

**Note:** Impatient readers may head straight to [Quick Start](#).

**Using previous version of Kubebuilder v1 or v2? Check the legacy documentation for [v1](#), [v2](#) or [v3](#)**

## Who is this for

### Users of Kubernetes

Users of Kubernetes will develop a deeper understanding of Kubernetes through learning the fundamental concepts behind how APIs are designed and implemented. This book will teach readers how to develop their own Kubernetes APIs and the principles from which the core Kubernetes APIs are designed.

Including:

- The structure of Kubernetes APIs and Resources
- API versioning semantics
- Self-healing
- Garbage Collection and Finalizers
- Declarative vs Imperative APIs
- Level-Based vs Edge-Based APIs
- Resources vs Subresources

### Kubernetes API extension developers

API extension developers will learn the principles and concepts behind implementing canonical Kubernetes APIs, as well as simple tools and libraries for rapid execution. This book covers pitfalls and misconceptions that extension developers commonly encounter.

Including:

- How to batch multiple events into a single reconciliation call
- How to configure periodic reconciliation
- *Forthcoming*
  - When to use the lister cache vs live lookups
  - Garbage Collection vs Finalizers
  - How to use Declarative vs Webhook Validation
  - How to implement API versioning

# Why Kubernetes APIs

Kubernetes APIs provide consistent and well defined endpoints for objects adhering to a consistent and rich structure.

This approach has fostered a rich ecosystem of tools and libraries for working with Kubernetes APIs.

Users work with the APIs through declaring objects as *yaml* or *json* config, and using common tooling to manage the objects.

Building services as Kubernetes APIs provides many advantages to plain old REST, including:

- Hosted API endpoints, storage, and validation.
- Rich tooling and CLIs such as `kubectl` and `kustomize`.
- Support for AuthN and granular AuthZ.
- Support for API evolution through API versioning and conversion.
- Facilitation of adaptive / self-healing APIs that continuously respond to changes in the system state without user intervention.
- Kubernetes as a hosting environment

Developers may build and publish their own Kubernetes APIs for installation into running Kubernetes clusters.

## Contribution

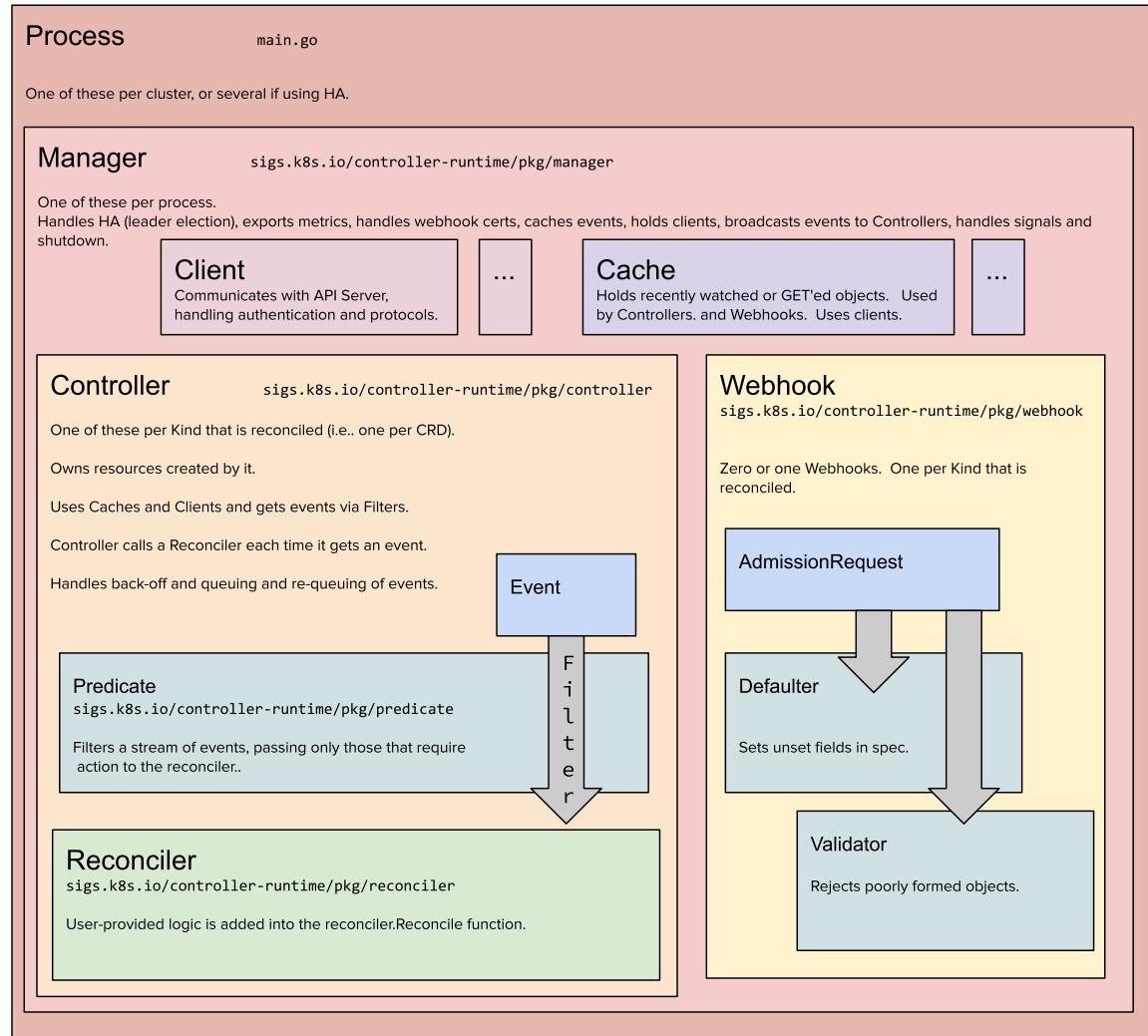
If you like to contribute to either this book or the code, please be so kind to read our [Contribution](#) guidelines first.

## Resources

- Repository: [sigs.k8s.io/kubebuilder](https://sigs.k8s.io/kubebuilder)
- Slack channel: [#kubebuilder](#)
- Google Group: [kubebuilder@googlegroups.com](mailto:kubebuilder@googlegroups.com)

# Architecture Concept Diagram

The following diagram will help you get a better idea over the Kubebuilder concepts and architecture.



# Quick Start

This Quick Start guide will cover:

- [Creating a project](#)
- [Creating an API](#)
- [Running locally](#)
- [Running in-cluster](#)

## Prerequisites

- [go](#) version v1.23.0+
- [docker](#) version 17.03+.
- [kubectl](#) version v1.11.3+.
- Access to a Kubernetes v1.11.3+ cluster.

### Versions Compatibility and Supportability

Please, ensure that you see the [guidance](#).

## Installation

Install [kubebuilder](#):

```
# download kubebuilder and install locally.  
curl -L -o kubebuilder "https://go.kubebuilder.io/dl/latest/$(go env GOOS)/$(go env GOARCH)"  
chmod +x kubebuilder && sudo mv kubebuilder /usr/local/bin/
```

### Using the Master Branch

You can work with the master branch by cloning the repository and running `make install` to generate the binary. Please follow the steps in the section **How to Build Kubebuilder Locally** from the [Contributing Guide](#).

## Enabling shell completion

Kubebuilder provides autocomplete support via the command `kubebuilder completion <bash|fish|powershell|zsh>`, which can save you a lot of typing. For further information see the [completion](#) document.

## Create a Project

Create a directory, and then run the init command inside of it to initialize a new project. Follows an example.

```
mkdir -p ~/projects/guestbook  
cd ~/projects/guestbook  
kubebuilder init --domain my.domain --repo my.domain/guestbook
```

### Developing in \$GOPATH

If your project is initialized within `GOPATH`, the implicitly called `go mod init` will interpolate the module path for you. Otherwise `--repo=<module path>` must be set.

Read the [Go modules blogpost](#) if unfamiliar with the module system.

## Create an API

Run the following command to create a new API (group/version) as `webapp/v1` and the new Kind(CRD) `Guestbook` on it:

```
kubebuilder create api --group webapp --version v1 --kind Guestbook
```

### Press Options

If you press `y` for Create Resource [y/n] and for Create Controller [y/n] then this will create the files `api/v1/guestbook_types.go` where the API is defined and the `internal/controllers/guestbook_controller.go` where the reconciliation business logic is implemented for this Kind(CRD).

**OPTIONAL:** Edit the API definition and the reconciliation business logic. For more info see [Designing an API](#) and [What's in a Controller](#).

If you are editing the API definitions, generate the manifests such as Custom Resources (CRs) or Custom Resource Definitions (CRDs) using

```
make manifests
```

- ▶ Click here to see an example. ([api/v1/guestbook\\_types.go](#))

## Test It Out

You'll need a Kubernetes cluster to run against. You can use [KIND](#) to get a local cluster for testing, or run against a remote cluster.

### Context Used

Your controller will automatically use the current context in your kubeconfig file (i.e. whatever cluster `kubectl cluster-info` shows).

Install the CRDs into the cluster:

```
make install
```

For quick feedback and code-level debugging, run your controller (this will run in the foreground, so switch to a new terminal if you want to leave it running):

```
make run
```

## Install Instances of Custom Resources

If you pressed `y` for Create Resource [y/n] then you created a CR for your CRD in your samples (make sure to edit them first if you've changed the API definition):

```
kubectl apply -k config/samples/
```

# Run It On the Cluster

When your controller is ready to be packaged and tested in other clusters.

Build and push your image to the location specified by `IMG`:

```
make docker-build docker-push IMG=<some-registry>/<project-name>:tag
```

Deploy the controller to the cluster with image specified by `IMG`:

```
make deploy IMG=<some-registry>/<project-name>:tag
```

## Registry Permission

This image ought to be published in the personal registry you specified. And it is required to have access to pull the image from the working environment. Make sure you have the proper permission to the registry if the above commands don't work.

Consider incorporating Kind into your workflow for a faster, more efficient local development and CI experience. Note that, if you're using a Kind cluster, there's no need to push your image to a remote container registry. You can directly load your local image into your specified Kind cluster:

```
kind load docker-image <your-image-name>:tag --name <your-kind-cluster-name>
```

To know more, see: [Using Kind For Development Purposes and CI](#)

## RBAC errors

If you encounter RBAC errors, you may need to grant yourself cluster-admin privileges or be logged in as admin. See [Prerequisites for using Kubernetes RBAC on GKE cluster v1.11.x and older](#) which may be your case.

# Uninstall CRDs

To delete your CRDs from the cluster:

```
make uninstall
```

# Undeploy controller

Undeploy the controller to the cluster:

```
make undeploy
```

## Next Step

- Now, take a look at the [Architecture Concept Diagram](#) for a clearer overview.
- Next, proceed with the [Getting Started Guide](#), which should take no more than 30 minutes and will provide a solid foundation. Afterward, dive into the [CronJob Tutorial](#) to deepen your understanding by developing a demo project.

# Getting Started

We will create a sample project to let you know how it works. This sample will:

- Reconcile a Memcached CR - which represents an instance of a Memcached deployed/managed on cluster
- Create a Deployment with the Memcached image
- Not allow more instances than the size defined in the CR which will be applied
- Update the Memcached CR status

## Why Operators?

By following the [Operator Pattern](#), it's possible not only to provide all expected resources but also to manage them dynamically, programmatically, and at execution time. To illustrate this idea, imagine if someone accidentally changed a configuration or removed a resource by mistake; in this case, the operator could fix it without any human intervention.

## Following Along vs Jumping Ahead

Note that most of this tutorial is generated from literate Go files that form a runnable project, and live in the book source directory: [docs/book/src/getting-started/testdata/project](#).

# Create a project

First, create and navigate into a directory for your project. Then, initialize it using `kubebuilder`:

```
mkdir $GOPATH/memcached-operator
cd $GOPATH/memcached-operator
kubebuilder init --domain=example.com
```

## Developing in \$GOPATH

If your project is initialized within `GOPATH`, the implicitly called `go mod init` will interpolate the module path for you. Otherwise `--repo=<module path>` must be set.

Read the [Go modules blogpost](#) if unfamiliar with the module system.

## Create the Memcached API (CRD):

Next, we'll create the API which will be responsible for deploying and managing Memcached(s) instances on the cluster.

```
kubebuilder create api --group cache --version v1alpha1 --kind Memcached
```

## Understanding APIs

This command's primary aim is to produce the Custom Resource (CR) and Custom Resource Definition (CRD) for the Memcached Kind. It creates the API with the group `cache.example.com` and version `v1alpha1`, uniquely identifying the new CRD of the Memcached Kind. By leveraging the KubeBuilder tool, we can define our APIs and objects representing our solutions for these platforms.

While we've added only one Kind of resource in this example, we can have as many `Groups` and `Kinds` as necessary. To make it easier to understand, think of CRDs as the definition of our custom Objects, while CRs are instances of them.

Please ensure that you check

[Groups and Versions and Kinds, oh my!](#).

## Defining our API

### Defining the Specs

Now, we will define the values that each instance of your Memcached resource on the cluster can assume. In this example, we will allow configuring the number of instances with the following:

```
type MemcachedSpec struct {
    ...
    Size int32 `json:"size,omitempty"`
}
```

## Creating Status definitions

We also want to track the status of our Operations which will be done to manage the Memcached CR(s). This allows us to verify the Custom Resource's description of our own API and determine if everything occurred successfully or if any errors were encountered, similar to how we do with any resource from the Kubernetes API.

```
// MemcachedStatus defines the observed state of Memcached
type MemcachedStatus struct {
    Conditions []metav1.Condition `json:"conditions,omitempty"
patchStrategy:"merge" patchMergeKey:"type" protobuf:"bytes,1,rep,name=conditions"`
}
```

### Status Conditions

Kubernetes has established conventions, and because of this, we use Status Conditions here. We want our custom APIs and controllers to behave like Kubernetes resources and their controllers, following these standards to ensure a consistent and intuitive experience.

Please ensure that you review: [Kubernetes API Conventions](#)

## Markers and validations

Furthermore, we want to validate the values added in our CustomResource to ensure that those are valid. To achieve this, we will use [markers](#), such as

```
+kubebuilder:validation:Minimum=1 .
```

Now, see our example fully completed.

```
$ vim ../getting-started/testdata/project/api/v1alpha1/memcached_types.go
// Apache License (hidden)
// Imports (hidden)
```

```

// MemcachedSpec defines the desired state of Memcached.
type MemcachedSpec struct {
    // INSERT ADDITIONAL SPEC FIELDS - desired state of cluster
    // Important: Run "make" to regenerate code after modifying this file

    // Size defines the number of Memcached instances
    // The following markers will use OpenAPI v3 schema to validate the value
    // More info: https://book.kubebuilder.io/reference/markers/crd-
validation.html
    // +kubebuilder:validation:Minimum=1
    // +kubebuilder:validation:Maximum=3
    // +kubebuilder:validation:ExclusiveMaximum=false
    Size int32 `json:"size,omitempty"`
}

// MemcachedStatus defines the observed state of Memcached.
type MemcachedStatus struct {
    // Represents the observations of a Memcached's current state.
    // Memcached.status.conditions.type are: "Available", "Progressing", and
    "Degraded"
    // Memcached.status.conditions.status are one of True, False, Unknown.
    // Memcached.status.conditions.reason the value should be a CamelCase string
    and producers of specific
    // condition types may define expected values and meanings for this field, and
    whether the values
    // are considered a guaranteed API.
    // Memcached.status.conditions.Message is a human readable message indicating
    details about the transition.
    // For further information see:
https://github.com/kubernetes/community/blob/master/contributors/devel/signature-architecture/api-conventions.md#typical-status-properties

    Conditions []metav1.Condition `json:"conditions,omitempty"`
    patchStrategy:"merge" patchMergeKey:"type" protobuf:"bytes,1,rep,name=conditions"`
}

// +kubebuilder:object:root=true
// +kubebuilder:subresource:status

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

    Spec   MemcachedSpec   `json:"spec,omitempty"`
    Status MemcachedStatus `json:"status,omitempty"`
}

// +kubebuilder:object:root=true

// MemcachedList contains a list of Memcached.
type MemcachedList struct {
    metav1.TypeMeta `json:",inline"`

```

```

    metav1.ListMeta `json:"metadata,omitempty"`
    Items          []Memcached `json:"items"`
}

func init() {
    SchemeBuilder.Register(&Memcached{}, &MemcachedList{})
}

```

## Generating manifests with the specs and validations

To generate all required files:

1. Run `make generate` to create the DeepCopy implementations in `api/v1alpha1/zz_generated.deepcopy.go`.
2. Then, run `make manifests` to generate the CRD manifests under `config/crd/bases` and a sample for it under `config/crd/samples`.

Both commands use `controller-gen` with different flags for code and manifest generation, respectively.

- `config/crd/bases/cache.example.com_memcacheds.yaml`: Our Memcached CRD

## Sample of Custom Resources

The manifests located under the `config/samples` directory serve as examples of Custom Resources that can be applied to the cluster. In this particular example, by applying the given resource to the cluster, we would generate a Deployment with a single instance size (see `size: 1`).

```

apiVersion: cache.example.com/v1alpha1
kind: Memcached
metadata:
  labels:
    app.kubernetes.io/name: project
    app.kubernetes.io/managed-by: kustomize
  name: memcached-sample
spec:
  # TODO(user): edit the following value to ensure the number
  # of Pods/Instances your Operand must have on cluster
  size: 1

```

## Reconciliation Process

In a simplified way, Kubernetes works by allowing us to declare the desired state of our system, and then its controllers continuously observe the cluster and take actions to ensure that the actual state matches the desired state. For our custom APIs and controllers, the process is similar. Remember, we are extending Kubernetes' behaviors and its APIs to fit our specific needs.

In our controller, we will implement a reconciliation process.

Essentially, the reconciliation process functions as a loop, continuously checking conditions and performing necessary actions until the desired state is achieved. This process will keep running until all conditions in the system align with the desired state defined in our implementation.

Here's a pseudo-code example to illustrate this:

```
reconcile App {  
  
    // Check if a Deployment for the app exists, if not, create one  
    // If there's an error, then restart from the beginning of the reconcile  
    if err != nil {  
        return reconcile.Result{}, err  
    }  
  
    // Check if a Service for the app exists, if not, create one  
    // If there's an error, then restart from the beginning of the reconcile  
    if err != nil {  
        return reconcile.Result{}, err  
    }  
  
    // Look for Database CR/CRD  
    // Check the Database Deployment's replicas size  
    // If deployment.replicas size doesn't match cr.size, then update it  
    // Then, restart from the beginning of the reconcile. For example, by returning  
    `reconcile.Result{Requeue: true}, nil`.  
    if err != nil {  
        return reconcile.Result{Requeue: true}, nil  
    }  
    ...  
  
    // If at the end of the loop:  
    // Everything was executed successfully, and the reconcile can stop  
    return reconcile.Result{}, nil  
}
```

### Return Options

The following are a few possible return options to restart the Reconcile:

- With the error:

```
return ctrl.Result{}, err
```

- Without an error:

```
return ctrl.Result{Requeue: true}, nil
```

- Therefore, to stop the Reconcile, use:

```
return ctrl.Result{}, nil
```

- Reconcile again after X time:

```
return ctrl.Result{RequeueAfter: nextRun.Sub(r.Now())}, nil
```

## In the context of our example

When our sample Custom Resource (CR) is applied to the cluster (i.e. `kubectl apply -f config/sample/cache_v1alpha1_memcached.yaml`), we want to ensure that a Deployment is created for our Memcached image and that it matches the number of replicas defined in the CR.

To achieve this, we need to first implement an operation that checks whether the Deployment for our Memcached instance already exists on the cluster. If it does not, the controller will create the Deployment accordingly. Therefore, our reconciliation process must include an operation to ensure that this desired state is consistently maintained. This operation would involve:

```

// Check if the deployment already exists, if not create a new one
found := &appsv1.Deployment{}
err = r.Get(ctx, types.NamespacedName{Name: memcached.Name, Namespace: memcached.Namespace}, found)
if err != nil && apierrors.IsNotFound(err) {
    // Define a new deployment
    dep := r.deploymentForMemcached()
    // Create the Deployment on the cluster
    if err = r.Create(ctx, dep); err != nil {
        log.Error(err, "Failed to create new Deployment",
            "Deployment.Namespace", dep.Namespace, "Deployment.Name", dep.Name)
        return ctrl.Result{}, err
    }
    ...
}

```

Next, note that the `deploymentForMemcached()` function will need to define and return the Deployment that should be created on the cluster. This function should construct the Deployment object with the necessary specifications, as demonstrated in the following example:

```

dep := &appsv1.Deployment{
    Spec: appsv1.DeploymentSpec{
        Replicas: &replicas,
        Template: corev1.PodTemplateSpec{
            Spec: corev1.PodSpec{
                Containers: []corev1.Container{
                    Image:          "memcached:1.6.26-alpine3.19",
                    Name:           "memcached",
                    ImagePullPolicy: corev1.PullIfNotPresent,
                    Ports:          []corev1.ContainerPort{
                        ContainerPort: 11211,
                        Name:           "memcached",
                    },
                    Command:         []string{"memcached", "--memory-limit=64", "-o",
                        "modern", "-v"},
                },
            },
        },
    },
}

```

Additionally, we need to implement a mechanism to verify that the number of Memcached replicas on the cluster matches the desired count specified in the Custom Resource (CR). If there is a discrepancy, the reconciliation must update the cluster to ensure consistency. This means that whenever a CR of the Memcached Kind is created or updated on the cluster, the controller will continuously reconcile the state until the actual number of replicas matches the desired count. The following example illustrates this process:

```

...
size := memcached.Spec.Size
if *found.Spec.Replicas != size {
    found.Spec.Replicas = &size
    if err = r.Update(ctx, found); err != nil {
        log.Error(err, "Failed to update Deployment",
            "Deployment.Namespace", found.Namespace, "Deployment.Name",
            found.Name)
        return ctrl.Result{}, err
    }
}
...

```

Now, you can review the complete controller responsible for managing Custom Resources of the Memcached Kind. This controller ensures that the desired state is maintained in the cluster, making sure that our Memcached instance continues running with the number of replicas specified by the users.

- ▶ `internal/controller/memcached_controller.go` : Our Controller Implementation

## Diving Into the Controller Implementation

### Setting Manager to Watching Resources

The whole idea is to be Watching the resources that matter for the controller. When a resource that the controller is interested in changes, the Watch triggers the controller's reconciliation loop, ensuring that the actual state of the resource matches the desired state as defined in the controller's logic.

Notice how we configured the Manager to monitor events such as the creation, update, or deletion of a Custom Resource (CR) of the Memcached kind, as well as any changes to the Deployment that the controller manages and owns:

```

// SetupWithManager sets up the controller with the Manager.
// The Deployment is also watched to ensure its
// desired state in the cluster.
func (r *MemcachedReconciler) SetupWithManager(mgr ctrl.Manager) error {
    return ctrl.NewControllerManagedBy(mgr).
        // Watch the Memcached Custom Resource and trigger reconciliation whenever
        it
        //is created, updated, or deleted
        For(&cachev1alpha1.Memcached{}).
        // Watch the Deployment managed by the Memcached controller. If any
        changes occur to the Deployment
        // owned and managed by this controller, it will trigger reconciliation,
        ensuring that the cluster
        // state aligns with the desired state.
        Owns(&appsv1.Deployment{}).
        Complete(r)
}

```

## But, How Does the Manager Know Which Resources Are Owned by It?

We do not want our Controller to watch any Deployment on the cluster and trigger our reconciliation loop. Instead, we only want to trigger reconciliation when the specific Deployment running our Memcached instance is changed. For example, if someone accidentally deletes our Deployment or changes the number of replicas, we want to trigger the reconciliation to ensure that it returns to the desired state.

The Manager knows which Deployment to observe because we set the `ownerRef` (Owner Reference):

```

if err := ctrl.SetControllerReference(memcached, dep, r.Scheme); err != nil {
    return nil, err
}

```

``ownerRef` and cascading event`

The `ownerRef` is crucial not only for allowing us to observe changes on the specific resource but also because, if we delete the Memcached Custom Resource (CR) from the cluster, we want all resources owned by it to be automatically deleted as well, in a cascading event.

This ensures that when the parent resource (Memcached CR) is removed, all associated resources (like Deployments, Services, etc.) are also cleaned up, maintaining a tidy and consistent cluster state.

For more information, see the Kubernetes documentation on [Owners and Dependents](#).

## Granting Permissions

It's important to ensure that the Controller has the necessary permissions(i.e. to create, get, update, and list) the resources it manages.

The [RBAC permissions](#) are now configured via [RBAC markers](#), which are used to generate and update the manifest files present in `config/rbac/`. These markers can be found (and should be defined) on the `Reconcile()` method of each controller, see how it is implemented in our example:

```
//  
+kubebuilder:rbac:groups=cache.example.com,resources=memcacheds,verbs=get;list;watch;create;update;patch;delete  
//  
+kubebuilder:rbac:groups=cache.example.com,resources=memcacheds/status,verbs=get;update;patch  
//  
+kubebuilder:rbac:groups=cache.example.com,resources=memcacheds/finalizers,verbs=update  
// +kubebuilder:rbac:groups=core,resources=events,verbs=create;patch  
//  
+kubebuilder:rbac:groups=apps,resources=deployments,verbs=get;list;watch;create;update;patch;delete  
// +kubebuilder:rbac:groups=core,resources=pods,verbs=get;list;watch
```

After making changes to the controller, run the `make generate` command. This will prompt [controller-gen](#) to refresh the files located under `config/rbac`.

- ▶ `config/rbac/role.yaml` : Our RBAC Role generated

## Manager (main.go)

The [Manager](#) in the `cmd/main.go` file is responsible for managing the controllers in your application.

- ▶ `cmd/main.go` : Our main.go

## Checking the Project running in the cluster

At this point you can check the steps to validate the project on the cluster by looking the steps defined in the Quick Start, see: [Run It On the Cluster](#)

## Next Steps

- To delve deeper into developing your solution, consider going through the [CronJob Tutorial](#)
- For insights on optimizing your approach, refer to the [Best Practices](#) documentation.

### Using Deploy Image plugin to generate APIs and source code

Now that you have a better understanding, you might want to check out the [Deploy Image Plugin](#). This plugin allows users to scaffold APIs/Controllers to deploy and manage an Operand (image) on the cluster. It will provide scaffolds similar to the ones in this guide, along with additional features such as tests implemented for your controller.

# Versions Compatibility and Supportability

Projects created by Kubebuilder contain a `Makefile` that installs tools at versions defined during project creation. The main tools included are:

- [kustomize](#)
- [controller-gen](#)
- [setup-envtest](#)

Additionally, these projects include a `go.mod` file specifying dependency versions. Kubebuilder relies on [controller-runtime](#) and its Go and Kubernetes dependencies. Therefore, the versions defined in the `Makefile` and `go.mod` files are the ones that have been tested, supported, and recommended.

Each minor version of Kubebuilder is tested with a specific minor version of client-go. While a Kubebuilder minor version *may* be compatible with other client-go minor versions, or other tools this compatibility is not guaranteed, supported, or tested.

The minimum Go version required by Kubebuilder is determined by the highest minimum Go version required by its dependencies. This is usually aligned with the minimum Go version required by the corresponding `k8s.io/*` dependencies.

Compatible `k8s.io/*` versions, client-go versions, and minimum Go versions can be found in the `go.mod` file scaffolded for each project for each [tag release](#).

**Example:** For the `4.1.1` release, the minimum Go version compatibility is `1.22`. You can refer to the samples in the `testdata` directory of the tag released `v4.1.1`, such as the `go.mod` file for `project-v4`. You can also check the tools versions supported and tested for this release by examining the [Makefile](#).

## Operating Systems Supported

Currently, Kubebuilder officially supports macOS and Linux platforms. If you are using a Windows OS, you may encounter issues. Contributions towards supporting Windows are welcome



Project customizations

After using the CLI to create your project, you are free to customize how you see fit. Bear in mind, that it is not recommended to deviate from the proposed

layout unless you know what you are doing.

For example, you should refrain from moving the scaffolded files, doing so will make it difficult in upgrading your project in the future. You may also lose the ability to use some of the CLI features and helpers. For further information on the project layout, see the doc [What's in a basic project?](#)

# Tutorial: Building CronJob

Too many tutorials start out with some really contrived setup, or some toy application that gets the basics across, and then stalls out on the more complicated stuff. Instead, this tutorial should take you through (almost) the full gamut of complexity with Kubebuilder, starting off simple and building up to something pretty full-featured.

Let's pretend (and sure, this is a teensy bit contrived) that we've finally gotten tired of the maintenance burden of the non-Kubebuilder implementation of the CronJob controller in Kubernetes, and we'd like to rewrite it using Kubebuilder.

The job (no pun intended) of the *CronJob* controller is to run one-off tasks on the Kubernetes cluster at regular intervals. It does this by building on top of the *Job* controller, whose task is to run one-off tasks once, seeing them to completion.

Instead of trying to tackle rewriting the Job controller as well, we'll use this as an opportunity to see how to interact with external types.

## Following Along vs Jumping Ahead

Note that most of this tutorial is generated from literate Go files that live in the book source directory: [docs/book/src/cronjob-tutorial/testdata](#). The full, runnable project lives in [project](#), while intermediate files live directly under the [testdata](#) directory.

## Scaffolding Out Our Project

As covered in the [quick start](#), we'll need to scaffold out a new project. Make sure you've [installed Kubebuilder](#), then scaffold out a new project:

```
# create a project directory, and then run the init command.  
mkdir project  
cd project  
# we'll use a domain of tutorial.kubebuilder.io,  
# so all API groups will be <group>.tutorial.kubebuilder.io.  
kubebuilder init --domain tutorial.kubebuilder.io --repo  
tutorial.kubebuilder.io/project
```

Your project's name defaults to that of your current working directory. You can pass --

```
project-name=<dns1123-label-string> to set a different project name.
```

Now that we've got a project in place, let's take a look at what Kubebuilder has scaffolded for us so far...

### Developing in \$GOPATH

If your project is initialized within `GOPATH`, the implicitly called `go mod init` will interpolate the module path for you. Otherwise `--repo=<module path>` must be set.

Read the [Go modules blogpost](#) if unfamiliar with the module system.

# What's in a basic project?

When scaffolding out a new project, Kubebuilder provides us with a few basic pieces of boilerplate.

## Build Infrastructure

First up, basic infrastructure for building your project:

- ▶ `go.mod` : A new Go module matching our project, with basic dependencies
- ▶ `Makefile` : Make targets for building and deploying your controller
- ▶ `PROJECT` : Kubebuilder metadata for scaffolding new components

## Launch Configuration

We also get launch configurations under the `config/` directory. Right now, it just contains `Kustomize` YAML definitions required to launch our controller on a cluster, but once we get started writing our controller, it'll also hold our CustomResourceDefinitions, RBAC configuration, and WebhookConfigurations.

`config/default` contains a `Kustomize base` for launching the controller in a standard configuration.

Each other directory contains a different piece of configuration, refactored out into its own base:

- `config/manager` : launch your controllers as pods in the cluster
- `config/rbac` : permissions required to run your controllers under their own service account

## The Entrypoint

Last, but certainly not least, Kubebuilder scaffolds out the basic entrypoint of our project: `main.go`. Let's take a look at that next...

# Every journey needs a start, every program needs a main

```
$ vim emptymain.go
// Apache License (hidden)
```

Our package starts out with some basic imports. Particularly:

- The core [controller-runtime](#) library
- The default controller-runtime logging, [Zap](#) (more on that a bit later)

```
package main

import (
    "flag"
    "os"

    // Import all Kubernetes client auth plugins (e.g. Azure, GCP, OIDC, etc.)
    // to ensure that exec-entrypoint and run can make use of them.
    - "k8s.io/client-go/plugin/pkg/client/auth"

    "k8s.io/apimachinery/pkg/runtime"
    utilruntime "k8s.io/apimachinery/pkg/util/runtime"
    clientgoscheme "k8s.io/client-go/kubernetes/scheme"
    - "k8s.io/client-go/plugin/pkg/client/auth/gcp"
    ctrl "sigs.k8s.io/controller-runtime"
    "sigs.k8s.io/controller-runtime/pkg/cache"
    "sigs.k8s.io/controller-runtime/pkg/healthz"
    "sigs.k8s.io/controller-runtime/pkg/log/zap"
    "sigs.k8s.io/controller-runtime/pkg/metrics/server"
    "sigs.k8s.io/controller-runtime/pkg/webhook"
    // +kubebuilder:scaffold:imports
)
```

Every set of controllers needs a [Scheme](#), which provides mappings between Kinds and their corresponding Go types. We'll talk a bit more about Kinds when we write our API definition, so just keep this in mind for later.

```
var (
    scheme = runtime.NewScheme()
    setupLog = ctrl.Log.WithName("setup")
)

func init() {
    utilruntime.Must(clientgoscheme.AddToScheme(scheme))
    // +kubebuilder:scaffold:scheme
}
```

At this point, our main function is fairly simple:

- We set up some basic flags for metrics.
- We instantiate a *manager*, which keeps track of running all of our controllers, as well as setting up shared caches and clients to the API server (notice we tell the manager about our Scheme).
- We run our manager, which in turn runs all of our controllers and webhooks. The manager is set up to run until it receives a graceful shutdown signal. This way, when we're running on Kubernetes, we behave nicely with graceful pod termination.

While we don't have anything to run just yet, remember where that `+kubebuilder:scaffold:builder` comment is – things'll get interesting there soon.

```

func main() {
    var metricsAddr string
    var enableLeaderElection bool
    var probeAddr string
    flag.StringVar(&metricsAddr, "metrics-bind-address", ":8080", "The address the
metric endpoint binds to.")
    flag.StringVar(&probeAddr, "health-probe-bind-address", ":8081", "The address
the probe endpoint binds to.")
    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,
        Metrics: server.Options{
            BindAddress: metricsAddr,
        },
        WebhookServer: webhook.NewServer(webhook.Options{Port: 9443}),
        HealthProbeBindAddress: probeAddr,
        LeaderElection: enableLeaderElection,
        LeaderElectionID: "80807133.tutorial.kubebuilder.io",
    })
    if err != nil {
        setupLog.Error(err, "unable to start manager")
        os.Exit(1)
    }
}

```

Note that the `Manager` can restrict the namespace that all controllers will watch for resources by:

```

mgr, err := ctrl.NewManager(ctrl.GetConfigOrDie(), ctrl.Options{
    Scheme: scheme,
    Cache: cache.Options{
        DefaultNamespaces: map[string]cache.Config{
            namespace: {},
        },
    },
    Metrics: server.Options{
        BindAddress: metricsAddr,
    },
    WebhookServer:           webhook.NewServer(webhook.Options{Port: 9443}),
    HealthProbeBindAddress: probeAddr,
    LeaderElection:          enableLeaderElection,
    LeaderElectionID:         "80807133.tutorial.kubebuilder.io",
)

```

The above example will change the scope of your project to a single `Namespace`. In this scenario, it is also suggested to restrict the provided authorization to this namespace by replacing the default `clusterRole` and `ClusterRoleBinding` to `Role` and `RoleBinding` respectively. For further information see the Kubernetes documentation about [Using RBAC Authorization](#).

Also, it is possible to use the `DefaultNamespaces` from `cache.Options{}` to cache objects in a specific set of namespaces:

```

var namespaces []string // List of Namespaces
defaultNamespaces := make(map[string]cache.Config)

for _, ns := range namespaces {
    defaultNamespaces[ns] = cache.Config{}
}

mgr, err := ctrl.NewManager(ctrl.GetConfigOrDie(), ctrl.Options{
    Scheme: scheme,
    Cache: cache.Options{
        DefaultNamespaces: defaultNamespaces,
    },
    Metrics: server.Options{
        BindAddress: metricsAddr,
    },
    WebhookServer:           webhook.NewServer(webhook.Options{Port: 9443}),
    HealthProbeBindAddress: probeAddr,
    LeaderElection:          enableLeaderElection,
    LeaderElectionID:         "80807133.tutorial.kubebuilder.io",
)

```

For further information see `cache.Options{}`

```
// +kubebuilder:scaffold:builder

if err := mgr.AddHealthzCheck("healthz", healthz.Ping); err != nil {
    setupLog.Error(err, "unable to set up health check")
    os.Exit(1)
}
if err := mgr.AddReadyzCheck("readyz", healthz.Ping); err != nil {
    setupLog.Error(err, "unable to set up ready check")
    os.Exit(1)
}

setupLog.Info("starting manager")
if err := mgr.Start(ctrl.SetupSignalHandler()); err != nil {
    setupLog.Error(err, "problem running manager")
    os.Exit(1)
}
```

With that out of the way, we can get on to scaffolding our API!

# Groups and Versions and Kinds, oh my!

Before we get started with our API, we should talk about terminology a bit.

When we talk about APIs in Kubernetes, we often use 4 terms: *groups*, *versions*, *kinds*, and *resources*.

## Groups and Versions

An *API Group* in Kubernetes is simply a collection of related functionality. Each group has one or more *versions*, which, as the name suggests, allow us to change how an API works over time.

## Kinds and Resources

Each API group-version contains one or more API types, which we call *Kinds*. While a Kind may change forms between versions, each form must be able to store all the data of the other forms, somehow (we can store the data in fields, or in annotations). This means that using an older API version won't cause newer data to be lost or corrupted. See the [Kubernetes API guidelines](#) for more information.

You'll also hear mention of *resources* on occasion. A resource is simply a use of a Kind in the API. Often, there's a one-to-one mapping between Kinds and resources. For instance, the `pods` resource corresponds to the `Pod` Kind. However, sometimes, the same Kind may be returned by multiple resources. For instance, the `Scale` Kind is returned by all scale subresources, like `deployments/scale` or `replicasets/scale`. This is what allows the Kubernetes HorizontalPodAutoscaler to interact with different resources. With CRDs, however, each Kind will correspond to a single resource.

Notice that resources are always lowercase, and by convention are the lowercase form of the Kind.

## So, how does that correspond to Go?

When we refer to a kind in a particular group version, we'll call it a *GroupVersionKind*, or GVK for short. Same with resources and GVR. As we'll see shortly, each GVK corresponds to a given root

Go type in a package.

Now that we have our terminology straight, we can *actually* create our API!

## So, how can we create our API?

In the next section, [Adding a new API](#), we will check how the tool helps us to create our own APIs with the command `kubebuilder create api`.

The goal of this command is to create a Custom Resource (CR) and Custom Resource Definition (CRD) for our Kind(s). To check it further see; [Extend the Kubernetes API with CustomResourceDefinitions](#).

## But, why create APIs at all?

New APIs are how we teach Kubernetes about our custom objects. The Go structs are used to generate a CRD which includes the schema for our data as well as tracking data like what our new type is called. We can then create instances of our custom objects which will be managed by our [controllers](#).

Our APIs and resources represent our solutions on the clusters. Basically, the CRDs are a definition of our customized Objects, and the CRs are an instance of it.

## Ah, do you have an example?

Let's think about the classic scenario where the goal is to have an application and its database running on the platform with Kubernetes. Then, one CRD could represent the App, and another one could represent the DB. By having one CRD to describe the App and another one for the DB, we will not be hurting concepts such as encapsulation, the single responsibility principle, and cohesion. Damaging these concepts could cause unexpected side effects, such as difficulty in extending, reuse, or maintenance, just to mention a few.

In this way, we can create the App CRD which will have its controller and which would be responsible for things like creating Deployments that contain the App and creating Services to access it and etc. Similarly, we could create a CRD to represent the DB, and deploy a controller that would manage DB instances.

## Err, but what's that Scheme thing?

The `Scheme` we saw before is simply a way to keep track of what Go type corresponds to a given GVK (don't be overwhelmed by its [godocs](#)).

For instance, suppose we mark the `"tutorial.kubebuilder.io/api/v1".CronJob{}` type as being in the `batch.tutorial.kubebuilder.io/v1` API group (implicitly saying it has the Kind `CronJob`).

Then, we can later construct a new `&CronJob{}` given some JSON from the API server that says

```
{  
    "kind": "CronJob",  
    "apiVersion": "batch.tutorial.kubebuilder.io/v1",  
    ...  
}
```

or properly look up the group version when we go to submit a `&CronJob{}` in an update.

# Adding a new API

To scaffold out a new Kind (you were paying attention to the [last chapter](#), right?) and corresponding controller, we can use `kubebuilder create api`:

```
kubebuilder create api --group batch --version v1 --kind CronJob
```

Press `y` for “Create Resource” and “Create Controller”.

The first time we call this command for each group-version, it will create a directory for the new group-version.

In this case, the `api/v1/` directory is created, corresponding to the `batch.tutorial.kubebuilder.io/v1` (remember our `--domain` setting from the beginning?).

It has also added a file for our `CronJob` Kind, `api/v1/cronjob_types.go`. Each time we call the command with a different kind, it’ll add a corresponding new file.

Let’s take a look at what we’ve been given out of the box, then we can move on to filling it out.

```
$ vim emptyapi.go
// Apache License (hidden)
```

We start out simply enough: we import the `meta/v1` API group, which is not normally exposed by itself, but instead contains metadata common to all Kubernetes Kinds.

```
package v1

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

Next, we define types for the Spec and Status of our Kind. Kubernetes functions by reconciling desired state (`Spec`) with actual cluster state (other objects’ `Status`) and external state, and then recording what it observed (`Status`). Thus, every *functional* object includes spec and status. A few types, like `ConfigMap` don’t follow this pattern, since they don’t encode desired state, but most types do.

```
// EDIT THIS FILE!  THIS IS SCAFFOLDING FOR YOU TO OWN!
// NOTE: json tags are required. Any new fields you add must have json tags for
// the fields to be serialized.

// CronJobSpec defines the desired state of CronJob
type CronJobSpec struct {
    // INSERT ADDITIONAL SPEC FIELDS - desired state of cluster
    // Important: Run "make" to regenerate code after modifying this file
}

// CronJobStatus defines the observed state of CronJob
type CronJobStatus struct {
    // INSERT ADDITIONAL STATUS FIELD - define observed state of cluster
    // Important: Run "make" to regenerate code after modifying this file
}
```

Next, we define the types corresponding to actual Kinds, `CronJob` and `CronJobList`. `CronJob` is our root type, and describes the `CronJob` kind. Like all Kubernetes objects, it contains `TypeMeta` (which describes API version and Kind), and also contains `ObjectMeta`, which holds things like name, namespace, and labels.

`CronJobList` is simply a container for multiple `CronJob`s. It's the Kind used in bulk operations, like LIST.

In general, we never modify either of these – all modifications go in either Spec or Status.

That little `+kubebuilder:object:root` comment is called a marker. We'll see more of them in a bit, but know that they act as extra metadata, telling [controller-tools](#) (our code and YAML generator) extra information. This particular one tells the `object` generator that this type represents a Kind. Then, the `object` generator generates an implementation of the [runtime.Object](#) interface for us, which is the standard interface that all types representing Kinds must implement.

```

// +kubebuilder:object:root=true
// +kubebuilder:subresource:status

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

    Spec   CronJobSpec   `json:"spec,omitempty"`
    Status CronJobStatus `json:"status,omitempty"`
}

// +kubebuilder:object:root=true

// CronJobList contains a list of CronJob
type CronJobList struct {
    metav1.TypeMeta `json:",inline"`
    metav1.ListMeta `json:"metadata,omitempty"`
    Items           []CronJob `json:"items"`
}

```

Finally, we add the Go types to the API group. This allows us to add the types in this API group to any [Scheme](#).

```

func init() {
    SchemeBuilder.Register(&CronJob{}, &CronJobList{})
}

```

Now that we've seen the basic structure, let's fill it out!

# Designing an API

In Kubernetes, we have a few rules for how we design APIs. Namely, all serialized fields *must* be `camelCase`, so we use JSON struct tags to specify this. We can also use the `omitempty` struct tag to mark that a field should be omitted from serialization when empty.

Fields may use most of the primitive types. Numbers are the exception: for API compatibility purposes, we accept three forms of numbers: `int32` and `int64` for integers, and `resource.Quantity` for decimals.

- ▶ Hold up, what's a `Quantity`?

There's one other special type that we use: `metav1.Time`. This functions identically to `time.Time`, except that it has a fixed, portable serialization format.

With that out of the way, let's take a look at what our CronJob object looks like!

```
$ vim project/api/v1/cronjob_types.go
// Apache License (hidden)

package v1

// Imports (hidden)
```

First, let's take a look at our spec. As we discussed before, spec holds *desired state*, so any "inputs" to our controller go here.

Fundamentally a CronJob needs the following pieces:

- A schedule (the `cron` in CronJob)
- A template for the Job to run (the `job` in CronJob)

We'll also want a few extras, which will make our users' lives easier:

- A deadline for starting jobs (if we miss this deadline, we'll just wait till the next scheduled time)
- What to do if multiple jobs would run at once (do we wait? stop the old one? run both?)
- A way to pause the running of a CronJob, in case something's wrong with it
- Limits on old job history

Remember, since we never read our own status, we need to have some other way to keep track of whether a job has run. We can use at least one old job to do this.

We'll use several markers ( // +comment ) to specify additional metadata. These will be used by controller-tools when generating our CRD manifest. As we'll see in a bit, controller-tools will also use GoDoc to form descriptions for the fields.

```
// CronJobSpec defines the desired state of CronJob.
type CronJobSpec struct {
    // +kubebuilder:validation:MinLength=0

    // The schedule in Cron format, see https://en.wikipedia.org/wiki/Cron.
    Schedule string `json:"schedule"`

    // +kubebuilder:validation:Minimum=0

    // Optional deadline in seconds for starting the job if it misses scheduled
    // time for any reason. Missed jobs executions will be counted as failed
    ones.

    // +optional
    StartingDeadlineSeconds *int64 `json:"startingDeadlineSeconds,omitempty"`

    // Specifies how to treat concurrent executions of a Job.
    // Valid values are:
    // - "Allow" (default): allows CronJobs to run concurrently;
    // - "Forbid": forbids concurrent runs, skipping next run if previous run
    hasn't finished yet;
    // - "Replace": cancels currently running job and replaces it with a new one
    // +optional
    ConcurrencyPolicy ConcurrencyPolicy `json:"concurrencyPolicy,omitempty"`

    // This flag tells the controller to suspend subsequent executions, it does
    // not apply to already started executions. Defaults to false.
    // +optional
    Suspend *bool `json:"suspend,omitempty"`

    // Specifies the job that will be created when executing a CronJob.
    JobTemplate batchv1.JobTemplateSpec `json:"jobTemplate"`

    // +kubebuilder:validation:Minimum=0

    // The number of successful finished jobs to retain.
    // This is a pointer to distinguish between explicit zero and not specified.
    // +optional
    SuccessfulJobsHistoryLimit *int32
    `json:"successfulJobsHistoryLimit,omitempty"`

    // +kubebuilder:validation:Minimum=0

    // The number of failed finished jobs to retain.
    // This is a pointer to distinguish between explicit zero and not specified.
    // +optional
    FailedJobsHistoryLimit *int32 `json:"failedJobsHistoryLimit,omitempty"`
}
```

We define a custom type to hold our concurrency policy. It's actually just a string under the hood, but the type gives extra documentation, and allows us to attach validation on the type instead of the field, making the validation more easily reusable.

```
// ConcurrencyPolicy describes how the job will be handled.  
// Only one of the following concurrent policies may be specified.  
// If none of the following policies is specified, the default one  
// is AllowConcurrent.  
// +kubebuilder:validation:Enum=Allow;Forbid;Replace  
type ConcurrencyPolicy string  
  
const (  
    // AllowConcurrent allows CronJobs to run concurrently.  
    AllowConcurrent ConcurrencyPolicy = "Allow"  
  
    // ForbidConcurrent forbids concurrent runs, skipping next run if previous  
    // hasn't finished yet.  
    ForbidConcurrent ConcurrencyPolicy = "Forbid"  
  
    // ReplaceConcurrent cancels currently running job and replaces it with a new  
    // one.  
    ReplaceConcurrent ConcurrencyPolicy = "Replace"  
)
```

Next, let's design our status, which holds observed state. It contains any information we want users or other controllers to be able to easily obtain.

We'll keep a list of actively running jobs, as well as the last time that we successfully ran our job. Notice that we use `metav1.Time` instead of `time.Time` to get the stable serialization, as mentioned above.

```
// CronJobStatus defines the observed state of CronJob.  
type CronJobStatus struct {  
    // INSERT ADDITIONAL STATUS FIELD - define observed state of cluster  
    // Important: Run "make" to regenerate code after modifying this file  
  
    // A list of pointers to currently running jobs.  
    // +optional  
    Active []corev1.ObjectReference `json:"active,omitempty"`  
  
    // Information when was the last time the job was successfully scheduled.  
    // +optional  
    LastScheduleTime *metav1.Time `json:"lastScheduleTime,omitempty"`  
}
```

Finally, we have the rest of the boilerplate that we've already discussed. As previously noted, we don't need to change this, except to mark that we want a status subresource, so that we behave like built-in kubernetes types.

```
// +kubebuilder:object:root=true
// +kubebuilder:subresource:status

// CronJob is the Schema for the cronjobs API.
type CronJob struct {

// Root Object Definitions (hidden)
```

Now that we have an API, we'll need to write a controller to actually implement the functionality.



# A Brief Aside: What's the rest of this stuff?

If you've taken a peek at the rest of the files in the `api/v1/` directory, you might have noticed two additional files beyond `cronjob_types.go`: `groupversion_info.go` and `zz_generated.deepcopy.go`.

Neither of these files ever needs to be edited (the former stays the same and the latter is autogenerated), but it's useful to know what's in them.

## groupversion\_info.go

`groupversion_info.go` contains common metadata about the group-version:

```
$ vim project/api/v1/groupversion_info.go
// Apache License (hidden)
```

First, we have some *package-level* markers that denote that there are Kubernetes objects in this package, and that this package represents the group `batch.tutorial.kubebuilder.io`. The `object` generator makes use of the former, while the latter is used by the CRD generator to generate the right metadata for the CRDs it creates from this package.

```
// Package v1 contains API Schema definitions for the batch v1 API group.
// +kubebuilder:object:generate=true
// +groupName=batch.tutorial.kubebuilder.io
package v1

import (
    "k8s.io/apimachinery/pkg/runtime/schema"
    "sigs.k8s.io/controller-runtime/pkg/scheme"
)
```

Then, we have the commonly useful variables that help us set up our Scheme. Since we need to use all the types in this package in our controller, it's helpful (and the convention) to have a convenient method to add all the types to some other `Scheme`. `SchemeBuilder` makes this easy for us.

```
var (
    // GroupVersion is group version used to register these objects.
    GroupVersion = schema.GroupVersion{Group: "batch.tutorial.kubebuilder.io",
Version: "v1"}

    // SchemeBuilder is used to add go types to the GroupVersionKind scheme.
    SchemeBuilder = &schema.Builder{GroupVersion: GroupVersion}

    // AddToScheme adds the types in this group-version to the given scheme.
    AddToScheme = SchemeBuilder.AddToScheme
)
```

## zz\_generated.deepcopy.go

`zz_generated.deepcopy.go` contains the autogenerated implementation of the aforementioned `runtime.Object` interface, which marks all of our root types as representing Kinds.

The core of the `runtime.Object` interface is a deep-copy method, `DeepCopyObject`.

The `object` generator in controller-tools also generates two other handy methods for each root type and all its sub-types: `DeepCopy` and `DeepCopyInto`.

# What's in a controller?

Controllers are the core of Kubernetes, and of any operator.

It's a controller's job to ensure that, for any given object, the actual state of the world (both the cluster state, and potentially external state like running containers for Kubelet or loadbalancers for a cloud provider) matches the desired state in the object. Each controller focuses on one *root Kind*, but may interact with other Kinds.

We call this process *reconciling*.

In controller-runtime, the logic that implements the reconciling for a specific kind is called a *Reconciler*. A reconciler takes the name of an object, and returns whether or not we need to try again (e.g. in case of errors or periodic controllers, like the HorizontalPodAutoscaler).

```
$ vim emptycontroller.go
// Apache License (hidden)
```

First, we start out with some standard imports. As before, we need the core controller-runtime library, as well as the client package, and the package for our API types.

```
package controllers

import (
    "context"

    "k8s.io/apimachinery/pkg/runtime"
    ctrl "sigs.k8s.io/controller-runtime"
    "sigs.k8s.io/controller-runtime/pkg/client"
    logf "sigs.k8s.io/controller-runtime/pkg/log"

    batchv1 "tutorial.kubebuilder.io/project/api/v1"
)
```

Next, kubebuilder has scaffolded a basic reconciler struct for us. Pretty much every reconciler needs to log, and needs to be able to fetch objects, so these are added out of the box.

```
// CronJobReconciler reconciles a CronJob object
type CronJobReconciler struct {
    client.Client
    Scheme *runtime.Scheme
}
```

Most controllers eventually end up running on the cluster, so they need RBAC permissions, which we specify using controller-tools [RBAC markers](#). These are the bare minimum

permissions needed to run. As we add more functionality, we'll need to revisit these.

```
//  
+kubebuilder:rbac:groups=batch.tutorial.kubebuilder.io,resources=cronjobs,verbs=ge  
t;list;watch;create;update;patch;delete  
//  
+kubebuilder:rbac:groups=batch.tutorial.kubebuilder.io,resources=cronjobs/status,v  
erbs=get;update;patch
```

The `ClusterRole` manifest at `config/rbac/role.yaml` is generated from the above markers via controller-gen with the following command:

```
// make manifests
```

NOTE: If you receive an error, please run the specified command in the error and re-run `make manifests`.

`Reconcile` actually performs the reconciling for a single named object. Our `Request` just has a name, but we can use the client to fetch that object from the cache.

We return an empty result and no error, which indicates to controller-runtime that we've successfully reconciled this object and don't need to try again until there's some changes.

Most controllers need a logging handle and a context, so we set them up here.

The `context` is used to allow cancellation of requests, and potentially things like tracing. It's the first argument to all client methods. The `Background` context is just a basic context without any extra data or timing restrictions.

The logging handle lets us log. controller-runtime uses structured logging through a library called `logr`. As we'll see shortly, logging works by attaching key-value pairs to a static message. We can pre-assign some pairs at the top of our reconcile method to have those attached to all log lines in this reconciler.

```
func (r *CronJobReconciler) Reconcile(ctx context.Context, req ctrl.Request)  
(ctrl.Result, error) {  
    _ = logf.FromContext(ctx)  
  
    // your logic here  
  
    return ctrl.Result{}, nil  
}
```

Finally, we add this reconciler to the manager, so that it gets started when the manager is started.

For now, we just note that this reconciler operates on `CronJob`s. Later, we'll use this to mark that we care about related objects as well.

```
func (r *CronJobReconciler) SetupWithManager(mgr ctrl.Manager) error {
    return ctrl.NewControllerManagedBy(mgr).
        For(&batchv1.CronJob{}).
        Complete(r)
}
```

Now that we've seen the basic structure of a reconciler, let's fill out the logic for `CronJob`s.

# Implementing a controller

The basic logic of our CronJob controller is this:

1. Load the named CronJob
2. List all active jobs, and update the status
3. Clean up old jobs according to the history limits
4. Check if we're suspended (and don't do anything else if we are)
5. Get the next scheduled run
6. Run a new job if it's on schedule, not past the deadline, and not blocked by our concurrency policy
7. Requeue when we either see a running job (done automatically) or it's time for the next scheduled run.

```
$ vim project/internal/controller/cronjob_controller.go
// Apache License (hidden)
```

We'll start out with some imports. You'll see below that we'll need a few more imports than those scaffolded for us. We'll talk about each one when we use it.

```
package controller

import (
    "context"
    "fmt"
    "sort"
    "time"

    "github.com/robfig/cron"
    kbatch "k8s.io/api/batch/v1"
    corev1 "k8s.io/api/core/v1"
    metav1 "k8s.io/apimachinery/pkg/apis/meta/v1"
    "k8s.io/apimachinery/pkg/runtime"
    ref "k8s.io/client-go/tools/reference"
    ctrl "sigs.k8s.io/controller-runtime"
    "sigs.k8s.io/controller-runtime/pkg/client"
    logf "sigs.k8s.io/controller-runtime/pkg/log"

    batchv1 "tutorial.kubebuilder.io/project/api/v1"
)
```

Next, we'll need a Clock, which will allow us to fake timing in our tests.

```
// CronJobReconciler reconciles a CronJob object
type CronJobReconciler struct {
    client.Client
    Scheme *runtime.Scheme
    Clock
}

// Clock (hidden)
```

Notice that we need a few more RBAC permissions – since we're creating and managing jobs now, we'll need permissions for those, which means adding a couple more [markers](#).

```
// +kubebuilder:rbac:groups=batch.tutorial.kubebuilder.io,resources=cronjobs,verbs=ge
t;list;watch;create;update;patch;delete
//
+kubebuilder:rbac:groups=batch.tutorial.kubebuilder.io,resources=cronjobs/status,v
erbs=get;update;patch
//
+kubebuilder:rbac:groups=batch.tutorial.kubebuilder.io,resources=cronjobs/finalize
rs,verbs=update
//
+kubebuilder:rbac:groups=batch,resources=jobs,verbs=get;list;watch;create;update;p
atch;delete
// +kubebuilder:rbac:groups=batch,resources=jobs/status,verbs=get
```

Now, we get to the heart of the controller – the reconciler logic.

```
var (
    scheduledTimeAnnotation = "batch.tutorial.kubebuilder.io/scheduled-at"
)

// Reconcile is part of the main kubernetes reconciliation loop which aims to
// move the current state of the cluster closer to the desired state.
// TODO(user): Modify the Reconcile function to compare the state specified by
// the CronJob object against the actual cluster state, and then
// perform operations to make the cluster state reflect the state specified by
// the user.
//
// For more details, check Reconcile and its Result here:
// - https://pkg.go.dev/sigs.k8s.io/controller-runtime@v0.20.4/pkg/reconcile
// nolint:gocyclo
func (r *CronJobReconciler) Reconcile(ctx context.Context, req ctrl.Request)
(ctrl.Result, error) {
    log := logf.FromContext(ctx)
```

## 1: Load the CronJob by name

We'll fetch the CronJob using our client. All client methods take a context (to allow for cancellation) as their first argument, and the object in question as their last. Get is a bit special, in that it takes a `NamespacedName` as the middle argument (most don't have a middle argument, as we'll see below).

Many client methods also take variadic options at the end.

```
var cronJob batchv1.CronJob
if err := r.Get(ctx, req.NamespacedName, &cronJob); err != nil {
    log.Error(err, "unable to fetch CronJob")
    // we'll ignore not-found errors, since they can't be fixed by an
immediate
    // requeue (we'll need to wait for a new notification), and we can get
them
    // on deleted requests.
    return ctrl.Result{}, client.IgnoreNotFound(err)
}
```

## 2: List all active jobs, and update the status

To fully update our status, we'll need to list all child jobs in this namespace that belong to this CronJob. Similarly to Get, we can use the List method to list the child jobs. Notice that we use variadic options to set the namespace and field match (which is actually an index lookup that we set up below).

```
var childJobs kbatch.JobList
if err := r.List(ctx, &childJobs, client.InNamespace(req.Namespace),
client.MatchingFields{jobOwnerKey: req.Name}); err != nil {
    log.Error(err, "unable to list child Jobs")
    return ctrl.Result{}, err
}
```

What is this index about?

The reconciler fetches all jobs owned by the cronjob for the status. As our number of cronjobs increases, looking these up can become quite slow as we have to filter through all of them. For a more efficient lookup, these jobs will be indexed locally on the controller's name. A jobOwnerKey field is added to the cached job objects. This key references the owning controller and functions as the index. Later in this document we will configure the manager to actually index this field.

Once we have all the jobs we own, we'll split them into active, successful, and failed jobs, keeping track of the most recent run so that we can record it in status. Remember, status should be able to be reconstituted from the state of the world, so it's generally not a good idea to read from the status of the root object. Instead, you should reconstruct it every run. That's what we'll do here.

We can check if a job is "finished" and whether it succeeded or failed using status conditions. We'll put that logic in a helper to make our code cleaner.

```
// find the active list of jobs
var activeJobs []*kbatch.Job
var successfulJobs []*kbatch.Job
var failedJobs []*kbatch.Job
var mostRecentTime *time.Time // find the last run so we can update the status

// isJobFinished (hidden)
// getScheduledTimeForJob (hidden)
```

```

for i, job := range childJobs.Items {
    _, finishedType := isJobFinished(&job)
    switch finishedType {
    case "": // ongoing
        activeJobs = append(activeJobs, &childJobs.Items[i])
    case kbatch.JobFailed:
        failedJobs = append(failedJobs, &childJobs.Items[i])
    case kbatch.JobComplete:
        successfulJobs = append(successfulJobs, &childJobs.Items[i])
    }

    // We'll store the launch time in an annotation, so we'll reconstitute
    // that from
    // the active jobs themselves.
    scheduledTimeForJob, err := getScheduledTimeForJob(&job)
    if err != nil {
        log.Error(err, "unable to parse schedule time for child job", "job",
&job)
        continue
    }
    if scheduledTimeForJob != nil {
        if mostRecentTime == nil ||
mostRecentTime.Before(*scheduledTimeForJob) {
            mostRecentTime = scheduledTimeForJob
        }
    }
}

if mostRecentTime != nil {
    cronJob.Status.LastScheduleTime = &metav1.Time{Time: *mostRecentTime}
} else {
    cronJob.Status.LastScheduleTime = nil
}
cronJob.Status.Active = nil
for _, activeJob := range activeJobs {
    jobRef, err := ref.GetReference(r.Scheme, activeJob)
    if err != nil {
        log.Error(err, "unable to make reference to active job", "job",
activeJob)
        continue
    }
    cronJob.Status.Active = append(cronJob.Status.Active, *jobRef)
}

```

Here, we'll log how many jobs we observed at a slightly higher logging level, for debugging. Notice how instead of using a format string, we use a fixed message, and attach key-value pairs with the extra information. This makes it easier to filter and query log lines.

```

log.V(1).Info("job count", "active jobs", len(activeJobs), "successful jobs",
len(successfulJobs), "failed jobs", len(failedJobs))

```

Using the data we've gathered, we'll update the status of our CRD. Just like before, we use our client. To specifically update the status subresource, we'll use the `Status` part of the client, with the `Update` method.

The status subresource ignores changes to spec, so it's less likely to conflict with any other updates, and can have separate permissions.

```
if err := r.Status().Update(ctx, &cronJob); err != nil {
    log.Error(err, "unable to update CronJob status")
    return ctrl.Result{}, err
}
```

Once we've updated our status, we can move on to ensuring that the status of the world matches what we want in our spec.

### 3: Clean up old jobs according to the history limit

First, we'll try to clean up old jobs, so that we don't leave too many lying around.

```

// NB: deleting these are "best effort" -- if we fail on a particular one,
// we won't requeue just to finish the deleting.
if cronJob.Spec.FailedJobsHistoryLimit != nil {
    sort.Slice(failedJobs, func(i, j int) bool {
        if failedJobs[i].Status.StartTime == nil {
            return failedJobs[j].Status.StartTime != nil
        }
        return
    })
    for i, job := range failedJobs {
        if int32(i) >= int32(len(failedJobs))- *cronJob.Spec.FailedJobsHistoryLimit {
            break
        }
        if err := r.Delete(ctx, job,
client.PropagationPolicy metav1.DeletePropagationBackground));
client.IgnoreNotFound(err) != nil {
            log.Error(err, "unable to delete old failed job", "job", job)
        } else {
            log.V(0).Info("deleted old failed job", "job", job)
        }
    }
}

if cronJob.Spec.SuccessfulJobsHistoryLimit != nil {
    sort.Slice(successfulJobs, func(i, j int) bool {
        if successfulJobs[i].Status.StartTime == nil {
            return successfulJobs[j].Status.StartTime != nil
        }
        return
    })
    for i, job := range successfulJobs {
        if int32(i) >= int32(len(successfulJobs))- *cronJob.Spec.SuccessfulJobsHistoryLimit {
            break
        }
        if err := r.Delete(ctx, job,
client.PropagationPolicy metav1.DeletePropagationBackground)); err != nil {
            log.Error(err, "unable to delete old successful job", "job", job)
        } else {
            log.V(0).Info("deleted old successful job", "job", job)
        }
    }
}

```

## 4: Check if we're suspended

If this object is suspended, we don't want to run any jobs, so we'll stop now. This is useful if something's broken with the job we're running and we want to pause runs to investigate or putz with the cluster, without deleting the object.

```
if cronJob.Spec.Suspend != nil && *cronJob.Spec.Suspend {  
    log.V(1).Info("cronjob suspended, skipping")  
    return ctrl.Result{}, nil  
}
```

## 5: Get the next scheduled run

If we're not paused, we'll need to calculate the next scheduled run, and whether or not we've got a run that we haven't processed yet.

```
// getNextSchedule (hidden)  
  
    // figure out the next times that we need to create  
    // jobs at (or anything we missed).  
    missedRun, nextRun, err := getNextSchedule(&cronJob, r.Now())  
    if err != nil {  
        log.Error(err, "unable to figure out CronJob schedule")  
        // we don't really care about requeuing until we get an update that  
        // fixes the schedule, so don't return an error  
        return ctrl.Result{}, nil  
    }
```

We'll prep our eventual request to requeue until the next job, and then figure out if we actually need to run.

```
scheduledResult := ctrl.Result{RequeueAfter: nextRun.Sub(r.Now())} // save  
this so we can re-use it elsewhere  
log = log.WithValues("now", r.Now(), "next run", nextRun)
```

## 6: Run a new job if it's on schedule, not past the deadline, and not blocked by our concurrency policy

If we've missed a run, and we're still within the deadline to start it, we'll need to run a job.

```

if missedRun.IsZero() {
    log.V(1).Info("no upcoming scheduled times, sleeping until next")
    return scheduledResult, nil
}

// make sure we're not too late to start the run
log = log.WithValues("current run", missedRun)
tooLate := false
if cronJob.Spec.StartingDeadlineSeconds != nil {
    tooLate =
missedRun.Add(time.Duration(*cronJob.Spec.StartingDeadlineSeconds) *
time.Second).Before(r.Now())
}
if tooLate {
    log.V(1).Info("missed starting deadline for last run, sleeping till next")
    // TODO(directxman12): events
    return scheduledResult, nil
}

```

If we actually have to run a job, we'll need to either wait till existing ones finish, replace the existing ones, or just add new ones. If our information is out of date due to cache delay, we'll get a requeue when we get up-to-date information.

```

// figure out how to run this job -- concurrency policy might forbid us from
running
// multiple at the same time...
if cronJob.SpecConcurrencyPolicy == batchv1.ForbidConcurrent &&
len(activeJobs) > 0 {
    log.V(1).Info("concurrency policy blocks concurrent runs, skipping", "num
active", len(activeJobs))
    return scheduledResult, nil
}

// ...or instruct us to replace existing ones...
if cronJob.SpecConcurrencyPolicy == batchv1.ReplaceConcurrent {
    for _, activeJob := range activeJobs {
        // we don't care if the job was already deleted
        if err := r.Delete(ctx, activeJob,
client.PropagationPolicy(metav1.DeletePropagationBackground));
client.IgnoreNotFound(err) != nil {
            log.Error(err, "unable to delete active job", "job", activeJob)
            return ctrl.Result{}, err
        }
    }
}

```

Once we've figured out what to do with existing jobs, we'll actually create our desired job

```
// constructJobForCronJob (hidden)
```

```

// actually make the job...
job, err := constructJobForCronJob(&cronJob, missedRun)
if err != nil {
    log.Error(err, "unable to construct job from template")
    // don't bother requeueing until we get a change to the spec
    return scheduledResult, nil
}

// ...and create it on the cluster
if err := r.Create(ctx, job); err != nil {
    log.Error(err, "unable to create Job for CronJob", "job", job)
    return ctrl.Result{}, err
}

log.V(1).Info("created Job for CronJob run", "job", job)

```

## 7: Requeue when we either see a running job or it's time for the next scheduled run

Finally, we'll return the result that we prepped above, that says we want to requeue when our next run would need to occur. This is taken as a maximum deadline – if something else changes in between, like our job starts or finishes, we get modified, etc, we might reconcile again sooner.

```

// we'll requeue once we see the running job, and update our status
return scheduledResult, nil
}

```

## Setup

Finally, we'll update our setup. In order to allow our reconciler to quickly look up Jobs by their owner, we'll need an index. We declare an index key that we can later use with the client as a pseudo-field name, and then describe how to extract the indexed value from the Job object. The indexer will automatically take care of namespaces for us, so we just have to extract the owner name if the Job has a CronJob owner.

Additionally, we'll inform the manager that this controller owns some Jobs, so that it will automatically call Reconcile on the underlying CronJob when a Job changes, is deleted, etc.

```

var (
    jobOwnerKey = ".metadata.controller"
    apiGVStr   = batchv1.GroupVersion.String()
)

// SetupWithManager sets up the controller with the Manager.
func (r *CronJobReconciler) SetupWithManager(mgr ctrl.Manager) error {
    // set up a real clock, since we're not in a test
    if r.Clock == nil {
        r.Clock = realClock{}
    }

    if err := mgr.GetFieldIndexer().IndexField(context.Background(),
&kbatch.Job{}, jobOwnerKey, func(rawObj client.Object) []string {
        // grab the job object, extract the owner...
        job := rawObj.(*kbatch.Job)
        owner := metav1.GetControllerOf(job)
        if owner == nil {
            return nil
        }
        // ...make sure it's a CronJob...
        if owner.APIVersion != apiGVStr || owner.Kind != "CronJob" {
            return nil
        }

        // ...and if so, return it
        return []string{owner.Name}
}); err != nil {
    return err
}

return ctrl.NewControllerManagedBy(mgr).
    For(&batchv1.CronJob{}).
    Owns(&kbatch.Job{}).
    Named("cronjob").
    Complete(r)
}

```

That was a doozy, but now we've got a working controller. Let's test against the cluster, then, if we don't have any issues, deploy it!

# You said something about main?

But first, remember how we said we'd [come back to `main.go` again](#)? Let's take a look and see what's changed, and what we need to add.

```
$ vim project/cmd/main.go
// Apache License (hidden)
// Imports (hidden)
```

The first difference to notice is that kubebuilder has added the new API group's package (`batchv1`) to our scheme. This means that we can use those objects in our controller.

If we would be using any other CRD we would have to add their scheme the same way. Built-in types such as `Job` have their scheme added by `clientgoscheme`.

```
var (
    scheme = runtime.NewScheme()
    setupLog = ctrl.Log.WithName("setup")
)

func init() {
    utilruntime.Must(clientgoscheme.AddToScheme(scheme))

    utilruntime.Must(batchv1.AddToScheme(scheme))
    // +kubebuilder:scaffold:scheme
}
```

The other thing that's changed is that kubebuilder has added a block calling our `CronJob` controller's `SetupWithManager` method.

```
// nolint:gocyclo
func main() {

    // old stuff (hidden)

    if err = (&controller.CronJobReconciler{
        Client: mgr.GetClient(),
        Scheme: mgr.GetScheme(),
    }).SetupWithManager(mgr); err != nil {
        setupLog.Error(err, "unable to create controller", "controller",
"CronJob")
        os.Exit(1)
    }

    // old stuff (hidden)
```

}

Now we can implement our controller.

# Implementing defaulting/validating webhooks

If you want to implement [admission webhooks](#) for your CRD, the only thing you need to do is to implement the `CustomDefaulter` and (or) the `CustomValidator` interface.

Kubebuilder takes care of the rest for you, such as

1. Creating the webhook server.
2. Ensuring the server has been added in the manager.
3. Creating handlers for your webhooks.
4. Registering each handler with a path in your server.

First, let's scaffold the webhooks for our CRD (CronJob). We'll need to run the following command with the `--defaulting` and `--programmatic-validation` flags (since our test project will use defaulting and validating webhooks):

```
kubebuilder create webhook --group batch --version v1 --kind CronJob --defaulting  
--programmatic-validation
```

This will scaffold the webhook functions and register your webhook with the manager in your `main.go` for you.

```
$ vim project/internal/webhook/v1/cronjob_webhook.go  
// Apache License (hidden)  
// Go imports (hidden)
```

Next, we'll setup a logger for the webhooks.

```
var cronjoblog = logf.Log.WithName("cronjob-resource")
```

Then, we set up the webhook with the manager.

```
// SetupCronJobWebhookWithManager registers the webhook for CronJob in the
manager.
func SetupCronJobWebhookWithManager(mgr ctrl.Manager) error {
    return ctrl.NewWebhookManagedBy(mgr).For(&batchv1.CronJob{}).
        WithValidator(&CronJobCustomValidator{}).
        WithDefaulter(&CronJobCustomDefaulter{
            DefaultConcurrencyPolicy:           batchv1.AllowConcurrent,
            DefaultSuspend:                   false,
            DefaultSuccessfulJobsHistoryLimit: 3,
            DefaultFailedJobsHistoryLimit:     1,
        }).
        Complete()
}
```

Notice that we use kubebuilder markers to generate webhook manifests. This marker is responsible for generating a mutating webhook manifest.

The meaning of each marker can be found [here](#).

This marker is responsible for generating a mutation webhook manifest.

```
// +kubebuilder:webhook:path=/mutate-batch-tutorial-kubebuilder-io-v1-
cronjob,mutating=true,failurePolicy=fail,sideEffects=None,groups=batch.tutorial.ku
bebuilder.io,resources=cronjobs,verbs=create;update,versions=v1,name=mcronjob-
v1.kb.io,admissionReviewVersions=v1

// CronJobCustomDefaulter struct is responsible for setting default values on the
// custom resource of the
// Kind CronJob when those are created or updated.
//
// NOTE: The +kubebuilder:object:generate=false marker prevents controller-gen
from generating DeepCopy methods,
// as it is used only for temporary operations and does not need to be deeply
copied.
type CronJobCustomDefaulter struct {

    // Default values for various CronJob fields
    DefaultConcurrencyPolicy      batchv1.ConcurrencyPolicy
    DefaultSuspend                bool
    DefaultSuccessfulJobsHistoryLimit int32
    DefaultFailedJobsHistoryLimit  int32
}

var _ webhook.CustomDefaulter = &CronJobCustomDefaulter{}
```

We use the `webhook.CustomDefaulter` interface to set defaults to our CRD. A webhook will automatically be served that calls this defaulting.

The `Default` method is expected to mutate the receiver, setting the defaults.

```

// Default implements webhook.CustomDefaulter so a webhook will be registered for
// the Kind CronJob.
func (d *CronJobCustomDefaulter) Default(ctx context.Context, obj runtime.Object)
error {
    cronjob, ok := obj.(*batchv1.CronJob)

    if !ok {
        return fmt.Errorf("expected an CronJob object but got %T", obj)
    }
    cronjoblog.Info("Defaulting for CronJob", "name", cronjob.GetName())

    // Set default values
    d.applyDefaults(cronjob)
    return nil
}

// applyDefaults applies default values to CronJob fields.
func (d *CronJobCustomDefaulter) applyDefaults(cronJob *batchv1.CronJob) {
    if cronJob.Spec.ConcurrencyPolicy == "" {
        cronJob.Spec.ConcurrencyPolicy = d.DefaultConcurrencyPolicy
    }
    if cronJob.Spec.Suspend == nil {
        cronJob.Spec.Suspend = new(bool)
        *cronJob.Spec.Suspend = d.DefaultSuspend
    }
    if cronJob.Spec.SuccessfulJobsHistoryLimit == nil {
        cronJob.Spec.SuccessfulJobsHistoryLimit = new(int32)
        *cronJob.Spec.SuccessfulJobsHistoryLimit =
d.DefaultSuccessfulJobsHistoryLimit
    }
    if cronJob.Spec.FailedJobsHistoryLimit == nil {
        cronJob.Spec.FailedJobsHistoryLimit = new(int32)
        *cronJob.Spec.FailedJobsHistoryLimit = d.DefaultFailedJobsHistoryLimit
    }
}

```

We can validate our CRD beyond what's possible with declarative validation. Generally, declarative validation should be sufficient, but sometimes more advanced use cases call for complex validation.

For instance, we'll see below that we use this to validate a well-formed cron schedule without making up a long regular expression.

If `webhook.CustomValidator` interface is implemented, a webhook will automatically be served that calls the validation.

The `ValidateCreate`, `ValidateUpdate` and `ValidateDelete` methods are expected to validate its receiver upon creation, update and deletion respectively. We separate out `ValidateCreate` from `ValidateUpdate` to allow behavior like making certain fields immutable, so that they can only be set on creation. `ValidateDelete` is also separated from `ValidateUpdate` to allow different

validation behavior on deletion. Here, however, we just use the same shared validation for `ValidateCreate` and `ValidateUpdate`. And we do nothing in `ValidateDelete`, since we don't need to validate anything on deletion.

This marker is responsible for generating a validation webhook manifest.

```
// +kubebuilder:webhook:path=/validate-batch-tutorial-kubebuilder-io-v1-
cronjob,mutating=false,failurePolicy=fail,sideEffects=None,groups=batch.tutorial.k
ubebuilder.io,resources=cronjobs,verbs=create;update,versions=v1,name=vcronjob-
v1.kb.io,admissionReviewVersions=v1

// CronJobCustomValidator struct is responsible for validating the CronJob
resource
// when it is created, updated, or deleted.
//
// NOTE: The +kubebuilder:object:generate=false marker prevents controller-gen
from generating DeepCopy methods,
// as this struct is used only for temporary operations and does not need to be
deeply copied.
type CronJobCustomValidator struct {
    // TODO(user): Add more fields as needed for validation
}

var _ webhook.CustomValidator = &CronJobCustomValidator{}

// ValidateCreate implements webhook.CustomValidator so a webhook will be
registered for the type CronJob.
func (v *CronJobCustomValidator) ValidateCreate(ctx context.Context, obj
runtime.Object) (admission.Warnings, error) {
    cronjob, ok := obj.(*batchv1.CronJob)
    if !ok {
        return nil, fmt.Errorf("expected a CronJob object but got %T", obj)
    }
    cronjoblog.Info("Validation for CronJob upon creation", "name",
cronjob.GetName())

    return nil, validateCronJob(cronjob)
}

// ValidateUpdate implements webhook.CustomValidator so a webhook will be
registered for the type CronJob.
func (v *CronJobCustomValidator) ValidateUpdate(ctx context.Context, oldObj,
newObj runtime.Object) (admission.Warnings, error) {
    cronjob, ok := newObj.(*batchv1.CronJob)
    if !ok {
        return nil, fmt.Errorf("expected a CronJob object for the newObj but got
%T", newObj)
    }
    cronjoblog.Info("Validation for CronJob upon update", "name",
cronjob.GetName())

    return nil, validateCronJob(cronjob)
}

// ValidateDelete implements webhook.CustomValidator so a webhook will be
registered for the type CronJob.
func (v *CronJobCustomValidator) ValidateDelete(ctx context.Context, obj
runtime.Object) (admission.Warnings, error) {
    cronjob, ok := obj.(*batchv1.CronJob)
```

```

if !ok {
    return nil, fmt.Errorf("expected a CronJob object but got %T", obj)
}
cronjoblog.Info("Validation for CronJob upon deletion", "name",
cronjob.GetName())

// TODO(user): fill in your validation logic upon object deletion.

return nil, nil
}

```

We validate the name and the spec of the CronJob.

```

// validateCronJob validates the fields of a CronJob object.
func validateCronJob(cronjob *batchv1.CronJob) error {
    var allErrs field.ErrorList
    if err := validateCronJobName(cronjob); err != nil {
        allErrs = append(allErrs, err)
    }
    if err := validateCronJobSpec(cronjob); err != nil {
        allErrs = append(allErrs, err)
    }
    if len(allErrs) == 0 {
        return nil
    }

    return apierrors.NewInvalid(
        schema.GroupKind{Group: "batch.tutorial.kubebuilder.io", Kind: "CronJob"},
        cronjob.Name, allErrs)
}

```

Some fields are declaratively validated by OpenAPI schema. You can find kubebuilder validation markers (prefixed with `// +kubebuilder:validation`) in the [Designing an API](#) section. You can find all of the kubebuilder supported markers for declaring validation by running `controller-gen crd -w`, or [here](#).

```

func validateCronJobSpec(cronjob *batchv1.CronJob) *field.Error {
    // The field helpers from the kubernetes API machinery help us return nicely
    // structured validation errors.
    return validateScheduleFormat(
        cronjob.Spec.Schedule,
        field.NewPath("spec").Child("schedule"))
}

```

We'll need to validate the `cron` schedule is well-formatted.

```
func validateScheduleFormat(schedule string, fldPath *field.Path) *field.Error {
    if _, err := cron.ParseStandard(schedule); err != nil {
        return field.Invalid(fldPath, schedule, err.Error())
    }
    return nil
}

// Validate object name (hidden)
```

# Running and deploying the controller

## Optional

If opting to make any changes to the API definitions, then before proceeding, generate the manifests like CRs or CRDs with

```
make manifests
```

To test out the controller, we can run it locally against the cluster. Before we do so, though, we'll need to install our CRDs, as per the [quick start](#). This will automatically update the YAML manifests using controller-tools, if needed:

```
make install
```

Now that we've installed our CRDs, we can run the controller against our cluster. This will use whatever credentials that we connect to the cluster with, so we don't need to worry about RBAC just yet.

### Running webhooks locally

If you want to run the webhooks locally, you'll have to generate certificates for serving the webhooks, and place them in the right directory ( `/tmp/k8s-webhook-server/serving-certs/tls.{crt,key}` , by default).

If you're not running a local API server, you'll also need to figure out how to proxy traffic from the remote cluster to your local webhook server. For this reason, we generally recommend disabling webhooks when doing your local code-run-test cycle, as we do below.

In a separate terminal, run

```
export ENABLE_WEBHOOKS=false  
make run
```

You should see logs from the controller about starting up, but it won't do anything just yet.

At this point, we need a CronJob to test with. Let's write a sample to `config/samples/batch_v1_cronjob.yaml`, and use that:

```
apiVersion: batch.tutorial.kubebuilder.io/v1
kind: CronJob
metadata:
  labels:
    app.kubernetes.io/name: project
    app.kubernetes.io/managed-by: kustomize
  name: cronjob-sample
spec:
  schedule: "*/1 * * * *"
  startingDeadlineSeconds: 60
  concurrencyPolicy: Allow # explicitly specify, but Allow is also default.
  jobTemplate:
    spec:
      template:
        spec:
          containers:
            - name: hello
              image: busybox
              args:
                - /bin/sh
                - -c
                - date; echo Hello from the Kubernetes cluster
          restartPolicy: OnFailure
```

```
kubectl create -f config/samples/batch_v1_cronjob.yaml
```

At this point, you should see a flurry of activity. If you watch the changes, you should see your cronjob running, and updating status:

```
kubectl get cronjob.batch.tutorial.kubebuilder.io -o yaml
kubectl get job
```

Now that we know it's working, we can run it in the cluster. Stop the `make run` invocation, and run

```
make docker-build docker-push IMG=<some-registry>/<project-name>:tag
make deploy IMG=<some-registry>/<project-name>:tag
```

### Registry Permission

This image ought to be published in the personal registry you specified. And it is required to have access to pull the image from the working environment. Make sure you have the proper permission to the registry if the above commands don't work.

Consider incorporating Kind into your workflow for a faster, more efficient local development and CI experience. Note that, if you're using a Kind cluster, there's no need to

push your image to a remote container registry. You can directly load your local image into your specified Kind cluster:

```
kind load docker-image <your-image-name>:tag --name <your-kind-cluster-name>
```

To know more, see: [Using Kind For Development Purposes and CI](#)

#### RBAC errors

If you encounter RBAC errors, you may need to grant yourself cluster-admin privileges or be logged in as admin. See [Prerequisites for using Kubernetes RBAC on GKE cluster v1.11.x and older](#) which may be your case.

If we list cronjobs again like we did before, we should see the controller functioning again!

# Deploying cert-manager

We suggest using [cert-manager](#) for provisioning the certificates for the webhook server. Other solutions should also work as long as they put the certificates in the desired location.

You can follow [the cert-manager documentation](#) to install it.

cert-manager also has a component called [CA Injector](#), which is responsible for injecting the CA bundle into the [MutatingWebhookConfiguration](#) / [ValidatingWebhookConfiguration](#).

To accomplish that, you need to use an annotation with key `cert-manager.io/inject-ca-from` in the [MutatingWebhookConfiguration](#) / [ValidatingWebhookConfiguration](#) objects. The value of the annotation should point to an existing [certificate request instance](#) in the format of `<certificate-namespace>/<certificate-name>`.

This is the [kustomize](#) patch we used for annotating the [MutatingWebhookConfiguration](#) / [ValidatingWebhookConfiguration](#) objects.

# Deploying Admission Webhooks

## cert-manager

You need to follow [this](#) to install the cert-manager bundle.

## Build your image

Run the following command to build your image locally.

```
make docker-build docker-push IMG=<some-registry>/<project-name>:tag
```

### Using Kind

Consider incorporating Kind into your workflow for a faster, more efficient local development and CI experience. Note that, if you're using a Kind cluster, there's no need to push your image to a remote container registry. You can directly load your local image into your specified Kind cluster:

```
kind load docker-image <your-image-name>:tag --name <your-kind-cluster-name>
```

To know more, see: [Using Kind For Development Purposes and CI](#)

## Deploy Webhooks

You need to enable the webhook and cert manager configuration through kustomize. `config/default/kustomization.yaml` should now look like the following:

```

# Adds namespace to all resources.
namespace: project-system

# Value of this field is prepended to the
# names of all resources, e.g. a deployment named
# "wordpress" becomes "alices-wordpress".
# Note that it should also match with the prefix (text before '-') of the
namespace
# field above.
namePrefix: project-

# Labels to add to all resources and selectors.
#labels:
#- includeSelectors: true
#  pairs:
#    someName: someValue

resources:
- ./crd
- ./rbac
- ./manager
# [WEBHOOK] To enable webhook, uncomment all the sections with [WEBHOOK] prefix
including the one in
# crd/kustomization.yaml
- ./webhook
# [CERTMANAGER] To enable cert-manager, uncomment all sections with 'CERTMANAGER'.
'WEBHOOK' components are required.
- ./certmanager
# [PROMETHEUS] To enable prometheus monitor, uncomment all sections with
'PROMETHEUS'.
- ./prometheus
# [METRICS] Expose the controller manager metrics service.
- metrics_service.yaml
# [NETWORK POLICY] Protect the /metrics endpoint and Webhook Server with
NetworkPolicy.
# Only Pod(s) running a namespace labeled with 'metrics: enabled' will be able to
gather the metrics.
# Only CR(s) which requires webhooks and are applied on namespaces labeled with
'webhooks: enabled' will
# be able to communicate with the Webhook Server.
#- ./network-policy

# Uncomment the patches line if you enable Metrics
patches:
# [METRICS] The following patch will enable the metrics endpoint using HTTPS and
the port :8443.
# More info: https://book.kubebuilder.io/reference/metrics
- path: manager_metrics_patch.yaml
  target:
    kind: Deployment

# Uncomment the patches line if you enable Metrics and CertManager
# [METRICS-WITH-CERTS] To enable metrics protected with certManager, uncomment the

```

```
following line.
# This patch will protect the metrics with certManager self-signed certs.
- path: cert_metrics_manager_patch.yaml
  target:
    kind: Deployment

# [WEBHOOK] To enable webhook, uncomment all the sections with [WEBHOOK] prefix
including the one in
# crd/kustomization.yaml
- path: manager_webhook_patch.yaml
  target:
    kind: Deployment

# [CERTMANAGER] To enable cert-manager, uncomment all sections with 'CERTMANAGER'
prefix.
# Uncomment the following replacements to add the cert-manager CA injection
annotations
replacements:
- source: # Uncomment the following block to enable certificates for metrics
  kind: Service
  version: v1
  name: controller-manager-metrics-service
  fieldPath: metadata.name
targets:
- select:
  kind: Certificate
  group: cert-manager.io
  version: v1
  name: metrics-certs
  fieldPaths:
  - spec.dnsNames.0
  - spec.dnsNames.1
  options:
    delimiter: '.'
    index: 0
    create: true
- select: # Uncomment the following to set the Service name for TLS config in
Prometheus ServiceMonitor
  kind: ServiceMonitor
  group: monitoring.coreos.com
  version: v1
  name: controller-manager-metrics-monitor
  fieldPaths:
  - spec.endpoints.0.tlsConfig.serverName
  options:
    delimiter: '.'
    index: 0
    create: true

- source:
  kind: Service
  version: v1
  name: controller-manager-metrics-service
  fieldPath: metadata.namespace
```

```
targets:
- select:
    kind: Certificate
    group: cert-manager.io
    version: v1
    name: metrics-certs
  fieldPaths:
    - spec.dnsNames.0
    - spec.dnsNames.1
  options:
    delimiter: '.'
    index: 1
    create: true
- select: # Uncomment the following to set the Service namespace for TLS in
Prometheus ServiceMonitor
    kind: ServiceMonitor
    group: monitoring.coreos.com
    version: v1
    name: controller-manager-metrics-monitor
  fieldPaths:
    - spec.endpoints.0.tlsConfig.serverName
  options:
    delimiter: '.'
    index: 1
    create: true

- source: # Uncomment the following block if you have any webhook
  kind: Service
  version: v1
  name: webhook-service
  fieldPath: .metadata.name # Name of the service
targets:
- select:
    kind: Certificate
    group: cert-manager.io
    version: v1
    name: serving-cert
  fieldPaths:
    - .spec.dnsNames.0
    - .spec.dnsNames.1
  options:
    delimiter: '.'
    index: 0
    create: true
- source:
  kind: Service
  version: v1
  name: webhook-service
  fieldPath: .metadata.namespace # Namespace of the service
targets:
- select:
    kind: Certificate
    group: cert-manager.io
    version: v1
```

```
    name: serving-cert
  fieldPaths:
    - .spec.dnsNames.0
    - .spec.dnsNames.1
  options:
    delimiter: '.'
    index: 1
    create: true

- source: # Uncomment the following block if you have a ValidatingWebhook (--programmatic-validation)
  kind: Certificate
  group: cert-manager.io
  version: v1
  name: serving-cert # This name should match the one in certificate.yaml
  fieldPath: .metadata.namespace # Namespace of the certificate CR
targets:
- select:
    kind: ValidatingWebhookConfiguration
    fieldPaths:
      - .metadata.annotations.[cert-manager.io/inject-ca-from]
  options:
    delimiter: '/'
    index: 0
    create: true

- source:
  kind: Certificate
  group: cert-manager.io
  version: v1
  name: serving-cert
  fieldPath: .metadata.name
targets:
- select:
    kind: ValidatingWebhookConfiguration
    fieldPaths:
      - .metadata.annotations.[cert-manager.io/inject-ca-from]
  options:
    delimiter: '/'
    index: 1
    create: true

- source: # Uncomment the following block if you have a DefaultingWebhook (--defaulting )
  kind: Certificate
  group: cert-manager.io
  version: v1
  name: serving-cert
  fieldPath: .metadata.namespace # Namespace of the certificate CR
targets:
- select:
    kind: MutatingWebhookConfiguration
    fieldPaths:
      - .metadata.annotations.[cert-manager.io/inject-ca-from]
  options:
```

```

    delimiter: '/'
    index: 0
    create: true
- source:
    kind: Certificate
    group: cert-manager.io
    version: v1
    name: serving-cert
    fieldPath: .metadata.name
targets:
- select:
    kind: MutatingWebhookConfiguration
    fieldPaths:
      - .metadata.annotations.[cert-manager.io/inject-ca-from]
options:
    delimiter: '/'
    index: 1
    create: true
#
# - source: # Uncomment the following block if you have a ConversionWebhook (--conversion)
#   kind: Certificate
#   group: cert-manager.io
#   version: v1
#   name: serving-cert
#   fieldPath: .metadata.namespace # Namespace of the certificate CR
#   targets: # Do not remove or uncomment the following scaffold marker; required to generate code for target CRD.
# +kubebuilder:scaffold:crdkustomizecainjectionns
# - source:
#   kind: Certificate
#   group: cert-manager.io
#   version: v1
#   name: serving-cert
#   fieldPath: .metadata.name
#   targets: # Do not remove or uncomment the following scaffold marker; required to generate code for target CRD.
# +kubebuilder:scaffold:crdkustomizecainjectionname

```

And `config/crd/kustomization.yaml` should now look like the following:

```
# This kustomization.yaml is not intended to be run by itself,
# since it depends on service name and namespace that are out of this kustomize
package.
# It should be run by config/default
resources:
- bases/batch.tutorial.kubebuilder.io_cronjobs.yaml
# +kubebuilder:scaffold:crdkustomizeresource

patches:
# [WEBHOOK] To enable webhook, uncomment all the sections with [WEBHOOK] prefix.
# patches here are for enabling the conversion webhook for each CRD
# +kubebuilder:scaffold:crdkustomizewebhookpatch

# [WEBHOOK] To enable webhook, uncomment the following section
# the following config is for teaching kustomize how to do kustomization for CRDs.
#configurations:
#- kustomizeconfig.yaml
```

Now you can deploy it to your cluster by

```
make deploy IMG=<some-registry>/<project-name>:tag
```

Wait a while till the webhook pod comes up and the certificates are provisioned. It usually completes within 1 minute.

Now you can create a valid CronJob to test your webhooks. The creation should successfully go through.

```
kubectl create -f config/samples/batch_v1_cronjob.yaml
```

You can also try to create an invalid CronJob (e.g. use an ill-formatted schedule field). You should see a creation failure with a validation error.

### The Bootstrapping Problem

If you are deploying a webhook for pods in the same cluster, be careful about the bootstrapping problem, since the creation request of the webhook pod would be sent to the webhook pod itself, which hasn't come up yet.

To make it work, you can either use `namespaceSelector` if your kubernetes version is 1.9+ or use `objectSelector` if your kubernetes version is 1.15+ to skip itself.

# Writing controller tests

Testing Kubernetes controllers is a big subject, and the boilerplate testing files generated for you by kubebuilder are fairly minimal.

To walk you through integration testing patterns for Kubebuilder-generated controllers, we will revisit the CronJob we built in our first tutorial and write a simple test for it.

The basic approach is that, in your generated `suite_test.go` file, you will use envtest to create a local Kubernetes API server, instantiate and run your controllers, and then write additional `*_test.go` files to test it using [Ginkgo](#).

If you want to tinker with how your envtest cluster is configured, see section [Configuring envtest for integration tests](#) as well as the [envtest docs](#).

## Test Environment Setup

```
$ vim ../../cronjob-tutorial/testdata/project/internal/controller/suite_test.go
// Apache License (hidden)
// Imports (hidden)
```

Now, let's go through the code generated.

```
var (
    ctx      context.Context
    cancel   context.CancelFunc
    testEnv  *envtest.Environment
    cfg      *rest.Config
    k8sClient client.Client // You'll be using this client in your tests.
)

func TestControllers(t *testing.T) {
    RegisterFailHandler(Fail)

    RunSpecs(t, "Controller Suite")
}

var _ = BeforeSuite(func() {
    logf.SetLogger(zap.New(zap.WriteTo(GinkgoWriter), zap.UseDevMode(true)))

    ctx, cancel = context.WithCancel(context.TODO())

    var err error
```

The CronJob Kind is added to the runtime scheme used by the test environment. This ensures that the CronJob API is registered with the scheme, allowing the test controller to recognize and interact with CronJob resources.

```
err = batchv1.AddToScheme(scheme.Scheme)
Expect(err).NotTo(HaveOccurred())
```

After the schemas, you will see the following marker. This marker is what allows new schemas to be added here automatically when a new API is added to the project.

```
// +kubebuilder:scaffold:scheme
```

The envtest environment is configured to load Custom Resource Definitions (CRDs) from the specified directory. This setup enables the test environment to recognize and interact with the custom resources defined by these CRDs.

```
By("bootstrapping test environment")
testEnv = &envtest.Environment{
    CRDDirectoryPaths: []string{filepath.Join("../", "..", "config", "crd",
"bases")},
    ErrorIfCRDPathMissing: true,
}

// Retrieve the first found binary directory to allow running tests from IDEs
if getFirstFoundEnvTestBinaryDir() != "" {
    testEnv.BinaryAssetsDirectory = getFirstFoundEnvTestBinaryDir()
}
```

Then, we start the envtest cluster.

```
// cfg is defined in this file globally.
cfg, err = testEnv.Start()
Expect(err).NotTo(HaveOccurred())
Expect(cfg).NotTo(BeNil())
```

A client is created for our test CRUD operations.

```
k8sClient, err = client.New(cfg, client.Options{Scheme: scheme.Scheme})
Expect(err).NotTo(HaveOccurred())
Expect(k8sClient).NotTo(BeNil())
```

One thing that this autogenerated file is missing, however, is a way to actually start your controller. The code above will set up a client for interacting with your custom Kind, but will not be able to test your controller behavior. If you want to test your custom controller logic, you'll

need to add some familiar-looking manager logic to your `BeforeSuite()` function, so you can register your custom controller to run on this test cluster.

You may notice that the code below runs your controller with nearly identical logic to your CronJob project's `main.go`! The only difference is that the manager is started in a separate goroutine so it does not block the cleanup of `envtest` when you're done running your tests.

Note that we set up both a "live" k8s client and a separate client from the manager. This is because when making assertions in tests, you generally want to assert against the live state of the API server. If you use the client from the manager (`k8sManager.GetClient`), you'd end up asserting against the contents of the cache instead, which is slower and can introduce flakiness into your tests. We could use the manager's `APIReader` to accomplish the same thing, but that would leave us with two clients in our test assertions and setup (one for reading, one for writing), and it'd be easy to make mistakes.

Note that we keep the reconciler running against the manager's cache client, though – we want our controller to behave as it would in production, and we use features of the cache (like indices) in our controller which aren't available when talking directly to the API server.

```
k8sManager, err := ctrl.NewManager(cfg, ctrl.Options{
    Scheme: scheme.Scheme,
})
Expect(err).ToNot(HaveOccurred())

err = (&CronJobReconciler{
    Client: k8sManager.GetClient(),
    Scheme: k8sManager.GetScheme(),
}).SetupWithManager(k8sManager)
Expect(err).ToNot(HaveOccurred())

go func() {
    defer GinkgoRecover()
    err = k8sManager.Start(ctx)
    Expect(err).ToNot(HaveOccurred(), "failed to run manager")
}()
})
```

Kubebuilder also generates boilerplate functions for cleaning up `envtest` and actually running your test files in your controllers/ directory. You won't need to touch these.

```
var _ = AfterSuite(func() {
    By("tearing down the test environment")
    cancel()
    err := testEnv.Stop()
    Expect(err).NotTo(HaveOccurred())
})
```

Now that you have your controller running on a test cluster and a client ready to perform operations on your CronJob, we can start writing integration tests!

```
// getFirstFoundEnvTestBinaryDir locates the first binary in the specified path.
// ENVTEST-based tests depend on specific binaries, usually located in paths set
by
// controller-runtime. When running tests directly (e.g., via an IDE) without
using
// Makefile targets, the 'BinaryAssetsDirectory' must be explicitly configured.
//
// This function streamlines the process by finding the required binaries, similar
to
// setting the 'KUBEBUILDER_ASSETS' environment variable. To ensure the binaries
are
// properly set up, run 'make setup-envtest' beforehand.
func getFirstFoundEnvTestBinaryDir() string {
    basePath := filepath.Join("../", "..", "bin", "k8s")
    entries, err := os.ReadDir(basePath)
    if err != nil {
        logf.Log.Error(err, "Failed to read directory", "path", basePath)
        return ""
    }
    for _, entry := range entries {
        if entry.IsDir() {
            return filepath.Join(basePath, entry.Name())
        }
    }
    return ""
}
```

## Testing your Controller's Behavior

```
$ vim ../../cronjob-tutorial/testdata/project/internal/controller/cronjob_controller_test.go
// Apache License (hidden)
```

Ideally, we should have one `<kind>_controller_test.go` for each controller scaffolded and called in the `suite_test.go`. So, let's write our example test for the CronJob controller (`cronjob_controller_test.go`.)

```
// Imports (hidden)
```

The first step to writing a simple integration test is to actually create an instance of CronJob you can run tests against. Note that to create a CronJob, you'll need to create a stub CronJob struct that contains your CronJob's specifications.

Note that when we create a stub CronJob, the CronJob also needs stubs of its required downstream objects. Without the stubbed Job template spec and the Pod template spec below, the Kubernetes API will not be able to create the CronJob.

```

var _ = Describe("CronJob controller", func() {
    // Define utility constants for object names and testing timeouts/durations
    // and intervals.
    const (
        CronjobName      = "test-cronjob"
        CronjobNamespace = "default"
        JobName         = "test-job"

        timeout   = time.Second * 10
        duration = time.Second * 10
        interval = time.Millisecond * 250
    )

    Context("When updating CronJob Status", func() {
        It("Should increase CronJob Status.Active count when new Jobs are
        created", func() {
            By("By creating a new CronJob")
            ctx := context.Background()
            cronJob := &cronjobv1.CronJob{
                TypeMeta: metav1.TypeMeta{
                    APIVersion: "batch.tutorial.kubebuilder.io/v1",
                    Kind:       "CronJob",
                },
                ObjectMeta: metav1.ObjectMeta{
                    Name:     CronjobName,
                    Namespace: CronjobNamespace,
                },
                Spec: cronjobv1.CronJobSpec{
                    Schedule: "1 * * * *",
                    JobTemplate: batchv1.JobTemplateSpec{
                        Spec: batchv1.JobSpec{
                            // For simplicity, we only fill out the required
                            fields.
                            Template: v1.PodTemplateSpec{
                                Spec: v1.PodSpec{
                                    // For simplicity, we only fill out the
                                    required fields.
                                    Containers: []v1.Container{
                                        {
                                            Name:   "test-container",
                                            Image:  "test-image",
                                        },
                                        },
                                        RestartPolicy: v1.RestartPolicyOnFailure,
                                    },
                                },
                            },
                        },
                    },
                },
            }
            Expect(k8sClient.Create(ctx, cronJob)).To(Succeed())
        })
    })
}

```

After creating this CronJob, let's check that the CronJob's Spec fields match what we passed in. Note that, because the k8s apiserver may not have finished creating a CronJob after our `Create()` call from earlier, we will use Gomega's `Eventually()` testing function instead of `Expect()` to give the apiserver an opportunity to finish creating our CronJob.

`Eventually()` will repeatedly run the function provided as an argument every interval seconds until (a) the assertions done by the passed-in `Gomega` succeed, or (b) the number of attempts \* interval period exceed the provided timeout value.

In the examples below, `timeout` and `interval` are Go Duration values of our choosing.

```
cronjobLookupKey := types.NamespacedName{Name: CronjobName, Namespace: CronjobNamespace}
createdCronjob := &cronjobv1.CronJob{}

    // We'll need to retry getting this newly created CronJob, given that creation may not immediately happen.
    Eventually(func(g Gomega) {
        g.Expect(k8sClient.Get(ctx, cronjobLookupKey,
createdCronjob)).To(Succeed())
    }, timeout, interval).Should(Succeed())
    // Let's make sure our Schedule string value was properly converted/handled.
    Expect(createdCronjob.Spec.Schedule).To(Equal("1 * * * *"))
```

Now that we've created a CronJob in our test cluster, the next step is to write a test that actually tests our CronJob controller's behavior. Let's test the CronJob controller's logic responsible for updating `CronJob.Status.Active` with actively running jobs. We'll verify that when a CronJob has a single active downstream job, its `CronJob.Status.Active` field contains a reference to this job.

First, we should get the test CronJob we created earlier, and verify that it currently does not have any active jobs. We use Gomega's `Consistently()` check here to ensure that the active job count remains 0 over a duration of time.

```
By("By checking the CronJob has zero active Jobs")
Consistently(func(g Gomega) {
    g.Expect(k8sClient.Get(ctx, cronjobLookupKey,
createdCronjob)).To(Succeed())
    g.Expect(createdCronjob.Status.Active).To(BeEmpty())
}, duration, interval).Should(Succeed())
```

Next, we actually create a stubbed Job that will belong to our CronJob, as well as its downstream template specs. We set the Job's status's "Active" count to 2 to simulate the Job running two pods, which means the Job is actively running.

We then take the stubbed Job and set its owner reference to point to our test CronJob. This ensures that the test Job belongs to, and is tracked by, our test CronJob. Once that's done, we create our new Job instance.

```

By("By creating a new Job")
testJob := &batchv1.Job{
    ObjectMeta: metav1.ObjectMeta{
        Name:      JobName,
        Namespace: CronjobNamespace,
    },
    Spec: batchv1.JobSpec{
        Template: v1.PodTemplateSpec{
            Spec: v1.PodSpec{
                // For simplicity, we only fill out the required
fields.
                    Containers: []v1.Container{
                        {
                            Name:  "test-container",
                            Image: "test-image",
                        },
                    },
                    RestartPolicy: v1.RestartPolicyOnFailure,
                },
            },
        },
    }
}

// Note that your CronJob's GroupVersionKind is required to set up
this owner reference.
kind := reflect.TypeOf(cronjobv1.CronJob{}).Name()
gvk := cronjobv1.GroupVersion.WithKind(kind)

controllerRef := metav1.NewControllerRef(createdCronjob, gvk)
testJob.SetOwnerReferences([]metav1.OwnerReference{*controllerRef})
Expect(k8sClient.Create(ctx, testJob)).To(Succeed())
// Note that you can not manage the status values while creating the
resource.
// The status field is managed separately to reflect the current state
of the resource.
// Therefore, it should be updated using a PATCH or PUT operation
after the resource has been created.
// Additionally, it is recommended to use StatusConditions to manage
the status. For further information see:
//
https://github.com/kubernetes/community/blob/master/contributors/devel/sig-architecture/api-conventions.md#spec-and-status
testJob.Status.Active = 2
Expect(k8sClient.Status().Update(ctx, testJob)).To(Succeed())

```

Adding this Job to our test CronJob should trigger our controller's reconciler logic. After that, we can write a test that evaluates whether our controller eventually updates our CronJob's Status field as expected!

```

        By("By checking that the CronJob has one active Job")
        Eventually(func(g *gomock.Callable) {
            g.Expect(k8sClient.Get(ctx, cronjobLookupKey,
createdCronjob)).To(Succeed(), "should GET the CronJob")
                g.Expect(createdCronjob.Status.Active).To(HaveLen(1), "should have
exactly one active job")
                    g.Expect(createdCronjob.Status.Active[0].Name).To(Equal(JobName),
"the wrong job is active")
                }, timeout, interval).Should(Succeed(), "should list our active job %s
in the active jobs list in status", JobName)
            })
        })
    }
)

```

After writing all this code, you can run `go test ./...` in your `controllers/` directory again to run your new test!

This Status update example above demonstrates a general testing strategy for a custom Kind with downstream objects. By this point, you hopefully have learned the following methods for testing your controller behavior:

- Setting up your controller to run on an envtest cluster
- Writing stubs for creating test objects
- Isolating changes to an object to test specific controller behavior

## Examples

You can use the plugin [DeployImage](#) to check examples. This plugin allows users to scaffold API/Controllers to deploy and manage an Operand (image) on the cluster following the guidelines and best practices. It abstracts the complexities of achieving this goal while allowing users to customize the generated code.

Therefore, you can check that a test using ENV TEST will be generated for the controller which has the purpose to ensure that the Deployment is created successfully. You can see an example of its code implementation under the `testdata` directory with the [DeployImage](#) samples [here](#).

# Epilogue

By this point, we've got a pretty full-featured implementation of the CronJob controller, made use of most of the features of Kubebuilder, and written tests for the controller using envtest.

If you want more, head over to the [Multi-Version Tutorial](#) to learn how to add new API versions to a project.

Additionally, you can try the following steps on your own – we'll have a tutorial section on them Soon™:

- adding [additional printer columns](#) `kubectl get`

# Tutorial: Multi-Version API

Most projects start out with an alpha API that changes release to release. However, eventually, most projects will need to move to a more stable API. Once your API is stable though, you can't make breaking changes to it. That's where API versions come into play.

Let's make some changes to the `CronJob` API spec and make sure all the different versions are supported by our CronJob project.

If you haven't already, make sure you've gone through the base [CronJob Tutorial](#).

## Following Along vs Jumping Ahead

Note that most of this tutorial is generated from literate Go files that form a runnable project, and live in the book source directory: [docs/book/src/multiversion-tutorial/testdata/project](#).

Next, let's figure out what changes we want to make...

# Changing things up

A fairly common change in a Kubernetes API is to take some data that used to be unstructured or stored in some special string format, and change it to structured data. Our `schedule` field fits the bill quite nicely for this – right now, in `v1`, our schedules look like

```
schedule: "*/1 * * * *"
```

That's a pretty textbook example of a special string format (it's also pretty unreadable unless you're a Unix sysadmin).

Let's make it a bit more structured. According to our [CronJob code](#), we support “standard” Cron format.

In Kubernetes, **all versions must be safely round-tripable through each other**. This means that if we convert from version 1 to version 2, and then back to version 1, we must not lose information. Thus, any change we make to our API must be compatible with whatever we supported in v1, and also need to make sure anything we add in v2 is supported in v1. In some cases, this means we need to add new fields to v1, but in our case, we won't have to, since we're not adding new functionality.

Keeping all that in mind, let's convert our example above to be slightly more structured:

```
schedule:  
  minute: */1
```

Now, at least, we've got labels for each of our fields, but we can still easily support all the different syntax for each field.

We'll need a new API version for this change. Let's call it v2:

```
kubebuilder create api --group batch --version v2 --kind CronJob
```

Press `y` for “Create Resource” and `n` for “Create Controller”.

Now, let's copy over our existing types, and make the change:

```
$ vim project/api/v2/cronjob_types.go  
// Apache License (hidden)
```

Since we're in a v2 package, controller-gen will assume this is for the v2 version automatically. We could override that with the `+versionName` marker.

```
package v2

// Imports (hidden)
```

We'll leave our spec largely unchanged, except to change the schedule field to a new type.

```
// CronJobSpec defines the desired state of CronJob.
type CronJobSpec struct {
    // The schedule in Cron format, see https://en.wikipedia.org/wiki/Cron.
    Schedule CronSchedule `json:"schedule"`

    // The rest of Spec (hidden)
}
```

Next, we'll need to define a type to hold our schedule. Based on our proposed YAML above, it'll have a field for each corresponding Cron "field".

```
// describes a Cron schedule.
type CronSchedule struct {
    // specifies the minute during which the job executes.
    // +optional
    Minute *CronField `json:"minute,omitempty"`
    // specifies the hour during which the job executes.
    // +optional
    Hour *CronField `json:"hour,omitempty"`
    // specifies the day of the month during which the job executes.
    // +optional
    DayOfMonth *CronField `json:"dayOfMonth,omitempty"`
    // specifies the month during which the job executes.
    // +optional
    Month *CronField `json:"month,omitempty"`
    // specifies the day of the week during which the job executes.
    // +optional
    DayOfWeek *CronField `json:"dayOfWeek,omitempty"`
}
```

Finally, we'll define a wrapper type to represent a field. We could attach additional validation to this field, but for now we'll just use it for documentation purposes.

```
// represents a Cron field specifier.
type CronField string

// Other Types (hidden)
```

# Storage Versions

```
$ vim project/api/v1/cronjob_types.go  
// Apache License (hidden)  
  
package v1  
  
// Imports (hidden)  
// old stuff (hidden)
```

Since we'll have more than one version, we'll need to mark a storage version. This is the version that the Kubernetes API server uses to store our data. We'll chose the v1 version for our project.

We'll use the `+kubebuilder:storageversion` to do this.

Note that multiple versions may exist in storage if they were written before the storage version changes – changing the storage version only affects how objects are created/updated after the change.

```
// +kubebuilder:object:root=true  
// +kubebuilder:storageversion  
// +kubebuilder:conversion:hub  
// +kubebuilder:subresource:status  
// +versionName=v1  
// +kubebuilder:storageversion  
// CronJob is the Schema for the cronjobs API.  
type CronJob struct {  
    metav1.TypeMeta `json:",inline"  
    metav1.ObjectMeta `json:"metadata,omitempty"  
    Spec   CronJobSpec   `json:"spec,omitempty"  
    Status CronJobStatus `json:"status,omitempty"  
}  
  
// old stuff (hidden)
```

Now that we've got our types in place, we'll need to set up conversion...

# Hubs, spokes, and other wheel metaphors

Since we now have two different versions, and users can request either version, we'll have to define a way to convert between our version. For CRDs, this is done using a webhook, similar to the defaulting and validating webhooks we [defined in the base tutorial](#). Like before, controller-runtime will help us wire together the nitty-gritty bits, we just have to implement the actual conversion.

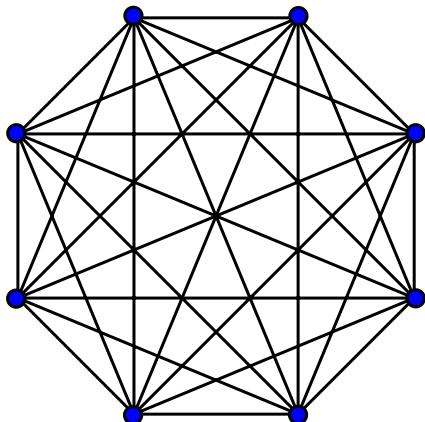
Before we do that, though, we'll need to understand how controller-runtime thinks about versions. Namely:

## Complete graphs are insufficiently nautical

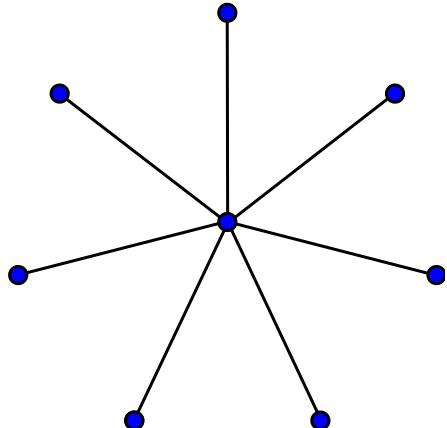
A simple approach to defining conversion might be to define conversion functions to convert between each of our versions. Then, whenever we need to convert, we'd look up the appropriate function, and call it to run the conversion.

This works fine when we just have two versions, but what if we had 4 types? 8 types? That'd be a lot of conversion functions.

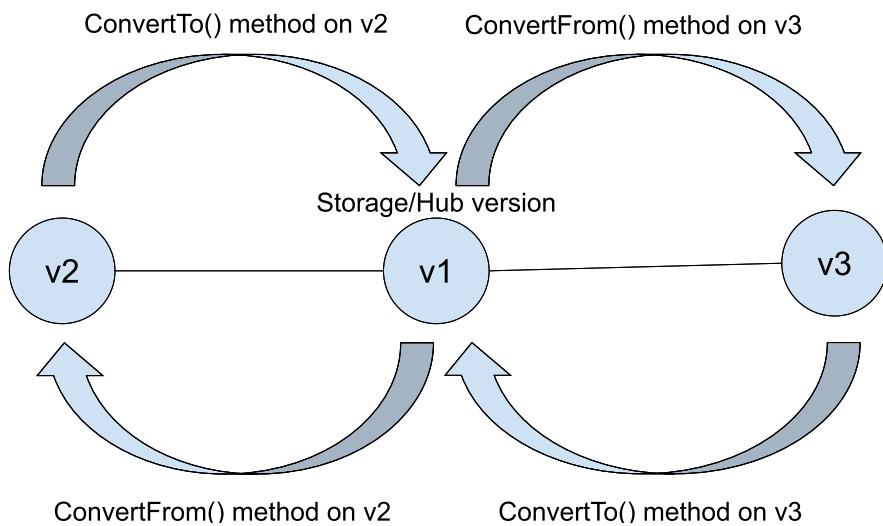
Instead, controller-runtime models conversion in terms of a “hub and spoke” model – we mark one version as the “hub”, and all other versions just define conversion to and from the hub:



**becomes**



Then, if we have to convert between two non-hub versions, we first convert to the hub version, and then to our desired version:



This cuts down on the number of conversion functions that we have to define, and is modeled off of what Kubernetes does internally.

## What does that have to do with Webhooks?

When API clients, like kubectl or your controller, request a particular version of your resource, the Kubernetes API server needs to return a result that's of that version. However, that version might not match the version stored by the API server.

In that case, the API server needs to know how to convert between the desired version and the stored version. Since the conversions aren't built in for CRDs, the Kubernetes API server calls out to a webhook to do the conversion instead. For Kubebuilder, this webhook is implemented by controller-runtime, and performs the hub-and-spoke conversions that we discussed above.

Now that we have the model for conversion down pat, we can actually implement our conversions.

# Implementing conversion

With our model for conversion in place, it's time to actually implement the conversion functions. We'll create a conversion webhook for our CronJob API version `v1` (Hub) to Spoke our CronJob API version `v2` see:

```
kubebuilder create webhook --group batch --version v1 --kind CronJob --conversion  
--spoke v2
```

The above command will generate the `cronjob_conversion.go` next to our `cronjob_types.go` file, to avoid cluttering up our main types file with extra functions.

## Hub...

First, we'll implement the hub. We'll choose the `v1` version as the hub:

```
$ vim project/api/v1/cronjob_conversion.go  
// Apache License (hidden)  
  
package v1
```

Implementing the hub method is pretty easy – we just have to add an empty method called `Hub()` to serve as a [marker](#). We could also just put this inline in our `cronjob_types.go` file.

```
// Hub marks this type as a conversion hub.  
func (*CronJob) Hub() {}
```

## ... and Spokes

Then, we'll implement our spoke, the `v2` version:

```
$ vim project/api/v2/cronjob_conversion.go  
// Apache License (hidden)  
  
package v2  
  
// Imports (hidden)
```

Our “spoke” versions need to implement the `Convertible` interface. Namely, they’ll need `ConvertTo()` and `ConvertFrom()` methods to convert to/from the hub version.

`ConvertTo` is expected to modify its argument to contain the converted object. Most of the conversion is straightforward copying, except for converting our changed field.

```
// ConvertTo converts this CronJob (v2) to the Hub version (v1).
func (src *CronJob) ConvertTo(dstRaw conversion.Hub) error {
    dst := dstRaw.(*batchv1.CronJob)
    log.Printf("ConvertTo: Converting CronJob from Spoke version v2 to Hub version
v1;"+
        "source: %s/%s, target: %s/%s", src.Namespace, src.Name, dst.Namespace,
dst.Name)

    sched := src.Spec.Schedule
    scheduleParts := []string{"*", "*", "*", "*", "*"}
    if sched.Minute != nil {
        scheduleParts[0] = string(*sched.Minute)
    }
    if sched.Hour != nil {
        scheduleParts[1] = string(*sched.Hour)
    }
    if sched.DayOfMonth != nil {
        scheduleParts[2] = string(*sched.DayOfMonth)
    }
    if sched.Month != nil {
        scheduleParts[3] = string(*sched.Month)
    }
    if sched.DayOfWeek != nil {
        scheduleParts[4] = string(*sched.DayOfWeek)
    }
    dst.Spec.Schedule = strings.Join(scheduleParts, " ")
}

// note conversion (hidden)

return nil
}
```

`ConvertFrom` is expected to modify its receiver to contain the converted object. Most of the conversion is straightforward copying, except for converting our changed field.

```

// ConvertFrom converts the Hub version (v1) to this CronJob (v2).
func (dst *CronJob) ConvertFrom(srcRaw conversion.Hub) error {
    src := srcRaw.(*batchv1.CronJob)
    log.Printf("ConvertFrom: Converting CronJob from Hub version v1 to Spoke
version v2;"+
        "source: %s/%s, target: %s/%s", src.Namespace, src.Name, dst.Namespace,
dst.Name)

    schedParts := strings.Split(src.Spec.Schedule, " ")
    if len(schedParts) != 5 {
        return fmt.Errorf("invalid schedule: not a standard 5-field schedule")
    }
    partIfNeeded := func(raw string) *CronField {
        if raw == "*" {
            return nil
        }
        part := CronField(raw)
        return &part
    }
    dst.Spec.Schedule.Minute = partIfNeeded(schedParts[0])
    dst.Spec.Schedule.Hour = partIfNeeded(schedParts[1])
    dst.Spec.Schedule.DayOfMonth = partIfNeeded(schedParts[2])
    dst.Spec.Schedule.Month = partIfNeeded(schedParts[3])
    dst.Spec.Schedule.DayOfWeek = partIfNeeded(schedParts[4])

// note conversion (hidden)

    return nil
}

```

Now that we've got our conversions in place, all that we need to do is wire up our main to serve the webhook!

# Setting up the webhooks

Our conversion is in place, so all that's left is to tell controller-runtime about our conversion.

## Webhook setup...

```
$ vim project/internal/webhook/v1/cronjob_webhook.go  
// Apache License (hidden)  
// Go imports (hidden)
```

Next, we'll setup a logger for the webhooks.

```
var cronjoblog = logf.Log.WithName("cronjob-resource")
```

This setup doubles as setup for our conversion webhooks: as long as our types implement the [Hub](#) and [Convertible](#) interfaces, a conversion webhook will be registered.

```
// SetupCronJobWebhookWithManager registers the webhook for CronJob in the  
manager.  
func SetupCronJobWebhookWithManager(mgr ctrl.Manager) error {  
    return ctrl.NewWebhookManagedBy(mgr).For(&batchv1.CronJob{}).  
        WithValidator(&CronJobCustomValidator{}).  
        WithDefaulter(&CronJobCustomDefaulter{  
            DefaultConcurrencyPolicy:           batchv1.AllowConcurrent,  
            DefaultSuspend:                   false,  
            DefaultSuccessfulJobsHistoryLimit: 3,  
            DefaultFailedJobsHistoryLimit:     1,  
        }).  
        Complete()  
}
```

Notice that we use kubebuilder markers to generate webhook manifests. This marker is responsible for generating a mutating webhook manifest.

The meaning of each marker can be found [here](#).

This marker is responsible for generating a mutation webhook manifest.

```

// +kubebuilder:webhook:path=/mutate-batch-tutorial-kubebuilder-io-v1-
cronjob,mutating=true,failurePolicy=fail,sideEffects=None,groups=batch.tutorial.ku
bebuilder.io,resources=cronjobs,verbs=create;update,versions=v1,name=mcronjob-
v1.kb.io,admissionReviewVersions=v1

// CronJobCustomDefaulter struct is responsible for setting default values on the
custom resource of the
// Kind CronJob when those are created or updated.
//
// NOTE: The +kubebuilder:object:generate=false marker prevents controller-gen
from generating DeepCopy methods,
// as it is used only for temporary operations and does not need to be deeply
copied.
type CronJobCustomDefaulter struct {

    // Default values for various CronJob fields
    DefaultConcurrencyPolicy      batchv1ConcurrencyPolicy
    DefaultSuspend                 bool
    DefaultSuccessfulJobsHistoryLimit int32
    DefaultFailedJobsHistoryLimit   int32
}

var _ webhook.CustomDefaulter = &CronJobCustomDefaulter{}

```

We use the `webhook.CustomDefaulter` interface to set defaults to our CRD. A webhook will automatically be served that calls this defaulting.

The `Default` method is expected to mutate the receiver, setting the defaults.

```

// Default implements webhook.CustomDefaulter so a webhook will be registered for
// the Kind CronJob.
func (d *CronJobCustomDefaulter) Default(ctx context.Context, obj runtime.Object)
error {
    cronjob, ok := obj.(*batchv1.CronJob)

    if !ok {
        return fmt.Errorf("expected an CronJob object but got %T", obj)
    }
    cronjoblog.Info("Defaulting for CronJob", "name", cronjob.GetName())

    // Set default values
    d.applyDefaults(cronjob)
    return nil
}

// applyDefaults applies default values to CronJob fields.
func (d *CronJobCustomDefaulter) applyDefaults(cronJob *batchv1.CronJob) {
    if cronJob.Spec.ConcurrencyPolicy == "" {
        cronJob.Spec.ConcurrencyPolicy = d.DefaultConcurrencyPolicy
    }
    if cronJob.Spec.Suspend == nil {
        cronJob.Spec.Suspend = new(bool)
        *cronJob.Spec.Suspend = d.DefaultSuspend
    }
    if cronJob.Spec.SuccessfulJobsHistoryLimit == nil {
        cronJob.Spec.SuccessfulJobsHistoryLimit = new(int32)
        *cronJob.Spec.SuccessfulJobsHistoryLimit =
d.DefaultSuccessfulJobsHistoryLimit
    }
    if cronJob.Spec.FailedJobsHistoryLimit == nil {
        cronJob.Spec.FailedJobsHistoryLimit = new(int32)
        *cronJob.Spec.FailedJobsHistoryLimit = d.DefaultFailedJobsHistoryLimit
    }
}

```

We can validate our CRD beyond what's possible with declarative validation. Generally, declarative validation should be sufficient, but sometimes more advanced use cases call for complex validation.

For instance, we'll see below that we use this to validate a well-formed cron schedule without making up a long regular expression.

If `webhook.CustomValidator` interface is implemented, a webhook will automatically be served that calls the validation.

The `ValidateCreate`, `ValidateUpdate` and `ValidateDelete` methods are expected to validate its receiver upon creation, update and deletion respectively. We separate out `ValidateCreate` from `ValidateUpdate` to allow behavior like making certain fields immutable, so that they can only be set on creation. `ValidateDelete` is also separated from `ValidateUpdate` to allow different

validation behavior on deletion. Here, however, we just use the same shared validation for `ValidateCreate` and `ValidateUpdate`. And we do nothing in `ValidateDelete`, since we don't need to validate anything on deletion.

This marker is responsible for generating a validation webhook manifest.

```
// +kubebuilder:webhook:path=/validate-batch-tutorial-kubebuilder-io-v1-
cronjob,mutating=false,failurePolicy=fail,sideEffects=None,groups=batch.tutorial.k
ubebuilder.io,resources=cronjobs,verbs=create;update,versions=v1,name=vcronjob-
v1.kb.io,admissionReviewVersions=v1

// CronJobCustomValidator struct is responsible for validating the CronJob
resource
// when it is created, updated, or deleted.
//
// NOTE: The +kubebuilder:object:generate=false marker prevents controller-gen
from generating DeepCopy methods,
// as this struct is used only for temporary operations and does not need to be
deeply copied.
type CronJobCustomValidator struct {
    // TODO(user): Add more fields as needed for validation
}

var _ webhook.CustomValidator = &CronJobCustomValidator{}

// ValidateCreate implements webhook.CustomValidator so a webhook will be
registered for the type CronJob.
func (v *CronJobCustomValidator) ValidateCreate(ctx context.Context, obj
runtime.Object) (admission.Warnings, error) {
    cronjob, ok := obj.(*batchv1.CronJob)
    if !ok {
        return nil, fmt.Errorf("expected a CronJob object but got %T", obj)
    }
    cronjoblog.Info("Validation for CronJob upon creation", "name",
cronjob.GetName())

    return nil, validateCronJob(cronjob)
}

// ValidateUpdate implements webhook.CustomValidator so a webhook will be
registered for the type CronJob.
func (v *CronJobCustomValidator) ValidateUpdate(ctx context.Context, oldObj,
newObj runtime.Object) (admission.Warnings, error) {
    cronjob, ok := newObj.(*batchv1.CronJob)
    if !ok {
        return nil, fmt.Errorf("expected a CronJob object for the newObj but got
%T", newObj)
    }
    cronjoblog.Info("Validation for CronJob upon update", "name",
cronjob.GetName())

    return nil, validateCronJob(cronjob)
}

// ValidateDelete implements webhook.CustomValidator so a webhook will be
registered for the type CronJob.
func (v *CronJobCustomValidator) ValidateDelete(ctx context.Context, obj
runtime.Object) (admission.Warnings, error) {
    cronjob, ok := obj.(*batchv1.CronJob)
```

```

if !ok {
    return nil, fmt.Errorf("expected a CronJob object but got %T", obj)
}
cronjoblog.Info("Validation for CronJob upon deletion", "name",
cronjob.GetName())

// TODO(user): fill in your validation logic upon object deletion.

return nil, nil
}

```

We validate the name and the spec of the CronJob.

```

// validateCronJob validates the fields of a CronJob object.
func validateCronJob(cronjob *batchv1.CronJob) error {
    var allErrs field.ErrorList
    if err := validateCronJobName(cronjob); err != nil {
        allErrs = append(allErrs, err)
    }
    if err := validateCronJobSpec(cronjob); err != nil {
        allErrs = append(allErrs, err)
    }
    if len(allErrs) == 0 {
        return nil
    }

    return apierrors.NewInvalid(
        schema.GroupKind{Group: "batch.tutorial.kubebuilder.io", Kind: "CronJob"},
        cronjob.Name, allErrs)
}

```

Some fields are declaratively validated by OpenAPI schema. You can find kubebuilder validation markers (prefixed with `// +kubebuilder:validation`) in the [Designing an API](#) section. You can find all of the kubebuilder supported markers for declaring validation by running `controller-gen crd -w`, or [here](#).

```

func validateCronJobSpec(cronjob *batchv1.CronJob) *field.Error {
    // The field helpers from the kubernetes API machinery help us return nicely
    // structured validation errors.
    return validateScheduleFormat(
        cronjob.Spec.Schedule,
        field.NewPath("spec").Child("schedule"))
}

```

We'll need to validate the `cron` schedule is well-formatted.

```
func validateScheduleFormat(schedule string, fldPath *field.Path) *field.Error {
    if _, err := cron.ParseStandard(schedule); err != nil {
        return field.Invalid(fldPath, schedule, err.Error())
    }
    return nil
}

// Validate object name (hidden)
```

## ...and main.go

Similarly, our existing main file is sufficient:

```
$ vim project/cmd/main.go
// Apache License (hidden)
// Imports (hidden)
// existing setup (hidden)

// nolint:gocyclo
func main() {
    // existing setup (hidden)
```

Our existing call to SetupWebhookWithManager registers our conversion webhooks with the manager, too.

```
// nolint:goconst
if os.Getenv("ENABLE_WEBHOOKS") != "false" {
    if err = webhookbatchv1.SetupCronJobWebhookWithManager(mgr); err != nil {
        setupLog.Error(err, "unable to create webhook", "webhook", "CronJob")
        os.Exit(1)
    }
}
// nolint:goconst
if os.Getenv("ENABLE_WEBHOOKS") != "false" {
    if err = webhookbatchv2.SetupCronJobWebhookWithManager(mgr); err != nil {
        setupLog.Error(err, "unable to create webhook", "webhook", "CronJob")
        os.Exit(1)
    }
}
// +kubebuilder:scaffold:builder

// existing setup (hidden)
```

Everything's set up and ready to go! All that's left now is to test out our webhooks.

# Deployment and Testing

Before we can test out our conversion, we'll need to enable them in our CRD:

Kubebuilder generates Kubernetes manifests under the `config` directory with webhook bits disabled. To enable them, we need to:

- Enable `patches/webhook_in_<kind>.yaml` and `patches/cainjection_in_<kind>.yaml` in `config/crd/kustomization.yaml` file.
- Enable `../certmanager` and `../webhook` directories under the `bases` section in `config/default/kustomization.yaml` file.
- Enable all the vars under the `CERTMANAGER` section in `config/default/kustomization.yaml` file.

Additionally, if present in our Makefile, we'll need to set the `CRD_OPTIONS` variable to just `"crd"`, removing the `trivialVersions` option (this ensures that we actually [generate validation for each version](#), instead of telling Kubernetes that they're the same):

```
CRD_OPTIONS ?= "crd"
```

Now we have all our code changes and manifests in place, so let's deploy it to the cluster and test it out.

You'll need [cert-manager](#) installed (version `0.9.0+`) unless you've got some other certificate management solution. The Kubebuilder team has tested the instructions in this tutorial with [0.9.0-alpha.0](#) release.

Once all our ducks are in a row with certificates, we can run `make install deploy` (as normal) to deploy all the bits (CRD, controller-manager deployment) onto the cluster.

## Testing

Once all of the bits are up and running on the cluster with conversion enabled, we can test out our conversion by requesting different versions.

We'll make a v2 version based on our v1 version (put it under `config/samples`)

```
apiVersion: batch.tutorial.kubebuilder.io/v2
kind: CronJob
metadata:
  labels:
    app.kubernetes.io/name: project
    app.kubernetes.io/managed-by: kustomize
  name: cronjob-sample
spec:
  schedule:
    minute: "*/1"
  startingDeadlineSeconds: 60
  concurrencyPolicy: Allow # explicitly specify, but Allow is also default.
  jobTemplate:
    spec:
      template:
        spec:
          containers:
            - name: hello
              image: busybox
              args:
                - /bin/sh
                - -c
                - date; echo Hello from the Kubernetes cluster
      restartPolicy: OnFailure
```

Then, we can create it on the cluster:

```
kubectl apply -f config/samples/batch_v2_cronjob.yaml
```

If we've done everything correctly, it should create successfully, and we should be able to fetch it using both the v2 resource

```
kubectl get cronjobs.v2.batch.tutorial.kubebuilder.io -o yaml
```

```
apiVersion: batch.tutorial.kubebuilder.io/v2
kind: CronJob
metadata:
  labels:
    app.kubernetes.io/name: project
    app.kubernetes.io/managed-by: kustomize
  name: cronjob-sample
spec:
  schedule:
    minute: "*/1"
  startingDeadlineSeconds: 60
  concurrencyPolicy: Allow # explicitly specify, but Allow is also default.
  jobTemplate:
    spec:
      template:
        spec:
          containers:
            - name: hello
              image: busybox
              args:
                - /bin/sh
                - -c
                - date; echo Hello from the Kubernetes cluster
        restartPolicy: OnFailure
```

and the v1 resource

```
kubectl get cronjobs.v1.batch.tutorial.kubebuilder.io -o yaml
```

```
apiVersion: batch.tutorial.kubebuilder.io/v1
kind: CronJob
metadata:
  labels:
    app.kubernetes.io/name: project
    app.kubernetes.io/managed-by: kustomize
  name: cronjob-sample
spec:
  schedule: "*/1 * * * *"
  startingDeadlineSeconds: 60
  concurrencyPolicy: Allow # explicitly specify, but Allow is also default.
  jobTemplate:
    spec:
      template:
        spec:
          containers:
            - name: hello
              image: busybox
              args:
                - /bin/sh
                - -c
                - date; echo Hello from the Kubernetes cluster
      restartPolicy: OnFailure
```

Both should be filled out, and look equivalent to our v2 and v1 samples, respectively. Notice that each has a different API version.

Finally, if we wait a bit, we should notice that our CronJob continues to reconcile, even though our controller is written against our v1 API version.

## kubectl and Preferred Versions

When we access our API types from Go code, we ask for a specific version by using that version's Go type (e.g. `batchv2.CronJob`).

You might've noticed that the above invocations of `kubectl` looked a little different from what we usually do – namely, they specify a *group-version-resource*, instead of just a resource.

When we write `kubectl get cronjob`, `kubectl` needs to figure out which group-version-resource that maps to. To do this, it uses the *discovery API* to figure out the preferred version of the `cronjob` resource. For CRDs, this is more-or-less the latest stable version (see the [CRD docs](#) for specific details).

With our updates to CronJob, this means that `kubectl get cronjob` fetches the `batch/v2` group-version.

If we want to specify an exact version, we can use `kubectl get resource.version.group`, as we do above.

**You should always use fully-qualified group-version-resource syntax in scripts.** `kubectl get resource` is for humans, self-aware robots, and other sentient beings that can figure out new versions. `kubectl get resource.version.group` is for everything else.

## Troubleshooting

[steps for troubleshooting](#)

# Migrations

Migrating between project structures in Kubebuilder generally involves a bit of manual work.

This section details what's required to migrate, between different versions of Kubebuilder scaffolding, as well as to more complex project layout structures.

# Migration guides from Legacy versions < 3.0.0

Follow the migration guides from the legacy Kubebuilder versions up the required latest v3x version. Note that from v3, a new ecosystem using plugins is introduced for better maintainability, reusability and user experience .

For more info, see the design docs of:

- [Extensible CLI and Scaffolding Plugins: phase 1](#)
- [Extensible CLI and Scaffolding Plugins: phase 1.5](#)
- [Extensible CLI and Scaffolding Plugins - Phase 2](#)

Also, you can check the [Plugins](#) section.

# Kubebuilder v1 vs v2 (Legacy v1.0.0+ to v2.0.0 Kubebuilder CLI versions)

This document cover all breaking changes when migrating from v1 to v2.

The details of all changes (breaking or otherwise) can be found in [controller-runtime](#), [controller-tools](#) and [kubebuilder](#) release notes.

## Common changes

V2 project uses go modules. But kubebuilder will continue to support `dep` until go 1.13 is out.

## controller-runtime

- `Client.List` now uses functional options (`List(ctx, list, ...option)`) instead of `List(ctx, ListOptions, list)`.
- `Client.DeleteAllOf` was added to the `Client` interface.
- Metrics are on by default now.
- A number of packages under `pkg/runtime` have been moved, with their old locations deprecated. The old locations will be removed before controller-runtime v1.0.0. See the [godocs](#) for more information.

## Webhook-related

- Automatic certificate generation for webhooks has been removed, and webhooks will no longer self-register. Use controller-tools to generate a webhook configuration. If you need certificate generation, we recommend using [cert-manager](#). Kubebuilder v2 will scaffold out cert manager configs for you to use – see the [Webhook Tutorial](#) for more details.
- The `builder` package now has separate builders for controllers and webhooks, which facilitates choosing which to run.

## controller-tools

The generator framework has been rewritten in v2. It still works the same as before in many cases, but be aware that there are some breaking changes. Please check [marker documentation](#) for more details.

## Kubebuilder

- Kubebuilder v2 introduces a simplified project layout. You can find the design doc [here](#).
- In v1, the manager is deployed as a `StatefulSet`, while it's deployed as a `Deployment` in v2.
- The `kubebuilder create webhook` command was added to scaffold mutating/validating/conversion webhooks. It replaces the `kubebuilder alpha webhook` command.
- v2 uses `distroless/static` instead of Ubuntu as base image. This reduces image size and attack surface.
- v2 requires kustomize v3.1.0+.

# Migration from v1 to v2

Make sure you understand the [differences between Kubebuilder v1 and v2](#) before continuing

Please ensure you have followed the [installation guide](#) to install the required components.

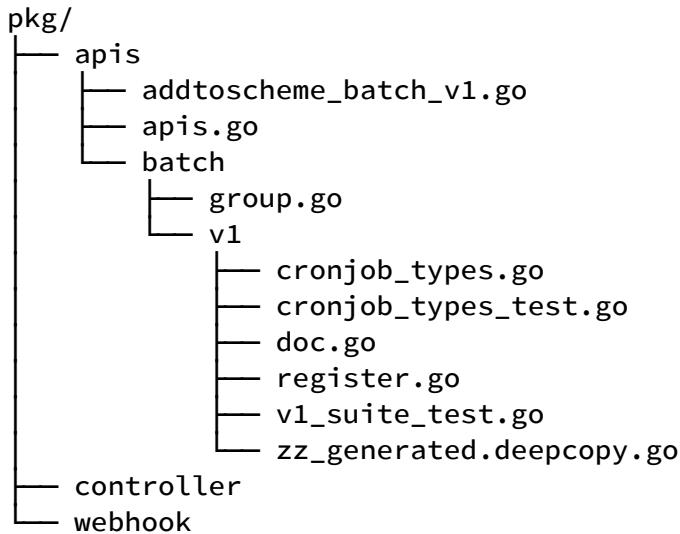
The recommended way to migrate a v1 project is to create a new v2 project and copy over the API and the reconciliation code. The conversion will end up with a project that looks like a native v2 project. However, in some cases, it's possible to do an in-place upgrade (i.e. reuse the v1 project layout, upgrading controller-runtime and controller-tools).

Let's take as example an V1 project and migrate it to Kubebuilder v2. At the end, we should have something that looks like the [example v2 project](#).

## Preparation

We'll need to figure out what the group, version, kind and domain are.

Let's take a look at our current v1 project structure:



All of our API information is stored in `pkg/apis/batch`, so we can look there to find what we need to know.

In `cronjob_types.go`, we can find

```
type CronJob struct {...}
```

In `register.go`, we can find

```
SchemeGroupVersion = schema.GroupVersion{Group: "batch.tutorial.kubebuilder.io",  
Version: "v1"}
```

Putting that together, we get `CronJob` as the kind, and `batch.tutorial.kubebuilder.io/v1` as the group-version

## Initialize a v2 Project

Now, we need to initialize a v2 project. Before we do that, though, we'll need to initialize a new go module if we're not on the `gopath`:

```
go mod init tutorial.kubebuilder.io/project
```

Then, we can finish initializing the project with `kubebuilder`:

```
kubebuilder init --domain tutorial.kubebuilder.io
```

## Migrate APIs and Controllers

Next, we'll re-scaffold out the API types and controllers. Since we want both, we'll say yes to both the API and controller prompts when asked what parts we want to scaffold:

```
kubebuilder create api --group batch --version v1 --kind CronJob
```

If you're using multiple groups, some manual work is required to migrate. Please follow [this](#) for more details.

## Migrate the APIs

Now, let's copy the API definition from `pkg/apis/batch/v1/cronjob_types.go` to `api/v1/cronjob_types.go`. We only need to copy the implementation of the `Spec` and `Status` fields.

We can replace the `+k8s:deepcopy-gen:interfaces=...` marker (which is [deprecated in kubebuilder](#)) with `+kubebuilder:object:root=true`.

We don't need the following markers any more (they're not used anymore, and are relics from much older versions of Kubebuilder):

```
// +genclient  
// +k8s:openapi-gen=true
```

Our API types should look like the following:

```
// +kubebuilder:object:root=true  
// +kubebuilder:subresource:status  
// CronJob is the Schema for the cronjobs API  
type CronJob struct {...}  
  
// +kubebuilder:object:root=true  
  
// CronJobList contains a list of CronJob  
type CronJobList struct {...}
```

## Migrate the Controllers

Now, let's migrate the controller reconciler code from

`pkg/controller/cronjob/cronjob_controller.go` to `controllers/cronjob_controller.go`.

We'll need to copy

- the fields from the `ReconcileCronJob` struct to `CronJobReconciler`
- the contents of the `Reconcile` function
- the [rbac related markers](#) to the new file.
- the code under `func add(mgr manager.Manager, r reconcile.Reconciler) error` to `func SetupWithManager`

## Migrate the Webhooks

If you don't have a webhook, you can skip this section.

## Webhooks for Core Types and External CRDs

If you are using webhooks for Kubernetes core types (e.g. Pods), or for an external CRD that is not owned by you, you can refer the [controller-runtime example for builtin types](#) and do

something similar. Kubebuilder doesn't scaffold much for these cases, but you can use the library in controller-runtime.

## Scaffold Webhooks for our CRDs

Now let's scaffold the webhooks for our CRD (CronJob). We'll need to run the following command with the `--defaulting` and `--programmatic-validation` flags (since our test project uses defaulting and validating webhooks):

```
kubebuilder create webhook --group batch --version v1 --kind CronJob --defaulting  
--programmatic-validation
```

Depending on how many CRDs need webhooks, we may need to run the above command multiple times with different Group-Version-Kinds.

Now, we'll need to copy the logic for each webhook. For validating webhooks, we can copy the contents from `func validatingCronJobFn` in `pkg/default_server/cronjob/validating/cronjob_create_handler.go` to `func ValidateCreate` in `api/v1/cronjob_webhook.go` and then the same for `update`.

Similarly, we'll copy from `func mutatingCronJobFn` to `func Default`.

## Webhook Markers

When scaffolding webhooks, Kubebuilder v2 adds the following markers:

```
// These are v2 markers

// This is for the mutating webhook
// +kubebuilder:webhook:path=/mutate-batch-tutorial-kubebuilder-io-v1-
cronjob,mutating=true,failurePolicy=fail,groups=batch.tutorial.kubebuilder.io,reso
urces=cronjobs,verbs=create;update,versions=v1,name=mcronjob.kb.io

...
// This is for the validating webhook
// +kubebuilder:webhook:path=/validate-batch-tutorial-kubebuilder-io-v1-
cronjob,mutating=false,failurePolicy=fail,groups=batch.tutorial.kubebuilder.io,res
ources=cronjobs,verbs=create;update,versions=v1,name=vcronjob.kb.io
```

The default verbs are `verbs=create;update`. We need to ensure `verbs` matches what we need. For example, if we only want to validate creation, then we would change it to `verbs=create`.

We also need to ensure `failure-policy` is still the same.

Markers like the following are no longer needed (since they deal with self-deploying certificate configuration, which was removed in v2):

```
// v1 markers
// +kubebuilder:webhook:port=9876,cert-dir=/tmp/cert
// +kubebuilder:webhook:service=test-system:webhook-service,selector=app:webhook-
server
// +kubebuilder:webhook:secret=test-system:webhook-server-secret
// +kubebuilder:webhook:mutating-webhook-config-name=test-mutating-webhook-cfg
// +kubebuilder:webhook:validating-webhook-config-name=test-validating-webhook-cfg
```

In v1, a single webhook marker may be split into multiple ones in the same paragraph. In v2, each webhook must be represented by a single marker.

## Others

If there are any manual updates in `main.go` in v1, we need to port the changes to the new `main.go`. We'll also need to ensure all of the needed schemes have been registered.

If there are additional manifests added under `config` directory, port them as well.

Change the image name in the Makefile if needed.

## Verification

Finally, we can run `make` and `make docker-build` to ensure things are working fine.

# Kubebuilder v2 vs v3 (Legacy Kubebuilder v2.0.0+ layout to 3.0.0+)

This document covers all breaking changes when migrating from v2 to v3.

The details of all changes (breaking or otherwise) can be found in [controller-runtime](#), [controller-tools](#) and [kb-releases](#) release notes.

## Common changes

v3 projects use Go modules and request Go 1.18+. Dep is no longer supported for dependency management.

## Kubebuilder

- Preliminary support for plugins was added. For more info see the [Extensible CLI and Scaffolding Plugins: phase 1](#), the [Extensible CLI and Scaffolding Plugins: phase 1.5](#) and the [Extensible CLI and Scaffolding Plugins - Phase 2](#) design docs. Also, you can check the [Plugins](#) section.
- The `PROJECT` file now has a new layout. It stores more information about what resources are in use, to better enable plugins to make useful decisions when scaffolding.

Furthermore, the `PROJECT` file itself is now versioned: the `version` field corresponds to the version of the `PROJECT` file itself, while the `layout` field indicates the scaffolding & primary plugin version in use.

- The version of the image `gcr.io/kubebuilder/kube-rbac-proxy`, which is an optional component enabled by default to secure the request made against the manager, was updated from `0.5.0` to `0.11.0` to address security concerns. The details of all changes can be found in [kube-rbac-proxy](#).

## TL;DR of the New go/v3 Plugin

*More details on this can be found at [here](#), but for the highlights, check below*

## Default plugin

Projects scaffolded with Kubebuilder v3 will use the `go.kubebuilder.io/v3` plugin by default.

- Scaffolded/Generated API version changes:
  - Use `apiextensions/v1` for generated CRDs (`apiextensions/v1beta1` was deprecated in Kubernetes 1.16)
  - Use `admissionregistration.k8s.io/v1` for generated webhooks (`admissionregistration.k8s.io/v1beta1` was deprecated in Kubernetes 1.16)
  - Use `cert-manager.io/v1` for the certificate manager when webhooks are used (`cert-manager.io/v1alpha2` was deprecated in Cert-Manager 0.14. More info: [CertManager v1.0 docs](#))
- Code changes:
  - The manager flags `--metrics-addr` and `enable-leader-election` now are named `--metrics-bind-address` and `--leader-elect` to be more aligned with core Kubernetes Components. More info: [#1839](#)
  - Liveness and Readiness probes are now added by default using `healthz.Ping`.
  - A new option to create the projects using ComponentConfig is introduced. For more info see its [enhancement proposal](#) and the [Component config tutorial](#)
  - Manager manifests now use `SecurityContext` to address security concerns. More info: [#1637](#)
- Misc:
  - Support for `controller-tools` v0.9.0 (for `go/v2` it is v0.3.0 and previously it was v0.2.5)
  - Support for `controller-runtime` v0.12.1 (for `go/v2` it is v0.6.4 and previously it was v0.5.0)
  - Support for `kustomize` v3.8.7 (for `go/v2` it is v3.5.4 and previously it was v3.1.0)
  - Required Envtest binaries are automatically downloaded
  - The minimum Go version is now 1.18 (previously it was 1.13).



### Project customizations

After using the CLI to create your project, you are free to customise how you see fit. Bear in mind, that it is not recommended to deviate from the proposed layout unless you know what you are doing.

For example, you should refrain from moving the scaffolded files, doing so will make it difficult in upgrading your project in the future. You may also lose the ability to use some of the CLI features and helpers. For further information on the project layout, see the doc [What's in a basic project?](#)

## Migrating to Kubebuilder v3

So you want to upgrade your scaffolding to use the latest and greatest features then, follow up the following guide which will cover the steps in the most straightforward way to allow you to upgrade your project to get all latest changes and improvements.

### Apple Silicon (M1)

The current scaffold done by the CLI (`go/v3`) uses [kubernetes-sigs/kustomize](#) v3 which does not provide a valid binary for Apple Silicon (`darwin/arm64`). Therefore, you can use the `go/v4` plugin instead which provides support for this platform:

```
kubebuilder init --domain my.domain --repo my.domain/guestbook --  
plugins=go/v4
```

- [Migration Guide v2 to V3 \(Recommended\)](#)

## By updating the files manually

So you want to use the latest version of Kubebuilder CLI without changing your scaffolding then, check the following guide which will describe the manually steps required for you to upgrade only your PROJECT version and starts to use the plugins versions.

This way is more complex, susceptible to errors, and success cannot be assured. Also, by following these steps you will not get the improvements and bug fixes in the default generated project files.

You will check that you can still using the previous layout by using the `go/v2` plugin which will not upgrade the [controller-runtime](#) and [controller-tools](#) to the latest version used with `go/v3` because of its breaking changes. By checking this guide you know also how to manually change the files to use the `go/v3` plugin and its dependencies versions.

- Migrating to Kubebuilder v3 by updating the files manually

# Migration from v2 to v3

Make sure you understand the [differences between Kubebuilder v2 and v3](#) before continuing.

Please ensure you have followed the [installation guide](#) to install the required components.

The recommended way to migrate a v2 project is to create a new v3 project and copy over the API and the reconciliation code. The conversion will end up with a project that looks like a native v3 project. However, in some cases, it's possible to do an in-place upgrade (i.e. reuse the v2 project layout, upgrading [controller-runtime](#) and [controller-tools](#)).

## Initialize a v3 Project

Project name

For the rest of this document, we are going to use `migration-project` as the project name and `tutorial.kubebuilder.io` as the domain. Please, select and use appropriate values for your case.

Create a new directory with the name of your project. Note that this name is used in the scaffolds to create the name of your manager Pod and of the Namespace where the Manager is deployed by default.

```
$ mkdir migration-project-name  
$ cd migration-project-name
```

Now, we need to initialize a v3 project. Before we do that, though, we'll need to initialize a new go module if we're not on the `GOPATH`. While technically this is not needed inside `GOPATH`, it is still recommended.

```
go mod init tutorial.kubebuilder.io/migration-project
```

The module of your project can found in the in the `go.mod` file at the root of your project:

```
module tutorial.kubebuilder.io/migration-project
```

Then, we can finish initializing the project with kubebuilder.

```
kubebuilder init --domain tutorial.kubebuilder.io
```

The domain of your project can be found in the PROJECT file:

```
...
domain: tutorial.kubebuilder.io
...
```

## Migrate APIs and Controllers

Next, we'll re-scaffold out the API types and controllers.

Scaffolding both the API types and controllers

For this example, we are going to consider that we need to scaffold both the API types and the controllers, but remember that this depends on how you scaffolded them in your original project.

```
kubebuilder create api --group batch --version v1 --kind CronJob
```

## Migrate the APIs

If you're using multiple groups

Please run `kubebuilder edit --multigroup=true` to enable multi-group support before migrating the APIs and controllers. Please see [this](#) for more details.

Now, let's copy the API definition from `api/v1/<kind>_types.go` in our old project to the new one.

These files have not been modified by the new plugin, so you should be able to replace your freshly scaffolded files by your old one. There may be some cosmetic changes. So you can choose to only copy the types themselves.

## Migrate the Controllers

Now, let's migrate the controller code from `controllers/cronjob_controller.go` in our old project to the new one. There is a breaking change and there may be some cosmetic changes.

The new `Reconcile` method receives the context as an argument now, instead of having to create it with `context.Background()`. You can copy the rest of the code in your old controller to the scaffolded methods replacing:

```
func (r *CronJobReconciler) Reconcile(req ctrl.Request) (ctrl.Result, error) {
    ctx := context.Background()
    log := r.Log.WithValues("cronjob", req.NamespacedName)
```

With:

```
func (r *CronJobReconciler) Reconcile(ctx context.Context, req ctrl.Request)
(ctrl.Result, error) {
    log := r.Log.WithValues("cronjob", req.NamespacedName)
```

 Controller-runtime version updated has breaking changes

Check [sigs.k8s.io/controller-runtime](https://sigs.k8s.io/controller-runtime) release docs from 0.8.0+ version for breaking changes.

## Migrate the Webhooks

Skip

If you don't have any webhooks, you can skip this section.

Now let's scaffold the webhooks for our CRD (CronJob). We'll need to run the following command with the `--defaulting` and `--programmatic-validation` flags (since our test project uses defaulting and validating webhooks):

```
kubebuilder create webhook --group batch --version v1 --kind CronJob --defaulting
--programmatic-validation
```

Now, let's copy the webhook definition from `api/v1/<kind>_webhook.go` from our old project to the new one.

## **Others**

If there are any manual updates in `main.go` in v2, we need to port the changes to the new `main.go`. We'll also need to ensure all of the needed schemes have been registered.

If there are additional manifests added under config directory, port them as well.

Change the image name in the Makefile if needed.

## **Verification**

Finally, we can run `make` and `make docker-build` to ensure things are working fine.

# Migration from v2 to v3 by updating the files manually

Make sure you understand the [differences between Kubebuilder v2 and v3](#) before continuing.

Please ensure you have followed the [installation guide](#) to install the required components.

The following guide describes the manual steps required to upgrade your config version and start using the plugin-enabled version.

This way is more complex, susceptible to errors, and success cannot be assured. Also, by following these steps you will not get the improvements and bug fixes in the default generated project files.

Usually you will only try to do it manually if you customized your project and deviated too much from the proposed scaffold. Before continuing, ensure that you understand the note about [project customizations](#). Note that you might need to spend more effort to do this process manually than organize your project customizations to follow up the proposed layout and keep your project maintainable and upgradable with less effort in the future.

The recommended upgrade approach is to follow the [Migration Guide v2 to V3](#) instead.

## Migration from project config version “2” to “3”

Migrating between project configuration versions involves additions, removals, and/or changes to fields in your project’s `PROJECT` file, which is created by running the `init` command.

The `PROJECT` file now has a new layout. It stores more information about what resources are in use, to better enable plugins to make useful decisions when scaffolding.

Furthermore, the `PROJECT` file itself is now versioned. The `version` field corresponds to the version of the `PROJECT` file itself, while the `layout` field indicates the scaffolding and the primary plugin version in use.

### Steps to migrate

The following steps describe the manual changes required to bring the project configuration file (`PROJECT`). These change will add the information that Kubebuilder would add when

generating the file. This file can be found in the root directory.

## Add the projectName

The project name is the name of the project directory in lowercase:

```
...
projectName: example
...
```

## Add the layout

The default plugin layout which is equivalent to the previous version is `go.kubebuilder.io/v2`:

```
...
layout:
- go.kubebuilder.io/v2
...
```

## Update the version

The `version` field represents the version of project's layout. Update this to "3":

```
...
version: "3"
...
```

## Add the resource data

The attribute `resources` represents the list of resources scaffolded in your project.

You will need to add the following data for each resource added to the project.

**Add the Kubernetes API version by adding resources[entry].api.crdVersion: v1beta1:**

```
...
resources:
- api:
  ...
    crdVersion: v1beta1
  domain: my.domain
  group: webapp
  kind: Guestbook
...
...
```

**Add the scope used do scaffold the CRDs by adding resources[entry].api.namespaced: true unless they were cluster-scoped:**

```
...
resources:
- api:
  ...
    namespaced: true
  group: webapp
  kind: Guestbook
...
...
```

**If you have a controller scaffolded for the API then, add resources[entry].controller: true:**

```
...
resources:
- api:
  ...
    controller: true
  group: webapp
  kind: Guestbook
...
```

**Add the resource domain such as resources[entry].domain: testproject.org which usually will be the project domain unless the API scaffold is a core type and/or an external type:**

```
...
resources:
- api:
  ...
    domain: testproject.org
  group: webapp
  kind: Guestbook
...
```

## Supportability

Kubebuilder only supports core types and the APIs scaffolded in the project by default unless you manually change the files you will be unable to work with external-types.

For core types, the domain value will be `k8s.io` or empty.

However, for an external-type you might leave this attribute empty. We cannot suggest what would be the best approach in this case until it become officially supported by the tool. For further information check the issue [#1999](#).

Note that you will only need to add the `domain` if your project has a scaffold for a core type API which the `Domain` value is not empty in Kubernetes API group qualified scheme definition. (For example, see [here](#) that for Kinds from the API `apps` it has not a domain when see [here](#) that for Kinds from the API `authentication` its domain is `k8s.io` )

Check the following the list to know the core types supported and its domain:

| Core Type             | Domain   |
|-----------------------|----------|
| admission             | "k8s.io" |
| admissionregistration | "k8s.io" |
| apps                  | empty    |
| auditregistration     | "k8s.io" |
| apiextensions         | "k8s.io" |
| authentication        | "k8s.io" |
| authorization         | "k8s.io" |
| autoscaling           | empty    |
| batch                 | empty    |
| certificates          | "k8s.io" |
| coordination          | "k8s.io" |
| core                  | empty    |
| events                | "k8s.io" |
| extensions            | empty    |
| imagepolicy           | "k8s.io" |
| networking            | "k8s.io" |
| node                  | "k8s.io" |
| metrics               | "k8s.io" |

| Core Type          | Domain   |
|--------------------|----------|
| policy             | empty    |
| rbac.authorization | "k8s.io" |
| scheduling         | "k8s.io" |
| setting            | "k8s.io" |
| storage            | "k8s.io" |

Following an example where a controller was scaffold for the core type Kind Deployment via the command `create api --group apps --version v1 --kind Deployment --controller=true --resource=false --make=false`:

```
- controller: true
group: apps
kind: Deployment
path: k8s.io/api/apps/v1
version: v1
```

**Add the resources[entry].path with the import path for the api:**

#### Path

If you did not scaffold an API but only generate a controller for the API(GKV) informed then, you do not need to add the path. Note, that it usually happens when you add a controller for an external or core type.

Kubebuilder only supports core types and the APIs scaffolded in the project by default unless you manually change the files you will be unable to work with external-types.

The path will always be the import path used in your Go files to use the API.

```
...
resources:
- api:
  ...
  ...
group: webapp
kind: Guestbook
path: example/api/v1
```

If your project is using webhooks then, add `resources[entry].webhooks.[type]: true` for each type generated and then, add `resources[entry].webhookVersion: v1beta1`:

## Webhooks

The valid types are: `defaulting`, `validation` and `conversion`. Use the webhook type used to scaffold the project.

The Kubernetes API version used to do the webhooks scaffolds in Kubebuilder v2 is `v1beta1`. Then, you will add the `webhookVersion: v1beta1` for all cases.

```
resources:  
- api:  
  ...  
  ...  
  group: webapp  
  kind: Guestbook  
  webhooks:  
    defaulting: true  
    validation: true  
    webhookVersion: v1beta1
```

## Check your PROJECT file

Now ensure that your `PROJECT` file has the same information when the manifests are generated via Kubebuilder V3 CLI.

For the QuickStart example, the `PROJECT` file manually updated to use `go.kubebuilder.io/v2` would look like:

```
domain: my.domain  
layout:  
- go.kubebuilder.io/v2  
projectName: example  
repo: example  
resources:  
- api:  
  crdVersion: v1  
  namespaced: true  
  controller: true  
  domain: my.domain  
  group: webapp  
  kind: Guestbook  
  path: example/api/v1  
  version: v1  
version: "3"
```

You can check the differences between the previous layout( `version 2` ) and the current format( `version 3` ) with the `go.kubebuilder.io/v2` by comparing an example scenario which involves more than one API and webhook, see:

### **Example (Project version 2)**

```
domain: testproject.org
repo: sigs.k8s.io/kubebuilder/example
resources:
- group: crew
  kind: Captain
  version: v1
- group: crew
  kind: FirstMate
  version: v1
- group: crew
  kind: Admiral
  version: v1
version: "2"
```

### **Example (Project version 3)**

```
domain: testproject.org
layout:
- go.kubebuilder.io/v2
projectName: example
repo: sigs.k8s.io/kubebuilder/example
resources:
- api:
  crdVersion: v1
  namespaced: true
  controller: true
  domain: testproject.org
  group: crew
  kind: Captain
  path: example/api/v1
  version: v1
  webhooks:
    defaulting: true
    validation: true
    webhookVersion: v1
- api:
  crdVersion: v1
  namespaced: true
  controller: true
  domain: testproject.org
  group: crew
  kind: FirstMate
  path: example/api/v1
  version: v1
  webhooks:
    conversion: true
    webhookVersion: v1
- api:
  crdVersion: v1
  controller: true
  domain: testproject.org
  group: crew
  kind: Admiral
  path: example/api/v1
  plural: admirales
  version: v1
  webhooks:
    defaulting: true
    webhookVersion: v1
version: "3"
```

## Verification

In the steps above, you updated only the `PROJECT` file which represents the project configuration. This configuration is useful only for the CLI tool. It should not affect how your project behaves.

There is no option to verify that you properly updated the configuration file. The best way to ensure the configuration file has the correct `v3+` fields is to initialize a project with the same API(s), controller(s), and webhook(s) in order to compare generated configuration with the manually changed configuration.

If you made mistakes in the above process, you will likely face issues using the CLI.

## Update your project to use go/v3 plugin

Migrating between project [plugins](#) involves additions, removals, and/or changes to files created by any plugin-supported command, e.g. `init` and `create`. A plugin supports one or more project config versions; make sure you upgrade your project's config version to the latest supported by your target plugin version before upgrading plugin versions.

The following steps describe the manual changes required to modify the project's layout enabling your project to use the `go/v3` plugin. These steps will not help you address all the bug fixes of the already generated scaffolds.



### Deprecated APIs

The following steps will not migrate the API versions which are deprecated  
`apiextensions.k8s.io/v1beta1`, `admissionregistration.k8s.io/v1beta1`,  
`cert-manager.io/v1alpha2`.

## Steps to migrate

### Update your plugin version into the PROJECT file

Before updating the `layout`, please ensure you have followed the above steps to upgrade your Project version to `3`. Once you have upgraded the project version, update the `layout` to the new plugin version `go.kubebuilder.io/v3` as follows:

```
domain: my.domain
layout:
- go.kubebuilder.io/v3
...
```

## Upgrade the Go version and its dependencies:

Ensure that your `go.mod` is using Go version `1.15` and the following dependency versions:

```
module example

go 1.18

require (
    github.com/onsi/ginkgo/v2 v2.1.4
    github.com/onsi/gomega v1.19.0
    k8s.io/api v0.24.0
    k8s.io/apimachinery v0.24.0
    k8s.io/client-go v0.24.0
    sigs.k8s.io/controller-runtime v0.12.1
)
```

## Update the golang image

In the Dockerfile, replace:

```
# Build the manager binary
FROM docker.io/golang:1.13 as builder
```

With:

```
# Build the manager binary
FROM docker.io/golang:1.16 as builder
```

## Update your Makefile

### To allow controller-gen to scaffold the new Kubernetes APIs

To allow `controller-gen` and the scaffolding tool to use the new API versions, replace:

```
CRD_OPTIONS ?= "crd:trivialVersions=true"
```

With:

```
CRD_OPTIONS ?= "crd"
```

## To allow automatic downloads

To allow downloading the newer versions of the Kubernetes binaries required by Envtest into the `testbin/` directory of your project instead of the global setup, replace:

```
# Run tests
test: generate fmt vet manifests
  go test ./... -coverprofile cover.out
```

With:

```
# Setting SHELL to bash allows bash commands to be executed by recipes.
# Options are set to exit when a recipe line exits non-zero or a piped command
fails.
SHELL = /usr/bin/env bash -o pipefail
.SHELLFLAGS = -ec

ENVTEST_ASSETS_DIR=$(shell pwd)/testbin
test: manifests generate fmt vet ## Run tests.
  mkdir -p ${ENVTEST_ASSETS_DIR}
  test -f ${ENVTEST_ASSETS_DIR}/setup-envtest.sh || curl -sSLo
${ENVTEST_ASSETS_DIR}/setup-envtest.sh
https://raw.githubusercontent.com/kubernetes-sigs/controller-
runtime/v0.8.3/hack/setup-envtest.sh
  source ${ENVTEST_ASSETS_DIR}/setup-envtest.sh; fetch_envtest_tools
$(ENVTEST_ASSETS_DIR); setup_envtest_env $(ENVTEST_ASSETS_DIR); go test ./... -
coverprofile cover.out
```

### Envtest binaries

The Kubernetes binaries that are required for the Envtest were upgraded from `1.16.4` to `1.22.1`. You can still install them globally by following [these installation instructions](#).

## To upgrade controller-gen and kustomize dependencies versions used

To upgrade the `controller-gen` and `kustomize` version used to generate the manifests replace:

```
# find or download controller-gen
# download controller-gen if necessary
controller-gen:
ifeq (, $(shell which controller-gen))
@{ \
set -e ;\
CONTROLLER_GEN_TMP_DIR=$(mktemp -d) ;\
cd $CONTROLLER_GEN_TMP_DIR ;\
go mod init tmp ;\
go get sigs.k8s.io/controller-tools/cmd/controller-gen@v0.2.5 ;\
rm -rf $CONTROLLER_GEN_TMP_DIR ;\
}
CONTROLLER_GEN=$(GOBIN)/controller-gen
else
CONTROLLER_GEN=$(shell which controller-gen)
endif
```

With:

```

##@ Build Dependencies

## Location to install dependencies to
LOCALBIN ?= $(shell pwd)/bin
$(LOCALBIN):
    mkdir -p $(LOCALBIN)

## Tool Binaries
KUSTOMIZE ?= $(LOCALBIN)/kustomize
CONTROLLER_GEN ?= $(LOCALBIN)/controller-gen
ENVTEST ?= $(LOCALBIN)/setup-envtest

## Tool Versions
KUSTOMIZE_VERSION ?= v3.8.7
CONTROLLER_TOOLS_VERSION ?= v0.9.0

KUSTOMIZE_INSTALL_SCRIPT ?= "https://raw.githubusercontent.com/kubernetes-
sigs/kustomize/master/hack/install_kustomize.sh"
.PHONY: kustomize
kustomize: $(KUSTOMIZE) ## Download kustomize locally if necessary.
$(KUSTOMIZE): $(LOCALBIN)
    test -s $(LOCALBIN)/kustomize || { curl -Ss $(KUSTOMIZE_INSTALL_SCRIPT) | bash
-s -- $(subst v,,,$(KUSTOMIZE_VERSION)) $(LOCALBIN); }

.PHONY: controller-gen
controller-gen: $(CONTROLLER_GEN) ## Download controller-gen locally if necessary.
$(CONTROLLER_GEN): $(LOCALBIN)
    test -s $(LOCALBIN)/controller-gen || GOBIN=$(LOCALBIN) go install
sig.k8s.io/controller-tools/cmd/controller-gen@$$(CONTROLLER_TOOLS_VERSION)

.PHONY: envtest
envtest: $(ENVTEST) ## Download envtest-setup locally if necessary.
$(ENVTEST): $(LOCALBIN)
    test -s $(LOCALBIN)/setup-envtest || GOBIN=$(LOCALBIN) go install
sig.k8s.io/controller-runtime/tools/setup-envtest@latest

```

And then, to make your project use the `kustomize` version defined in the Makefile, replace all usage of `kustomize` with `$(KUSTOMIZE)`

### Makefile

You can check all changes applied to the Makefile by looking in the samples projects generated in the `testdata` directory of the Kubebuilder repository or by just by creating a new project with the Kubebuilder CLI.

## Update your controllers

-   Controller-runtime version updated has breaking changes

Check [sigs.k8s.io/controller-runtime](https://sigs.k8s.io/controller-runtime) release docs from 0.7.0+ version for breaking changes.

Replace:

```
func (r *MyKindReconciler) Reconcile(req ctrl.Request) (ctrl.Result, error) {
    ctx := context.Background()
    log := r.Log.WithValues("cronjob", req.NamespacedName)
```

With:

```
func (r *MyKindReconciler) Reconcile(ctx context.Context, req ctrl.Request)
(ctrl.Result, error) {
    log := r.Log.WithValues("cronjob", req.NamespacedName)
```

## Update your controller and webhook test suite

-   Ginkgo V2 version update has breaking changes

Check [Ginkgo V2 Migration Guide](#) for breaking changes.

Replace:

- "github.com/onsi/ginkgo"

With:

- "github.com/onsi/ginkgo/v2"

Also, adjust your test suite.

For Controller Suite:

```
RunSpecsWithDefaultAndCustomReporters(t,
    "Controller Suite",
    []Reporter{printer.NewlineReporter{}})
```

With:

```
RunSpecs(t, "Controller Suite")
```

For Webhook Suite:

```
RunSpecsWithDefaultAndCustomReporters(t,
    "Webhook Suite",
    []Reporter{printer.NewlineReporter{}})
```

With:

```
RunSpecs(t, "Webhook Suite")
```

Last but not least, remove the timeout variable from the `BeforeSuite` blocks:

Replace:

```
var _ = BeforeSuite(func(done Done) {
    ...
}, 60)
```

With

```
var _ = BeforeSuite(func(done Done) {
    ...
})
```

## Change Logger to use flag options

In the `main.go` file replace:

```
flag.Parse()

ctrl.SetLogger(zap.New(zap.UseDevMode(true)))
```

With:

```

opts := zap.Options{
    Development: true,
}
opts.BindFlags(flag.CommandLine)
flag.Parse()

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

```

## Rename the manager flags

The manager flags `--metrics-addr` and `enable-leader-election` were renamed to `--metrics-bind-address` and `--leader-elect` to be more aligned with core Kubernetes Components. More info: [#1839](#).

In your `main.go` file replace:

```

func main() {
    var metricsAddr string
    var enableLeaderElection bool
    flag.StringVar(&metricsAddr, "metrics-addr", ":8080", "The address the metric endpoint binds to.")
    flag.BoolVar(&enableLeaderElection, "enable-leader-election", false,
        "Enable leader election for controller manager. "+
            "Enabling this will ensure there is only one active controller manager.")
}

```

With:

```

func main() {
    var metricsAddr string
    var enableLeaderElection bool
    flag.StringVar(&metricsAddr, "metrics-bind-address", ":8080", "The address the metric endpoint binds to.")
    flag.BoolVar(&enableLeaderElection, "leader-elect", false,
        "Enable leader election for controller manager. "+
            "Enabling this will ensure there is only one active controller manager.")
}

```

And then, rename the flags in the `config/default/manager_auth_proxy_patch.yaml` and `config/default/manager.yaml`:

```

- name: manager
args:
- "--health-probe-bind-address=:8081"
- "--metrics-bind-address=127.0.0.1:8080"
- "--leader-elect"

```

## Verification

Finally, we can run `make` and `make docker-build` to ensure things are working fine.

# Change your project to remove the Kubernetes deprecated API versions usage

Before continuing

Make sure you understand [Versions in CustomResourceDefinitions](#).

The following steps describe a workflow to upgrade your project to remove the deprecated Kubernetes APIs: `apiextensions.k8s.io/v1beta1`, `admissionregistration.k8s.io/v1beta1`, `cert-manager.io/v1alpha2`.

The Kubebuilder CLI tool does not support scaffolded resources for both Kubernetes API versions such as; an API/CRD with `apiextensions.k8s.io/v1beta1` and another one with `apiextensions.k8s.io/v1`.

Cert Manager API

If you scaffold a webhook using the Kubernetes API `admissionregistration.k8s.io/v1` then, by default, it will use the API `cert-manager.io/v1` in the manifests.

The first step is to update your `PROJECT` file by replacing the `api.crdVersion:v1beta` and `webhooks.WebhookVersion:v1beta` with `api.crdVersion:v1` and `webhooks.WebhookVersion:v1` which would look like:

```
domain: my.domain
layout: go.kubebuilder.io/v3
projectName: example
repo: example
resources:
- api:
  crdVersion: v1
  namespaced: true
  group: webapp
  kind: Guestbook
  version: v1
  webhooks:
    defaulting: true
    webhookVersion: v1
version: "3"
```

You can try to re-create the APIS(CRDs) and Webhooks manifests by using the `--force` flag.

  Before re-create

Note, however, that the tool will re-scaffold the files which means that you will lose their content.

Before executing the commands ensure that you have the files content stored in another place. An easy option is to use `git` to compare your local change with the previous version to recover the contents.

Now, re-create the APIS(CRDs) and Webhooks manifests by running the `kubebuilder create api` and `kubebuilder create webhook` for the same group, kind and versions with the flag `--force`, respectively.

# V3 - Plugins Layout Migration Guides

Following the migration guides from the plugins versions. Note that the plugins ecosystem was introduced with Kubebuilder v3.0.0 release where the go/v3 version is the default layout since [28 Apr 2021](#).

Therefore, you can check here how to migrate the projects built from Kubebuilder 3.x with the plugin go/v3 to the latest.

# go/v3 vs go/v4

This document covers all breaking changes when migrating from projects built using the plugin go/v3 (default for any scaffold done since 28 Apr 2021) to the next version of the Golang plugin go/v4.

The details of all changes (breaking or otherwise) can be found in:

- [controller-runtime](#)
- [controller-tools](#)
- [kustomize](#)
- [kb-releases](#) release notes.

## Common changes

- go/v4 projects use Kustomize v5x (instead of v3x)
- note that some manifests under `config/` directory have been changed in order to no longer use the deprecated Kustomize features such as env vars.
- A `kustomization.yaml` is scaffolded under `config/samples`. This helps simply and flexibly generate sample manifests: `kustomize build config/samples`.
- adds support for Apple Silicon M1 (darwin/arm64)
- remove support to CRD/WebHooks Kubernetes API v1beta1 version which are no longer supported since k8s 1.22
- no longer scaffold webhook test files with "`k8s.io/api/admission/v1beta1`" the k8s API which is no longer served since k8s 1.25. By default webhooks test files are scaffolding using "`k8s.io/api/admission/v1`" which is support from k8s 1.20
- no longer provide backwards compatible support with k8s versions < 1.16
- change the layout to accommodate the community request to follow the [Standard Go Project Layout](#) by moving the api(s) under a new directory called `api`, controller(s) under a new directory called `internal` and the `main.go` under a new directory named `cmd`

TL;DR of the New `go/v4` Plugin

Further details can be found in the [go/v4 plugin section](#)

# TL;DR of the New go/v4 Plugin

*More details on this can be found at [here](#), but for the highlights, check below*



## Project customizations

After using the CLI to create your project, you are free to customize how you see fit. Bear in mind, that it is not recommended to deviate from the proposed layout unless you know what you are doing.

For example, you should refrain from moving the scaffolded files, doing so will make it difficult in upgrading your project in the future. You may also lose the ability to use some of the CLI features and helpers. For further information on the project layout, see the doc [What's in a basic project?](#)

## Migrating to Kubebuilder go/v4

If you want to upgrade your scaffolding to use the latest and greatest features then, follow the guide which will cover the steps in the most straightforward way to allow you to upgrade your project to get all latest changes and improvements.

- [Migration Guide go/v3 to go/v4 \(Recommended\)](#)

## By updating the files manually

If you want to use the latest version of Kubebuilder CLI without changing your scaffolding then, check the following guide which will describe the steps to be performed manually to upgrade only your PROJECT version and start using the plugins versions.

This way is more complex, susceptible to errors, and success cannot be assured. Also, by following these steps you will not get the improvements and bug fixes in the default generated project files.

- [Migrating to go/v4 by updating the files manually](#)

# Migration from go/v3 to go/v4

Make sure you understand the differences between Kubebuilder go/v3 and go/v4 before continuing.

Please ensure you have followed the [installation guide](#) to install the required components.

The recommended way to migrate a `go/v3` project is to create a new `go/v4` project and copy over the API and the reconciliation code. The conversion will end up with a project that looks like a native `go/v4` project layout (latest version).

 Your Upgrade Assistant: The ``alpha generate`` command

To upgrade your project you might want to use the command `kubebuilder alpha generate [OPTIONS]`. This command will re-scaffold the project using the current Kubebuilder version. You can run `kubebuilder alpha generate --plugins=go/v4` to regenerate your project using `go/v4` based in your `PROJECT` file config. ([More info](#))

However, in some cases, it's possible to do an in-place upgrade (i.e. reuse the `go/v3` project layout, upgrading the `PROJECT` file, and scaffolds manually). For further information see [Migration from go/v3 to go/v4 by updating the files manually](#)

## Initialize a go/v4 Project

Project name

For the rest of this document, we are going to use `migration-project` as the project name and `tutorial.kubebuilder.io` as the domain. Please, select and use appropriate values for your case.

Create a new directory with the name of your project. Note that this name is used in the scaffolds to create the name of your manager Pod and of the Namespace where the Manager is deployed by default.

```
$ mkdir migration-project-name  
$ cd migration-project-name
```

Now, we need to initialize a go/v4 project. Before we do that, we'll need to initialize a new go module if we're not on the `GOPATH`. While technically this is not needed inside `GOPATH`, it is still recommended.

```
go mod init tutorial.kubebuilder.io/migration-project
```

The module of your project can be found in the `go.mod` file at the root of your project:

```
module tutorial.kubebuilder.io/migration-project
```

Now, we can finish initializing the project with kubebuilder.

```
kubebuilder init --domain tutorial.kubebuilder.io --plugins=go/v4
```

The domain of your project can be found in the PROJECT file:

```
...
domain: tutorial.kubebuilder.io
...
```

## Migrate APIs and Controllers

Next, we'll re-scaffold out the API types and controllers.

Scaffolding both the API types and controllers

For this example, we are going to consider that we need to scaffold both the API types and the controllers, but remember that this depends on how you scaffolded them in your original project.

```
kubebuilder create api --group batch --version v1 --kind CronJob
```

## Migrate the APIs

If you're using multiple groups

Please run `kubebuilder edit --multigroup=true` to enable multi-group support before migrating the APIs and controllers. Please see [this](#) for more details.

Now, let's copy the API definition from `api/v1/<kind>_types.go` in our old project to the new one.

These files have not been modified by the new plugin, so you should be able to replace your freshly scaffolded files by your old one. There may be some cosmetic changes. So you can choose to only copy the types themselves.

## Migrate the Controllers

Now, let's migrate the controller code from `controllers/cronjob_controller.go` in our old project to `internal/controller/cronjob_controller.go` in the new one.

## Migrate the Webhooks

Skip

If you don't have any webhooks, you can skip this section.

Now let's scaffold the webhooks for our CRD (CronJob). We'll need to run the following command with the `--defaulting` and `--programmatic-validation` flags (since our test project uses defaulting and validating webhooks):

```
kubebuilder create webhook --group batch --version v1 --kind CronJob --defaulting  
--programmatic-validation
```

Now, let's copy the webhook definition from `api/v1/<kind>_webhook.go` from our old project to the new one.

## Others

If there are any manual updates in `main.go` in v3, we need to port the changes to the new `main.go`. We'll also need to ensure all of needed controller-runtime schemes have been registered.

If there are additional manifests added under config directory, port them as well. Please, be aware that the new version go/v4 uses Kustomize v5x and no longer Kustomize v4. Therefore, if added customized implementations in the config you need to ensure that they can work with Kustomize v5 and if not update/upgrade any breaking change that you might face.

In v4, installation of Kustomize has been changed from bash script to `go get`. Change the `kustomize` dependency in Makefile to

```
.PHONY: kustomize
kustomize: $(KUSTOMIZE) ## Download kustomize locally if necessary. If wrong
version is installed, it will be removed before downloading.
$(KUSTOMIZE): $(LOCALBIN)
    @if test -x $(LOCALBIN)/kustomize && ! $(LOCALBIN)/kustomize version | grep -q
$(KUSTOMIZE_VERSION); then \
    echo "$(LOCALBIN)/kustomize version is not expected $(KUSTOMIZE_VERSION).
Removing it before installing."; \
    rm -rf $(LOCALBIN)/kustomize; \
fi
    test -s $(LOCALBIN)/kustomize || GOBIN=$(LOCALBIN) G0111MODULE=on go install
sig.sigs.k8s.io/kustomize/kustomize/v5@$(KUSTOMIZE_VERSION)
```

Change the image name in the Makefile if needed.

## Verification

Finally, we can run `make` and `make docker-build` to ensure things are working fine.

# Migration from go/v3 to go/v4 by updating the files manually

Make sure you understand the differences between Kubebuilder go/v3 and go/v4 before continuing.

Please ensure you have followed the [installation guide](#) to install the required components.

The following guide describes the manual steps required to upgrade your PROJECT config file to begin using go/v4 .

This way is more complex, susceptible to errors, and success cannot be assured. Also, by following these steps you will not get the improvements and bug fixes in the default generated project files.

Usually it is suggested to do it manually if you have customized your project and deviated too much from the proposed scaffold. Before continuing, ensure that you understand the note about [project customizations][project-customizations]. Note that you might need to spend more effort to do this process manually than to organize your project customizations. The proposed layout will keep your project maintainable and upgradable with less effort in the future.

The recommended upgrade approach is to follow the [Migration Guide go/v3 to go/v4](#) instead.

## Migration from project config version “go/v3” to “go/v4”

Update the PROJECT file layout which stores information about the resources that are used to enable plugins make useful decisions while scaffolding. The layout field indicates the scaffolding and the primary plugin version in use.

### Steps to migrate

#### Migrate the layout version into the PROJECT file

The following steps describe the manual changes required to bring the project configuration file (PROJECT). These change will add the information that Kubebuilder would add when generating the file. This file can be found in the root directory.

Update the PROJECT file by replacing:

```
layout:  
- go.kubebuilder.io/v3
```

With:

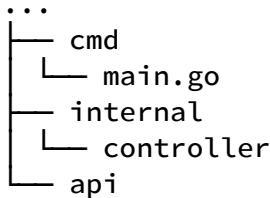
```
layout:  
- go.kubebuilder.io/v4
```

## Changes to the layout

### New layout:

- The directory `apis` was renamed to `api` to follow the standard
- The `controller(s)` directory has been moved under a new directory called `internal` and renamed to singular as well `controller`
- The `main.go` previously scaffolded in the root directory has been moved under a new directory called `cmd`

Therefore, you can check the changes in the layout results into:



### Migrating to the new layout:

- Create a new directory `cmd` and move the `main.go` under it.
- If your project support multi-group the APIs are scaffold under a directory called `apis`. Rename this directory to `api`
- Move the `controllers` directory under the `internal` and rename it for `controller`
- Now ensure that the imports will be updated accordingly by:
  - Update the `main.go` imports to look for the new path of your controllers under the `internal/controller` directory

### Then, let's update the scaffolds paths

- Update the Dockerfile to ensure that you will have:

```
COPY cmd/main.go cmd/main.go
COPY api/ api/
COPY internal/controller/ internal/controller/
```

Then, replace:

```
RUN CGO_ENABLED=0 GOOS=${TARGETOS:-linux} GOARCH=${TARGETARCH} go build -a -o manager main.go
```

With:

```
RUN CGO_ENABLED=0 GOOS=${TARGETOS:-linux} GOARCH=${TARGETARCH} go build -a -o manager cmd/main.go
```

- Update the Makefile targets to build and run the manager by replacing:

```
.PHONY: build
build: manifests generate fmt vet ## Build manager binary.
      go build -o bin/manager main.go

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

With:

```
.PHONY: build
build: manifests generate fmt vet ## Build manager binary.
      go build -o bin/manager cmd/main.go

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

- Update the `internal/controller/suite_test.go` to set the path for the CRDDirectoryPaths :

Replace:

```
CRDDirectoryPaths:      []string{filepath.Join("../", "config", "crd", "bases")},
```

With:

```
CRDDirectoryPaths:      []string{filepath.Join("../", "..", "config", "crd",  
"bases")},
```

Note that if your project has multiple groups (`multigroup:true`) then the above update should result into `..`, `..`, `..`, instead of `..`,`..`

## Now, let's update the PATHs in the PROJECT file accordingly

The PROJECT tracks the paths of all APIs used in your project. Ensure that they now point to `api/...` as the following example:

### Before update:

```
group: crew  
kind: Captain  
path: sigs.k8s.io/kubebuilder/testdata/project-v4/apis/crew/v1
```

### After Update:

```
group: crew  
kind: Captain  
path: sigs.k8s.io/kubebuilder/testdata/project-v4/api/crew/v1
```

## Update kustomize manifests with the changes made so far

- Update the manifest under `config/` directory with all changes performed in the default scaffold done with `go/v4` plugin. (see for example `testdata/project-v4/config/`) to get all changes in the default scaffolds to be applied on your project
- Create `config/samples/kustomization.yaml` with all Custom Resources samples specified into `config/samples`. (see for example `testdata/project-v4/config/samples/kustomization.yaml`)

### i `config/` directory with changes into the scaffold files

Note that under the `config/` directory you will find scaffolding changes since using `go/v4` you will ensure that you are no longer using Kustomize v3x.

You can mainly compare the `config/` directory from the samples scaffolded under the `testdata` directory by checking the differences between the `testdata/project-`

`v3/config/` with `testdata/project-v4/config/` which are samples created with the same commands with the only difference being versions.

However, note that if you create your project with Kubebuilder CLI 3.0.0, its scaffolds might change to accommodate changes up to the latest releases using `go/v3` which are not considered breaking for users and/or are forced by the changes introduced in the dependencies used by the project such as [controller-runtime](#) and [controller-tools](#).

## If you have webhooks:

Replace the import `admissionv1beta1 "k8s.io/api/admission/v1beta1"` with `admissionv1 "k8s.io/api/admission/v1"` in the webhook test files

## Makefile updates

Update the Makefile with the changes which can be found in the samples under `testdata` for the release tag used. (see for example `testdata/project-v4/Makefile` )

## Update the dependencies

Update the `go.mod` with the changes which can be found in the samples under `testdata` for the release tag used. (see for example `testdata/project-v4/go.mod` ). Then, run `go mod tidy` to ensure that you get the latest dependencies and your Golang code has no breaking changes.

## Verification

In the steps above, you updated your project manually with the goal of ensuring that it follows the changes in the layout introduced with the `go/v4` plugin that update the scaffolds.

There is no option to verify that you properly updated the `PROJECT` file of your project. The best way to ensure that everything is updated correctly, would be to initialize a project using the `go/v4` plugin, (ie) using `kubebuilder init --domain tutorial.kubebuilder.io plugins=go/v4` and generating the same API(s), controller(s), and webhook(s) in order to compare the generated configuration with the manually changed configuration.

Also, after all updates you would run the following commands:

- `make manifests` (to re-generate the files using the latest version of the controller-gen after you update the Makefile)
- `make all` (to ensure that you are able to build and perform all operations)

# Single Group to Multi-Group

## Note

While Kubebuilder will not scaffold out a project structure compatible with multiple API groups in the same repository by default, it's possible to modify the default project structure to support it.

Note that the process mainly is to ensure that your API(s) and controller(s) will be moved under new directories with their respective group name.

Let's migrate the [CronJob example](#).

## Instructions vary per project layout

You can verify the version by looking at the PROJECT file. The currently default and recommended version is go/v4.

The layout go/v3 is **deprecated**, if you are using go/v3 it is recommended that you migrate to go/v4, however this documentation is still valid. [Migration from go/v3 to go/v4](#).

To change the layout of your project to support Multi-Group run the command `kubebuilder edit --multigroup=true .`. Once you switch to a multi-group layout, the new Kinds will be generated in the new layout but additional manual work is needed to move the old API groups to the new layout.

Generally, we use the prefix for the API group as the directory name. We can check `api/v1/groupversion_info.go` to find that out:

```
// +groupName=batch.tutorial.kubebuilder.io
package v1
```

Then, we'll move our existing APIs into a new subdirectory named after the group. Considering the [CronJob example](#), subdirectory name is "batch":

```
mkdir api/batch
mv api/* api/batch
```

After moving the APIs to a new directory, the same needs to be applied to the controllers. For go/v4:

```
mkdir internal/controller/batch  
mv internal/controller/* internal/controller/batch/
```

The same needs to be applied for any pre-existing [webhooks](#):

```
mkdir internal/webhook/batch  
mv internal/webhook/* internal/webhook/batch/
```

For any new webhook created for a new group, the respective functions will be created under subdirectory `internal/webhook/<group>/`.

If you are using the deprecated layout go/v3

Then, your layout has not the internal directory. So, you will move the controller(s) under a directory with the name of the API group which it is responsible for manage. ````bash mkdir controller/batch mv controller/\* controller/batch/ ````

Next, we'll need to update all the references to the old package name. For CronJob, the paths to be replaced would be `main.go` and `controllers/batch/cronjob_controller.go` to their respective locations in the new project structure.

If you've added additional files to your project, you'll need to track down imports there as well.

Finally, fix the PROJECT file manually, the command `kubebuilder edit --multigroup=true` sets our project to multigroup, but it doesn't fix the path of the existing APIs. For each resource we need to modify the path.

For instance, for a file:

```

# Code generated by tool. DO NOT EDIT.
# This file is used to track the info used to scaffold your project
# and allow the plugins properly work.
# More info: https://book.kubebuilder.io/reference/project-config.html
domain: tutorial.kubebuilder.io
layout:
- go.kubebuilder.io/v4
multigroup: true
projectName: test
repo: tutorial.kubebuilder.io/project
resources:
- api:
  crdVersion: v1
  namespaced: true
  controller: true
  domain: tutorial.kubebuilder.io
  group: batch
  kind: CronJob
  path: tutorial.kubebuilder.io/project/api/v1beta1
  version: v1beta1
version: "3"

```

Replace `path: tutorial.kubebuilder.io/project/api/v1beta1` for `path: tutorial.kubebuilder.io/project/api/batch/v1beta1`

In this process, if the project is not new and has previous implemented APIs they would still need to be modified as needed. Notice that with the `multi-group` project the Kind API's files are created under `api/<group>/<version>` instead of `api/<version>`. Also, note that the controllers will be created under `internal/controller/<group>` instead of `internal/controller`.

That is the reason why we moved the previously generated APIs to their respective locations in the new structure. Remember to update the references in imports accordingly.

For envtest to install CRDs correctly into the test environment, the relative path to the CRD directory needs to be updated accordingly in each

`internal/controller/<group>/suite_test.go` file. We need to add additional `.."` to our CRD directory relative path as shown below.

```

By("bootstrapping test environment")
testEnv = &envtest.Environment{
    CRDDirectoryPaths: []string{filepath.Join("..", "..", "config", "crd",
"bases")},
}

```

The [CronJob tutorial](#) explains each of these changes in more detail (in the context of how they're generated by KubeBuilder for single-group projects).

## Overview

Please note that all input utilized via the Kubebuilder tool is tracked in the PROJECT file ([example](#)). This file is responsible for storing essential information, representing various facets of the Project such as its layout, plugins, APIs, and more. ([More info](#)).

With the release of new plugin versions/layouts or even a new Kubebuilder CLI version with scaffold changes, an easy way to upgrade your project is by re-scaffolding. This process allows users to employ tools like IDEs to compare changes, enabling them to overlay their code implementation on the new scaffold or integrate these changes into their existing projects.

## When to use it ?

This command is useful when you want to upgrade an existing project to the latest version of the Kubebuilder project layout. It makes it easier for the users to migrate their operator projects to the new scaffolding.

## How to use it ?

### To upgrade the scaffold of your project to use a new plugin version:

```
kubebuilder alpha generate --plugins="pluginkey/version"
```

### To upgrade the scaffold of your project to get the latest changes:

Currently, it supports two optional params, `input-dir` and `output-dir`.

`input-dir` is the path to the existing project that you want to re-scaffold. Default is the current working directory.

`output-dir` is the path to the directory where you want to generate the new project. Default is a subdirectory in the current working directory.

```
kubebuilder alpha generate --input-dir=/path/to/existing/project --output-dir=/path/to/new/project
```



! Regarding `input-dir` and `output-dir`:

If neither `input-dir` nor `output-dir` are specified, the project will be regenerated in the current directory. This approach facilitates comparison between your current local branch and the version stored upstream (e.g., GitHub main branch). This way, you can easily overlay your project's code changes atop the new scaffold.

## Further Resources:

- Check out [video](#) to show how it works
- See the [design proposal documentation](#)

## Reference

- Generating CRDs
- [Using Finalizers](#) Finalizers are a mechanism to execute any custom logic related to a resource before it gets deleted from Kubernetes cluster.
- [Watching Resources](#) Watch resources in the Kubernetes cluster to be informed and take actions on changes.
  - [Watching Secondary Resources that are Owned](#)
  - [Watching Secondary Resources that are NOT Owned](#)
  - [Using Predicates to Refine Watches](#)
- Kind cluster
- [What's a webhook?](#) Webhooks are HTTP callbacks, there are 3 types of webhooks in k8s: 1) admission webhook 2) CRD conversion webhook 3) authorization webhook
  - [Admission webhook](#) Admission webhooks are HTTP callbacks for mutating or validating resources before the API server admit them.
- [Markers for Config/Code Generation](#)
  - [CRD Generation](#)
  - [CRD Validation](#)
  - [Webhook](#)
  - [Object/DeepCopy](#)
  - [RBAC](#)
  - [Scaffold](#)
- [Monitoring with Pprof](#)
- [controller-gen CLI](#)
- [completion](#)
- [Artifacts](#)
- [Platform Support](#)
- [Sub-Module Layouts](#)
- [Using an external Resource / API](#)
- [Metrics](#)

- Reference
- CLI plugins

## Generating CRDs

Kubebuilder uses a tool called [controller-gen](#) to generate utility code and Kubernetes object YAML, like CustomResourceDefinitions.

To do this, it makes use of special “marker comments” (comments that start with `// +`) to indicate additional information about fields, types, and packages. In the case of CRDs, these are generally pulled from your `_types.go` files. For more information on markers, see the [marker reference docs](#).

Kubebuilder provides a `make manifests` target to run controller-gen and generate CRDs: `make manifests`.

When you run `make manifests`, you should see CRDs generated under the `config/crd/bases` directory. `make manifests` can generate a number of other artifacts as well – see the [marker reference docs](#) for more details.

## Validation

CRDs support declarative validation using an [OpenAPI v3 schema](#) in the `validation` section.

In general, [validation markers](#) may be attached to fields or to types. If you’re defining complex validation, if you need to re-use validation, or if you need to validate slice elements, it’s often best to define a new type to describe your validation.

For example:

```

type ToySpec struct {
    // +kubebuilder:validation:MaxLength=15
    // +kubebuilder:validation:MinLength=1
    Name string `json:"name,omitempty"`

    // +kubebuilder:validation:MaxItems=500
    // +kubebuilder:validation:MinItems=1
    // +kubebuilder:validation:UniqueItems=true
    Knights []string `json:"knights,omitempty"`

    Alias Alias `json:"alias,omitempty"`
    Rank   Rank  `json:"rank"`
}

// +kubebuilder:validation:Enum=Lion;Wolf;Dragon
type Alias string

// +kubebuilder:validation:Minimum=1
// +kubebuilder:validation:Maximum=3
// +kubebuilder:validation:ExclusiveMaximum=false
type Rank int32

```

## Additional Printer Columns

Starting with Kubernetes 1.11, `kubectl get` can ask the server what columns to display. For CRDs, this can be used to provide useful, type-specific information with `kubectl get`, similar to the information provided for built-in types.

The information that gets displayed can be controlled with the [additionalPrinterColumns field](#) on your CRD, which is controlled by the `+kubebuilder:printcolumn` marker on the Go type for your CRD.

For instance, in the following example, we add fields to display information about the knights, rank, and alias fields from the validation example:

```

// +kubebuilder:printcolumn:name="Alias",type=string,JSONPath=`.spec.alias`
// +kubebuilder:printcolumn:name="Rank",type=integer,JSONPath=`.spec.rank`
// +kubebuilder:printcolumn:name="Bravely Run
Away",type=boolean,JSONPath=`.spec.knights[?(@ == "Sir
Robin")]`,description="when danger rears its ugly head, he bravely turned his
tail and fled",priority=10
//
+kubebuilder:printcolumn:name="Age",type="date",JSONPath=".metadata.creationTi
mestamp"
type Toy struct {
    metav1.TypeMeta `json:",inline"`
    metav1.ObjectMeta `json:"metadata,omitempty"`

    Spec ToySpec `json:"spec,omitempty"`
    Status ToyStatus `json:"status,omitempty"`
}

```

## Subresources

CRDs can choose to implement the `/status` and `/scale` [subresources](#) as of Kubernetes 1.13.

It's generally recommended that you make use of the `/status` subresource on all resources that have a status field.

Both subresources have a corresponding [marker](#).

## Status

The status subresource is enabled via `+kubebuilder:subresource:status`. When enabled, updates at the main resource will not change status. Similarly, updates to the status subresource cannot change anything but the status field.

For example:

```

// +kubebuilder:subresource:status
type Toy struct {
    metav1.TypeMeta `json:",inline"`
    metav1.ObjectMeta `json:"metadata,omitempty"`

    Spec ToySpec `json:"spec,omitempty"`
    Status ToyStatus `json:"status,omitempty"`
}

```

## Scale

The scale subresource is enabled via `+kubebuilder:subresource:scale`. When enabled, users will be able to use `kubectl scale` with your resource. If the `selectorpath` argument pointed to the string form of a label selector, the HorizontalPodAutoscaler will be able to autoscale your resource.

For example:

```
type CustomSetSpec struct {
    Replicas *int32 `json:"replicas"`
}

type CustomSetStatus struct {
    Replicas int32 `json:"replicas"`
    Selector string `json:"selector"` // this must be the string form of the
                                     selector
}

// +kubebuilder:subresource:status
//
+kubebuilder:subresource:scale:specpath=.spec.replicas,statuspath=.status.replicas,selectorpath=.status.selector
type CustomSet struct {
    metav1.TypeMeta `json:",inline"`
    metav1.ObjectMeta `json:"metadata,omitempty"`

    Spec CustomSetSpec `json:"spec,omitempty"`
    Status CustomSetStatus `json:"status,omitempty"`
}
```

## Multiple Versions

As of Kubernetes 1.13, you can have multiple versions of your Kind defined in your CRD, and use a webhook to convert between them.

For more details on this process, see the [multiversion tutorial](#).

By default, KubeBuilder disables generating different validation for different versions of the Kind in your CRD, to be compatible with older Kubernetes versions.

You'll need to enable this by switching the line in your makefile that says `CRD_OPTIONS ?= "crd:trivialVersions=true,preserveUnknownFields=false` to `CRD_OPTIONS ?=`

`crd:preserveUnknownFields=false` if using v1beta CRDs, and `CRD_OPTIONS ?= crd` if using v1 (recommended).

Then, you can use the `+kubebuilder:storageversion` marker to indicate the GVK that should be used to store data by the API server.

## Under the hood

Kubebuilder scaffolds out make rules to run `controller-gen`. The rules will automatically install controller-gen if it's not on your path using `go install` with Go modules.

You can also run `controller-gen` directly, if you want to see what it's doing.

Each controller-gen "generator" is controlled by an option to controller-gen, using the same syntax as markers. controller-gen also supports different output "rules" to control how and where output goes. Notice the `manifests` make rule (condensed slightly to only generate CRDs):

```
# Generate manifests for CRDs
manifests: controller-gen
    $(CONTROLLER_GEN) rbac:roleName=manager-role crd webhook paths="./*"
output:crd:artifacts:config=config/crd/bases
```

It uses the `output:crd:artifacts` output rule to indicate that CRD-related config (non-code) artifacts should end up in `config/crd/bases` instead of `config/crd`.

To see all the options including generators for `controller-gen`, run

```
$ controller-gen -h
```

or, for more details:

```
$ controller-gen -hhh
```

## Using Finalizers

Finalizers allow controllers to implement asynchronous pre-delete hooks. Let's say you create an external resource (such as a storage bucket) for each object of your API type, and you want to delete the associated external resource on object's deletion from Kubernetes, you can use a finalizer to do that.

You can read more about the finalizers in the [Kubernetes reference docs](#). The section below demonstrates how to register and trigger pre-delete hooks in the `Reconcile` method of a controller.

The key point to note is that a finalizer causes "delete" on the object to become an "update" to set deletion timestamp. Presence of deletion timestamp on the object indicates that it is being deleted. Otherwise, without finalizers, a delete shows up as a reconcile where the object is missing from the cache.

Highlights:

- If the object is not being deleted and does not have the finalizer registered, then add the finalizer and update the object in Kubernetes.
- If object is being deleted and the finalizer is still present in finalizers list, then execute the pre-delete logic and remove the finalizer and update the object.
- Ensure that the pre-delete logic is idempotent.

```
$ vim ../../cronjob-tutorial/testdata/finalizer_example.go
// Apache License (hidden)
// Imports (hidden)
```

By default, kubebuilder will include the RBAC rules necessary to update finalizers for CronJobs.

```
//
+kubebuilder:rbac:groups=batch.tutorial.kubebuilder.io,resources=cronjobs,verbs=get;list;watch;create;update;patch;delete
//
+kubebuilder:rbac:groups=batch.tutorial.kubebuilder.io,resources=cronjobs/status,verbs=get;update;patch
//
+kubebuilder:rbac:groups=batch.tutorial.kubebuilder.io,resources=cronjobs/finalizers,verbs=update
```

The code snippet below shows skeleton code for implementing a finalizer.

```
func (r *CronJobReconciler) Reconcile(ctx context.Context, req ctrl.Request)
(ctrl.Result, error) {
    log := r.Log.WithValues("cronjob", req.NamespacedName)

    cronJob := &batchv1.CronJob{}
    if err := r.Get(ctx, req.NamespacedName, cronJob); err != nil {
        log.Error(err, "unable to fetch CronJob")
        // we'll ignore not-found errors, since they can't be fixed by an
immediate
        // requeue (we'll need to wait for a new notification), and we can get
them
        // on deleted requests.
        return ctrl.Result{}, client.IgnoreNotFound(err)
    }

    // name of our custom finalizer
    myFinalizerName := "batch.tutorial.kubebuilder.io/finalizer"

    // examine DeletionTimestamp to determine if object is under deletion
    if cronJob.ObjectMeta.DeletionTimestamp.IsZero() {
        // The object is not being deleted, so if it does not have our
finalizer,
        // then let's add the finalizer and update the object. This is
equivalent
        // to registering our finalizer.
        if !controllerutil.ContainsFinalizer(cronJob, myFinalizerName) {
            controllerutil.AddFinalizer(cronJob, myFinalizerName)
            if err := r.Update(ctx, cronJob); err != nil {
                return ctrl.Result{}, err
            }
        }
    } else {
        // The object is being deleted
        if controllerutil.ContainsFinalizer(cronJob, myFinalizerName) {
            // our finalizer is present, so let's handle any external
dependency
            if err := r.deleteExternalResources(cronJob); err != nil {
                // if fail to delete the external dependency here, return with
error
                // so that it can be retried.
                return ctrl.Result{}, err
            }

            // remove our finalizer from the list and update it.
            controllerutil.RemoveFinalizer(cronJob, myFinalizerName)
            if err := r.Update(ctx, cronJob); err != nil {
                return ctrl.Result{}, err
            }
        }
    }

    // Stop reconciliation as the item is being deleted
    return ctrl.Result{}, nil
}
```

```
// Your reconcile logic

return ctrl.Result{}, nil
}

func (r *Reconciler) deleteExternalResources(cronJob *batch.CronJob) error {
//
// delete any external resources associated with the cronJob
//
// Ensure that delete implementation is idempotent and safe to invoke
// multiple times for same object.
}
```

## What is “Reconciliation” in Operators?

When you create a project using Kubebuilder, see the scaffolded code generated under `cmd/main.go`. This code initializes a [Manager](#), and the project relies on the [controller-runtime](#) framework. The Manager manages [Controllers](#), which offer a reconcile function that synchronizes resources until the desired state is achieved within the cluster.

Reconciliation is an ongoing loop that executes necessary operations to maintain the desired state, adhering to Kubernetes principles, such as the [control loop](#). For further information, check out the [Operator patterns](#) documentation from Kubernetes to better understand those concepts.

## Why should reconciliations be idempotent?

When developing operators, the controller’s reconciliation loop needs to be idempotent. By following the [Operator pattern](#) we create [controllers](#) that provide a reconcile function responsible for synchronizing resources until the desired state is reached on the cluster. Developing idempotent solutions will allow the reconciler to correctly respond to generic or unexpected events, easily deal with application startup or upgrade. More explanation on this is available [here](#).

Writing reconciliation logic according to specific events, breaks the recommendation of operator pattern and goes against the design principles of [controller-runtime](#). This may lead to unforeseen consequences, such as resources becoming stuck and requiring manual intervention.

## Understanding Kubernetes APIs and following API conventions

Building your operator commonly involves extending the Kubernetes API itself. It is helpful to understand precisely how Custom Resource Definitions (CRDs) interact with the Kubernetes API. Also, the [Kubebuilder documentation](#) on Groups and Versions and Kinds may be helpful to understand these concepts better as they relate to operators.

Additionally, we recommend checking the documentation on [Operator patterns](#) from Kubernetes to better understand the purpose of the standard solutions built with KubeBuilder.

## Why you should adhere to the Kubernetes API conventions and standards

Embracing the [Kubernetes API conventions and standards](#) is crucial for maximizing the potential of your applications and deployments. By adhering to these established practices, you can benefit in several ways.

Firstly, adherence ensures seamless interoperability within the Kubernetes ecosystem. Following conventions allows your applications to work harmoniously with other components, reducing compatibility issues and promoting a consistent user experience.

Secondly, sticking to API standards enhances the maintainability and troubleshooting of your applications. Adopting familiar patterns and structures makes debugging and supporting your deployments easier, leading to more efficient operations and quicker issue resolution.

Furthermore, leveraging the Kubernetes API conventions empowers you to harness the platform's full capabilities. By working within the defined framework, you can leverage the rich set of features and resources offered by Kubernetes, enabling scalability, performance optimization, and resilience.

Lastly, embracing these standards future-proofs your native solutions. By aligning with the evolving Kubernetes ecosystem, you ensure compatibility with future updates, new features, and enhancements introduced by the vibrant Kubernetes community.

In summary, by adhering to the Kubernetes API conventions and standards, you unlock the potential for seamless integration, simplified maintenance, optimal performance, and future-readiness, all contributing to the success of your applications and deployments.

# **Why should one avoid a system design where a single controller is responsible for managing multiple CRDs (Custom Resource Definitions)(for example, an '*install\_all\_controller.go*)?**

Avoid a design solution where the same controller reconciles more than one Kind. Having many Kinds (such as CRDs), that are all managed by the same controller, usually goes against the design proposed by controller-runtime. Furthermore, this might hurt concepts such as encapsulation, the Single Responsibility Principle, and Cohesion. Damaging these concepts may cause unexpected side effects and increase the difficulty of extending, reusing, or maintaining the operator. Having one controller manage many Custom Resources (CRs) in an Operator can lead to several issues:

- **Complexity:** A single controller managing multiple CRs can increase the complexity of the code, making it harder to understand, maintain, and debug.
- **Scalability:** Each controller typically manages a single kind of CR for scalability. If a single controller handles multiple CRs, it could become a bottleneck, reducing the overall efficiency and responsiveness of your system.
- **Single Responsibility Principle:** Following this principle from software engineering, each controller should ideally have only one job. This approach simplifies development and debugging, and makes the system more robust.
- **Error Isolation:** If one controller manages multiple CRs and an error occurs, it could potentially impact all the CRs it manages. Having a single controller per CR ensures that an issue with one controller or CR does not directly affect others.
- **Concurrency and Synchronization:** A single controller managing multiple CRs could lead to race conditions and require complex synchronization, especially if the CRs have interdependencies.

In conclusion, while it might seem efficient to have a single controller manage multiple CRs, it often leads to higher complexity, lower scalability, and potential stability issues. It's generally better to adhere to the single responsibility principle, where each CR is managed by its own controller.

## **Why You Should Adopt Status Conditions**

We recommend you manage your solutions using Status Conditionals following the [K8s Api conventions](#) because:

- **Standardization:** Conditions provide a standardized way to represent the state of an Operator's custom resources, making it easier for users and tools to understand and interpret the resource's status.
- **Readability:** Conditions can clearly express complex states by using a combination of multiple conditions, making it easier for users to understand the current state and progress of the resource.
- **Extensibility:** As new features or states are added to your Operator, conditions can be easily extended to represent these new states without requiring significant changes to the existing API or structure.
- **Observability:** Status conditions can be monitored and tracked by cluster administrators and external monitoring tools, enabling better visibility into the state of the custom resources managed by the Operator.
- **Compatibility:** By adopting the common pattern of using conditions in Kubernetes APIs, Operator authors ensure their custom resources align with the broader ecosystem, which helps users to have a consistent experience when interacting with multiple Operators and resources in their clusters.

#### Example of Usage

Check out the [Deploy Image Plugin](#). This plugin allows users to scaffold API/Controllers to deploy and manage an Operand (image) on the cluster following the guidelines and best practices. It abstracts the complexities of achieving this goal while allowing users to customize the generated code.

Therefore, you can check an example of Status Conditional usage by looking at its API(s) scaffolded and code implemented under the Reconciliation into its Controllers.

## Creating Events

It is often useful to publish *Event* objects from the controller Reconcile function as they allow users or any automated processes to see what is going on with a particular object and respond to them.

Recent Events for an object can be viewed by running `$ kubectl describe <resource kind> <resource name>`. Also, they can be checked by running `$ kubectl get events`.

### **Events should be raised in certain circumstances only**

Be aware that it is **not** recommended to emit Events for all operations. If authors raise too many events, it brings bad UX experiences for those consuming the solutions on the cluster, and they may find it difficult to filter an actionable event from the cluster. For more information, please take a look at the [Kubernetes APIs convention](#).

## Writing Events

Anatomy of an Event:

```
Event(object runtime.Object, eventtype, reason, message string)
```

- `object` is the object this event is about.
- `eventtype` is this event type, and is either *Normal* or *Warning*. ([More info](#))
- `reason` is the reason this event is generated. It should be short and unique with `UpperCamelCase` format. The value could appear in `switch` statements by automation. ([More info](#))
- `message` is intended to be consumed by humans. ([More info](#))

### Example Usage

Following is an example of a code implementation that raises an Event.

```
// The following implementation will raise an event
r.Recorder.Eventf(cr, "Warning", "Deleting",
    "Custom Resource %s is being deleted from the namespace %s",
    cr.Name, cr.Namespace)
```

## How to be able to raise Events?

Following are the steps with examples to help you raise events in your controller's reconciliations. Events are published from a Controller using an [EventRecorder](#) type `CorrelatorOptions` struct , which can be created for a Controller by calling `GetRecorder(name string)` on a Manager. See that we will change the implementation scaffolded in `cmd/main.go` :

```
if err = (&controller.MyKindReconciler{
    Client:   mgr.GetClient(),
    Scheme:   mgr.GetScheme(),
    // Note that we added the following line:
    Recorder: mgr.GetEventRecorderFor("mykind-controller"),
).SetupWithManager(mgr); err != nil {
    setupLog.Error(err, "unable to create controller", "controller",
"MyKind")
    os.Exit(1)
}
```

## Allowing usage of EventRecorder on the Controller

To raise an event, you must have access to `record.EventRecorder` in the Controller. Therefore, firstly let's update the controller implementation:

```
import (
...
"k8s.io/client-go/tools/record"
...
)
// MyKindReconciler reconciles a MyKind object
type MyKindReconciler struct {
    client.Client
    Scheme   *runtime.Scheme
    // See that we added the following code to allow us to pass the
    record.EventRecorder
    Recorder record.EventRecorder
}
```

## Passing the EventRecorder to the Controller

Events are published from a Controller using an [EventRecorder] type `CorrelatorOptions` struct , which can be created for a Controller by calling `GetRecorder(name string)` on a Manager. See that we will change the implementation scaffolded in `cmd/main.go` :

```
if err = (&controller.MyKindReconciler{
    Client:   mgr.GetClient(),
    Scheme:   mgr.GetScheme(),
    // Note that we added the following line:
    Recorder: mgr.GetEventRecorderFor("mykind-controller"),
}).SetupWithManager(mgr); err != nil {
    setupLog.Error(err, "unable to create controller", "controller",
"MyKind")
    os.Exit(1)
}
```

## Granting the required permissions

You must also grant the RBAC rules permissions to allow your project to create Events. Therefore, ensure that you add the [RBAC](#) into your controller:

```
...
// +kubebuilder:rbac:groups=core,resources=events,verbs=create;patch
...
func (r *MyKindReconciler) Reconcile(ctx context.Context, req ctrl.Request)
(ctrl.Result, error) {
```

And then, run `$ make manifests` to update the rules under `config/rbac/role.yaml`.

## Watching Resources

When extending the Kubernetes API, we aim to ensure that our solutions behave consistently with Kubernetes itself. For example, consider a `Deployment` resource, which is managed by a controller. This controller is responsible for responding to changes in the cluster—such as when a `Deployment` is created, updated, or deleted—by triggering reconciliation to ensure the resource's state matches the desired state.

Similarly, when developing our controllers, we want to watch for relevant changes in resources that are crucial to our solution. These changes—whether creations, updates, or deletions—should trigger the reconciliation loop to take appropriate actions and maintain consistency across the cluster.

The [controller-runtime](#) library provides several ways to watch and manage resources.

## Primary Resources

The **Primary Resource** is the resource that your controller is responsible for managing. For example, if you create a custom resource definition (CRD) for `MyApp`, the corresponding controller is responsible for managing instances of `MyApp`.

In this case, `MyApp` is the **Primary Resource** for that controller, and your controller's reconciliation loop focuses on ensuring the desired state of these primary resources is maintained.

When you create a new API using Kubebuilder, the following default code is scaffolded, ensuring that the controller watches all relevant events—such as creations, updates, and deletions—for (`For()`) the new API.

This setup guarantees that the reconciliation loop is triggered whenever an instance of the API is created, updated, or deleted:

```
// Watches the primary resource (e.g., MyApp) for create, update, delete
events
if err := ctrl.NewControllerManagedBy(mgr).
    For(&YourAPISpec{}). <-- See there that the Controller is For this API
    Complete(r); err != nil {
    return err
}
```

# Secondary Resources

Your controller will likely also need to manage **Secondary Resources**, which are the resources required on the cluster to support the **Primary Resource**.

Changes to these **Secondary Resources** can directly impact the **Primary Resource**, so the controller must watch and reconcile these resources accordingly.

## Which are Owned by the Controller

These **Secondary Resources**, such as `Services`, `ConfigMaps`, or `Deployments`, when Owned by the controllers, are created and managed by the specific controller and are tied to the **Primary Resource** via [OwnerReferences](#).

For example, if we have a controller to manage our CR(s) of the Kind `MyApp` on the cluster, which represents our application solution, all resources required to ensure that `MyApp` is up and running with the desired number of instances will be **Secondary Resources**. The code responsible for creating, deleting, and updating these resources will be part of the `MyApp` Controller. We would add the appropriate [OwnerReferences](#) using the [controllerutil.SetControllerReference](#) function to indicate that these resources are owned by the same controller responsible for managing `MyApp` instances, which will be reconciled by the `MyAppReconciler`.

Additionally, if the **Primary Resource** is deleted, Kubernetes' garbage collection mechanism ensures that all associated **Secondary Resources** are automatically deleted in a cascading manner.

## Which are NOT Owned by the Controller

Note that **Secondary Resources** can either be APIs/CRDs defined in your project or in other projects that are relevant to the **Primary Resources**, but which the specific controller is not responsible for creating or managing.

For example, if we have a CRD that represents a backup solution (i.e. `MyBackup`) for our `MyApp`, it might need to watch changes in the `MyApp` resource to trigger reconciliation in `MyBackup` to ensure the desired state. Similarly, `MyApp`'s behavior might also be impacted by CRDs/APIs defined in other projects.

In both scenarios, these resources are treated as **Secondary Resources**, even if they are not owned (i.e., not created or managed) by the `MyAppController`.

In Kubebuilder, resources that are not defined in the project itself and are not a **Core Type** (those not defined in the Kubernetes API) are called **External Types**.

An **External Type** refers to a resource that is not defined in your project but one that you need to watch and respond to. For example, if **Operator A** manages a `MyApp` CRD for application deployment, and **Operator B** handles backups, **Operator B** can watch the `MyApp` CRD as an external type to trigger backup operations based on changes in `MyApp`.

In this scenario, **Operator B** could define a `BackupConfig` CRD that relies on the state of `MyApp`. By treating `MyApp` as a **Secondary Resource**, **Operator B** can watch and reconcile changes in **Operator A's** `MyApp`, ensuring that backup processes are initiated whenever `MyApp` is updated or scaled.

## General Concept of Watching Resources

Whether a resource is defined within your project or comes from an external project, the concept of **Primary** and **Secondary Resources** remains the same:

- The **Primary Resource** is the resource the controller is primarily responsible for managing.
- **Secondary Resources** are those that are required to ensure the primary resource works as desired.

Therefore, regardless of whether the resource was defined by your project or by another project, your controller can watch, reconcile, and manage changes to these resources as needed.

## Why does watching the secondary resources matter?

When building a Kubernetes controller, it's crucial to not only focus on **Primary Resources** but also to monitor **Secondary Resources**. Failing to track these resources can lead to inconsistencies in your controller's behavior and the overall cluster state.

Secondary resources may not be directly managed by your controller, but changes to these resources can still significantly impact the primary resource and your controller's functionality. Here are the key reasons why it's important to watch them:

- **Ensuring Consistency:**

- Secondary resources (e.g., child objects or external dependencies) may diverge from their desired state. For instance, a secondary resource may be modified or deleted, causing the system to fall out of sync.
- Watching secondary resources ensures that any changes are detected immediately, allowing the controller to reconcile and restore the desired state.

- **Avoiding Random Self-Healing:**

- Without watching secondary resources, the controller may “heal” itself only upon restart or when specific events are triggered. This can cause unpredictable or delayed reactions to issues.
- Monitoring secondary resources ensures that inconsistencies are addressed promptly, rather than waiting for a controller restart or external event to trigger reconciliation.

- **Effective Lifecycle Management:**

- Secondary resources might not be owned by the controller directly, but their state still impacts the behavior of primary resources. Without watching these, you risk leaving orphaned or outdated resources.
- Watching non-owned secondary resources lets the controller respond to lifecycle events (create, update, delete) that might affect the primary resource, ensuring consistent behavior across the system.

See [Watching Secondary Resources That Are Not Owned](#) for an example.

## Why not use `RequeueAfter X` for all scenarios instead of watching resources?

Kubernetes controllers are fundamentally **event-driven**. When creating a controller, the **Reconciliation Loop** is typically triggered by **events** such as `create`, `update`, or `delete` actions on resources. This event-driven approach is more efficient and responsive compared to constantly requeuing or polling resources using `RequeueAfter`. This ensures that the system only takes action when necessary, maintaining both performance and efficiency.

In many cases, **watching resources** is the preferred approach for ensuring Kubernetes resources remain in the desired state. It is more efficient, responsive, and aligns with Kubernetes’ event-driven architecture. However, there are scenarios where `RequeueAfter` is appropriate and necessary, particularly for managing external systems that do not emit events or for handling resources that take time to converge, such as long-running

processes. Relying solely on `RequeueAfter` for all scenarios can lead to unnecessary overhead and delayed reactions. Therefore, it is essential to prioritize **event-driven reconciliation** by configuring your controller to **watch resources** whenever possible, and reserving `RequeueAfter` for situations where periodic checks are required.

## When `RequeueAfter X` is Useful

While `RequeueAfter` is not the primary method for triggering reconciliations, there are specific cases where it is necessary, such as:

- **Observing External Systems:** When working with external resources that do not generate events (e.g., external databases or third-party services), `RequeueAfter` allows the controller to periodically check the status of these resources.
- **Time-Based Operations:** Some tasks, such as rotating secrets or renewing certificates, must happen at specific intervals. `RequeueAfter` ensures these operations are performed on schedule, even when no other changes occur.
- **Handling Errors or Delays:** When managing resources that encounter errors or require time to self-heal, `RequeueAfter` ensures the controller waits for a specified duration before checking the resource's status again, avoiding constant reconciliation attempts.

## Usage of Predicates

For more complex use cases, [Predicates](#) can be used to fine-tune when your controller should trigger reconciliation. Predicates allow you to filter events based on specific conditions, such as changes to particular fields, labels, or annotations, ensuring that your controller only responds to relevant events and operates efficiently.

## Watching Secondary Resources Owned by the Controller

In Kubernetes controllers, it's common to manage both **Primary Resources** and **Secondary Resources**. A **Primary Resource** is the main resource that the controller is responsible for, while **Secondary Resources** are created and managed by the controller to support the **Primary Resource**.

In this section, we will explain how to manage **Secondary Resources** which are owned by the controller. This example shows how to:

- Set the [Owner Reference](#) between the primary resource (`Busybox`) and the secondary resource (`Deployment`) to ensure proper lifecycle management.
- Configure the controller to `Watch` the secondary resource using `Owns()` in `SetupWithManager()`. See that `Deployment` is owned by the `Busybox` controller because it will be created and managed by it.

## Setting the Owner Reference

To link the lifecycle of the secondary resource (`Deployment`) to the primary resource (`Busybox`), we need to set an [Owner Reference](#) on the secondary resource. This ensures that Kubernetes automatically handles cascading deletions: if the primary resource is deleted, the secondary resource will also be deleted.

Controller-runtime provides the [`controllerutil.SetControllerReference`](#) function, which you can use to set this relationship between the resources.

### Setting the Owner Reference

Below, we create the `Deployment` and set the Owner reference between the `Busybox` custom resource and the `Deployment` using `controllerutil.SetControllerReference()`.

```

// deploymentForBusybox returns a Deployment object for Busybox
func (r *BusyboxReconciler) deploymentForBusybox(busybox
*examplecomv1alpha1.Busybox) *appsv1.Deployment {
    replicas := busybox.Spec.Size

    dep := &appsv1.Deployment{
        ObjectMeta: metav1.ObjectMeta{
            Name:      busybox.Name,
            Namespace: busybox.Namespace,
        },
        Spec: appsv1.DeploymentSpec{
            Replicas: &replicas,
            Selector: &metav1.LabelSelector{
                MatchLabels: map[string]string{"app": busybox.Name},
            },
            Template: metav1.PodTemplateSpec{
                ObjectMeta: metav1.ObjectMeta{
                    Labels: map[string]string{"app": busybox.Name},
                },
                Spec: corev1.PodSpec{
                    Containers: []corev1.Container{
                        {
                            Name:  "busybox",
                            Image: "busybox:latest",
                        },
                    },
                },
            },
        },
    }

    // Set the ownerRef for the Deployment, ensuring that the Deployment
    // will be deleted when the Busybox CR is deleted.
    controllerutil.SetControllerReference(busybox, dep, r.Scheme)
    return dep
}

```

## Explanation

By setting the `OwnerReference`, if the `Busybox` resource is deleted, Kubernetes will automatically delete the `Deployment` as well. This also allows the controller to watch for changes in the `Deployment` and ensure that the desired state (such as the number of replicas) is maintained.

For example, if someone modifies the `Deployment` to change the replica count to 3, while the `Busybox` CR defines the desired state as 1 replica, the controller will reconcile this and ensure the `Deployment` is scaled back to 1 replica.

## Reconcile Function Example

```

// Reconcile handles the main reconciliation loop for Busybox and the
Deployment
func (r *BusyboxReconciler) Reconcile(ctx context.Context, req ctrl.Request)
(ctrl.Result, error) {
    log := log.FromContext(ctx)

    // Fetch the Busybox instance
    busybox := &examplecomv1alpha1.Busybox{}
    if err := r.Get(ctx, req.NamespacedName, busybox); err != nil {
        if apierrors.NotFound(err) {
            log.Info("Busybox resource not found. Ignoring since it must be
deleted")
            return ctrl.Result{}, nil
        }
        log.Error(err, "Failed to get Busybox")
        return ctrl.Result{}, err
    }

    // Check if the Deployment already exists, if not create a new one
    found := &appsv1.Deployment{}
    err := r.Get(ctx, types.NamespacedName{Namespace: busybox.Namespace, Name: busybox.Name}, found)
    if err != nil && apierrors.NotFound(err) {
        // Define a new Deployment
        dep := r.deploymentForBusybox(busybox)
        log.Info("Creating a new Deployment", "Deployment.Namespace",
dep.Namespace, "Deployment.Name", dep.Name)
        if err := r.Create(ctx, dep); err != nil {
            log.Error(err, "Failed to create new Deployment",
"Deployment.Namespace", dep.Namespace, "Deployment.Name", dep.Name)
            return ctrl.Result{}, err
        }
        // Requeue the request to ensure the Deployment is created
        return ctrl.Result{RequeueAfter: time.Minute}, nil
    } else if err != nil {
        log.Error(err, "Failed to get Deployment")
        return ctrl.Result{}, err
    }

    // Ensure the Deployment size matches the desired state
    size := busybox.Spec.Size
    if *found.Spec.Replicas != size {
        found.Spec.Replicas = &size
        if err := r.Update(ctx, found); err != nil {
            log.Error(err, "Failed to update Deployment size",
"Deployment.Namespace", found.Namespace, "Deployment.Name", found.Name)
            return ctrl.Result{}, err
        }
        // Requeue the request to ensure the correct state is achieved
        return ctrl.Result{Requeue: true}, nil
    }

    // Update Busybox status to reflect that the Deployment is available
}

```

```

busybox.Status.AvailableReplicas = found.Status.AvailableReplicas
if err := r.Status().Update(ctx, busybox); err != nil {
    log.Error(err, "Failed to update Busybox status")
    return ctrl.Result{}, err
}

return ctrl.Result{}, nil
}

```

## Watching Secondary Resources

To ensure that changes to the secondary resource (such as the `Deployment`) trigger a reconciliation of the primary resource (`Busybox`), we configure the controller to watch both resources.

The `Owns()` method allows you to specify secondary resources that the controller should monitor. This way, the controller will automatically reconcile the primary resource whenever the secondary resource changes (e.g., is updated or deleted).

### Example: Configuring `SetupWithManager` to Watch Secondary Resources

```

// SetupWithManager sets up the controller with the Manager.
// The controller will watch both the Busybox primary resource and the
// Deployment secondary resource.
func (r *BusyboxReconciler) SetupWithManager(mgr ctrl.Manager) error {
    return ctrl.NewControllerManagedBy(mgr).
        For(&examplecomv1alpha1.Busybox{}).           // Watch the primary resource
        Owns(&appsv1.Deployment{}).                  // Watch the secondary resource
        (Deployment)
        Complete(r)
}

```

## Ensuring the Right Permissions

Kubebuilder uses [markers](#) to define RBAC permissions required by the controller. In order for the controller to properly watch and manage both the primary (`Busybox`) and secondary (`Deployment`) resources, it must have the appropriate permissions granted; i.e. to `watch`, `get`, `list`, `create`, `update`, and `delete` permissions for those resources.

## Example: RBAC Markers

Before the `Reconcile` method, we need to define the appropriate RBAC markers. These markers will be used by [controller-gen](#) to generate the necessary roles and permissions when you run `make manifests`.

```
//  
+kubebuilder:rbac:groups=example.com,resources=busyboxes,verbs=get;list;watch;  
create;update;patch;delete  
//  
+kubebuilder:rbac:groups=apps,resources=deployments,verbs=get;list;watch;creat  
e;update;patch;delete
```

- The first marker gives the controller permission to manage the `Busybox` custom resource (the primary resource).
- The second marker grants the controller permission to manage `Deployment` resources (the secondary resource).

Note that we are granting permissions to `watch` the resources.

## Watching Secondary Resources that are NOT Owned

In some scenarios, a controller may need to watch and respond to changes in resources that it does not `Own`, meaning those resources are created and managed by another controller.

The following examples demonstrate how a controller can monitor and reconcile resources that it doesn't directly manage. This applies to any resource not `Owned` by the controller, including **Core Types** or **Custom Resources** managed by other controllers or projects and reconciled in separate processes.

For instance, consider two custom resources—`Busybox` and `BackupBusybox`. If changes to `Busybox` should trigger reconciliation in the `BackupBusybox` controller, we can configure the `BackupBusybox` controller to watch for updates in `Busybox`.

### Example: Watching a Non-Owned Busybox Resource to Reconcile BackupBusybox

Consider a controller that manages a custom resource `BackupBusybox` but also needs to monitor changes to `Busybox` resources across the cluster. We only want to trigger reconciliation when `Busybox` instances have the `Backup` feature enabled.

- **Why Watch Secondary Resources?**

- The `BackupBusybox` controller is not responsible for creating or owning `Busybox` resources, but changes in these resources (such as updates or deletions) directly affect the primary resource (`BackupBusybox`).
- By watching `Busybox` instances with a specific label, the controller ensures that the necessary actions (e.g., backups) are triggered only for the relevant resources.

### Configuration Example

Here's how to configure the `BackupBusyboxReconciler` to watch changes in the `Busybox` resource and trigger reconciliation for `BackupBusybox`:

```

// SetupWithManager sets up the controller with the Manager.
// The controller will watch both the BackupBusybox primary resource and the
Busybox resource.
func (r *BackupBusyboxReconciler) SetupWithManager(mgr ctrl.Manager) error {
    return ctrl.NewControllerManagedBy(mgr).
        For(&examplecomv1alpha1.BackupBusybox{}). // Watch the primary
resource (BackupBusybox)
        Watches(
            &source.Kind{Type: &examplecomv1alpha1.Busybox{}}, // Watch the
Busybox CR
            handler.EnqueueRequestsFromMapFunc(func(ctx context.Context, obj
client.Object) []reconcile.Request {
                // Trigger reconciliation for the BackupBusybox in the same
namespace
                return []reconcile.Request{
                    {
                        NamespacedName: types.NamespacedName{
                            Name:      "backupbusybox", // Reconcile the
associated BackupBusybox resource
                            Namespace: obj.GetNamespace(), // Use the
namespace of the changed Busybox
                        },
                    },
                },
            }),
        ). // Trigger reconciliation when the Busybox resource changes
    Complete(r)
}

```

Here's how we can configure the controller to filter and watch for changes to only those Busybox resources that have the specific label:

```
// SetupWithManager sets up the controller with the Manager.
// The controller will watch both the BackupBusybox primary resource and the
Busybox resource, filtering by a label.
func (r *BackupBusyboxReconciler) SetupWithManager(mgr ctrl.Manager) error {
    return ctrl.NewControllerManagedBy(mgr).
        For(&examplecomv1alpha1.BackupBusybox{}). // Watch the primary
resource (BackupBusybox)
        Watches(
            &source.Kind{Type: &examplecomv1alpha1.Busybox{}}, // Watch the
Busybox CR
            handler.EnqueueRequestsFromMapFunc(func(ctx context.Context, obj
client.Object) []reconcile.Request {
                // Check if the Busybox resource has the label 'backup-needed:
"true"
                if val, ok := obj.GetLabels()["backup-enable"]; ok && val ==
"true" {
                    // If the label is present and set to "true", trigger
reconciliation for BackupBusybox
                    return []reconcile.Request{
                        {
                            NamespacedName: types.NamespacedName{
                                Name:      "backupbusybox", // Reconcile the
associated BackupBusybox resource
                                Namespace: obj.GetNamespace(), // Use the
namespace of the changed Busybox
                            },
                            },
                        },
                    }
                }
                // If the label is not present or doesn't match, don't trigger
reconciliation
                return []reconcile.Request{}
            }),
        ). // Trigger reconciliation when the labeled Busybox resource
changes
        Complete(r)
}
```

## Using Predicates to Refine Watches

When working with controllers, it's often beneficial to use **Predicates** to filter events and control when the reconciliation loop should be triggered.

**Predicates** allow you to define conditions based on events (such as create, update, or delete) and resource fields (such as labels, annotations, or status fields). By using **Predicates**, you can refine your controller's behavior to respond only to specific changes in the resources it watches.

This can be especially useful when you want to refine which changes in resources should trigger a reconciliation. By using predicates, you avoid unnecessary reconciliations and can ensure that the controller only reacts to relevant changes.

## When to Use Predicates

**Predicates are useful when:**

- You want to ignore certain changes, such as updates that don't impact the fields your controller is concerned with.
- You want to trigger reconciliation only for resources with specific labels or annotations.
- You want to watch external resources and react only to specific changes.

## Example: Using Predicates to Filter Update Events

Let's say that we only want our `BackupBusybox` controller to reconcile when certain fields of the `Busybox` resource change, for example, when the `spec.size` field changes, but we want to ignore all other changes (such as status updates).

### Defining a Predicate

In the following example, we define a predicate that only allows reconciliation when there's a meaningful update to the `Busybox` resource:

```

import (
    "sigs.k8s.io/controller-runtime/pkg/predicate"
    "sigs.k8s.io/controller-runtime/pkg/event"
)

// Predicate to trigger reconciliation only on size changes in the Busybox
spec
updatePred := predicate.Funcs{
    // Only allow updates when the spec.size of the Busybox resource changes
    UpdateFunc: func(e event.UpdateEvent) bool {
        oldObj := e.ObjectOld.(*examplecomv1alpha1.Busybox)
        newObj := e.ObjectNew.(*examplecomv1alpha1.Busybox)

        // Trigger reconciliation only if the spec.size field has changed
        return newObj.Spec.Size != newObj.Spec.Size
    },
    // Allow create events
    CreateFunc: func(e event.CreateEvent) bool {
        return true
    },
    // Allow delete events
    DeleteFunc: func(e event.DeleteEvent) bool {
        return true
    },
    // Allow generic events (e.g., external triggers)
    GenericFunc: func(e event.GenericEvent) bool {
        return true
    },
}

```

## Explanation

In this example:

- The `UpdateFunc` returns `true` only if the `spec.size` field has changed between the old and new objects, meaning that all other changes in the `spec`, like annotations or other fields, will be ignored.
- `CreateFunc`, `DeleteFunc`, and `GenericFunc` return `true`, meaning that create, delete, and generic events are still processed, allowing reconciliation to happen for these event types.

This ensures that the controller reconciles only when the specific field `spec.size` is modified, while ignoring any other modifications in the `spec` that are irrelevant to your logic.

## Example: Using Predicates in Watches

Now, we apply this predicate in the `Watches()` method of the `BackupBusyboxReconciler` to trigger reconciliation only for relevant events:

```
// SetupWithManager sets up the controller with the Manager.  
// The controller will watch both the BackupBusybox primary resource and the  
Busybox resource, using predicates.  
func (r *BackupBusyboxReconciler) SetupWithManager(mgr ctrl.Manager) error {  
    return ctrl.NewControllerManagedBy(mgr).  
        For(&examplecomv1alpha1.BackupBusybox{}). // Watch the primary  
resource (BackupBusybox)  
        Watches(  
            &source.Kind{Type: &examplecomv1alpha1.Busybox{}}, // Watch the  
Busybox CR  
            handler.EnqueueRequestsFromMapFunc(func(ctx context.Context, obj  
client.Object) []reconcile.Request {  
                return []reconcile.Request{  
                    {  
                        NamespacedName: types.NamespacedName{  
                            Name: "backupbusybox", // Reconcile the  
associated BackupBusybox resource  
                            Namespace: obj.GetNamespace(), // Use the  
namespace of the changed Busybox  
                        },  
                    },  
                }  
            }),  
            builder.WithPredicates(updatePred), // Apply the predicate  
. // Trigger reconciliation when the Busybox resource changes (if it  
meets predicate conditions)  
            Complete(r)  
        }  
}
```

## Explanation

- `builder.WithPredicates(updatePred)` : This method applies the predicate, ensuring that reconciliation only occurs when the `spec.size` field in `Busybox` changes.
- **Other Events:** The controller will still trigger reconciliation on `Create`, `Delete`, and `Generic` events.

## Why Use Kind

- **Fast Setup:** Launch a multi-node Kubernetes cluster locally in under a minute.
- **Quick Teardown:** Dismantle the cluster in just a few seconds, streamlining your development workflow.
- **Local Image Usage:** Deploy your container images directly without the need to push to a remote registry.
- **Lightweight and Efficient:** Kind is a minimalistic Kubernetes distribution, making it perfect for local development and CI/CD pipelines.

This only cover the basics to use a kind cluster. You can find more details at [kind documentation](#).

## Installation

You can follow [this](#) to install `kind`.

## Create a Cluster

You can simply create a `kind` cluster by

```
kind create cluster
```

To customize your cluster, you can provide additional configuration. For example, the following is a sample `kind` configuration.

```
kind: Cluster
apiVersion: kind.x-k8s.io/v1alpha4
nodes:
  - role: control-plane
  - role: worker
  - role: worker
  - role: worker
```

Using the configuration above, run the following command will give you a k8s v1.17.2 cluster with 1 control-plane node and 3 worker nodes.

```
kind create cluster --config hack/kind-config.yaml --  
image=kindest/node:v1.17.2
```

You can use `--image` flag to specify the cluster version you want, e.g. `--image=kindest/node:v1.17.2`, the supported version are listed [here](#).

## Load Docker Image into the Cluster

When developing with a local kind cluster, loading docker images to the cluster is a very useful feature. You can avoid using a container registry.

```
kind load docker-image your-image-name:your-tag
```

See [Load a local image into a kind cluster](#) for more information.

## Delete a Cluster

```
kind delete cluster
```

## Webhook

Webhooks are requests for information sent in a blocking fashion. A web application implementing webhooks will send a HTTP request to other applications when a certain event happens.

In the kubernetes world, there are 3 kinds of webhooks: [admission webhook](#), [authorization webhook](#) and [CRD conversion webhook](#).

In [controller-runtime](#) libraries, we support admission webhooks and CRD conversion webhooks.

Kubernetes supports these dynamic admission webhooks as of version 1.9 (when the feature entered beta).

Kubernetes supports the conversion webhooks as of version 1.15 (when the feature entered beta).

## Admission Webhooks

Admission webhooks are HTTP callbacks that receive admission requests, process them and return admission responses.

Kubernetes provides the following types of admission webhooks:

- **Mutating Admission Webhook:** These can mutate the object while it's being created or updated, before it gets stored. It can be used to default fields in a resource requests, e.g. fields in Deployment that are not specified by the user. It can be used to inject sidecar containers.
- **Validating Admission Webhook:** These can validate the object while it's being created or updated, before it gets stored. It allows more complex validation than pure schema-based validation. e.g. cross-field validation and pod image whitelisting.

The apiserver by default doesn't authenticate itself to the webhooks. However, if you want to authenticate the clients, you can configure the apiserver to use basic auth, bearer token, or a cert to authenticate itself to the webhooks. You can find detailed steps [here](#).

### Execution Order

**Validating webhooks run after all mutating webhooks**, so you don't need to worry about another webhook changing an object after your validation has accepted it.

## Handling Resource Status in Admission Webhooks

### Modify status

You cannot modify or default the status of a resource using a mutating admission webhook. Set initial status in your controller when you first see a new object.

### Understanding Why:

#### Mutating Admission Webhooks

Mutating Admission Webhooks are primarily designed to intercept and modify requests concerning the creation, modification, or deletion of objects. Though they possess the

capability to modify an object's specification, directly altering its status isn't deemed a standard practice, often leading to unintended results.

```
// MutatingWebhookConfiguration allows for modification of objects.  
// However, direct modification of the status might result in unexpected  
behavior.  
type MutatingWebhookConfiguration struct {  
    ...  
}
```

## Setting Initial Status

For those diving into custom controllers for custom resources, it's imperative to grasp the concept of setting an initial status. This initialization typically takes place within the controller itself. The moment the controller identifies a new instance of its managed resource, primarily through a watch mechanism, it holds the authority to assign an initial status to that resource.

```
// Custom controller's reconcile function might look something like this:  
func (r *ReconcileMyResource) Reconcile(request reconcile.Request)  
(reconcile.Result, error) {  
    // ...  
    // Upon discovering a new instance, set the initial status  
    instance.Status = SomeInitialStatus  
    // ...  
}
```

## Status Subresource

Delving into Kubernetes custom resources, a clear demarcation exists between the spec (depicting the desired state) and the status (illustrating the observed state). Activating the /status subresource for a custom resource definition (CRD) bifurcates the status and spec, each assigned to its respective API endpoint. This separation ensures that changes introduced by users, such as modifying the spec, and system-driven updates, like status alterations, remain distinct. Leveraging a mutating webhook to tweak the status during a spec-modifying operation might not pan out as expected, courtesy of this isolation.

```
apiVersion: apiextensions.k8s.io/v1
kind: CustomResourceDefinition
metadata:
  name: myresources.mygroup.mydomain
spec:
  ...
  subresources:
    status: {} # Enables the /status subresource
```

## Conclusion

While certain edge scenarios might allow a mutating webhook to seamlessly modify the status, treading this path isn't a universally acclaimed or recommended strategy. Entrusting the controller logic with status updates remains the most advocated approach.

## Markers for Config/Code Generation

Kubebuilder makes use of a tool called [controller-gen](#) for generating utility code and Kubernetes YAML. This code and config generation is controlled by the presence of special “marker comments” in Go code.

Markers are single-line comments that start with a plus, followed by a marker name, optionally followed by some marker specific configuration:

```
// +kubebuilder:validation:Optional  
// +kubebuilder:validation:MaxItems=2  
//  
+kubebuilder:printcolumn:JSONPath=".status.replicas",name=Replicas,type=string
```

difference between `// +optional` and `// +kubebuilder:validation:Optional`

Controller-gen supports both (see the output of `controller-gen crd -www`).

`+kubebuilder:validation:Optional` and `+optional` can be applied to fields.

But `+kubebuilder:validation:Optional` can also be applied at the package level such that it applies to every field in the package.

If you’re using controller-gen only then they’re redundant, but if you’re using other generators or you want developers that need to build their own clients for your API, you’ll want to also include `+optional`.

The most reliable way in 1.x to get `+optional` is `omitempty`.

See each subsection for information about different types of code and YAML generation.

## Generating Code & Artifacts in Kubebuilder

Kubebuilder projects have two `make` targets that make use of controller-gen:

- `make manifests` generates Kubernetes object YAML, like [CustomResourceDefinitions](#), [WebhookConfigurations](#), and [RBAC roles](#).
- `make generate` generates code, like [runtime.Object/DeepCopy implementations](#).

See [Generating CRDs](#) for a comprehensive overview.

# Marker Syntax

Exact syntax is described in the [godocs for controller-tools](#).

In general, markers may either be:

- **Empty** (`+kubebuilder:validation:Optional`): empty markers are like boolean flags on the command line – just specifying them enables some behavior.
- **Anonymous** (`+kubebuilder:validation:MaxItems=2`): anonymous markers take a single value as their argument.
- **Multi-option**  
(`+kubebuilder:printcolumn:JSONPath=".status.replicas",name=Replicas,type=string`): multi-option markers take one or more named arguments. The first argument is separated from the name by a colon, and latter arguments are comma-separated. Order of arguments doesn't matter. Some arguments may be optional.

Marker arguments may be strings, ints, bools, slices, or maps thereof. Strings, ints, and bools follow their Go syntax:

```
// +kubebuilder:validation:ExclusiveMaximum=false
// +kubebuilder:validation:Format="date-time"
// +kubebuilder:validation:Maximum=42
```

For convenience, in simple cases the quotes may be omitted from strings, although this is not encouraged for anything other than single-word strings:

```
// +kubebuilder:validation>Type=string
```

Slices may be specified either by surrounding them with curly braces and separating with commas:

```
// +kubebuilder:webhooks:Enum={"crackers, Gromit, we forgot the
crackers!","not even wensleydale?"}
```

or, in simple cases, by separating with semicolons:

```
// +kubebuilder:validation:Enum=Wallace;Gromit;Chicken
```

Maps are specified with string keys and values of any type (effectively `map[string]interface{}``). A map is surrounded by curly braces (`{}`), each key and value is separated by a colon (`:`), and each key-value pair is separated by a comma:

```
// +kubebuilder:default={magic: {numero: 42, stringified: forty-two}}
```

## CRD Generation

These markers describe how to construct a custom resource definition from a series of Go types and packages. Generation of the actual validation schema is described by the [validation markers](#).

See [Generating CRDs](#) for examples.

### ► Show Detailed Argument Help

```
// +kubebuilder:deprecatedversion:warning=<string> on type
    marks this version as deprecated.

// +kubebuilder:metadata:annotations=<[]string>,labels=<[]string> on type
    ► configures the additional annotations or labels for this CRD.

// +kubebuilder:printcolumn
:JSONPath=<string>,description=<string>,format=<string>,name=<string>,priority=<int>,type=<string>
on type
    adds a column to "kubectl get" output for this CRD.

// +kubebuilder:resource
:categories=<[]string>,path=<string>,scope=<string>,shortName=<[]string>,singular=<string>
on type
    configures naming and scope for a CRD.

// +kubebuilder:selectablefield:JSONPath=<string> on type
    adds a field that may be used with field selectors.

// +kubebuilder:skipversion on type
    ► removes the particular version of the CRD from the CRDs spec.

// +kubebuilder:storageversion on type
    ► marks this version as the "storage version" for the CRD for conversion.

// +kubebuilder:subresource:scale
:selectorpath=<string>,specpath=<string>,statuspath=<string> on type
    enables the "/scale" subresource on a CRD.

// +kubebuilder:subresource:status on type
    enables the "/status" subresource on a CRD.

// +kubebuilder:unservedversion on type
    ► does not serve this version.

// +groupName:=<string> on package
    specifies the API group name for this package.

// +kubebuilder:skip on package
    don't consider this package as an API version.

// +versionName:=<string> on package
    overrides the API group version for this package (defaults to the package name).
```

## CRD Validation

These markers modify how the CRD validation schema is produced for the types and fields they modify. Each corresponds roughly to an OpenAPI/JSON schema option.

See [Generating CRDs](#) for examples.

### Understanding Marker Grouping in Documentation

Certain markers may seem duplicated. However, these markers are grouped based on their context of use — such as fields, types, or arrays. For instance, a marker like `+kubebuilder:validation:Enum` can be applied to individual fields or array items, and this flexibility is reflected in the documentation.

The grouping ensures clarity by showing how the same marker can be reused for different purposes.

- ▶ Show Detailed Argument Help

```
// +default:value=<any> on field
    ► Default sets the default value for this field.

// +kubebuilder:default:=<any> on field
    ► sets the default value for this field.

// +kubebuilder:example:=<any> on field
    ► sets the example value for this field.

// +kubebuilder:validation:EmbeddedResource on field
    ► EmbeddedResource marks a fields as an embedded resource with apiVersion, kind
    and metadata fields.

// +kubebuilder:validation:Enum:=<[]any> on field
    specifies that this (scalar) field is restricted to the *exact* values specified here.

// +kubebuilder:validation:ExclusiveMaximum:=<bool> on field
    indicates that the maximum is "up to" but not including that value.

// +kubebuilder:validation:ExclusiveMinimum:=<bool> on field
    indicates that the minimum is "up to" but not including that value.

// +kubebuilder:validation:Format:=<string> on field
    ► specifies additional "complex" formatting for this field.

// +kubebuilder:validation:MaxItems:=<int> on field
    specifies the maximum length for this list.

// +kubebuilder:validation:MaxLength:=<int> on field
    specifies the maximum length for this string.

// +kubebuilder:validation:MaxProperties:=<int> on field
    restricts the number of keys in an object

// +kubebuilder:validation:Maximum:=<> on field
    specifies the maximum numeric value that this field can have.

// +kubebuilder:validation:MinItems:=<int> on field
    specifies the minimum length for this list.

// +kubebuilder:validation:MinLength:=<int> on field
    specifies the minimum length for this string.

// +kubebuilder:validation:MinProperties:=<int> on field
    restricts the number of keys in an object

// +kubebuilder:validation:Minimum:=<> on field
    specifies the minimum numeric value that this field can have. Negative numbers are
    supported.

// +kubebuilder:validation:MultipleOf:=<> on field
    specifies that this field must have a numeric value that's a multiple of this one.

// +kubebuilder:validation:Optional on field
    specifies that this field is optional.

// +kubebuilder:validation:Pattern:=<string> on field
    specifies that this string must match the given regular expression.
```

```
// +kubebuilder:validation:Required on field
    specifies that this field is required.

// +kubebuilder:validation:Schemaless on field
    ▶ marks a field as being a schemaless object.

// +kubebuilder:validation>Type:=<string> on field
    ▶ overrides the type for this field (which defaults to the equivalent of the Go type).

// +kubebuilder:validation:UniqueItems:=<bool> on field
    specifies that all items in this list must be unique.

// +kubebuilder:validation:XEmbeddedResource on field
    ▶ EmbeddedResource marks a fields as an embedded resource with apiVersion, kind
    and metadata fields.

// +kubebuilder:validation:XIntOrString on field
    ▶ IntOrString marks a fields as an IntOrString.

// +kubebuilder:validation:XValidation
:fieldPath=<string>, message=<string>, messageExpression=<string>, reason=<string>, rule=<string>
on field
    ▶ marks a field as requiring a value for which a given

// +kubebuilder:validation:items:Enum:=</> on field
    for array items specifies that this (scalar) field is restricted to the *exact* values
    specified here.

// +kubebuilder:validation:items:ExclusiveMaximum:=<bool> on field
    for array items indicates that the maximum is "up to" but not including that value.

// +kubebuilder:validation:items:ExclusiveMinimum:=<bool> on field
    for array items indicates that the minimum is "up to" but not including that value.

// +kubebuilder:validation:items:Format:=<string> on field
    ▶ for array items specifies additional "complex" formatting for this field.

// +kubebuilder:validation:items:MaxItems:=<int> on field
    for array items specifies the maximum length for this list.

// +kubebuilder:validation:items:MaxLength:=<int> on field
    for array items specifies the maximum length for this string.

// +kubebuilder:validation:items:MaxProperties:=<int> on field
    for array items restricts the number of keys in an object

// +kubebuilder:validation:items:Maximum:=<> on field
    for array items specifies the maximum numeric value that this field can have.

// +kubebuilder:validation:items:MinItems:=<int> on field
    for array items specifies the minimum length for this list.

// +kubebuilder:validation:items:MinLength:=<int> on field
    for array items specifies the minimum length for this string.

// +kubebuilder:validation:items:MinProperties:=<int> on field
    for array items restricts the number of keys in an object

// +kubebuilder:validation:items:Minimum:=<> on field
```

for array items specifies the minimum numeric value that this field can have.  
Negative numbers are supported.

// **+kubebuilder:validation:items:MultipleOf:=<bool>** on field  
for array items specifies that this field must have a numeric value that's a multiple of this one.

// **+kubebuilder:validation:items:Pattern:=<string>** on field  
for array items specifies that this string must match the given regular expression.

// **+kubebuilder:validation:items:Type:=<string>** on field  
▶ for array items overrides the type for this field (which defaults to the equivalent of the Go type).

// **+kubebuilder:validation:items:UniqueItems:=<bool>** on field  
for array items specifies that all items in this list must be unique.

// **+kubebuilder:validation:items:XEmbeddedResource** on field  
▶ for array items EmbeddedResource marks a fields as an embedded resource with apiVersion, kind and metadata fields.

// **+kubebuilder:validation:items:XIntOrString** on field  
▶ for array items IntOrString marks a fields as an IntOrString.

// **+kubebuilder:validation:items:XValidation**  
`:fieldPath=<string>, message=<string>, messageExpression=<string>, reason=<string>, rule=<string>`  
on field  
▶ for array items marks a field as requiring a value for which a given

// **+nullable** on field  
▶ marks this field as allowing the "null" value.

// **+optional** on field  
specifies that this field is optional.

// **+required** on field  
specifies that this field is required.

// **+kubebuilder:validation:Enum:=<[]any>** on type  
specifies that this (scalar) field is restricted to the \*exact\* values specified here.

// **+kubebuilder:validation:ExclusiveMaximum:=<bool>** on type  
indicates that the maximum is "up to" but not including that value.

// **+kubebuilder:validation:ExclusiveMinimum:=<bool>** on type  
indicates that the minimum is "up to" but not including that value.

// **+kubebuilder:validation:Format:=<string>** on type  
▶ specifies additional "complex" formatting for this field.

// **+kubebuilder:validation:MaxItems:=<int>** on type  
specifies the maximum length for this list.

// **+kubebuilder:validation:MaxLength:=<int>** on type  
specifies the maximum length for this string.

```
// +kubebuilder:validation:MaxProperties:=<int> on type
    restricts the number of keys in an object

// +kubebuilder:validation:Maximum:=<> on type
    specifies the maximum numeric value that this field can have.

// +kubebuilder:validation:MinItems:=<int> on type
    specifies the minimum length for this list.

// +kubebuilder:validation:MinLength:=<int> on type
    specifies the minimum length for this string.

// +kubebuilder:validation:MinProperties:=<int> on type
    restricts the number of keys in an object

// +kubebuilder:validation:Minimum:=<> on type
    specifies the minimum numeric value that this field can have. Negative numbers are
    supported.

// +kubebuilder:validation:MultipleOf:=<> on type
    specifies that this field must have a numeric value that's a multiple of this one.

// +kubebuilder:validation:Pattern:=<string> on type
    specifies that this string must match the given regular expression.

// +kubebuilder:validation>Type:=<string> on type
    ▶ overrides the type for this field (which defaults to the equivalent of the Go type).

// +kubebuilder:validation:UniqueItems:=<bool> on type
    specifies that all items in this list must be unique.

// +kubebuilder:validation:XEmbeddedResource on type
    ▶ EmbeddedResource marks a fields as an embedded resource with apiVersion, kind
    and metadata fields.

// +kubebuilder:validation:XIntOrString on type
    ▶ IntOrString marks a fields as an IntOrString.

// +kubebuilder:validation:XValidation
:fieldPath=<string>, message=<string>, messageExpression=<string>, reason=<string>, rule=<string>
on type
    ▶ marks a field as requiring a value for which a given

// +kubebuilder:validation:items:Enum:=<[]any> on type
    for array items specifies that this (scalar) field is restricted to the *exact* values
    specified here.

// +kubebuilder:validation:items:ExclusiveMaximum:=<bool> on type
    for array items indicates that the maximum is "up to" but not including that value.

// +kubebuilder:validation:items:ExclusiveMinimum:=<bool> on type
    for array items indicates that the minimum is "up to" but not including that value.

// +kubebuilder:validation:items:Format:=<string> on type
    ▶ for array items specifies additional "complex" formatting for this field.

// +kubebuilder:validation:items:MaxItems:=<int> on type
    for array items specifies the maximum length for this list.
```

```
// +kubebuilder:validation:items:MaxLength:=<int> on type
    for array items specifies the maximum length for this string.

// +kubebuilder:validation:items:MaxProperties:=<int> on type
    for array items restricts the number of keys in an object

// +kubebuilder:validation:items:Maximum:=<> on type
    for array items specifies the maximum numeric value that this field can have.

// +kubebuilder:validation:items:MinItems:=<int> on type
    for array items specifies the minimum length for this list.

// +kubebuilder:validation:items:MinLength:=<int> on type
    for array items specifies the minimum length for this string.

// +kubebuilder:validation:items:MinProperties:=<int> on type
    for array items restricts the number of keys in an object

// +kubebuilder:validation:items:Minimum:=<> on type
    for array items specifies the minimum numeric value that this field can have.
    Negative numbers are supported.

// +kubebuilder:validation:items:MultipleOf:=<> on type
    for array items specifies that this field must have a numeric value that's a multiple of
    this one.

// +kubebuilder:validation:items:Pattern:=<string> on type
    for array items specifies that this string must match the given regular expression.

// +kubebuilder:validation:items:Type:=<string> on type
    ▶ for array items overrides the type for this field (which defaults to the equivalent of
    the Go type).

// +kubebuilder:validation:items:UniqueItems:=<bool> on type
    for array items specifies that all items in this list must be unique.

// +kubebuilder:validation:items:XEmbeddedResource on type
    ▶ for array items EmbeddedResource marks a fields as an embedded resource with
    apiVersion, kind and metadata fields.

// +kubebuilder:validation:items:XIntOrString on type
    ▶ for array items IntOrString marks a fields as an IntOrString.

// +kubebuilder:validation:items:XValidation
:fieldPath=<string>, message=<string>, messageExpression=<string>, reason=<string>, rule=<string>
on type
    ▶ for array items marks a field as requiring a value for which a given

// +kubebuilder:validation:Optional on package
    specifies that all fields in this package are optional by default.

// +kubebuilder:validation:Required on package
    specifies that all fields in this package are required by default.
```

## CRD Processing

These markers help control how the Kubernetes API server processes API requests involving your custom resources.

See [Generating CRDs](#) for examples.

### ► Show Detailed Argument Help

```
// +kubebuilder:pruning:PreserveUnknownFields  on field
    ▶ PreserveUnknownFields stops the apiserver from pruning fields which are not
        specified.

// +kubebuilder:validation:XPreserveUnknownFields  on field
    ▶ PreserveUnknownFields stops the apiserver from pruning fields which are not
        specified.

// +kubebuilder:validation:items:XPreserveUnknownFields  on field
    ▶ for array items PreserveUnknownFields stops the apiserver from pruning fields
        which are not specified.

// +listMapKey:=<string>  on field
    ▶ specifies the keys to map listTypes.

// +listType:=<string>  on field
    ▶ specifies the type of data-structure that the list

// +mapType:=<string>  on field
    ▶ specifies the level of atomicity of the map;

// +structType:=<string>  on field
    ▶ specifies the level of atomicity of the struct;

// +kubebuilder:pruning:PreserveUnknownFields  on type
    ▶ PreserveUnknownFields stops the apiserver from pruning fields which are not
        specified.

// +kubebuilder:validation:XPreserveUnknownFields  on type
    ▶ PreserveUnknownFields stops the apiserver from pruning fields which are not
        specified.

// +kubebuilder:validation:items:XPreserveUnknownFields  on type
    ▶ for array items PreserveUnknownFields stops the apiserver from pruning fields
        which are not specified.

// +listMapKey:=<string>  on type
    ▶ specifies the keys to map listTypes.

// +listType:=<string>  on type
    ▶ specifies the type of data-structure that the list

// +mapType:=<string>  on type
    ▶ specifies the level of atomicity of the map;
```

```
// +structType:=<string> on type
▶ specifies the level of atomicity of the struct;
```

## Webhook

These markers describe how [webhook configuration](#) is generated. Use these to keep the description of your webhooks close to the code that implements them.

### ► Show Detailed Argument Help

```
// +kubebuilder:webhook
:admissionReviewVersions=<[]string>,failurePolicy=<string>,groups=<[]string>,matchPolicy=
<string>,mutating=<bool>,name=<string>,path=<string>,reinvocationPolicy=<string>,resources=
<[]string>,serviceName=<string>,serviceNamespace=<string>,servicePort=<int>,sideEffects=
<string>,timeoutSeconds=<int>,url=<string>,verbs=<[]string>,versions=<[]string>,webhookVersions=
<[]string>
```

on package

- specifies how a webhook should be served.

```
// +kubebuilder:webhookconfiguration:mutating=<bool>,name=<string> on package
```

- specifies how a webhook should be served.

## Object/DeepCopy

These markers control when `DeepCopy` and `runtime.Object` implementation methods are generated.

### ► Show Detailed Argument Help

```
// +kubebuilder:object:generate:=<bool>  on type
    overrides enabling or disabling deepcopy generation for this type

// +kubebuilder:object:root:=<bool>  on type
    enables object interface implementation generation for this type

// +kubebuilder:object:generate:=<bool>  on package
    enables or disables object interface & deepcopy implementation generation for this
    package

// +k8s:deepcopy-gen:=<raw>  use kubebuilder:object:generate (on package)
    enables or disables object interface & deepcopy implementation generation for this
    package

// +k8s:deepcopy-gen:=<raw>  use kubebuilder:object:generate (on type)
    overrides enabling or disabling deepcopy generation for this type

// +k8s:deepcopy-gen:interfaces:=<string>  use kubebuilder:object:root (on type)
    enables object interface implementation generation for this type
```

## RBAC

These markers cause an [RBAC ClusterRole](#) to be generated. This allows you to describe the permissions that your controller requires alongside the code that makes use of those permissions.

- ▶ Show Detailed Argument Help

```
// +kubebuilder:rbac
:groups=<[]string>, namespace=<string>, resourceName=<[]string>, resources=<[]string>, urls=<[]string>, verbs=<[]string>
on package
```

specifies an RBAC rule to all access to some resources or non-resource URLs.

## Scaffold

The `+kubebuilder:scaffold` marker is a key part of the Kubebuilder scaffolding system. It marks locations in generated files where additional code will be injected as new resources (such as controllers, webhooks, or APIs) are scaffolded. This enables Kubebuilder to seamlessly integrate newly generated components into the project without affecting user-defined code.

-  ! If you delete or change the `+kubebuilder:scaffold` markers

The Kubebuilder CLI specifically looks for these markers in expected files during code generation. If the marker is moved or removed, the CLI will not be able to inject the necessary code, and the scaffolding process may fail or behave unexpectedly.

## How It Works

When you scaffold a new resource using the Kubebuilder CLI (e.g., `kubebuilder create api`), the CLI identifies `+kubebuilder:scaffold` markers in key locations and uses them as placeholders to insert the required imports and registration code.

## Example Usage in `main.go`

Here is how the `+kubebuilder:scaffold` marker is used in a typical `main.go` file. To illustrate how it works, consider the following command to create a new API:

```
kubebuilder create api --group crew --version v1 --kind Admiral --  
controller=true --resource=true
```

## To Add New Imports

The `+kubebuilder:scaffold:imports` marker allows the Kubebuilder CLI to inject additional imports, such as for new controllers or webhooks. When we create a new API, the CLI automatically adds the required import paths in this section.

For example, after creating the `Admiral` API in a single-group layout, the CLI will add `crewv1 "<repo-path>/api/v1"` to the imports:

```
import (
    "crypto/tls"
    "flag"
    "os"

    // Import all Kubernetes client auth plugins (e.g. Azure, GCP, OIDC, etc.)
    // to ensure that exec-entrypoint and run can make use of them.
    _ "k8s.io/client-go/plugin/pkg/client/auth"
    ...
    crewv1 "sigs.k8s.io/kubebuilder/testdata/project-v4/api/v1"
    // +kubebuilder:scaffold:imports
)
```

## To Register a New Scheme

The `+kubebuilder:scaffold:scheme` marker is used to register newly created API versions with the runtime scheme, ensuring the API types are recognized by the manager.

For example, after creating the Admiral API, the CLI will inject the following code into the `init()` function to register the scheme:

```
func init() {
    ...
    utilruntime.Must(crewv1.AddToScheme(scheme))
    // +kubebuilder:scaffold:scheme
}
```

## To Set Up a Controller

When we create a new controller (e.g., for Admiral), the Kubebuilder CLI injects the controller setup code into the manager using the `+kubebuilder:scaffold:builder` marker. This marker indicates where the setup code for new controllers should be added.

For example, after creating the `AdmiralReconciler`, the CLI will add the following code to register the controller with the manager:

```

if err = (&crewv1.AdmiralReconciler{
    Client: mgr.GetClient(),
    Scheme: mgr.GetScheme(),
}).SetupWithManager(mgr); err != nil {
    setupLog.Error(err, "unable to create controller", "controller",
    "Admiral")
    os.Exit(1)
}
// +kubebuilder:scaffold:builder

```

The `+kubebuilder:scaffold:builder` marker ensures that newly scaffolded controllers are properly registered with the manager, so that the controller can reconcile the resource.

## List of `+kubebuilder:scaffold` Markers

| Marker  | Usual Location                          |                                |
|---|---|--------------------------------|
| <code>+kubebuilder:scaffold:imports</code>                      | <code>main.go</code>                    | Marker                         |
| <code>+kubebuilder:scaffold:scheme</code>                       | <code>init() in main.go</code>          | Used                           |
| <code>+kubebuilder:scaffold:builder</code>                      | <code>main.go</code>                    | Marker                         |
| <code>+kubebuilder:scaffold:webhook</code>                      | <code>webhooks suite tests files</code> | Marker                         |
| <code>+kubebuilder:scaffold:crdkustomizeresource</code>         | <code>config/crd</code>                 | Marker                         |
| <code>+kubebuilder:scaffold:crdkustomizewebhookpatch</code>     | <code>config/crd</code>                 | Marker                         |
| <code>+kubebuilder:scaffold:crdkustomizecainjectionns</code>    | <code>config/default</code>             | Marker                         |
| <code>+kubebuilder:scaffold:crdkustomizecainjectionname</code>  | <code>config/default</code>             | Marker                         |
| <b>(No longer supported)</b>                                    |   |                                |
| <code>+kubebuilder:scaffold:crdkustomizecainjectionpatch</code> | <code>config/crd</code>                 | Marker<br>we<br>+k<br>an<br>+k |
| <code>+kubebuilder:scaffold:manifestskustomizesamples</code>    | <code>config/samples</code>             | Marker                         |

| Marker                                    | Usual Location |       |
|---|----------------|-------|
| +kubebuilder:scaffold:e2e-webhooks-checks | test/e2e       | Ad of |



! \*\*(No longer supported)\*\*  
`+kubebuilder:scaffold:crdkustomizecainjectionpatch`

If you find this marker in your code please:

### 1. Remove the CERTMANAGER Section from

`config/crd/kustomization.yaml`:

Delete the `CERTMANAGER` section to prevent unintended CA injection patches for CRDs. Ensure the following lines are removed or commented out:

```
# [CERTMANAGER] To enable cert-manager, uncomment all the
sections with [CERTMANAGER] prefix.
# patches here are for enabling the CA injection for each CRD
#- path: patches/cainjection_in_firstmates.yaml
# +kubebuilder:scaffold:crdkustomizecainjectionpatch
```

### 2. Ensure CA Injection Configuration in

`config/default/kustomization.yaml`:

Under the `[CERTMANAGER]` replacement in `config/default/kustomization.yaml`, add the following code for proper CA injection generation:

**NOTE:** You must ensure that the code contains the following target markers:

- +kubebuilder:scaffold:crdkustomizecainjectionns
- +kubebuilder:scaffold:crdkustomizecainjectionname

```
# - source: # Uncomment the following block if you have a
ConversionWebhook (--conversion)
#   kind: Certificate
#   group: cert-manager.io
#   version: v1
#   name: serving-cert # This name should match the one in
certificate.yaml
#   fieldPath: .metadata.namespace # Namespace of the
certificate CR
#   targets: # Do not remove or uncomment the following scaffold
marker; required to generate code for target CRD.
# +kubebuilder:scaffold:crdkustomizecainjectionns
# - source:
#   kind: Certificate
#   group: cert-manager.io
#   version: v1
#   name: serving-cert # This name should match the one in
certificate.yaml
#   fieldPath: .metadata.name
#   targets: # Do not remove or uncomment the following scaffold
marker; required to generate code for target CRD.
# +kubebuilder:scaffold:crdkustomizecainjectionname
```

### 3. Ensure Only Conversion Webhook Patches in config/crd/patches :

The `config/crd/patches` directory and the corresponding entries in `config/crd/kustomization.yaml` should only contain files for conversion webhooks. Previously, a bug caused the patch file to be generated for any webhook, but only patches for webhooks scaffolded with the `--conversion` option should be included.

For further guidance, you can refer to examples in the `testdata/` directory in the Kubebuilder repository.

---

**Alternatively:** You can use the `alpha generate` command to re-generate the project from scratch using the latest release available. Afterward, you can re-add only your code implementation on top to ensure your project includes all the latest bug fixes and enhancements.

---

## Creating Your Own Markers

If you are using Kubebuilder as a library to create [your own plugins](#) and extend its CLI functionalities, you have the flexibility to define and use your own markers. To implement your own markers, refer to the [kubebuilder/v4/pkg/machinery](#), which provides tools to create and manage markers effectively.

## controller-gen CLI

Kubebuilder makes use of a tool called [controller-gen](#) for generating utility code and Kubernetes YAML. This code and config generation is controlled by the presence of special “[marker comments](#)” in Go code.

controller-gen is built out of different “generators” (which specify what to generate) and “output rules” (which specify how and where to write the results).

Both are configured through command line options specified in [marker format](#).

For instance, the following command:

```
controller-gen paths=./... crd:trivialVersions=true rbac:roleName=controller-
perms output:crd:artifacts:config=config/crd/bases
```

generates CRDs and RBAC, and specifically stores the generated CRD YAML in `config/crd/bases`. For the RBAC, it uses the default output rules (`config/rbac`). It considers every package in the current directory tree (as per the normal rules of the go ... wildcard).

## Generators

Each different generator is configured through a CLI option. Multiple generators may be used in a single invocation of `controller-gen`.

### ► Show Detailed Argument Help

```
// +webhook:headerFile=<string>,year=<string> on package
    generates (partial) {Mutating,Validating}WebhookConfiguration objects.

// +schemapatch:generateEmbeddedObjectMeta=<bool>,manifests=<string>,maxDescLen=<int>
    on package
        ► patches existing CRDs with new schemata.

// +rbac:headerFile=<string>,roleName=<string>,year=<string> on package
    generates ClusterRole objects.

// +object:headerFile=<string>,year=<string> on package
        ► generates code containing DeepCopy, DeepCopyInto, and

// +crd
:allowDangerousTypes=<bool>,crdVersions=<[]string>
,deprecatedV1beta1CompatibilityPreserveUnknownFields=<bool>
,generateEmbeddedObjectMeta=<bool>,headerFile=<string>,ignoreUnexportedFields=<bool>
,maxDescLen=<int>,year=<string>
```

on package  
generates CustomResourceDefinition objects.

## Output Rules

Output rules configure how a given generator outputs its results. There is always one global “fallback” output rule (specified as `output:<rule>`), plus per-generator overrides (specified as `output:<generator>:<rule>`).

### Default Rules

When no fallback rule is specified manually, a set of default per-generator rules are used which result in YAML going to `config/<generator>`, and code staying where it belongs.

The default rules are equivalent to `output:`

`<generator>:artifacts:config=config/<generator>` for each generator.

When a “fallback” rule is specified, that’ll be used instead of the default rules.

For example, if you specify `crd rbac:roleName=controller-perms`  
`output:crd:stdout`, you’ll get CRDs on standard out, and rbac in a file in  
`config/rbac`. If you were to add in a global rule instead, like `crd`  
`rbac:roleName=controller-perms output:crd:stdout output:none`, you’d get CRDs  
to standard out, and everything else to /dev/null, because we’ve explicitly specified a  
fallback.

For brevity, the per-generator output rules (`output:<generator>:<rule>`) are omitted below. They are equivalent to the global fallback options listed here.

#### ► Show Detailed Argument Help

```
// +output:artifacts:code=<string>,config=<string> on package
  ► outputs artifacts to different locations, depending on
// +output:dir:=<string> on package
  ► outputs each artifact to the given directory, regardless
// +output:none on package
  skips outputting anything.
// +output:stdout on package
  ► outputs everything to standard-out, with no separation.
```

## Other Options

- ▶ Show Detailed Argument Help

// **+paths**:=⟨[]string⟩ on package

- ▶ represents paths and go-style path patterns to use as package roots.

## Enabling shell completion

The Kubebuilder completion script can be generated with the command `kubebuilder completion [bash|fish|powershell|zsh]`. Note that sourcing the completion script in your shell enables Kubebuilder autocompletion.

### Prerequisites for Bash

The completion Bash script depends on [bash-completion](#), which means that you have to install this software first (you can test if you have bash-completion already installed). Also, ensure that your Bash version is 4.1+.

- Once installed, go ahead and add the path `/usr/local/bin/bash` in the `/etc/shells`.

```
echo "/usr/local/bin/bash" > /etc/shells
```

- Make sure to use installed shell by current user.

```
chsh -s /usr/local/bin/bash
```

- Add following content in `/.bash_profile` or `~/.bashrc`

```
# kubebuilder completion
if [ -f /usr/local/share/bash-completion/bash_completion ]; then
  . /usr/local/share/bash-completion/bash_completion
fi
. <(kubebuilder completion bash)
```

- Restart terminal for the changes to be reflected or `source` the changed bash file.

### Zsh

Follow a similar protocol for `zsh` completion.

### Fish

```
source (kubebuilder completion fish | psub)
```

## Artifacts

To test your controllers, you will need to use the tarballs containing the required binaries:

```
./bin/k8s/
└── 1.25.0-darwin-amd64
    ├── etcd
    ├── kube-apiserver
    └── kubectl
```

These tarballs are released by [controller-tools](#), and you can find the list of available versions at: [envtest-releases.yaml](#).

When you run `make envtest` or `make test`, the necessary tarballs are downloaded and properly configured for your project.

### Setup ENV TEST tool

To learn more about the tooling used to configure ENVTEST, which is utilized in the `setup-envtest` target in the Makefile of projects built with Kubebuilder, see the [README](#) of its tooling. Additionally, you can find more information by reviewing the Kubebuilder [ENVTEST](#) documentation.



**!** **IMPORTANT:** Action Required: Ensure that you no longer use <https://storage.googleapis.com/kubebuilder-tools>

**Artifacts provided under <https://storage.googleapis.com/kubebuilder-tools> are deprecated and Kubebuilder maintainers are no longer able to support, build, or ensure the promotion of these artifacts.**

You will find the ENVTEST binaries available in the new location from k8s release 1.28 , see: <https://github.com/kubernetes-sigs/controller-tools/blob/main/envtest-releases.yaml>. Also, binaries to test your controllers after k8s 1.29.3 will no longer be found in the old location.

**New binaries are only promoted in the new location.**

**You should ensure that your projects are using the new location.** Please ensure you use `setup-envtest` from the controller-runtime release v0.19.0 to be able to download those. **This update is fully transparent for Kubebuilder users.**

The artefacts for [ENVTEST k8s 1.31](#) are exclusively available at: [Controller Tools Releases](#).

You can refer to the Makefile of the Kubebuilder scaffold and observe that the envtest setup is consistently aligned across all controller-runtime releases.

Starting from `release-0.19`, it is configured to automatically download the artefact from the correct location, **ensuring that kubebuilder users are not impacted**.

```
## Tool Binaries
...
ENVTEST ?= $(LOCALBIN)/setup-envtest
...

## Tool Versions
...
#ENVTEST_VERSION is the version of controller-runtime release branch
# to fetch the envtest setup script (i.e. release-0.20)
ENVTEST_VERSION ?= $(shell go list -m -f "{{ .Version }}" \
    sigs.k8s.io/controller-runtime | awk -F'[v.]' '{printf "release-
    %d.%d", $2, $3}')
#ENVTEST_K8S_VERSION is the version of Kubernetes to use for setting
# up ENVTEST binaries (i.e. 1.31)
ENVTEST_K8S_VERSION ?= $(shell go list -m -f "{{ .Version }}" \
    k8s.io/api | awk -F'[v.]' '{printf "1.%d", $3}')
...
.PHONY: setup-envtest
setup-envtest: envtest ## Download the binaries required for ENVTEST
in the local bin directory.
    @echo "Setting up envtest binaries for Kubernetes version
$(ENVTEST_K8S_VERSION)..."
    @$(ENVTEST) use $(ENVTEST_K8S_VERSION) --bin-dir $(LOCALBIN) -p
path || { \
    echo "Error: Failed to set up envtest binaries for version
$(ENVTEST_K8S_VERSION)."; \
    exit 1; \
}

.PHONY: envtest
envtest: $(ENVTEST) ## Download setup-envtest locally if necessary.
$(ENVTEST): $(LOCALBIN)
    $(call go-install-tool,$(ENVTEST),sig.s.k8s.io/controller-
    runtime/tools/setup-envtest,$(ENVTEST_VERSION))
```

## Platforms Supported

Kubebuilder produces solutions that by default can work on multiple platforms or specific ones, depending on how you build and configure your workloads. This guide aims to help you properly configure your projects according to your needs.

## Overview

To provide support on specific or multiple platforms, you must ensure that all images used in workloads are built to support the desired platforms. Note that they may not be the same as the platform where you develop your solutions and use KubeBuilder, but instead the platform(s) where your solution should run and be distributed. It is recommended to build solutions that work on multiple platforms so that your project works on any Kubernetes cluster regardless of the underlying operating system and architecture.

## How to define which platforms are supported

The following covers what you need to do to provide the support for one or more platforms or architectures.

### 1) Build workload images to provide the support for other platform(s)

The images used in workloads such as in your Pods/Deployments will need to provide the support for this other platform. You can inspect the images using a ManifestList of supported platforms using the command `docker manifest inspect`, i.e.:

```
$ docker manifest inspect myregistry/example/myimage:v0.0.1
{
  "schemaVersion": 2,
  "mediaType": "application/vnd.docker.distribution.manifest.list.v2+json",
  "manifests": [
    {
      "mediaType": "application/vnd.docker.distribution.manifest.v2+json",
      "size": 739,
      "digest":
"sha256:a274a1a2af811a1daf3fd6b48ff3d08feb757c2c3f3e98c59c7f85e550a99a32",
      "platform": {
        "architecture": "arm64",
        "os": "linux"
      }
    },
    {
      "mediaType": "application/vnd.docker.distribution.manifest.v2+json",
      "size": 739,
      "digest":
"sha256:d801c41875f12ffd8211ffffef2b3a3d1a301d99f149488d31f245676fa8bc5d9",
      "platform": {
        "architecture": "amd64",
        "os": "linux"
      }
    },
    {
      "mediaType": "application/vnd.docker.distribution.manifest.v2+json",
      "size": 739,
      "digest":
"sha256:f4423c8667edb5372fb0eafb6ec599bae8212e75b87f67da3286f0291b4c8732",
      "platform": {
        "architecture": "s390x",
        "os": "linux"
      }
    },
    {
      "mediaType": "application/vnd.docker.distribution.manifest.v2+json",
      "size": 739,
      "digest":
"sha256:621288f6573c012d7cf6642f6d9ab20dbaa35de3be6ac2c7a718257ec3aff333",
      "platform": {
        "architecture": "ppc64le",
        "os": "linux"
      }
    },
  ],
}
}
```

## 2) (Recommended as a Best Practice) Ensure that node affinity expressions are set to match the supported platforms

Kubernetes provides a mechanism called [nodeAffinity](#) which can be used to limit the possible node targets where a pod can be scheduled. This is especially important to ensure correct scheduling behavior in clusters with nodes that span across multiple platforms (i.e. heterogeneous clusters).

### Kubernetes manifest example

```
affinity:  
  nodeAffinity:  
    requiredDuringSchedulingIgnoredDuringExecution:  
      nodeSelectorTerms:  
        - matchExpressions:  
            - key: kubernetes.io/arch  
              operator: In  
              values:  
                - amd64  
                - arm64  
                - ppc64le  
                - s390x  
            - key: kubernetes.io/os  
              operator: In  
              values:  
                - linux
```

### Golang Example

```
Template: corev1.PodTemplateSpec{
    ...
    Spec: corev1.PodSpec{
        Affinity: &corev1.Affinity{
            NodeAffinity: &corev1.NodeAffinity{
                RequiredDuringSchedulingIgnoredDuringExecution:
&corev1.NodeSelector{
                NodeSelectorTerms: []corev1.NodeSelectorTerm{
                    {
                        MatchExpressions:
[]corev1.NodeSelectorRequirement{
                        {
                            Key:      "kubernetes.io/arch",
                            Operator: "In",
                            Values:   []string{"amd64"},
                        },
                        {
                            Key:      "kubernetes.io/os",
                            Operator: "In",
                            Values:   []string{"linux"},
                        },
                        },
                    },
                },
            },
        },
        SecurityContext: &corev1.PodSecurityContext{
            ...
        },
        Containers: []corev1.Container{
            ...
        },
    },
},
```

#### Example(s)

You can look for some code examples by checking the code which is generated via the Deploy Image plugin. ([More info](#))

## Producing projects that support multiple platforms

You can use `docker buildx` to cross-compile via emulation (QEMU) to build the manager image. See that projects scaffold with the latest versions of Kubebuilder have the Makefile target `docker-buildx`.

## Example of Usage

```
$ make docker-buildx IMG=myregistry/myoperator:v0.0.1
```

Note that you need to ensure that all images and workloads required and used by your project will provide the same support as recommended above, and that you properly configure the [nodeAffinity](#) for all your workloads. Therefore, ensure that you uncomment the following code in the `config/manager/manager.yaml` file

```
# TODO(user): Uncomment the following code to configure the nodeAffinity
# expression
# according to the platforms which are supported by your solution.
# It is considered best practice to support multiple architectures. You can
# build your manager image using the makefile target docker-buildx.
# affinity:
#   nodeAffinity:
#     requiredDuringSchedulingIgnoredDuringExecution:
#       nodeSelectorTerms:
#         - matchExpressions:
#             - key: kubernetes.io/arch
#               operator: In
#               values:
#                 - amd64
#                 - arm64
#                 - ppc64le
#                 - s390x
#             - key: kubernetes.io/os
#               operator: In
#               values:
#                 - linux
```

### Building images for releases

You will probably want to automate the releases of your projects to ensure that the images are always built for the same platforms. Note that Goreleaser also supports [docker buildx](#). See its [documentation](#) for more detail.

Also, you may want to configure GitHub Actions, Prow jobs, or any other solution that you use to build images to provide multi-platform support. Note that you can also use other options like `docker manifest create` to customize your solutions to achieve the same goals with other tools.

By using Docker and the target provided by default you should NOT change the Dockerfile to use any specific GOOS and GOARCH to build the manager binary. However, if you are looking for to customize the default scaffold and create your own

implementations you might want to give a look in the Golang [doc](#) to know its available options.

## Which (workload) images are created by default?

Projects created with the Kubebuilder CLI have two workloads which are:

### Manager

The container to run the manager implementation is configured in the `config/manager/manager.yaml` file. This image is built with the Dockerfile file scaffolded by default and contains the binary of the project which will be built via the command `go build -a -o manager main.go`.

Note that when you run `make docker-build` OR `make docker-build IMG=myregistry/myprojectname:<tag>` an image will be built from the client host (local environment) and produce an image for the client os/arch, which is commonly linux/amd64 or linux/arm64.

#### Mac Os

If you are running from a Mac Os environment then, Docker also will consider it as linux/\$arch. Be aware that when, for example, is running Kind on a Mac OS operational system the nodes will end up labeled with `kubernetes.io/os=linux`

## Monitoring Performance with Pprof

Pprof, a Go profiling tool, helps identify performance bottlenecks in areas like CPU and memory usage. It's integrated with the controller-runtime library's HTTP server, enabling profiling via HTTP endpoints. You can visualize the data using go tool pprof. Since [Pprof](#) is built into controller-runtime, no separate installation is needed. [Manager options](#) make it easy to enable pprof and gather runtime metrics to optimize controller performance.

### Not Recommended for Production

While [Pprof](#) is an excellent tool for profiling and debugging, it is not recommended to leave it enabled in production environments. The primary reasons are:

1. **Security Risk:** The profiling endpoints expose detailed information about your application's performance and resource usage, which could be exploited if accessed by unauthorized users.
2. **Overhead:** Running profiling can introduce performance overhead, mainly CPU usage, especially under heavy load, potentially impacting production workloads.

## How to use Pprof?

### 1. Enabling Pprof

In your `cmd/main.go` file, add the field:

```
mgr, err := ctrl.NewManager(ctrl.GetConfigOrDie(), ctrl.Options{
    ...
    // PprofBindAddress is the TCP address that the controller should bind
    to
    // for serving pprof. Specify the manager address and the port that
    should be bind.
    PprofBindAddress:      ":8082",
    ...
})
```

### 2. Test It Out

After enabling [Pprof](#), you need to build and deploy your controller to test it out. Follow the steps in the [Quick Start guide](#) to run your project locally or on a cluster.

Then, you can apply your CRs/samples in order to monitor the performance of its controllers.

### 3. Exporting the data

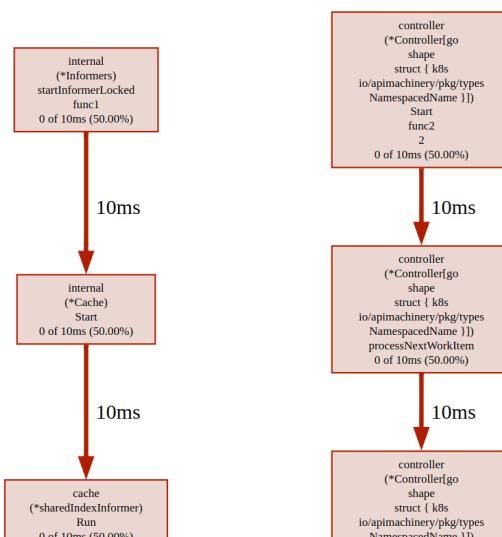
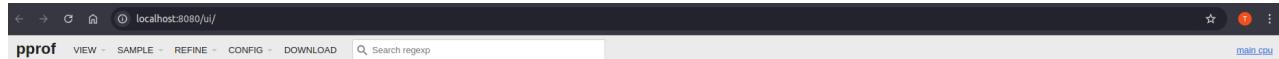
Using `curl`, export the profiling statistics to a file like this:

```
# Note that we are using the bind host and port configured via the
# Manager Options in the cmd/main.go
curl -s "http://127.0.0.1:8082/debug/pprof/profile" > ./cpu-profile.out
```

### 4. Visualizing the results on Browser

```
# Go tool will open a session on port 8080.
# You can change this as per your own need.
go tool pprof -http=:8080 ./cpu-profile.out
```

Visualization results will vary depending on the deployed workload, and the Controller's behavior. However, you'll see the result on your browser similar to this one:



## Understanding and Setting Scopes for Managers (Operators) and CRDs

This section covers the configuration of the operational and resource scopes within a Kubebuilder project. Managers (“Operators”) in Kubernetes can be scoped to either specific namespaces or the entire cluster, influencing how resources are watched and managed.

Additionally, CustomResourceDefinitions (CRDs) can be defined to be either namespace-scoped or cluster-scoped, affecting their availability across the cluster.

## Configuring Manager Scope

Managers can operate under different scopes depending on the resources they need to handle:

### (Default) Watching All Namespaces

By default, if no namespace is specified, the manager will observe all namespaces. This is configured as follows:

```
mgr, err := ctrl.NewManager(ctrl.GetConfigOrDie(), ctrl.Options{
...
})
```

### Watching a Single Namespace

To constrain the manager to monitor resources within a specific namespace, set the Namespace option:

```
mgr, err := ctrl.NewManager(ctrl.GetConfigOrDie(), ctrl.Options{
...
    Cache: cache.Options{
        DefaultNamespaces: map[string]cache.Config{"operator-namespace": cache.Config{}},
    },
})
```

## Watching Multiple Namespaces

A manager can also be configured to watch a specified set of namespaces using [Cache Config](#):

```
mgr, err := ctrl.NewManager(ctrl.GetConfigOrDie(), ctrl.Options{
    ...
    Cache: cache.Options{
        DefaultNamespaces: map[string]cache.Config{
            "operator-namespace1": cache.Config{},
            "operator-namespace2": cache.Config{},
        },
    },
})
```

## Configuring CRD Scope

The scope of CRDs determines their visibility either within specific namespaces or across the entire cluster.

### Namespace-scoped CRDs

Namespace-scoped CRDs are suitable when resources need to be isolated to specific namespaces. This setting helps manage resources related to particular teams or applications. However, it is important to note that due to the unique definition of CRDs (Custom Resource Definitions) in Kubernetes, testing a new version of a CRD is not straightforward. Proper versioning and conversion strategies need to be implemented (example in our [kubebuilder tutorial](#)), and coordination is required to manage which manager instance handles the conversion (see the official [kubernetes documentation](#) about this). Additionally, the namespace scope must be taken into account for mutating and validating webhook configurations to ensure they are correctly applied within the intended scope. This facilitates a more controlled and phased rollout strategy.

### Cluster-scoped CRDs

For resources that need to be accessible and manageable across the entire cluster, such as shared configurations or global resources, cluster-scoped CRDs are used.

## Configuring CRDs Scopes

### When the API is created

The scope of a CRD is defined when generating its manifest. Kubebuilder facilitates this through its API creation command.

By default, APIs are created with CRD scope as namespaced. However, for cluster-wide you use `--namespaced=false`, i.e.:

```
kubebuilder create api --group cache --version v1alpha1 --kind Memcached --  
resource=true --controller=true --namespaced=false
```

This command generates the CRD with the Cluster scope, meaning it will be accessible and manageable across all namespaces in the cluster.

### By updating existing APIs

After you create an API you are still able to change the scope. For example, to configure a CRD to be cluster-wide, add the `+kubebuilder:resource:scope=Cluster` marker above the API type definition in your Go file. Here is an example:

```
//+kubebuilder:object:root=true  
//+kubebuilder:subresource:status  
//+kubebuilder:resource:scope=Cluster,shortName=mc  
  
...
```

After setting the desired scope with markers, run `make manifests` to generate the files. This command invokes `controller-gen` to generate the CRD manifests according to the markers specified in your Go files.

The generated manifests will then correctly reflect the scope as either Cluster or Namespaced without needing manual adjustment in the YAML files.

## Sub-Module Layouts

This part describes how to modify a scaffolded project for use with multiple `go.mod` files for APIs and Controllers.

Sub-Module Layouts (in a way you could call them a special form of [Monorepo's](#)) are a special use case and can help in scenarios that involve reuse of APIs without introducing indirect dependencies that should not be available in the project consuming the API externally.

### Using External Resources/APIs

If you are looking to do operations and reconcile via a controller a Type(CRD) which are owned by another project or By Kubernetes API then, please see [Using an external Resources/API](#) for more info.

## Overview

Separate `go.mod` modules for APIs and Controllers can help for the following cases:

- There is an enterprise version of an operator available that wants to reuse APIs from the Community Version
- There are many (possibly external) modules depending on the API and you want to have a more strict separation of transitive dependencies
- If you want to reduce impact of transitive dependencies on your API being included in other projects
- If you are looking to separately manage the lifecycle of your API release process from your controller release process.
- If you are looking to modularize your codebase without splitting your code between multiple repositories.

They introduce however multiple caveats into typical projects which is one of the main factors that makes them hard to recommend in a generic use-case or plugin:

- Multiple `go.mod` modules are not recommended as a go best practice and [multiple modules are mostly discouraged](#)
- There is always the possibility to extract your APIs into a new repository and arguably also have more control over the release process in a project spanning multiple repos relying on the same API types.

- It requires at least one [replace directive](#) either through `go.work` which is at least 2 more files plus an environment variable for build environments without `GO_WORK` or through `go.mod` replace, which has to be manually dropped and added for every release.

### ! Implications on Maintenance efforts

When deciding to deviate from the standard `kubebuilder` `PROJECT` setup or the extended layouts offered by its plugins, it can result in increased maintenance overhead as there can be breaking changes in upstream that could break with the custom module structure described here.

Splitting your codebase to multiple repos and/or multiple modules incurs costs that will grow over time. You'll need to define clear version dependencies between your own modules, do phased upgrades carefully, etc. Especially for small-to-medium projects, one repo and one module is the best way to go.

Bear in mind, that it is not recommended to deviate from the proposed layout unless you know what you are doing. You may also lose the ability to use some of the CLI features and helpers. For further information on the project layout, see the doc [What's in a basic project?](#)

## Adjusting your Project

For a proper Sub-Module layout, we will use the generated APIs as a starting point.

For the steps below, we will assume you created your project in your `GOPATH` with

```
kubebuilder init
```

and created an API & controller with

```
kubebuilder create api --group operator --version v1alpha1 --kind Sample --resource --controller --make
```

## Creating a second module for your API

Now that we have a base layout in place, we will enable you for multiple modules.

1. Navigate to `api/v1alpha1`
2. Run `go mod init` to create a new submodule
3. Run `go mod tidy` to resolve the dependencies

Your api go.mod file could now look like this:

```
module YOUR_GO_PATH/test-operator/api/v1alpha1

go 1.21.0

require (
    k8s.io/apimachinery v0.28.4
    sigs.k8s.io/controller-runtime v0.16.3
)

require (
    github.com/go-logr/logr v1.2.4 // indirect
    github.com/gogo/protobuf v1.3.2 // indirect
    github.com/google/gofuzz v1.2.0 // indirect
    github.com/json-iterator/go v1.1.12 // indirect
    github.com/modern-go/concurrent v0.0.0-20180306012644-bacd9c7ef1dd //
indirect
    github.com/modern-go/reflect2 v1.0.2 // indirect
    golang.org/x/net v0.17.0 // indirect
    golang.org/x/text v0.13.0 // indirect
    gopkg.in/inf.v0 v0.9.1 // indirect
    gopkg.in/yaml.v2 v2.4.0 // indirect
    k8s.io/klog/v2 v2.100.1 // indirect
    k8s.io/utils v0.0.0-20230406110748-d93618cff8a2 // indirect
    sigs.k8s.io/json v0.0.0-20221116044647-bc3834ca7abd // indirect
    sigs.k8s.io/structured-merge-diff/v4 v4.2.3 // indirect
)
```

As you can see it only includes apimachinery and controller-runtime as dependencies and any dependencies you have declared in your controller are not taken over into the indirect imports.

## Using replace directives for development

When trying to resolve your main module in the root folder of the operator, you will notice an error if you use a VCS path:

```
go mod tidy
go: finding module for package YOUR_GO_PATH/test-operator/api/v1alpha1
YOUR_GO_PATH/test-operator imports
  YOUR_GO_PATH/test-operator/api/v1alpha1: cannot find module providing
package YOUR_GO_PATH/test-operator/api/v1alpha1: module YOUR_GO_PATH/test-
operator/api/v1alpha1: git ls-remote -q origin in LOCALVCSPATH: exit status
128:
  remote: Repository not found.
  fatal: repository 'https://YOUR_GO_PATH/test-operator/' not found
```

The reason for this is that you may have not pushed your modules into the VCS yet and resolving the main module will fail as it can no longer directly access the API types as a package but only as a module.

To solve this issue, we will have to tell the go tooling to properly replace the API module with a local reference to your path.

You can do this with 2 different approaches: go modules and go workspaces.

## Using go modules

For go modules, you will edit the main `go.mod` file of your project and issue a replace directive.

You can do this by editing the `go.mod` with ``

```
go mod edit -require YOUR_GO_PATH/test-operator/api/v1alpha1@v0.0.0 # Only if
you didn't already resolve the module
go mod edit -replace YOUR_GO_PATH/test-
operator/api/v1alpha1@v0.0.0=./api/v1alpha1
go mod tidy
```

Note that we used the placeholder version `v0.0.0` of the API Module. In case you already released your API module once, you can use the real version as well. However this will only work if the API Module is already available in the VCS.

### Implications on controller releases

Since the main `go.mod` file now has a replace directive, it is important to drop it again before releasing your controller module. To achieve this you can simply run

```
go mod edit -dropreplace YOUR_GO_PATH/test-operator/api/v1alpha1
go mod tidy
```

## Using go workspaces

For go workspaces, you will not edit the `go.mod` files yourself, but rely on the workspace support in go.

To initialize a workspace for your project, run `go work init` in the project root.

Now let us include both modules in our workspace:

```
go work use . # This includes the main module with the controller  
go work use api/v1alpha1 # This is the API submodule  
go work sync
```

This will lead to commands such as `go run` or `go build` to respect the workspace and make sure that local resolution is used.

You will be able to work with this locally without having to build your module.

When using `go.work` files, it is recommended to not commit them into the repository and add them to `.gitignore`.

```
go.work  
go.work.sum
```

When releasing with a present `go.work` file, make sure to set the environment variable `GOWORK=off` (verifiable with `go env GOWORK`) to make sure the release process does not get impeded by a potentially committed `go.work` file.

## Adjusting the Dockerfile

When building your controller image, kubebuilder by default is not able to work with multiple modules. You will have to manually add the new API module into the download of dependencies:

```

# Build the manager binary
FROM docker.io/golang:1.20 as builder
ARG TARGETOS
ARG TARGETARCH

WORKDIR /workspace
# Copy the Go Modules manifests
COPY go.mod go.mod
COPY go.sum go.sum
# Copy the Go Sub-Module manifests
COPY api/v1alpha1/go.mod api/go.mod
COPY api/v1alpha1/go.sum api/go.sum
# cache deps before building and copying source so that we don't need to re-
download as much
# and so that source changes don't invalidate our downloaded layer
RUN go mod download

# Copy the go source
COPY cmd/main.go cmd/main.go
COPY api/ api/
COPY internal/controller/ internal/controller/

# Build
# the GOARCH has not a default value to allow the binary be built according to
the host where the command
# was called. For example, if we call make docker-build in a local env which
has the Apple Silicon M1 SO
# the docker BUILDPLATFORM arg will be linux/arm64 when for Apple x86 it will
be linux/amd64. Therefore,
# by leaving it empty we can ensure that the container and binary shipped on
it will have the same platform.
RUN CGO_ENABLED=0 GOOS=${TARGETOS:-linux} GOARCH=${TARGETARCH} go build -a -o
manager cmd/main.go

# Use distroless as minimal base image to package the manager binary
# Refer to https://github.com/GoogleContainerTools/distroless for more details
FROM gcr.io/distroless/static:nonroot
WORKDIR /
COPY --from=builder /workspace/manager .
USER 65532:65532

ENTRYPOINT ["/manager"]

```

## Creating a new API and controller release

Because you adjusted the default layout, before releasing your first version of your operator, make sure to [familiarize yourself with mono-repo/multi-module releases](#) with multiple `go.mod` files in different subdirectories.

Assuming a single API was created, the release process could look like this:

```
git commit  
git tag v1.0.0 # this is your main module release  
git tag api/v1.0.0 # this is your api release  
go mod edit -require YOUR_GO_PATH/test-operator/api@v1.0.0 # now we depend on  
the api module in the main module  
go mod edit -dropreplace YOUR_GO_PATH/test-operator/api/v1alpha1 # this will  
drop the replace directive for local development in case you use go modules,  
meaning the sources from the VCS will be used instead of the ones in your  
monorepo checked out locally.  
git push origin main v1.0.0 api/v1.0.0
```

After this, your modules will be available in VCS and you do not need a local replacement anymore. However if you're making local changes, make sure to adopt your behavior with `replace` directives accordingly.

## Reusing your extracted API module

Whenever you want to reuse your API module with a separate kubebuilder, we will assume you follow the guide for [using an external Type](#). When you get to the step `Edit the API files` simply import the dependency with

```
go get YOUR_GO_PATH/test-operator/api@v1.0.0
```

and then use it as explained in the guide.

## Using External Resources

In some cases, your project may need to work with resources that aren't defined by your own APIs. These external resources fall into two main categories:

- **Core Types**: API types defined by Kubernetes itself, such as `Pods`, `Services`, and `Deployments`.
- **External Types**: API types defined in other projects, such as CRDs defined by another solution.

## Managing External Types

### Creating a Controller for External Types

To create a controller for an external type without scaffolding a resource, use the `create api` command with the `--resource=false` option and specify the path to the external API type using the `--external-api-path` and `--external-api-domain` flag options. This generates a controller for types defined outside your project, such as CRDs managed by other Operators.

The command looks like this:

```
kubebuilder create api --group <theirgroup> --version <theirversion> --kind <theirKind> --controller --resource=false --external-api-path=<their Golang path import> --external-api-domain=<theirdomain>
```

- `--external-api-path`: Provide the Go import path where the external types are defined.
- `--external-api-domain`: Provide the domain for the external types. This value will be used to generate RBAC permissions and create the QualifiedGroup, such as -  
`apiGroups: <group>.<domain>`

For example, if you're managing Certificates from Cert Manager:

```
kubebuilder create api --group certmanager --version v1 --kind Certificate --controller=true --resource=false --external-api-path=github.com/cert-manager/cert-manager/pkg/apis/certmanager/v1 --external-api-domain=io
```

See the RBAC markers generated for this:

```
// +kubebuilder:rbac:groups=cert-
manager.io,resources=certificates,verbs=get;list;watch;create;update;patch;del
ete
// +kubebuilder:rbac:groups=cert-
manager.io,resources=certificates/status,verbs=get;update;patch
// +kubebuilder:rbac:groups=cert-
manager.io,resources=certificates/finalizers,verbs=update
```

Also, the RBAC role:

```
- apiGroups:
  - cert-manager.io
    resources:
      - certificates
    verbs:
      - create
      - delete
      - get
      - list
      - patch
      - update
      - watch
- apiGroups:
  - cert-manager.io
    resources:
      - certificates/finalizers
    verbs:
      - update
- apiGroups:
  - cert-manager.io
    resources:
      - certificates/status
    verbs:
      - get
      - patch
      - update
```

This scaffolds a controller for the external type but skips creating new resource definitions since the type is defined in an external project.

## Creating a Webhook to Manage an External Type

Following an example:

```
kubebuilder create webhook --group certmanager --version v1 --kind Issuer --
defaulting --programmatic-validation --external-api-path=github.com/cert-
manager/cert-manager/pkg/apis/certmanager/v1 --external-api-domain=cert-
manager.io
```

# Managing Core Types

Core Kubernetes API types, such as `Pods`, `Services`, and `Deployments`, are predefined by Kubernetes. To create a controller for these core types without scaffolding the resource, use the Kubernetes group name described in the following table and specify the version and kind.

| Group                 | K8s API Group                |
|-----------------------|------------------------------|
| admission             | k8s.io/admission             |
| admissionregistration | k8s.io/admissionregistration |
| apps                  | apps                         |
| auditregistration     | k8s.io/auditregistration     |
| apiextensions         | k8s.io/apiextensions         |
| authentication        | k8s.io/authentication        |
| authorization         | k8s.io/authorization         |
| autoscaling           | autoscaling                  |
| batch                 | batch                        |
| certificates          | k8s.io/certificates          |
| coordination          | k8s.io/coordination          |
| core                  | core                         |
| events                | k8s.io/events                |
| extensions            | extensions                   |
| imagepolicy           | k8s.io/imagepolicy           |
| networking            | k8s.io/networking            |
| node                  | k8s.io/node                  |
| metrics               | k8s.io/metrics               |
| policy                | policy                       |
| rbac.authorization    | k8s.io/rbac.authorization    |
| scheduling            | k8s.io/scheduling            |
| setting               | k8s.io/setting               |
| storage               | k8s.io/storage               |

The command to create a controller to manage `Pods` looks like this:

```
kubebuilder create api --group core --version v1 --kind Pod --controller=true  
--resource=false
```

For instance, to create a controller to manage Deployment the command would be like:

```
create api --group apps --version v1 --kind Deployment --controller=true --  
resource=false
```

See the RBAC markers generated for this:

```
//  
+kubebuilder:rbac:groups=apps,resources=deployments,verbs=get;list;creat  
e;update;patch;delete  
//  
+kubebuilder:rbac:groups=apps,resources=deployments/status,verbs=get;update;pa  
tch  
// +kubebuilder:rbac:groups=apps,resources=deployments/finalizers,verbs=update
```

Also, the RBAC for the above markers:

```
- apiGroups:  
  - apps  
resources:  
  - deployments  
verbs:  
  - create  
  - delete  
  - get  
  - list  
  - patch  
  - update  
  - watch  
- apiGroups:  
  - apps  
resources:  
  - deployments/finalizers  
verbs:  
  - update  
- apiGroups:  
  - apps  
resources:  
  - deployments/status  
verbs:  
  - get  
  - patch  
  - update
```

This scaffolds a controller for the Core type `corev1.Pod` but skips creating new resource definitions since the type is already defined in the Kubernetes API.

## Creating a Webhook to Manage a Core Type

You will run the command with the Core Type data, just as you would for controllers. See an example:

```
kubebuilder create webhook --group core --version v1 --kind Pod --  
programmatic-validation
```

## Configuring envtest for integration tests

The [controller-runtime/pkg/envtest](#) Go library helps write integration tests for your controllers by setting up and starting an instance of etcd and the Kubernetes API server, without kubelet, controller-manager or other components.

## Installation

Installing the binaries is as simple as running `make envtest`. `envtest` will download the Kubernetes API server binaries to the `bin/` folder in your project by default. `make test` is the one-stop shop for downloading the binaries, setting up the test environment, and running the tests.

You can refer to the Makefile of the Kubebuilder scaffold and observe that the envtest setup is consistently aligned across all controller-runtime releases. Starting from `release-0.19`, it is configured to automatically download the artefact from the correct location, **ensuring that kubebuilder users are not impacted.**

```

## Tool Binaries
..
ENVTEST ?= $(LOCALBIN)/setup-envtest
...

## Tool Versions
...
#ENVTEST_VERSION is the version of controller-runtime release branch to fetch
the envtest setup script (i.e. release-0.20)
ENVTEST_VERSION ?= $(shell go list -m -f "{{ .Version }}" \
sigs.k8s.io/controller-runtime | awk -F'[v.]' '{printf "release-%d.%d", $2,
$3}')
#ENVTEST_K8S_VERSION is the version of Kubernetes to use for setting up
ENVTEST binaries (i.e. 1.31)
ENVTEST_K8S_VERSION ?= $(shell go list -m -f "{{ .Version }}" k8s.io/api | awk
-F'[v.]' '{printf "1.%d", $3}')
...
.PHONY: setup-envtest
setup-envtest: envtest ## Download the binaries required for ENVTEST in the
local bin directory.
    @echo "Setting up envtest binaries for Kubernetes version
$(ENVTEST_K8S_VERSION)..."
    @$(ENVTEST) use $(ENVTEST_K8S_VERSION) --bin-dir $(LOCALBIN) -p path || {
\
        echo "Error: Failed to set up envtest binaries for version
$(ENVTEST_K8S_VERSION)."; \
        exit 1; \
    }

.PHONY: envtest
envtest: $(ENVTEST) ## Download setup-envtest locally if necessary.
$(ENVTEST): $(LOCALBIN)
    $(call go-install-tool,$(ENVTEST),sigs.k8s.io/controller-
runtime/tools/setup-envtest,$(ENVTEST_VERSION))

```

## Installation in Air Gapped/disconnected environments

If you would like to download the tarball containing the binaries, to use in a disconnected environment you can use `setup-envtest` to download the required binaries locally. There are a lot of ways to configure `setup-envtest` to avoid talking to the internet you can read about them [here](#). The examples below will show how to install the Kubernetes API binaries using mostly defaults set by `setup-envtest`.

### Download the binaries

`make envtest` will download the `setup-envtest` binary to `./bin/`.

```
make envtest
```

Installing the binaries using `setup-envtest` stores the binary in OS specific locations, you can read more about them [here](#)

```
./bin/setup-envtest use 1.31.0
```

## Update the test make target

Once these binaries are installed, change the `test` make target to include a `-i` like below. `-i` will only check for locally installed binaries and not reach out to remote resources. You could also set the `ENVTEST_INSTALLED_ONLY` env variable.

```
test: manifests generate fmt vet
    KUBEBUILDER_ASSETS="$(shell $(ENVTEST) use $(ENVTEST_K8S_VERSION) -i --
bin-dir $(LOCALBIN) -p path)" go test ./... -coverprofile cover.out
```

NOTE: The `ENVTEST_K8S_VERSION` needs to match the `setup-envtest` you downloaded above. Otherwise, you will see an error like the below

```
no such version (1.24.5) exists on disk for this architecture (darwin/amd64) --
try running `list -i` to see what's on disk
```

## Writing tests

Using `envtest` in integration tests follows the general flow of:

```
import sigs.k8s.io/controller-runtime/pkg/envtest

//specify testEnv configuration
testEnv = &envtest.Environment{
    CRDDirectoryPaths: []string{filepath.Join("../", "config", "crd",
"bases")},
}

//start testEnv
cfg, err = testEnv.Start()

//write test logic

//stop testEnv
err = testEnv.Stop()
```

`kubebuilder` does the boilerplate setup and teardown of `testEnv` for you, in the ginkgo test suite that it generates under the `/controllers` directory.

Logs from the test runs are prefixed with `test-env`.

## Examples

You can use the plugin [DeployImage](#) to check examples. This plugin allows users to scaffold API/Controllers to deploy and manage an Operand (image) on the cluster following the guidelines and best practices. It abstracts the complexities of achieving this goal while allowing users to customize the generated code.

Therefore, you can check that a test using ENV TEST will be generated for the controller which has the purpose to ensure that the Deployment is created successfully. You can see an example of its code implementation under the `testdata` directory with the [DeployImage](#) samples [here](#).

## Configuring your test control plane

Controller-runtime's `envtest` framework requires `kubectl`, `kube-apiserver`, and `etcd` binaries be present locally to simulate the API portions of a real cluster.

The `make test` command will install these binaries to the `bin/` directory and use them when running tests that use `envtest`. i.e,

```

./bin/k8s/
└── 1.25.0-darwin-amd64
    ├── etcd
    ├── kube-apiserver
    └── kubectl

```

You can use environment variables and/or flags to specify the `kubectl`, `api-server` and `etcd` setup within your integration tests.

## Environment Variables

| Variable name   | Type   | When to use  |
|---|--|--|
| <code>USE_EXISTING_CLUSTER</code>   | boolean  | Instead of a local connection, point to the control plane of an existing cluster.  |
| <code>KUBEBUILDER_ASSETS</code>   | path to directory  | Point integration tests to a directory containing <code>(api-server, etcd, kubectl)</code> .   |
| <code>TEST_ASSET_KUBE_APISERVER</code> ,<br><code>TEST_ASSET_ETCD</code> , <code>TEST_ASSET_KUBECTL</code>    | paths to, respectively, api-server, etcd and kubectl binaries    | Similar to <code>KUBEBUILD_ASSETS</code> but more flexible. Point integration tests to user-specified assets other than default ones. Environment variables can be used to enable specific test cases with expected values for these binaries. |
| <code>KUBEBUILDER_CONTROLPLANE_START_TIMEOUT</code> and<br><code>KUBEBUILDER_CONTROLPLANE_STOP_TIMEOUT</code> | durations in format supported by <code>time.ParseDuration</code> | Specify timeouts for different from the default for control plane (respectively start and stop; a  |

| Variable name                           | Type    | When to use  |
|---|---------|--|
| KUBEBUILDER_ATTACH_CONTROL_PLANE_OUTPUT | boolean | Set to <code>true</code> if you want the control plane to attach to the stdout and stderr streams. This is useful when debugging failures, as it will include output from the control plane. |

See that the `test` makefile target will ensure that all is properly setup when you are using it. However, if you would like to run the tests without use the Makefile targets, for example via an IDE, then you can set the environment variables directly in the code of your `suite_test.go`:

```

var _ = BeforeSuite(func(done Done) {
    Expect(os.Getenv("TEST_ASSET_KUBE_APISERVER", "../bin/k8s/1.25.0-darwin-amd64/kube-apiserver")).To(Succeed())
    Expect(os.Getenv("TEST_ASSET_ETCD", "../bin/k8s/1.25.0-darwin-amd64/etcd")).To(Succeed())
    Expect(os.Getenv("TEST_ASSET_KUBECTL", "../bin/k8s/1.25.0-darwin-amd64/kubectl")).To(Succeed())
    // OR
    Expect(os.Getenv("KUBEBUILDER_ASSETS", "../bin/k8s/1.25.0-darwin-amd64")).To(Succeed())
}

logf.SetLogger(zap.New(zap.WriteTo(GinkgoWriter), zap.UseDevMode(true)))
testenv = &envtest.Environment{}

_, err := testenv.Start()
Expect(err).NotTo(HaveOccurred())

    close(done)
}, 60)

var _ = AfterSuite(func() {
    Expect(testenv.Stop()).To(Succeed())

    Expect(os.Unsetenv("TEST_ASSET_KUBE_APISERVER")).To(Succeed())
    Expect(os.Unsetenv("TEST_ASSET_ETCD")).To(Succeed())
    Expect(os.Unsetenv("TEST_ASSET_KUBECTL")).To(Succeed())
})

```

## ENV TEST Config Options

You can look at the controller-runtime docs to know more about its configuration options, see [here](#). On top of that, if you are looking to use ENV TEST to test your webhooks then you might want to give a look at its install [options](#).

## Flags

Here's an example of modifying the flags with which to start the API server in your integration tests, compared to the default values in `envtest.DefaultKubeAPIServerFlags`:

```
customApiServerFlags := []string{
    "--secure-port=6884",
    "--admission-control=MutatingAdmissionWebhook",
}

apiServerFlags := append([]string(nil), envtest.DefaultKubeAPIServerFlags...)
apiServerFlags = append(apiServerFlags, customApiServerFlags...)

testEnv = &envtest.Environment{
    CRDDirectoryPaths: []string{filepath.Join("../", "config", "crd",
"bases")},
    KubeAPIServerFlags: apiServerFlags,
}
```

## Testing considerations

Unless you're using an existing cluster, keep in mind that no built-in controllers are running in the test context. In some ways, the test control plane will behave differently from "real" clusters, and that might have an impact on how you write tests. One common example is garbage collection; because there are no controllers monitoring built-in resources, objects do not get deleted, even if an `OwnerReference` is set up.

To test that the deletion lifecycle works, test the ownership instead of asserting on existence. For example:

```
expectedOwnerReference := v1.OwnerReference{
    Kind:      "MyCoolCustomResource",
    APIVersion: "my.api.example.com/v1beta1",
    UID:       "d9607e19-f88f-11e6-a518-42010a800195",
    Name:      "userSpecifiedResourceName",
}
Expect(deployment.ObjectMeta.OwnerReferences).To(ContainElement(expectedOwnerReference))
```

## i Namespace usage limitation

EnvTest does not support namespace deletion. Deleting a namespace will seem to succeed, but the namespace will just be put in a Terminating state, and never actually be reclaimed. Trying to recreate the namespace will fail. This will cause your reconciler to continue reconciling any objects left behind, unless they are deleted.

To overcome this limitation you can create a new namespace for each test. Even so, when one test completes (e.g. in “namespace-1”) and another test starts (e.g. in “namespace-2”), the controller will still be reconciling any active objects from “namespace-1”. This can be avoided by ensuring that all tests clean up after themselves as part of the test teardown. If teardown of a namespace is difficult, it may be possible to wire the reconciler in such a way that it ignores reconcile requests that come from namespaces other than the one being tested:

```
type MyCoolReconciler struct {
    client.Client
    ...
    Namespace     string // restrict namespaces to reconcile
}
func (r *MyCoolReconciler) Reconcile(ctx context.Context, req ctrl.Request) (ctrl.Result, error) {
    _ = r.Log.WithValues("myreconciler", req.NamespacedName)
    // Ignore requests for other namespaces, if specified
    if r.Namespace != "" && req.Namespace != r.Namespace {
        return ctrl.Result{}, nil
    }
}
```

Whenever your tests create a new namespace, it can modify the value of reconciler.Namespace. The reconciler will effectively ignore the previous namespace. For further information see the issue raised in the controller-runtime [controller-runtime/issues/880](#) to add this support.

# Cert-Manager and Prometheus options

Projects scaffolded with Kubebuilder can enable the `metrics` and the `cert-manager` options. Note that when we are using the ENV TEST we are looking to test the controllers and their reconciliation. It is considered an integrated test because the ENV TEST API will do the test against a cluster and because of this the binaries are downloaded and used to configure its pre-requirements, however, its purpose is mainly to `unit` test the controllers.

Therefore, to test a reconciliation in common cases you do not need to care about these options. However, if you would like to do tests with the Prometheus and the Cert-manager installed you can add the required steps to install them before running the tests. Following an example.

```
// Add the operations to install the Prometheus operator and the cert-
manager
// before the tests.
BeforeEach(func() {
    By("installing prometheus operator")
    Expect(utils.InstallPrometheusOperator()).To(Succeed())

    By("installing the cert-manager")
    Expect(utils.InstallCertManager()).To(Succeed())
})

// You can also remove them after the tests::
AfterEach(func() {
    By("uninstalling the Prometheus manager bundle")
    utils.UninstallPrometheusOperManager()

    By("uninstalling the cert-manager bundle")
    utils.UninstallCertManager()
})
```

Check the following example of how you can implement the above operations:

```

const (
    prometheusOperatorVersion = "0.51"
    prometheusOperatorURL      = "https://raw.githubusercontent.com/prometheus-
operator/" + "prometheus-operator/release-%s/bundle.yaml"
    certmanagerVersion = "v1.5.3"
    certmanagerURLTmpl = "https://github.com/cert-manager/cert-
manager/releases/download/%s/cert-manager.yaml"
)

func warnError(err error) {
    _, _ = fmt.Fprintf(GinkgoWriter, "warning: %v\n", err)
}

// InstallPrometheusOperator installs the prometheus Operator to be used to
// export the enabled metrics.
func InstallPrometheusOperator() error {
    url := fmt.Sprintf(prometheusOperatorURL, prometheusOperatorVersion)
    cmd := exec.Command("kubectl", "apply", "-f", url)
    _, err := Run(cmd)
    return err
}

// UninstallPrometheusOperator uninstalls the prometheus
func UninstallPrometheusOperator() {
    url := fmt.Sprintf(prometheusOperatorURL, prometheusOperatorVersion)
    cmd := exec.Command("kubectl", "delete", "-f", url)
    if _, err := Run(cmd); err != nil {
        warnError(err)
    }
}

// UninstallCertManager uninstalls the cert manager
func UninstallCertManager() {
    url := fmt.Sprintf(certmanagerURLTmpl, certmanagerVersion)
    cmd := exec.Command("kubectl", "delete", "-f", url)
    if _, err := Run(cmd); err != nil {
        warnError(err)
    }
}

// InstallCertManager installs the cert manager bundle.
func InstallCertManager() error {
    url := fmt.Sprintf(certmanagerURLTmpl, certmanagerVersion)
    cmd := exec.Command("kubectl", "apply", "-f", url)
    if _, err := Run(cmd); err != nil {
        return err
    }
    // Wait for cert-manager-webhook to be ready, which can take time if cert-
    manager
    //was re-installed after uninstalling on a cluster.
    cmd = exec.Command("kubectl", "wait", "deployment.apps/cert-manager-
webhook",
        "--for", "condition=Available",

```

```

    "--namespace", "cert-manager",
    "--timeout", "5m",
)
}

_, err := Run(cmd)
return err
}

// LoadImageToKindClusterWithName loads a local docker image to the kind
cluster
func LoadImageToKindClusterWithName(name string) error {
    cluster := "kind"
    if v, ok := os.LookupEnv("KIND_CLUSTER"); ok {
        cluster = v
    }

    kindOptions := []string{"load", "docker-image", name, "--name", cluster}
    cmd := exec.Command("kind", kindOptions...)
    _, err := Run(cmd)
    return err
}

```

However, see that tests for the metrics and cert-manager might fit better well as e2e tests and not under the tests done using ENV TEST for the controllers. You might want to give a look at the [sample example](#) implemented into [Operator-SDK](#) repository to know how you can write your e2e tests to ensure the basic workflows of your project. Also, see that you can run the tests against a cluster where you have some configurations in place they can use the option to test using an existing cluster:

```

testEnv = &envtest.Environment{
    UseExistingCluster: true,
}

```

#### Setup ENV TEST tool

To know more about the tooling used to configure ENVTEST which is used in the setup-envtest target in the Makefile of the projects build with Kubebuilder see the [README] [readme] of its tooling.

## Metrics

By default, controller-runtime builds a global prometheus registry and publishes [a collection of performance metrics](#) for each controller.

 **IMPORTANT:** If you are using `kube-rbac-proxy`

Please stop using the image `gcr.io/kubebuilder/kube-rbac-proxy` as soon as possible. Your projects will be affected and may fail to work if the image cannot be pulled.

**Images provided under `gcr.io/kubebuilder/` will be unavailable from early 2025.**

- **Projects initialized with Kubebuilder versions v3.14 or lower** utilize `kube-rbac-proxy` to protect the metrics endpoint. In this case, you might want to upgrade your project to the latest release or ensure that you have applied the same or similar code changes.
- **However, projects initialized with Kubebuilder versions v4.1.0 or higher** have similar protection using `authn/authz` enabled by default via Controller-Runtime's feature [WithAuthenticationAndAuthorization](#).

If you want to continue using `kube-rbac-proxy` then you MUST change your project to use the image from another source.

For further information, see: [kubebuilder/discussions/3907](#)

## Metrics Configuration

By looking at the file `config/default/kustomization.yaml` you can check the metrics are exposed by default:

```
# [METRICS] Expose the controller manager metrics service.  
- metrics_service.yaml
```

```
patches:
  # [METRICS] The following patch will enable the metrics endpoint using
  # HTTPS and the port :8443.
  # More info: https://book.kubebuilder.io/reference/metrics
  - path: manager_metrics_patch.yaml
    target:
      kind: Deployment
```

Then, you can check in the `cmd/main.go` where metrics server is configured:

```
// Metrics endpoint is enabled in 'config/default/kustomization.yaml'. The
// Metrics options configure the server.
// For more info: https://pkg.go.dev/sigs.k8s.io/controller-
// runtime/pkg/metrics/server
Metrics: metricsserver.Options{
  ...
},
```

## Metrics Protection

Unprotected metrics endpoints can expose valuable data to unauthorized users, such as system performance, application behavior, and potentially confidential operational metrics. This exposure can lead to security vulnerabilities where an attacker could gain insights into the system's operation and exploit weaknesses.

### By using authn/authz (Enabled by default)

To mitigate these risks, Kubebuilder projects utilize authentication (authn) and authorization (authz) to protect the metrics endpoint. This approach ensures that only authorized users and service accounts can access sensitive metrics data, enhancing the overall security of the system.

In the past, the [kube-rbac-proxy](#) was employed to provide this protection. However, its usage has been discontinued in recent versions. Since the release of `v4.1.0`, projects have had the metrics endpoint enabled and protected by default using the [WithAuthenticationAndAuthorization](#) feature provided by controller-runtime.

Therefore, you will find the following configuration:

- In the `cmd/main.go`:

```
if secureMetrics {  
    ...  
    metricsServerOptions.FilterProvider =  
filters.WithAuthenticationAndAuthorization  
}
```

This configuration leverages the FilterProvider to enforce authentication and authorization on the metrics endpoint. By using this method, you ensure that the endpoint is accessible only to those with the appropriate permissions.

- In the `config/rbac/kustomization.yaml`:

```
# The following RBAC configurations are used to protect  
# the metrics endpoint with authn/authz. These configurations  
# ensure that only authorized users and service accounts  
# can access the metrics endpoint.  
- metrics_auth_role.yaml  
- metrics_auth_role_binding.yaml  
- metrics_reader_role.yaml
```

In this way, only Pods using the `ServiceAccount` token are authorized to read the metrics endpoint. For example:

```
apiVersion: v1  
kind: Pod  
metadata:  
  name: metrics-consumer  
  namespace: system  
spec:  
  # Use the scaffolded service account name to allow authn/authz  
  serviceAccountName: controller-manager  
  containers:  
  - name: metrics-consumer  
    image: curlimages/curl:latest  
    command: ["/bin/sh"]  
    args:  
      - "-c"  
      - >  
        while true;  
        do  
          # Note here that we are passing the token obtained from the  
          ServiceAccount to curl the metrics endpoint  
          curl -s -k -H "Authorization: Bearer $(cat  
/var/run/secrets/kubernetes.io/serviceaccount/token)"  
            https://controller-manager-metrics-  
service.system.svc.cluster.local:8443/metrics;  
          sleep 60;  
        done
```

## (Recommended) Enabling certificates for Production (Disabled by default)

### Why Is This Not Enabled by Default?

This option is not enabled by default because it introduces a dependency on CertManager. To keep the project as lightweight and beginner-friendly as possible, it is disabled by default.

### Recommended for Production

The default scaffold in `cmd/main.go` uses a **controller-runtime feature** to automatically generate a self-signed certificate to secure the metrics server. While this is convenient for development and testing, it is **not** recommended for production.

Those certificates are used to secure the transport layer (TLS). The token authentication using `authn/authz`, which is enabled by default serves as the application-level credential. However, for example, when you enable the integration of your metrics with Prometheus, those certificates can be used to secure the communication.

Projects built with KubeBuilder releases `4.4.0` and above have the logic scaffolded to enable the usage of certificates managed by [CertManager](#) for securing the metrics server. Following the steps below, you can configure your project to use certificates managed by CertManager.

#### 1. Enable Cert-Manager in `config/default/kustomization.yaml`:

- Uncomment the cert-manager resource to include it in your project:
  - `./certmanager`

#### 2. Enable the Patch to configure the usage of the certs in the Controller Deployment in `config/default/kustomization.yaml`:

- Uncomment the `cert_metrics_manager_patch.yaml` to mount the `serving-cert` secret in the Manager Deployment.

```
# Uncomment the patches line if you enable Metrics and CertManager
# [METRICS-WITH-CERTS] To enable metrics protected with certManager,
uncomment the following line.
# This patch will protect the metrics with certManager self-signed
certs.
- path: cert_metrics_manager_patch.yaml
  target:
    kind: Deployment
```

### 3. Enable the CertManager replaces for the Metrics Server certificates in config/default/kustomization.yaml :

- Uncomment the replacements block bellow. It is required to properly set the DNS names for the certificates configured under config/certmanager .

```
# [CERTMANAGER] To enable cert-manager, uncomment all sections with
'CERTMANAGER' prefix.

# Uncomment the following replacements to add the cert-manager CA
injection annotations

#replacements:

# - source: # Uncomment the following block to enable certificates
for metrics

#     kind: Service
#     version: v1
#     name: controller-manager-metrics-service
#     fieldPath: metadata.name

#     targets:
#         - select:
#             kind: Certificate
#             group: cert-manager.io
#             version: v1
#             name: metrics-certs
#             fieldPaths:
#                 - spec.dnsNames.0
#                 - spec.dnsNames.1
#             options:
#                 delimiter: '.'
#                 index: 0
#                 create: true
#
# - source:
#     kind: Service
#     version: v1
#     name: controller-manager-metrics-service
#     fieldPath: metadata.namespace

#     targets:
#         - select:
#             kind: Certificate
#             group: cert-manager.io
#             version: v1
#             name: metrics-certs
#             fieldPaths:
#                 - spec.dnsNames.0
#                 - spec.dnsNames.1
#             options:
#                 delimiter: '.'
```

```
#           index: 1
#           create: true
#
```

#### 4. Enable the Patch for the `ServiceMonitor` to Use the Cert-Manager-Managed Secret `config/prometheus/kustomization.yaml`:

- Add or uncomment the `ServiceMonitor` patch to securely reference the cert-manager-managed secret, replacing insecure configurations with secure certificate verification:

```
# [PROMETHEUS-WITH-CERTS] The following patch configures the
ServiceMonitor in ../prometheus
# to securely reference certificates created and managed by cert-
manager.
# Additionally, ensure that you uncomment the [METRICS WITH
CERTMANAGER] patch under config/default/kustomization.yaml
# to mount the "metrics-server-cert" secret in the Manager
Deployment.
patches:
  - path: monitor_tls_patch.yaml
    target:
      kind: ServiceMonitor
```

---

**NOTE** that the `ServiceMonitor` patch above will ensure that if you enable the Prometheus integration, it will securely reference the certificates created and managed by CertManager. But it will **not** enable the integration with Prometheus. To enable the integration with Prometheus, you need uncomment the `#- ../certmanager` in the `config/default/kustomization.yaml`. For more information, see [Exporting Metrics for Prometheus](#).

#### (Optional) By using Network Policy (Disabled by default)

NetworkPolicy acts as a basic firewall for pods within a Kubernetes cluster, controlling traffic flow at the IP address or port level. However, it doesn't handle `authn/authz`.

Uncomment the following line in the `config/default/kustomization.yaml`:

```
# [NETWORK POLICY] Protect the /metrics endpoint and Webhook Server with NetworkPolicy.  
# Only Pod(s) running a namespace labeled with 'metrics: enabled' will be able to gather the metrics.  
# Only CR(s) which uses webhooks and applied on namespaces labeled 'webhooks: enabled' will be able to work properly.  
#- ./network-policy
```

## Exporting Metrics for Prometheus

Follow the steps below to export the metrics using the Prometheus Operator:

1. Install Prometheus and Prometheus Operator. We recommend using [kube-prometheus](#) in production if you don't have your own monitoring system. If you are just experimenting, you can only install Prometheus and Prometheus Operator.
2. Uncomment the line `- ./prometheus` in the `config/default/kustomization.yaml`. It creates the `ServiceMonitor` resource which enables exporting the metrics.

```
# [PROMETHEUS] To enable prometheus monitor, uncomment all sections with 'PROMETHEUS'.  
- ./prometheus
```

Note that, when you install your project in the cluster, it will create the `ServiceMonitor` to export the metrics. To check the `ServiceMonitor`, run `kubectl get ServiceMonitor -n <project>-system`. See an example:

```
$ kubectl get ServiceMonitor -n monitor-system  
NAME                                AGE  
monitor-controller-manager-metrics-monitor   2m8s
```

 **If you are using Prometheus Operator ensure that you have the required permissions**

If you are using Prometheus Operator, be aware that, by default, its RBAC rules are only enabled for the `default` and `kube-system` namespaces. See its guide to know [how to configure kube-prometheus to monitor other namespaces using the .jsonnet file](#).

Alternatively, you can give the Prometheus Operator permissions to monitor other namespaces using RBAC. See the Prometheus Operator [Enable RBAC rules for Prometheus pods](#) documentation to know how to enable the permissions on the namespace where the `ServiceMonitor` and manager exist.

Also, notice that the metrics are exported by default through port `8443`. In this way, you are able to check the Prometheus metrics in its dashboard. To verify it, search for the metrics exported from the namespace where the project is running `{namespace="<project>-system"}`. See an example:

The screenshot shows the Prometheus UI interface. At the top, there are tabs for Prometheus, Alerts, Graph, Status, Help, and a dropdown for 'Enable query history'. Below the tabs is a search bar containing the query `{namespace="monitor-system"}`. There are two buttons: 'Execute' (highlighted in blue) and '- insert metric at cursor'. Below the search bar is a navigation bar with 'Graph' and 'Console' tabs, and a moment selector. The main area displays a table of search results with columns for 'Element', 'Value', and 'Label'. The table contains numerous rows of Prometheus metric definitions, such as alert definitions and counter metrics for CPU throttling and system seconds. The table is scrollable, and the bottom right corner shows performance metrics: Load time: 52ms, Resolution: 14s, Total time series: 137.

| Element  | Value | Label |
|--|-------|-------|
| ALERTS{alertname="CPUThrottlingHigh",alertstate="pending",container="manager",namespace="monitor-system",pod="monitor-controller-manager-5686c8c89-slgsl",severity="warning"}  | 1     |       |
| ALERTS_FOR_STATE{alertname="CPUThrottlingHigh",container="manager",namespace="monitor-system",pod="monitor-controller-manager-5686c8c89-slgsl",severity="warning"}   | 157   |       |
| container_cpu_cfs_periods_total{container="manager",container_name="manager",endpoint="https-metrics",id="/kubepods/burstable/pod6d912b20-e50b-11e9-acf8-0e1eba7688b6/474668f98b4df2277c76e8e2d94ac9c134284e6aab6ed21144556d64c9c5a6f4",image="cmacdoo/monitor@sha256:049ad34393559213b2d5b1b81b24fa09e56a9b5e7cf61333d5681f76667d8a",instance="192.168.64.39:10250",job="k8s_manager_monitor-controller-manager-5686c8c89-slgsl",monitor_system,"_6d912b20-e50b-11e9-acf8-0e1eba7688b6_0",namespace="monitor-system",node="minikube",pod="monitor-controller-manager-5686c8c89-slgsl",pod_name="monitor-controller-manager-5686c8c89-slgsl",service="kubelet"}                                      | 137   |       |
| container_cpu_cfs_throttled_periods_total{container="manager",container_name="manager",endpoint="https-metrics",id="/kubepods/burstable/pod6d912b20-e50b-11e9-acf8-0e1eba7688b6/474668f98b4df2277c76e8e2d94ac9c134284e6aab6ed21144556d64c9c5a6f4",image="cmacdoo/monitor@sha256:049ad34393559213b2d5b1b81b24fa09e56a9b5e7cf61333d5681f76667d8a",instance="192.168.64.39:10250",job="k8s_manager_monitor-controller-manager-5686c8c89-slgsl",monitor_system,"_6d912b20-e50b-11e9-acf8-0e1eba7688b6_0",namespace="monitor-system",node="minikube",pod="monitor-controller-manager-5686c8c89-slgsl",pod_name="monitor-controller-manager-5686c8c89-slgsl",service="kubelet"}                            | 398   |       |
| container_cpu_cfs_throttled_seconds_total{container="manager",container_name="manager",endpoint="https-metrics",id="/kubepods/burstable/pod6d912b20-e50b-11e9-acf8-0e1eba7688b6/474668f98b4df2277c76e8e2d94ac9c134284e6aab6ed21144556d64c9c5a6f4",image="cmacdoo/monitor@sha256:049ad34393559213b2d5b1b81b24fa09e56a9b5e7cf61333d5681f76667d8a",instance="192.168.64.39:10250",job="k8s_manager_monitor-controller-manager-5686c8c89-slgsl",monitor_system,"_6d912b20-e50b-11e9-acf8-0e1eba7688b6_0",namespace="monitor-system",node="minikube",pod="monitor-controller-manager-5686c8c89-slgsl",pod_name="monitor-controller-manager-5686c8c89-slgsl",service="kubelet"}                            | 36..  |       |
| container_cpu_system_seconds_total{container="POD",container_name="POD",endpoint="https-metrics",id="/kubepods/burstable/pod6d912b20-e50b-11e9-acf8-0e1eba7688b6_0",namespace="monitor-system",node="minikube",pod="monitor-controller-manager-5686c8c89-slgsl",pod_name="monitor-controller-manager-5686c8c89-slgsl",service="kubelet"}   | 0.01  |       |
| container_cpu_system_seconds_total{container="k8s_POD_monitor-controller-manager-5686c8c89-slgsl",container_name="k8s_POD_monitor-controller-manager-5686c8c89-slgsl",endpoint="https-metrics",id="/kubepods/burstable/pod6d912b20-e50b-11e9-acf8-0e1eba7688b6_0",namespace="monitor-system",node="minikube",pod="monitor-controller-manager-5686c8c89-slgsl",pod_name="monitor-controller-manager-5686c8c89-slgsl",service="kubelet"}   | 0.01  |       |
| container_cpu_system_seconds_total{container="kube-rbac-proxy",container_name="kube-rbac-proxy",endpoint="https-metrics",id="/kubepods/burstable/pod6d912b20-e50b-11e9-acf8-0e1eba7688b6/aff6d212373f3826a8ccb2bc69d9656b243853d64ab73d9f13d69eb6",image="crs.joi.kubebuilder/kube-rbac-proxy@sha256:c6c915d484d781d366300de765d678309410f78319f0fec21c7744053eff",instance="192.168.64.39:10250",job="k8s_kube-rbac-proxy_monitor-controller-manager-5686c8c89-slgsl",monitor_system,"_6d912b20-e50b-11e9-acf8-0e1eba7688b6_0",namespace="monitor-system",node="minikube",pod="monitor-controller-manager-5686c8c89-slgsl",pod_name="monitor-controller-manager-5686c8c89-slgsl",service="kubelet"} | 0.01  |       |
| container_cpu_system_seconds_total{container="k8s_kube-rbac-proxy_monitor-controller-manager-5686c8c89-slgsl",container_name="k8s_kube-rbac-proxy_monitor-controller-manager-5686c8c89-slgsl",endpoint="https-metrics",id="/kubepods/burstable/pod6d912b20-e50b-11e9-acf8-0e1eba7688b6",image="k8s_gcr.io/pause:3.1",instance="192.168.64.39:10250",job="k8s_POD_monitor-controller-manager-5686c8c89-slgsl",monitor_system,"_6d912b20-e50b-11e9-acf8-0e1eba7688b6_0",namespace="monitor-system",node="minikube",pod="monitor-controller-manager-5686c8c89-slgsl",pod_name="monitor-controller-manager-5686c8c89-slgsl",service="kubelet"}   | 0.01  |       |
| container_cpu_usage_seconds_total{container="POD",cpu="total",endpoint="https-metrics",id="/kubepods/burstable/pod6d912b20-e50b-11e9-acf8-0e1eba7688b6_0",image="k8s_gcr.io/pause:3.1",instance="192.168.64.39:10250",job="k8s_POD_monitor-controller-manager-5686c8c89-slgsl",monitor_system,"_6d912b20-e50b-11e9-acf8-0e1eba7688b6_0",namespace="monitor-system",node="minikube",pod="monitor-controller-manager-5686c8c89-slgsl",pod_name="monitor-controller-manager-5686c8c89-slgsl",service="kubelet"}   | 0.01  |       |

## Publishing Additional Metrics

If you wish to publish additional metrics from your controllers, this can be easily achieved by using the global registry from `controller-runtime/pkg/metrics`.

One way to achieve this is to declare your collectors as global variables and then register them using `init()` in the controller's package.

For example:

```

import (
    "github.com/prometheus/client_golang/prometheus"
    "sigs.k8s.io/controller-runtime/pkg/metrics"
)

var (
    goobers = prometheus.NewCounter(
        prometheus.CounterOpts{
            Name: "goobers_total",
            Help: "Number of goobers processed",
        },
    )
    gooberFailures = prometheus.NewCounter(
        prometheus.CounterOpts{
            Name: "goober_failures_total",
            Help: "Number of failed goobers",
        },
    )
)

func init() {
    // Register custom metrics with the global prometheus registry
    metrics.Registry.MustRegister(goobers, gooberFailures)
}

```

You may then record metrics to those collectors from any part of your reconcile loop. These metrics can be evaluated from anywhere in the operator code.

### Enabling metrics in Prometheus UI

In order to publish metrics and view them on the Prometheus UI, the Prometheus instance would have to be configured to select the Service Monitor instance based on its labels.

Those metrics will be available for Prometheus or other openmetrics systems to scrape.

The screenshot shows the Prometheus UI interface. At the top, there's a navigation bar with links for Prometheus, Alerts, Graph, Status, Help, and Classic UI. Below the navigation bar are several configuration checkboxes: 'Use local time' (unchecked), 'Enable query history' (checked), 'Enable autocomplete' (checked), 'Use experimental editor' (unchecked), 'Enable highlighting' (unchecked), and 'Enable Interpolate' (checked). A search bar contains the query 'goobers\_total'. Under the search bar, there are tabs for 'Table' (selected) and 'Graph'. The main area displays a single data series named 'goobers\_total' with a value of 10. The query details below the graph show: container="kube-rbac-proxy", endpoint="https", instance="172.17.0.14:8443", job="memcached-operator-controller-manager-metrics-service", namespace="memcached-operator-system", pod="memcached-operator-controller-manager-68d4bf9f14-mdq4x", service="memcached-operator-controller-manager-metrics-service". At the bottom right, there are buttons for 'Remove Panel' and 'Add Panel'.

## Controller-Runtime Auth/Authz Feature Current Known Limitations and Considerations

Some known limitations and considerations have been identified. The settings for `cache TTL`, `anonymous access`, and `timeouts` are currently hardcoded, which may lead to performance and security concerns due to the inability to fine-tune these parameters. Additionally, the current implementation lacks support for configurations like `alwaysAllow` for critical paths (e.g., `/healthz`) and `alwaysAllowGroups` (e.g., `system:masters`), potentially causing operational challenges. Furthermore, the system heavily relies on stable connectivity to the `kube-apiserver`, making it vulnerable to metrics outages during network instability. This can result in the loss of crucial metrics data, particularly during critical periods when monitoring and diagnosing issues in real-time is essential.

An [issue](#) has been opened to enhance the controller-runtime and address these considerations.

## Default Exported Metrics References

Following the metrics which are exported and provided by [controller-runtime](#) by default:

| Metrics name                                      | Type      | Description  |
|---|-----------|--|
| <a href="#">workqueue_depth</a>                   | Gauge     | Current depth of workqueue.  |
| <a href="#">workqueue_adds_total</a>              | Counter   | Total number of adds handled by workqueue.   |
| <a href="#">workqueue_queue_duration_seconds</a>  | Histogram | How long in seconds an item stays in workqueue before being requested.   |
| <a href="#">workqueue_work_duration_seconds</a>   | Histogram | How long in seconds processing an item from workqueue takes.   |
| <a href="#">workqueue_unfinished_work_seconds</a> | Gauge     | How many seconds of work has been done that is in progress and hasn't been observed by work_duration. Large values indicate stuck threads. One can deduce the number of stuck threads by observing the rate at which this increases. |

| Metrics name                                       | Type      | Description  |
|--|-----------|--|
| workqueue_longest_running_processor_seconds        | Gauge     | How many seconds has the longest running processor for workqueue been running. |
| workqueue_retries_total                            | Counter   | Total number of retries handled by workqueue.                                  |
| rest_client_requests_total                         | Counter   | Number of HTTP requests, partitioned by status code, method, and host.         |
| controller_runtime_reconcile_total                 | Counter   | Total number of reconciliations per controller.                                |
| controller_runtime_reconcile_errors_total          | Counter   | Total number of reconciliation errors per controller.                          |
| controller_runtime_terminal_reconcile_errors_total | Counter   | Total number of terminal errors from the reconciler.                           |
| controller_runtime_reconcile_time_seconds          | Histogram | Length of time per reconciliation per controller.                              |
| controller_runtime_max_concurrent_reconciles       | Gauge     | Maximum number of concurrent reconciles per controller.                        |

| Metrics name                                  | Type      | Description  |
|---|-----------|--|
| controller_runtime_active_workers             | Gauge     | Number of currently used workers per controller.           |
| controller_runtime_webhook_latency_seconds    | Histogram | Histogram of the latency of processing admission requests. |
| controller_runtime_webhook_requests_total     | Counter   | Total number of admission requests by HTTP status code.    |
| controller_runtime_webhook_requests_in_flight | Gauge     | Current number of admission requests being served          |

## Overview

The Project Config represents the configuration of a KubeBuilder project. All projects that are scaffolded with the CLI (KB version 3.0 and higher) will generate the `PROJECT` file in the projects' root directory. Therefore, it will store all plugins and input data used to generate the project and APIs to better enable plugins to make useful decisions when scaffolding.

## Example

Following is an example of a PROJECT config file which is the result of a project generated with two APIs using the [Deploy Image Plugin](#).

```
# Code generated by tool. DO NOT EDIT.
# This file is used to track the info used to scaffold your project
# and allow the plugins properly work.
# More info: https://book.kubebuilder.io/reference/project-config.html
domain: testproject.org
layout:
  - go.kubebuilder.io/v4
plugins:
  deploy-image.go.kubebuilder.io/v1-alpha:
    resources:
      - domain: testproject.org
        group: example.com
        kind: Memcached
        options:
          containerCommand: memcached,--memory-limit=64,-o,modern,-v
          containerPort: "11211"
          image: memcached:1.4.36-alpine
          runAsUser: "1001"
        version: v1alpha1
      - domain: testproject.org
        group: example.com
        kind: Busybox
        options:
          image: busybox:1.28
        version: v1alpha1
projectName: project-v4-with-deploy-image
repo: sigs.k8s.io/kubebuilder/testdata/project-v4-with-deploy-image
resources:
  - api:
      crdVersion: v1
      namespaced: true
      controller: true
      domain: testproject.org
      group: example.com
      kind: Memcached
      path: sigs.k8s.io/kubebuilder/testdata/project-v4-with-deploy-
image/api/v1alpha1
      version: v1alpha1
      webhooks:
        validation: true
        webhookVersion: v1
    - api:
        crdVersion: v1
        namespaced: true
        controller: true
        domain: testproject.org
        group: example.com
        kind: Busybox
        path: sigs.k8s.io/kubebuilder/testdata/project-v4-with-deploy-
image/api/v1alpha1
        version: v1alpha1
version: "3"
```

# Why do we need to store the plugins and data used?

Following some examples of motivations to track the input used:

- check if a plugin can or cannot be scaffolded on top of an existing plugin (i.e.) plugin compatibility while chaining multiple of them together.
- what operations can or cannot be done such as verify if the layout allow API(s) for different groups to be scaffolded for the current configuration or not.
- verify what data can or not be used in the CLI operations such as to ensure that WebHooks can only be created for pre-existent API(s)

Note that KubeBuilder is not only a CLI tool but can also be used as a library to allow users to create their plugins/tools, provide helpers and customizations on top of their existing projects - an example of which is [Operator-SDK](#). SDK leverages KubeBuilder to create plugins to allow users to work with other languages and provide helpers for their users to integrate their projects with, for example, the [Operator Framework solutions/OLM](#). You can check the [plugin's documentation](#) to know more about creating custom plugins.

Additionally, another motivation for the PROJECT file is to help us to create a feature that allows users to easily upgrade their projects by providing helpers that automatically re-scaffold the project. By having all the required metadata regarding the APIs, their configurations and versions in the PROJECT file. For example, it can be used to automate the process of re-scaffolding while migrating between plugin versions. ([More info](#)).

## Versioning

The Project config is versioned according to its layout. For further information see [Versioning](#).

## Layout Definition

The PROJECT version 3 layout looks like:

```

domain: testproject.org
layout:
  - go.kubebuilder.io/v4
plugins:
  deploy-image.go.kubebuilder.io/v1-alpha:
    resources:
      - domain: testproject.org
        group: example.com
        kind: Memcached
        options:
          containerCommand: memcached,--memory-limit=64,-o,modern,-v
          containerPort: "11211"
          image: memcached:memcached:1.6.26-alpine3.19
          runAsUser: "1001"
        version: v1alpha1
      - domain: testproject.org
        group: example.com
        kind: Busybox
        options:
          image: busybox:1.36.1
        version: v1alpha1
projectName: project-v4-with-deploy-image
repo: sigs.k8s.io/kubebuilder/testdata/project-v4-with-deploy-image
resources:
  - api:
      crdVersion: v1
      namespaced: true
      controller: true
      domain: testproject.org
      group: example.com
      kind: Memcached
      path: sigs.k8s.io/kubebuilder/testdata/project-v4-with-deploy-
image/api/v1alpha1
      version: v1alpha1
      webhooks:
        validation: true
        webhookVersion: v1
  - api:
      crdVersion: v1
      namespaced: true
      controller: true
      domain: testproject.org
      group: example.com
      kind: Busybox
      path: sigs.k8s.io/kubebuilder/testdata/project-v4-with-deploy-
image/api/v1alpha1
      version: v1alpha1
version: "3"

```

Now let's check its layout fields definition:

| Field                                 | Description   |
|---------------------------------------|---|
| <code>layout</code>                   | Defines the global plugins, e.g. a project <code>init</code> with <code>--plugins="go/v4,deploy-image/v1-alpha"</code> means that any sub-command used will always call its implementation for both plugins in a chain.   |
| <code>domain</code>                   | Store the domain of the project. This information can be provided by the user when the project is generate with the <code>init</code> sub-command and the <code>domain</code> flag.   |
| <code>plugins</code>                  | Defines the plugins used to do custom scaffolding, e.g. to use the optional <code>deploy-image/v1-alpha</code> plugin to do scaffolding for just a specific api via the command <code>kubebuilder create api [options] --plugins=deploy-image/v1-alpha</code> . |
| <code>projectName</code>              | The name of the project. This will be used to scaffold the manager data. By default it is the name of the project directory, however, it can be provided by the user in the <code>init</code> sub-command via the <code>--project-name</code> flag.             |
| <code>repo</code>                     | The project repository which is the Golang module, e.g <code>github.com/example/myproject-operator</code> .   |
| <code>resources</code>                | An array of all resources which were scaffolded in the project.   |
| <code>resources.api</code>            | The API scaffolded in the project via the sub-command <code>create api</code> .   |
| <code>resources.api.crdVersion</code> | The Kubernetes API version ( <code>apiVersion</code> ) used to do the scaffolding for the CRD resource.   |
| <code>resources.api.namespaced</code> | The API RBAC permissions which can be namespaced or cluster scoped.   |
| <code>resources.controller</code>     | Indicates whether a controller was scaffolded for the API.  |

| Field  | Description  |
|--|--|
| <code>resources.domain</code>                  | The domain of the resource which was provided by the <code>--domain</code> flag when the project was initialized or via the flag <code>--external-api-domain</code> when it was used to scaffold controllers for an <a href="#">External Type</a> .  |
| <code>resources.group</code>                   | The GKV group of the resource which is provided by the <code>--group</code> flag when the sub-command <code>create api</code> is used.   |
| <code>resources.version</code>                 | The GKV version of the resource which is provided by the <code>--version</code> flag when the sub-command <code>create api</code> is used.   |
| <code>resources.kind</code>                    | Store GKV Kind of the resource which is provided by the <code>--kind</code> flag when the sub-command <code>create api</code> is used.   |
| <code>resources.path</code>                    | The import path for the API resource. It will be <code>&lt;repo&gt;/api/&lt;kind&gt;</code> unless the API added to the project is an external or core-type. For the core-types scenarios, the paths used are mapped <a href="#">here</a> . Or either the path informed by the flag <code>--external-api-path</code> |
| <code>resources.core</code>                    | It is <code>true</code> when the group used is from Kubernetes API and the API resource is not defined on the project.   |
| <code>resources.external</code>                | It is <code>true</code> when the flag <code>--external-api-path</code> was used to generated the scaffold for an <a href="#">External Type</a> .   |
| <code>resources.webhooks</code>                | Store the webhooks data when the sub-command <code>create webhook</code> is used.  |
| <code>resources.webhooks.spoke</code>          | Store the API version that will act as the Spoke with the designated Hub version for conversion webhooks.  |
| <code>resources.webhooks.webhookVersion</code> | The Kubernetes API version ( <code>apiVersion</code> ) used to scaffold the webhook resource.  |
| <code>resources.webhooks.conversion</code>     | It is <code>true</code> when the webhook was scaffold with the <code>--conversion</code> flag  |

| Field                                      | Description  |
|--|--|
|  | which means that is a conversion webhook.  |
| <code>resources.webhooks.defaulting</code> | It is <code>true</code> when the webhook was scaffold with the <code>--defaulting</code> flag which means that is a defaulting webhook.              |
| <code>resources.webhooks.validation</code> | It is <code>true</code> when the webhook was scaffold with the <code>--programmatic-validation</code> flag which means that is a validation webhook. |

## Plugins

Kubebuilder's architecture is fundamentally plugin-based. This design enables the Kubebuilder CLI to evolve while maintaining backward compatibility with older versions, allowing users to opt-in or opt-out of specific features, and enabling seamless integration with external tools.

By leveraging plugins, projects can extend Kubebuilder and use it as a library to support new functionalities or implement custom scaffolding tailored to their users' needs. This flexibility allows maintainers to build on top of Kubebuilder's foundation, adapting it to specific use cases while benefiting from its powerful scaffolding engine.

Plugins offer several key advantages:

- **Backward compatibility:** Ensures older layouts and project structures remain functional with newer versions.
- **Customization:** Allows users to opt-in or opt-out for specific features (i.e. [Grafana](#) and [Deploy Image](#) plugins)
- **Extensibility:** Facilitates integration with third-party tools and projects that wish to provide their own [External Plugins](#), which can be used alongside Kubebuilder to modify and enhance project scaffolding or introduce new features.

**For example, to initialize a project with multiple global plugins:**

```
kubebuilder init --plugins=pluginA,pluginB,pluginC
```

**For example, to apply custom scaffolding using specific plugins:**

```
kubebuilder create api --plugins=pluginA,pluginB,pluginC  
OR  
kubebuilder create webhook --plugins=pluginA,pluginB,pluginC  
OR  
kubebuilder edit --plugins=pluginA,pluginB,pluginC
```

This section details the available plugins, how to extend Kubebuilder, and how to create your own plugins while following the same layout structures.

- [Available Plugins](#)
- [Extending](#)
- [Plugins Versioning](#)

## Available plugins

This section describes the plugins supported and shipped in with the Kubebuilder project.

## To scaffold the projects

The following plugins are useful to scaffold the whole project with the tool.

| Plugin  | Key   | Description  |
|---|-------|--|
| <a href="#">go.kubebuilder.io/v4 - (Default scaffold with Kubebuilder init)</a> | go/v4 | Scaffold composite by <a href="#">base.go.kubebuilder.io/v4</a> and <a href="#">kustomize.common.kubebuilder.io/v2</a> . Responsible for scaffolding Golang projects and its configurations. |

## To add optional features

The following plugins are useful to generate code and take advantage of optional features

| Plugin  | Key                   | Description   |
|---|-----------------------|---|
| <a href="#">grafana.kubebuilder.io/v1-alpha</a>         | grafana/v1-alpha      | Optional helper plugin which can be used to scaffold Grafana Manifests Dashboards for the default metrics which are exported by controller-runtime. |
| <a href="#">deploy-image.go.kubebuilder.io/v1-alpha</a> | deploy-image/v1-alpha | Optional helper plugin which can be used to scaffold APIs and controller with code implementation to Deploy and Manage an Operand(image).           |
| <a href="#">helm.kubebuilder.io/v1-alpha</a>            | helm/v1-alpha         | Optional helper plugin which can be used to scaffold a Helm Chart to distribute the project under the <code>dist</code> directory                   |

## To be extended

The following plugins are useful for other tools and [External Plugins](#) which are looking to extend the Kubebuilder functionality.

You can use the `kustomize` plugin, which is responsible for scaffolding the `kustomize` files under `config/`. The base language plugins are responsible for scaffolding the necessary Golang files, allowing you to create your own plugins for other languages (e.g., [Operator-SDK](#) enables users to work with Ansible/Helm) or add additional functionality.

For example, [Operator-SDK](#) has a plugin which integrates the projects with [OLM](#) by adding its own features on top.

| Plugin   | Key                       | Description  |
|--|---------------------------|--|
| <a href="#">kustomize.common.kubebuilder.io/v2</a> | <code>kustomize/v2</code> | Responsible for scaffolding all <code>kustomize</code> files under the <code>config/</code> directory  |
| <a href="#">base.go.kubebuilder.io/v4</a>          | <code>base/v4</code>      | Responsible for scaffolding all files which specifically requires Golang. This plugin is used in the composition to create the plugin ( <code>go/v4</code> ) |

## (Default Scaffold)

Kubebuilder will scaffold using the `go/v4` plugin only if specified when initializing the project. This plugin is a composition of the `kustomize.common.kubebuilder.io/v2` and `base.go.kubebuilder.io/v4` plugins using the [Bundle Plugin](#). It scaffolds a project template that helps in constructing sets of [controllers](#).

By following the [quickstart](#) and creating any project, you will be using this plugin by default.

### Examples

You can check samples using this plugin by looking at the `project-v4-<options>` projects under the `testdata` directory on the root directory of the Kubebuilder project.

## How to use it ?

To create a new project with the `go/v4` plugin the following command can be used:

```
kubebuilder init --domain tutorial.kubebuilder.io --repo
tutorial.kubebuilder.io/project --plugins=go/v4
```

## Subcommands supported by the plugin

- Init - `kubebuilder init [OPTIONS]`
- Edit - `kubebuilder edit [OPTIONS]`
- Create API - `kubebuilder create api [OPTIONS]`
- Create Webhook - `kubebuilder create webhook [OPTIONS]`

## Further resources

- To see the composition of plugins, you can check the source code for the Kubebuilder [main.go](#).

- Check the code implementation of the base Golang plugin [base.go.kubebuilder.io/v4](https://base.go.kubebuilder.io/v4) .
- Check the code implementation of the Kustomize/v2 plugin.
- Check [controller-runtime](#) to know more about controllers.

## Grafana Plugin (grafana/v1-alpha)

The Grafana plugin is an optional plugin that can be used to scaffold Grafana Dashboards to allow you to check out the default metrics which are exported by projects using [controller-runtime](#).

### Examples

You can check its default scaffold by looking at the `project-v4-with-plugins` projects under the [testdata](#) directory on the root directory of the Kubebuilder project.

## When to use it ?

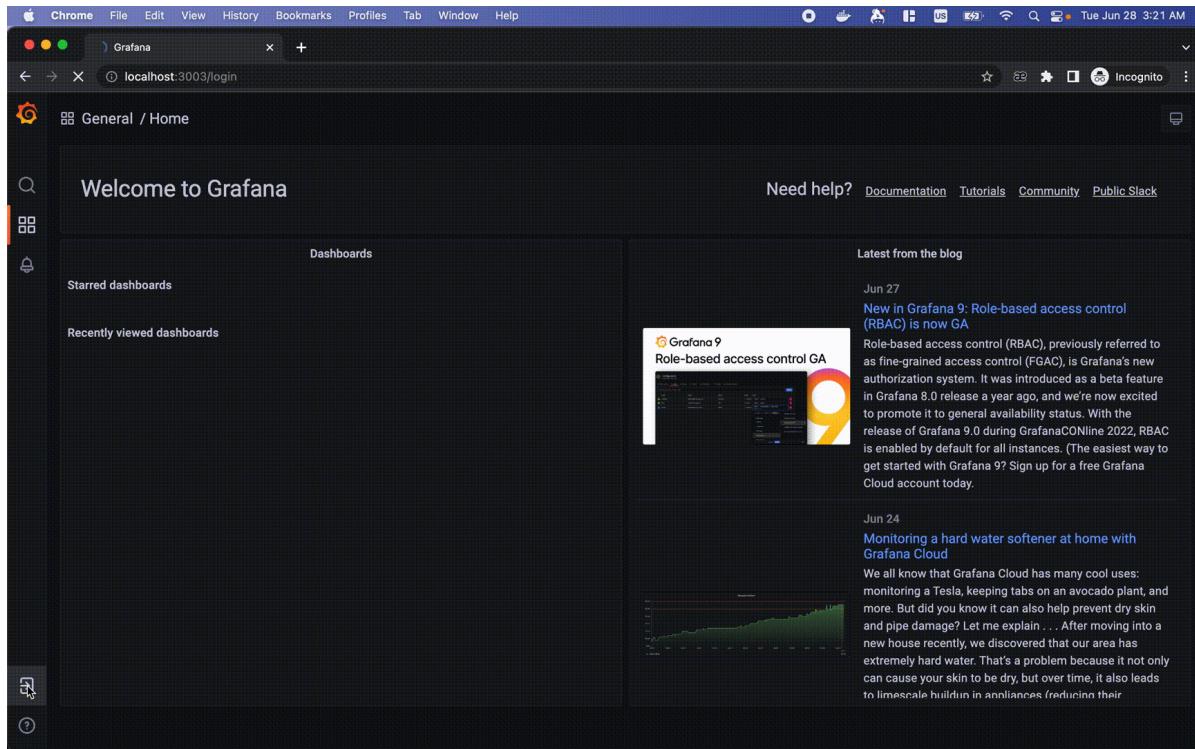
- If you are looking to observe the metrics exported by [controller metrics](#) and collected by Prometheus via [Grafana](#).

## How to use it ?

### Prerequisites:

- Your project must be using [controller-runtime](#) to expose the metrics via the [controller default metrics](#) and they need to be collected by Prometheus.
- Access to [Prometheus](#).
  - Prometheus should have an endpoint exposed. (For `prometheus-operator`, this is similar as: `http://prometheus-k8s.monitoring.svc:9090` )
  - The endpoint is ready to/already become the datasource of your Grafana. See [Add a data source](#)
- Access to [Grafana](#). Make sure you have:
  - [Dashboard edit permission](#)

- Prometheus Data source



Check the [metrics](#) to know how to enable the metrics for your projects scaffold with Kubebuilder.

See that in the [config/prometheus](#) you will find the ServiceMonitor to enable the metrics in the default endpoint `/metrics`.

## Basic Usage

The Grafana plugin is attached to the `init` subcommand and the `edit` subcommand:

```
# Initialize a new project with grafana plugin
kubebuilder init --plugins grafana.kubebuilder.io/v1-alpha

# Enable grafana plugin to an existing project
kubebuilder edit --plugins grafana.kubebuilder.io/v1-alpha
```

The plugin will create a new directory and scaffold the JSON files under it (i.e. `grafana/controller-runtime-metrics.json`).

### Show case:

See an example of how to use the plugin in your project:

The screenshot displays a Mac desktop environment. On the left, the iTerm2 application is open with two tabs: 'Home - Grafana' and another tab. The terminal session in the 'Home - Grafana' tab shows commands like 'grafana' and 'kubebuilder-with-grafana-plugin'. On the right, a web browser window is open to the Grafana home page at 'localhost:3003?orgId=1'. A 'Basic' tutorial panel is currently active, providing steps to quickly finish setting up your Grafana installation. Below the tutorial, there are sections for 'Dashboards' (Starred dashboards and Recently viewed dashboards) and 'Latest from the blog' (an article titled 'How to send logs to Grafana Loki with the OpenTelemetry Collector using Fluent Forward and Filelog receivers'). The status bar at the bottom of the screen shows system information including battery level (100%), signal strength, and the date and time (15:28 | 23 Jun).

## Now, let's check how to use the Grafana dashboards

1. Copy the JSON file
2. Visit `<your-grafana-url>/dashboard/import` to import a new dashboard.

3. Paste the JSON content to `Import via panel json`, then press Load button

The screenshot shows the "Import via panel json" section of the Grafana dashboard import interface. A blue box highlights the JSON code:

```
"time": {  
  "from": "now-15m",  
  "to": "now"  
},  
"timepicker": 0,  
"timezone": "",  
"title": "Controller-Runtime-Metrics",  
"weekStart": ""  
}
```

Below the JSON code is a blue "Load" button.

4. Select the data source for Prometheus metrics

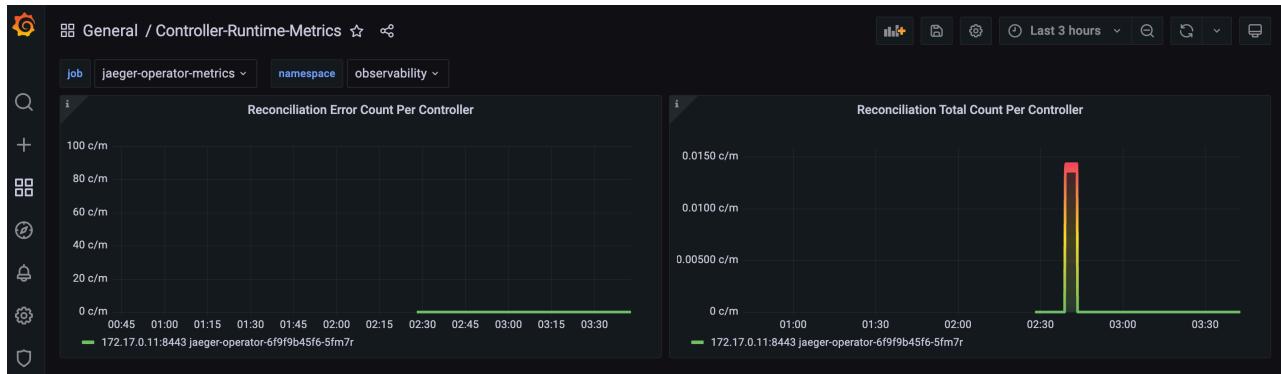
The screenshot shows the "Options" dialog for selecting a Prometheus data source. The "Prometheus" section is highlighted with a blue border. It contains a dropdown menu with the placeholder text "Select a Prometheus data source" and a single option: "prometheus".

5. Once the json is imported in Grafana, the dashboard is ready.

# Grafana Dashboard

## Controller Runtime Reconciliation total & errors

- Metrics:
  - controller\_runtime\_reconcile\_total
  - controller\_runtime\_reconcile\_errors\_total
- Query:
  - sum(rate(controller\_runtime\_reconcile\_total{job="\$job"}[5m])) by (instance, pod)
  - sum(rate(controller\_runtime\_reconcile\_errors\_total{job="\$job"}[5m])) by (instance, pod)
- Description:
  - Per-second rate of total reconciliation as measured over the last 5 minutes
  - Per-second rate of reconciliation errors as measured over the last 5 minutes
- Sample:



## Controller CPU & Memory Usage

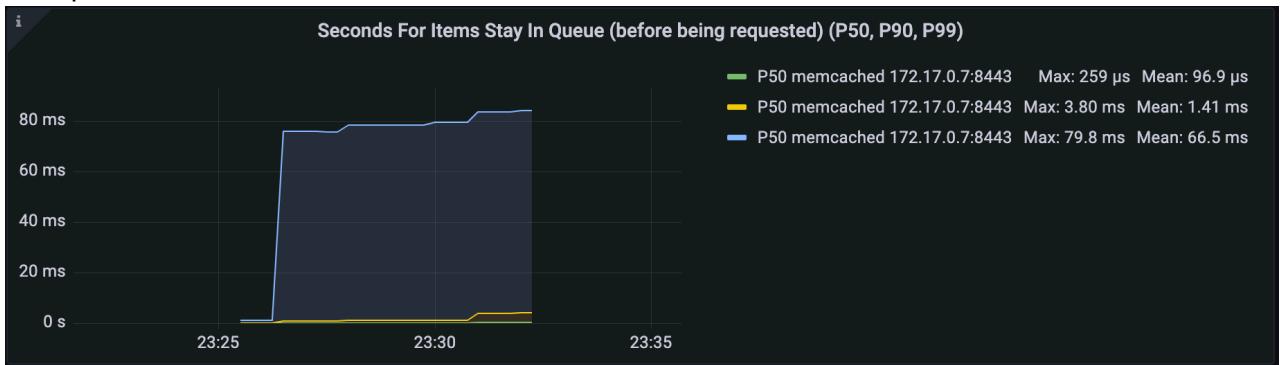
- Metrics:
  - process\_cpu\_seconds\_total
  - process\_resident\_memory\_bytes
- Query:
  - rate(process\_cpu\_seconds\_total{job="\$job", namespace="\$namespace", pod="\$pod"}[5m]) \* 100
  - process\_resident\_memory\_bytes{job="\$job", namespace="\$namespace", pod="\$pod"}
- Description:
  - Per-second rate of CPU usage as measured over the last 5 minutes
  - Allocated Memory for the running controller

- Sample:



## Seconds of P50/90/99 Items Stay in Work Queue

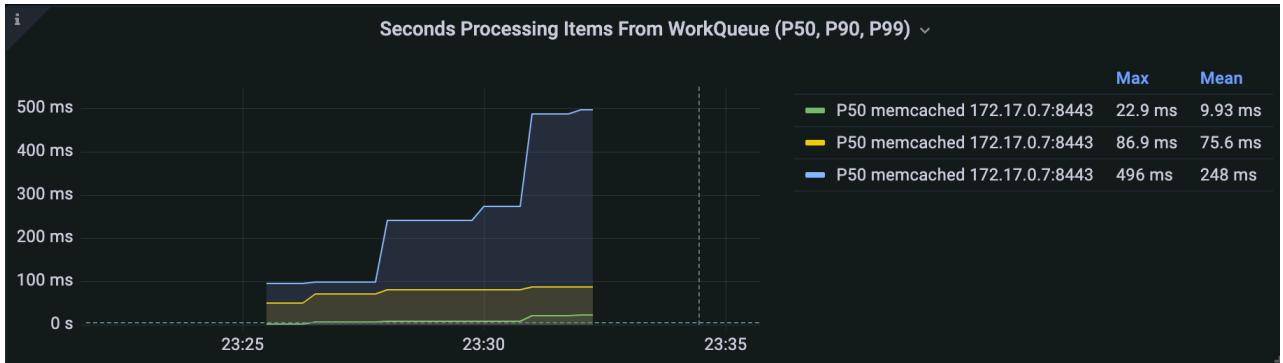
- Metrics
  - workqueue\_queue\_duration\_seconds\_bucket
- Query:
  - histogram\_quantile(0.50,  
sum(rate(workqueue\_queue\_duration\_seconds\_bucket{job="\$job",  
namespace="\$namespace"}[5m])) by (instance, name, le))
- Description
  - Seconds an item stays in workqueue before being requested.
- Sample:



## Seconds of P50/90/99 Items Processed in Work Queue

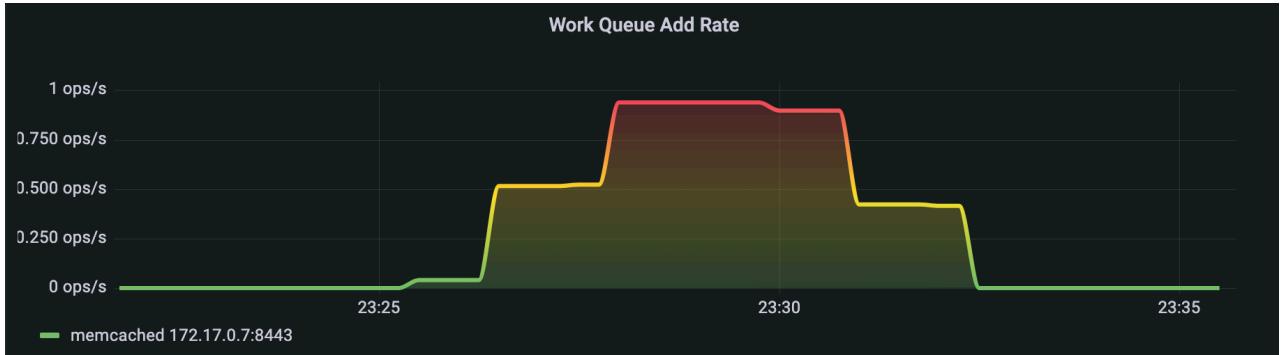
- Metrics
  - workqueue\_work\_duration\_seconds\_bucket
- Query:
  - histogram\_quantile(0.50,  
sum(rate(workqueue\_work\_duration\_seconds\_bucket{job="\$job",  
namespace="\$namespace"}[5m])) by (instance, name, le))
- Description
  - Seconds of processing an item from workqueue takes.

- Sample:



## Add Rate in Work Queue

- Metrics
  - workqueue\_adds\_total
- Query:
  - sum(rate(workqueue\_adds\_total{\$job="\$job", namespace="\$namespace"}[5m])) by (instance, name)
- Description
  - Per-second rate of items added to work queue
- Sample:



## Retries Rate in Work Queue

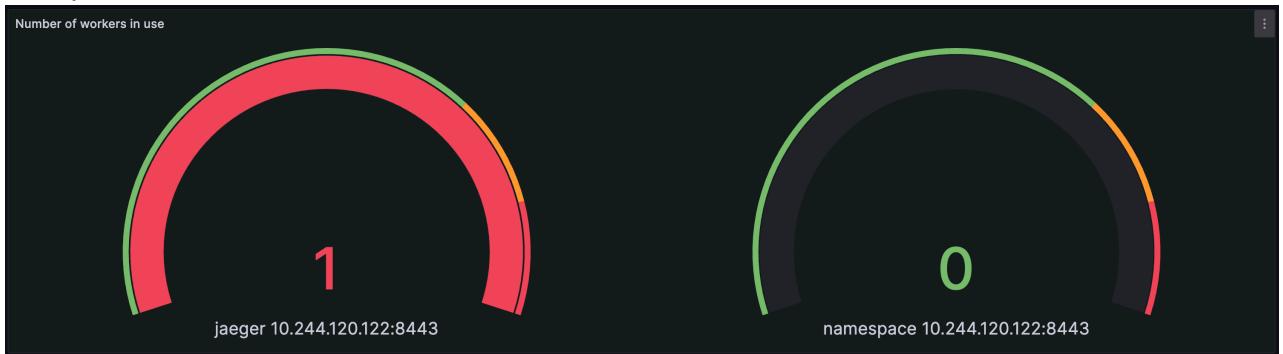
- Metrics
  - workqueue\_retries\_total
- Query:
  - sum(rate(workqueue\_retries\_total{\$job="\$job", namespace="\$namespace"}[5m])) by (instance, name)
- Description
  - Per-second rate of retries handled by workqueue

- Sample:



## Number of Workers in Use

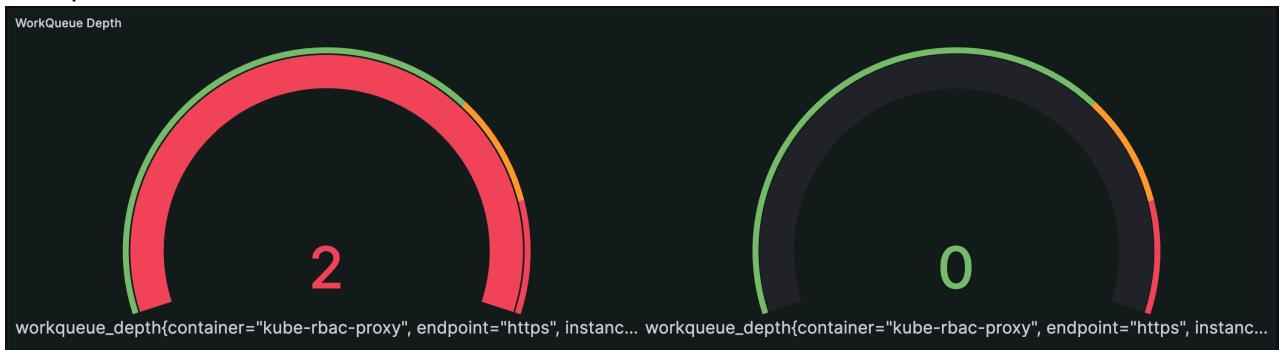
- Metrics
  - controller\_runtime\_active\_workers
- Query:
  - controller\_runtime\_active\_workers{job="\$job", namespace="\$namespace"}
- Description
  - The number of active controller workers
- Sample:



## WorkQueue Depth

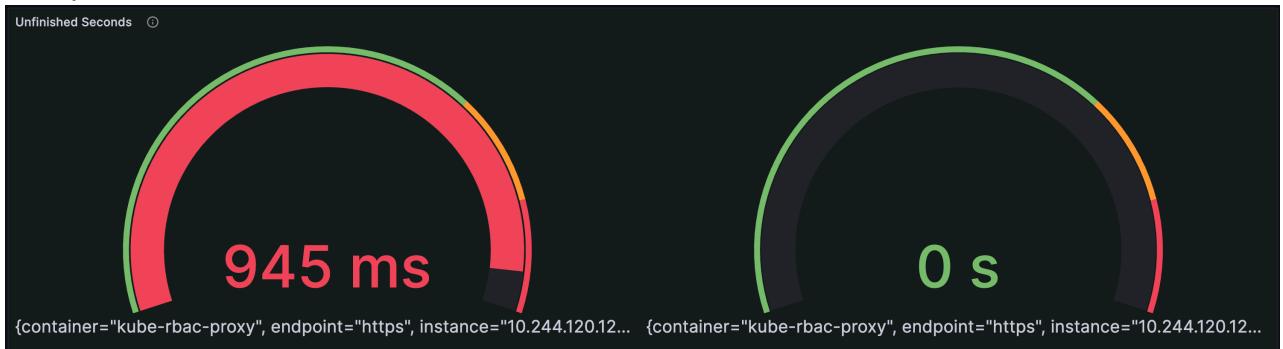
- Metrics
  - workqueue\_depth
- Query:
  - workqueue\_depth{job="\$job", namespace="\$namespace"}
- Description
  - Current depth of workqueue

- Sample:



## Unfinished Seconds

- Metrics
  - workqueue\_unfinished\_work\_seconds
- Query:
  - $\text{rate}(\text{workqueue\_unfinished\_work\_seconds}\{\text{job}=\text{"\$job"}, \text{namespace}=\text{"\$namespace"}\}[5m])$
- Description
  - How many seconds of work has done that is in progress and hasn't been observed by work\_duration.
- Sample:



## Visualize Custom Metrics

The Grafana plugin supports scaffolding manifests for custom metrics.

### Generate Config Template

When the plugin is triggered for the first time, `grafana/custom-metrics/config.yaml` is generated.

```
---
```

```
customMetrics:  
# - metric: # Raw custom metric (required)  
#   type:    # Metric type: counter/gauge/histogram (required)  
#   expr:    # Prom_ql for the metric (optional)  
#   unit:    # Unit of measurement, examples: s,none,bytes,percent,etc.  
(optional)
```

## Add Custom Metrics to Config

You can enter multiple custom metrics in the file. For each element, you need to specify the `metric` and its `type`. The Grafana plugin can automatically generate `expr` for visualization. Alternatively, you can provide `expr` and the plugin will use the specified one directly.

```
---
```

```
customMetrics:  
- metric: memcached_operator_reconcile_total # Raw custom metric (required)  
  type: counter # Metric type: counter/gauge/histogram (required)  
  unit: none  
- metric: memcached_operator_reconcile_time_seconds_bucket  
  type: histogram
```

## Scaffold Manifest

Once `config.yaml` is configured, you can run `kubebuilder edit --plugins grafana.kubebuilder.io/v1-alpha` again. This time, the plugin will generate `grafana/custom-metrics/custom-metrics-dashboard.json`, which can be imported to Grafana UI.

## Show case:

See an example of how to visualize your custom metrics:

```
..ched-operator (-zsh)          31          cool-gu kb 2 zsh (ssh)      32
> l
.rw-r--r-- 129 jintony 24 Mar 19:23 .dockerignore
.rw-r--r-- 367 jintony 24 Mar 19:23 .gitignore
drwxr-xr-x - jintony 24 Mar 19:23 api
drwxr-xr-x - jintony 1 Aug 17:30 bin
drwxr-xr-x - jintony 24 Mar 19:23 bundle
.rw-r--r-- 695 jintony 24 Mar 19:23 bundle.Dockerfile
drwxr-xr-x - jintony 24 Mar 19:23 config
drwxr-xr-x - jintony 5 Aug 11:53 controllers
.rw-r--r-- 10 jintony 5 Aug 12:07 cover.out
.rw-r--r-- 776 jintony 14 Jul 11:26 Dockerfile
.rw-r--r-- 3.9k jintony 14 Jul 11:26 go.mod
.rw-r--r-- 96k jintony 14 Jul 11:26 go.sum
drwxr-xr-x - jintony 24 Mar 19:23 hack
.rw-r--r-- 4.1k jintony 14 Jul 11:26 main.go
.rw-r--r-- 10k jintony 14 Jul 11:26 Makefile
.rw-r--r-- 504 jintony 26 Aug 10:42 PROJECT
.rw-r--r-- 2.7k jintony 14 Jul 11:26 README.md
drwxr-xr-x - jintony 14 Jul 11:26 test

Apple ~ /pro/operator-sdk/testdata/go/v3/memcached-operator tony/play-grafana-plugin    ⚡ 10:43:22
>
```

## Subcommands

The Grafana plugin implements the following subcommands:

- `edit( $ kubebuilder edit [OPTIONS] )`
- `init( $ kubebuilder init [OPTIONS] )`

## Affected files

The following scaffolds will be created or updated by this plugin:

- `grafana/*.json`

## Further resources

- Check out [video](#) to show how it works

- Checkout the [video](#) to show how the custom metrics feature works
- Refer to a sample of `serviceMonitor` provided by [kustomize](#) plugin
- Check the [plugin implementation](#)
- [Grafana Docs](#) of importing JSON file
- The usage of serviceMonitor by [Prometheus Operator](#)

## Deploy Image Plugin (deploy-image/v1-alpha)

The `deploy-image` plugin allows users to create [controllers](#) and custom resources that deploy and manage container images on the cluster, following Kubernetes best practices. It simplifies the complexities of deploying images while allowing users to customize their projects as needed.

By using this plugin, you will get:

- A controller implementation to deploy and manage an Operand (image) on the cluster.
- Tests to verify the reconciliation logic, using [ENVTEST](#).
- Custom resource samples updated with the necessary specifications.
- Environment variable support for managing the Operand (image) within the manager.

### Examples

See the `project-v4-with-plugins` directory under the `testdata` directory in the Kubebuilder project to check an example of scaffolding created using this plugin.

The `Memcached` API and its controller was scaffolded using the command:

```
kubebuilder create api \
--group example.com \
--version v1alpha1 \
--kind Memcached \
--image=memcached:memcached:1.6.26-alpine3.19 \
--image-container-command="memcached,--memory-limit=64,-o,modern,-v" \
--image-container-port="11211" \
--run-as-user="1001" \
--plugins="deploy-image/v1-alpha"
```

The `Busybox` API was created with:

```
kubebuilder create api \
--group example.com \
--version v1alpha1 \
--kind Busybox \
--image=busybox:1.36.1 \
--plugins="deploy-image/v1-alpha"
```

## When to use it?

- This plugin is ideal for users who are just getting started with Kubernetes operators.
- It helps users deploy and manage an image (Operand) using the [Operator pattern](#).
- If you're looking for a quick and efficient way to set up a custom controller and manage a container image, this plugin is a great choice.

## How to use it?

1. **Initialize your project:** After creating a new project with `kubebuilder init`, you can use this plugin to create APIs. Ensure that you've completed the [quick start guide](#) before proceeding.
2. **Create APIs:** With this plugin, you can [create APIs](#) to specify the image (Operand) you want to deploy on the cluster. You can also optionally specify the command, port, and security context using various flags:

Example command:

```
kubebuilder create api --group example.com --version v1alpha1 --kind Memcached --image=memcached:1.6.15-alpine --image-container-command="memcached,--memory-limit=64,modern,-v" --image-container-port="11211" --run-as-user="1001" --plugins="deploy-image/v1-alpha"
```

### Note on make run:

When running the project locally with `make run`, the Operand image provided will be stored as an environment variable in the `config/manager/manager.yaml` file.

Ensure you export the environment variable before running the project locally, such as:

```
export MEMCACHED_IMAGE="memcached:1.4.36-alpine"
```

## Subcommands

The `deploy-image` plugin includes the following subcommand:

- `create api` : Use this command to scaffold the API and controller code to manage the container image.

## Affected files

When using the `create api` command with this plugin, the following files are affected, in addition to the existing Kubebuilder scaffolding:

- `controllers/*_controller_test.go` : Scaffolds tests for the controller.
- `controllers/*_suite_test.go` : Scaffolds or updates the test suite.
- `api/<version>/*_types.go` : Scaffolds the API specs.
- `config/samples/*_.yaml` : Scaffolds default values for the custom resource.
- `main.go` : Updates the file to add the controller setup.
- `config/manager/manager.yaml` : Updates to include environment variables for storing the image.

## Further Resources:

- Check out this [video](#) to see how it works.

## Helm Plugin (helm/v1-alpha)

The Helm plugin is an optional plugin that can be used to scaffold a Helm chart, allowing you to distribute the project using Helm.

By default, users can generate a bundle with all the manifests by running the following command:

```
make build-installer IMG=<some-registry>/<project-name>:tag
```

This allows the project consumer to install the solution by applying the bundle with:

```
kubectl apply -f https://raw.githubusercontent.com/<org>/project-v4/<tag or branch>/dist/install.yaml
```

However, in many scenarios, you might prefer to provide a Helm chart to package your solution. If so, you can use this plugin to generate the Helm chart under the `dist` directory.

### Examples

You can check the plugin usage by looking at `project-v4-with-plugins` samples under the `testdata` directory on the root directory of the Kubebuilder project.

## When to use it

- If you want to provide a Helm chart for users to install and manage your project.
- If you need to update the Helm chart generated under `dist/chart/` with the latest project changes:
  - After generating new manifests, use the `edit` option to sync the Helm chart.
  - **IMPORTANT:** If you have created a webhook or an API using the `DeployImage` plugin, you must run the `edit` command with the `--force` flag to regenerate the Helm chart values based on the latest manifests (*after running make manifests*) to ensure that the HelmChart values are updated accordingly. In this case, if you have customized the files under `dist/chart/values.yaml`, and the `templates/manager/manager.yaml`, you will need to manually reapply your customizations on top of the latest changes after regenerating the Helm chart.

# How to use it ?

## Basic Usage

The Helm plugin is attached to the `init` subcommand and the `edit` subcommand:

```
# Initialize a new project with helm chart
kubebuilder init --plugins=helm/v1-alpha

# Enable or Update the helm chart via the helm plugin to an existing project
# Before run the edit command, run `make manifests` to generate the manifest
under `config/`
make manifests
kubebuilder edit --plugins=helm/v1-alpha
```

Use the `edit` command to update the Helm Chart with the latest changes

After making changes to your project, ensure that you run `make manifests` and then use the command `kubebuilder edit --plugins=helm/v1-alpha` to update the Helm Chart.

Note that the following files will **not** be updated unless you use the `--force` flag:

```
dist/chart/
├── values.yaml
└── templates/
    └── manager/
        └── manager.yaml
```

The files `chart/Chart.yaml`, `chart/templates/_helpers.tpl`, and `chart/.helmignore` are never updated after their initial creation unless you remove them.

## Subcommands

The Helm plugin implements the following subcommands:

- `edit( $ kubebuilder edit [OPTIONS] )`
- `init( $ kubebuilder init [OPTIONS] )`

## Affected files

The following scaffolds will be created or updated by this plugin:

- `dist/chart/*`

## (Default Scaffold)

The Kustomize plugin allows you to scaffold all kustomize manifests used with the language base plugin `base.go.kubebuilder.io/v4`. This plugin is used to generate the manifest under the `config/` directory for projects built within the `go/v4` plugin (default scaffold).

Projects like [Operator-sdk](#) use the Kubebuilder project as a library and provide options for working with other languages such as Ansible and Helm. The Kustomize plugin helps them maintain consistent configuration across languages. It also simplifies the creation of plugins that perform changes on top of the default scaffold, removing the need for manual updates across multiple language plugins. This approach allows the creation of “helper” plugins that work with different projects and languages.

### Examples

You can check the kustomize content by looking at the `config/` directory provided in the sample `project-v4-*` under the `testdata` directory of the Kubebuilder project.

## How to use it

If you want your language plugin to use kustomize, use the [Bundle Plugin](#) to specify that your language plugin is composed of your language-specific plugin and kustomize for its configuration, as shown:

```
import (
    ...
    kustomizecommonv2
    "sigs.k8s.io/kubebuilder/v4/pkg/plugins/common/kustomize/v2"
    golangv4 "sigs.k8s.io/kubebuilder/v4/pkg/plugins/golang/v4"
    ...
)

// Bundle plugin for Golang projects scaffolded by Kubebuilder go/v4
gov4Bundle, _ :=
    plugin.NewBundle(plugin.WithName(golang.DefaultNameQualifier),
        plugin.WithVersion(plugin.Version{Number: 4}),
        plugin.WithPlugins(kustomizecommonv2.Plugin{}, golangv4.Plugin{}), // Scaffold the config/ directory and all kustomize files
    )
```

You can also use kustomize/v2 alone via:

```
kubebuilder init --plugins=kustomize/v2
$ ls -la
total 24
drwxr-xr-x  6 camilamacedo86  staff  192 31 Mar 09:56 .
drwxr-xr-x 11 camilamacedo86  staff  352 29 Mar 21:23 ..
-rw-----  1 camilamacedo86  staff  129 26 Mar 12:01 .dockerignore
-rw-----  1 camilamacedo86  staff  367 26 Mar 12:01 .gitignore
-rw-----  1 camilamacedo86  staff   94 31 Mar 09:56 PROJECT
drwx----- 6 camilamacedo86  staff  192 31 Mar 09:56 config
```

Or combined with the base language plugins:

```
# Provides the same scaffold of go/v4 plugin which is composition but with
# kustomize/v2
kubebuilder init --plugins=kustomize/v2,base.go.kubebuilder.io/v4 --domain
example.org --repo example.org/guestbook-operator
```

## Subcommands

The kustomize plugin implements the following subcommands:

- `init ( $ kubebuilder init [OPTIONS] )`
- `create api ( $ kubebuilder create api [OPTIONS] )`
- `create webhook ( $ kubebuilder create api [OPTIONS] )`

### Create API and Webhook

The implementation for the `create api` subcommand scaffolds the kustomize manifests specific to each API. See more [here](#). The same applies to `create webhook`.

## Affected files

The following scaffolds will be created or updated by this plugin:

- `config/*`

## Further resources

- Check the kustomize [plugin implementation](#)
- Check the [kustomize documentation](#)
- Check the [kustomize repository](#)

## Extending Kubebuilder

Kubebuilder provides an extensible architecture to scaffold projects using plugins. These plugins allow you to customize the CLI behavior or integrate new features.

## Overview

Kubebuilder's CLI can be extended through custom plugins, allowing you to:

- Build new scaffolds.
- Enhance existing ones.
- Add new commands and functionality to Kubebuilder's scaffolding.

This flexibility enables you to create custom project setups tailored to specific needs.

### Why use the Kubebuilder style?

Kubebuilder and SDK are both broadly adopted projects which leverage the [controller-runtime](#) project. They both allow users to build solutions using the [Operator Pattern](#) and follow common standards.

Adopting these standards can bring significant benefits, such as joining forces on maintaining the common standards as the features provided by Kubebuilder and take advantage of the contributions made by the community. This allows you to focus on the specific needs and requirements for your plugin and use-case.

And then, you will also be able to use custom plugins and options currently or in the future which might be provided by these projects as any other which decides to persuade the same standards.

## Options to Extend

Extending Kubebuilder can be achieved in two main ways:

1. **Extending CLI features and Plugins:** You can import and build upon existing Kubebuilder plugins to [extend its features and plugins](#). This is useful when you need to add specific features to a tool that already benefits from Kubebuilder's scaffolding system. For example, [Operator SDK](#) leverages the [kustomize plugin](#) to provide

language support for tools like Ansible or Helm. So that the project can be focused to keep maintained only what is specific language based.

2. **Creating External Plugins:** You can build standalone, independent plugins as binaries. These plugins can be written in any language and should follow an execution pattern that Kubebuilder recognizes. For more information, see [Creating external plugins](#).

For further details on how to extend Kubebuilder, explore the following sections:

- [CLI and Plugins](#) to learn how to extend CLI features and plugins.
- [External Plugins](#) for creating standalone plugins.
- [E2E Tests](#) to ensure your plugin functions as expected.

Kubebuilder provides an extensible architecture to scaffold projects using plugins. These plugins allow you to customize the CLI behavior or integrate new features.

In this guide, we'll explore how to extend CLI features, create custom plugins, and bundle multiple plugins.

## Creating Custom Plugins

To create a custom plugin, you need to implement the [Kubebuilder Plugin interface](#).

This interface allows your plugin to hook into Kubebuilder's commands (`init`, `create api`, `create webhook`, etc.) and add custom logic.

## Example of a Custom Plugin

You can create a plugin that generates both language-specific scaffolds and the necessary configuration files, using the [Bundle Plugin](#). This example shows how to combine the Golang plugin with a Kustomize plugin:

```
import (
    kustomizecommonv2
"sigs.k8s.io/kubebuilder/v4/pkg/plugins/common/kustomize/v2"
    golangv4 "sigs.k8s.io/kubebuilder/v4/pkg/plugins/golang/v4"
)

mylanguagev1Bundle, _ := plugin.NewBundle(
    plugin.WithName("mylanguage.kubebuilder.io"),
    plugin.WithVersion(plugin.Version{Number: 1}),
    plugin.WithPlugins(kustomizecommonv2.Plugin{}, mylanguagev1.Plugin{}),
)
```

This composition allows you to scaffold a common configuration base (via Kustomize) and the language-specific files (via `mylanguagev1`).

You can also use your plugin to scaffold specific resources like CRDs and controllers, using the `create api` and `create webhook` subcommands.

## Plugin Subcommands

Plugins are responsible for implementing the code that will be executed when the sub-commands are called. You can create a new plugin by implementing the [Plugin interface](#).

On top of being a `Base`, a plugin should also implement the [SubcommandMetadata](#) interface so it can be run with a CLI. Optionally, a custom help text for the target command can be set; this method can be a no-op, which will preserve the default help text set by the [cobra](#) command constructors.

Kubebuilder CLI plugins wrap scaffolding and CLI features in conveniently packaged Go types that are executed by the `kubebuilder` binary, or any binary which imports them. More specifically, a plugin configures the execution of one of the following CLI commands:

- `init`: Initializes the project structure.
- `create api`: Scaffolds a new API and controller.
- `create webhook`: Scaffolds a new webhook.
- `edit`: edit the project structure.

Here's an example of using the `init` subcommand with a custom plugin:

```
kubebuilder init --plugins=mylanguage.kubebuilder.io/v1
```

This would initialize a project using the `mylanguage` plugin.

## Plugin Keys

Plugins are identified by a key of the form `<name>/<version>`. There are two ways to specify a plugin to run:

- Setting `kubebuilder init --plugins=<plugin key>`, which will initialize a project configured for plugin with key `<plugin key>`.
- A `layout: <plugin key>` in the scaffolded [PROJECT configuration file](#). Commands (except for `init`, which scaffolds this file) will look at this value before running to choose which plugin to run.

By default, `<plugin key>` will be `go.kubebuilder.io/vX`, where `X` is some integer.

For a full implementation example, check out Kubebuilder's native `go.kubebuilder.io` plugin.

## Plugin naming

Plugin names must be DNS1123 labels and should be fully qualified, i.e. they have a suffix like `.example.com`. For example, the base Go scaffold used with `kubebuilder` commands has name `go.kubebuilder.io`. Qualified names prevent conflicts between plugin names; both `go.kubebuilder.io` and `go.example.com` can both scaffold Go code and can be specified by a user.

## Plugin versioning

A plugin's `Version()` method returns a `plugin.Version` object containing an integer value and optionally a stage string of either "alpha" or "beta". The integer denotes the current version of a plugin. Two different integer values between versions of plugins indicate that the two plugins are incompatible. The stage string denotes plugin stability:

- `alpha` : should be used for plugins that are frequently changed and may break between uses.
- `beta` : should be used for plugins that are only changed in minor ways, ex. bug fixes.

## Boilerplates

The Kubebuilder internal plugins use boilerplates to generate the files of code. Kubebuilder uses templating to scaffold files for plugins. For instance, when creating a new project, the `go/v4` plugin scaffolds the `go.mod` file using a template defined in its implementation.

You can extend this functionality in your custom plugin by defining your own templates and using [Kubebuilder's machinery library](#) to generate files. This library allows you to:

- Define file I/O behaviors.
- Add [markers](#) to the scaffolded files.
- Specify templates for your scaffolds.

### Example: Boilerplate

For instance, the `go/v4` scaffolds the `go.mod` file by defining an object that [implements the machinery interface](#). The raw template is set to the `TemplateBody` field on the `Template.SetTemplateDefaults` method:

```
/*
Copyright 2022 The Kubernetes Authors.

Licensed under the Apache License, Version 2.0 (the "License");
you may not use this file except in compliance with the License.
You may obtain a copy of the License at

    http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software
distributed under the License is distributed on an "AS IS" BASIS,
WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
See the License for the specific language governing permissions and
limitations under the License.
*/

package templates

import (
    "sigs.k8s.io/kubebuilder/v4/pkg/machinery"
)

var _ machinery.Template = &GoMod{}


// GoMod scaffolds a file that defines the project dependencies
type GoMod struct {
    machinery.TemplateMixin
    machinery.RepositoryMixin

    ControllerRuntimeVersion string
}

// SetTemplateDefaults implements machinery.Template
func (f *GoMod) SetTemplateDefaults() error {
    if f.Path == "" {
        f.Path = "go.mod"
    }

    f.TemplateBody = goModTemplate

    f.IfExistsAction = machinery.OverwriteFile

    return nil
}

const goModTemplate = `module {{ .Repo }}

go 1.23.0

godebug default=go1.23

require (
    sigs.k8s.io/controller-runtime {{ .ControllerRuntimeVersion }}`
```

)

Such object that implements the machinery interface will later pass to the execution of scaffold:

```
// Scaffold implements cmdutil.Scaffolder
func (s *initScaffolder) Scaffold() error {
    log.Println("Writing scaffold for you to edit...")

    // Initialize the machinery.Scaffold that will write the boilerplate file
    // to disk
    // The boilerplate file needs to be scaffolded as a separate step as it is
    // going to
    // be used by the rest of the files, even those scaffolded in this command
    // call.
    scaffold := machinery.NewScaffold(s.fs,
        machinery.WithConfig(s.config),
    )

    ...

    return scaffold.Execute(
        ...
        &templates.GoMod{
            ControllerRuntimeVersion: ControllerRuntimeVersion,
        },
        ...
    )
}
```

## Example: Overwriting a File in a Plugin

Let's imagine that when a subcommand is called, you want to overwrite an existing file.

For example, to modify the `Makefile` and add custom build steps, in the definition of your Template you can use the following option:

```
f.IfExistsAction = machinery.OverwriteFile
```

By using those options, your plugin can take control of certain files generated by Kubebuilder's default scaffolds.

# Customizing Existing Scaffolds

Kubebuilder provides utility functions to help you modify the default scaffolds. By using the [plugin utilities](#), you can insert, replace, or append content to files generated by Kubebuilder, giving you full control over the scaffolding process.

These utilities allow you to:

- **Insert content:** Add content at a specific location within a file.
- **Replace content:** Search for and replace specific sections of a file.
- **Append content:** Add content to the end of a file without removing or altering the existing content.

## Example

If you need to insert custom content into a scaffolded file, you can use the `InsertCode` function provided by the plugin utilities:

```
pluginutil.InsertCode(filename, target, code)
```

This approach enables you to extend and modify the generated scaffolds while building custom plugins.

For more details, refer to the [Kubebuilder plugin utilities](#).

# Bundle Plugin

Plugins can be bundled to compose more complex scaffolds. A plugin bundle is a composition of multiple plugins that are executed in a predefined order. For example:

```
myPluginBundle, _ := plugin.NewBundle(
    plugin.WithName("myplugin.example.com"),
    plugin.WithVersion(plugin.Version{Number: 1}),
    plugin.WithPlugins(pluginA.Plugin{}, pluginB.Plugin{}, pluginC.Plugin{}),
)
```

This bundle will execute the `init` subcommand for each plugin in the specified order:

1. pluginA
2. pluginB
3. pluginC

The following command will run the bundled plugins:

```
kubebuilder init --plugins=myplugin.example.com/v1
```

## CLI system

Plugins are run using a [CLI](#) object, which maps a plugin type to a subcommand and calls that plugin's methods. For example, writing a program that injects an `Init` plugin into a `CLI` then calling `CLI.Run()` will call the plugin's [SubcommandMetadata](#), [UpdatesMetadata](#) and `Run` methods with information a user has passed to the program in `kubebuilder init`. Following an example:

```
package cli

import (
    log "github.com/sirupsen/logrus"
    "github.com/spf13/cobra"

    "sigs.k8s.io/kubebuilder/v4/pkg/cli"
    cfgv3 "sigs.k8s.io/kubebuilder/v4/pkg/config/v3"
    "sigs.k8s.io/kubebuilder/v4/pkg/plugin"
    kustomizecommonv2
    "sigs.k8s.io/kubebuilder/v4/pkg/plugins/common/kustomize/v2"
    "sigs.k8s.io/kubebuilder/v4/pkg/plugins/golang"
    deployimage "sigs.k8s.io/kubebuilder/v4/pkg/plugins/golang/deploy-
image/v1alpha1"
    golangv4 "sigs.k8s.io/kubebuilder/v4/pkg/plugins/golang/v4"
)

var (
    // The following is an example of the commands
    // that you might have in your own binary
    commands = []*cobra.Command{
        myExampleCommand.NewCmd(),
    }
    alphaCommands = []*cobra.Command{
        myExampleAlphaCommand.NewCmd(),
    }
)

// GetPluginsCLI returns the plugins based CLI configured to be used in your
// CLI binary
func GetPluginsCLI() (*cli.CLI) {
    // Bundle plugin which built the golang projects scaffold by Kubebuilder
    go/v4
    gov3Bundle, _ := plugin.NewBundleWithOptions(plugin.WithName(golang.DefaultNameQualifier),
        plugin.WithVersion(plugin.Version{Number: 3}),
        plugin.WithPlugins(kustomizecommonv2.Plugin{}, golangv4.Plugin{}),
    )

    c, err := cli.New(
        // Add the name of your CLI binary
        cli.WithCommandName("example-cli"),
        // Add the version of your CLI binary
        cli.WithVersion(versionString()),

        // Register the plugins options which can be used to do the scaffolds
        // via your CLI tool. See that we are using as example here the plugins which are
        // implemented and provided by Kubebuilder
        cli.WithPlugins(
            gov3Bundle,
```

```

        &deployimage.Plugin{},
    ),

    // Defines what will be the default plugin used by your binary. It
    means that will be the plugin used if no info be provided such as when the
    user runs `kubebuilder init`
    cli.WithDefaultPlugins(cfgv3.Version, gov3Bundle),

    // Define the default project configuration version which will be used
    by the CLI when none is informed by --project-version flag.
    cli.WithDefaultProjectVersion(cfgv3.Version),

    // Adds your own commands to the CLI
    cli.WithExtraCommands(commands...),

    // Add your own alpha commands to the CLI
    cli.WithExtraAlphaCommands(alphaCommands...),

    // Adds the completion option for your CLI
    cli.WithCompletion(),
)
if err != nil {
    log.Fatal(err)
}

return c
}

// versionString returns the CLI version
func versionString() string {
    // return your binary project version
}

```

This program can then be built and run in the following ways:

Default behavior:

```

# Initialize a project with the default Init plugin, "go.example.com/v1".
# This key is automatically written to a PROJECT config file.
$ my-bin-builder init
# Create an API and webhook with "go.example.com/v1" CreateAPI and
# CreateWebhook plugin methods. This key was read from the config file.
$ my-bin-builder create api [flags]
$ my-bin-builder create webhook [flags]

```

Selecting a plugin using --plugins :

```
# Initialize a project with the "ansible.example.com/v1" Init plugin.  
# Like above, this key is written to a config file.  
$ my-bin-builder init --plugins ansible  
# Create an API and webhook with "ansible.example.com/v1" CreateAPI  
# and CreateWebhook plugin methods. This key was read from the config file.  
$ my-bin-builder create api [flags]  
$ my-bin-builder create webhook [flags]
```

## Inputs should be tracked in the PROJECT file

The CLI is responsible for managing the [PROJECT file configuration](#), which represents the configuration of the projects scaffolded by the CLI tool.

When extending Kubebuilder, it is recommended to ensure that your tool or [External Plugin](#) properly uses the [PROJECT file](#) to track relevant information. This ensures that other external tools and plugins can properly integrate with the project. It also allows tools features to help users re-scaffold their projects such as the [Project Upgrade Assistant](#) provided by Kubebuilder, ensuring the tracked information in the PROJECT file can be leveraged for various purposes.

For example, plugins can check whether they support the project setup and re-execute commands based on the tracked inputs.

## Example

By running the following command to use the [Deploy Image](#) plugin to scaffold an API and its controller:

```
kubebyilder create api --group example.com --version v1alpha1 --kind Memcached  
--image=memcached:memcached:1.6.26-alpine3.19 --image-container-  
command="memcached,--memory-limit=64,-o,modern,-v" --image-container-  
port="11211" --run-as-user="1001" --plugins="deploy-image/v1-alpha" --  
make=false
```

The following entry would be added to the PROJECT file:

```
...
plugins:
  deploy-image.go.kubebuilder.io/v1-alpha:
    resources:
      - domain: testproject.org
        group: example.com
        kind: Memcached
        options:
          containerCommand: memcached,--memory-limit=64,-o,modern,-v
          containerPort: "11211"
          image: memcached:memcached:1.6.26-alpine3.19
          runAsUser: "1001"
          version: v1alpha1
      - domain: testproject.org
        group: example.com
        kind: Busybox
        options:
          image: busybox:1.36.1
        version: v1alpha1
...
...
```

By inspecting the PROJECT file, it becomes possible to understand how the plugin was used and what inputs were provided. This not only allows re-execution of the command based on the tracked data but also enables creating features or plugins that can rely on this information.

## Overview

Kubebuilder's functionality can be extended through external plugins. These plugins are executables (written in any language) that follow an execution pattern recognized by Kubebuilder. Kubebuilder interacts with these plugins via `stdin` and `stdout`, enabling seamless communication.

## Why Use External Plugins?

External plugins enable third-party solution maintainers to integrate their tools with Kubebuilder. Much like Kubebuilder's own plugins, these can be opt-in, offering users flexibility in tool selection. By developing plugins in their repositories, maintainers ensure updates are aligned with their CI pipelines and can manage any changes within their domain of responsibility.

If you are interested in this type of integration, collaborating with the maintainers of the third-party solution is recommended. Kubebuilder's maintainers are always willing to provide support in extending its capabilities.

## How to Write an External Plugin

Communication between Kubebuilder and an external plugin occurs via standard I/O. Any language can be used to create the plugin, as long as it follows the [PluginRequest](#) and [PluginResponse](#) structures.

### PluginRequest

`PluginRequest` contains the data collected from the CLI and any previously executed plugins. Kubebuilder sends this data as a JSON object to the external plugin via `stdin`.

**Example `PluginRequest` (triggered by `kubebuilder init --plugins sampleexternalplugin/v1 --domain my.domain`):**

```
{  
    "apiVersion": "v1alpha1",  
    "args": ["--domain", "my.domain"],  
    "command": "init",  
    "universe": {}  
}
```

## PluginResponse

`PluginResponse` contains the modifications made by the plugin to the project. This data is serialized as JSON and returned to Kubebuilder through `stdout`.

**Example `PluginResponse`:**

```
{  
    "apiVersion": "v1alpha1",  
    "command": "init",  
    "metadata": {  
        "description": "The `init` subcommand initializes a project via  
Kubebuilder. It scaffolds a single file: `initFile`.",  
        "examples": "kubebuilder init --plugins sampleexternalplugin/v1 --domain  
my.domain"  
    },  
    "universe": {  
        "initFile": "A file created with the `init` subcommand."  
    },  
    "error": false,  
    "errorMsgs": []  
}
```

Avoid printing directly to `stdout` in your external plugin. Since communication between Kubebuilder and the plugin occurs through `stdin` and `stdout` using structured JSON, any unexpected output (like debug logs) may cause errors. Write logs to a file if needed.

## How to Use an External Plugin

### Prerequisites

- Kubebuilder CLI version > 3.11.0
- An executable for the external plugin
- Plugin path configuration using `.getExternalPluginsPath` or default OS-based paths:

- Linux: \${HOME}/.config/kubebuilder/plugins/\${name}/\${version}/\${name}
- macOS: ~/Library/Application Support/kubebuilder/plugins/\${name}/\${version}/\${name}

**Example:** For a plugin `foo.acme.io` version `v2` on Linux, the path would be `${HOME}/.config/kubebuilder/plugins/foo.acme.io/v2/foo.acme.io`.

## Available Subcommands

External plugins can support the following Kubebuilder subcommands:

- `init`: Project initialization
- `create api`: Scaffold Kubernetes API definitions
- `create webhook`: Scaffold Kubernetes webhooks
- `edit`: Update project configuration

### Optional subcommands for enhanced user experience:

- `metadata`: Provide plugin descriptions and examples with the `--help` flag.
- `flags`: Inform Kubebuilder of supported flags, enabling early error detection.

#### More about `flags` subcommand

The `flags` subcommand in an external plugin allows for early error detection by informing Kubebuilder about the flags the plugin supports. If an unsupported flag is identified, Kubebuilder can issue an error before the plugin is called to execute. If a plugin does not implement the `flags` subcommand, Kubebuilder will pass all flags to the plugin, making it the external plugin's responsibility to handle any invalid flags.

## Configuring Plugin Path

Set the environment variable `$EXTERNAL_PLUGINS_PATH` to specify a custom plugin binary path:

```
export EXTERNAL_PLUGINS_PATH=<custom-path>
```

Otherwise, Kubebuilder would search for the plugins in a default path based on your OS.

## Example CLI Commands

You can now use it by calling the CLI commands:

```
# Initialize a new project with the external plugin named `sampleplugin`  
kubebuilder init --plugins sampleplugin/v1  
  
# Display help information of the `init` subcommand of the external plugin  
kubebuilder init --plugins sampleplugin/v1 --help  
  
# Create a new API with the above external plugin with a customized flag  
`number`  
kubebuilder create api --plugins sampleplugin/v1 --number 2  
  
# Create a webhook with the above external plugin with a customized flag  
`hooked`  
kubebuilder create webhook --plugins sampleplugin/v1 --hooked  
  
# Update the project configuration with the above external plugin  
kubebuilder edit --plugins sampleplugin/v1  
  
# Create new APIs with external plugins v1 and v2 by respecting the plugin  
chaining order  
kubebuilder create api --plugins sampleplugin/v1,sampleplugin/v2  
  
# Create new APIs with the go/v4 plugin and then pass those files to the  
external plugin by respecting the plugin chaining order  
kubebuilder create api --plugins go/v4,sampleplugin/v1
```

## Further resources

- A [sample external plugin written in Go](#)
- A [sample external plugin written in Python](#)
- A [sample external plugin written in JavaScript](#)

## Write E2E Tests

You can check the [Kubebuilder/v4/test/e2e/utils](#) package, which offers `TestContext` with rich methods:

- [NewTestContext](#) helps define:
  - A temporary folder for testing projects.
  - A temporary controller-manager image.
  - The [Kubectl execution method](#).
  - The CLI executable (whether `kubebuilder`, `operator-sdk`, or your extended CLI).

Once defined, you can use `TestContext` to:

### 1. Setup the testing environment, e.g.:

- Clean up the environment and create a temporary directory. See [Prepare](#).
- Install prerequisite CRDs. See [InstallCertManager](#), [InstallPrometheusManager](#).

### 2. Validate the plugin behavior, e.g.:

- Trigger the plugin's bound subcommands. See [Init](#), [CreateAPI](#).
- Use [PluginUtil](#) to verify scaffolded outputs. See [InsertCode](#), [ReplaceInFile](#), [UncommentCode](#).

### 3. Ensure the scaffolded output works, e.g.:

- Execute commands in your `Makefile`. See [Make](#).
- Temporarily load an image of the testing controller. See [LoadImageToKindCluster](#).
- Call Kubectl to validate running resources. See [Kubectl](#).

### 4. Cleanup temporary resources after testing:

- Uninstall prerequisite CRDs. See [UninstallPrometheusOpenManager](#).
- Delete the temporary directory. See [Destroy](#).

## References:

- [operator-sdk e2e tests](#)
- [kubebuilder e2e tests](#)

# Generate Test Samples

It's straightforward to view the content of sample projects generated by your plugin.

For example, KubeBuilder generates [sample projects](#) based on different plugins to validate the layouts.

You can also use `TestContext` to generate folders of scaffolded projects from your plugin. The commands are similar to those mentioned in [Extending CLI Features and Plugins](#).

Here's a general workflow to create a sample project using the `go/v4` plugin (`kbc` is an instance of `TestContext`):

- **To initialize a project:**

```
By("initializing a project")
err = kbc.Init(
    "--plugins", "go/v4",
    "--project-version", "3",
    "--domain", kbc.Domain,
    "--fetch-deps=false",
)
Expect(err).NotTo(HaveOccurred(), "Failed to initialize a project")
```

- **To define API:**

```
By("creating API definition")
err = kbc.CreateAPI(
    "--group", kbc.Group,
    "--version", kbc.Version,
    "--kind", kbc.Kind,
    "--namespaced",
    "--resource",
    "--controller",
    "--make=false",
)
Expect(err).NotTo(HaveOccurred(), "Failed to create an API")
```

- **To scaffold webhook configurations:**

```
By("scaffolding mutating and validating webhooks")
err = kbc.CreateWebhook(
    "--group", kbc.Group,
    "--version", kbc.Version,
    "--kind", kbc.Kind,
    "--defaulting",
    "--programmatic-validation",
)
Expect(err).NotTo(HaveOccurred(), "Failed to create an webhook")
```

## Plugins Versioning

| Name                | Example                           | Description   |
|---------------------|-----------------------------------|---|
| Kubebuilder version | v2.2.0 , v2.3.0 , v2.3.1 , v4.2.0 | Tagged versions of the Kubebuilder project, representing changes to the source code in this repository. See the <a href="#">releases</a> page for binary releases.  |
| Project version     | "1" , "2" , "3"                   | Project version defines the scheme of a <code>PROJECT</code> configuration file. This version is defined in a <code>PROJECT</code> file's <code>version</code> .  |
| Plugin version      | v2 , v3 , v4                      | Represents the version of an individual plugin, as well as the corresponding scaffolding that it generates. This version is defined in a plugin key, ex. <code>go.kubebuilder.io/v2</code> . See the <a href="#">design doc</a> for more details. |

## Incrementing versions

For more information on how Kubebuilder release versions work, see the [semver](#) documentation.

Project versions should only be increased if a breaking change is introduced in the `PROJECT` file scheme itself. Changes to the Go scaffolding or the Kubebuilder CLI *do not* affect project version.

Similarly, the introduction of a new plugin version might only lead to a new minor version release of Kubebuilder, since no breaking change is being made to the CLI itself. It'd only be a breaking change to Kubebuilder if we remove support for an older plugin version. See the plugins design doc [versioning section](#) for more details on plugin versioning.

## Introducing changes to plugins

Changes made to plugins only require a plugin version increase if and only if a change is made to a plugin that breaks projects scaffolded with the previous plugin version. Once a plugin version `vx` is stabilized (it doesn't have an "alpha" or "beta" suffix), a new plugin package should be created containing a new plugin with version `v(X+1)-alpha`. Typically this is done by (semantically) `cp -r pkg/plugins/golang/vX pkg/plugins/golang/v(X+1)` then updating version numbers and paths. All further breaking changes to the plugin should be made in this package; the `vx` plugin would then be frozen to breaking changes.

You must also add a migration guide to the [migrations](#) section of the Kubebuilder book in your PR. It should detail the steps required for users to upgrade their projects from `vX` to `v(X+1)-alpha`.

### Example

Kubebuilder scaffolds projects with plugin `go.kubebuilder.io/v4` by default.

You create a feature that adds a new marker to the file `main.go` scaffolded by `init` that `create api` will use to update that file. The changes introduced in your feature would cause errors if used with projects built with plugins `go.kubebuilder.io/v4` without users manually updating their projects.

Thus, your changes introduce a breaking change to plugin `go.kubebuilder.io`, and can only be merged into plugin version `v5-alpha`. This plugin's package should exist already.

### Controller-Runtime FAQ

Kubebuilder is developed on top of the [controller-runtime](#) and [controller-tools](#) libraries. We recommend you also check the [Controller-Runtime FAQ page](#).

## How does the value informed via the domain flag (i.e. `kubebuilder init --domain example.com`) when we init a project?

After creating a project, usually you will want to extend the Kubernetes APIs and define new APIs which will be owned by your project. Therefore, the domain value is tracked in the [PROJECT](#) file which defines the config of your project and will be used as a domain to create the endpoints of your API(s). Please, ensure that you understand the [Groups and Versions and Kinds, oh my!](#).

The domain is for the group suffix, to explicitly show the resource group category. For example, if set `--domain=example.com`:

```
kubebuilder init --domain example.com --repo xxx --plugins=go/v4  
kubebuilder create api --group mygroup --version v1beta1 --kind Mykind
```

Then the result resource group will be `mygroup.example.com`.

---

If domain field not set, the default value is `my.domain`.

---

## I'd like to customize my project to use [klog](#) instead of the [zap](#) provided by controller-runtime. How to use [klog](#) or other loggers as the project logger?

In the `main.go` you can replace:

```
opts := zap.Options{
Development: true,
}
opts.BindFlags(flag.CommandLine)
flag.Parse()

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

ctrl.SetLogger(klog.NewKlogr())
```

with:

```
flag.Parse()
ctrl.SetLogger(klog.NewKlogr())
```

## After make run, I see errors like “unable to find leader election namespace: not running in-cluster...”

You can enable the leader election. However, if you are testing the project locally using the `make run` target which will run the manager outside of the cluster then, you might also need to set the namespace the leader election resource will be created, as follows:

```
mgr, err := ctrl.NewManager(ctrl.GetConfigOrDie(), ctrl.Options{
    Scheme:                 scheme,
    MetricsBindAddress:     metricsAddr,
    Port:                   9443,
    HealthProbeBindAddress: probeAddr,
    LeaderElection:         enableLeaderElection,
    LeaderElectionID:       "14be1926.testproject.org",
    LeaderElectionNamespace: "<project-name>-system",
```

If you are running the project on the cluster with `make deploy` target then, you might not want to add this option. So, you might want to customize this behaviour using environment variables to only add this option for development purposes, such as:

```

leaderElectionNS := ""
if os.Getenv("ENABLE_LEADER_ELECTION_NAMESPACE") != "false" {
    leaderElectionNS = "<project-name>-system"
}

mgr, err := ctrl.NewManager(ctrl.GetConfigOrDie(), ctrl.Options{
    Scheme:                 scheme,
    MetricsBindAddress:     metricsAddr,
    Port:                   9443,
    HealthProbeBindAddress: probeAddr,
    LeaderElection:          enableLeaderElection,
    LeaderElectionNamespace: leaderElectionNS,
    LeaderElectionID:        "14be1926.testproject.org",
    ...
}

```

## I am facing the error “open **/var/run/secrets/kubernetes.io/serviceaccount/token: permission denied**” when I deploy my project against Kubernetes old versions. How to sort it out?

If you are facing the error:

```

1.6656687258729894e+09 ERROR controller-runtime.client.config      unable
to get kubeconfig      {"error": "open
/var/run/secrets/kubernetes.io/serviceaccount/token: permission denied"}
sigs.k8s.io/controller-runtime/pkg/client/config.GetConfigOrDie
    /go/pkg/mod/sigs.k8s.io/controller-
runtime@v0.13.0/pkg/client/config/config.go:153
main.main
    /workspace/main.go:68
runtime.main
    /usr/local/go/src/runtime/proc.go:250

```

when you are running the project against a Kubernetes old version (maybe <= 1.21) , it might be caused by the [issue](#) , the reason is the mounted token file set to 0600 , see [solution](#) here. Then, the workaround is:

Add `fsGroup` in the manager.yaml

```

securityContext:
    runAsNonRoot: true
    fsGroup: 65532 # add this fsGroup to make the token file readable

```

However, note that this problem is fixed and will not occur if you deploy the project in high versions (maybe  $\geq 1.22$ ).

## **The error Too long: must have at most 262144 bytes is faced when I run make install to apply the CRD manifests. How to solve it? Why this error is faced?**

When attempting to run `make install` to apply the CRD manifests, the error `Too long: must have at most 262144 bytes` may be encountered. This error arises due to a size limit enforced by the Kubernetes API. Note that the `make install` target will apply the CRD manifest under `config/crd` using `kubectl apply -f -`. Therefore, when the `apply` command is used, the API annotates the object with the `last-applied-configuration` which contains the entire previous configuration. If this configuration is too large, it will exceed the allowed byte size. ([More info](#))

In ideal approach might use client-side apply might seem like the perfect solution since with the entire object configuration doesn't have to be stored as an annotation (`last-applied-configuration`) on the server. However, it's worth noting that as of now, it isn't supported by controller-gen or kubebuilder. For more on this, refer to: [Controller-tool-discussion](#).

Therefore, you have a few options to workaround this scenario such as:

### **By removing the descriptions from CRDs:**

Your CRDs are generated using [controller-gen](#). By using the option `maxDescLen=0` to remove the description, you may reduce the size, potentially resolving the issue. To do it you can update the Makefile as the following example and then, call the target `make manifest` to regenerate your CRDs without description, see:

```
.PHONY: manifests
manifests: controller-gen ## Generate WebhookConfiguration, ClusterRole and
CustomResourceDefinition objects.
    # Note that the option maxDescLen=0 was added in the default scaffold in
    # order to sort out the issue
    # Too long: must have at most 262144 bytes. By using kubectl apply to
    # create / update resources an annotation
    # is created by K8s API to store the latest version of the resource (
    # kubectl.kubernetes.io/last-applied-configuration).
    # However, it has a size limit and if the CRD is too big with so many
    # long descriptions as this one it will cause the failure.
    $(CONTROLLER_GEN) rbac:roleName=manager-role crd:maxDescLen=0 webhook
paths="./..." output:crd:artifacts:config=config/crd/bases
```

### By re-design your APIs:

You can review the design of your APIs and see if it has not more specs than should be by hurting single responsibility principle for example. So that you might to re-design them.

## How can I validate and parse fields in CRDs effectively?

To enhance user experience, it is recommended to use [OpenAPI v3 schema](#) validation when writing your CRDs. However, this approach can sometimes require an additional parsing step. For example, consider this code

```
type StructName struct {
    // +kubebuilder:validation:Format=date-time
    TimeField string `json:"timeField,omitempty"`
}
```

### What happens in this scenario?

- Users will receive an error notification from the Kubernetes API if they attempt to create a CRD with an invalid timeField value.
- On the developer side, the string value needs to be parsed manually before use.

### Is there a better approach?

To provide both a better user experience and a streamlined developer experience, it is advisable to use predefined types like `metav1.Time`. For example, consider this code

```
type StructName struct {
    TimeField metav1.Time `json:"timeField,omitempty"`
}
```

## What happens in this scenario?

- Users still receive error notifications from the Kubernetes API for invalid `timeField` values.
- Developers can directly use the parsed `TimeField` in their code without additional parsing, reducing errors and improving efficiency.

