



KubeCon



CloudNativeCon

North America 2022

BUILDING FOR THE ROAD AHEAD

DETROIT 2022

So you want to develop a Cluster API provider?

*Richard Case, Anusha Hegde, Winnie Kwon,
Avishay Traeger*

So you want to develop a Cluster API provider?



KubeCon



CloudNativeCon

North America 2022

BUILDING FOR THE ROAD AHEAD

DETROIT 2022

October 24-28, 2021



Anusha Hegde

Technical Product Manager

Nirmata



Richard Case

Principal Engineer

SUSE



Winnie Kwon

Engineering Manager

VMware



Avishay Traeger

Senior Principal Software
Engineer

Red Hat



BUILDING FOR THE ROAD AHEAD

DETROIT 2022

What we will be learning

- What is Cluster API?
- Cluster API Provider Theory
- Hands-on: building a provider

NOTE: this is a hands-on session where you will be building a provider. Time will be given for each section during which the speakers can help

Accompanying Docs



<https://capi-samples.github.io/kubecon-na-2022-tutorial/>

Please complete the prerequisites

Discuss now and after the session:

CNCF Slack #kubekon-na-2022-capi-provider-tutorial

What is Cluster API?

- Declarative specification of clusters
- Built on the premise that “Cluster lifecycle management is difficult”
- Designed around interchangeable components via “providers”
- **clusterctl** handles the lifecycle of a CAPI management cluster
- Community calls every week on Wednesday @ 6pm GMT / 10am PT
- For a walkthrough of CAPI see the “lets talk about...” series by Stefan & Fabrizio:
 - <https://github.com/kubernetes-sigs/cluster-api/discussions/6106>

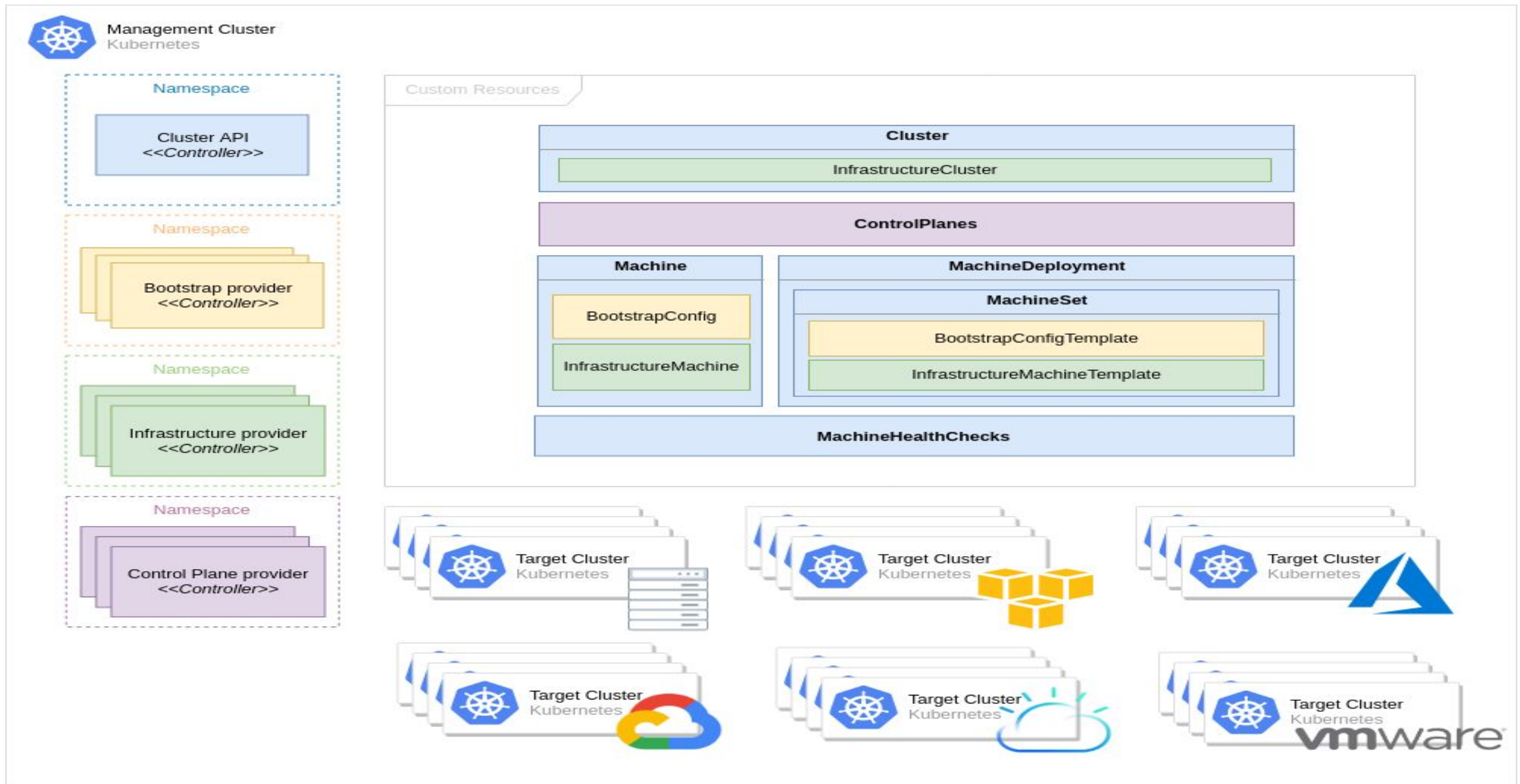
What is CAPI?

- The [Cluster API](#) (CAPI) brings declarative, Kubernetes-style APIs to cluster creation, configuration and management.
- Establishes building blocks for higher order functionality:
 - Cluster templating
 - Automation of scaling, repair & upgrades
 - Distributing nodes across failure domains
 - Machine health checks to replace unhealthy nodes
 - Managed control planes
- Many infrastructure vendors maintain CAPI providers
 - You can see the complete list [here](#)

What is CAPI?

- Designed around interchangeable components via “providers”
 - Infrastructure (i.e. vpc, machines, security groups.....)
 - Bootstrap
 - Control plane
- **clusterctl** handles the lifecycle of a CAPI management cluster
 - Cluster templates / flavors are very useful
 - Providers operator being developed.....GitOps friendly provider operations :)
- Kubernetes sub-project under SIG Cluster Lifecycle
 - Community calls every week on Wednesday @ 6pm GMT / 10am PT
- For a walkthrough of CAPI see the “lets talk about...” series by Stefan:
 - <https://github.com/kubernetes-sigs/cluster-api/discussions/6106>

CAPI in a nutshell



What is a Cluster API provider?

A Kubernetes **operator** that implements infrastructure / operating environment specific functionality that is utilized by core Cluster API when managing the lifecycle of a K8s cluster.

The operator implements a contract via its custom resources (i.e. CRDs) depending on the type of provider, which enables interaction between core CAPI and the provider.

CAPI Concepts

- Useful definitions can be found in [The Cluster API Book](#)
- Management cluster vs. workload cluster
- Control plane
- Machine
- MachineDeployment / MachineSet
- MachineHealthCheck

Provider types

- **Infrastructure** - used to provision any infrastructure that is required to create and run a Kubernetes cluster. For example, networking, security groups, virtual or physical host machines
- **Bootstrap** - used to create the “user-data” that is passed to the infrastructure machines that contains the instructions to bootstrap a Kubernetes node on that machine. 2 parts to it:
 - Action: how Kubernetes is bootstrapped (e.g. invoking kubeadm)
 - Format: how the action is encoded and passed to the machine (e.g. cloud-init, ignition)
- **Control plane** - used to control the creation & lifecycle of the Kubernetes control plane. It can utilize resources created by bootstrap and infrastructure providers.
 - Kubeadm control plane is the original
 - Managed Kubernetes (i.e. EKS, AKS) implementations - no nodes


CAPI CRDs

- CAPI defines several CRDs that are used for all providers
- In general, each of these refers to one or more implementation-specific CRDs

```
kind: DockerCluster
apiVersion:
  infrastructure.cluster.x-k8s.io/v1beta1
metadata:
  name: my-cluster-docker
---
kind: Cluster
apiVersion: cluster.x-k8s.io/v1beta1
metadata:
  name: my-cluster
spec:
  infrastructureRef:
    kind: DockerCluster
    apiVersion:
      infrastructure.cluster.x-k8s.io/v1beta1
    name: my-cluster-docker
```



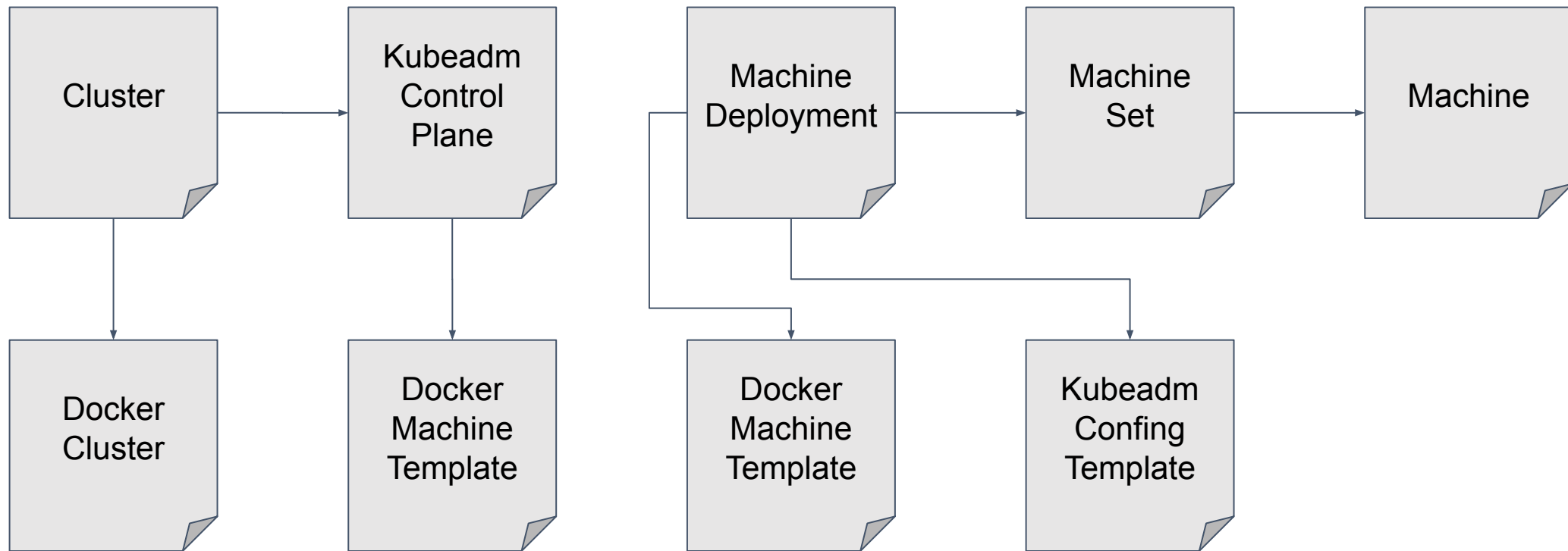
```
kind: KubeadmConfig
apiVersion:
  bootstrap.cluster.x-k8s.io/v1beta1
metadata:
  name: my-control-plane1-config
spec:
  initConfiguration:
    nodeRegistration:
      kubeletExtraArgs:
        eviction-hard:
          nodefs.available<0%,nodefs.inodesFree<0%,ima
          gefs.available<0%
  clusterConfiguration:
    controllerManager:
      extraArgs:
        enable-hostpath-provisioner: "true"
---
kind: DockerMachine
apiVersion:
  infrastructure.cluster.x-k8s.io/v1beta1
metadata:
  name: my-control-plane1-docker
```



```
kind: Machine
apiVersion: cluster.x-k8s.io/v1beta1
metadata:
  name: my-control-plane1
  labels:
    cluster.x-k8s.io/cluster-name:
      my-cluster
    cluster.x-k8s.io/control-plane: "true"
  set: controlplane
spec:
  bootstrap:
    configRef:
      kind: KubeadmConfig
      apiVersion:
        bootstrap.cluster.x-k8s.io/v1beta1
      name: my-control-plane1-config
  infrastructureRef:
    kind: DockerMachine
    apiVersion:
      infrastructure.cluster.x-k8s.io/v1beta1
    name: my-control-plane1-docker
    version: "v1.19.1"
```

CAPI CRDs

- CAPI defines several CRDs that are used for all providers
- Some of these CRDs refer to one or more implementation-specific CRDs



Cluster

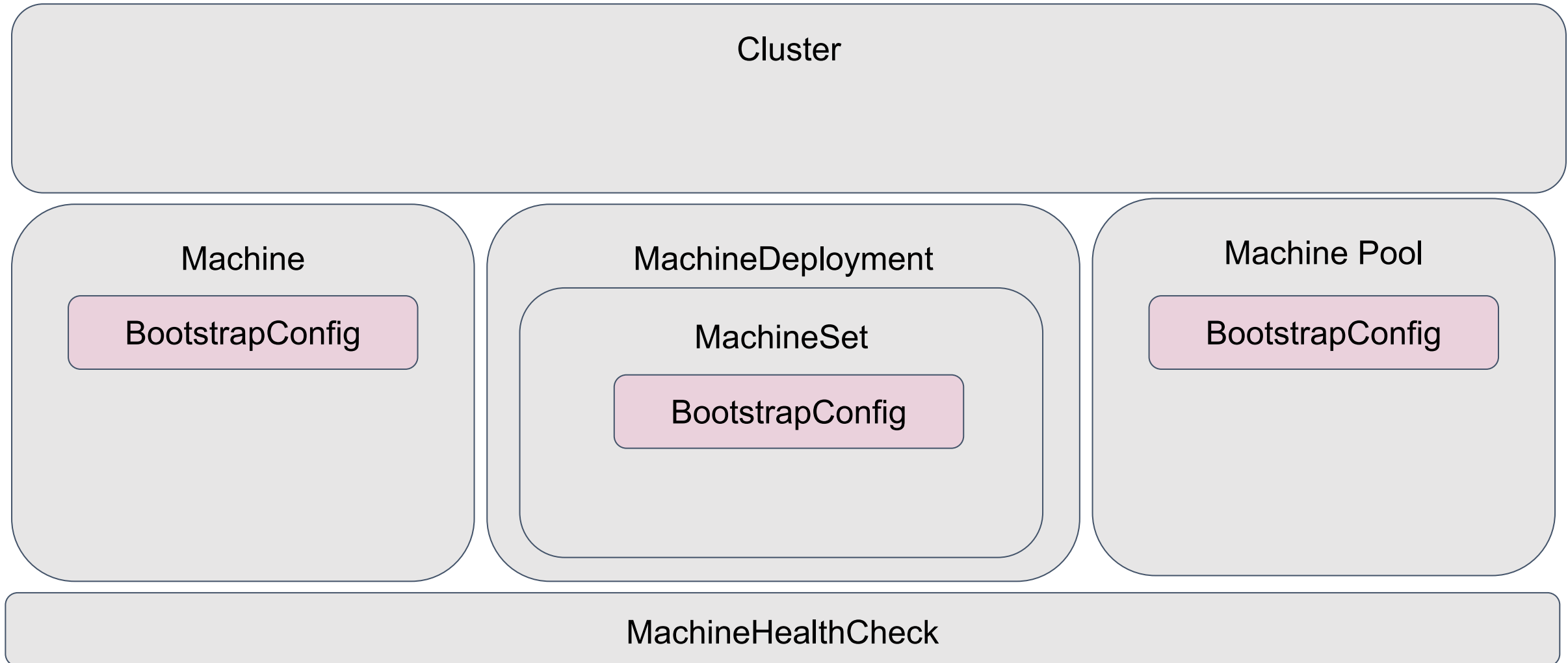
Machine

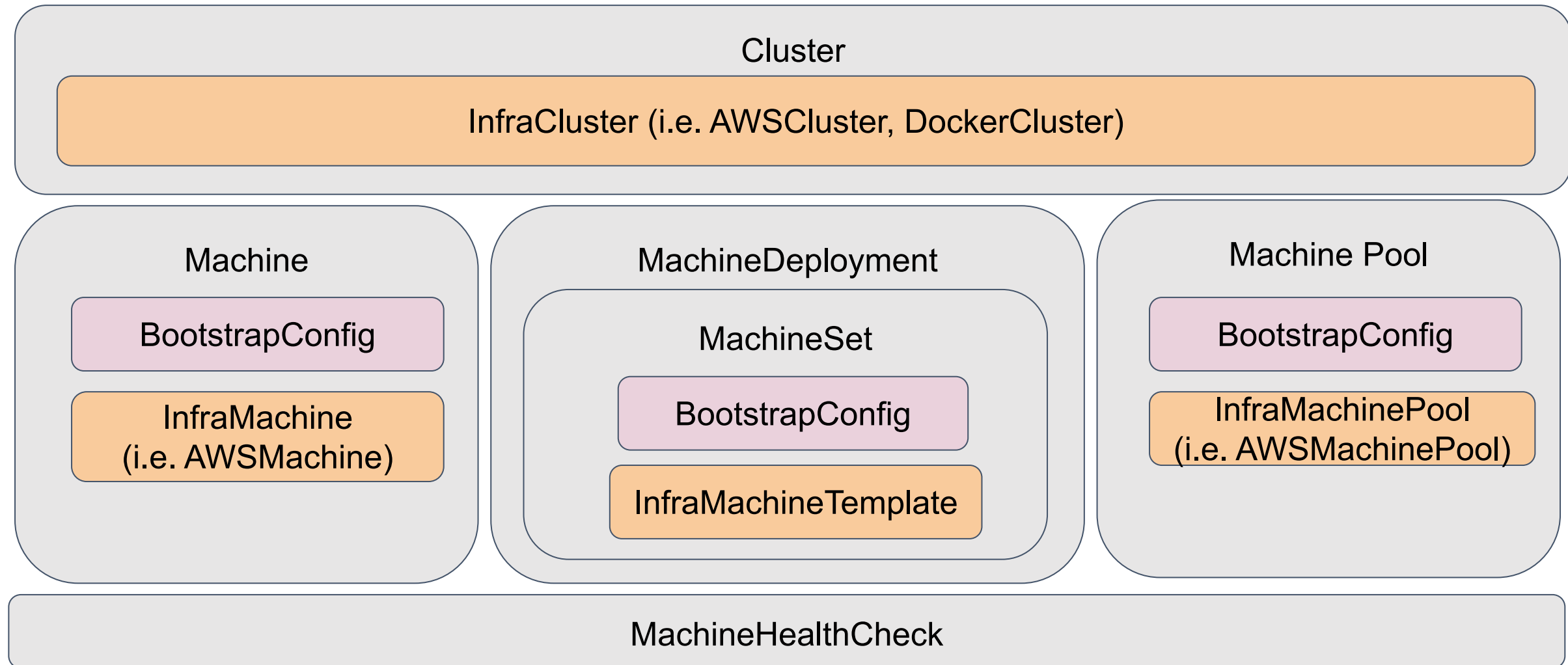
MachineDeployment

Machine Pool

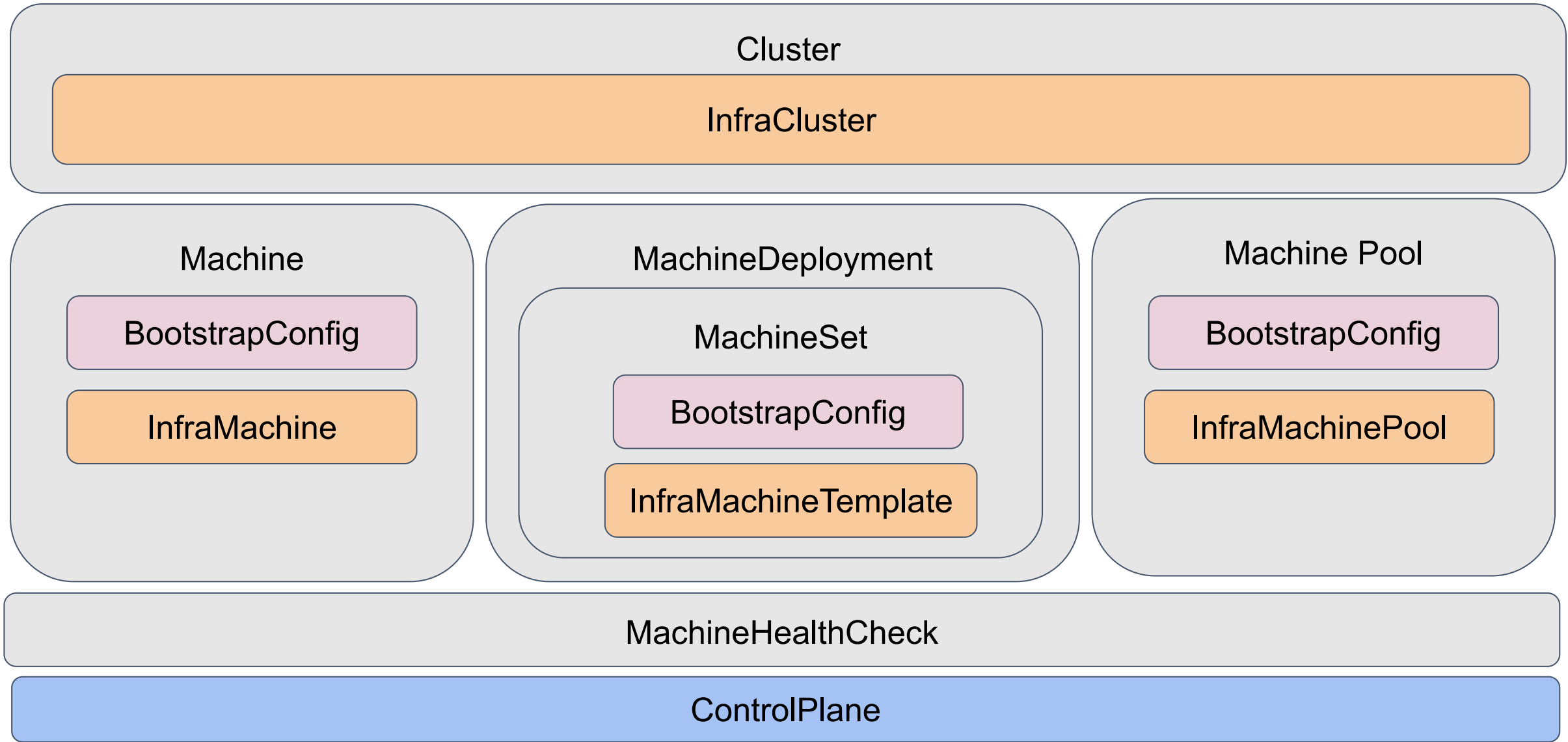
MachineSet

MachineHealthCheck





Control Plane



What makes up a provider?

- CRD
 - Adhere to contract (i.e. specific fields)
 - Contract is specific to the provider type
- Controller that reconciles the CRD
- k8s resources to deploy the controller
- Metadata / repo layout

Hands-on

Tutorial Scenario

- We will build an **infrastructure provider** for **Docker**
- Simplified version of the existing provider (a.k.a CAPD)
- Infrastructure that we will provision:
 - Loadbalancer container
 - Machine containers

Tutorial Format

- You will be working through the tutorial docs
 - Go at your own pace
 - Don't panic
- We will give a period of time for section
 - Overview
- If you get stuck raise your hand
 - We are here to help

Accompanying Docs



<https://capi-samples.github.io/kubecon-na-2022-tutorial/>

Please complete the prerequisites

Tutorial Overview

1. Pre-reqs
2. Setup
3. Kubebuilder scaffolding
4. Setting up tilt
5. **DockerCluster** implementation
6. **DockerMachine** implementation
7. Webhooks
8. Create a Cluster
9. Releasing

**We are here to help...so please ask
questions and raise your hands if you get
stuck**





Please scan the QR Code above to
leave feedback on this session

What is Cluster API?

- Declarative specification of clusters
- Built on the premise that “Cluster lifecycle management is difficult”
- Designed around interchangeable components via “providers”
- **clusterctl** handles the lifecycle of a CAPI management cluster
- Community calls every week on Wednesday @ 6pm GMT / 10am PT
- For a walkthrough of CAPI see the “lets talk about...” series by Stefan & Fabrizio:
 - <https://github.com/kubernetes-sigs/cluster-api/discussions/6106>

What is a Cluster API provider?

A Kubernetes **operator** that implements infrastructure / operating environment specific functionality that is utilized by core Cluster API when managing the lifecycle of a K8s cluster.

The operator implements a contract via its custom resources (i.e. CRDs) depending on the type of provider, which enables interaction between core CAPI and the provider.

- **Infrastructure** - used to provision any infrastructure that is required to create and run a Kubernetes cluster. For example, networking, security groups, virtual or physical host machines
- **Bootstrap** - used to create the “user-data” that is passed to the infrastructure machines that contains the instructions to bootstrap a Kubernetes node on that machine. 2 parts to it:
 - Action: how Kubernetes is bootstrapped (e.g. invoking kubeadm)
 - Format: how the action is encoded and passed to the machine (e.g. cloud-init, ignition)
- **Control plane** - used to control the creation & lifecycle of the Kubernetes control plane. It can utilize resources created by bootstrap and infrastructure providers.
 - Kubeadm control plane is the original
 - Managed Kubernetes (i.e. EKS, AKS) implementations - no nodes

What constitutes a Cluster API provider?

- A Kubernetes operator (a.k.a controller manager)
 - CRDs
 - Controllers that reconcile the CRDs
- k8s resources to deploy the controller
 - Plain old yaml
 - (Optional) tokens that will be replaced an installation time
 - Kustomize
- Metadata / repo layout

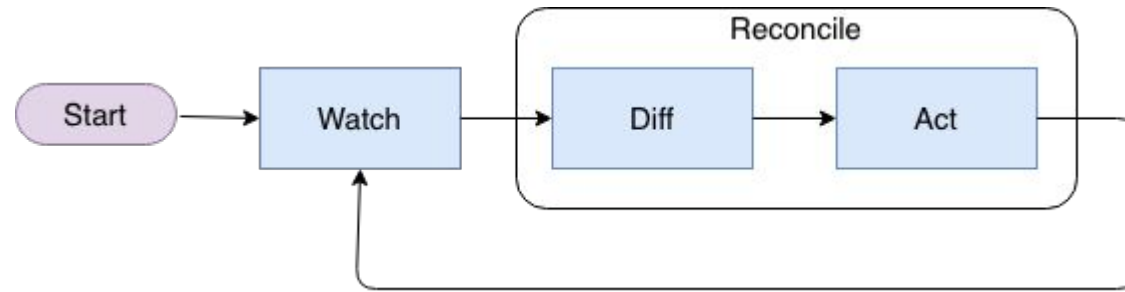
What is an operator?

- A way to create, manage and configure complex applications in Kubernetes.
- Codifies the steps a human would do to deploy and operate a complex application
 - For example, the steps to create Kubernetes cluster involved creating infrastructure, bootstrapping k8s, managing version upgrades.
- Surface to user via declarative API (i.e. CRDs)
- Contains one or more controllers that understand & reconcile the CRDs

What is a controller?

“ A controller is a control loop that watches the desired state of the cluster through the API server and makes changes attempting to move the current state towards the desired state. “

Control Loop



- Watch - for changes in custom resource(s)
- Diff - work out the difference between desired & actual state
- Act - take action to remediate the difference (if any)

.....And repeat!

Implement controllers for your API types (2/2)

For **Reconcile** the following pattern is generally used:

1. Get the instance of the API type being reconciled
2. Get the owning CAPI type (i.e. if we reconciling **PodmanMachine** then get **Machine**)
3. If we don't have the **Machine** then exit (owner reference isn't set yet)
4. Optionally, get the owning **Cluster** and Infra Cluster (i.e. **PodManCluster**)
5. If instance has a deletion timestamp, then in **reconcileDelete**:
 - a. do any actions to delete
 - b. remove finalizer and save
6. If instance has no deletion timestamp, then in **reconcileNormal**:
 - a. Add finalizer to instance and save
 - b. do any actions to create OR update

When building a provider you should set ownerReference

- A link to a resource that is the owner
 - Example: Deployment owns Pods
 - Example: Cluster owns PodmanCluster
- Used heavily in Cluster API
- Implemented via the **metadata.ownerReference** field
- If the owner is deleted then either:
 - Cascading deletion (controlled via policy) - this what Cluster API uses.
 - Orphaned resources



```
kubectl delete cluster my-dev-cluster
```


Local testing / development (1/2)

- Developing and debugging operators in Kubernetes can be painful.
- Tilt will save you a lot of time, pain and tears!
 - We need to tell Tilt about our provider via the **tilt-provider.json** file in repo root

```
[
  {
    "name": "podman",
    "config": {
      "image": "ghcr.io/capi-samples/cluster-api-provider-podman:dev",
      "live_reload_deps": [
        "main.go",
        "go.mod",
        "go.sum",
        "api",
        "controllers",
        "pkg"
      ],
      "label": "CAPP0D"
    }
  }
]
```

Local testing / development (2/2)

- We can then follow the instructions from the CAPI docs to configure Tilt:
 - <https://cluster-api.sigs.k8s.io/developer/tilt.html>

```
{
  "default_registry": "gcr.io/capi-samples",
  "provider_repos": ["../github.com/capi-samples/cluster-api-provider-podman"],
  "enable_providers": ["podman", "kubeadm-bootstrap", "kubeadm-control-plane"],
  "kustomize_substitutions": {
    "EXP_CLUSTER_RESOURCE_SET": "true",
  },
  "extra_args": {
    "podman": ["--v=4"],
    "kubeadm-control-plane": ["--v=4"],
    "kubeadm-bootstrap": ["--v=4"],
    "core": ["--v=4"]
  },
  "debug": {
    "podman": {
      "continue": true,
      "port": 30000
    }
  }
}
```

- Unit and integration tests...its up to the provider which approach/frameworks to use
- Envtest is often used for unit and integration tests
 - Part of the controller runtime
 - Interact with your provider as if its in a real cluster

```
testEnv = &envtest.Environment{
    CRDDirectoryPaths: []string{
        filepath.Join("../", "../", "config", "crd", "bases"),
        filepath.Join(build.Default.GOPATH, "pkg", "mod", "sigs.k8s.io", "cluster-api@v1.1.3",
"config", "crd", "bases"),
    },
    ErrorIfCRDPathMissing: true,
}

var err error
cfg, err = testEnv.Start()
Expect(err).ToNot(HaveOccurred())
Expect(cfg).ToNot(BeNil())
```

```
func TestAWSMachinePoolConversion(t *testing.T) {
    g := NewWithT(t)
    ns, err := testEnv.CreateNamespace(ctx, fmt.Sprintf("conversion-webhook-%s", util.RandomString(5)))
    g.Expect(err).ToNot(HaveOccurred())
    machinepool := &AWSMachinePool{
        ObjectMeta: metav1.ObjectMeta{
            Name:      fmt.Sprintf("test-machinepool-%s", util.RandomString(5)),
            Namespace: ns.Name,
        },
        Spec: AWSMachinePoolSpec{
            MinSize: 1,
            MaxSize: 3,
        },
    }

    g.Expect(testEnv.Create(ctx, machinepool)).To(Succeed())
    defer func(do ...client.Object) {
        g.Expect(testEnv.Cleanup(ctx, do...)).To(Succeed())
    }(ns, machinepool)
}
```

- CAPI provides e2e framework - most of the code is reusable

```
import (
    "sigs.k8s.io/cluster-api/test/framework"
)

cluster := framework.GetClusterByName(ctx, framework.GetClusterByNameInput{
    Getter:    e2eCtx.Environment.BootstrapClusterProxy.GetClient(),
    Namespace: namespace.Name,
    Name:      clusterName,
})
Expect(cluster).NotTo(BeNil(), "couldn't find cluster")

framework.DeleteCluster(ctx, framework.DeleteClusterInput{
    Deleter: e2eCtx.Environment.BootstrapClusterProxy.GetClient(),
    Cluster: cluster,
})

framework.WaitForClusterDeleted(ctx, framework.WaitForClusterDeletedInput{
    Getter:    e2eCtx.Environment.BootstrapClusterProxy.GetClient(),
    Cluster:   cluster,
}, e2eCtx.E2EConfig.GetIntervals("", "wait-delete-cluster")...)
```

To be installable via **clusterctl init** you must:

- Publish your provider as a container to a registry
- Create a GitHub release:
 - Release name should be a version number following the semver convention
 - Attach the following assets:
 - **metadata.yaml**
 - **infrastructure-components.yaml**
 - **cluster-template*.yaml**

```
kustomize build config/default/ > infrastructure-components.yaml  
# NOTE: replace container image with the one from the registry
```

- Building a provider is just the start
- It's advisable to get involved in the wider CAPI community
 - Attend the office hours calls on Wednesdays
 - Read & comment on issues and enhancement proposals (CAEP)
 - Update your provider when new CAPI versions are released
- To raise awareness or to increase adoption for your new provider
 - Host regular Office Hours
 - Encourage new contributors by having a well-defined README, good first issues
 - Use forums like CAPI Office Hours to talk about your provider
- To donate to kubernetes-sigs
 - Check if your repo follows the [kubernetes template project](#) format
 - Fill out the repo [migration request](#)
 - Stay on top of the request and answer any queries :)