

GETTING STARTED WITH CROSSPLANE CONFIGURATION PACKAGES

**VIKTOR
FARCIC**



Configuration Packages

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

If you're curious to see some of the things Crossplane can do, [check out this blog](#)² or the intro of the paper. This is the third of the series, of which we covered [Providers and Managed Resources](#)³ and [Compositions](#)⁴ in the previous two. I recommend seeing all resources (or the full book) beforehand to give context to what we will cover in this paper.

Let's explore Crossplane Configuration Packages. We won't be talking about theory without touching the keyboard, so I'll keep this introduction short and jump straight into Crossplane Configuration Packages... right after we set up the environment we'll use in this paper. If you prefer a video version of this paper, [view my YouTube tutorial here](#)⁵.

In the previous paper, 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.

¹<https://www.upbound.io/resources/lp/book/crossplane-cloud-native-control-plane>

²<https://blog.upbound.io/why-choose-crossplane>

³<https://www.upbound.io/resources/lp/whitepaper-b/crossplane-providers-and-managed-resources-getting-started>

⁴<https://www.upbound.io/resources/lp/whitepaper-b/crossplane-compositions-getting-started>

⁵<https://youtu.be/ompdPvaHn0U?si=5HlQhqsoVaMLigZz>

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](#)⁶.

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,...

⁶<https://gist.github.com/vfarcic/3f4f9bf05c937b9f12e6bcb43f3c0bc7>

Configuration Packages

```
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.

Configuration Packages

```
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/)⁷.

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.

⁷<https://marketplace.upbound.io/>

Configuration Packages

```
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.io⁸, 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](#)⁹, 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
```

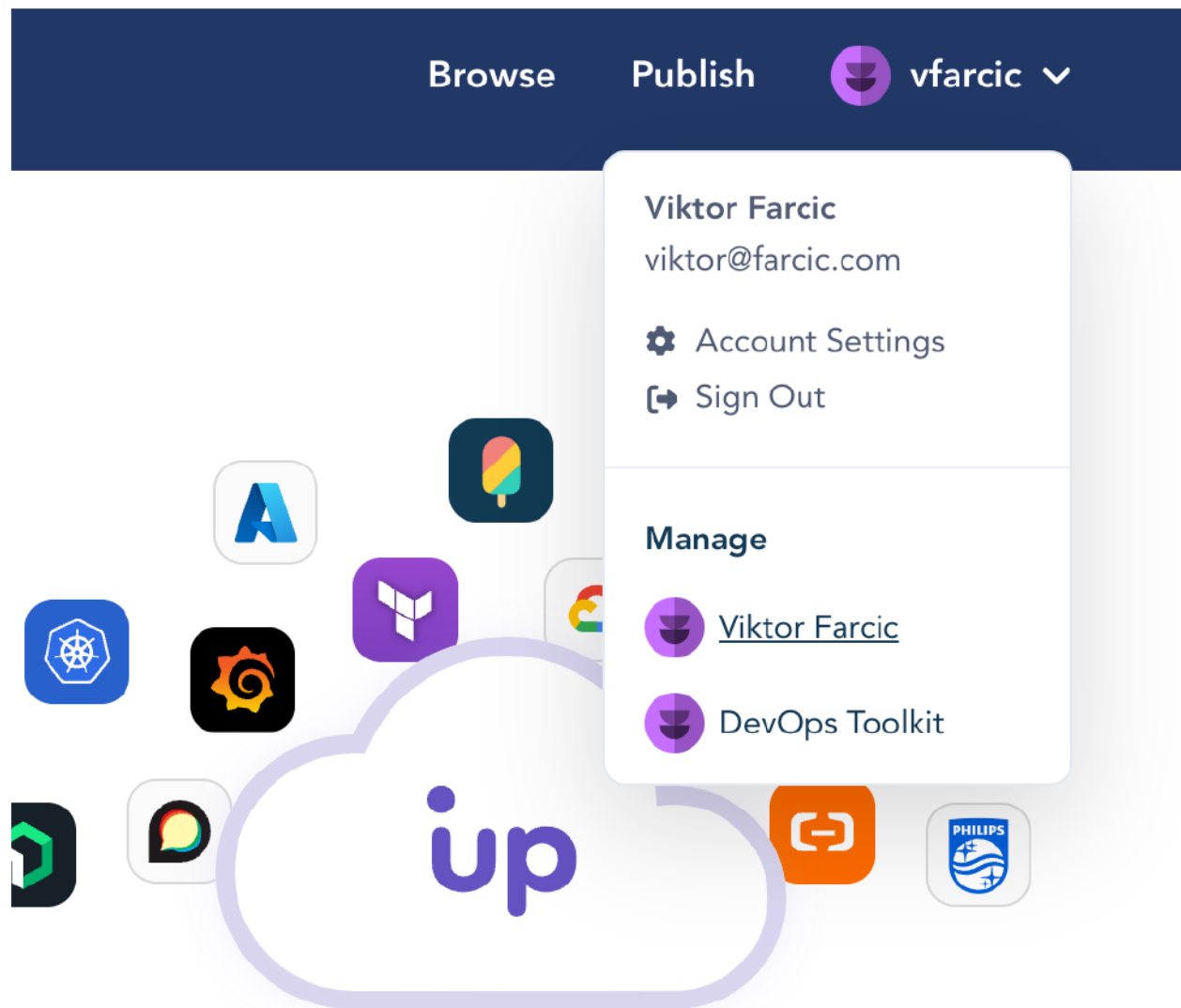
For those who do prefer “pretty colors”, we can see the newly created repository by opening the [marketplace](#)¹⁰ in a browser.

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.

⁸<https://accounts.upbound.io>

⁹<https://docs.upbound.io/reference/cli>

¹⁰<https://marketplace.upbound.io>



We can see the newly created repository.

upbound Marketplace

Vfarcic
User Settings ⚙️

Create Repository

Repository ↑
dot-sql PUBLIC


Visibility
Newest Version


⋮

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

upbound Marketplace

Browse Publish vfarcic ▾

vfarcic /
 **dot-sql**



No Packages

Only Repository Administrators can push a package to this repository.

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.

Configuration Packages

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.

vfaric /

 **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,...

Configuration Packages

```
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.

Configuration Packages

```
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.

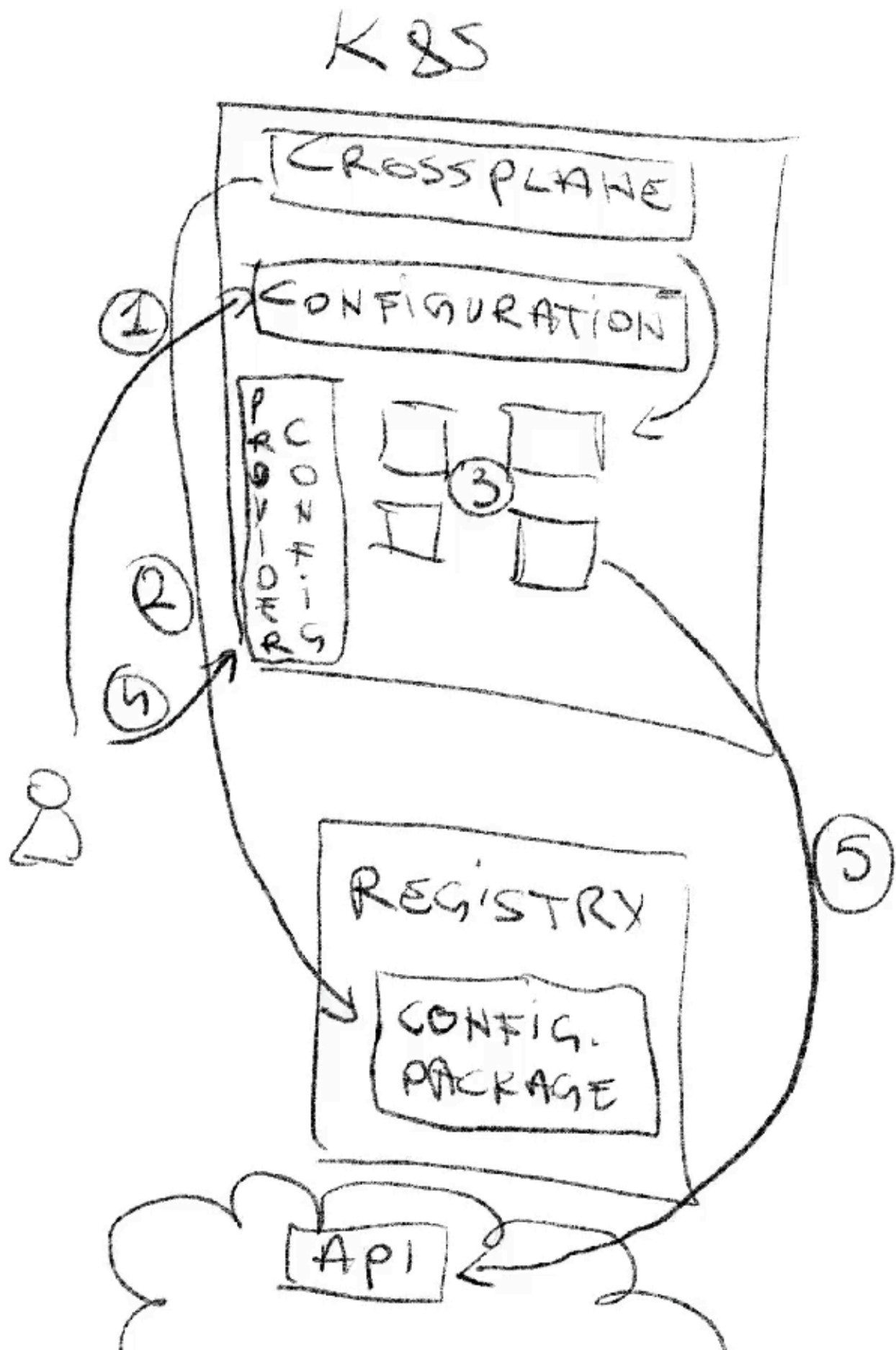
```
1 kubectl apply --filename providers/$HYPERSCALER-config.yaml
```

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.

Conclusion

To take the next step in your `Crossplane` journey to learn about `Composition Functions`, check out my full book, [Crossplane: the Cloud Native Control Plane](https://www.upbound.io/resources/lp/book/crossplane-cloud-native-control-plane)¹¹. It'll give you the whole picture on how to implement `Crossplane` and more!

¹¹<https://www.upbound.io/resources/lp/book/crossplane-cloud-native-control-plane>