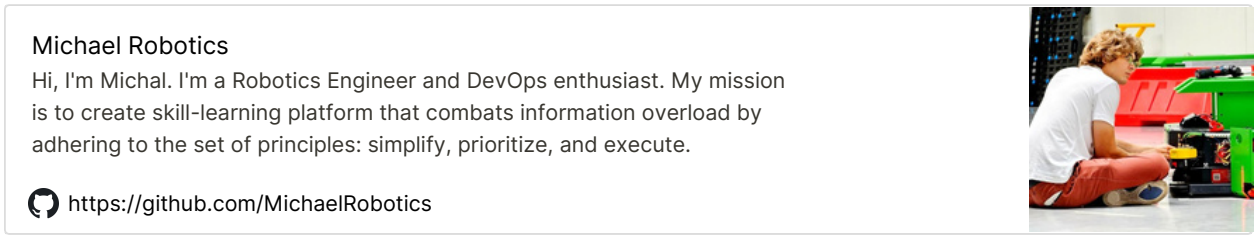
Kubernetes Custom APIs: Create Custom k8 API (CRD&controllers) for Slack alerting using Metacontroller or Kubebuilder in Python & Go, incorporating graceful shutdown techniques

Check GitHub for helpful DevOps tools:

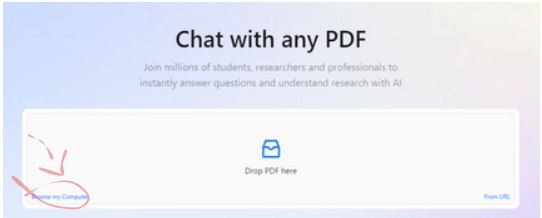


Ask Personal AI Document assistant to learn interactively (FASTER)!

1. Download PDF
2. Go to website
3. Browse file
4. Chat with Document

1 https://github.com/MichaelRobotics/DevOpsTools/blob/main/KubernetesCustomAPIs.pdf

Click there to go to ChatPdf website 2



3

Ask questions about document! 4

Complety new to Linux and Networking?

Essential for this PDF is a thorough knowledge of networking. I highly recommend the HTB platform's networking module, which offers extensive information to help build a comprehensive understanding.

HTB - Your Cyber Performance Center

We provide a human-first platform creating and maintaining high performing cybersecurity individuals and organizations.

https://www.hackthebox.com/

What is Kubernetes?

Kubernetes is an open-source platform that automates the deployment, scaling, and management of containerized applications. It helps manage clusters of nodes running containers, ensuring efficient and reliable operation.

How Kubernetes clusters are made?

Kubernetes clusters consist of a control plane and multiple worker nodes. The control plane manages cluster operations, while worker nodes run the actual container workloads.

Why and When use Kubernetes

Kubernetes is ideal for deploying scalable, resilient, and automated containerized applications. It is used when managing multiple containers across different environments is necessary.

Example: Running a microservices-based e-commerce platform that scales up during peak hours.

System Requirements

- RAM: 2 GB per node (1 GB can work for testing but may lead to limited performance)
- 10 GB free storage
- Ubuntu

Kubernetes: Main components & packages

- **kube-apiserver:** Central management component that exposes the Kubernetes API; acts as the front-end for the cluster.
- **etcd:** Distributed key-value store for storing all cluster data, ensuring data consistency across nodes.
- **kube-scheduler:** Assigns pods to available nodes based on resource requirements and policies.
- **kube-controller-manager:** Manages core controllers that handle various functions like node status, replication, and endpoints.
- **kubelet:** Agent that runs on each node, responsible for managing pods and their containers.
- **kube-proxy:** Manages networking on each node, ensuring communication between pods and services within the cluster.

Kubernetes Custom APIs: CRDs, controllers, operators

1) Custom APIs Introduction

Kubernetes is designed as an extensible API. While often associated with containers, it was built to manage any resource.

Vanilla Kubernetes is basic. It supports creating deployments that manage replicasets and pods, basic service discovery, ingress for pod access, and attaching persistent storage.

Provisioning storage requires CSI drivers, as Kubernetes doesn't handle that by default. Network resources also lack built-in controllers, so operators like Cilium are needed for tasks like routing and network policies. If you use Kubernetes, you've likely already installed CRDs (Custom Resource Definitions).

2) CRD (Custom Resource Definition):

A Custom Resource Definition (CRD) lets you create new resource types in Kubernetes, extending its default capabilities. Changes to these resources trigger eventsthrough Kubernetes components, which can be handled and passed to custom controllers.

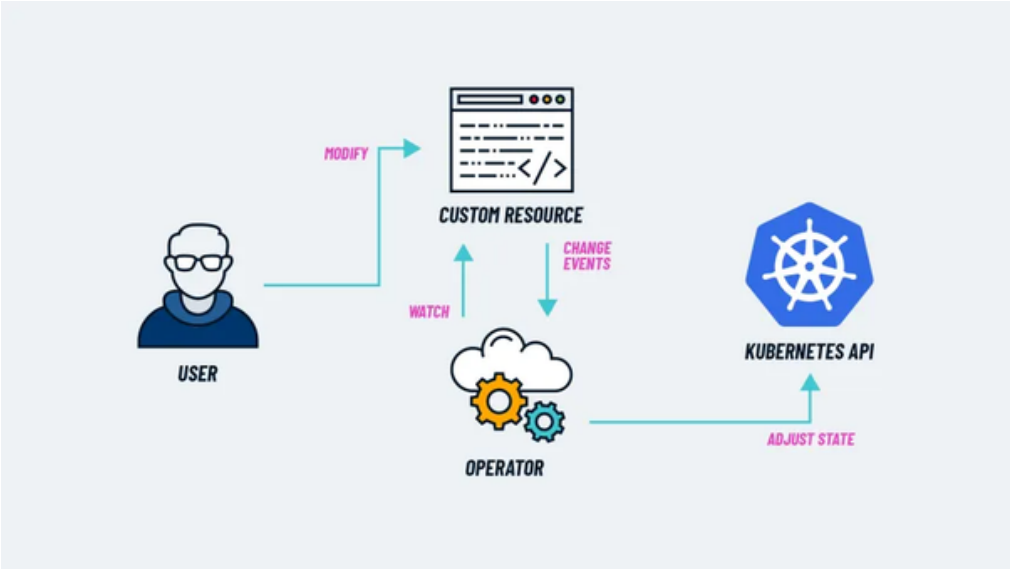
3) Controllers:

Controllers are processes that continuously listen for events related to specific resources (including custom resources) and take action when those events are triggered. They ensure the desired state of a resource is maintained.

4) Operators:

Operators are a design pattern for creating controllers. They encapsulate domain-specific knowledge and automate complex operations for managing applications or resources in Kubernetes.

We can use CRD for gel m templating. We can define to change k8s resources to way we want in case of some event automatically and dont need to apply manually helm templates.



5) Operators use cases

We can use CRDs In substitute of Helm templating. CRDs allow us to define Kubernetes resources to automatically respond to events and make changes as needed, eliminating the need to manually apply Helm templates.

When comparing annotations to CRDs, annotations may be sufficient if you're managing just a single instance. However, when scaling to manage thousands of instances, the added complexity of CRDs becomes worthwhile, offering better scalability and automation.

Kubernetes Custom APIs: Create k8s API's with Metacontroller

1) Intro to Metacontrollers

Metacontroller is an add-on for Kubernetes that makes it easy to write and deploy custom controllers

2) Setup

Download repo:

```
git clone https://github.com/vfarcic/metacontroller-demo
cd metacontroller-demo
```

3) Install application with kustomize:

```
export INGRESS_HOST=127.0.0.1

kubectl apply \
--kustomize https://github.com/metacontroller/metacontroller/manifests/production

kubectl create namespace production

kubectl create namespace controllers
```

```
wget https://github.com/mikefarah/yq/releases/latest/download/yq_linux_amd64 -O
/usr/bin/yq &&\
chmod +x /usr/bin/yq

yq --inplace \
".spec.host = \"silly-demo.$INGRESS_HOST.nip.io\"" \
my-app.yaml

yq --inplace \
".spec.host = \"silly-demo.$INGRESS_HOST.nip.io\"" \
my-app-1-0-7.yaml
```

4) Custom CRDs

check crd:

```
cat crds.yaml
```

```yaml
---

apiVersion: apiextensions.k8s.io/v1
kind: CustomResourceDefinition
metadata:
  name: apps.devopstoolkitseries.com
spec:
  group: devopstoolkitseries.com
  names:
    kind: App
    plural: apps
    singular: app
  scope: Namespaced
  versions:
```

This CRD defines a new Kubernetes resource type named App within the devopstoolkitseries.com API group, with the plural name apps and singular name app, enabling custom resource management under this schema.

```
      openAPIV3Schema:
        type: object
        properties:
          spec:
            type: object
            properties:
              image:
                type: string
              port:
                type: integer
              cpuLimit:
                type: string
              memLimit:
                type: string
```

The schema defines spec as an object with properties like image, port, cpuLimit, and others, specifying their types for resource validation.

```
                replicas:
                  type: integer
                  default: 1
              required:
              - image
              - host
    subresources:
      status: {}
```

This addition specifies that the replicas property has a default value of 1, marks image and host as required fields in the spec, and enables the status subresource to allow Kubernetes to track and manage the resource's status.

```yaml
apiVersion: apiextensions.k8s.io/v1
kind: CustomResourceDefinition
metadata:
  name: slacks.devopstoolkitseries.com
spec:
  group: devopstoolkitseries.com
  names:
    kind: Slack
    plural: slacks
    singular: slack
    shortNames:
    - sl
  scope: Cluster
  versions:
  - name: v1
    served: true
    storage: true
    schema:
      openAPIV3Schema:
        x-kubernetes-preserve-unknown-fields: true
    subresources:
      status: {}
```

The x-kubernetes-preserve-unknown-fields: true setting allows the custom resource to include and retain fields not defined in the schema, providing flexibility but bypassing strict validation.

As you can see, we created two CustomResourceDefinition. One is for Slack and second for app

apply

```
kubectl apply -f crds.yaml
```

```
controlplane $ kubectl apply -f crds.yaml
customresourcedefinition.apiextensions.k8s.io/apps.devopstoolkitseries.com created
customresourcedefinition.apiextensions.k8s.io/slacks.devopstoolkitseries.com created
```

We created our resources! Now list all CRDs

kubectl get crds

```
controlplane $ kubectl get crds
NAME                                                  CREATED AT
apps.devopstoolkitseries.com                          2025-01-08T15:34:17Z
bgpconfigurations.crd.projectcalico.org               2025-01-02T09:48:14Z
bgppeers.crd.projectcalico.org                        2025-01-02T09:48:14Z
blockaffinities.crd.projectcalico.org                 2025-01-02T09:48:14Z
caliconodestatuses.crd.projectcalico.org              2025-01-02T09:48:15Z
clusterinformations.crd.projectcalico.org             2025-01-02T09:48:15Z
compositecontrollers.metacontroller.k8s.io            2025-01-08T15:17:32Z
controllerrevisions.metacontroller.k8s.io             2025-01-08T15:17:32Z
decoratorcontrollers.metacontroller.k8s.io            2025-01-08T15:17:32Z
felixconfigurations.crd.projectcalico.org             2025-01-02T09:48:15Z
globalnetworkpolicies.crd.projectcalico.org           2025-01-02T09:48:15Z
globalnetworksets.crd.projectcalico.org               2025-01-02T09:48:15Z
hostendpoints.crd.projectcalico.org                   2025-01-02T09:48:15Z
ipamblocks.crd.projectcalico.org                      2025-01-02T09:48:15Z
ipamconfigs.crd.projectcalico.org                     2025-01-02T09:48:15Z
ipamhandles.crd.projectcalico.org                     2025-01-02T09:48:15Z
ippools.crd.projectcalico.org                         2025-01-02T09:48:15Z
ipreservations.crd.projectcalico.org                  2025-01-02T09:48:15Z
kubecontrollersconfigurations.crd.projectcalico.org   2025-01-02T09:48:15Z
networkpolicies.crd.projectcalico.org                 2025-01-02T09:48:15Z
networksets.crd.projectcalico.org                     2025-01-02T09:48:15Z
slacks.devopstoolkitseries.com                        2025-01-08T15:34:17Z
```

Now its time to create resource, based on our definition:

cat slack.yaml

```
controlplane $ cat slack.yaml
apiVersion: devopstoolkitseries.com/v1
kind: Slack
metadata:
  name: apps
spec: {}
```

our resource spec, have nothing under spec label and its fine, beacause as we specified in CRD

```
      openAPIV3Schema:
        x-kubernetes-preserve-unknown-fields: true
    subresources:
```

It doesnt need to have specs to be valid.

Now check slack resource yaml:

```
cat slack.yaml
```

```
controlplane $ cat my-app.yaml
apiVersion: devopstoolkitseries.com/v1
kind: App
metadata:
  name: my-app
spec:
  image: vfarcic/silly-demo:1.0.6
  port: 8080
  host: silly-demo.127.0.0.1.nip.io
```

Difference is instantly seen. We defined image name, beacause CRDs demand this resource to have image and host definitions under spec label as we specified:

```
      required:
      - image
      - host
```

Now apply both custom resources

```
kubectl apply -f my-app.yaml
kubectl apply -f slack.yaml
```

```
controlplane $ kubectl apply -f my-app.yaml
app.devopstoolkitseries.com/my-app created
controlplane $ kubectl apply -f slack.yaml
slack.devopstoolkitseries.com/apps created
controlplane $ kubectl get pods
No resources found in default namespace.
```

and check both custom resources

```
controlplane $ kubectl get apps
NAME     AGE
my-app   36s
controlplane $ kubectl get slack
NAME   AGE
apps   31s
```

You have created your first custom k8s resource! But resource itself dont do anything. There is nothing in k8s that decide what to do, when changes happen to anything defined in this resource. To make it do actually something with our cluster or other apps, we need controllers

5) Controllers resource

Check custom controllers:

`cat composite-controllers.yaml`

```yaml
apiVersion: metacontroller.k8s.io/v1alpha1
kind: CompositeController
metadata:
  name: app
spec:
  generateSelector: true
  parentResource:
    apiVersion: devopstoolkitseries.com/v1
    resource: apps
  childResources:
  - apiVersion: apps/v1
    resource: deployments
    updateStrategy:
      method: InPlace
  - apiVersion: v1
    resource: services
    updateStrategy:
      method: InPlace
  - apiVersion: networking.k8s.io/v1
    resource: ingresses
    updateStrategy:
      method: InPlace
```

The CompositeController specifies apps (from devopstoolkitseries.com/v1) as the parent resource, which generates events caught by the controller, while deployments, services, and ingresses are child resources managed and updated by the controller.

```yaml
  hooks:
    sync:
      webhook:
        url: http://app-controller.controllers/sync
```

The sync hook triggers a webhook to http://app-controller.controllers/sync during synchronization, enabling custom actions. We will specify app as python application and deploy it in k8s later

second part defines second controller:

```
kind: CompositeController
metadata:
  name: slack
spec:
  generateSelector: true
  parentResource:
    apiVersion: devopstoolkitseries.com/v1
    resource: apps
  hooks:
    sync:
      webhook:
        url: http://slack-controller.controllers:8080/slack
```

Its less complicated. There is only 1 parent resource whose events will be passed to slack app .Child resources dont exist what implies, that this controller only passes k8s events to slack app and doesnt change anything within kubernetes resources.

apply custom controllers and check their existence in k8s:

```
kubectl apply \
   --filename composite-controllers.yaml
kubectl get compositecontrollers
```

```
controlplane $ kubectl apply -f crds.yaml
customresourcedefinition.apiextensions.k8s.io/apps.devopstoolkitseries.com created
customresourcedefinition.apiextensions.k8s.io/slacks.devopstoolkitseries.com created
controlplane $
controlplane $ kubectl apply \
>     --filename composite-controllers.yaml
compositecontroller.metacontroller.k8s.io/app created
compositecontroller.metacontroller.k8s.io/slack created
controlplane $ kubectl get compositecontrollers
NAME    AGE
app     55s
slack   55s
```

6) Controllers scripts

Firstly, lets check python script for app controler. Check what it does

cat app.py

```
class Controller(BaseHTTPRequestHandler):
  def sync(self, parent, children):
    # Compute status based on observed state.
    desired_status = {
      "deployments": len(children["Deployment.apps/v1"]),
      "services": len(children["Service.v1"]),
      "ingresses": len(children["Ingress.networking.k8s.io/v1"])
    }

    image = parent.get("spec", {}).get("image")
    port = parent.get("spec", {}).get("port", "80")
    cpu_limit = parent.get("spec", {}).get("cpuLimit", "500m")
    mem_limit = parent.get("spec", {}).get("memLimit", "512Mi")
    cpu_req = parent.get("spec", {}).get("cpuReq", "250m")
    mem_req = parent.get("spec", {}).get("memReq", "256Mi")
    host = parent.get("spec", {}).get("host")
    replicas = parent.get("spec", {}).get("replicas")
    desired_resources = [
```

Class Controller(BaseHTTPRequestHandler) is used to handle controller related tasks. It have sync method which takes parent and children resources from our k8s cluster.

image, port and other objects, catch values related to specific yamls labels and specify default values like port 80 or cpuLimit 500.

```
desired_resources = [
  {
    "apiVersion": "apps/v1",
    "kind": "Deployment",
    "metadata": {
      "name": parent["metadata"]["name"],
      "labels": {
        "app.kubernetes.io/name": parent["metadata"]["name"]
      }
    },
    "spec": {
      "replicas": replicas,
      "selector": {
        "matchLabels": {
          "app.kubernetes.io/name": parent["metadata"]["name"]
        }
```

Then we have desired_resource which is a list of dictionaries. Each list define specific resource like ingress, deployment etc. supplied with atributes.

```
    }
  ]

  return {"status": desired_status, "children": desired_resources}
```

sync function returns desired_status which track number of resources and desired_resources which consist of updated resources state.

```
  def do_POST(self):
    observed = json.loads(self.rfile.read(int(self.headers.get("content
    print("xxx")
    # print(observed["children"])
    print("yyy")
    desired = self.sync(observed["parent"], observed["children"])

    self.send_response(200)
    self.send_header("Content-type", "application/json")
    self.end_headers()
    self.wfile.write(json.dumps(desired).encode())

HTTPServer(("", 80), Controller).serve_forever()
```

Post function basically tells python what to do, when k8s controller calls this script. Firstye get state of the content (observed = json.lodads(...) change state desired=self.sync(...) and send response to kubernetes.

Now check what Go script for slack app does:

cat main.go

```
func main() {
    r := gin.Default()
    r.POST("/slack", slackHandler)
    port := os.Getenv("PORT")
    if len(port) == 0 {
        port = "8080"
    }
    r.Run(fmt.Sprintf(":%s", port))
}
```

The script creates a routing object r to listen on the /slack endpoint. Upon receiving a request, it is forwarded to the slackHandler function. Server runs on port 8080.

```
type Controller struct {
    ApiVersion      string `json:"apiVersion"`
    Kind            string `json:"kind"`
    meta.ObjectMeta `json:"metadata"`
}

type SyncRequest struct {
    Parent Controller `json:"parent"`
}
```

SynRequest is a struct used to represent incoming JSON data to a Kubernetes-like resource format.

slackHandler handles HTTP requests and responses. If passed object dont have JSON data, notify and exit. Json type data passed from http request will be unmarshalled into Go object.

```
func slackHandler(c *gin.Context) {
    body, err := ioutil.ReadAll(c.Request.Body)
    if err != nil {
        println("JSON could not be retrieved")
        c.String(http.StatusBadRequest, "JSON could not be retrieved")
        return
    }
    request := &SyncRequest{}
    // Unmarshal the JSON into the struct.
    err = json.Unmarshal(body, request)
    if err != nil {
        println("JSON could not be unmarshalled")
        c.String(http.StatusBadRequest, "JSON could not be unmarshalled")
        return
    }
```

MessageField will store info about type of operation done to resource. Object “message” constructs a message in yaml format, that will be sent to Slack. Info will include ApiVersion, Kind, and the Operation from the ManagedFields.

```
    managedField := request.Parent.ObjectMeta.ManagedFields[0]
    message := fmt.Sprintf(`API Version: %s
Kind: %s
Operation: %s`,
        request.Parent.ApiVersion,
        request.Parent.Kind,
        managedField.Operation)
```

Next step is to retrieve variable under “channel” field if specified in URL for example: http://example.com/slack?channel=#channel, set “#general” if not specified and Get SLACK_TOKEN variable from host.

```
channel := c.Query("channel")
// Throw error if channel is empty
if channel == "" {
    channel = "#general"
}
// Get token from environment variable
token := os.Getenv("SLACK_TOKEN")
// Throw error if token is empty
if token == "" {
    println("Environment variable `SLACK_TOKEN` is empty")
    c.String(http.StatusBadRequest, "Environment variable `SLACK_TOK
    return
```

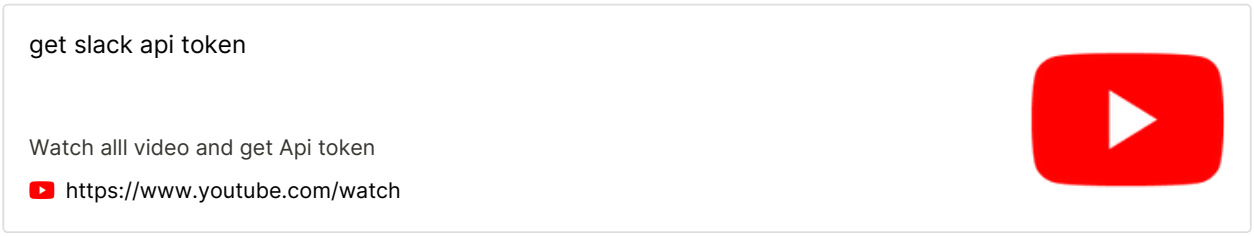
At the end of a script we create connection with slack and send our message.

```
}
// Send message to Slack
api := slack.New(token)
_, _, err = api.PostMessage(channel, slack.MsgOptionText(message, false))
// Throw error if message could not be sent
if err != nil {
    c.String(http.StatusInternalServerError, err.Error())
    return
}
// Return success
c.String(http.StatusOK, "Success")
```

There are better solutions available for alerting. The ones provided here are merely examples to demonstrate the functionality of metacontrollers.

7) Deploy controllers

follow this tutorial to Get Slack token:



Modify controller.yaml, and pass api key to your SLACK_TOKEN parameter.

```
nano controller.yaml
```

add to to slack-controller:

```
env:
- name: SLACK_TOKEN
value: xoxp-81[...]
```

```
  name: slack-controller
spec:
  replicas: 1
  selector:
    matchLabels:
      app: slack-controller
  template:
    metadata:
      labels:
        app: slack-controller
    spec:
      containers:
      - name: controller
        image: vfarcic/metacontroller-demo:0.0.7
        env:
        - name: SLACK_TOKEN
          value: xoxp-8148002[...]
```

We have defined controllers, the resources they utilize, and created a script that allows the controllers to modify specified resources. Now, let's create our app that uses these components

cat my-app-1-0-7.yaml

```
apiVersion: devopstoolkitseries.com/v1
kind: App
metadata:
  name: my-app
spec:
  image: vfarcic/silly-demo:1.0.7
  port: 8080
  host: silly-demo.127.0.0.1.nip.io
  replicas: 3
```

```
kubectl --namespace production apply \
  --filename my-app-1-0-7.yaml
```

cat slack.yaml

```
apiVersion: devopstoolkitseries.com/v1
kind: Slack
metadata:
  name: apps
spec: {}
```

```
kubectl --namespace production apply \
  --filename slack.yaml
```

The apps application doesn't perform any significant actions—more precisely, it does nothing beacause we didnt specified in CR any resource. To make it track we need to add resource and re apply. The my-app application creates a container called silly-demo, and after implementing this, we should be able to curl its DNS to verify its functionality.

Controllers capture events from resources and pass them to the hook URLs. We need to create Deployments with pods maintaing code that react to hook requests.

cat controllers.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: app-controller
spec:
  replicas: 1
[...]
    spec:
      containers:
      - name: controller
        image: python:3
        command: ["python3", "/hooks/app.py"]
        volumeMounts:
        - name: hooks
          mountPath: /hooks
      volumes:
      - name: hooks
        configMap:
          name: app
```

The app-controller deployment mounts a Python script as a config map to the pod and then executes it.

The slack-controller pulls the previously built image and attaches the SLACK_TOKEN variable.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: slack-controller
spec:
  replicas: 1
  selector:
    matchLabels:
      app: slack-controller
  template:
    metadata:
      labels:
        app: slack-controller
    spec:
      containers:
      - name: controller
        image: vfarcic/metacontroller-demo:0.0.7
        env:
        - name: SLACK_TOKEN
          value: xoxp-8148002[...]
```

Each deployment has a related service:

```
apiVersion: v1
kind: Service
metadata:
  name: app-controller
[...]
  ports:
  - port: 80
```

and

```
apiVersion: v1
kind: Service
metadata:
  name: slack-controller
[...]
  ports:
  - port: 8080
```

create a ConfigMap from a Python script and mount it to the app-controller

```
kubectl --namespace controllers \
 create configmap app \
 --from-file=app.py
```

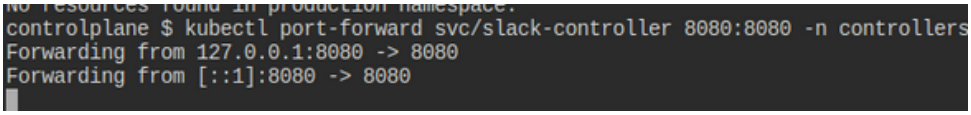
Apply manifest composite-controllers.yaml

```
kubectl --namespace controllers apply \
   --filename controllers.yaml
```

By default, services are in ClusterIP mode. To access the /slack endpoint or silly-demo DNS, we must expose them to the host or access them from other pods.

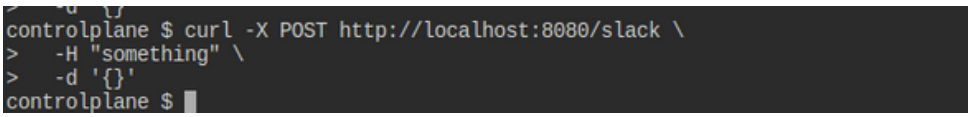
lets expose service related to slack controller:

```
kubectl port-forward svc/slack-controller 8080:8080 -n controllers
```



curl X POST with anything to check if service understand request:

```
curl -X POST http://localhost:8080/slack \
 -H "something" \
 -d '{}'
```



Note that in the my-app-1-0-7.yaml resource, we specified the creation of 3 pods with the image vfarcic/silly-demo:1.0.7. However, when we check all resources in the default namespace:

```
kubectl --namespace production \
  get all,ingresses
```

```
NAME                              READY   STATUS    RESTARTS   AGE
pod/my-app-57d9b4d7-9hf6f         1/1     Running   0          14m
pod/my-app-57d9b4d7-ps28n         1/1     Running   0          14m
pod/my-app-57d9b4d7-vp5nj         1/1     Running   0          14m

NAME                      TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)          AGE
service/my-app            ClusterIP   10.107.180.99    <none>        8080/TCP         14m
service/my-app-service    NodePort    10.109.251.146   <none>        8080:32609/TCP   12m

NAME                      READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/my-app    3/3     3            3           14m

NAME                                DESIRED   CURRENT   READY   AGE
replicaset.apps/my-app-57d9b4d7     3         3         3       14m

NAME                                CLASS    HOSTS                          ADDRESS   PORTS   AGE
ingress.networking.k8s.io/my-app    <none>   silly-demo.127.0.0.1.nip.io              80      14m
controlplane $
```

This happens because our app controller continuously monitors events from my-app and ensures that the state of all resources defined in the Python script is maintained. The Python script instructs the Kubernetes cluster to create resources like Ingress, Services, and Deployments. When it detects that these resources are missing, it automatically creates them.

Kubernetes Custom APIs: Create k8s API's with Kubebuilder

1) Intro

Kubebuilder is a framework for building Kubernetes APIs using Custom Resource Definitions (CRDs). It automatically creates the necessary files, allowing you to quickly start developing the business logic for your operator. Kubebuilder makes it easier to build and manage Kubernetes operators by generating controllers, CRDs, and API definitions.

2) Setup

Lets install go and kubebuilder:

```
git clone https://github.com/MichaelRobotics/Kubernetes.git
cd Kubernetes/Kubebuilder
./Setup.sh
```

Create directory and build our development environemt

```
mkdir -p projects/guestbook
cd  projects/guestbook
ls
kubebuilder init --domain devops.toolbox --repo devops.toolbox/controller
```

```
export PATH=$PATH:/usr/local/go/bin
source ~/.bashrc # For bash
```

Make utility file, Makefile creates our environment. It generates manifests for k8s cluster using kustomize, installs crds on the cluster it can run program locally or push into registry

```
.PHONY: run
run: manifests generate fmt vet ## Run a controller from your host.
    go run ./main.go

.PHONY: docker-build
docker-build: test ## Build docker image with the manager.
    docker build -t ${IMG} .

.PHONY: docker-push
docker-push: ## Push docker image with the manager.
    docker push ${IMG}

##@ Deployment

ifndef ignore-not-found
  ignore-not-found = false
endif

.PHONY: install
install: manifests kustomize ## Install CRDs into the K8s cluster specified in ~/.kube/config.
    $(KUSTOMIZE) build config/crd | kubectl apply -f -

.PHONY: uninstall
uninstall: manifests kustomize ## Uninstall CRDs from the K8s cluster specified in ~/.kube/conf
    $(KUSTOMIZE) build config/crd | kubectl delete --ignore-not-found=$(ignore-not-found) -f -
```

main.go starts controllers process, manages high availability, reports health on dedicated endpoint and much more

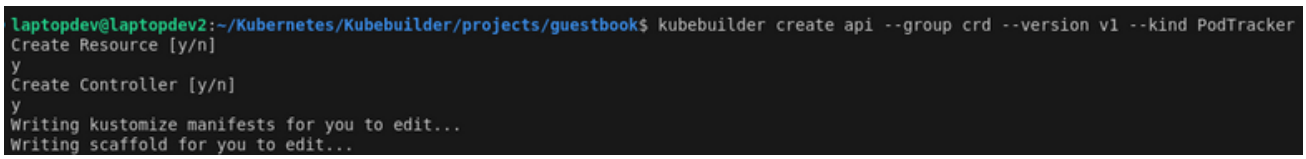
```
s > guestbook > main.go
func init() {
}

func main() {
    var metricsAddr string
    var enableLeaderElection bool
    var probeAddr string
    flag.StringVar(&metricsAddr, "metrics-bind-address", ":8080", "The address the metric endpoin
    flag.StringVar(&probeAddr, "health-probe-bind-address", ":8081", "The address the probe endpo
    flag.BoolVar(&enableLeaderElection, "leader-elect", false,
        "Enable leader election for controller manager. "+
            "Enabling this will ensure there is only one active controller manager.")
    opts := zap.Options{
        Development: true,
    }
    opts.BindFlags(flag.CommandLine)
    flag.Parse()

    ctrl.SetLogger(zap.New(zap.UseFlagOptions(&opts)))

    mgr, err := ctrl.NewManager(ctrl.GetConfigOrDie(), ctrl.Options{
        Scheme:                 scheme,
        MetricsBindAddress:     metricsAddr,
        Port:                   9443,
        HealthProbeBindAddress: probeAddr,
        LeaderElection:         enableLeaderElection,
        LeaderElectionID:       "22381abe.devops.toolbox",
```

lets generate our controller and crds

```
kubebuilder create api --group crd --version v1 --kind PodTracker
```

You can create controller that use existing resources or resources that are not managed by any controller. We will create resources and controller so choose Y in both cases. Now generate k8s manifests

```
make manifests
```



Navigate towards Kubernetes/Kubebuilder directory and run PostSetup.sh script to replace controller and crd files with those already prepared

```
./PostSetup.sh
```

3) CRDs

Navigate towards Kubernetes/Kubebuilder. Our goal, is to crate resource, which will be specified in this way:

```
cat Tracker.yaml
```

```yaml
apiVersion: "crd.devops.toolbox/v1"
kind: "PodTracker"
metadata:
  name: my-tracker
spec:
  name: "slack-tracker"
  report:
    kind: "slack"
    channel: "bots"
    key: "YOUR BOT KEY"
```

Lets check how this CRD is defined in kubebuilder:

```
cat projects/guestbook/api/v1/podtracker_types.go
```

Core of our yaml is defined in PodTracker struct:

```go
// PodTracker is the Schema for the podtrackers API.
type PodTracker struct {
    metav1.TypeMeta   `json:",inline"`
    metav1.ObjectMeta `json:"metadata,omitempty"`

    Spec   PodTrackerSpec   `json:"spec,omitempty"`
    Status PodTrackerStatus `json:"status,omitempty"`
}
```

In the CRD creation, the metadata in the YAML corresponds to the metav1.ObjectMeta field in the PodTracker struct, the spec section maps to the PodTrackerSpec field, and the metav1.TypeMeta field with json:",inline" corresponds to the apiVersion and kind fields in the YAML, allowing them to be included directly without nesting.

So we defined this part of yaml:

```yaml
apiVersion: "crd.devops.toolbox/v1"
kind: "PodTracker"
metadata:
  name: my-tracker
```

now lets check PodTrackerSpec. PodTrackerStatus will not be specified in this tutorial but if we want to run this crd in prodution, its highly recommeded to specify this field.

```go
// PodTrackerSpec defines the desired state of PodTracker
type PodTrackerSpec struct {
    Name   string   `json:"name,omitempty"`
    Report Reporter `json:"report,omitempty"`
```

it basically adds name to our resource and calls another struct Reporter

```go
type Reporter struct {
    Kind    string `json:"kind,omitempty"`
    Key     string `json:"key,omitempty"`
    Channel string `json:"channel,omitempty"`
}
```

The reporter will have the necessary data, including the app to which we want to send the pod status, the key for that app, and the channel for posting. The definitions for PodTrackerSpec and Reporter are as follows:

```yaml
spec:
  name: "slack-tracker"
  report:
    kind: "slack"
    channel: "bots"
    key: ""
```

3) Controller

controller manages custom resource PodTracker, monitoring PodTracker and Pod objects, generating reports (e.g., Slack messages) on events like pod creation, and annotating pods to mark them as managed.

`cat projects/guestbook/controllers/podtracker_controller.go`

Lets read most important function, that manages reconcilation loop:

```go
// SetupWithManager sets up the controller with the Manager.
func (r *PodTrackerReconciler) SetupWithManager(mgr ctrl.Manager) error {
    return ctrl.NewControllerManagedBy(mgr).
        For(&crdv1.PodTracker{}).
        Watches(
            &source.Kind{Type: &corev1.Pod{}},
            handler.EnqueueRequestsFromMapFunc(r.HandlePodEvents),
            builder.WithPredicates(predicate.ResourceVersionChangedPredicate{}),
        ).
        Complete(r)
}
```

The SetupWithManager method configures the PodTrackerReconciler to manage PodTracker resources using the For function, which specifies that the controller will reconcile PodTracker objects.

It also watches Pod resources using the Watches function
handler.EnqueueRequestsFromMapFunc(r.HandlePodEvents) is event handler that processes resource events from pod and by calling r.HandlePodEvents it checks if Pod should be passed to reconcilation loop

ResourceVersionChangedPredicate filter out unnecessary events to optimize performance.
Now check how HandlePodEvents decide if pod should be passed to reconcile fucntion

```go
func (r *PodTrackerReconciler) HandlePodEvents(pod client.Object) []reconcile.Request {
    if pod.GetNamespace() != "default" {
        return []reconcile.Request{}
    }

    namesapcedName := types.NamespacedName{
        Namespace: pod.GetNamespace(),
        Name:      pod.GetName(),
    }

    var podObject corev1.Pod
    err := r.Get(context.Background(), namesapcedName, &podObject)

    if err != nil {
        return []reconcile.Request{}
    }

    if len(podObject.Annotations) == 0 {
        log.Log.V(1).Info("No annotations set, so this pod is becoming a tracked one now", "pod", podObject.Name)
    } else if podObject.GetAnnotations()["exampleAnnotation"] == "crd.devops.toolbox" {
        log.Log.V(1).Info("Found a managed pod, lets report it", "pod", podObject.Name)
    } else {
        return []reconcile.Request{}
    }

    podObject.SetAnnotations(map[string]string{
        "exampleAnnotation": "crd.devops.toolbox",
    })

    if err := r.Update(context.TODO(), &podObject); err != nil {
        log.Log.V(1).Info("error trying to update pod", "err", err)
    }
    requests := []reconcile.Request{
        {NamespacedName: namesapcedName},
    }
    return requests
    // return []reconcile.Request{}
}
```

The HandlePodEvents function processes pod events in the default namespace by checking for any annotations.If a pod is doesnt have anotation, it adds an annotation to mark it as managed and enqueues a reconciliation request so pod will be tracked by reconciliation loop

Lets check reconciliation loop itself:

```go
func (r *PodTrackerReconciler) Reconcile(ctx context.Context, req ctrl.Request) (ctrl.Result, error) {
    logger := log.FromContext(ctx)

    var podTrackerList crdv1.PodTrackerList

    if err := r.List(ctx, &podTrackerList); err != nil {
        logger.Error(err, "unable to fetch pod tracker list")
        return ctrl.Result{}, client.IgnoreNotFound(err)
    }
    if len(podTrackerList.Items) == 0 {
        logger.V(1).Info("no pod trackers configured")
        return ctrl.Result{}, nil
    } else {
        var podObject corev1.Pod
        err := r.Get(context.Background(), req.NamespacedName, &podObject)
        if err != nil {
            return ctrl.Result{}, client.IgnoreNotFound(err)
        }
        logger.V(1).Info("found repoter configured. sending report")
        report(podTrackerList.Items[0], podObject)
    }
    return ctrl.Result{}, nil
}
```

The Reconcile method fetches the list of PodTracker objects and, if any are found, retrieves the associated Pod object for the current request. If both the PodTracker and Pod are found, it sends a report (e.g., Slack message) and completes the reconciliation process.

Lets see what is actually send to slack:

```go
func report(reporter crdv1.PodTracker, pod corev1.Pod) {
    // report to slack
    log.Log.V(1).Info("Reporting to reporter", "name", reporter.Spec.Name, "endpoint", reporter.Spec.Report.Key)
    slackChannel := reporter.Spec.Report.Channel
    app := slack.New(reporter.Spec.Report.Key, slack.OptionDebug(true))

    message := fmt.Sprintf("New pod created: %s", pod.Name)
    msgText := slack.NewTextBlockObject("mrkdwn", message, false, false)
    msgSection := slack.NewSectionBlock(msgText, nil, nil)
    msg := slack.MsgOptionBlocks(
        msgSection,
    )
    fmt.Print(msg)
    log.Log.V(1).Info("Reporting", "message", "", "channel", slackChannel)
    _, _, _, err := app.SendMessage(slackChannel, msg)

    if err != nil {
        log.Log.V(1).Info(err.Error())
    }
}
```

The report function sends a notification to a Slack channel when a new pod is created. It formats a message with the pod's name, creates a Slack message block, and sends the message using the Slack API, logging the success or any errors encountered during the process.

4) Setup slack

The last part is setting up Slack and passing the correct parameters to the Tracker yaml.

To configure Slack check tutorial:

Slack Bot OAuth Access Token

Watch alll video and get Api token

https://www.youtube.com/watch

With those permissions:

Bot Token Scopes

Scopes that govern what your app can access.

OAuth Scope	Description
app_mentions:read	View messages that directly mention @OperatorTest in conversations that the app is in
calls:read	View information about ongoing and past calls
calls:write	Start and manage calls in a workspace
channels:join	Join public channels in a workspace
channels:read	View basic information about public channels in a workspace
channels:write.topic	Set the description of public channels
chat:write	Send messages as @OperatorTest
chat:write.public	Send messages to channels @OperatorTest isn't a member of

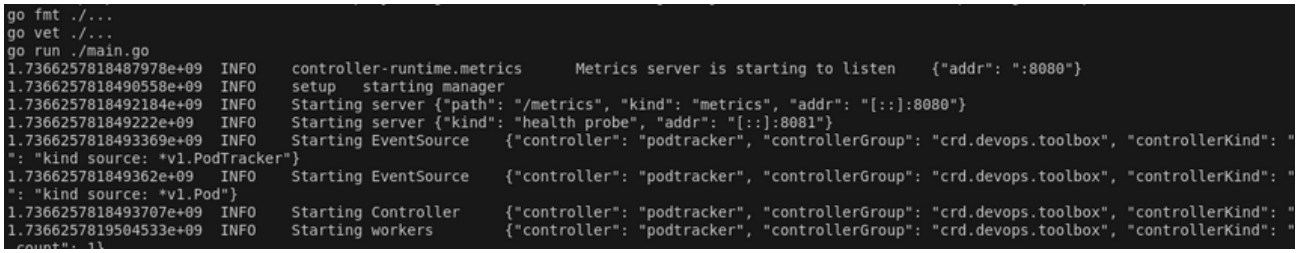
Install CRDs:

```
make install
```

and run our controller

```
make run
```

Everything is fine:



Deploy the PodTracker by passing your token and desired Slack channel to Track.yaml.

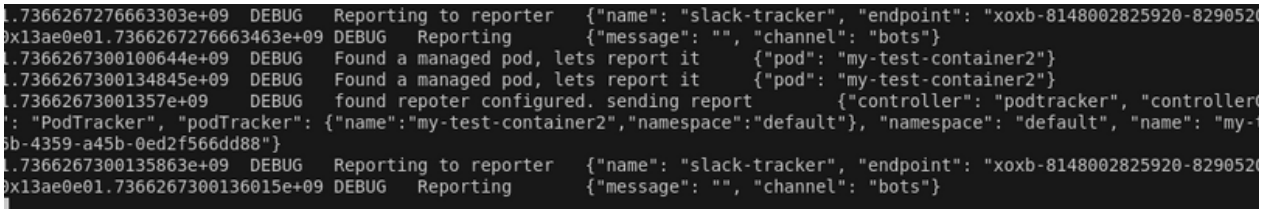
```
nano tracker
```

```yaml
apiVersion: "crd.devops.toolbox/v1"
kind: "PodTracker"
metadata:
  name: my-tracker
spec:
  name: "slack-tracker"
  report:
    kind: "slack"
    channel: "bots"
    key: "YOUR BOT KEY"
```

```
kubectl apply -f Tracker.yaml
```

Create pod to check if slack will recieve message

```
kubectl run nginx-pod --image=nginx:latest --restart=Never --port=80
```



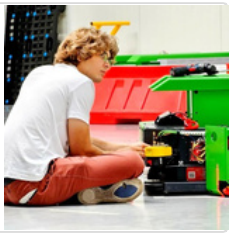
Logs show the report was sent. Check Slack for the update.

It works! Now our Slack channel will be updated any time we create Pod or do something else with those we created!

last part is about Packing our controller into container and CRDs into helmfile or kustomize to simplify deployment process on any cluster. I will do it in the future, everything will be avaialble under this repo:

Michael Robotics
Hi, I'm Michal. I'm a Robotics Engineer and DevOps enthusiast. My mission is to create skill-learning platform that combats information overload by adhering to the set of principles: simplify, prioritize, and execute.

https://github.com/MichaelRobotics



Kubernetes Custom APIs: k8s Gracefull shutdown techniques

1) Introduction

When a server process is abruptly deleted before responding to client requests, errors like 502 Bad Gateway occur. The client sends requests but receives no response because the server was terminated early, leading to failure and potential crashes.

In Kubernetes, avoiding this issue is challenging. Applications or their replicas can be terminated for many reasons—upgrades, scaling, or node updates. Replicas may be deleted, and it’s more about when than if.

2) Solution

The solution is graceful shutdown, managed by signals. Most Linux-based systems assume apps can handle specific signals, but many developers are unaware of this. Today, we’ll explore how these signals can ensure a proper shutdown, even in unexpected termination scenarios.

3) Server with gracefull shutdown implementation

Download repo

Run Setup script

```
/.InstallGo.sh
```

```
export PATH=$PATH:/usr/local/go/bin
source ~/.bashrc # For bash
```

cat goapp/main.go

```go
func main() {
    r := gin.Default()
    r.POST("/slack", slackHandler)

    // Retrieve the port from the environment variable or default to 8080
    port := os.Getenv("PORT")
    if len(port) == 0 {
        port = "10808"
    }

    // Create the HTTP server
    server := &http.Server{
        Addr:    ":" + port,
        Handler: r,
    }
```

The setup initializes a Gin router with a POST route at /slack handled by slackHandler. It retrieves the server port from the PORT environment variable (defaulting to 10808 if not set) and creates an HTTP server using this port and the Gin router to handle requests. Its important to use http.Server with manual handling:

```go
    // Create the HTTP server
    server := &http.Server{
        Addr:    ":" + port,
        Handler: r,
    }
```

to give more control over the server's behavior and lifecycle

Now lets move to greacefull shutdown implementation:

```
// Graceful shutdown logic
go func() {
    // Start the server in a goroutine to avoid blocking
    if err := server.ListenAndServe(); err != nil && err != http.ErrServerClosed {
        log.Fatalf("HTTP server error: %v", err)
    }
}()
```

The server runs in a goroutine, allowing the main function to handle shutdown signals while listening for incoming requests with server.ListenAndServe(). Errors are logged, except for http.ErrServerClosed, which is expected during a graceful shutdown.

```
}()

// Listen for OS signals for graceful shutdown
sigChan := make(chan os.Signal, 1)
signal.Notify(sigChan, syscall.SIGINT, syscall.SIGTERM)
<-sigChan

log.Println("Shutdown signal received. Shutting down gracefully...")
```

A new channel sigChan listens for SIGINT (Ctrl+C) and SIGTERM signals. The <-sigChan line blocks until a shutdown signal is received.When a shutdown signal is received, the program logs that it is shutting down gracefully.

```
// Create a context with timeout for shutdown
shutdownCtx, cancel := context.WithTimeout(context.Background(), 60*time.Second)
defer cancel()
```

context.WithTimeout creates a context that cancels after 60 seconds, limiting the shutdown time. If the server doesn't shut down in time, it will be forcefully terminated. defer cancel() ensures the context is canceled after shutdown.

```
// Attempt to gracefully shut down the server
if err := server.Shutdown(shutdownCtx); err != nil {
    log.Fatalf("Server forced to shutdown: %v", err)
}

log.Println("Server shutdown complete.")
```

server.Shutdown(shutdownCtx) initiates graceful shutdown, allowing ongoing requests to finish. Any shutdown errors (except context cancellation) are logged as fatal. Log message is printed after the server shuts down completely.

```
func slackHandler(c *gin.Context) {
    // Read the entire payload from the request body
    atomic.AddInt64(&requestCount, 1) // Increment request count
    body, err := io.ReadAll(c.Request.Body)
    atomic.AddInt64(&responseCount, 1) // Increment response count after request is handled
    if err != nil {
        c.JSON(http.StatusBadRequest, gin.H{"error": "Failed to read request body"})
        return
    }

    // Log the size of the payload
    log.Printf("Received payload of size: %d bytes", len(body))

    // Respond to the client
    c.JSON(http.StatusOK, gin.H{"message": "Slack endpoint hit!"})
}
```

The slackHandler increments the request and response counts, reads the request body, logs its size, and sends a 200 OK response with a success message. If reading the body fails, it returns a 400 Bad Request.

5) Server without gracefull shutdown implementation

```
cat nograce/main.go
```

instead http.Server we used gin to started server. Its easier way if we need basic server manipulation functionalities:

```
    // Start the server
    log.Printf("Starting server on port %s...", port)
    if err := r.Run(":" + port); err != nil {
        log.Fatalf("Failed to start server: %v", err)
    }
}
```

Instead server shutdown function we use basic sigterm catch and server shutdown. Befor we exit, status of requests and responses is printed:

```
go func() {
    // Block until a signal is received
    <-signalChannel
    // Log the total requests and responses
    log.Printf("Total requests: %d, Total responses: %d\n", atomic.LoadInt64(&requestCount), atomic.LoadInt64(&responseCount))
    // Exit the program immediately
    os.Exit(0)
}()
```

6) Test

run server without gracefull shutdown and client in seperate terminals:

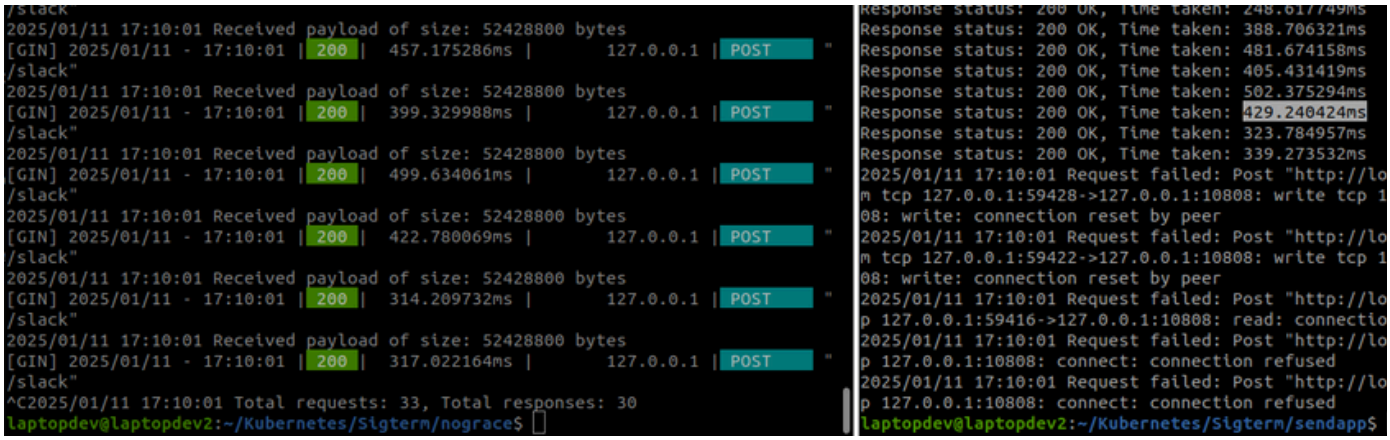
server:

```
go run nograce/main.go
```

client:

```
go run sendapp/main.go
```

Perform shutdown (ctrl+c) on server



Total requests: 33, total responses: 30. The right terminal shows 2 requests were reset by the peer (server). The server received and processed the requests, but something interrupted the task. "Connection refused" indicates attempts to reach an unavailable server.

run server with gracefull shutdown and client in seperate terminals:

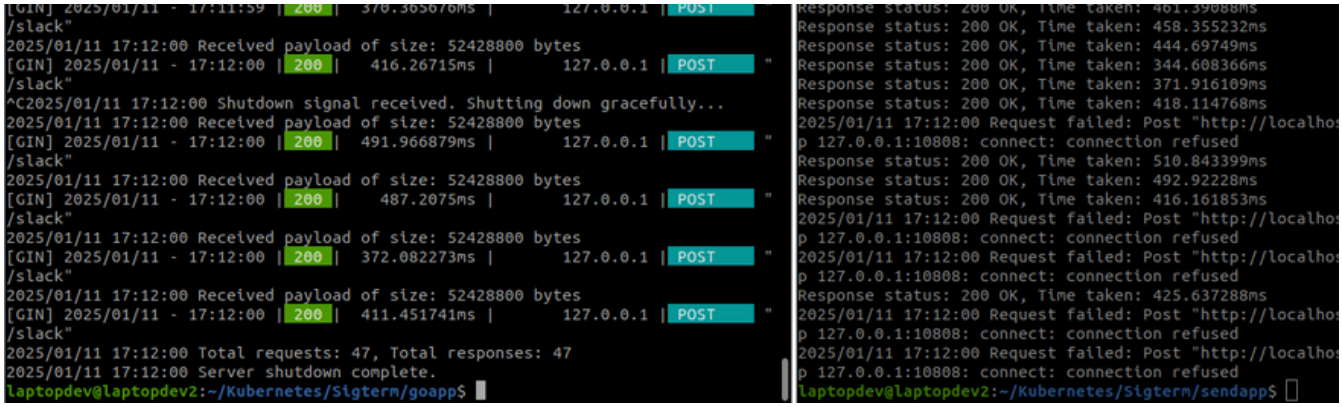
server:

```
go run goapp/main.go
```

client

```
go run sendapp/main.go
```

Perform shutdown (ctrl+c) on server



After receiving the SIGTERM signal, the graceful shutdown procedure was triggered. In the right terminal, tasks related to active connections were completed. The server processed 47 requests and responses, confirming no requests were lost due to the SIGTERM. The right terminal also shows that four pre-SIGTERM requests were completed after the signal. Once all connections were closed, the client stopped.

7) Gracefull shutdown and docker containers

In case of running our aps in docker containers, beahviour should be same. With excveption that docker runtime will send sigterm signals.

Kubernetes Custom APIs: Metacontroller vs Kubebuilder

1) Kubebuilder Overview

Kubebuilder is a framework for building complex Kubernetes controllers using Go. It provides full control over resource management, making it ideal for advanced, stateful, and production-grade controllers. Developers can directly interact with the Kubernetes API and write custom reconciliation logic.

2) Metacontroller Overview

Metacontroller is a higher-level framework that simplifies controller development. It abstracts much of the complexity, allowing developers to define desired states declaratively and write business logic in simpler languages like Python or JavaScript. It’s perfect for lightweight controllers and external resource management.

3) Summary

Kubebuilder is suited for complex, highly customizable controllers with deep Kubernetes API integration, while Metacontroller is better for faster development of simpler controllers, focusing on ease of use and minimal code.

common troubleshooting

1) Custom API Resource (CRD) Not Created

Cause: Invalid CRD YAML.
Solution: Run kubectl apply -f <crd.yaml> and check with kubectl get crds.

2) Controller Not Responding to CRD

Cause: Missing event handling in controller.
Solution: Check logs with kubectl logs <controller-pod> and verify controller setup.

3) Slack Alert Not Triggering

Cause: Incorrect webhook URL or controller issue.
Solution: Test with curl and check controller logs for errors.

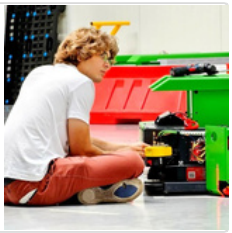
4) Controller Fails to Reconcile

Cause: Logic error or permission issue.
Solution: Review reconcile function and verify RBAC permissions.

5) Check my Kubernetes Troubleshooting series:

Michael Robotics
Hi, I'm Michal. I'm a Robotics Engineer and DevOps enthusiast. My mission is to create skill-learning platform that combats skill information overload by adhering to the set of principles: simplify, prioritize, and execute.

https://github.com/MichaelRobotics



Learn more about Kubernetes

Check Kubernetes and piyushsachdeva - great docs!

Setup a Multi Node Kubernetes Cluster

kubeadm is a tool to bootstrap the Kubernetes cluster

https://github.com/piyushsachdeva/CKA-2024/tree/main/Resources/Day27

Kubernetes Documentation

This section lists the different ways to set up and run Kubernetes

https://kubernetes.io/docs/setup/

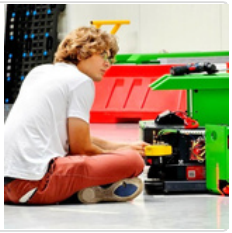
Share, comment, DM and check GitHub for scripts & playbooks created to automate process.

Check my GitHub

Michael Robotics

Hi, I'm Michal. I'm a Robotics Engineer and DevOps enthusiast. My mission is to create skill-learning platform that combats skill information overload by adhering to the set of principles: simplify, prioritize, and execute.

https://github.com/MichaelRobotics



PS.

If you need a playbook or bash script to manage KVM on a specific Linux distribution, feel free to ask me in the comments or send a direct message!