
Artículo

[Eduardo Anglada](#) · 12 ago, 2021 Lectura de 13 min

Análisis en detalle del Operador InterSystems Kubernetes: Parte 2

En el [artículo anterior](#), vimos una forma de crear un operador personalizado que administra el estado de la instancia de IRIS. Esta vez, vamos a echar un vistazo a un operador listo para usar - el Operador InterSystems Kubernetes (IKO). [La documentación oficial](#) nos ayudará a recorrer los pasos de la implementación.

Requisitos previos

Para implementar IRIS, necesitamos un clúster de Kubernetes. En este ejemplo, utilizaremos Google Kubernetes Engine ([GKE](#)), por lo que tendremos que utilizar una cuenta de Google, configurar un proyecto de Google Cloud [e instalar las herramientas de línea de comandos gcloud y kubectl](#).

También necesitarás instalar la herramienta [Helm3](#):

```
$ helm version
version.BuildInfo{Version:"v3.3.4"...}
```

Nota: ten en cuenta que en [el nivel gratuito de Google](#), no todos los recursos son gratuitos.

En nuestro caso no importa el tipo de GKE que utilicemos: [zonal](#), [regional](#), o [privada](#). Después de crear uno, vamos a conectarnos al clúster. Hemos creado un clúster llamado "iko" en un proyecto llamado "iko-project". En el siguiente texto, utiliza el nombre de tu propio proyecto en vez de "iko-project".

Este comando añade el clúster a nuestra configuración de clústeres locales:

```
$ gcloud container clusters get-credentials iko --zone europe-west2-b --project iko-project
```

Instalar IKO

Vamos a implementar IKO en nuestro clúster recién creado. La forma recomendada de instalar paquetes en Kubernetes es usando Helm. IKO no es una excepción y se puede instalar como un gráfico de Helm. Elige [la versión 3 de Helm](#), ya que es más segura.

Descarga IKO desde la página [Componentes de InterSystems](#) del Centro de Soporte Internacional (WRC), creando una cuenta de desarrollador gratuita si aún no tienes una. En el momento de escribir este artículo, la última versión era 2.0.223.0.

Descarga y descomprime el archivo. Nos referiremos al directorio descomprimido como el directorio actual.

El gráfico se encuentra en el directorio *chart/iris-operator*. Si solo implementas este gráfico, recibirás un error al describir los contenedores implementados:

```
Failed to pull image "intersystems/iris-operator:2.0.0.223.0": rpc error: code = Unknown desc = Error response from daemon: pull access denied for intersystems/iris-operator, repository does not exist or may require 'docker login'.
```

Por lo tanto, es necesario hacer que una imagen IKO esté disponible desde el clúster Kubernetes. Primero, vamos a incorporar esta imagen al Google Container Registry:

```
$ docker load -i image/iris_operator-2.0.0.223.0-docker.tgz
$ docker tag intersystems/iris-operator:2.0.0.223.0 eu.gcr.io/iko-project/iris-operator:2.0.0.223.0
$ docker push eu.gcr.io/iko-project/iris-operator:2.0.0.223.0
```

Después, debemos indicar al IKO que utilice esta nueva imagen. Debes hacerlo editando el archivo de valores de Helm:

```
$ vi chart/iris-operator/values.yaml
...
operator:
  registry: eu.gcr.io/iko-project
...
```

Ahora, estamos listos para implementar IKO en GKE:

```
$ helm upgrade iko chart/iris-operator --install --namespace iko --create-namespace

$ helm ls --all-namespaces --output json | jq '.[].status'
"deployed"

$ kubectl -n iko get pods # Should be Running with Readiness 1/1
```

Veamos los registros de IKO:

```
$ kubectl -n iko logs -f --tail 100 -l app=iris-operator
...
I1212 17:10:38.119363 1 secure_serving.go:116] Serving securely on [::]:8443
I1212 17:10:38.122306 1 operator.go:77] Starting Iris operator
```

[La definición de recursos personalizados](#) *irisclusters.intersystems.com* fue creada durante la implementación de IKO.

Puedes consultar el esquema compatible con la API, aunque es bastante largo:

```
$ kubectl get crd irisclusters.intersystems.com -oyaml | less
```

Una forma de ver todos los parámetros disponibles es utilizar el comando "explain":

```
$ kubectl explain irisclusters.intersystems.com
```

Otra forma es utilizar [jq](#). Por ejemplo, viendo todos los ajustes de la configuración de nivel superior:

```
$ kubectl get crd irisclusters.intersystems.com -ojson | jq '.spec.versions[].schema.openAPIV3Schema.properties.spec.properties | to_entries[] | .key'
"configSource"
"licenseKeySecret"
"passwordHash"
"serviceTemplate"
"topology"
```

Al utilizar jq de esta manera (viendo los campos de configuración y sus propiedades), podemos encontrar la siguiente estructura de configuración:

```
configSource
  name
licenseKeySecret
  name
passwordHash
serviceTemplate
  metadata
    annotations
  spec
    clusterIP
    externalIPs
    externalTrafficPolicy
    healthCheckNodePort
    loadBalancerIP
    loadBalancerSourceRanges
    ports
    type
topology
  arbiter
    image
  podTemplate
    controller
      annotations
    metadata
      annotations
    spec
      affinity
        nodeAffinity
          preferredDuringSchedulingIgnoredDuringExecution
          requiredDuringSchedulingIgnoredDuringExecution
        podAffinity
          preferredDuringSchedulingIgnoredDuringExecution
          requiredDuringSchedulingIgnoredDuringExecution
        podAntiAffinity
          preferredDuringSchedulingIgnoredDuringExecution
```

```
        requiredDuringSchedulingIgnoredDuringExecution
    args
    env
    imagePullSecrets
    initContainers
    lifecycle
    livenessProbe
    nodeSelector
    priority
    priorityClassName
    readinessProbe
    resources
    schedulerName
    securityContext
    serviceAccountName
    tolerations
preferredZones
updateStrategy
    rollingUpdate
    type
compute
    image
    podTemplate
        controller
            annotations
        metadata
            annotations
    spec
        affinity
            nodeAffinity
                preferredDuringSchedulingIgnoredDuringExecution
                requiredDuringSchedulingIgnoredDuringExecution
            podAffinity
                preferredDuringSchedulingIgnoredDuringExecution
                requiredDuringSchedulingIgnoredDuringExecution
            podAntiAffinity
                preferredDuringSchedulingIgnoredDuringExecution
                requiredDuringSchedulingIgnoredDuringExecution
        args
        env
        imagePullSecrets
        initContainers
        lifecycle
        livenessProbe
        nodeSelector
        priority
        priorityClassName
        readinessProbe
        resources
            limits
            requests
        schedulerName
        securityContext
        serviceAccountName
        tolerations
preferredZones
```

```
replicas
storage
  accessModes
  dataSource
  apiGroup
  kind
  name
resources
  limits
  requests
selector
storageClassName
volumeMode
volumeName
updateStrategy
  rollingUpdate
  type
data
  image
  mirrored
  podTemplate
    controller
    annotations
  metadata
    annotations
  spec
    affinity
      nodeAffinity
        preferredDuringSchedulingIgnoredDuringExecution
        requiredDuringSchedulingIgnoredDuringExecution
      podAffinity
        preferredDuringSchedulingIgnoredDuringExecution
        requiredDuringSchedulingIgnoredDuringExecution
      podAntiAffinity
        preferredDuringSchedulingIgnoredDuringExecution
        requiredDuringSchedulingIgnoredDuringExecution
    args
    env
    imagePullSecrets
    initContainers
    lifecycle
    livenessProbe
    nodeSelector
    priority
    priorityClassName
    readinessProbe
    resources
      limits
      requests
    schedulerName
    securityContext
    serviceAccountName
    tolerations
  preferredZones
  shards
  storage
```

```
accessModes
dataSource
  apiGroup
  kind
  name
resources
  limits
  requests
selector
storageClassName
volumeMode
volumeName
updateStrategy
  rollingUpdate
  type
```

Hay muchos ajustes, pero no necesitas configurar todos. Los valores predeterminados son adecuados. Puedes ver ejemplos de configuración en el archivo *irisoperator-2.0.0.223.0/samples* .

Para ejecutar un IRIS viable mínimo, necesitamos especificar solo unos pocos ajustes, como la versión de IRIS (o de la aplicación basada en IRIS), el tamaño del almacenamiento y la clave de licencia.

Nota sobre la clave de la licencia: utilizaremos la versión Community de IRIS, por lo que no necesitamos una clave. Como no podemos omitir esta configuración, vamos a crear una información confidencial que contenga una pseudo-licencia. La generación de la información confidencial de la licencia es sencilla:

```
$ touch iris.key # remember that a real license file is used in the most cases
$ kubectl create secret generic iris-license --from-file=iris.key
```

Una descripción de IRIS entendible por IKO es:

```
$ cat iko.yaml
apiVersion: intersystems.com/v1alpha1
kind: IrisCluster
metadata:
  name: iko-test
spec:
  passwordHash: '' # use a default password SYS
  licenseKeySecret:
    name: iris-license # use a Secret name bolded above
  topology:
    data:
      image: intersystemsdc/iris-community:2020.4.0.524.0-zpm # Take a community IRIS
      storage:
        resources:
          requests:
            storage: 10Gi
```

Envía este manifiesto al clúster:

```
$ kubectl apply -f iko.yaml
```

```
$ kubectl get iriscluster
```

NAME	DATA	COMPUTE	MIRRORED	STATUS	AGE
iko-test	1			Creating	76s

```
$ kubectl -n iko logs -f --tail 100 -l app=iris-operator
```

```
db.Spec.Topology.Data.Shards = 0
```

```
I1219 15:55:57.989032 1 iriscluster.go:39] Sync/Add/Update for IrisCluster default/iko-test
```

```
I1219 15:55:58.016618 1 service.go:19] Creating Service default/iris-svc.
```

```
I1219 15:55:58.051228 1 service.go:19] Creating Service default/iko-test.
```

```
I1219 15:55:58.216363 1 statefulset.go:22] Creating StatefulSet default/iko-test-data.
```

Vemos que algunos recursos (Service, StatefulSet) se van a crear en un clúster en el *namespace* "predeterminado".

En pocos segundos, deberías ver un contenedor de IRIS en el *namespace* "predeterminado":

```
$ kubectl get po -w
```

NAME	READY	STATUS	RESTARTS	AGE
iko-test-data-0	0/1	ContainerCreating	0	2m10s

Espera un poco hasta que se extraiga la imagen de IRIS, es decir, hasta que el Estado sea Ready y Ready sea 1/1. Puedes verificar qué tipo de disco se creó:

```
$ kubectl get pv
```

NAME	CLAIM	STORAGECLASS	REASON	AGE	CAPACITY	ACCESS	MODES	RECLAIM	POLICY	STATUS
pvc-b356a943-219e-4685-9140-d911dea4c106	iko-test-data-0	standard		5m	10Gi	RWO	Delete	Bound		default/iris-data-iko-test-data-0

La política de recuperación ["Delete"](#) (Eliminar) significa que cuando se elimina el Volumen Persistente, el disco persistente GCE también se eliminará. Hay otra política, ["Retain"](#) (Retener), que te permite guardar discos persistentes de Google para conservar los volúmenes persistentes eliminados de Kubernetes. Puedes definir una [StorageClass](#) personalizada para utilizar esta política y otras configuraciones no predeterminadas. Un ejemplo está presente en la documentación de IKO: [Crear una clase de almacenamiento para el almacenamiento persistente](#).

Ahora, vamos a comprobar nuestro IRIS recién creado. En general, el tráfico hacia los contenedores pasa a través de los Services o Ingresses. De forma predeterminada, IKO crea un servicio de tipo ClusterIP con un nombre del campo iko.yaml *metadata.name*:

```
$ kubectl get svc iko-test
```

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
iko-test	ClusterIP	10.40.6.33	<none>	1972/TCP, 52773/TCP	14m

Podemos llamar a este servicio usando *port-forward*:

```
$ kubectl port-forward svc/iko-test 52773
```

Ve a <http://localhost:52773/csp/sys/UtilHome.csp> y escribe `system/SYS`.

Deberías ver una interfaz de usuario (IU) de IRIS que te resultará familiar.

Aplicación personalizada

Vamos a sustituir un IRIS puro por una aplicación basada en IRIS. En primer lugar, descarga la [aplicación COVID-19](#). No vamos a considerar aquí una implementación completa y continua, solo consideraremos unos pasos mínimos:

```
$ git clone https://github.com/intersystems-community/covid-19.git
$ cd covid-19
$ docker build --no-cache -t covid-19:v1 .
```

Como nuestro Kubernetes se ejecuta en una nube de Google, vamos a utilizar Google Docker Container Registry como un almacén de imágenes. Asumimos aquí que tienes una cuenta en Google Cloud que te permite enviar imágenes. Utiliza tu propio nombre de proyecto en los siguientes comandos:

```
$ docker tag covid-19:v1 eu.gcr.io/iko-project/covid-19:v1
$ docker push eu.gcr.io/iko-project/covid-19:v1
```

Vamos al directorio con `iko.yaml`, cambiamos la imagen que se encuentra allí y volvamos a implementarla. Debes considerar eliminar primero el ejemplo anterior:

```
$ cat iko.yaml
...
  data:
    image: eu.gcr.io/iko-project/covid-19:v1
...
$ kubectl delete -f iko.yaml
$ kubectl -n iko delete deploy -l app=iris-operator
$ kubectl delete pvc iris-data-iko-test-data-0
$ kubectl apply -f iko.yaml
```

Deberás volver a crear el contenedor IRIS con esta nueva imagen.

Esta vez, vamos a proporcionar acceso externo por medio del [Ingress Resource](#). Para que funcione, debemos implementar un [Ingress Controller](#) (elige [nginx](#) por su flexibilidad). Para proporcionar un cifrado de tráfico (TLS), también añadiremos otro componente – [cert-manager](#).

Para instalar estos dos componentes, utilizamos una [herramienta Helm](#), versión 3.

```
$ helm repo add ingress-nginx https://kubernetes.github.io/ingress-nginx

$ helm upgrade nginx-ingress \
  --namespace nginx-ingress \
  ingress-nginx/ingress-nginx \
  --install \
  --atomic \
  --version 3.7.0 \
```



```
--create-namespace
```

Observa una IP de servicio nginx (es dinámica, pero puedes [hacerla estática](#)):

```
$ kubectl -n nginx-ingress get svc
NAME                                TYPE                CLUSTER-IP      EXTERNAL-IP
nginx-ingress-                      LoadBalancer        10.40.0.103      xx.xx.xx.xx
ingress-nginx-controller            LoadBalancer        10.40.0.103      80:32032/TCP,443:32374/TCP 88s
```

Nota: tu IP será distinta.

Ve a tu registrador de dominios para dar de alta y crear un nombre de dominio para esta IP. Por ejemplo, crea un registro A:

```
covid19.myardyas.club = xx.xx.xx.xx
```

Pasará algún tiempo hasta que este nuevo registro se propague a los servidores DNS. El resultado final debería ser similar a:

```
$ dig +short covid19.myardyas.club
xx.xx.xx.xx
```

Después de implementar el Ingress Controller, ahora necesitamos crear un Ingress Resource en sí mismo (usa tu propio nombre de dominio):

```
$ cat ingress.yaml
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: iko-test
  annotations:
    kubernetes.io/ingress.class: nginx
    nginx.ingress.kubernetes.io/use-regex: "true"
    nginx.ingress.kubernetes.io/ssl-redirect: "true"
    certmanager.k8s.io/cluster-issuer: lets-encrypt-
production # Cert manager will be deployed below
spec:
  rules:
  - host: covid19.myardyas.club
    http:
      paths:
      - backend:
          serviceName: iko-test
          servicePort: 52773
        path: /
  tls:
  - hosts:
    - covid19.myardyas.club
    secretName: covid19.myardyas.club
```

```
$ kubectl apply -f ingress.yaml
```

Después de un minuto, más o menos, IRIS debería estar disponible en <http://covid19.myardyas.club/csp/sys/UtilHome.csp> (recuerda utilizar tu nombre de dominio) y la aplicación COVID-19 en <http://covid19.myardyas.club/dsw/index.html> (selecciona el *namespace* IRISAPP).

Nota: arriba, expusimos el puerto HTTP de IRIS. Si necesitas exponer a través de nginx el puerto super-servidor TCP (1972 o 51773), lee las instrucciones en [Exponer servicios TCP y UDP](#).

Añadir cifrado de tráfico

El último paso es añadir cifrado de tráfico. Para ello, vamos a implementar cert-manager:

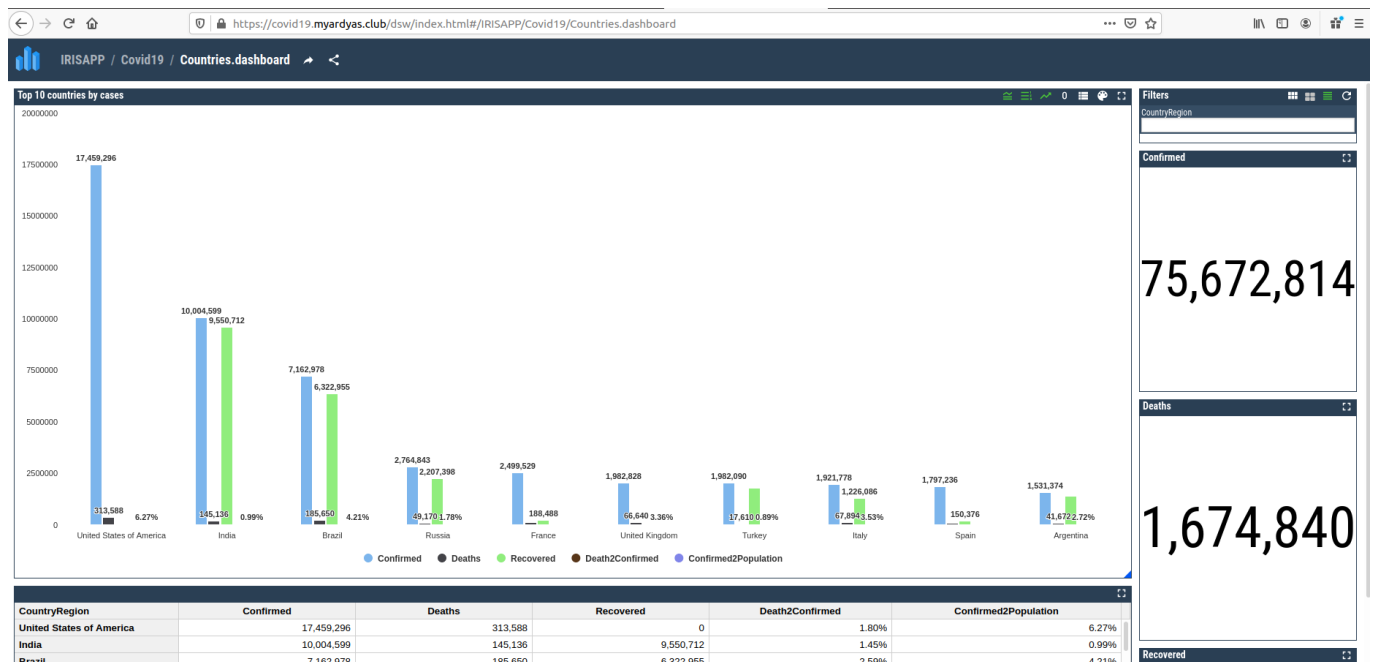
```
$ kubectl apply -f https://raw.githubusercontent.com/jetstack/cert-manager/v0.10.0/deploy/manifests/00-crds.yaml
$ helm upgrade cert-manager \
  --namespace cert-manager \
  jetstack/cert-manager \
  --install \
  --atomic \
  --version v0.10.0 \
  --create-namespace
$ cat lets-encrypt-production.yaml
apiVersion: certmanager.k8s.io/v1alpha1
kind: ClusterIssuer
metadata:
  name: lets-encrypt-production
spec:
  acme:
    # Set your email. Let's Encrypt will send notifications about certificates expiration
    email: mvhoma@gmail.com
    server: https://acme-v02.api.letsencrypt.org/directory
    privateKeySecretRef:
      name: lets-encrypt-production
    solvers:
      - http01:
          ingress:
            class: nginx
$ kubectl apply -f lets-encrypt-production.yaml
```

Espera unos minutos **hasta que cert-manager note el acceso de la aplicación de IRIS Ingress y vaya a Let's Encrypt para obtener un certificado**. Puedes observar los recursos Order y Certificate en curso:

```
$ kubectl get order
NAME                                STATE AGE
covid19.myardyas.club-3970469834 valid 52s
$ kubectl get certificate
NAME                                READY SECRET                                AGE
```

covid19.myardyas.club True covid19.myardyas.club 73s

Esta vez, puedes visitar una versión más segura del sitio - <https://covid19.myardyas.club/dsw/index.html>:



Sobre el controlador de acceso nativo de Google y los certificados administrados

Google admite su propio controlador de acceso, [GCE](#), que puedes utilizar en vez de un controlador nginx. Sin embargo, tiene algunos inconvenientes, por ejemplo, [la falta de compatibilidad con las reglas para reescribir](#), al menos en el momento de escribir.

Además, puedes utilizar [certificados administrados por Google](#) en lugar de cert-manager. Es útil, pero la recuperación inicial del certificado y cualquier actualización de los recursos de Ingress (como la nueva ruta) provoca un tiempo de inactividad evidente. Además, los certificados administrados por Google solo funcionan con GCE, no con nginx, como se indica en los [certificados administrados](#).

Siguientes pasos

Hemos implementado una aplicación basada en IRIS en el clúster GKE. Para exponerlo en Internet, hemos añadido un Ingress Controller y un administrador de certificaciones. Hemos probado la configuración de IrisCluster para resaltar que la configuración de IKO es sencilla. Puedes leer sobre más configuraciones en la documentación: [Uso del Operador InterSystems Kubernetes](#).

Un solo servidor de datos está bien, pero la verdadera diversión comienza cuando añadimos ECP, *mirroring* y monitorización, que también están disponibles con IKO. En el próximo artículo

trataremos *mirroring* en detalle.

[#DevOps](#) [#Kubernetes](#) [#InterSystems](#) [IRIS](#)

URL de
fuentes: <https://es.community.intersystems.com/post/an%C3%A1lisis-en-detalle-del-operador-intersystems-kubernetes-parte-2>