

GETTING STARTED

WITH CROSSPLANE PROVIDERS AND MANAGED RESOURCES

VIKTOR FARCIC



Providers and Managed Resources

This paper is the first chapter in the larger book, [Crossplane: the Cloud Native Control Plane](https://www.upbound.io/resources/lp/book/crossplane-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](https://blog.upbound.io/why-choose-crossplane)² or the intro of the paper. Either will give context to what we will cover in this paper. Now, let's dive into Crossplane implementation by going back to the very beginning and exploring some of the basics.

Let's explore Crossplane providers and managed resources. 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 paper. If you prefer a video version of this paper, view my [YouTube tutorial here](https://www.youtube.com/watch?v=o53_7vuWjw4)³. Otherwise, let's get started.

Chapter Setup

To run setup scripts as well as the instructions that follow in the hands-on parts of this paper, we'll need tools. We'll need quite a few CLIs like, for example, kubectl, 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. 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](https://docs.docker.com/engine/install/)⁴.

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

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

³https://www.youtube.com/watch?v=o53_7vuWjw4

⁴<https://docs.docker.com/engine/install/>

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](#)⁴ if you do not have it already.

You can find additional information about GitHub CLI in the [GitHub CLI \(gh\) - How to manage repositories](#)⁵ more efficiently video.

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

To see the full instructions on the full setup for each paper in this series, check out the full version of this paper, my book, [Crossplane: the Cloud Native Control Plane](#)⁶.

Back to exploring Crossplane providers and managed resources. All the commands used in this chapter are available in the [Gist](#)⁷. Run the associated script there.

```
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.

⁴<https://github.com/cli/cli?tab=readme-ov-file#installation>

⁵<https://www.youtube.com/watchv=BII6ZY2Rnlc&feature=youtu.be>

⁶<https://www.upbound.io/resources/lp/book/crossplane-cloud-native-control-plane>

⁷<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)⁸ 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.

⁸<https://marketplace.upbound.io>

Providers and Managed Resources

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

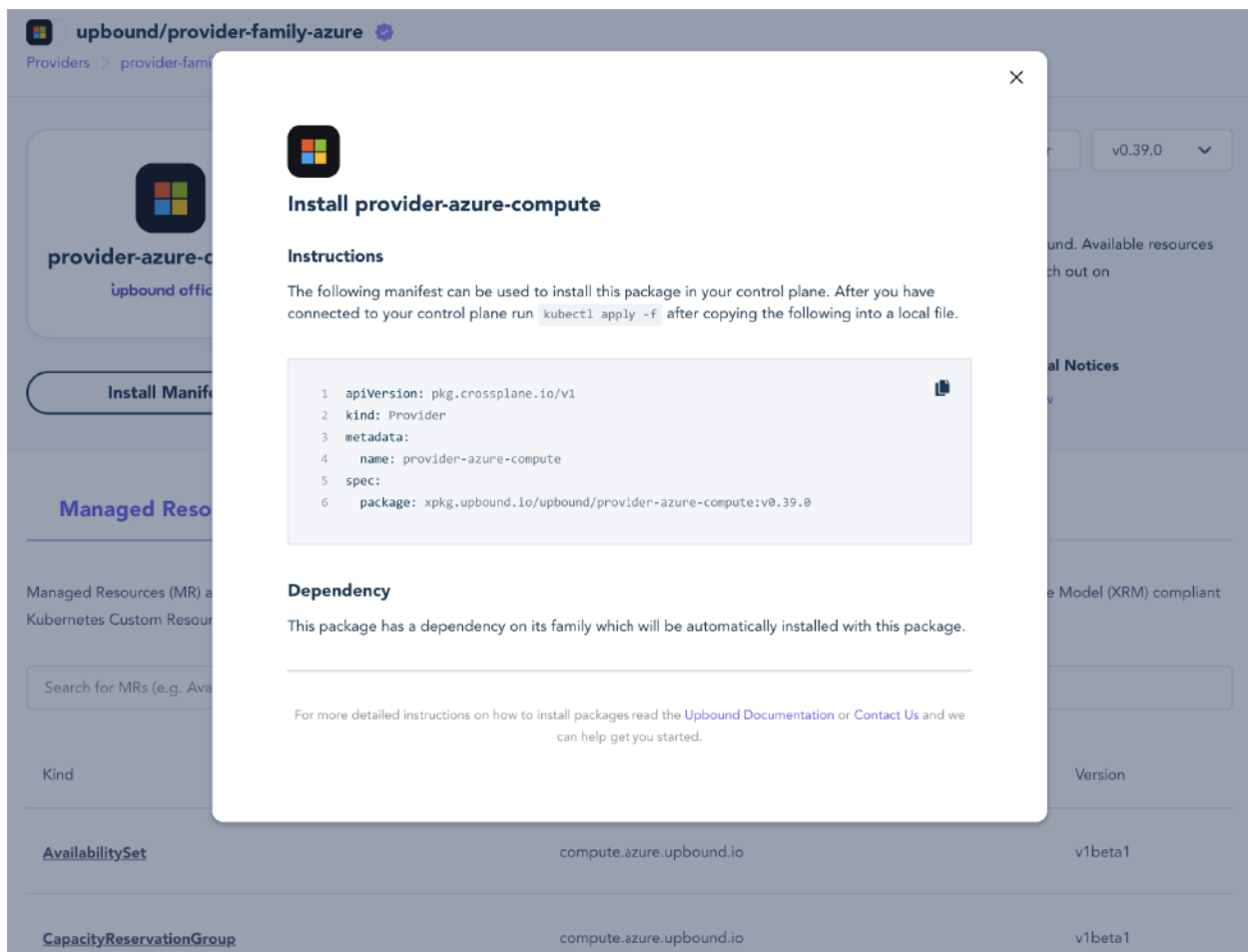
ankasoftco

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.

Providers and Managed Resources



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.

Providers and Managed Resources

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.

Providers and Managed Resources

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

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.

Providers and Managed Resources

```
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.

Providers and Managed Resources

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 (7f9f9b08-7d00-43c9-9d30-f10bb79e9a61). 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`.

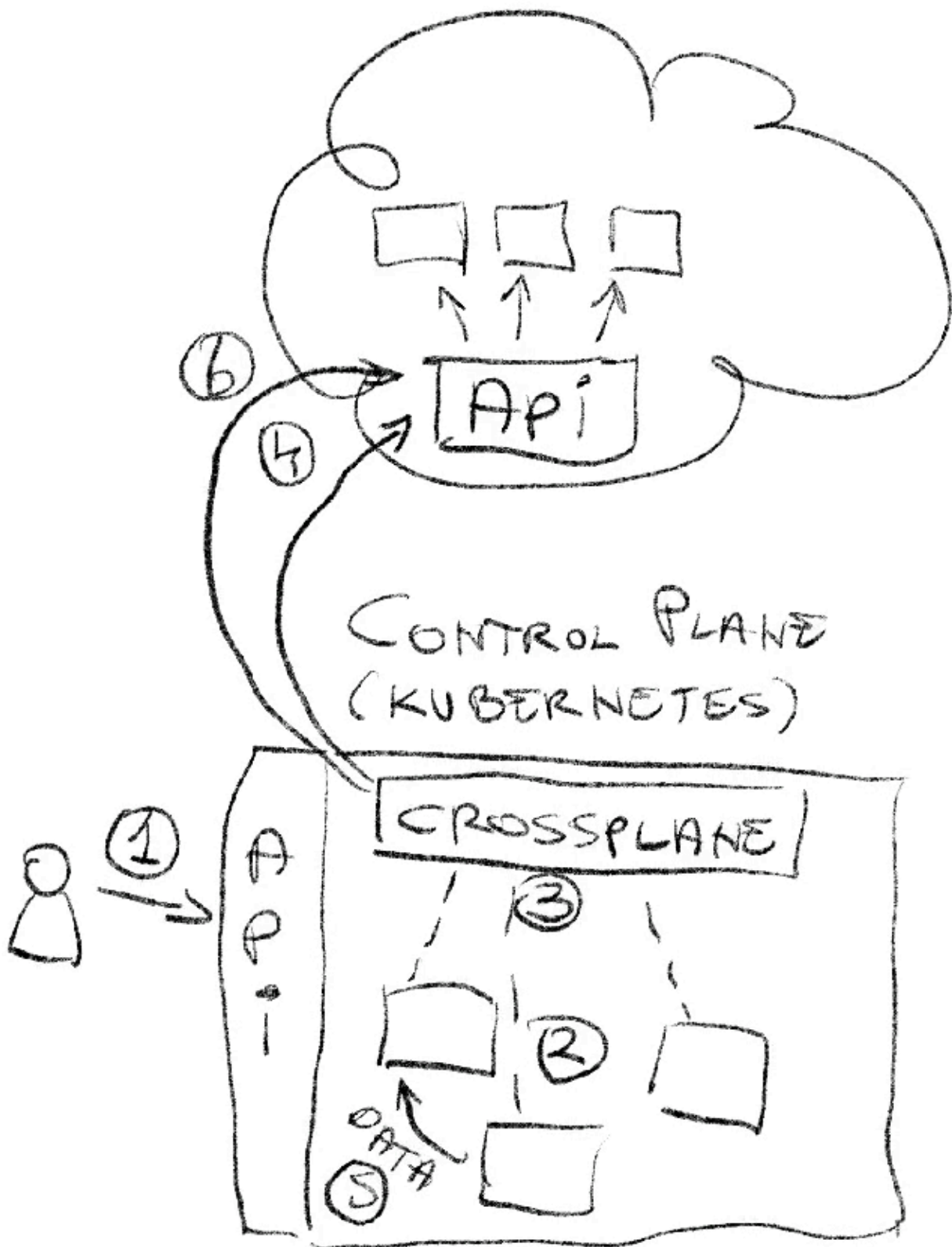
```
1 kubectl get managed
```

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 ↑↓



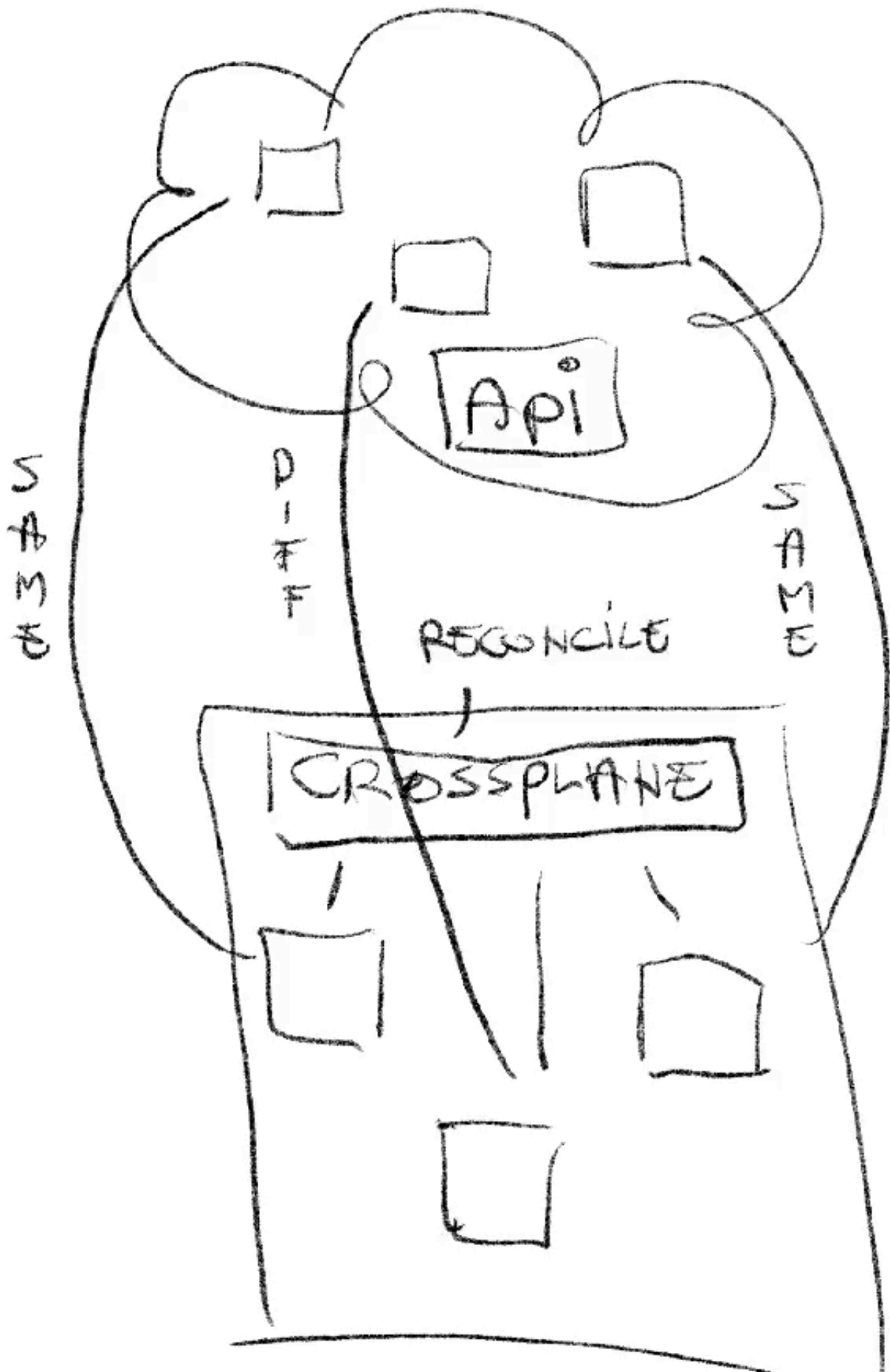
No virtual machines to display

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
```

Conclusion

To take the next step in your Crossplane journey to learn about Compositions, check out my full book, [Crossplane: the Cloud Native Control Plane](https://www.upbound.io/resources/lp/book/crossplane-cloud-native-control-plane)⁹. It’ll give you the whole picture on how to implement Crossplane and more!

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

Providers and Managed Resources

This paper is the first chapter in the larger book, [Crossplane: the Cloud Native Control Plane](https://www.upbound.io/resources/lp/book/crossplane-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](https://blog.upbound.io/why-choose-crossplane)² or the intro of the paper. Either will give context to what we will cover in this paper. Now, let's dive into Crossplane implementation by going back to the very beginning and exploring some of the basics.

Let's explore Crossplane providers and managed resources. 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 paper. If you prefer a video version of this paper, view my [YouTube tutorial here](https://www.youtube.com/watch?v=o53_7vuWjw4)³. Otherwise, let's get started.

Chapter Setup

To run setup scripts as well as the instructions that follow in the hands-on parts of this paper, we'll need tools. We'll need quite a few CLIs like, for example, kubectl, 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. 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](https://docs.docker.com/engine/install/)⁴.

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

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

³https://www.youtube.com/watch?v=o53_7vuWjw4

⁴<https://docs.docker.com/engine/install/>

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](#)⁴ if you do not have it already.

You can find additional information about GitHub CLI in the [GitHub CLI \(gh\) - How to manage repositories](#)⁵ more efficiently video.

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

To see the full instructions on the full setup for each paper in this series, check out the full version of this paper, my book, [Crossplane: the Cloud Native Control Plane](#)⁶.

Back to exploring Crossplane providers and managed resources. All the commands used in this chapter are available in the [Gist](#)⁷. Run the associated script there.

```
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.

⁴<https://github.com/cli/cli?tab=readme-ov-file#installation>

⁵<https://www.youtube.com/watchv=BII6ZY2Rnlc&feature=youtu.be>

⁶<https://www.upbound.io/resources/lp/book/crossplane-cloud-native-control-plane>

⁷<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)⁸ 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.

⁸<https://marketplace.upbound.io>

Providers and Managed Resources

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

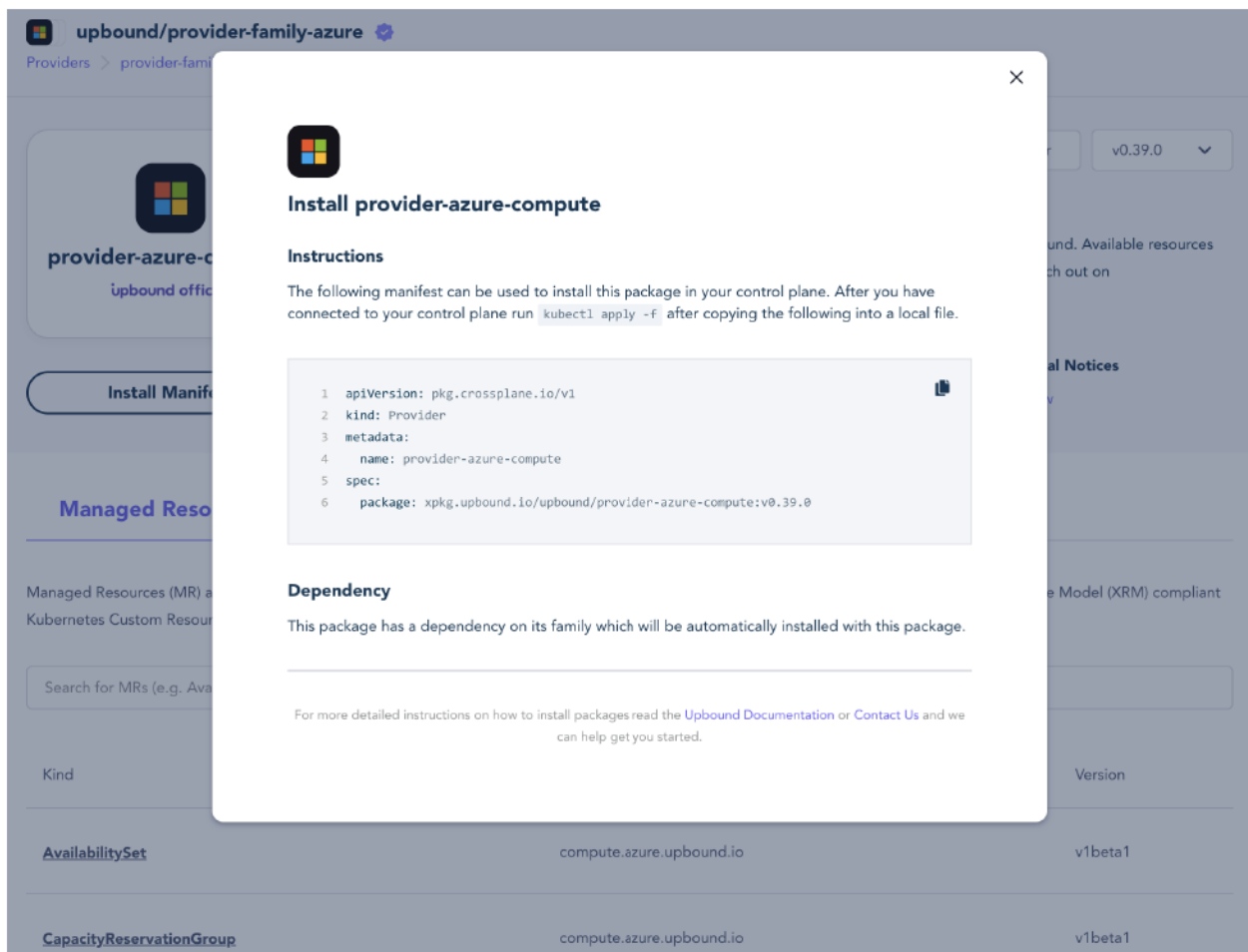
ankasoftco

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.

Providers and Managed Resources



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.

Providers and Managed Resources

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.

Providers and Managed Resources

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

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.

Providers and Managed Resources

```
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.

Providers and Managed Resources

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 (7f9f9b08-7d00-43c9-9d30-f10bb79e9a61). 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`.

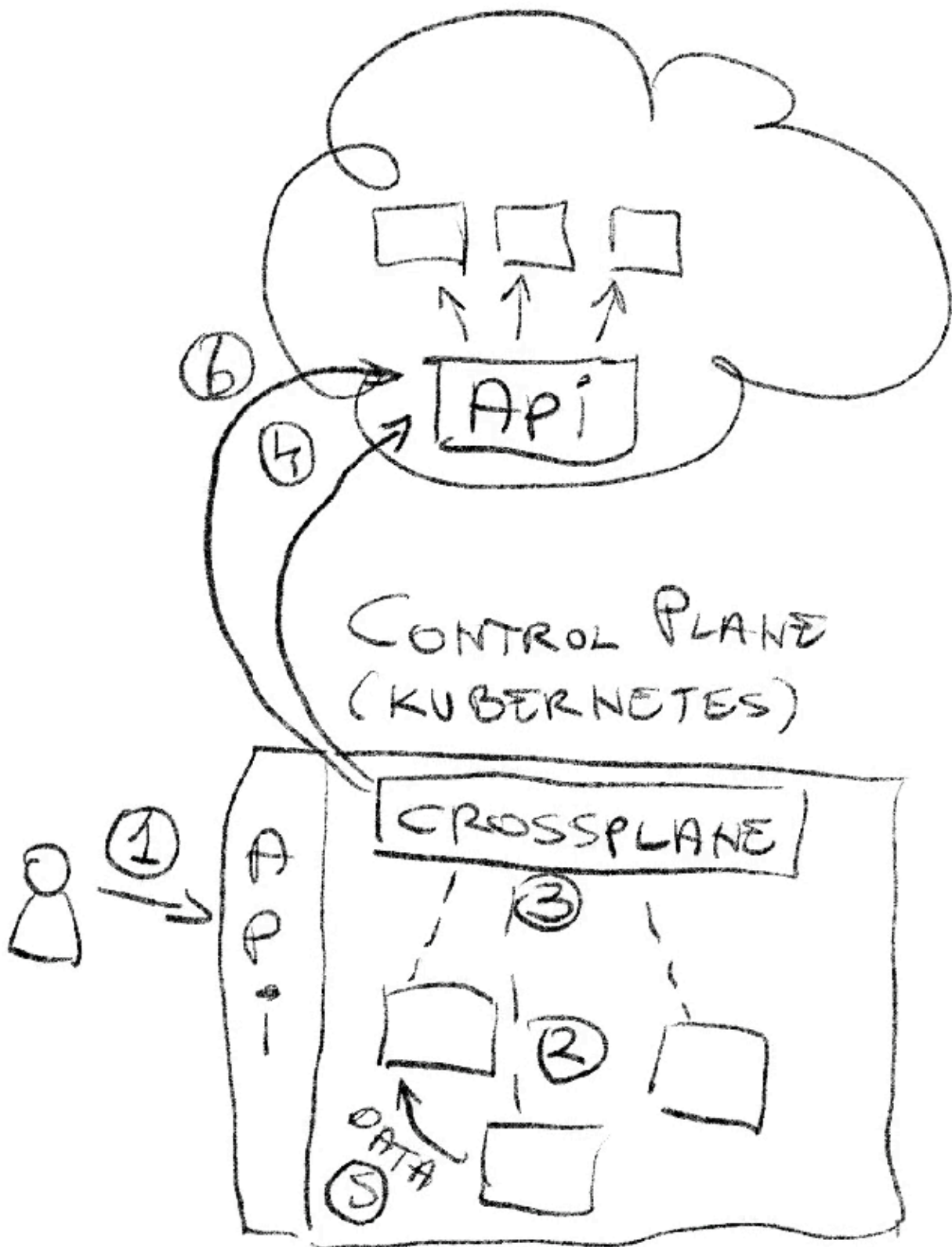
```
1 kubectl get managed
```

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 ↑↓



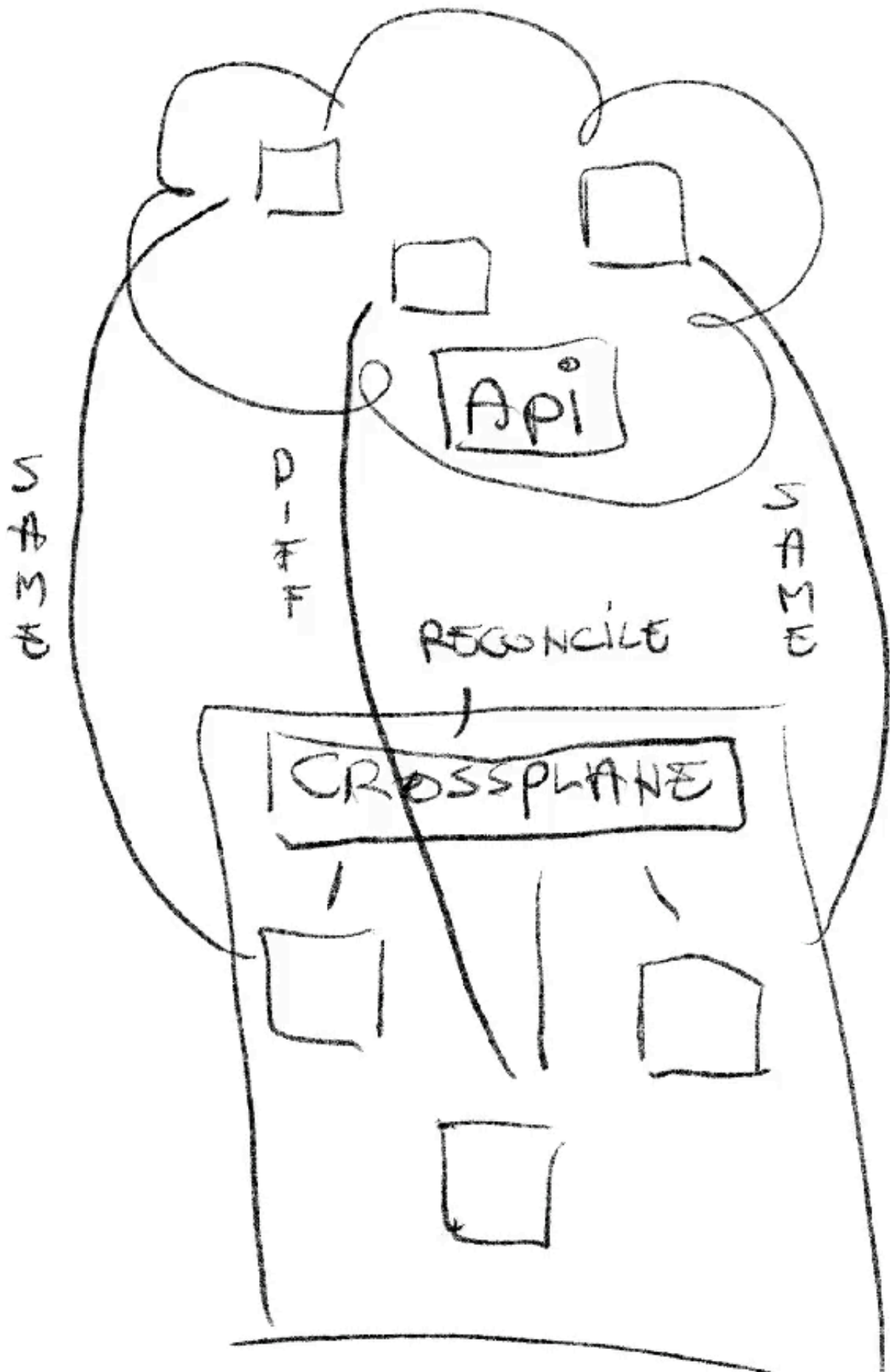
No virtual machines to display

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
```

Conclusion

To take the next step in your Crossplane journey to learn about Compositions, check out my full book, [Crossplane: the Cloud Native Control Plane](https://www.upbound.io/resources/lp/book/crossplane-cloud-native-control-plane)⁹. It’ll give you the whole picture on how to implement Crossplane and more!

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