KubeCon | CloudNativeCon

Europe 2022

WELCOME TO VALENCIA

KubeCon CloudNativeCon Europe 2022

Build your own Cluster API Provider

The ~~hard~~ easy way

Who are we?

Anusha Hegde

Senior Engineer @ VMWare
Cluster API Provider BYOH Maintainer

Richard Case

Principal Engineer @ Weaveworks
Cluster API Provider AWS Maintainer
Cluster API Provider Microvm Maintainer

Lego & retro gaming addict

Who uses Cluster API (CAPI) already?

Who contributes to a CAPI or a provider?

Who’s thinking of building a Cluster API Provider?

What will we be covering?

- What is Cluster API
- Different Provider types
- Designing a Provider
- Development & Testing
- Community

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.

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)

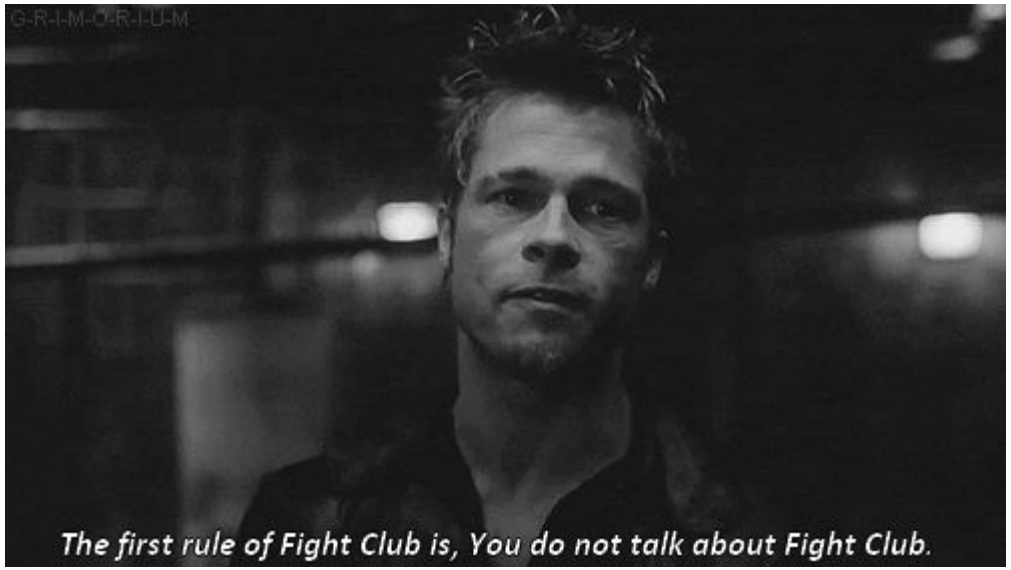
- **Control plane -** used to control the creation & lifecycle of the Kubernetes control plane. It can utilize resources created by bootstrap and infrastructure providers.
  - Kubeadm control plane is the original
  - Managed Kubernetes (i.e. EKS, AKS) implementations - no nodes

First rule of creating a provider…

KubeCon CloudNativeCon Europe 2022

…you don't ~~talk about~~ need to create a provider!

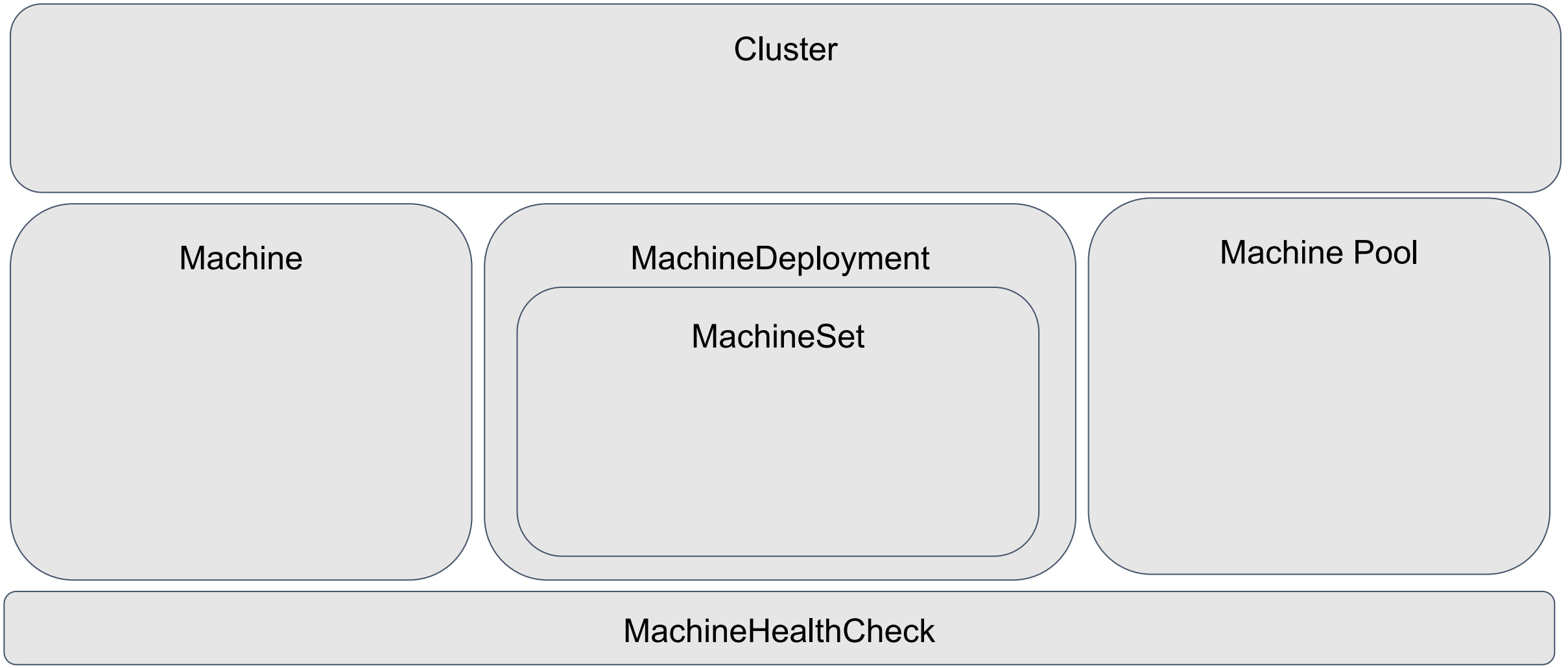
(hopefully)



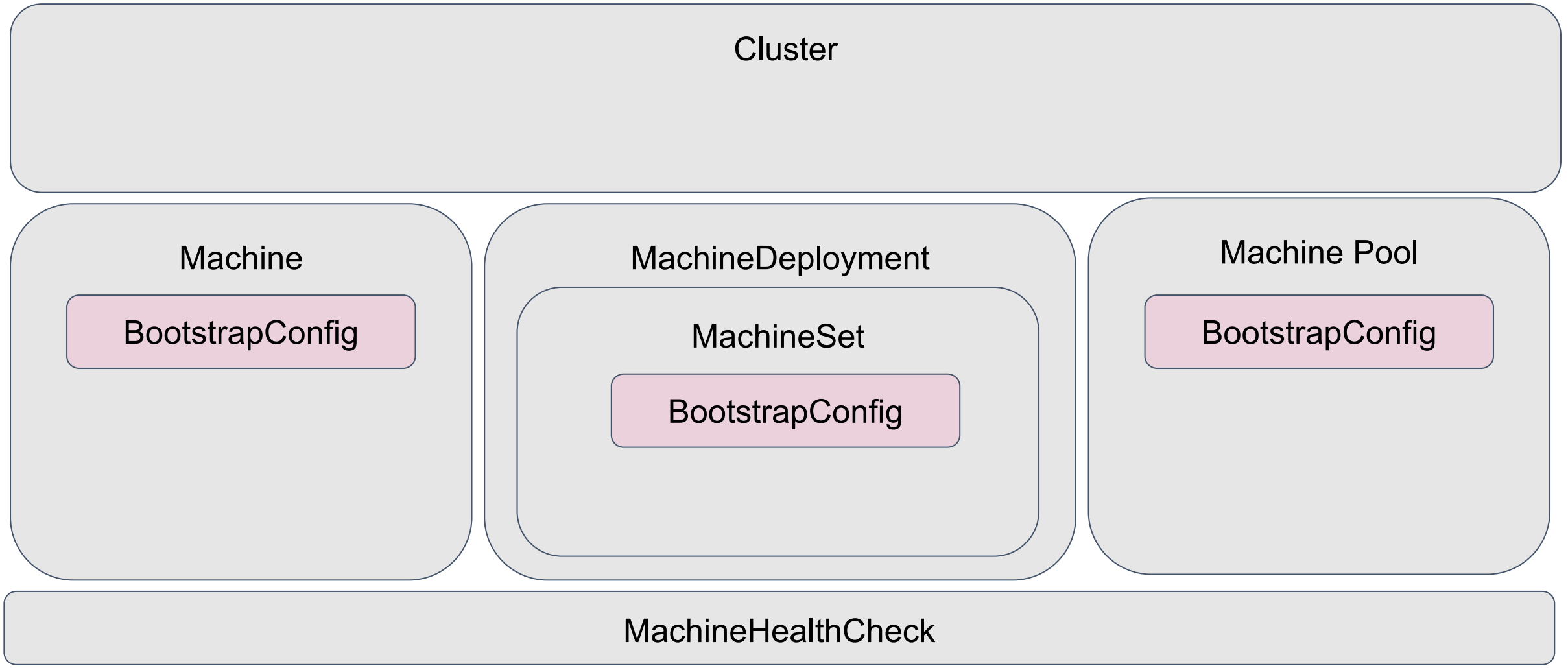
What constitutes a Cluster API provider?

- 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

Core


Cluster
Machine
MachineDeployment
MachineSet
Machine Pool
MachineHealthCheck

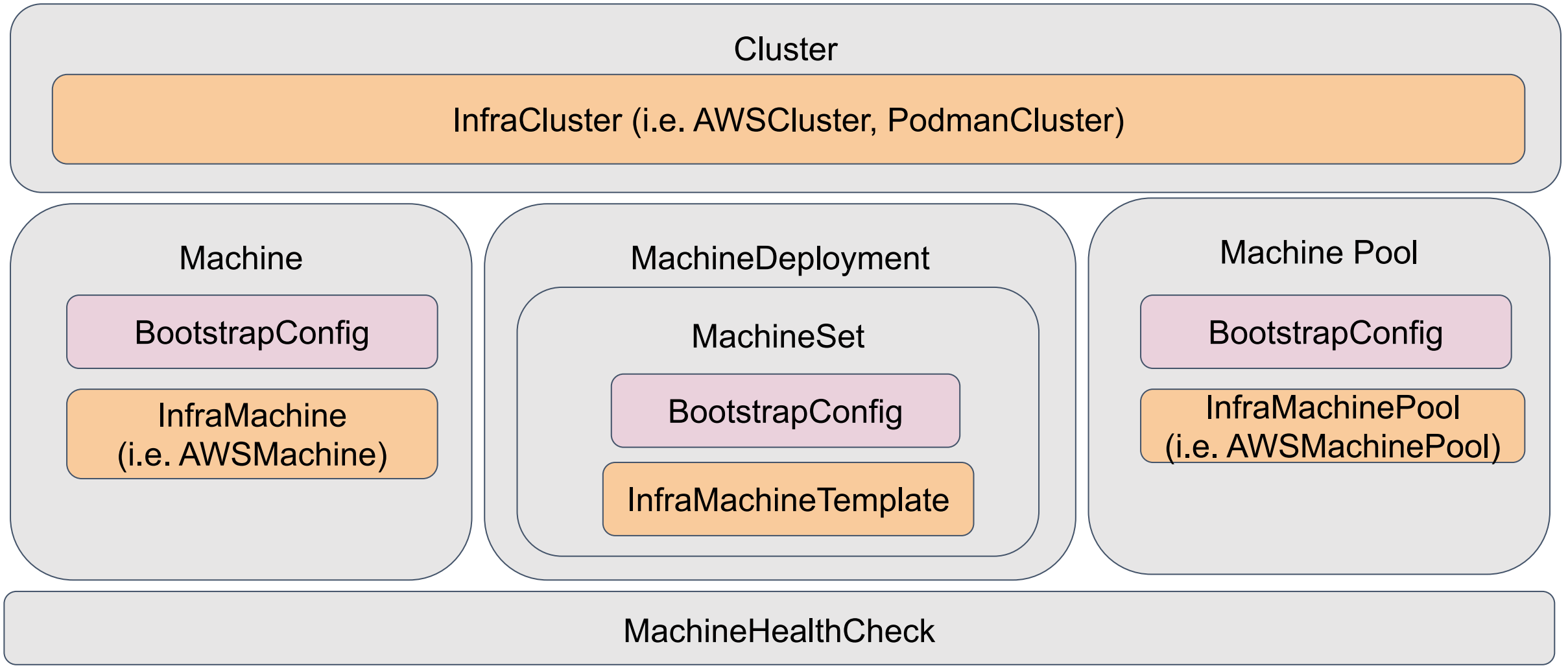
Bootstrap


Cluster
Machine
BootstrapConfig
MachineDeployment
MachineSet
BootstrapConfig
Machine Pool
BootstrapConfig
MachineHealthCheck

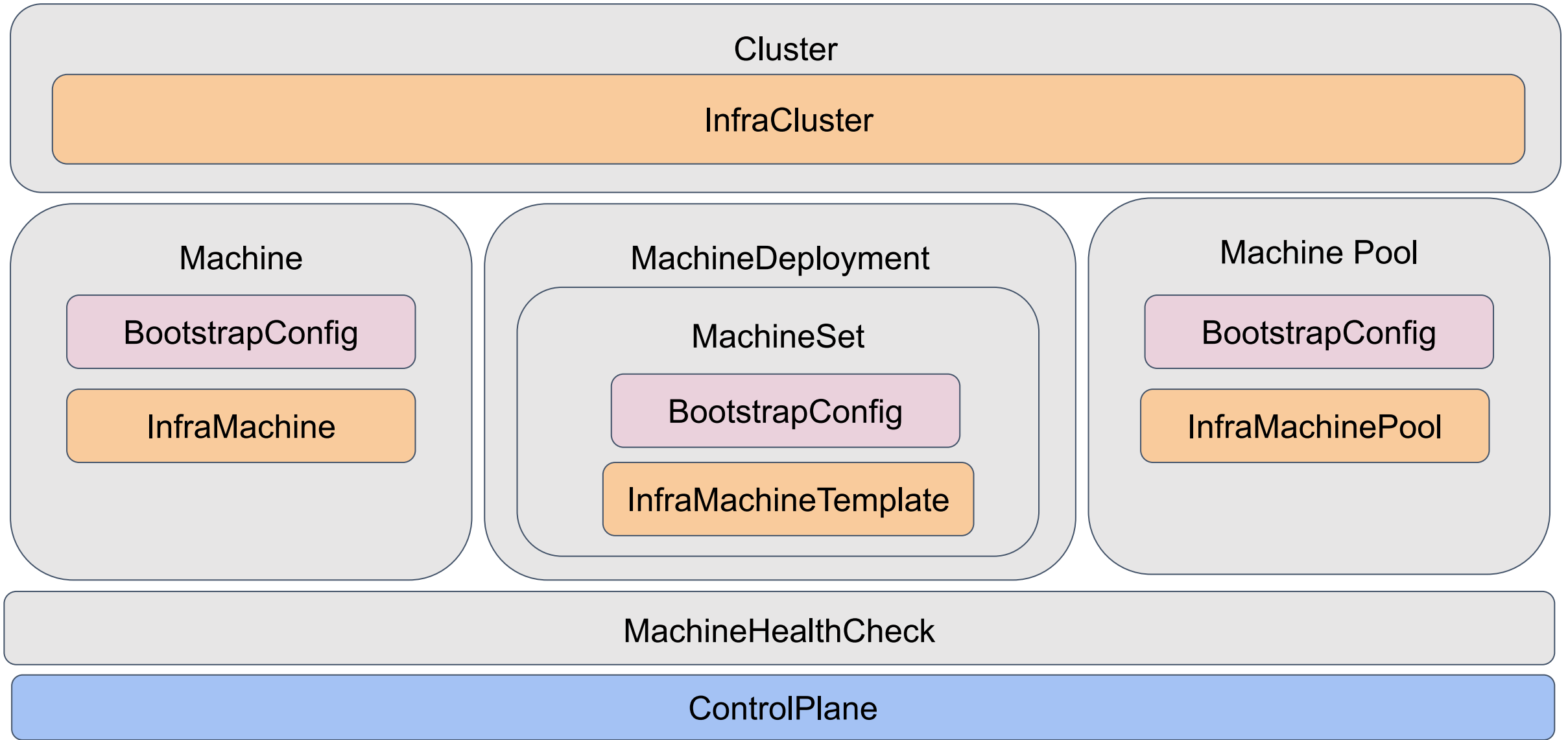
Provider References

```
apiVersion: cluster.x-k8s.io/v1beta1
kind: Cluster
metadata:
  name: "test1"
spec:
  clusterNetwork:
    pods:
      cidrBlocks:
        - 172.25.0.0/16
  infrastructureRef:
    apiVersion: infrastructure.cluster.x-k8s.io/v1alpha1
    kind: PodmanCluster
    name: "test1"
  controlPlaneRef:
    kind: KubeadmControlPlane
    apiVersion: controlplane.cluster.x-k8s.io/v1beta1
    name: "test1-control-plane"}
  return go(f, seed, [])
}
```

Infrastructure



Control Plane



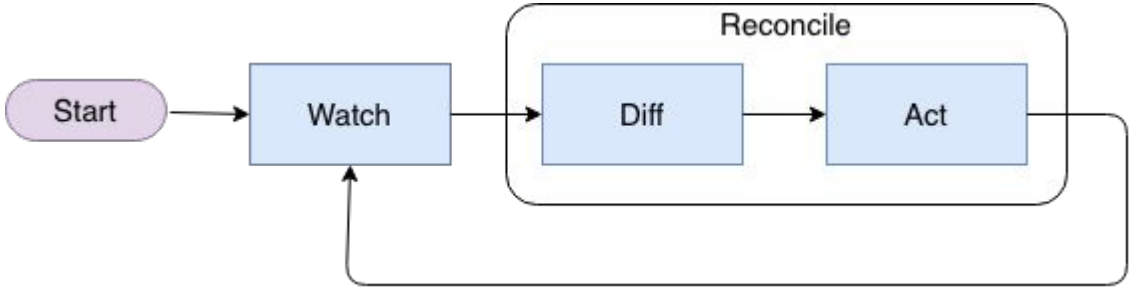
What is an operator?

- 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!

What provider type do you need?

- Do you operate a cloud / baremetal service?
  - You'll need an Infrastructure provider
- Do you want a different way to bootstrap Kubernetes instead of Kubeadm?
  - You'll need a bootstrap provider and maybe a control plane provider
- Do you have a hosted Kubernetes control plane service?
  - You may need a control plane provider
- Do you want to use virtualization technology like KVM or vSphere?
  - You may be covered by the vSphere, Microvm or Kubevirt providers
  - If not then may need to create a infrastructure provider
- Do you want to provision your own infrastructure and get CAPI to manage Kubernetes?
  - You may be covered by one of the existing providers that allow you to bring your own infrastructure (like CAPA, CAPZ)
  - You may be covered by the "bring your own host" provider
  - If not then you may need any of the 3 types of provider

Scaffolding the provider (1/2)

- Kubebuilder and controller-runtime are your friends:

```
kubebuilder init --domain cluster.x-k8s.io --repo github.com/capi-samples/cluster-api-provider-podman
kubebuilder create api --group infrastructure --version v1alpha1 --kind PodmanCluster
kubebuilder create api --group infrastructure --version v1alpha1 --kind PodmanMachine
```

Important parts:

- **–domain cluster.x-k8s.io** - this generally used as the domain part of the GVK by providers
- **–group infrastructure** - this by convention indicates that the CRD relates to an infrastructure provider
- **–version v1alpha1** - providers need to follow the Kubernetes API versioning standard and the guarantees these provide to the user
- **–kind PodmanCluster/Machine** - by convention, a provider uses a consistent prefix on the CRD names

Scaffolding the provider (2/2)

- We also need a machine template API type **BUT no controller**
- Add a reference to CAPI so we can use their API definitions / utility functions

Why do we not need a controller for the machine template?

- It's used as a template to create new instances of **PodmanMachine**
- **PodmanMachine** has a controller to handle reconciliation

Provider Metadata

A provider must specify which versions are compatible with which CAPI API version.

- One of the requirements to be installable via **clusterctl init**
- Create a **metadata.yaml** file in the root of the repo
- clusterctl init –infrastructure podman

```yaml
# maps release series of major.minor to cluster-api contract version
# the contract version may change between minor or major versions, but *not*
# between patch versions.
#
# update this file only when a new major or minor version is released
apiVersion: clusterctl.cluster.x-k8s.io/v1alpha3
releaseSeries:
  - major: 0
    minor: 1
    contract: v1beta1
```

Define API for your provider (1/2)

- Add fields to the **Spec** & **Status** of your API types to **conform to your provider types contract**
  - CAPI documentation helps: https://cluster-api.sigs.k8s.io/developer/providers/implementers.html

```go
// PodmanClusterSpec defines the desired state of PodmanCluster
type PodmanClusterSpec struct {
    // ControlPlaneEndpoint represents the endpoint used to communicate with the control plane.
    //
    // See https://cluster-api.sigs.k8s.io/developer/architecture/controllers/cluster.html
    // for more details.
    //
    // +optional
    ControlPlaneEndpoint clusterv1.APIEndpoint `json:"controlPlaneEndpoint"`
}

// PodmanClusterStatus defines the observed state of PodmanCluster
type PodmanClusterStatus struct {
    // Ready indicates that the cluster is ready.
    // +optional
    // +kubebuilder:default=false
    Ready bool `json:"ready"`

    // FailureDomains is a list of the failure domains that CAPI should spread the machines across. For
    // the CAPPOD provider this doesn't mean anything.
    FailureDomains clusterv1.FailureDomains `json:"failureDomains,omitempty"`
}
```

Define API for your provider (2/2)

- Add custom fields to **the Spec** & **Status** of your API types that are specific to your provider

```go
// PodmanMachineSpec defines the desired state of PodmanMachine
type PodmanMachineSpec struct {
    // ProviderID will be the container name in ProviderID format (podman:////<containername>)
    // +optional
    ProviderID *string `json:"providerID,omitempty"`

    // ExtraMounts describes additional mount points for the node container
    // These may be used to bind a hostPath
    // +optional
    ExtraMounts []Mount `json:"extraMounts,omitempty"`
}
```

Add finalizers (1/2)

- **As providers generally create external resources you will need to define finalizers**
  - A finalizer allows the controller to clean up external resources before allowing the API type to be deleted from API server.
  - See docs: https://book.kubebuilder.io/reference/using-finalizers.html

```
const (
    // MachineFinalizer allows ReconcilePodmanMachine to clean up resources associated with
    // PodmanMachine before removing it from the apiserver.
    MachineFinalizer = "podmanmachine.infrastructure.cluster.x-k8s.io"
)
```

Add finalizers (2/2)

- The controllers will add and remove the finalizers

```
func (r *PodmanMachineReconciler) reconcileNormal(ctx context.Context, podmanMachine
*infrav1.PodmanMachine, machine *clusterv1.Machine) (res ctrl.Result, reterr error) {
    logger := log.FromContext(ctx)
    logger.Info("Reconciling PodmanMachine")

    controllerutil.AddFinalizer(podmanMachine, infrav1.MachineFinalizer)
    if err := r.patchObject(podmanMachine); err != nil {
        return ctrl.Result{}, err
    }
    ...
}
```

** Opposite will need to be done for deletion

Implement controllers for your API types (1/2)

- Kubebuilder will have created an “empty” controller with
  - **Reconcile** function
  - Controller setup to watch its CRD type

- Setup tasks that you need to do:
  - Watch any additional CRDs from CAPI
  - Ensure reconciliation doesn’t occur if paused or if the resource is externally managed

```go
// SetupWithManager sets up the controller with the Manager.
func (r *PodmanMachineReconciler) SetupWithManager(ctx context.Context, mgr ctrl.Manager, options
controller.Options) error {
    log := ctrl.LoggerFrom(ctx)

    builder := ctrl.NewControllerManagedBy(mgr).
        WithOptions(options).
        For(&infrav1.PodmanMachine{}).
        WithEventFilter(predicates.ResourceNotPaused(log)).
        WithEventFilter(predicates.ResourceIsNotExternallyManaged(log)).
        Watches(
            &source.Kind{Type: &clusterv1.Machine{}},
            handler.EnqueueRequestsFromMapFunc(
                util.MachineToInfrastructureMapFunc(infrav1.GroupVersion.WithKind("PodmanMachine")),
            ),
        ) //TODO: add additional watches for PodmanCluster and Cluster if needed

    return builder.Complete(r)
}
```

Implement controllers for your API types (2/2)

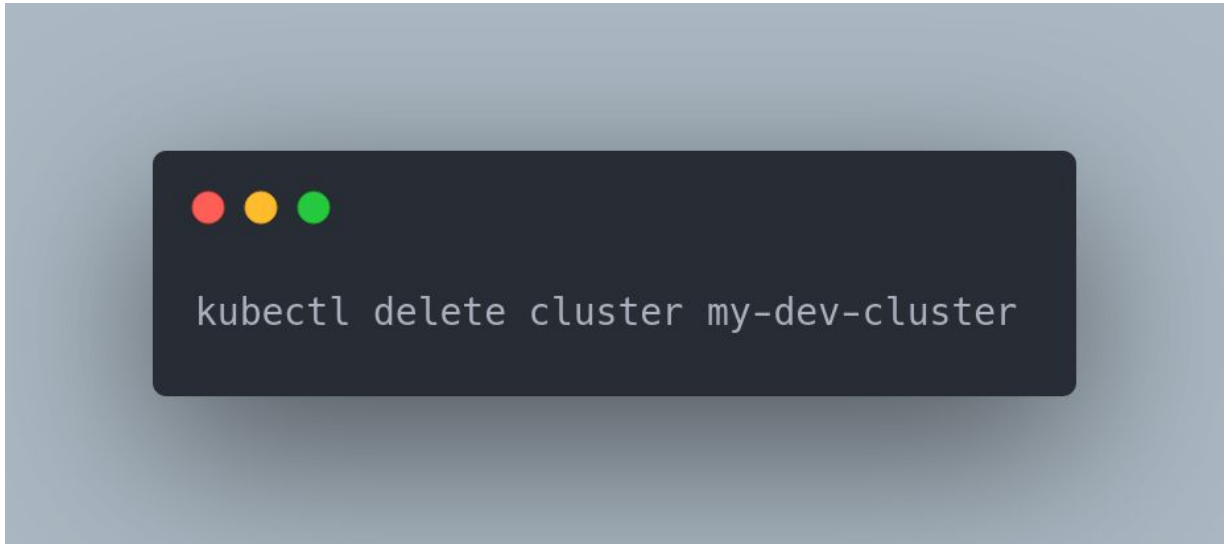
For **Reconcile** the following pattern is generally used:

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

Owner Reference

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



Webhooks

If you need custom logic for defaults or validation you can create webhooks:

```
→ kubebuilder create webhook --group infrastructure --version v1alpha1 --kind PodmanCluster
--defaulting --programmatic-validation
→ kubebuilder create webhook --group infrastructure --version v1alpha1 --kind PodmanMachine
--defaulting --programmatic-validation
kubebuilder create webhook --group infrastructure --version v1alpha1 --kind PodmanMachineTemplate
--defaulting --programmatic-validatio
```

- Webhooks are Kubernetes Admission controllers
- This will scaffold both a **defaulting** & **validating** webhook
  - It's your responsibility to fill in the logic
- Defaulting webhook should only be used where kubebuilder defaults are not sufficient

Webhook - Validation Implementation


```go
var _ webhook.Validator = &PodmanMachine{}

// ValidateCreate implements webhook.Validator so a webhook will be registered for the type
func (r *PodmanMachine) ValidateCreate() error {
    podmanmachinelog.Info("validate create", "name", r.Name)

    var allErrs field.ErrorList

    for _, mount := range r.Spec.ExtraMounts {
        if mount.HostPath == "" || mount.ContainerPath == "" {
            allErrs = append(allErrs, field.Invalid(
                field.NewPath("spec", "extraMounts"), "", "must specify both host and container path",
            ),
            )
        }
    }

    if len(allErrs) == 0 {
        return nil
    }

    return apierrors.NewInvalid(GroupVersion.WithKind("Cluster").GroupKind(), r.Name, allErrs)
}
```

Webhook - Defaulting Implementation


```go
type PodmanMachineSpec struct {
    // CPUs specifies the number of CPUs for the machine.
    // +kubebuilder:default:=2
    CPUs int `json:cpus`
```

<-Kubebuilder defaults

Custom defaulter ->

```go
// Default implements webhook.Defaulter so a webhook will be registered for the type
func (r *PodmanMachine) Default() {
    podmanmachinelog.Info("default", "name", r.Name)

    if r.Spec.Image =="" {
        r.Spec.Image = lookupDefaultImageForFamily(r.Spec.ImageFamily)
    }
}
```

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": "CAPPOD"
      }
    }
]
```

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
        }
    }
}
```

Testing (1/3)

- 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())
```

Testing (2/3)


```go
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)
}
```

Testing (3/3)

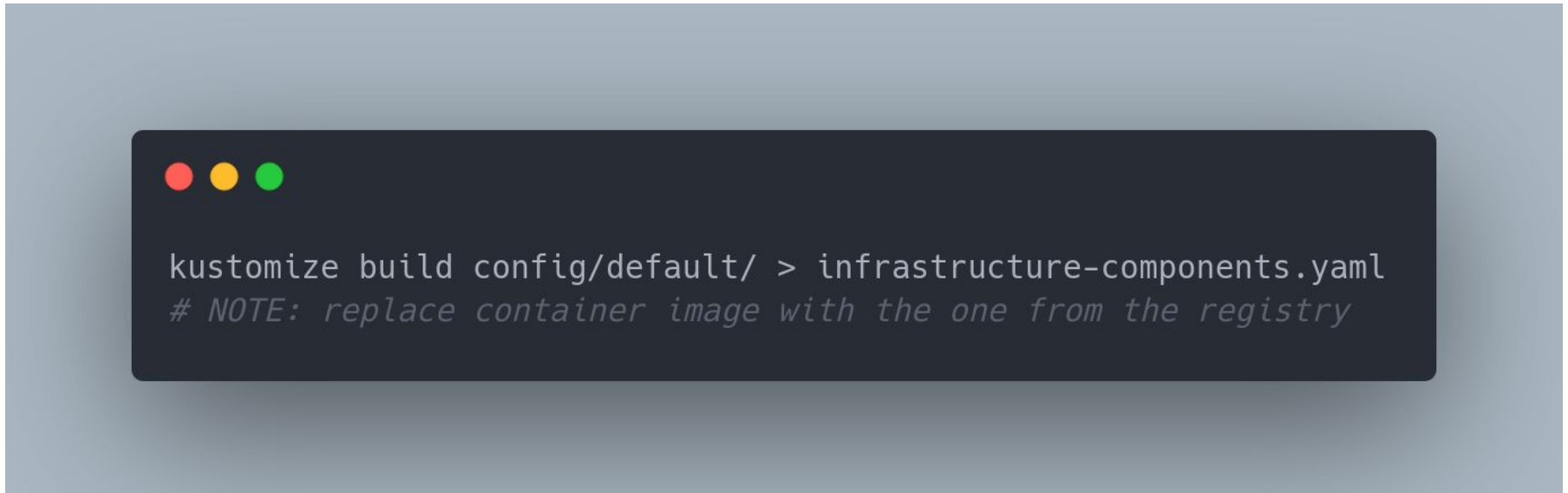
- CAPI provides e2e framework - most of the code is reusable

```
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")...)
```

Releasing

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



Community

- 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 :)

Wrapping up

There is a lot we haven't covered. Some important areas:

- **Multiple API versions, conversions and Hub/Spoke**
- **Cluster templates & Cluster Classes**
- **Upgrades**

Some resources when you implement your own provider:

- CAPI Repo: https://github.com/kubernetes-sigs/cluster-api
- CAPI Provider Implementers docs: https://cluster-api.sigs.k8s.io/developer/providers/implementers.html
- List of existing providers: **https://cluster-api.sigs.k8s.io/reference/providers.html**
- Kubebuilder docs: https://book.kubebuilder.io/

Thanks for listening…..any questions?