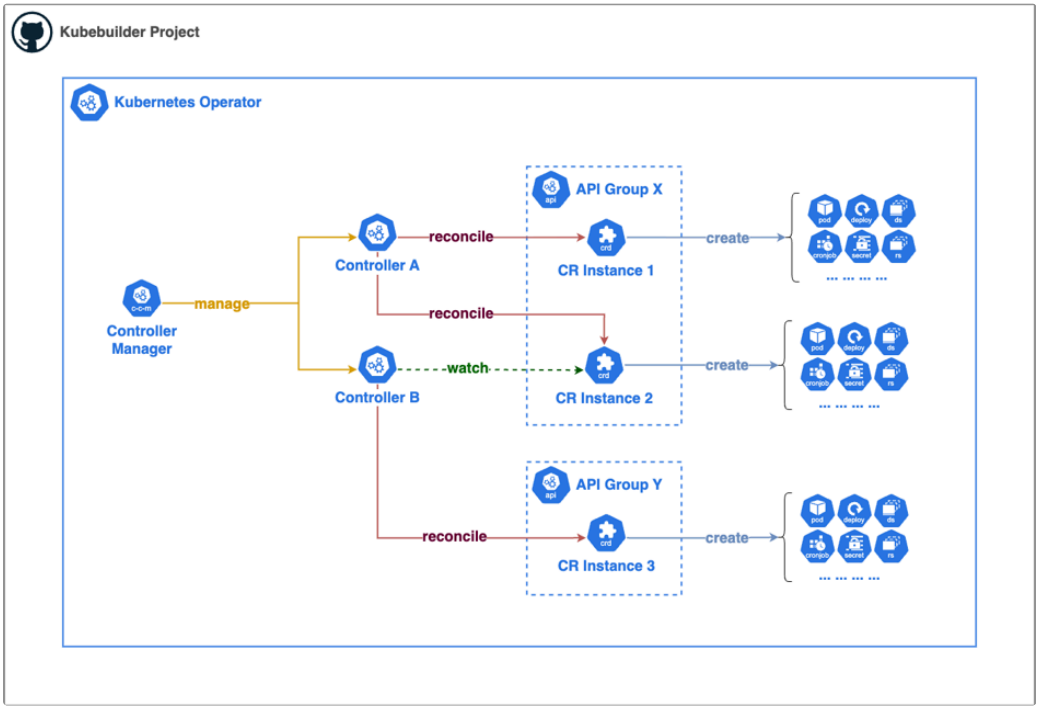
Experiments with Kubebuilder for Implementing Kubernetes Operators

Inside a Kubebuilder Project Through My Lens

In this page, I share two experiments I ran with Kubebuilder and highlight insights about how controllers, CRDs, and projects are structured and managed, summarized in the diagram, which illustrates my understanding of the relationships between a Kubebuilder project, Operator, API Groups, Controllers, and CRs—including ownership, watching, and how CR instances produce Kubernetes resources.



Project / Operator Structure:

- A Kubebuilder project produces a **single Kubernetes Operator binary**.
- The Operator runs a **single Controller Manager**.
- All controllers implemented within the project are **managed by this Controller Manager** and are **independent of API Groups**.
- A **Controller** can reconcile or watch CRs across one or more API Groups, depending on its logic.

CRD / CR Relationships:

- A single CRD can result in **one or many Kubernetes resources**.
- Each CR instance is **owned and reconciled by a single controller**.
- Each CR can be **watched by multiple controllers**.
- Multiple CRs can be created under the **same API Group or across different API Groups**.
- **API Groups** organize CRDs and CRs under a common name (e.g., `apps.example.com`) to separate resources and avoid naming collisions. They exist at the CRD/CR level, not the controller level.

Experiment 1: Single-Controller Operator

Purpose

Deploy a configurable application where a Custom Resource (CR) defines both the Deployment parameters and a greeting message. The operator ensures a Deployment is created with the specified parameters, and the greeting message is stored in a ConfigMap for pods to consume.

Setup / Context

Project & Operator

- Kubebuilder Project in GitHub: config-to-deploy
- Operator Name: config-to-deploy, which matches GitHub repository Name

Custom Resources (CRDs)

- One CRD, named `ConfigDeployment`, is defined to configure the application.
- A single CR created from the `ConfigDeployment` CRD with the following specification:
- ```
  type ConfigDeploymentSpec struct {
      DeployNamespace string `json:"deployNamespace,omitempty"`
  ```
  ```

```
    DeployName      string `json:"deployName,omitempty"`
    DeployImage     string `json:"deployImage,omitempty"`
    ConfigGreetingMsg string `json:"configMsg,omitempty"`
    // +optional
    // +kubebuilder:default=1
    // +kubebuilder:validation:Minimum=1
    // +kubebuilder:validation:Maximum=3
    DeploySize int32 `json:"deploySize,omitempty"`
}
```

- ***Field Explanations***
  - **DeployNamespace** (required) - used for creating both Deployment and ConfigMap
  - **DeployName** (required) - used for naming both Deployment and ConfigMap
  - **DeployImage** (required) - used for spinning up container for the Deployment
  - **ConfigGreetingMsg** (required) - stored in ConfigMap and consumed by deployment containers
  - **DeploySize** (optional) - default to 1, min 1 max 3
- ***Resources Created by the CR in the Target Namespace***
  - **Deployment**: Spins up 1–3 pods based on the CR's `DeploySize` and `DeployImage`.
  - **ConfigMap**: Stores the greeting message from `ConfigGreetingMsg` and is mounted as a local file in the pod.

> **Thinking out loud: Design CRDs**
> When designing a CRD, I first consider all the Kubernetes resources involved in the complete workflow. Knowing how to create each resource individually helps identify the key parameters. The CRD then acts as an encapsulated interface, wrapping these parameters together, like defining a function that exposes just the inputs while hiding the details of the underlying resources.

Controller & Reconciliation Behaviour

- **Controller Purpose:**
  The `ConfigDeploymentReconciler` watches `ConfigDeployment` CRs and ensures that the actual cluster state matches the desired specification.
- **Reconciliation Steps:**
  a. **Fetch CR:** Retrieve the `ConfigDeployment` instance for the requested namespace and name.
  b. **ConfigMap Management:**

- Initialize a ConfigMap containing the `ConfigGreetingMsg`.
- Set the CR as the owner for garbage collection.
- Create the ConfigMap if it doesn't exist.
- Update the ConfigMap if the greeting message changes.

c. **Deployment Management:**

- Initialize a Deployment using `DeployName`, `DeployNamespace`, `DeployImage`, and `DeploySize`.
- Mount the ConfigMap as a volume for the pods.
- Set the CR as the owner.
- Create the Deployment if it doesn't exist.
- Update the Deployment if `Replicas` or `Image` changes.

d. **Idempotency:** The reconciliation is repeatable; it ensures the Deployment and ConfigMap always reflect the CR spec, even if modified manually.

- **Controller Scope:**
  - Watches `ConfigDeployment` CRs.
  - Manages owned `ConfigMap` and `Deployment` resources in the same namespace.
  - Uses `ctrl.SetControllerReference` for both ConfigMap and Deployment to mark them as owned by the CR, ensuring proper cleanup when the CR is deleted.
  - Declares ownership in `SetupWithManager` so the controller automatically reconciles changes to these resources.
- **Hashing for Change Detection:**
  - A SHA256 hash of the ConfigMap data is stored in the pod annotations.
  - This triggers pod updates when the greeting message changes.
- **Manager Setup:**
  - The controller is registered with the manager to watch `ConfigDeployment` CRs and their owned resources (`ConfigMap` and `Deployment`).

Demo

1. **Generate and Install CRD Manifests Within the config-to-deploy Project**

```
config-to-deploy % make manifests
/Users/meng.xu/Workspace/MengWS/kubebuilder-experiments/config-to-deploy/bin/controller-gen rbac:roleName=manager-role crd webhook paths="./..."
output:crd:artifacts:config=config/crd/bases
```

```
config-to-deploy % make install
/Users/meng.xu/Workspace/MengWS/kubebuilder-experiments/config-to-deploy/bin/controller-gen rbac:roleName=manager-role crd webhook paths="./..." output:crd:artifacts:config=config/crd/bases
/Users/meng.xu/Workspace/MengWS/kubebuilder-experiments/config-to-deploy/bin/kustomize build config/crd | kubectl apply -f -
customresourcedefinition.apiextensions.k8s.io/configdeployments.cfg2deploy.meng.xu created
```

2. **Create and Apply a ConfigDeployment CR**

```yaml
apiVersion: cfg2deploy.meng.xu/v1
kind: ConfigDeployment
metadata:
  labels:
    app.kubernetes.io/name: config-to-deploy
    app.kubernetes.io/managed-by: kustomize
  name: configdeployment
spec:
  deployNamespace: cfg2deploy
  deployName: spark
  deployImage: httpd:latest
  deploySize: 2
  configMsg: "What a wonderful world"
```

```
% kubectl get configdeployment
NAME               AGE
configdeployment   5s
```

3. **Build the Operator and Start Local Testing Within the config-to-deploy Project**

```
config-to-deploy % make build
/Users/meng.xu/Workspace/MengWS/kubebuilder-experiments/config-to-deploy/bin/controller-gen rbac:roleName=manager-role crd webhook paths="./..." output:crd:artifacts:config=config/crd/bases
/Users/meng.xu/Workspace/MengWS/kubebuilder-experiments/config-to-deploy/bin/controller-gen object:headerFile="hack/boilerplate.go.txt" paths="./..."
go fmt ./...
go vet ./...
go build -o bin/manager cmd/main.go
```

```
config-to-deploy % make run
/Users/meng.xu/Workspace/MengWS/kubebuilder-experiments/config-to-deploy/bin/controller-gen rbac:roleName=manager-role crd webhook paths="./..." output:crd:artifacts:config=config/crd/bases
/Users/meng.xu/Workspace/MengWS/kubebuilder-experiments/config-to-deploy/bin/controller-gen object:headerFile="hack/boilerplate.go.txt" paths="./..."
go fmt ./...
go vet ./...
go run ./cmd/main.go
2025-08-18T16:01:56+10:00   INFO    setup   starting manager
2025-08-18T16:01:56+10:00   INFO    starting server {"name": "health probe", "addr": "[::]:8081"}
2025-08-18T16:01:56+10:00   INFO    Starting EventSource    {"controller": "configdeployment", "controllerGroup": "cfg2deploy.meng.xu", "controllerKind": "ConfigDeployment", "source": "kind source: *v1.Deployment"}
2025-08-18T16:01:56+10:00   INFO    Starting EventSource    {"controller": "configdeployment", "controllerGroup": "cfg2deploy.meng.xu", "controllerKind":
```

```
"ConfigDeployment", "source": "kind source: *v1.ConfigDeployment"}
2025-08-18T16:01:56+10:00   INFO    Starting EventSource    {"controller":
"configdeployment", "controllerGroup": "cfg2deploy.meng.xu", "controllerKind":
"ConfigDeployment", "source": "kind source: *v1.ConfigMap"}
2025-08-18T16:01:56+10:00   INFO    Starting Controller {"controller":
"configdeployment", "controllerGroup": "cfg2deploy.meng.xu", "controllerKind":
"ConfigDeployment"}
2025-08-18T16:01:56+10:00   INFO    Starting workers    {"controller":
"configdeployment", "controllerGroup": "cfg2deploy.meng.xu", "controllerKind":
"ConfigDeployment", "worker count": 1}
... ...
```

4. **Update ConfigMap Greeting Message**

```
# Namespace to cfg2deploy
% kubens cfg2deploy
Context "kind-kubebuilder" modified.
Active namespace is "cfg2deploy".

# Current running pods
% kubectl get po
NAME                              READY   STATUS    RESTARTS   AGE
spark-deploy-789df4fd6c-w72mt   1/1     Running   0          10s
spark-deploy-789df4fd6c-ws7ss   1/1     Running   0          10s

# Inspect the current greeting message from running pod and configmap
% kubectl exec -it spark-deploy-789df4fd6c-w72mt -- cat
/usr/local/apache2/htdocs/index.html
What a wonderful world

% kubectl get configmap spark-config -o yaml | yq .data.greetingMsg
What a wonderful world
```

```
# Apply the update to the CR with a different greeting message
apiVersion: cfg2deploy.meng.xu/v1
kind: ConfigDeployment
metadata:
  labels:
    app.kubernetes.io/name: config-to-deploy
    app.kubernetes.io/managed-by: kustomize
  name: configdeployment
spec:
  deployNamespace: cfg2deploy
  deployName: spark
  deployImage: httpd:latest
  deploySize: 2
  configMsg: "What a marvelous world" # Original: What a wonderful world

```

```
# Capture Operator log for reconciliation
2025-08-18T16:10:31+10:00   INFO    Updating Deployment with new configuration
{"controller": "configdeployment", "controllerGroup": "cfg2deploy.meng.xu",
"controllerKind": "ConfigDeployment", "ConfigDeployment":
{"name":"configdeployment","namespace":"cfg2deploy"}, "namespace": "cfg2deploy", "name":
"configdeployment", "reconcileID": "941d4c8b-e1aa-4120-80b1-1209ed660c11"}
2025-08-18T16:10:31+10:00   INFO    Deployment updated successfully {"controller":
"configdeployment", "controllerGroup": "cfg2deploy.meng.xu", "controllerKind":
"ConfigDeployment", "ConfigDeployment":
```

{"name":"configdeployment","namespace":"cfg2deploy"}, "namespace": "cfg2deploy", "name": "configdeployment", "reconcileID": "941d4c8b-e1aa-4120-80b1-1209ed660c11"}

```
# Verify configmap update
% kubectl get configmap spark-config -o yaml | yq .data.greetingMsg
What a marvelous world

# Verify deployment update, new pods are up
% kubectl get po
NAME                            READY   STATUS    RESTARTS   AGE
spark-deploy-c5988dd48-jnzhv    1/1     Running   0          17s
spark-deploy-c5988dd48-k2s72    1/1     Running   0          13s

% kubectl exec -it spark-deploy-c5988dd48-jnzhv -- cat
/usr/local/apache2/htdocs/index.html
What a marvelous world
```

Experiment 2: Multi-Controller Operator

Purpose

- Automate pod lifecycle management using custom resources (CRDs).
- Use **CheckIn CRDs** to ensure exactly one pod is running:
  - Set `ActivePod` to the current pod.
  - Append new pod UIDs to `PodHistory`.
- Use **LongLivingPod CRDs** to monitor pods exceeding a runtime threshold and record them in the CR status.
- Demonstrate **multi-controller patterns**:
  - Multiple CRDs managed by a single operator.
  - CR ownership, status updates, and cross-controller observation.

Thinking out loud: Design Multi-Controller Operator

When designing a multi-controller operator with Kubebuilder, it's essential to **clearly define the role of each controller** in relation to the Custom Resource (CR).

While **multiple controllers can watch the same CR**, only **one controller should reconcile it**. This reconciler is responsible for enforcing the desired state and updating the CR. Other controllers may observe the CR to trigger related actions, but they must **not perform reconciliation** or modify the CR directly.

⚠️ Kubernetes doesn't prevent multiple controllers from reconciling the same CR, but doing so is **strongly discouraged**. It can lead to race conditions, conflicting updates, and unpredictable behavior. So while it's technically possible, it's not a pattern you want to rely on.

> This distinction is important because it helps you decide whether you actually need multiple controllers or just one with broader responsibilities. It also guides how and when **cross-controller access** should happen—whether through shared status fields, annotations, or event-driven coordination.

Setup / Context

Project & Operator

- Kubebuilder Project in GitHub: checkin
- A single Kubernetes operator managing multiple CRDs.
- Operator manages pod lifecycle based on custom resources.

Custom Resources (CRDs)

- **CheckIn CRD**
  - Ensures exactly one pod is running.
  - Tracks active pod (`ActivePod`) and pod history (`PodHistory`).
- **LongLivingPod CRD**
  - Monitors pods exceeding a runtime threshold (e.g., 2 minutes).
  - Updates CR status when pods exceed the threshold.

Controller & Reconciliation Behaviour

- **CheckIn Controller**:
  - Watches CheckIn CRs.
  - Creates pods if none exist.
  - Updates `ActivePod` and appends pod UID to `PodHistory` .
- **LongLivingPod Controller**:
  - Observes pods in `ActivePod` .
  - Appends pods exceeding runtime threshold to the CR status.
- Demonstrates **cross-controller observation**: controllers can read and react to each other's CRs.

Demo

1. Under the `checkin` project folder, generate and install manifests.
2. Create both `CheckIn` and `LongLivingPod` CRs with the following YAMLs:

```
# CheckIn CR
```

```yaml
apiVersion: tracker.meng.xu/v1alpha1
kind: CheckIn
metadata:
  labels:
    app.kubernetes.io/name: checkin
    app.kubernetes.io/managed-by: kustomize
  name: "checkin-rsc"
spec:
  podImage: "nginx:latest"
```

```yaml
# LongLivingPod CR
apiVersion: tracker.meng.xu/v1alpha1
kind: LongLivingPod
metadata:
  labels:
    app.kubernetes.io/name: checkin
    app.kubernetes.io/managed-by: kustomize
  name: longlivingpod-sample
spec: {}
```

3. Build the Operator and Start Local Testing Within the Checkin Project

```
checkin % make build
checkin % make run
```

4. Verify reconciliation

```
Namespace to default
% kubens default
Context "kind-kubebuilder" modified.
Active namespace is "default".

# Inspect current running pod and checkin status
% kubectl get po checkin-rsc-pod -o yaml | yq .metadata.uid
2f4ef39e-c320-4b2c-afea-a51147c1f958

# Verify in checkin CR status
% kubectl describe checkin checkin-rsc | yq e '.Status.["Active Pod"]'
2f4ef39e-c320-4b2c-afea-a51147c1f958

# Verify for long living pod also gets appended to LongRunningPod CR status
% kubectl get po
NAME              READY   STATUS    RESTARTS      AGE
checkin-rsc-pod   1/1     Running   1 (43h ago)   47h

% kubectl describe longlivingpod longlivingpod-sample | yq e '.Status.["Long Living
Pods"]' | tr ' ' '\n'
3143f768-f7de-4529-beea-fc02a17a508c
e0e5fd0e-7a15-43cd-ab90-b2385ec9ff44
b5ea63f9-b716-4b50-90ca-7583bbb135f0
87413fac-c4d3-4cae-b822-a4f0cced2c85
89006e39-8fad-4f85-8bfb-d5541387e632
2448caf2-ead4-4b9c-a663-907dfea82da1
62bf2e57-cddf-4d27-b977-b777d711abec
2f4ef39e-c320-4b2c-afea-a51147c1f958 # Current pod UID aged 47h is appended
```