Black lives matter.

We stand in solidarity with the Black community.

Racism is unacceptable.

It conflicts with the core values of the Kubernetes project and our community does not tolerate it.

Note: Impatient readers may head straight to Quick Start.

Using Kubebuilder v1? Check the legacy documentation

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-Base APIs
- Resources vs Subresources

Kubernetes API extension developers

API extension developers will learn the principals 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

Resources

- Repository: sigs.k8s.io/kubebuilder
- Slack channel: #kubebuilder
- Google Group: kubebuilder@googlegroups.com

Quick Start

This Quick Start guide will cover:

- Creating a project
- Creating an API
- Running locally
- Running in-cluster

Prerequisites

- go version v1.13+.
- docker version 17.03+.
- kubectl version v1.11.3+.
- Access to a Kubernetes v1.11.3+ cluster.

Versions and Supportability

Projects created by Kubebuilder contain a Makefile that will install tools at versions defined at creation time. Those tools are:

- kustomize
- controller-gen

The versions which are defined in the `Makefile` and `go.mod` files are the versions tested and therefore is recommend to use the specified versions.

Installation

Install kubebuilder:

```
os=$(go env GOOS)
arch=$(go env GOARCH)

# download kubebuilder and extract it to tmp
curl -L https://go.kubebuilder.io/dl/2.3.1/${os}/${arch} | tar -xz -C /tmp/

# move to a long-term location and put it on your path
# (you'll need to set the KUBEBUILDER_ASSETS env var if you put it somewhere else)
sudo mv /tmp/kubebuilder_2.3.1_${os}_${arch} /usr/local/kubebuilder
export PATH=$PATH:/usr/local/kubebuilder/bin
```

Using master branch

Also, you can install a master snapshot from `https://go.kubebuilder.io/dl/latest/${os}/${arch}`.

Create a Project

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

```
mkdir $GOPATH/src/example
cd $GOPATH/src/example
kubebuilder init --domain my.domain
```

Not in $GOPATH

If you're not in `GOPATH`, you'll need to run `go mod init <modulename>` in order to tell kubebuilder and Go the base import path of your module.

For a further understanding of `GOPATH` see The GOPATH environment variable in the How to Write Go Code golang page doc.

Go package issues

Ensure that you activate the module support by running `$ export GO111MODULE=on` to solve issues as `cannot find package ... (from $GOROOT)`.

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 `controller/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.

▶ 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
```

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 an (CR)Custom Resource for your (CRD)Custom Resource Definition in your samples (make sure to edit them first if you've changed the API definition):

```
kubectl apply -f config/samples/
```

Run It On the Cluster

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
```

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
```

Next Step

Now, follow up the CronJob tutorial to better understand how it works by developing a demo example 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:

```
# 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
```

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

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 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"
    "fmt"
    "os"

    "k8s.io/apimachinery/pkg/runtime"
    _ "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/log/zap"
    // +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() {

    // +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.

```go
func main() {
    var metricsAddr string
    flag.StringVar(&metricsAddr, "metrics-addr", ":8080", "The address the metric
endpoint binds to.")
    flag.Parse()

    ctrl.SetLogger(zap.New(zap.UseDevMode(true)))

    mgr, err := ctrl.NewManager(ctrl.GetConfigOrDie(), ctrl.Options{Scheme: scheme,
MetricsBindAddress: metricsAddr})
    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:

```go
    mgr, err = ctrl.NewManager(ctrl.GetConfigOrDie(), ctrl.Options{
        Scheme:             scheme,
        Namespace:          namespace,
        MetricsBindAddress: metricsAddr,
    })
```

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 MultiNamespacedCacheBuilder to watch a specific set of namespaces:

```go
    var namespaces []string // List of Namespaces

    mgr, err = ctrl.NewManager(ctrl.GetConfigOrDie(), ctrl.Options{
        Scheme:             scheme,
        NewCache:           cache.MultiNamespacedCacheBuilder(namespaces),
        MetricsBindAddress: fmt.Sprintf("%s:%d", metricsHost, metricsPort),
    })
```

For further information see MultiNamespacedCacheBuilder

```go
    // +kubebuilder:scaffold:builder
    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!

Actually, before we get started with our API, we should talk 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!

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 that 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
```

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

// 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) ◀
// 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:

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.

```go
// 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"`
    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 batchv1beta1.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.
    // 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?

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 = &scheme.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"

    "github.com/go-logr/logr"
    "k8s.io/apimachinery/pkg/runtime"
    ctrl "sigs.k8s.io/controller-runtime"
    "sigs.k8s.io/controller-runtime/pkg/client"
    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
    Log    logr.Logger
    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=get;

//
+kubebuilder:rbac:groups=batch.tutorial.kubebuilder.io,resources=cronjobs/status,ver
```

`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 cancelation 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(req ctrl.Request) (ctrl.Result, error) {
    _ = context.Background()
    _ = r.Log.WithValues("cronjob", req.NamespacedName)

    // your logic here
    // 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/controllers/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 controllers

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

    "github.com/go-logr/logr"
    "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"

    batch "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
    Log    logr.Logger
    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=get;

//
+kubebuilder:rbac:groups=batch.tutorial.kubebuilder.io,resources=cronjobs/status,ver

//
+kubebuilder:rbac:groups=batch,resources=jobs,verbs=get;list;watch;create;update;pat

// +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"
)

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

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 batch.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 = scheduledTimeForJob
            } else if 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 date 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 is "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
failedJobs[i].Status.StartTime.Before(failedJobs[j].Status.StartTime)
        })
        for i, job := range failedJobs {
            if int32(i) >= int32(len(failedJobs))-
*cronJob.Spec.FailedJobsHistoryLimit {
```

```
*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
successfulJobs[i].Status.StartTime.Before(successfulJobs[j].Status.StartTime)
        })
        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

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.Spec.ConcurrencyPolicy == batch.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.Spec.ConcurrencyPolicy == batch.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 requeuing 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    = batch.GroupVersion.String()
)

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(&kbatch.Job{}, jobOwnerKey,
func(rawObj runtime.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(&batch.CronJob{}).
        Owns(&kbatch.Job{}).
        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/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. Builtin types such as Job have their scheme added by `clientgoscheme` .

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

func init() {
    _ = clientgoscheme.AddToScheme(scheme)

    _ = 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.

```
func main() {
```

```
// old stuff (hidden)
```

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

We'll also set up webhooks for our type, which we'll talk about next. We just need to add them to the manager. Since we might want to run the webhooks separately, or not run them when testing our controller locally, we'll put them behind an environment variable.

We'll just make sure to set `ENABLE_WEBHOOKS=false` when we run locally.

```
    if os.Getenv("ENABLE_WEBHOOKS") != "false" {
        if err = (&batchv1.CronJob{}).SetupWebhookWithManager(mgr); err != nil {
            setupLog.Error(err, "unable to create webhook", "webhook", "Captain")
            os.Exit(1)
        }
    }
    // +kubebuilder:scaffold:builder
```

```
// 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 `Defaulter` and (or) the `Validator` 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/api/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.

```
func (r *CronJob) SetupWebhookWithManager(mgr ctrl.Manager) error {
    return ctrl.NewWebhookManagedBy(mgr).
        For(r).
        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.

```
// +kubebuilder:webhook:path=/mutate-batch-tutorial-kubebuilder-io-v1-
cronjob,mutating=true,failurePolicy=fail,groups=batch.tutorial.kubebuilder.io,resour
```

We use the `webhook.Defaulter` 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.

```go
var _ webhook.Defaulter = &CronJob{}

// Default implements webhook.Defaulter so a webhook will be registered for the
type
func (r *CronJob) Default() {
    cronjoblog.Info("default", "name", r.Name)

    if r.Spec.ConcurrencyPolicy == "" {
        r.Spec.ConcurrencyPolicy = AllowConcurrent
    }
    if r.Spec.Suspend == nil {
        r.Spec.Suspend = new(bool)
    }
    if r.Spec.SuccessfulJobsHistoryLimit == nil {
        r.Spec.SuccessfulJobsHistoryLimit = new(int32)
        *r.Spec.SuccessfulJobsHistoryLimit = 3
    }
    if r.Spec.FailedJobsHistoryLimit == nil {
        r.Spec.FailedJobsHistoryLimit = new(int32)
        *r.Spec.FailedJobsHistoryLimit = 1
    }
}
```

This marker is responsible for generating a validating webhook manifest.

```go
// TODO(user): change verbs to "verbs=create;update;delete" if you want to enable
deletion validation.
// +kubebuilder:webhook:verbs=create;update,path=/validate-batch-tutorial-
kubebuilder-io-v1-
cronjob,mutating=false,failurePolicy=fail,groups=batch.tutorial.kubebuilder.io,resou
```

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

The `ValidateCreate`, `ValidateUpdate` and `ValidateDelete` methods are expected to validate that 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 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.

```
var _ webhook.Validator = &CronJob{}

// ValidateCreate implements webhook.Validator so a webhook will be registered for
the type
func (r *CronJob) ValidateCreate() error {
	cronjoblog.Info("validate create", "name", r.Name)

	return r.validateCronJob()
}

// ValidateUpdate implements webhook.Validator so a webhook will be registered for
the type
func (r *CronJob) ValidateUpdate(old runtime.Object) error {
	cronjoblog.Info("validate update", "name", r.Name)

	return r.validateCronJob()
}

// ValidateDelete implements webhook.Validator so a webhook will be registered for
the type
func (r *CronJob) ValidateDelete() error {
	cronjoblog.Info("validate delete", "name", r.Name)

	// TODO(user): fill in your validation logic upon object deletion.
	return nil
}
```

We validate the name and the spec of the CronJob.

```
func (r *CronJob) validateCronJob() error {
	var allErrs field.ErrorList
	if err := r.validateCronJobName(); err != nil {
		allErrs = append(allErrs, err)
	}
	if err := r.validateCronJobSpec(); err != nil {
		allErrs = append(allErrs, err)
	}
	if len(allErrs) == 0 {
		return nil
	}

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

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

```
func (r *CronJob) validateCronJobSpec() *field.Error {
    // The field helpers from the kubernetes API machinery help us return nicely
    // structured validation errors.
    return validateScheduleFormat(
        r.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

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 ([illegible] by default)

`/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 reccomended disabling webhooks when doing your local code-run-test cycle, as we do below.

In a separate terminal, run

```
make run ENABLE_WEBHOOKS=false
```

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:
  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
```

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

Deploying the 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 Mutating|ValidatingWebhookConfiguration.

To accomplish that, you need to use an annotation with key `cert-manager.io/inject-ca-from` in the Mutating|ValidatingWebhookConfiguration objects. The value of the annotation should point to an existing certificate CR instance in the format of `<certificate-namespace>/<certificate-name>`.

This is the kustomize patch we used for annotating the Mutating|ValidatingWebhookConfiguration objects.

```
# This patch add annotation to admission webhook config and
# the variables $(CERTIFICATE_NAMESPACE) and $(CERTIFICATE_NAME) will be
substituted by kustomize.
apiVersion: admissionregistration.k8s.io/v1beta1
kind: MutatingWebhookConfiguration
metadata:
  name: mutating-webhook-configuration
  annotations:
    cert-manager.io/inject-ca-from: $(CERTIFICATE_NAMESPACE)/$(CERTIFICATE_NAME)
---
apiVersion: admissionregistration.k8s.io/v1beta1
kind: ValidatingWebhookConfiguration
metadata:
  name: validating-webhook-configuration
  annotations:
    cert-manager.io/inject-ca-from: $(CERTIFICATE_NAMESPACE)/$(CERTIFICATE_NAME)
```

Deploying Admission Webhooks

Kind Cluster

It is recommended to develop your webhook with a kind cluster for faster iteration. Why?

- You can bring up a multi-node cluster locally within 1 minute.
- You can tear it down in seconds.
- You don't need to push your images to remote registry.

Cert Manager

You need follow this to install the cert manager bundle.

Build your image

Run the following command to build your image locally.

```
make docker-build
```

You don't need to push the image to a remote container registry if you are using a kind cluster. You can directly load your local image to your kind cluster:

```
kind load docker-image your-image-name:your-tag
```

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.
#commonLabels:
#  someName: someValue
```

```
bases:
- ../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

patchesStrategicMerge:
  # Protect the /metrics endpoint by putting it behind auth.
  # If you want your controller-manager to expose the /metrics
  # endpoint w/o any authn/z, please comment the following line.
- manager_auth_proxy_patch.yaml

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

# [CERTMANAGER] To enable cert-manager, uncomment all sections with 'CERTMANAGER'.
# Uncomment 'CERTMANAGER' sections in crd/kustomization.yaml to enable the CA
injection in the admission webhooks.
# 'CERTMANAGER' needs to be enabled to use ca injection
- webhookcainjection_patch.yaml

# the following config is for teaching kustomize how to do var substitution
vars:
# [CERTMANAGER] To enable cert-manager, uncomment all sections with 'CERTMANAGER'
prefix.
- name: CERTIFICATE_NAMESPACE # namespace of the certificate CR
  objref:
    kind: Certificate
    group: cert-manager.io
    version: v1alpha2
    name: serving-cert # this name should match the one in certificate.yaml
  fieldref:
    fieldpath: metadata.namespace
- name: CERTIFICATE_NAME
  objref:
    kind: Certificate
    group: cert-manager.io
    version: v1alpha2
    name: serving-cert # this name should match the one in certificate.yaml
- name: SERVICE_NAMESPACE # namespace of the service
  objref:
    kind: Service
    version: v1
    name: webhook-service
  fieldref:
    fieldpath: metadata.namespace
- name: SERVICE_NAME
  objref:
    kind: Service
    version: v1
    name: webhook-service
```

Now you can deploy it to your cluster by

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

Wait a while til 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 Ginko.

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/controllers/suite_test.go
```

When we created the CronJob API with `kubebuilder create api` in a previous chapter, Kubebuilder already did some test work for you. Kubebuilder scaffolded a `controllers/suite_test.go` file that does the bare bones of setting up a test environment.

First, it will contain the necessary imports.

```
// Apache License (hidden)
// Imports (hidden)
```

Now, let's go through the code generated.

```
var cfg *rest.Config
var k8sClient client.Client // You'll be using this client in your tests.
var testEnv *envtest.Environment

var _ = BeforeSuite(func(done Done) {
    logf.SetLogger(zap.LoggerTo(GinkgoWriter, true))
```

First, the envtest cluster is configured to read CRDs from the CRD directory Kubebuilder scaffolds for you.

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

Then, we start the envtest cluster.

```
    var err error
    cfg, err = testEnv.Start()
    Expect(err).ToNot(HaveOccurred())
    Expect(cfg).ToNot(BeNil())
```

The autogenerated test code will add the CronJob Kind schema to the default client-go k8s scheme. This ensures that the CronJob API/Kind will be used in our test controller.

```
    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
```

A client is created for our test CRUD operations.

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

Once you've added the code below, you can actually delete the k8sClient above, because you can get k8sClient from the manager (as shown below).

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

    err = (&CronJobReconciler{
        Client: k8sManager.GetClient(),
        Log:    ctrl.Log.WithName("controllers").WithName("CronJob"),
    }).SetupWithManager(k8sManager)
    Expect(err).ToNot(HaveOccurred())

    go func() {
        err = k8sManager.Start(ctrl.SetupSignalHandler())
```

```go
        err = k8sManager.Start(ctrl.SetupSignalHandler())
        Expect(err).ToNot(HaveOccurred())
    }()

    k8sClient = k8sManager.GetClient()
    Expect(k8sClient).ToNot(BeNil())

    close(done)
}, 60)
```

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.

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

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

    RunSpecsWithDefaultAndCustomReporters(t,
        "Controller Suite",
        []Reporter{envtest.NewlineReporter{}})
}
```

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!

Testing your Controller's Behavior

$ vim *../../cronjob-tutorial/testdata/project/controllers/cronjob_controller_test.go*

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

```
// Apache License (hidden)
// 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 = "test-cronjob-namespace"
        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: batchv1beta1.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)).Should(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 function's output matches what's expected in the subsequent `Should()` call, 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() bool {
                err := k8sClient.Get(ctx, cronjobLookupKey, createdCronjob)
                if err != nil {
                    return false
                }
                return true
            }, timeout, interval).Should(BeTrue())
            // Let's make sure our Schedule string value was properly
converted/handled.
            Expect(createdCronjob.Spec.Schedule).Should(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() (int, error) {
                err := k8sClient.Get(ctx, cronjobLookupKey, createdCronjob)
                if err != nil {
                    return -1, err
                }
                return len(createdCronjob.Status.Active), nil
            }, duration, interval).Should(Equal(0))
```

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,
                        },
                    },
                },
                Status: batchv1.JobStatus{
                    Active: 2,
                },
            }

            // 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)).Should(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() ([]string, error) {
                err := k8sClient.Get(ctx, cronjobLookupKey, createdCronjob)
                if err != nil {
                    return nil, err
                }

                names := []string{}
                for _, job := range createdCronjob.Status.Active {
                    names = append(names, job.Name)
                }
                return names, nil
            }, timeout, interval).Should(ConsistOf(JobName), "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

Advanced Examples

There are more involved examples of using envtest to rigorously test controller behavior. Examples include:

- Azure Databricks Operator: see their fully fleshed-out `suite_test.go` as well as any `*_test.go` file in that directory like this one.

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.

Minimum Kubernetes Versions Incoming!

CRD conversion support was introduced as an alpha feature in Kubernetes 1.13 (which means it's not on by default, and needs to be enabled via a feature gate), and became beta in Kubernetes 1.15 (which means it's on by default).

If you're on Kubernetes 1.13-1.14, make sure to enable the feature gate. If you're on Kubernetes 1.12 or below, you'll need a new cluster to use conversion. Check out the KinD instructions for instructions on how to set up a all-in-one cluster.

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 the 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
```

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:subresource:status
// +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:

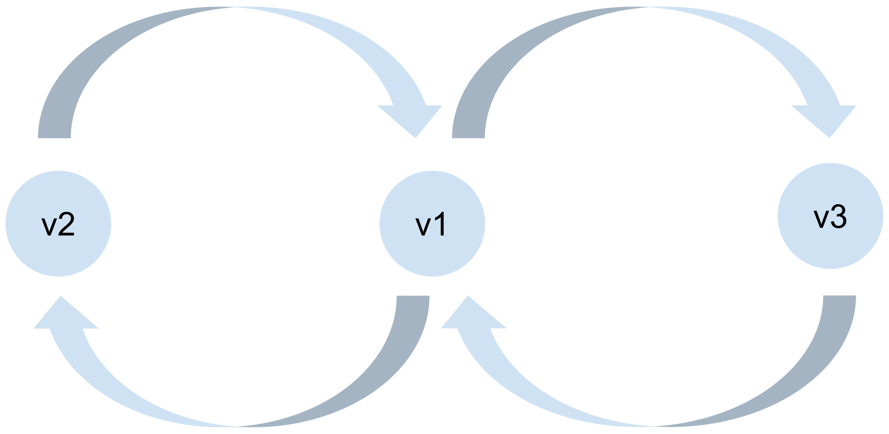
Complete graphs are insufficiently nautical

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:

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 put them in a file called `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)
```

```go
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.

```go
// 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)
```

```go
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.

```go
// ConvertTo converts this CronJob to the Hub version (v1).
func (src *CronJob) ConvertTo(dstRaw conversion.Hub) error {
	dst := dstRaw.(*v1.CronJob)

	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 {
    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, " ")
```

// rote 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 from the Hub version (v1) to this version.
func (dst *CronJob) ConvertFrom(srcRaw conversion.Hub) error {
    src := srcRaw.(*v1.CronJob)

    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])
```

// rote 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.

Normally, we'd run

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

to scaffold out the webhook setup. However, we've already got webhook setup, from when we built our defaulting and validating webhooks!

Webhook setup...

```
$ vim project/api/v1/cronjob_webhook.go
```

```
// Apache License (hidden)
// Go imports (hidden)
```

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

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

```
func (r *CronJob) SetupWebhookWithManager(mgr ctrl.Manager) error {
    return ctrl.NewWebhookManagedBy(mgr).
        For(r).
        Complete()
}
```

```
// Existing Defaulting and Validation (hidden)
```

...and `main.go`

Similarly, our existing main file is sufficient:

```
$ vim project/main.go
```

```
// Apache License (hidden)
// Imports (hidden)
// existing setup (hidden)
```

```
func main() {
```

```
// existing setup (hidden)
```

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

```
    if err = (&batchv1.CronJob{}).SetupWebhookWithManager(mgr); err != nil {
        setupLog.Error(err, "unable to create webhook", "webhook", "Captain")
        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 conversion 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 `manager_webhook_patch.yaml` under the `patches` section in `config/default/kustomization.yaml` file.
- Enable all the vars under the `CERTMANAGER` section in `config/default/kustomization.yaml` file.

Additionally, 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 an 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`)

```yaml
apiVersion: batch.tutorial.kubebuilder.io/v2
kind: CronJob
metadata:
  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
```

```yaml
apiVersion: batch.tutorial.kubebuilder.io/v2
kind: CronJob
metadata:
  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:
  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.

-

Kubebuilder v1 vs v2

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 the example 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:

```
pkg/
├── apis
│   ├── addtoscheme_batch_v1.go
│   ├── apis.go
│   └── batch
│       ├── group.go
│       └── v1
│           ├── cronjob_types.go
│           ├── cronjob_types_test.go
│           ├── doc.go
│           ├── register.go
│           ├── v1_suite_test.go
│           └── zz_generated.deepcopy.go
├── controller
└── webhook
```

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

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

// 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,resour


...

// 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,resou
```

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.

Single Group to Multi-Group

! Note

Multi-group scaffolding support was not present in the initial version of the KubeBuilder v2 scaffolding (as of KubeBuilder v2.0.0).

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.

While KubeBuilder v2 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.

Let's migrate the CronJob example.

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 rename `api` to `apis` to be more clear, and we'll move our existing APIs into a new subdirectory, "batch":

```
mkdir apis/batch
mv api/* apis/batch
# After ensuring that all was moved successfully remove the old directory `api/`
rm -rf api/
```

After moving the APIs to a new directory, the same needs to be applied to the controllers:

```
mkdir controllers/batch
mv controllers/* controllers/batch/
```

Next, we'll need to update all the references to the old package name. For CronJob, that'll be `main.go` and `controllers/batch/cronjob_controller.go`.

If you've added additional files to your project, you'll need to track down imports there as well.

Finally, we'll run the command which enable the multi-group layout in the project:

```
kubebuilder edit --multigroup=true
```

When the command `kubebuilder edit --multigroup=true` is executed it will add a new line to `PROJECT` that marks this a multi-group project:

```
version: "2"
domain: tutorial.kubebuilder.io
repo: tutorial.kubebuilder.io/project
multigroup: true
```

Note that this option indicates to KubeBuilder that this is a multi-group project.

In this way, if the project is not new and has previous APIs already implemented will be in the previous structure. Notice that with the `multi-group` project the Kind API's files are created under `apis/<group>/<version>` instead of `api/<version>`. Also, note that the controllers will be created under `controllers/<group>` instead of `controllers`. That is the reason why we moved the previously generated APIs with the provided scripts in the previous steps. Remember to update the references afterwards.

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).

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.

- 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
- controller-gen CLI
- Artifacts
- Writing controller tests
- Metrics

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` 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:

```go
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][kube-additional-printer-columns] 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:

```go
// +kubebuilder:printcolumn:name="Alias",type=string,JSONPath=`.spec.alias`
// +kubebuilder:printcolumn:name="Rank",type=integer,JSONPath=`.spec.rank`
// +kubebuilder:printcolumn:name="Bravely Run
// +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
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 reccomended 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,s

type CustomSet struct {
    metav1.TypeMeta   `json:",inline"`
    metav1.ObjectMeta `json:"metadata,omitempty"`

    Spec   ToySpec   `json:"spec,omitempty"`
    Status ToyStatus `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` to `CRD_OPTIONS ?= crd`

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 get` 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. For instance, to generate CRDs with "trivial versions" (no version conversion webhooks), we call `controller-gen crd:trivialVersions=true paths=./api/...`.

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) crd:trivialVersions=true 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 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)
```

The code snippet below shows skeleton code for implementing a finalizer.

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

    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)
    }
    // name of our custom finalizer
    myFinalizerName := "storage.finalizers.tutorial.kubebuilder.io"

    // 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 lets add the finalizer and update the object. This is equivalent
        // registering our finalizer.
        if !containsString(cronJob.ObjectMeta.Finalizers, myFinalizerName) {
            cronJob.ObjectMeta.Finalizers = append(cronJob.ObjectMeta.Finalizers,
myFinalizerName)
            if err := r.Update(context.Background(), cronJob); err != nil {
                return ctrl.Result{}, err
            }
        }
    } else {
        // The object is being deleted
        if containsString(cronJob.ObjectMeta.Finalizers, myFinalizerName) {
            // our finalizer is present, so lets 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.
            cronJob.ObjectMeta.Finalizers =
removeString(cronJob.ObjectMeta.Finalizers, myFinalizerName)
            if err := r.Update(context.Background(), 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 types for same object.
}

// Helper functions to check and remove string from a slice of strings.
func containsString(slice []string, s string) bool {
    for _, item := range slice {
        if item == s {
            return true
        }
    }
    return false
}

func removeString(slice []string, s string) (result []string) {
    for _, item := range slice {
        if item == s {
            continue
        }
        result = append(result, item)
    }
    return
}
```

Kind Cluster

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.sigs.k8s.io/v1alpha3
nodes:
  - role: control-plane
  - role: worker
  - role: worker
  - role: worker
  role: worker
```

Using the configuration above, run the following command will give you a k8s 1.14.2 cluster with 1 master and 3 workers.

```
kind create cluster --config hack/kind-config.yaml --image=kindest/node:v1.14.2
```

You can use `--image` flag to specify the cluster version you want, e.g. `--image=kindest/node:v1.13.6`, the supported version are listed here

Cheetsheet

- Load a local image into a kind cluster.

```
kind load docker-image your-image-name:your-tag
```

- Point `kubectl` to the kind cluster

```
kind export kubeconfig
```

- Delete a kind cluster

```
kind delete cluster
```

Webhook

Webhooks are requests for information sent in a blocking fashion. A web application implementing webhooks will send an HTTP request to other application when 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.

Admission Webhook for Core Types

It is very easy to build admission webhooks for CRDs, which has been covered in the CronJob tutorial. Given that kubebuilder doesn't support webhook scaffolding for core types, you have to use the library from controler-runtime to handle it. There is an example in controller-runtime.

It is suggested to use kubebuilder to initialize a project, and then you can follow the steps below to add admission webhooks for core types.

Implement Your Handler

You need to have your handler implements the admission.Handler interface.

```
type podAnnotator struct {
    Client  client.Client
    decoder *admission.Decoder
}

func (a *podAnnotator) Handle(ctx context.Context, req admission.Request)
admission.Response {
    pod := &corev1.Pod{}
    err := a.decoder.Decode(req, pod)
    if err != nil {
        return admission.Errored(http.StatusBadRequest, err)
    }

    // mutate the fields in pod

    marshaledPod, err := json.Marshal(pod)
    if err != nil {
        return admission.Errored(http.StatusInternalServerError, err)
    }
    return admission.PatchResponseFromRaw(req.Object.Raw, marshaledPod)
}
```

If you need a client, just pass in the client at struct construction time.

If you add the `InjectDecoder` method for your handler, a decoder will be injected for you.

```
func (a *podAnnotator) InjectDecoder(d *admission.Decoder) error {
    a.decoder = d
    return nil
}
```

Note: in order to have controller-gen generate the webhook configuration for you, you need to add markers. For example,

```
// +kubebuilder:webhook:path=/mutate-v1-
pod,mutating=true,failurePolicy=fail,groups="",resources=pods,verbs=create;update,ve
```

Update main.go

Now you need to register your handler in the webhook server.

```
mgr.GetWebhookServer().Register("/mutate-v1-pod", &webhook.Admission{Handler:
&podAnnotator{Client: mgr.GetClient()}})
```

You need to ensure the path here match the path in the marker.

Deploy

Deploying it is just like deploying a webhook server for CRD. You need to

1. provision the serving certificate 2) deploy the server

You can follow the tutorial.

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
```

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:validation: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: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: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.

`// +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.

▶ Show Detailed Argument Help

`// +kubebuilder:default:=<any>` on field
  ▶ sets the default 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

// +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:Maximum:=*<int>* on field

specifies the maximum numeric value that this field can have.

// +kubebuilder:validation:MinItems:=*<int>* on field

specifies the minimun length for this list.

// +kubebuilder:validation:MinLength:=*<int>* on field

specifies the minimum length for this string.

// +kubebuilder:validation:Minimum:=*<int>* on field

specifies the minimum numeric value that this field can have.

// +kubebuilder:validation:MultipleOf:=*<int>* 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, if fields are required by default.

// +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, if fields are optional by default.

// +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.

// +nullable on field

▶ marks this field as allowing the "null" value.

// +optional on field

specifies that this field is optional, if fields are required by default.

// +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:Maximum:=*<int>* on type

specifies the maximum numeric value that this field can have.

// +kubebuilder:validation:MinItems:=*<int>* on type

specifies the minimun length for this list.

// +kubebuilder:validation:MinLength:=*<int>* on type

specifies the minimum length for this string.

// +kubebuilder:validation:Minimum:=*<int>* on type

specifies the minimum numeric value that this field can have.

// +kubebuilder:validation:MultipleOf:=*<int>* 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: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.

`// +listMapKey:=`*<string>* on field

▶ specifies the keys to map listTypes.

`// +listType:=`*<string>* on field

▶ specifies the type of data-structure that the list represents (map, set, atomic).

`// +mapType:=`*<string>* on field

▶ specifies the level of atomicity of the map; i.e. whether each item in the map is independent of the others, or all fields are treated as a single unit.

`// +structType:=`*<string>* on field

▶ specifies the level of atomicity of the struct; i.e. whether each field in the struct is independent of the others, or all fields are treated as a single unit.

`// +kubebuilder:validation:XPreserveUnknownFields` on type

▶ PreserveUnknownFields stops the apiserver from pruning fields which are not specified.

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` `:failurePolicy=`*<string>*`,groups=`*<[]string>*`,mutating=`*<bool>*`,name=`*<string>*`,path=`*<string>*`,resources=`*<[]string>*`,verbs=`*<[]string>*`,versions=`*<[]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>*`,resources=`*<[]string>*`,urls=`*<[]string>*`,verbs=`*<[]string>*
on package

specifies an RBAC rule to all access to some resources or non-resource URLs.

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,

```
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 `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` on package
: generates (partial) {Mutating,Validating}WebhookConfiguration objects.

`// +schemapatch``:manifests=`*<string>*`,maxDescLen=`*<int>* on package
: ▶ patches existing CRDs with new schemata.

`// +rbac``:roleName=`*<string>* on package
: generates ClusterRole objects.

`// +object``:headerFile=`*<string>*`,year=`*<string>* on package
: generates code containing DeepCopy, DeepCopyInto, and DeepCopyObject method implementations.

`// +crd`
`:crdVersions=`*<[]string>*`,maxDescLen=`*<int>*`,preserveUnknownFields=`*<bool>*`,trivialVersions=`*<bool>*
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 whether they're package-associated or not.

`// +output:dir``:=`*<string>* on package

outputs each artifact to the given directory, regardless of if it's package-associated or not.

`// +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.

Artifacts

Kubebuilder publishes test binaries and container images in addition to the main binary releases.

Test Binaries

You can find all of the test binaries at `https://go.kubebuilder.io/test-tools`. You can find

You can find all of the test binaries at `https://go.kubebuilder.io/test-tools`. You can find individual test binaries at `https://go.kubebuilder.io/test-tools/${version}/${os}/${arch}`.

Container Images

You can find all container images for your os at `https://go.kubebuilder.io/images/${os}` or at `gcr.io/kubebuilder/thirdparty-${os}`. You can find individual container images at `https://go.kubebuilder.io/images/${os}/${version}` or at `gcr.io/kubebuilder/thirdparty-${os}:${version}`.

Configuring envtest for integration tests

`controller-runtime` offers `envtest` (godoc), a package that 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.

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`.

Configuring your test control plane

You can use environment variables and/or flags to specify the `api-server` and `etcd` setup within your integration tests.

within your integration tests.

Environment Variables

Variable name	Type	When to use
`USE_EXISTING_CLUSTER`	boolean	Instead of settir a local control p point to the con plane of an exis cluster.
`KUBEBUILDER_ASSETS`	path to directory	Point integratio tests to a direct containing all binaries (api-ser etcd and kubect
`TEST_ASSET_KUBE_APISERVER`, `TEST_ASSET_ETCD`, `TEST_ASSET_KUBECTL`	paths to, respectively, api-server, etcd and kubectl binaries	Similar to `KUBEBUILDER_AS` , but more gran Point integratio tests to use bin other than the default ones. Th environment variables can al used to ensure specific tests ru with expected versions of thes binaries.
`KUBEBUILDER_CONTROLPLANE_START_TIMEOUT` and `KUBEBUILDER_CONTROLPLANE_STOP_TIMEOUT`	durations in format supported by `time.ParseDuration`	Specify timeout different from t default for the t control plane to (respectively) st and stop; any te run that exceed them will fail.
		Set to `true` to a

Variable name	Type	When to use
`KUBEBUILDER_ATTACH_CONTROL_PLANE_OUTPUT`	boolean	the control plan stdout and stde os.Stdout and os.Stderr. This c useful when debugging test failures, as outp will include outp from the contro plane.

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(expectedOwnerReferen
```

Metrics

By default, controller-runtime builds a global prometheus registry and publishes a collection of performance metrics for each controller.

Protecting the Metrics

These metrics are protected by kube-auth-proxy by default if using kubebuilder. Kubebuilder v2.2.0+ scaffold a clusterrole which can be found at `config/rbac/auth_proxy_client_clusterrole.yaml`.

You will need to grant permissions to your Prometheus server so that it can scrape the protected metrics. To achieve that, you can create a `clusterRoleBinding` to bind the `clusterRole` to the service account that your Prometheus server uses.

You can run the following kubectl command to create it. If using kubebuilder `<project-prefix>` is the `namePrefix` field in `config/default/kustomization.yaml`.

```
kubectl create clusterrolebinding metrics --clusterrole=<project-prefix>-metrics-
reader --serviceaccount=<namespace>:<service-account-name>
```

Exporting Metrics for Prometheus

Follow the steps below to export the metrics using the Prometheus Operator:

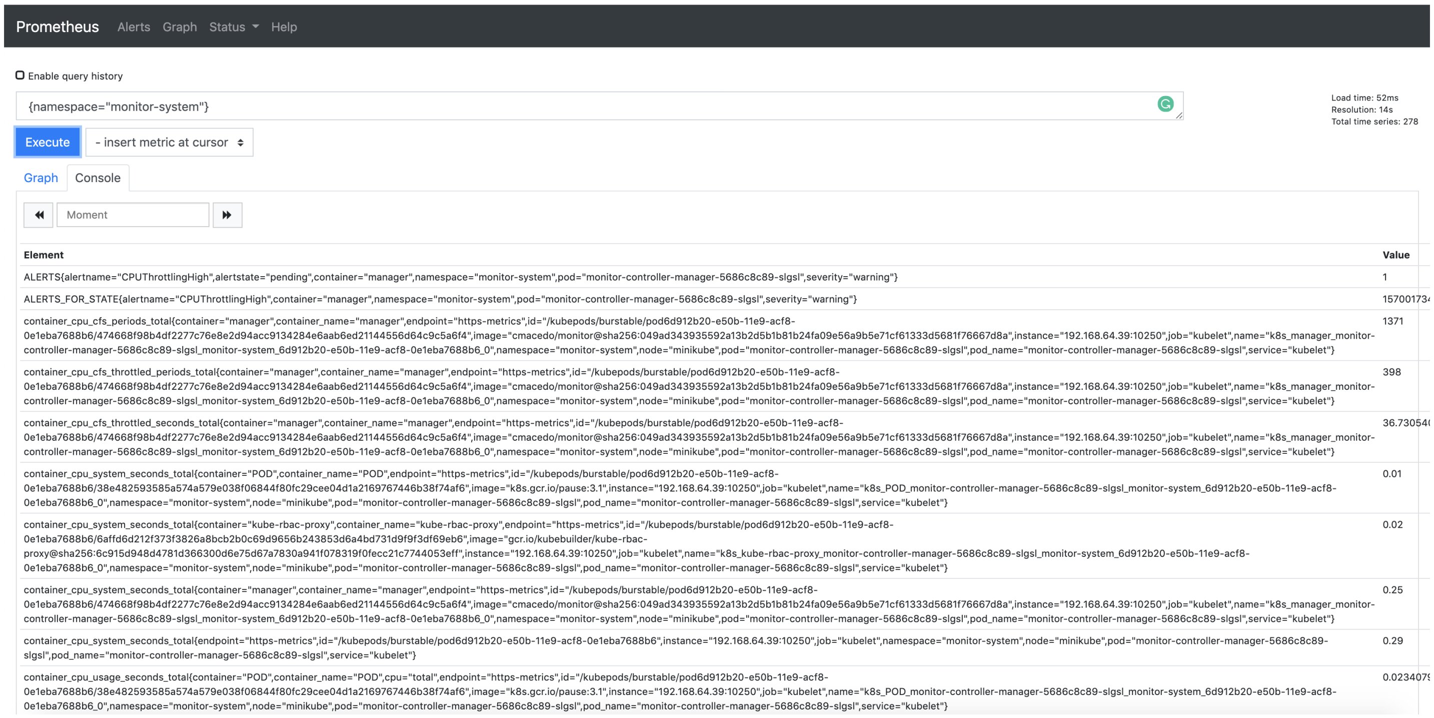
1. Install Prometheus and Prometheus Operator. We recommend using kube-prometheus in production if you don't have your own monitoring system. If you are just experimenting, you can only install Prometheus and Prometheus Operator.
2. Uncomment the line `- ../prometheus` in the `config/default/kustomization.yaml`. 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
```

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:



```
{namespace="monitor-system"}
```

Load time: 52ms
Resolution: 14s
Total time series: 278

Element	Value
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"}	15700173
container_cpu_cfs_periods_total{container="manager",container_name="manager",endpoint="https-metrics",id="/kubepods/burstable/pod6d912b20-e50b-11e9-acf8-0e1eba7688b6/474668f98b4df2277c76e8e2d94acc9134284e6aab6ed21144556d64c9c5a6f4",image="cmacedo/monitor@sha256:049ad343935592a13b2d5b1b81b24fa09e56a9b5e71cf61333d5681f76667d8a",instance="192.168.64.39:10250",job="kubelet",name="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"}	1371
container_cpu_cfs_throttled_periods_total{container="manager",container_name="manager",endpoint="https-metrics",id="/kubepods/burstable/pod6d912b20-e50b-11e9-acf8-0e1eba7688b6/474668f98b4df2277c76e8e2d94acc9134284e6aab6ed21144556d64c9c5a6f4",image="cmacedo/monitor@sha256:049ad343935592a13b2d5b1b81b24fa09e56a9b5e71cf61333d5681f76667d8a",instance="192.168.64.39:10250",job="kubelet",name="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-acf8-0e1eba7688b6/474668f98b4df2277c76e8e2d94acc9134284e6aab6ed21144556d64c9c5a6f4",image="cmacedo/monitor@sha256:049ad343935592a13b2d5b1b81b24fa09e56a9b5e71cf61333d5681f76667d8a",instance="192.168.64.39:10250",job="kubelet",name="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"}	36.73054
container_cpu_system_seconds_total{container="POD",container_name="POD",endpoint="https-metrics",id="/kubepods/burstable/pod6d912b20-e50b-11e9-acf8-0e1eba7688b6/38e482593585a574a579e038f06844f80fc29cee04d1a2169767446b38f74af6",image="k8s.gcr.io/pause:3.1",instance="192.168.64.39:10250",job="kubelet",name="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_system_seconds_total{container="kube-rbac-proxy",container_name="kube-rbac-proxy",endpoint="https-metrics",id="/kubepods/burstable/pod6d912b20-e50b-11e9-acf8-0e1eba7688b6/6affd6d212f373f3826a8bcb2b0c69d9656b243853d6a4bd731d9f9f3df69eb6",image="gcr.io/kubebuilder/kube-rbac-proxy@sha256:6c915d948d4781d366300d6e75d67a7830a941f078319f0fecc21c7744053eff",instance="192.168.64.39:10250",job="kubelet",name="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"}	0.02
container_cpu_system_seconds_total{container="manager",container_name="manager",endpoint="https-metrics",id="/kubepods/burstable/pod6d912b20-e50b-11e9-acf8-0e1eba7688b6/474668f98b4df2277c76e8e2d94acc9134284e6aab6ed21144556d64c9c5a6f4",image="cmacedo/monitor@sha256:049ad343935592a13b2d5b1b81b24fa09e56a9b5e71cf61333d5681f76667d8a",instance="192.168.64.39:10250",job="kubelet",name="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"}	0.25
container_cpu_system_seconds_total{endpoint="https-metrics",id="/kubepods/burstable/pod6d912b20-e50b-11e9-acf8-0e1eba7688b6",instance="192.168.64.39:10250",job="kubelet",namespace="monitor-system",node="minikube",pod="monitor-controller-manager-5686c8c89-slgsl",pod_name="monitor-controller-manager-5686c8c89-slgsl",service="kubelet"}	0.29
container_cpu_usage_seconds_total{container="POD",container_name="POD",cpu="total",endpoint="https-metrics",id="/kubepods/burstable/pod6d912b20-e50b-11e9-acf8-0e1eba7688b6/38e482593585a574a579e038f06844f80fc29cee04d1a2169767446b38f74af6",image="k8s.gcr.io/pause:3.1",instance="192.168.64.39:10250",job="kubelet",name="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"}	0.023407

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()`.

For example:

```go
import (
    "github.com/prometheus/client_golang/prometheus"
    "sigs.k8s.io/controller-runtime/pkg/metrics"
)
```

```go
var (
    goobers = prometheus.NewCounter(
        prometheus.CounterOpts{
            Name: "goobers_total",
            Help: "Number of goobers proccessed",
        },
    )
    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, and those metrics will be available for prometheus or other openmetrics systems to scrape.

TODO

If you're seeing this page, it's probably because something's not done in the book yet. Go see if anyone else has found this or bug the maintainers.