GETTING STARTED WITH CROSSPLANE COMPOSITIONS

VIKTOR FARCIC

Compositions

This paper is the second chapter in the larger book, Crossplane: the Cloud Native Control Plane[1]. It is part of a series of papers that break down the book.

If you're curious to see some of the things Crossplane can do, check out this blog[2] or the intro of the paper. This is the second of the series, of which we covered Providers and Managed Resources in the first[3]. I recommend seeing all resources beforehand to give context to what we will cover in this paper.

Let's explore Crossplane compositions. We won't be talking about theory without touching the keyboard, so I'll keep this introduction short and jump straight into Crossplane compositions... right after we set up the environment we'll use in this paper. If you prefer a video version of this paper, view my YouTube tutorial here[4].

[1] https://www.upbound.io/resources/lp/book/crossplane-cloud-native-control-plane
[2] https://blog.upbound.io/why-choose-crossplane
[3] https://www.upbound.io/resources/lp/whitepaper-b/crossplane-providers-and-managed-resources-getting-started
[4] https://youtu.be/X7E6YfXWgvE?si=2lfYnPp3b9WAHEK2

We saw how we can manage individual resources with **Crossplane**. If, for example, we want a VM in **AWS**, **Azure**, or **Google Cloud**, all we have to do is define a **Kubernetes** resource that represents it, apply it to the control plane cluster running Crossplane, and… that's it. **Crossplane takes care of everything** else.

The problem with that approach is that it is often very **low-level**. For example, creating and managing a production-ready database can consist of quite a few low-level resources that require a certain level of expertise that not everyone in an organization might have.

Most of the resources we are managing today are low-level. There is no such thing as a database in AWS. Instead, we need to combine **RDS**, with a **VPC**, with **subnets**, with a **gateway**, and so on and so forth. Even when we do that, we still need at least one database inside the database server, a user, and a schema. Similarly, there is no such thing as an application in Kubernetes. We need to combine a **Deployment** with an **Ingress**, with a **Service**, with **Secrets**, and quite a few other resources. The same can be said for almost anything else. Software, services, and infrastructure are complex and providers we are using are intentionally focusing on low-level services so that they can cater wide audience. Resources are more like **building blocks** than final solutions.

On the other hand, we are all trying to **shift left**. We are trying to enable our colleagues to be autonomous instead of waiting for someone else to assemble those building blocks for them.

As a result, we need to use experts in certain fields to create services that can be consumed by others. A database expert can create services that will enable others to manage databases in a way appropriate for production. A Kubernetes expert can create abstractions that define what an application is. A security expert can bake security and policies into those services. There are many other examples and it all boils down to experts in certain fields using their experience to create and manage services that can be consumed by others.

The end result is an **Internal Developer Platform** that exposes services that simplify workflow for developers and other software engineers.

That's where **Crossplane Compositions** come in. They enable us to define what something is. They enable us to codify our expertise and expose services as higher-level abstractions.

In this chapter, we'll explore how to leverage Crossplane's ability to create Custom Resource Definitions and Controllers that will act as such services.

Chapter Setup

The setup in this chapter continues using the pattern from the previous one.

All the commands user in this chapter are in the Gist[5].

We'll enter into the directory of the forked repository...

```
cd crossplane-tutorial
```

...and start Nix shell that brings all the tools we'll need.

```
nix-shell --run $SHELL
```

Next, we'll make the setup script executable,...

```
chmod +x setup/02-compositions.sh
```

...and execute it.

```
./setup/02-compositions.sh
```

The only thing left is to source the environment variables in case we need them later.

```
source .env
```

Now we can explore **Composite Resource Definitions**.

Composite Resource Definitions

Crossplane Compositions consist of a few components. There are Composite Resource Definitions, Compositions, and Composite Resources.

Right now, we'll focus on the first of those. We'll create a **Composite Resource Definition**.

For now, the only important thing to know about Composite Resource Definitions is that they **extend Kubernetes API** by creating **Custom Resource Definitions**. They enable us to define what something is.

I'll explain everything else you need to know about them in a moment. For now, let's take a look at a simple example, which we'll improve as we're progressing through this chapter.

```
cat compositions/sql-v1/definition.yaml
```

The output is as follows.

[5] https://gist.github.com/vfarcic/08162d1f3f4954c1f420fae59704b629

```
apiVersion: apiextensions.crossplane.io/v1
kind: CompositeResourceDefinition
metadata:
  name: sqls.devopstoolkitseries.com
spec:
  group: devopstoolkitseries.com
  names:
    kind: SQL
    plural: sqls
  claimNames:
    kind: SQLClaim
    plural: sqlclaims
  versions:
  - name: v1alpha1
    served: true
    referenceable: true
    schema:
      openAPIV3Schema: {}
```

Everything we do with Crossplane is defined as Kubernetes resources, and Composite Resource Definitions are no exception.

The `apiVersion`, `kind`, and `metadata` should be self-explanatory if you have at least a basic understanding of **Kubernetes**. If you don't, I'm surprised you got this far without giving up.

The "magic" is in the `spec`.

The first in line is the `spec.group` field that defines the API group that will be created in the Kubernetes API. That follows the rules that any Kubernetes resource definition must follow. `Deployment`, for example, is in the group `apps/v1`. By defining groups, we are making sure that resources are uniquely identified even if some of them have the same `kind`.

Further on, there is `spec.names` that defines the names of that resource definition, both in singular and plural.

You can probably guess what the goal of that definition is by looking at the `spec.names.kind` value. We'll use it to define **SQL (database) servers** and everything they need.

I'll leave the `spec.claimNames` fields a mystery for now. We'll explore them later.

Finally, there is `spec.versions` that, as the name suggests, defines the versions of that resource. It can be anything, but my recommendation is to follow Kubernetes versioning guidelines with `v1alpha1`, `v1beta2`, `v1`, and so on.

Further on, there is `spec.served` which, essentially, tells Crossplane whether that specific version is served to users (enabled).

Then there is `spec.referenceable` which indicates which specific version is currently active or, to use Crossplane terminology, which one can be referenced. A Composite Resource Definition can have any number of versions, but only one can be referenceable.

Finally, there is, arguably, the most important field `spec.schema` which is based on Open API. I intentionally left it blank in an attempt to create the simplest possible definition which we'll expand later.

Before we proceed, please bookmark the Crossplane API[6] page. You'll find the complete schema for `CompositeResourceDefinition` or any other Crossplane API over there. By doing that, by redirecting you to the docs, I can avoid going through every single API in detail and risk repeating details written over there.

Let's apply the definition,...

```
kubectl apply --filename compositions/sql-v1/definition.yaml
```

...and retrieve it from the control plane cluster.

```
kubectl get compositeresourcedefinitions
```

Over time, you will probably get tired from typing long resource kinds like `compositeresourcedefinition` so you can also use short names like, in this case, `xrd` to get the same outcome.

```
kubectl get xrds
```

I will continue using long names most of the time because I believe they are easier to understand, even though they are harder to type.

For now, the main takeaway you should get from Composite Resource Definitions is that they create and manage Kubernetes Custom Resource Definitions which is a way to extend Kubernetes API.

As proof that's what they do, we can list all `crds`.

```
kubectl get crds | grep sql
```

The output is as follows.

```
sqlclaims.devopstoolkitseries.com 2024-01-03T16:04:16Z
sqls.devopstoolkitseries.com      2024-01-03T16:04:16Z
```

We can see that a specific Composite Resource Definition created two CRDs. We'll ignore the first one (`sqlclaims`) for now, and focus on the second (`sqls`).

One nice thing about Kubernetes CRDs and, through them, about Crossplane **Composite Resource Definitions**, is that they are discoverable. We can, for example, ask Kubernetes to `explain sqls`.

[6] https://docs.crossplane.io/latest/api

```
kubectl explain sqls.devopstoolkitseries.com --recursive
```

The output is as follows (truncated for brevity).

```
GROUP:      devopstoolkitseries.com
KIND:       SQL
VERSION:    v1alpha1

DESCRIPTION:
    <empty>
FIELDS:
  apiVersion    <string>
  kind  <string>
  metadata  <ObjectMeta>
    ...
  spec  <Object> -required-
    claimRef    <Object>
      apiVersion    <string> -required-
      kind  <string> -required-
      name  <string> -required-
      namespace <string> -required-
    compositionRef  <Object>
      name  <string> -required-
    compositionRevisionRef  <Object>
      name  <string> -required-
    ...
  status    <Object>
    ...
```

The ability to retrieve a schema or, as in this case, to explain it, might not sound very exciting but, if that's what you think, I will have to strongly disagree. Kubernetes' API allows us to discover the type of resources we can use and their schemas and that means that attempts to visualize them through UIs, CLIs, or any other means is trivial. It would be trivial to build a "dumb" front-end that would be able to discover what is what, to help us define resources, and to operate those resources. With a standard, but extensible API like the one Kubernetes offers, we can easily build the tools we need for an **Internal Developer Platform**. We'll explore that in more detail in one of the next chapters. For now, the key takeaway is that everything in Kubernetes is discoverable and since Crossplane is Kubernetes-native, everything we do with Crossplane is discoverable as well.

From now on, we can create any number of resources based on that definition. We are yet to discover whether that also means that we can create any number of database servers (SQLs).

Here's one example.

```
cat examples/sql-v1.yaml
```

The output is as follows.

```
apiVersion: devopstoolkitseries.com/v1alpha1
kind: SQL
metadata:
  name: my-db
spec: {}
```

That manifest is a resource based on that definition. Values of the `apiVersion` and `kind` fields match those of the definition. `metadata` contains an arbitrary `name` that can be anything, as long as it is unique.

Finally, we did not set any `spec` fields in the definition, so that one is empty (for now).

All that's left is to apply that `SQL`,...

```
kubectl apply --filename examples/sql-v1.yaml
```

...and celebrate.

We got our first SQL server! Our first database was born!

Let's take a look at it by listing all `sqls`.

```
kubectl get sqls
```

The output is as follows.

```
NAME  SYNCED READY COMPOSITION AGE
my-db False                   2m41s
```

That does not look right. `my-db` is not synced. Crossplane could not even start working on it.

We can confirm that nothing really happened by outputting managed resources.

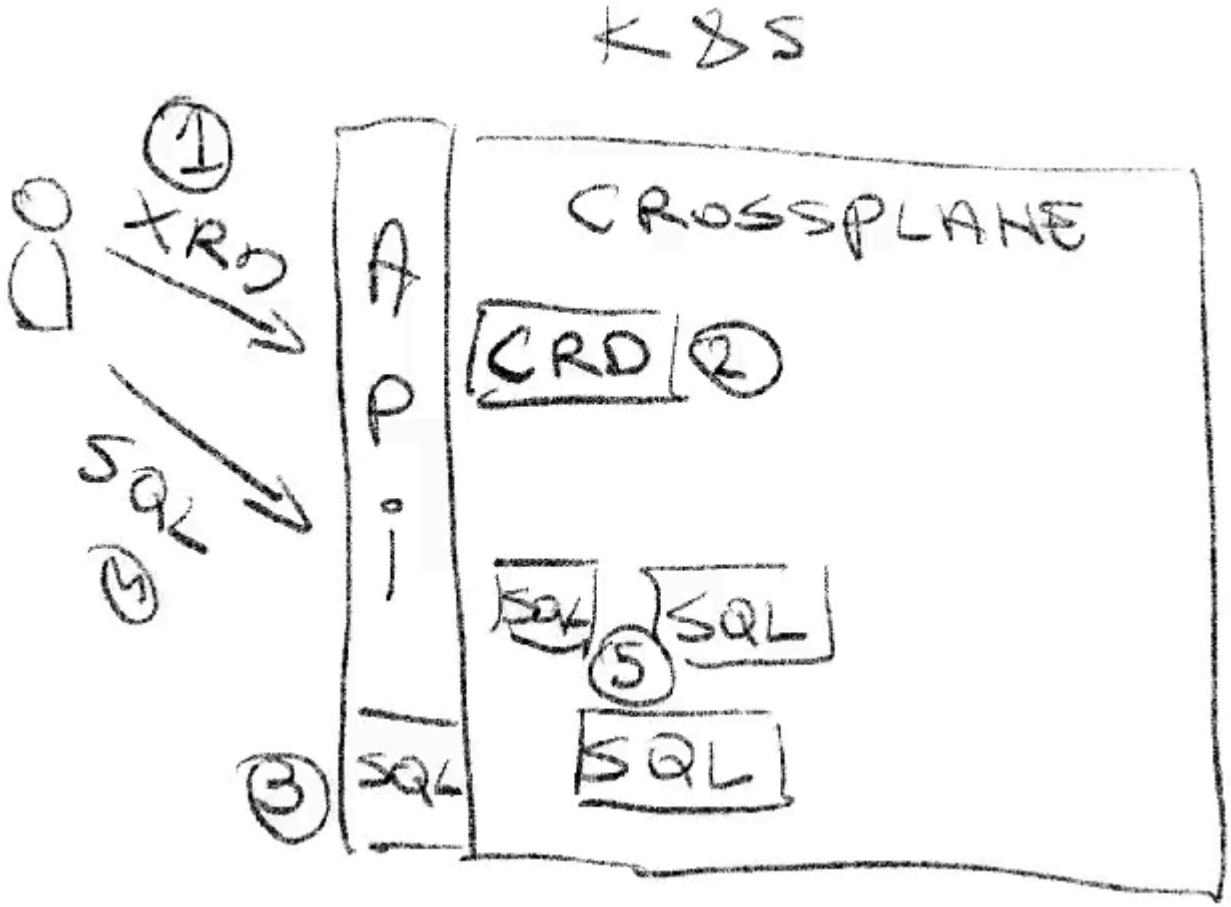
```
kubectl get managed
```

The output is as follows.

```
error: the server doesn't have a resource type "managed"
```

There are no managed resources. Crossplane did nothing, and that was to be expected.

All we did, so far, was to define a **Composite Resource Definition** or XRD (1) which, essentially, created a **Kubernetes Custom Resource Definition** or CRD which extended Kubernetes API with a new resource type called `SQL` (2). There is no controller that would detect **Composite Resources** or, in Kubernetes terminology, Custom Resources. Simply put, we extended Kubernetes API (3) but we did not tell it what to do when resources based on that API are created.



We are missing **Crossplane Compositions**. Right now, we have none, and we can confirm that by retrieving all `compositions`.

```
kubectl get compositions
```

So, for now, we can create as many `SQL` resources (4, 5) as we want, but there are no controllers so there is no process that will do anything with those resources. They are just entries in etcd.

The output shows `No resources found`. Let's fix that. Let's tell Crossplane what to do when a resource based on our definition is created.

Defining Compositions

Before we proceed, let me state that I chose to use Google Cloud in this chapter. The first chapter used **AWS**, and the second used **Azure**, so now it is time for **Google Cloud**.

That being said, just as in previous chapters, you can use any of the "big three" hyperscalers. Explanations in this chapter will be based on examples for **Google Cloud**, but the logic is the same no matter which one you chose so you should not have any trouble following along even if what you see in your terminal is different from what you see here.

With that out of the way, let's take a look at a Composition I prepared in advance.

```
cat compositions/sql-v1/$HYPERSCALER.yaml
```

The output is as follows.

```
---
apiVersion: apiextensions.crossplane.io/v1
kind: Composition
metadata:
  name: google-postgresql
  labels:
    provider: google
    db: postgresql
spec:
  compositeTypeRef:
    apiVersion: devopstoolkitseries.com/v1alpha1
    kind: SQL
  resources:
  - name: sql
    base:
      apiVersion: sql.gcp.upbound.io/v1beta1
      kind: DatabaseInstance
      spec:
        forProvider:
          region: us-east1
          rootPasswordSecretRef:
            namespace: crossplane-system
            key: password
            name: my-db-password
          databaseVersion: "POSTGRES_13"
          settings:
          - availabilityType: REGIONAL
```

```
            tier: db-custom-1-3840
            backupConfiguration:
            - enabled: true
              binaryLogEnabled: false
            ipConfiguration:
            - ipv4Enabled: true
              authorizedNetworks:
              - name: all
                value: 0.0.0.0/0
          deletionProtection: false
  - name: user
    base:
      apiVersion: sql.gcp.upbound.io/v1beta1
      kind: User
      spec:
        forProvider:
          passwordSecretRef:
            key: password
            name: my-db-password
            namespace: crossplane-system
          instanceSelector:
            matchLabels:
              crossplane.io/composite: my-db
```

That is a definition of a `Composition`. Think of it as one of the implementations of the definition we applied earlier.

We have `metadata` with a `name` and `labels`. Those are important since, as we'll see later, they will allow us to choose which implementation we want to use when we declare a **Composite Resource**. You'll see those in action later. For now, remember that this Composition can be identified through a `name` or `labels` which, in my case, are set to `provider: google` and `db: postgresql`.

Next, there is `spec.compositeTypeRef` which tells Crossplane what the associated Composite Resource Definition this Composition is associated with. In other words, this Composition (this implementation) will be used whenever someone defines an `SQL` that has matching `name` or `labels`. We'll see how that works soon. For now, let's take a look at the second component of the `spec`.

`spec.resources` array contains the list of resources that should be managed whenever someone defines the `SQL` resource. In the case of Google Cloud, there are only two resources. If you're using AWS, you'll notice that there are many more since AWS often forces us to combine more resources to get something meaningful.

Each `resource` has a `name` and a `base`. The `name` is a unique identifier within a Composition, while the `base` defines all the details of a resource that should be managed by that Composition. In the case of Google Cloud, we are defining a `DatabaseInstance` and a `User`. Those two are, essentially,

Managed Resources just like those we used in the previous chapter. The major difference is that we are not defining those resources every single time but, instead, grouping them all together and, by doing that, creating a new service and exposing it to others.

You'll notice selectors like, in the case of Google Cloud, `instanceSelector`. Selectors deserve special attention so we'll go through them separately. For now, think of them as Crossplane's way of saying: "Let this resource get some information from this other resource or a group of resources".

The important part is that, among other things, we are defining `spec.resources[0].base.spec.forProvider.rootPasswordSecretRef` that references a Kubernetes secret that will contain the initial password for the database. In the case of AWS, that would be `spec.resources[14].base.spec.forProvider.passwordSecretRef` and, in the case of Azure it's `spec.resources[1].base.spec.forProvider.administratorLoginPasswordSecretRef`. For now, remember that a secret with the password is required. We'll need that knowledge later.

The rest is following the logic we explored in the previous chapter. Each resource defines specific parameters like the `region`, `databaseVersion`, and so on and so forth.

As a result, we should have a database server with everything it needs.

Actually, that's wrong. That database server will not have everything we need but, rather, it is a start that leads us towards the path that ends with everything we might need.

Even though my examples are based on Google Cloud, that is only one of the three Compositions we are defining and relating to the Composite Resource Definition. We can see that all three of them are in the `compositions` directory.

```
ls -1 compositions/sql-v1
```

The output is as follows.

```
aws.yaml
azure.yaml
definition.yaml
google.yaml
```

We are about to apply all the manifests in that directory and, as a result, we'll get the definition that we already applied, and three Compositions, one for each of the major hyperscalers.

```
kubectl apply --filename compositions/sql-v1
```

The output is as follows (truncated for brevity).

```
Warning: ... "VPC.ec2.aws.upbound.io" not found
Warning: ... "Subnet.ec2.aws.upbound.io" not found
Warning: ... "SubnetGroup.rds.aws.upbound.io" not found
Warning: ... "InternetGateway.ec2.aws.upbound.io" not found
Warning: ... "RouteTable.ec2.aws.upbound.io" not found
Warning: ... "Route.ec2.aws.upbound.io" not found
Warning: ... "MainRouteTableAssociation.ec2.aws.upbound.io" not found
Warning: ... "RouteTableAssociation.ec2.aws.upbound.io" not found
Warning: ... "SecurityGroup.ec2.aws.upbound.io" not found
Warning: ... "SecurityGroupRule.ec2.aws.upbound.io" not found
Warning: ... "Instance.rds.aws.upbound.io" not found
composition.apiextensions.../aws-postgresql created
Warning: ... "ResourceGroup.azure.upbound.io" not found
Warning: ... "Server.dbforpostgresql.azure.upbound.io" not found
Warning: ... "FirewallRule.dbforpostgresql.azure.upbound.io" not found
composition.apiextensions.../azure-postgresql created
compositeresourcedefinition.apiextensions.../sqls.devopstoolkitseries.com unchanged
Warning: ... "DatabaseInstance.sql.gcp.upbound.io" not found
Warning: ... "User.sql.gcp.upbound.io" not found
composition.apiextensions.../google-postgresql created
```

We can see that three Compositions (`aws-postgresql`, `azure-postgresql`, and `google-postgresql`) were created. As a result, Crossplane spun up a controller that will manage resources based on that definition. We'll see the controller in action in a moment.

We can also notice from that output that that we got quite a few warnings.

The definitions of individual resources that constitute Compositions are not available. Just as we did in the previous chapter, we need to apply Providers that contain definitions behind those resources.

Later on, we'll see how we can package Compositions into Configurations that will auto-install the required providers. But that's the story for later so, for now, we'll apply the providers manually. They are defined in `providers/sql-v1.yaml`, so let's take a look at it.

```
cat providers/sql-v1.yaml
```

The output is as follows.

```
---
apiVersion: pkg.crossplane.io/v1
kind: Provider
metadata:
  name: provider-aws-ec2
spec:
  package: xpkg.upbound.io/upbound/provider-aws-ec2:v0.47.1
---
apiVersion: pkg.crossplane.io/v1
kind: Provider
metadata:
  name: provider-aws-rds
spec:
  package: xpkg.upbound.io/upbound/provider-aws-rds:v0.47.1
---
apiVersion: pkg.crossplane.io/v1
kind: Provider
metadata:
  name: provider-gcp-sql
spec:
  package: xpkg.upbound.io/upbound/provider-gcp-sql:v0.41.0
---
apiVersion: pkg.crossplane.io/v1
kind: Provider
metadata:
  name: provider-azure-dbforpostgresql
spec:
  package: xpkg.upbound.io/upbound/provider-azure-dbforpostgresql:v0.40.0
```

Over there, just as we did in the previous chapter, we have a few providers. There are `ec2` and `rds` providers from the AWS family. We need both since RDS (SQL) in AWS requires some EC2 resources as well. Then there is `sql` provider from GCP (Google Cloud Platform), and `dbforpostgresql` from Azure.

Let's apply those,...

```
kubectl apply --filename providers/sql-v1.yaml
```

...and take a look at package revisions.

```
kubectl get pkgrev
```

The output is as follows (truncated for brevity).

```
NAME                                    HEALTHY REVISION IMAGE       ...
.../provider-aws-rds-...                        1        .../provider...
.../provider-azure-dbforpostgresql-... False    1        .../provider...
.../provider-gcp-sql-...                Unknown 1        .../provider...
```

As we saw earlier, each of the providers can have any number of resource definitions so we might end up with hundreds of CRDs. As a result, it might take a while until they are all loaded and ready to go. So, we might need to wait for a bit until we see the status of all of the Providers as `HEALTHY`.

After a while, once all the Providers are healthy, we can re-run `kubectl get pkgrev`.

```
kubectl get pkgrev
```

The output is as follows (truncated for brevity).

```
NAME                                             HEALTHY REVISION IMAGE       ...
.../provider-aws-ec2-bc4e31f08ec6                True    1        .../provider...
.../provider-aws-rds-410139ed4243                True    1        .../provider...
.../provider-azure-dbforpostgresql-7905967328cb True    1        .../provider...
.../provider-gcp-sql-ac45452bc4d2                True    1        .../provider...
.../upbound-provider-family-aws-461aea25f5b4     True    1        .../provider...
.../upbound-provider-family-azure-f70e43ba7cb1   True    1        .../provider...
.../upbound-provider-family-gcp-d0f27e03505b     True    1        .../provider...
```

Now that all the providers are `HEALTHY`, we can proceed to configure them.

Here's the configuration for the provider that matches your choice of the hyperscaler.

```
cat providers/$HYPERSCALER-config.yaml
```

The output is as follows.

```
---
apiVersion: gcp.upbound.io/v1beta1
kind: ProviderConfig
metadata:
  name: default
spec:
  projectID: dot-20231226202303
  credentials:
    source: Secret
    secretRef:
      namespace: crossplane-system
      name: gcp-creds
      key: creds
```

We already learned how to work with providers so there's probably no need to explain that Provider Configuration. The only important note is that, even though we are creating Compositions for all three hyperscalers, we'll configure only one of them since that should be sufficient for what we are about to explore next.

So, let's apply the provider config,...

```
kubectl apply --filename providers/$HYPERSCALER-config.yaml
```

...and, retrieve the Compositions we created earlier.

```
kubectl get compositions
```

```
NAME              XR-KIND XR-APIVERSION                    AGE
aws-postgresql    SQL     devopstoolkitseries.com/v1alpha1 6m55s
azure-postgresql  SQL     devopstoolkitseries.com/v1alpha1 6m55s
google-postgresql SQL     devopstoolkitseries.com/v1alpha1 6m55s
```

All three Compositions associated with the `SQL` kind are up and running and we can, finally, start managing database servers. To do that, we'll have to modify the Composite Resource we used before. Here's the updated version.

```
cat examples/$HYPERSCALER-sql-v1.yaml
```

The output is as follows.

```
---
apiVersion: v1
kind: Secret
metadata:
  name: my-db-password
  namespace: crossplane-system
data:
  password: cG9zdGdyZXM=
---
apiVersion: devopstoolkitseries.com/v1alpha1
kind: SQL
metadata:
  name: my-db
spec:
  compositionSelector:
    matchLabels:
      provider: google
      db: postgresql
```

To begin with, we're defining a "standard" Kubernetes Secret that contains the `password` that will be used as the initial password for the database server. We already saw the reference to that Secret when we explored the Composition itself.

Besides the Secret, there's a modified version of the `SQL` definition we created earlier. While the `spec` field was empty before, now it contains `spec.compositionSelector`. That's one of the ways to select which variation, which implementation of the `SQL` one wants to use. In this case, it's clear that it is **Google Cloud**, but it could be **AWS**, or **Azure** as well.

The interesting thing about that `SQL` definition is that we are letting consumers of the service choose what they want without having to deal with all the implementation details. A user could choose to run a database in any of the major hyperscalers with a change to the `provider` label. What will happen in the background is completely different since each hyperscaler works differently yet, to a user of the `SQL` service it is all the same. Those differences are becoming implementation details he or she does not care about.

If we created additional Compositions, we could have enabled people to choose between, let's say, **PostgreSQL** and **MySQL**, or anything else we want. Still, for the sake of keeping this chapter relatively short, I did not include additional Compositions. PostgreSQL running in **AWS**, **Azure**, and **Google Cloud** should be more than enough, for now.

Another thing you'll notice is that we are not letting users choose anything but the provider. We could create a better experience by, for example, letting people choose the size of the database server or the version of PostgreSQL. We might do that later. For now, we're keeping it simple. The only choice that can be made is the `provider`.

Let's apply the `Secret` and `SQL` Composite Resource,...

```
kubectl apply --filename examples/$HYPERSCALER-sql-v1.yaml
```

...and execute `crossplane trace` (we explored it in the first chapter) to see the `SQL` and all the child resources it might create.

```
crossplane beta trace sql my-db
```

The output is as follows.

```
NAME                              SYNCED READY STATUS                                \

SQL/my-db                         True   False Creating...
├─ DatabaseInstance/my-db-schrw True   False Creating                                \

└─ User/my-db-mdmsv               True   False Creating
```

In the case of Google Cloud, we can see that `SQL/my-db` created two resources; `DatabaseInstance` and `User`. Both are not `READY` and the status says that it is `Creating` them. As a result, the parent resource `SQL/my-db` is also not `READY`.

It will take a while for all the resources spun up from the `SQL` **Composite Resource** to be ready. How much it takes varies from one hyperscaler to another as well as the number of resources that should be created. Take a break. Get some coffee. When you're back, the process should finish, and we can execute the `crossplane trace` command again.

```
crossplane beta trace sql my-db
```

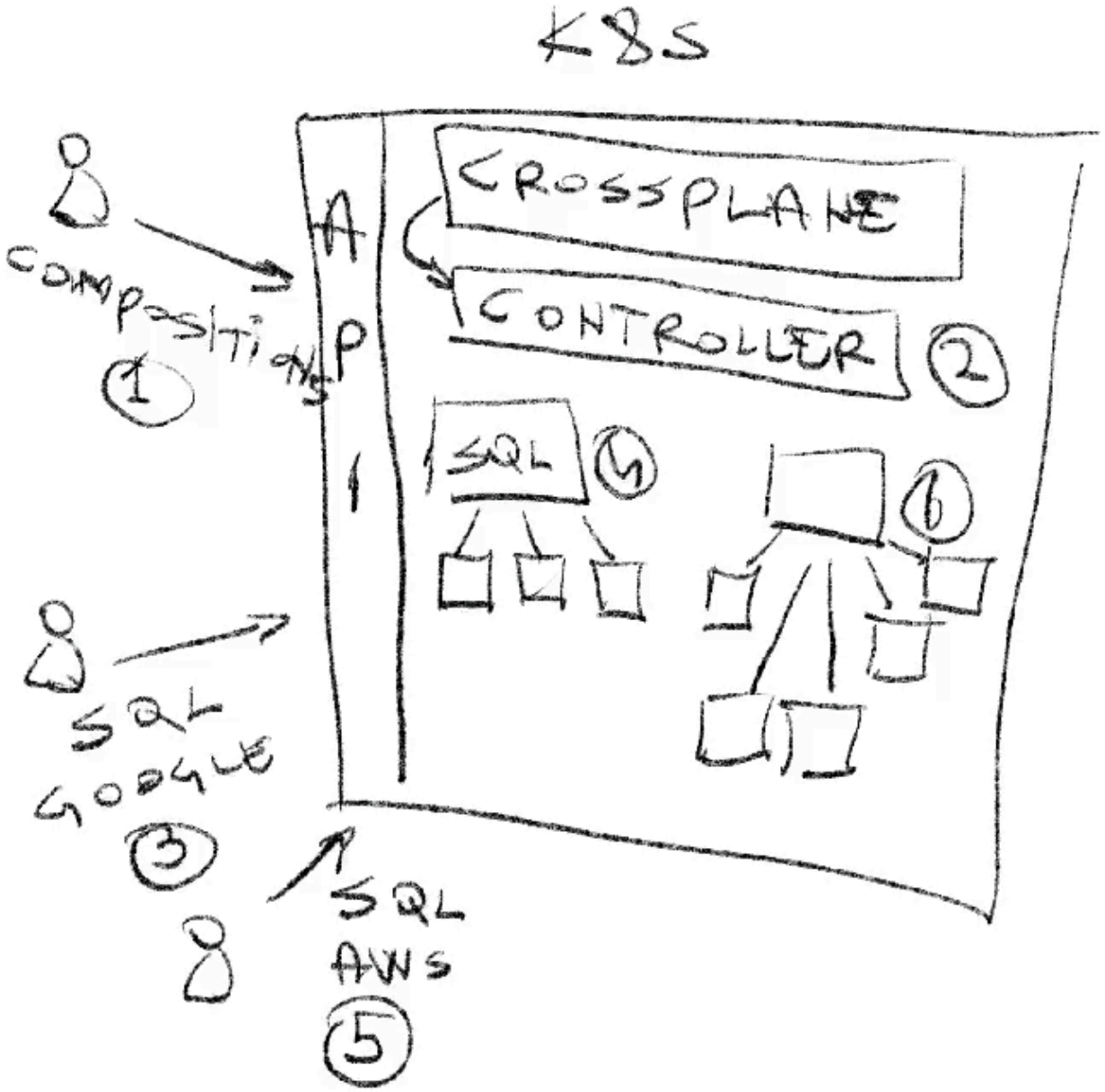
The `STATUS` of all the resources is now `Available`.

We did it. We managed to create a service that enables everyone to create and manage **PostgreSQL** in any of the three major hyperscalers and we made it as easy for them as it can get.

If you are a skeptic and do not take my word for granted, you can open the console of your favorite hyperscaler and see that the database server and all the related resources are indeed up and running.

So, what did we build?

We created **Compositions** for SQL database servers in AWS, Google Cloud, and Azure (1). Crossplane, in turn, created a Controller that watches for **Composite Resources** (2). From now on, when someone applies a Composite Resource to the control plane cluster, the controller will "expand" it into all the Managed Resources required to run the PostgreSQL server in the selected hyperscaler. If, for example, the `matchLabels.provider` is set to `google` (3), the Crossplane controller will expand the Composite Resource into Managed Resources required to run the database server in Google Cloud (4). Similarly, if someone applies a Composite Resource that sets the `matchLabels.provider` to `aws` (5), the controller will expand it into Managed Resources required to run the database server in AWS.



Now, to be honest, what we have done so far is far from perfect. Consumers of the `SQL` service have **no influence** over the outcome, all Crossplane resources are **cluster-scoped**, which is far from perfect and potentially insecure, database servers we are creating have **no databases** inside them, and quite a few other things. We'll fix or implement all of those, and quite a few others. We just started. From now on, we'll be improving those compositions until we reach **perfection**.

The first thing we should fix is the selectors we used.

Resource References and Selectors

Let's get back for a moment and take a look at one of the definitions we used in the previous chapter.

```
cat examples/$HYPERSCALER-vm.yaml
```

The output is as follows (truncated for brevity).

```
---
apiVersion: compute.gcp.upbound.io/v1beta1
kind: Instance
metadata:
  name: my-vm
spec:
  forProvider:
    ...
    networkInterface:
      - networkRef:
          name: dot-network
    ...
---
apiVersion: compute.gcp.upbound.io/v1beta1
kind: Network
metadata:
  name: dot-network
...
```

The `Instance` resource is referencing the `Network` through the `spec.forProvider.networkInterface.networkRef` set to hard-coded `dot-network`. Essentially, we told Crossplane that it can take the information it needs for `Instance` from the `Network` resource named `dot-network`.

That worked, but that wasn't necessarily the best way to reference a resource.

We used a better approach earlier in this chapter when we defined the Compositions, so let's take another look at what we did (and what I did not yet explain).

```
cat compositions/sql-v1/$HYPERSCALER.yaml
```

The output is as follows (truncated for brevity).

```
---
apiVersion: apiextensions.crossplane.io/v1
kind: Composition
metadata:
  name: google-postgresql
  ...
spec:
  ...
  resources:
  - name: sql
    base:
      apiVersion: sql.gcp.upbound.io/v1beta1
      kind: DatabaseInstance
      spec:
        ...
  - name: user
    base:
      apiVersion: sql.gcp.upbound.io/v1beta1
      kind: User
      spec:
        forProvider:
          ...
          instanceSelector:
            matchLabels:
              crossplane.io/composite: my-db
```

In this case, the `User` resource needs information about the `DatabaseInstance` where the user should reside. Since, among other things, Crossplane automatically injects labels `crossplane.io/composite` into all resources managed by a Composition, we used that one to tell the `User` how to find the `DatabaseInstance`. That, however, is a bad solution. It contains the hard-coded value `my-db`. If we created a **Composite Resource** named anything else, the `User` would either fail to find the `DatabaseInstance` or it would find a wrong one (the one from some other Composition).

Fortunately, there is a much better and easier way to reference resources within a Composition.

Let's take a look at an updated version of the Composition.

```
cat compositions/sql-v2/$HYPERSCALER.yaml
```

The output is as follows (truncated for brevity).

```
---
apiVersion: apiextensions.crossplane.io/v1
kind: Composition
metadata:
  name: google-postgresql
  ...
spec:
  ..
  resources:
  - name: sql
    base:
      apiVersion: sql.gcp.upbound.io/v1beta1
      kind: DatabaseInstance
      spec:
        forProvider:
          ...
  - name: user
    base:
      apiVersion: sql.gcp.upbound.io/v1beta1
      kind: User
      spec:
        forProvider:
          ...
          instanceSelector:
            matchControllerRef: true
```

This time, instead of referencing (selecting) resources using names or labels we are setting `matchControllerRef` to `true`. That can be translated to: "If you need information from the `DatabaseInstance`, find it yourself. It's somewhere in the Composition. Don't ask me how to find it. Figure it out."

More often than not, `matchControllerRef` will be the main, if not the only way you'll reference managed resources within a Composition. Nevertheless, that's not the only way to select resources. We'll explore others later and, if you are impatient, you can consult the documentation.

All that's left is to apply modified **Compositions**.

```
kubectl apply --filename compositions/sql-v1
```

You will not notice any tangible change to managed resources since the change we just applied is only a better way to accomplish the same result as what we had before. If we created a Composite Resource that was not named `my-db`, then we would see a different outcome since the previous version would fail due to the hard-coded `my-db` selector.

Next, we'll explore **patching** which is probably one of the most important features of Compositions. Patching enables us to customize the experience. But, before we dive into it, we'll delete the Composition and, through it, all the child resources it manages.

```
kubectl delete --filename examples/$HYPERSCALER-sql-v1.yaml
```

There's one more thing I want to show by listing all the Managed Resources. We need to do that before all the resources are removed.

```
kubectl get managed
```

The output is as follows (truncated for brevity).

```
NAME                              READY SYNCED EXTERNAL-NAME AGE
databaseinstance.../my-db-schrw False True   my-db-schrw   17m
```

Take a look at the names of the resources. My `databaseinstance` is called `my-db-schrw`. That's bad. A small annoyance is that we might want to have predictable names for Kubernetes resources. Maybe we would prefer it to be called `my-db` (without the randomized suffix). More importantly, that is the name of the database server that was created in Google Cloud. You can observe that through the `EXTERNAL-NAME` column in that output. We should be able to name resources any way we want. Right? As a matter of fact, that will be a good example we can use next when we explore patching.

Wait until all the resources are deleted before moving to patching.

Patching

The time has come to extend our Composite Resource Definition.

Let's say that we would like to enable users of our Database-as-a-Service solution to be able to specify the **version** of the database and the **size**. To make it more interesting, we'll try to avoid people having to know what are all the available sizes in **AWS**, **Azure**, and **Google Cloud** by allowing them to choose from three sizes; **small**, **medium**, and **large**. We'll figure out how to map those sizes to the correct values in the hyperscaler they choose.

How does that sound?

A potential solution is in the updated version of the definition so let's take a look at it.

```
cat compositions/sql-v3/definition.yaml
```

The output is as follows (truncated for brevity).

```
---
apiVersion: apiextensions.crossplane.io/v1
kind: CompositeResourceDefinition
metadata:
  name: sqls.devopstoolkitseries.com
spec:
  ...
  versions:
  - name: v1alpha1
    ...
    schema:
      openAPIV3Schema:
        type: object
        properties:
          spec:
            type: object
            properties:
              id:
                type: string
                description: Database ID
              parameters:
                type: object
                properties:
                  version:
                    description: The DB version depends...
                    type: string
                  size:
                    description: "Supported sizes: small, medium, large"
                    type: string
                    default: small
                required:
                - version
            required:
            - parameters
```

The previous version had the `openAPIV3Schema` empty. Now it defines two fields; `id` and `parameters`. While `id` is a `string`, `parameters` is an `object` meaning that it contains additional fields `version` and `size`.

Besides definitions of the fields that form the schema, `version`, and `parameters` fields are set as `required`.

Everything we defined in `spec.versions[].schema.openAPIV3Schema` follows the same rules as those we'd follow when defining Kubernetes `CustomResourceDefinition`. So, the experience

required to write `CompositeResourceDefinition` is the same as creating `CustomResourceDefinition` (plus a few additional fields which we might comment on later).

That's it. That's all there is to it (for now), so let's `apply` the modified version of the definition.

```
kubectl apply --filename compositions/sql-v3/definition.yaml
```

Now that we introduced a few additional fields in the Composite Resource Definition, we should modify our Compositions to take advantage of those changes.

Let's take a look at a modified version of the Composition.

```
cat compositions/sql-v3/$HYPERSCALER.yaml
```

There are a few changes we should discuss, so I'll break the output into smaller pieces.

A part of the output is as follows.

```
---
apiVersion: apiextensions.crossplane.io/v1
kind: Composition
...
spec:
  ...
  patchSets:
  - name: metadata
    patches:
    - fromFieldPath: metadata.annotations
      toFieldPath: metadata.annotations
    - fromFieldPath: spec.id
      toFieldPath: metadata.name
  ...
```

To begin with, there is `spec.patchSets` with a `name` and a list of `patches`. Those happen to be using the most common pattern for patching with `fromFieldPath` and `toFieldPath`. We can translate them as take `metadata.annotations` value from the Composite Resource and put it into `metadata.annotations` of a Managed Resource. So, whichever annotations we define in the Composite Resource will be the annotations propagated to the Managed Resource. As you already know, `metadata.annotations` are "standard" Kubernetes fields available in any resource.

A more interesting patch is the one that takes `spec.id` and puts it into `metadata.name`. Actually, there is nothing special about it from the patching perspective. "Take the value from the parent resource and put it to a resource managed by that Composition". What makes it interesting, rather than special, is that `spec.id` is a custom field. That's one of the fields we added to the new definition as a way to enable users to specify a unique identifier for database resources.

I mentioned that a patch like the one we're discussing (there are other types) takes values from the Composition Resources and puts them into a resource managed by that Composition. However, that `patchSet` does not specify which resources will be patched. We'll see how to tell Crossplane which resources to patch soon. Right now we'll move to the next change.

Not only that parts of that Composition were added or updated, but some were removed.

For example, `spec.resources[0].base.spec.forProvider.rootPasswordSecretRef.name` from the Google Composition is now gone completely. In the previous iteration, it contained the hard-coded value `my-db-password` and we already established that hard-coded values are not a good idea if they vary from one resource to another. The name of the secret that contains the password should not be `my-db-password` but the name of the Composite Resource (whichever one chooses) with a suffix `-password`. So, I removed the hard-coded value, and we'll see soon what we'll use instead.

Similarly, `spec.resources[0].base.spec.forProvider.databaseVersion` that was set to hard-coded value `POSTGRES_13` is gone as well. The value of that field should be replaced by whatever someone chooses to put as the value of the `version` field we added to the definition. The same is true for the `tier`. It's gone as well and it should be replaced with the `size`.

Another snippet of the output is as follows.

```
---
apiVersion: apiextensions.crossplane.io/v1
kind: Composition
...
spec:
  ...
  resources:
  - name: sql
    ...
    patches:
    - type: PatchSet
      patchSetName: metadata
    - fromFieldPath: spec.parameters.version
      toFieldPath: spec.forProvider.databaseVersion
      transforms:
      - type: string
        string:
          fmt: POSTGRES_%s
    - fromFieldPath: spec.parameters.size
      toFieldPath: spec.forProvider.settings[0].tier
      transforms:
      - type: map
        map:
          small: db-custom-1-3840
```

```
25            medium: db-custom-16-61440
26            large: db-custom-64-245760
27      ...
```

This is the part where we patch specific resources. The first one in the Google Composition (`sql`) got the `patches` section that has the `type` set to `PatchSet` followed by the `patchSetName` set to `metadata`. That's where the `patchSets` we commented on earlier are used. Instead of defining repetitive patches over and over again, we defined them as `patchSets` and now we're telling Crossplane to apply them to the `sql` resource. As a result, the `annotations` and the `name` will be patched with values from the Composite Resource.

The next patch contains the `fromFieldPath` and `toFieldPath` just as those we saw in `patchSets`. Its goal is to replace the `databaseVersion` in Google Cloud managed database with whatever someone chooses to be the `version`. But there is a problem.

While we expect users to specify something like `13` as the PostgreSQL version, Google Cloud expects it to be something like `POSTGRES_13`.

I do not want to force users to deal with the intricacies of specific hyperscalers. Instead, I consider it my job to translate expected input (e.g., `13`) into input required by a hyperscaler (`POSTGRES_13`). To mitigate that, the second patch adds `transforms` to the mix. In this specific case, it can be translated to "transform the input value defined as a `string` into this format (`fmt`): `POSTGRES_%s`. As a result, `%s` will be replaced with whichever value is retrieved from `spec.parameters.version`. So, apart from retrieving the value from one resource and using it to patch another, we are also applying a transformation which, in this case, is to format a string.

The next one is also using a transformation but in a very different way. It takes the `spec.parameters.size` value and uses it to patch the `spec.forProvider.settings[0].tier` field. That's the `size` field we added to the definition and, as you probably remember, the goal of the `size` field is to allow users to specify whether they want a `small`, a `medium`, or a `large` database server. Now, as you can probably imagine, hyperscalers have a large variety of sizes we can choose from, and none of them is `small`, `medium`, or `large`. So, we have to do the translation and that's why this patch has `type` set to `map`. It allows us to map an input value into something else. It acts as a map in any programming language.

In this case, we are telling Crossplane that it should convert `small` to `db-custom-1-3840`, `medium` to `db-custom-16-61440`, and `large` to `db-custom-64-245760`. If you're looking at AWS or Azure, you'll see a similar mapping but with different values. They are based on the sizes those hyperscalers offer.

There are a few other patches but they all follow the same pattern so there's probably no need to go through them. Instead, we'll apply modified Compositions before we see them in action.

```
1 kubectl apply --filename compositions/sql-v3
```

Finally, we can incorporate those new "capabilities" into the **Composite Resource** we used so far. Here's the updated version.

```
cat examples/$HYPERSCALER-sql-v3.yaml
```

The output is as follows.

```
...
apiVersion: devopstoolkitseries.com/v1alpha1
kind: SQL
metadata:
  name: my-db
  annotations:
    organization: DevOps Toolkit
    author: Viktor Farcic <viktor@farcic.com>
spec:
  id: my-db
  compositionSelector:
    matchLabels:
      provider: google
      db: postgresql
  parameters:
    version: "13"
    size: small
```

We added a few `metadata.annotations` so that we can test that one of the patches we defined works. Further on, `spec.id`, `spec.parameters.version`, and `spec.parameters.size` fields were added as well.

That manifest can be translated to "give me a `postgresql` server in `google`, make sure that the version is `13`, and make it `small` without making me learn which nodes in Google classify as small." The user, the person who defined that manifest, gained more freedom to specify what matters while still not having to deal with low-level details and intricacies of the hyperscaler of choice.

*If you are using **Azure**, you'll notice that we changed the name from `my-db` to `my-db-2`. Azure does not allow repeated names for some of its resources like SQL, even if those resources were deleted. So, the `spec.id` changed to `my-db-2`. Otherwise, since we already have `my-db` and deleted it, it would fail to create a new one with the same name.*

Let's apply the Composite Resource...

```
kubectl apply --filename examples/$HYPERSCALER-sql-v3.yaml
```

...and trace the progress.

```
crossplane beta trace sql my-db
```

The output is as follows (truncated for brevity).

```
NAME                        SYNCED READY STATUS
SQL/my-db                   True   False Creating...
├─ DatabaseInstance/my-db True   False Creating
└─ User/my-db               False  False ReconcileError:...
```

Those resources will eventually be ready and, while waiting for that to happen, we can make a few observations.

To begin with, the names of managed resources are now `my-db`. There is no auto-generated suffix anymore. We got that change because one of the patches made sure that the name of a resource is the same as the value of the new `spec.id` field we added to the definition. Using auto-generated suffixes is a good practice that helps us avoid conflicts, but I like my resources to have "proper" names so we ignored the "best practice".

Next, we'll check whether the annotations we added to the Composite Resource were indeed added to Managed Resources. To do that, we'll create an environment variable `XR` with the full name of any of the Managed Resources we created...

```
# Replace `[...]` with the full name of the one the Managed Resources.
export XR=[...]
```

...and output that resource as YAML.

```
kubectl get $XR --output yaml
```

The output is as follows (truncated for brevity).

```
apiVersion: sql.gcp.upbound.io/v1beta1
kind: DatabaseInstance
metadata:
  annotations:
    author: Viktor Farcic <viktor@farcic.com>
    ...
    organization: DevOps Toolkit
  ...
```

We can see that the Managed Resource was indeed patched with annotations from the Composite Resource.

Feel free to confirm that other patches were applied as well, or simply trust me when I say that they all did. The `size` and the `version` we specified were applied to the relevant resources.

Now that the **PostgreSQL** server is up and running, we need to figure out how to connect to it. Otherwise, what's the point of having a database that cannot be used?

Managing Connection Secrets

We have a **PostgreSQL** database but, right now, it is just sitting there not being used by anyone or anything. We need a way to connect to it. Fortunately, Crossplane can **combine all the secrets** generated by the Managed Resources into a single Kubernetes Secret. All we have to do is tell it where to put that secret.

Let's take a look at a modified version of the Composition.

```
cat compositions/sql-v4/$HYPERSCALER.yaml
```

The output is as follows (truncated for brevity).

```
---
apiVersion: apiextensions.crossplane.io/v1
kind: Composition
...
spec:
  writeConnectionSecretsToNamespace: crossplane-system
  ...
  resources:
  - name: sql
    base:
      apiVersion: sql.gcp.upbound.io/v1beta1
      kind: DatabaseInstance
      spec:
        ...
        writeConnectionSecretToRef:
          namespace: crossplane-system
    patches:
    ...
    - fromFieldPath: spec.id
      toFieldPath: spec.writeConnectionSecretToRef.name
    ...
```

To begin with, we're telling Crossplane through `spec.writeConnectionSecretsToNamespace` to store the secret that contains all the confidential and connection information generated through Managed Resources in the `crossplane-system` Namespace. Think of that field as being the default location for the Secret which can be overwritten for specific resources.

Further down, we are overwriting the Namespace where the secret will be stored through the `spec.resources[0].base.spec.writeConnectionSecretToRef.namespace` value. That wasn't really necessary since the value (`crossplane-system`) is the same as what we set through

`spec.writeConnectionSecretsToNamespace` but I wanted to show that we can overwrite the Namespace for the Secret from that specific resource. That capability will become important later when we switch to Namespace-scoped resources.

Finally, since it would be silly to have all SQL Secrets with the same name, we are using patching to set the value of `spec.writeConnectionSecretToRef.name` of that resource to whatever the `spec.id` is in the Composite Resource.

That's it, for now, so let's apply the Compositions...

```
kubectl apply --filename compositions/sql-v4
```

...and output the Secrets in the `crossplane-system` Namespace.

```
kubectl --namespace crossplane-system get secrets
```

The output is as follows (truncated for brevity).

```
NAME           TYPE                                  DATA AGE
...
my-db          connection.crossplane.io/v1alpha1 10   13s
my-db-password Opaque                                1    8m24s
...
```

We can see that, besides the `my-db-password` secret we created as a way to provide the initial password, there is now `my-db` that should contain all the information on how to connect to the database server.

We have a slight complication with the demo since the database and the secret are called `my-db` if you're using AWS or Google Cloud, or `my-db-` with a timestamp suffix if it's **Azure**. To mitigate that discrepancy, we'll store the name of the database to an environment variable.

Locate the secret with the name that starts with `my-db-` and has a timestamp suffix if you are using **Azure**, copy it, and use it as the value in the command that follows.

```
export DB=my-db
```

Now we can retrieve the Secret and output it to YAML to get a sneak peak into the data it contains.

```
kubectl --namespace crossplane-system get secret $DB \
    --output yaml
```

The output is as follows (truncated for brevity).

```
apiVersion: v1
data:
  attribute.root_password: cG9zdGdyZXM=
  connectionName: ZG90LTIwMjQwMTAzMTk0MDU2OnVzLWVhc3QxOm15LWRi
  password: cG9zdGdyZXM=
  privateIP: ""
  publicIP: MzUuMTk2LjQ3LjEwNQ==
  serverCACertificateCert: LS0tLS1CRUd...
  serverCACertificateCommonName: Qz1VUyxP...=
  serverCACertificateCreateTime: MjAyNC0wMS0wM1QxOT...
  serverCACertificateExpirationTime: MjAzMy0xMi0zMV...
  serverCACertificateSha1Fingerprint: OWMyNzVjNjB...
kind: Secret
...
```

The output will differ from one hyperscaler to another since the information each provides and the keys used to represent that information might be different. We'll see, later on, how we can unify that. For now, what matters, is that all the information is available. In the case of **Google Cloud**, we can see that, among other information, the `password` and the `publicIP` are available.

As you probably already know, data in Kubernetes Secrets is base64 encoded so we need to decode the data if we would like to use it to connect to the database server.

We'll do that soon.

Combining Providers in Compositions

We created a database server so the next logical step would be to try to connect to it and confirm that it works as expected.

We'll get the information like the user, the password, and the host from the Secret Crossplane provided.

Since we have not yet unified the format of that Secret, the commands might differ from one hyperscaler to another.

Let's start with the user.

*Execute the command that follows only if you are using **Azure** or **AWS**.*

```
export PGUSER=$(kubectl --namespace crossplane-system \
    get secret $DB --output jsonpath="{.data.username}" \
    | base64 -d)
```

*Execute the command that follows only if you are using **Google Cloud**.*

```
export PGUSER=postgres
```

Next, we'll retrieve the password.

```
export PGPASSWORD=$(kubectl --namespace crossplane-system \
    get secret $DB --output jsonpath="{.data.password}" \
    | base64 -d)
```

Then we'll get the key that holds the host in the Secret.

*Execute the command that follows only if you are using **Azure**.*

```
export HOST_KEY=endpoint
```

*Execute the command that follows only if you are using **AWS**.*

```
export HOST_KEY=host
```

*Execute the command that follows only if you are using **Google Cloud**.*

```
export HOST_KEY=publicIP
```

Finally, we'll use that key to retrieve the host itself.

```
export PGHOST=$(kubectl --namespace crossplane-system \
    get secret $DB --output jsonpath="{.data.$HOST_KEY}" \
    | base64 -d)
```

Now we're ready to connect to the database.

Since I did not want to assume that you have `psql` on your laptop, and since I was too lazy to add it to `shell.nix`, we'll run it inside a container based on the `bitnami/postgresql` image.

```
kubectl run postgresql-client --rm -ti --restart='Never' \
    --image docker.io/bitnami/postgresql:16 \
    --env PGPASSWORD=$PGPASSWORD --env PGHOST=$PGHOST \
    --env PGUSER=$PGUSER --command -- sh
```

Once inside the container with `psql`, we can execute the command that connects to the remote PostgreSQL server...

```
psql --host $PGHOST -U $PGUSER -d postgres -p 5432
```

...and list all the databases.

```
\l
```

We have a problem though. We can see system-level databases inside that server, but not one that we would use from, let's say an application.

We should try to extend our Compositions to add a database to the PostgreSQL server. While we're at it, we might just as well extend them to provide a unified format for the Secret we're generating.

So, let's get out of the `psql` shell...

```
exit
```

...and out of the container.

```
exit
```

Right now, we are missing a way to generate a database inside the server and to create a uniform Secret with the information on how to connect to the database. We cannot do either of those tasks with the providers we're currently using. **AWS**, **Azure**, and **Google Cloud** providers do not have resource definitions for those types of operations. Fortunately, other providers can do just what we need and there is no limit to which resources we can include in Compositions. The fact that, let's say, one of the compositions manages PostgreSQL in Google, does not mean that we are limited only to what we can do through the Google Cloud API. We can, for example, include the **SQL provider** to manage databases inside database servers and we can add the **Kubernetes provider** to create any Kubernetes resource, including the Secret we discussed.

Let's take a look at yet another version of the Composition.

```
cat compositions/sql-v5/$HYPERSCALER.yaml
```

There are four new resources in that Composition, so I'll break the output into smaller pieces; one for each of the new resources.

A part of the output is as follows (truncated for brevity).

```
---
apiVersion: apiextensions.crossplane.io/v1
kind: Composition
metadata:
  name: google-postgresql
  ...
spec:
  ...
  resources:
  ...
  - name: sql-config
    base:
      apiVersion: postgresql.sql.crossplane.io/v1alpha1
      kind: ProviderConfig
      metadata:
        name: default
      spec:
        credentials:
          source: PostgreSQLConnectionSecret
          connectionSecretRef:
            namespace: crossplane-system
        sslMode: require
    patches:
    - type: PatchSet
      patchSetName: metadata
    - fromFieldPath: spec.id
      toFieldPath: spec.credentials.connectionSecretRef.name
  ...
```

We can add a database through the SQL Provider[7]. For it to work correctly, it needs to be configured so that it can authenticate to the PostgreSQL server. Hence, we are doing the `ProviderConfig` as a new resource to the Composition. That config will use `credentials` from the Secret we're generating. Since the Secret name is the same as the value of the `spec.id` field in Composite Resources, we're patching the `ProviderConfig` so that `spec.credentials.connectionSecretRef.name` field of the Managed Resource has the value taken from the `spec.id` field of the Composite Resource.

There's one important thing to note here. The SQL `ProviderConfig` expects a Secret with credentials in a specific format. Hence, besides the need to have secrets with credentials in the same format no matter which hyperscaler provider we're using, this `ProviderConfig` forces us to use the specific format.

*Please note that **Azure** PostgreSQL `Server` already stores the credentials in the Secret with exactly the same format, so there's no need to create it separately. Hence, if you are using **Azure**, you'll*

[7] https://github.com/crossplane-contrib/provider-sql

notice that the third and the fourth new resources we'll explore are missing.

Now that we saw that we added `ProviderConfig` which will enable the **SQL provider** to talk to the PostgreSQL database server we are managing, we can move on to the next new resource.

The snippet with the second new resource is as follows.

```
---
apiVersion: apiextensions.crossplane.io/v1
kind: Composition
metadata:
  name: google-postgresql
  ...
spec:
  ...
  resources:
  ...
  - name: sql-db
    base:
      apiVersion: postgresql.sql.crossplane.io/v1alpha1
      kind: Database
      spec:
        forProvider: {}
    patches:
    - type: PatchSet
      patchSetName: metadata
    - fromFieldPath: spec.id
      toFieldPath: spec.providerConfigRef.name
  ...
```

This is a simple one. We're adding `Database` Managed Resource, from the SQL provider we configured earlier, and making sure that it is using the correct configuration by patching `spec.providerConfigRef.name` with the value from `spec.id`. That resource will create and manage a database with the same name as the name of the resource itself, so there's nothing special to do given the `metadata.name` field is patched through the `metadata` `PatchSet`.

Let's move to the third new resource.

The snippet with the third new resource is as follows.

```
---
apiVersion: apiextensions.crossplane.io/v1
kind: Composition
metadata:
  name: google-postgresql
  ...
spec:
  ...
  resources:
  ...
  - name: kubernetes
    base:
      apiVersion: kubernetes.crossplane.io/v1alpha1
      kind: ProviderConfig
      spec:
        credentials:
          source: InjectedIdentity
    patches:
    - fromFieldPath: metadata.annotations
      toFieldPath: metadata.annotations
    - fromFieldPath: spec.id
      toFieldPath: metadata.name
      transforms:
      - type: string
        string:
          fmt: "%s-sql"
  ...
```

We can use `object` resources from the Kubernetes provider[8] to create any Kubernetes resource, including Secrets. But, just as with the SQL provider, first, we need to apply `ProviderConfig` which will tell the **Kubernetes provider** how to find the cluster where it should create the secret. We're doing that by saying that the `source` of the `credentials` is `InjectedIdentity` which, essentially, means that it should manage resources in the cluster where it's running. It could be a different cluster and we'll explore that option in one of the upcoming chapters.

Now we can explore the last resource we added to the Composition.

The snippet with the fourth new resource is as follows.

[8] https://marketplace.upbound.io/providers/crossplane-contrib/provider-kubernetes

```yaml
---
apiVersion: apiextensions.crossplane.io/v1
kind: Composition
metadata:
  name: google-postgresql
  ...
spec:
  ...
  resources:
  ...
  - name: sql-secret
    base:
      apiVersion: kubernetes.crossplane.io/v1alpha1
      kind: Object
      metadata:
        name: sql-secret
      spec:
        forProvider:
          manifest:
            apiVersion: v1
            kind: Secret
            metadata:
              namespace: crossplane-system
            data:
              port: NTQzMg==
        references:
        - patchesFrom:
            apiVersion: sql.gcp.upbound.io/v1beta1
            kind: User
            namespace: crossplane-system
            fieldPath: metadata.name
          toFieldPath: stringData.username
        - patchesFrom:
            apiVersion: v1
            kind: Secret
            namespace: crossplane-system
            fieldPath: data.password
          toFieldPath: data.password
        - patchesFrom:
            apiVersion: v1
            kind: Secret
            namespace: crossplane-system
            fieldPath: data.publicIP
```

```
44            toFieldPath: data.endpoint
45      patches:
46      - type: PatchSet
47        patchSetName: metadata
48      - fromFieldPath: spec.id
49        toFieldPath: spec.references[0].patchesFrom.name
50      - fromFieldPath: spec.id
51        toFieldPath: spec.references[1].patchesFrom.name
52        transforms:
53        - type: string
54          string:
55            fmt: "%s-password"
56      - fromFieldPath: spec.id
57        toFieldPath: spec.references[2].patchesFrom.name
58      - fromFieldPath: spec.id
59        toFieldPath: spec.forProvider.manifest.metadata.name
60      - fromFieldPath: spec.id
61        toFieldPath: spec.providerConfigRef.name
62        transforms:
63        - type: string
64          string:
65            fmt: "%s-sql"
```

That's a long one.

We are creating an `Object` which can be any Kubernetes resource. In this case, we're creating a `Secret` with a hard-coded `port` set to encoded value `NTQzMg==` which, in its decoded form is `5432`. There's no reason not to have it hard-coded since it is always that port.

The interesting part is the `references` section with a few `patchesFrom`. Those allow us to get information from any resources in a Kubernetes cluster, no matter whether that resource was created by Crossplane or any other process. The first one is getting a value from the `User` resource and adding it to the secret as the `username`. The second takes the value from one Secret and puts it into the `password`, and the third one from a different Secret which will end up being the `endpoint`. All that might not be obvious just by looking at the `patchesFrom` sections. Parts of that puzzle are in the `patches` which, in this case, are patching `patchesFrom`.

We're still missing something. We introduced two new Providers into our Compositions and we need to deploy them as well, at least until we figure out how to automate that part. This time, however, one of the providers will require a bit of extra work.

Let's take a look at the updated version of the providers.

```
1 cat providers/sql-v5.yaml
```

The output is as follows (truncated for brevity).

```yaml
...
apiVersion: pkg.crossplane.io/v1
kind: Provider
metadata:
  name: provider-sql
spec:
  package: crossplane/provider-sql:v0.7.0
---
apiVersion: v1
kind: ServiceAccount
metadata:
  name: crossplane-provider-kubernetes
  namespace: crossplane-system
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: crossplane-provider-kubernetes
subjects:
- kind: ServiceAccount
  name: crossplane-provider-kubernetes
  namespace: crossplane-system
roleRef:
  kind: ClusterRole
  name: cluster-admin
  apiGroup: rbac.authorization.k8s.io
---
apiVersion: pkg.crossplane.io/v1alpha1
kind: ControllerConfig
metadata:
  name: crossplane-provider-kubernetes
spec:
  serviceAccountName: crossplane-provider-kubernetes
---
apiVersion: pkg.crossplane.io/v1
kind: Provider
metadata:
  name: crossplane-provider-kubernetes
spec:
  package: xpkg.upbound.io/crossplane-contrib/provider-kubernetes:v0.9.0
  controllerConfigRef:
    name: crossplane-provider-kubernetes
```

The first resource is easy to explain. It is the `crossplane/provider-sql` Provider. The Kubernetes provider is a bit trickier since we need to make sure that it has permissions to create additional resources through the Kubernetes API. So, we are creating a `ServiceAccount` and `ClusterRoleBinding` that gives that `ServiceAccount` permissions. Further on, there is `ControllerConfig` which references that `ServiceAccount`. Finally, the Kubernetes `Provider` is configured through `controllerConfigRef` to use that `ControllerConfig`. It's a handful, but it works and is pretty much how you would create `ServiceAccount` and `ClusterRoleBinding` for any other non-Crossplane process that needs to perform some operations through the Kubernetes API inside the same cluster.

Let's apply those providers,...

```
kubectl apply --filename providers/sql-v5.yaml
```

...output all package revisions,...

```
kubectl get pkgrev
```

...and wait until they are all `HEALTHY`.

Once all package revisions are `HEALTHY`, we can go ahead and apply modified Compositions.

```
kubectl apply --filename compositions/sql-v5
```

Now we can go back to the container with `psql`,...

```
kubectl run postgresql-client --rm -ti --restart='Never' \
    --image docker.io/bitnami/postgresql:16 \
    --env PGPASSWORD=$PGPASSWORD --env PGHOST=$PGHOST \
    --env PGUSER=$PGUSER --command -- sh
```

...enter the client,...

```
psql --host $PGHOST -U $PGUSER -d postgres -p 5432
```

...and list all databases.

```
\l
```

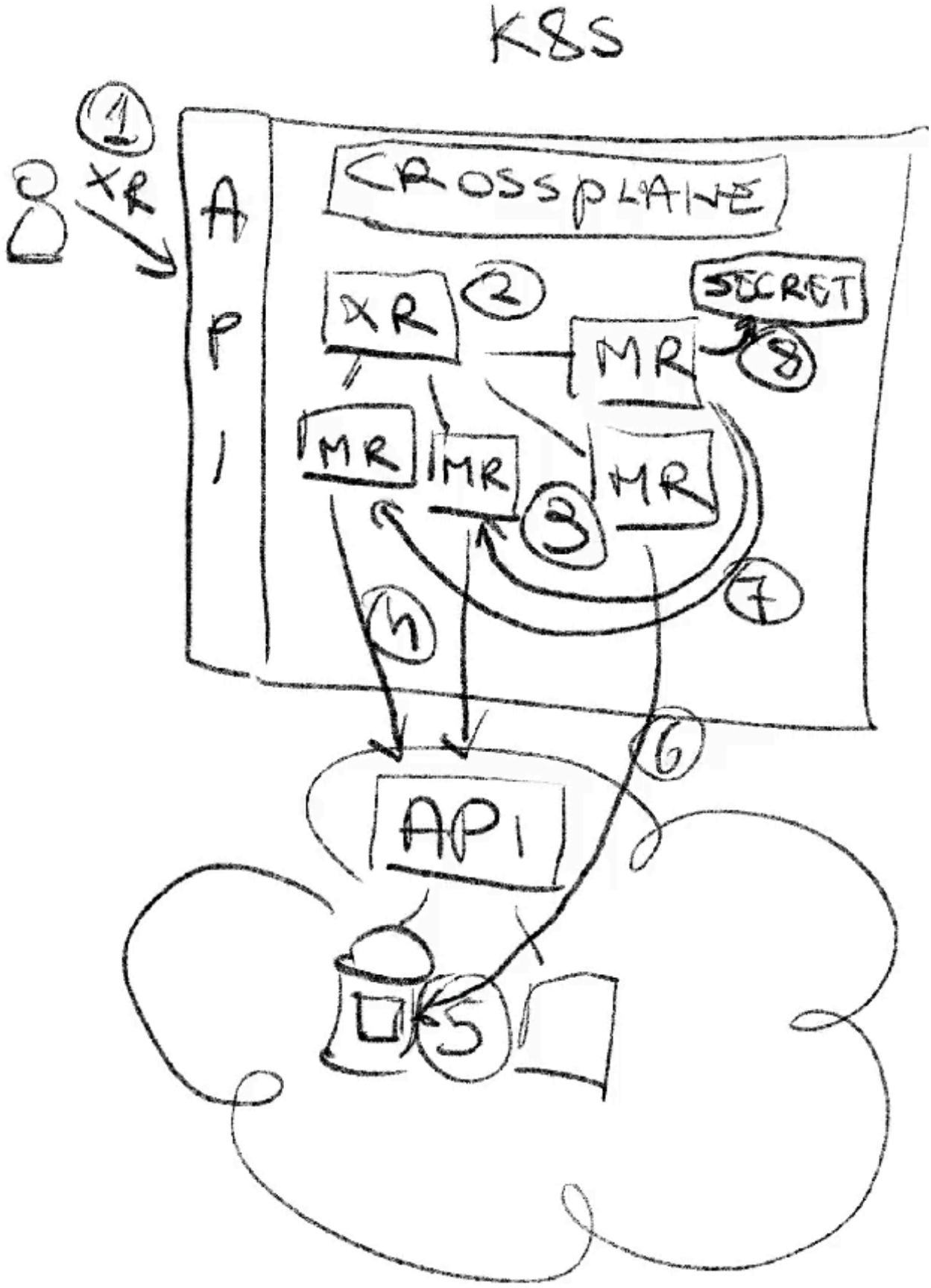
The output is as follows (truncated for brevity).

```
...
     Name      |       Owner       |...
---------------+-------------------+...
 cloudsqladmin | cloudsqladmin     |...
 my-db         | my-db             |...
 postgres      | cloudsqlsuperuser |...
 template0     | cloudsqladmin     |...
               |                   |...
 template1     | cloudsqlsuperuser |...
               |                   |...
(5 rows)
```

We can see that this time, there is a new database `my-db`. The mission was successful. From now on, every time someone chooses to create a PostgreSQL server in any of the hyperscalers, a database and a Secret with a uniform format will be created as well.

Here's the summary of what we did.

We created a Composite Resource (XR) (1, 2) which created Managed Resources (MR) (3). Some of those Managed Resources (MR) created a database server and networking (4, 5) through the hyperscaler API. We had that before. What's new is that one of those Managed Resources (MR) created a database directly inside the database server (6) and the other collected information from other resources (7) and generated the Kubernetes Secret inside the same cluster (8).


K8S
1
XR
A
P
I
CROSSPLANE
XR
2
SECRET
MR
8
MR
MR
MR
3
4
7
6
API
5

This is still not a good solution though. We need to switch from Cluster-scoped Composite Resources to Namespace-scoped Composite Claims. But, before we do that, we'll remove the Composite Resource so that we can start over.

Let's exit the `psql` client,...

```
exit
```

...and the container,...

```
exit
```

...delete the Composite Resource,...

```
kubectl delete --filename examples/$HYPERSCALER-sql-v3.yaml
```

...and list all Managed Resources.

```
kubectl get managed
```

It might take a while until all the Managed Resources are deleted. So, repeat the `kubectl get managed` command until you see that everything is gone.

*The `database` might not be deleted if the database server it was created in ends up being deleted first. That issue will be fixed later when we explore Crossplane `Usage`. Execute the command that follows only if the `database` resource is left.

```
kubectl patch database.postgresql.sql.crossplane.io $DB \
    --patch '{"metadata":{"finalizers":[]}}' --type=merge
```

Now we can discuss Composite Claims.

Defining Composite Claims

All the resources we created so far are cluster-scoped. That means that they are managed on the cluster level and not inside Namespaces. That is potentially problematic since Namespaces are typically used to separate teams or types of resources, apply RBAC and policies, and quite a few other things we normally do in Kubernetes, and that we will explore later. The bad news is that we cannot change the scope of Composite and Managed Resources. They are always **cluster-scoped**. However, we can add new types of Crossplane resources to the mix. We can use **Composite Claims** which are **Namespace-scoped** and which can be used to **create Composite Resources**. Not only that, but with Claims we can also have more control over the location of Secrets, Objects, and other Namespace-scoped resources. As a matter of fact, we already defined everything we need to use Claims but I sneakily avoided mentioning those parts.

Let's take another look at the Composite Resource Definition we applied earlier.

```
cat compositions/sql-v5/definition.yaml
```

The output is as follows (truncated for brevity).

```
---
apiVersion: apiextensions.crossplane.io/v1
kind: CompositeResourceDefinition
metadata:
  name: sqls.devopstoolkitseries.com
spec:
  ...
  names:
    kind: SQL
    plural: sqls
  claimNames:
    kind: SQLClaim
    plural: sqlclaims
  ...
```

The `spec.names` field is the one we used as the `kind` of Composite Resources. Just below it is `spec.claimNames` which does the same but for Composite Claims.

That's it. There is nothing "special" we have to do. With `spec.claimNames` we can create **Namespace-scoped Claims** instead of going directly for cluster-scoped Compositions. All we have to do, in this case, is set `kind` to `SQLClaim` instead of `SQL`. Hence, we could create a Claim right away, but we won't do that just yet. Instead, we'll make a few modifications to our Compositions. Specifically, we'll change Compositions so that the Secret with database authentication is created in the same Namespace where we'll apply the Claim.

Let's take a look at a modified version of the Composition.

```
cat compositions/sql-v6/$HYPERSCALER.yaml
```

The output is as follows (truncated for brevity).

```
---
apiVersion: apiextensions.crossplane.io/v1
kind: Composition
metadata:
  name: google-postgresql
  ...
spec:
  ...
  resources:
  - name: sql
    ...
    patches:
    ...
    - fromFieldPath: spec.claimRef.namespace
      toFieldPath: spec.forProvider.rootPasswordSecretRef.namespace
  - name: user
    ...
    patches:
    ...
    - fromFieldPath: spec.claimRef.namespace
      toFieldPath: spec.forProvider.passwordSecretRef.namespace
  - name: sql-config
    ...
    patches:
    ...
    - fromFieldPath: spec.claimRef.namespace
      toFieldPath: spec.credentials.connectionSecretRef.namespace
    ...
  - name: sql-secret
    ...
    patches:
    ...
    - fromFieldPath: spec.claimRef.namespace
      toFieldPath: spec.references[1].patchesFrom.namespace
    ...
    - fromFieldPath: spec.claimRef.namespace
      toFieldPath: spec.forProvider.manifest.metadata.namespace
```

That is a very straightforward change. The `spec.writeConnectionSecretsToNamespace` entry that was previously set to hard-coded `crossplane-system` is now gone. Similarly, all the places where we had `namespace` set to `crossplane-system` are gone as well. We don't want Secrets to be created in the `crossplane-system` Namespace anymore. We want them to be in the same Namespace where Claims are created so we replaced those hard-coded `namespace` values with `patches`. Crossplane

will automatically add `spec.claimRef.namespace` to all Composite Resources created by Claims so we're using those in patches to populate `namespace` fields.

That's it, for now. We can proceed and apply changes to the Compositions.

```
kubectl apply --filename compositions/sql-v6
```

Next, we'll make two tiny modifications to the Composite Resource we used so far, so let's take a look at a modified version.

```
cat examples/$HYPERSCALER-sql-v6.yaml
```

The output is as follows (truncated for brevity).

```
apiVersion: devopstoolkitseries.com/v1alpha1
kind: SQLClaim
...
```

The first thing you'll notice, even though it's not in the output, is that the `metadata.namespace` entry in the Secret is now gone. That's the Secret we're using to define the initial password (the one we'll get rid of later). From now on, that Secret will be created in whichever Namespace we choose to run the Claim instead of the `crossplane-system` Namespace. We already modified the Composition to know how to fetch it through `patches` that retrieve the value from the `spec.claimRef.namespace` field in Compositions.

The second change is that the `kind` is now `SQLClaim` (it was `SQL` before). We specified in the Composite Resource Definition that Composite Resources (cluster-scoped) have `kind` `SQL` and Claims have it set to `SQLClaim`. Those can be any names and you do not have to add `Claim` as a suffix, but you do need to make them unique (within the API group).

Now we can apply the Claim by executing `kubectl apply`. What makes the command different is that this time, we're specifying `--namespace` so that the Secret and the Claim are created in a specific Namespace.

```
kubectl --namespace a-team apply \
    --filename examples/$HYPERSCALER-sql-v6.yaml
```

As a result, we can retrieve all the `sqlclaims`.

```
kubectl --namespace a-team get sqlclaims
```

The output is as follows.

```
NAME  SYNCED READY CONNECTION-SECRET AGE
my-db True   False                  15s
```

If we try to create some kind of a developer portal (which we will do later), users would be working exclusively with Claims and we could hide access to Compositions and Managed resources with RBAC. That's not mandatory but, if we do something like that, we will have a clear separation between resources end-users create and manage (Claims) and resources administrators can see (everything). Those rules can be easily enforced through policies, RBAC, and quite a few other means.

Hence, end-users would only see the resources in specific Namespaces, which, in this case, would be the `SQLClaim` in the `a-team` Namespace.

Administrators, on the other hand, could still see everything through, among other means, `crossplane trace` commands.

```
crossplane beta trace sqlclaim my-db --namespace a-team
```

The output is as follows (truncated for brevity).

```
NAME                           SYNCED READY STATUS
SQLClaim/my-db (a-team)        True   False Waiting...
└─ SQL/my-db-tzvhj             True   False Creating...
   ├─ DatabaseInstance/my-db   True   False Creating
   ├─ User/my-db               False  False ReconcileError:...
   ├─ ProviderConfig/my-db     -      -
   ├─ Database/my-db           False  -     ReconcileError:...
   ├─ ProviderConfig/my-db-sql -      -
   └─ Object/my-db             True   True  Available
```

We can see that the `SQLClaim` created Composition `SQL` which, in turn, created several Managed Resources. The Claim is in the `a-team` Namespace while all other resources are cluster-scoped.

It'll take a while until all the hyperscaler resources are created.

Once all the resources are `Available`, we can confirm that the `my-db` Secret with the authentication is, this time, created in the `a-team` Namespace (the same one where the Claim is running) instead of the `crossplane-system` Namespace we hard-coded in previous iterations of the Compositions.

```
kubectl --namespace a-team get secrets
```

The output is as follows.

```
NAME            TYPE   DATA AGE
my-db           Opaque 4    2m18s
my-db-password Opaque 1    2m18s
```

The `my-db` Secret is there.

Everything seems to be working correctly and the exploration of Crossplane Compositions concluded, for now. **Compositions** are probably the most important feature of Crossplane and we will continue improving them as we progress exploring other features of Crossplane as well as the Kubernetes ecosystem as a whole.

Nevertheless, we are done with this chapter and we can proceed towards destruction.

Destroy Everything

Please execute the commands that follow to destroy everything we did in this chapter. The next chapter will start with a fresh setup.

```
chmod +x destroy/02-compositions.sh

./destroy/02-compositions.sh

exit
```

Conclusion

To take the next step in your Crossplane journey to learn about Configuration Packages and Composition Functions, check out my full book, Crossplane: the Cloud Native Control Plane[9]. It'll give you the whole picture on how to implement Crossplane and more!

[9] https://www.upbound.io/resources/lp/book/crossplane-cloud-native-control-plane