Services Toolkit for VMware Tanzu Application Platform v0.6

Services Toolkit for VMware Tanzu Application Platform 0.6

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 6

Motivation 7

Component Overview 8

- Resource Claims 8
- Service Offering 8
- Service API Projection & Resource Replication (experimental) 9
- Resource Classes 9

Release Notes 10

- v0.6.0 10
- v0.5.1 10
- v0.5.0 10
  - Breaking changes 10

Install 12

Getting Started 13

Uninstall 14

Service API Projection and Service Resource Replication for VMware Tanzu 15

- Install 15
- Terminology 15
- Concepts 15
- Resources 15
  - Projection Plane 15
    - UpstreamClusterLink and DownstreamClusterLink 15
  - API Projection 17
    - APIExportRoleBinding 17
    - ClusterAPIGroupImport 17
    - APIResourceImport 18
  - Resource Replication 18
    - SecretExport 19

SecretImport 19
ClusterResourceImportMonitor 20
ResourceImportMonitorBinding 20
ClusterResourceExportMonitor 21
ResourceExportMonitorBinding 21

Service Offering for VMware Tanzu 23
Install 23
Terminology 23
Resources 23
ClusterResource 23
GVKDescriptor (duck type) 24
ClusterExampleUsage (GVKDescriptor) 24
Scope, Discoverability and Usability 24
RBAC Rules for Discoverability 25

Service Resource Claims 26
Install 26
Terminology 26
Resources 26
ResourceClaim 26
ResourceClaimPolicy 27
Permissions (RBAC) 27

Services Plug-in for Tanzu CLI 29
Use Cases 29
Discovery of Service Types 29
Listing Service Instances 29
Claiming Service Instances with Resource Claims 29
Listing and getting Resource Claims 30
Unclaiming Service Instances 30

Reference 31

Resource Requirements 31
Deployments 31

Known Limitations 33
Service Resource Replication Limitations 33
Limitation 1: Updates to Secrets are not automatically replicated 33
Service API Projection Limitations 33

Limitation 1: CRD and Aggregation layer conflict 33
Behaviour when local CRD is created before Service Resource API has been projected 33
Behaviour when local CRD is created after Service Resource API is projected 34
Limitation 2: No built-in support for cluster-scoped requests against projected APIs in the Workload Cluster 35
Service Resource Claims Limitations 36
Limitation 1: Can only claim service resources that adhere to the Kubernetes Binding specification 36
Limitation 2: Can only claim service resources once 36
Supported Kubernetes Distributions 36
Topology 36
Supported Topologies 37
Provide a Service Resource Lifecycle API 37
From one Service cluster to one Workload cluster 37
From a Service cluster to multiple Workload clusters 38
Provide different Service Resource Lifecycle APIs 39
From a Service cluster to a Workload cluster 39
Provide multiple Service Resource Lifecycle APIs 40
From a Service Cluster to a Workload cluster 40
From multiple Service Clusters to one Workload cluster 41
From multiple Service Clusters to multiple Workload clusters 42
User Roles 43
Service Author (SA) 44
Service Operator (SO) 44
Application Operator (AO) 44
Application Developer (AD) 44
Advanced Use Cases 44
Direct Secret References 44
Dedicated Service Clusters (using experimental Projection and Replication APIs) 46
Pre-Requisites 46
Walkthrough 47

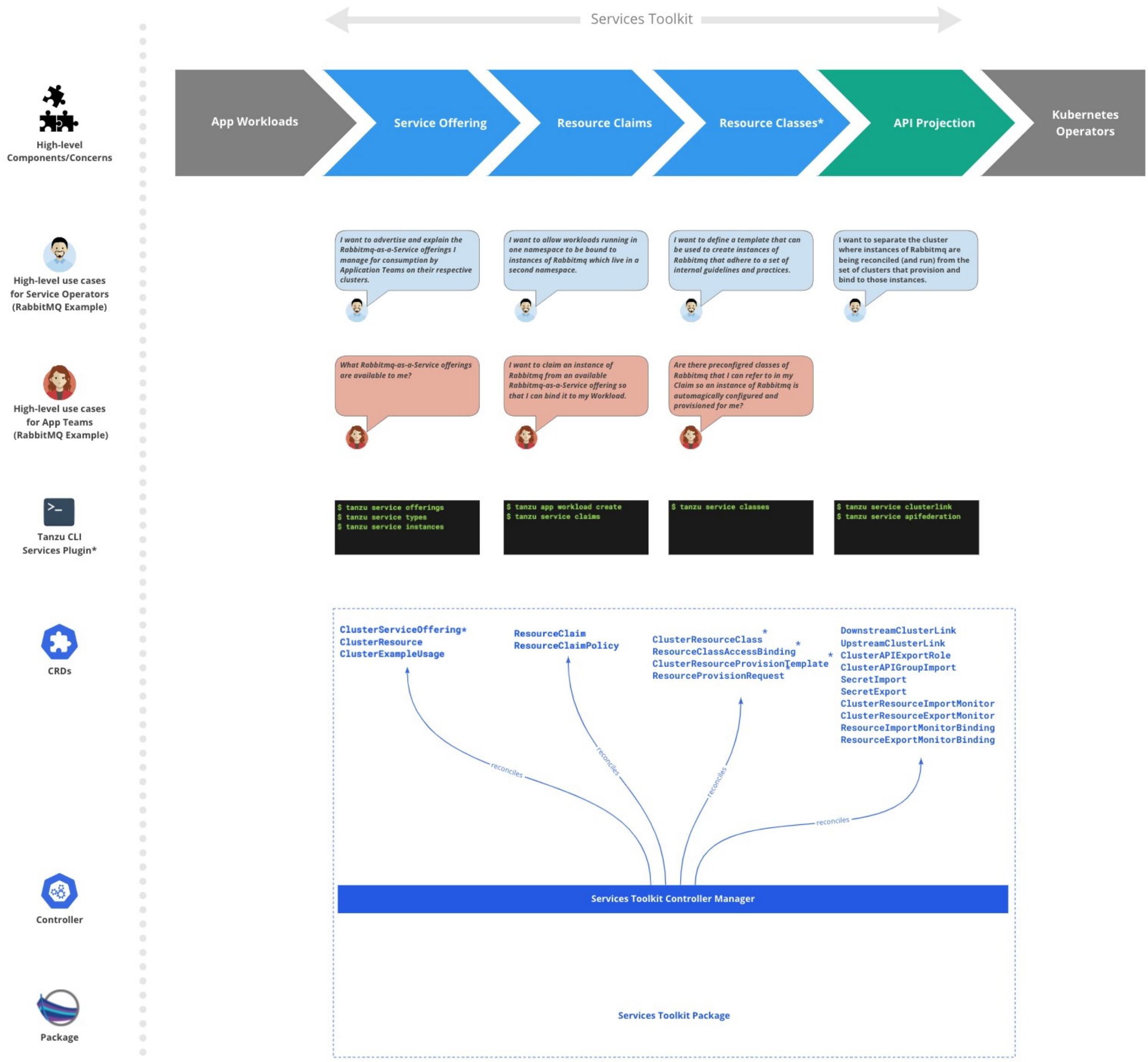
About Services Toolkit

Services Toolkit (“STK”) is a collection of Kubernetes native components supporting the discoverability, lifecycle management (CRUD) and connectivity of Service Resources (databases, message queues, DNS records, etc.) on Kubernetes.

The toolkit is currently comprised of the following components:

1. Resource Claims
2. Service Offering
3. Service API Projection (experimental)
4. Resource Replication (experimental)
5. Resource Classes (coming soon)

Each component has value independent of the others, however the most powerful and valuable use cases are unlocked by combining them together in unique and interesting ways.



* indicates item is on the roadmap. No concrete design available yet. Early prototype and/or proposal might exist.

Motivation

Application teams need supporting Service Resources (e.g. databases, message queues, DNS records, etc.) to develop and run their applications. They do not want the burden of having to run these services themselves, so often organizations provide ticketing systems that allow Application teams to make manual 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 can be provisioned with simple API calls, for example RDS. Services Toolkit aims to provide a set of modular tools that can be used to provide a similar self-service experience to that of the cloud for Service Resources running on Tanzu.

Component Overview

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

Resource Claims

Resource Claims allows 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, e.g. type, protocol, provider, version. The claim is then fulfilled against an existing (or in the future automatically created) Service Resource using rules decided by the Service Operator. This allows Application teams to focus on their application and its dependencies.

To learn more about Resource Claims, see Resource Claims.

Service Offering

In order 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 do already define some metadata, however, this only consists of Kubernetes level API descriptions, e.g. name, field descriptions. While this metadata is useful, Application Operators require more holistic information covering details such as service level management capabilities, QoS guarantees, relationships between different resource types the API exposes, as well as other information that aids discovery by Application Operators and higher-level tooling aimed at that role (e.g. keywords, icons, etc).

It is worth noting that some metadata surfaced by Service Description and Offering relate not only to the Service Resource Lifecycle API itself, but also 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. As an example, a Service Resource that is concerned with 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, we consider it the responsibility of the Service Operator to make sure that the right level of accurate metadata has been specified for a given Service Resource. Service Description and

Offering enables associating metadata with Service Resources and surfacing it to Application Operators. This metadata can be provided by Service Operator or, for infrastructure agnostic metadata (e.g. data that describes the relationships between different API resource types), by Service Authors.

To learn more about Service Offering, see Service Offering.

Service API Projection & Resource Replication (experimental)

We also believe Application and Service infrastructure should be separated, and we have observed customers doing this in production environments. A few examples of the benefits of this segmentation of infrastructure are:

- Dedicated cluster requirements for workload or service clusters. For example, Service clusters may need access to SSDs.
- Different cluster lifecycle management. Upgrades to Service clusters may occur more cautiously.
- Unique Compliance requirements. As data is stored on a Service cluster it may have different compliance needs.
- 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 multi-cluster world is to split clusters into Application Workload clusters and Service clusters, 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.

Resource Classes

Coming soon.

Release Notes

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` has been 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.
- `tanzu services` CLI plug-in 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 a issue that caused `kapp-controller` to unnecessarily reconcile continuously.
- `tanzu services` CLI plug-in 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.
- Do not block claim deletion when not able to 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` will need to be updated to remove the old finalizer in order 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`.

Install

Services Toolkit is packaged and distributed using the carvel set of tools. The Services Toolkit carvel Package is currently published to the Tanzu Application Platform Package Repository. It can be installed either as part of a wider Tanzu Application Platform installation (see here) or as an individual Package on its own (see here).

Getting Started

The quickest and easist way to get started with Services Toolkit is to experience it as part of Tanzu Application Platform. A comprehensive walkthrough demonstrating the main use cases, tools and APIs powered by the toolkit is published in Tanzu Application Platform's Getting Started Guide, which can be found here.

In addition, a number of additional Use Cases can be found in Use Cases.

Uninstall

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

Service API Projection and Service Resource Replication for VMware Tanzu

Install

See the documentation on installing the latest release of the Services Toolkit to get started and refer to Topology for information on supported topologies.

Terminology

- **Service Resources** - Things like databases, message queues, caches, DNS records, firewall rules, virtual networks, etc.
- **Service Resource Lifecycle API** - Any Kubernetes API that can be used to manage the lifecycle (CRUD) of a Service Resource.
- **Service Cluster** - A Kubernetes cluster that has Service Resource Lifecycle APIs installed and a corresponding controller managing their lifecycle.
- **Workload Cluster** - A Kubernetes cluster that has developer-created applications running on it.

Concepts

This document introduces a number of concepts. These are briefly summarised below:

- **Projection Plane** - Defines an “upstream” and “downstream” relationship between a pair of Kubernetes clusters, namely between a Service Cluster (upstream) and a Workload Cluster (downstream).
- **API Projection** - Makes **custom** Kubernetes APIs installed on a Service Cluster (upstream) available in a Workload Cluster (downstream).
- **Resource Replication** - Synchronizes **core** Kubernetes resources across Kubernetes clusters.

Resources

Projection Plane

UpstreamClusterLink and DownstreamClusterLink

The `UpstreamClusterLink` resource is created on a Service Cluster. Its main purpose is to manage a Service Account that will be used by components running in a Workload Cluster.

```
apiVersion: projection.apiresources.multicluster.x-tanzu.vmware.com/v1alpha1
kind: UpstreamClusterLink
metadata:
  name: workload-3c
  namespace: services-toolkit
spec:
  downstream:
    # Name of the Workload Cluster. This will be used for debugging.
    name: workload-3c
status:
  # Created Service Account that will be used by the Workload Cluster
  serviceAccount:
    name: managed-service-account
  observedGeneration: 1
  conditions:
  - lastTransitionTime: "2021-02-02T18:41:22Z"
    status: "True"
    type: Ready
  - lastTransitionTime: "2021-02-02T18:41:22Z"
    status: "True"
    type: ServiceAccountReady
```

The `DownstreamClusterLink` resource is created on a Workload Cluster. Its primary purpose is to manage an API aggregation server that will eventually be used to project specific APIs. This resource does the following:

- Contains information about the corresponding Service Cluster - url, name, ca cert and service account token.
- Deploys the API-aggregation server that is configured to proxy to the Service Cluster using the provided service account token.

```
apiVersion: projection.apiresources.multicluster.x-tanzu.vmware.com/v1alpha1
kind: DownstreamClusterLink
metadata:
  name: services-2b
  namespace: services-toolkit
spec:
  proxy:
    TLS:
      # TLS cert to be used for the API proxy
      secretName: omnia-isla
  upstream:
    kubeconfig:
      # Secret containing the kubeconfig to connect to the Service Cluster
      secretName: pumpkin-seeds
    name: services-2b
status:
  proxy:
    # base64-encoded CA for the API proxy
    caBundle: facade0ff1cebadc0ffee...
    # Reference to the kubernetes Service providing access to the API proxy
    serviceReference:
      name: services-2b-proxy
      namespace: services-toolkit
      port: 443
  conditions:
```

```
  - lastTransitionTime: "2021-02-02T18:41:22Z"
    status: "True"
    type: Ready
  - lastTransitionTime: "2021-02-02T18:41:22Z"
    status: "True"
    type: ServiceAccountReady
  - lastTransitionTime: "2021-02-02T18:41:22Z"
    status: "True"
    type: ProxyDeploymentReady
  - lastTransitionTime: "2021-02-02T18:41:22Z"
    status: "True"
    type: ProxyServiceReady
  - lastTransitionTime: "2021-02-02T18:41:22Z"
    status: "True"
    type: ProxyConfigMapReady
  - lastTransitionTime: "2021-02-02T18:41:22Z"
    status: "True"
    type: ProxyServiceAccountReady
```

Note: the service account used by the proxy `Deployment` must have the following RBAC set up for it: * A `ClusteRoleBinding` to the `system:auth-delegator ClusterRole` to delegate auth decisions to the Kubernetes core API server. * A `RoleBinding` to the `extension-apiserver-authentication-reader` role in the `kube-system` namespace. This allows your extension api-server to access the extension-apiserver-authentication configmap. * A `ClusterRoleBinding` to a `ClusterRole` that provides “get”, “list” and “watch” for namespaces, if such a `ClusterRole` doesn’t exist you will need to create one.

API Projection

APIExportRoleBinding

The purpose of the `APIExportRoleBinding` is to provide downstream users with necessary permissions on the Upstream Cluster. It does so by binding a user-specified `ClusterRole` to the service account referred to in the provided `UpstreamClusterLink` resource.

```
apiVersion: projection.apiresources.multicluster.x-tanzu.vmware.com/v1alpha1
kind: APIExportRoleBinding
spec:
  upstreamClusterLinkRef:
    name: fish-sauce
    namespace: project-alpha
  clusterRoleRef:
    name: cluster-1-a
```

ClusterAPIGroupImport

The `ClusterAPIGroupImport` resource is a cluster-scoped resource created on the Workload Cluster. It expresses the intent to import an API group using the specified DownstreamClusterLink. Only one `ClusterAPIGroupImport` can exist per API Group.

Once created, if a corresponding `APIExportRole` exists in the Service Cluster, a new custom Kubernetes API will be available in the Workload Cluster and can be discovered via kubectl api-resources.

```
apiVersion: projection.apiresources.multicluster.x-tanzu.vmware.com/v1alpha1
kind: ClusterAPIGroupImport
metadata:
  name: rabbitmq.com
spec:
  # This is the reference to the DownstreamClusterLink resources
  downstreamClusterLinkRef:
    name: services-2b
    namespace: services-toolkit
  # The api group that is to be projected
  group: rabbitmq.com
  # Version of the api to be projected. Optional, if not specified register all discov
ered versions
  version: v1beta1
status:
  conditions:
  - type: Ready
    lastTransitionTime: "2020-12-01T13:03:32Z"
    status: "True"
  - type: APIServicesReady
    lastTransitionTime: "2020-12-01T13:03:28Z"
    status: "True"
```

APIResourceImport

The APIResourceImport resource is a namespace-scoped resource created on the downstream cluster. Its presence indicates to the proxy whether a projected Group and Resource is available in a given namespace. This information allows the proxy to decide if a particular request should be forwarded to upstream. It is worth noting this is for convenience rather than policy enforcement, which is achieved by the RBAC in upstream.

Resources are specified at the namespace scope rather than the cluster scope to allow different resources to be made available in different namespaces.

```
apiVersion: projection.apiresources.multicluster.x-tanzu.vmware.com/v1alpha1
kind: APIResourceImport
metadata:
  name: rabbitmq.com-import
  namespace: team-1 # namespace scoped resource as it sets up ns RBAC
 spec:
   clusterApiImportRef:
      name: rabbitmq.com
   resources: [“rabbitmqclusters”]
status:
  conditions:
  - type: Ready
    message: "Successfully reconciled"
    lastTransitionTime: "2020-12-01T13:03:30Z"
    status: "True"
  - type: ResourcesAvailable
    message: "Resources Ready"
    lastTransitionTime: "2020-12-01T13:03:32Z"
    status: "True"
```

Resource Replication

The resource replication components are responsible for synchronizing core kubernetes resources across multiple clusters. As of version v0.5.0, the resource replication only handles the `Secret` resources.

SecretExport

`SecretExport` is a namespaced resource indicating that the named Secret is involved in the replication process. Services toolkit will place these resources on the services cluster. This resource is used to set up permissions for the local service account, which will be used by the Workload Clusters when pulling the secret across.

```
apiVersion: replication.apiresources.multicluster.x-tanzu.vmware.com/v1alpha1
kind: SecretExport
metadata:
  name: small-postgres-23.status.binding.name
  namespace: project-1
  labels:
    # The following labels will be applied automatically
    # to help with filtering and searching of SecretExport resources
    replication.apiresources.multicluster.x-tanzu.vmware.com/secret-owner-group: sql.t
anzu.vmware.com
    replication.apiresources.multicluster.x-tanzu.vmware.com/secret-owner-version: v1
    replication.apiresources.multicluster.x-tanzu.vmware.com/secret-owner-kind: Postgr
es
    replication.apiresources.multicluster.x-tanzu.vmware.com/secret-owner-name: small-
postgres-23
    replication.apiresources.multicluster.x-tanzu.vmware.com/secret-owner-uid: cafe012
3d09e
    replication.apiresources.multicluster.x-tanzu.vmware.com/monitor-binding-uid: 0ff1
ceca5cade
spec:
  secret:
    # The name of the secret in the current namespace to be replicated
    name: pg-binding
  serviceAccount:
    # The name of the service account in the current namespace that will be used for r
eplication
    name: upstream-replication-sa
```

SecretImport

SecretImport is responsible for replicating the secret from the Service Cluster. Services Toolkit places the `SecretImport` in a user namespace of the Workload Cluster for each secret. Currently, the namespace on the Service Cluster has to match the namespace on the Workload Cluster.

```
apiVersion: replication.apiresources.multicluster.x-tanzu.vmware.com/v1alpha1
kind: SecretImport
metadata:
  namespace: project-1
  name: small-postgres-23.status.binding.name
  labels:
    # The following labels will be applied automatically
    # to help with filtering and searching of SecretImport resources
    replication.apiresources.multicluster.x-tanzu.vmware.com/secret-owner-group: sql.t
anzu.vmware.com
```

```
    replication.apiresources.multicluster.x-tanzu.vmware.com/secret-owner-version: v1
    replication.apiresources.multicluster.x-tanzu.vmware.com/secret-owner-kind: Postgr
es
    replication.apiresources.multicluster.x-tanzu.vmware.com/secret-owner-name: small-
postgres-23
    replication.apiresources.multicluster.x-tanzu.vmware.com/secret-owner-uid: cafe012
3d09e
    replication.apiresources.multicluster.x-tanzu.vmware.com/monitor-binding-uid: 0b5e
55ed90dde55
spec:
  secret:
    # The name of the secret in the current namespace to be replicated
    name: dumbo
  remoteKubeconfig:
    # The name of a secret in the current namespace holding a kubeconfig for the Servi
ce Cluster
    name: energy-source
```

The two resources above handle a single `Secret` object replication. In order to automatically set up replication of the specified secrets for every service instance of a given type, cluster-scoped resources `ClusterResourceImportMonitor` and `ClusterResourceExportMonitor` are used. Additionally, `ResourceImportMonitorBinding` and `ResourceExportMonitorBinding` are used to enable automatic replication in a given namespace, and specify the connection details for replication for this namespace.

ClusterResourceImportMonitor

`ClusterResourceImportMonitor` is responsible for setting up watching on service instances, so that as a result, `SecretImport` resources could be produced when needed. `ClusterResourceImportMonitor` resources are defined on the Workload Cluster.

```
apiVersion: replication.apiresources.multicluster.x-tanzu.vmware.com/v1alpha1
kind: ClusterResourceImportMonitor
metadata:
  name: postgres
  labels:
    # The following labels are required and must match the values in spec.resource
    replication.apiresources.multicluster.x-tanzu.vmware.com/monitored-resource-group:
 sql.tanzu.vmware.com
    replication.apiresources.multicluster.x-tanzu.vmware.com/monitored-resource-versio
n: v1
    replication.apiresources.multicluster.x-tanzu.vmware.com/monitored-resource-kind:
Postgres
spec:
  # The type of the resource owning the secrets to be replicated
  resource:
    group: sql.tanzu.vmware.com
    version: v1
    kind: Postgres
  # The list of secrets to be replicated expressed as JSON path on the resource
  secretPaths:
  - .status.binding.name
```

ResourceImportMonitorBinding

By default, defining an `ClusterResourceImportMonitor` resource configures the resource type and secrets to be replicated, but does not enable replication. `ResourceImportMonitorBinding` is used to enable the replication of secrets for service instances within a given namespace. It references a secret containing the kubeconfig of the Service Cluster to pull the secrets from.

```
apiVersion: replication.apiresources.multicluster.x-tanzu.vmware.com/v1alpha1
kind: ResourceImportMonitorBinding
spec:
  monitorRef:
    # Name of the related cluster-scoped ClusterResourceImportMonitor
    name: postgres
  remoteKubeconfig:
    # The name of a secret in the current namespace holding a kubeconfig for the Servi
ce Cluster
    name: energy-source
```

ClusterResourceExportMonitor

`ClusterResourceExportMonitor` is responsible for setting up watching on service instances, so that as a result, `SecretExport` resources could be produced when needed. `ClusterResourceExportMonitor` resources are defined on the services cluster.

```
apiVersion: replication.apiresources.multicluster.x-tanzu.vmware.com/v1alpha1
kind: ClusterResourceExportMonitor
metadata:
  name: postgres
  labels:
    # The following labels are required and must match the values in spec.resource
    replication.apiresources.multicluster.x-tanzu.vmware.com/monitored-resource-group:
 sql.tanzu.vmware.com
    replication.apiresources.multicluster.x-tanzu.vmware.com/monitored-resource-versio
n: v1
    replication.apiresources.multicluster.x-tanzu.vmware.com/monitored-resource-kind:
Postgres
spec:
  # The type of the resource owning the secrets to be replicated
  resource:
    group: sql.tanzu.vmware.com
    version: v1
    kind: Postgres
  # The list of secrets to be replicated expressed as JSON path on the resource
  secretPaths:
  - .status.binding.name
```

ResourceExportMonitorBinding

By default, defining an `ClusterResourceExportMonitor` resource configures the resource type and secrets to be replicated, but does not enable replication. `ResourceExportMonitorBinding` is used to enable the replication of secrets for service instances within a given namespace. It provides the service account in the current namespace of the Service Cluster to pull the secrets from.

```
apiVersion: replication.apiresources.multicluster.x-tanzu.vmware.com/v1alpha1
kind: ResourceExportMonitorBinding
metadata:
```

```
  name: cluster1-postgres
  namespace: project-1
spec:
  monitorRef:
    # Name of the related cluster-scoped ClusterResourceImportMonitor
    name: postgres
  serviceAccount:
    # Name of the service account in the current namespace used by the Workload Cluste
r to pull secrets.
    name: upstream-replication-sa
```

Service Offering for VMware Tanzu

Install

See the documentation on installing the latest release of the Services Toolkit to get started.

Terminology

- **Service Resources** - Things like databases, message queues, caches, DNS records, firewall rules, virtual networks, etc.
- **Service Resource Lifecycle API** - Any Kubernetes API that can be used to manage the lifecycle (CRUD) of a Service Resource.

Resources

ClusterResource

The ClusterResource CR is a place to store metadata regarding a Service Resource Lifecycle API. The only required field is `.spec.resourceRef`, which defines the Kubernetes API Group and Kind that a given ClusterResource CR is describing.

```
apiVersion: services.apps.tanzu.vmware.com/v1alpha1
kind: ClusterResource
metadata:
  name: rabbitmqcluster
  labels:
    # The following labels will be applied automatically by the ClusterResource contro
ller
    # to help with filtering and searching of ClusterResource resources
    services.apps.tanzu.vmware.com/api-group: rabbitmq.com
    services.apps.tanzu.vmware.com/api-kind: RabbitmqCluster
spec:
  # A reference to the Kubernetes API Group and Kind that this ClusterResource is desc
ribing
  resourceRef:
    # The Kubernetes API Group the resource belongs to
    group: rabbitmq.com
    # The Kubernetes API Kind of the resource
    kind: RabbitmqCluster
  # Short description of the resource (optional; string)
  shortDescription: "It's a RabbitMQ Cluster"
  # Long description of the resource (optional; string)
  longDescription: "RabbitMQ is an open source ..."
```

Note that metadata stored in ClusterResource CRs is not specific to a particular version of the API. Version-specific API metadata is stored in GVKDescriptor CRs.

GVKDescriptor (duck type)

GVKDescriptor is not a concrete CRD itself, but rather a duck type of the following shape:

```
apiVersion: group/version
kind: Kind
spec:
  # A reference to the Kubernetes API Group/Version/Kind
  gvkRef:
    # The Kubernetes API Group the resource belongs to
    group: rabbitmq.com
    # The Kubernetes API Version of the API
    version: v1beta1
    # The Kubernetes API Kind of the resource
    kind: RabbitmqCluster
```

Any CR that contains `.spec.gvkRef` with the `group`, `version` and `kind` fields can be considered an GVKDescriptor.

ClusterExampleUsage (GVKDescriptor)

ClusterExampleUsage CR adheres to the GVKDescriptor duck type and is used to store a YAML document for a given Service Resource LifecycleAPI.

```
apiVersion: services.apps.tanzu.vmware.com/v1alpha1
kind: ClusterExampleUsage
metadata:
  name: rabbitmqcluster-hello-world
  labels:
    # The following labels will be applied automatically by the ClusterExampleUsage co
ntroller
    # to help with filtering and searching of ClusterExampleUsage resources
    services.apps.tanzu.vmware.com/api-group: rabbitmq.com
    services.apps.tanzu.vmware.com/api-kind: RabbitmqCluster
    services.apps.tanzu.vmware.com/api-version: v1beta1
spec:
  # Adherence to GVKDescriptor duck type
  gvkRef:
    group: rabbitmq.com
    version: v1beta1
    kind: RabbitmqCluster
  # Description of the example
  description: |
    "Hello World" example for the RabbitmqCluster resource
  # YAML document for the example
  yaml: |
    ---
    apiVersion: rabbitmq.com/v1beta1
    kind: RabbitmqCluster
    metadata:
      name: hello-world
    spec:
      ...
```

Scope, Discoverability and Usability

All Service Offering APIs are cluster scoped meaning that, assuming relevant RBAC has been configured (see below), any user can get, list and watch CRs from these APIs. This configuration helps to support *discoverability*, in that just as any user can run `kubectl api-resources`, so they can run `kubectl get clusterresources`. The former command outputs all API resources on the server, while the latter outputs only the Service Resource Lifecycle APIs on the server (a subset).

Ability to discover Service Resource Lifecycle APIs does not automatically mean a user has permission to use the APIs. *Accessibility* of a given Service Resource Lifecycle API depends on whether the user has relevant RBAC permissions on the API that has been discovered.

RBAC Rules for Discoverability

By default Services Toolkit carvel package allows the `system:authenticated` Group to get, list and watch Service Offering resources via the ClusterRole `service-offering-api-discoverability`.

Service Resource Claims

Install

See the documentation on installing the latest release of the Services Toolkit to get started.

Terminology

- **Service Resource** - Represents a concrete resource that provides a certain service like databases, message queues, caches, DNS records, firewall rules, virtual networks, etc.
- **Service Bindings** Represents the intent of providing information about a well-known Service Resource object to a well-known Application.
- **Provisioned service** used to refer to any kubernetes object that adheres to the Provisioned Service duck type
- **Service Resource Claim** - Represents a request by an Application to use any Service Resource of a certain category as long as it satisfies a set of specified requirements

Resources

ResourceClaim

The main purpose of `ResourceClaim` is to identify the concrete Kubernetes object within the cluster that satisfies the requirements stated in the claim.

Once the object is identified the status condition `ResourceMatched` is set to true.
If the reference object adheres to the Provisioned Service duck type the `.status.binding.name` will be copied to the `ResourceClaim .status.binding.name` and the `ResourceClaimed` condition will be set to true. The claim object itself is a Provisioned Service, so it can be used to define a Service Binding.

ResourceClaims are currently exclusive. A Service Resource can only have ONE successfully claimed ResourceClaim in the cluster.

```
apiVersion: services.apps.tanzu.vmware.com/v1alpha1
kind: ResourceClaim
metadata:
  name: rmq-claim
  namespace: accounts
spec:
  ref:
    apiVersion: rabbitmq.com/v1alpha1
    kind: RabbitmqCluster
    name: my-rmq
```

```
    namespace: my-rmq-namespace # optional (if claiming across namespaces)
status:
  binding:
    name: my-rmq-secret # copied from RabbitmqCluster/my-rmq
  conditions:
    - lastTransitionTime: "2019-10-22T16:29:25Z"
      status: "True"
      type: Ready
    - lastTransitionTime: "2019-10-22T16:29:24Z"
      status: "True"
      type: ResourceClaimed
    - lastTransitionTime: "2019-10-22T16:29:23Z"
      status: "True"
      type: ResourceMatched
```

ResourceClaimPolicy

`ResourceClaimPolicy` enables `ResourceClaims` to work across namespaces.

The Policy refers to two pieces of information. Service Resources (e.g. RabbitmqClusters) that this policy applies to and which namespaces are allowed to claim these resources. * The matching Service Resources MUST reside in the same namespace as the `ResourceClaimPolicy` and their type must also be specified in `.spec.type`. * Namespaces that are allowed to claim these service resources must have their namespace name in the `.spec.consumingNamespaces` array. A value of `*` would allow claiming from ALL namespaces in this cluster.

```
apiVersion: services.apps.tanzu.vmware.com/v1alpha1
kind: ResourceClaimPolicy
metadata:
  name: rmq-policy
  namespace: my-rmq-namespace
spec:
  consumingNamespaces:
  - accounts # or "*" for all namespaces
  type:
    group: rabbitmq.com
    kind: RabbitmqCluster
```

Permissions (RBAC)

The `ResourceClaim` controller MUST have read access to Resources specified in the `ResourceClaim` spec. As these resources are not known upfront, the appropriate RBAC must be setup on the Cluster. To accomplish this RBAC must be setup using Aggregated ClusterRoles with the `resourceclaims.services.apps.tanzu.vmware.com/controller: "true"` label.

An example of a ClusterRole that allows `RabbitmqCluster` resources to be read by the `ResourceClaim` controller:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: resource-claims-rmq-role
  labels:
    resourceclaims.services.apps.tanzu.vmware.com/controller: "true"
```

```
rules:
- apiGroups:
  - rabbitmq.com
  resources:
  - rabbitmqclusters
  verbs:
  - get
  - list
  - watch
  - update
```

Services Plug-in for Tanzu CLI

Warning: The `services` plug-in is currently in ALPHA. Commands and arguments might change without notice.

The `services` plug-in improves the user experience of working with services on Tanzu Application Platform. After installation, the plug-in is invoked by using the `tanzu services` command.

The plug-in is currently distributed with Tanzu Application Platform. Please see here for information on how to acquire and install the plug-in.

Use Cases

The `services` plug-in is currently of most use to the Application Developer and Application Operator roles. See User Roles for more details. The following use cases are currently covered by the plug-in as documented below. We hope to unlock more use cases for the `services` plug-in in the near future.

Discovery of Service Types

Application Developers can discover the list of service types available on their target cluster by running `tanzu service types list`.

For further information including help text and usage, please run `tanzu service types list --help`.

Listing Service Instances

Application Developers can list existing Service Instances on their target cluster by running `tanzu Service Instances list`.

For further information including help text and usage, please run `tanzu Service Instances list --help`.

Claiming Service Instances with Resource Claims

Application Developers can claim Service Instances on their target cluster by running:

```
tanzu service claims create
CLAIM-NAME --resource-name SERVICE-INSTANCE-NAME --resource-kind SERVICE-INSTANCE-KIND
 --resource-api-version
SERVICE-INSTANCE-API-VERSION
```

Where:

- `CLAIM-NAME` is the desired name of the Resource Claim to be created and

- `SERVICE-INSTANCE-NAME`, `SERVICE-INSTANCE-KIND` and `SERVICE-INSTANCE-API-VERSION` are the name, kind and apiVersion, respectively, of the Service Instance to be claimed.
- `--resource-namespace` is an optional flag that can be passed in along with a namespace in order to claim a Service Instance in a different namespace.

For further information including help text and usage, please run `tanzu service claims create --help`.

Listing and getting Resource Claims

Application Developers can view existing claims on their target cluster by running `tanzu service claims list`. In addition, Application Developers can use this command to output Claim References by passing in `-o wide`, which can then be passed to the `--service-ref` flag of the `tanzu apps workload create` command in order to bind Application Workloads to Service Instances.

For further information including help text and usage, please run `tanzu service claims list --help`.

Unclaiming Service Instances

Application Developers can unclaim a claimed Service Instance on their target cluster by running:

```
tanzu service claims delete CLAIM-NAME
```

Where `CLAIM-NAME` is the name of the resource claim that currently claims the Service Instance.

For further information including help text and usage, please run `tanzu service claims delete --help`.

Reference

This section provides further references regarding Services Toolkit:

- Resource Requirements
- Known Limitations
- Supported Kubernetes Distributions
- Topology
- User Roles
- Use Cases

Resource Requirements

This page provides information that can be used to help you understand how much resource (such as CPU and RAM) is required to install and use Services Toolkit.

> **Note**
>
> : At present it is not possible to alter default resource configurations for Services Toolkit as part of the installation process. We are planning to add support for this at some point in the future.

Deployments

In order to better understand resource requirements and utilisation, it is important to consider the various Kubernetes `Deployment`s that get created as part of installation, and subsequent usage of, Services Toolkit.

Upon installation of Services Toolkit to a cluster, a single Deployment named `services-toolkit-controller-manager` will be created and it defines a container with the following resource configuration:

```
resources:
  limits:
    cpu: 200m
    memory: 500Mi
  requests:
    cpu: 100m
    memory: 100Mi
```

Note: Please refer to the Kubernetes documentation on Managing Resources for Containers for further information on resource management in Kubernetes.

Then, for each `DownstreamClusterLink` resource created as part of configuring a *Projection Plane* (see Service API Projection and Service Resource Replication), 1 additional `Deployment` will be created on the downstream cluster. This `Deployment` defines a container with the following resource configuration:

```
resources:
  limits:
    cpu: 100m
    memory: 100Mi
  requests:
    cpu: 100m
    memory: 20Mi
```

And finally, there will be one additional `Deployment` for each `ClusterResourceExportMonitor` and `ClusterResourceImportMonitor` resource that gets created upon configuration of *Resource Replication* (see Service API Projection and Service Resource Replication). This `Deployment` defines a container with the following resource configuration:

```
resources:
  limits:
    cpu: 100m
    memory: 100Mi
  requests:
    cpu: 100m
    memory: 20Mi
```

Taking the above into consideration, the minimum set of resources required to support the federation of an API between a Workload Cluster and a Service Cluster can be broken down as follows:

- Workload Cluster
  - 1 x Services Toolkit controller manager deployment
  - requests 100m CPU and 100Mi memory
  - 1 x API proxy deployment
  - requests 100m CPU and 20Mi memory
  - 1 x ClusterResourceImportMonitor deployment
  - requests 100m CPU and 20Mi memory
- Service Cluster
  - 1 x Services Toolkit controller manager deployment
  - requests 100m CPU and 100Mi memory
  - 1 x ClusterResourceExportMonitor deployment
  - requests 100m CPU and 20Mi memory
- Total min resource requirements
  - Workload Cluster = 300m CPU and 140Mi memory

- Service Cluster = 200m CPU and 120Mi

Note: Services Toolkit does not require the use of volumes or any external storage.

Known Limitations

This page lists known limitations and issues with Services Toolkit.

Service Resource Replication Limitations

Limitation 1: Updates to `Secrets` are not automatically replicated

Currently, after a `Secret` has been replicated from a Service Cluster to a Workload Cluster, any further updates to the original `Secret` in the Service Cluster are not propagated to the replica `Secret` in the Workload Cluster. We are aiming to remove this limitation in a future release of Services Toolkit.

Service API Projection Limitations

Limitation 1: CRD and Aggregation layer conflict

We use api-aggregation as the mechanism to project APIs. Once an API is registered via this aggregation layer (the APIService is available), even if you create a CRD pointing to the same path, the requests will still be proxied by the aggregation layer. If you do it the other way around, as in first create the CRD and then “project” the API (or register the APIService), the APIService won’t be available.

Behaviour when local CRD is created before Service Resource API has been projected

For example, let’s say you created `rabbitmqclusters.rabbitmq.com/v1beta1` on your workload cluster by creating a `CustomResourceDefinition` before you project the `rabbitmq.com/v1beta1` API. When you try to project it, the APIService `v1beta1.rabbitmq.com` won’t be ready:

`rabbitmqclusters.rabbitmq.com` CRD status:

```
status:
  acceptedNames:
    categories:
    - all
    kind: RabbitmqCluster
    listKind: RabbitmqClusterList
    plural: rabbitmqclusters
    shortNames:
    - rmq
    singular: rabbitmqcluster
  conditions:
  - lastTransitionTime: "2021-08-18T13:01:31Z"
    message: no conflicts found
    reason: NoConflicts
    status: "True"
    type: NamesAccepted
  - lastTransitionTime: "2021-08-18T13:01:31Z"
    message: the initial names have been accepted
```

```
    reason: InitialNamesAccepted
    status: "True"
    type: Established
  storedVersions:
  - v1beta1
```

`rabbitmq.com-v1beta1-api-group-import` ClusterAPIGroupImport status:

```
status:
  conditions:
  - lastTransitionTime: "2021-08-18T13:01:47Z"
    message: apiservices.apiregistration.k8s.io "v1beta1.rabbitmq.com" already exists
    reason: APIServiceNotReady
    status: "False"
    type: APIServiceReady
  - lastTransitionTime: "2021-08-18T13:01:47Z"
    message: apiservices.apiregistration.k8s.io "v1beta1.rabbitmq.com" already exists
    reason: APIServiceNotReady
    status: "False"
    type: Ready
  observedGeneration: 1
```

The workaround in this case, if you want to use Service API Projection on your cluster (and you don't have any Custom Resources provisioned from this CRD) is to delete the local CRD and delete/recreate the ClusterAPIGroupImport.

Behaviour when local CRD is created after Service Resource API is projected

If you did things in the other order however, the APIService will be available but also the `rabbitmqclusters.rabbitmq.com` CRD won't show any errors on the status, which can be confusing as when you provision/delete a Custom Resource, the requests will be proxied and will run on the linked Service cluster, not on your local cluster.

`rabbitmqclusters.rabbitmq.com` CRD status:

```
status:
  acceptedNames:
    categories:
    - all
    kind: RabbitmqCluster
    listKind: RabbitmqClusterList
    plural: rabbitmqclusters
    shortNames:
    - rmq
    singular: rabbitmqcluster
  conditions:
  - lastTransitionTime: "2021-08-18T09:40:35Z"
    message: no conflicts found
    reason: NoConflicts
    status: "True"
    type: NamesAccepted
  - lastTransitionTime: "2021-08-18T09:40:35Z"
    message: the initial names have been accepted
    reason: InitialNamesAccepted
    status: "True"
    type: Established
```

```
  storedVersions:
  - v1beta1
```

`rabbitmq.com-v1beta1-api-group-import` ClusterAPIGroupImport status:

```
status:
  conditions:
  - lastTransitionTime: "2021-08-18T13:10:48Z"
    status: "True"
    type: APIServiceReady
  - lastTransitionTime: "2021-08-18T13:10:48Z"
    status: "True"
    type: Ready
  observedGeneration: 1
```

Limitation 2: No built-in support for cluster-scoped requests against projected APIs in the Workload Cluster

By default, Services Toolkit does not support projection of cluster-scoped requests in the Workload Cluster. It supports namespace-scoped requests only.

This poses a problem with certain controllers watching these APIs in the Workload Cluster, e.g. Service Binding implementation. They might require cluster-scoped read access verbs on projected APIs in the Workload Cluster.

There is a workaround for these types of scenarios:

We provide a ClusterRole through our prototypical `kubectl-scp` plugin's `federate` command on the Service Cluster. For example:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
 name: "example"
rules:
- apiGroups:
   - rabbitmq.com
 resources:
   - rabbitmqcluster
 verbs: ["get", "list", "watch"]
```

The ClusterRole is then bound to the Proxy Service Account on the Service Cluster.

This workaround has significant implications to be aware of:

- It represents a potential attack vector in which a malicious user operating in Workload Cluster A might obtain the secret access token used by the Proxy and, in turn, use that token to perform read actions (e.g. get/watch/list) on resources in the Service Cluster that are owned by an entirely different Workload Cluster B. In other words, this workaround circumvents proper isolation of projected resources between different Workload Clusters.
- It's confusing to the App Operator who might see resources that belong to non-existing namespaces.
- Projected resources belonging to a Workload Cluster A are potentially being leaked to users in Workload Cluster B. It's similar to the security issue stated earlier in this list, but different in

that the user doesn't even have to have any sort of malicious intent.

Future versions of the Services Toolkit will add first-class support for cluster-scoped requests against projected APIs and, thus, remove the need for the laid out workaround and its problematic characteristics.

Service Resource Claims Limitations

Limitation 1: Can only claim service resources that adhere to the Kubernetes Binding specification

Currently, a `ResourceClaim` will only be successful in claiming a service resource if that service resource adheres to the Provisioned Service duck type or if directly referring to a compatible Secret. Eventually future iterations of the Services Toolkit will loosen this requirement through an extension of the `ResourceClaim` functionality or another API.

Limitation 2: Can only claim service resources once

Currently, only a single `ResourceClaim` can successful claim a service resource. If a second `ResourceClaim` is created for the same service resource it will fail with `ResourceAlreadyClaimed`. Eventually future iterations of the Services Toolkit may allow shared service resources.

Supported Kubernetes Distributions

Kubernetes Distribution	GA Functionality Tested?	Experimental / Beta Functionality Tested?
kind	Yes (used for our local development)	Yes
GKE	Yes (continuously tested in CI)	Yes
AKS	Yes	Not yet
EKS	Yes	Not yet
VMware Tanzu Kubernetes Grid (TKGm) clusters	Yes (TKGm v1.5.0 on vSphere)*	Not yet
Other	Unknown - we haven't tested Services Toolkit on other distributions yet, but it should** work.	Unknown

* TKGm 1.5+ is required.

** Services Toolkit leverages core Kubernetes APIs to provide functionality, as such we would expect it to work on most reasonably up-to-date distributions.

Topology

Topology is a combination of Service and Workload Clusters, their namespaces and the Service Resource Lifecycle APIs that are to be made available from Service Clusters to one or more Workload Clusters.

Note: The following two assumptions that must hold true for topologies currently supported by the Services Toolkit.

1. The presence of a “flat” network is assumed, which is to say that workloads running in one cluster are able to establish network connections (resolution and routing) to the Kubernetes API Server endpoints of all other clusters without any additional setup
2. Application workloads can establish network connections to the endpoints of Service Instances without any additional setup

We are considering ideas that will allow us to relax these assumptions in the future but do not yet have a firm date in mind for when such functionality may be released.

Supported Topologies

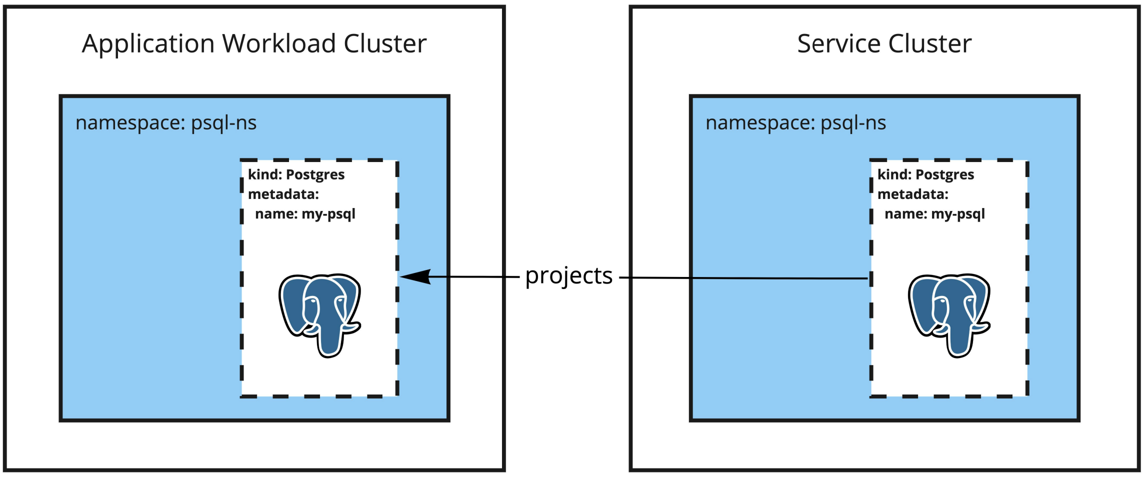
Topologies that are currently supported by the Services Toolkit are documented below. Please also note the following rules that apply.

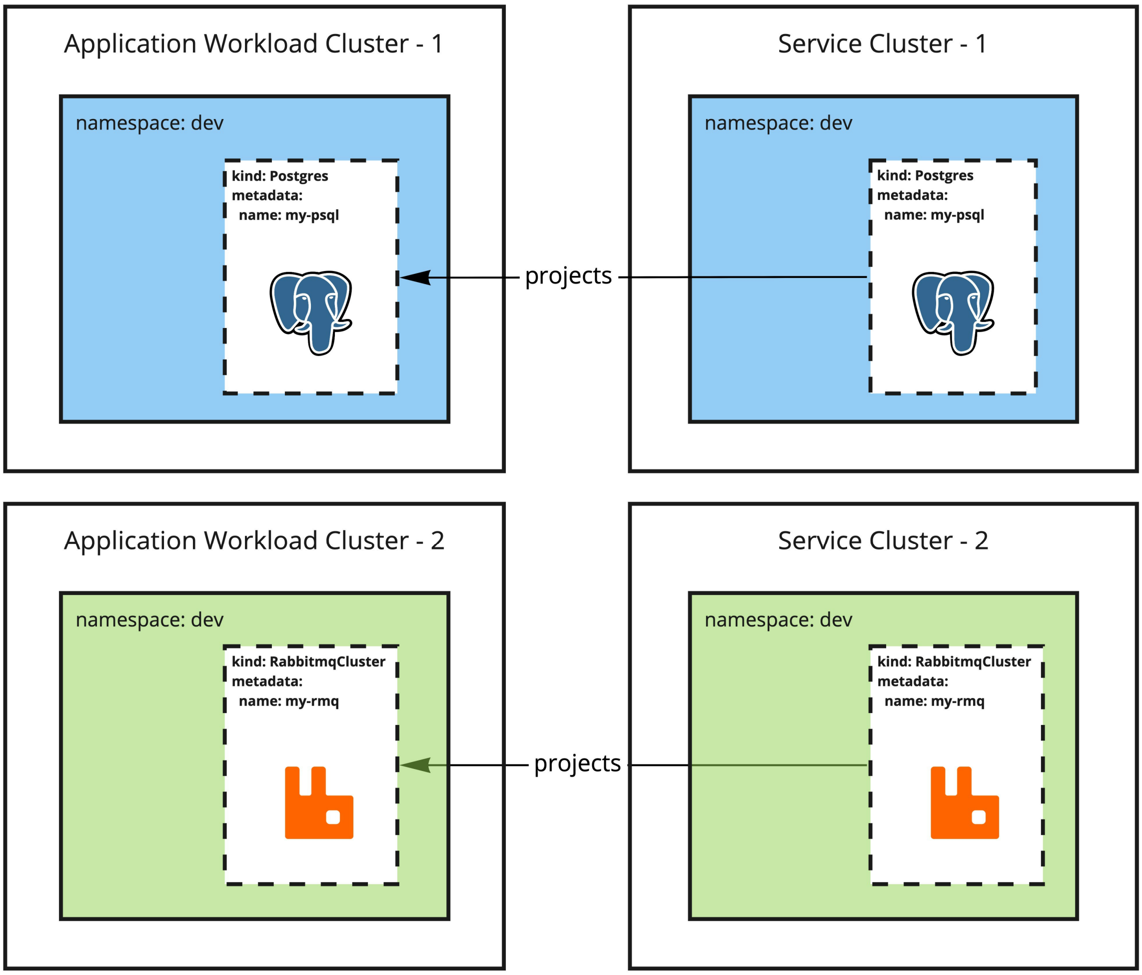
- API Projection does NOT work within a single cluster but only across a set of distinct service and workload clusters.
  - We have no plans on changing this with subsequent releases.
- An API group can be either projected into a given cluster or installed/reconciled within that cluster, not both.
  - For example, you cannot install the RabbitmqCluster Operator and project RabbitmqClusters from a Service cluster in the same Workload cluster.
  - Right now, we have no plans on changing this with subsequent releases.
  - Refer to Limitations for further details.
- Resources of a projected API group must exist in identically named namespaces in the workload and service clusters.
  - For a given workload cluster, there can only be a single service cluster for a given API group projection.
  - For example, a workload cluster cannot receive projections of a RabbitmqCluster API from service cluster 1 as well as from service cluster 2.
  - We think this is a legitimate use case, so we may change this in the future.

Provide a Service Resource Lifecycle API

From one Service cluster to one Workload cluster

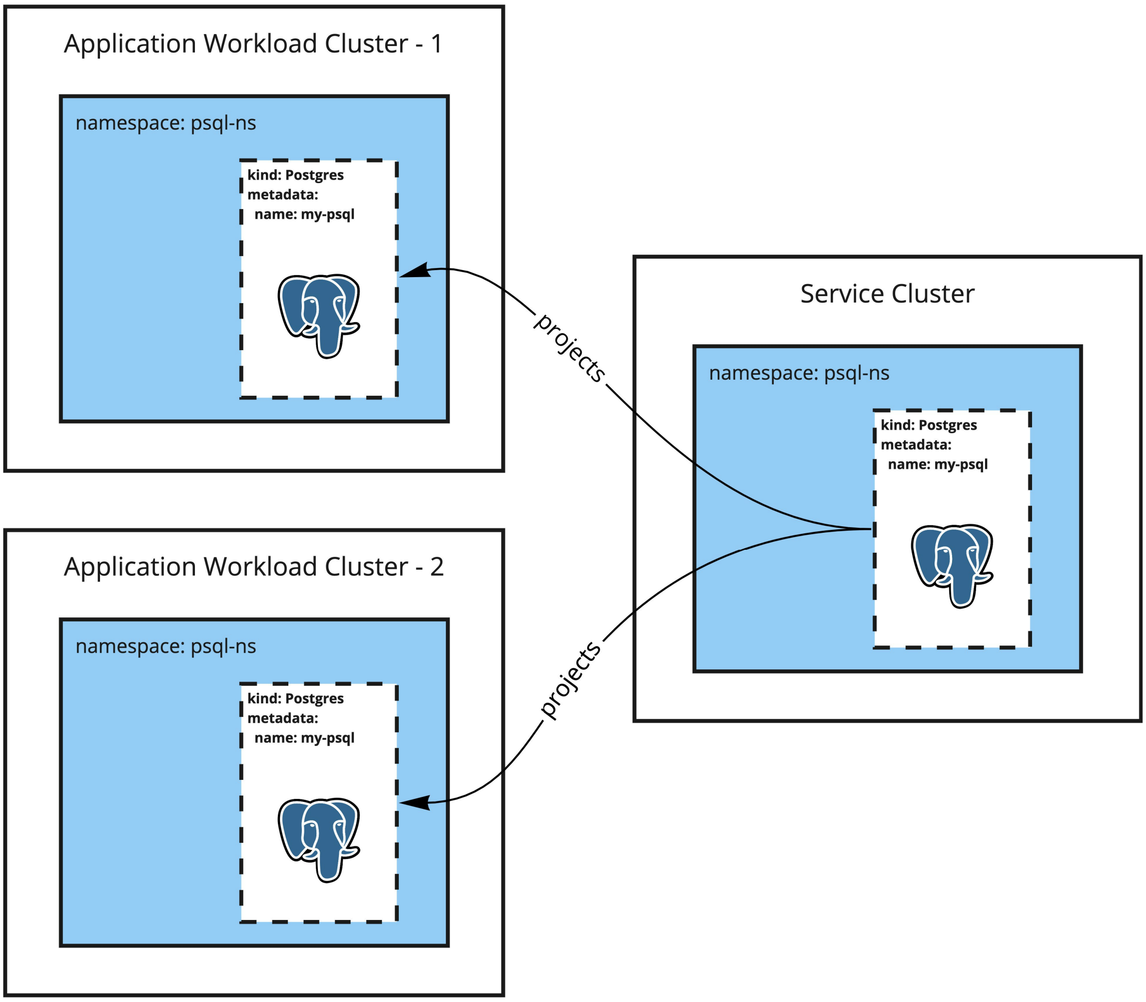
As a Service Operator I want to provide a Service Resource Lifecycle API from one Service cluster to one Workload cluster in the same named namespace.





From a Service cluster to multiple Workload clusters

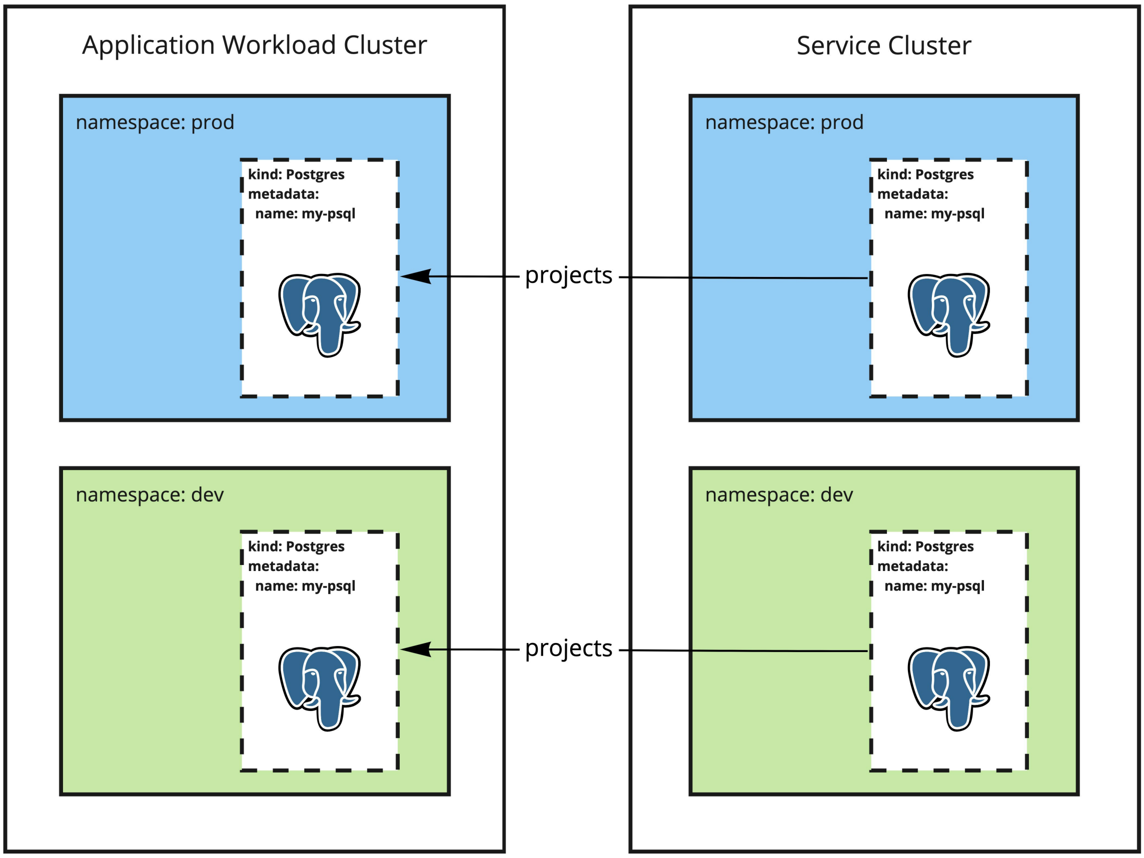
As Service Operator I want to provide a Service Resource Lifecycle API from a Service cluster to multiple Workload clusters with the same named namespace.



Provide different Service Resource Lifecycle APIs

From a Service cluster to a Workload cluster

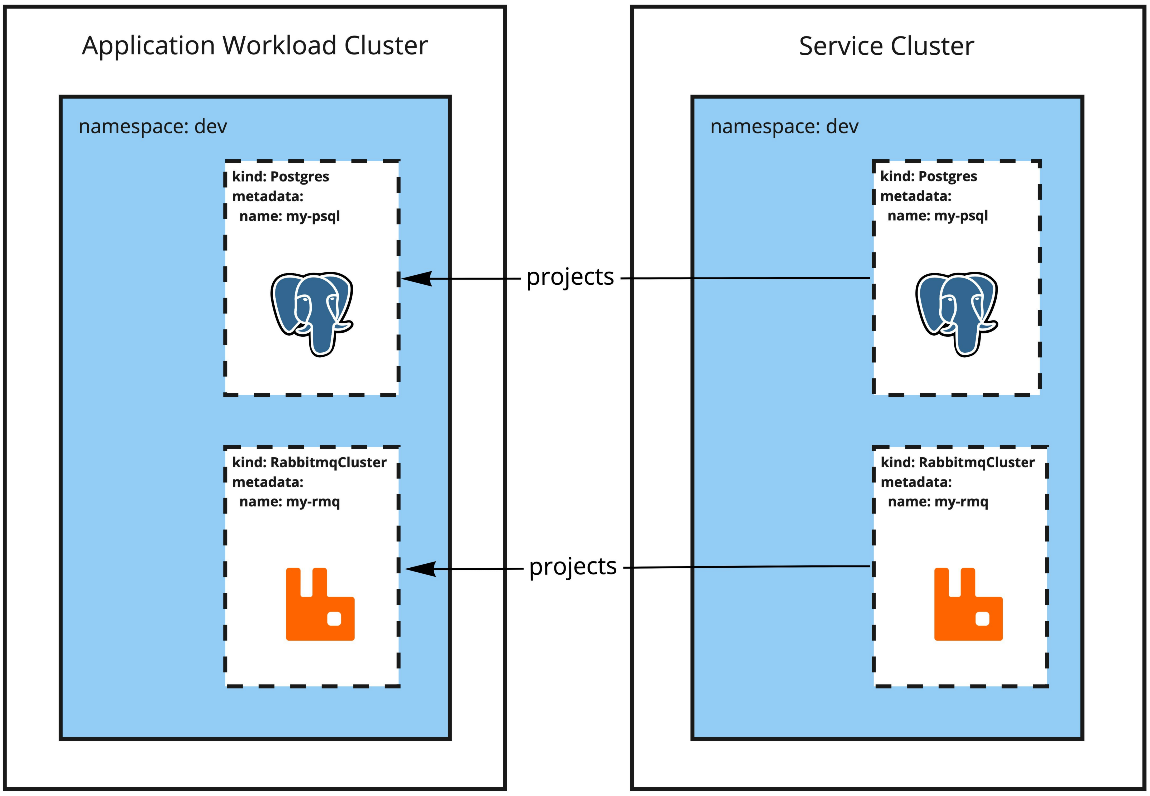
As a Service Operator I want to provide different Service Resource Lifecycle APIs from one Service cluster and distinct namespaces to one Workload cluster in matching named namespaces.



Provide multiple Service Resource Lifecycle APIs

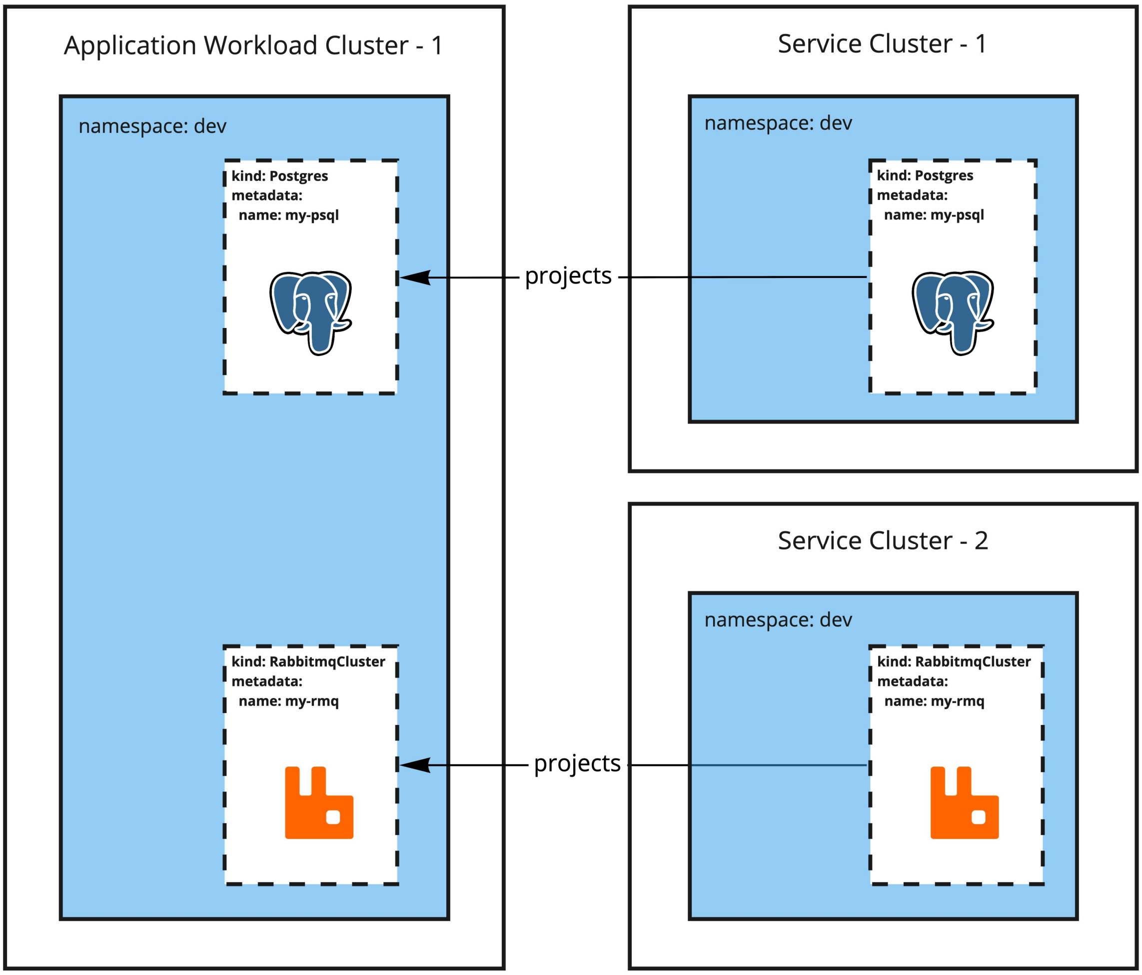
From a Service Cluster to a Workload cluster

As Service Operator I want to provide multiple Service Resource Lifecycle APIs from one Service Cluster and one namespace to one Workload cluster with the same named namespace.



From multiple Service Clusters to one Workload cluster

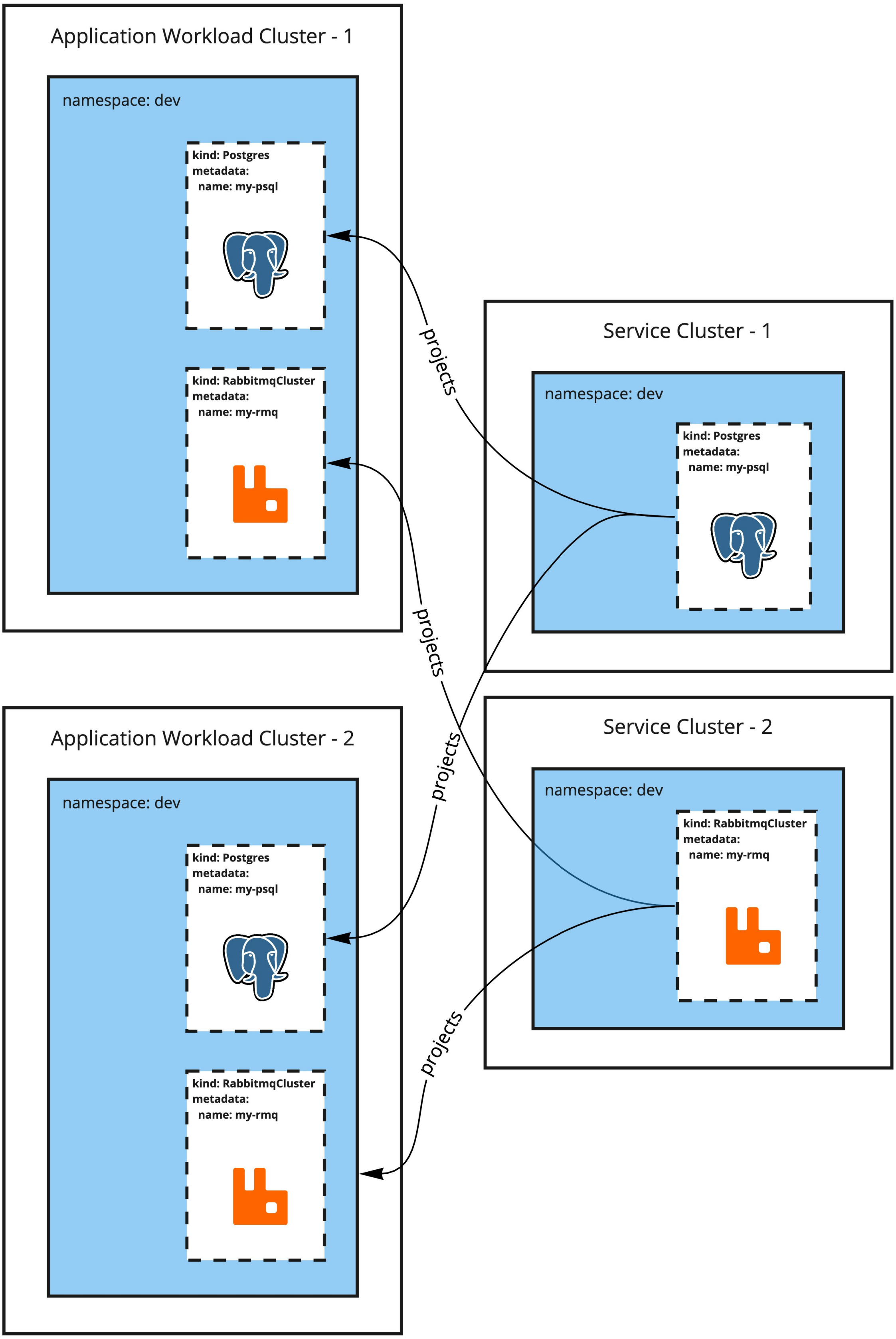
As Service Operator I want to provide multiple Service Resource Lifecycle APIs from multiple Service Clusters with the same namespace to one Workload cluster with the same named namespace.



Warning: In this particular scenario, you might encounter name collisions in the Application Workload Clusters for the core resources like secrets. For example, if API-1 creates a secret called binding-secret and API-2 also creates a secret called binding-secret, Resource Replication component will copy both of these secrets in the Application Workload Cluster but one will be overridden by the other depending on which one is replicated second.

From multiple Service Clusters to multiple Workload clusters

As Service Operator I want to provide multiple Service Resource from multiple distinct Service Clusters with the same namespace name to multiple Workload clusters with matching named namespace.


Application Workload Cluster - 1
namespace: dev
kind: Postgres
metadata:
name: my-psql
kind: RabbitmqCluster
metadata:
name: my-rmq
Service Cluster - 1
namespace: dev
kind: Postgres
metadata:
name: my-psql
projects
projects
Application Workload Cluster - 2
namespace: dev
kind: Postgres
metadata:
name: my-psql
kind: RabbitmqCluster
metadata:
name: my-rmq
Service Cluster - 2
namespace: dev
kind: RabbitmqCluster
metadata:
name: my-rmq
projects
projects

User Roles

Services Toolkit caters to the following user roles, which can be considered groupings of jobs to be done. Each role can be carried out by the same or alternatively by different people, and each individual person can play more than one role.

Service Author (SA)

- Responsible for the development and release of Kubernetes Operators and their Service Resource Lifecycle APIs.
- May optionally provide recommendations regarding configuration of Service Resources (e.g. production-ready configuration provided by the RabbitMQ Cluster Operator SAs here).

Service Operator (SO)

- Responsible for the installation, operation and ongoing maintenance of one or more Kubernetes Operators providing Service Resource Lifecycle APIs.
- Responsible for offering out Service Resource Lifecycle APIs and making them available to Application Operators and Developers.
- Lifecycle management (Create, Read, Update, Delete) of Service Instances
- Lifecycle management (Create, Read, Update, Delete) of Resource Claim Policies

Application Operator (AO)

- Discover Service Resource Lifecycle APIs and assesses their capabilities through provided metadata.
- Make decisions about which APIs to consume, taking into consideration the needs of the Application (e.g. QoS, persistence, HA, etc.).
- Lifecycle management (Create, Read, Update, Delete) of Resource Claims

Application Developer (AD)

- Lifecycle management (Create, Read, Update, Delete) of Application Workloads
- Binding Application Workloads to Service Instances

Advanced Use Cases

This page contains a number of use cases for Tanzu Application Platform powered by the Services Toolkit. It is highly recommended to have first completed the Getting Started Walkthrough in the Tanzu Application Platform Getting Started Guide as this covers the most common day-to-day use cases.

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.

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

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"
```

   Note: Kubernetes Secret resources must abide by the Well-known Secret Entries specifications. **Note:** 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.

2. Apply the YAML file by running:

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

3. 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
```

4. 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
```

5. Create an Application Workload by running:

   Example:

```
tanzu apps workload create <WORKLOAD-NAME> \
  --git-repo https://github.com/sample-accelerators/spring-petclinic \
```

```
  --git-branch main \
  --git-tag tap-1.0 \
  --type web \
  --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)

Note: This Use Case make use of experimental APIs and is not recommended for use in production environments.

This use case make use of the experimental API Projection and Resource Replication APIs in order to separate Application Workloads and Service Instances onto separate Kubernetes clusters. There are several reasons as to why you may want to do this.

- **Dedicated cluster requirements for Workload or Service clusters:** service clusters, for instance, 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.

Pre-Requisites

Please note the following assumptions / pre-requisites for completing this use case walkthrough:

1. You have access to a cluster with Tanzu Application Platform installed (henceforth referred to as the “Application Workload Cluster”)
2. You have access to a second, separate cluster with the Services Toolkit package installed (henceforth referred to as the “Service Cluster”)
3. You have downloaded and installed the `tanzu` CLI along with the corresponding plug-ins

4. You have downloaded and installed the experimental `kubectl-scp` plug-in (see Install the kubectl-scp plug-in)

5. You have setup the `default` namespace on the Application Workload Cluster to use installed packages (see Set up developer namespaces to use installed packages) and will use it as your “developer namespace”

6. The Application Workload Cluster is able to pull source code from GitHub

7. The Service Cluster is able to pull the images required by the RabbitMQ Cluster Kubernetes Operator

8. The Service Cluster is able to create LoadBalancer services

Important: If you have previously installed the RabbitMQ Cluster Operator to the Application Workload Cluster (i.e. as part of running through the Getting Started Walkthrough), you must first uninstall it from that cluster. This is due to a known limitation with the experimental API Projection APIs. Further information regarding this limitation can be found in Limitations.

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

Walkthrough

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

Important: Some of the commands listed in the following steps have placeholder values `WORKLOAD-CONTEXT` and `SERVICE-CONTEXT`. Change these values before running the commands.

1. As the Service Operator, run the following command to link the Workload Cluster and Service Cluster together by using the `kubectl scp` plug-in:

   ```
   kubectl scp link --workload-kubeconfig-context=<WORKLOAD-CONTEXT> --service-kub
   econfig-context=<SERVICE-CONTEXT>
   ```

2. Install the RabbitMQ Kubernetes Operator in the Services Cluster using kapp.

   Note: This Operator is installed in the Service Cluster, but `RabbitmqCluster` service instances can still have their lifecycles managed (CRUD) from the Workload Cluster.

   Note: Use the exact `deploy.yml` specified in the command as this RabbitMQ Operator deployment includes specific changes to enable cross-cluster service binding.

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

3. Verify that the Operator is installed by running:

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

   The following steps federate the `rabbitmq.com/v1beta1` API group, which is available in the Service Cluster, into 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 service-instance namespace in both clusters.

   API Projection ocurrs between clusters 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
```

5. Federate using the `kubectl-scp` plug-in 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
```

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

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

7. Discover the new service and provision an instance from the Workload cluster by running:

   Note: Ensure the `tanzu` CLI is configured to target the Workload cluster.

```
tanzu service types list
```

   The following output appears:

```
Warning: This is an ALPHA command and may change without notice.

NAME      DESCRIPTION                APIVERSION             KIND
rabbitmq  It's a RabbitMQ cluster!  rabbitmq.com/v1beta1   RabbitmqCluster
```

8. 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
```

9. Apply the YAML file by running:

```
kubectl --context <WORKLOAD-CONTEXT> -n service-instances apply -f rabbitmq-clu
ster.yaml
```

10. Confirm that the RabbitmqCluster resource reconciles successfully from the Workload Cluster by running:

```
kubectl --context <WORKLOAD-CONTEXT> -n service-instances get -f rabbitmq-clust
er.yaml
```

11. Confirm 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
```

12. 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
```

13. Create the application workload by running:

```
tanzu apps workload create multi-cluster-binding-sample \
  --git-repo https://github.com/sample-accelerators/rabbitmq-sample \
  --git-branch main \
  --git-tag tap-1.0 \
  --type web \
  --service-ref "rmq=services.apps.tanzu.vmware.com/v1alpha1:ResourceClaim:proj
ected-rmq-claim"
```

14. Get the web-app URL by running:

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

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