

CROSSPLANE

THE CLOUD NATIVE CONTROL PLANE

VIKTOR FARCIC



Crossplane

The Cloud Native Control Plane

Viktor Farcic

This book is for sale at <http://leanpub.com/crossplane>

This version was published on 2024-02-27



© 2024 Viktor Farcic

Contents

Introduction	1
Chapter Setup	2
A Glimpse Into the Future	4
Destroy Everything	15
Providers and Managed Resources	17
Chapter Setup	17
Crossplane Providers	17
Create Managed Resources	25
Continuous Drift-Detection and Reconciliation	35
Update Managed Resources	38
Delete Managed Resources	39
Destroy Everything	40
Compositions	41
Chapter Setup	41
Composite Resource Definitions	42
Defining Compositions	47
Resource References and Selectors	57
Patching	61
Managing Connection Secrets	67
Combining Providers in Compositions	70
Defining Composite Claims	82
Destroy Everything	87
Configuration Packages	88
Chapter Setup	88
Building Configuration Packages	89
Installing Configuration Packages	98
Destroy Everything	103
Composition Functions	104
Chapter Setup	107
What's Missing?	108
Patch and Transform Function	110

Go Templating Function	116
Auto-Ready Function	131
Building and Pushing Configuration Package	136
Destroy Everything	137
The End?	139

Introduction

Imagine if you could create, for people in your company, a platform that would provide them with the same experience they have when working with **AWS**, **Google Cloud**, **Azure**, or any other public Cloud provider. Imagine if there would be a **service** for everything they do.

Do you need a **database** that works exactly as we expect it to work in this company with all the **security**, **backup**, **compliance**, and other policies we have?

Well...

There is a service for that.

Do you need to run a **backend application in Kubernetes**? There is a service for that as well.

Do you need a **Kubernetes cluster** itself? There is yet another service for that.

All you have to do is define a **simple manifest** that contains only the things you care about and abstracts all the unavoidable complexity. From there on, all you have to do is submit that manifest, the desired state, to the control plane API and observe the actual state. Even better, you can forget about the API and just push it to **Git**.

Wouldn't that be awesome?

Wouldn't it be great if we could replicate the experience of using a **public Cloud provider** but made specifically for our needs? Wouldn't it be great if there were a clearly defined API and a clear separation between the tasks end-users need to perform and the tasks that are the responsibility of the platform itself?

If that sounds like something you might need, then **Crossplane** is just the project that will get you there. It enables us with capabilities of creating control planes based on the same principles public cloud providers have. It democratizes technology that was previously reserved mostly for big Cloud providers like **AWS**, **Azure**, and **Google Cloud**. It enables us to create **internal developer platforms**.

Let me give you a **sneak peek** into some of the capabilities we'll build throughout this book.

But, before I do that, I need to give you an early warning so that you can decide whether this book is for you or not. I will not use a typical teaching approach where there is a long introduction, followed by an even longer theory, followed by an explanation of each of the components, followed by a bunch of diagrams, and so on and so forth. This is a book that reflects the way I learn. I learn by doing. I learn by having my hands on the keyboard at all times. I learn by making mistakes and fixing them. I learn by experimenting. That's the approach I'll take with this book. I'll explain theory and concepts, and I'll show diagrams, but only through **hands-on** exercises. I'll explain what we did rather than do what I explained.

Does that make sense?

If it does, you're in the right place. Otherwise... Go away, while you still can.

With that warning out of the way, let's take a quick glimpse at some of the things we'll build throughout this book. That should give you a good idea of what you can expect and let you decide whether **Crossplane** is something you want to learn and adopt.

Chapter Setup

I lied when I said that we'll take "a quick glimpse at some of the things we'll build throughout this book". We will, but not right away. First, we need to create a control plane cluster with Crossplane, Compositions, Argo CD, and a few other tools that will be required to show you what we're building.

Now, to make things easy, instead of providing step-by-step setup instructions for this and all other chapters, I created scripts that will do that for us. You can run them and follow the instructions they present through questions, or you can choose to inspect the scripts if you prefer setting up everything manually.

However, we have a bit of a problem. To run those setup scripts as well as the instructions that follow in the hands-on parts of the book, we'll need tools. We'll need quite a few CLIs like, for example, `kubect1`, `crossplane`, `gum`, `gh`, hyperscaler-specific CLIs, and so on and so forth.

One option would be for me to give you the instructions on how to install all the CLIs we'll need. That, however, might result in you spending considerable time reading those instructions and installing those CLIs. We'll do something else. We'll run everything in **Nix**. It allows us to create ad-hoc Shell environments with all the tools we might need. It's awesome and it will help us streamline this book and avoid complications that might arise from using different operating systems.

Apart from **Nix**, we'll need to install one more thing. I don't think we should run Docker in **Nix**, so we'll need it on the host machine. You probably already have it. If you don't, please install it by following the [install instructions](#)¹.

Finally, we'll need `gh` (GitHub CLI) to fork the repository with examples we'll use throughout this book, including the `shell.nix` file that will bring in all the tools we'll need. Please [install it](#)² if you do not have it already.

You can find additional information about GitHub CLI in the [GitHub CLI \(gh\) - How to manage repositories more efficiently](#)³ video.

Finally, each chapter has an associated Gist that contains all the commands we'll execute.

That's it. Now we're ready to set up everything required for me to show you a glimpse of the future.

First, we'll fork the repo,...

¹<https://docs.docker.com/engine/install>

²<https://github.com/cli/cli?tab=readme-ov-file#installation>

³<https://youtu.be/BII6ZY2RnIc>

All the commands user in this chapter are in the [Gist](#)⁴.

```
1 gh repo fork vfaric/crossplane-tutorial --clone --remote
```

...then enter the `crossplane-tutorial` directory, ...

```
1 cd crossplane-tutorial
```

...and select the fork as the default repository.

```
1 gh repo set-default
```

Next, we are going to start a Nix Shell with all the tools we'll need in this chapter.

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

Executing `nix-shell` with all the required packages might take a while since there are quite a few CLIs we'll need install. It takes longer the first time. The packages it's downloading are cached so subsequent executions will be almost instant. Be patient. It will be worth it. Once it's done, we'll get a Shell with everything we'll need.

As an alternative, you can skip running `nix-shell` but, in that case, you'll need to ensure that all the requirements are fulfilled. You'll see what those are in a moment when we execute the setup script.

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

```
1 chmod +x setup/00-intro.sh
```

...and execute it.

```
1 ./setup/00-intro.sh
```

Confirm that you are ready to start unless you prefer to inspect the script and set up everything manually.

Next, double-check that you meet all the requirements. If you are running the script from inside a Nix Shell, as I recommended, you do meet all the requirements, except for Docker. Nix took care of that. So, if you do meet all the requirements, just say... Yes!

The script starts by creating a `kind` cluster with NGINX Ingress. Once it's up and running, it installs Crossplane and a few Configurations. Don't worry if you don't know what they are. We'll dive

⁴<https://gist.github.com/vfaric/732bf76feb51489add89567433019460>

into configurations and everything else related to Crossplane and the ecosystem later. Right now, we're just fast-forwarding through all that so that you can see some of the features we'll be learning throughout this book.

Now, those configurations will, ultimately, create hundreds of custom resource definitions. That takes time so expect to wait for a while until all the providers are up and running.

Once all the providers are ready, we're presented with a choice of a hyperscaler.

I'll choose **AWS** for this chapter, and you can choose any of the "big three". To keep things interesting, and for me not to be labeled as biased, I'll pick a different one in each chapter so the outputs you'll see in the book might differ from those you'll see on your terminal. The logic for all the providers is the same so that should not pose a problem, as long as you're aware that what you see in this book might not be exactly the same as what you'll see in your terminal, unless you happen to pick the same hyperscaler which, as I mentioned, is AWS in this chapter.

I assumed that the downside of having a discrepancy between what you see in this book and what you'll experience by, potentially, choosing a different hyperscaler, is justified by you being able to pick a hyperscaler that best suits your needs.

Once the hyperscaler is selected, the script sets up everything required for Crossplane to work with it.

Finally, the script installed and configured Argo CD.

Throughout the execution of the script, a few environment variables were placed into the `.env` file. We'll need those, so let's source it.

```
1 source .env
```

That's it. Now we're ready to take a look at some of the features we might expect to learn through this book.

A Glimpse Into the Future

Here's what the future looks like.

```
1 cat examples/$HYPERSCALER-intro.yaml
```

The output is as follows.


```
1  ---
2  apiVersion: devopstoolkitseries.com/v1alpha1
3  kind: ClusterClaim
4  metadata:
5    name: cluster-01
6  spec:
7    id: cluster-01
8    compositionSelector:
9      matchLabels:
10        provider: aws-official
11        cluster: eks
12    parameters:
13      nodeSize: small
14      minNodeCount: 3
15  ---
16  apiVersion: v1
17  kind: Secret
18  metadata:
19    name: silly-demo-db-password
20  data:
21    password: cG9zdGdyZXM=
22  ---
23  apiVersion: devopstoolkitseries.com/v1alpha1
24  kind: SQLClaim
25  metadata:
26    name: silly-demo-db
27  spec:
28    id: silly-demo-db
29    compositionSelector:
30      matchLabels:
31        provider: aws-official
32        db: postgresql
33    parameters:
34      version: "13"
35      size: small
36  ---
37  apiVersion: devopstoolkitseries.com/v1alpha1
38  kind: AppClaim
39  metadata:
40    name: silly-demo
41  spec:
42    id: silly-demo
43    compositionSelector:
```

```
44     matchLabels:
45       type: backend-db
46       location: remote
47   parameters:
48     namespace: production
49     image: c8n.io/vfarcic/silly-demo:1.4.52
50     port: 8080
51     host: silly-demo.acme.com
52     dbSecret:
53       name: silly-demo-db
54       namespace: a-team
55     kubernetesProviderConfigName: cluster-01
```

There are around fifty lines of YAML. That might look intimidating but, once you see what that YAML creates, you'll realize that it is infinitely simpler than if we tried to accomplish the same result any other way.

It starts with a `ClusterClaim` that will create a Kubernetes cluster in your favorite hyperscaler. Which hyperscaler it will be depends on the `matchLabels`. In my case, it will be an AWS EKS cluster. Then there are a few parameters that specify that the nodes of the cluster should be `small` and that the minimum number of nodes (`minNodeCount`) should be 3. I don't have to worry about exact node sizes. Crossplane will translate `small` to whatever that means in AWS.

*Please note that if you're using **Google Cloud**, `minNodeCount` is set to 1 and not 3 because it will be a regional cluster running in three zones, and GKE will create 1 in each, so there will be 3 in total.*

To make things more interesting, `ClusterClaim` will not only create a managed Kubernetes cluster, but also make it production-ready by setting up Cilium, installing Crossplane, creating a few Namespaces, and quite a few other things.

Further on we have a `Secret` that contains the encoded password for the database. We will improve on that later when we integrate Crossplane with solutions for managing secrets.

The third definition is `SQLClaim` which will create a managed PostgreSQL database in the hyperscaler of choice. But that's not all. Not only that a database server will be created, but a new user and a database will be created in that server as well.

Finally, there is `AppClaim` which will run a backend application in the yet-to-be-created cluster defined through `ClusterClaim`. That backend app will be automatically connected to the database defined through the `SQLClaim`.

In other words, those fifty or so lines of YAML will create a production-ready managed Kubernetes cluster, a managed PostgreSQL database, and a backend application. Moreover, those three will be interconnected. The application will be running in that cluster and will be connected to the database.

There are probably a few other things that will happen and we'll comment on them later.

Now, as you'll see soon, creating all that often requires experience in AWS, Google, Azure, Kubernetes, databases, and quite a few other technologies. But, as a user of Crossplane Compositions,

we don't need to worry about any of that. Right now, we are consumers of a service created by someone else.

Now, you might expect me to execute a command that will make all that happen, but that's not what we'll do. Instead, we'll treat it as the desired state and push it to Git where it belongs.

So, we'll copy the manifest to the a-team directory in the repo which happens to be monitored by Argo CD,...

```
1 cp examples/$HYPERSCALER-intro.yaml a-team/intro.yaml
```

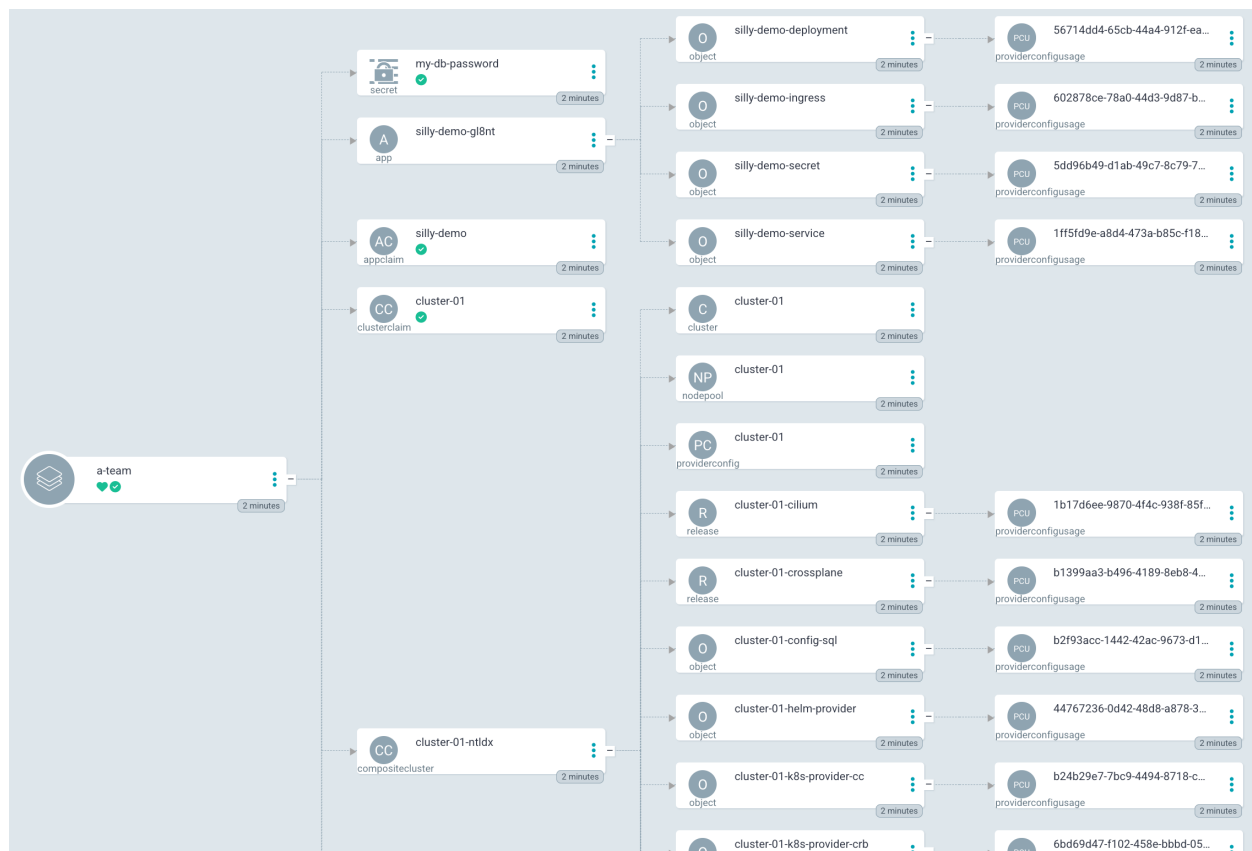
...and add, commit, and push changes to the Git repo.

```
1 git add .
2
3 git commit -m "Intro"
4
5 git push
```

That's it. That's the only action I'll do in this chapter. That's all it takes.

Now, please open <http://argocd.127.0.0.1.nip.io> in a browser. Use admin as the username and admin123 as the password, and sign in.

There is only one application managed by Argo CD. That's the one that monitors the a-team directory. To see the "real" action, I'll enter into the application and... Lo and behold! Those fifty or so lines of YAML expanded into quite a few resources.



We can also observe what's going on by interacting directly with the control plane cluster through `kubectl` by retrieving `clusterclaims`, `sqlclaims`, `appclaims`.

```
1 kubectl --namespace a-team get clusterclaims,sqlclaims,appclaims
```

The output is as follows (truncated for brevity).

```
1 NAME          CLUSTERNAME CONTROLPLANE  NODEPOOL          SYNCED READY ...
2 .../cluster-01 cluster-01  ReconcileError ReconcileError True   False ...
3
4 NAME          SYNCED READY CONNECTION-SECRET AGE
5 .../silly-demo-db True   False              7s
6
7 NAME          HOST          SYNCED READY CONNECTION-SECRET AGE
8 .../silly-demo silly-demo.acme.com True   False              7s
```

We can see that the `clusterclaim`, `sqlclaim`, and `appclaim`, the resources we defined in YAML, are there. None of them are `READY`. It'll take a bit of time until the Kubernetes cluster, the database, the application, and everything in between are created and fully operational.

Now, those three resources “expanded” into quite a few lower-level resources which we can observe by retrieving all managed resources.

```
1 kubectl get managed
```

The output is as follows (truncated for brevity).

```
1 NAME                                READY SYNCED EXTERNAL-NAME AGE
2 internetgateway.../cluster-01      False True                                18s
3 internetgateway.../silly-demo-db    False                                18s
4
5 NAME                                READY SYNCED EXTERNAL-NAME AGE
6 mainroutetableassociation.../cluster-01      False                                18s
7 mainroutetableassociation.../silly-demo-db    False                                18s
8
9 NAME                                READY SYNCED EXTERNAL-NAME AGE
10 route.../cluster-01                  False                                18s
11 route.../silly-demo-db                False                                18s
12
13 NAME                                READY SYNCED EXTERNAL-NAME AGE
14 routetableassociation.../cluster-01-1a        False                                18s
15 routetableassociation.../cluster-01-1b        False                                18s
16 routetableassociation.../cluster-01-1c        False                                18s
17 routetableassociation.../silly-demo-db-1a      False                                18s
18 routetableassociation.../silly-demo-db-1b      False                                18s
19 routetableassociation.../silly-demo-db-1c      False                                18s
20
21 NAME                                READY SYNCED EXTERNAL-NAME AGE
22 routetable.../cluster-01               False True                                18s
23 routetable.../silly-demo-db            False                                18s
24 ...
```

That output shows **AWS** resources (and a few others). If you chose **Google Cloud** or **Azure**, your output will differ.

In my case, I got AWS internet gateways, route table associations, main route table associations, routes, route table associations, route tables, security group rules, security groups, subnets, VPCs, an addon, a cluster auth, a cluster, a node group, role policy attachments, roles, an RDS instance, and a subnet group. Those are the AWS resources that Crossplane is managing. Further on, there are some Helm releases and Kubernetes objects that will be created in the control plane and the new cluster. Finally, there is a database that will be created in the managed database server.

If you're using **Google Cloud** or **Azure**, the number of managed resources will be smaller. Nevertheless, no matter what you chose, it's clear that those fifty or so lines of YAML are very simple to define and understand when compared to everything they produce.

An easier way to explore resources managed by Crossplane is through the crossplane CLI. Among other features, it contains the `trace` command which is, at the time of this writing, a beta release. By

the time you read this, it might become GA so you might need to remove beta from the command that follows.

```
1 crossplane beta trace clusterclaim cluster-01 --namespace a-team
```

The output is as follows (truncated for brevity).

```
1 NAME                               SYNCED READY STATUS
2 ClusterClaim/cluster-01 (a-team)   True   False Waiting: ...
3 └─ CompositeCluster/cluster-01-lkm96 True   False Creating...
4   │─ InternetGateway/cluster-01     True   True   Available
5   │─ MainRouteTableAssociation/cluster-01 True   True   Available
6   │─ RouteTableAssociation/cluster-01-1a True   False Creating
7   │─ RouteTableAssociation/cluster-01-1b False  -     ReconcileError...
8   │─ RouteTableAssociation/cluster-01-1c True   False Creating
9   │─ RouteTable/cluster-01          True   True   Available
10  │─ Route/cluster-01                False  -     ReconcileError...
11  ...
```

That output shows a tree-like structure of all the resources managed by the `ClusterClaim` resource (one of the resources we defined in YAML).

We can see that `ClusterClaim` claim created the `CompositeCluster` composite resource which, in turn, created a bunch of managed resources. We'll go through claims, Composite Resources, and Managed Resources later. What matters, for now, is that we can see what are all the resources that are managed by the `ClusterClaim` we defined earlier.

Another important note is that some of those resources are synced while others are not. Some are ready, and others are not. Unlike some other tools, resources managed by Crossplane are eventually consistent. Some cannot be synchronized because they miss some information from other resources and some are not ready because their preconditions were not met or they are in the process of being created.

As you'll see later, everything will be eventually consistent.

We can observe a similar output if we trace the `SQLClaim`, the second Crossplane resource we defined.

```
1 crossplane beta trace sqlclaim silly-demo-db --namespace a-team
```

The output is as follows (truncated for brevity).

1	NAME	SYNCED	READY	STATUS
2	SQLClaim/silly-demo-db (a-team)	True	False	Waiting:...
3	└ SQL/silly-demo-db-lp9xm	True	False	Creating...
4	└ VPC/silly-demo-db	True	True	Available
5	└ Subnet/silly-demo-db-a	True	False	Creating
6	└ Subnet/silly-demo-db-b	True	True	Available
7	└ Subnet/silly-demo-db-c	True	False	Creating
8	└ SubnetGroup/silly-demo-db	False	-	ReconcileError...
9	└ InternetGateway/silly-demo-db	True	True	Available
10	└ RouteTable/silly-demo-db	True	True	Available
11	└ Route/silly-demo-db	False	-	ReconcileError...
12	└ MainRouteTableAssociation/silly-demo-db	False	-	ReconcileError...
13	└ RouteTableAssociation/silly-demo-db-1a	False	-	ReconcileError...
14	└ RouteTableAssociation/silly-demo-db-1b	False	-	ReconcileError...
15	└ RouteTableAssociation/silly-demo-db-1c	False	-	ReconcileError...
16	└ SecurityGroup/silly-demo-db	True	True	Available
17	└ SecurityGroupRule/silly-demo-db	False	-	ReconcileError...
18	└ Instance/silly-demo-db	False	-	ReconcileError...
19	└ ProviderConfig/silly-demo-db	-	-	
20	└ Database/silly-demo-db	False	-	ReconcileError...
21	└ ProviderConfig/silly-demo-db-sql	-	-	

The database server and everything else required to run the database are also not yet ready. It will be ready soon.

Finally, we can trace the application itself defined as the AppClaim.

```
1 crossplane beta trace appclaim silly-demo --namespace a-team
```

The output is as follows (truncated for brevity).

1	NAME	SYNCED	READY	STATUS
2	AppClaim/silly-demo (a-team)	True	False	Waiting...
3	└ App/silly-demo-9pfkj	True	False	Creating...
4	└ Object/silly-demo-deployment	False	-	ReconcileError...
5	└ Object/silly-demo-service	False	-	ReconcileError...
6	└ Object/silly-demo-ingress	False	-	ReconcileError...
7	└ Object/silly-demo-secret	False	-	ReconcileError...

It's "normal" that the application is not yet running. It should run on the new cluster and until the cluster is created, there is no place for it to run. As with the other two, all I can say is that it will become eventually consistent.

Here's what you should do next.

Take a short break. Get a coffee.

Once you're back, we can trace the SQLClaim again.

```
1 crossplane beta trace sqlclaim silly-demo-db --namespace a-team
```

The output is as follows.

```

1 NAME                               SYNCED READY STATUS
2 SQLClaim/silly-demo-db (a-team)    True   True  Available
3 └─ SQL/silly-demo-db-lp9xm         True   True  Available
4   │ VPC/silly-demo-db               True   True  Available
5   │ Subnet/silly-demo-db-a          True   True  Available
6   │ Subnet/silly-demo-db-b          True   True  Available
7   │ Subnet/silly-demo-db-c          True   True  Available
8   │ SubnetGroup/silly-demo-db       True   True  Available
9   │ InternetGateway/silly-demo-db   True   True  Available
10  │ RouteTable/silly-demo-db         True   True  Available
11  │ Route/silly-demo-db              True   True  Available
12  │ MainRouteTableAssociation/silly-demo-db True   True  Available
13  │ RouteTableAssociation/silly-demo-db-1a True   True  Available
14  │ RouteTableAssociation/silly-demo-db-1b True   True  Available
15  │ RouteTableAssociation/silly-demo-db-1c True   True  Available
16  │ SecurityGroup/silly-demo-db      True   True  Available
17  │ SecurityGroupRule/silly-demo-db  True   True  Available
18  │ Instance/silly-demo-db           True   True  Available
19  │ ProviderConfig/silly-demo-db      -      -
20  │ Database/silly-demo-db           True   True  Available
21  │ ProviderConfig/silly-demo-db-sql  -      -
22  └─ Object/silly-demo-db            True   True  Available

```

This time, all the resources required to run PostgreSQL are Available. The managed database is up and running.

Let's see what's going on with the cluster.

```
1 crossplane beta trace clusterclaim cluster-01 --namespace a-team
```

The output is as follows (truncated for brevity).


```

1  NAME                               SYNCED  READY  STATUS
2  ClusterClaim/cluster-01 (a-team)   True    True   Available
3  └─ CompositeCluster/cluster-01-lkm96 True    True   Available
4    └─ InternetGateway/cluster-01    True    True   Available
5      └─ MainRouteTableAssociation/cluster-01 True    True   Available
6        └─ RouteTableAssociation/cluster-01-1a True    True   Available
7          └─ RouteTableAssociation/cluster-01-1b True    True   Available
8            └─ RouteTableAssociation/cluster-01-1c True    True   Available
9              └─ RouteTable/cluster-01 True     True   Available
10                └─ Route/cluster-01 True     True   Available
11  ...

```

All the resources required to run a cluster are up and running as well. That probably means that Crossplane could apply resources related to the backend application. It should run inside that cluster and it should be connected to the database. Since both of those are fully operational now, the application should be running as well.

Let's check it out.

```
1 crossplane beta trace appclaim silly-demo --namespace a-team
```

The output is as follows.

```

1  NAME                               SYNCED  READY  STATUS
2  AppClaim/silly-demo (a-team)       True    True   Available
3  └─ App/silly-demo-9pfkj            True    True   Available
4    └─ Object/silly-demo-deployment True    True   Available
5      └─ Object/silly-demo-service   True    True   Available
6        └─ Object/silly-demo-ingress True    True   Available
7          └─ Object/silly-demo-secret True    True   Available

```

Everything is up and running. Those three resources we defined at the beginning expanded into dozens of other resources. Some of them are resources in a hyperscaler while others are resources in the new Kubernetes cluster, with a few others sprinkled for good taste.

We are almost done. The last thing we'll do is feed my paranoia. I need to confirm that everything is indeed working as expected. We'll do that by entering into the newly created cluster and taking a peek to see whether the backend application is indeed running. To do that, we'll export the KUBECONFIG variable...

```
1 export KUBECONFIG=$PWD/kubeconfig.yaml
```

...and retrieve the Kube config from the hyperscaler of choice.

```

1  # Execute only if using Google Cloud
2  gcloud container clusters get-credentials cluster-01 \
3      --region us-east1 --project $PROJECT_ID
4
5  # Execute only if using AWS
6  aws eks update-kubeconfig --region us-east-1 \
7      --name cluster-01 --kubeconfig $KUBECONFIG
8
9  # Execute only if using Azure
10 az aks get-credentials --resource-group cluster-01 \
11     --name cluster-01 --file $KUBECONFIG

```

This was one of the cases when instructions differed depending on the hyperscaler you chose. You'll find the instructions for your choice in the [Gist](#)⁵.

Now that the cluster config is set, we can check whether it indeed has three nodes I specified.

```
1 kubectl get nodes
```

The output is as follows.

```

1 NAME                                STATUS ROLES  AGE  VERSION
2 ip-10-0-0-241.ec2.internal Ready  <none> 30s  v1.27.7-eks-e71965b
3 ip-10-0-1-82.ec2.internal   Ready  <none> 26s  v1.27.7-eks-e71965b
4 ip-10-0-2-216.ec2.internal   Ready  <none> 32s  v1.27.7-eks-e71965b

```

The nodes are there.

How about the application? It should be running inside the production Namespace.

```
1 kubectl --namespace production get all,ingresses
```

The output is as follows.

⁵<https://gist.github.com/vfarcic/732bf76feb51489add89567433019460>

```

1  NAME                                READY STATUS  RESTARTS AGE
2  pod/silly-demo-998c464fc-8j9jj 1/1   Running 0          17m
3
4  NAME                                TYPE          CLUSTER-IP    EXTERNAL-IP  PORT(S)    AGE
5  service/silly-demo ClusterIP 172.20.219.47 <none>        8080/TCP   17m
6
7  NAME                                READY UP-TO-DATE AVAILABLE AGE
8  deployment.apps/silly-demo 1/1    1            1          17m
9
10 NAME                                DESIRED CURRENT READY AGE
11 replicaset.apps/silly-demo-998c464fc 1        1        1        17m
12
13 NAME                                CLASS  HOSTS                                ADDRESS PORTS AGE
14 ingress.networking.k8s.io/silly-demo <none> silly-demo.acme.com                80    17m

```

The application is indeed up and running and it is connected to the PostgreSQL database.

As I mentioned earlier, this was only a glimpse of some of the features Crossplane offers. There's much more to it and I did not want to prolong this section more than necessary.

The goal was not to teach you how to use Crossplane and all its nitty-gritty details. That's coming next. So far, the objective was for you to see some of the things it can do so that you can decide whether that's something you might be interested in. I hope that the answer is a resounding YES.

Crossplane is **unique**. It's **special**. It changes the way we manage resources of any kind. It enables us to create **control planes**. It enables us to do things that were, in the past, reserved for only a handful of companies.

So... What do you say? Shall we continue?

Destroy Everything

Each section starts with instructions on how to create everything needed for that section and ends with instructions on how to destroy everything we created. That way you can explore each section independently from others. You can take a break without having to run and pay for, all the resources we created. I wanted to make it easy, but also cheap. No one should spend more time and more money than necessary.

In that spirit, here are the instructions on how to destroy everything we created.

```
1  unset KUBECONFIG
2
3  chmod +x destroy/00-intro.sh
4
5  ./destroy/00-intro.sh
6
7  exit
```

Providers and Managed Resources

Now that we have seen some of the things Crossplane can do, we'll go back to the very beginning and explore some of the basics.

Specifically, we'll explore **Crossplane providers and managed resources**. Now, as I already mentioned in the introduction, we won't be talking about theory without touching the keyboard, so I'll keep this introduction short and jump straight into Crossplane providers... right after we set up the environment we'll use in this chapter.

Chapter Setup

You already know the setup drill from the previous section. Run the script!

All the commands used in this chapter are available in the [Gist](#)¹.

```
1 cd crossplane-tutorial
2
3 nix-shell --run $SHELL
4
5 chmod +x setup/01-managed-resources.sh
6
7 ./setup/01-managed-resources.sh
8
9 source .env
```

Finally, we'll install Crossplane itself. In the subsequent chapters, Crossplane installation will be part of setup scripts but, since this is the first time we're doing a "real" hands-on, I thought it would be beneficial to see how it's done.

There's not much to do though.

It's a single `helm` command.

```
1 helm upgrade --install crossplane crossplane \
2   --repo https://charts.crossplane.io/stable \
3   --namespace crossplane-system --create-namespace --wait
```

With that out of the way, we're ready to dive into Crossplane Providers.

¹<https://gist.github.com/vfarcic/aa5ecfa315608d1257ba56df18088f2f>

Crossplane Providers

Providers are a way to **extend Crossplane capabilities through custom resource definitions (CRDs) and controllers**.

A provider is typically associated with a set of APIs. We have, for example, **AWS**, **Google Cloud**, and **Azure** providers. Installing any of them extends Kubernetes API with hundreds of CRDs. Most of the time, each of those corresponds with an API endpoint.

Now, the important note is that providers can be anything. Besides those I mentioned, there is a **Kubernetes** provider, **SQL** provider, **Helm** provider, and many others.

We'll see what providers do soon. For now, let's take a quick look at the [Upbound marketplace](https://marketplace.upbound.io)² which serves as a place where providers are collected and catalogued.

Over there we can search for providers or simply Browse. The latter is probably a good start if we're new to them.













On the left side, we can switch to Configurations or Functions which we'll explore later.

Inside the providers screen, there is a list of all those currently available. Feel free to spend a few moments taking a look at what's available. Once you're done, we'll install specific providers we'll use in this chapter.

²<https://marketplace.upbound.io>

Providers

Providers are Crossplane packages that bundle a set of Managed Resources and controllers to allow Crossplane to provision and manage the respective infrastructure resources.

 provider-aviatrix aviatrix <small>partner</small> Upbound Partner Provider for configuring Aviatrix Secure Multi Cloud functionality	 crossplane-provider-castai crossplane-contrib <small>partner</small> Upbound Partner Provider for configuring CastAI automation & cost optimization functionality.
 provider-aws Upbound Official Upbound's official Crossplane provider to manage Amazon Web Services (AWS) monolith services in Kubernetes.	 provider-aws-accessanalyzer Upbound Official Upbound's official Crossplane provider to manage Amazon Web Services (AWS) accessanalyzer services in Kubernetes.
 provider-aws-account Upbound Official Upbound's official Crossplane provider to manage Amazon Web Services (AWS) account services in Kubernetes.	 provider-aws-acm Upbound Official Upbound's official Crossplane provider to manage Amazon Web Services (AWS) acm services in Kubernetes.
 provider-aws-acmpca Upbound Official Upbound's official Crossplane provider to manage Amazon Web Services (AWS) acmpca services in Kubernetes.	 provider-aws-amp Upbound Official Upbound's official Crossplane provider to manage Amazon Web Services (AWS) amp services in Kubernetes.
 provider-aws-amplify Upbound Official Upbound's official Crossplane provider to manage Amazon Web Services (AWS) amplify services in Kubernetes.	 provider-aws-apigateway Upbound Official Upbound's official Crossplane provider to manage Amazon Web Services (AWS) apigateway services in Kubernetes.
 provider-aws-apigatewayv2 Upbound Official Upbound's official Crossplane provider to manage Amazon Web Services (AWS) apigatewayv2 services in Kubernetes.	 provider-aws-appautoscaling Upbound Official Upbound's official Crossplane provider to manage Amazon Web Services (AWS) appautoscaling services in Kubernetes.

Now that we had a glimpse of the providers, and before we dive into them, let's make a decision on what we'll build in this chapter. Since we're just starting, we'll make something simple. A good candidate for something "simple" is a VM in your favorite hyperscaler.

To create and manage virtual machines, we need to know the API group. We could find it by browsing the marketplace, but that would probably take too much time, so let's search for it instead.

Search for AWS, Azure, or GCP (Google Cloud) depending on which provider you chose. Select the provider that contains family in the name, select Providers, and search for compute if using Azure or GCP or ec2 if using AWS.

In this chapter I'll use Azure, so my examples might be slightly different from yours.

Search Results

Showing 3 matching results for "compute". [Clear Search Results](#)



provider-azure-compute

Upbound Official

Upbound's official Crossplane provider to manage Microsoft Azure compute services in Kubernetes.



provider-gcp-compute

Upbound Official

Upbound's official Crossplane provider to manage Google Cloud Platform (GCP) compute services in Kubernetes.



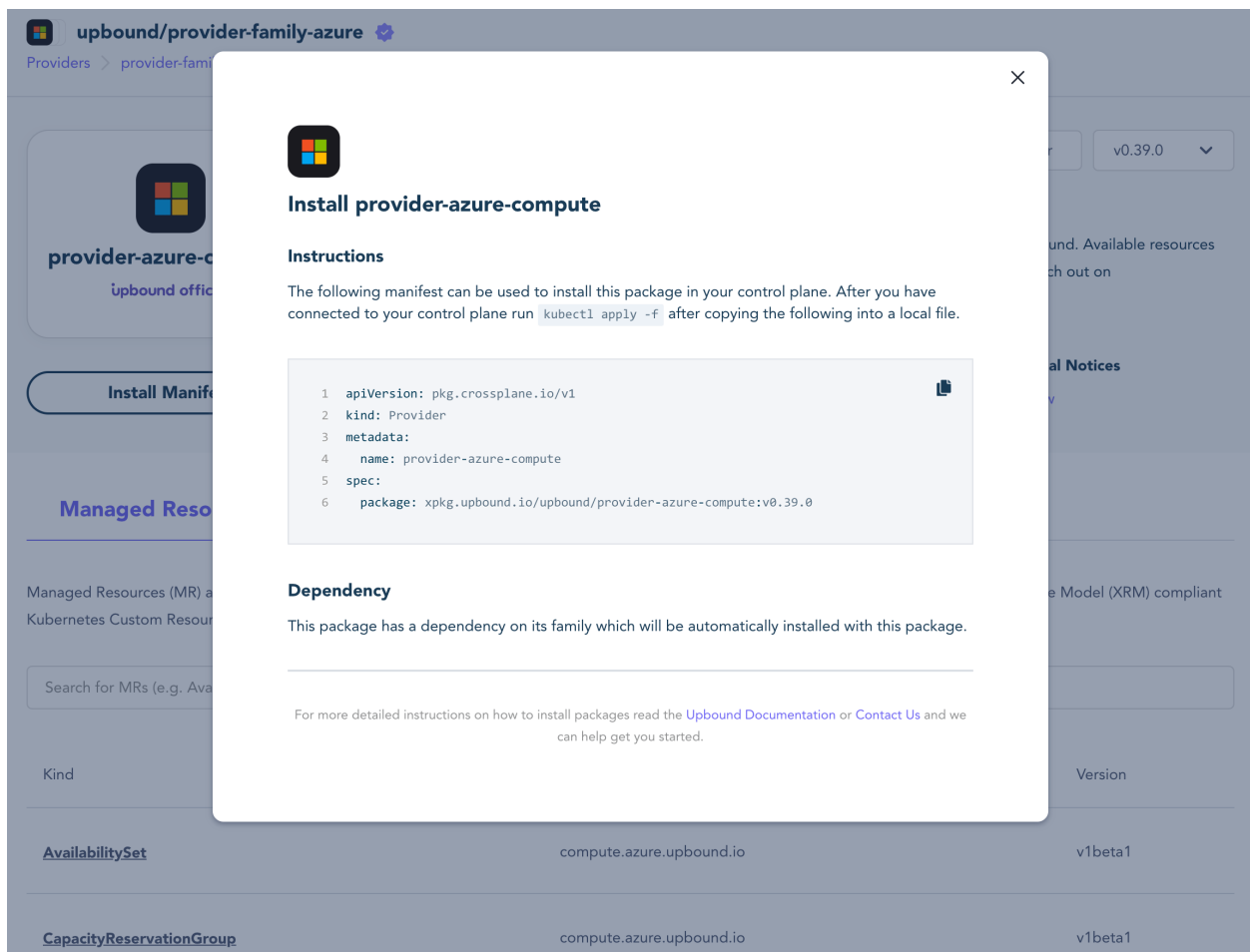
provider-vra

ankasoftco

VMware Aria Automation (vRA) Crossplane provider adds support for managing vRA resources in Kubernetes.



Click on the provider of choice (e.g. `provider-aws-ec2`, `provider-gcp-compute`, or `provider-azure-compute`). Over there, on the page of a specific provider, we can see quite a few things which we'll explore later. For now, what matters is the `Install Manifest` button that gives us instructions on how to define the resource that represents the provider of choice.



We could copy that manifest and paste it into a YAML file, but we won't do that since I already prepared it in advance. Let's take a look at it.

```
1 cat providers/$HYPERSCALER-vm.yaml
```

The output is as follows.

```

1  ---
2  apiVersion: pkg.crossplane.io/v1
3  kind: Provider
4  metadata:
5    name: provider-azure-compute
6  spec:
7    package: xpkg.upbound.io/upbound/provider-azure-compute:v0.39.0
8  ---
9  apiVersion: pkg.crossplane.io/v1
10 kind: Provider

```

```

11 metadata:
12   name: provider-azure-network
13 spec:
14   package: xpkg.upbound.io/upbound/provider-azure-network:v0.39.0

```

In this case, since I’m using Azure in this chapter (and you can be using any of the “big three”), there is a definition of the `provider-azure-compute` that contains the managed resource definitions related to computing in Azure. There is also the `provider-azure-network` provider since we’ll need to define networking for our VM.

If you chose AWS or Google Cloud, you’ll see only one provider.

Let’s install it by executing `kubectl apply...`

```
1 kubectl apply --filename providers/$HYPERSCALER-vm.yaml
```

... and list all available package versions.

```
1 kubectl get pkgrev
```

The output is as follows (truncated for brevity).

1	NAME	HEALTHY	REVISION	IMAGE	...
2	.../provider-azure-compute-...	1		.../provider...	
3	.../provider-azure-network-...	1		.../provider...	

This might be confusing.

We installed a provider, or two, but then we listed something called package versions.

Let me explain...

Packages allow Crossplane to be extended to include new functionality. This typically looks like bundling a set of Kubernetes CRDs and controllers that represent some API endpoints. There are three types of packages; **providers**, **configurations**, and **functions**.

In other words, providers, together with configurations and functions, are a type of package so by listing all package versions we got all packages. If we had configurations or functions, we would see them as well.

Let’s get back to the output of the previous commands.

We can see that the provider(s) we defined were applied, but they did not yet report as `HEALTHY`. That might take a few moments since a provider can, sometimes, contain tens of even hundreds of CRDs.

As a side note, we could have listed only providers, instead of packages that include providers, with `kubectl get providers`. Most of the time, I’m interested in all types of packages and not only providers so we’ll probably use `kubectl get pkgrev` throughout the rest of this book.

Let’s retrieve packages again.

```
1 kubectl get pkgrev
```

The output is as follows (truncated for brevity).

```
1 NAME                                HEALTHY REVISION IMAGE      ...
2 .../provider-azure-compute-...      True    1          .../provider...
3 .../provider-azure-network-...      True    1          .../provider...
4 .../upbound-provider-family-azure-... True    1          .../provider...
```

You'll notice two things. First, after a while, all the providers became HEALTHY. That's good news.

Second, a new provider appeared. In my case, that's `provider-family-azure`.

Let me give you a short background of provider families.

In the beginning, there was a single provider for each Hyperscaler. Since a provider creates a CRD for each API endpoint, and hyperscalers tend to have hundreds of endpoints, installing a provider like, for example, AWS, could end up creating close to a thousand CRDs. If providers for all three hyperscalers are installed, a cluster can easily end up having thousands of CRDs. That can result in performance issues or even cluster crashes on smaller control plane clusters.

Issues with too many CRDs are directly related to Kubernetes itself and the situation is improving with each new Kubernetes release.

Nevertheless, apart from working with the Kubernetes community to resolve those issues, the Crossplane team decided to split big providers into provider families. As a result, instead of having a single provider for AWS, Google Cloud, or Azure, they are split into smaller providers like the one we just defined and applied.

Now, let's get back to the mysterious `family` provider that appeared out of nowhere.

That's the "parent" provider that is installed automatically whenever we apply one of the providers from a family. It contains additional Managed Resource Definitions that are mandatory no matter which of the Providers from a family we install.

I mentioned a few times CRDs and controllers and managed resource definitions and now that we installed a few providers, we can see what those are by listing all CRDs.

```
1 kubectl get crds
```

The output is as follows (truncated for brevity).

```

1 NAME                                CREATED AT
2 ...
3 linuxvirtualmachines...             2023-12-24T23:00:53Z
4 linuxvirtualmachinescalesets...     2023-12-24T23:00:53Z
5 loadbalancerbackendaddresspooladdresses... 2023-12-24T23:01:24Z
6 loadbalancerbackendaddresspools...  2023-12-24T23:01:24Z
7 loadbalancernatpools...             2023-12-24T23:01:24Z
8 ...

```

The output should show tens or even hundreds of CRDs. Each of those represents a hyperscaler resource we can define. For example, since I'm using Azure right now, and I want to create and manage a virtual machine, there is the `linuxvirtualmachines.compute.azure.upbound.io` CRD that contains the extended Kubernetes API endpoint with a schema we can use to define a VM. That's exactly what we'll do soon, right after we finish configuring the providers.

As you can probably imagine, Crossplane cannot manage **AWS**, **Azure**, or **Google Cloud** resources without being able to authenticate to an account. We need to give it credentials with sufficient permissions to manage the resources we're planning to define.

We can provide that through a `ProviderConfig` that will reference a `Secret` with credentials. The setup script we executed earlier already created the credentials file, and we can jump directly into creating the secret.

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

```

1 kubectl --namespace crossplane-system \
2     create secret generic aws-creds \
3     --from-file creds=./aws-creds.conf

```

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

```

1 kubectl --namespace crossplane-system \
2     create secret generic gcp-creds \
3     --from-file creds=./gcp-creds.json

```

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

```

1 kubectl --namespace crossplane-system \
2     create secret generic azure-creds \
3     --from-file creds=./azure-creds.json

```

Next, we need to tell Crossplane where to find the secret we just created. We do that through a `ProviderConfig` associated with the providers we installed.

I prepared that one as well, so let's take a look.

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

The output is as follows.

```
1 ---
2 apiVersion: azure.upbound.io/v1beta1
3 kind: ProviderConfig
4 metadata:
5   name: default
6 spec:
7   credentials:
8     source: Secret
9     secretRef:
10      namespace: crossplane-system
11      name: azure-creds
12      key: creds
```

There's nothing special there apart from the `apiVersion` that is specific to the provider we're running and the `secretRef` that tells it where the secret is.

We're almost done with the providers. All that's left is to apply the `ProviderConfig`.

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

Crossplane is now ready to manage resources in whichever hyperscaler you chose to use and we can jump into the more interesting part of this chapter.

Create Managed Resources

A Crossplane **Managed Resource** represents a resource managed by Crossplane. That resource can be anything. It can be an **AWS EC2 instance**, a **managed PostgreSQL database in Azure**, a **Google Cloud Run instance**, a **Kubernetes object**, a **Helm release**, a **GitHub repository**, or any other type of resource. As long as the Managed Resource Definition exists in the control plane cluster, we can create Managed Resources based on it.

Managed Resource Definitions and their corresponding controllers are installed through providers like the one we applied in the previous section. So, installing a provider results in the installation of a number of Managed Resource Definitions which come with **Kubernetes Custom Definitions and Controllers**.

If we go back to the Marketplace screen, we can see the list of Managed Resources we can create. That way we can deduce whether the provider we're interested in contains the resource definition we're interested in.

Managed Resources (23)

Managed Resources (MR) are Crossplane's representation of a resource in a cloud provider. Managed Resources are opinionated, Crossplane Resource Model (XRM) compliant Kubernetes Custom Resources that are installed by the provider.

Search for MRs (e.g. AvailabilitySet, CapacityReservationGroup)

Kind	Group	Version
AvailabilitySet	compute.azure.upbound.io	v1beta1
CapacityReservationGroup	compute.azure.upbound.io	v1beta1
CapacityReservation	compute.azure.upbound.io	v1beta1
DedicatedHost	compute.azure.upbound.io	v1beta1
DiskAccess	compute.azure.upbound.io	v1beta1
DiskEncryptionSet	compute.azure.upbound.io	v1beta1
GalleryApplication	compute.azure.upbound.io	v1beta1
GalleryApplicationVersion	compute.azure.upbound.io	v1beta1

Please select Instance if you are using AWS or Google Cloud, or LinuxVirtualMachine if you prefer Azure.

Once we select the resource we'd like to manage, we can see the API documentation that contains the full schema with all the fields we might need to manage that resource.

API Documentation	Examples (3)
+ apiVersion	string
+ kind	string
+ metadata	object
- spec	object
LinuxVirtualMachineSpec defines the desired state of LinuxVirtualMachine	
+ deletionPolicy	string
- forProvider	required object
No description provided.	
- additionalCapabilities	array
A additional_capabilities block as defined below.	
+ ultraSsdEnabled	boolean
- adminPasswordSecretRef	object
The Password which should be used for the local-administrator on this Virtual Machine. Changing this forces a new resource to be created.	
+ key	required string

I already prepared an example that we'll use to create and manage a VM in the hyperscaler of choice.

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

The output of the first manifest is as follows (truncated for brevity).

```
1 ---
2 apiVersion: compute.azure.upbound.io/v1beta1
3 kind: LinuxVirtualMachine
4 metadata:
5   name: my-vm
6 spec:
7   forProvider:
8     location: eastus
9     resourceGroupNameRef:
10      name: dot-group
11     size: Standard_A1_v2
12     sourceImageReference:
```

```
13     - offer: UbuntuServer
14     publisher: Canonical
15     sku: 16.04-LTS
16     version: latest
17   adminSshKey:
18     - publicKey: ssh-rsa
19       AAAAB3NzaC1yc2EAAAADAQABAAQAC...
20       you@me.com
21     username: adminuser
22   adminUsername: adminuser
23   osDisk:
24     - caching: ReadWrite
25       storageAccountType: Standard_LRS
26   networkInterfaceIdsRefs:
27     - name: dot-interface
```

As I already mentioned, I’m using **Azure** in this chapter so, depending on what your choice is, you might see a different output. Nevertheless, even though the definitions might differ, the logic behind the explanation that follows is the same.

That is a “standard” Kubernetes manifest with `apiVersion`, `kind`, `metadata`, and `spec`. Assuming that you are familiar with **Kubernetes**, there’s probably no need to explain those. If you are a stranger to Kubernetes, it’s probably too early for you to adopt Crossplane.

The important part is the `spec.forProvider` section. Every **Crossplane Managed Resource** has it. Typically, the fields inside it map the parameters of the resource it manages.

In this specific case, there are fields like `location`, `size`, `adminUsername`, and others that you should be familiar with if you are familiar with Azure. They are almost identical mappings to Azure API for that resource.

There are also “special” fields like `resourceGroupNameRef` and `networkInterfaceIdsRefs`.

Instead of specifying the Resource Group and the network interface, we are letting Crossplane know that it can find the information about those from other resources (from `dot-group` and `dot-interface`). Azure cannot create VMs without the Resource Group and without the network interface. We could have hardcoded that information into the manifest, but that would not be a good idea. It’s much better to let Crossplane figure it out dynamically. Instead of hard-coding information from dependencies, we reference them.

Crossplane Managed Resources do not have a mechanism, like some other tools, to define dependencies. We cannot orchestrate the order in which resources are defined. Instead, Crossplane follows **Kubernetes logic** where everything is eventually consistent. If we decide to apply five resources at once, Crossplane will start creating all five at once, as long as it has all the information it needs. If some information is missing, it will wait until the information is provided.

All that means that the VM manifest requires information about the Resource Group and the network interface and, in this specific case, we are referencing them by name. There are other, potentially better ways to reference resources which we'll explore later.

As a result, Crossplane might not be able to work on the VM if `dot-group` and `dot-interface` are not ready since it cannot get the information it needs. We'll see what that looks like in a moment. For now, let's move on to the other manifests from the output of the previous command.

The rest of the output is as follows.

```
1  ---
2  apiVersion: azure.upbound.io/v1beta1
3  kind: ResourceGroup
4  metadata:
5    name: dot-group
6  spec:
7    forProvider:
8      location: eastus
9  ---
10 apiVersion: network.azure.upbound.io/v1beta1
11 kind: NetworkInterface
12 metadata:
13   name: dot-interface
14 spec:
15   forProvider:
16     ipConfiguration:
17       - name: my-vm
18         privateIpAddressAllocation: Dynamic
19         subnetIdRef:
20           name: dot-subnet
21     location: eastus
22     resourceGroupNameRef:
23       name: dot-group
24   ---
25 apiVersion: network.azure.upbound.io/v1beta1
26 kind: Subnet
27 metadata:
28   name: dot-subnet
29 spec:
30   forProvider:
31     addressPrefixes:
32       - 10.0.1.0/24
33     resourceGroupNameRef:
34       name: dot-group
```

```
35     virtualNetworkNameRef:
36       name: dot-network
37   ---
38   apiVersion: network.azure.upbound.io/v1beta1
39   kind: VirtualNetwork
40   metadata:
41     name: dot-network
42   spec:
43     forProvider:
44       addressSpace:
45         - 10.0.0.0/16
46       location: eastus
47       resourceGroupNameRef:
48         name: dot-group
```

The second manifest defines the Azure ResourceGroup. That is the dot-group resource that the LinuxVirtualMachine is referencing through the `spec.forProvider.resourceGroupNameRef.name` field.

Then there is the NetworkInterface which is the one LinuxVirtualMachine referenced through the `spec.forProvider.networkInterfaceIdsRefs[].name` field. However, NetworkInterface also needs to be inside a Resource Group, so it contains `spec.forProvider.resourceGroupNameRef.name` reference as well. It also requires a subnet so it is referencing it through the `spec.forProvider.ipConfiguration[].subnetIdRef.name`.

Then we have a Subnet manifest referenced by the NetworkInterface which, in turn, references the VirtualNetwork.

Before we proceed, I will say something that might make you think that I'm wasting your time.

You will probably not define **Managed Resources** like that. That would result in a lot of duplication and a lot of confusion by the end users. We'll see a much better way to define Managed Resources when we dive into **Crossplane Compositions**. More importantly, as you will see later, learning how to use Managed Resources will be critical even though you will probably not define them as we're doing it now, so the time learning them is not a waste. Quite the contrary.

With that "depressing" note out of the way, let's apply the manifests we explored...

```
1 kubectl apply --filename examples/\$HYPERSCALER-vm.yaml
```

...and retrieve all managed resources.

```
1 kubectl get managed
```

The output is as follows.

```

1  NAME                                READY SYNCED EXTERNAL-NAME AGE
2  resourcegroup.azure.../dot-group True  True  dot-group      12s
3
4  NAME                                READY SYNCED EXTERNAL-NAME AGE
5  linuxvirtualmachine.compute.azure.../my-vm      False my-vm          12s
6
7  NAME                                READY SYNCED EXTERNAL-NAME AGE
8  networkinterface.network.../dot-interface      False dot-interface 12s
9
10 NAME                                READY SYNCED EXTERNAL-NAME AGE
11 subnet.network.azure.../dot-subnet False True  dot-subnet     12s
12
13 NAME                                READY SYNCED EXTERNAL-NAME AGE
14 virtualnetwork.network.azure.../dot-network      dot-network    12s

```

managed is a shortcut, of sorts, that allows us to retrieve all resources managed by Crossplane. It is, in a way, equivalent to `kubectl get all` which outputs all “core” Kubernetes resources.

Apart from seeing the APIs and the names of the resources we applied, we can see whether they are READY and SYNCED. Suspiciously, in my case, only the `resourcegroup` and the `subnet` are synced. The rest is not, and that brings us back to the references we discussed earlier. `linuxvirtualmachine`, for example, references the `networkinterface`. It needs information from it so until that information is available, Crossplane considers `linuxvirtualmachine` not synced meaning that it cannot start working on it. The same can be said for other resources that are not yet synced. Information from some other referenced resources is missing and that information might be available after Crossplane obtains it from Azure (or whichever hyperscaler you might be using).

The READY field is easier to explain. It indicates whether the actual state, in this case, Azure resource, is ready. It shows whether that specific resource is up and running.

After a while, Crossplane will have all the information it needs to create the VM.

We can see the current state through the `kubectl describe` command or by going to the console of the hyperscaler of choice.

Please open the hyperscaler console and navigate to the EC2 instance if you’re using AWS or the virtual machine if you’re using Azure or Google Cloud.

If you are using AWS, resources are being created in the us-east-1 region, so make sure to have it selected. In the case of Google Cloud, you’ll need to go inside the newly created Project or the Resource Group in the case of Azure.

Once inside the console page of the VM (or AWS EC2), we can see that it was indeed created, or that it is in the process of being created, or that it was not yet created, in which case you might need to wait for a while longer.

The screenshot shows the Azure portal interface for a virtual machine named 'my-vm'. The left sidebar contains navigation options like Overview, Activity log, Access control (IAM), Tags, Diagnose and solve problems, Settings, Networking, Connect, Disks, Size, Microsoft Defender for Cloud, Advisor recommendations, Extensions + applications, Availability + scaling, Configuration, Identity, Properties, Locks, Operations, and Bastion. The main content area is divided into 'Essentials' and 'Properties' sections. The 'Essentials' section shows details like Resource group (dot-group), Status (Creating), Location (East US), Subscription (Pay-As-You-Go), and Subscription ID. The 'Properties' section is further divided into 'Virtual machine' and 'Networking' tabs, showing details like Computer name (my-vm), Operating system (Linux), Image publisher (Canonical), Image offer (UbuntuServer), Image plan (16.04-LTS), VM generation (V1), VM architecture (x64), Public IP address, Private IP address (10.0.1.4), and Virtual network/subnet (dot-network/dot-subnet).

Depending on the hyperscaler you chose, it might take a few minutes until everything is ready. As you already saw, we can check the state of all Managed Resources with `kubectl get managed`.

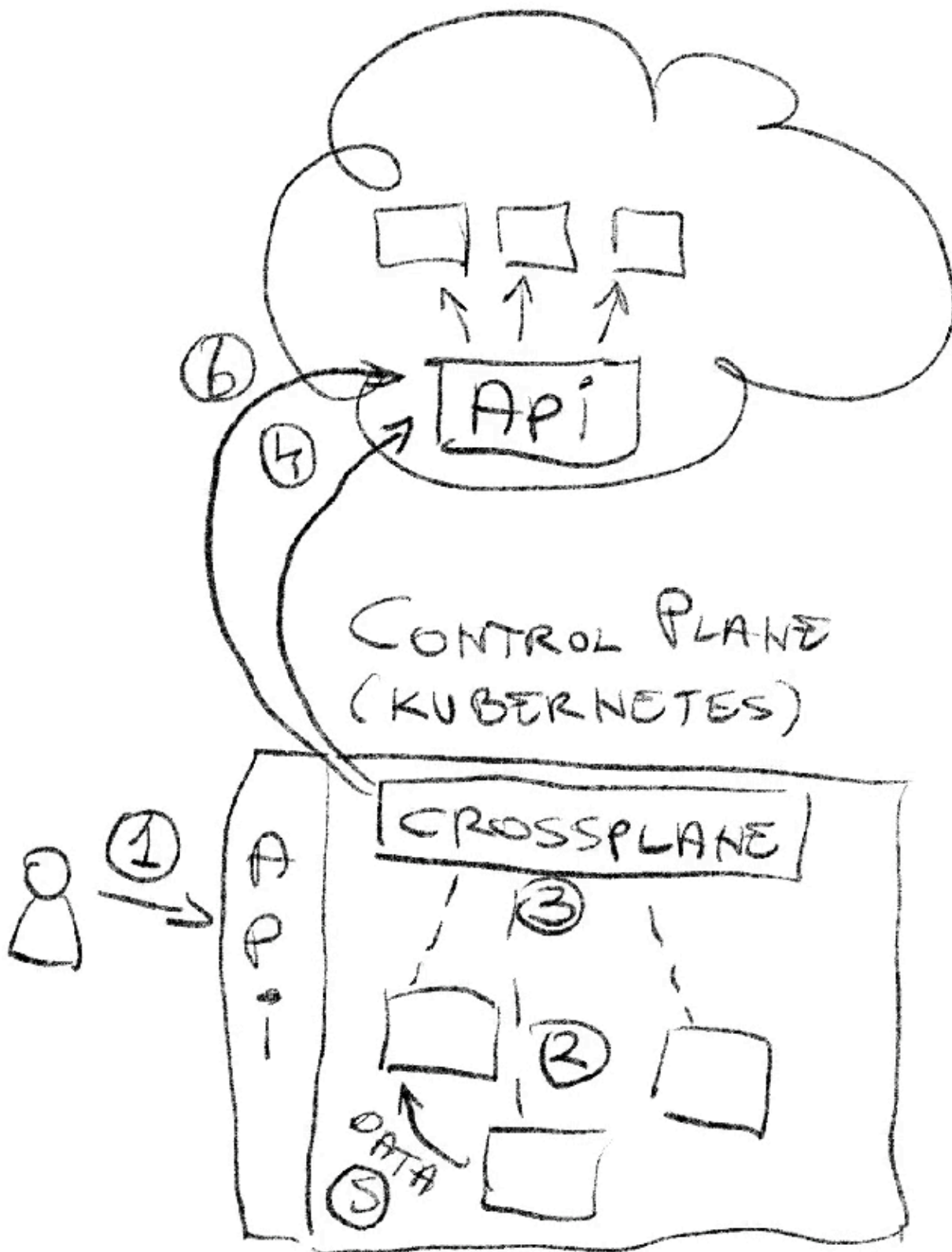
```
1 kubectl get managed
```

The output is as follows.

```
1 NAME                                READY SYNCED EXTERNAL-NAME AGE
2 resourcegroup.azure.../dot-group True  True  dot-group 7m47s
3
4 NAME                                READY SYNCED EXTERNAL-NAME AGE
5 linuxvirtualmachine.compute.azure.../my-vm True  True  my-vm 7m47s
6
7 NAME                                READY SYNCED EXTERNAL-NAME AGE
8 networkinterface.network.azure.../dot-interface True  True  dot-interface 7m47s
9
10 NAME                                READY SYNCED EXTERNAL-NAME AGE
11 subnet.network.azure.../dot-subnet True  True  dot-subnet 7m47s
12
13 NAME                                READY SYNCED EXTERNAL-NAME AGE
14 virtualnetwork.network.azure.../dot-network True  True  dot-network 7m47s
```

Here's what we did so far.

We created a few Custom Resources through Kubernetes API (1, 2). Those Custom Resources are Crossplane Managed Resources associated with a hyperscaler we chose. Crossplane Controllers detected those resources (3) and started talking with the hyperscaler API (4) to create some of those resources, while it was waiting with those that needed data from other resources (5). Once those other resources were created, it could retrieve the data it needs from them and create the rest of the resources (6).



All the resources are fully operational, and we can explore one of the big advantages of Crossplane; continuous drift-detection and reconciliation.

Continuous Drift-Detection and Reconciliation

One of the things we all love about Kubernetes is **continuous drift detection and reconciliation**. If, for example, we create a ReplicaSet (through a Deployment) it creates Pods. But that's only part of the story. That ReplicaSet will continuously watch the Pods it is responsible for and, if the state of those Pods differs from the desired state, it will detect it as a drift and reconcile the states. It will update the Pods to match the desired state. As a result, if we manually change the specification of the Pods, those changes will be undone by the ReplicaSet since there is a drift. If we manually delete one of the Pods, ReplicaSet will create a new one. In that example, the ReplicaSet is ensuring that the actual state of the Pods it is in charge of is always the same as the desired state.

Crossplane takes those concepts to the next level or, to be more precise, it extends them to... **everything**. No matter which type of resources we are managing with Crossplane, it will ensure that their state always matches the desired state.

Let's see if we can prove that.

Please go back to the VM in the console of your hyperscaler of choice. Stop the instance if you are using Google Cloud or AWS or, if you're using Azure, delete the instance.

*Crossplane is limited by the capabilities of the API it talks to. Azure API does not have a mechanism in its API to start a VM that is stopped, so Crossplane cannot do that either. For that reason, in the case of **Azure**, we'll demonstrate drift-detection and reconciliation by deleting it instead.**

We should be able to see that the VM disappeared in the case of Azure or that it was stopped in the case of AWS or Google Cloud.

[Home](#) >

Virtual machines



Default Directory

[+ Create](#) [↔ Switch to classic](#) [🕒 Reservations](#) [⚙️ Manage view](#) [🔄 Refresh](#) ...

Filter for any field...

Subscription equals **all**[+ Add filter](#)[More \(3\)](#)

Showing 0 to 0 of 0 records.

No grouping

List view

Name ↑↓

Type ↑↓

Subscription ↑↓

Resource group ↑↓

Location ↑↓

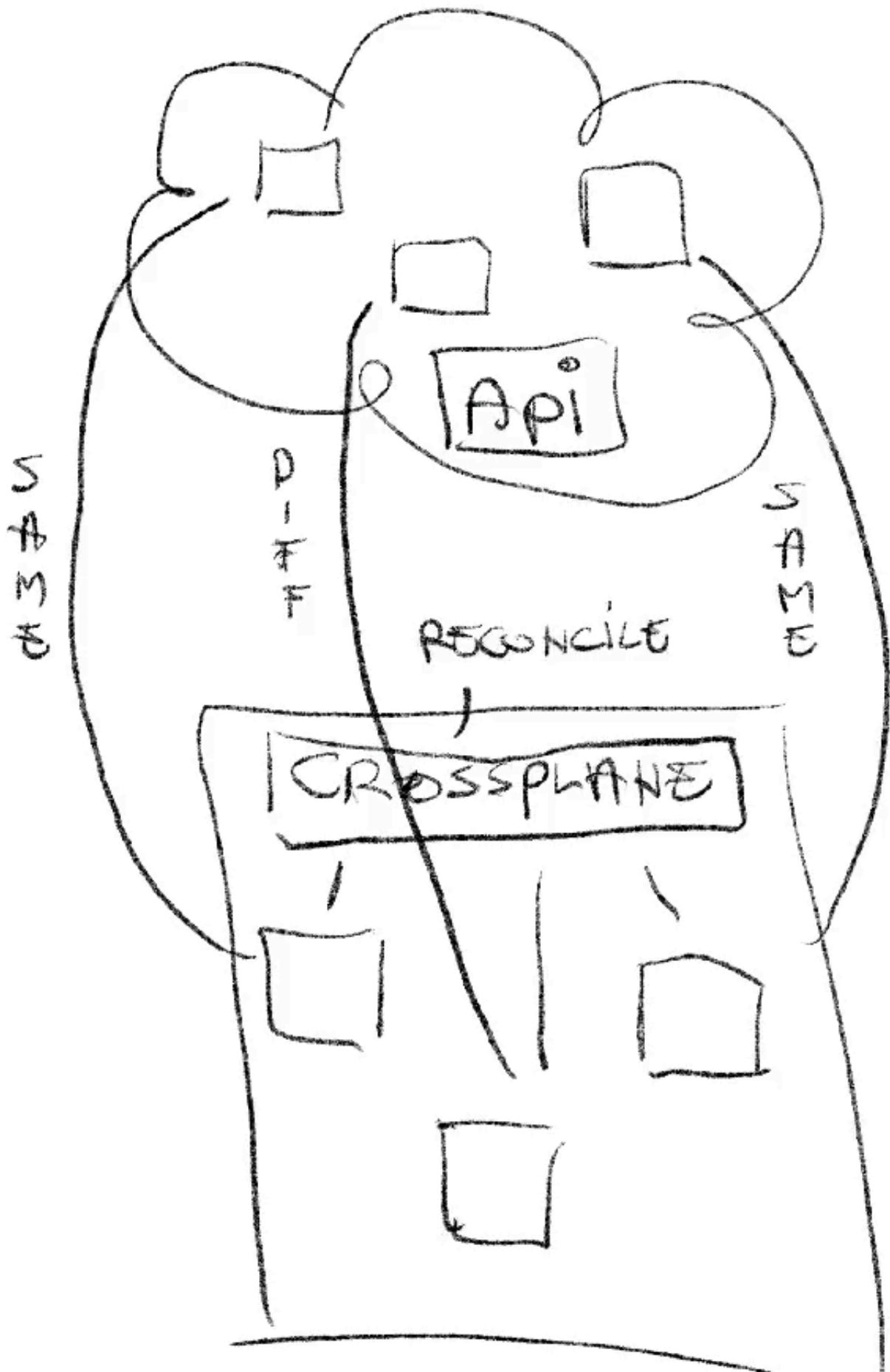


No virtual machines to display

Create a virtual machine that runs Linux or Windows. Select an image from the marketplace or use your own customized image.

[Give feedback](#)[Create](#)

All that's left now is to wait for a few moments so that Crossplane detects the drift and reconciles the differences of the states. A few moments later, we should see that the VM is back in the correct state. It is up and running! Crossplane did the same with the VM as what a ReplicaSet would do with a Pod it manages if we changed its state.



Next, we'll explore how we can update Managed Resources.

Update Managed Resources

Updating Managed resources follows the same drift-detection and reconciliation process we just observed. If we change the desired state by modifying and applying the manifests, Crossplane will detect it as a drift and reconcile it.

Let's take a look at an example by outputting a difference between the manifests we have running in the control plane right now and a modified version.

```
1 diff examples/$HYPERSCALER-vm.yaml \
2     examples/$HYPERSCALER-vm-bigger.yaml
```

The output is as follows.

```
1 <     size: Standard_A1_v2
2 ---
3 >     size: Standard_A2_v2
```

We can see that, in the case of **Azure**, the size of the node changed from `Standard_A1_v2` to `Standard_A2_v2`.

Let's apply the modified manifest,...

```
1 kubectl apply --filename examples/$HYPERSCALER-vm-bigger.yaml
```

...wait for a few moments for Crossplane to detect the drift and reconcile the states, and take another look at the console.

Operating system	: Linux (ubuntu 16.04)
Size	: Standard A2 v2 (1 vcpu, 2 GiB memory)
Public IP address	: -
Virtual network/subnet	: dot-network/dot-subnet
DNS name	: -
Health state	: -

We can see that, in my case, the size of the VM indeed changed to Standard_A2_v2.

Delete Managed Resources

As you can probably guess, the same logic with drift detection and reconciliation is applied if we delete a managed resource.

If, for example, we delete the manifests we applied,...

```
1 kubectl delete --filename examples/$HYPERSCALER-vm-bigger.yaml
```

...wait for a while, and go back to the console, we can see that the VM and all other resources we were managing are now gone.

[Home](#) >

Virtual machines

Default Directory

[+](#) Create [↔](#) Switch to classic [🕒](#) Reservations [⚙️](#) Manage view [🔄](#) Refresh ...

Filter for any field...

Subscription equals all

[+](#) Add filter

[More \(3\)](#)

Showing 0 to 0 of 0 records.

No grouping

List view

Name ↑↓

Type ↑↓

Subscription ↑↓

Resource group ↑↓

Location ↑↓



No virtual machines to display

Create a virtual machine that runs Linux or Windows. Select an image from the marketplace or use your own customized image.

[Give feedback](#)

Create

Crossplane detected the drift between the desired and the actual state and deduced that our desired state is to not have those resources. Hence, Crossplane reconciled the drift by removing them from the hyperscaler.

In some cases, the hyperscaler might choose to spawn a child resource from the resource managed by Crossplane. In those cases, since that resource is not managed by Crossplane, it might be left

“dangling” after we remove the parent resource by deleting the Crossplane Managed Resource. An example of that would be an AWS ELB spun as a result of creating an Ingress controller. It will stay intact even if we remove the Kubernetes cluster through Crossplane since that ELB is not managed by it. In some cases, Hyperscalers have internal mechanisms to clean up orphaned resources, while in others they don’t.

Destroy Everything

That’s it. That’s all you should know about Crossplane Managed Resources, for now.

Let’s destroy everything we did before we jump into the next chapter.

```
1  chmod +x destroy/01-managed-resources.sh
2
3  ./destroy/01-managed-resources.sh
4
5  exit
```

Compositions

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](#)¹.

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

```
1 cd crossplane-tutorial
```

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

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

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

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

...and execute it.

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

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

```
1 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.

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

The output is as follows.

¹<https://gist.github.com/vfarcic/08162d1f3f4954c1f420fae59704b629>

```
1  apiVersion: apiextensions.crossplane.io/v1
2  kind: CompositeResourceDefinition
3  metadata:
4    name: sqls.devopstoolkitseries.com
5  spec:
6    group: devopstoolkitseries.com
7    names:
8      kind: SQL
9      plural: sqls
10   claimNames:
11     kind: SQLClaim
12     plural: sqlclaims
13   versions:
14     - name: v1alpha1
15       served: true
16       referenceable: true
17       schema:
18         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](https://docs.crossplane.io/latest/api)² 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,...

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

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

```
1 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.

```
1 kubectl get xrd
```

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`.

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

The output is as follows.

```
1 sqlclaims.devopstoolkitseries.com 2024-01-03T16:04:16Z
2 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`.

²<https://docs.crossplane.io/latest/api>


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

The output is as follows (truncated for brevity).

```
1 GROUP:      devopstoolkitseries.com
2 KIND:       SQL
3 VERSION:    v1alpha1
4
5 DESCRIPTION:
6   <empty>
7 FIELDS:
8   apiVersion    <string>
9   kind          <string>
10  metadata      <ObjectMeta>
11  ...
12  spec          <Object> -required-
13    claimRef     <Object>
14      apiVersion  <string> -required-
15      kind        <string> -required-
16      name        <string> -required-
17      namespace   <string> -required-
18    compositionRef <Object>
19      name        <string> -required-
20    compositionRevisionRef <Object>
21      name        <string> -required-
22    ...
23  status        <Object>
24  ...
```

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.

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

The output is as follows.

```
1 apiVersion: devopstoolkitseries.com/v1alpha1
2 kind: SQL
3 metadata:
4   name: my-db
5 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,...

```
1 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`.

```
1 kubectl get sqls
```

The output is as follows.

```
1 NAME   SYNCED READY COMPOSITION AGE
2 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.

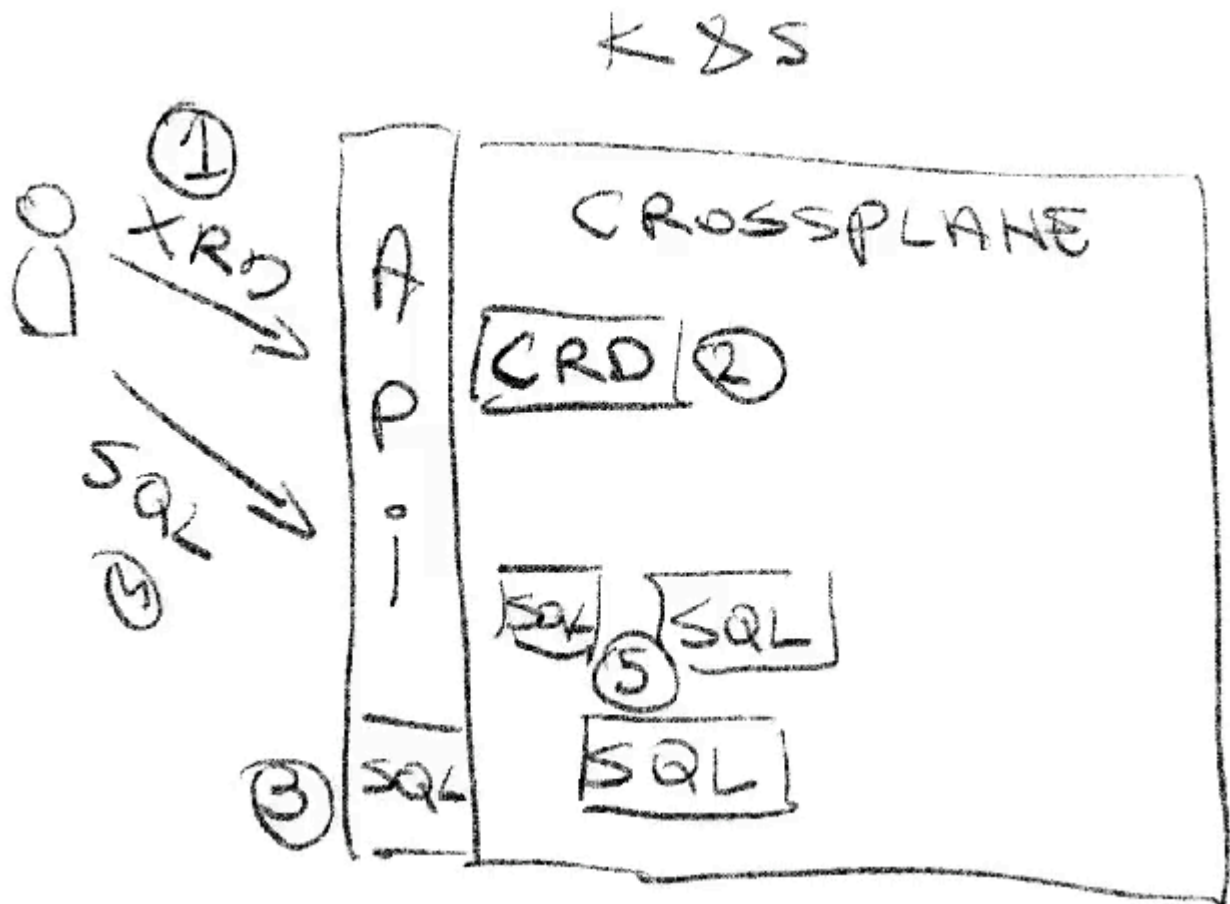
```
1 kubectl get managed
```

The output is as follows.

```
1 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.

```
1 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.

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

The output is as follows.

```
1  ---
2  apiVersion: apiextensions.crossplane.io/v1
3  kind: Composition
4  metadata:
5    name: google-postgresql
6    labels:
7      provider: google
8      db: postgresql
9  spec:
10   compositeTypeRef:
11     apiVersion: devopstoolkitseries.com/v1alpha1
12     kind: SQL
13   resources:
14     - name: sql
15       base:
16         apiVersion: sql.gcp.upbound.io/v1beta1
17         kind: DatabaseInstance
18         spec:
19           forProvider:
20             region: us-east1
21             rootPasswordSecretRef:
22               namespace: crossplane-system
23               key: password
24               name: my-db-password
25             databaseVersion: "POSTGRES_13"
26             settings:
27               - availabilityType: REGIONAL
```

```

28     tier: db-custom-1-3840
29     backupConfiguration:
30     - enabled: true
31       binaryLogEnabled: false
32     ipConfiguration:
33     - ipv4Enabled: true
34       authorizedNetworks:
35     - name: all
36       value: 0.0.0.0/0
37     deletionProtection: false
38 - name: user
39   base:
40     apiVersion: sql.gcp.upbound.io/v1beta1
41     kind: User
42     spec:
43       forProvider:
44         passwordSecretRef:
45         key: password
46         name: my-db-password
47         namespace: crossplane-system
48       instanceSelector:
49       matchLabels:
50       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.

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

The output is as follows.

```
1 aws.yaml
2 azure.yaml
3 definition.yaml
4 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.

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

The output is as follows (truncated for brevity).

```
1 Warning: ... "VPC.ec2.aws.upbound.io" not found
2 Warning: ... "Subnet.ec2.aws.upbound.io" not found
3 Warning: ... "SubnetGroup.rds.aws.upbound.io" not found
4 Warning: ... "InternetGateway.ec2.aws.upbound.io" not found
5 Warning: ... "RouteTable.ec2.aws.upbound.io" not found
6 Warning: ... "Route.ec2.aws.upbound.io" not found
7 Warning: ... "MainRouteTableAssociation.ec2.aws.upbound.io" not found
8 Warning: ... "RouteTableAssociation.ec2.aws.upbound.io" not found
9 Warning: ... "SecurityGroup.ec2.aws.upbound.io" not found
10 Warning: ... "SecurityGroupRule.ec2.aws.upbound.io" not found
11 Warning: ... "Instance.rds.aws.upbound.io" not found
12 composition.apiextensions.../aws-postgresql created
13 Warning: ... "ResourceGroup.azure.upbound.io" not found
14 Warning: ... "Server.dbforpostgresql.azure.upbound.io" not found
15 Warning: ... "FirewallRule.dbforpostgresql.azure.upbound.io" not found
16 composition.apiextensions.../azure-postgresql created
17 compositeresourcedefinition.apiextensions.../sqls.devopstoolkitseries.com unchanged
18 Warning: ... "DatabaseInstance.sql.gcp.upbound.io" not found
19 Warning: ... "User.sql.gcp.upbound.io" not found
20 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.

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

The output is as follows.

```

1  ---
2  apiVersion: pkg.crossplane.io/v1
3  kind: Provider
4  metadata:
5    name: provider-aws-ec2
6  spec:
7    package: xpkg.upbound.io/upbound/provider-aws-ec2:v0.47.1
8  ---
9  apiVersion: pkg.crossplane.io/v1
10 kind: Provider
11 metadata:
12   name: provider-aws-rds
13 spec:
14   package: xpkg.upbound.io/upbound/provider-aws-rds:v0.47.1
15 ---
16 apiVersion: pkg.crossplane.io/v1
17 kind: Provider
18 metadata:
19   name: provider-gcp-sql
20 spec:
21   package: xpkg.upbound.io/upbound/provider-gcp-sql:v0.41.0
22 ---
23 apiVersion: pkg.crossplane.io/v1
24 kind: Provider
25 metadata:
26   name: provider-azure-dbforpostgresql
27 spec:
28   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,...

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

...and take a look at package revisions.

```
1 kubectl get pkgrev
```

The output is as follows (truncated for brevity).


```

1 NAME                                HEALTHY REVISION IMAGE      ...
2 .../provider-aws-rds-...             1      .../provider...
3 .../provider-azure-dbforpostgresql-... False   1      .../provider...
4 .../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`.

```
1 kubectl get pkgrev
```

The output is as follows (truncated for brevity).

```

1 NAME                                HEALTHY REVISION IMAGE      ...
2 .../provider-aws-ec2-bc4e31f08ec6    True    1      .../provider...
3 .../provider-aws-rds-410139ed4243    True    1      .../provider...
4 .../provider-azure-dbforpostgresql-7905967328cb True    1      .../provider...
5 .../provider-gcp-sql-ac45452bc4d2    True    1      .../provider...
6 .../upbound-provider-family-aws-461aea25f5b4 True    1      .../provider...
7 .../upbound-provider-family-azure-f70e43ba7cb1 True    1      .../provider...
8 .../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.

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

The output is as follows.

```

1 ---
2 apiVersion: gcp.upbound.io/v1beta1
3 kind: ProviderConfig
4 metadata:
5   name: default
6 spec:
7   projectID: dot-20231226202303
8   credentials:
9     source: Secret
10    secretRef:
11      namespace: crossplane-system
12      name: gcp-creds
13      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,...

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

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

```
1 kubectl get compositions
```

```
1 NAME                XR-KIND XR-APIVERSION                AGE
2 aws-postgresql      SQL     devopstoolkitseries.com/v1alpha1 6m55s
3 azure-postgresql    SQL     devopstoolkitseries.com/v1alpha1 6m55s
4 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.

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

The output is as follows.

```
1 ---
2 apiVersion: v1
3 kind: Secret
4 metadata:
5   name: my-db-password
6   namespace: crossplane-system
7 data:
8   password: cG9zdGdyZXM=
9 ---
10 apiVersion: devopstoolkitseries.com/v1alpha1
11 kind: SQL
12 metadata:
13   name: my-db
14 spec:
15   compositionSelector:
16     matchLabels:
17       provider: google
18       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,...

```
1 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.

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

The output is as follows.

```
1 NAME                                SYNCED READY STATUS                                \
2
3 SQL/my-db                           True   False Creating...
4 └─ DatabaseInstance/my-db-schrw True   False Creating                                \
5
6 └─ 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 beta trace sql my-db` command again.

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

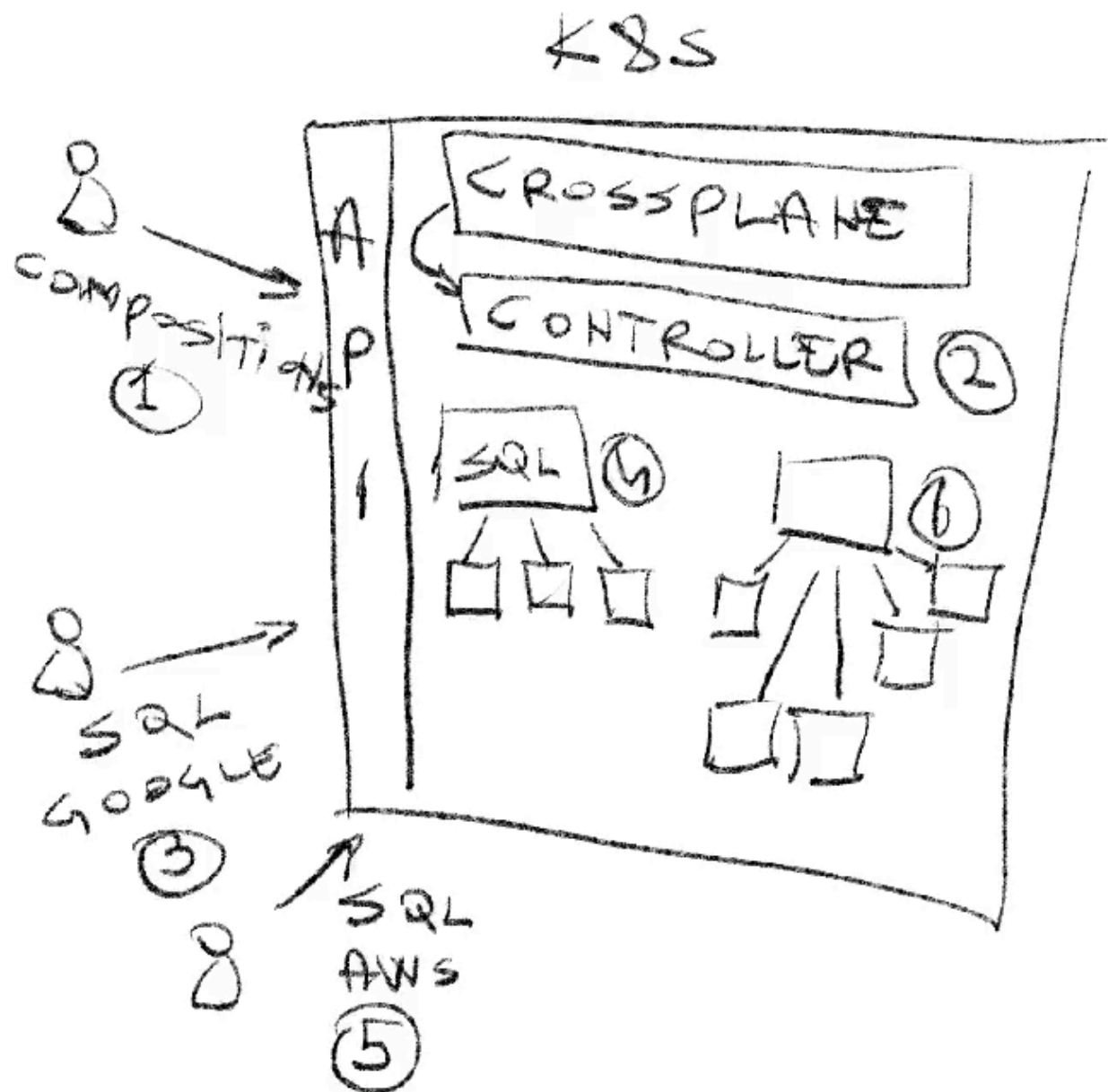
The STATUS of all the resources is now Available.

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.

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

The output is as follows (truncated for brevity).

```
1 ---
2 apiVersion: compute.gcp.upbound.io/v1beta1
3 kind: Instance
4 metadata:
5   name: my-vm
6 spec:
7   forProvider:
8     ...
9   networkInterface:
10     - networkRef:
11       name: dot-network
12     ...
13 ---
14 apiVersion: compute.gcp.upbound.io/v1beta1
15 kind: Network
16 metadata:
17   name: dot-network
18 ...
```

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).

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

The output is as follows (truncated for brevity).

```

1  ---
2  apiVersion: apiextensions.crossplane.io/v1
3  kind: Composition
4  metadata:
5    name: google-postgresql
6    ...
7  spec:
8    ...
9  resources:
10 - name: sql
11   base:
12     apiVersion: sql.gcp.upbound.io/v1beta1
13     kind: DatabaseInstance
14     spec:
15       ...
16 - name: user
17   base:
18     apiVersion: sql.gcp.upbound.io/v1beta1
19     kind: User
20     spec:
21       forProvider:
22         ...
23       instanceSelector:
24         matchLabels:
25           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`.

```

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

```

The output is as follows (truncated for brevity).

```

1  ---
2  apiVersion: apiextensions.crossplane.io/v1
3  kind: Composition
4  metadata:
5    name: google-postgresql
6    ...
7  spec:
8    ..
9    resources:
10   - name: sql
11     base:
12       apiVersion: sql.gcp.upbound.io/v1beta1
13       kind: DatabaseInstance
14       spec:
15         forProvider:
16           ...
17   - name: user
18     base:
19       apiVersion: sql.gcp.upbound.io/v1beta1
20       kind: User
21       spec:
22         forProvider:
23           ...
24         instanceSelector:
25           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**.

```

1  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.

```
1 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.

```
1 kubectl get managed
```

The output is as follows (truncated for brevity).

```
1 NAME                                READY SYNCED EXTERNAL-NAME AGE
2 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.

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

The output is as follows (truncated for brevity).

```

1  ---
2  apiVersion: apiextensions.crossplane.io/v1
3  kind: CompositeResourceDefinition
4  metadata:
5    name: sqls.devopstoolkitseries.com
6  spec:
7    ...
8  versions:
9    - name: v1alpha1
10    ...
11  schema:
12    openAPIV3Schema:
13      type: object
14      properties:
15        spec:
16          type: object
17          properties:
18            id:
19              type: string
20              description: Database ID
21          parameters:
22            type: object
23            properties:
24              version:
25                description: The DB version depends...
26                type: string
27              size:
28                description: "Supported sizes: small, medium, large"
29                type: string
30                default: small
31            required:
32              - version
33          required:
34            - 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.

```
1 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.

```
1 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.

```
1 ---
2 apiVersion: apiextensions.crossplane.io/v1
3 kind: Composition
4 ...
5 spec:
6   ...
7   patchSets:
8     - name: metadata
9       patches:
10        - fromFieldPath: metadata.annotations
11          toFieldPath: metadata.annotations
12        - fromFieldPath: spec.id
13          toFieldPath: metadata.name
14   ...
```

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.

```

1  ---
2  apiVersion: apiextensions.crossplane.io/v1
3  kind: Composition
4  ...
5  spec:
6    ...
7    resources:
8      - name: sql
9        ...
10     patches:
11       - type: PatchSet
12         patchSetName: metadata
13       - fromFieldPath: spec.parameters.version
14         toFieldPath: spec.forProvider.databaseVersion
15         transforms:
16           - type: string
17             string:
18               fmt: POSTGRES_%s
19       - fromFieldPath: spec.parameters.size
20         toFieldPath: spec.forProvider.settings[0].tier
21         transforms:
22           - type: map
23             map:
24               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.

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

The output is as follows.

```
1 ...
2 apiVersion: devopstoolkitseries.com/v1alpha1
3 kind: SQL
4 metadata:
5   name: my-db
6   annotations:
7     organization: DevOps Toolkit
8     author: Viktor Farcic <viktor@farcic.com>
9 spec:
10   id: my-db
11   compositionSelector:
12     matchLabels:
13       provider: google
14       db: postgresql
15   parameters:
16     version: "13"
17     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...

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

...and trace the progress.

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

The output is as follows (truncated for brevity).

```
1 NAME                               SYNCED READY STATUS
2 SQL/my-db                          True   False Creating...
3 └─ DatabaseInstance/my-db True   False Creating
4 └─ 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...

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

...and output that resource as YAML.

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

The output is as follows (truncated for brevity).

```
1 apiVersion: sql.gcp.upbound.io/v1beta1
2 kind: DatabaseInstance
3 metadata:
4   annotations:
5     author: Viktor Farcic <viktor@farcic.com>
6     ...
7   organization: DevOps Toolkit
8   ...
```

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.

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

The output is as follows (truncated for brevity).

```
1 ---
2 apiVersion: apiextensions.crossplane.io/v1
3 kind: Composition
4 ...
5 spec:
6   writeConnectionSecretsToNamespace: crossplane-system
7   ...
8   resources:
9     - name: sql
10       base:
11         apiVersion: sql.gcp.upbound.io/v1beta1
12         kind: DatabaseInstance
13         spec:
14           ...
15           writeConnectionSecretToRef:
16             namespace: crossplane-system
17       patches:
18         ...
19         - fromFieldPath: spec.id
20           toFieldPath: spec.writeConnectionSecretToRef.name
21         ...
```

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...

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

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

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

The output is as follows (truncated for brevity).

```
1 NAME                TYPE                                DATA AGE
2 ...
3 my-db                connection.crossplane.io/v1alpha1 10    13s
4 my-db-password Opaque                             1     8m24s
5 ...
```

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.

```
1 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.

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

The output is as follows (truncated for brevity).

```

1  apiVersion: v1
2  data:
3    attribute.root_password: cG9zdGdyZXM=
4    connectionName: ZG90LTIwMjQwMTAzMTk0MDU2OnVzLWVhc3QxOm15LWRi
5    password: cG9zdGdyZXM=
6    privateIP: ""
7    publicIP: MzUuMTk2LjQ3LjEwNQ==
8    serverCACertificateCert: LS0tLS1CRUd...
9    serverCACertificateCommonName: Qz1VUyxP...=
10   serverCACertificateCreateTime: MjAyNC0wMS0wM1QxOT...
11   serverCACertificateExpirationTime: MjAzMy0xMi0zMV...
12   serverCACertificateSha1Fingerprint: OWMyNzVjNjB...
13 kind: Secret
14 ...

```

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**.*

```

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

```

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

```
1 export PGUSER=postgres
```

Next, we'll retrieve the password.

```
1 export PGPASSWORD=$(kubectl --namespace crossplane-system \  
2     get secret $DB --output jsonpath="{.data.password}" \  
3     | 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**.*

```
1 export HOST_KEY=endpoint
```

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

```
1 export HOST_KEY=host
```

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

```
1 export HOST_KEY=publicIP
```

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

```
1 export PGHOST=$(kubectl --namespace crossplane-system \  
2     get secret $DB --output jsonpath="{.data.$HOST_KEY}" \  
3     | 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.

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

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

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

...and list all the databases.

```
1  \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...

```
1  exit
```

...and out of the container.

```
1  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.

```
1  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).

```

1  ---
2  apiVersion: apiextensions.crossplane.io/v1
3  kind: Composition
4  metadata:
5    name: google-postgresql
6    ...
7  spec:
8    ...
9  resources:
10   ...
11   - name: sql-config
12     base:
13       apiVersion: postgresql.sql.crossplane.io/v1alpha1
14       kind: ProviderConfig
15       metadata:
16         name: default
17       spec:
18         credentials:
19           source: PostgreSQLConnectionSecret
20           connectionSecretRef:
21             namespace: crossplane-system
22           sslMode: require
23       patches:
24         - type: PatchSet
25           patchSetName: metadata
26         - fromFieldPath: spec.id
27           toFieldPath: spec.credentials.connectionSecretRef.name
28     ...

```

We can add a database through the [SQL Provider](#)³. 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*

³<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.

```
1  ---
2  apiVersion: apiextensions.crossplane.io/v1
3  kind: Composition
4  metadata:
5    name: google-postgresql
6    ...
7  spec:
8    ...
9  resources:
10   ...
11   - name: sql-db
12     base:
13       apiVersion: postgresql.sql.crossplane.io/v1alpha1
14       kind: Database
15       spec:
16         forProvider: {}
17     patches:
18     - type: PatchSet
19       patchSetName: metadata
20     - fromFieldPath: spec.id
21       toFieldPath: spec.providerConfigRef.name
22     ...
```

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.

```
1  ---
2  apiVersion: apiextensions.crossplane.io/v1
3  kind: Composition
4  metadata:
5    name: google-postgresql
6    ...
7  spec:
8    ...
9  resources:
10   ...
11   - name: kubernetes
12     base:
13       apiVersion: kubernetes.crossplane.io/v1alpha1
14       kind: ProviderConfig
15       spec:
16         credentials:
17           source: InjectedIdentity
18     patches:
19     - fromFieldPath: metadata.annotations
20       toFieldPath: metadata.annotations
21     - fromFieldPath: spec.id
22       toFieldPath: metadata.name
23     transforms:
24     - type: string
25       string:
26         fmt: "%s-sql"
27     ...
```

We can use object resources from the [Kubernetes provider](https://marketplace.upbound.io/providers/crossplane-contrib/provider-kubernetes)⁴ 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.

⁴<https://marketplace.upbound.io/providers/crossplane-contrib/provider-kubernetes>

```
1  ---
2  apiVersion: apiextensions.crossplane.io/v1
3  kind: Composition
4  metadata:
5    name: google-postgresql
6    ...
7  spec:
8    ...
9  resources:
10   ...
11   - name: sql-secret
12     base:
13       apiVersion: kubernetes.crossplane.io/v1alpha1
14       kind: Object
15       metadata:
16         name: sql-secret
17       spec:
18         forProvider:
19           manifest:
20             apiVersion: v1
21             kind: Secret
22             metadata:
23               namespace: crossplane-system
24             data:
25               port: NTQzMg==
26         references:
27         - patchesFrom:
28             apiVersion: sql.gcp.upbound.io/v1beta1
29             kind: User
30             namespace: crossplane-system
31             fieldPath: metadata.name
32             toFieldPath: stringData.username
33         - patchesFrom:
34             apiVersion: v1
35             kind: Secret
36             namespace: crossplane-system
37             fieldPath: data.password
38             toFieldPath: data.password
39         - patchesFrom:
40             apiVersion: v1
41             kind: Secret
42             namespace: crossplane-system
43             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).

```
1  ...
2  apiVersion: pkg.crossplane.io/v1
3  kind: Provider
4  metadata:
5    name: provider-sql
6  spec:
7    package: crossplane/provider-sql:v0.7.0
8  ---
9  apiVersion: v1
10 kind: ServiceAccount
11 metadata:
12   name: crossplane-provider-kubernetes
13   namespace: crossplane-system
14 ---
15 apiVersion: rbac.authorization.k8s.io/v1
16 kind: ClusterRoleBinding
17 metadata:
18   name: crossplane-provider-kubernetes
19 subjects:
20 - kind: ServiceAccount
21   name: crossplane-provider-kubernetes
22   namespace: crossplane-system
23 roleRef:
24   kind: ClusterRole
25   name: cluster-admin
26   apiGroup: rbac.authorization.k8s.io
27 ---
28 apiVersion: pkg.crossplane.io/v1alpha1
29 kind: ControllerConfig
30 metadata:
31   name: crossplane-provider-kubernetes
32 spec:
33   serviceAccountName: crossplane-provider-kubernetes
34 ---
35 apiVersion: pkg.crossplane.io/v1
36 kind: Provider
37 metadata:
38   name: crossplane-provider-kubernetes
39 spec:
40   package: xpkg.upbound.io/crossplane-contrib/provider-kubernetes:v0.9.0
41   controllerConfigRef:
42     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 `KubernetesProvider` 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,...

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

...output all package revisions,...

```
1 kubectl get pkgrev
```

...and wait until they are all HEALTHY.

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

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

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

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

...enter the client,...

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

...and list all databases.

```
1 \l
```

The output is as follows (truncated for brevity).

```

1  ...
2      Name      |      Owner      | ...
3  -----+-----+...
4  cloudsqladmin | cloudsqladmin   | ...
5  my-db         | my-db           | ...
6  postgres      | cloudsqlsuperuser | ...
7  template0     | cloudsqladmin    | ...
8                |                  | ...
9  template1     | cloudsqlsuperuser | ...
10               |                  | ...
11 (5 rows)

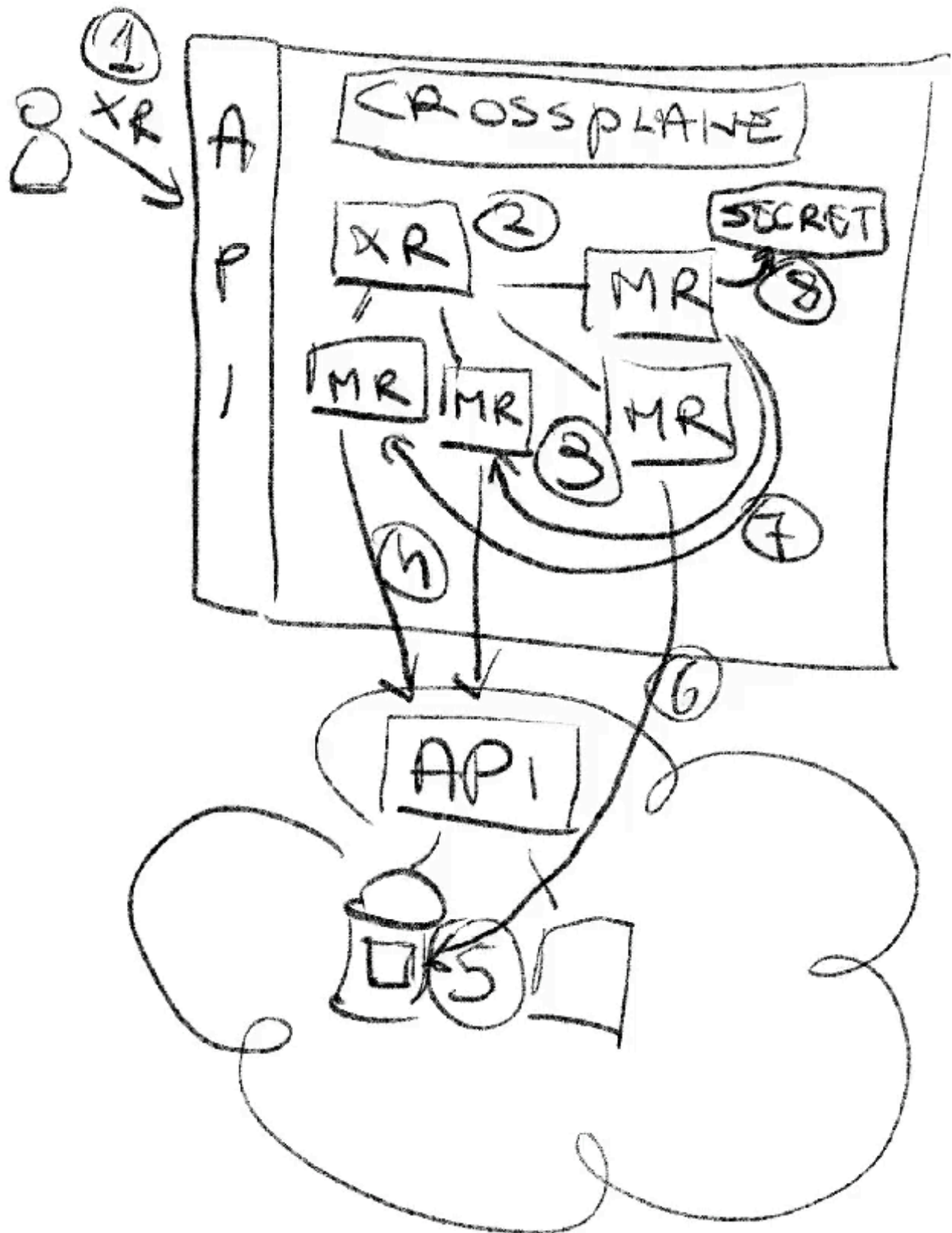
```

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



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,...

```
1 exit
```

...and the container,...

```
1 exit
```

...delete the Composite Resource,...

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

...and list all Managed Resources.

```
1 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.

```
1 kubectl patch database.postgresql.sql.crossplane.io $DB \  
2 --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.

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

The output is as follows (truncated for brevity).

```
1 ---
2 apiVersion: apiextensions.crossplane.io/v1
3 kind: CompositeResourceDefinition
4 metadata:
5   name: sqls.devopstoolkitseries.com
6 spec:
7   ...
8   names:
9     kind: SQL
10    plural: sqls
11    claimNames:
12      kind: SQLClaim
13      plural: sqlclaims
14    ...
```

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.

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

The output is as follows (truncated for brevity).

```

1  ---
2  apiVersion: apiextensions.crossplane.io/v1
3  kind: Composition
4  metadata:
5    name: google-postgresql
6    ...
7  spec:
8    ...
9  resources:
10 - name: sql
11   ...
12   patches:
13   ...
14   - fromFieldPath: spec.claimRef.namespace
15     toFieldPath: spec.forProvider.rootPasswordSecretRef.namespace
16 - name: user
17   ...
18   patches:
19   ...
20   - fromFieldPath: spec.claimRef.namespace
21     toFieldPath: spec.forProvider.passwordSecretRef.namespace
22 - name: sql-config
23   ...
24   patches:
25   ...
26   - fromFieldPath: spec.claimRef.namespace
27     toFieldPath: spec.credentials.connectionSecretRef.namespace
28   ...
29 - name: sql-secret
30   ...
31   patches:
32   ...
33   - fromFieldPath: spec.claimRef.namespace
34     toFieldPath: spec.references[1].patchesFrom.namespace
35   ...
36   - fromFieldPath: spec.claimRef.namespace
37     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.

```
1 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.

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

The output is as follows (truncated for brevity).

```
1 apiVersion: devopstoolkitseries.com/v1alpha1
2 kind: SQLClaim
3 ...
```

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.

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

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

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

The output is as follows.

```

1 NAME SYNCED READY CONNECTION-SECRET AGE
2 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.

```

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

```

The output is as follows (truncated for brevity).

```

1 NAME SYNCED READY STATUS
2 SQLClaim/my-db (a-team) True False Waiting...
3 └─ SQL/my-db-tzvhj True False Creating...
4   └─ DatabaseInstance/my-db True False Creating
5   └─ User/my-db False False ReconcileError:...
6   └─ ProviderConfig/my-db - -
7   └─ Database/my-db False - ReconcileError:...
8   └─ ProviderConfig/my-db-sql - -
9   └─ 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.

```

1 kubectl --namespace a-team get secrets

```

The output is as follows.

	NAME	TYPE	DATA	AGE
2	my-db	Opaque	4	2m18s
3	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.

```
1 chmod +x destroy/02-compositions.sh
2
3 ./destroy/02-compositions.sh
4
5 exit
```

Configuration Packages

In the previous chapter, we built **Compositions** that encapsulate managed **PostgreSQL** databases in **AWS**, **Azure**, and **Google Cloud**. Those enabled us to provide a simple service that allows anyone to create a database and everything required for it to run successfully, while, at the same time, converting the complexity into an implementation detail.

We still have a **distribution problem** though. We would need to instruct people managing the control plane to apply the **Providers**, **Composite Resource Definitions**, and **Compositions**. We would need to distribute all those manifests we wrote. While that is not necessarily a bad idea, especially if we are using GitOps tools like **Argo CD** and **Flux**, there might be a better way to distribute all that.

We can build OCI (what you might call Docker) images that package everything Crossplane needs to run a set of Compositions. We call them **Configuration Packages** which, just like **Providers** and, as you will discover later, **Functions**, are all variations of **Crossplane Packages**. Those are the ones we saw when we were retrieving Package Revisions with the `kubectl get pkgrev` command.

Hence, this chapter is dedicated to **Crossplane Configuration Packages** which provide a mechanism to distribute Composite Resource Definitions, Compositions, and Providers they depend on.

Let's start by setting up everything we'll need for the hands-on part of this chapter.

Chapter Setup

You already know what to do.

All the commands user in this chapter are in the [Gist](https://gist.github.com/vfarcic/3f4f9bf05c937b9f12e6bcb43f3c0bc7)¹.

We'll enter the directory with the `crossplane-tutorial` fork unless you're already there,...

```
1 cd crossplane-tutorial
```

...run Nix Shell,...

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

...make the setup script executable,...

¹<https://gist.github.com/vfarcic/3f4f9bf05c937b9f12e6bcb43f3c0bc7>

```
1 chmod +x setup/03-configurations.sh
```

...and run it.

```
1 ./setup/03-configurations.sh
```

The only thing left is to source the environment variables.

```
1 source .env
```

Building Configuration Packages

I have a feeling that you might think that I am trying to trick you by hiding something so let's start by showing that there is (almost) nothing in the control plane cluster we're currently using.

Are there any Compositions?

```
1 kubectl get compositions
```

Nope. The output says `No resources found`.

How about Packages?

```
1 kubectl get pkgrev
```

It's still `no resources found`.

I'll leave it to you to discover that there is indeed nothing in the cluster except Crossplane itself and a Secret with the credentials for whichever hyperscaler you chose.

Unlike in the previous chapters, we did not apply **Packages**, **Compositions**, **Composite Resource Definitions**, or any other type of resources we used so far. I wanted to ensure that we can package all of those into a container image and apply it to a "virgin" control plane.

Actually, that's not really true, but we'll get to the exceptions later.

Now, let's take a look at a directory that contains a new version of the Compositions by entering into the directory,...

```
1 cd compositions/sql-v7
```

...and listing everything inside.

```
1 ls -l
```

The output is as follows.

```
1 aws.yaml
2 azure.yaml
3 crossplane.yaml
4 definition.yaml
5 google.yaml
```

That whole directory is a copy of the last one we used in the previous chapter. I did not modify Compositions or the Composite Resource Definition. Those are exactly the same. But there is a new file over there. A **Configuration** was added to the `crossplane.yaml` file. That's the new addition to the mix, so let's take a look at it.

```
1 cat crossplane.yaml
```

The output is as follows (truncated for brevity).

```
1  apiVersion: meta.pkg.crossplane.io/v1
2  kind: Configuration
3  metadata:
4    name: dot-sql
5    annotations:
6      meta.crossplane.io/maintainer: Viktor Farcic (@vfarcic)
7      meta.crossplane.io/source: github.com/vfarcic/crossplane-tutorial
8      meta.crossplane.io/license: MIT
9      meta.crossplane.io/description: Fully operational PostgreSQL...
10     meta.crossplane.io/readme: A Configuration package that defines...
11  spec:
12    crossplane:
13      version: ">=v1.14.0"
14    dependsOn:
15      - provider: xpkg.upbound.io/upbound/provider-aws-ec2
16        version: ">=v0.36.0"
17      - provider: xpkg.upbound.io/upbound/provider-aws-rds
18        version: ">=v0.36.0"
19      - provider: xpkg.upbound.io/upbound/provider-azure-dbforpostgresql
20        version: ">=v0.33.0"
21      - provider: xpkg.upbound.io/upbound/provider-gcp-sql
22        version: ">=v0.33.0"
23      - provider: crossplane/provider-sql
```

```

24     version: ">=v0.5.0"
25     # - provider: xpkg.upbound.io/crossplane-contrib/provider-kubernetes
26     #   version: ">=v0.10.0"

```

That Configuration is a Kubernetes resource like anything else related to Crossplane. It contains a few informative annotations unless we publish the Configuration to [Upbound Marketplace](https://marketplace.upbound.io/)².

The “real” action is in the spec section.

Over the, we are specifying that the minimum crossplane version is v1.14.0. That one is important if we use features in Compositions that were added to a specific Crossplane version. We’ll see one of those that became available in Crossplane 1.14 in one of the upcoming chapters.

Finally, there is the spec.dependsOn array with a list of Providers and their versions. Since our Compositions use AWS ec2 and rds, Azure dbforpostgresql, Google Cloud’s sql, and sql Providers, those are the ones we specified here. As you’ll see soon, all of those Providers will be installed automatically when we apply the Configuration.

We also used the kubernetes Provider, but that one is commented on in the Configuration. If you remember from the previous chapter, the Kubernetes Provider needs “extra” resources like the ServiceAccount, ClusterRoleBinding, and ControllerConfig. Also, the Kubernetes Provider needs to be modified to use that ControllerConfig. Otherwise, if we do not apply all those, the Kubernetes Provider would not have permissions to create resources through the Kubernetes API, just as AWS, Azure, and Google Cloud Providers need configurations with authentication. Unfortunately, those cannot be put into the Configuration Package. I should have removed it from the Configuration but I thought to leave it there commented as a reminder that it needs to be applied separately.

Now that we have seen the **Configuration**, we can build a **Configuration Package** that will contain that Configuration together with all **Compositions** and **Composite Resource Definitions** from that directory.

All we have to do is execute `crossplane xpkg build`,...

```
1 crossplane xpkg build
```

...and take another look at the files in that directory.

```
1 ls -l
```

The output is as follows.

²<https://marketplace.upbound.io/>

```
1 aws.yaml
2 azure.yaml
3 crossplane.yaml
4 definition.yaml
5 dot-sql-b1062aa3bfc8.xpkg
6 google.yaml
```

We can see that `dot-sql-*.xpkg` file was added. That's the file that we'll push to a container image registry. That's the OCI image that contains a Configuration Package with (almost) everything needed to run Composite Resources and Composite Resource Claims.

Next, we'll push that package to a container image registry.

We could use any registry. It could be **Docker Hub**, **GitHub Registry**, **Harbor**, or whatever you might be using to store your container images. But, to keep it simple and the same for all following this book, we'll use **Upbound Registry**. It's free and it is the home to most, if not all public Configuration Packages, Providers, and most of the other resources we are or will be using in this book.

First, we need to create an Upbound account, so please open accounts.upbound.io³, and create an account if you do not have it already.

Since I prefer a terminal over a console in a browser, at least when there are some actions to be performed, we'll do the rest of the steps through the **Up CLI**⁴, even though we could perform some of them through the GUI as well.

Let's log in...

```
1 up login
```

...and create a new repository.

```
1 up repository create dot-sql
```

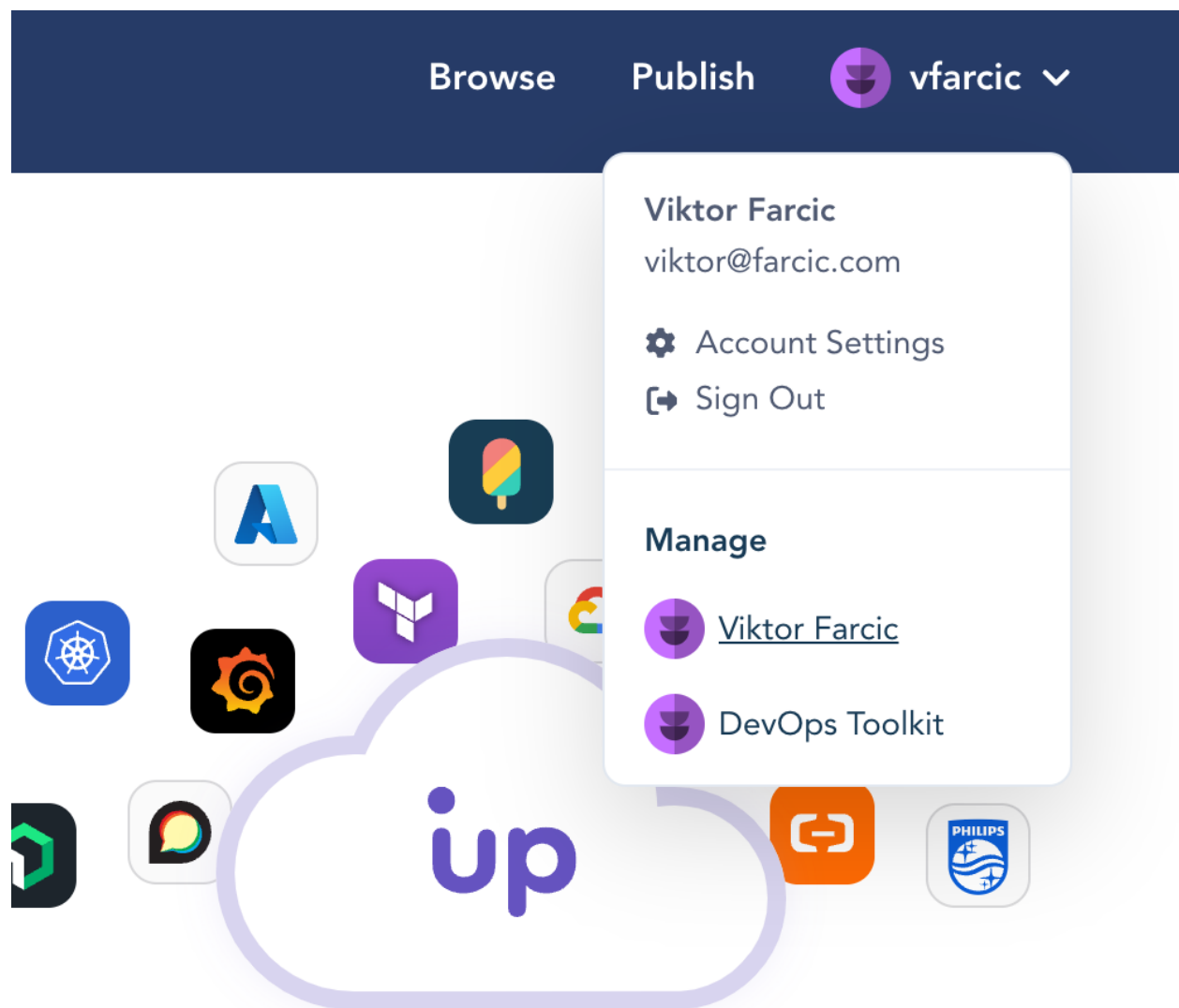
For those who do prefer “pretty colors”, we can see the newly created repository by opening the [marketplace](https://marketplace.upbound.io)⁵ in a browser.

From there on, open the user in the top-right corner of the screen, and choose which account you'd like to manage. If you just registered, you have only one.

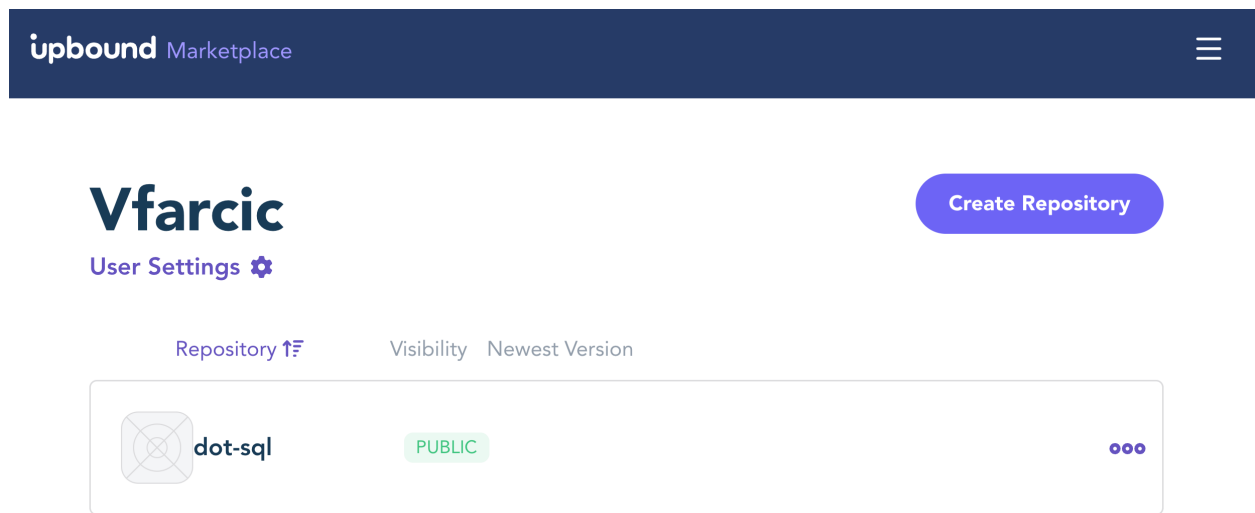
³<https://accounts.upbound.io>

⁴<https://docs.upbound.io/reference/cli>

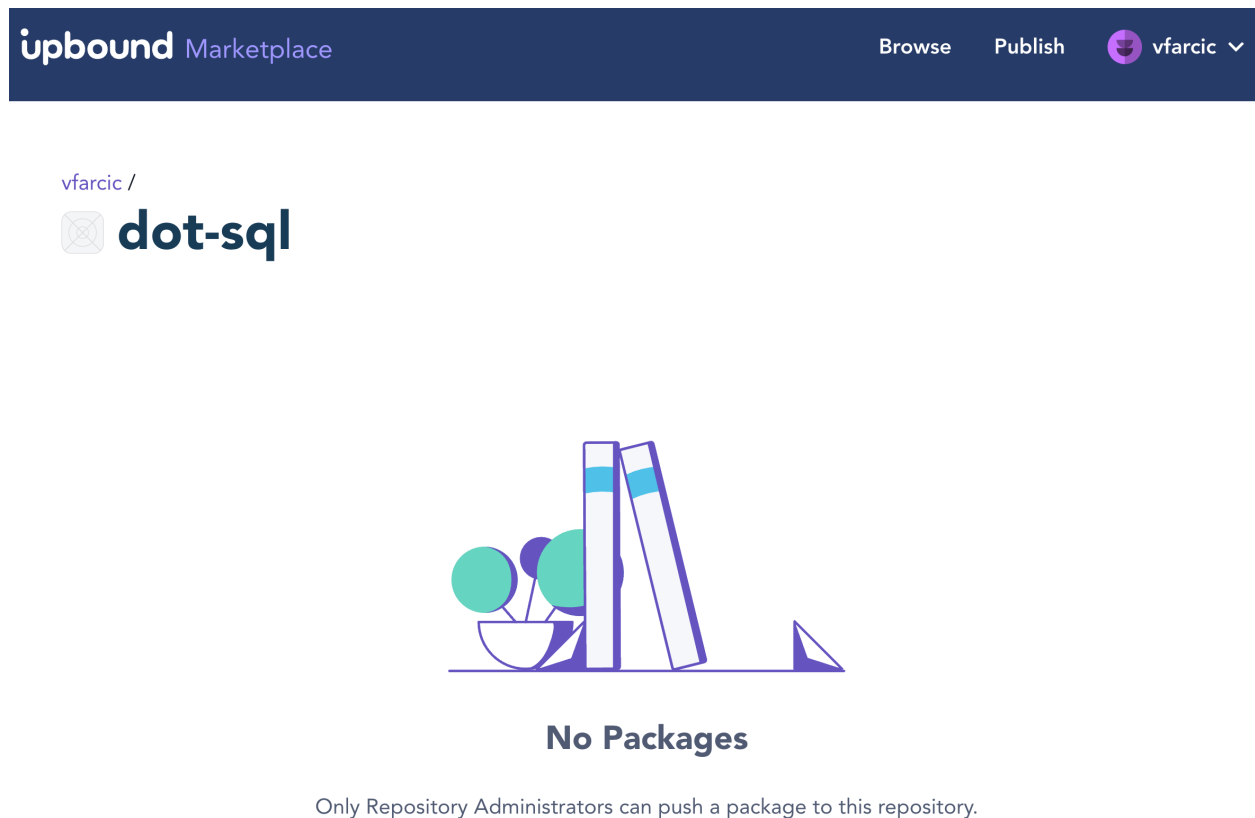
⁵<https://marketplace.upbound.io>



We can see the newly created repository.



If we enter into it, we get the depressing message that there are no packages inside it. We'll change that soon.



We're done with the Marketplace Repository, and now we can turn our attention toward pushing the package we built earlier. But, before we do that, we need to authenticate crossplane CLI with the repository we just created.

First, we'll store the username in the UP_USER variable,...

```
1 # Replace `[...]` with the username
2 export UP_USER=[...]
```

...and login.


```
1 crossplane xpkg login --username $UP_USER
```

Now we are ready to push the package.

```
1 crossplane xpkg push xpkg.upbound.io/$UP_USER/dot-sql:v0.0.7
```


That's it. The Configuration Package was pushed and now it is available to anyone who has permissions to use it which, in this case, is anyone since we created a public repository.



We can refresh the repository in the browser to confirm that the package is there.

upbound Marketplace 

You currently have an accepted but unpublished package. Please reference this [documentation](#) for more information regarding publishing packages.

vfarcic /

 **dot-sql**

Version 	Status	Pushed On	
v0.0.7	ACCEPTED	11/01/2024	

Apart from the fact that the Configuration Package is now stored in the Upbound Marketplace Registry, it could be any other container image registry, we can also see that it is not yet published. That means that the Package is not listed in the Marketplace. We'll keep it like that, mainly to avoid hundreds of dot-sql packages appearing in the Marketplace. After all, this is a demo and not a "real" Package that you would like to share with the world. The instructions are straightforward and I'll assume that, if you do decide to share your creation, you will have no trouble following them.

We'll delete the package since we do not need it anymore on our local file system,...

```
1 rm dot-sql-*.xpkg
```

...and we'll get back to the root of the directory with the fork.

```
1 cd ../../..
```

Here's what we did.

We took our Compositions, Composite Resource Definition, and Configuration and packaged it all into an OCI image. We pushed that image to the container image registry, and now it is available to be deployed to any cluster that contains Crossplane.



Now we are ready to explore how we, and others, can consume the Configuration Package we just built and stored in the registry.

Installing Configuration Packages

Let's get back to our unexciting cluster. The one that only has Crossplane installed. The one without Packages, Compositions, Composite Resource Definitions, or any other type of resources we explored in previous chapters.

Let's transform that uninspiring empty cluster into a control plane that can manage the PostgreSQL database in hyperscalers. We should be able to manage anything else, but PostgreSQL should be a good start.

Everything we need is, probably, in this Configuration.

```
1 cat providers/dot-sql-v7.yaml
```

The output is as follows.

```
1 ---
2 apiVersion: pkg.crossplane.io/v1
3 kind: Configuration
4 metadata:
5   name: crossplane-sql
6 spec:
7   package: xpkg.upbound.io/vfarcic/dot-sql:v0.0.7
```

That's all there is to it (probably). It's a Configuration that references a package we just built and pushed to the registry.

Actually, that's not the one you pushed to the registry, so let's tweak it a bit by replacing `vfarcic` with whichever user you used.

```
1 yq --inplace \
2   ".spec.package = \"xpkg.upbound.io/$UP_USER/dot-sql:v0.0.7\"" \
3   providers/dot-sql-v7.yaml
```

Now we can apply it,...

```
1 kubectl apply --filename providers/dot-sql-v7.yaml
```

...and now is the time to take a break for a few minutes since, as you already know, it takes a while until all the packages are deployed and all the CRDs are created. Get some coffee.

Once you're back, we can take a look at the Package revisions.

```
1 kubectl get pkgrev
```

The output is as follows (truncated for brevity).

```
1 NAME                HEALTHY REVISION IMAGE
2 .../crossplane-sql-... True    1      xpkg.upbound.io/vfarcic/dot-sql:v0.0.7...
3
4 NAME                HEALTHY REVISION...
5 .../crossplane-provider-sql-... True    1      ...
6 .../upbound-provider-aws-ec2-... True    1      ...
7 .../upbound-provider-aws-rds-... True    1      ...
8 .../upbound-provider-azure-dbforpostgresql-... True    1      ...
9 .../upbound-provider-family-aws-... True    1      ...
10 .../upbound-provider-family-azure-... True    1      ...
11 .../upbound-provider-family-gcp-... True    1      ...
12 .../upbound-provider-gcp-sql-... True    1      ...
```

We can see two types of Packages.

At the top, there is the Configuration. That's the `crossplane-sql` Configuration we just applied.

Below are all the Providers we specified in the `crossplane.yaml` file.

Two things are missing though. The Kubernetes Provider is not there. We already established that one is “complicated” and that we’ll have to apply it outside the Configuration Package. Provider Configs, those that contain credentials for hyperscalers, are also missing. We’ll get to them in a moment. For now, let’s confirm that the Compositions are there as well.

```
1 kubectl get compositions
```

The output is as follows.

```
1 NAME                XR-KIND XR-APIVERSION          AGE
2 aws-postgresql      SQL     devopstoolkitseries.com/v1alpha1 8m28s
3 azure-postgresql    SQL     devopstoolkitseries.com/v1alpha1 8m28s
4 google-postgresql   SQL     devopstoolkitseries.com/v1alpha1 8m28s
```

All three Compositions we defined in the previous chapter are available, and we can get back to the things we’re missing.

It would be unreasonable to add Provider Configs into Configuration Packages. They contain credentials or, to be more precise, references to Secrets with credentials for, in this case, operating hyperscalers. We would not get far if we added those to container images. So, we need to apply them separately, just as we did before.

Here’s the one we used in previous chapters.

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

The output is as follows.

```
1 ---
2 apiVersion: aws.upbound.io/v1beta1
3 kind: ProviderConfig
4 metadata:
5   name: default
6 spec:
7   credentials:
8     source: Secret
9     secretRef:
10      namespace: crossplane-system
11      name: aws-creds
12      key: creds
```

You already saw `ProviderConfig` manifests, including that one, in previous chapters, so I won't repeat what they do and how they work. The only note I have is that, in this chapter, I am using **AWS**, so the config references a `Secret` with AWS credentials. You'll see in your terminal the config that matches whichever hyperscaler you chose.

That's it. Let's apply it.

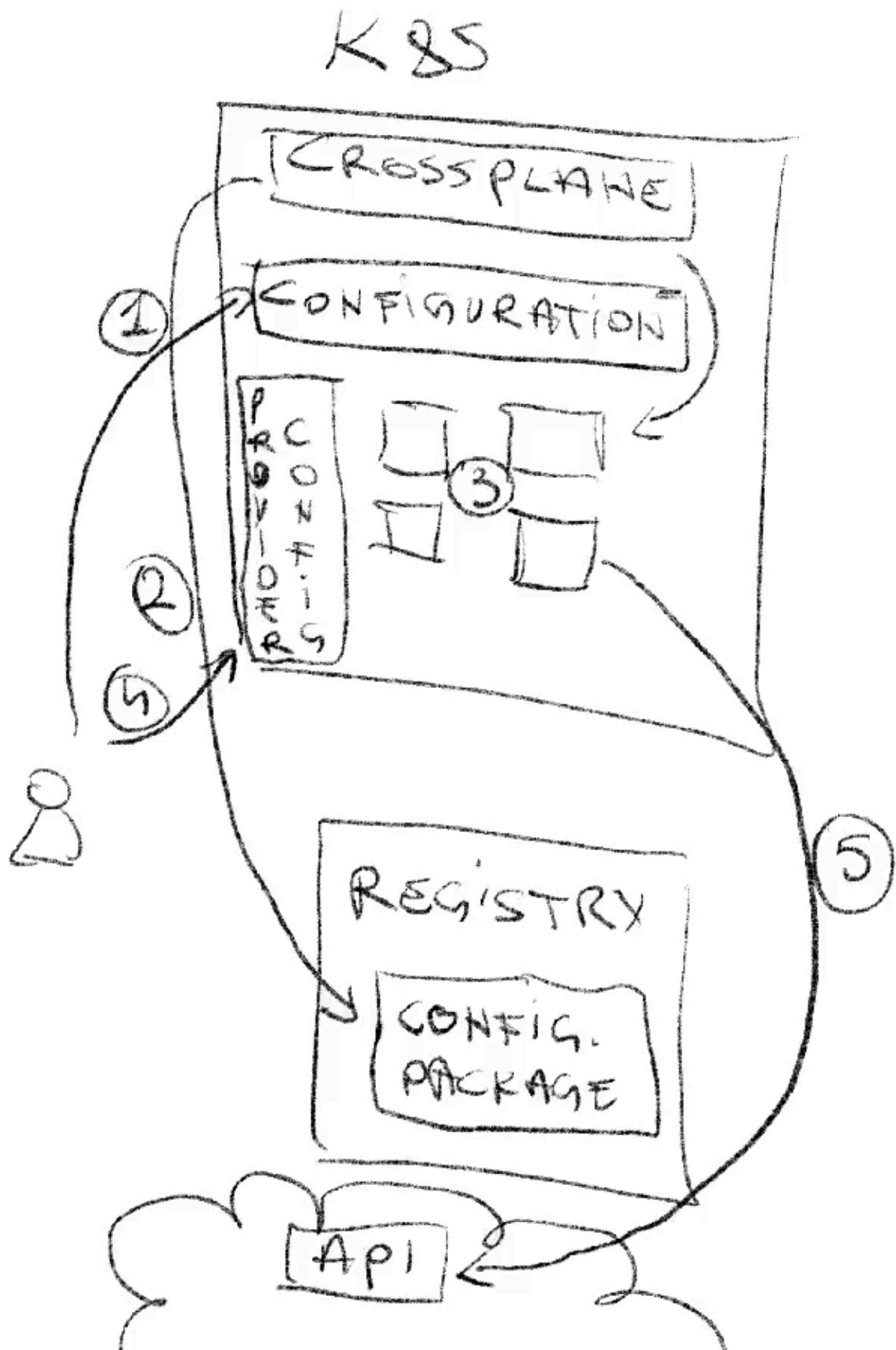
```
1 kubectl apply --filename providers/$HYPERSCALER-config.yaml
```

Finally, the last piece of the puzzle is the “unfortunate” Kubernetes Provider which requires “extra care”. We discussed it in the previous chapter, so let's just apply it.

```
1 kubectl apply \
2   --filename providers/provider-kubernetes-incluster.yaml
```

Here's what we did.

We applied Configuration (1) which downloaded the image (2) which Crossplane used to install the Providers, Compositions, and Composite Resource Definitions (3). The only thing missing was to apply Provider Configs with credentials (4) those Providers can use to authenticate with APIs like AWS, Azure, and Google Cloud (5).



Now we're done. From now on, anyone can apply Composite Claims. Everything we did in this chapter is related to how we manage Compositions, Composite Resource Definitions, and Providers. Managing Composite Claims which manage Composite Resource which manage Managed Resources stays the same. As proof, we can just apply the same Claim manifest we used in the previous chapter, without even explaining it or showing it since... You already know what it looks like and what it does.

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

To confirm that everything works as expected, we can execute `crossplane trace`.

```
1 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: ...resource claim...
└ SQL/my-db-2z1wb	True	False	Creating: ...ce,...
└ VPC/my-db	True	True	Available
└ Subnet/my-db-a	True	True	Available
└ Subnet/my-db-b	True	True	Available
└ Subnet/my-db-c	True	True	Available
└ SubnetGroup/my-db	True	True	Available
└ InternetGateway/my-db	True	True	Available
└ RouteTable/my-db	True	True	Available
└ Route/my-db	True	True	Available
└ MainRouteTableAssociation/my-db	True	True	Available
└ RouteTableAssociation/my-db-1a	False	-	ReconcileError:...
└ RouteTableAssociation/my-db-1b	False	-	ReconcileError:...
└ RouteTableAssociation/my-db-1c	False	-	ReconcileError:...
└ SecurityGroup/my-db	True	True	Available
└ SecurityGroupRule/my-db	True	True	Available
└ Instance/my-db	True	False	Creating
└ ProviderConfig/my-db	-	-	
└ Database/my-db	False	-	ReconcileError:...
└ ProviderConfig/my-db-sql	-	-	
└ Object/my-db	False	-	ReconcileError:...

The Claim created the Composite Resource which created Managed Resources. Everything works as expected or, to be more precise, everything will be available eventually.

You can wait until all the resources are `Available`, or move on. You already saw the end result. We just changed the path how to get to it.

From now on, you can keep updating `Compositions` and, once a new release is ready, build a new version of the **Configuration**, **package** it, **push** it to the registry. As a result, whichever changes you made to `Compositions` or the dependencies like `Providers`, will be applied whenever you apply the new `Configuration` to your control plane cluster(s).

There are still some things missing though. We shouldn't release new `Configuration Packages` without testing them and we should automate the process of testing, building, and pushing packages through pipelines (what you might call CI). We'll get to that in one of the upcoming chapters.

We are finished, for now, so let's destroy everything.

Destroy Everything

You know the drill.

Make the "destroy" script executable,...

```
1 chmod +x destroy/03-configurations.sh
```

...run the script,...

```
1 ./destroy/03-configurations.sh
```

...exit Nix Shell,...

```
1 exit
```

...and take a break. You deserve it.

Composition Functions

So far, we have been successful in **building and publishing Crossplane Compositions**, but it wasn't easy and the results were not necessarily what we need them to be.

We are facing two potential problems, even though I did not mention them in the previous chapters.

To begin with, we had to write **a lot of YAML**. The AWS Composition has over four hundred lines of YAML. Google is much better with half of that at around two hundred lines, and Azure has only around one hundred and twenty. Those Compositions do a lot of work so those sizes are justified. Yet, there is no denying that having hundreds of lines of YAML might not be the most optimal way to write manifests. The users, those that create Claims, are shielded from all that since their needs are covered with only a few lines. Still, while using Claims is easy, defining the Compositions is not.

Fortunately, it's only YAML that represents **Kubernetes resources** so we can use any format and any tool that we normally use to generate those manifests. It could be **Helm** templates, **Kustomize** overlays, **cdk8s** written in Go, Python, TypeScript, or Java, **Jsonnet**, or whichever other format you might be using today. That's the beauty of Kubernetes. It expects YAML, yet it does not care how we generate it before submitting it to its API.

Nevertheless, having too much YAML is not the problem we'll try to solve in this chapter. I'm saying that in part since it would be hard for me to guess what is your preference, but also since you probably already know how to write **Helm** templates, **Kustomize** overlays, or whatever else might be your format and tool of choice when managing Kubernetes resources. Remember, those are all Kubernetes resources so there is probably no good reason not to use whatever you're already using. We might get to it in one of the upcoming chapters but, for now, we'll focus on solving a different problem.

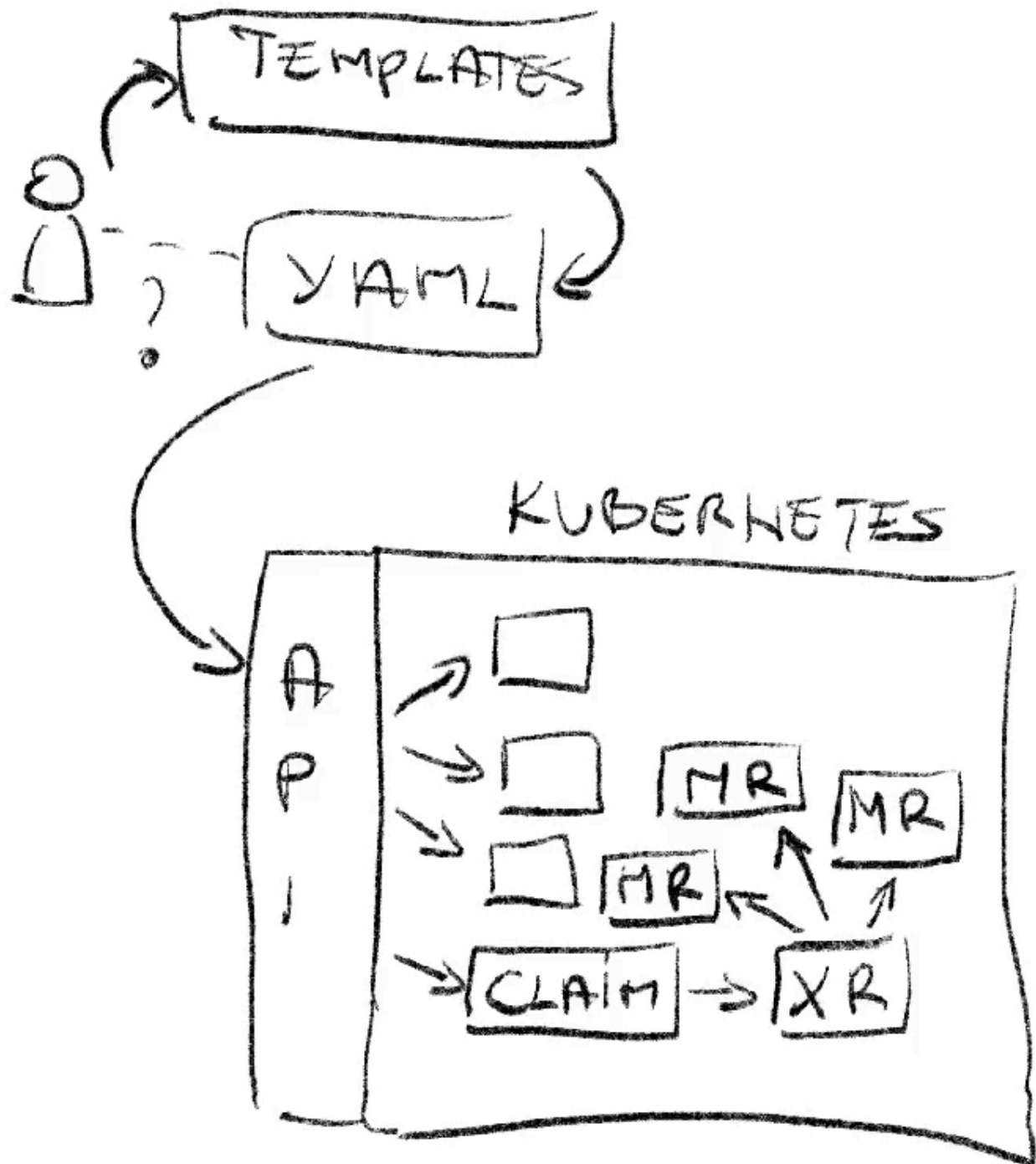
Crossplane **Compositions** offer **no flexibility with Managed Resources**. Whatever we specify in Compositions is what will be managed. We can customize the experience through Patching, but that's limited to how we retrieve and populate values. Crossplane Compositions do not allow conditionals, loops, or more complex logic.

That's the problem we'll tackle in this chapter. We'll see how to use **Composition Functions** to "make" Crossplane Compositions do whatever we want them to do.

But, before we proceed, I want to be clear that those two problems I mentioned are very distinct.

The first one, typically solved with templating solutions, is the problem of how we generate Kubernetes manifests that are applied to Kubernetes API. **Helm**, **Kustomize**, and other similar tools provide a way to convert templates or overlays, or whatever we're using, to Kubernetes YAML which is submitted to Kubernetes API. Kubernetes API does not care about templates. It works only with YAML which contains manifests that are converted into resources in a cluster.

The second issue is related to processes happening inside the cluster. Once a Composite Claim is sent to Kubernetes API, a controller “expands” that Claim into Composite Resources (XRs) which, in turn, create and manage Managed Resources (MRs). Unlike Helm, Kustomize, and similar tools that perform some operations on the client side (e.g., your laptop), **Composition Functions** enable us to be flexible with the processes happening inside the cluster. The difference is whether we are “flexible” before or after submitting something to Kubernetes API.



Today we are tackling the latter. We'll use **Composition Functions** to provide flexibility to processes happening inside the control plane cluster.

Let me give you the background story that led us towards Composition Functions.

Since Crossplane's early days, users have been asking the community to add new features related

to the generation of Managed Resources.

People asked for conditionals so that they could decide programmatically which resources are created. That's easy to add, but we did not add it.

People asked for loops so that they could iterate over values to create a variable number of Managed Resources. That's easy as well, but we did not add that either.

People asked for more complex features like using lookups into a hyperscaler to find a list of available IPs. We certainly did not add that.

There were countless other requests, and none of them made it into Crossplane Compositions.

Why is that so? Does that mean that the Crossplane community does not care about feature requests?

The community is trying to accommodate the needs of Crossplane users but is also trying to avoid converting Crossplane APIs into a mess. Adding everything everyone needs is a recipe for an API to become too complex and to serve everyone's needs poorly. So, the community decided to keep Crossplane APIs, specifically Composition API, simple. We are not trying to convert it into a DSL.

Nevertheless, the problem was still there. People needed additional capabilities and the community did not want to add them to the Composition spec. Instead, the community came up with ways to extend Composition's capabilities without baking any of those capabilities into its spec. That "extension" is today known as **Composition Functions**. Let's check them out.

As in every chapter, we'll start with a setup.

Chapter Setup

You know the drill by now.

All the commands user in this chapter are in the [Gist](https://gist.github.com/vfarcic/9ae7abe89ebab199aa6e52a364e01edc)¹.

Enter the `crossplane-tutorial` directory, if you're not there already,...

```
1 cd crossplane-tutorial
```

...start the Nix Shell that will bring all the tools we'll need,...

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

...give executable permission to the script,...

```
1 chmod +x setup/04-functions.sh
```

...and run it.

¹<https://gist.github.com/vfarcic/9ae7abe89ebab199aa6e52a364e01edc>

```
1 ./setup/04-functions.sh
```

Once we're done, the only thing left is to source the environment variables.

```
1 source .env
```

Now we're almost ready to dive into Composition Functions. The only thing missing is to figure out what we're missing.

What's Missing?

Let's take a look at the Composite Claim that was applied through the setup script.

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

The output is as follows (truncated for brevity).

```
1 NAME                               SYNCED READY STATUS
2 SQLClaim/my-db (a-team)           True   True  Available
3 └─ SQL/my-db-...                   True   True  Available
4   └─ ResourceGroup/my-db-...       True   True  Available
5     └─ Server/my-db-...             True   True  Available
6       └─ FirewallRule/my-db-...     True   True  Available
7         └─ ProviderConfig/my-db-... -      -    -
8       └─ Database/my-db-...         True   True  Available
```

That's the same Claim we used in the previous chapter, which, I hope, you already went through.

As a side note, your Managed Resources might not yet be available. They will, in time, and there's no need to wait.

Are we missing something?

One potential issue is the Database we're creating inside the PostgreSQL Server. The name of the database is always the same as the Claim ID, which needs to be unique. People creating such Claims might want their databases to be called differently.

We could easily solve that issue by adding yet another field to the **Composite Resource Definition** and employing patching to the Database resource.

A potentially bigger problem is that some users might not need a Database inside the Server. They might be using some other tool to manage databases and rely on Crossplane to manage database servers. Hence, we might want to wrap the Database into a conditional and expose a boolean field

people could use to decide whether to have a database inside the server they're managing through Composite Claims. That's something we could solve through a **Composition Function**.

Others, on the other hand, might need more than one database so, instead of having a conditional, we might want to add a **loop over an array**. So, if people set the array to an empty value, there would be no databases. If they put a single item in the array, there would be one database. Two items would result in two databases, and so on and so forth. The items of that array could be database names.

That should work.

So, the plan is to **extend the Composite Resource Definition** to allow people to specify an array with database names. That should solve all three downsides of our current design of the Composition. People will be able to specify names for databases and they will be able to have any number of databases, including none.

There's more though.

This might be a good opportunity to introduce database schemas as well. If users of our Compositions can create PostgreSQL servers with all the related infrastructure and services, as well as a variable number of databases in those servers, they should also be able to define schemas as well.

Now, Crossplane itself does not have a mechanism to manage database schemas. It could have it, but, as far as I know, no one wrote a Provider for that, so we'll use a separate tool.

I picked [Atlas Operator](https://atlasgo.io/integrations/kubernetes/operator)². It works great and is one of only a handful of schema management tools that were designed to work as Kubernetes controllers. Feel free to check [this video](https://youtu.be/1iZoEFzlvhM)³ if you are not familiar with it. As a matter of fact, I included it in the setup script, so it's already running inside the cluster.

What's missing is to **add AtlasSchema resources to the Compositions**. It will need a Secret with the authentication to the database server. We are already generating Secrets in the Compositions, and we already saw that we can use patching so that AtlasSchema resources are "hooked" into the correct Secrets. We could do all that with the knowledge we have so far. There's nothing in those requirements we cannot do.

However, just as with databases, we should make management of database schemas optional so that people can choose whether they would like to manage them through Composite Claims or through some other means unrelated to Crossplane. Similarly, since databases are variable, schemas should be variable too. Hence, we need another loop for them.

All in all, we'll **extend the Composite Resource Definition** with new fields for databases and schemas, and we'll modify our Compositions to manage a **variable number of databases and schemas**.

That's it. That's all the theory and bla bla bla I'll give. I cannot stand reading for too long without touching my keyboard, so let's jump into practical examples. I know that, by now, you're probably

²<https://atlasgo.io/integrations/kubernetes/operator>

³<https://youtu.be/1iZoEFzlvhM>

very confused and that you probably have no idea what **Composition Functions** are and why we want them. That's normal since I did not really explain them. I will, but through examples, so bear with me.

What matters, for now, is that we know what the requirements for the features we should enable through Compositions, and that **Composite Functions** can help us accomplish that.

The first function we'll explore is not going to provide any new feature and it will not help us fulfill our new requirements. Yet it will be a great way to understand what Composition Functions do before we start working on those requirements.

Patch and Transform Function

















We can choose to use existing Composition Functions or write our own. As a personal rule of thumb, I tend to start by checking whether there is already a Function that does what I need it to do before I start writing my own.

TOOD: Lower-third: Upbound Marketplace, <https://marketplace.upbound.io>, upbound.webp

A good start is the [Upbound Marketplace](https://marketplace.upbound.io)⁴, so let's open it, click the Browse button on the top-right side of the screen, and choose Functions type from the left-hand menu. We can see a few functions.

Functions

Composition Functions allow you to use advanced logic, programming languages, and tools to configure your Crossplane Compositions.

 function-auto-ready Upbound Official Upbound's official composition function that automatically detects when resources are ready		 function-go-templating Upbound Official Upbound's official Go templating composition function	
 function-patch-and-transform Upbound Official Upbound's official patch & transform composition function		 function-auto-ready crossplane-contrib A composition function that automatically detects when resources are ready	
 function-dummy crossplane-contrib A composition function that returns what you tell it to		 function-go-templating crossplane-contrib A Go templating composition function	
 function-patch-and-transform crossplane-contrib A patch & transform composition function		 function-argo-eks-discovery upbound A composition function that search for eks cluster and create argocd cluster managed resource	

⁴<https://marketplace.upbound.io>

At the time of this writing, Composition Function just reached beta and there aren't many available in the Marketplace. That does not mean that there aren't many since quite a few teams are already writing and using their own Functions. However, some of them are private, while others are publicly available, just not through the Marketplace. Many are spread across GitHub repositories. Hopefully, we'll see those published in the Marketplace soon.

We'll pick the Patch and Transform Function, so let's enter into it. Over there, we can see that the Function has only one input Resources. If we enter inside Resources, we can see the spec just as we'd see the spec of a Provider or a Configuration.

API Documentation

+ apiVersion string

- environment object

Environment represents the Composition environment. THIS IS AN ALPHA FIELD. Do not use it in production. It may be changed or removed without notice.

- patches array

Patches is a list of environment patches that are executed before a composition's resources are composed. These patches are between the XR and the Environment. Either from the Environment to the XR, or vice versa.

- combine object

Combine is the patch configuration for a CombineFromComposite, CombineToComposite patch.

+ strategy required string

- string object

String declares that input variables should be combined into a single string, using the relevant settings for formatting purposes.

+ fmt required string

- variables required array

I already prepared everything we need to use the Patch and Transform function, so let's take a look at the manifests I wrote.

To begin with, we need to deploy the Function. Functions are a specific type of Packages just as Providers and Configurations are, so we have a manifest similar to those.

```
1 cat providers/function-patch-and-transform.yaml
```

The output is as follows.

```

1  apiVersion: pkg.crossplane.io/v1beta1
2  kind: Function
3  metadata:
4    name: crossplane-contrib-function-patch-and-transform
5  spec:
6    package: xpkg.upbound.io/crossplane-contrib/function-patch-and-transform:v0.1.4

```

The only structural difference between that manifest and those we used to define Providers and Configurations is the `kind` field which is now set to `Function`, so there's probably nothing new to say about it.

*Bear in mind that, as we saw in the previous chapter, we should package Compositions, Composite Resource Definitions, and Configuration into **Configuration Packages**. Hence, you probably guessed that functions should be defined in `crossplane.yaml`. If that's what you're thinking, you're right. However, for the sake of simplicity and agility, we'll skip Configuration Packages for now and apply changes directly. That will allow us to move faster. We will package and publish them as a new OCI image at the very end of the chapter, once we're happy with the changes we'll make.*

Let's apply the Function.

```

1  kubectl apply \
2    --filename providers/function-patch-and-transform.yaml

```

As a result, we have a **Composition Function** running in our control plane cluster. It's not doing anything though. It's just sitting there waiting for someone, or something, to send it some input. We'll let it enjoy idleness for a while longer.

Let's see a modified version of the Composition we worked on.

```

1  cat compositions/sql-v8/$HYPERSCALER.yaml

```

The output is as follows (truncated for brevity).

```

1  ---
2  apiVersion: apiextensions.crossplane.io/v1
3  kind: Composition
4  metadata:
5    name: azure-postgresql
6    labels:
7      provider: azure
8      db: postgresql
9  spec:
10   compositeTypeRef:
11     apiVersion: devopstoolkitseries.com/v1alpha1

```

```

12     kind: SQL
13     mode: Pipeline
14     pipeline:
15     - functionRef:
16         name: crossplane-contrib-function-patch-and-transform
17     step: patch-and-transform
18     input:
19         apiVersion: pt.fn.crossplane.io/v1beta1
20         kind: Resources
21         patchSets:
22         - name: metadata
23           patches:
24             - fromFieldPath: metadata.annotations
25               toFieldPath: metadata.annotations
26             - fromFieldPath: spec.id
27               toFieldPath: metadata.name
28         resources:
29         - name: resourcegroup
30           base:
31             apiVersion: azure.upbound.io/v1beta1
32             kind: ResourceGroup
33             spec:
34               forProvider:
35                 location: eastus
36           patches:
37             - type: PatchSet
38               patchSetName: metadata
39         - name: server
40           base:
41             ...

```

This time we have `spec.mode` set to `Pipeline`. The alternative is `Resources`, which is the default value. Since we did not specify it in the past, all our Compositions were running in the `Resources` mode.

The `Pipeline` mode acts similarly to CI pipelines you're probably experienced with. It is an array of steps executed one after another. Each of the steps is processed by a function which, in this case, is the `Patch and Transform Function` referenced through the `spec.pipeline[].functionRef.name` field. Finally, each Function defines an optional input. Input is what is passed to a Function which is expected to produce an output. That output is effectively acting in the same way as `Resources` we were defining in the past. The difference is that, instead of hard-coding resources inside the Composition, we are letting Functions generate those resources inside the control plane cluster.

Inside input are a few fields that are common to all the Functions. Those are the fields that any

Kubernetes resource needs to define; `apiVersion`, `kind`, and optional `metadata`. The rest depends on the Function. The author of a Function is free to define any schema. In the case of the Patch and Transform Function, the one we're using right now, only two additional fields are available; `patchSets` and `resources`. You'll notice that those are the same fields we used in the past, except that now they are nested inside `spec.pipeline[].input`. As a matter of fact, this new Composition is exactly the same as the previous one, except that I moved `spec.patchSets` and `spec.resources` fields inside `spec.pipeline[].input`.

Now, you might be asking: **“What is the purpose of the Patch and Transform Function?”** The answer is simple. It provides the patch and transform functionality we used before. That might lead you to ask a follow-up question: **“What’s the difference between patch and transform functionality baked into Compositions and the one provided as the Function?”** The answer to that one is **“none that matters.”** Both are the same. As a matter of fact, there might be no good reason to rewrite your Compositions to use the Patch and Transform Function. The end result is going to be the same, except that patching and transformations will not be done by Crossplane itself, but by a Function running independently of Crossplane.

So, why are we doing all this? Why did we rewrite our Compositions to use the Patch and Transform Function if there are no tangible benefits?

There are two reasons for that.

First, we switched to the `Pipeline` mode which forces us to use Functions for all steps. The `Pipeline` mode does not have patching and transformations baked in, so we had to use that Function. Second, the patch and transform functionality that is baked into the `Resources` mode we used so far is likely to be deprecated in favor of using the Patch and Transform Function. That effectively means not only that Crossplane itself is not likely to add additional features to Compositions, like those we were discussing earlier, but it might even remove some of them, in favor of Functions.

Let's apply the new Composition and...

```
1 kubectl apply --filename compositions/sql-v8/$HYPERSCALER.yaml
```

...trace the Claim.

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

The output is as follows (truncated for brevity).

1	NAME	SYNCED	READY	STATUS
2	SQLClaim/my-db (a-team)	True	True	Available
3	└─ SQL/my-db-rmwkj	True	True	Available
4	└─ ResourceGroup/my-db-...	True	True	Available
5	└─ Server/my-db-...	True	True	Available
6	└─ FirewallRule/my-db-...	True	True	Available
7	└─ ProviderConfig/my-db-...	-	-	
8	└─ Database/my-db-...	True	True	Available

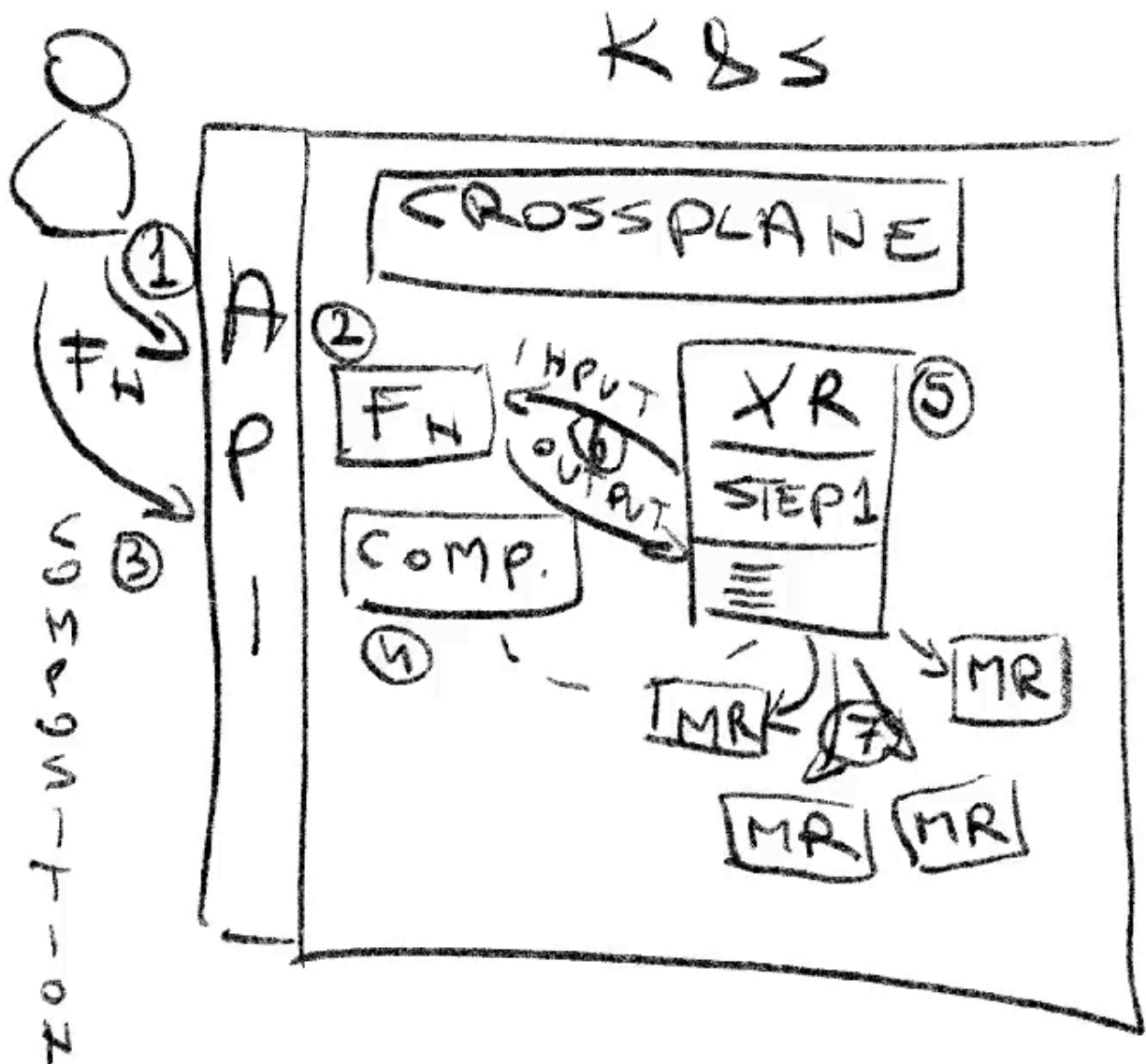
The outcome is exactly the same. Nothing changed, except that the Composite Resource is not performing patching and transformations anymore but **delegating it to the Function**. The Function, in turn, returned patched resources to the Composite Resource which ended up being the same Managed Resources we had before.

We can consider the changes we just applied as refactoring. We switched to the potentially better design without changing the functionality.

Here's what happened.

We applied a Function in the control plane cluster (1). That Function is a separate process (2) that expects an input and produces an output. From that perspective, all Functions are the same. The process that produces the output differs from one Function to another.

Further on, we applied the Composition set to run in the Pipeline mode (3, 4). As a result, the Composite Resource (XR) that was already running in that cluster, iterated over all the steps (5). We had only one step which sent input to the Patch and Transform Function (6). The Function returned resources to the Composition and those resources became Managed Resources (MRs) (7).



In other words, Compositions in the Pipeline mode do not define the resources anymore. Instead, they contain steps with inputs that are sent to Functions which, in turn, return resources to the Composition. Those Functions can be anything. They could add resources, modify them, or remove them. The logic inside a Function can do anything. As long as it can accept an input and return resources as output, Crossplane does not care what a Function does. The one we used happens to have the same Patch and Transform functionality as the one baked into Composition Resources running in the Resources mode. Soon we'll see more interesting examples of Functions that do something truly different; something that is currently not possible to do in Compositions.

One of those is the Go Templating Function.

Go Templating Function

Let's take another look at the databases we have in the PostgreSQL server.

```
1 kubectl get databases.postgresql.sql.crossplane.io
```

The output is as follows (truncated for brevity).

```
1 NAME          READY SYNCED AGE
2 my-db-... True  True   9m54s
```

There is only one database in that server and its name is the same as the ID of the Claim. As we already commented, we'd like to change the functionality of our Compositions so that users can request any number of databases in a given server, and that those databases can have user-specified names.

To accomplish that, the first step is to change our Composite Resource Definition.

I already prepared a new version, so let's take a look at it.

```
1 cat compositions/sql-v8/definition.yaml
```

The output is as follows (truncated for brevity).

```
1 ---
2 apiVersion: apiextensions.crossplane.io/v1
3 kind: CompositeResourceDefinition
4 metadata:
5   name: sqls.devopstoolkitseries.com
6 spec:
7   ...
8   versions:
9     ...
10   schema:
11     openAPIV3Schema:
12       type: object
13       properties:
14         spec:
15           type: object
16           properties:
17             ...
18           parameters:
```

```

19         type: object
20         properties:
21             ...
22         databases:
23             description: The list of databases to create...
24             type: array
25             items:
26                 type: string
27         ...

```

We added the `databases` field. The type is `array` meaning that it can accept a list of `items`. Those `items` are strings, each representing a name of a database.

There's not much more to it, so let's apply it.

```
1 kubectl apply --filename compositions/sql-v8/definition.yaml
```

Now that we extended the CRD with the additional field that represents a list of databases, we need to figure out how to generate as many resources as there are databases set in a Claim. We already know that Compositions do not have the loop functionality, so we need to do it through Functions.

One possible solution is the [Go Templating Function](#)⁵. It allows us to generate resources using templates. It's conceptually similar to Helm Templates which are also based on Go Templates. The major difference is that the templates are processed inside a cluster.

I already prepared the Function manifest.

```
1 cat providers/function-go-templating.yaml
```

The output is as follows.

```

1 ---
2 apiVersion: pkg.crossplane.io/v1beta1
3 kind: Function
4 metadata:
5     name: upbound-function-go-templating
6 spec:
7     package: xpkg.upbound.io/crossplane-contrib/function-go-templating:v0.4.0

```

There's probably no need to explain it, so let's just apply it.

⁵<https://marketplace.upbound.io/functions/crossplane-contrib/function-go-templating>

```
1 kubectl apply --filename providers/function-go-templating.yaml
```

Finally, we should add the Function as the second step in the Pipeline.
Here's the updated version of the Composition.

```
1 cat compositions/sql-v9/$HYPERSCALER.yaml
```

The output is as follows (truncated for brevity).

```
1 ---
2 apiVersion: apiextensions.crossplane.io/v1
3 kind: Composition
4 metadata:
5   name: azure-postgresql
6   ...
7 spec:
8   ...
9   pipeline:
10    - functionRef:
11      name: crossplane-contrib-function-patch-and-transform
12      ...
13    - functionRef:
14      name: upbound-function-go-templating
15      step: sql-db
16      input:
17        apiVersion: gotemplating.fn.crossplane.io/v1beta1
18        kind: GoTemplate
19        source: Inline
20        inline:
21          template: |
22            {{ range .observed.composite.resource.spec.parameters.databases }}
23            ---
24            apiVersion: postgresql.sql.crossplane.io/v1alpha1
25            kind: Database
26            metadata:
27              name: {{ $.observed.composite.resource.spec.id }}-{{ . }}
28              annotations:
29                crossplane.io/external-name: {{ . }}
30                gotemplating.fn.crossplane.io/composition-resource-name: {{ $.observed\
31 .composite.resource.spec.id }}-{{ . }}
32            spec:
33              providerConfigRef:
```

```
34         name: {{ $.observed.composite.resource.spec.id }}
35     forProvider: {}
36     {{ end }}
```

The first thing you'll notice, if you pay close attention, is that the Database we had as one of the resources in the first step is now gone.

Second, there is a new step which is using the Go Templating Function with a template. That template is the standard Go template (similar to Helm templates). What makes it "special" is the usage of the observed field.

As I already mentioned, when a Composition invokes a Function, it sends the contents of the input. What I (probably) did not mention is that it also sends it the current state of the Composite Resource. That's the observed field we're referencing in the template. That one contains everything, including the definition of the Composite Resource, the one created by the Claim, as well as all the resources assembled by Functions executed as previous steps. We don't care for the resources assembled previously since that's mostly relevant to those who write Functions. We are, for now, only Function users, not builders.

The part we care about is the ability to access fields from the Composite Resource. More specifically, we care about the `spec.parameters.databases` and `spec.id` fields. Those, and all other Composite Resource fields are available inside the `observed.composite.resource` field.

With that explanation out of the way, let's take a quick look at the template itself.

We're iterating through the values of the `spec.parameters.databases` field, which is an array, and generating a Database for each.

There's probably no need to explain anything else, as long as we understand that everything is available in the observed field and that at least a basic understanding of Go Templating is required to use this Function. Even if you never worked with Go Templates, I bet you worked with Helm templates which are also based on Go Templates. Hence, I'm confident that you're familiar with the syntax.

The only things that might be interesting to comment on are the values of the `metadata.name` and `metadata.annotations.crossplane.io/external-name` fields.

We could have left the `metadata.name` empty, resulting in Crossplane autogenerating a unique name. Since I don't like dealing with random names without meaning, we are using a combination of `spec.id` and the database name which is the item from the range (`{{ . }}`). So, we are generating a unique resource name that combines those two. However, while that works for the name of the Managed Resource, I did not want the "real" names of databases to be anything but what users specify in the `spec.parameters.databases` field of the Claim.

By default, the "real" name of a resource, the one that, in my case, ends up being something in Azure, is the same as the Kubernetes resource name, except when names are auto-generated like in the case of AWS VPCs. Hence, those are sometimes the same, and at other times differ. Crossplane

solves that through the `crossplane.io/external-name` annotation. That's where it keeps the "real" name of a resource, no matter whether it matches the name of the Managed Resource or not.

But, as I already mentioned, I did not want those two to be the same, so we are specifying the `crossplane.io/external-name` annotation ourselves by setting it to the `spec.parameters.databases[]` value (`{{ . }}`).

That's it. The rest is all about knowing how Go Templating works, which is not the subject I'll cover here.

Let's apply the Composition...

```
1 kubectl apply --filename compositions/sql-v9/$HYPERSCALER.yaml
```

...and retrieve all databases.

TODO: Ignore screen coming up!

```
1 kubectl get databases.postgresql.sql.crossplane.io
```

The output is as follows.

```
1 No resources found
```

The old database was removed, but no new databases were added. Did we make a mistake?

We applied a template that iterates over the values in the `spec.parameters.databases` field, but we did not add that field to the Claim. Hence, we have no databases, thus demonstrating one of the requirements. Users who do not specify any databases, get no databases. It's the opt-in model.

Let's add some databases.

```
1 cat examples/$HYPERSCALER-sql-v9.yaml
```

The output is as follows (truncated for brevity).

```
1 ...
2 apiVersion: devopstoolkitseries.com/v1alpha1
3 kind: SQLClaim
4 metadata:
5   name: my-db
6   ...
7 spec:
8   ...
9   parameters:
```

```
10     ...
11     databases:
12         - db-01
13         - db-02
```

The updated manifest added two databases; db-01 and db-02.

Let's apply it...

```
1 kubectl --namespace a-team apply \
2     --filename examples/$HYPERSCALER-sql-v9.yaml
```

...and retrieve all databases.

```
1 kubectl get databases.postgresql.sql.crossplane.io
```

We can see that we got two new databases.

```
1 NAME                                READY SYNCED AGE
2 my-db-...-db-01                      7s
3 my-db-...-db-02                      7s
```

It'll take a few seconds until they are up and running inside the server, so let's use that time to double-check that everything works as expected. I don't trust myself, so you should not trust me either.

So, let's double-check that the databases were indeed created inside the server. We already did go through the hurdle of getting authentication data and running a container with `psql`, so we can do this without much of an explanation.

Here we go.

Retrieve the Secrets in the a-team Namespace,...

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

...and store the server name (the same as the Secret name) in the `DB_NAME` variable if it's Azure (and its insistence to have unique names). For everyone else, it'll be `my-db`.

```
1 export DB_NAME=my-db
```

Next, we'll retrieve the user,...

```
1 export PGUSER=$(kubectl --namespace a-team \  
2     get secret $DB_NAME --output jsonpath="{.data.username}" \  
3     | base64 -d)
```

...the password,...

```
1 export PGPASSWORD=$(kubectl --namespace a-team \  
2     get secret $DB_NAME --output jsonpath="{.data.password}" \  
3     | base64 -d)
```

...and the host.

```
1 export PGHOST=$(kubectl --namespace a-team \  
2     get secret $DB_NAME --output jsonpath="{.data.endpoint}" \  
3     | base64 -d)
```

Now we can run a Pod with psql,...

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

...connect to the server,...

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

...and list all databases.

```
1 \l
```

The output is as follows (truncated for brevity).

```

1  ...List of databases...
2      Name      |...
3  -----+...
4  azure_maintenance |...
5  azure_sys        |...
6  db-01            |...
7  db-02            |...
8  postgres         |...
9  template0        |...
10 template1        |...
11 (7 rows)

```

We can see that both databases (db-01 and db-02) were added to the server. Hurray!

We have a problem though, so let's exit the psql CLI,...

```
1  exit
```

...and the container.

```
1  exit
```

We can see the problem we are facing now by running a trace of the Claim again.

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

The output is as follows (truncated for brevity).

```

1  NAME                                SYNCED READY STATUS
2  SQLClaim/my-db (a-team)             True   False Waiting: ...resource...
3  └─ SQL/my-db-rmwkj                 True   False Creating: ...resources: my-db-...-db-01,\
4  my-db-...-db-02
5  │─ ResourceGroup/my-db-...         True   True   Available
6  │─ FirewallRule/my-db-...          True   True   Available
7  │─ Server/my-db-...                True   True   Available
8  │─ Database/my-db-...-db-01        True   True   Available
9  │─ Database/my-db-...-db-02        True   True   Available
10 └─ ProviderConfig/my-db-...         -     -

```

We can see that the STATUS of the SQL Composite Resource is Creating followed by the list of resources that are not ready. To be more precise, the Composition thinks that the Database resources are not Available even though we can see that they are. Let me explain...

The output of Functions is a list of resources and it is the responsibility of a Function to provide the status, which is not what the Go Templating one does. Hence, even though Managed Resources are Available, the SQL Composite Resource, the parent, thinks they are not. We'll fix that later. For now, I just want you to take a mental note that we need to fix that issue.

We solved the requirements around databases, and now we can turn towards the Schemas.

As I already mentioned, Atlas Operator was deployed through the setup script and you can watch [this video](#)⁶ if you are not familiar with it.

So, the operator is running, but we're still missing the means to apply `AtlasSchema` resources through Composite Resources. The process is similar to what we did with databases, so let's go through it quickly.

Here's the updated Composite Resource Definition.

```
1 cat compositions/sql-v10/definition.yaml
```

The output is as follows (truncated for brevity).

```
1 ---
2 apiVersion: apiextensions.crossplane.io/v1
3 kind: CompositeResourceDefinition
4 metadata:
5   name: sqls.devopstoolkitseries.com
6 spec:
7   ...
8   versions:
9     ...
10    schema:
11      openAPIV3Schema:
12        type: object
13        properties:
14          spec:
15            type: object
16            properties:
17              ...
18            parameters:
19              type: object
20              properties:
21                ...
22            schemas:
23              description: Database schema. Atlas operator...
24              type: array
```

⁶<https://youtu.be/1iZoEFzlvhM>

```

25         items:
26             type: object
27         properties:
28             database:
29                 description: The name of the database...
30                 type: string
31             sql:
32                 description: The SQL to apply.
33                 type: string
34         required:
35             - version
36     required:
37         - parameters

```

We added schemas as another array. However, the type of items is, this time, object instead of string, since we need two fields as items of the array. One is the database that should contain the name of the database where the schema should be applied, and the other is sql with the schema itself. Since we are using Atlas Operator, schema consists of SQL statements, hence the name sql.

Let's apply the Composite Resource Definition.

```
1 kubectl apply --filename compositions/sql-v10/definition.yaml
```

Next, there are changes to Compositions.

```
1 cat compositions/sql-v10/$HYPERSCALER.yaml
```

The output is as follows (truncated for brevity).

```

1  ---
2  apiVersion: apiextensions.crossplane.io/v1
3  kind: Composition
4  metadata:
5      name: azure-postgresql
6      ...
7  spec:
8      ...
9  pipeline:
10     ...
11     - functionRef:
12         name: upbound-function-go-templating
13     step: schema
14     input:

```

```

15     apiVersion: gotemplating.fn.crossplane.io/v1beta1
16     kind: GoTemplate
17     source: Inline
18     inline:
19         template: |
20             {{ range .observed.composite.resource.spec.parameters.schemas }}
21             ---
22             apiVersion: kubernetes.crossplane.io/v1alpha1
23             kind: Object
24             metadata:
25                 name: {{ $.observed.composite.resource.spec.id }}-schema-{{ .database }}
26                 annotations:
27                     gotemplating.fn.crossplane.io/composition-resource-name:...
28             spec:
29                 providerConfigRef:
30                     name: {{ $.observed.composite.resource.spec.id }}-sql
31                 forProvider:
32                     manifest:
33                         apiVersion: db.atlasgo.io/v1alpha1
34                         kind: AtlasSchema
35                         metadata:
36                             name: {{ $.observed.composite.resource.spec.id }}-{{ .database }}
37                             namespace: {{ $.observed.composite.resource.spec.claimRef.namespac\
38 e }}
39                 toFieldPath: spec.credentials.connectionSecretRef.namespace
40                 spec:
41                     credentials:
42                         scheme: postgres
43                         hostFrom:
44                             secretKeyRef:
45                                 key: endpoint
46                                 name: {{ $.observed.composite.resource.spec.id }}
47                         port: 5432
48                         userFrom:
49                             secretKeyRef:
50                                 key: username
51                                 name: {{ $.observed.composite.resource.spec.id }}
52                         passwordFrom:
53                             secretKeyRef:
54                                 key: password
55                                 name: {{ $.observed.composite.resource.spec.id }}
56                         database: {{ .database }}
57                     parameters:

```

```

58         sslmode: disable
59     schema:
60         sql: "{{ .sql }}"
61 {{ end }}

```

The logic is pretty much the same as before.

We added a new step that references the same Go Templating Function. Inside the template, we're iterating over all items of the `schemas` field. Each iteration constructs a new Object that applies `AtlasSchema` with the credentials, taken from the Secret with auth data, and the schema itself, which is filled with the `schemas[].sql` field defined in the Composite Resource Definition.

That's it. That's all the explanation I can give without diving into Go Templating, so let's apply it...

```
1 kubectl apply --filename compositions/sql-v10/$HYPERSCALER.yaml
```

...and check whether `atlasschemas` resources were created.

```
1 kubectl --namespace a-team get atlasschemas
```

We can see that no resources were found in `a-team` namespace. That's normal since we did not yet specify the schemas in the Claim, so let's do just that.

Here's the updated Composite Claim.

```
1 cat examples/$HYPERSCALER-sql-v10.yaml
```

The output is as follows (truncated for brevity).

```

1  ...
2  apiVersion: devopstoolkitseries.com/v1alpha1
3  kind: SQLClaim
4  ...
5  spec:
6    ...
7    parameters:
8      ...
9    schemas:
10     - database: db-01
11       sql: |
12         create table videos (
13           id varchar(50) not null,
14           description text,
15           primary key (id)

```

```

16         );
17     create table comments (
18         id serial,
19         video_id varchar(50) not null,
20         description text not null,
21         primary key (id),
22         CONSTRAINT fk_videos FOREIGN KEY(video_id) REFERENCES videos(id)
23     );

```

We added schemas with one item. The database db-01 has the sql set to create tables `videos` and `comments`. Don't be confused by the `create table` statements.

Atlas Operator will create a temporary database, create those tables, compare the schema against the “real” server, and apply the differences. In this case, it will create both tables but, later on, if we modify one of them or add a new one, it will apply one the differences. It's awesome, and you should explore it in more detail on your own or, as I mentioned, watch [this video](#)⁷.

All in all, a schema will be applied to the db-01 while there will be no schema in db-02 since we did not specify any.

Let's apply the modified Composite Claim...

```

1 kubectl --namespace a-team apply \
2     --filename examples/$HYPERSCALER-sql-v10.yaml

```

...and retrieve `atlasschemas`.

```

1 kubectl --namespace a-team get atlasschemas

```

The output is as follows (truncated for brevity).

```

1 NAME          READY REASON
2 my-db-...-db-01 False GettingDevDB

```

We can see that the `atlasschema` was created.

We can confirm that the Composite Resource did what was supposed to do by creating a Pod with `psql` client,...

⁷<https://youtu.be/1iZoEFzlvhM>

```

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

```

...connecting psql with the server,...

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

...switching to the db-01 database,...

```
1 \c db-01
```

...and listing all the tables.

```
1 \dt
```

The output is as follows.

```

1          List of relations
2  Schema |   Name   | Type | Owner
3  -----+-----+-----+-----
4  public | comments | table | postgres
5  public | videos   | table | postgres
6  (2 rows)

```

Both tables were created.

We can confirm it further by outputting the schema of the videos table.

```
1 \d videos
```

The output is as follows.

```

1          Table "public.videos"
2  Column      |          Type          | Collation | Nullable | Default
3  -----+-----+-----+-----+-----
4  id           | character varying(50) |           | not null |
5  description  | text                  |           |          |
6  Indexes:
7      "videos_pkey" PRIMARY KEY, btree (id)
8  Referenced by:
9      TABLE "comments" CONSTRAINT "fk_videos" FOREIGN KEY (video_id) REFERENCES videos\
10 (id)

```

We can see that the `videos` table contains the fields we specified and that it is referencing the `comments` table.

Everything seems to be working correctly, so let's get out of `psql`,...

```
1 exit
```

...and the Pod.

```
1 exit
```

We're left with only one issue to solve. We need to make the Composite Resource understand the "real" status of the Managed Resources created through the Go Templating Function.

Auto-Ready Function

Let's trace the Composite Claim again.

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

The output is as follows (truncated for brevity).

```

1  NAME                                SYNCED READY STATUS
2  SQLClaim/my-db (a-team)             True   False Waiting: ...resource claim is waiti\
3  ng...
4  └─ SQL/my-db-rmwkj                  True   False Creating: ...y-db-...-db-02, and my\
5  -db-...-schema-db-01
6      └─ ResourceGroup/my-db-...       True   True   Available
7      └─ FirewallRule/my-db-...       True   True   Available
8      └─ Server/my-db-...              True   True   Available
9      └─ Object/my-db-...-schema-db-01 True   True   Available
10     └─ ProviderConfig/my-db-...-sql  -      -
11     └─ Database/my-db-...-db-01      True   True   Available
12     └─ Database/my-db-...-db-02      True   True   Available
13     └─ ProviderConfig/my-db-...      -      -

```

We can see that both Database as well as the Object with the schema Managed Resources are Available, but also that the SQL Composite Resource still thinks that those three Managed Resources are being created (Creating).

We could solve that in a few ways, with one of them being by far the easiest. Since I'm lazy, we'll go with the easiest and fastest choice. We'll use yet another Composition Function.

Here's the Package manifest.

```
1 cat providers/function-auto-ready.yaml
```

The output is as follows.

```

1  ---
2  apiVersion: pkg.crossplane.io/v1beta1
3  kind: Function
4  metadata:
5    name: upbound-function-auto-ready
6  spec:
7    package: xpkg.upbound.io/crossplane-contrib/function-auto-ready:v0.2.1

```

This time we're adding the function-auto-ready Function which serves a very simple purpose. It automatically detects Composed Resources that are ready and updates their statuses. Since Functions in each step receive, among other things, all the Composed Resources assembled in previous steps, if we put that Function as the last step, it'll be able to evaluate all Composed Resources assembled thus far and update statuses. This is an example of how a Function can solve tasks that are not necessarily as typical as Go Templating.

Let's apply the Package,...


```
1 kubectl apply --filename providers/function-auto-ready.yaml
```

...and take a look at the modified version of the Composition.

```
1 cat compositions/sql-v11/$HYPERSCALER.yaml
```

The output is as follows (truncated for brevity).

```
1 ---
2 apiVersion: apiextensions.crossplane.io/v1
3 kind: Composition
4 ...
5 spec:
6   ...
7   pipeline:
8     ...
9   - functionRef:
10     name: upbound-function-auto-ready
11     step: automatically-detect-ready-composed-resources
```

That's an easy one. It (automatically-detect-ready-composed-resources) does not even define input. The function does not need any additional information except the list of all the Composed Resources assembled thus far. Every Function receives them no matter whether they require additional input.

Let's see whether it works by applying the new version of the Composition,...

```
1 kubectl apply --filename compositions/sql-v11/$HYPERSCALER.yaml
```

...and tracing the Claim.

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

The output is as follows (truncated for brevity).

	NAME	SYNCED	READY	STATUS
1	SQLClaim/my-db (a-team)	True	True	Available
2	└ SQL/my-db-rmwkj	True	True	Available
3	└─ ResourceGroup/my-db-...	True	True	Available
4	└─ FirewallRule/my-db-...	True	True	Available
5	└─ Server/my-db-...	True	True	Available
6	└─ Object/my-db-...-schema-db-01	True	True	Available
7	└─ ProviderConfig/my-db-...-sql	-	-	
8	└─ Database/my-db-...-db-01	True	True	Available
9	└─ Database/my-db-...-db-02	True	True	Available
10	└─ ProviderConfig/my-db-...	-	-	
11				

This time the status of the SQL Composite Resource is Available. The newly added Auto-Detect Function correctly propagated the status of the Managed Resources to the Composite Resource.

Here's what we did so far.

We deployed three Functions to the control plane cluster. One to perform **Patch and Transform** operations (1), one that executes **Go Templating** (2), and one, the **Auto-Ready** Function (3), that ensures the proper statuses of the resources.

Further on, we applied the Composition (4) set to run in the Pipeline mode, with four steps.

The first step uses the **Patch and Transform Function** to deal with resources that do not need "special" treatment beyond patching and basic transformations. Composite Resource (XR) (5), which already had running in the cluster, sent all the resources we specified in that step to the Function which returned those same resources **patched** (6).

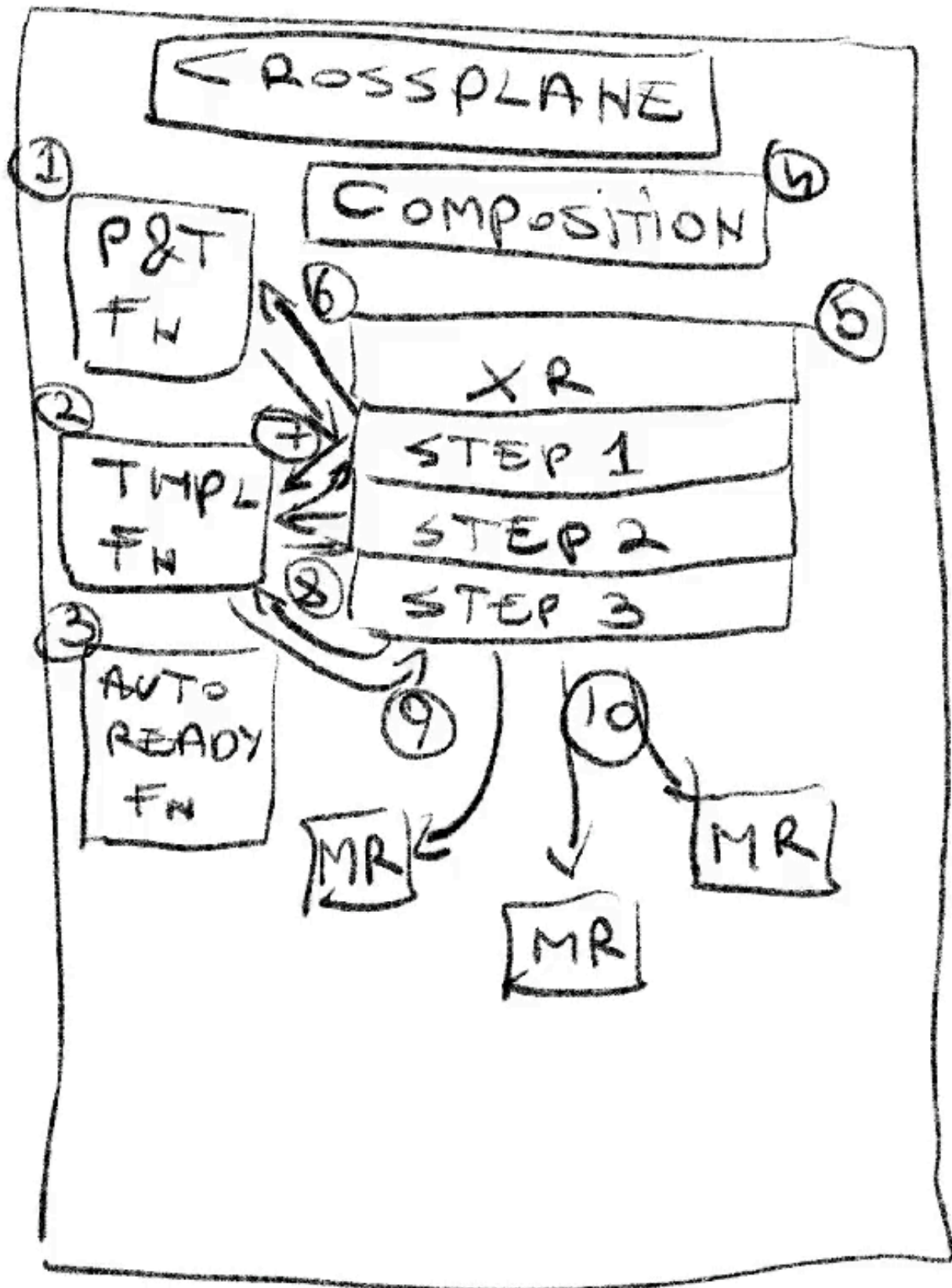
After that, the Composite Resource moved to the second step which uses the **Go Templating Function**. It sent to that Function the template that iterates over the list of Databases as well as all the **previously assembled resources**. The Function **added the Databases** to the list of resources and **returned them all back** to the Composite Resource (7).

Next, it went to the third step which also uses the **Patch and Transform Function**. It sent the **input** (the template) together with all the **previously assembled resources**, the Function **added the Schemas** to the list of resources and **returned them all back** to the Composite Resource (8).

The last step used the **Auto-Ready** Function which has **no input** so it only received the **previously assembled resources** and returned them **back with correct statuses** (9).

Once all the steps were executed, each modifying the state of resources, the Composite Resource (XR) created, updated or deleted Managed Resources (MRs) (10).

K.85



Almost all the heavy lifting was done by Functions with the Composite Resource acting as orchestrator of pipeline steps that invoke Functions by sending them inputs and expecting resources as outputs.

Building and Pushing Configuration Package

Now that we finished improving our Compositions and Composite Resource Definition, the time has come to package it all and push a new version to the registry so that others can easily use the improved version of the Database-as-a-Service we're building.

But, before we build the Package, we need to make a few changes to the Configuration we created in the previous chapter.

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

```
1 cat compositions/sql-v11/crossplane.yaml
```

The output is as follows (truncated for brevity).

```
1  apiVersion: meta.pkg.crossplane.io/v1
2  kind: Configuration
3  ...
4  spec:
5    ...
6    dependsOn:
7    ...
8    - provider: crossplane/provider-sql
9      version: ">=v0.5.0"
10   - function: xpkg.upbound.io/crossplane-contrib/function-patch-and-transform
11     version: ">=v0.1.4"
12   - function: xpkg.upbound.io/crossplane-contrib/function-go-templating
13     version: ">=v0.4.0"
14   - function: xpkg.upbound.io/crossplane-contrib/function-auto-ready
15     version: ">=v0.2.1"
16   ...
```

We used the Configuration to define the Providers that it depends on. As a result, those providers are deployed every time we deploy the Configuration Package. Those were defined as `provider` and we should do the same with Functions. They are also dependencies of the Configuration.

Adding Functions as dependencies is the same as Providers. The only difference is that we need to use `function` instead of `provider`. That's all there is to it, so let's build a new version of the Configuration Package.

I won't go into details of the steps that follow since they are the same as what we explored in the previous chapter. Instead, we'll move fast through what is already known.

We'll enter the directory with the Compositions, Composite Resource Definition, and Configuration,...

```
1 cd compositions/sql-v11
```

...and build the Configuration Package.

```
1 crossplane xpkg build
```

Next, we'll create UP_USER environment variable with the registry username,...

```
1 export UP_USER=[...]
```

...log in,...

```
1 crossplane xpkg login --username $UP_USER
```

...push the package,...

```
1 crossplane xpkg push xpkg.upbound.io/$UP_USER/dot-sql:v0.0.11
```

...remove it from the file system,...

```
1 rm dot-sql-*.xpkg
```

...and go back to the root of the repository.

```
1 cd ../../
```

That's it. The new release of the Configuration Package is in the registry and anyone can use it.

We improved the capabilities of our Database-as-a-Service without increasing the complexity for end-users beyond adding a few additional fields to the Composite Resource Definition. From the user's perspective, none of the things we did are relevant. All they have to do is continue creating, updating, and deleting Composite Claims. They need to deal with around 10 lines of YAML. The complexity increased, but it's still "hidden". The complexity is still an implementation detail, at least from the perspective of the end-users; those who create Claims.

We did not yet explore how to build our own Functions. All those we used so far are public. We acted as Function users, not as Function builders.

The day might come when none of the publicly available Functions fit our needs. When that day comes, we'll want to build our own Functions with our own logic. Yet, that day is not today.

This chapter is almost finished. All that's left is to destroy everything.

Destroy Everything

You know what to do.

Make the script executable,...

```
1  chmod +x destroy/04-functions.sh
```

...run it,...

```
1  ./destroy/04-functions.sh
```

TODO: Fast forward

...and exit Nix Shell.

```
1  exit
```

The End?

This is **not the end**. As Crossplane continues growing, I will continue adding chapters to the book. If you got it from LeanPub, you'll receive notifications to download the updated version of the book. The same should be true for eBook versions purchased from Amazon. If you got a “dead tree” (printed) version from Amazon, this is it, in this format. However, you can send me a message on [LinkedIn](#)¹ or [Twitter](#)² and I'll get back to you with a link to download the ebook from LeanPub for free (and get notifications when it's updated).

Finally, please subscribe to <https://youtube.com/@DevOpsToolkit>. You'll find a constant influx of videos on that channel, many of them related to Crossplane.

¹<https://linkedin.com/in/viktorfarcic>

²<https://twitter.com/vfarcic>