Hands-on Labs

Étendre l'API Kubernetes avec vos propres services

20-04-2018

Déroulement du workshop

- Présentation
- 1ère partie: Vue d'ensemble de l'API Kubernetes
- 2e partie: création d'une Custom Resource Definition (CRD)
- Pause
- 3è partie: création d'un API Server
- Questions

Slides et Repository Github du workshop

- Dans votre $GOPATH/src/github.com/simonferquel:
- git clone https://github.com/simonferquel/devoxx-2018-k8s-workshop.git
- Les slides sont à la racine

Disclaimer: Nous n'avons pas choisi les couleurs du template devoxx...

Intervenants

- Simon Ferquel@Docker Paris
  - https://github.com/simonferquel

- Jean-Christophe Sirot@Docker Paris
  - https://github.com/jcsirot

- Silvin Lubecki@Docker Paris
  - https://github.com/silvin-lubecki

But du workshop

- Comprendre comment fonctionne l'API Kubernetes et comment jouer avec
- Comprendre les différents points d'extension de Kubernetes
- Dérouler un exemple en fil conducteur
  - Déployer un *etcd* en Haute Disponibilité
- Non-but: déployer vraiment un etcd en Haute Disponibilité

Pré-requis

- Docker for Desktop + kubernetes / ou Minikube
- Installer golang 1.10
- Ajouter $GOPATH/bin à votre $PATH
- Familier avec
  - Golang
  - Docker
  - Kubernetes

Première partie: API Kubernetes

Disclaimer:

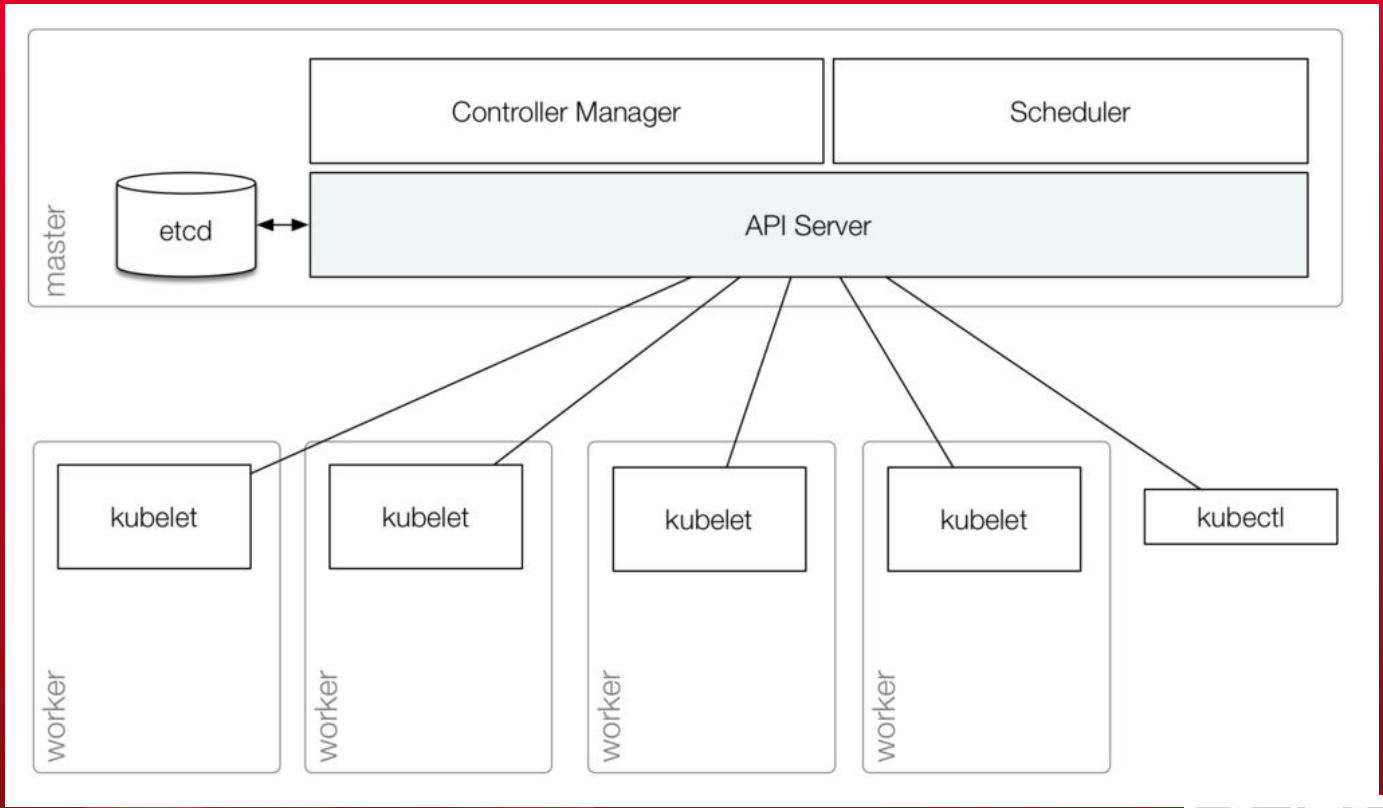
Fortement inspiré des posts https://blog.openshift.com/kubernetes-deep-dive-api-server-part-1/

(très complet sur le sujet)

Ressources Kubernetes

- Ressources Kubernetes standards
  - Pod
  - Deployment
  - ...
- Déclaratif (vs impératif)
- Persistant dans etcd
- Accessibles via une API REST Json (CRUD)
  - Utilisé par des Controllers k8s
    - boucle de réconciliation
    - un controller est un simple client k8s
  - Par des clients k8s (ex: kubectl)
- Client go: https://github.com/kubernetes/client-go

Architecture de l'API



Structure d'une ressource k8s

- TypeMeta
  - Kind
  - APIVersion
- ObjectMeta
  - Name
  - Labels
  - Annotations
  - OwnerReferences
  - ...
- Spec (état voulu)
- Status (état courant)
- Certains champs sont définis par l'utilisateur à la création, d'autres sont remplis par les controllers

```
type ReplicationController struct {
    metav1.TypeMeta
    metav1.ObjectMeta
    Spec ReplicationControllerSpec
    Status ReplicationControllerStatus
}
```

```
type ReplicationControllerSpec struct {
      Replicas *int32
      MinReadySeconds int32
      Selector map[string]string
      Template *PodTemplateSpec
}
```

```
type ReplicationControllerStatus struct {
      Replicas int32
      FullyLabeledReplicas int32
      ReadyReplicas int32
      AvailableReplicas int32
      ObservedGeneration int64
      Conditions []ReplicationControllerCondition
}
```

Versioning de ressource

- Version
  - Alpha: non activé par défaut, fortement déconseillé pour la prod
  - Beta: activé par défaut, déconseillé pour la prod
  - Stable 👍

Jouons avec l'API 1/3

```
# Dans un term à part, pour accéder à l'API kubernetes
$ kubectl proxy
Starting to serve on 127.0.0.1:8001

# Pour lister les groupes d'APIs disponibles avec leurs versions
$ curl http://127.0.0.1:8001/apis
{
"kind": "APIGroupList",
"apiVersion": "v1",
"groups": [
...
{
    "name": "batch",
    "versions": [
     {
       "groupVersion": "batch/v1",
       "version": "v1"
     },
     {
       "groupVersion": "batch/v1beta1",
       "version": "v1beta1"
     }
    ],
    "preferredVersion": {
      "groupVersion": "batch/v1",
      "version": "v1"
    },
    "serverAddressByClientCIDRs": null
  },
...
```

Jouons avec l'API 2/3

```
# Pour lister les ressources disponibles dans un groupe et une version particulière
$ curl http://127.0.0.1:8001/apis/batch/v1
{
  "kind": "APIResourceList",
  "apiVersion": "v1",
  "groupVersion": "batch/v1",
  "resources": [
    {
      "name": "jobs",
      "singularName": "",
      "namespaced": true,
      "kind": "Job",
      "verbs": [
        "create",
        "delete",
...
# Listons tous les Jobs
$ curl http://127.0.0.1:8001/apis/batch/v1/jobs
{
  "kind": "JobList",
  "apiVersion": "batch/v1",
  "metadata": {
    "selfLink": "/apis/batch/v1/jobs",
    "resourceVersion": "171071"
  },
  "items": []
}
# Attention, pour les Pods c'est un autre schéma (c'est legacy)!
$ curl http://127.0.0.1:8001/api/v1/pods
...
# Avec un namespace
$ curl http://127.0.0.1:8001/api/v1/namespaces/default/pods
...
```

Jouons avec l'API 3/3

```
# On va maintenant créer un Pod dans le namespace default
$ curl -H "Content-Type: application/json" -X POST -d
'{"metadata":{"name":"webserver","namespace":"default"},"spec":{"containers":[{"name":"nginx","image":"nginx:1.9"}],"ports":[{"containerPort":80,"protocol":"TCP"
}]}}' http://localhost:8001/api/v1/namespaces/default/pods
# retourne la ressource Pod créée
{
  "kind": "Pod",
  "apiVersion": "v1",
  "metadata": {
    "name": "webserver",
    "namespace": "default",
    "selfLink": "/api/v1/namespaces/default/pods/webserver",
...

# Vérifions avec kubectl
$ kubectl get pods
NAME        READY   STATUS              RESTARTS   AGE
webserver   0/1     ContainerCreating   0          9s

# EXERCICE: Essayer de supprimer ce pod avec un curl, et sinon kubectl y arrivera...
```

Deuxième partie

Custom Resource Definition

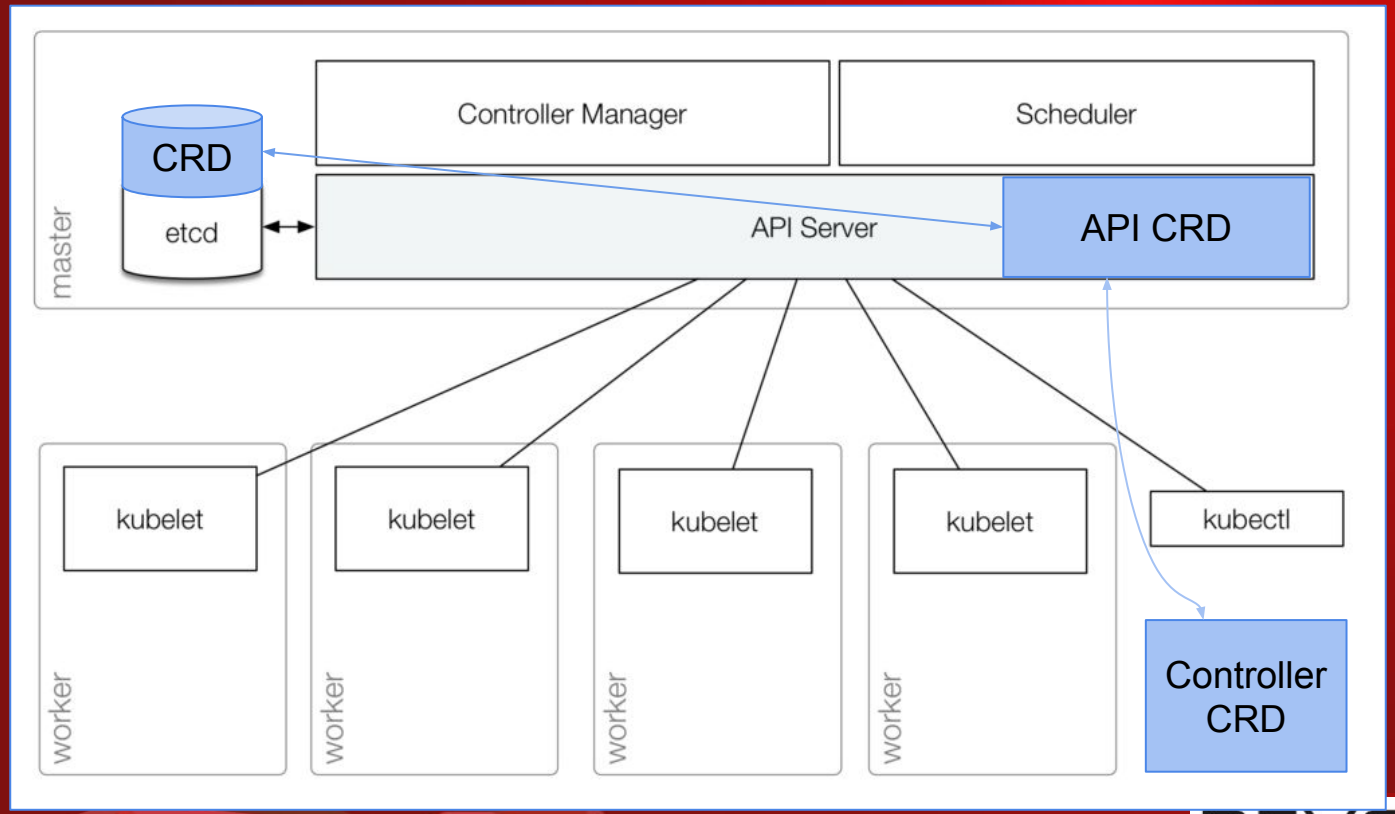
Qu’est-ce qu’une CRD?

- Premier point d’extension de Kubernetes, le plus simple
- Utilise le même etcd que Kubernetes
- Utilise le même API Server que Kubernetes
- API de la ressource définie dans un YAML, déployée par kubectl
- Structure de la ressource définie en Go
- Attention aucune validation de la structure, l'api server central ne connait rien de notre type
- Génération du code du client à partir du type Go + annotations
  - API CRUD
  - Listers (pour lister les entités)
  - Watchers (pour écouter les évènements sur les entités)
  - Informers (système de cache sur l’état des entités)
  - DeepCopy
  - Fake
  - ...

Rôle du controller

- Le plus simple possible
- Simple client de l'API Kubernetes d'une ressource
  - Ecoute les évènements Create/Update/Delete
  - Cas particulier de Kubelet (réconcilie les PODs en containers)
- Boucle de réconciliation (ex: toutes les 30s)
  - Tente de concilier l'état courant avec l'état voulu
  - Ex: ReplicaSet crée ou supprime des Pods si un scale a eu lieu ou si un Pod a quitté prématurément

Architecture CRD



Déclarons notre CRD à Kubernetes 1/2

```
# Step 1

$ cd devoxx-2018-k8s-workshop

$ git checkout step-1

$ cat k8s-assets/crd.yml
apiVersion: apiextensions.k8s.io/v1beta1
kind: CustomResourceDefinition
metadata:
  # name must match the spec fields below, and be in the form: <plural>.<group>
  name: etcdinstances.etcdaas.devoxx2018
spec:
  # group name to use for REST API: /apis/<group>/<version>
  group: etcdaas.devoxx2018
  # version name to use for REST API: /apis/<group>/<version>
  version: v1alpha1
  # either Namespaced or Cluster
  scope: Namespaced
  names:
    # plural name to be used in the URL: /apis/<group>/<version>/<plural>
    plural: etcdinstances
    # singular name to be used as an alias on the CLI and for display
    singular: etcdinstance
    # kind is normally the CamelCased singular type. Your resource manifests use this.
    kind: ETCDInstance
    # shortNames allow shorter string to match your resource on the CLI
    shortNames:
    - etcd
```

Déclarons notre CRD à Kubernetes 2/2

```
# C'est parti, on envoie tout à kubernetes
$ kubectl apply -f  k8s-assets/crd.yml
customresourcedefinition "etcdinstances.etcdaas.devoxx2018" configured

# Liste de toutes les API disponibles
$ kubectl api-versions
...
etcdaas.devoxx2018/v1alpha1
...

# On peut lister nos instances etcd
$ kubectl get etcdinstances
No resources found.

# Création d'une simple instance
$ cat k8s-assets/etcdinstance-sample.yml
apiVersion: etcdaas.devoxx2018/v1alpha1
kind: ETCDInstance
metadata:
  name: myinstance
$ kubectl create -f k8s-assets/etcdinstance-sample.yml
etcdinstance "myinstance" created
# Tadaaaa
$ kubectl get etcdinstance
NAME           AGE
myinstance   0s

# Ne pas oublier de la supprimer
$ kubectl delete etcdinstance myinstance
etcdinstance "myinstance" deleted

# EXERCICE: jouer un peu avec ce que l'on vient de voir
```

Mise en place de la CRD 1/2

- Ajouter un package pkg/apis/etcdaas/v1alpha1
- Définition de notre type ETCDInstance dans types.go
  - Ne pas oublier les “annotations” pour le générateur de code
  - TypeMeta
  - ObjectMeta
  - Spec
  - Status
- Doc.go
  - // +k8s:deepcopy-gen=package
- Ajout d’un type ETCDInstanceList
- Définition de notre spec
  - Replicas int
  - WithTLSBundle
- Ajout d’un type Status

Définition de la struct CRD 1/2

pkg/apis/etcdass/v1alpha1/types.go

```go
package v1alpha1

import metav1 "k8s.io/apimachinery/pkg/apis/meta/v1"

// +genclient
// +k8s:deepcopy-gen:interfaces=k8s.io/apimachinery/pkg/runtime.Object
type ETCDInstance struct {
    metav1.TypeMeta   `json:",inline"`
    metav1.ObjectMeta `json:"metadata,omitempty"`
    Spec              ETCDInstanceSpec   `json:"spec,omitempty"`
    Status            ETCDInstanceStatus `json:"status,omitempty"`
}

type ETCDInstanceSpec struct {
    Replicas      int  `json:"replicas,omitempty"`
    WithTLSBundle bool `json:"with_tls_bundle,omitempty"`
}
```

Définition de la struct CRD 2/2

pkg/apis/etcdass/v1alpha1/types.go

```
type ETCDInstanceState string

const (
    ETCDNone       = ETCDInstanceState("")
    ETCDDeploying  = ETCDInstanceState("deploying")
    ETCDRunning    = ETCDInstanceState("running")
    ETCDFailed     = ETCDInstanceState("failed")
)

type ETCDInstanceStatus struct {
    State   ETCDInstanceState  `json:"state,omitempty"`
    Message string             `json:"message,omitempty"`
}

// +k8s:deepcopy-gen:interfaces=k8s.io/apimachinery/pkg/runtime.Object
type ETCDInstanceList struct {
    metav1.TypeMeta  `json:",inline"`
    metav1.ListMeta  `json:"metadata,omitempty"`
    Items           []ETCDInstance  `json:"items"`
}
```

Mise en place de la CRD 2/2

Register.go

- Déclare les types à Kubernetes via le SchemeBuilder
- Déclare le groupe d'API et la version ("etcdaas.devoxx2018" et "v1alpha1"

```
var (
	SchemeBuilder = runtime.NewSchemeBuilder(addKnownTypes)
	AddToScheme   = SchemeBuilder.AddToScheme
)
func addKnownTypes(scheme *runtime.Scheme) error {
	scheme.AddKnownTypes(SchemeGroupVersion,
		&ETCDInstance{},
		&ETCDInstanceList{},
	)
	scheme.AddKnownTypes(SchemeGroupVersion, &metav1.Status{})
	metav1.AddToGroupVersion(scheme, SchemeGroupVersion)
	return nil
}
```

Génération du code client

- Build une image docker avec les générateurs
- Monte votre répertoire courant en volume
- Lance la génération du code à partir de la définition de notre struct CRD
- Génère le code client
  - Listers
  - Informers
  - Deepcopy
  - Fake
  - ...

Génération du code client

```
# Step 2
# Heureusement il y a une image toute prête pour ça
$ cat tools/generators/Dockerfile
FROM golang:1.10
RUN go get -u k8s.io/code-generator/cmd/client-gen \
&&  go get -u k8s.io/code-generator/cmd/conversion-gen \
&&  go get -u k8s.io/code-generator/cmd/deepcopy-gen \
&&  go get -u k8s.io/code-generator/cmd/defaulter-gen \
&&  go get -u k8s.io/code-generator/cmd/go-to-protobuf \
&&  go get -u k8s.io/code-generator/cmd/import-boss \
&&  go get -u k8s.io/code-generator/cmd/informer-gen \
&&  go get -u k8s.io/code-generator/cmd/lister-gen \
&&  go get -u k8s.io/code-generator/cmd/openapi-gen \
&&  go get -u k8s.io/code-generator/cmd/set-gen
WORKDIR /go/src/github.com/simonferquel/devoxx-2018-k8s-workshop

$ make generator-image
...
Successfully built 9d1cb924ce02
Successfully tagged k8s-generators:latest

# On peut lancer la génération du code
$ make generate-all
...
# EXERCICE:
# Le répertoire pkg/client est créé, jetez-y un oeil
# Essayez de modifier la Spec et regénérez le code
```

Premier client: CLI

- Utilisation du code client généré pour
  - Créer un etcd
  - Lister nos etcd
  - Supprimer un etcd
- Créer un nouveau package cmd/etcdaas-cli
- Nouveau fichier main.go

CLI - Parser les flags 1/3

cmd/etcdaas-cli/main.go

```
import (
...
    types "github.com/simonferquel/devoxx-2018-k8s-workshop/pkg/apis/etcdaas/v1alpha1"
    client
"github.com/simonferquel/devoxx-2018-k8s-workshop/pkg/client/clientset/versioned/typed/etcdaas/v1alpha1"
    metav1 "k8s.io/apimachinery/pkg/apis/meta/v1"
    "k8s.io/client-go/tools/clientcmd"
)

func main() {
    var create, list, delete bool
    var name string
    var namespace string
    var replicas int
    var tls bool
    flag.BoolVar(&create, "create", false, "create an etcd")
    flag.BoolVar(&list, "list", false, "list etcds")
    flag.BoolVar(&delete, "delete", false, "delete an etcd")
    flag.BoolVar(&tls, "enable-tls", false, "enable tls")
    flag.StringVar(&name, "name", "", "etcd name")
    flag.StringVar(&namespace, "namespace", "default", "k8s namespace")
    flag.IntVar(&replicas, "replicas", 1, "etcd replicas")
    flag.Parse()
```

CLI - Config Kubernetes 2/3

cmd/etcdaas-cli/main.go

```
home := os.Getenv("HOME")
if home == "" {
    home = os.Getenv("HOMEDRIVE") + os.Getenv("HOMEPATH")
}
cfg, err := clientcmd.BuildConfigFromFlags("", filepath.Join(home, ".kube", "config"))
if err != nil {
    panic(err)
}
c := client.NewForConfigOrDie(cfg)
```

CLI - Commandes 3/3

cmd/etcdaas-cli/main.go

```
        switch {
        case create:
            i := &types.ETCDInstance{
                ObjectMeta: metav1.ObjectMeta{
                    Name: name,
                },
                Spec: types.ETCDInstanceSpec{
                    Replicas:      replicas,
                    WithTLSBundle: tls,
                },
            }
            _, err := c.ETCDInstances(namespace).Create(i)
            if err != nil {
                panic(err)
            }
        case list:
            lst, err := c.ETCDInstances(namespace).List(metav1.ListOptions{})
            if err != nil {
                panic(err)
            }
            for _, item := range lst.Items {
                fmt.Printf("%s:\n\tspec: %#v,\n\tstatus: %#v\n", item.Name, item.Spec, item.Status)
            }
        case delete:
            if err := c.ETCDInstances(namespace).Delete(name, nil); err != nil {
                panic(err)
            }
        }
    }
```

CLI - Build and Run

```
# Step 4 ou Step 3 (avec juste le code client généré)
# Builder le tout
$ make bin/etcdaas-cli
go build -i -o ./bin/etcdaas-cli ./cmd/etcdaas-cli

# On tente de créer une instance etcd
$ ./bin/etcdaas-cli -create -name my-super-etcd

# kubectl peut lister nos instances
$ kubectl get etcdinstances
NAME              AGE
my-super-etcd   5s

# et les inspecter
$ kubectl get etcdinstance my-super-etcd -o yaml
apiVersion: etcdaas.devoxx2018/v1alpha1
kind: ETCDInstance
metadata:
  clusterName: ""
  creationTimestamp: 2018-04-19T11:41:38Z
  name: my-super-etcd
  namespace: default
  resourceVersion: "228055"
  selfLink: /apis/etcdaas.devoxx2018/v1alpha1/namespaces/default/etcdinstances/my-super-etcd
  uid: 9f1368cf-43c6-11e8-9a00-025000000001
spec:
  replicas: 1
status: {}

# EXERCICE: A vous de jouer avec -list et -delete, les namespaces... !
$ ./bin/etcdaas-cli -list
$ ./bin/etcdaas-cli -delete -name my-super-etcd
$ ./bin/etcdaas-cli -create -name namespaced-etcd -namespace ....
```

Ajout du controller

- Ecoute les évènements kubernetes liés aux instances Etcd
  - Evènements create/update/delete
  - rôle des informers de synchroniser tout
- Réconcilie régulièrement l'état courant avec l'état désiré exprimé via la Spec de notre Etcd
- Ajout d'un package cmd/etcdaas-controller
- Implémenter l'interface et afficher les objets dans la console
  - OnAdd(obj interface{})
  - OnUpdate(oldObj, newObj interface{})
  - OnDelete(obj interface{})

Controller - Flags et configuration 1/4

cmd/etcdaas-controller/main.go

```go
package main

import (
…
    types "github.com/simonferquel/devoxx-2018-k8s-workshop/pkg/apis/etcdaas/v1alpha1"
    clientset "github.com/simonferquel/devoxx-2018-k8s-workshop/pkg/client/clientset/versioned"
    informers "github.com/simonferquel/devoxx-2018-k8s-workshop/pkg/client/informers/externalversions"
    restclient "k8s.io/client-go/rest"
    "k8s.io/client-go/tools/clientcmd"
)

func main() {
    var kubeconfig string
    flag.StringVar(&kubeconfig, "kubeconfig", "", "kubeconfig path (keep unset for using ambient config)" )
    flag.Parse()

    cfg, err := clientcmd.BuildConfigFromFlags("", kubeconfig)
    if err != nil {
        panic(err)
    }

    c := clientset.NewForConfigOrDie(cfg)
    informerFactory := informers.NewSharedInformerFactory(c, time.Minute)
    informer := informerFactory.Etcdaas().V1alpha1().ETCDInstances().Informer()

    ctx := context.Background()
    ctr := &controller{config: cfg, client: c}
    informer.AddEventHandler(ctr)
    informer.Run(ctx.Done())
}
```

Controller - OnDelete 2/4

cmd/etcdaas-controller/main.go

```
type controller struct {
    config *restclient.Config
    client clientset.Interface
}

func (c *controller) OnDelete(obj interface{}) {
    etcd, ok := obj.(*types.ETCDInstance)
    if !ok {
        panic("unexpected object")
    }
    fmt.Printf("OnDelete:\n %#v\n", *etcd)
}
```

Controller - OnAdd 3/4

cmd/etcdaas-controller/main.go

```
func (c *controller) OnAdd(obj interface{}) {
    etcd, ok := obj.(*types.ETCDInstance)
    if !ok {
        panic("unexpected object")
    }
    fmt.Printf("OnAdd:\n %#v\n", *etcd)

    etcd.Status = types.ETCDInstanceStatus{
        State:   types.ETCDDeploying,
        Message: "deploying",
    }
    etcd, _ = c.client.EtcdaasV1alpha1().ETCDInstances(etcd.Namespace).Update(etcd)

    time.Sleep(10 * time.Second)

    etcd.Status = types.ETCDInstanceStatus{
        State:   types.ETCDRunning,
        Message: "deployment successfull",
    }
    c.client.EtcdaasV1alpha1().ETCDInstances(etcd.Namespace).Update(etcd)
}
```

Controller - OnUpdate 4/4

cmd/etcdaas-controller/main.go

```
func (c *controller) OnUpdate(oldObj, newObj interface{}) {
    oldetcd, ok := oldObj.(*types.ETCDInstance)
    if !ok {
        panic("unexpected object")
    }
    newetcd, ok := newObj.(*types.ETCDInstance)
    if !ok {
        panic("unexpected object")
    }
    fmt.Printf("OnUpdate:\n %#v\n to\n %#v\n", *oldetcd, *newetcd)

    if oldetcd.Spec == newetcd.Spec {
        return // don't do anything
    }

    newetcd.Status = types.ETCDInstanceStatus{
        State:   types.ETCDDeploying,
        Message: "updating",
    }
    newetcd, _ = c.client.EtcdaasV1alpha1().ETCDInstances(newetcd.Namespace).Update(newetcd)

    time.Sleep(10 * time.Second)

    newetcd.Status = types.ETCDInstanceStatus{
        State:   types.ETCDRunning,
        Message: "update successfull",
    }
    c.client.EtcdaasV1alpha1().ETCDInstances(newetcd.Namespace).Update(newetcd)
}
```

Controller - Build and Run

```
# Step 5
# Builder le tout
$ make bin/etcdaas-controller
go build -i -o ./bin/etcdaas-controller ./cmd/etcdaas-controller

# Lancer le controller dans un nouveau terminal
$ ./bin/etcdaas-controller -kubeconfig ~/.kube/config
...

# Inspecter l'instance, le status est rempli
$ kubectl describe etcd my-super-etcd
Name:         my-super-etcd
Namespace:    default
Labels:       <none>
Annotations:  <none>
API Version:  etcdaas.devoxx2018/v1alpha1
Kind:         ETCDInstance
Metadata:
  Cluster Name:
  Creation Timestamp:  2018-04-19T11:41:38Z
  Generation:          0
  Resource Version:    232133
  Self Link:           /apis/etcdaas.devoxx2018/v1alpha1/namespaces/default/etcdinstances/my-super-etcd
  UID:                 9f1368cf-43c6-11e8-9a00-025000000001
Spec:
  Replicas:  1
Status:
  Message:  deployment successfull
  State:    running

# EXERCICE: Ajouter/Supprimer des instances avec etcdaas-cli, les inspecter, et regarder les logs du controller
```

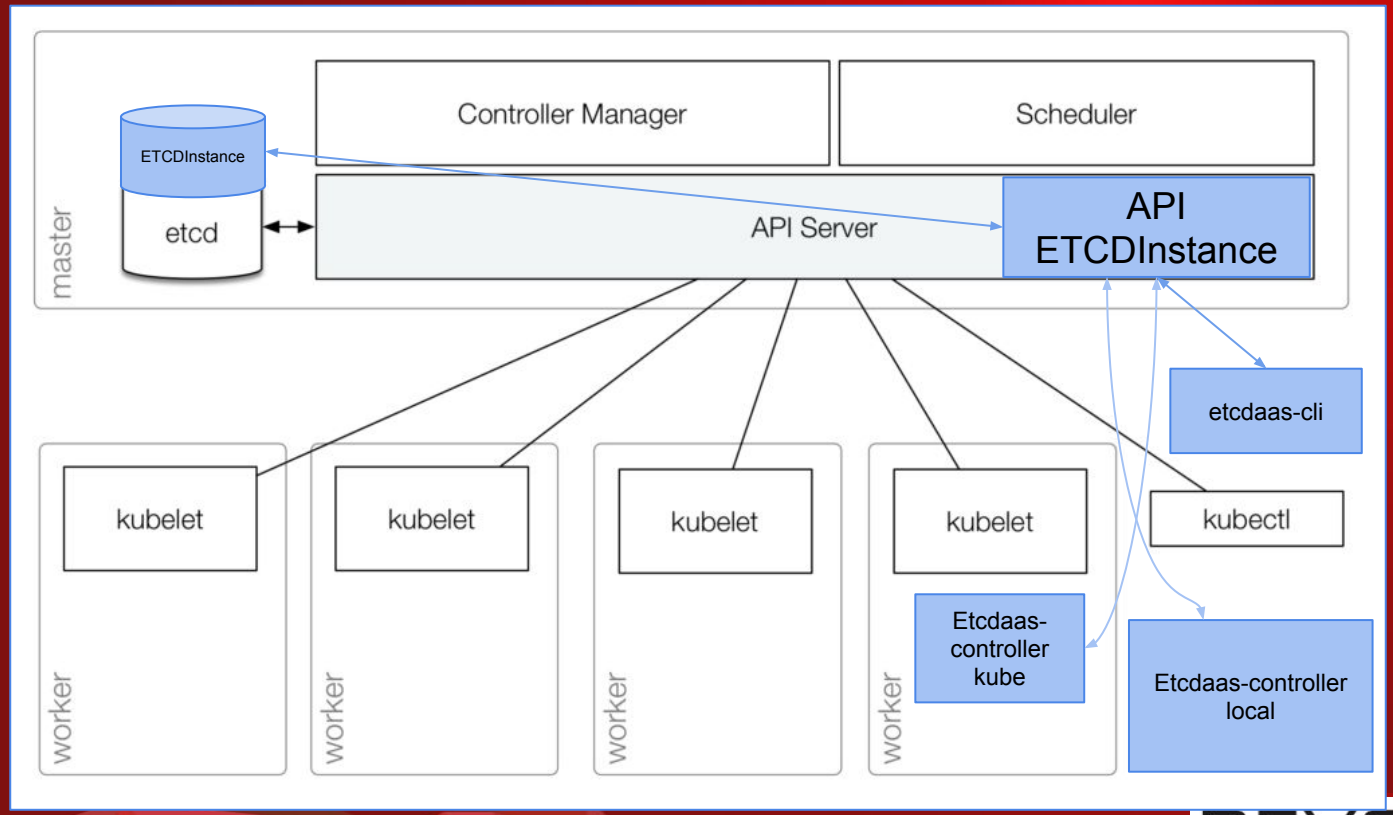
Installation du controller dans Kubernetes

```
# Step 6
# Builder une image
$ make image/etcdaas-controller
...
Successfully tagged etcdaas-controller:latest

# Ajout d'un ServiceAccount et d'un ClusterRoleBinding
$ kubectl apply -f k8s-assets/sa.yml
serviceaccount "etcdaas" created
clusterrolebinding "etcdaas:clusteradmin" created

# !! Ne pas oublier de tuer le controller qui tourne dans le term

# Ajout d'un Deployment pour deployer notre image
$ kubectl apply -f k8s-assets/controller-deployment.yml
deployment "etcdaas-controller" created

# On vérifie que c'est bien déployé
$ kubectl get pods
NAME                                  READY   STATUS    RESTARTS   AGE
etcdaas-controller-5fb66d74dd-ls4z9   1/1     Running   0          47s

# Suivre les logs du controller
$ kubectl logs -f etcdaas-controller-5fb66d74dd-ls4z9

# EXERCICE: vérifier que tout fonctionne comme avant avec le
# controller en local
```

```
apiVersion: apps/v1
kind: Deployment
metadata:
 name: etcdaas-controller
 labels:
   app: etcdaas-controller
spec:
 replicas: 1
 selector:
   matchLabels:
     app: etcdaas-controller
 template:
   metadata:
     labels:
       app: etcdaas-controller
   spec:
     serviceAccount: etcdaas
     containers:
     - name: etcdaas-controller
       image:
etcdaas-controller
       imagePullPolicy: Never
```

```
kind: ServiceAccount
apiVersion: v1
metadata:
 name: etcdaas
 namespace: default
---
apiVersion:
rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
 name: etcdaas:clusteradmin
roleRef:
 apiGroup: rbac.authorization.k8s.io
 kind: ClusterRole
 name: ClusterAdmin
subjects:
- kind: ServiceAccount
 name: etcdaas
 namespace: default
```

Récapitulatif



Pause

Troisième partie

API Server

Limitations de la CRD

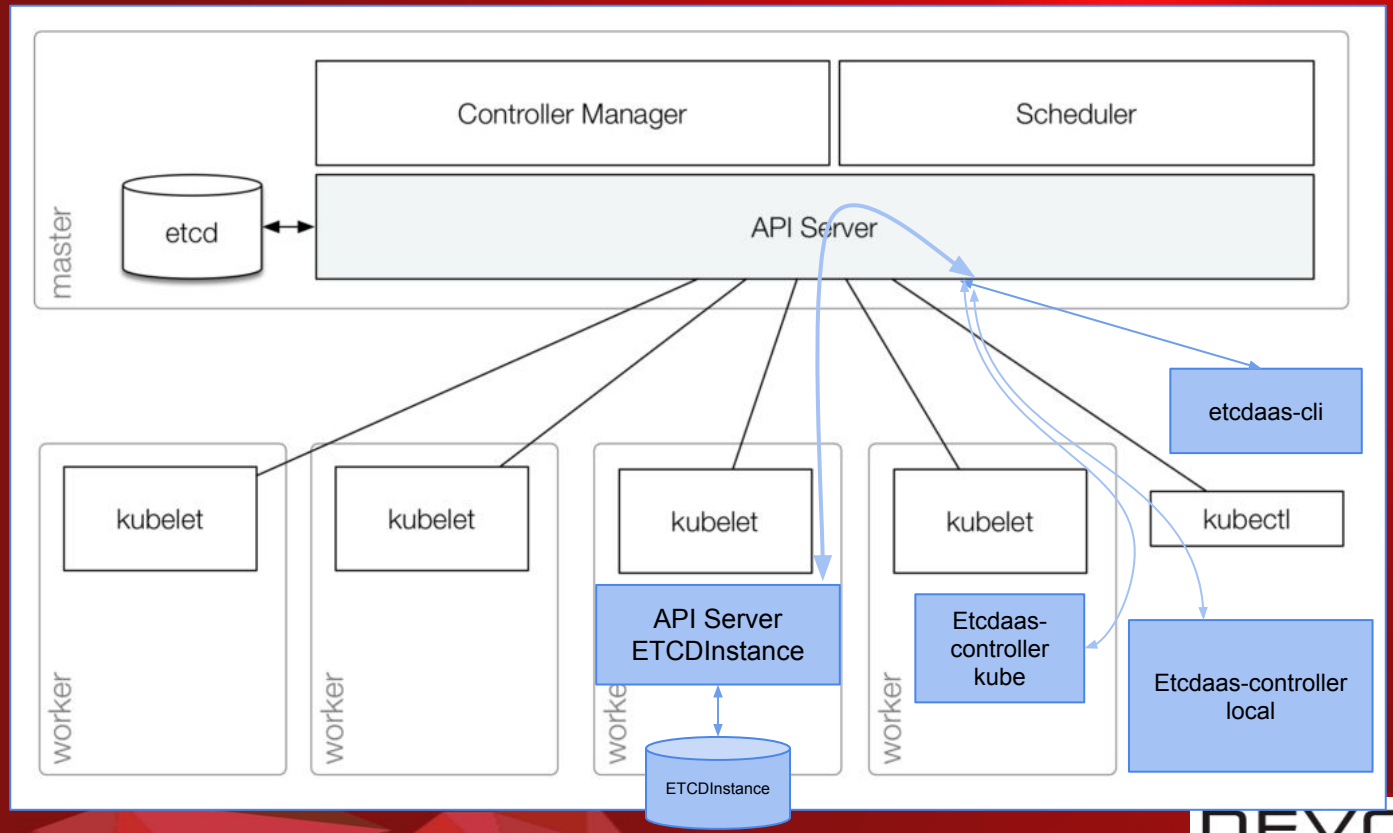
- CRD très pratique et rapide à développer
- Mais vite limitée si l'on veut enrichir notre API
  - Pas de subressource possible (Ex: on veut ajouter une commande de backup etcd ou un scale)
  - Ne supporte pas plusieurs versions
  - Pas de validation de la CRD en amont, un client peut poster n'importe quoi

API Server

- Serveur d'API custom:
  - on ne profite pas de celui de Kubernetes
  - Persistance à gérer soi-même (etcd à part, potentiel problème de Haute Disponibilité)
- Beaucoup plus lourd à mettre en place
  - TLS / enregistrement de l'API sécurisée
  - RBAC
- Mais customisable à souhait
  - admission/validation
  - Conversion entre plusieurs versions
  - Ajout de subressources

```
v1beta1 ⇒ internal ⇒    |    ⇒       |    ⇒  v1  ⇒ json/yaml ⇒ etcd
                    admission    validation
```

Architecture API-Server



API-Server

- Nouveau binaire etcdaas-api
- Parcours du code directement, trop long/compliqué à faire pas-à-pas, beaucoup de boilerplate code
- Package apiserver
  - Ajout d'une représentation interne "internalversion"
  - Pour simplifier ici on réutilise le même type que v1alpha1
  - L'API-Server ne manipule que des internalversion
  - Déclarer l'internalversion à kubernetes
- API-Server sur kubernetes
  - On triche et on utilise le même etcd que docker-for-desktop

API Server - Build and Run 1/2

```
# Step 7
# Builder l'image de l'api-server
$ make image/etcdaas-api
...
Successfully tagged etcdaas-api:latest

# Supprimer la CRD, sinon tout continuera d'aller vers elle
$ kubectl delete -f k8s-assets/crd.yml
customresourcedefinition "etcdinstances.etcdaas.devoxx2018" deleted

# Deployer l'API Server sur Kubernetes (crée un Service, un Deployment et APIService
$ kubectl apply -f k8s-assets/api-deployment.yml
deployment "etcdaas-api" created
service "etcdaas-api" created
apiservice "v1alpha1.etcdaas.devoxx2018" created

# S'assurer que tout est déployé
$ kubectl get pods
NAME                                READY   STATUS    RESTARTS   AGE
etcdaas-api-9b9cf499-9gc5m          1/1     Running   0          14s
etcdaas-controller-5fb66d74dd-ls4z9 1/1     Running   0          1h

$ kubectl get service
NAME          TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)   AGE
etcdaas-api   ClusterIP   10.107.189.226   <none>        443/TCP   22s
```

API Server - Build and Run 2/2

```
# Creation d'une instance
$ ./bin/etcdaas-cli -create -name my-super-etcd -replicas 2

# Afficher les logs de l'API-server dans un nouveau terminal
$ kubectl logs -f etcdaas-api-9b9cf499-9gc5m
...
PrepareForCreate my-super-etcd spec: v1alpha1.ETCDInstanceSpec{Replicas:2, WithTLSBundle:false}, status: v1alpha1.ETCDInstanceStatus{State:"",
Message:""}
Validate my-super-etcd spec: v1alpha1.ETCDInstanceSpec{Replicas:2, WithTLSBundle:false}, status: v1alpha1.ETCDInstanceStatus{State:"", Message:""}
Canonicalize my-super-etcd spec: v1alpha1.ETCDInstanceSpec{Replicas:2, WithTLSBundle:false}, status: v1alpha1.ETCDInstanceStatus{State:"", Message:""}
PrepareForUpdate my-super-etcd spec: v1alpha1.ETCDInstanceSpec{Replicas:2, WithTLSBundle:false}, status: v1alpha1.ETCDInstanceStatus{State:"deploying",
Message:"deploying"}
ValidateUpdate my-super-etcd spec: v1alpha1.ETCDInstanceSpec{Replicas:2, WithTLSBundle:false}, status: v1alpha1.ETCDInstanceStatus{State:"deploying",
Message:"deploying"}
Canonicalize my-super-etcd spec: v1alpha1.ETCDInstanceSpec{Replicas:2, WithTLSBundle:false}, status: v1alpha1.ETCDInstanceStatus{State:"deploying",
Message:"deploying"}
PrepareForUpdate my-super-etcd spec: v1alpha1.ETCDInstanceSpec{Replicas:2, WithTLSBundle:false}, status: v1alpha1.ETCDInstanceStatus{State:"running",
Message:"deployment successfull"}
ValidateUpdate my-super-etcd spec: v1alpha1.ETCDInstanceSpec{Replicas:2, WithTLSBundle:false}, status: v1alpha1.ETCDInstanceStatus{State:"running",
Message:"deployment successfull"}
Canonicalize my-super-etcd spec: v1alpha1.ETCDInstanceSpec{Replicas:2, WithTLSBundle:false}, status: v1alpha1.ETCDInstanceStatus{State:"running",
Message:"deployment successfull"}
```

Ajout d'une subresource backup

- Nouvelle registry "backup registry" qui va servir les requêtes HTTP sur /backup
- C'est un Storage Object, ici Connecter pour les "long running requests", permet de streamer des choses, mais d'autres sont disponibles
  - Watcher
  - Create/Update
  - ...
- On implemente l'interface rest.Connecter
  - NewConnectOptions -> pour déclarer à kube un objet Option passé au moment du connect
  - ConnectMethods -> []string{http.MethodGet}
  - Connect -> retourne un handler à implémenter
- On ajoute notre registry au v1alpha1storage["etcdinstances/backup"] (main.go)

Subresource Backup 1/2

cmd/etcdaas-api/registry/backupregistry.go

```
import (
...
    "github.com/simonferquel/devoxx-2018-k8s-workshop/pkg/apis/etcdaas/v1alpha1"
    "k8s.io/apimachinery/pkg/runtime"
    genericapirequest "k8s.io/apiserver/pkg/endpoints/request"
    "k8s.io/apiserver/pkg/registry/rest"
)

func NewBackupREST() rest.Storage {
    return &backupREST{}
}

type backupREST struct {}

var _ rest.Storage = &backupREST{}
var _ rest.Connecter = &backupREST{}

func (b *backupREST) New() runtime.Object {
    return &v1alpha1.ETCDInstance{}
}
```

Subresource Backup 2/2

cmd/etcdaas-api/registry/backupregistry.go

```go
func (b *backupREST) Connect(ctx genericapirequest.Context, id string, options runtime.Object, r rest.Responder)
(http.Handler, error) {
    ns, _ := genericapirequest.NamespaceFrom(ctx)
    fmt.Printf("received backup connect request for %s in namespace %s", id, ns)
    return &backupHandler{id: id, ns: ns}, nil
}

func (b *backupREST) NewConnectOptions() (runtime.Object, bool, string) {
    return nil, false, ""
}

func (b *backupREST) ConnectMethods() []string {
    return []string{http.MethodGet}
}

type backupHandler struct {
    id string
    ns string
}

func (h *backupHandler) ServeHTTP(w http.ResponseWriter, r *http.Request) {
    w.Write([]byte(fmt.Sprintf("this is a backup for %s/%s", h.ns, h.id)))
}
```

CLI - Backup

cmd/etcdaas-cli/main.go

```
func main() {
    var create, list, delete, backup bool
...
    flag.BoolVar(&backup, "backup", false, "backup an etcd")
...
    case backup:
        r, err := c.RESTClient().
            Get().
            Namespace(namespace).
            Resource("etcdinstances").
            Name(name).
            SubResource("backup").
            Stream()
        if err != nil {
            panic(err)
        }
        defer r.Close()
        io.Copy(os.Stdout, r)
        fmt.Println()
```

Backup - Build and Run

```
# Step 8
# Re-Builder l'image de l'api-server
$ make image/etcdaas-api
...
Successfully tagged etcdaas-api:latest

# Re-builder la CLI
$ make bin/etcdaas-cli
...

# Re-Deployer l'API Server sur Kubernetes
$ kubectl apply -f k8s-assets/api-deployment.yml
deployment "etcdaas-api" unchanged
service "etcdaas-api" unchanged
apiservice "v1alpha1.etcdaas.devoxx2018" configured

# Effectuer un backup
$ ./bin/etcdaas-cli -backup -name my-super-etcd
this is a backup for default/my-super-etcd

# EXERCICE: Essayer d'appeler la subresource backup via un curl
# Le premier qui réussit gagne un tshirt Docker :)

# Final Step
```

Conclusion

CRD:

- **Énormément de code généré**
- **Pas forcément de bonne qualité**
- **Pratique pour bootstrapper un projet.**

API Server:

- **Beaucoup plus customisable**
- **Beaucoup plus lourd à maintenir**

Pour Compose for Kubernetes, on a commencé par une CRD, puis tout refait à la main avec l'API Server.

Pour aller plus loin

Tous les sujets non traités:

- Authentification/Admission/Impersonation
- Upgrade de version de ressource/Migration

Les excellents posts d'OpenShift sur le sujet

- https://blog.openshift.com/kubernetes-deep-dive-api-server-part-1/
- https://blog.openshift.com/kubernetes-deep-dive-api-server-part-2/
- https://blog.openshift.com/kubernetes-deep-dive-api-server-part-3a/
- https://blog.openshift.com/kubernetes-deep-dive-code-generation-custom resources/

La doc Kubernetes

- https://kubernetes.io/docs/concepts/api-extension/custom-resources/
- https://kubernetes.io/docs/concepts/api-extension/apiserver-aggregation/

Questions?

Merci de nous avoir écouté!