REF` for the newly created claim in the output of the last step.

Dedicated Service Clusters (using experimental Projection and Replication APIs)

Caution: This use case leverages experimental APIs. Do not use it in a production environment.

This use case is currently not supported on Kubernetes v1.24 or later.

This use case leverages the experimental API Projection and Resource Replication APIs to separate application workloads and service instances onto separate Kubernetes clusters. There are several reasons for it:

- **Dedicated cluster requirements for workload or service clusters:** Service clusters, for example, might need access to more powerful SSDs.
- **Different cluster life cycle management:** Upgrades to service clusters can occur more cautiously.
- **Unique compliance requirements:** Data is stored on a service cluster, which might have different compliance needs.
- **Separation of permissions and access:** Application teams can only access the clusters where their applications are running.

The benefits of implementing this use case include:

- The experience for application developers and application operators working on their Tanzu Application Platform cluster is unaltered.
- All complexity in the setup and management of backing infrastructure is abstracted away from application developers, which gives them more time to focus on developing their applications.

For information about network requirements and possible topology setups, see Topology.

Prerequisites

Meet the following prerequisites before completing this use case walkthrough:

- You have access to a cluster with Tanzu Application Platform installed, henceforth called the application workload cluster.
- You have access to a second, separate cluster with the Services Toolkit package installed, henceforth called the service cluster.
- You downloaded and installed the `tanzu` CLI and the corresponding plug-ins.
- You downloaded and installed the experimental `kubectl-scp` plug-in. For instructions, see

Install the kubectl-scp plug-in.

- You set up the `default` namespace on the application workload cluster as your developer namespace to use installed packages. For more information, see Set up developer namespaces to use installed packages.
- The application workload cluster can pull source code from GitHub.
- The service cluster can pull the images required by the RabbitMQ Cluster Kubernetes Operator.
- The service cluster can create LoadBalancer services.
- If you have previously installed the RabbitMQ cluster operator to the application workload cluster as part of Getting started with Tanzu Application Platform, uninstall it from that cluster. This is necessary because of a limitation of the experimental API Projection APIs. To delete the operator, run:

```
kapp delete -a rmq-operator -y
```

Walkthrough

Follow these steps to bind an application to a service instance running on a different Kubernetes cluster:

1. As the service operator, link the workload cluster and service cluster together by using the `kubectl scp` plug-in. To do so, run:

   ```
   kubectl scp link --workload-kubeconfig-context=WORKLOAD-CONTEXT --service-kubec
   onfig-context=SERVICE-CONTEXT
   ```

   Where `WORKLOAD-CONTEXT` is your workload context and `SERVICE-CONTEXT` is your service context.

2. Install the RabbitMQ Kubernetes operator in the services cluster by running:

   ```
   kapp -y deploy --app rmq-operator \
    --file https://raw.githubusercontent.com/rabbitmq/cluster-operator/lb-binding/
   hack/deploy.yml \
    --kubeconfig-context SERVICE-CONTEXT
   ```

   Where `SERVICE-CONTEXT` is your service context.

   This operator is installed in the service cluster, but `RabbitmqCluster` service instance life cycles (CRUD) can still be managed from the workload cluster. Use the exact `deploy.yml` specified in the command because this RabbitMQ operator deployment includes specific changes to enable cross-cluster service binding.

3. Verify that you installed the operator by running:

   ```
   kubectl --context SERVICE-CONTEXT get crds rabbitmqclusters.rabbitmq.com
   ```

   Where `SERVICE-CONTEXT` is your service context.

   The `rabbitmq.com/v1beta1` API group is available in the service cluster. The following steps federate the `rabbitmq.com/v1beta1` in the workload cluster. This occurs in two parts, projection and replication.

- Projection applies to custom API groups.
- Replication applies to core Kubernetes resources, such as secrets.

4. Create a `service-instance` namespace in both clusters. API projection occurs between clusters by using namespaces with the same name and that are said to have a quality of namespace sameness.

   For example:

   ```
   kubectl --context WORKLOAD-CONTEXT create namespace service-instances
   kubectl --context SERVICE-CONTEXT create namespace service-instances
   ```

   Where `WORKLOAD-CONTEXT` is your workload context and `SERVICE-CONTEXT` is your service context.

5. Use the `kubectl-scp` plug-in to federate by running:

   ```
   kubectl scp federate \
   --workload-kubeconfig-context=WORKLOAD-CONTEXT \
   --service-kubeconfig-context=SERVICE-CONTEXT \
   --namespace=service-instances \
   --api-group=rabbitmq.com \
   --api-version=v1beta1 \
   --api-resource=rabbitmqclusters
   ```

   Where `WORKLOAD-CONTEXT` is your workload context and `SERVICE-CONTEXT` is your service context.

6. After federation, verify the `rabbitmq.com/v1beta1` API is also available in the workload cluster by running:

   ```
   kubectl --context WORKLOAD-CONTEXT api-resources
   ```

   Where `WORKLOAD-CONTEXT` is your workload context

7. Advertise that the RabbitmqCluster API is available to developers by applying the following YAML to your workload cluster. Ensure the Tanzu CLI is configured to target the workload cluster for the rest of the steps.

   ```
   ---
   apiVersion: services.apps.tanzu.vmware.com/v1alpha1
   kind: ClusterInstanceClass
   metadata:
    name: rabbitmq
   spec:
    description:
      short: It's a RabbitMQ cluster!
    pool:
      kind: RabbitmqCluster
      group: rabbitmq.com
   ```

8. Discover the new service and provision an instance from the workload cluster by running:

```
tanzu services classes list
```

The following output appears:

```
tanzu services classes list

NAME      DESCRIPTION
rabbitmq  It's a RabbitMQ cluster!
```

9. Provision a service instance on the Tanzu Application Platform cluster.

   For example:

```
# rabbitmq-cluster.yaml
---
apiVersion: rabbitmq.com/v1beta1
kind: RabbitmqCluster
metadata:
 name: projected-rmq
spec:
 service:
   type: LoadBalancer
```

10. Apply the YAML file by running:

```
kubectl --context WORKLOAD-CONTEXT -n service-instances apply -f rabbitmq-clust
er.yaml
```

   Where `WORKLOAD-CONTEXT` is your workload context

11. Confirm that the RabbitmqCluster resource reconciles successfully from the workload cluster by running:

```
kubectl --context WORKLOAD-CONTEXT -n service-instances get -f rabbitmq-cluster
.yaml
```

   Where `WORKLOAD-CONTEXT` is your workload context

12. Verify that RabbitMQ pods are running in the service cluster, but not in the workload cluster, by running:

```
kubectl --context WORKLOAD-CONTEXT -n service-instances get pods
kubectl --context SERVICE-CONTEXT -n service-instances get pods
```

   Where `WORKLOAD-CONTEXT` is your workload context and `SERVICE-CONTEXT` is your service context.

13. Enable cross-namespace claims by creating a `ResourceClaimPolicy` on your workload cluster:

```
# rabbitmq-cluster-policy.yaml
---
apiVersion: services.apps.tanzu.vmware.com/v1alpha1
kind: ResourceClaimPolicy
metadata:
  name: rabbitmq-cluster-policy
  namespace: service-instances
spec:
  consumingNamespaces:
  - default
  subject:
    group: rabbitmq.com
    kind: RabbitmqCluster
```

14. Apply the YAML file by running:

    ```
    kubectl --context WORKLOAD-CONTEXT apply -f rabbitmq-cluster-policy.yaml
    ```

    Where `WORKLOAD-CONTEXT` is your workload context

15. Create a claim for the projected service instance by running:

    ```
    tanzu service claim create projected-rmq-claim \
      --resource-name projected-rmq \
      --resource-kind RabbitmqCluster \
      --resource-api-version rabbitmq.com/v1beta1 \
      --resource-namespace service-instances \
      --namespace default
    ```

16. Create the application workload by running:

    ```
    tanzu apps workload create multi-cluster-binding-sample \
      --namespace default \
      --git-repo https://github.com/sample-accelerators/rabbitmq-sample \
      --git-branch main \
      --git-tag 0.0.1 \
      --type web \
      --label app.kubernetes.io/part-of=rabbitmq-sample \
      --annotation autoscaling.knative.dev/minScale=1 \
      --service-ref "rmq=services.apps.tanzu.vmware.com/v1alpha1:ResourceClaim:proj
    ected-rmq-claim"
    ```

17. Get the `web-app` URL by running:

    ```
    tanzu apps workload get multi-cluster-binding-sample -n default
    ```

18. Visit the URL and refresh the page to confirm the app is running by viewing the new message IDs.

Consuming Cloud Services (AWS, Azure and GCP) on Tanzu Application Platform

This section of the documentation covers integrations of various Cloud Service Providers (AWS, Azure and GCP) into Tanzu Application Platform.

Consuming AWS RDS on Tanzu Application Platform

This section of the documentation covers integrations of AWS RDS into Tanzu Application Platform. Documentation is provided for both an integration using AWS Controllers for Kubernetes (ACK), as well as an integration using Crossplane.

Consuming AWS RDS on Tanzu Application Platform with AWS Controllers for Kubernetes (ACK)

This topic describes how to use Services Toolkit to allow Tanzu Application Platform workloads to consume AWS RDS PostgreSQL databases.

This topic makes use of AWS Controllers for Kubernetes (ACK) to manage RDS instances in AWS. As such, it is an alternative approach to using Crossplane to achieve the same outcomes.

Prerequisites

- Prerequisites
- Configure your AWS RDS environment

Create service instances that are compatible with Tanzu Application Platform

Installing the ACK service controller for RDS makes available new Kubernetes APIs for interacting with RDS resources from within the Tanzu Application Platform cluster.

```
$ kubectl api-resources --api-group rds.services.k8s.aws

NAME                       SHORTNAMES   APIVERSION                       NAMESPACED   K
IND
dbclusterparametergroups                rds.services.k8s.aws/v1alpha1    true         D
BClusterParameterGroup
dbclusters                              rds.services.k8s.aws/v1alpha1    true         D
BCluster
dbinstances                             rds.services.k8s.aws/v1alpha1    true         D
BInstance
dbparametergroups                       rds.services.k8s.aws/v1alpha1    true         D
BParameterGroup
dbsubnetgroups                          rds.services.k8s.aws/v1alpha1    true         D
BSubnetGroup
globalclusters                          rds.services.k8s.aws/v1alpha1    true         G
lobalCluster
```

`DBInstance` is of most interest here because this is the primary API for creating RDS databases. However, there are two important obstacles with this API when considering compatibility with Tanzu Application Platform.

Obstacle 1: `DBInstance` does not adhere to the binding specification

`DBInstance` does not adhere to the Service Binding Specification for Kubernetes. Tanzu Application Platform uses this specification as a contract for ensuring compatibility between different parts of the system. Given that `DBInstance` does not adhere to the specification it means that, by default, it is not possible to claim and bind application workloads to `DBInstance` resources.

Obstacle 2: Creating a `DBInstance` resource on its own is not

sufficient

Creating a `DBInstance` resource on its own might not always be enough to create a working, usable instance that can be connected to and utilized.

For example, `DBInstance` defines the field `.spec.masterUserPassword`, which must refer to a secret containing credentials for the instance. As such, the secret resource can be considered a dependent resource of `DBInstance`. Without both of these resources, it is not possible to properly configure the RDS instance as wanted. In many cases, a group of related resources must be created to create something usable.

Solutions

Tanzu Application Platform v1.2 and later enables solutions for both these obstacles.

For example, consider the first obstacle where `DBInstance` does not adhere to the Kubernetes binding specification. One solution is for the authors of the RDS ACK service controller to update the `DBInstance` API to make it adhere to the binding specification. However, this requires code changes to the operator itself, and the authors of the operator might choose not to prioritize it.

Fortunately, there is an alternative solution that doesn’t require any code changes to the operator itself while still enabling claiming and binding to RDS instances from within a Tanzu Application Platform cluster.

This solution uses the `SecretTemplate` API provided by Carvel’s secretgen-controller. This API can be used to create binding specification-conforming secrets by identifying and collecting information that resources from the RDS APIs provide.

Next, consider the second obstacle where multiple resources must be created to produce a usable RDS database. One solution to this obstacle is to just document all the resources that must be created to produce something that can be used. This solution is laborious, error-prone, and is generally a poor developer experience.

Fortunately, there is an alternative solution that abstracts away the complexities of creating instances that are known to work well with application workloads.

This solution uses the `ClusterInstanceClass` API provided by Services Toolkit. Instance classes allow for logical service instances to be presented to Application Operators, allowing them to discover, reason about, and, most importantly, claim service instances that they can then bind to their application workloads.

The rest of this topic describes how both these solutions can come together to form an end-to-end integration for RDS services on Tanzu Application Platform.

Create an RDS service instance

This section describes how to create an RDS service instance in Tanzu Application Platform by using a ready-made reference Carvel Package. This step is typically performed by the Service Operator role. Follow the steps in Creating an RDS service instance by using a Carvel Package.

Alternatively, if you want to author your own reference package and want to learn about the underlying APIs and how they come together to produce a useable service instance for Tanzu Application Platform, you can achieve the same outcome by using the more advanced Creating an RDS service instance manually.

After you complete either of these steps and have a running RDS service instance, return here to continue with the rest of the use case.

Create a service instance class for RDS

Now that you know how to create RDS service instances it's time to learn how to make those instances discoverable to Application Operators. This step is typically performed by the Service Operator role.

You can use Services Toolkit's `ClusterInstanceClass` API to create a service instance class to represent RDS service instances within the cluster. The existence of such classes make these logical service instances discoverable to Application Operators. This allows them to create Resource Claims for such instances and to then bind them to application workloads.

Create the following Kubernetes resource on your EKS cluster:

```
# clusterinstanceclass.yaml
---
apiVersion: services.apps.tanzu.vmware.com/v1alpha1
kind: ClusterInstanceClass
metadata:
  name: aws-rds-postgres
spec:
  description:
    short: AWS RDS instances with a postgresql engine
  pool:
    kind: Secret
    labelSelector:
      matchLabels:
        services.apps.tanzu.vmware.com/class: rds-postgres
```

Apply it by running:

```
kubectl apply -f clusterinstanceclass.yaml
```

In this example, the class states that claimable instances of RDS PostgreSQL are represented by `Secret` objects with the label `services.apps.tanzu.vmware.com/class` set to `rds-postgres`. A `Secret` with this label was created in the earlier step when you provisioned an RDS service instance.

Although this example uses `services.apps.tanzu.vmware.com/class`, there is no special meaning to that key. The Service Operator role can choose arbitrary label names and values. They might also decide to select multiple labels or combine a label selector with a field selector when defining the `ClusterInstanceClass`.

After creating a `ClusterInstanceClass`, you must grant sufficient RBAC permissions to enable Services Toolkit to read the resources that match the pool definition of the instance class. For this example, create the following aggregated `ClusterRole` in your EKS cluster:

```
# stk-secret-reader.yaml
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: stk-secret-reader
  labels:
    servicebinding.io/controller: "true"
rules:
- apiGroups:
  - ""
  resources:
  - secrets
  verbs:
  - get
  - list
  - watch
```

Apply it by running:

```
kubectl apply -f stk-secret-reader.yaml
```

If you want to claim resources across namespace boundaries, you must create a corresponding `ResourceClaimPolicy`. For example, if the provisioned RDS PostgreSQL instances exist in the namespace `service-instances`, and you want to allow Application Operators to claim them for workloads residing in the `default` namespace, create the following `ResourceClaimPolicy`:

```
# resourceclaimpolicy.yaml
apiVersion: services.apps.tanzu.vmware.com/v1alpha1
kind: ResourceClaimPolicy
metadata:
  name: default-can-claim-rds-postgres
  namespace: service-instances
spec:
  subject:
    kind: Secret
    group: ""
    selector:
      matchLabels:
        services.apps.tanzu.vmware.com/class: rds-postgres
  consumingNamespaces: [ "default" ]
```

Apply it by running:

```
kubectl apply -f resourceclaimpolicy.yaml
```

Discover, Claim, and Bind to an RDS

Creating the `ClusterInstanceClass` and the corresponding RBAC informs Application Operators that RDS is available to use with their application workloads on Tanzu Application Platform. In this section you learn how to discover, claim, and bind to the RDS service instance previously created. The Application Operator is typically the role that discovers and claims service instances. The Application Developer is typically the role that handles binding.

To discover what service instances are available to them, Application Operators can run

```
tanzu services classes list

  NAME                      DESCRIPTION
  aws-rds-postgres      AWS RDS instances with a postgresql engine
```

Here you can see information about the `ClusterInstanceClass` created in the earlier step. Each `ClusterInstanceClass` created is added to the list of classes returned here.

The next step is to claim an instance of the wanted class, but to do that, Application Operators must first discover the list of currently claimable instances for the class. Many variables, including namespace boundaries, claim policies, and the exclusivity of claims, affect the capacity to claim instances. Therefore Services Toolkit provides the CLI command `tanzu service claimable list` to help inform Application Operators of the instances that can enable successful claims. Example:

```
tanzu services claimable list --class aws-rds-postgres

  NAME          NAMESPACE  API KIND  API GROUP/VERSION
  rds-bindable  default    Secret    v1
```

Because of the setup performed as part of Creating a claimable class for RDS instances, the secrets created from the `SecretTemplate` as part of Create an RDS service instance now appear as claimable to the Application Operator. From here on it is simply a case of creating a resource claim for the instance and then binding the claim to an application workload.

Create a claim for the newly created secret by running:

```
tanzu service claim create ack-rds-claim \
  --resource-name rds-bindable \
  --resource-kind Secret \
  --resource-api-version v1
```

Obtain the claim reference of the claim by running:

```
tanzu service claim list -o wide
```

Verify that the output is similar to the following:

```
NAME                        READY  REASON  CLAIM REF
ack-rds-claim               True           services.apps.tanzu.vmware.com/v1alpha1:Resourc
eClaim:ack-rds-claim
```

Create an application workload that consumes the claimed RDS PostgreSQL Database. Example:

```
tanzu apps workload create my-workload \
  --git-repo https://github.com/sample-accelerators/spring-petclinic \
  --git-branch main \
  --git-tag tap-1.2 \
  --type web \
  --label app.kubernetes.io/part-of=spring-petclinic \
  --annotation autoscaling.knative.dev/minScale=1 \
  --env SPRING_PROFILES_ACTIVE=postgres \
  --service-ref db=services.apps.tanzu.vmware.com/v1alpha1:ResourceClaim:ack-rds-claim
```

`--service-ref` is set to the claim reference obtained previously.

Your application workload now starts up and connects automatically to the RDS service instance. You can verify this by visiting the app in the browser and, for example, creating a new owner through the UI.

Prerequisites

Meet these prerequisites to follow along with Consuming AWS RDS on Tanzu Application Platform (TAP) with AWS Controllers for Kubernetes (ACK).

1. Install the AWS CLI or gain access to the Amazon Cloud Console
2. Gain the AWS privileges required to configure the IAM permissions and identity used by the ACK service controller for RDS
3. Create an Amazon EKS cluster. The quickest and simplest way to create an EKS cluster is to use eksctl, as in this example:

   ```
   eksctl create cluster -r YOUR-REGION -m 6 -M 8 -n YOUR-CLUSTER-NAME
   ```

4. Tanzu Application Platform v1.2.0 or later and Cluster Essentials v1.2.0 or later have to be installed on the Kubernetes cluster.

   Note: To check if you have an appropriate version, run the following:

   ```
   kubectl api-resources | grep secrettemplate
   ```

   This command returns the `SecretTemplate` API. If it does not for you, verify that Cluster Essentials for VMware Tanzu v1.2.0 or later is installed.

5. Install the ACK service controller for RDS and configure it in the cluster. It is recommended to install the latest stable version of the Operator (v0.0.25 is known to work with this specific use case). For instructions, see Install an ACK Controller. This entails installing the RDS ACK service controller, which entails updating some of the environment variables used throughout the official documentation. In particular, note the following changes:

   - Set the `SERVICE` environment variable to `rds` by running:

     ```
     export SERVICE=rds
     ```

   - Set the `AWS_REGION` environment variable to the AWS region where the RDS instances is created by running:

     ```
     export AWS_REGION=us-east-1
     ```

6. After the operator is installed, configure IAM permissions. Set the following environment variables accordingly:

   - Set the `SERVICE` environment variable to `rds` by running:

     ```
     export SERVICE=rds
     ```

   - Set the `EKS_CLUSTER_NAME` environment variable to the name of your EKS cluster by running:

     ```
     export EKS_CLUSTER_NAME=<YOUR_CLUSTER_NAME>
     ```

- Set the `AWS_REGION` environment variable to the AWS region where the RDS instances is created by running:

  ```
  export AWS_REGION=us-east-1
  ```

Configuring the AWS RDS environment

This topic tells you how to configure your AWS environment for Consuming AWS RDS on Tanzu Application Platform (TAP) with AWS Controllers for Kubernetes (ACK).

Prerequisites

Meet the prerequisites for consuming AWS RDS on Tanzu Application Platform with AWS Controllers for Kubernetes (ACK), including using `eksctl` to create an EKS cluster. This procedure entails reusing the resources created when you created the cluster.

You can still create separate VPCs, subnets and security groups if you want. Ensure that these are configured such that Tanzu Application Platform workloads on EKS can discover and connect to RDS instances.

Configure the AWS RDS environment

To configure the AWS RDS environment:

1. Use the AWS cloud console to determine the VPC ID of the EKS cluster, or run this command:

   ```
   aws eks describe-cluster --name YOUR-CLUSTER-NAME --region YOUR-REGION | \
     jq -r .cluster.resourcesVpcConfig.vpcId
   ```

   RDS instances must be configured with a subnet group consisting of two or more subnets. The subnets within the subnet group must adhere to the following rules:

   - The subnets must be in different availability zones, such as us-west-1a and us-west-1b.
   - All subnets must either be public or private, which the `MapPublicIpOnLaunch` value reveals.

2. Discover existing subnets within your VPC by using the AWS Cloud console or by running:

   ```
   aws ec2 describe-subnets --filters "Name=vpc-id,Values=YOUR-VPC-ID" --region YO
   UR-REGION | \
     jq -r '.Subnets[] | select(.MapPublicIpOnLaunch == false) | .SubnetId'
   ```

3. Create the following Kubernetes resource on your EKS cluster by using the subnet IDs output:

   ```
   # dbsubnetgroup.yaml
   ---
   apiVersion: rds.services.k8s.aws/v1alpha1
   kind: DBSubnetGroup
   metadata:
   ```

```
  name: DB-SUBNET-GROUP-NAME
  namespace: ack-system
spec:
  name: DB-SUBNET-GROUP-NAME
  description: rds-subnet-group
  subnetIDs:
  - SUBNET-ID-1
  - SUBNET-ID-2
  - SUBNET-ID-3
```

Where `DB-SUBNET-GROUP-NAME`, `SUBNET-ID-1`, `SUBNET-ID-2`, and `SUBNET-ID-3` are your own values.

4. Run

```
kubectl apply -f dbsubnetgroup.yaml
```

5. Confirm that you created `DBSubnetGroup` by running:

```
kubectl get DBSubnetGroup -n ack-system DB-SUBNET-GROUP-NAME -o yaml
```

6. Identify a suitable security group to use for the RDS instance that allows workloads running on the Tanzu Application Platform cluster to establish a connection. Do so by searching for a suitable security group within the AWS cloud console, or by running the following command, which identifies the `Communication between all nodes in the cluster` security group:

```
aws ec2 describe-security-groups --filters "Name=vpc-id,Values=YOUR-VPC-ID" --r
egion YOUR-REGION | \
  jq -r '.SecurityGroups[] | select(.Description == "Communication between all
nodes in the cluster").GroupId'
```

7. Record `DB-SUBNET-GROUP-NAME` and the security group ID output from the previous command. You need both when creating RDS instances as part of this use case.

Creating AWS RDS Instances manually using kubectl (experimental)

This topic is for users who want to understand the underlying APIs involved in making a bindable service instance using `DBInstance` and `SecretTemplate` resources. For a simpler user experience, see Creating an RDS service instance through a Carvel Package.

Prerequisite

Meet the prerequisites in Consuming AWS RDS on Tanzu Application Platform (TAP) with AWS Controllers for Kubernetes (ACK) and keep the following information to hand:

- `DB-SUBNET-GROUP-NAME` - the name of the `DBSubnetGroup` resource previously created
- `SECURITY-GROUP-ID` - the security group ID to use for this RDS instance

Create an RDS service instance by using kubectl

Follow these procedures to create an RDS service instance by using kubectl.

Create the `DBInstance` resource

This example uses secret-gen to generate a `Password` for the `DBInstance`. You can also provide an explicit password through a `Secret`.

1. Create Kubernetes resources on your EKS cluster by using the following example. This YAML creates the `DBInstance` resource in the `default` namespace.

```
# dbinstance.yaml
---
apiVersion: secretgen.k14s.io/v1alpha1
kind: Password
metadata:
 name: rds-psql-password
 namespace: default
spec:
 length: 64
 secretTemplate:
   type: Opaque
   stringData:
     password: $(value) # do not edit, this will auto generate a password.
---
apiVersion: rds.services.k8s.aws/v1alpha1
kind: DBInstance
metadata:
 name: rds-psql-1
 namespace: default
spec:
 allocatedStorage: 20
 dbInstanceClass: db.t3.micro
 dbInstanceIdentifier: rds-psql-1
 dbName: postgres
 engine: postgres
 engineVersion: "14.1"
 masterUsername: adminUser
 masterUserPassword:
   namespace: default
   name: rds-psql-password
   key: password
 vpcSecurityGroupIDs:
 - SECURITY-GROUP-ID                     # modify value
 dbSubnetGroupName: DB-SUBNET-GROUP-NAME # modify value

 # note: due to an issue in the RDS ACK controller, it is recommended to explic
itly set the
 # following optional spec fields.
 # default values for the optional fields are provided below.
 # https://github.com/aws-controllers-k8s/community/issues/1346
 autoMinorVersionUpgrade: true
 backupRetentionPeriod: 1
 copyTagsToSnapshot: false
 deletionProtection: false
 licenseModel: postgresql-license
 monitoringInterval: 0
 multiAZ: false
 preferredBackupWindow: 23:00-23:30
 preferredMaintenanceWindow: wed:23:34-thu:00:04
 publiclyAccessible: false
 storageEncrypted: false
 storageType: gp2
```

Where:

- `DB-SUBNET-GROUP-NAME` is the name of the `DBSubnetGroup` resource previously created
- `SECURITY-GROUP-ID` is the security group ID to use for this RDS instance

2. Run:

```
kubectl apply -f dbinstance.yaml
```

3. Verify the creation status of the `DBInstance` by inspecting the conditions in the Kubernetes API. To do so, run:

```
kubectl get DBInstance rds-psql-1 -o yaml -n default
```

Create a Binding Specification Compatible Secret

As mentioned in Creating service instances that are compatible with Tanzu Application Platform, for Tanzu Application Platform workloads to be able to claim and bind to services such as RDS, a resource compatible with Service Binding Specification must exist in the cluster.

This can take the form of either a `ProvisionedService` or a Kubernetes `Secret` with some known keys. Both are defined in the specification.

The RDS `DBInstance` you created does not adhere to `ProvisionedService` and does not create a spec-compatible secret. So, you must create one using the resources you have available.

In this topic, you create a Kubernetes secret in the necessary format using the secret-gen tooling. You do so by using the `SecretTemplate` API to extract values from the `DBInstance` resource and populate a new spec-compatible secret with the values.

Create a ServiceAccount for secret templating

As part of using the `SecretTemplate` API, a Kubernetes `ServiceAccount` must be provided. The `ServiceAccount` is used for reading the `DBInstance` resource and the `Secret` created from the `Password` resource.

1. Create the following Kubernetes resources on your EKS cluster:

```
# secrettemplate-sa.yaml
---
apiVersion: v1
kind: ServiceAccount
metadata:
 name: rds-resources-reader
 namespace: default
---
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
 name: rds-resources-reading
```

```
  namespace: default
rules:
- apiGroups:
  - ""
  resources:
  - secrets
  verbs:
  - get
  - list
  - watch
  resourceNames:
  - rds-psql-password
- apiGroups:
  - rds.services.k8s.aws
  resources:
  - dbinstances
  verbs:
  - get
  - list
  - watch
  resourceNames:
  - rds-psql-1
---
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: rds-resources-reader-to-read
  namespace: default
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: Role
  name: rds-resources-reading
subjects:
  - kind: ServiceAccount
  name: rds-resources-reader
  namespace: default
```

2. Run:

```
kubectl apply -f secrettemplate-sa.yaml
```

Create a SecretTemplate

In combination with the `ServiceAccount` you created, a `SecretTemplate` can be used to declaratively create a secret that is compatible with the service binding specification.

The `.spec.inputResources` fields list the resources with information needed to create the secret. The `.spec.template` field defines how that information is interpolated as a secret.

To specify fields on an input resource, you can use JSONPath syntax that is very similar to kubectl syntax. The only difference is the delimiters, which are `\$(` and `)` instead of `{` and `}`.

For example, `$(.rds.status.endpoint.address)` produces the host address of an RDS instance if the input resource is an ACK controller `DBInstance` resource.

This syntax can currently be used in the following fields of the `SecretTemplate` API:

- `.spec.inputResource[].ref.name` for dynamically loading input resources of the APIs of input resources previously in the list

- `.spec.template` values for taking values from the input resources and interpolating them into the secret you create

In this case, you directly reference the `DBInstance` resource and then dynamically load the secret containing the password from its specification.

You then create a `Secret` conforming to the Postgres auto-configuration for Spring Cloud Bindings to enable a fully automated, end-to-end binding experience for application workloads on Tanzu Application Platform.

1. Create the following Kubernetes resources on your EKS cluster:

```
# bindable-rds-secrettemplate.yaml
---
apiVersion: secretgen.carvel.dev/v1alpha1
kind: SecretTemplate
metadata:
 name: rds-bindable
 namespace: default
spec:
 serviceAccountName: rds-resources-reader
 inputResources:
 - name: rds
   ref:
     apiVersion: rds.services.k8s.aws/v1alpha1
     kind: DBInstance
     name: rds-psql-1
 - name: creds
   ref:
     apiVersion: v1
     kind: Secret
     name: "$(.rds.spec.masterUserPassword.name)"
template:
 metadata:
   labels:
     app.kubernetes.io/component: rds-postgres
     app.kubernetes.io/instance: "$(.rds.metadata.name)"
     services.apps.tanzu.vmware.com/class: rds-postgres
 type: postgresql
 stringData:
   type: postgresql
   port: "$(.rds.status.endpoint.port)"
   database: "$(.rds.spec.dbName)"
   host: "$(.rds.status.endpoint.address)"
   username: "$(.rds.spec.masterUsername)"
 data:
   password: "$(.creds.data.password)"
```

2. Run:

```
kubectl apply -f bindable-rds-secrettemplate.yaml
```

Verify

Find the name of the secret produced by reading the status of `SecretTemplate`. To do so, run:

```
kubectl get secrettemplate -n default rds-bindable -o jsonpath="{.status.secret.name}"
```

Delete an RDS service instance

Delete an RDS service instance by running:

```
kubectl delete DBInstance rds-psql-1 -n default
kubectl delete SecretTemplate rds-bindable -n default
kubectl delete Password rds-psql-password -n default
kubectl delete ServiceAccount rds-resources-reader -n default
kubectl delete RoleBinding rds-resources-reader-to-read -n default
kubectl delete Role rds-resources-reading -n default
```

Summary and Next Steps

You learned how to use Carvel's `SecretTemplate` API to construct a secret that is compatible with the binding specification to create an AWS RDS service instance.

Now that you have this available in the cluster, you can learn how to make use of it by continuing where you left off in Consuming AWS RDS on Tanzu Application Platform (TAP) with AWS Controllers for Kubernetes (ACK).

Creating AWS RDS instances by using a Carvel package (experimental)

This topic describes how to create, update, and delete RDS service instances by using a Carvel package. For a more detailed and low-level alternative procedure, see Creating AWS RDS Instances manually by using kubectl.

Prerequisite

Meet the prerequisites in Consuming AWS RDS on Tanzu Application Platform (TAP) with AWS Controllers for Kubernetes (ACK).

The package repository and service instance package bundles for this topic are in the tanzu-application-platform-reference-packages GitHub repository.

Create an RDS service instance using a Carvel package

Follow the steps in the following procedures.

Add a reference package repository to the in the cluster

To add a reference package repository to the in the cluster:

1. Use the Tanzu CLI to add the new Service Reference packages repository by running:

   ```
   tanzu package repository add tap-service-reference-packages --url ghcr.io/vmwar
   e-tanzu/tanzu-application-platform-reference-packages/tap-service-reference-pac
   kage-repo:0.0.1 -n tanzu-package-repo-global
   ```

2. Use the following example to create a `ServiceAccount` that you use to provision `PackageInstall` resources. The namespace of this `ServiceAccount` must match the namespace of the `tanzu package install` command in the next step.

```
# rds-service-account-installer.yaml
---
apiVersion: v1
kind: ServiceAccount
metadata:
 name: rds-install
 namespace: default
---
kind: Role
apiVersion: rbac.authorization.k8s.io/v1
metadata:
 name: rds-install
 namespace: default
rules:
- apiGroups: ["*"] # TODO: use more fine-grained RBAC permissions
 resources: ["*"]
 verbs: ["*"]
---
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
 name: rds-install
 namespace: default
subjects:
- kind: ServiceAccount
 name: rds-install
roleRef:
 apiGroup: rbac.authorization.k8s.io
 kind: Role
 name: rds-install
```

3. Run:

```
kubectl apply -f rds-service-account-installer.yaml
```

Create an RDS service instance through the Tanzu CLI

To create an RDS service instance through the Tanzu CLI:

1. Create the following Kubernetes resources on your EKS cluster:

```
# RDS-INSTANCE-NAME-values.yaml
---
name: "RDS-INSTANCE-NAME"
namespace: "default"
dbSubnetGroupName: "DB-SUBNET-GROUP-NAME"
vpcSecurityGroupIDs:
- "SECURITY-GROUP-ID"
```

Where:

- `RDS-INSTANCE-NAME` is a chosen name for the RDS instance to create

    - `DB-SUBNET-GROUP-NAME` is the name of the `DBSubnetGroup` resource previously created
    - `SECURITY-GROUP-ID` is the security group ID to use for this RDS instance

2. Use the Tanzu CLI to install an instance of the reference service instance Package by running:

   ```
   tanzu package install RDS-INSTANCE-NAME --package-name psql.aws.references.services.apps.tanzu.vmware.com --version 0.0.1-alpha --service-account-name rds-install -f RDS-INSTANCE-NAME-values.yaml -n default
   ```

You can install the `psql.aws.references.services.apps.tanzu.vmware.com` package multiple times to produce multiple RDS service instances.

To do so, prepare a separate `RDS-INSTANCE-NAME-values.yaml` file and then install the package with a different name and the earlier mentioned separate data values file for each RDS service instance.

Verify

To verify:

1. Verify the creation status for the RDS instance by inspecting the conditions in the Kubernetes API. To do so, run:

   ```
   kubectl get DBInstance RDS-INSTANCE-NAME -n default -o yaml
   ```

2. Wait for up to 20 minutes.
3. Find the binding-compliant secret that `PackageInstall` produced by running:

   ```
   kubectl get secrettemplate RDS-INSTANCE-NAME-bindable -n default -o jsonpath="{.status.secret.name}"
   ```

Delete an RDS service instance

Delete the RDS service instance by running:

```
tanzu package installed delete RDS-INSTANCE-NAME -n default
```

Summary

You learned how to use Carvel's `Package` and `PackageInstall` APIs to create an RDS service instance. To learn more about the pieces that comprise this service instance package, see Create an RDS service instance manually.

Now that you have an RDS service instance in the cluster, you can learn how to make use of it by continuing from where you left off in Consuming AWS RDS on Tanzu Application Platform (TAP) with AWS Controllers for Kubernetes (ACK).

Consuming AWS RDS on Tanzu Application Platform with Crossplane

Overview

This topic describes how to use Services Toolkit to enable Tanzu Application Platform workloads to consume AWS RDS PostgreSQL databases.

This topic makes use of Crossplane to manage RDS instances in AWS. It is an alternative approach to using the AWS Controllers for Kubernetes (ACK) to achieve the same outcomes.

Prerequisites

Meet these prerequisites:

- Create a Kubernetes cluster that supports running both Tanzu Application Platform and Crossplane
- Install Tanzu Application Platform on the Kubernetes cluster
- Gain access to an AWS account with permissions to manage RDS database instances
- Install AWS CLI
- Configure a named profile for an AWS account that has permissions to manage RDS databases.

Install Crossplane

Run the following commands to install Crossplane to your existing Kubernetes cluster:

```
kubectl create namespace crossplane-system

helm repo add crossplane-stable https://charts.crossplane.io/stable
helm repo update

helm install crossplane --namespace crossplane-system crossplane-stable/crossplane \
  --set 'args={--enable-external-secret-stores}'
```

Note: For the latest steps for installing Crossplane, see the Crossplane documentation. As of Crossplane 1.9.0, the feature flag `--enable-external-secret-stores` is still needed.

For this topic, you do not need to install the Crossplane CLI or any additional configuration package.

Install AWS Provider for Crossplane

To install the AWS Provider for Crossplane:

1. Run:

```
kubectl apply -f -<<EOF
---
apiVersion: pkg.crossplane.io/v1
kind: Provider
metadata:
 name: provider-aws
spec:
 package: xpkg.upbound.io/crossplane/provider-aws:v0.24.1
EOF
```

2. After installing the provider, you see a new `rdsinstances.database.aws.crossplane.io` API resource available in your Kubernetes cluster. See the health of the installed provider by running:

```
kubectl get provider.pkg.crossplane.io provider-aws
```

Configure AWS provider

To configure an AWS provider:

1. Create a new key file:

```
AWS_PROFILE=default && echo -e "[default]\naws_access_key_id = $(aws configure
get aws_access_key_id --profile $AWS_PROFILE)\naws_secret_access_key = $(aws co
nfigure get aws_secret_access_key --profile $AWS_PROFILE)\naws_session_token =
$(aws configure get aws_session_token --profile $AWS_PROFILE)" > creds.conf
```

   If your AWS profile is not named `default`, change `AWS_PROFILE` to the actual name.

2. Verify that you a created a new key file by reading the content of the newly created `creds.conf` file.

3. Create a new secret from the key file by running:

```
kubectl create secret generic aws-provider-creds -n crossplane-system --from-fi
le=creds=./creds.conf
```

4. Delete the key file by running:

```
rm -f creds.conf
```

5. Configure the AWS provider to use the newly created secret by running:

```
kubectl apply -f -<<EOF
---
apiVersion: aws.crossplane.io/v1beta1
kind: ProviderConfig
metadata:
 name: default
spec:
 credentials:
   source: Secret
   secretRef:
     namespace: crossplane-system
     name: aws-provider-creds
     key: creds
EOF
```

Define composite resource types

Now that the AWS provider for Crossplane is installed and configured, you can create a new `CompositeResourceDefinition` (XRD) and corresponding `Composition` representing individual instances of RDS PostgreSQL by following the steps in this section. For more information about these concepts see the Crossplane composition documentation.

Instead of creating your own custom XRD and composition, you can also install an existing Crossplane configuration package for AWS that includes pre-configured XRDs and compositions for RDS.

The primary reason for choosing to create a new XRD and composition is to ensure the connection secrets for newly provisioned RDS PostgreSQL instances support the Service Binding Specification for Kubernetes and automatic Spring Boot configuration using Spring Cloud Bindings.

1. Create a new XRD by running:

   ```
   kubectl apply -f -<<EOF
   ---
   apiVersion: apiextensions.crossplane.io/v1
   kind: CompositeResourceDefinition
   metadata:
    name: xpostgresqlinstances.bindable.database.example.org
   spec:
    claimNames:
      kind: PostgreSQLInstance
      plural: postgresqlinstances
    connectionSecretKeys:
    - type
    - provider
    - host
    - port
    - database
    - username
    - password
    group: bindable.database.example.org
    names:
      kind: XPostgreSQLInstance
      plural: xpostgresqlinstances
    versions:
    - name: v1alpha1
      referenceable: true
      schema:
        openAPIV3Schema:
          properties:
            spec:
              properties:
                parameters:
                  properties:
                    storageGB:
                      type: integer
                  required:
                  - storageGB
                  type: object
              required:
              - parameters
              type: object
          type: object
      served: true
   EOF
   ```

   After the newly created XRD is reconciled there are two new API resources available in your Kubernetes cluster, `xpostgresqlinstances.bindable.database.example.org` and postgresqlinstances.bindable.database.example.org.

2. Create a corresponding composition by running:

```
kubectl apply -f -<<EOF
---
apiVersion: apiextensions.crossplane.io/v1
kind: Composition
metadata:
 labels:
   provider: "aws"
   vpc: "default"
 name: xpostgresqlinstances.bindable.aws.database.example.org
spec:
 compositeTypeRef:
   apiVersion: bindable.database.example.org/v1alpha1
   kind: XPostgreSQLInstance
 publishConnectionDetailsWithStoreConfigRef:
   name: default
 resources:
 - base:
     apiVersion: database.aws.crossplane.io/v1beta1
     kind: RDSInstance
     spec:
       forProvider:
         dbInstanceClass: db.t2.micro
         engine: postgres
         dbName: postgres
         engineVersion: "12"
         masterUsername: masteruser
         publiclyAccessible: true
         region: us-east-1
         skipFinalSnapshotBeforeDeletion: true
       writeConnectionSecretToRef:
         namespace: crossplane-system
   connectionDetails:
   - name: type
     value: postgresql
   - name: provider
     value: aws
   - name: database
     value: postgres
   - fromConnectionSecretKey: username
   - fromConnectionSecretKey: password
   - name: host
     fromConnectionSecretKey: endpoint
   - fromConnectionSecretKey: port
   name: rdsinstance
   patches:
   - fromFieldPath: metadata.uid
     toFieldPath: spec.writeConnectionSecretToRef.name
     transforms:
     - string:
         fmt: '%s-postgresql'
         type: Format
       type: string
     type: FromCompositeFieldPath
   - fromFieldPath: spec.parameters.storageGB
     toFieldPath: spec.forProvider.allocatedStorage
```

```
      type: FromCompositeFieldPath
EOF
```

This composition ensures that all RDS PostgreSQL instances are placed in the `us-east-1` region and use the default VPC for the respective AWS account.

3. Take one of these actions:

   - Connect to those instances from outside the default VPC by assigning an appropriate inbound rule for TCP on port `5432` to the security group of that VPC.
   - Define a composition that creates a separate VPC for each RDS PostgreSQL instance and automatically configures inbound rules. See this example.

Create an instance class

To make instances of a service easy for application operators to discover and claim, the service operator persona creates a `ClusterInstanceClass`. In this example, the class states that claimable instances of RDS PostgreSQL are represented by secret objects of type `connection.crossplane.io/v1alpha1` with the label `services.apps.tanzu.vmware.com/class` set to `rds-postgres`:

```
kubectl apply -f -<<EOF
---
apiVersion: services.apps.tanzu.vmware.com/v1alpha1
kind: ClusterInstanceClass
metadata:
  name: rds-postgres
spec:
  description:
    short: AWS RDS Postgresql database instances
  pool:
    kind: Secret
    labelSelector:
      matchLabels:
        services.apps.tanzu.vmware.com/class: rds-postgres
    fieldSelector: type=connection.crossplane.io/v1alpha1
EOF
```

In addition, grant RBAC permissions to Services Toolkit to enable reading the secrets specified by the class.

```
kubectl apply -f -<<EOF
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: stk-secret-reader
  labels:
    servicebinding.io/controller: "true"
rules:
- apiGroups:
  - ""
  resources:
  - secrets
  verbs:
```

```
  - get
  - list
  - watch
EOF
```

Provision RDS PostgreSQL instance

As the service operator persona, you now provision an instance of RDS PostgreSQL using the `postgresqlinstances.bindable.database.example.org` API managed by the XRD you previously created.

`.spec.publishConnectionDetailsTo` provides Crossplane with the name and a label for the secret that stores the connection details for the newly created database. You can see that the label specified here matches the drop-down menu value defined in the `ClusterInstanceClass` you created earlier.

1. Create an RDS database instance in your AWS account by running:

   ```
   kubectl apply -f -<<EOF
   ---
   apiVersion: bindable.database.example.org/v1alpha1
   kind: PostgreSQLInstance
   metadata:
    name: rds-postgres-db
    namespace: default
   spec:
    parameters:
      storageGB: 20
    compositionSelector:
      matchLabels:
        provider: aws
        vpc: default
    publishConnectionDetailsTo:
      name: rds-postgres-db
      metadata:
        labels:
          services.apps.tanzu.vmware.com/class: rds-postgres
   EOF
   ```

2. Verify that you created the RDS database instance by running:

   ```
   aws rds describe-db-instances --region us-east-1 --profile default
   ```

   Expect the status of the newly created `PostgreSQLInstance` resource to be `READY=True`. This might take a few minutes. You can wait for this by running:

   ```
   kubectl wait --for=condition=Ready=true postgresqlinstances.bindable.database.e
   xample.org rds-postgres-db
   ```

As soon as the RDS PostgreSQL instance is ready, it is claimable by the application operator persona as shown in the next section.

Claim the RDS PostgreSQL instance and connect to it from

the Tanzu Application Platform workload

Thanks to the `ClusterInstanceClass` created in the earlier section, application operators can now use the Tanzu CLI to discover and claim secrets representing RDS PostgreSQL instances.

1. Show available classes of service instances by running:

   ```
   tanzu service classes list

   NAME            DESCRIPTION
   rds-postgres    AWS RDS Postgresql database instances
   ```

2. Show claimable instances belonging to the RDS PostgreSQL class by running:

   ```
   tanzu services claimable list --class rds-postgres

   NAME                NAMESPACE  API KIND  API GROUP/VERSION
   rds-postgres-db     default    Secret    v1
   ```

3. Create a claim for the discovered secret by running:

   ```
   tanzu service claim create rds-claim \
   --resource-name rds-postgres-db \
   --resource-kind Secret \
   --resource-api-version v1
   ```

4. Obtain the claim reference by running:

   ```
   tanzu service claim list -o wide
   ```

   Expect to see the following output:

   ```
   NAME                         READY  REASON   CLAIM REF
   rds-claim                    True            services.apps.tanzu.vmware.com/v1alpha1
   :ResourceClaim:rds-claim
   ```

5. Create an application workload that consumes the claimed RDS PostgreSQL database. In this example, `--service-ref` is set to the claim reference obtained earlier.

   ```
   tanzu apps workload create my-workload \
   --git-repo https://github.com/sample-accelerators/spring-petclinic \
   --git-branch main \
   --git-tag tap-1.2 \
   --type web \
   --label app.kubernetes.io/part-of=spring-petclinic \
   --annotation autoscaling.knative.dev/minScale=1 \
   --env SPRING_PROFILES_ACTIVE=postgres \
   --service-ref db=services.apps.tanzu.vmware.com/v1alpha1:ResourceClaim:rds-clai
   m
   ```

Consuming Azure Flexible Server Tanzu Application Platform

This section of the documentation covers integrations of Azure Flexible Server into Tanzu Application Platform. Documentation is provided for both an integration using Azure Service

Operator (ASO), as well as an integration using Crossplane.

Consuming Azure Flexible Server for PostgreSQL on Tanzu Application Platform with Azure Service Operator (ASO)

This topic describes using Services Toolkit to allow Tanzu Application Platform workloads to consume Azure Flexible Server PostgreSQL. This particular topic makes use of Azure Service Operator v2 to manage PostgreSQL instances in Azure.

Important: This use case is not currently compatible with air-gapped Tanzu Application Platform installations.

Prerequisites

Meet these prerequisites

Create service instances that are compatible with Tanzu Application Platform

To create an Azure PostgreSQL service instance for Tanzu Application Platform to consume, you can use a ready-made, reference Carvel package. The Service Operator typically performs this step. Follow the steps in Creating an Azure PostgreSQL service instance using a Carvel package.

```
$ kubectl api-resources --api-group=dbforpostgresql.azure.com
```

```
NAME                                SHORTNAMES   APIVERSION
  NAMESPACED   KIND
flexibleservers                                  dbforpostgresql.azure.com/v1beta20210601
  true         FlexibleServer
flexibleserversconfigurations                    dbforpostgresql.azure.com/v1beta20210601
  true         FlexibleServersConfiguration
flexibleserversdatabases                         dbforpostgresql.azure.com/v1beta20210601
  true         FlexibleServersDatabase
flexibleserversfirewallrules                     dbforpostgresql.azure.com/v1beta20210601
  true         FlexibleServersFirewallRule
```

There is also the Resource Group, which is in another API group.

```
$ kubectl api-resources --api-group=resources.azure.com
```

```
NAME              SHORTNAMES   APIVERSION                           NAMESPACED   KIND
resourcegroups                 resources.azure.com/v1beta20200601   true         Resour
ceGroup
```

To create an Azure PostgreSQL service instance for Tanzu Application Platform to consume, you can use a ready-made, reference Carvel package. The Service Operator typically performs this step. Follow the steps in Creating an Azure PostgreSQL service instance using a Carvel package.

Alternatively, if you are interested in authoring your own reference package and want to learn about the underlying APIs and how they come together to produce a useable service instance for Tanzu Application Platform, you can achieve the same outcome by using the more advanced Creating an

Azure PostgreSQL service instance manually topic.

After creating a running Azure PostgreSQL service instance, return here to continue the use case.

Create a service instance class for PSQL

After creating Flexible Server service instances, you must make it possible for application operators to discover them. The service operator role typically performs this step.

You can use Services Toolkit's `ClusterInstanceClass` API to create a service instance class that represents psql service instances within the cluster. The existence of such classes enables application operators to discover logical service instances. This, in turn, enables application operators to create Resource Claims for such instances and to then bind them to application workloads.

Create the following Kubernetes resource on your AKS cluster by running:

```
cat <<EOF | kubectl apply -f -
---
apiVersion: services.apps.tanzu.vmware.com/v1alpha1
kind: ClusterInstanceClass
metadata:
  name: azure-postgres
spec:
  description:
    short: Azure Flexible Server instances with a postgresql engine
  pool:
    kind: Secret
    labelSelector:
      matchLabels:
        services.apps.tanzu.vmware.com/class: azure-postgres
EOF
```

In this particular example, the class represents claimable instances of PostgreSQL by a `Secret` object with the label `services.apps.tanzu.vmware.com/class` set to `azure-postgres`.

In addition, you must grant RBAC permissions to Services Toolkit for reading the secrets that the class specifies. Create the following RBAC on your AKS cluster by running:

```
cat <<EOF | kubectl apply -f -
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: stk-secret-reader
  labels:
    servicebinding.io/controller: "true"
rules:
- apiGroups:
  - ""
  resources:
  - secrets
  verbs:
  - get
  - list
  - watch
EOF
```

To claim resources across namespace boundaries, create a corresponding `ResourceClaimPolicy`.

For example, if the provisioned Azure Flexible Server instance exists in the namespace `service-instances`, and you want to allow application operators to claim them for workloads residing in the `default` namespace, you must create the following `ResourceClaimPolicy` by running:

```
cat <<EOF | kubectl apply -f -
---
apiVersion: services.apps.tanzu.vmware.com/v1alpha1
kind: ResourceClaimPolicy
metadata:
  name: default-can-claim-azure-postgres
  namespace: service-instances
spec:
  subject:
    kind: Secret
    group: ""
    selector:
      matchLabels:
        services.apps.tanzu.vmware.com/class: azure-postgres
  consumingNamespaces: [ "default" ]
EOF
```

Discover, Claim, and Bind to a PostgreSQL

Creating the `ClusterInstanceClass` and the corresponding RBAC informs application operators that Azure PostgreSQL is available to use with their application workloads on Tanzu Application Platform.

This section describes how to discover, claim, and bind to the PostgreSQL service instance previously created.

Discovering and claiming service instances is typically the responsibility of the application operator role. Binding is typically an action for application developers.

To discover which service instances are available to them, application operators can run:

```
tanzu services classes list

NAME                    DESCRIPTION
azure-postgres          Azure Flexible Server instances with a postgresql engine
```

You can see information about the `ClusterInstanceClass` created in the earlier step. Each `ClusterInstanceClass` created is added to the list of classes.

Next, the application operator claims an instance of the class they want. But to do that the application operator must first discover the list of currently claimable instances for the class.

Many variables affect the capacity to claim instances, including namespace boundaries, claim policies, and the exclusivity of claims. Therefore, Services Toolkit provides the CLI command `tanzu service claimable list` to help inform application operators of the instances that can cause successful claims.

Example:

```
tanzu services claimable list --class azure-postgres
```

```
  NAME                NAMESPACE  API KIND  API GROUP/VERSION
  aso-psql-bindable  default    Secret    v1
```

Create a claim for the newly created secret by running:

```
tanzu services claim create aso-psql-claim \
  --resource-name aso-psql-bindable \
  --resource-kind Secret \
  --resource-api-version v1
```

Obtain the claim reference of the claim by running:

```
tanzu services claim list -o wide
```

Verify the output is similar to the following:

```
NAME                    READY  REASON  CLAIM REF
aso-psql-claim          True           services.apps.tanzu.vmware.com/v1alpha1:Resourc
eClaim:aso-psql-claim
```

Test claim With Tanzu Application Platform workload

Create an application workload that consumes the claimed Azure PostgreSQL database by running:

```
tanzu apps workload create my-workload
```

Example:

```
tanzu apps workload create my-workload \
  --git-repo https://github.com/sample-accelerators/spring-petclinic \
  --git-tag tap-1.2 \
  --type web \
  --label app.kubernetes.io/part-of=spring-petclinic \
  --annotation autoscaling.knative.dev/minScale=1 \
  --env SPRING_PROFILES_ACTIVE=postgres \
  --service-ref db=services.apps.tanzu.vmware.com/v1alpha1:ResourceClaim:aso-psql-clai
m
```

`--service-ref` is set to the claim reference obtained previously.

Your application workload starts and connects automatically to the Azure PostgreSQL service instance. You can verify this by visiting the app in the browser and, for example, creating a new owner through the UI.

Delete a PostgreSQL service instance

To delete the Azure PostgreSQL service instance, run the appropriate cleanup commands for how you created the service.

Delete a PostgreSQL service instance by using a Carvel package

```
tanzu package installed delete demo-psql-instance
```

Delete a PostgreSQL service instance by using kubectl

Delete the Azure PostgreSQL service instance by running:

```
kubectl delete flexibleservers.dbforpostgresql.azure.com aso-psql
kubectl delete flexibleserversfirewallrules.dbforpostgresql.azure.com aso-psql
kubectl delete flexibleserversdatabases.dbforpostgresql.azure.com aso-psql
kubectl delete SecretTemplate aso-psql-bindable
kubectl delete Password aso-psql
kubectl delete ServiceAccount aso-psql-reader
kubectl delete RoleBinding aso-psql-reader-to-read
kubectl delete Role aso-psql-reading
```

Troubleshooting Azure Service Operator

Azure Service Operator is still in beta and doesn't always behave as expected. For help with most common scenarios, see Troubleshooting.

Prerequisites

To follow the procedures in Consuming Azure Flexible Server for PostgreSQL on Tanzu Application Platform with Azure Service Operator (ASO) you need:

- An Azure AKS Kubernetes cluster
  - This cluster should have a Paid SKU tier. Using the Free tier may cause resource limitation issues.
- Tanzu Application Platform v1.2.0 or later
- Azure Service Operator (ASO) installed on the cluster

If you do not already have a cluster that meets these requirements, you can follow this procedure to create and configure a cluster:

1. Install the Azure CLI. For how to do so, see the Microsoft documentation.
2. Ensure that you are logged in to Azure by running:

   ```
   az login
   ```

3. Create an Azure Kubernetes Service (AKS) cluster. The quickest and simplest way to create an AKS cluster is to use the Azure CLI, as in the following example that creates a new ResourceGroup and AKS cluster:

   ```
   # Name of the resource group to contain the AKS cluster
   RESOURCE_GROUP_NAME=tap-psql-demo

   # Location of the Cluster
   LOCATION=centralus

   # Cluster name
   CLUSTER_NAME=tap-psql-demo-cluster

   # Arbitrary labels for the cluster
   LABELS="key=value key2=value2"
   ```

```
# Number of k8s nodes
NODES=2

az group create --name "${RESOURCE_GROUP_NAME}" --location "${LOCATION}"

az aks create -g "${RESOURCE_GROUP_NAME}" -n "${CLUSTER_NAME}" --enable-managed
-identity --node-count "${NODES}" --enable-addons monitoring --tags "${LABELS}"
 -s Standard_DS3_v2 --generate-ssh-keys  --uptime-sla

az aks get-credentials --resource-group "${RESOURCE_GROUP_NAME}" --name "${CLUS
TER_NAME}"
```

Note: This creates an AKS cluster with a paid tier using the `--uptime-sla` flag. Not setting this flag will cause the Kubernetes Control plane to potentially have resource limitation issues. See https://learn.microsoft.com/en-us/azure/aks/quotas-skus-regions#service-quotas-and-limits

For more information about AKS, see the Microsoft documentation.

4. Install Tanzu Application Platform v1.2.0 or later and Cluster Essentials v1.2.0 or later on the Kubernetes cluster. For more information, see Installing Tanzu Application Platform

5. Verify that you have the appropriate versions by running:

```
kubectl api-resources | grep secrettemplate
```

This command returns the `SecretTemplate` API. If it does not work for you, you might not have Cluster Essentials for VMware Tanzu v1.2.0 or later installed.

6. Install the Azure Service Operator (ASO) and configure it in the cluster. You must have the appropriate permission in Azure to create a service principal and configure Azure access. v2.0.0-beta.2 is known to work with this use case. Install the latest stable version of the operator by running:

```
AZURE_TENANT_ID=$(az account show | jq -r '.tenantId')
AZURE_SUBSCRIPTION_ID=$(az account show | jq -r '.id')

az ad sp create-for-rbac -n tap-azure-service-operator --role contributor \
--scopes /subscriptions/"${AZURE_SUBSCRIPTION_ID}" > /tmp/aso-creds.json

AZURE_CLIENT_ID=$(cat /tmp/aso-creds.json | jq -r '.appId')
AZURE_CLIENT_SECRET=$(cat /tmp/aso-creds.json | jq -r '.password' )

rm -f  /tmp/aso-creds.json

# requires carvel kapp v0.46+
kapp deploy -a aso -f https://github.com/Azure/azure-service-operator/releases/
download/v2.0.0-beta.2/azureserviceoperator_v2.0.0-beta.2.yaml -y --wait=false

cat <<EOF | kubectl apply -f -
apiVersion: v1
kind: Secret
metadata:
  name: aso-controller-settings
  namespace: azureserviceoperator-system
stringData:
  AZURE_SUBSCRIPTION_ID: "${AZURE_SUBSCRIPTION_ID}"
  AZURE_TENANT_ID: "${AZURE_TENANT_ID}"
  AZURE_CLIENT_ID: "${AZURE_CLIENT_ID}"
  AZURE_CLIENT_SECRET: "${AZURE_CLIENT_SECRET}"
EOF

kubectl wait deployment -n azureserviceoperator-system -l app=azure-service-ope
rator-v2 --for=condition=Available=True
```

Next Steps

See Consuming Azure Flexible Server for PostgreSQL on Tanzu Application Platform with Azure Service Operator (ASO).

Creating Azure PostgreSQL Instances manually using kubectl (experimental)

This topic describes how to use Services Toolkit to allow Tanzu Application Platform workloads to consume Azure Flexible Server PostgreSQL. This particular topic makes use of Azure Service Operator v2 to manage PostgreSQL instances in Azure.

Create a resource group

First of all, a ResourceGroup for all PSQL Instances to reside in will be created:

```
cat <<EOF | kubectl apply -f -
---
apiVersion: resources.azure.com/v1beta20200601
kind: ResourceGroup
metadata:
  name: aso-psql
spec:
  location: centralus
EOF
```

Create a Flexible Server service instance

Next, you will create a Flexible Server PSQL Instance, a Database and a Firewall Rule in Azure as well as a Secret for credentials. In this guide you will leverage the `Password` API from Carvel's secretgen controller, which will create the `Secrets` for you. However, any other mechanism to manage those secrets works too.

Change the `.spec.azureName` of the `FlexibleServer` resource below from "aso-psql" to something unique, using only lowercase letters, digits and hyphens. This avoids naming conflicts as Azure has a global naming namespace and this resource may already exist.

```
cat <<'EOF' | kubectl apply -f -
---
apiVersion: secretgen.k14s.io/v1alpha1
kind: Password
metadata:
  name: aso-psql
spec:
  length: 64
  secretTemplate:
    type: Opaque
    stringData:
      password: $(value)
---
apiVersion: dbforpostgresql.azure.com/v1beta20210601
kind: FlexibleServersDatabase
metadata:
  name: aso-psql
spec:
  azureName: mydb
  owner:
    name: aso-psql
  charset: utf8
---
apiVersion: dbforpostgresql.azure.com/v1beta20210601
kind: FlexibleServersFirewallRule
metadata:
  name: aso-psql
spec:
  owner:
    name: aso-psql
  startIpAddress: 0.0.0.0 #! only allow traffic from azure. See https://docs.microsoft
.com/en-us/azure/postgresql/single-server/concepts-firewall-rules#connecting-from-azur
e. Warning not for production use.
  endIpAddress: 0.0.0.0
---
apiVersion: dbforpostgresql.azure.com/v1beta20210601
kind: FlexibleServer
metadata:
  name: aso-psql
spec:
  location: centralus
  azureName: aso-psql #! CHANGE THIS NAME
  owner:
    name: aso-psql #! the ResourceGroup above
  version: "13" #! only 11,12,13 supported
  sku:
    name: Standard_D4s_v3
    tier: GeneralPurpose
  administratorLogin: myAdmin
  administratorLoginPassword:
    name: aso-psql
    key: password
  storage:
    storageSizeGB: 128
EOF
```

Create a Binding Specification Compatible Secret

As mentioned in Creating service instances that are compatible with Tanzu Application Platform, in order for Tanzu Application Platform workloads to be able to claim and bind to services such as Azure PostgreSQL, a resource compatible with Service Binding Specification must exist in the cluster. This can take the form of either a `ProvisionedService`, as defined by the specification, or a

Kubernetes `Secret` with some known keys, also as defined in the specification.

In this guide, you create a Kubernetes secret in the necessary format using the secretgen-controller tooling. You do so by using the `SecretTemplate` API to extract values from the Azure Service Operator resources and populate a new spec-compatible secret with the values.

Create a ServiceAccount for Secret Templating

As part of using the `SecretTemplate` API, a Kubernetes `ServiceAccount` must be provided. The `ServiceAccount` is used for reading the `FlexibleServer` resource and the `Secret` created from the `Password` resource above.

Create the following Kubernetes resources on your AKS cluster:

```
cat <<EOF | kubectl apply -f -
---
apiVersion: v1
kind: ServiceAccount
metadata:
  name: aso-psql-reader
  namespace: default
---
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: aso-psql-reading
  namespace: default
rules:
- apiGroups:
  - ""
  resources:
  - secrets
  verbs:
  - get
  - list
  - watch
  resourceNames:
  - aso-psql
- apiGroups:
  - dbforpostgresql.azure.com
  resources:
  - flexibleservers
  - flexibleserversdatabases
  verbs:
  - get
  - list
  - watch
  resourceNames:
  - aso-psql
---
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: aso-psql-reader-to-read
  namespace: default
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: Role
  name: aso-psql-reading
subjects:
- kind: ServiceAccount
  name: aso-psql-reader
  namespace: default
EOF
```

Create a SecretTemplate

In combination with the `ServiceAccount` just created, a `SecretTemplate` can be used to declaratively create a secret that is compatible with the service binding specification. For more information on this API see the Secret Template Documentation.

Create the following Kubernetes resources on your AKS cluster:

```
cat <<'EOF' | kubectl apply -f -
---
apiVersion: secretgen.carvel.dev/v1alpha1
kind: SecretTemplate
metadata:
  name: aso-psql-bindable
  namespace: default
spec:
  serviceAccountName: aso-psql-reader
  inputResources:
  - name: server
    ref:
      apiVersion: dbforpostgresql.azure.com/v1alpha1api20210601
      kind: FlexibleServer
      name: aso-psql
  - name: db
    ref:
      apiVersion: dbforpostgresql.azure.com/v1alpha1api20210601
      kind: FlexibleServersDatabase
      name: aso-psql
  - name: creds
    ref:
      apiVersion: v1
      kind: Secret
      name: "$(.server.spec.administratorLoginPassword.name)"
  template:
    metadata:
      labels:
        app.kubernetes.io/component: aso-psql
        app.kubernetes.io/instance: "$(.server.metadata.name)"
        services.apps.tanzu.vmware.com/class: azure-postgres
    type: postgresql
    stringData:
      type: postgresql
      port: "5432"
      database: "$(.db.status.name)"
      host: "$(.server.status.fullyQualifiedDomainName)"
      username: "$(.server.status.administratorLogin)"
    data:
      password: "$(.creds.data.password)"
EOF
```

Verify the Service Instance

Firstly wait until the PostgreSQL instance is ready. This may take 5 to 10 minutes.

```
kubectl wait flexibleservers.dbforpostgresql.azure.com aso-psql -n default --for=condi
tion=Ready --timeout=5m
```

Next, ensure a bindable `Secret` was produced by the `SecretTemplate`. To do so, run:

```
kubectl wait SecretTemplate -n default aso-psql-bindable --for=condition=ReconcileSucc
eeded --timeout=5m

kubectl get Secret -n default aso-psql-bindable
```

Creating Azure PostgreSQL instances by using a Carvel package (experimental)

This topic describes creating, updating, and deleting Azure PostgreSQL service instances using a Carvel package. For a more detailed and low-level alternative procedure, see Creating Service Instances that are compatible with Tanzu Application Platform.

Prerequisite

Meet the prerequisites:

The Package Repository and service instance Package Bundles for this guide can be found in the Reference Service Packages GitHub repository.

Create an Azure PostgreSQL service instance using a Carvel package

Follow the steps in the following procedures.

Add a reference package repository to the cluster

The namespace `tanzu-package-repo-global` has a special significance. The kapp-controller defines a Global Packaging namespace. In this namespace, any package the is made available through a Package Respository, is available in every namespace.

When the kapp-controller is installed via Tanzu Application Platform, the namespace is `tanzu-package-repo-global`. If you install the controller in another way, verify which namespace is considered the Global Packaging namespace.

To add a reference package repository to the cluster:

1. Use the Tanzu CLI to add the new Service Reference packages repository:

   ```
   tanzu package repository add tap-reference-service-packages \
       --url ghcr.io/vmware-tanzu/tanzu-application-platform-reference-service-pac
   kages:0.0.3 \
       -n tanzu-package-repo-global
   ```

2. Create a `ServiceAccount` to provision `PackageInstall` resources by using the following example. The namespace of this `ServiceAccount` must match the namespace of the `tanzu package install` command in the next step.

```
kubectl apply -f - <<'EOF'
---
apiVersion: v1
kind: ServiceAccount
metadata:
  name: psql-install
---
kind: Role
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: psql-install
rules:
- apiGroups: ["dbforpostgresql.azure.com"]
  resources: ["flexibleservers","flexibleserversdatabases","flexibleserversfire
wallrules"]
  verbs:     ["*"]
- apiGroups: ["resources.azure.com"]
  resources: ["resourcegroups"]
  verbs:     ["*"]
- apiGroups: ["secretgen.carvel.dev", "secretgen.k14s.io"]
  resources: ["secrettemplates","passwords"]
  verbs:     ["*"]
- apiGroups: [""]
  resources: ["serviceaccounts","configmaps"]
  verbs:     ["*"]
- apiGroups: [""]
  resources: ["namespaces"]
  verbs:     ["get", "list"]
- apiGroups: ["rbac.authorization.k8s.io"]
  resources: ["roles","rolebindings"]
  verbs:     ["*"]
---
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: psql-install
subjects:
- kind: ServiceAccount
  name: psql-install
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: Role
  name: psql-install
```

Create a Azure PostgreSQL service instance through the Tanzu CLI

Before you create the values file, here are some values highlighted.

- **aso_controller_namespace**: the Azure Service Operator has some potential conflicting behaviors with the kapp-controller. We reduce the conflicts by annotating the resources with the ASO installation namespace.
- **firewall_rules**: by default, the FlexibleServer is not accessible. Setting `0.0.0.0` as the start

and end IP addresses for a firewall rule makes the server available from within Azure.

- **resource_group.use_existing**: if you cannot create a Resource Group in Azure or have other reasons for using an existing one, set this to `true`. Else, the package makes a Resource Group with the name specified by the `resource_group.name` value.

The `server.name` field will be used for the FlexibleServer resource name on Azure, otherwise `name` will be used. It is recommended to set the value of the `name` (and the optional `server.name`) field below from `aso-psql` to something unique, using only lowercase letters, digits and hyphens. This avoids naming conflicts, as Azure has a global naming namespace for FlexibleServer instances and this resource may already exist. Do make sure you also change the commands below using a `aso-psql` value, such as the `aso-psql-bindable` from the SecretTemplate,and replace `aso-psql` with the actual `name`.

1. Create a file holding the configuration of the Azure PostgreSQL service instance:

```
cat <<'EOF' > aso-psql-instance-values.yml
---
name: aso-psql
namespace: service-instances
location: westeurope
aso_controller_namespace: azureserviceoperator-system
create_namespace: false

server:
    administrator_name: trpadmin

database:
    name: testdb

firewall_rules:
    - startIpAddress: 0.0.0.0
      endIpAddress: 0.0.0.0

resource_group:
    use_existing: false
    name: aso-psql
EOF
```

Note

: To understand which settings are available for this package you can run:

```
tanzu package available get \
  --values-schema \
  psql.azure.references.services.apps.tanzu.vmware.com/0.0.1-alpha
```

This shows a list of all configuration options you can use in the `aso-psql-instance-values.yml` file.

2. Use the Tanzu CLI to install an instance of the reference service instance Package.

```
tanzu package install aso-psql-instance \
  --package-name psql.azure.references.services.apps.tanzu.vmware.com \
  --version 0.0.1-alpha \
  --service-account-name psql-install \
  --values-file aso-psql-instance-values.yml \
  --wait
```

You can install the `psql.azure.references.services.apps.tanzu.vmware.com` package multiple times to produce various Azure PostgreSQL Service instances. You create a separate `<INSTANCE-NAME>-values.yml` for each instance, set a different `name` value, and then install the package with the instance-specific data values file.

Verify the Azure Resources

1. Verify the creation status for the Azure PostgreSQL instance by inspecting the conditions in the Kubernetes API. To do so, run:

   ```
   kubectl get flexibleservers.dbforpostgresql.azure.com aso-psql -o yaml
   ```

2. After some time has passed, sometimes up to 10 minutes, you can find the binding-compliant secret produced by `PackageInstall`. To do so, run:

   ```
   kubectl get secrettemplate aso-psql-bindable -o jsonpath="{.status.secret.name}
   "
   ```

Verify the Service Instance

Firstly wait until the PostgreSQL instance is ready. This may take 5 to 10 minutes.

```
kubectl wait flexibleservers.dbforpostgresql.azure.com aso-psql -n default --for=condi
tion=Ready --timeout=5m
```

Next, ensure a bindable `Secret` was produced by the `SecretTemplate`. To do so, run:

```
kubectl wait SecretTemplate -n default aso-psql-bindable --for=condition=ReconcileSucc
eeded --timeout=5m

kubectl get Secret -n default aso-psql-bindable
```

Summary

You have learnt to use Carvel's `Package` and `PackageInstall` APIs to create a Azure PostgreSQL service instance. If you want to learn more about the pieces that comprise this service instance package, see Creating Azure PostgreSQL Instances manually using kubectl.

Now that you have this available in the cluster, you can learn how to make use of it by continuing where you left off in Consuming Azure PostgreSQL on Tanzu Application Platform (TAP) with ASO.

Azure Service Operator Troubleshooting

Increase Log Level

There is a guide on the Azure Service Operator (ASO) controller for aiding you in diagnosing problems.

We recommend temporarily change the Controller's binary log level from `v=2` to `v=6`. Setting it higher than six prints a lot more things, such as the HTTP requests with headers, and usually doesn't add more value.

```
kubectl edit deploy -n azureserviceoperator-system azureserviceoperator-controller-man
ager
```

```
spec:
  template:
    spec:
      containers:
      - name: manager
        args:
        - --metrics-addr=0.0.0.0:8080
        - --health-addr=:8081
        - --enable-leader-election
        - --v=6
```

Not Updating The Kubernetes Resources

The ASO controller sometimes conflicts when updating the resource status in Kubernetes. The resource in Azure exists, but is not reflected properly in its corresponding Kubernetes resource.

In the logs you will see a `409 conflict` message when updating the Kubernetes resource. To resolve this, you can restart the Pod, which will take a few seconds.

```
kubectl -n azureserviceoperator-system rollout restart deployment azureserviceoperator
-controller-manager
```

Consuming Azure Flexible Server for PostgreSQL on Tanzu Application Platform with Crossplane

Introduction

This topic demonstrates how to use Services Toolkit to allow Tanzu Application Platform workloads to consume Azure Flexible Server for PostgreSQL. This particular topic makes use of Crossplane to manage those Flexible Server for PostgreSQL instances. As such, it can be thought of as an alternative approach to Consuming Azure Flexible Server for PostgreSQL on Tanzu Application Platform with Azure Service Operator (ASO) to achieve the similar outcomes.

Prerequisites

Meet these prerequisites:

- Install Azure CLI
- Create an AKS cluster
- Install Tanzu Application Platform (v1.2.0 or later) and Cluster Essentials (v1.2.0 or later)

Note

In this example we use an AKS Cluster to deploy Crossplane and Tanzu Application Platform too. However, any other cluster which supports running those two systems should suffice.

Install Crossplane

Note

For the latest steps for installing Crossplane, see these instructions. For the instructions in this topic, it is important to enable support for external secret stores in Crossplane. This is currently an Alpha feature. As such, you will have to explicitly set command line flag `--enable-external-secret-stores` when starting the Crossplane controller.

Run the following commands to install Crossplane to your existing Kubernetes cluster:

```
kubectl create namespace crossplane-system

helm repo add crossplane-stable https://charts.crossplane.io/stable
helm repo update

helm install crossplane --namespace crossplane-system crossplane-stable/crossplane \
  --set 'args={--enable-external-secret-stores}'
```

For this topic, you do not need to install the Crossplane CLI or any additional configuration package.

Install the Azure Provider for Crossplane

To install the Azure Provider for Crossplane, run:

```
kubectl apply -f - <<'EOF'
apiVersion: pkg.crossplane.io/v1alpha1
kind: ControllerConfig
metadata:
  name: jet-azure-config
spec:
  image: crossplane/provider-jet-azure-controller:v0.12.0
  args: ["-d"]
---
apiVersion: pkg.crossplane.io/v1
kind: Provider
metadata:
  name: provider-jet-azure
spec:
  package: crossplane/provider-jet-azure:v0.12.0
  controllerConfigRef:
    name: jet-azure-config
EOF
```

After you have installed the provider, you see a new `flexibleservers.dbforpostgresql.azure.jet.crossplane.io` API resource available in your Kubernetes cluster. You can wait for the provider to become healthy by running:

```
kubectl -n crossplane-system wait provider/provider-jet-azure \
  --for=condition=Healthy=True --timeout=3m
```

Install the Kubernetes Provider for Crossplane

To install the Kubernetes Provider for Crossplane, run:

```
kubectl apply -f - <<'EOF'
apiVersion: pkg.crossplane.io/v1
kind: Provider
metadata:
  name: provider-kubernetes
spec:
  package: "crossplane/provider-kubernetes:main"
EOF
```

Configure the Azure Provider

This section creates a new Service Principal to be used by the Crossplane system to allow it to manage PostgreSQL Servers.

1. Setup some configuration in the current shell session

```
# Set the name of the Service Principal to be created
AZURE_SP_NAME='sql-crossplane-demo'

# Get the subscription ID
AZURE_SUBSCRIPTION_ID="$( az account show -o json | jq -r '.id' )"
```

2. Create a new Service Principal and set up the kubernetes secret

```
kubectl create secret generic jet-azure-creds -o yaml --dry-run=client --from-l
iteral=creds="$(
 az ad sp create-for-rbac -n "${AZURE_SP_NAME}" \
   --sdk-auth \
   --role "Contributor" \
   --scopes "/subscriptions/${AZURE_SUBSCRIPTION_ID}" \
   -o json
)" | kubectl apply -n crossplane-system -f -
```

Note

You'll see the following warning:

`WARNING: Option '--sdk-auth' has been deprecated and will be removed in a future release.`

which you can ignore for now. There is some context about that in this issue for the Azure CLI and this issue for the Crossplane Azure Provider.

3. Deploy a `ProviderConfig` which uses the previously created secret for the Azure crossplane provider

```
kubectl apply -f - <<'EOF'
apiVersion: azure.jet.crossplane.io/v1alpha1
kind: ProviderConfig
metadata:
 name: default
spec:
 credentials:
   source: Secret
   secretRef:
     namespace: crossplane-system
     name: jet-azure-creds
     key: creds
EOF
```

Configure the Kubernetes Provider

```
SA=$(kubectl -n crossplane-system get sa -o name | grep provider-kubernetes | sed -e '
s|serviceaccount\/|crossplane-system:|g')
kubectl create role -n crossplane-system password-manager --resource=passwords.secretg
en.k14s.io --verb=create,get,update,delete
kubectl create rolebinding -n crossplane-system provider-kubernetes-password-manager -
-role password-manager --serviceaccount="${SA}"

kubectl apply -f - <<'EOF'
apiVersion: kubernetes.crossplane.io/v1alpha1
kind: ProviderConfig
metadata:
  name: default
spec:
  credentials:
    source: InjectedIdentity
EOF
```

Define Composite Resource Types

Now that the Azure Provider for Crossplane has been installed and configured, create a new `CompositeResourceDefinition` (XRD) and corresponding `Composition` representing individual instances of Azure PostgreSQL Server. For more information about these concepts see the Crossplane Composition documentation.

1. Create a new XRD by running:

```
kubectl apply -f - <<'EOF'
apiVersion: apiextensions.crossplane.io/v1
kind: CompositeResourceDefinition
metadata:
 name: xpostgresqlinstances.bindable.database.example.org
spec:
 claimNames:
   kind: PostgreSQLInstance
   plural: postgresqlinstances
```

```
  connectionSecretKeys:
  - type
  - provider
  - host
  - port
  - database
  - username
  - password
  group: bindable.database.example.org
  names:
    kind: XPostgreSQLInstance
    plural: xpostgresqlinstances
  versions:
  - name: v1alpha1
    referenceable: true
    schema:
      openAPIV3Schema:
        properties:
          spec:
            properties:
              parameters:
                properties:
                  storageGB:
                    type: integer
                required:
                - storageGB
                type: object
            required:
            - parameters
            type: object
        type: object
    served: true
EOF
```

After the newly created XRD has been successfully reconciled, there are two new API resources available in your Kubernetes cluster, `xpostgresqlinstances.bindable.database.example.org` and `postgresqlinstances.bindable.database.example.org`.

2. Create a corresponding composition (not in a production environment) by running:

```
kubectl apply -f - <<'EOF'
apiVersion: apiextensions.crossplane.io/v1
kind: Composition
metadata:
  labels:
    provider: azure
  name: xpostgresqlinstances.bindable.gcp.database.example.org
spec:
  compositeTypeRef:
    apiVersion: bindable.database.example.org/v1alpha1
    kind: XPostgreSQLInstance
  publishConnectionDetailsWithStoreConfigRef:
    name: default
  resources:
  - name: dbinstance
    base:
      apiVersion: dbforpostgresql.azure.jet.crossplane.io/v1alpha2
      kind: FlexibleServer
      spec:
        forProvider:
          administratorLogin: myPgAdmin
          administratorPasswordSecretRef:
            name: ""
            namespace: crossplane-system
            key: password
          location: westeurope
          skuName: GP_Standard_D2s_v3
          version: "12" #! 11,12 and 13 are supported
          resourceGroupName: tap-psql-demo
        writeConnectionSecretToRef:
          namespace: crossplane-system
    connectionDetails:
    - name: type
      value: postgresql
    - name: provider
      value: azure
    - name: database
      value: postgres
    - name: username
      fromFieldPath: spec.forProvider.administratorLogin
    - name: password
      fromConnectionSecretKey: "attribute.administrator_password"
    - name: host
      fromFieldPath: status.atProvider.fqdn
    - name: port
      type: FromValue
      value: "5432"
    patches:
    - fromFieldPath: metadata.uid
      toFieldPath: spec.writeConnectionSecretToRef.name
      transforms:
      - string:
          fmt: '%s-postgresql'
          type: Format
        type: string
      type: FromCompositeFieldPath
    - type: FromCompositeFieldPath
      fromFieldPath: metadata.name
      toFieldPath: spec.forProvider.administratorPasswordSecretRef.name
    - fromFieldPath: spec.parameters.storageGB
      toFieldPath: spec.forProvider.storageMb
      type: FromCompositeFieldPath
      transforms:
      - type: math
        math:
          multiply: 1024
  - name: dbfwrule
    base:
      apiVersion: dbforpostgresql.azure.jet.crossplane.io/v1alpha2
      kind: FlexibleServerFirewallRule
      spec:
        forProvider:
          serverIdSelector:
            matchControllerRef: true
          #! not recommended for production deployments!
          startIpAddress: 0.0.0.0
          endIpAddress: 255.255.255.255
  - name: password
    base:
      apiVersion: kubernetes.crossplane.io/v1alpha1
      kind: Object
      spec:
        forProvider:
          manifest:
            apiVersion: secretgen.k14s.io/v1alpha1
            kind: Password
            metadata:
              name: ""
              namespace: crossplane-system
            spec:
              length: 64
              secretTemplate:
                type: Opaque
                stringData:
                  password: $(value)
    patches:
    - type: FromCompositeFieldPath
      fromFieldPath: metadata.name
      toFieldPath: spec.forProvider.manifest.metadata.name
EOF
```

The composition defined above makes sure that all `FlexibleServers` are placed in the `westeurope` region and under the resource group `tap-psql-demo`. This composition fulfils the XRD previously created.

Warning: Setting the `FlexibleServerFirewallRule` to start at `0.0.0.0` and end at `255.255.255.255` will allow access to the PostgreSQL Server from any IP and is not recommended in a production environment.

Create an Instance Class

In order to make instances of a service easily discoverable and claimable by Application Operators, the role of the Service Operator creates a `ClusterInstanceClass`. In this particular example, the class states that claimable instances of PostgreSQL instances are represented by `Secret` objects of type `connection.crossplane.io/v1alpha1` with label `services.apps.tanzu.vmware.com/class` set to `azure-postgres`:

```
kubectl apply -f - <<'EOF'
apiVersion: services.apps.tanzu.vmware.com/v1alpha1
kind: ClusterInstanceClass
metadata:
  name: azure-postgres
spec:
  description:
    short: Azure Postgresql database instances
  pool:
    kind: Secret
    labelSelector:
      matchLabels:
        services.apps.tanzu.vmware.com/class: azure-postgres
    fieldSelector: type=connection.crossplane.io/v1alpha1
EOF
```

In addition, you need to grant sufficient RBAC permissions to Services Toolkit to be able to read the secrets specified by the class.

```
kubectl apply -f - <<'EOF'
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: stk-secret-reader
  labels:
    servicebinding.io/controller: "true"
rules:
- apiGroups:
  - ""
  resources:
  - secrets
  verbs:
  - get
  - list
  - watch
EOF
```

Provision Azure Flexible Server for PostgreSQL instances

Playing the role of the Service Operator, you now provision an instance of an Azure Flexible Server for PostgreSQL using the `postgresqlinstances.bindable.database.example.org` API managed by the XRD you previously created. Note that `.spec.publishConnectionDetailsTo` provides Crossplane with the name and a label for the secret that is being used to store the connection details for the newly created database. You can see that the label specified here matches the label selector defined on the `ClusterInstanceClass` you created in the previous step.

The `PostgreSQLInstance` has a dependency on a `Secret` where the Service Operator needs to specify the password for the admin user. Here we use Carvel's `Password` API to create this `Secret` for us.

Run the following command:

```
kubectl apply -f - <<'EOF'
apiVersion: bindable.database.example.org/v1alpha1
kind: PostgreSQLInstance
metadata:
  name: postgresql-server
  namespace: default
spec:
  parameters:
    #! supported storage sizes: 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384,
32768
    storageGB: 32
  compositionSelector:
    matchLabels:
      provider: azure
  publishConnectionDetailsTo:
    name: postgresql-server
    metadata:
      labels:
        services.apps.tanzu.vmware.com/class: azure-postgres
EOF
```

Running this command will cause the creation of a Azure Flexible Server for PostgreSQL instance in your Azure account. You can use the Azure CLI to verify this:

```
az postgres flexible-server list -o table
```

After the instance has been successfully created, the status of the newly created `PostgreSQLInstance` resource should show `READY=True`. This might take a few minutes. You can wait for this by running:

```
kubectl wait postgresqlinstances.bindable.database.example.org/postgresql-server \
    --for=condition=Ready=true --timeout=10m
```

As soon as the Azure Flexible Server for PostgreSQL instance is ready, it is claimable by the role of the Application Operator as shown in the next section.

Note

There is currently a bug in Crossplane 1.7.2 onwards with the `--enable-external-secret-stores` feature gate enabled where the controller will fail to clean up a local secret created by the field `.spec.publishConnectionDetailsTo` after the deletion of the claim. A workaround is to temporarily give the crossplane controller the necessary i.e. permissions:

```
kubectl create clusterrole crossplane-cleaner --verb=delete --resource=s
ecrets
kubectl create clusterrolebinding crossplane-cleaner --clusterrole=cross
plane-cleaner --serviceaccount=crossplane-system:crossplane
```

Claim the Azure Flexible Server for PostgreSQL Server instance and connect to it from the Tanzu Application Platform Workload

Thanks to the previously created `ClusterInstanceClass`, `Secrets` representing PostgreSQL Server instances can now be discovered and claimed by Application Operators through the Tanzu CLI as shown below.

1. Show available classes of service instances by running:

   ```
   tanzu service classes list

     NAME              DESCRIPTION
     azure-postgres    Azure Postgresql database instances
   ```

2. Show claimable instances belonging to the PostgreSQL Server instance class by running:

   ```
   tanzu services claimable list --class azure-postgres
   ```

```
  NAME                 NAMESPACE  API KIND  API GROUP/VERSION
  postgresql-server    default    Secret    v1
```

3. Create a claim for the discovered instance by running:

```
tanzu service claim create postgresql-server-claim \
  --resource-name postgresql-server\
  --resource-kind Secret \
  --resource-api-version v1
```

4. Obtain the claim reference by running:

```
tanzu service claim list -o wide
```

Expect to see the following output:

```
NAME                         READY  REASON  CLAIM REF
postgresql-server-claim      True           services.apps.tanzu.vmware.com/v1alp
ha1:ResourceClaim:postgresql-server-claim
```

5. Create an application workload that consumes the claimed PostgreSQL Server instance by running:

Example:

```
tanzu apps workload create my-workload \
  --git-repo https://github.com/sample-accelerators/spring-petclinic \
  --git-branch main \
  --git-tag tap-1.2 \
  --type web \
  --label app.kubernetes.io/part-of=spring-petclinic \
  --annotation autoscaling.knative.dev/minScale=1 \
  --env SPRING_PROFILES_ACTIVE=postgres \
  --service-ref db=services.apps.tanzu.vmware.com/v1alpha1:ResourceClaim:postgr
esql-server-claim
```

Note that `--service-ref` is being set to the claim reference obtained previously.

Consuming Google Cloud SQL on Tanzu Application Platform

This section of the documentation covers integrations of Google Cloud SQL into Tanzu Application Platform. Documentation is provided for both an integration using Config Connector, as well as an integration using Crossplane.

Consuming Google Cloud SQL on Tanzu Application Platform (TAP) with Config Connector

Introduction

This topic demonstrates how to use Services Toolkit to allow TAP Workloads to consume Google Cloud SQL for PostgreSQL databases. This particular guide makes use of Config Connector to manage PostgreSQL instances in GCP.

This is describing the procedures to produce similar outcomes as in "Consuming AWS RDS on Tanzu Application Platform (TAP) with AWS Controllers for Kubernetes (ACK)". The same points discussed in "Creating Service Instances that are compatible with Tanzu Application Platform" apply here too:

- Neither of the resources discussed below adhere to the Service Binding Specification
- We need to manage the lifecycle of multiple resources which together form a usable database instance

Note Please ensure you have met all prerequisites before reading on.

Creating Service Instances that are compatible with Tanzu Application Platform

The installation of the Config Connector Addon results in the availability of new Kubernetes APIs for interacting with Google Cloud resources, specifically Cloud SQL resources, from within the TAP cluster.

```
$ kubectl api-resources --api-group sql.cnrm.cloud.google.com

NAME            SHORTNAMES                         APIVERSION                          NA
MESPACED    KIND
sqldatabases    gcpsqldatabase,gcpsqldatabases    sql.cnrm.cloud.google.com/v1beta1    tr
ue          SQLDatabase
sqlinstances    gcpsqlinstance,gcpsqlinstances    sql.cnrm.cloud.google.com/v1beta1    tr
ue          SQLInstance
sqlsslcerts     gcpsqlsslcert,gcpsqlsslcerts      sql.cnrm.cloud.google.com/v1beta1    tr
ue          SQLSSLCert
sqlusers        gcpsqluser,gcpsqlusers            sql.cnrm.cloud.google.com/v1beta1    tr
ue          SQLUser
```

To create a CloudSQL service instance for consumption by Tanzu Application Platform, you can use a ready-made, reference Carvel Package. This step is typically performed by the role of the Service Operator. Follow the steps in Creating an CloudSQL service instance by using a Carvel Package.

Alternatively, if you are interested in authoring your own Reference Package and want to learn about the underlying APIs and how they come together to produce a useable service instance for Tanzu Application Platform, you can achieve the same outcome by using the more advanced Creating an CloudSQL service instance manually.

Once you have completed either of these steps and have a running CloudSQL service instance, please return here to continue with the rest of the use case.

Creating a Service Instance Class for Cloud SQL

We can now make the Cloud SQL Service Instance discoverable to Application Operators. This step is typically performed by the role of the Service Operator.

You can use Services Toolkit's `ClusterInstanceClass` API to create a "Service Instance Class" to represent Cloud SQL Service Instances within the cluster. The existence of such classes make these logical Service Instances discoverable to Application Operators, thus allowing them to create Resource Claims for such instances and to then bind them to Application Workloads.

Create the following Kubernetes resource::

```
apiVersion: services.apps.tanzu.vmware.com/v1alpha1
kind: ClusterInstanceClass
metadata:
  name: cloudsql-postgres
spec:
  description:
    short: Google Cloud SQL with a postgresql engine
  pool:
    kind: Secret
    labelSelector:
      matchLabels:
        services.apps.tanzu.vmware.com/class: cloudsql-postgres
```

In this particular example, the class states that claimable instances of Cloud SQL Postgresql are represented by `Secret` objects with label `services.apps.tanzu.vmware.com/class` set to `cloudsql-postgres`. A `Secret` with this label was created earlier when you created the CloudSQL service instance.

Although this example uses `services.apps.tanzu.vmware.com/class`, there is no special meaning to that key. The Service Operator persona can choose arbitrary label names and values. They might also decide to select on multiple labels or combine a label selector with a field selector when defining the `ClusterInstanceClass`.

Now that you have created a `ClusterInstanceClass`, you need to grant sufficient RBAC permissions to enable Services Toolkit to read the resources that match the pool definition of the instance class. For this example, create the following aggregated `ClusterRole` in your cluster:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: stk-secret-reader
  labels:
    servicebinding.io/controller: "true"
rules:
- apiGroups: [ "" ]
  resources: [ "secrets" ]
  verbs:     [ "get", "list", "watch" ]
```

If you want to claim resources across namespace boundaries, you will have to create a corresponding `ResourceClaimPolicy`. For example, if the provisioned Cloud SQL instances exist in namespace `service-instances` and you want to allow App Operators to claim them for workloads residing in the `default` namespace, you would have to create the following `ResourceClaimPolicy`:

```
#! optional, when workload and services are in different namespaces
apiVersion: services.apps.tanzu.vmware.com/v1alpha1
kind: ResourceClaimPolicy
metadata:
  name: default-can-claim-cloudsql-postgres
  namespace: service-instances
spec:
  subject:
    kind: Secret
    group: ""
    selector:
      matchLabels:
```

```
      services.apps.tanzu.vmware.com/class: cloudsql-postgres
  consumingNamespaces: [ "default" ]
```

Discover, Claim and Bind to a Google Cloud SQL Postgresql Instance

The act of creating the `ClusterInstanceClass` and the corresponding RBAC essentially advertises to Application Operators that Cloud SQL Instances are available to use with their Application Workloads on Tanzu Application Platform. In this step you will learn how to discover, claim and bind to the Cloud SQL Service Instance previously created. Discovery and claiming of Service Instances is typically the role of the Application Operator while binding is typically a step for Application Developers.

To discover what Service Instances are available to them, Application Operators can use the `tanzu services classes list` command.

```
tanzu services classes list

  NAME                    DESCRIPTION
  cloudsql-postgres       Google Cloud SQL with a postgresql engine
```

Here you can see information about the `ClusterInstanceClass` created in the previous step. Each `ClusterInstanceClass` created will be added to the list of classes returned here.

The next step is to “claim” an instance of the desired class, but in order to do that, Application Operators must first discover the list of currently claimable instances for the class. Claimability of instances is affected by many variables (including namespace boundaries, claim policies and the exclusivity of claims) and so Services Toolkit provides a CLI command to help inform Application Operators of the instances that will result in successful claims. This command is the `tanzu service claimable list` command.

```
tanzu services claimable list --class cloudsql-postgres

  NAME                      NAMESPACE          KIND    APIVERSION
  sql-instance-claimable    service-instances  Secret  v1
```

Due to the setup done as part of creating a claimable class for Cloud SQL instances, the `Secret`s created from the `SecretTemplate` now appears as “claimable” to the Application Operator. From here on it is simply a case of creating a resource claim for the instance and then binding the claim to an Application Workload.

Create a claim for the newly created secret by running:

```
tanzu service claim create cloudsql-postgres-claim \
  --resource-name sql-instance-claimable \
  --resource-namespace service-instances \
  --resource-kind Secret \
  --resource-api-version v1
```

Obtain the claim reference of the claim by running:

```
tanzu service claim list -o wide
```

Expect to see the following output:

```
NAME                       READY  REASON   CLAIM REF
cloudsql-postgres-claim  True   Ready    services.apps.tanzu.vmware.com/v1alpha1:Resour
ceClaim:cloudsql-postgres-claim
```

Create an Application Workload that consumes the claimed Cloud SQL Postgresql Database by running:

Example:

```
tanzu apps workload create my-workload \
  --git-repo https://github.com/sample-accelerators/spring-petclinic \
  --git-branch main \
  --git-tag tap-1.2 \
  --type web \
  --label app.kubernetes.io/part-of=spring-petclinic \
  --annotation autoscaling.knative.dev/minScale=1 \
  --env SPRING_PROFILES_ACTIVE=postgres \
  --service-ref db=services.apps.tanzu.vmware.com/v1alpha1:ResourceClaim:cloudsql-post
gres-claim
```

`--service-ref` is set to the claim reference obtained previously.

Congratulations - your Application Workload will now start up and will connect automatically to the Cloud SQL Service Instance. This can be verified by visiting the app in the browser and, for example, creating a new “Owner” through the GUI.

Prerequisites

The following prerequisites must be met in order to follow along with Consuming Cloud SQL on Tanzu Application Platform (TAP) with Config Connector.

The `gcloud` CLI

You need to have the `gcloud` CLI installed and authenticated.

A Kubernetes cluster

- with the Config Connector installed & configured
- with a stable Egress IP/CIDR range to allow access to the Cloud SQL instance (see further down at A Cloud NAT service)

In this example we went standard GKE cluster with the Config Connector pre-installed.

It is recommended to install the latest stable version of the Operator (1.71.0 is known to work with this specific use case).

```
GCP_PROJECT='<GCP project ID>'
LABELS='<label1=value1,label2=value2,...>'
CLUSTER_NAME='<GKE cluster name>'

# The Google Cloud Service Account to be used by the Config Connector
```

```
SA_NAME="${CLUSTER_NAME}-sa"

# The cluster's node count
# We suggest to start at 6 nodes to host all the TAP systems and to ensure
# the (automatically provisioned and managed) control plane is also scaled
# accordingly.
NODE_COUNT=6

# The namespace you want to deploy the Config Connector / service instance
# objects into
SI_NAMESPACE="service-instances"

# In this example we deploy a zonal cluster, thus you need to provide the
# zone you want your cluster to land in
ZONE='europe-west6-b'

# For Cloud NAT we need to provide the region we want to deploy the router
# to, this needs to be the region the zonal cluster resides in
REGION='europe-west6'

# Will be used for the name of the Cloud NAT router and the NAT config we
# deploy on it
NAT_NAME="${REGION}-nat"

gcloud container --project "${GCP_PROJECT}" \
    clusters create "${CLUSTER_NAME}" \
    --zone "${ZONE}" \
    --release-channel "regular" \
    --machine-type "e2-standard-4" \
    --disk-type "pd-standard" \
    --disk-size "70" \
    --metadata disable-legacy-endpoints=true \
    --num-nodes "${NODE_COUNT}" \
    --node-labels "${LABELS}" \
    --logging=SYSTEM \
    --monitoring=SYSTEM \
    --enable-ip-alias \
    --enable-network-policy \
    --addons ConfigConnector,HorizontalPodAutoscaling,HttpLoadBalancing,GcePersistentD
iskCsiDriver \
    --workload-pool="${GCP_PROJECT}.svc.id.goog" \
    --labels "${LABELS}"

gcloud iam service-accounts create \
    "${SA_NAME}" \
    --description "${LABELS}"

gcloud projects add-iam-policy-binding "${GCP_PROJECT}" \
    --member="serviceAccount:${SA_NAME}@${GCP_PROJECT}.iam.gserviceaccount.com" \
    --role="roles/editor"

gcloud iam service-accounts add-iam-policy-binding \
    "${SA_NAME}@${GCP_PROJECT}.iam.gserviceaccount.com" \
    --member="serviceAccount:${GCP_PROJECT}.svc.id.goog[cnrm-system/cnrm-controller-ma
nager]" \
    --role="roles/iam.workloadIdentityUser"
```

Configure a stable egress IP

By default egress traffic from pods will get their source IP translated to the node's public IP (SNAT) on the way out. Thus, when we need to configure allowed ingress networks for a Cloud SQL instance, we'd need to add each node of the cluster. Everytime the cluster scales or nodes get repaved, their public IP would change and we would need to make sure to keep the list of authorized networks up to date.

To make this easier we will: - turn off SNAT on the nodes, so egress traffic is not translated to the node's public IP - deploy a Cloud NAT service, which then handles the source IP translation and gives us a stable egress IP

Configure the `ip-masq-agent`

Each cluster comes with a `DaemonSet` `ip-masq-agent` in the `kube-system` namespace. By deploying a configuration for this service and restarting the `DaemonSet`, we can turn off SNAT for egress traffic.

```
cat <<'EOF' | kubectl -n kube-system create cm ip-masq-agent --from-file=config=/dev/s
tdin
nonMasqueradeCIDRs:
- 0.0.0.0/0
EOF

kubectl -n kube-system rollout restart daemonset ip-masq-agent
```

With this config none of the outbound traffic is translated to the node's public IP.

Note: You can also set specfic destination network CIDRs in `nonMasqueradeCIDRs` for which the SNAT on the nodes should be turned off. In that case, any traffic's source IP will still be translated to the node's public IP, except if the destination is explicitly configured in that list.

Set up a Cloud NAT service

After we've turned off SNAT on the nodes, we will employ a Cloud NAT service.

Conceptually this does the same thing as the SNAT on the nodes. However, the difference is, that we don't translate to a node's public IP address, but rather to a reserved IP address that is explicitly used by the Cloud NAT router. Therefore this IP is stable as long as this Cloud NAT router exists and all traffic originating from any pod, regardless which node it resides on, will get its source IP translated to that stable IP.

```
gcloud compute routers create "${NAT_NAME}-router" --region "${REGION}" --network defa
ult
gcloud compute routers nats create "${NAT_NAME}-config" \
    --router-region "${REGION}" \
    --router "${NAT_NAME}-router" \
    --auto-allocate-nat-external-ips \
    --nat-all-subnet-ip-ranges
```

A Tanzu Application Platform installation on the cluster (v1.2.0+).

Tanzu Application Platform (v1.2.0 or newer) and Cluster Essentials (v1.2.0 or newer) have to be installed on the kubernetes cluster.

Note: To check if you have an appropriate version, please run the following:

```
kubectl api-resources | grep secrettemplate
```

This command should return the `SecretTemplate` API. If it does not, ensure Cluster Essentials for VMware Tanzu (v1.2.0 or newer) is installed.

Configure the Config Connector

```
cat <<EOF | kubectl apply -f -
apiVersion: core.cnrm.cloud.google.com/v1beta1
kind: ConfigConnector
metadata:
  name: configconnector.core.cnrm.cloud.google.com
spec:
  mode: cluster
  googleServiceAccount: "${SA_NAME}@${GCP_PROJECT}.iam.gserviceaccount.com"
EOF

kubectl create namespace "${SI_NAMESPACE}"

kubectl annotate namespace "${SI_NAMESPACE}" "cnrm.cloud.google.com/project-id=${GCP_P
ROJECT}"

kubectl wait -n cnrm-system --for=condition=Ready pod --all

gcloud services enable serviceusage.googleapis.com
```

Get the NAT IP(s) for egress from the cluster

```
gcloud compute routers get-status "${NAT_NAME}-router" --region "${REGION}" --format=j
son \
  | jq -r '.result.natStatus[].autoAllocatedNatIps[]'
```

This IP(s) will later be used for allowing access to the CloudSQL instance from the cluster.

Creating Google CloudSQL Instances manually using kubectl (experimental)

Note

: This document is for users who are looking to understand the underlying APIs involved in making a bindable service instance using `SQLInstance`, `SQLDatabase`, `SQLUser` and `SecretTemplate` resources. For a simpler user experience, the alternative Creating an CloudSQL service instance through a Carvel Package topic is recommended.

Prerequisite

Meet the prerequisites and keep the following information to hand:

- `NAT-IP` - the cluster's egress NAT IP

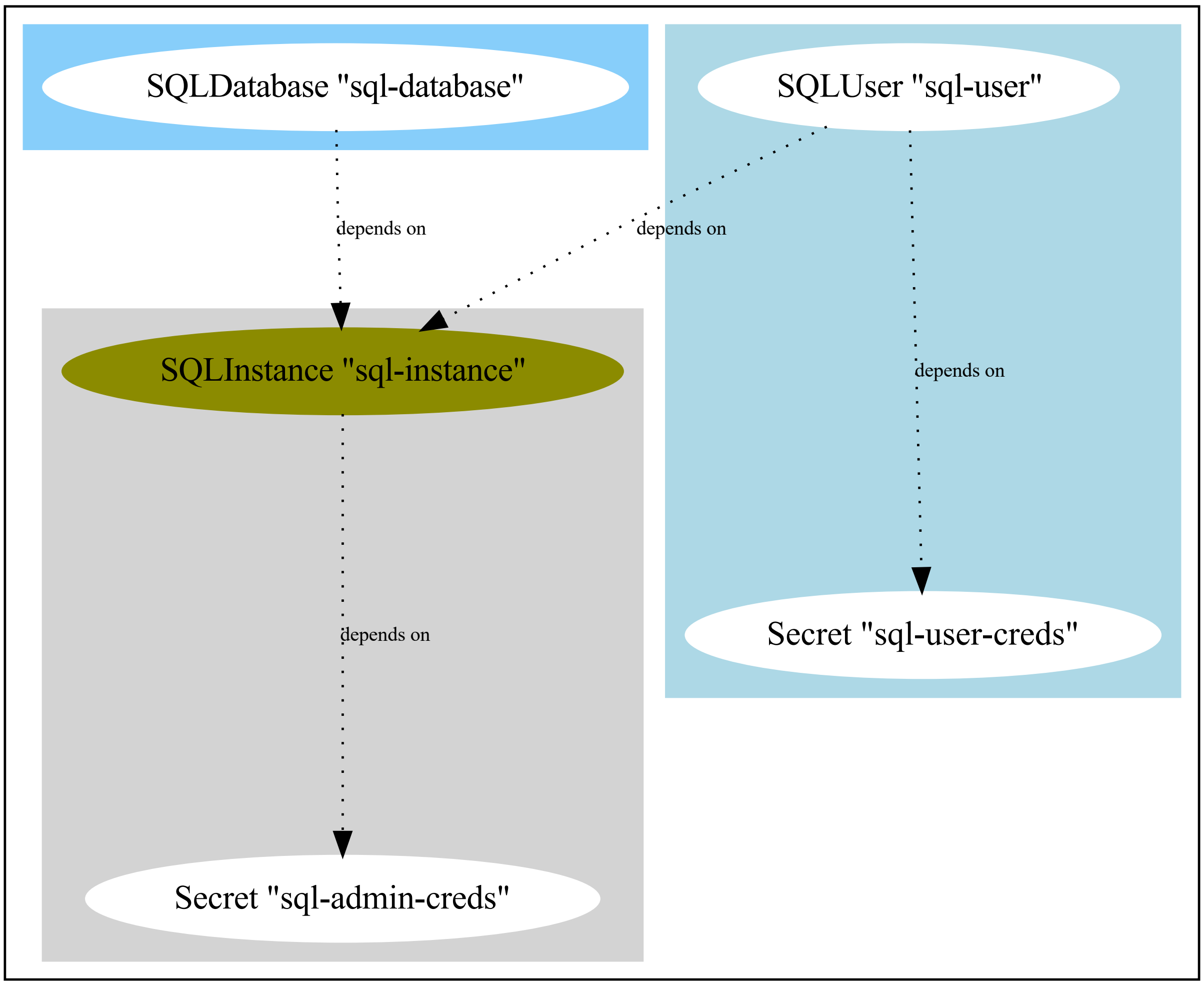
Create a CloudSQL service instance by using kubectl

At a minimum, a useable database instance consists of a `SQLInstance`, a `SQLDatabase`, and a `SQLUser`.

Realistically, in addition to that we will also want another set of `Secrets`:

- one `Secret` per `SQLInstance` to hold the password for the instance's admin role
- one `Secret` per `SQLUser` to hold that user's password

In the simplest case, with one `SQLInstance`, one `SQLDatabase`, and one `SQLUser`, we need to manage the following set of interrelated resources:



Create the `Secrets` for the Database admin & user

First we need to ensure that the `Secrets` which hold the admin's and user's password exist, so we can reference them in the `SQLInstance` and `SQLUser` objects.

Those secrets can be created by any means. In this guide will leverage the `Password` API from Carvel's secretgen controller, which will create the `Secrets` for us. However, any other mechanism to manage those secrets works too.

```
kind: List
apiVersion: v1
items:
- kind: Password
  apiVersion: secretgen.k14s.io/v1alpha1
  metadata:
    name: sql-admin-creds
    namespace: service-instances
  spec: &passwordSpec
    length: 64
    secretTemplate:
      type: Opaque
      stringData:
        password: $(value)
- kind: Password
  apiVersion: secretgen.k14s.io/v1alpha1
  metadata:
    name: sql-user-creds
    namespace: service-instances
  spec: *passwordSpec
```

Applying this will create two `Passwords` which in turn will have two `Secrets` created:

```
kubectl -n service-instances get passwords,secrets sql-user-creds sql-admin-creds
```

```
NAME                                           DESCRIPTION           AGE
password.secretgen.k14s.io/sql-user-creds     Reconcile succeeded   4m41s
password.secretgen.k14s.io/sql-admin-creds    Reconcile succeeded   4m41s

NAME                        TYPE      DATA   AGE
secret/sql-user-creds      Opaque    1      4m41s
secret/sql-admin-creds     Opaque    1      4m41s
```

Create a usable postgres database

Now we can reference those two secrets and use the Config Connector APIs to create our database objects:

Note

: You need to allow access from the Kubernetes cluster's NAT IP. You can get the NAT IP via the command described in the prerequisites. This NAT IP then needs to be used in the `SQLInstance`'s `spec.settings.ipConfiguration.authorizedNetworks`.

```
apiVersion: sql.cnrm.cloud.google.com/v1beta1
kind: SQLInstance
metadata:
  name: sql-instance
  namespace: service-instances
spec:
  databaseVersion: POSTGRES_14
  #! If you have deployed your cluster into a different region, you might want
  #! to change this and deploy the SQLInstance into the same region as the
  #! cluster, to avoid traffic going across regions.
  region: europe-west6
  rootPassword:
    valueFrom:
      secretKeyRef:
        key: password
        name: sql-admin-creds
  settings:
    tier: db-g1-small
    ipConfiguration:
      authorizedNetworks:
      - name: cluster-NAT-IP
        #! Update this value with your NAT IP address in CIDR notation (e.g. 8.8.8.8/3
2). See above.
        value: <NAT-IP>
      ipv4Enabled: true
---
apiVersion: sql.cnrm.cloud.google.com/v1beta1
kind: SQLDatabase
metadata:
  name: sql-database
  namespace: service-instances
spec:
  charset: UTF8
  collation: en_US.UTF8
  instanceRef:
    name: sql-instance
---
apiVersion: sql.cnrm.cloud.google.com/v1beta1
kind: SQLUser
metadata:
  name: sql-user
  namespace: service-instances
spec:
  instanceRef:
    name: sql-instance
  password:
    valueFrom:
      secretKeyRef:
        key: password
        name: sql-user-creds
```

Once those objects are committed to the Kubernetes API, the Config Connector will cause the creation of those resources on GCP. This will take a short amount of time.

The three resources report their status and potential problems/errors back. If all goes well we should see all of those resources as “Ready” & “UpToDate” after a couple of minutes.

```
# kubectl -n service-instances get sqlinstance,sqldatabase,sqluser
NAME                                                    AGE     READY   STATUS     STATUS
 AGE
sqlinstance.sql.cnrm.cloud.google.com/sql-instance     3d20h   True    UpToDate   3d20h

NAME                                                    AGE     READY   STATUS     STATUS
 AGE
sqldatabase.sql.cnrm.cloud.google.com/sql-database     3d20h   True    UpToDate   3d20h

NAME                                            AGE     READY   STATUS     STATUS  AGE
sqluser.sql.cnrm.cloud.google.com/sql-user     3d20h   True    UpToDate   3d20h
```

You can also see this Cloud SQL instance in the Google Cloud Console.

> **Note**
>
> : Cloud SQL does not allow you to reuse the name of a deleted instance for a week. If you try to create a new `SQLInstance` with a name you have already used previously, you will see an error like
>
> > **Note**
> >
> > [...] When you delete an instance, you can't reuse the name of the deleted instance until one week from the deletion date. [...]
>
> You can use a different name for the `SQLInstance`; make sure to use replace that name in all examples going forward.

Create a Binding Specification compatible Secret for the database

As pointed out, none of the created objects are compatible with the Service Binding Specification. To help with that, we can create a secret which holds the data we need to know to connect to and use the Cloud SQL instance and which allows the platform to discover the fact that this instance can be "claimed" and "bound" to application workloads.

For this to be an automated process, we can use the `SecretTemplate` API of the secretgen controller. The secretgen controller needs to be able to read the resources created, thus we also need to deploy some RBAC rules to allow for that:

```
apiVersion: secretgen.carvel.dev/v1alpha1
kind: SecretTemplate
metadata:
  name: sql-instance-claimable
  namespace: service-instances
spec:
  inputResources:
  - name: sqlInstance
    ref:
      apiVersion: sql.cnrm.cloud.google.com/v1beta1
      kind: SQLInstance
      name: sql-instance
  - name: sqlDatabase
    ref:
      apiVersion: sql.cnrm.cloud.google.com/v1beta1
      kind: SQLDatabase
      name: sql-database
  - name: sqlUser
    ref:
      apiVersion: sql.cnrm.cloud.google.com/v1beta1
      kind: SQLUser
      name: sql-user
  - name: sqlUserSecret
    ref:
      apiVersion: v1
      kind: Secret
      name: $(.sqlUser.spec.password.valueFrom.secretKeyRef.name)
  serviceAccountName: sql-objects-reader
  template:
    data:
      password: $(.sqlUserSecret.data.password)
    metadata:
      labels:
        app.kubernetes.io/component: cloudsql-postgres
        app.kubernetes.io/instance: "$(.sqlInstance.metadata.name)"
        services.apps.tanzu.vmware.com/class: cloudsql-postgres
    stringData:
      database: $(.sqlDatabase.metadata.name)
      host: $(.sqlInstance.status.publicIpAddress)
      port: "5432"
      type: postgresql
      username: $(.sqlUser.metadata.name)
---
apiVersion: v1
kind: ServiceAccount
metadata:
  name: sql-objects-reader
  namespace: service-instances
---
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: sql-objects-reader
  namespace: service-instances
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: Role
  name: sql-objects-reader
subjects:
- kind: ServiceAccount
  name: sql-objects-reader
---
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: sql-objects-reader
  namespace: service-instances
rules:
- apiGroups:     [ "" ]
  resources:     [ "secrets" ]
  verbs:         &objReaderVerbs [ "get", "list", "watch" ]
  resourceNames: [ "sql-user-creds", "sql-admin-creds" ]
- apiGroups:     [ "sql.cnrm.cloud.google.com" ]
  resources:     [ "sqlinstances", "sqldatabases", "sqlusers" ]
  verbs:         *objReaderVerbs
  resourceNames: [ "sql-instance", "sql-database", "sql-user" ]
```

Verify

Find the name of the secret produced by reading the status of `SecretTemplate`. To do so, run:

```
kubectl get secrettemplate -n service-instances sql-instance-claimable -o jsonpath="{.
status.secret.name}"
```

Delete a CloudSQL service instance

Delete an CloudSQL service instance and all additional and related objects by running:

```
kubectl -n service-instances delete \
  sqlinstance/sql-instance \
  sqldatabase/sql-database \
  sqluser/sql-user \
  secrettemplate/sql-instance-claimable \
  password/sql-admin-creds \
  password/sql-user-creds \
  serviceaccount/sql-objects-reader \
  rolebinding/sql-objects-reader \
  roles/sql-objects-reader
```

Summary and Next Steps

You have learned how to use Carvel's `SecretTemplate` API to construct a secret that is compatible with the binding specification in order to create an Google CloudSQL service instance.

Now that you have this available in the cluster, you can learn how to make use of it by continuing where you left off in Consuming Google Cloud SQL on Tanzu Application Platform (TAP) with Config Connector.

Creating Google CloudSQL instances by using a Carvel package (experimental)

This topic describes how to create, update, and delete CloudSQL service instances using a Carvel package. For a more detailed and low-level alternative procedure, see Creating Service Instances that are compatible with Tanzu Application Platform.

Prerequisite

Meet the prerequisites and keep the following information to hand:

- `NAT-IP` - the cluster's egress NAT IP

The Package Repository and service instance Package Bundles for this guide can be found in the Reference Service Packages GitHub repository.

Create an CloudSQL service instance using a Carvel package

Follow the steps in the following procedures.

Add a reference package repository to the cluster

To add a reference package repository to the cluster:

1. Use the Tanzu CLI to add the new Service Reference packages repository:

```
tanzu package repository add tap-reference-service-packages \
  --url ghcr.io/vmware-tanzu/tanzu-application-platform-reference-packages/tap-
service-reference-package-repo:0.0.2 \
  -n tanzu-package-repo-global
```

2. Create a `ServiceAccount` that is used to provision `PackageInstall` resources by using the following example. The namespace of this `ServiceAccount` must match the namespace of the `tanzu package install` command in the next step.

```
kubectl apply -f - <<'EOF'
apiVersion: v1
kind: ServiceAccount
metadata:
  name: cloudsql-install
  namespace: service-instances
---
kind: Role
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: cloudsql-install
  namespace: service-instances
rules:
- apiGroups: ["sql.cnrm.cloud.google.com"]
  resources: ["sqlinstances","sqldatabases","sqlusers"]
  verbs:     ["*"]
- apiGroups: ["secretgen.carvel.dev", "secretgen.k14s.io"]
  resources: ["secrettemplates","passwords"]
  verbs:     ["*"]
- apiGroups: [""]
  resources: ["serviceaccounts","configmaps"]
  verbs:     ["*"]
- apiGroups: ["rbac.authorization.k8s.io"]
  resources: ["roles","rolebindings"]
  verbs:     ["*"]
---
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: cloudsql-install
  namespace: service-instances
subjects:
- kind: ServiceAccount
  name: cloudsql-install
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: Role
  name: cloudsql-install
EOF
```

Create a CloudSQL service instance through the Tanzu CLI

1. Create a file holding the configuration of the CloudSQL service instance:

```
cat <<'EOF' > demo-pg-instance-values.yml
---
name: demo-pg-instance
namespace: service-instances
```

```
allowedNetworks:
- name: service-instances-cluster-snat
  #! replace that with the cluster's egress IP, see NAT-IP in Prerequisite
  value: 34.65.178.24/32
EOF
```

> **Note**
>
> : To understand which settings are available for this package you can run:
>
> ```
> tanzu package available get \
>   --values-schema \
>   psql.google.references.services.apps.tanzu.vmware.com/0.0.1-alph
> a
> ```
>
> This shows a list of all configuration options you can use in the `demo-pg-instance-values.yml` file.
>
> : By default the package will create a claimable `Secret` which is labeled with `services.apps.tanzu.vmware.com/class: cloudsql-postgres`. While you can overwrite that by setting the `serviceInstanceLabels` setting above, you don't have to do that and it will still be aligned with the `ClusterInstanceClass` we will set up later.

2. Use the Tanzu CLI to install an instance of the reference service instance Package.

```
tanzu package install demo-pg-instance \
   --package-name psql.google.references.services.apps.tanzu.vmware.com \
   --version 0.0.1-alpha \
   --namespace service-instances \
   --service-account-name cloudsql-install \
   --values-file demo-pg-instance-values.yml \
   --wait
```

You can install the `psql.google.references.services.apps.tanzu.vmware.com` package multiple times to produce multiple CloudSQL Service instances. For that you need to prepare a separate `<INSTANCE-NAME>-values.yml` and then install the package with a different name and the above mentioned separate data values file for each CloudSQL service instance.

Verify

1. Verify the creation status for the CloudSQL instance by inspecting the conditions in the Kubernetes API. To do so, run:

```
kubectl get sqlinstance demo-pg-instance -n service-instances -o yaml
```

2. After some time has passed, sometimes up to 20 minutes, you are able to find the binding-compliant secret produced by `PackageInstall`. To do so, run:

```
kubectl get secrettemplate demo-pg-instance -n service-instances -o jsonpath="{
.status.secret.name}"
```

Delete a CloudSQL service instance

To delete the CloudSQL service instance run:

```
tanzu package installed delete demo-pg-instance -n service-instances
```

Summary

You have learned how to use Carvel's `Package` and `PackageInstall` APIs to create a CloudSQL service instance. If you want to learn more about the pieces that comprise this service instance package, see Creating Google CloudSQL Instances manually using kubectl.

Now that you have this available in the cluster, you can learn how to make use of it by continuing where you left off in [Consuming Google Cloud SQL on Tanzu Application Platform (TAP) with Config Connector][create-class].

Consuming GCP CloudSQL on Tanzu Application Platform with Crossplane

Introduction

This topic demonstrates how to use Services Toolkit to allow Tanzu Application Platform workloads to consume GCP CloudSQL PostgreSQL databases. This particular guide makes use of Crossplane to manage CloudSQL instances in GCP. As such, it can be thought of as an alternative approach to Consuming Google Cloud SQL on Tanzu Application Platform (TAP) with Config Connector to achieve the same outcomes.

Prerequisites

Meet these prerequisites:

- Create a Kubernetes cluster that supports running both Tanzu Application Platform and Crossplane
- Install Tanzu Application Platform (v1.2+) on the Kubernetes cluster
- Install gcloud CLI

Install Crossplane

Note: For the latest steps for installing Crossplane, see these instructions. For the instructions in this topic, it is important to enable support for external secret stores in Crossplane. This is currently an Alpha feature. As such, you will have to explicitly set command line flag `--enable-external-secret-stores` when starting the Crossplane controller.

Run the following commands to install Crossplane to your existing Kubernetes cluster:

```
kubectl create namespace crossplane-system
```

```
helm repo add crossplane-stable https://charts.crossplane.io/stable
helm repo update

helm install crossplane --namespace crossplane-system crossplane-stable/crossplane \
  --set 'args={--enable-external-secret-stores}'
```

For this topic, you do not need to install the Crossplane CLI or any additional configuration package.

Install GCP Provider for Crossplane

To install the GCP Provider for Crossplane, run:

```
kubectl apply -f -<<EOF
---
apiVersion: pkg.crossplane.io/v1
kind: Provider
metadata:
  name: crossplane-provider-gcp
spec:
  package: crossplane/provider-gcp:v0.21.0
EOF
```

After you have installed the provider, you see a new `cloudsqlinstances.database.gcp.crossplane.io` API resource available in your Kubernetes cluster. See the health of the installed provider by running:

```
kubectl get provider.pkg.crossplane.io crossplane-provider-gcp
```

Configure GCP Provider

This section creates a new GCP Service Account and gives it permissions to manage CloudSQL databases which are necessary to use Crossplane to manage CloudSQL instances.

1. Create a new GCP ServiceAccount, give it `Cloud SQL Admin` and create a key file:

   ```
   PROJECT_ID=<GCP Project ID>
   SA_NAME=crossplane-cloudsql

   gcloud iam service-accounts create "${SA_NAME}" --project "${PROJECT_ID}"
   gcloud projects add-iam-policy-binding "${PROJECT_ID}" \
     --role="roles/cloudsql.admin" \
     --member "serviceAccount:${SA_NAME}@${PROJECT_ID}.iam.gserviceaccount.com"
   gcloud iam service-accounts keys create creds.json --project "${PROJECT_ID}" --
   iam-account "${SA_NAME}@${PROJECT_ID}.iam.gserviceaccount.com"
   ```

2. Create a new secret from the key file by running:

   ```
   kubectl create secret generic gcp-creds -n crossplane-system --from-file=creds=
   ./creds.json
   ```

3. Delete the key file by running:

   ```
   rm -f creds.json
   ```

4. Configure the GCP provider to use the newly created secret by running:

```
kubectl apply -f -<<EOF
apiVersion: gcp.crossplane.io/v1beta1
kind: ProviderConfig
metadata:
  name: default
spec:
  projectID: ${PROJECT_ID}
  credentials:
    source: Secret
    secretRef:
      namespace: crossplane-system
      name: gcp-creds
      key: creds
EOF
```

Define Composite Resource Types

Now that the GCP provider for Crossplane has been installed and configured, create a new `CompositeResourceDefinition` (XRD) and corresponding `Composition` representing individual instances of CloudSQL Postgresql. For more information about these concepts see the Crossplane Composition documentation.

Note: Instead of creating your own custom XRD and Composition as shown below, you can also install an existing Crossplane configuration package for GCP that includes pre-configured XRDs and compositions for CloudSQL. The primary reason for creating a new XRD and composition from scratch is to make sure the connection secrets for newly provisioned CloudSQL Postgresql instances support the Service Binding Specification for Kubernetes and automatic Spring Boot configuration using Spring Cloud Bindings.

1. Create a new XRD by running:

```
kubectl apply -f -<<EOF
---
apiVersion: apiextensions.crossplane.io/v1
kind: CompositeResourceDefinition
metadata:
  name: xpostgresqlinstances.bindable.database.example.org
spec:
  claimNames:
    kind: PostgreSQLInstance
    plural: postgresqlinstances
  connectionSecretKeys:
  - type
  - provider
  - host
  - port
  - database
  - username
  - password
  group: bindable.database.example.org
  names:
    kind: XPostgreSQLInstance
    plural: xpostgresqlinstances
  versions:
  - name: v1alpha1
```

```
    referenceable: true
    schema:
      openAPIV3Schema:
        properties:
          spec:
            properties:
              parameters:
                properties:
                  storageGB:
                    type: integer
                required:
                - storageGB
                type: object
            required:
            - parameters
            type: object
        type: object
    served: true
EOF
```

After the newly created XRD has been successfully reconciled, there are two new API resources available in your Kubernetes cluster, `xpostgresqlinstances.bindable.database.example.org` and `postgresqlinstances.bindable.database.example.org`. The XRD created is agnostic to the underlying cloud managed service, so could also be fulfilled by a Composition that makes use of AWS RDS Postgresql or Azure Database for PostgreSQL.

2. Create a corresponding composition (not in a production environment) by running:

```
kubectl apply -f -<<EOF
---
apiVersion: apiextensions.crossplane.io/v1
kind: Composition
metadata:
  labels:
    provider: gcp
  name: xpostgresqlinstances.bindable.gcp.database.example.org
spec:
  compositeTypeRef:
    apiVersion: bindable.database.example.org/v1alpha1
    kind: XPostgreSQLInstance
  publishConnectionDetailsWithStoreConfigRef:
    name: default
  resources:
  - base:
      apiVersion: database.gcp.crossplane.io/v1beta1
      kind: CloudSQLInstance
      spec:
        forProvider:
          databaseVersion: POSTGRES_14
          region: us-central1
          settings:
            dataDiskType: PD_SSD
            ipConfiguration:
              authorizedNetworks:
              - value: 0.0.0.0/0 # not recommended for production deployments!
              ipv4Enabled: true
            tier: db-custom-1-3840
        writeConnectionSecretToRef:
          namespace: crossplane-system
    connectionDetails:
    - name: type
      value: postgresql
    - name: provider
      value: gcp
    - name: database
      value: postgres
    - fromConnectionSecretKey: username
    - fromConnectionSecretKey: password
    - name: host
      fromConnectionSecretKey: endpoint
    - name: port
      type: FromValue
      value: "5432"
    name: cloudsqlinstance
    patches:
    - fromFieldPath: metadata.uid
      toFieldPath: spec.writeConnectionSecretToRef.name
      transforms:
      - string:
          fmt: '%s-postgresql'
          type: Format
        type: string
      type: FromCompositeFieldPath
    - fromFieldPath: spec.parameters.storageGB
      toFieldPath: spec.forProvider.settings.dataDiskSizeGb
      type: FromCompositeFieldPath
EOF
```

The composition defined above makes sure that all CloudSQL Postgresql instances are placed in the `us-central1` region. This composition fulfils the XRD previously created by creating GCP CloudSQL databases.

Warning: The authorized network CIDR `0.0.0.0/0` provided above, will allow access to the Cloud SQL from any IP and is not recommended in a production environment.

Create an Instance Class

In order to make instances of a service easily discoverable and claimable by application operators, the role of the service operator creates a `ClusterInstanceClass`. In this particular example, the class states that claimable instances of CloudSQL Postgresql are represented by secret objects of type `connection.crossplane.io/v1alpha1` with label `services.apps.tanzu.vmware.com/class` set to `cloudsql-postgres`:

```
kubectl apply -f -<<EOF
---
apiVersion: services.apps.tanzu.vmware.com/v1alpha1
kind: ClusterInstanceClass
metadata:
  name: cloudsql-postgres
spec:
  description:
    short: GCP CloudSQL Postgresql database instances
  pool:
    kind: Secret
    labelSelector:
      matchLabels:
        services.apps.tanzu.vmware.com/class: cloudsql-postgres
    fieldSelector: type=connection.crossplane.io/v1alpha1
EOF
```

In addition, you need to grant sufficient RBAC permissions to Services Toolkit to be able to read the secrets specified by the class.

```
kubectl apply -f -<<EOF
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: stk-secret-reader
  labels:
    servicebinding.io/controller: "true"
rules:
- apiGroups:
  - ""
  resources:
  - secrets
  verbs:
  - get
  - list
  - watch
EOF
```

Provision GCP CloudSQL Postgresql Instance

Playing the role of the Service Operator, you now provision an instance of GCP CloudSQL Postgresql using the `postgresqlinstances.bindable.database.example.org` API managed by the XRD you previously created. Note that `.spec.publishConnectionDetailsTo` provides Crossplane with the name and a label for the secret that is being used to store the connection details for the newly created database. You can see that the label specified here matches the label selector defined on the `ClusterInstanceClass` you created in the previous step.

Run the following command:

```
kubectl apply -f -<<EOF
---
apiVersion: bindable.database.example.org/v1alpha1
kind: PostgreSQLInstance
metadata:
  name: cloudsql-postgres-db
  namespace: default
spec:
  parameters:
    storageGB: 20
  compositionSelector:
    matchLabels:
      provider: gcp
  publishConnectionDetailsTo:
    name: cloudsql-postgres-db
    metadata:
      labels:
        services.apps.tanzu.vmware.com/class: cloudsql-postgres
EOF
```

Running this command will cause the creation of a CloudSQL database instance in your GCP account. You can use the gcloud CLI to verify this:

```
gcloud sql instances list
```

After the instance has been successfully created in GCP, the status of the newly created `PostgreSQLInstance` resource should show `READY=True`. This might take a few minutes. You can wait for this by running:

```
kubectl wait --for=condition=Ready=true postgresqlinstances.bindable.database.example.
org cloudsql-postgres-db --timeout=10m
```

As soon as the CloudSQL Postgresql instance is ready, it is claimable by the role of the application operator as shown in the next section.

Note: There is currently a bug in Crossplane 1.7.2 onwards with the `--enable-external-secret-stores` feature gate enabled where the controller will fail to clean up a local secret created by the field `.spec.publishConnectionDetailsTo` after the deletion of the claim. A workaround is to temporarily give the crossplane controller the necessary i.e. permissions:

```
kubectl create clusterrole crossplane-cleaner --verb=delete --resource=secrets
kubectl create clusterrolebinding crossplane-cleaner --clusterrole=crossplane-cleaner
--serviceaccount=crossplane-system:crossplane
```

Claim the CloudSQL Postgresql instance and connect to it from the Tanzu Application Platform Workload

Thanks to the previously created `ClusterInstanceClass`, secrets representing CloudSQL Postgresql instances can now be discovered and claimed by application operators through the Tanzu CLI as shown below.

1. Show available classes of service instances by running:

   ```
   tanzu service classes list

     NAME                 DESCRIPTION
     cloudsql-postgres    GCP CloudSQL Postgresql database instances
   ```

2. Show claimable instances belonging to the CloudSQL Postgresql class by running:

   ```
   tanzu services claimable list --class cloudsql-postgres

     NAME                    NAMESPACE  API KIND  API GROUP/VERSION
     cloudsql-postgres-db    default    Secret    v1
   ```

3. Create a claim for the discovered instance by running:

   ```
   tanzu service claim create cloudsql-claim \
   ```

```
  --resource-name cloudsql-postgres-db \
  --resource-kind Secret \
  --resource-api-version v1
```

4. Obtain the claim reference by running:

   ```
   tanzu service claim list -o wide
   ```

   Expect to see the following output:

   ```
   NAME                      READY  REASON  CLAIM REF
   cloudsql-claim            True           services.apps.tanzu.vmware.com/v1alpha1
   :ResourceClaim:cloudsql-claim
   ```

5. Create an application workload that consumes the claimed CloudSQL Postgresql database by running:

   Example:

   ```
   tanzu apps workload create my-workload \
     --git-repo https://github.com/sample-accelerators/spring-petclinic \
     --git-branch main \
     --git-tag tap-1.2 \
     --type web \
     --label app.kubernetes.io/part-of=spring-petclinic \
     --annotation autoscaling.knative.dev/minScale=1 \
     --env SPRING_PROFILES_ACTIVE=postgres \
     --service-ref db=services.apps.tanzu.vmware.com/v1alpha1:ResourceClaim:clouds
   ql-claim
   ```

   Note that `--service-ref` is being set to the claim reference obtained previously.

Direct Secret References

This use case leverages direct references to Kubernetes `Secret` resources to enable developers to connect their application workloads to almost any backing service, including backing services that:

- are running external to Tanzu Application Platform
- do not adhere to the ProvisionedService of the Service Binding Specification for Kubernetes in GitHub.

The following example demonstrates a procedure to bind a new application on Tanzu Application Platform to an existing PostgreSQL database that exists in Azure.

Depending on your Kubernetes distribution and the backing Service you are hoping to connect to your Tanzu Application Platform workloads, there could be extra work to set up networking between the workload and the service endpoint and to obtain the credentials for the backing service. This example assumes the credentials are available and networking has been set up.

1. Create a Kubernetes secret resource similar to the following example:

   ```
   # external-azure-db-binding-compatible.yaml
   ---
   apiVersion: v1
   kind: Secret
   ```

```
metadata:
  name: external-azure-db-binding-compatible
type: Opaque
stringData:
  type: postgresql
  provider: azure
  host: EXAMPLE.DATABASE.AZURE.COM
  port: "5432"
  database: "EXAMPLE-DB-NAME"
  username: "USER@EXAMPLE"
  password: "PASSWORD"
```

Kubernetes secret resources must abide by the Well-known Secret Entries specifications in GitHub. If you are planning to bind this secret to a Spring-based application workload and want to take advantage of the auto-wiring feature, this secret must also contain the properties required by Spring Cloud Bindings in GitHub.

2. Apply the YAML file by running:

```
kubectl apply -f external-azure-db-binding-compatible.yaml
```

3. Grant sufficient RBAC permissions to Services Toolkit to be able to read the secrets specified by the class:

```
# stk-secret-reader.yaml
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: stk-secret-reader
  labels:
    servicebinding.io/controller: "true"
rules:
- apiGroups:
  - ""
  resources:
  - secrets
  verbs:
  - get
  - list
  - watch
```

4. Apply your changes by running:

```
kubectl apply -f stk-secret-reader.yaml
```

5. Create a claim for the newly created secret by running:

```
tanzu service claim create external-azure-db-claim \
  --resource-name external-azure-db-binding-compatible \
  --resource-kind Secret \
  --resource-api-version v1
```

6. Obtain the claim reference of the claim by running:

```
tanzu service claim list -o wide
```

Expect to see the following output:

```
NAME                       READY  REASON   CLAIM REF
external-azure-db-claim    True            services.apps.tanzu.vmware.com/v1alpha1
:ResourceClaim:external-azure-db-claim
```

7. Create an application workload by running a command similar to the following example:

   Example:

```
tanzu apps workload create WORKLOAD-NAME \
  --git-repo https://github.com/sample-accelerators/spring-petclinic \
  --git-branch main \
  --git-tag tap-1.2 \
  --type web \
  --label app.kubernetes.io/part-of=spring-petclinic \
  --annotation autoscaling.knative.dev/minScale=1 \
  --env SPRING_PROFILES_ACTIVE=postgres \
  --service-ref db=REFERENCE
```

   Where:

   - `WORKLOAD-NAME` is the name of the Application Workload. For example, `pet-clinic`.
   - `REFERENCE` is the value of the `CLAIM REF` for the newly created claim in the output of the last step.

Dedicated Service Clusters (using experimental Projection and Replication APIs)

Caution: This use case leverages experimental APIs. Do not use it in a production environment.

This use case is currently not supported on Kubernetes v1.24 or later.

This use case leverages the experimental API Projection and Resource Replication APIs to separate application workloads and service instances onto separate Kubernetes clusters. There are several reasons for it:

- **Dedicated cluster requirements for workload or service clusters:** Service clusters, for example, might need access to more powerful SSDs.
- **Different cluster life cycle management:** Upgrades to service clusters can occur more cautiously.
- **Unique compliance requirements:** Data is stored on a service cluster, which might have different compliance needs.
- **Separation of permissions and access:** Application teams can only access the clusters where their applications are running.

The benefits of implementing this use case include:

- The experience for application developers and application operators working on their Tanzu Application Platform cluster is unaltered.
- All complexity in the setup and management of backing infrastructure is abstracted away

from application developers, which gives them more time to focus on developing their applications.

For information about network requirements and possible topology setups, see Topology.

Prerequisites

Meet the following prerequisites before completing this use case walkthrough:

- You have access to a cluster with Tanzu Application Platform installed, henceforth called the application workload cluster.
- You have access to a second, separate cluster with the Services Toolkit package installed, henceforth called the service cluster.
- You downloaded and installed the `tanzu` CLI and the corresponding plug-ins.
- You downloaded and installed the experimental `kubectl-scp` plug-in. For instructions, see Install the kubectl-scp plug-in.
- You set up the `default` namespace on the application workload cluster as your developer namespace to use installed packages. For more information, see Set up developer namespaces to use installed packages.
- The application workload cluster can pull source code from GitHub.
- The service cluster can pull the images required by the RabbitMQ Cluster Kubernetes Operator.
- The service cluster can create LoadBalancer services.
- If you have previously installed the RabbitMQ cluster operator to the application workload cluster as part of Getting started with Tanzu Application Platform, uninstall it from that cluster. This is necessary because of a limitation of the experimental API Projection APIs. To delete the operator, run:

  ```
  kapp delete -a rmq-operator -y
  ```

Walkthrough

Follow these steps to bind an application to a service instance running on a different Kubernetes cluster:

1. As the service operator, link the workload cluster and service cluster together by using the `kubectl scp` plug-in. To do so, run:

   ```
   kubectl scp link --workload-kubeconfig-context=WORKLOAD-CONTEXT --service-kubec
   onfig-context=SERVICE-CONTEXT
   ```

   Where `WORKLOAD-CONTEXT` is your workload context and `SERVICE-CONTEXT` is your service context.

2. Install the RabbitMQ Kubernetes operator in the services cluster by running:

   ```
   kapp -y deploy --app rmq-operator \
   ```

```
 --file https://raw.githubusercontent.com/rabbitmq/cluster-operator/lb-binding/
hack/deploy.yml \
 --kubeconfig-context SERVICE-CONTEXT
```

Where `SERVICE-CONTEXT` is your service context.

This operator is installed in the service cluster, but `RabbitmqCluster` service instance life cycles (CRUD) can still be managed from the workload cluster. Use the exact `deploy.yml` specified in the command because this RabbitMQ operator deployment includes specific changes to enable cross-cluster service binding.

3. Verify that you installed the operator by running:

   ```
   kubectl --context SERVICE-CONTEXT get crds rabbitmqclusters.rabbitmq.com
   ```

   Where `SERVICE-CONTEXT` is your service context.

   The `rabbitmq.com/v1beta1` API group is available in the service cluster. The following steps federate the `rabbitmq.com/v1beta1` in the workload cluster. This occurs in two parts, projection and replication.

   - Projection applies to custom API groups.
   - Replication applies to core Kubernetes resources, such as secrets.

4. Create a `service-instance` namespace in both clusters. API projection occurs between clusters by using namespaces with the same name and that are said to have a quality of namespace sameness.

   For example:

   ```
   kubectl --context WORKLOAD-CONTEXT create namespace service-instances
   kubectl --context SERVICE-CONTEXT create namespace service-instances
   ```

   Where `WORKLOAD-CONTEXT` is your workload context and `SERVICE-CONTEXT` is your service context.

5. Use the `kubectl-scp` plug-in to federate by running:

   ```
   kubectl scp federate \
   --workload-kubeconfig-context=WORKLOAD-CONTEXT \
   --service-kubeconfig-context=SERVICE-CONTEXT \
   --namespace=service-instances \
   --api-group=rabbitmq.com \
   --api-version=v1beta1 \
   --api-resource=rabbitmqclusters
   ```

   Where `WORKLOAD-CONTEXT` is your workload context and `SERVICE-CONTEXT` is your service context.

6. After federation, verify the `rabbitmq.com/v1beta1` API is also available in the workload cluster by running:

   ```
   kubectl --context WORKLOAD-CONTEXT api-resources
   ```

   Where `WORKLOAD-CONTEXT` is your workload context

7. Advertise that the RabbitmqCluster API is available to developers by applying the following YAML to your workload cluster. Ensure the Tanzu CLI is configured to target the workload cluster for the rest of the steps.

   ```
   ---
   apiVersion: services.apps.tanzu.vmware.com/v1alpha1
   kind: ClusterInstanceClass
   metadata:
    name: rabbitmq
   spec:
    description:
      short: It's a RabbitMQ cluster!
    pool:
      kind: RabbitmqCluster
      group: rabbitmq.com
   ```

8. Discover the new service and provision an instance from the workload cluster by running:

   ```
   tanzu services classes list
   ```

   The following output appears:

   ```
   tanzu services classes list

   NAME      DESCRIPTION
   rabbitmq  It's a RabbitMQ cluster!
   ```

9. Provision a service instance on the Tanzu Application Platform cluster.

   For example:

   ```
   # rabbitmq-cluster.yaml
   ---
   apiVersion: rabbitmq.com/v1beta1
   kind: RabbitmqCluster
   metadata:
    name: projected-rmq
   spec:
    service:
      type: LoadBalancer
   ```

10. Apply the YAML file by running:

    ```
    kubectl --context WORKLOAD-CONTEXT -n service-instances apply -f rabbitmq-clust
    er.yaml
    ```

    Where `WORKLOAD-CONTEXT` is your workload context

11. Confirm that the RabbitmqCluster resource reconciles successfully from the workload cluster by running:

    ```
    kubectl --context WORKLOAD-CONTEXT -n service-instances get -f rabbitmq-cluster
    .yaml
    ```

    Where `WORKLOAD-CONTEXT` is your workload context

12. Verify that RabbitMQ pods are running in the service cluster, but not in the workload cluster, by running:

    ```
    kubectl --context WORKLOAD-CONTEXT -n service-instances get pods
    kubectl --context SERVICE-CONTEXT -n service-instances get pods
    ```

    Where `WORKLOAD-CONTEXT` is your workload context and `SERVICE-CONTEXT` is your service context.

13. Enable cross-namespace claims by creating a `ResourceClaimPolicy` on your workload cluster:

    ```
    # rabbitmq-cluster-policy.yaml
    ---
    apiVersion: services.apps.tanzu.vmware.com/v1alpha1
    kind: ResourceClaimPolicy
    metadata:
      name: rabbitmq-cluster-policy
      namespace: service-instances
    spec:
      consumingNamespaces:
      - default
      subject:
        group: rabbitmq.com
        kind: RabbitmqCluster
    ```

14. Apply the YAML file by running:

    ```
    kubectl --context WORKLOAD-CONTEXT apply -f rabbitmq-cluster-policy.yaml
    ```

    Where `WORKLOAD-CONTEXT` is your workload context

15. Create a claim for the projected service instance by running:

    ```
    tanzu service claim create projected-rmq-claim \
      --resource-name projected-rmq \
      --resource-kind RabbitmqCluster \
      --resource-api-version rabbitmq.com/v1beta1 \
      --resource-namespace service-instances \
      --namespace default
    ```

16. Create the application workload by running:

    ```
    tanzu apps workload create multi-cluster-binding-sample \
      --namespace default \
      --git-repo https://github.com/sample-accelerators/rabbitmq-sample \
      --git-branch main \
      --git-tag 0.0.1 \
      --type web \
      --label app.kubernetes.io/part-of=rabbitmq-sample \
      --annotation autoscaling.knative.dev/minScale=1 \
      --service-ref "rmq=services.apps.tanzu.vmware.com/v1alpha1:ResourceClaim:proj
    ected-rmq-claim"
    ```

17. Get the `web-app` URL by running:

```
tanzu apps workload get multi-cluster-binding-sample -n default
```

18. Visit the URL and refresh the page to confirm the app is running by viewing the new message IDs.

Consuming Cloud Services (AWS, Azure and GCP) on Tanzu Application Platform

This section of the documentation covers integrations of various Cloud Service Providers (AWS, Azure and GCP) into Tanzu Application Platform.

Consuming AWS RDS on Tanzu Application Platform

This section of the documentation covers integrations of AWS RDS into Tanzu Application Platform. Documentation is provided for both an integration using AWS Controllers for Kubernetes (ACK), as well as an integration using Crossplane.

Consuming AWS RDS on Tanzu Application Platform with AWS Controllers for Kubernetes (ACK)

This topic describes how to use Services Toolkit to allow Tanzu Application Platform workloads to consume AWS RDS PostgreSQL databases.

This topic makes use of AWS Controllers for Kubernetes (ACK) to manage RDS instances in AWS. As such, it is an alternative approach to using Crossplane to achieve the same outcomes.

Prerequisites

- Prerequisites
- Configure your AWS RDS environment

Create service instances that are compatible with Tanzu Application Platform

Installing the ACK service controller for RDS makes available new Kubernetes APIs for interacting with RDS resources from within the Tanzu Application Platform cluster.

```
$ kubectl api-resources --api-group rds.services.k8s.aws

NAME                         SHORTNAMES   APIVERSION                      NAMESPACED   K
IND
dbclusterparametergroups                  rds.services.k8s.aws/v1alpha1   true         D
BClusterParameterGroup
dbclusters                                rds.services.k8s.aws/v1alpha1   true         D
BCluster
dbinstances                               rds.services.k8s.aws/v1alpha1   true         D
BInstance
dbparametergroups                         rds.services.k8s.aws/v1alpha1   true         D
BParameterGroup
dbsubnetgroups                            rds.services.k8s.aws/v1alpha1   true         D
```

```
BSubnetGroup
globalclusters                              rds.services.k8s.aws/v1alpha1   true          G
lobalCluster
```

`DBInstance` is of most interest here because this is the primary API for creating RDS databases. However, there are two important obstacles with this API when considering compatibility with Tanzu Application Platform.

Obstacle 1: `DBInstance` does not adhere to the binding specification

`DBInstance` does not adhere to the Service Binding Specification for Kubernetes. Tanzu Application Platform uses this specification as a contract for ensuring compatibility between different parts of the system. Given that `DBInstance` does not adhere to the specification it means that, by default, it is not possible to claim and bind application workloads to `DBInstance` resources.

Obstacle 2: Creating a `DBInstance` resource on its own is not sufficient

Creating a `DBInstance` resource on its own might not always be enough to create a working, usable instance that can be connected to and utilized.

For example, `DBInstance` defines the field `.spec.masterUserPassword`, which must refer to a secret containing credentials for the instance. As such, the secret resource can be considered a dependent resource of `DBInstance`. Without both of these resources, it is not possible to properly configure the RDS instance as wanted. In many cases, a group of related resources must be created to create something usable.

Solutions

Tanzu Application Platform v1.2 and later enables solutions for both these obstacles.

For example, consider the first obstacle where `DBInstance` does not adhere to the Kubernetes binding specification. One solution is for the authors of the RDS ACK service controller to update the `DBInstance` API to make it adhere to the binding specification. However, this requires code changes to the operator itself, and the authors of the operator might choose not to prioritize it.

Fortunately, there is an alternative solution that doesn't require any code changes to the operator itself while still enabling claiming and binding to RDS instances from within a Tanzu Application Platform cluster.

This solution uses the `SecretTemplate` API provided by Carvel's secretgen-controller. This API can be used to create binding specification-conforming secrets by identifying and collecting information that resources from the RDS APIs provide.

Next, consider the second obstacle where multiple resources must be created to produce a usable RDS database. One solution to this obstacle is to just document all the resources that must be created to produce something that can be used. This solution is laborious, error-prone, and is generally a poor developer experience.

Fortunately, there is an alternative solution that abstracts away the complexities of creating instances that are known to work well with application workloads.

This solution uses the `ClusterInstanceClass` API provided by Services Toolkit. Instance classes allow for logical service instances to be presented to Application Operators, allowing them to discover, reason about, and, most importantly, claim service instances that they can then bind to their application workloads.

The rest of this topic describes how both these solutions can come together to form an end-to-end integration for RDS services on Tanzu Application Platform.

Create an RDS service instance

This section describes how to create an RDS service instance in Tanzu Application Platform by using a ready-made reference Carvel Package. This step is typically performed by the Service Operator role. Follow the steps in Creating an RDS service instance by using a Carvel Package.

Alternatively, if you want to author your own reference package and want to learn about the underlying APIs and how they come together to produce a useable service instance for Tanzu Application Platform, you can achieve the same outcome by using the more advanced Creating an RDS service instance manually.

After you complete either of these steps and have a running RDS service instance, return here to continue with the rest of the use case.

Create a service instance class for RDS

Now that you know how to create RDS service instances it's time to learn how to make those instances discoverable to Application Operators. This step is typically performed by the Service Operator role.

You can use Services Toolkit's `ClusterInstanceClass` API to create a service instance class to represent RDS service instances within the cluster. The existence of such classes make these logical service instances discoverable to Application Operators. This allows them to create Resource Claims for such instances and to then bind them to application workloads.

Create the following Kubernetes resource on your EKS cluster:

```
# clusterinstanceclass.yaml
---
apiVersion: services.apps.tanzu.vmware.com/v1alpha1
kind: ClusterInstanceClass
metadata:
  name: aws-rds-postgres
spec:
  description:
    short: AWS RDS instances with a postgresql engine
  pool:
    kind: Secret
    labelSelector:
      matchLabels:
        services.apps.tanzu.vmware.com/class: rds-postgres
```

Apply it by running:

```
kubectl apply -f clusterinstanceclass.yaml
```

In this example, the class states that claimable instances of RDS PostgreSQL are represented by

`Secret` objects with the label `services.apps.tanzu.vmware.com/class` set to `rds-postgres`. A `Secret` with this label was created in the earlier step when you provisioned an RDS service instance.

Although this example uses `services.apps.tanzu.vmware.com/class`, there is no special meaning to that key. The Service Operator role can choose arbitrary label names and values. They might also decide to select multiple labels or combine a label selector with a field selector when defining the `ClusterInstanceClass`.

After creating a `ClusterInstanceClass`, you must grant sufficient RBAC permissions to enable Services Toolkit to read the resources that match the pool definition of the instance class. For this example, create the following aggregated `ClusterRole` in your EKS cluster:

```
# stk-secret-reader.yaml
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: stk-secret-reader
  labels:
    servicebinding.io/controller: "true"
rules:
- apiGroups:
  - ""
  resources:
  - secrets
  verbs:
  - get
  - list
  - watch
```

Apply it by running:

```
kubectl apply -f stk-secret-reader.yaml
```

If you want to claim resources across namespace boundaries, you must create a corresponding `ResourceClaimPolicy`. For example, if the provisioned RDS PostgreSQL instances exist in the namespace `service-instances`, and you want to allow Application Operators to claim them for workloads residing in the `default` namespace, create the following `ResourceClaimPolicy`:

```
# resourceclaimpolicy.yaml
apiVersion: services.apps.tanzu.vmware.com/v1alpha1
kind: ResourceClaimPolicy
metadata:
  name: default-can-claim-rds-postgres
  namespace: service-instances
spec:
  subject:
    kind: Secret
    group: ""
    selector:
      matchLabels:
        services.apps.tanzu.vmware.com/class: rds-postgres
  consumingNamespaces: [ "default" ]
```

Apply it by running:

```
kubectl apply -f resourceclaimpolicy.yaml
```

Discover, Claim, and Bind to an RDS

Creating the `ClusterInstanceClass` and the corresponding RBAC informs Application Operators that RDS is available to use with their application workloads on Tanzu Application Platform. In this section you learn how to discover, claim, and bind to the RDS service instance previously created. The Application Operator is typically the role that discovers and claims service instances. The Application Developer is typically the role that handles binding.

To discover what service instances are available to them, Application Operators can run

```
tanzu services classes list

  NAME                      DESCRIPTION
  aws-rds-postgres          AWS RDS instances with a postgresql engine
```

Here you can see information about the `ClusterInstanceClass` created in the earlier step. Each `ClusterInstanceClass` created is added to the list of classes returned here.

The next step is to claim an instance of the wanted class, but to do that, Application Operators must first discover the list of currently claimable instances for the class. Many variables, including namespace boundaries, claim policies, and the exclusivity of claims, affect the capacity to claim instances. Therefore Services Toolkit provides the CLI command `tanzu service claimable list` to help inform Application Operators of the instances that can enable successful claims. Example:

```
tanzu services claimable list --class aws-rds-postgres

  NAME           NAMESPACE  API KIND  API GROUP/VERSION
  rds-bindable   default    Secret    v1
```

Because of the setup performed as part of Creating a claimable class for RDS instances, the secrets created from the `SecretTemplate` as part of Create an RDS service instance now appear as claimable to the Application Operator. From here on it is simply a case of creating a resource claim for the instance and then binding the claim to an application workload.

Create a claim for the newly created secret by running:

```
tanzu service claim create ack-rds-claim \
  --resource-name rds-bindable \
  --resource-kind Secret \
  --resource-api-version v1
```

Obtain the claim reference of the claim by running:

```
tanzu service claim list -o wide
```

Verify that the output is similar to the following:

```
NAME                        READY  REASON  CLAIM REF
ack-rds-claim               True           services.apps.tanzu.vmware.com/v1alpha1:Resourc
eClaim:ack-rds-claim
```

Create an application workload that consumes the claimed RDS PostgreSQL Database. Example:

```
tanzu apps workload create my-workload \
  --git-repo https://github.com/sample-accelerators/spring-petclinic \
  --git-branch main \
  --git-tag tap-1.2 \
  --type web \
  --label app.kubernetes.io/part-of=spring-petclinic \
  --annotation autoscaling.knative.dev/minScale=1 \
  --env SPRING_PROFILES_ACTIVE=postgres \
  --service-ref db=services.apps.tanzu.vmware.com/v1alpha1:ResourceClaim:ack-rds-claim
```

`--service-ref` is set to the claim reference obtained previously.

Your application workload now starts up and connects automatically to the RDS service instance. You can verify this by visiting the app in the browser and, for example, creating a new owner through the UI.

Prerequisites

Meet these prerequisites to follow along with Consuming AWS RDS on Tanzu Application Platform (TAP) with AWS Controllers for Kubernetes (ACK).

1. Install the AWS CLI or gain access to the Amazon Cloud Console
2. Gain the AWS privileges required to configure the IAM permissions and identity used by the ACK service controller for RDS
3. Create an Amazon EKS cluster. The quickest and simplest way to create an EKS cluster is to use eksctl, as in this example:

   ```
   eksctl create cluster -r YOUR-REGION -m 6 -M 8 -n YOUR-CLUSTER-NAME
   ```

4. Tanzu Application Platform v1.2.0 or later and Cluster Essentials v1.2.0 or later have to be installed on the Kubernetes cluster.

   Note: To check if you have an appropriate version, run the following:

   ```
   kubectl api-resources | grep secrettemplate
   ```

   This command returns the `SecretTemplate` API. If it does not for you, verify that Cluster Essentials for VMware Tanzu v1.2.0 or later is installed.

5. Install the ACK service controller for RDS and configure it in the cluster. It is recommended to install the latest stable version of the Operator (v0.0.25 is known to work with this specific use case). For instructions, see Install an ACK Controller. This entails installing the RDS ACK service controller, which entails updating some of the environment variables used throughout the official documentation. In particular, note the following changes:

   - Set the `SERVICE` environment variable to `rds` by running:

     ```
     export SERVICE=rds
     ```

   - Set the `AWS_REGION` environment variable to the AWS region where the RDS instances is created by running:

```
export AWS_REGION=us-east-1
```

6. After the operator is installed, configure IAM permissions. Set the following environment variables accordingly:

   - Set the `SERVICE` environment variable to `rds` by running:

     ```
     export SERVICE=rds
     ```

   - Set the `EKS_CLUSTER_NAME` environment variable to the name of your EKS cluster by running:

     ```
     export EKS_CLUSTER_NAME=<YOUR_CLUSTER_NAME>
     ```

   - Set the `AWS_REGION` environment variable to the AWS region where the RDS instances is created by running:

     ```
     export AWS_REGION=us-east-1
     ```

Configuring the AWS RDS environment

This topic tells you how to configure your AWS environment for Consuming AWS RDS on Tanzu Application Platform (TAP) with AWS Controllers for Kubernetes (ACK).

Prerequisites

Meet the prerequisites for consuming AWS RDS on Tanzu Application Platform with AWS Controllers for Kubernetes (ACK), including using `eksctl` to create an EKS cluster. This procedure entails reusing the resources created when you created the cluster.

You can still create separate VPCs, subnets and security groups if you want. Ensure that these are configured such that Tanzu Application Platform workloads on EKS can discover and connect to RDS instances.

Configure the AWS RDS environment

To configure the AWS RDS environment:

1. Use the AWS cloud console to determine the VPC ID of the EKS cluster, or run this command:

   ```
   aws eks describe-cluster --name YOUR-CLUSTER-NAME --region YOUR-REGION | \
     jq -r .cluster.resourcesVpcConfig.vpcId
   ```

   RDS instances must be configured with a subnet group consisting of two or more subnets. The subnets within the subnet group must adhere to the following rules:

   - The subnets must be in different availability zones, such as us-west-1a and us-west-1b.

- All subnets must either be public or private, which the `MapPublicIpOnLaunch` value reveals.

2. Discover existing subnets within your VPC by using the AWS Cloud console or by running:

```
aws ec2 describe-subnets --filters "Name=vpc-id,Values=YOUR-VPC-ID" --region YO
UR-REGION | \
  jq -r '.Subnets[] | select(.MapPublicIpOnLaunch == false) | .SubnetId'
```

3. Create the following Kubernetes resource on your EKS cluster by using the subnet IDs output:

```
# dbsubnetgroup.yaml
---
apiVersion: rds.services.k8s.aws/v1alpha1
kind: DBSubnetGroup
metadata:
  name: DB-SUBNET-GROUP-NAME
  namespace: ack-system
spec:
  name: DB-SUBNET-GROUP-NAME
  description: rds-subnet-group
  subnetIDs:
  - SUBNET-ID-1
  - SUBNET-ID-2
  - SUBNET-ID-3
```

   Where `DB-SUBNET-GROUP-NAME`, `SUBNET-ID-1`, `SUBNET-ID-2`, and `SUBNET-ID-3` are your own values.

4. Run

```
kubectl apply -f dbsubnetgroup.yaml
```

5. Confirm that you created `DBSubnetGroup` by running:

```
kubectl get DBSubnetGroup -n ack-system DB-SUBNET-GROUP-NAME -o yaml
```

6. Identify a suitable security group to use for the RDS instance that allows workloads running on the Tanzu Application Platform cluster to establish a connection. Do so by searching for a suitable security group within the AWS cloud console, or by running the following command, which identifies the `Communication between all nodes in the cluster` security group:

```
aws ec2 describe-security-groups --filters "Name=vpc-id,Values=YOUR-VPC-ID" --r
egion YOUR-REGION | \
  jq -r '.SecurityGroups[] | select(.Description == "Communication between all
nodes in the cluster").GroupId'
```

7. Record `DB-SUBNET-GROUP-NAME` and the security group ID output from the previous command. You need both when creating RDS instances as part of this use case.

Creating AWS RDS Instances manually using kubectl (experimental)

This topic is for users who want to understand the underlying APIs involved in making a bindable service instance using `DBInstance` and `SecretTemplate` resources. For a simpler user experience, see Creating an RDS service instance through a Carvel Package.

Prerequisite

Meet the prerequisites in Consuming AWS RDS on Tanzu Application Platform (TAP) with AWS Controllers for Kubernetes (ACK) and keep the following information to hand:

- `DB-SUBNET-GROUP-NAME` - the name of the `DBSubnetGroup` resource previously created
- `SECURITY-GROUP-ID` - the security group ID to use for this RDS instance

Create an RDS service instance by using kubectl

Follow these procedures to create an RDS service instance by using kubectl.

Create the `DBInstance` resource

This example uses secret-gen to generate a `Password` for the `DBInstance`. You can also provide an explicit password through a `Secret`.

1. Create Kubernetes resources on your EKS cluster by using the following example. This YAML creates the `DBInstance` resource in the `default` namespace.

```
# dbinstance.yaml
---
apiVersion: secretgen.k14s.io/v1alpha1
kind: Password
metadata:
 name: rds-psql-password
 namespace: default
spec:
 length: 64
 secretTemplate:
   type: Opaque
   stringData:
     password: $(value) # do not edit, this will auto generate a password.
---
apiVersion: rds.services.k8s.aws/v1alpha1
kind: DBInstance
metadata:
 name: rds-psql-1
 namespace: default
spec:
 allocatedStorage: 20
 dbInstanceClass: db.t3.micro
 dbInstanceIdentifier: rds-psql-1
 dbName: postgres
 engine: postgres
 engineVersion: "14.1"
 masterUsername: adminUser
 masterUserPassword:
   namespace: default
   name: rds-psql-password
   key: password
```

```
  vpcSecurityGroupIDs:
  - SECURITY-GROUP-ID                     # modify value
  dbSubnetGroupName: DB-SUBNET-GROUP-NAME # modify value

  # note: due to an issue in the RDS ACK controller, it is recommended to explic
itly set the
  # following optional spec fields.
  # default values for the optional fields are provided below.
  # https://github.com/aws-controllers-k8s/community/issues/1346
  autoMinorVersionUpgrade: true
  backupRetentionPeriod: 1
  copyTagsToSnapshot: false
  deletionProtection: false
  licenseModel: postgresql-license
  monitoringInterval: 0
  multiAZ: false
  preferredBackupWindow: 23:00-23:30
  preferredMaintenanceWindow: wed:23:34-thu:00:04
  publiclyAccessible: false
  storageEncrypted: false
  storageType: gp2
```

Where:

- `DB-SUBNET-GROUP-NAME` is the name of the `DBSubnetGroup` resource previously created
- `SECURITY-GROUP-ID` is the security group ID to use for this RDS instance

2. Run:

```
kubectl apply -f dbinstance.yaml
```

3. Verify the creation status of the `DBInstance` by inspecting the conditions in the Kubernetes API. To do so, run:

```
kubectl get DBInstance rds-psql-1 -o yaml -n default
```

Create a Binding Specification Compatible Secret

As mentioned in Creating service instances that are compatible with Tanzu Application Platform, for Tanzu Application Platform workloads to be able to claim and bind to services such as RDS, a resource compatible with Service Binding Specification must exist in the cluster.

This can take the form of either a `ProvisionedService` or a Kubernetes `Secret` with some known keys. Both are defined in the specification.

The RDS `DBInstance` you created does not adhere to `ProvisionedService` and does not create a spec-compatible secret. So, you must create one using the resources you have available.

In this topic, you create a Kubernetes secret in the necessary format using the secret-gen tooling. You do so by using the `SecretTemplate` API to extract values from the `DBInstance` resource and populate a new spec-compatible secret with the values.

Create a ServiceAccount for secret templating

As part of using the `SecretTemplate` API, a Kubernetes `ServiceAccount` must be provided. The

`ServiceAccount` is used for reading the `DBInstance` resource and the `Secret` created from the `Password` resource.

1. Create the following Kubernetes resources on your EKS cluster:

```
# secrettemplate-sa.yaml
---
apiVersion: v1
kind: ServiceAccount
metadata:
 name: rds-resources-reader
 namespace: default
---
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
 name: rds-resources-reading
 namespace: default
rules:
- apiGroups:
 - ""
 resources:
 - secrets
 verbs:
 - get
 - list
 - watch
 resourceNames:
 - rds-psql-password
- apiGroups:
 - rds.services.k8s.aws
 resources:
 - dbinstances
 verbs:
 - get
 - list
 - watch
 resourceNames:
 - rds-psql-1
---
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
 name: rds-resources-reader-to-read
 namespace: default
roleRef:
 apiGroup: rbac.authorization.k8s.io
 kind: Role
 name: rds-resources-reading
subjects:
 - kind: ServiceAccount
 name: rds-resources-reader
 namespace: default
```

2. Run:

```
kubectl apply -f secrettemplate-sa.yaml
```

Create a SecretTemplate

In combination with the `ServiceAccount` you created, a `SecretTemplate` can be used to declaratively create a secret that is compatible with the service binding specification.

The `.spec.inputResources` fields list the resources with information needed to create the secret. The `.spec.template` field defines how that information is interpolated as a secret.

To specify fields on an input resource, you can use JSONPath syntax that is very similar to kubectl syntax. The only difference is the delimiters, which are `\$(` and `)` instead of `{` and `}`.

For example, `$(.rds.status.endpoint.address)` produces the host address of an RDS instance if the input resource is an ACK controller `DBInstance` resource.

This syntax can currently be used in the following fields of the `SecretTemplate` API:

- `.spec.inputResource[].ref.name` for dynamically loading input resources of the APIs of input resources previously in the list
- `.spec.template` values for taking values from the input resources and interpolating them into the secret you create

In this case, you directly reference the `DBInstance` resource and then dynamically load the secret containing the password from its specification.

You then create a `Secret` conforming to the Postgres auto-configuration for Spring Cloud Bindings to enable a fully automated, end-to-end binding experience for application workloads on Tanzu Application Platform.

1. Create the following Kubernetes resources on your EKS cluster:

```
# bindable-rds-secrettemplate.yaml
---
apiVersion: secretgen.carvel.dev/v1alpha1
kind: SecretTemplate
metadata:
 name: rds-bindable
 namespace: default
spec:
 serviceAccountName: rds-resources-reader
 inputResources:
 - name: rds
   ref:
     apiVersion: rds.services.k8s.aws/v1alpha1
     kind: DBInstance
     name: rds-psql-1
 - name: creds
   ref:
     apiVersion: v1
     kind: Secret
     name: "$(.rds.spec.masterUserPassword.name)"
template:
 metadata:
   labels:
     app.kubernetes.io/component: rds-postgres
     app.kubernetes.io/instance: "$(.rds.metadata.name)"
     services.apps.tanzu.vmware.com/class: rds-postgres
 type: postgresql
```

```
  stringData:
    type: postgresql
    port: "$(.rds.status.endpoint.port)"
    database: "$(.rds.spec.dbName)"
    host: "$(.rds.status.endpoint.address)"
    username: "$(.rds.spec.masterUsername)"
  data:
    password: "$(.creds.data.password)"
```

2. Run:

```
kubectl apply -f bindable-rds-secrettemplate.yaml
```

Verify

Find the name of the secret produced by reading the status of `SecretTemplate`. To do so, run:

```
kubectl get secrettemplate -n default rds-bindable -o jsonpath="{.status.secret.name}"
```

Delete an RDS service instance

Delete an RDS service instance by running:

```
kubectl delete DBInstance rds-psql-1 -n default
kubectl delete SecretTemplate rds-bindable -n default
kubectl delete Password rds-psql-password -n default
kubectl delete ServiceAccount rds-resources-reader -n default
kubectl delete RoleBinding rds-resources-reader-to-read -n default
kubectl delete Role rds-resources-reading -n default
```

Summary and Next Steps

You learned how to use Carvel's `SecretTemplate` API to construct a secret that is compatible with the binding specification to create an AWS RDS service instance.

Now that you have this available in the cluster, you can learn how to make use of it by continuing where you left off in Consuming AWS RDS on Tanzu Application Platform (TAP) with AWS Controllers for Kubernetes (ACK).

Creating AWS RDS instances by using a Carvel package (experimental)

This topic describes how to create, update, and delete RDS service instances by using a Carvel package. For a more detailed and low-level alternative procedure, see Creating AWS RDS Instances manually by using kubectl.

Prerequisite

Meet the prerequisites in Consuming AWS RDS on Tanzu Application Platform (TAP) with AWS Controllers for Kubernetes (ACK).

The package repository and service instance package bundles for this topic are in the tanzu-application-platform-reference-packages GitHub repository.

Create an RDS service instance using a Carvel package

Follow the steps in the following procedures.

Add a reference package repository to the in the cluster

To add a reference package repository to the in the cluster:

1. Use the Tanzu CLI to add the new Service Reference packages repository by running:

```
tanzu package repository add tap-service-reference-packages --url ghcr.io/vmwar
e-tanzu/tanzu-application-platform-reference-packages/tap-service-reference-pac
kage-repo:0.0.1 -n tanzu-package-repo-global
```

2. Use the following example to create a `ServiceAccount` that you use to provision `PackageInstall` resources. The namespace of this `ServiceAccount` must match the namespace of the `tanzu package install` command in the next step.

```
# rds-service-account-installer.yaml
---
apiVersion: v1
kind: ServiceAccount
metadata:
 name: rds-install
 namespace: default
---
kind: Role
apiVersion: rbac.authorization.k8s.io/v1
metadata:
 name: rds-install
 namespace: default
rules:
- apiGroups: ["*"] # TODO: use more fine-grained RBAC permissions
 resources: ["*"]
 verbs: ["*"]
---
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
 name: rds-install
 namespace: default
subjects:
- kind: ServiceAccount
 name: rds-install
roleRef:
 apiGroup: rbac.authorization.k8s.io
 kind: Role
 name: rds-install
```

3. Run:

```
kubectl apply -f rds-service-account-installer.yaml
```

Create an RDS service instance through the Tanzu CLI

To create an RDS service instance through the Tanzu CLI:

1. Create the following Kubernetes resources on your EKS cluster:

   ```
   # RDS-INSTANCE-NAME-values.yaml
   ---
   name: "RDS-INSTANCE-NAME"
   namespace: "default"
   dbSubnetGroupName: "DB-SUBNET-GROUP-NAME"
   vpcSecurityGroupIDs:
   - "SECURITY-GROUP-ID"
   ```

   Where:

   - `RDS-INSTANCE-NAME` is a chosen name for the RDS instance to create
   - `DB-SUBNET-GROUP-NAME` is the name of the `DBSubnetGroup` resource previously created
   - `SECURITY-GROUP-ID` is the security group ID to use for this RDS instance

2. Use the Tanzu CLI to install an instance of the reference service instance Package by running:

   ```
   tanzu package install RDS-INSTANCE-NAME --package-name psql.aws.references.serv
   ices.apps.tanzu.vmware.com --version 0.0.1-alpha --service-account-name rds-ins
   tall -f RDS-INSTANCE-NAME-values.yaml -n default
   ```

You can install the `psql.aws.references.services.apps.tanzu.vmware.com` package multiple times to produce multiple RDS service instances.

To do so, prepare a separate `RDS-INSTANCE-NAME-values.yaml` file and then install the package with a different name and the earlier mentioned separate data values file for each RDS service instance.

Verify

To verify:

1. Verify the creation status for the RDS instance by inspecting the conditions in the Kubernetes API. To do so, run:

   ```
   kubectl get DBInstance RDS-INSTANCE-NAME -n default -o yaml
   ```

2. Wait for up to 20 minutes.

3. Find the binding-compliant secret that `PackageInstall` produced by running:

   ```
   kubectl get secrettemplate RDS-INSTANCE-NAME-bindable -n default -o jsonpath="{
   .status.secret.name}"
   ```

Delete an RDS service instance

Delete the RDS service instance by running:

```
tanzu package installed delete RDS-INSTANCE-NAME -n default
```

Summary

You learned how to use Carvel's `Package` and `PackageInstall` APIs to create an RDS service instance. To learn more about the pieces that comprise this service instance package, see Create an RDS service instance manually.

Now that you have an RDS service instance in the cluster, you can learn how to make use of it by continuing from where you left off in Consuming AWS RDS on Tanzu Application Platform (TAP) with AWS Controllers for Kubernetes (ACK).

Consuming AWS RDS on Tanzu Application Platform with Crossplane

Overview

This topic describes how to use Services Toolkit to enable Tanzu Application Platform workloads to consume AWS RDS PostgreSQL databases.

This topic makes use of Crossplane to manage RDS instances in AWS. It is an alternative approach to using the AWS Controllers for Kubernetes (ACK) to achieve the same outcomes.

Prerequisites

Meet these prerequisites:

- Create a Kubernetes cluster that supports running both Tanzu Application Platform and Crossplane
- Install Tanzu Application Platform on the Kubernetes cluster
- Gain access to an AWS account with permissions to manage RDS database instances
- Install AWS CLI
- Configure a named profile for an AWS account that has permissions to manage RDS databases.

Install Crossplane

Run the following commands to install Crossplane to your existing Kubernetes cluster:

```
kubectl create namespace crossplane-system

helm repo add crossplane-stable https://charts.crossplane.io/stable
helm repo update

helm install crossplane --namespace crossplane-system crossplane-stable/crossplane \
  --set 'args={--enable-external-secret-stores}'
```

Note: For the latest steps for installing Crossplane, see the Crossplane documentation. As of Crossplane 1.9.0, the feature flag `--enable-external-secret-stores` is still needed.

For this topic, you do not need to install the Crossplane CLI or any additional configuration package.

Install AWS Provider for Crossplane

To install the AWS Provider for Crossplane:

1. Run:

   ```
   kubectl apply -f -<<EOF
   ---
   apiVersion: pkg.crossplane.io/v1
   kind: Provider
   metadata:
    name: provider-aws
   spec:
    package: xpkg.upbound.io/crossplane/provider-aws:v0.24.1
   EOF
   ```

2. After installing the provider, you see a new `rdsinstances.database.aws.crossplane.io` API resource available in your Kubernetes cluster. See the health of the installed provider by running:

   ```
   kubectl get provider.pkg.crossplane.io provider-aws
   ```

Configure AWS provider

To configure an AWS provider:

1. Create a new key file:

   ```
   AWS_PROFILE=default && echo -e "[default]\naws_access_key_id = $(aws configure
   get aws_access_key_id --profile $AWS_PROFILE)\naws_secret_access_key = $(aws co
   nfigure get aws_secret_access_key --profile $AWS_PROFILE)\naws_session_token =
   $(aws configure get aws_session_token --profile $AWS_PROFILE)" > creds.conf
   ```

   If your AWS profile is not named `default`, change `AWS_PROFILE` to the actual name.

2. Verify that you a created a new key file by reading the content of the newly created `creds.conf` file.

3. Create a new secret from the key file by running:

   ```
   kubectl create secret generic aws-provider-creds -n crossplane-system --from-fi
   le=creds=./creds.conf
   ```

4. Delete the key file by running:

   ```
   rm -f creds.conf
   ```

5. Configure the AWS provider to use the newly created secret by running:

   ```
   kubectl apply -f -<<EOF
   ---
   apiVersion: aws.crossplane.io/v1beta1
   ```

```
kind: ProviderConfig
metadata:
 name: default
spec:
 credentials:
   source: Secret
   secretRef:
     namespace: crossplane-system
     name: aws-provider-creds
     key: creds
EOF
```

Define composite resource types

Now that the AWS provider for Crossplane is installed and configured, you can create a new `CompositeResourceDefinition` (XRD) and corresponding `Composition` representing individual instances of RDS PostgreSQL by following the steps in this section. For more information about these concepts see the Crossplane composition documentation.

Instead of creating your own custom XRD and composition, you can also install an existing Crossplane configuration package for AWS that includes pre-configured XRDs and compositions for RDS.

The primary reason for choosing to create a new XRD and composition is to ensure the connection secrets for newly provisioned RDS PostgreSQL instances support the Service Binding Specification for Kubernetes and automatic Spring Boot configuration using Spring Cloud Bindings.

1. Create a new XRD by running:

```
kubectl apply -f -<<EOF
---
apiVersion: apiextensions.crossplane.io/v1
kind: CompositeResourceDefinition
metadata:
 name: xpostgresqlinstances.bindable.database.example.org
spec:
 claimNames:
   kind: PostgreSQLInstance
   plural: postgresqlinstances
 connectionSecretKeys:
 - type
 - provider
 - host
 - port
 - database
 - username
 - password
 group: bindable.database.example.org
 names:
   kind: XPostgreSQLInstance
   plural: xpostgresqlinstances
 versions:
 - name: v1alpha1
   referenceable: true
   schema:
     openAPIV3Schema:
       properties:
         spec:
           properties:
             parameters:
               properties:
                 storageGB:
                   type: integer
               required:
               - storageGB
               type: object
           required:
           - parameters
           type: object
       type: object
   served: true
EOF
```

After the newly created XRD is reconciled there are two new API resources available in your Kubernetes cluster, `xpostgresqlinstances.bindable.database.example.org` and `postgresqlinstances.bindable.database.example.org`.

2. Create a corresponding composition by running:

```
kubectl apply -f -<<EOF
---
apiVersion: apiextensions.crossplane.io/v1
kind: Composition
metadata:
 labels:
   provider: "aws"
   vpc: "default"
 name: xpostgresqlinstances.bindable.aws.database.example.org
spec:
 compositeTypeRef:
   apiVersion: bindable.database.example.org/v1alpha1
   kind: XPostgreSQLInstance
 publishConnectionDetailsWithStoreConfigRef:
   name: default
 resources:
 - base:
     apiVersion: database.aws.crossplane.io/v1beta1
     kind: RDSInstance
     spec:
       forProvider:
         dbInstanceClass: db.t2.micro
         engine: postgres
         dbName: postgres
         engineVersion: "12"
         masterUsername: masteruser
         publiclyAccessible: true
         region: us-east-1
         skipFinalSnapshotBeforeDeletion: true
       writeConnectionSecretToRef:
         namespace: crossplane-system
   connectionDetails:
   - name: type
     value: postgresql
   - name: provider
     value: aws
```

```
    - name: database
      value: postgres
    - fromConnectionSecretKey: username
    - fromConnectionSecretKey: password
    - name: host
      fromConnectionSecretKey: endpoint
    - fromConnectionSecretKey: port
    name: rdsinstance
    patches:
    - fromFieldPath: metadata.uid
      toFieldPath: spec.writeConnectionSecretToRef.name
      transforms:
      - string:
          fmt: '%s-postgresql'
          type: Format
        type: string
      type: FromCompositeFieldPath
    - fromFieldPath: spec.parameters.storageGB
      toFieldPath: spec.forProvider.allocatedStorage
      type: FromCompositeFieldPath
EOF
```

This composition ensures that all RDS PostgreSQL instances are placed in the `us-east-1` region and use the default VPC for the respective AWS account.

3. Take one of these actions:

   - Connect to those instances from outside the default VPC by assigning an appropriate inbound rule for TCP on port `5432` to the security group of that VPC.
   - Define a composition that creates a separate VPC for each RDS PostgreSQL instance and automatically configures inbound rules. See this example.

Create an instance class

To make instances of a service easy for application operators to discover and claim, the service operator persona creates a `ClusterInstanceClass`. In this example, the class states that claimable instances of RDS PostgreSQL are represented by secret objects of type `connection.crossplane.io/v1alpha1` with the label `services.apps.tanzu.vmware.com/class` set to `rds-postgres`:

```
kubectl apply -f -<<EOF
---
apiVersion: services.apps.tanzu.vmware.com/v1alpha1
kind: ClusterInstanceClass
metadata:
  name: rds-postgres
spec:
  description:
    short: AWS RDS Postgresql database instances
  pool:
    kind: Secret
    labelSelector:
      matchLabels:
        services.apps.tanzu.vmware.com/class: rds-postgres
    fieldSelector: type=connection.crossplane.io/v1alpha1
EOF
```

In addition, grant RBAC permissions to Services Toolkit to enable reading the secrets specified by the class.

```
kubectl apply -f -<<EOF
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: stk-secret-reader
  labels:
    servicebinding.io/controller: "true"
rules:
- apiGroups:
  - ""
  resources:
  - secrets
  verbs:
  - get
  - list
  - watch
EOF
```

Provision RDS PostgreSQL instance

As the service operator persona, you now provision an instance of RDS PostgreSQL using the `postgresqlinstances.bindable.database.example.org` API managed by the XRD you previously created.

`.spec.publishConnectionDetailsTo` provides Crossplane with the name and a label for the secret that stores the connection details for the newly created database. You can see that the label specified here matches the drop-down menu value defined in the `ClusterInstanceClass` you created earlier.

1. Create an RDS database instance in your AWS account by running:

```
kubectl apply -f -<<EOF
---
apiVersion: bindable.database.example.org/v1alpha1
kind: PostgreSQLInstance
metadata:
 name: rds-postgres-db
 namespace: default
spec:
 parameters:
   storageGB: 20
 compositionSelector:
   matchLabels:
     provider: aws
     vpc: default
 publishConnectionDetailsTo:
   name: rds-postgres-db
   metadata:
     labels:
       services.apps.tanzu.vmware.com/class: rds-postgres
EOF
```

2. Verify that you created the RDS database instance by running:

   ```
   aws rds describe-db-instances --region us-east-1 --profile default
   ```

   Expect the status of the newly created `PostgreSQLInstance` resource to be `READY=True`. This might take a few minutes. You can wait for this by running:

   ```
   kubectl wait --for=condition=Ready=true postgresqlinstances.bindable.database.e
   xample.org rds-postgres-db
   ```

As soon as the RDS PostgreSQL instance is ready, it is claimable by the application operator persona as shown in the next section.

Claim the RDS PostgreSQL instance and connect to it from the Tanzu Application Platform workload

Thanks to the `ClusterInstanceClass` created in the earlier section, application operators can now use the Tanzu CLI to discover and claim secrets representing RDS PostgreSQL instances.

1. Show available classes of service instances by running:

   ```
   tanzu service classes list

   NAME            DESCRIPTION
   rds-postgres    AWS RDS Postgresql database instances
   ```

2. Show claimable instances belonging to the RDS PostgreSQL class by running:

   ```
   tanzu services claimable list --class rds-postgres

   NAME               NAMESPACE  API KIND  API GROUP/VERSION
   rds-postgres-db    default    Secret    v1
   ```

3. Create a claim for the discovered secret by running:

   ```
   tanzu service claim create rds-claim \
   --resource-name rds-postgres-db \
   --resource-kind Secret \
   --resource-api-version v1
   ```

4. Obtain the claim reference by running:

   ```
   tanzu service claim list -o wide
   ```

   Expect to see the following output:

   ```
   NAME                         READY  REASON   CLAIM REF
   rds-claim                    True            services.apps.tanzu.vmware.com/v1alpha1
   :ResourceClaim:rds-claim
   ```

5. Create an application workload that consumes the claimed RDS PostgreSQL database. In this example, `--service-ref` is set to the claim reference obtained earlier.

```
tanzu apps workload create my-workload \
--git-repo https://github.com/sample-accelerators/spring-petclinic \
--git-branch main \
--git-tag tap-1.2 \
--type web \
--label app.kubernetes.io/part-of=spring-petclinic \
--annotation autoscaling.knative.dev/minScale=1 \
--env SPRING_PROFILES_ACTIVE=postgres \
--service-ref db=services.apps.tanzu.vmware.com/v1alpha1:ResourceClaim:rds-clai
m
```

Consuming Azure Flexible Server Tanzu Application Platform

This section of the documentation covers integrations of Azure Flexible Server into Tanzu Application Platform. Documentation is provided for both an integration using Azure Service Operator (ASO), as well as an integration using Crossplane.

Consuming Azure Flexible Server for PostgreSQL on Tanzu Application Platform with Azure Service Operator (ASO)

This topic describes using Services Toolkit to allow Tanzu Application Platform workloads to consume Azure Flexible Server PostgreSQL. This particular topic makes use of Azure Service Operator v2 to manage PostgreSQL instances in Azure.

Important: This use case is not currently compatible with air-gapped Tanzu Application Platform installations.

Prerequisites

Meet these prerequisites

Create service instances that are compatible with Tanzu Application Platform

To create an Azure PostgreSQL service instance for Tanzu Application Platform to consume, you can use a ready-made, reference Carvel package. The Service Operator typically performs this step. Follow the steps in Creating an Azure PostgreSQL service instance using a Carvel package.

```
$ kubectl api-resources --api-group=dbforpostgresql.azure.com
```

```
NAME                              SHORTNAMES   APIVERSION
  NAMESPACED   KIND
flexibleservers                                dbforpostgresql.azure.com/v1beta20210601
  true         FlexibleServer
flexibleserversconfigurations                  dbforpostgresql.azure.com/v1beta20210601
  true         FlexibleServersConfiguration
flexibleserversdatabases                       dbforpostgresql.azure.com/v1beta20210601
  true         FlexibleServersDatabase
flexibleserversfirewallrules                   dbforpostgresql.azure.com/v1beta20210601
  true         FlexibleServersFirewallRule
```

There is also the Resource Group, which is in another API group.

```
$ kubectl api-resources --api-group=resources.azure.com
```

```
NAME              SHORTNAMES   APIVERSION                            NAMESPACED   KIND
resourcegroups                 resources.azure.com/v1beta20200601   true         Resour
ceGroup
```

To create an Azure PostgreSQL service instance for Tanzu Application Platform to consume, you can use a ready-made, reference Carvel package. The Service Operator typically performs this step. Follow the steps in Creating an Azure PostgreSQL service instance using a Carvel package.

Alternatively, if you are interested in authoring your own reference package and want to learn about the underlying APIs and how they come together to produce a useable service instance for Tanzu Application Platform, you can achieve the same outcome by using the more advanced Creating an Azure PostgreSQL service instance manually topic.

After creating a running Azure PostgreSQL service instance, return here to continue the use case.

Create a service instance class for PSQL

After creating Flexible Server service instances, you must make it possible for application operators to discover them. The service operator role typically performs this step.

You can use Services Toolkit's `ClusterInstanceClass` API to create a service instance class that represents psql service instances within the cluster. The existence of such classes enables application operators to discover logical service instances. This, in turn, enables application operators to create Resource Claims for such instances and to then bind them to application workloads.

Create the following Kubernetes resource on your AKS cluster by running:

```
cat <<EOF | kubectl apply -f -
---
apiVersion: services.apps.tanzu.vmware.com/v1alpha1
kind: ClusterInstanceClass
metadata:
  name: azure-postgres
spec:
  description:
    short: Azure Flexible Server instances with a postgresql engine
  pool:
    kind: Secret
    labelSelector:
      matchLabels:
        services.apps.tanzu.vmware.com/class: azure-postgres
EOF
```

In this particular example, the class represents claimable instances of PostgreSQL by a `Secret` object with the label `services.apps.tanzu.vmware.com/class` set to `azure-postgres`.

In addition, you must grant RBAC permissions to Services Toolkit for reading the secrets that the class specifies. Create the following RBAC on your AKS cluster by running:

```
cat <<EOF | kubectl apply -f -
```

```
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: stk-secret-reader
  labels:
    servicebinding.io/controller: "true"
rules:
- apiGroups:
  - ""
  resources:
  - secrets
  verbs:
  - get
  - list
  - watch
EOF
```

To claim resources across namespace boundaries, create a corresponding `ResourceClaimPolicy`.

For example, if the provisioned Azure Flexible Server instance exists in the namespace `service-instances`, and you want to allow application operators to claim them for workloads residing in the `default` namespace, you must create the following `ResourceClaimPolicy` by running:

```
cat <<EOF | kubectl apply -f -
---
apiVersion: services.apps.tanzu.vmware.com/v1alpha1
kind: ResourceClaimPolicy
metadata:
  name: default-can-claim-azure-postgres
  namespace: service-instances
spec:
  subject:
    kind: Secret
    group: ""
    selector:
      matchLabels:
        services.apps.tanzu.vmware.com/class: azure-postgres
  consumingNamespaces: [ "default" ]
EOF
```

Discover, Claim, and Bind to a PostgreSQL

Creating the `ClusterInstanceClass` and the corresponding RBAC informs application operators that Azure PostgreSQL is available to use with their application workloads on Tanzu Application Platform.

This section describes how to discover, claim, and bind to the PostgreSQL service instance previously created.

Discovering and claiming service instances is typically the responsibility of the application operator role. Binding is typically an action for application developers.

To discover which service instances are available to them, application operators can run:

```
tanzu services classes list

NAME                      DESCRIPTION
```

```
azure-postgres        Azure Flexible Server instances with a postgresql engine
```

You can see information about the `ClusterInstanceClass` created in the earlier step. Each `ClusterInstanceClass` created is added to the list of classes.

Next, the application operator claims an instance of the class they want. But to do that the application operator must first discover the list of currently claimable instances for the class.

Many variables affect the capacity to claim instances, including namespace boundaries, claim policies, and the exclusivity of claims. Therefore, Services Toolkit provides the CLI command `tanzu service claimable list` to help inform application operators of the instances that can cause successful claims.

Example:

```
tanzu services claimable list --class azure-postgres

  NAME               NAMESPACE  API KIND  API GROUP/VERSION
  aso-psql-bindable  default    Secret    v1
```

Create a claim for the newly created secret by running:

```
tanzu services claim create aso-psql-claim \
  --resource-name aso-psql-bindable \
  --resource-kind Secret \
  --resource-api-version v1
```

Obtain the claim reference of the claim by running:

```
tanzu services claim list -o wide
```

Verify the output is similar to the following:

```
NAME                        READY  REASON  CLAIM REF
aso-psql-claim              True           services.apps.tanzu.vmware.com/v1alpha1:Resourc
eClaim:aso-psql-claim
```

Test claim With Tanzu Application Platform workload

Create an application workload that consumes the claimed Azure PostgreSQL database by running:

```
tanzu apps workload create my-workload
```

Example:

```
tanzu apps workload create my-workload \
  --git-repo https://github.com/sample-accelerators/spring-petclinic \
  --git-tag tap-1.2 \
  --type web \
  --label app.kubernetes.io/part-of=spring-petclinic \
  --annotation autoscaling.knative.dev/minScale=1 \
  --env SPRING_PROFILES_ACTIVE=postgres \
  --service-ref db=services.apps.tanzu.vmware.com/v1alpha1:ResourceClaim:aso-psql-clai
m
```

`--service-ref` is set to the claim reference obtained previously.

Your application workload starts and connects automatically to the Azure PostgreSQL service instance. You can verify this by visiting the app in the browser and, for example, creating a new owner through the UI.

Delete a PostgreSQL service instance

To delete the Azure PostgreSQL service instance, run the appropriate cleanup commands for how you created the service.

Delete a PostgreSQL service instance by using a Carvel package

```
tanzu package installed delete demo-psql-instance
```

Delete a PostgreSQL service instance by using kubectl

Delete the Azure PostgreSQL service instance by running:

```
kubectl delete flexibleservers.dbforpostgresql.azure.com aso-psql
kubectl delete flexibleserversfirewallrules.dbforpostgresql.azure.com aso-psql
kubectl delete flexibleserversdatabases.dbforpostgresql.azure.com aso-psql
kubectl delete SecretTemplate aso-psql-bindable
kubectl delete Password aso-psql
kubectl delete ServiceAccount aso-psql-reader
kubectl delete RoleBinding aso-psql-reader-to-read
kubectl delete Role aso-psql-reading
```

Troubleshooting Azure Service Operator

Azure Service Operator is still in beta and doesn't always behave as expected. For help with most common scenarios, see Troubleshooting.

Prerequisites

To follow the procedures in Consuming Azure Flexible Server for PostgreSQL on Tanzu Application Platform with Azure Service Operator (ASO) you need:

- An Azure AKS Kubernetes cluster
    - This cluster should have a Paid SKU tier. Using the Free tier may cause resource limitation issues.
- Tanzu Application Platform v1.2.0 or later
- Azure Service Operator (ASO) installed on the cluster

If you do not already have a cluster that meets these requirements, you can follow this procedure to create and configure a cluster:

1. Install the Azure CLI. For how to do so, see the Microsoft documentation.
2. Ensure that you are logged in to Azure by running:

```
az login
```

3. Create an Azure Kubernetes Service (AKS) cluster. The quickest and simplest way to create an AKS cluster is to use the Azure CLI, as in the following example that creates a new ResourceGroup and AKS cluster:

```
# Name of the resource group to contain the AKS cluster
RESOURCE_GROUP_NAME=tap-psql-demo

# Location of the Cluster
LOCATION=centralus

# Cluster name
CLUSTER_NAME=tap-psql-demo-cluster

# Arbitrary labels for the cluster
LABELS="key=value key2=value2"

# Number of k8s nodes
NODES=2

az group create --name "${RESOURCE_GROUP_NAME}" --location "${LOCATION}"

az aks create -g "${RESOURCE_GROUP_NAME}" -n "${CLUSTER_NAME}" --enable-managed
-identity --node-count "${NODES}" --enable-addons monitoring --tags "${LABELS}"
 -s Standard_DS3_v2 --generate-ssh-keys  --uptime-sla

az aks get-credentials --resource-group "${RESOURCE_GROUP_NAME}" --name "${CLUS
TER_NAME}"
```

Note: This creates an AKS cluster with a paid tier using the `--uptime-sla` flag. Not setting this flag will cause the Kubernetes Control plane to potentially have resource limitation issues. See https://learn.microsoft.com/en-us/azure/aks/quotas-skus-regions#service-quotas-and-limits

For more information about AKS, see the Microsoft documentation.

4. Install Tanzu Application Platform v1.2.0 or later and Cluster Essentials v1.2.0 or later on the Kubernetes cluster. For more information, see Installing Tanzu Application Platform

5. Verify that you have the appropriate versions by running:

```
kubectl api-resources | grep secrettemplate
```

This command returns the `SecretTemplate` API. If it does not work for you, you might not have Cluster Essentials for VMware Tanzu v1.2.0 or later installed.

6. Install the Azure Service Operator (ASO) and configure it in the cluster. You must have the appropriate permission in Azure to create a service principal and configure Azure access. v2.0.0-beta.2 is known to work with this use case. Install the latest stable version of the operator by running:

```
AZURE_TENANT_ID=$(az account show | jq -r '.tenantId')
AZURE_SUBSCRIPTION_ID=$(az account show | jq -r '.id')

az ad sp create-for-rbac -n tap-azure-service-operator --role contributor \
--scopes /subscriptions/"${AZURE_SUBSCRIPTION_ID}" > /tmp/aso-creds.json

AZURE_CLIENT_ID=$(cat /tmp/aso-creds.json | jq -r '.appId')
AZURE_CLIENT_SECRET=$(cat /tmp/aso-creds.json | jq -r '.password' )

rm -f  /tmp/aso-creds.json

# requires carvel kapp v0.46+
kapp deploy -a aso -f https://github.com/Azure/azure-service-operator/releases/
download/v2.0.0-beta.2/azureserviceoperator_v2.0.0-beta.2.yaml -y --wait=false

cat <<EOF | kubectl apply -f -
apiVersion: v1
kind: Secret
metadata:
  name: aso-controller-settings
  namespace: azureserviceoperator-system
stringData:
  AZURE_SUBSCRIPTION_ID: "${AZURE_SUBSCRIPTION_ID}"
  AZURE_TENANT_ID: "${AZURE_TENANT_ID}"
  AZURE_CLIENT_ID: "${AZURE_CLIENT_ID}"
  AZURE_CLIENT_SECRET: "${AZURE_CLIENT_SECRET}"
EOF

kubectl wait deployment -n azureserviceoperator-system -l app=azure-service-ope
rator-v2 --for=condition=Available=True
```

Next Steps

See Consuming Azure Flexible Server for PostgreSQL on Tanzu Application Platform with Azure Service Operator (ASO).

Creating Azure PostgreSQL Instances manually using kubectl (experimental)

This topic describes how to use Services Toolkit to allow Tanzu Application Platform workloads to consume Azure Flexible Server PostgreSQL. This particular topic makes use of Azure Service Operator v2 to manage PostgreSQL instances in Azure.

Create a resource group

First of all, a ResourceGroup for all PSQL Instances to reside in will be created:

```
cat <<EOF | kubectl apply -f -
---
apiVersion: resources.azure.com/v1beta20200601
kind: ResourceGroup
metadata:
  name: aso-psql
spec:
  location: centralus
EOF
```

Create a Flexible Server service instance

Next, you will create a Flexible Server PSQL Instance, a Database and a Firewall Rule in Azure as well as a Secret for credentials. In this guide you will leverage the `Password` API from Carvel's secretgen controller, which will create the `Secrets` for you. However, any other mechanism to manage those secrets works too.

Change the `.spec.azureName` of the `FlexibleServer` resource below from "aso-psql" to something unique, using only lowercase letters, digits and hyphens. This avoids naming conflicts as Azure has a global naming namespace and this resource may already exist.

```
cat <<'EOF' | kubectl apply -f -
---
apiVersion: secretgen.k14s.io/v1alpha1
kind: Password
metadata:
  name: aso-psql
spec:
  length: 64
  secretTemplate:
    type: Opaque
    stringData:
      password: $(value)
---
apiVersion: dbforpostgresql.azure.com/v1beta20210601
kind: FlexibleServersDatabase
metadata:
  name: aso-psql
spec:
  azureName: mydb
  owner:
    name: aso-psql
  charset: utf8
---
apiVersion: dbforpostgresql.azure.com/v1beta20210601
kind: FlexibleServersFirewallRule
metadata:
  name: aso-psql
spec:
  owner:
    name: aso-psql
  startIpAddress: 0.0.0.0 #! only allow traffic from azure. See https://docs.microsoft
.com/en-us/azure/postgresql/single-server/concepts-firewall-rules#connecting-from-azur
e. Warning not for production use.
  endIpAddress: 0.0.0.0
---
apiVersion: dbforpostgresql.azure.com/v1beta20210601
kind: FlexibleServer
metadata:
  name: aso-psql
spec:
  location: centralus
  azureName: aso-psql #! CHANGE THIS NAME
  owner:
    name: aso-psql #! the ResourceGroup above
  version: "13" #! only 11,12,13 supported
  sku:
    name: Standard_D4s_v3
    tier: GeneralPurpose
  administratorLogin: myAdmin
  administratorLoginPassword:
    name: aso-psql
    key: password
  storage:
    storageSizeGB: 128
EOF
```

Create a Binding Specification Compatible Secret

As mentioned in Creating service instances that are compatible with Tanzu Application Platform, in order for Tanzu Application Platform workloads to be able to claim and bind to services such as Azure PostgreSQL, a resource compatible with Service Binding Specification must exist in the cluster. This can take the form of either a `ProvisionedService`, as defined by the specification, or a Kubernetes `Secret` with some known keys, also as defined in the specification.

In this guide, you create a Kubernetes secret in the necessary format using the secretgen-controller tooling. You do so by using the `SecretTemplate` API to extract values from the Azure Service Operator resources and populate a new spec-compatible secret with the values.

Create a ServiceAccount for Secret Templating

As part of using the `SecretTemplate` API, a Kubernetes `ServiceAccount` must be provided. The `ServiceAccount` is used for reading the `FlexibleServer` resource and the `Secret` created from the `Password` resource above.

Create the following Kubernetes resources on your AKS cluster:

```
cat <<EOF | kubectl apply -f -
---
apiVersion: v1
kind: ServiceAccount
metadata:
  name: aso-psql-reader
  namespace: default
---
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: aso-psql-reading
  namespace: default
rules:
- apiGroups:
  - ""
  resources:
  - secrets
  verbs:
  - get
  - list
  - watch
  resourceNames:
  - aso-psql
- apiGroups:
  - dbforpostgresql.azure.com
  resources:
  - flexibleservers
  - flexibleserversdatabases
  verbs:
  - get
  - list
  - watch
  resourceNames:
  - aso-psql
---
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: aso-psql-reader-to-read
  namespace: default
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: Role
  name: aso-psql-reading
subjects:
- kind: ServiceAccount
  name: aso-psql-reader
  namespace: default
EOF
```

Create a SecretTemplate

In combination with the `ServiceAccount` just created, a `SecretTemplate` can be used to declaratively create a secret that is compatible with the service binding specification. For more information on this API see the Secret Template Documentation.

Create the following Kubernetes resources on your AKS cluster:

```
cat <<'EOF' | kubectl apply -f -
---
apiVersion: secretgen.carvel.dev/v1alpha1
kind: SecretTemplate
metadata:
  name: aso-psql-bindable
  namespace: default
spec:
  serviceAccountName: aso-psql-reader
  inputResources:
  - name: server
    ref:
      apiVersion: dbforpostgresql.azure.com/v1alpha1api20210601
      kind: FlexibleServer
      name: aso-psql
  - name: db
    ref:
      apiVersion: dbforpostgresql.azure.com/v1alpha1api20210601
      kind: FlexibleServersDatabase
      name: aso-psql
  - name: creds
    ref:
      apiVersion: v1
      kind: Secret
      name: "$(.server.spec.administratorLoginPassword.name)"
  template:
    metadata:
      labels:
        app.kubernetes.io/component: aso-psql
        app.kubernetes.io/instance: "$(.server.metadata.name)"
        services.apps.tanzu.vmware.com/class: azure-postgres
    type: postgresql
    stringData:
      type: postgresql
      port: "5432"
      database: "$(.db.status.name)"
      host: "$(.server.status.fullyQualifiedDomainName)"
      username: "$(.server.status.administratorLogin)"
    data:
      password: "$(.creds.data.password)"
EOF
```

Verify the Service Instance

Firstly wait until the PostgreSQL instance is ready. This may take 5 to 10 minutes.

```
kubectl wait flexibleservers.dbforpostgresql.azure.com aso-psql -n default --for=condi
tion=Ready --timeout=5m
```

Next, ensure a bindable `Secret` was produced by the `SecretTemplate`. To do so, run:

```
kubectl wait SecretTemplate -n default aso-psql-bindable --for=condition=ReconcileSucc
eeded --timeout=5m

kubectl get Secret -n default aso-psql-bindable
```

Creating Azure PostgreSQL instances by using a Carvel package (experimental)

This topic describes creating, updating, and deleting Azure PostgreSQL service instances using a Carvel package. For a more detailed and low-level alternative procedure, see Creating Service Instances that are compatible with Tanzu Application Platform.

Prerequisite

Meet the prerequisites:

The Package Repository and service instance Package Bundles for this guide can be found in the Reference Service Packages GitHub repository.

Create an Azure PostgreSQL service instance using a Carvel package

Follow the steps in the following procedures.

Add a reference package repository to the cluster

The namespace `tanzu-package-repo-global` has a special significance. The kapp-controller defines a Global Packaging namespace. In this namespace, any package the is made available through a Package Respository, is available in every namespace.

When the kapp-controller is installed via Tanzu Application Platform, the namespace is `tanzu-package-repo-global`. If you install the controller in another way, verify which namespace is considered the Global Packaging namespace.

To add a reference package repository to the cluster:

1. Use the Tanzu CLI to add the new Service Reference packages repository:

```
tanzu package repository add tap-reference-service-packages \
    --url ghcr.io/vmware-tanzu/tanzu-application-platform-reference-service-pac
kages:0.0.3 \
    -n tanzu-package-repo-global
```

2. Create a `ServiceAccount` to provision `PackageInstall` resources by using the following example. The namespace of this `ServiceAccount` must match the namespace of the `tanzu package install` command in the next step.

```
kubectl apply -f - <<'EOF'
---
apiVersion: v1
kind: ServiceAccount
metadata:
  name: psql-install
---
kind: Role
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: psql-install
rules:
- apiGroups: ["dbforpostgresql.azure.com"]
  resources: ["flexibleservers","flexibleserversdatabases","flexibleserversfire
wallrules"]
  verbs:     ["*"]
- apiGroups: ["resources.azure.com"]
  resources: ["resourcegroups"]
  verbs:     ["*"]
- apiGroups: ["secretgen.carvel.dev", "secretgen.k14s.io"]
  resources: ["secrettemplates","passwords"]
  verbs:     ["*"]
- apiGroups: [""]
  resources: ["serviceaccounts","configmaps"]
  verbs:     ["*"]
- apiGroups: [""]
  resources: ["namespaces"]
  verbs:     ["get", "list"]
- apiGroups: ["rbac.authorization.k8s.io"]
  resources: ["roles","rolebindings"]
  verbs:     ["*"]
---
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: psql-install
```

```
subjects:
- kind: ServiceAccount
  name: psql-install
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: Role
  name: psql-install
```

Create a Azure PostgreSQL service instance through the Tanzu CLI

Before you create the values file, here are some values highlighted.

- **aso_controller_namespace**: the Azure Service Operator has some potential conflicting behaviors with the kapp-controller. We reduce the conflicts by annotating the resources with the ASO installation namespace.
- **firewall_rules**: by default, the FlexibleServer is not accessible. Setting `0.0.0.0` as the start and end IP addresses for a firewall rule makes the server available from within Azure.
- **resource_group.use_existing**: if you cannot create a Resource Group in Azure or have other reasons for using an existing one, set this to `true`. Else, the package makes a Resource Group with the name specified by the `resource_group.name` value.

The `server.name` field will be used for the FlexibleServer resource name on Azure, otherwise `name` will be used. It is recommended to set the value of the `name` (and the optional `server.name`) field below from `aso-psql` to something unique, using only lowercase letters, digits and hyphens. This avoids naming conflicts, as Azure has a global naming namespace for FlexibleServer instances and this resource may already exist. Do make sure you also change the commands below using a `aso-psql` value, such as the `aso-psql-bindable` from the SecretTemplate,and replace `aso-psql` with the actual `name`.

1. Create a file holding the configuration of the Azure PostgreSQL service instance:

```
cat <<'EOF' > aso-psql-instance-values.yml
---
name: aso-psql
namespace: service-instances
location: westeurope
aso_controller_namespace: azureserviceoperator-system
create_namespace: false

server:
    administrator_name: trpadmin

database:
    name: testdb

firewall_rules:
    - startIpAddress: 0.0.0.0
      endIpAddress: 0.0.0.0

resource_group:
    use_existing: false
    name: aso-psql
EOF
```

Note

: To understand which settings are available for this package you can run:

```
tanzu package available get \
  --values-schema \
  psql.azure.references.services.apps.tanzu.vmware.com/0.0.1-alpha
```

This shows a list of all configuration options you can use in the `aso-psql-instance-values.yml` file.

2. Use the Tanzu CLI to install an instance of the reference service instance Package.

```
tanzu package install aso-psql-instance \
   --package-name psql.azure.references.services.apps.tanzu.vmware.com \
   --version 0.0.1-alpha \
   --service-account-name psql-install \
   --values-file aso-psql-instance-values.yml \
   --wait
```

You can install the `psql.azure.references.services.apps.tanzu.vmware.com` package multiple times to produce various Azure PostgreSQL Service instances. You create a separate `<INSTANCE-NAME>-values.yml` for each instance, set a different `name` value, and then install the package with the instance-specific data values file.

Verify the Azure Resources

1. Verify the creation status for the Azure PostgreSQL instance by inspecting the conditions in the Kubernetes API. To do so, run:

```
kubectl get flexibleservers.dbforpostgresql.azure.com aso-psql -o yaml
```

2. After some time has passed, sometimes up to 10 minutes, you can find the binding-compliant secret produced by `PackageInstall`. To do so, run:

```
kubectl get secrettemplate aso-psql-bindable -o jsonpath="{.status.secret.name}
"
```

Verify the Service Instance

Firstly wait until the PostgreSQL instance is ready. This may take 5 to 10 minutes.

```
kubectl wait flexibleservers.dbforpostgresql.azure.com aso-psql -n default --for=condi
tion=Ready --timeout=5m
```

Next, ensure a bindable `Secret` was produced by the `SecretTemplate`. To do so, run:

```
kubectl wait SecretTemplate -n default aso-psql-bindable --for=condition=ReconcileSucc
eeded --timeout=5m

kubectl get Secret -n default aso-psql-bindable
```

Summary

You have learnt to use Carvel's `Package` and `PackageInstall` APIs to create a Azure PostgreSQL service instance. If you want to learn more about the pieces that comprise this service instance package, see Creating Azure PostgreSQL Instances manually using kubectl.

Now that you have this available in the cluster, you can learn how to make use of it by continuing where you left off in Consuming Azure PostgreSQL on Tanzu Application Platform (TAP) with ASO.

Azure Service Operator Troubleshooting

Increase Log Level

There is a guide on the Azure Service Operator (ASO) controller for aiding you in diagnosing problems.

We recommend temporarily change the Controller's binary log level from `v=2` to `v=6`. Setting it higher than six prints a lot more things, such as the HTTP requests with headers, and usually doesn't add more value.

```
kubectl edit deploy -n azureserviceoperator-system azureserviceoperator-controller-man
ager
```

```
spec:
  template:
    spec:
      containers:
      - name: manager
        args:
        - --metrics-addr=0.0.0.0:8080
        - --health-addr=:8081
        - --enable-leader-election
        - --v=6
```

Not Updating The Kubernetes Resources

The ASO controller sometimes conflicts when updating the resource status in Kubernetes. The resource in Azure exists, but is not reflected properly in its corresponding Kubernetes resource.

In the logs you will see a `409 conflict` message when updating the Kubernetes resource. To resolve this, you can restart the Pod, which will take a few seconds.

```
kubectl -n azureserviceoperator-system rollout restart deployment azureserviceoperator
-controller-manager
```

Consuming Azure Flexible Server for PostgreSQL on Tanzu Application Platform with Crossplane

Introduction

This topic demonstrates how to use Services Toolkit to allow Tanzu Application Platform workloads to consume Azure Flexible Server for PostgreSQL. This particular topic makes use of Crossplane to manage those Flexible Server for PostgreSQL instances. As such, it can be thought of as an alternative approach to Consuming Azure Flexible Server for PostgreSQL on Tanzu Application Platform with Azure Service Operator (ASO) to achieve the similar outcomes.

Prerequisites

Meet these prerequisites:

- Install Azure CLI
- Create an AKS cluster
- Install Tanzu Application Platform (v1.2.0 or later) and Cluster Essentials (v1.2.0 or later)

Note

In this example we use an AKS Cluster to deploy Crossplane and Tanzu Application Platform too. However, any other cluster which supports running those two systems should suffice.

Install Crossplane

Note

For the latest steps for installing Crossplane, see these instructions. For the instructions in this topic, it is important to enable support for external secret stores in Crossplane. This is currently an Alpha feature. As such, you will have to explicitly set command line flag `--enable-external-secret-stores` when starting the Crossplane controller.

Run the following commands to install Crossplane to your existing Kubernetes cluster:

```
kubectl create namespace crossplane-system

helm repo add crossplane-stable https://charts.crossplane.io/stable
helm repo update

helm install crossplane --namespace crossplane-system crossplane-stable/crossplane \
  --set 'args={--enable-external-secret-stores}'
```

For this topic, you do not need to install the Crossplane CLI or any additional configuration package.

Install the Azure Provider for Crossplane

To install the Azure Provider for Crossplane, run:

```
kubectl apply -f - <<'EOF'
```

```
apiVersion: pkg.crossplane.io/v1alpha1
kind: ControllerConfig
metadata:
  name: jet-azure-config
spec:
  image: crossplane/provider-jet-azure-controller:v0.12.0
  args: ["-d"]
---
apiVersion: pkg.crossplane.io/v1
kind: Provider
metadata:
  name: provider-jet-azure
spec:
  package: crossplane/provider-jet-azure:v0.12.0
  controllerConfigRef:
    name: jet-azure-config
EOF
```

After you have installed the provider, you see a new `flexibleservers.dbforpostgresql.azure.jet.crossplane.io` API resource available in your Kubernetes cluster. You can wait for the provider to become healthy by running:

```
kubectl -n crossplane-system wait provider/provider-jet-azure \
  --for=condition=Healthy=True --timeout=3m
```

Install the Kubernetes Provider for Crossplane

To install the Kubernetes Provider for Crossplane, run:

```
kubectl apply -f - <<'EOF'
apiVersion: pkg.crossplane.io/v1
kind: Provider
metadata:
  name: provider-kubernetes
spec:
  package: "crossplane/provider-kubernetes:main"
EOF
```

Configure the Azure Provider

This section creates a new Service Principal to be used by the Crossplane system to allow it to manage PostgreSQL Servers.

1. Setup some configuration in the current shell session

   ```
   # Set the name of the Service Principal to be created
   AZURE_SP_NAME='sql-crossplane-demo'

   # Get the subscription ID
   AZURE_SUBSCRIPTION_ID="$( az account show -o json | jq -r '.id' )"
   ```

2. Create a new Service Principal and set up the kubernetes secret

   ```
   kubectl create secret generic jet-azure-creds -o yaml --dry-run=client --from-literal=creds="$(
   ```

```
 az ad sp create-for-rbac -n "${AZURE_SP_NAME}" \
   --sdk-auth \
   --role "Contributor" \
   --scopes "/subscriptions/${AZURE_SUBSCRIPTION_ID}" \
   -o json
)" | kubectl apply -n crossplane-system -f -
```

Note

You'll see the following warning:

`WARNING: Option '--sdk-auth' has been deprecated and will be removed in a future release.`

which you can ignore for now. There is some context about that in this issue for the Azure CLI and this issue for the Crossplane Azure Provider.

3. Deploy a `ProviderConfig` which uses the previously created secret for the Azure crossplane provider

```
kubectl apply -f - <<'EOF'
apiVersion: azure.jet.crossplane.io/v1alpha1
kind: ProviderConfig
metadata:
 name: default
spec:
 credentials:
   source: Secret
   secretRef:
     namespace: crossplane-system
     name: jet-azure-creds
     key: creds
EOF
```

Configure the Kubernetes Provider

```
SA=$(kubectl -n crossplane-system get sa -o name | grep provider-kubernetes | sed -e '
s|serviceaccount\/|crossplane-system:|g')
kubectl create role -n crossplane-system password-manager --resource=passwords.secretg
en.k14s.io --verb=create,get,update,delete
kubectl create rolebinding -n crossplane-system provider-kubernetes-password-manager -
-role password-manager --serviceaccount="${SA}"

kubectl apply -f - <<'EOF'
apiVersion: kubernetes.crossplane.io/v1alpha1
kind: ProviderConfig
metadata:
  name: default
spec:
  credentials:
    source: InjectedIdentity
EOF
```

Define Composite Resource Types

Now that the Azure Provider for Crossplane has been installed and configured, create a new `CompositeResourceDefinition` (XRD) and corresponding `Composition` representing individual instances of Azure PostgreSQL Server. For more information about these concepts see the Crossplane Composition documentation.

1. Create a new XRD by running:

```
kubectl apply -f - <<'EOF'
apiVersion: apiextensions.crossplane.io/v1
kind: CompositeResourceDefinition
metadata:
 name: xpostgresqlinstances.bindable.database.example.org
spec:
 claimNames:
   kind: PostgreSQLInstance
   plural: postgresqlinstances
 connectionSecretKeys:
 - type
 - provider
 - host
 - port
 - database
 - username
 - password
 group: bindable.database.example.org
 names:
   kind: XPostgreSQLInstance
   plural: xpostgresqlinstances
 versions:
 - name: v1alpha1
   referenceable: true
   schema:
     openAPIV3Schema:
       properties:
         spec:
           properties:
             parameters:
               properties:
                 storageGB:
                   type: integer
               required:
               - storageGB
               type: object
           required:
           - parameters
           type: object
       type: object
   served: true
EOF
```

After the newly created XRD has been successfully reconciled, there are two new API resources available in your Kubernetes cluster, `xpostgresqlinstances.bindable.database.example.org` and `postgresqlinstances.bindable.database.example.org`.

2. Create a corresponding composition (not in a production environment) by running:

```
kubectl apply -f - <<'EOF'
apiVersion: apiextensions.crossplane.io/v1
kind: Composition
metadata:
  labels:
    provider: azure
  name: xpostgresqlinstances.bindable.gcp.database.example.org
spec:
  compositeTypeRef:
    apiVersion: bindable.database.example.org/v1alpha1
    kind: XPostgreSQLInstance
  publishConnectionDetailsWithStoreConfigRef:
    name: default
  resources:
  - name: dbinstance
    base:
      apiVersion: dbforpostgresql.azure.jet.crossplane.io/v1alpha2
      kind: FlexibleServer
      spec:
        forProvider:
          administratorLogin: myPgAdmin
          administratorPasswordSecretRef:
            name: ""
            namespace: crossplane-system
            key: password
          location: westeurope
          skuName: GP_Standard_D2s_v3
          version: "12" #! 11,12 and 13 are supported
          resourceGroupName: tap-psql-demo
        writeConnectionSecretToRef:
          namespace: crossplane-system
    connectionDetails:
    - name: type
      value: postgresql
    - name: provider
      value: azure
    - name: database
      value: postgres
    - name: username
      fromFieldPath: spec.forProvider.administratorLogin
    - name: password
      fromConnectionSecretKey: "attribute.administrator_password"
    - name: host
      fromFieldPath: status.atProvider.fqdn
    - name: port
      type: FromValue
      value: "5432"
    patches:
    - fromFieldPath: metadata.uid
      toFieldPath: spec.writeConnectionSecretToRef.name
      transforms:
      - string:
          fmt: '%s-postgresql'
          type: Format
        type: string
      type: FromCompositeFieldPath
    - type: FromCompositeFieldPath
      fromFieldPath: metadata.name
      toFieldPath: spec.forProvider.administratorPasswordSecretRef.name
    - fromFieldPath: spec.parameters.storageGB
      toFieldPath: spec.forProvider.storageMb
      type: FromCompositeFieldPath
      transforms:
      - type: math
        math:
          multiply: 1024
  - name: dbfwrule
    base:
      apiVersion: dbforpostgresql.azure.jet.crossplane.io/v1alpha2
      kind: FlexibleServerFirewallRule
      spec:
        forProvider:
          serverIdSelector:
            matchControllerRef: true
          #! not recommended for production deployments!
          startIpAddress: 0.0.0.0
          endIpAddress: 255.255.255.255
  - name: password
    base:
      apiVersion: kubernetes.crossplane.io/v1alpha1
      kind: Object
      spec:
        forProvider:
          manifest:
            apiVersion: secretgen.k14s.io/v1alpha1
            kind: Password
            metadata:
              name: ""
              namespace: crossplane-system
            spec:
              length: 64
              secretTemplate:
                type: Opaque
                stringData:
                  password: $(value)
    patches:
    - type: FromCompositeFieldPath
      fromFieldPath: metadata.name
      toFieldPath: spec.forProvider.manifest.metadata.name
EOF
```

The composition defined above makes sure that all `FlexibleServers` are placed in the `westeurope` region and under the resource group `tap-psql-demo`. This composition fulfils the XRD previously created.

Warning: Setting the `FlexibleServerFirewallRule` to start at `0.0.0.0` and end at `255.255.255.255` will allow access to the PostgreSQL Server from any IP and is not recommended in a production environment.

Create an Instance Class

In order to make instances of a service easily discoverable and claimable by Application Operators, the role of the Service Operator creates a `ClusterInstanceClass`. In this particular example, the class states that claimable instances of PostgreSQL instances are represented by `Secret` objects of type `connection.crossplane.io/v1alpha1` with label `services.apps.tanzu.vmware.com/class` set to `azure-postgres`:

```
kubectl apply -f - <<'EOF'
apiVersion: services.apps.tanzu.vmware.com/v1alpha1
kind: ClusterInstanceClass
metadata:
  name: azure-postgres
spec:
  description:
    short: Azure Postgresql database instances
  pool:
    kind: Secret
    labelSelector:
      matchLabels:
        services.apps.tanzu.vmware.com/class: azure-postgres
    fieldSelector: type=connection.crossplane.io/v1alpha1
EOF
```

In addition, you need to grant sufficient RBAC permissions to Services Toolkit to be able to read the secrets specified by the class.

```
kubectl apply -f - <<'EOF'
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: stk-secret-reader
  labels:
    servicebinding.io/controller: "true"
rules:
- apiGroups:
  - ""
  resources:
  - secrets
  verbs:
  - get
  - list
  - watch
EOF
```

Provision Azure Flexible Server for PostgreSQL instances

Playing the role of the Service Operator, you now provision an instance of an Azure Flexible Server for PostgreSQL using the `postgresqlinstances.bindable.database.example.org` API managed by the XRD you previously created. Note that `.spec.publishConnectionDetailsTo` provides Crossplane with the name and a label for the secret that is being used to store the connection details for the newly created database. You can see that the label specified here matches the label selector defined on the `ClusterInstanceClass` you created in the previous step.

The `PostgreSQLInstance` has a dependency on a `Secret` where the Service Operator needs to specify the password for the admin user. Here we use Carvel's `Password` API to create this `Secret` for us.

Run the following command:

```
kubectl apply -f - <<'EOF'
```

```
apiVersion: bindable.database.example.org/v1alpha1
kind: PostgreSQLInstance
metadata:
  name: postgresql-server
  namespace: default
spec:
  parameters:
    #! supported storage sizes: 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384,
32768
    storageGB: 32
  compositionSelector:
    matchLabels:
      provider: azure
  publishConnectionDetailsTo:
    name: postgresql-server
    metadata:
      labels:
        services.apps.tanzu.vmware.com/class: azure-postgres
EOF
```

Running this command will cause the creation of a Azure Flexible Server for PostgreSQL instance in your Azure account. You can use the Azure CLI to verify this:

```
az postgres flexible-server list -o table
```

After the instance has been successfully created, the status of the newly created `PostgreSQLInstance` resource should show `READY=True`. This might take a few minutes. You can wait for this by running:

```
kubectl wait postgresqlinstances.bindable.database.example.org/postgresql-server \
    --for=condition=Ready=true --timeout=10m
```

As soon as the Azure Flexible Server for PostgreSQL instance is ready, it is claimable by the role of the Application Operator as shown in the next section.

Note

There is currently a bug in Crossplane 1.7.2 onwards with the `--enable-external-secret-stores` feature gate enabled where the controller will fail to clean up a local secret created by the field `.spec.publishConnectionDetailsTo` after the deletion of the claim. A workaround is to temporarily give the crossplane controller the necessary i.e. permissions:

```
kubectl create clusterrole crossplane-cleaner --verb=delete --resource=s
ecrets
kubectl create clusterrolebinding crossplane-cleaner --clusterrole=cross
plane-cleaner --serviceaccount=crossplane-system:crossplane
```

Claim the Azure Flexible Server for PostgreSQL Server instance and connect to it from the Tanzu Application Platform Workload

Thanks to the previously created `ClusterInstanceClass`, `Secrets` representing PostgreSQL Server instances can now be discovered and claimed by Application Operators through the Tanzu CLI as shown below.

1. Show available classes of service instances by running:

   ```
   tanzu service classes list

     NAME              DESCRIPTION
     azure-postgres    Azure Postgresql database instances
   ```

2. Show claimable instances belonging to the PostgreSQL Server instance class by running:

   ```
   tanzu services claimable list --class azure-postgres

     NAME                 NAMESPACE  API KIND  API GROUP/VERSION
     postgresql-server    default    Secret    v1
   ```

3. Create a claim for the discovered instance by running:

   ```
   tanzu service claim create postgresql-server-claim \
     --resource-name postgresql-server\
     --resource-kind Secret \
     --resource-api-version v1
   ```

4. Obtain the claim reference by running:

   ```
   tanzu service claim list -o wide
   ```

   Expect to see the following output:

   ```
   NAME                         READY  REASON  CLAIM REF
   postgresql-server-claim      True           services.apps.tanzu.vmware.com/v1alp
   ha1:ResourceClaim:postgresql-server-claim
   ```

5. Create an application workload that consumes the claimed PostgreSQL Server instance by running:

   Example:

   ```
   tanzu apps workload create my-workload \
     --git-repo https://github.com/sample-accelerators/spring-petclinic \
     --git-branch main \
     --git-tag tap-1.2 \
     --type web \
     --label app.kubernetes.io/part-of=spring-petclinic \
     --annotation autoscaling.knative.dev/minScale=1 \
     --env SPRING_PROFILES_ACTIVE=postgres \
     --service-ref db=services.apps.tanzu.vmware.com/v1alpha1:ResourceClaim:postgr
   esql-server-claim
   ```

   Note that `--service-ref` is being set to the claim reference obtained previously.

Consuming Google Cloud SQL on Tanzu Application Platform

This section of the documentation covers integrations of Google Cloud SQL into Tanzu Application Platform. Documentation is provided for both an integration using Config Connector, as well as an integration using Crossplane.

Consuming Google Cloud SQL on Tanzu Application Platform (TAP) with Config Connector

Introduction

This topic demonstrates how to use Services Toolkit to allow TAP Workloads to consume Google Cloud SQL for PostgreSQL databases. This particular guide makes use of Config Connector to manage PostgreSQL instances in GCP.

This is describing the procedures to produce similar outcomes as in "Consuming AWS RDS on Tanzu Application Platform (TAP) with AWS Controllers for Kubernetes (ACK)". The same points discussed in "Creating Service Instances that are compatible with Tanzu Application Platform" apply here too:

- Neither of the resources discussed below adhere to the Service Binding Specification
- We need to manage the lifecycle of multiple resources which together form a usable database instance

Note Please ensure you have met all prerequisites before reading on.

Creating Service Instances that are compatible with Tanzu Application Platform

The installation of the Config Connector Addon results in the availability of new Kubernetes APIs for interacting with Google Cloud resources, specifically Cloud SQL resources, from within the TAP cluster.

```
$ kubectl api-resources --api-group sql.cnrm.cloud.google.com

NAME            SHORTNAMES                        APIVERSION                         NA
MESPACED   KIND
sqldatabases   gcpsqldatabase,gcpsqldatabases   sql.cnrm.cloud.google.com/v1beta1   tr
ue         SQLDatabase
sqlinstances   gcpsqlinstance,gcpsqlinstances   sql.cnrm.cloud.google.com/v1beta1   tr
ue         SQLInstance
sqlsslcerts    gcpsqlsslcert,gcpsqlsslcerts     sql.cnrm.cloud.google.com/v1beta1   tr
ue         SQLSSLCert
sqlusers       gcpsqluser,gcpsqlusers           sql.cnrm.cloud.google.com/v1beta1   tr
ue         SQLUser
```

To create a CloudSQL service instance for consumption by Tanzu Application Platform, you can use a ready-made, reference Carvel Package. This step is typically performed by the role of the Service Operator. Follow the steps in Creating an CloudSQL service instance by using a Carvel Package.

Alternatively, if you are interested in authoring your own Reference Package and want to learn about the underlying APIs and how they come together to produce a useable service instance for Tanzu Application Platform, you can achieve the same outcome by using the more advanced Creating an CloudSQL service instance manually.

Once you have completed either of these steps and have a running CloudSQL service instance, please return here to continue with the rest of the use case.

Creating a Service Instance Class for Cloud SQL

We can now make the Cloud SQL Service Instance discoverable to Application Operators. This step is typically performed by the role of the Service Operator.

You can use Services Toolkit's `ClusterInstanceClass` API to create a "Service Instance Class" to represent Cloud SQL Service Instances within the cluster. The existence of such classes make these logical Service Instances discoverable to Application Operators, thus allowing them to create Resource Claims for such instances and to then bind them to Application Workloads.

Create the following Kubernetes resource::

```
apiVersion: services.apps.tanzu.vmware.com/v1alpha1
kind: ClusterInstanceClass
metadata:
  name: cloudsql-postgres
spec:
  description:
    short: Google Cloud SQL with a postgresql engine
  pool:
    kind: Secret
    labelSelector:
      matchLabels:
        services.apps.tanzu.vmware.com/class: cloudsql-postgres
```

In this particular example, the class states that claimable instances of Cloud SQL Postgresql are represented by `Secret` objects with label `services.apps.tanzu.vmware.com/class` set to `cloudsql-postgres`. A `Secret` with this label was created earlier when you created the CloudSQL service instance.

Although this example uses `services.apps.tanzu.vmware.com/class`, there is no special meaning to that key. The Service Operator persona can choose arbitrary label names and values. They might also decide to select on multiple labels or combine a label selector with a field selector when defining the `ClusterInstanceClass`.

Now that you have created a `ClusterInstanceClass`, you need to grant sufficient RBAC permissions to enable Services Toolkit to read the resources that match the pool definition of the instance class. For this example, create the following aggregated `ClusterRole` in your cluster:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: stk-secret-reader
  labels:
    servicebinding.io/controller: "true"
rules:
- apiGroups: [ "" ]
  resources: [ "secrets" ]
  verbs:     [ "get", "list", "watch" ]
```

If you want to claim resources across namespace boundaries, you will have to create a corresponding `ResourceClaimPolicy`. For example, if the provisioned Cloud SQL instances exist in namespace `service-instances` and you want to allow App Operators to claim them for workloads residing in the `default` namespace, you would have to create the following `ResourceClaimPolicy`:

```
#! optional, when workload and services are in different namespaces
apiVersion: services.apps.tanzu.vmware.com/v1alpha1
kind: ResourceClaimPolicy
metadata:
  name: default-can-claim-cloudsql-postgres
  namespace: service-instances
spec:
  subject:
    kind: Secret
    group: ""
    selector:
      matchLabels:
        services.apps.tanzu.vmware.com/class: cloudsql-postgres
  consumingNamespaces: [ "default" ]
```

Discover, Claim and Bind to a Google Cloud SQL Postgresql Instance

The act of creating the `ClusterInstanceClass` and the corresponding RBAC essentially advertises to Application Operators that Cloud SQL Instances are available to use with their Application Workloads on Tanzu Application Platform. In this step you will learn how to discover, claim and bind to the Cloud SQL Service Instance previously created. Discovery and claiming of Service Instances is typically the role of the Application Operator while binding is typically a step for Application Developers.

To discover what Service Instances are available to them, Application Operators can use the `tanzu services classes list` command.

```
tanzu services classes list

  NAME                      DESCRIPTION
  cloudsql-postgres         Google Cloud SQL with a postgresql engine
```

Here you can see information about the `ClusterInstanceClass` created in the previous step. Each `ClusterInstanceClass` created will be added to the list of classes returned here.

The next step is to “claim” an instance of the desired class, but in order to do that, Application Operators must first discover the list of currently claimable instances for the class. Claimability of instances is affected by many variables (including namespace boundaries, claim policies and the exclusivity of claims) and so Services Toolkit provides a CLI command to help inform Application Operators of the instances that will result in successful claims. This command is the `tanzu service claimable list` command.

```
tanzu services claimable list --class cloudsql-postgres

  NAME                        NAMESPACE            KIND     APIVERSION
  sql-instance-claimable      service-instances    Secret   v1
```

Due to the setup done as part of creating a claimable class for Cloud SQL instances, the `Secret`s created from the `SecretTemplate` now appears as "claimable" to the Application Operator. From here on it is simply a case of creating a resource claim for the instance and then binding the claim to an Application Workload.

Create a claim for the newly created secret by running:

```
tanzu service claim create cloudsql-postgres-claim \
  --resource-name sql-instance-claimable \
  --resource-namespace service-instances \
  --resource-kind Secret \
  --resource-api-version v1
```

Obtain the claim reference of the claim by running:

```
tanzu service claim list -o wide
```

Expect to see the following output:

```
NAME                     READY  REASON  CLAIM REF
cloudsql-postgres-claim  True   Ready   services.apps.tanzu.vmware.com/v1alpha1:Resour
ceClaim:cloudsql-postgres-claim
```

Create an Application Workload that consumes the claimed Cloud SQL Postgresql Database by running:

Example:

```
tanzu apps workload create my-workload \
  --git-repo https://github.com/sample-accelerators/spring-petclinic \
  --git-branch main \
  --git-tag tap-1.2 \
  --type web \
  --label app.kubernetes.io/part-of=spring-petclinic \
  --annotation autoscaling.knative.dev/minScale=1 \
  --env SPRING_PROFILES_ACTIVE=postgres \
  --service-ref db=services.apps.tanzu.vmware.com/v1alpha1:ResourceClaim:cloudsql-post
gres-claim
```

`--service-ref` is set to the claim reference obtained previously.

Congratulations - your Application Workload will now start up and will connect automatically to the Cloud SQL Service Instance. This can be verified by visiting the app in the browser and, for example, creating a new "Owner" through the GUI.

Prerequisites

The following prerequisites must be met in order to follow along with Consuming Cloud SQL on Tanzu Application Platform (TAP) with Config Connector.

The `gcloud` CLI

You need to have the `gcloud` CLI installed and authenticated.

A Kubernetes cluster

- with the Config Connector installed & configured
- with a stable Egress IP/CIDR range to allow access to the Cloud SQL instance (see further down at A Cloud NAT service)

In this example we went standard GKE cluster with the Config Connector pre-installed.

It is recommended to install the latest stable version of the Operator (1.71.0 is known to work with this specific use case).

```
GCP_PROJECT='<GCP project ID>'
LABELS='<label1=value1,label2=value2,...>'
CLUSTER_NAME='<GKE cluster name>'

# The Google Cloud Service Account to be used by the Config Connector
SA_NAME="${CLUSTER_NAME}-sa"

# The cluster's node count
# We suggest to start at 6 nodes to host all the TAP systems and to ensure
# the (automatically provisioned and managed) control plane is also scaled
# accordingly.
NODE_COUNT=6

# The namespace you want to deploy the Config Connector / service instance
# objects into
SI_NAMESPACE="service-instances"

# In this example we deploy a zonal cluster, thus you need to provide the
# zone you want your cluster to land in
ZONE='europe-west6-b'

# For Cloud NAT we need to provide the region we want to deploy the router
# to, this needs to be the region the zonal cluster resides in
REGION='europe-west6'

# Will be used for the name of the Cloud NAT router and the NAT config we
# deploy on it
NAT_NAME="${REGION}-nat"

gcloud container --project "${GCP_PROJECT}" \
    clusters create "${CLUSTER_NAME}" \
    --zone "${ZONE}" \
    --release-channel "regular" \
    --machine-type "e2-standard-4" \
    --disk-type "pd-standard" \
    --disk-size "70" \
    --metadata disable-legacy-endpoints=true \
    --num-nodes "${NODE_COUNT}" \
    --node-labels "${LABELS}" \
    --logging=SYSTEM \
    --monitoring=SYSTEM \
    --enable-ip-alias \
    --enable-network-policy \
    --addons ConfigConnector,HorizontalPodAutoscaling,HttpLoadBalancing,GcePersistentD
iskCsiDriver \
    --workload-pool="${GCP_PROJECT}.svc.id.goog" \
    --labels "${LABELS}"
```

```
gcloud iam service-accounts create \
    "${SA_NAME}" \
    --description "${LABELS}"

gcloud projects add-iam-policy-binding "${GCP_PROJECT}" \
    --member="serviceAccount:${SA_NAME}@${GCP_PROJECT}.iam.gserviceaccount.com" \
    --role="roles/editor"

gcloud iam service-accounts add-iam-policy-binding \
    "${SA_NAME}@${GCP_PROJECT}.iam.gserviceaccount.com" \
    --member="serviceAccount:${GCP_PROJECT}.svc.id.goog[cnrm-system/cnrm-controller-ma
nager]" \
    --role="roles/iam.workloadIdentityUser"
```

Configure a stable egress IP

By default egress traffic from pods will get their source IP translated to the node's public IP (SNAT) on the way out. Thus, when we need to configure allowed ingress networks for a Cloud SQL instance, we'd need to add each node of the cluster. Everytime the cluster scales or nodes get repaved, their public IP would change and we would need to make sure to keep the list of authorized networks up to date.

To make this easier we will: - turn off SNAT on the nodes, so egress traffic is not translated to the node's public IP - deploy a Cloud NAT service, which then handles the source IP translation and gives us a stable egress IP

Configure the `ip-masq-agent`

Each cluster comes with a `DaemonSet` `ip-masq-agent` in the `kube-system` namespace. By deploying a configuration for this service and restarting the `DaemonSet`, we can turn off SNAT for egress traffic.

```
cat <<'EOF' | kubectl -n kube-system create cm ip-masq-agent --from-file=config=/dev/s
tdin
nonMasqueradeCIDRs:
- 0.0.0.0/0
EOF

kubectl -n kube-system rollout restart daemonset ip-masq-agent
```

With this config none of the outbound traffic is translated to the node's public IP.

Note: You can also set specfic destination network CIDRs in `nonMasqueradeCIDRs` for which the SNAT on the nodes should be turned off. In that case, any traffic's source IP will still be translated to the node's public IP, except if the destination is explicitly configured in that list.

Set up a Cloud NAT service

After we've turned off SNAT on the nodes, we will employ a Cloud NAT service.

Conceptually this does the same thing as the SNAT on the nodes. However, the difference is, that we don't translate to a node's public IP address, but rather to a reserved IP address that is explicitly used by the Cloud NAT router. Therefore this IP is stable as long as this Cloud NAT router exists and all traffic originating from any pod, regardless which node it resides on, will get its source IP translated to that stable IP.

```
gcloud compute routers create "${NAT_NAME}-router" --region "${REGION}" --network defa
ult
gcloud compute routers nats create "${NAT_NAME}-config" \
    --router-region "${REGION}" \
    --router "${NAT_NAME}-router" \
    --auto-allocate-nat-external-ips \
    --nat-all-subnet-ip-ranges
```

A Tanzu Application Platform installation on the cluster (v1.2.0+).

Tanzu Application Platform (v1.2.0 or newer) and Cluster Essentials (v1.2.0 or newer) have to be installed on the kubernetes cluster.

Note: To check if you have an appropriate version, please run the following:

```
kubectl api-resources | grep secrettemplate
```

This command should return the `SecretTemplate` API. If it does not, ensure Cluster Essentials for VMware Tanzu (v1.2.0 or newer) is installed.

Configure the Config Connector

```
cat <<EOF | kubectl apply -f -
apiVersion: core.cnrm.cloud.google.com/v1beta1
kind: ConfigConnector
metadata:
  name: configconnector.core.cnrm.cloud.google.com
spec:
  mode: cluster
  googleServiceAccount: "${SA_NAME}@${GCP_PROJECT}.iam.gserviceaccount.com"
EOF

kubectl create namespace "${SI_NAMESPACE}"

kubectl annotate namespace "${SI_NAMESPACE}" "cnrm.cloud.google.com/project-id=${GCP_P
ROJECT}"

kubectl wait -n cnrm-system --for=condition=Ready pod --all

gcloud services enable serviceusage.googleapis.com
```

Get the NAT IP(s) for egress from the cluster

```
gcloud compute routers get-status "${NAT_NAME}-router" --region "${REGION}" --format=j
son \
  | jq -r '.result.natStatus[].autoAllocatedNatIps[]'
```

This IP(s) will later be used for allowing access to the CloudSQL instance from the cluster.

Creating Google CloudSQL Instances manually using kubectl (experimental)

Note

: This document is for users who are looking to understand the underlying APIs involved in making a bindable service instance using `SQLInstance`, `SQLDatabase`, `SQLUser` and `SecretTemplate` resources. For a simpler user experience, the alternative Creating an CloudSQL service instance through a Carvel Package topic is recommended.

Prerequisite

Meet the prerequisites and keep the following information to hand:

- `NAT-IP` - the cluster's egress NAT IP

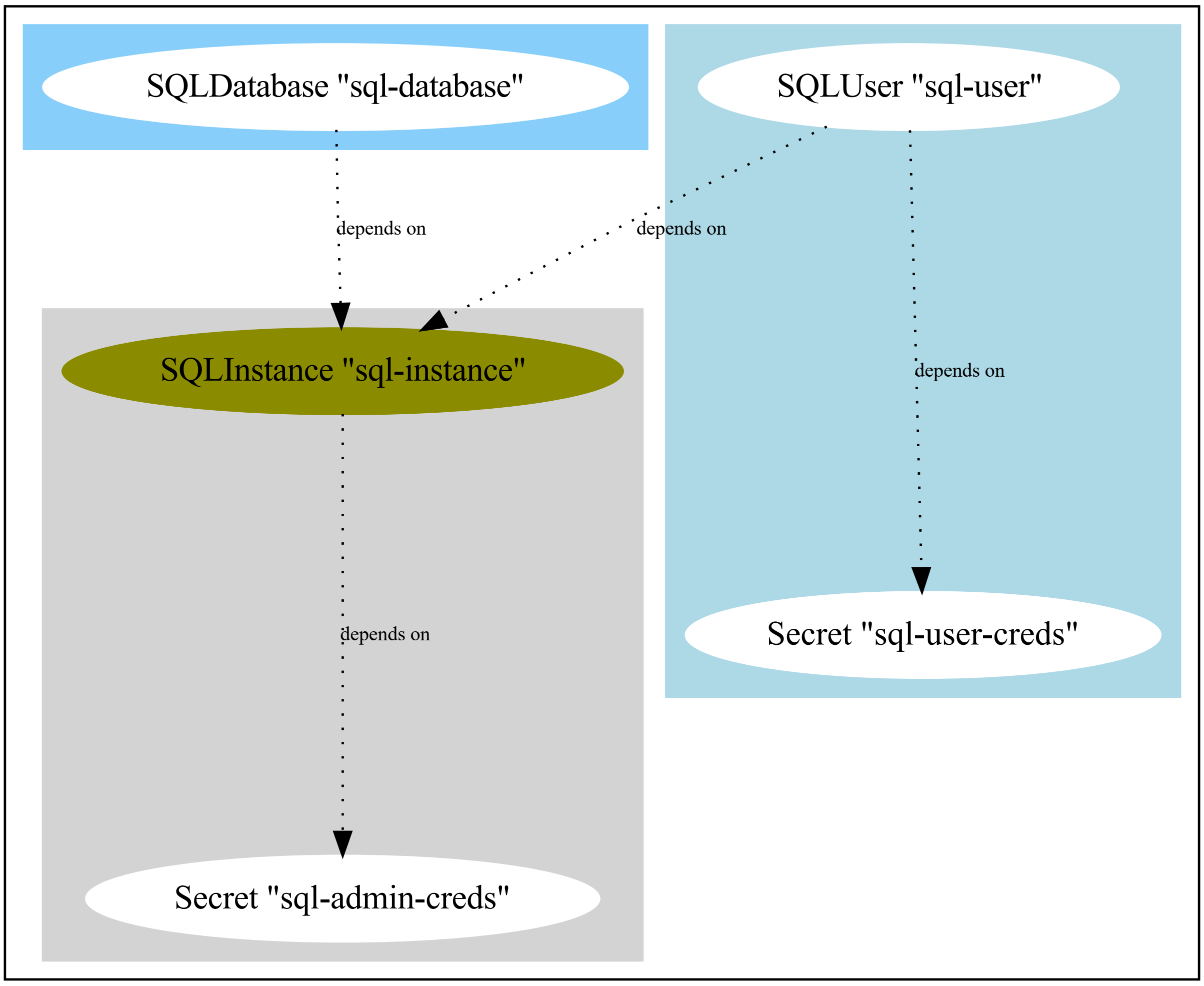
Create a CloudSQL service instance by using kubectl

At a minimum, a useable database instance consists of a `SQLInstance`, a `SQLDatabase`, and a `SQLUser`.

Realistically, in addition to that we will also want another set of `Secrets`:

- one `Secret` per `SQLInstance` to hold the password for the instance's admin role
- one `Secret` per `SQLUser` to hold that user's password

In the simplest case, with one `SQLInstance`, one `SQLDatabase`, and one `SQLUser`, we need to manage the following set of interrelated resources:



Create the `Secrets` for the Database admin & user

First we need to ensure that the `Secrets` which hold the admin's and user's password exist, so we can reference them in the `SQLInstance` and `SQLUser` objects.

Those secrets can be created by any means. In this guide will leverage the `Password` API from Carvel's secretgen controller, which will create the `Secrets` for us. However, any other mechanism to manage those secrets works too.

```
kind: List
apiVersion: v1
items:
- kind: Password
  apiVersion: secretgen.k14s.io/v1alpha1
  metadata:
    name: sql-admin-creds
    namespace: service-instances
  spec: &passwordSpec
    length: 64
    secretTemplate:
      type: Opaque
      stringData:
        password: $(value)
- kind: Password
  apiVersion: secretgen.k14s.io/v1alpha1
  metadata:
    name: sql-user-creds
    namespace: service-instances
  spec: *passwordSpec
```

Applying this will create two `Passwords` which in turn will have two `Secrets` created:

```
kubectl -n service-instances get passwords,secrets sql-user-creds sql-admin-creds
```

```
NAME                                            DESCRIPTION           AGE
password.secretgen.k14s.io/sql-user-creds     Reconcile succeeded   4m41s
password.secretgen.k14s.io/sql-admin-creds    Reconcile succeeded   4m41s

NAME                        TYPE      DATA   AGE
secret/sql-user-creds     Opaque    1      4m41s
secret/sql-admin-creds    Opaque    1      4m41s
```

Create a usable postgres database

Now we can reference those two secrets and use the Config Connector APIs to create our database objects:

Note

: You need to allow access from the Kubernetes cluster's NAT IP. You can get the NAT IP via the command described in the prerequisites. This NAT IP then needs to be used in the `SQLInstance`'s `spec.settings.ipConfiguration.authorizedNetworks`.

```
apiVersion: sql.cnrm.cloud.google.com/v1beta1
kind: SQLInstance
metadata:
  name: sql-instance
  namespace: service-instances
spec:
  databaseVersion: POSTGRES_14
  #! If you have deployed your cluster into a different region, you might want
  #! to change this and deploy the SQLInstance into the same region as the
  #! cluster, to avoid traffic going across regions.
  region: europe-west6
  rootPassword:
    valueFrom:
      secretKeyRef:
        key: password
        name: sql-admin-creds
  settings:
    tier: db-g1-small
    ipConfiguration:
      authorizedNetworks:
      - name: cluster-NAT-IP
        #! Update this value with your NAT IP address in CIDR notation (e.g. 8.8.8.8/3
2). See above.
        value: <NAT-IP>
      ipv4Enabled: true
---
```

```
apiVersion: sql.cnrm.cloud.google.com/v1beta1
kind: SQLDatabase
metadata:
  name: sql-database
  namespace: service-instances
spec:
  charset: UTF8
  collation: en_US.UTF8
  instanceRef:
    name: sql-instance
---
apiVersion: sql.cnrm.cloud.google.com/v1beta1
kind: SQLUser
metadata:
  name: sql-user
  namespace: service-instances
spec:
  instanceRef:
    name: sql-instance
  password:
    valueFrom:
      secretKeyRef:
        key: password
        name: sql-user-creds
```

Once those objects are committed to the Kubernetes API, the Config Connector will cause the creation of those resources on GCP. This will take a short amount of time.

The three resources report their status and potential problems/errors back. If all goes well we should see all of those resources as “Ready” & “UpToDate” after a couple of minutes.

```
# kubectl -n service-instances get sqlinstance,sqldatabase,sqluser
NAME                                                 AGE     READY   STATUS     STATUS
 AGE
sqlinstance.sql.cnrm.cloud.google.com/sql-instance   3d20h   True    UpToDate   3d20h

NAME                                                 AGE     READY   STATUS     STATUS
 AGE
sqldatabase.sql.cnrm.cloud.google.com/sql-database   3d20h   True    UpToDate   3d20h

NAME                                         AGE     READY   STATUS     STATUS  AGE
sqluser.sql.cnrm.cloud.google.com/sql-user   3d20h   True    UpToDate   3d20h
```

You can also see this Cloud SQL instance in the Google Cloud Console.

Note

: Cloud SQL does not allow you to reuse the name of a deleted instance for a week. If you try to create a new `SQLInstance` with a name you have already used previously, you will see an error like

Note

[…] When you delete an instance, you can’t reuse the name of the deleted instance until one week from the deletion date. […]

You can use a different name for the `SQLInstance`; make sure to use replace that name in all examples going forward.

Create a Binding Specification compatible Secret for the database

As pointed out, none of the created objects are compatible with the Service Binding Specification. To help with that, we can create a secret which holds the data we need to know to connect to and use the Cloud SQL instance and which allows the platform to discover the fact that this instance can be “claimed” and “bound” to application workloads.

For this to be an automated process, we can use the `SecretTemplate` API of the secretgen controller. The secretgen controller needs to be able to read the resources created, thus we also need to deploy some RBAC rules to allow for that:

```
apiVersion: secretgen.carvel.dev/v1alpha1
kind: SecretTemplate
metadata:
  name: sql-instance-claimable
  namespace: service-instances
spec:
  inputResources:
  - name: sqlInstance
    ref:
      apiVersion: sql.cnrm.cloud.google.com/v1beta1
      kind: SQLInstance
      name: sql-instance
  - name: sqlDatabase
    ref:
      apiVersion: sql.cnrm.cloud.google.com/v1beta1
      kind: SQLDatabase
      name: sql-database
  - name: sqlUser
    ref:
      apiVersion: sql.cnrm.cloud.google.com/v1beta1
      kind: SQLUser
      name: sql-user
  - name: sqlUserSecret
    ref:
      apiVersion: v1
      kind: Secret
      name: $(.sqlUser.spec.password.valueFrom.secretKeyRef.name)
  serviceAccountName: sql-objects-reader
  template:
    data:
      password: $(.sqlUserSecret.data.password)
    metadata:
      labels:
        app.kubernetes.io/component: cloudsql-postgres
        app.kubernetes.io/instance: "$(.sqlInstance.metadata.name)"
        services.apps.tanzu.vmware.com/class: cloudsql-postgres
    stringData:
      database: $(.sqlDatabase.metadata.name)
      host: $(.sqlInstance.status.publicIpAddress)
      port: "5432"
      type: postgresql
      username: $(.sqlUser.metadata.name)
---
apiVersion: v1
kind: ServiceAccount
metadata:
  name: sql-objects-reader
  namespace: service-instances
---
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: sql-objects-reader
  namespace: service-instances
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: Role
  name: sql-objects-reader
subjects:
- kind: ServiceAccount
  name: sql-objects-reader
---
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: sql-objects-reader
  namespace: service-instances
rules:
- apiGroups:     [ "" ]
  resources:     [ "secrets" ]
  verbs:         &objReaderVerbs [ "get", "list", "watch" ]
  resourceNames: [ "sql-user-creds", "sql-admin-creds" ]
- apiGroups:     [ "sql.cnrm.cloud.google.com" ]
  resources:     [ "sqlinstances", "sqldatabases", "sqlusers" ]
  verbs:         *objReaderVerbs
  resourceNames: [ "sql-instance", "sql-database", "sql-user" ]
```

Verify

Find the name of the secret produced by reading the status of `SecretTemplate`. To do so, run:

```
kubectl get secrettemplate -n service-instances sql-instance-claimable -o jsonpath="{.
status.secret.name}"
```

Delete a CloudSQL service instance

Delete an CloudSQL service instance and all additional and related objects by running:

```
kubectl -n service-instances delete \
  sqlinstance/sql-instance \
  sqldatabase/sql-database \
  sqluser/sql-user \
  secrettemplate/sql-instance-claimable \
  password/sql-admin-creds \
  password/sql-user-creds \
  serviceaccount/sql-objects-reader \
  rolebinding/sql-objects-reader \
  roles/sql-objects-reader
```

Summary and Next Steps

You have learned how to use Carvel's `SecretTemplate` API to construct a secret that is compatible with the binding specification in order to create an Google CloudSQL service instance.

Now that you have this available in the cluster, you can learn how to make use of it by continuing where you left off in Consuming Google Cloud SQL on Tanzu Application Platform (TAP) with Config Connector.

Creating Google CloudSQL instances by using a Carvel package (experimental)

This topic describes how to create, update, and delete CloudSQL service instances using a Carvel package. For a more detailed and low-level alternative procedure, see Creating Service Instances that are compatible with Tanzu Application Platform.

Prerequisite

Meet the prerequisites and keep the following information to hand:

- `NAT-IP` - the cluster's egress NAT IP

The Package Repository and service instance Package Bundles for this guide can be found in the Reference Service Packages GitHub repository.

Create an CloudSQL service instance using a Carvel package

Follow the steps in the following procedures.

Add a reference package repository to the cluster

To add a reference package repository to the cluster:

1. Use the Tanzu CLI to add the new Service Reference packages repository:

   ```
   tanzu package repository add tap-reference-service-packages \
     --url ghcr.io/vmware-tanzu/tanzu-application-platform-reference-packages/tap-
   service-reference-package-repo:0.0.2 \
     -n tanzu-package-repo-global
   ```

2. Create a `ServiceAccount` that is used to provision `PackageInstall` resources by using the following example. The namespace of this `ServiceAccount` must match the namespace of the `tanzu package install` command in the next step.

   ```
   kubectl apply -f - <<'EOF'
   apiVersion: v1
   kind: ServiceAccount
   metadata:
     name: cloudsql-install
     namespace: service-instances
   ```

```
---
kind: Role
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: cloudsql-install
  namespace: service-instances
rules:
- apiGroups: ["sql.cnrm.cloud.google.com"]
  resources: ["sqlinstances","sqldatabases","sqlusers"]
  verbs:     ["*"]
- apiGroups: ["secretgen.carvel.dev", "secretgen.k14s.io"]
  resources: ["secrettemplates","passwords"]
  verbs:     ["*"]
- apiGroups: [""]
  resources: ["serviceaccounts","configmaps"]
  verbs:     ["*"]
- apiGroups: ["rbac.authorization.k8s.io"]
  resources: ["roles","rolebindings"]
  verbs:     ["*"]
---
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: cloudsql-install
  namespace: service-instances
subjects:
- kind: ServiceAccount
  name: cloudsql-install
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: Role
  name: cloudsql-install
EOF
```

Create a CloudSQL service instance through the Tanzu CLI

1. Create a file holding the configuration of the CloudSQL service instance:

```
cat <<'EOF' > demo-pg-instance-values.yml
---
name: demo-pg-instance
namespace: service-instances
allowedNetworks:
- name: service-instances-cluster-snat
  #! replace that with the cluster's egress IP, see NAT-IP in Prerequisite
  value: 34.65.178.24/32
EOF
```

Note

: To understand which settings are available for this package you can run:

```
tanzu package available get \
  --values-schema \
  psql.google.references.services.apps.tanzu.vmware.com/0.0.1-alph
a
```

> This shows a list of all configuration options you can use in the `demo-pg-instance-values.yml` file.
>
> : By default the package will create a claimable `Secret` which is labeled with `services.apps.tanzu.vmware.com/class: cloudsql-postgres`. While you can overwrite that by setting the `serviceInstanceLabels` setting above, you don't have to do that and it will still be aligned with the `ClusterInstanceClass` we will set up later.

2. Use the Tanzu CLI to install an instance of the reference service instance Package.

```
tanzu package install demo-pg-instance \
   --package-name psql.google.references.services.apps.tanzu.vmware.com \
   --version 0.0.1-alpha \
   --namespace service-instances \
   --service-account-name cloudsql-install \
   --values-file demo-pg-instance-values.yml \
   --wait
```

You can install the `psql.google.references.services.apps.tanzu.vmware.com` package multiple times to produce multiple CloudSQL Service instances. For that you need to prepare a separate `<INSTANCE-NAME>-values.yml` and then install the package with a different name and the above mentioned separate data values file for each CloudSQL service instance.

Verify

1. Verify the creation status for the CloudSQL instance by inspecting the conditions in the Kubernetes API. To do so, run:

```
kubectl get sqlinstance demo-pg-instance -n service-instances -o yaml
```

2. After some time has passed, sometimes up to 20 minutes, you are able to find the binding-compliant secret produced by `PackageInstall`. To do so, run:

```
kubectl get secrettemplate demo-pg-instance -n service-instances -o jsonpath="{
.status.secret.name}"
```

Delete a CloudSQL service instance

To delete the CloudSQL service instance run:

```
tanzu package installed delete demo-pg-instance -n service-instances
```

Summary

You have learned how to use Carvel's `Package` and `PackageInstall` APIs to create a CloudSQL service instance. If you want to learn more about the pieces that comprise this service instance package, see Creating Google CloudSQL Instances manually using kubectl.

Now that you have this available in the cluster, you can learn how to make use of it by continuing where you left off in [Consuming Google Cloud SQL on Tanzu Application Platform (TAP) with Config Connector][create-class].

Consuming GCP CloudSQL on Tanzu Application Platform with Crossplane

Introduction

This topic demonstrates how to use Services Toolkit to allow Tanzu Application Platform workloads to consume GCP CloudSQL PostgreSQL databases. This particular guide makes use of Crossplane to manage CloudSQL instances in GCP. As such, it can be thought of as an alternative approach to Consuming Google Cloud SQL on Tanzu Application Platform (TAP) with Config Connector to achieve the same outcomes.

Prerequisites

Meet these prerequisites:

- Create a Kubernetes cluster that supports running both Tanzu Application Platform and Crossplane
- Install Tanzu Application Platform (v1.2+) on the Kubernetes cluster
- Install gcloud CLI

Install Crossplane

Note: For the latest steps for installing Crossplane, see these instructions. For the instructions in this topic, it is important to enable support for external secret stores in Crossplane. This is currently an Alpha feature. As such, you will have to explicitly set command line flag `--enable-external-secret-stores` when starting the Crossplane controller.

Run the following commands to install Crossplane to your existing Kubernetes cluster:

```
kubectl create namespace crossplane-system

helm repo add crossplane-stable https://charts.crossplane.io/stable
helm repo update

helm install crossplane --namespace crossplane-system crossplane-stable/crossplane \
  --set 'args={--enable-external-secret-stores}'
```

For this topic, you do not need to install the Crossplane CLI or any additional configuration package.

Install GCP Provider for Crossplane

To install the GCP Provider for Crossplane, run:

```
kubectl apply -f -<<EOF
---
apiVersion: pkg.crossplane.io/v1
kind: Provider
```

```
metadata:
  name: crossplane-provider-gcp
spec:
  package: crossplane/provider-gcp:v0.21.0
EOF
```

After you have installed the provider, you see a new `cloudsqlinstances.database.gcp.crossplane.io` API resource available in your Kubernetes cluster. See the health of the installed provider by running:

```
kubectl get provider.pkg.crossplane.io crossplane-provider-gcp
```

Configure GCP Provider

This section creates a new GCP Service Account and gives it permissions to manage CloudSQL databases which are necessary to use Crossplane to manage CloudSQL instances.

1. Create a new GCP ServiceAccount, give it `Cloud SQL Admin` and create a key file:

   ```
   PROJECT_ID=<GCP Project ID>
   SA_NAME=crossplane-cloudsql

   gcloud iam service-accounts create "${SA_NAME}" --project "${PROJECT_ID}"
   gcloud projects add-iam-policy-binding "${PROJECT_ID}" \
     --role="roles/cloudsql.admin" \
     --member "serviceAccount:${SA_NAME}@${PROJECT_ID}.iam.gserviceaccount.com"
   gcloud iam service-accounts keys create creds.json --project "${PROJECT_ID}" --
   iam-account "${SA_NAME}@${PROJECT_ID}.iam.gserviceaccount.com"
   ```

2. Create a new secret from the key file by running:

   ```
   kubectl create secret generic gcp-creds -n crossplane-system --from-file=creds=
   ./creds.json
   ```

3. Delete the key file by running:

   ```
   rm -f creds.json
   ```

4. Configure the GCP provider to use the newly created secret by running:

   ```
   kubectl apply -f -<<EOF
   apiVersion: gcp.crossplane.io/v1beta1
   kind: ProviderConfig
   metadata:
     name: default
   spec:
     projectID: ${PROJECT_ID}
     credentials:
       source: Secret
       secretRef:
         namespace: crossplane-system
         name: gcp-creds
         key: creds
   EOF
   ```

Define Composite Resource Types

Now that the GCP provider for Crossplane has been installed and configured, create a new `CompositeResourceDefinition` (XRD) and corresponding `Composition` representing individual instances of CloudSQL Postgresql. For more information about these concepts see the Crossplane Composition documentation.

Note: Instead of creating your own custom XRD and Composition as shown below, you can also install an existing Crossplane configuration package for GCP that includes pre-configured XRDs and compositions for CloudSQL. The primary reason for creating a new XRD and composition from scratch is to make sure the connection secrets for newly provisioned CloudSQL Postgresql instances support the Service Binding Specification for Kubernetes and automatic Spring Boot configuration using Spring Cloud Bindings.

1. Create a new XRD by running:

```
kubectl apply -f -<<EOF
---
apiVersion: apiextensions.crossplane.io/v1
kind: CompositeResourceDefinition
metadata:
  name: xpostgresqlinstances.bindable.database.example.org
spec:
  claimNames:
    kind: PostgreSQLInstance
    plural: postgresqlinstances
  connectionSecretKeys:
  - type
  - provider
  - host
  - port
  - database
  - username
  - password
  group: bindable.database.example.org
  names:
    kind: XPostgreSQLInstance
    plural: xpostgresqlinstances
  versions:
  - name: v1alpha1
    referenceable: true
    schema:
      openAPIV3Schema:
        properties:
          spec:
            properties:
              parameters:
                properties:
                  storageGB:
                    type: integer
                required:
                - storageGB
                type: object
            required:
            - parameters
            type: object
        type: object
    served: true
EOF
```

After the newly created XRD has been successfully reconciled, there are two new API resources available in your Kubernetes cluster, `xpostgresqlinstances.bindable.database.example.org` and `postgresqlinstances.bindable.database.example.org`. The XRD created is agnostic to the underlying cloud managed service, so could also be fulfilled by a Composition that makes use of AWS RDS Postgresql or Azure Database for PostgreSQL.

2. Create a corresponding composition (not in a production environment) by running:

```
kubectl apply -f -<<EOF
---
apiVersion: apiextensions.crossplane.io/v1
kind: Composition
metadata:
  labels:
    provider: gcp
  name: xpostgresqlinstances.bindable.gcp.database.example.org
spec:
  compositeTypeRef:
    apiVersion: bindable.database.example.org/v1alpha1
    kind: XPostgreSQLInstance
  publishConnectionDetailsWithStoreConfigRef:
    name: default
  resources:
  - base:
      apiVersion: database.gcp.crossplane.io/v1beta1
      kind: CloudSQLInstance
      spec:
        forProvider:
          databaseVersion: POSTGRES_14
          region: us-central1
          settings:
            dataDiskType: PD_SSD
            ipConfiguration:
              authorizedNetworks:
              - value: 0.0.0.0/0 # not recommended for production deployments!
              ipv4Enabled: true
            tier: db-custom-1-3840
        writeConnectionSecretToRef:
          namespace: crossplane-system
    connectionDetails:
    - name: type
      value: postgresql
    - name: provider
      value: gcp
    - name: database
      value: postgres
    - fromConnectionSecretKey: username
    - fromConnectionSecretKey: password
    - name: host
      fromConnectionSecretKey: endpoint
    - name: port
      type: FromValue
      value: "5432"
    name: cloudsqlinstance
    patches:
    - fromFieldPath: metadata.uid
      toFieldPath: spec.writeConnectionSecretToRef.name
      transforms:
      - string:
          fmt: '%s-postgresql'
          type: Format
        type: string
      type: FromCompositeFieldPath
    - fromFieldPath: spec.parameters.storageGB
      toFieldPath: spec.forProvider.settings.dataDiskSizeGb
      type: FromCompositeFieldPath
EOF
```

The composition defined above makes sure that all CloudSQL Postgresql instances are placed in the `us-central1` region. This composition fulfils the XRD previously created by creating GCP CloudSQL databases.

Warning: The authorized network CIDR `0.0.0.0/0` provided above, will allow access to the Cloud SQL from any IP and is not recommended in a production environment.

Create an Instance Class

In order to make instances of a service easily discoverable and claimable by application operators, the role of the service operator creates a `ClusterInstanceClass`. In this particular example, the class states that claimable instances of CloudSQL Postgresql are represented by secret objects of type `connection.crossplane.io/v1alpha1` with label `services.apps.tanzu.vmware.com/class` set to `cloudsql-postgres`:

```
kubectl apply -f -<<EOF
---
apiVersion: services.apps.tanzu.vmware.com/v1alpha1
kind: ClusterInstanceClass
metadata:
  name: cloudsql-postgres
spec:
  description:
    short: GCP CloudSQL Postgresql database instances
  pool:
    kind: Secret
    labelSelector:
      matchLabels:
        services.apps.tanzu.vmware.com/class: cloudsql-postgres
    fieldSelector: type=connection.crossplane.io/v1alpha1
EOF
```

In addition, you need to grant sufficient RBAC permissions to Services Toolkit to be able to read the secrets specified by the class.

```
kubectl apply -f -<<EOF
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: stk-secret-reader
  labels:
    servicebinding.io/controller: "true"
rules:
- apiGroups:
  - ""
  resources:
  - secrets
  verbs:
  - get
  - list
  - watch
EOF
```

Provision GCP CloudSQL Postgresql Instance

Playing the role of the Service Operator, you now provision an instance of GCP CloudSQL Postgresql using the `postgresqlinstances.bindable.database.example.org` API managed by the XRD you previously created. Note that `.spec.publishConnectionDetailsTo` provides Crossplane with the name and a label for the secret that is being used to store the connection details for the newly created database. You can see that the label specified here matches the label selector defined on the `ClusterInstanceClass` you created in the previous step.

Run the following command:

```
kubectl apply -f -<<EOF
---
apiVersion: bindable.database.example.org/v1alpha1
kind: PostgreSQLInstance
metadata:
  name: cloudsql-postgres-db
  namespace: default
spec:
  parameters:
    storageGB: 20
  compositionSelector:
    matchLabels:
      provider: gcp
  publishConnectionDetailsTo:
    name: cloudsql-postgres-db
    metadata:
      labels:
        services.apps.tanzu.vmware.com/class: cloudsql-postgres
EOF
```

Running this command will cause the creation of a CloudSQL database instance in your GCP account. You can use the gcloud CLI to verify this:

```
gcloud sql instances list
```

After the instance has been successfully created in GCP, the status of the newly created `PostgreSQLInstance` resource should show `READY=True`. This might take a few minutes. You can wait for this by running:

```
kubectl wait --for=condition=Ready=true postgresqlinstances.bindable.database.example.
org cloudsql-postgres-db --timeout=10m
```

As soon as the CloudSQL Postgresql instance is ready, it is claimable by the role of the application operator as shown in the next section.

Note: There is currently a bug in Crossplane 1.7.2 onwards with the `--enable-external-secret-stores` feature gate enabled where the controller will fail to clean up a local secret created by the field `.spec.publishConnectionDetailsTo` after the deletion of the claim. A workaround is to temporarily give the crossplane controller the necessary i.e. permissions:

```
kubectl create clusterrole crossplane-cleaner --verb=delete --resource=secrets
kubectl create clusterrolebinding crossplane-cleaner --clusterrole=crossplane-cleaner
--serviceaccount=crossplane-system:crossplane
```

Claim the CloudSQL Postgresql instance and connect to it from the Tanzu Application Platform Workload

Thanks to the previously created `ClusterInstanceClass`, secrets representing CloudSQL Postgresql instances can now be discovered and claimed by application operators through the Tanzu CLI as shown below.

1. Show available classes of service instances by running:

   ```
   tanzu service classes list

     NAME                 DESCRIPTION
     cloudsql-postgres    GCP CloudSQL Postgresql database instances
   ```

2. Show claimable instances belonging to the CloudSQL Postgresql class by running:

   ```
   tanzu services claimable list --class cloudsql-postgres

     NAME                    NAMESPACE  API KIND  API GROUP/VERSION
     cloudsql-postgres-db    default    Secret    v1
   ```

3. Create a claim for the discovered instance by running:

   ```
   tanzu service claim create cloudsql-claim \
     --resource-name cloudsql-postgres-db \
     --resource-kind Secret \
     --resource-api-version v1
   ```

4. Obtain the claim reference by running:

   ```
   tanzu service claim list -o wide
   ```

   Expect to see the following output:

   ```
   NAME                     READY  REASON  CLAIM REF
   cloudsql-claim           True           services.apps.tanzu.vmware.com/v1alpha1
   :ResourceClaim:cloudsql-claim
   ```

5. Create an application workload that consumes the claimed CloudSQL Postgresql database by running:

Example:

```
tanzu apps workload create my-workload \
  --git-repo https://github.com/sample-accelerators/spring-petclinic \
  --git-branch main \
  --git-tag tap-1.2 \
  --type web \
  --label app.kubernetes.io/part-of=spring-petclinic \
  --annotation autoscaling.knative.dev/minScale=1 \
  --env SPRING_PROFILES_ACTIVE=postgres \
  --service-ref db=services.apps.tanzu.vmware.com/v1alpha1:ResourceClaim:clouds
ql-claim
```

Note that `--service-ref` is being set to the claim reference obtained previously.