Kubernetes Resources & Extensions

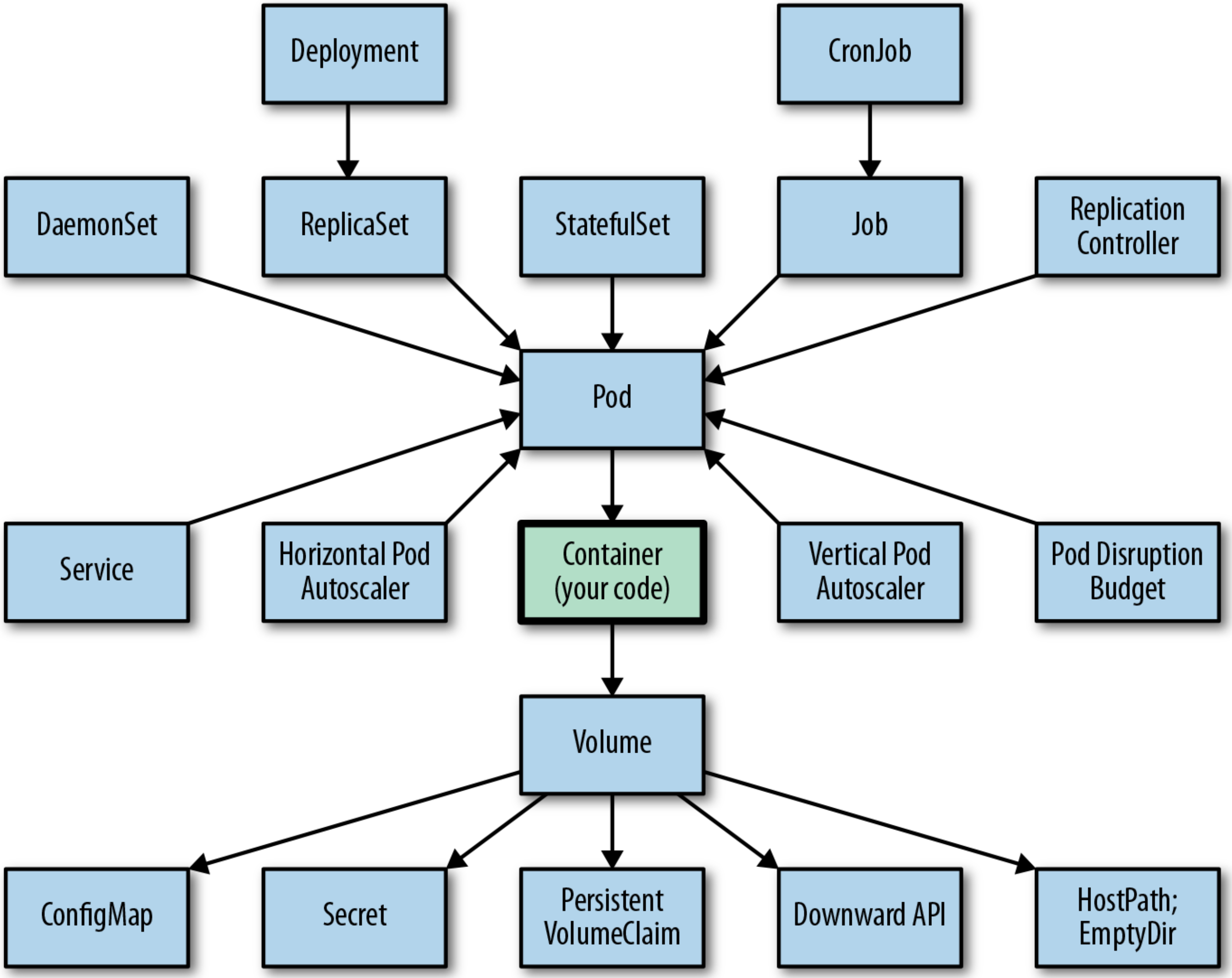
SoC Summer Workshop
Cloud Computing with Big Data

Richard T. B. Ma
School of Computing
National University of Singapore

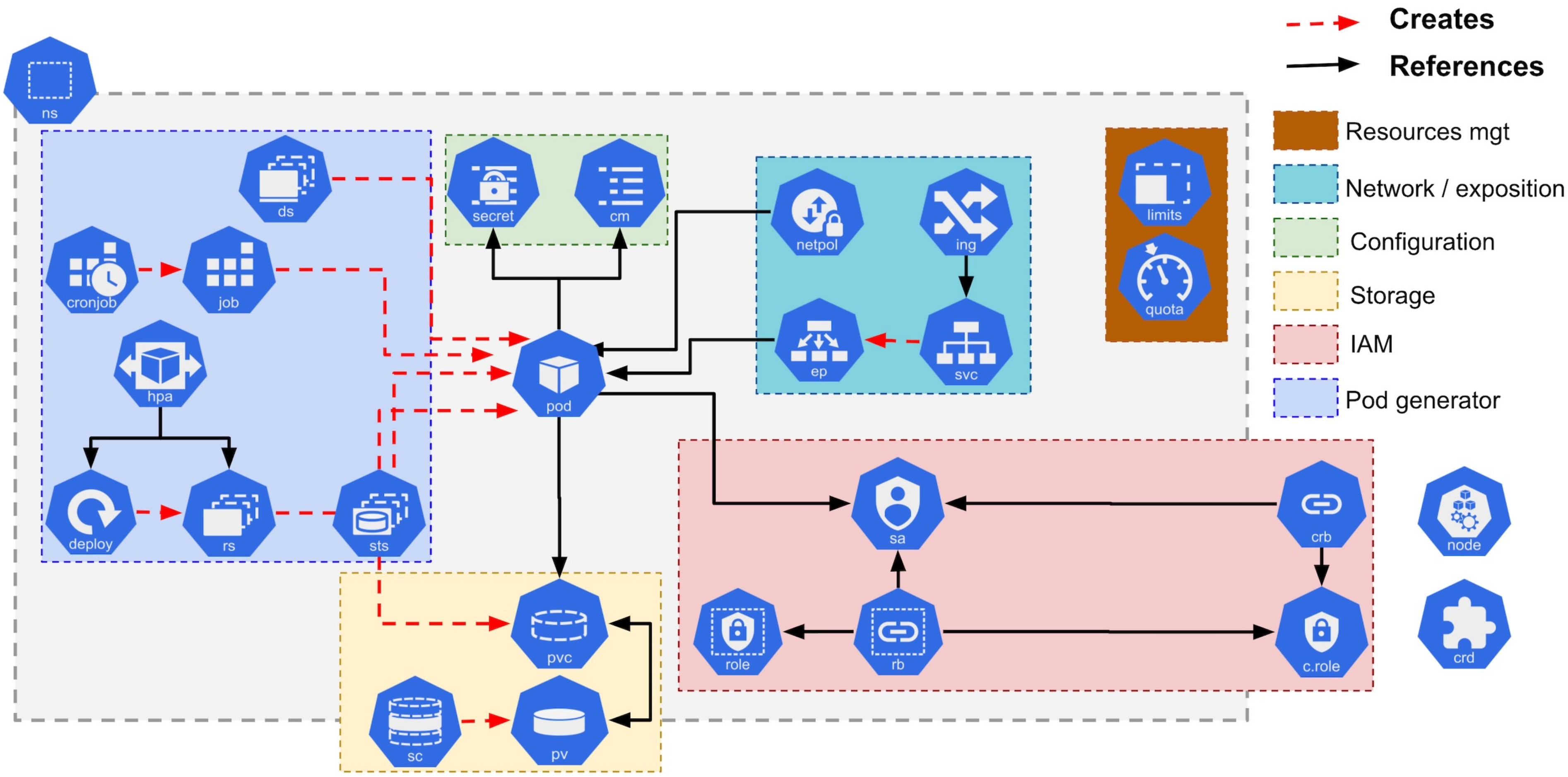
Roadmap

- Kubernetes API Resources
- Kubernetes Extensions
  - Controllers
  - Custom Resources
  - The Operator Pattern

A view of K8s for app developers



Kubernetes Resources Map



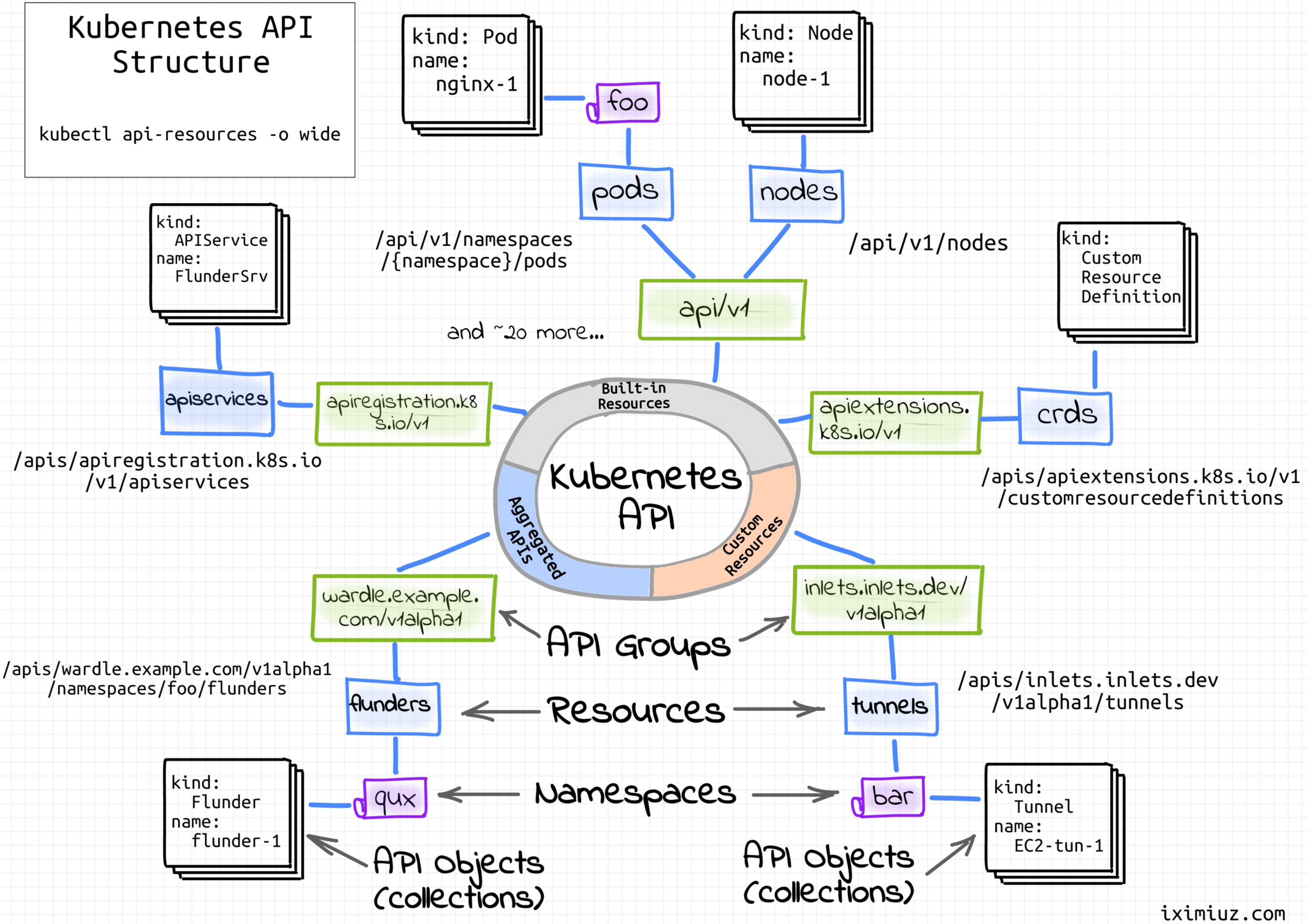
Kubernetes Resources

- Resources refer to the API endpoints that allow you to interact with the objects in the cluster.

- Resources vs Objects
  - Objects are persistent entities in K8s
    - represent an *intent* (desired state)
    - and the *status* (actual state) of the cluster
  - Resource is more general concept
    - computing components and entities that can be managed within a Kubernetes cluster
    - include both object and non-object resources
    - RESTful like APIs via the API Server

Kubernetes API Structure

```
kubectl api-resources -o wide
```



kind: Pod
name:
  nginx-1

kind: Node
name:
  node-1

/api/v1/namespaces
/{namespace}/pods

/api/v1/nodes

kind:
  APIService
name:
  FlunderSrv

kind:
  Custom
  Resource
  Definition

/apis/apiregistration.k8s.io
/v1/apiservices

/apis/apiextensions.k8s.io/v1
/customresourcedefinitions

/apis/wardle.example.com/v1alpha1
/namespaces/foo/flunders

/apis/inlets.inlets.dev
/v1alpha1/tunnels

kind:
  Flunder
name:
  flunder-1

kind:
  Tunnel
name:
  EC2-tun-1

iximiuz.com

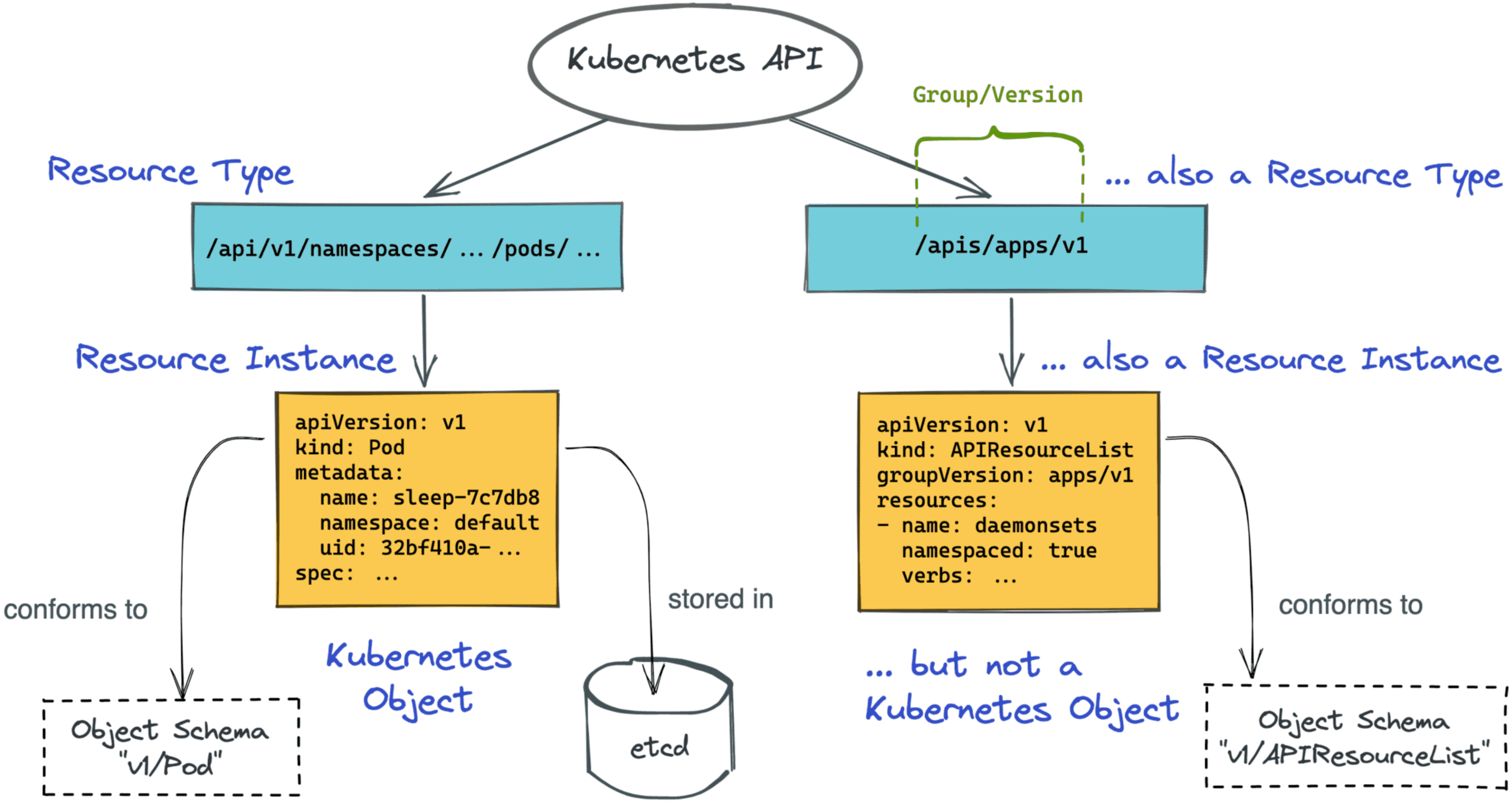
Types of API Resources

- ❑ Built-in Resources
  - include all native objects
  - has a special `CustomResourceDefinition` kind
- ❑ Custom Resources
  - created by CRDs
  - the Operator pattern
- ❑ Aggregated APIs
  - extend the functionality of API server
  - create new API endpoints via `APIService` kind
  - hosted outside of the main K8s API server

API Resource Groups

- Resources are bound together in API groups
  - each group may have one or more versions that evolve independent of other API groups,
  - each version within the group has one or more resources.
- Group names are typically in domain name form
  - the Kubernetes project reserves use of the empty group, all single word names ("extensions", "apps"), and any group name ending in "*.k8s.io" for its sole use.
- When choosing a group name, recommend a subdomain your group or organization owns
  - e.g., "widget.mycompany.com".

Object vs Non-Object Resources



```
apiVersion: v1
kind: Pod
metadata:
  name: sleep-7c7db8
  namespace: default
  uid: 32bf410a-…
spec: …
```

```
apiVersion: v1
kind: APIResourceList
groupVersion: apps/v1
resources:
- name: daemonsets
  namespaced: true
  verbs: …
```

iximiuz.com

Non-Object Resources

- Resources not treated as Object:
  - **Nodes**: the machines (physical or virtual) that form the cluster.
  - **Namespaces**: provide a way to partition a cluster into multiple virtual clusters and used to organize and isolate resources within a cluster.
  - **ConfigMaps and Secrets**: are used to store configuration data and sensitive information, respectively, in a Kubernetes cluster.
- Resources managed by the Kubernetes control plane.
  - **Persistent Volumes (PVs) and Persistent Volume Claims (PVCs)**: are used to manage persistent storage. PVs represent storage volumes provisioned by the admin; PVCs are requests for storage made by Pods.
  - **Service Accounts**: provide an identity for Pods running in a Kubernetes cluster. They are used by Pods to authenticate and authorize with other cluster components, such as the API server.
  - **Cluster Roles and Cluster Role Bindings**: are used to define sets of permissions (RBAC) for accessing cluster-wide resources. They are similar to roles and role bindings but operate at the cluster level.

RESTful style API

- Create, delete, retrieve, or update a description of an object via the standard HTTP verbs (POST, PUT, DELETE, GET)
  - APIs preferentially accept and return JSON.
  - has a schema, identified by `kind` and `apiVersion` fields.
  - also exposes additional endpoints for non-standard verbs.
- Create a local proxy that acts as an intermediary between local machine and the API server:

  ```
  $ kubectl proxy --port=8001
  ```

  - allows access the API securely from local machine without complex authentication settings. Now try:

```
$ curl -X GET http://localhost:8001/api/v1/namespaces/<namespace>/pods/<pod-name>
```

Roadmap

- Kubernetes API Resources
- Kubernetes Extensions
  - Controllers
  - Custom Resources
  - The Operator Pattern

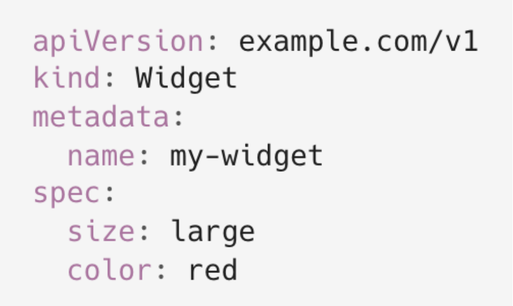
Controller as a design pattern

- Conceptually, it is easily a loop of the following
  - obtain runtime status
  - obtain current spec
  - check difference, make changes for both to converge
- Can be considered as a design pattern
  - an example on ConfigMap:

  https://github.com/k8spatterns/examples/tree/main/advanced/Controller
- Simple controllers can be built on native objects
  - as an extension to enable new logics and functionalities
  - typically run as a `Deployment` with just one replica
  - but invisible to the users of the cluster

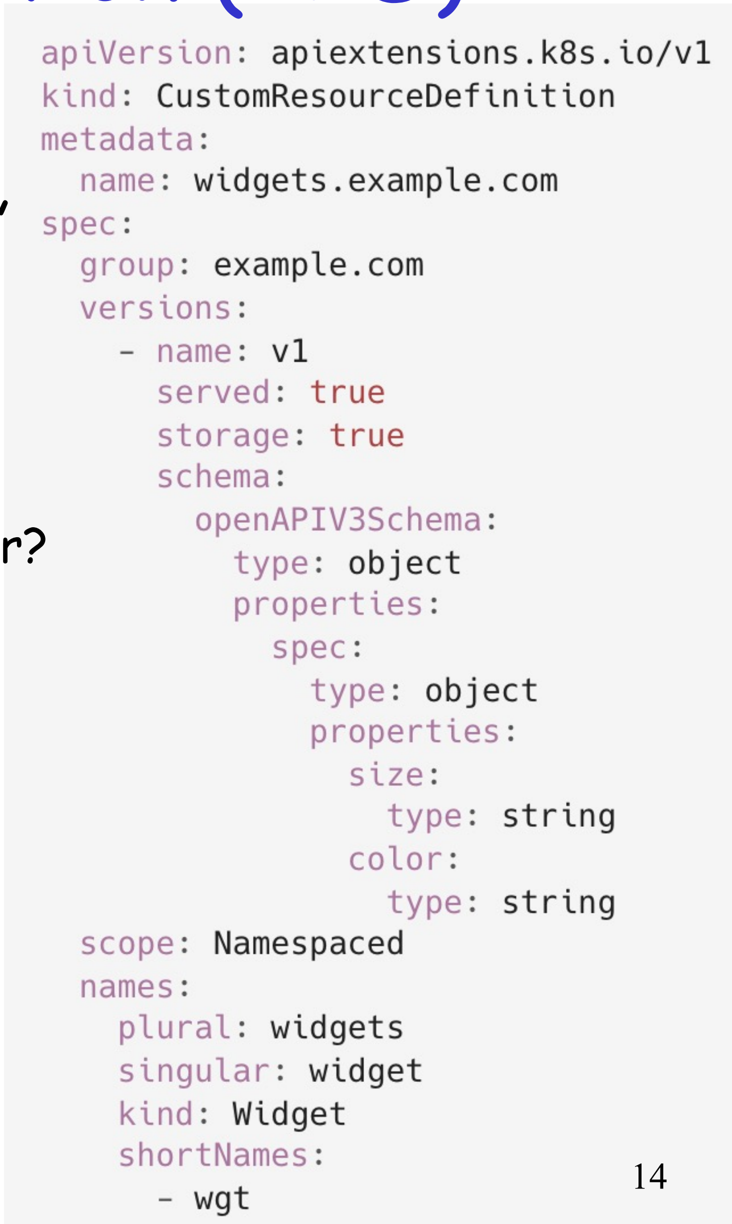
Custom Resource Definition (CRD)

- ❑ used to create custom resources, i.e., extensions of the K8s API.
  - ❖ spec.scope: Namespaced or Cluster
  - ❖ spec.group: the API group
  - ❖ spec.versions.served: by API server?
  - ❖ spec.versions.storage: by etcd cluster?
  - ❖ spec.versions.schema: open standard
- ❑ create a custom resource (CR):



```yaml
apiVersion: example.com/v1
kind: Widget
metadata:
  name: my-widget
spec:
  size: large
  color: red
```

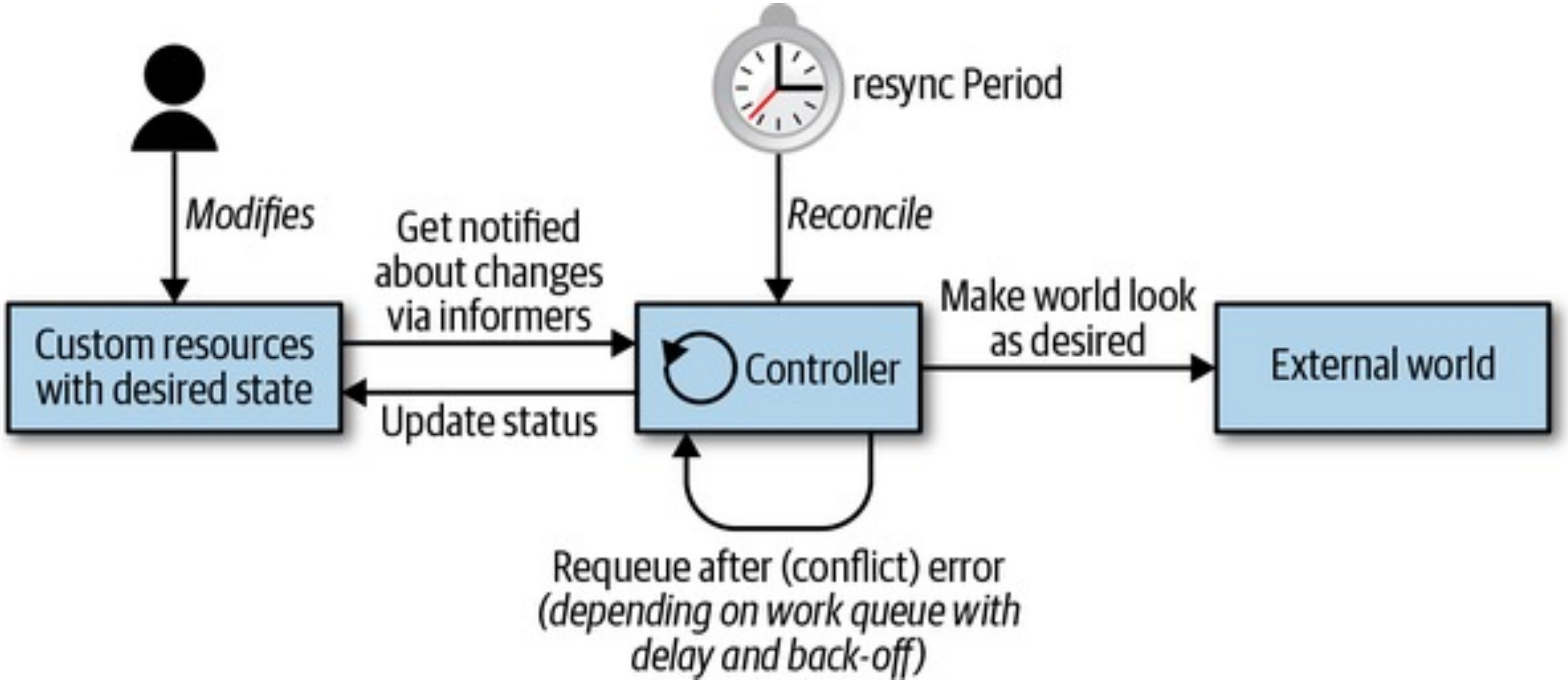
- ❑ controller?



```yaml
apiVersion: apiextensions.k8s.io/v1
kind: CustomResourceDefinition
metadata:
  name: widgets.example.com
spec:
  group: example.com
  versions:
    - name: v1
      served: true
      storage: true
      schema:
        openAPIV3Schema:
          type: object
          properties:
            spec:
              type: object
              properties:
                size:
                  type: string
                color:
                  type: string
  scope: Namespaced
  names:
    plural: widgets
    singular: widget
    kind: Widget
    shortNames:
      - wgt
```

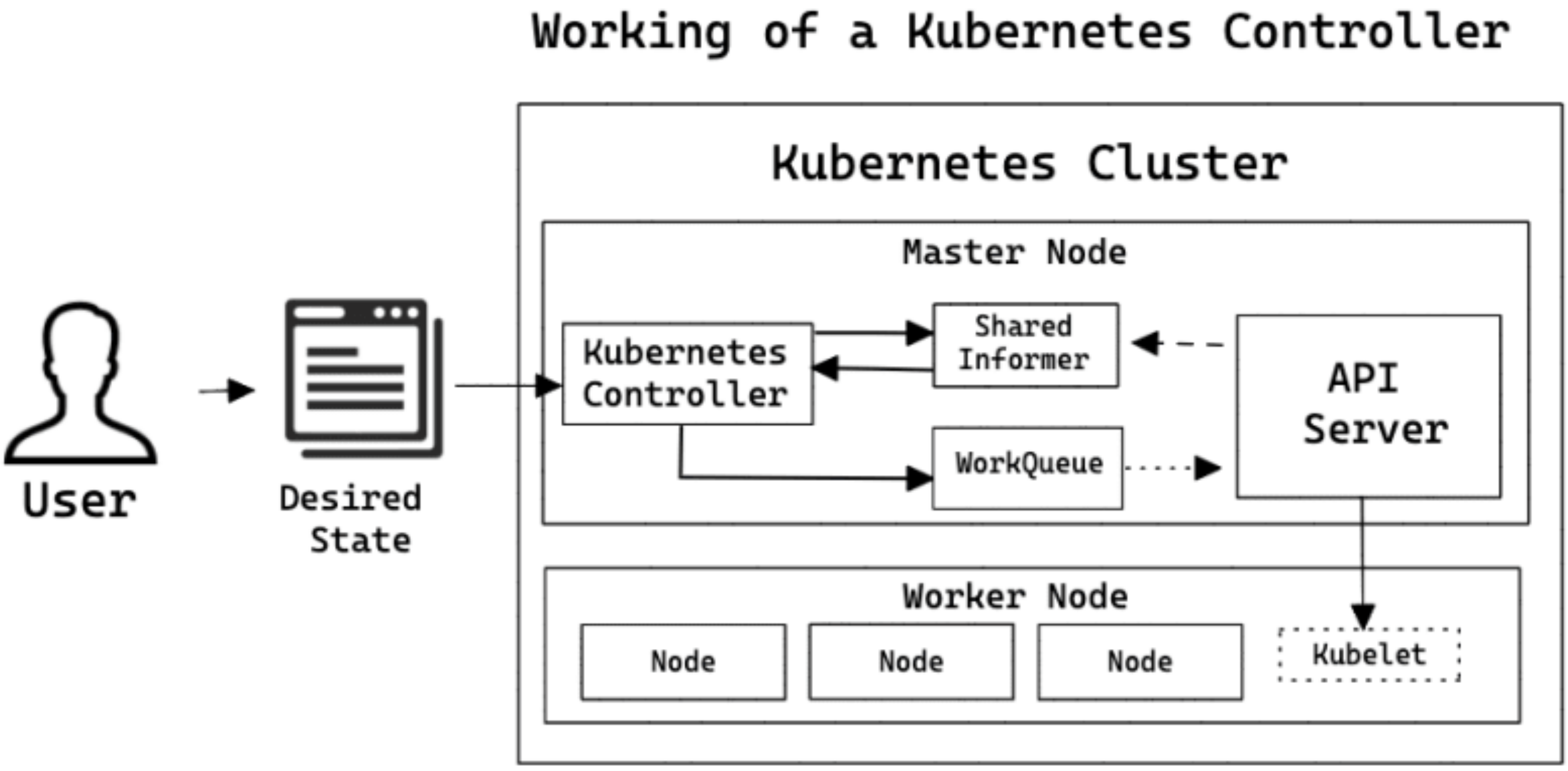
Kubernetes Controllers

- The detailed architecture of Kubernetes native controllers is more complicated.



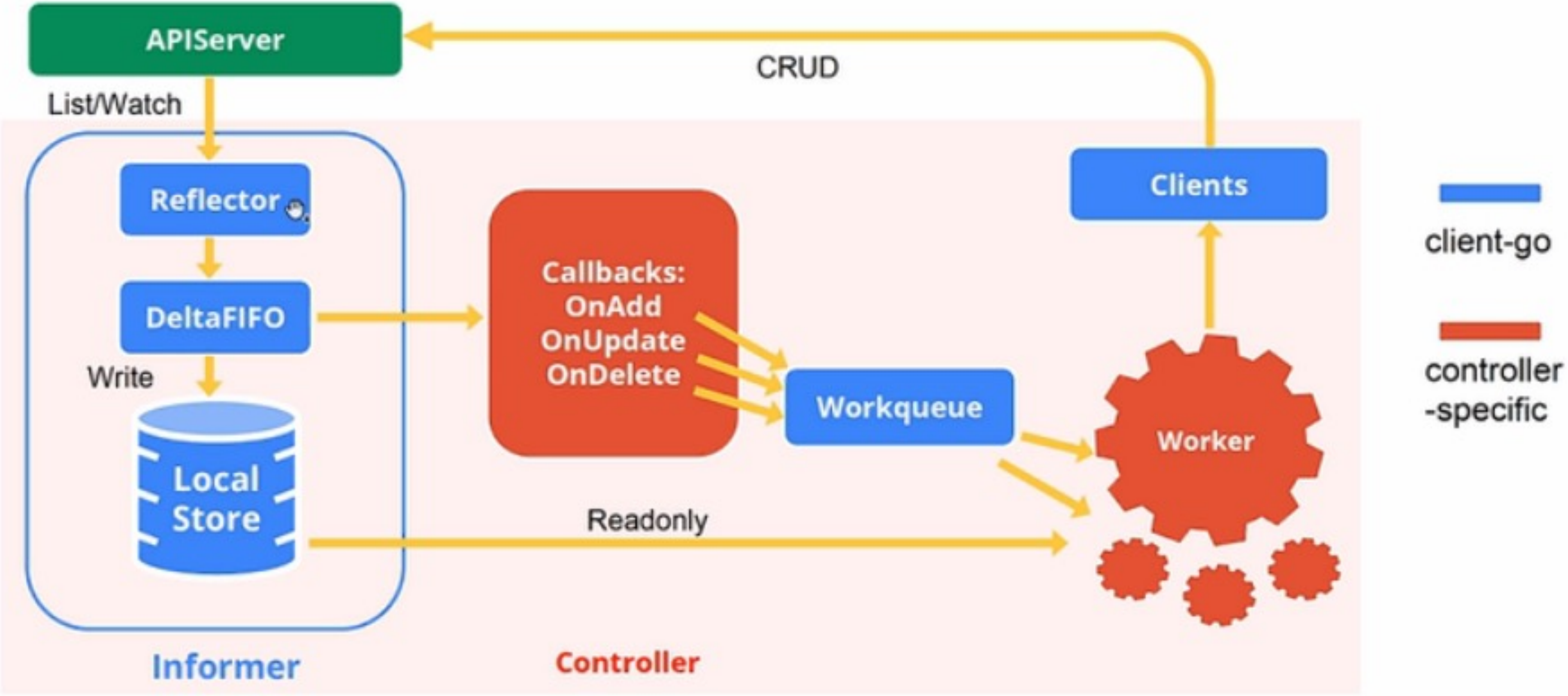
resync Period
Modifies
Reconcile
Get notified about changes via informers
Make world look as desired
Custom resources with desired state
Controller
External world
Update status
Requeue after (conflict) error (depending on work queue with delay and back-off)

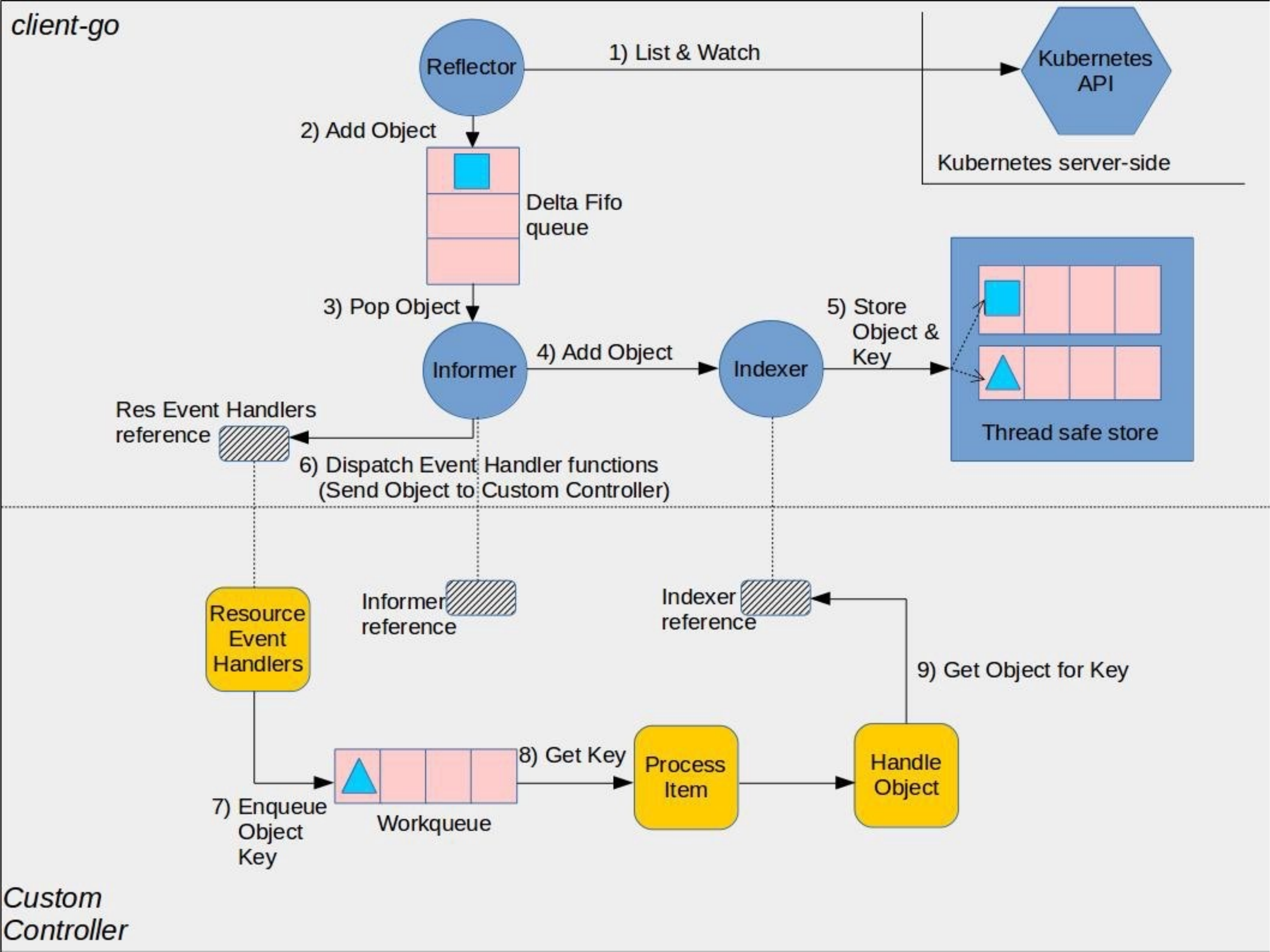
Kubernetes Controllers



HackerDraw

General pattern of a Kubernetes controller




client-go
Reflector
1) List & Watch
Kubernetes API
Kubernetes server-side
2) Add Object
Delta Fifo queue
3) Pop Object
Informer
4) Add Object
Indexer
5) Store Object & Key
Thread safe store
Res Event Handlers reference
6) Dispatch Event Handler functions (Send Object to Custom Controller)
Resource Event Handlers
Informer reference
Indexer reference
9) Get Object for Key
7) Enqueue Object Key
Workqueue
8) Get Key
Process Item
Handle Object
Custom Controller

A custom controller written in Go

- watches for changes to `ConfigMaps` and logs these changes and leverages the `client-go` library
- step 1: init a new GO module:

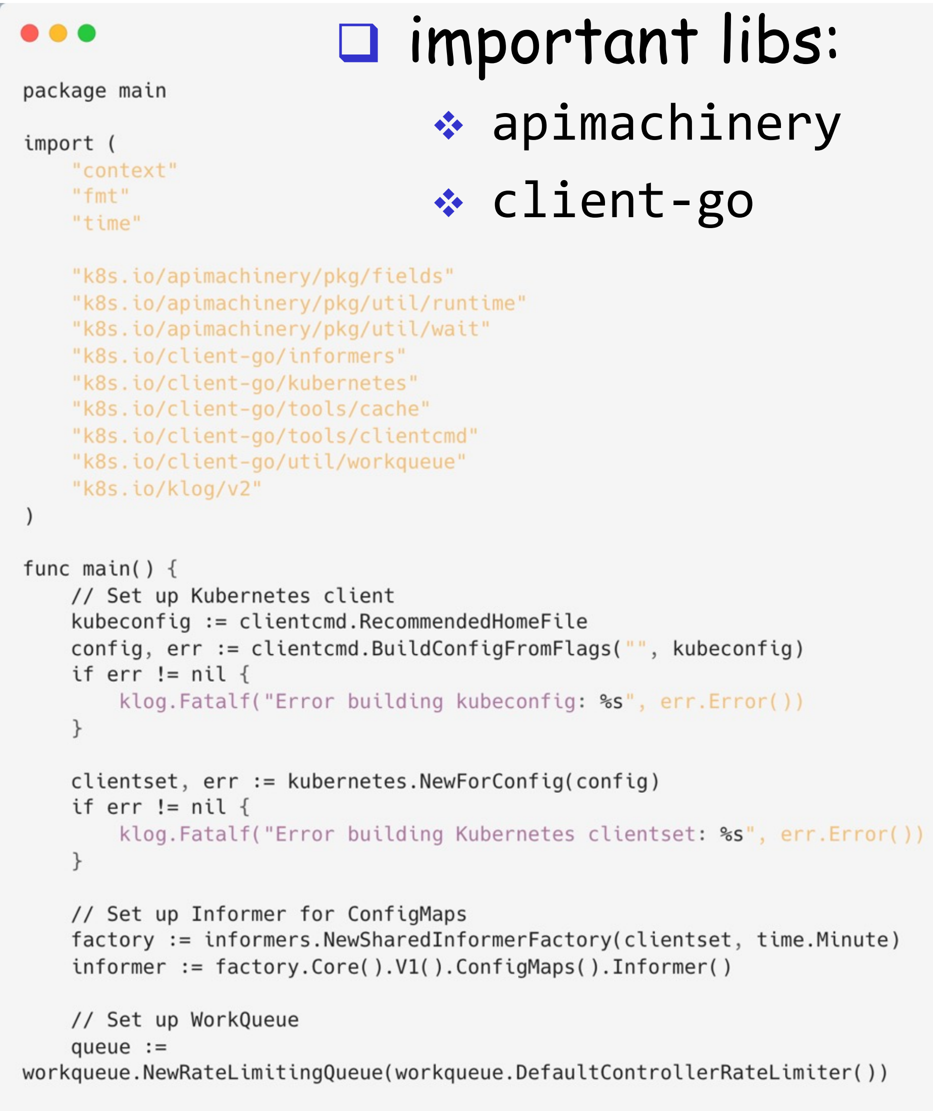
```
$ mkdir configmap-controller
$ cd configmap-controller
$ go mod init configmap-controller
$ go get k8s.io/client-go@kubernetes-1.26.1
```

- step 2: write the controller code `main.go`
- step 3: build and run controller:

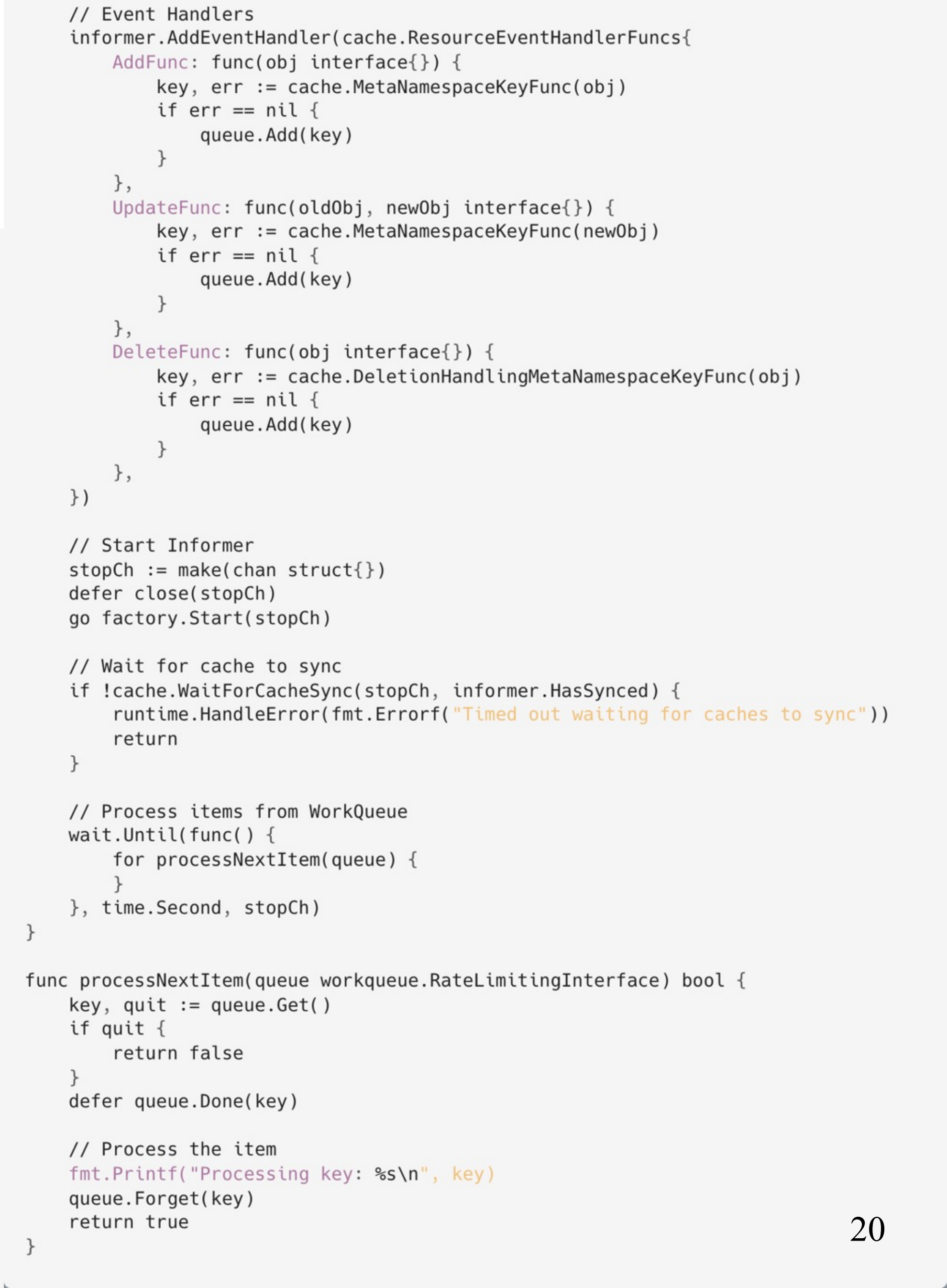
```
$ go build -o configmap-controller .
$ ./configmap-controller
```

main.go

- ❑ important libs:
  - ❖ apimachinery
  - ❖ client-go



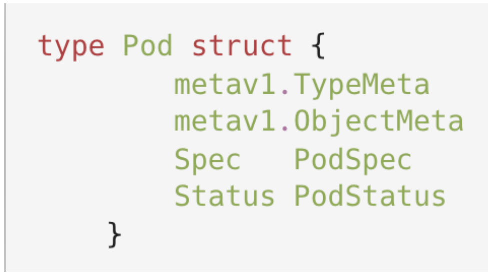
```go
package main

import (
    "context"
    "fmt"
    "time"

    "k8s.io/apimachinery/pkg/fields"
    "k8s.io/apimachinery/pkg/util/runtime"
    "k8s.io/apimachinery/pkg/util/wait"
    "k8s.io/client-go/informers"
    "k8s.io/client-go/kubernetes"
    "k8s.io/client-go/tools/cache"
    "k8s.io/client-go/tools/clientcmd"
    "k8s.io/client-go/util/workqueue"
    "k8s.io/klog/v2"
)

func main() {
    // Set up Kubernetes client
    kubeconfig := clientcmd.RecommendedHomeFile
    config, err := clientcmd.BuildConfigFromFlags("", kubeconfig)
    if err != nil {
        klog.Fatalf("Error building kubeconfig: %s", err.Error())
    }

    clientset, err := kubernetes.NewForConfig(config)
    if err != nil {
        klog.Fatalf("Error building Kubernetes clientset: %s", err.Error())
    }

    // Set up Informer for ConfigMaps
    factory := informers.NewSharedInformerFactory(clientset, time.Minute)
    informer := factory.Core().V1().ConfigMaps().Informer()

    // Set up WorkQueue
    queue :=
workqueue.NewRateLimitingQueue(workqueue.DefaultControllerRateLimiter())
```



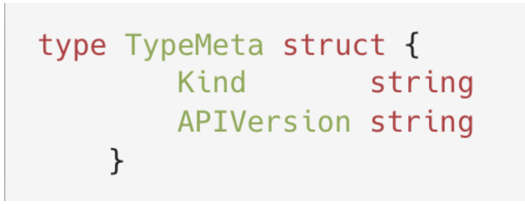
```go
    // Event Handlers
    informer.AddEventHandler(cache.ResourceEventHandlerFuncs{
        AddFunc: func(obj interface{}) {
            key, err := cache.MetaNamespaceKeyFunc(obj)
            if err == nil {
                queue.Add(key)
            }
        },
        UpdateFunc: func(oldObj, newObj interface{}) {
            key, err := cache.MetaNamespaceKeyFunc(newObj)
            if err == nil {
                queue.Add(key)
            }
        },
        DeleteFunc: func(obj interface{}) {
            key, err := cache.DeletionHandlingMetaNamespaceKeyFunc(obj)
            if err == nil {
                queue.Add(key)
            }
        },
    })

    // Start Informer
    stopCh := make(chan struct{})
    defer close(stopCh)
    go factory.Start(stopCh)

    // Wait for cache to sync
    if !cache.WaitForCacheSync(stopCh, informer.HasSynced) {
        runtime.HandleError(fmt.Errorf("Timed out waiting for caches to sync"))
        return
    }

    // Process items from WorkQueue
    wait.Until(func() {
        for processNextItem(queue) {
        }
    }, time.Second, stopCh)
}

func processNextItem(queue workqueue.RateLimitingInterface) bool {
    key, quit := queue.Get()
    if quit {
        return false
    }
    defer queue.Done(key)

    // Process the item
    fmt.Printf("Processing key: %s\n", key)
    queue.Forget(key)
    return true
}
```

A taste of GO

- the go struct for a Pod

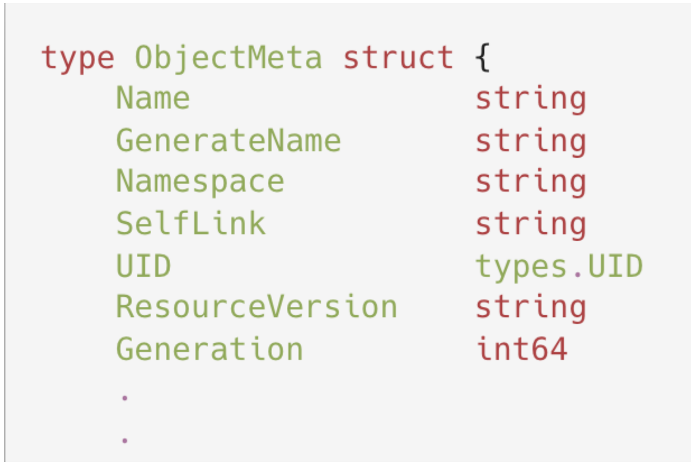


```go
type Pod struct {
        metav1.TypeMeta
        metav1.ObjectMeta
        Spec    PodSpec
        Status  PodStatus
    }
```



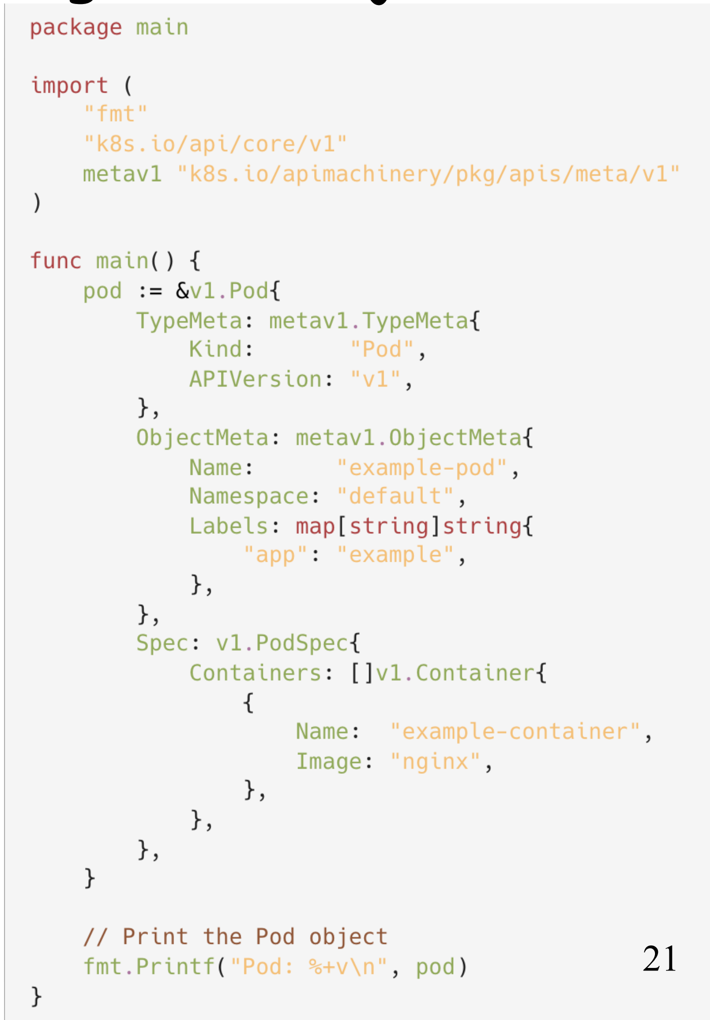
```go
type TypeMeta struct {
        Kind        string
        APIVersion  string
    }
```

- metav1: the API machinery package that provides metadata for API objects



```go
type ObjectMeta struct {
    Name                string
    GenerateName        string
    Namespace           string
    SelfLink            string
    UID                 types.UID
    ResourceVersion     string
    Generation          int64
    .
    .
```

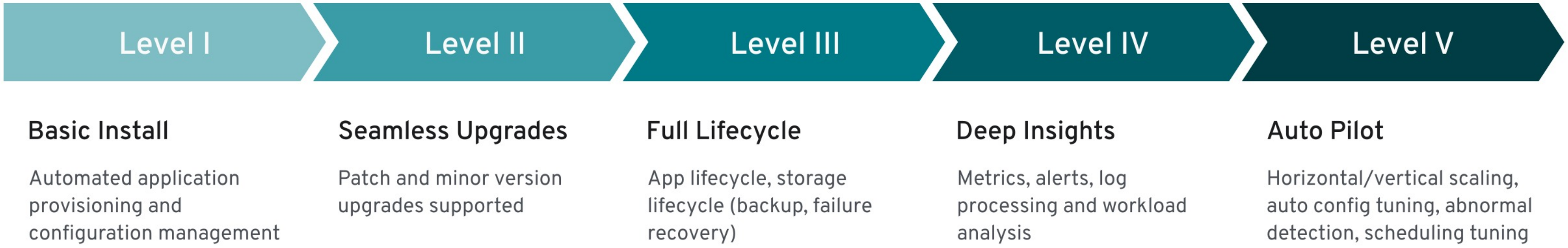
- defining a Pod object



```go
package main

import (
    "fmt"
    "k8s.io/api/core/v1"
    metav1 "k8s.io/apimachinery/pkg/apis/meta/v1"
)

func main() {
    pod := &v1.Pod{
        TypeMeta: metav1.TypeMeta{
            Kind:       "Pod",
            APIVersion: "v1",
        },
        ObjectMeta: metav1.ObjectMeta{
            Name:      "example-pod",
            Namespace: "default",
            Labels: map[string]string{
                "app": "example",
            },
        },
        Spec: v1.PodSpec{
            Containers: []v1.Container{
                {
                    Name:  "example-container",
                    Image: "nginx",
                },
            },
        },
    }

    // Print the Pod object
    fmt.Printf("Pod: %+v\n", pod)
}
```

The Operator Pattern

- What is an Operator?
  - *"An operator is a K8s controller that understands 2 domains: K8s and something else. It can automate tasks that usually require a human operator that understands both domains" - CNCF*
- CNCF Operator White Paper - Final Version
  - https://github.com/cncf/tag-app-delivery/blob/main/operator-wg/whitepaper/Operator-WhitePaper_v1-0.md
- The registry for Kubernetes Operators
  - https://operatorhub.io/
  - 5-level operator maturity model



Level I

Basic Install

Automated application provisioning and configuration management

Level II

Seamless Upgrades

Patch and minor version upgrades supported

Level III

Full Lifecycle

App lifecycle, storage lifecycle (backup, failure recovery)

Level IV

Deep Insights

Metrics, alerts, log processing and workload analysis

Level V

Auto Pilot

Horizontal/vertical scaling, auto config tuning, abnormal detection, scheduling tuning

Controllers vs Operators

- Controllers
  - A simple reconciliation process that monitors and acts on standard K8s resources. More often, they enhance platform behavior and add new platform features.
- Operators
  - A sophisticated reconciliation process that interacts with `CustomResourceDefinitions` (CRDs). Typically, these Operators encapsulate complex application domain logic and manage the full application lifecycle.

The Operator Pattern - how to build?

- Use `client-go` library directly
- Kubebuilder
  - Owned and maintained by the K8S SIG API Machinery, a tool and set of libs
- The Operator SDK
  - From CoreOS/Red Hat
- Metacontroller, KUDO and etc.