- Efficient Detection of Changes
- Add ImagePullSecrets to a Service Account
- RBAC Dev
- Rakkess
- How the Basics of Kubernetes Auth Scale for Organizations
- Kubernetes CVE-2020-8559 Proof of Concept PoC Exploit
- OAuth Is Not Authentication

PART VI
Advanced Patterns

The patterns in this category cover more complex topics that do not fit in any of the other categories. Some of the patterns here such as *Controller* or *Operator* are timeless, and Kubernetes itself is built on them. However, some of the other pattern implementations are still evolving. To keep up with this, we will keep our online examples up to date and reflect the latest developments in this space.

In the following chapters, we explore these advanced patterns:

- Chapter 27, "Controller", is essential to Kubernetes itself and shows how custom controllers can extend the platform.
- Chapter 28, "Operator", combines a controller with custom domain-specific resources to encapsulate operational knowledge in an automated form.
- Chapter 29, "Elastic Scale", describes how Kubernetes can handle dynamic loads by scaling in various dimensions.
- Chapter 30, "Image Builder", moves the aspect of building application images onto the cluster itself.

CHAPTER 27
Controller

A controller actively monitors and maintains a set of Kubernetes resources in a desired state. The heart of Kubernetes itself consists of a fleet of controllers that regularly watch and reconcile the current state of applications with the declared target state. In this chapter, we see how to leverage this *Controller* pattern to extend the platform for our needs.

Problem

You've already seen that Kubernetes is a sophisticated and comprehensive platform that provides many features out of the box. However, it is a general-purpose orchestration platform that does not cover all application use cases. Luckily, it provides natural extension points where specific use cases can be implemented elegantly on top of proven Kubernetes building blocks.

The main questions that arise here are how to extend Kubernetes without changing and breaking it and how to use its capabilities for custom use cases.

By design, Kubernetes is based on a declarative resource-centric API. What exactly do we mean by *declarative*? As opposed to an *imperative* approach, a declarative approach does not tell Kubernetes how it should act but instead describes how the target state should look. For example, when we scale up a Deployment, we do not actively create new Pods by telling Kubernetes to "create a new Pod." Instead, we change the Deployment resource's `replicas` property via the Kubernetes API to the desired number.

So, how are the new Pods created? This is done internally by the controllers. For every change in the resource status (like changing the `replicas` property value of a Deployment), Kubernetes creates an event and broadcasts it to all interested listeners. These listeners can then react by modifying, deleting, or creating new resources,

which in turn creates other events, like Pod-created events. These events are then potentially picked up again by other controllers, which perform their specific actions.

The whole process is also known as *state reconciliation*, where a target state (the number of desired replicas) differs from the current state (the actual running instances), and it is the task of a controller to reconcile and reach the desired target state again. When looked at from this angle, Kubernetes essentially represents a distributed state manager. You give it the desired state for a component instance, and it attempts to maintain that state should anything change.

How can we now hook into this reconciliation process without modifying Kubernetes code and create a controller customized for our specific needs?

Solution

Kubernetes comes with a collection of built-in controllers that manage standard Kubernetes resources like ReplicaSets, DaemonSets, StatefulSets, Deployments, or Services. These controllers run as part of the controller manager, which is deployed (as a standalone process or a Pod) on the control plane node. These controllers are not aware of one another. They run in an endless reconciliation loop, to monitor their resources for the actual and desired state and to act accordingly to get the actual state closer to the desired state.

However, in addition to these out-of-the-box controllers, the Kubernetes event-driven architecture allows us to natively plug in other custom controllers. Custom controllers can add extra functionality to the behavior by reacting to state-changing events, the same way that internal controllers do. A common characteristic of controllers is that they are reactive and react to events in the system to perform their specific actions. At a high level, this reconciliation process consists of the following main steps:

Observe
: Discover the actual state by watching for events issued by Kubernetes when an observed resource changes.

Analyze
: Determine the differences from the desired state.

Act
: Perform operations to drive the actual state to the desired state.

For example, the ReplicaSet controller watches for ReplicaSet resource changes, analyzes how many Pods need to be running, and acts by submitting Pod definitions to the API Server. The Kubernetes backend is then responsible for starting up the requested Pod on a node.

Figure 27-1 shows how a controller registers itself as an event listener for detecting changes on the managed resources. It observes the current state and changes it by calling out to the API Server to get closer to the target state (if necessary).

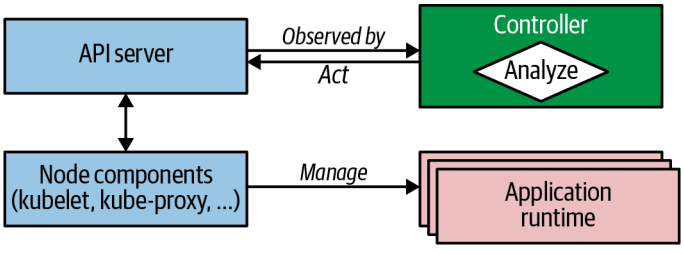

Figure 27-1. Observe-Analyze-Act cycle

Controllers are part of the Kubernetes control plane, and it became clear early on that they would also allow you to extend the platform with custom behavior. Moreover, they have become the standard mechanism for extending the platform and enable complex application lifecycle management. And as a result, a new generation of more sophisticated controllers was born, called *Operators*. From an evolutionary and complexity point of view, we can classify the active reconciliation components into two groups:

Controllers
: A simple reconciliation process that monitors and acts on standard Kubernetes resources. More often, these controllers enhance platform behavior and add new platform features.

Operators
: A sophisticated reconciliation process that interacts with CustomResourceDefinitions (CRDs), which are at the heart of the *Operator* pattern. Typically, these operators encapsulate complex application domain logic and manage the full application lifecycle.

As stated previously, these classifications help introduce new concepts gradually. Here, we focus on the simpler controllers, and in Chapter 28, we introduce CRDs and build up to the *Operator* pattern.

To avoid having multiple controllers acting on the same resources simultaneously, controllers use the *Singleton Service* pattern explained in Chapter 10. Most controllers are deployed just as Deployments but with one replica, as Kubernetes uses optimistic locking at the resource level to prevent concurrency issues when changing resource objects. In the end, a controller is nothing more than an application that runs permanently in the background.

Because Kubernetes itself is written in Go, and a complete client library for accessing Kubernetes is also written in Go, many controllers are written in Go too. However, you can write controllers in any programming language by sending requests to the Kubernetes API Server. We see a controller written in a pure shell script later in Example 27-1.

The most straightforward kind of controllers extend the way Kubernetes manages its resources. They operate on the same standard resources and perform similar tasks as the Kubernetes internal controllers operating on the standard Kubernetes resources, but they are invisible to the user of the cluster. Controllers evaluate resource definitions and conditionally perform some actions. Although they can monitor and act upon any field in the resource definition, metadata and ConfigMaps are most suitable for this purpose. The following are a few considerations to keep in mind when choosing where to store controller data:

Labels
: Labels as part of a resource's metadata can be watched by any controller. They are indexed in the backend database and can be efficiently searched for in queries. We should use labels when a selector-like functionality is required (e.g., to match Pods of a Service or a Deployment). A limitation of labels is that only alphanumeric names and values with restrictions can be used. See the Kubernetes documentation for which syntax and character sets are allowed for labels.

Annotations
: Annotations are an excellent alternative to labels. They have to be used instead of labels if the values do not conform to the syntax restrictions of label values. Annotations are not indexed, so we use annotations for nonidentifying information not used as keys in controller queries. Preferring annotations over labels for arbitrary metadata also has the advantage that it does not negatively impact the internal Kubernetes performance.

ConfigMaps
: Sometimes controllers need additional information that does not fit well into labels or annotations. In this case, ConfigMaps can be used to hold the target state definition. These ConfigMaps are then watched and read by the controllers. However, CRDs are much better suited for designing the custom target state specification and are recommended over plain ConfigMaps. For registering CRDs, however, you need elevated cluster-level permissions. If you don't have these, ConfigMaps are still the best alternative to CRDs. We will explain CRDs in detail in Chapter 28, "Operator".

Here are a few reasonably simple example controllers you can study as a sample implementation of this pattern:

jenkins-x/exposecontroller
: This controller watches Service definitions, and if it detects an annotation named `expose` in the metadata, the controller automatically exposes an Ingress object for external access of the Service. It also removes the Ingress object when someone removes the Service. This project is now archived but still serves as a good example of implementing a simple controller.

stakater/Reloader
: This is a controller that watches ConfigMap and Secret objects for changes and performs rolling upgrades of their associated workloads, which can be Deployment, DaemonSet, StatefulSet and other workload resources. We can use this controller with applications that are not capable of watching the ConfigMap and updating themselves with new configurations dynamically. That is particularly true when a Pod consumes this ConfigMap as environment variables or when your application cannot quickly and reliably update itself on the fly without a restart. As a proof of concept, we implement a similar controller with a plain shell script in Example 27-2.

Flatcar Linux Update Operator
: This is a controller that reboots a Kubernetes node running on Flatcar Container Linux when it detects a particular annotation on the Node resource object.

Now let's take a look at a concrete example: a controller that consists of a single shell script and that watches the Kubernetes API for changes on ConfigMap resources. If we annotate such a ConfigMap with `k8spatterns.io/podDeleteSelector`, all Pods selected with the given label selector are deleted when the ConfigMap changes. Assuming we back these Pods with a high-order resource like Deployment or ReplicaSet, these Pods are restarted and pick up the changed configuration.

For example, the following ConfigMap would be monitored by our controller for changes and would restart all Pods that have a label `app` with value `webapp`. The ConfigMap in Example 27-1 is used in our web application to provide a welcome message.

Example 27-1. ConfigMap use by web application

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: webapp-config
  annotations:
    k8spatterns.io/podDeleteSelector: "app=webapp"  ❶
data:
  message: "Welcome to Kubernetes Patterns !"
```

❶ Annotation used as selector for the controller in Example 27-2 to find the application Pods to restart.

Our controller shell script now evaluates this ConfigMap. You can find the source in its full glory in our Git repository. In short, the controller starts a *hanging GET* HTTP request for opening an endless HTTP response stream to observe the lifecycle events pushed by the API Server to us. These events are in the form of plain JSON objects, which are then analyzed to detect whether a changed ConfigMap carries our annotation. As events arrive, the controller acts by deleting all Pods matching the selector provided as the value of the annotation. Let's have a closer look at how the controller works.

The main part of this controller is the reconciliation loop, which listens on ConfigMap lifecycle events, as shown in Example 27-2.

Example 27-2. Controller script

```
namespace=${WATCH_NAMESPACE:-default}  ❶

base=http://localhost:8001             ❷
ns=namespaces/$namespace

curl -N -s $base/api/v1/${ns}/configmaps?watch=true | \
while read -r event                    ❸
do
   # ...
done
```

❶ Namespace to watch (or *default* if not given).

❷ Access to the Kubernetes API via a proxy running in the same Pod.

❸ Loop with watches for events on ConfigMaps.

The environment variable `WATCH_NAMESPACE` specifies the namespace in which the controller should watch for ConfigMap updates. We can set this variable in the Deployment descriptor of the controller itself. In our example, we're using the Downward API described in Chapter 14, "Self Awareness", to monitor the namespace in which we have deployed the controller as configured in Example 27-3 as part of the controller Deployment.

Example 27-3. `WATCH_NAMESPACE` extracted from the current namespace

```
env:
 - name: WATCH_NAMESPACE
   valueFrom:
     fieldRef:
       fieldPath: metadata.namespace
```

With this namespace, the controller script constructs the URL to the Kubernetes API endpoint to watch the ConfigMaps.

Note the `watch=true` query parameter in Example 27-2. This parameter indicates to the API Server not to close the HTTP connection but to send events along the response channel as soon as they happen (*hanging GET* or *Comet* are other names for this kind of technique). The loop reads every individual event as it arrives as a single item to process.

As you can see, our controller contacts the Kubernetes API Server via localhost. We won't deploy this script directly on the Kubernetes API control plane node, but then how can we use localhost in the script? As you may have probably guessed, another pattern kicks in here. We deploy this script in a Pod together with an ambassador container that exposes port 8001 on localhost and proxies it to the real Kubernetes Service. See Chapter 18 for more details on the *Ambassador* pattern. We see the actual Pod definition with this ambassador in detail later in this chapter.

Watching events this way is not very robust, of course. The connection can stop anytime, so there should be a way to restart the loop. Also, one could miss events, so production-grade controllers should not only watch on events but from time to time should also query the API Server for the entire current state and use that as the new base. For the sake of demonstrating the pattern, this is good enough.

Within the loop, the logic shown in Example 27-4 is performed.

Example 27-4. Controller reconciliation loop

```
curl -N -s $base/api/v1/${ns}/configmaps?watch=true | \
while read -r event
do
  type=$(echo "$event"        | jq -r '.type')                    ❶
  config_map=$(echo "$event"  | jq -r '.object.metadata.name')
  annotations=$(echo "$event" | jq -r '.object.metadata.annotations')

  if [ "$annotations" != "null" ]; then
    selector=$(echo $annotations | \                              ❷
     jq -r "\
        to_entries                                          |\
```

```
        .[]                                                    |\
        select(.key == \"k8spatterns.io/podDeleteSelector\") |\
        .value                                                 |\
         @uri                                                   \
    ")
  fi

  if [ $type = "MODIFIED" ] && [ -n "$selector" ]; then          ❸
    pods=$(curl -s $base/api/v1/${ns}/pods?labelSelector=$selector |\
           jq -r .items[].metadata.name)

    for pod in $pods; do                                         ❹
      curl -s -X DELETE $base/api/v1/${ns}/pods/$pod
    done
  fi
done
```

❶ Extract the type and name of the ConfigMap from the event.

❷ Extract all annotations on the ConfigMap with the key `k8spatterns.io/podDeleteSelector`. See the following sidebar for an explanation of this `jq` expression.

❸ If the event indicates an update of the ConfigMap and our annotation is attached, then find all Pods matching this label selector.

❹ Delete all Pods that match the selector.

First, the script extracts the event type that specifies what action happened to the ConfigMap. Then, we derive the annotations with `jq`. `jq` is an excellent tool for parsing JSON documents from the command line, and the script assumes it is available in the container the script is running in.

If the ConfigMap has annotations, we check for the annotation `k8spatterns.io/podDeleteSelector` by using a more complex `jq` query. The purpose of this query is to convert the annotation value to a Pod selector that can be used in an API query option in the next step: an annotation `k8spatterns.io/podDeleteSelector: "app=webapp"` is transformed to `app%3Dwebapp` that is used as a Pod selector. This conversion is performed with `jq` and is explained next if you are interested in how this extraction works.

If the script can extract a `selector`, we can now use it directly to select the Pods to delete. First, we look up all Pods that match the selector, and then we delete them one by one with direct API calls.

This shell script-based controller is, of course, not production-grade (e.g., the event loop can stop any time), but it nicely reveals the base concepts without too much boilerplate code for us.

Some jq Fu

Extracting the ConfigMap's `k8spatterns.io/podDeleteSelector` annotation value and converting it to a Pod selector is performed with `jq`. This is an excellent JSON command-line tool, but some concepts can be a bit confusing. Let's have a close look at how the expressions work in detail:

```
selector=$(echo $annotations | \
   jq -r "\
    to_entries                                          |\
    .[]                                                 |\
    select(.key == \"k8spatterns.io/podDeleteSelector\") |\
    .value                                              |\
     @uri                                                \
")
```

- `$annotations` holds all annotations as a JSON object, with annotation names as properties.
- With `to_entries`, we convert a JSON object like `{ "a": "b"}` into an array with entries like `{ "key": "a", "value": "b" }`. See the jq documentation for more details.
- `.[]` selects the array entries individually.
- From these entries, we pick only the ones with the matching key. There can be only zero or one matches that survive this filter.
- Finally, we extract the value (`.value`) and convert it with `@uri` so that it can be used as part of a URI.

This expression converts a JSON structure such as

```
{
  "k8spatterns.io/pattern": "Controller",
  "k8spatterns.io/podDeleteSelector": "app=webapp"
}
```

to a selector, `app%3Dwebapp`.

The remaining work is about creating resource objects and container images. The controller script itself is stored in a ConfigMap `config-watcher-controller`, and can be easily edited later if required.

We use a Deployment to create a Pod for our controller with two containers:

- One Kubernetes API ambassador container that exposes the Kubernetes API on localhost on port 8001. The image `k8spatterns/kubeapi-proxy` is an Alpine Linux with a local `kubectl` installed and `kubectl proxy` started with the proper

CA and token mounted. The original version, kubectl-proxy, was written by Marko Lukša, who introduced this proxy in *Kubernetes in Action*.

- The main container that executes the script contained in the just-created ConfigMap. Here, we use an Alpine base image with `curl` and `jq` installed.

You can find the Dockerfiles for the `k8spatterns/kubeapi-proxy` and `k8spatterns/curl-jq` images in the example Git repository.

Now that we have the images for our Pod, the final step is to deploy the controller by using a Deployment. We can see the main parts of the Deployment in Example 27-5 (the full version is available in our example repository).

Example 27-5. Controller Deployment

```
apiVersion: apps/v1
kind: Deployment
# ....
spec:
  template:
    # ...
    spec:
      serviceAccountName: config-watcher-controller ❶
      containers:
      - name: kubeapi-proxy                         ❷
        image: k8spatterns/kubeapi-proxy
      - name: config-watcher                        ❸
        image: k8spatterns/curl-jq
        # ...
        command:                                    ❹
        - "sh"
        - "/watcher/config-watcher-controller.sh"
        volumeMounts:                               ❺
        - mountPath: "/watcher"
          name: config-watcher-controller
      volumes:
      - name: config-watcher-controller             ❻
        configMap:
          name: config-watcher-controller
```

❶ ServiceAccount with proper permissions for watching events and restarting Pods.

❷ Ambassador container for proxying localhost to the Kubeserver API.

❸ Main container holding all tools and mounting the controller script.

❹ Startup command calling the controller script.

❺ Volume mapped to the ConfigMap holding our script.

❻ Mount of the ConfigMap-backed volume into the main Pod.

As you can see, we mount the `config-watcher-controller-script` from the ConfigMap we created previously and directly use it as the startup command for the primary container. For simplicity, we omitted any liveness and readiness checks as well as resource limit declarations. Also, we need a ServiceAccount `config-watcher-controller` that is allowed to monitor ConfigMaps. Refer to the example repository for the full security setup.

Let's see the controller in action. For this, we are using a straightforward web server, which serves the value of an environment variable as the only content. The base image uses plain `nc` (netcat) for serving the content. You can find the Dockerfile for this image in the example repository. We deploy the HTTP server with a ConfigMap and Deployment, as is sketched in Example 27-6.

Example 27-6. Sample web app with Deployment and ConfigMap

```
apiVersion: v1
kind: ConfigMap                                          ❶
metadata:
  name: webapp-config
  annotations:
    k8spatterns.io/podDeleteSelector: "app=webapp"       ❷
data:
  message: "Welcome to Kubernetes Patterns !"            ❸
---
apiVersion: apps/v1
kind: Deployment                                         ❹
# ...
spec:
  # ...
  template:
    spec:
      containers:
      - name: app
        image: k8spatterns/mini-http-server              ❺
        ports:
        - containerPort: 8080
        env:
        - name: MESSAGE                                  ❻
          valueFrom:
            configMapKeyRef:
              name: webapp-config
              key: message
```

❶ ConfigMap for holding the data to serve.

❷ Annotation that triggers a restart of the web app's Pod.

❸ Message used in web app in HTTP responses.

❹ Deployment for the web app.

❺ Simplistic image for HTTP serving with netcat.

❻ Environment variable used as an HTTP response body and fetched from the watched ConfigMap.

This concludes our example of our ConfigMap controller implemented in a plain shell script. Although this is probably the most complex example in this book, it also shows that it does not take much to write a basic controller.

Obviously, for real-world scenarios, you would write this sort of controller in a real programming language that provides better error-handling capabilities and other advanced features.

Discussion

To sum up, a controller is an active reconciliation process that monitors objects of interest for the world's desired state and the world's actual state. Then, it sends instructions to try to change the world's current state to be more like the desired state. Kubernetes uses this mechanism with its internal controllers, and you can also reuse the same mechanism with custom controllers. We demonstrated what is involved in writing a custom controller and how it functions and extends the Kubernetes platform.

Controllers are possible because of the highly modular and event-driven nature of the Kubernetes architecture. This architecture naturally leads to a decoupled and asynchronous approach for controllers as extension points. The significant benefit here is that we have a precise technical boundary between Kubernetes itself and any extensions. However, one issue with the asynchronous nature of controllers is that they are often hard to debug because the flow of events is not always straightforward. As a consequence, you can't easily set breakpoints in your controller to stop everything to examine a specific situation.

In Chapter 28, you'll learn about the related *Operator* pattern, which builds on this *Controller* pattern and provides an even more flexible way to configure operations.

More Information

- Controller Example
- Writing Controllers
- Writing a Kubernetes Controller
- A Deep Dive into Kubernetes Controllers
- Expose Controller
- Reloader: ConfigMap Controller
- Writing a Custom Controller: Extending the Functionality of Your Cluster
- Writing Kubernetes Custom Controllers
- Contour Ingress Controller
- Syntax and Character Set
- Kubectl-Proxy

CHAPTER 28

Operator

An operator is a controller that uses a CRD to encapsulate operational knowledge for a specific application in an algorithmic and automated form. The *Operator* pattern allows us to extend the *Controller* pattern from the preceding chapter for more flexibility and greater expressiveness.

Problem

You learned in Chapter 27, "Controller", how to extend the Kubernetes platform in a simple and decoupled way. However, for extended use cases, plain custom controllers are not powerful enough, as they are limited to watching and managing Kubernetes intrinsic resources only. Moreover, sometimes we want to add new concepts to the Kubernetes platform, which requires additional domain objects. For example, let's say we chose Prometheus as our monitoring solution and want to add it as a monitoring facility to Kubernetes in a well-defined way. Wouldn't it be wonderful to have a Prometheus resource describing our monitoring setup and all the deployment details, similar to how we define other Kubernetes resources? Moreover, could we have resources relating to services we have to monitor (e.g., with a label selector)?

These situations are precisely the kind of use cases where CustomResourceDefinition (CRD) resources are very helpful. They allow extensions of the Kubernetes API, by adding custom resources to your Kubernetes cluster and using them as if they were native resources. Custom resources, together with a controller acting on these resources, form the *Operator* pattern.

This quote by Jimmy Zelinskie probably describes the characteristics of operators best:

> An operator is a Kubernetes controller that understands two domains: Kubernetes and something else. By combining knowledge of both areas, it can automate tasks that usually require a human operator that understands both domains.

Solution

As you saw in Chapter 27, "Controller", we can efficiently react to state changes of default Kubernetes resources. Now that you understand one half of the *Operator* pattern, let's have a look at the other half—representing custom resources on Kubernetes using CRD resources.

Custom Resource Definitions

With a CRD, we can extend Kubernetes to manage our domain concepts on the Kubernetes platform. Custom resources are managed like any other resource, through the Kubernetes API, and are eventually stored in the backend store etcd.

The preceding scenario is actually implemented with these new custom resources by the CoreOS Prometheus operator to allow seamless integration of Prometheus to Kubernetes. The Prometheus CRD is defined in Example 28-1, which also explains most of the available fields for a CRD.

Example 28-1. CustomResourceDefinition

```
apiVersion: apiextensions.k8s.io/v1
kind: CustomResourceDefinition
metadata:
  name: prometheuses.monitoring.coreos.com ❶
spec:
  group: monitoring.coreos.com             ❷
  names:
    kind: Prometheus                       ❸
    plural: prometheuses                   ❹
  scope: Namespaced                        ❺
  versions:                                ❻
  - name: v1                               ❼
    storage: true                          ❽
    served: true                           ❾
    schema:
      openAPIV3Schema: ....                ❿
```

❶ Name.

❷ API group it belongs to.

❸ Kind used to identify instances of this resource.

❹ Naming rule for creating the plural form, used for specifying a list of those objects.

❺ Scope—whether the resource can be created cluster-wide or is specific to a namespace.

❻ Versions available for this CRD.

❼ Name of a supported version.

❽ Exactly one version has to be the storage version used for storing the definition in the backend.

❾ Whether this version is served via the REST API.

❿ OpenAPI V3 schema for validation (not shown here).

An OpenAPI V3 schema can also be specified to allow Kubernetes to validate a custom resource. For simple use cases, this schema can be omitted, but for production-grade CRDs, the schema should be provided so that configuration errors can be detected early.

Additionally, Kubernetes allows us to specify two possible subresources for our CRD via the spec field `subresources`:[1]

scale
: With this property, a CRD can specify how it manages its replica count. This field can be used to declare the JSON path, where the number of desired replicas of this custom resource is specified: the path to the property that holds the actual number of running replicas and an optional path to a label selector that can be used to find copies of custom resource instances. This label selector is usually optional but is required if you want to use this custom resource with the HorizontalPodAutoscaler explained in Chapter 29, "Elastic Scale".

status
: When this property is set, a new API call becomes available that allows you to update only the `status` field of a resource. This API call can be secured individually and allows the operator to reflect the *actual* status of the resource, which might differ from the *declared* state in the `spec` field. When a custom

1 Kubernetes subresources are additional API endpoints that provide further functionality within a resource type.

resource is updated as a whole, any sent `status` section is ignored, as is the case with standard Kubernetes resources.

Example 28-2 shows a potential subresource path as is also used for a regular Pod.

Example 28-2. Subresource definition for a CustomResourceDefinition

```
kind: CustomResourceDefinition
# ...
spec:
  subresources:
    status: {}
    scale:
      specReplicasPath: .spec.replicas          ❶
      statusReplicasPath: .status.replicas      ❷
      labelSelectorPath: .status.labelSelector  ❸
```

❶ JSON path to the number of declared replicas.

❷ JSON path to the number of active replicas.

❸ JSON path to a label selector to query for the number of active replicas.

Once we define a CRD, we can easily create such a resource, as shown in Example 28-3.

Example 28-3. A Prometheus custom resource

```
apiVersion: monitoring.coreos.com/v1
kind: Prometheus
metadata:
  name: prometheus
spec:
  serviceMonitorSelector:
    matchLabels:
      team: frontend
  resources:
    requests:
      memory: 400Mi
```

The `metadata` section has the same format and validation rules as any other Kubernetes resource. The `spec` contains the CRD-specific content, and Kubernetes validates against the given validation rule from the CRD.

Custom resources alone are not of much use without an active component to act on them. To give them some meaning, we need again our well-known controller, which watches the lifecycle of these resources and acts according to the declarations found within the resources.

Controller and Operator Classification

Before we dive into writing our operator, let's look at a few kinds of classifications for controllers, operators, and especially CRDs. Based on the operator's action, broadly, the classifications are as follows:

Installation CRDs
: Meant for installing and operating applications on the Kubernetes platform. Typical examples are the Prometheus CRDs, which we can use for installing and managing Prometheus itself.

Application CRDs
: In contrast, these are used to represent an application-specific domain concept. This kind of CRD allows applications deep integration with Kubernetes, which involves combining Kubernetes with an application-specific domain behavior. For example, the ServiceMonitor CRD is used by the Prometheus operator to register specific Kubernetes Services to be scraped by a Prometheus server. The Prometheus operator takes care of adapting the Prometheus server configuration accordingly.

Note that an operator can act on different kinds of CRDs as the Prometheus operator does in this case. The boundary between these two categories of CRDs is blurry.

In our categorization of controller and operator, an operator is-a controller that uses CRDs.[2] However, even this distinction is a bit fuzzy as there are variations in between.

One example is a controller, which uses a ConfigMap as a kind of replacement for a CRD. This approach makes sense in scenarios where default Kubernetes resources are not enough but creating CRDs is not feasible either. In this case, ConfigMap is an excellent middle ground, allowing encapsulation of domain logic within the content of a ConfigMap. An advantage of using a plain ConfigMap is that you don't need to have the cluster-admin rights you need when registering a CRD. In certain cluster setups, it is just not possible for you to register such a CRD (e.g., when running on public clusters like OpenShift Online).

However, you can still use the concept of *Observe-Analyze-Act* when you replace a CRD with a plain ConfigMap that you use as your domain-specific configuration. The drawback is that you don't get essential tool support like `kubectl get` for CRDs; you have no validation on the API Server level and no support for API versioning.

2 is-a emphasizes the inheritance relationship between operator and controller, that an operator has all characteristics of a controller plus a bit more.

Also, you don't have much influence on how you model the `status` field of a ConfigMap, whereas for a CRD, you are free to define your status model as you wish.[3]

Another advantage of CRDs is that you have a fine-grained permission model based on the kind of CRD, which you can tune individually, as is explained in Chapter 26, "Access Control". This kind of RBAC security is not possible when all your domain configuration is encapsulated in ConfigMaps, as all ConfigMaps in a namespace share the same permission setup.

From an implementation point of view, it matters whether we implement a controller by restricting its usage to vanilla Kubernetes objects or whether we have custom resources managed by the controller. In the former case, we already have all types available in the Kubernetes client library of our choice. For the CRD case, we don't have the type information out of the box, and we can either use a schemaless approach for managing CRD resources or define the custom types on our own, possibly based on an OpenAPI schema contained in the CRD definition. Support for typed CRDs varies by client library and framework used.

Figure 28-1 shows our controller and operator categorization starting from simpler resource definition options to more advanced with the boundary between controller and operator being the use of custom resources.

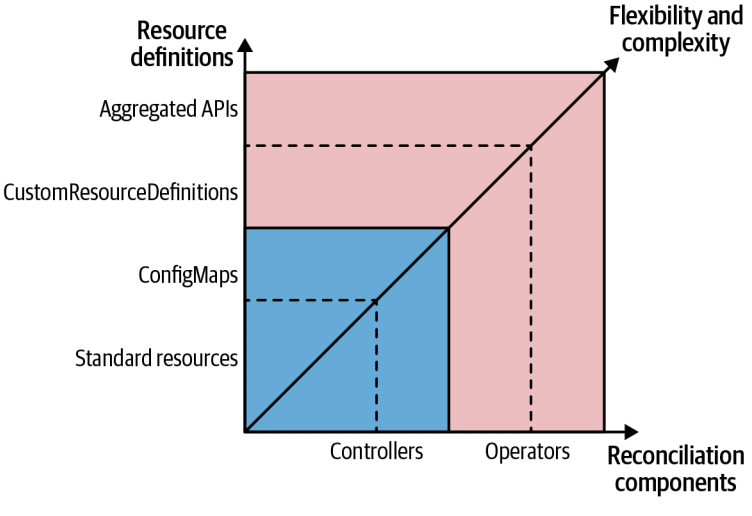

Figure 28-1. Spectrum of controllers and operators

3 However, you should be aware of common API conventions for `status` and other fields when designing your CRDs. Following common community conventions makes it easier for people and tooling to read your new API objects.

For operators, there is even a more advanced Kubernetes extension hook option. When Kubernetes-managed CRDs are not sufficient to represent a problem domain, you can extend the Kubernetes API with its own aggregation layer. We can add a custom-implemented `APIService` resource as a new URL path to the Kubernetes API.

To connect a Service that is backed by a Pod with the `APIService`, you can use a resource like that shown in Example 28-4.

Example 28-4. API aggregation with a custom APIService

```
apiVersion: apiregistration.k8s.io/v1beta1
kind: APIService
metadata:
  name: v1alpha1.sample-api.k8spatterns.io
spec:
  group: sample-api.k8spattterns.io
  service:
    name: custom-api-server
  version: v1alpha1
```

Besides the Service and Pod implementation, we need some additional security configuration for setting up the ServiceAccount under which the Pod is running.

After it is set up, every request to the API Server `https://<api server ip>/apis/sample-api.k8spatterns.io/v1alpha1/namespaces/<ns>/...` is directed to our custom Service implementation. It's up to this custom Service implementation to handle these requests, including persisting the resources managed via this API. This approach is different from the preceding CRD case, where Kubernetes itself completely manages the custom resources.

With a custom API Server, you have many more degrees of freedom, which allows you to go beyond watching resource lifecycle events. On the other hand, you also have to implement much more logic, so for typical use cases, an operator dealing with plain CRDs is often good enough.

A detailed exploration of the API Server capabilities is beyond the scope of this chapter. The official documentation as well as a complete sample-apiserver have more detailed information. Also, you can use the apiserver-builder library, which helps with implementing API Server aggregation.

Now, let's see how you can develop and deploy operators with CRDs.

Operator Development and Deployment

Several toolkits and frameworks are available for developing operators. The three main projects aiding in the creation of operators are as follows:

- Kubebuilder developed under the SIG API Machinery of Kubernetes itself
- Operator Framework, a CNCF project
- Metacontroller from Google Cloud Platform

We touch on each of these very briefly to give you a good starting point for developing and maintaining your own Operators.

Kubebuilder

Kubebuilder, a project by the SIG API Machinery,[4] is a framework and library for creating Kubernetes APIs via CustomResourceDefinitions.

It comes with outstanding documentation that also covers general aspects for programming Kubernetes. Kubebuilder's focus is on creating Golang-based operators by adding higher-level abstractions on top of the Kubernetes API to remove some of the overhead. It also offers scaffolding of new projects and supports multiple CRDs that can be watched by a single operator. Other projects can consume Kubebuilder as a library, and it also offers a plugin architecture to extend the support to languages and platforms beyond Golang. For programming against the Kubernetes API, Kubebuilder is an excellent starting point.

Operator framework

The Operator Framework provides extensive support for developing operators. It offers several subcomponents:

- The *Operator SDK* provides a high-level API for accessing a Kubernetes cluster and a scaffolding to start an operator project.
- The *Operator Lifecycle Manager* manages the release and updates of operators and their CRDs. You can think of it as a kind of "operator operator."
- *Operator Hub* is a publicly available catalog of operators dedicated to sharing operators built by the community.

4 Special Interest Groups (SIGs) are how the Kubernetes community organizes feature areas. You can find a list of current SIGs on the Kubernetes community site.

In the first edition of this book in 2019, we mentioned the high feature overlap of Kubebuilder and the Operator-SDK, and we speculated that both projects might eventually merge. It turned out that instead of a full merge, a different strategy was chosen by the community: all the overlapping parts have been moved to Kubebuilder, and the Operator-SDK uses Kubebuilder now as a dependency. This move is a good example of the power and self-healing effect of community-driven open source projects. The article "What Are the Differences Between Kubebuilder and Operator-SDK?" contains more information about the relationship between Kubebuilder and the Operator-SDK. The *Operator-SDK* offers everything needed for developing and maintaining Kubernetes operators. It is built on top of Kubebuilder and uses it directly for scaffolding and managing operators written in Golang. Beyond that, it benefits from Kubebuilder's plugin system for creating operators based on other technologies. As of 2023, the Operator-SDK provides plugins for creating operators based on Ansible playbooks or Helm Charts and Java-based operators that use a Quarkus runtime. When scaffolding a project, the SDK also adds the appropriate hooks for integration with the Operator Lifecycle Manager and the Operator Hub.

The *Operator Lifecycle Manager* (OLM) provides valuable help when using operators. One issue with CRDs is that these resources can be registered only cluster-wide and require cluster-admin permissions. While regular Kubernetes users can typically manage all aspects of the namespaces they have granted access to, they can't just use operators without interaction with a cluster administrator.

To streamline this interaction, the OLM is a cluster service running in the background under a service account with permission to install CRDs. A dedicated CRD called ClusterServiceVersion (CSV) is registered along with the OLM and allows us to specify the Deployment of an operator together with references to the CRD definitions associated with this operator. As soon as we have created such a CSV, one part of the OLM waits for that CRD and all its dependent CRDs to be registered. If this is the case, the OLM deploys the operator specified in the CSV. Then, another part of the OLM can be used to register these CRDs on behalf of a nonprivileged user. This approach is an elegant way to allow regular cluster users to install their operators.

Operators can be easily published at the Operator Hub. Operator Hub makes it easy to discover and install operators. The metadata-like name, icon, description, and more is extracted from the operator's CSV and rendered in a friendly web UI. Operator Hub also introduces the concept of *channels* that allow you to provide different streams like "stable" or "alpha," to which users can subscribe for automatic updates of various maturity levels.

Metacontroller

Metacontroller is very different from the other two operator building frameworks as it extends Kubernetes with APIs that encapsulate the common parts of writing custom controllers. It acts similarly to Kubernetes Controller Manager by running multiple controllers that are not hardcoded but are defined dynamically through Metacontroller-specific CRDs. In other words, it's a delegating controller that calls out to the service providing the actual controller logic.

Another way to describe Metacontroller is as declarative behavior. While CRDs allow us to store new types in Kubernetes APIs, Metacontroller makes it easy to define the behavior for standard or custom resources declaratively.

When we define a controller through Metacontroller, we have to provide a function that contains only the business logic specific to our controller. Metacontroller handles all interactions with the Kubernetes APIs, runs a reconciliation loop on our behalf, and calls our function through a webhook. The webhook gets called with a well-defined payload describing the CRD event. As the function returns the value, we return a definition of the Kubernetes resources that should be created (or deleted) on behalf of our controller function.

This delegation allows us to write functions in any language that can understand HTTP and JSON and that do not have any dependency on the Kubernetes API or its client libraries. The functions can be hosted on Kubernetes, or externally on a Functions-as-a-Service provider, or somewhere else.

We cannot go into many details here, but if your use case involves extending and customizing Kubernetes with simple automation or orchestration, and you don't need any extra functionality, you should have a look at Metacontroller, especially when you want to implement your business logic in a language other than Go. Some controller examples will demonstrate how to implement StatefulSet, Blue-Green Deployment, Indexed Job, and Service per Pod by using Metacontroller only.

Example

Let's look at a concrete operator example. We extend our example in Chapter 27, "Controller", and introduce a CRD of the type ConfigWatcher. An instance of this CRD then specifies a reference to the ConfigMap to watch and specifies which Pods to restart if this ConfigMap changes. With this approach, we remove the dependency of the ConfigMap on the Pods, as we don't have to modify the ConfigMap itself to add triggering annotations. Also, with our simple annotation-based approach in the Controller example, we can connect only a ConfigMap to a single application too. With a CRD, arbitrary combinations of ConfigMaps and Pods are possible.

This ConfigWatcher custom resource is shown in Example 28-5.

Example 28-5. Simple ConfigWatcher resource

```
apiVersion: k8spatterns.io/v1
kind: ConfigWatcher
metadata:
  name: webapp-config-watcher
spec:
  configMap: webapp-config ❶
  podSelector:             ❷
    app: webapp
```

❶ Reference to ConfigMap to watch.

❷ Label selector to determine Pods to restart.

In this definition, the attribute `configMap` references the name of the `ConfigMap` to watch. The field `podSelector` is a collection of labels and their values, which identify the Pods to restart.

We define the type of this custom resource with a CRD (shown in Example 28-6).

Example 28-6. ConfigWatcher CRD

```
apiVersion: apiextensions.k8s.io/v1
kind: CustomResourceDefinition
metadata:
  name: configwatchers.k8spatterns.io
spec:
  scope: Namespaced          ❶
  group: k8spatterns.io      ❷
  names:
    kind: ConfigWatcher      ❸
    singular: configwatcher  ❹
    plural: configwatchers
  versions:
  - name: v1                 ❺
    storage: true
    served: true
    schema:
      openAPIV3Schema:       ❻
        type: object
        properties:
          configMap:
            type: string
            description: "Name of the ConfigMap"
          podSelector:
            type: object
            description: "Label selector for Pods"
            additionalProperties:
              type: string
```

❶ Connected to a namespace.

❷ Dedicated API group.

❸ Unique kind of this CRD.

❹ Labels of the resource as used in tools like `kubectl`.

❺ Initial version.

❻ OpenAPI V3 schema specification for this CRD.

For our operator to be able to manage custom resources of this type, we need to attach a ServiceAccount with the proper permissions to our operator's Deployment. For this task, we introduce a dedicated Role used later in a RoleBinding to attach it to the ServiceAccount in Example 28-7. We explain the concept and usage of ServiceAccounts, Roles, and RoleBindings in much more details in Chapter 26, "Access Control". For now, it is sufficient to know that the Role definition in Example 28-6 grants permission for all API operations to any instance of ConfigWatcher resources.

Example 28-7. Role definition allowing access to custom resource

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: config-watcher-crd
rules:
- apiGroups:
  - k8spatterns.io
  resources:
  - configwatchers
  - configwatchers/finalizers
  verbs: [ get, list, create, update, delete, deletecollection, watch ]
```

With these CRDs in place, we can now define custom resources as in Example 28-5.

To make sense of these resources, we have to implement a controller that evaluates these resources and triggers a Pod restart when the ConfigMap changes.

We expand here on our controller script in Example 27-2 and adapt the event loop in the controller script.

In the case of a ConfigMap update, instead of checking for a specific annotation, we do a query on all resources of the kind ConfigWatcher and check whether the modified ConfigMap is included as a `configMap` value. Example 28-8 shows the reconciliation loop. Refer to our Git repository for the full example, which also includes detailed instructions for installing this operator.

Example 28-8. WatchConfig controller reconciliation loop

```
curl -Ns $base/api/v1/${ns}/configmaps?watch=true | \          ❶
while read -r event
do
  type=$(echo "$event" | jq -r '.type')
  if [ $type = "MODIFIED" ]; then                              ❷

    watch_url="$base/apis/k8spatterns.io/v1/${ns}/configwatchers"
    config_map=$(echo "$event" | jq -r '.object.metadata.name')

    watcher_list=$(curl -s $watch_url | jq -r '.items[]')      ❸

    watchers=$(echo $watcher_list | \                          ❹
               jq -r "select(.spec.configMap == \"$config_map\") | .metadata.name")

    for watcher in watchers; do                                ❺
      label_selector=$(extract_label_selector $watcher)
      delete_pods_with_selector "$label_selector"
    done
  fi
done
```

❶ Start a watch stream to watch for ConfigMap changes for a given namespace.

❷ Check for a `MODIFIED` event only.

❸ Get a list of all installed ConfigWatcher custom resources.

❹ Extract from this list all ConfigWatcher elements that refer to this ConfigMap.

❺ For every ConfigWatcher found, delete the configured Pod via a selector. The logic for calculating a label selector as well as the deletion of the Pods are omitted here for clarity. Refer to the example code in our Git repository for the full implementation.

As for the controller example, this controller can be tested with a sample web application that is provided in our example Git repository. The only difference with this Deployment is that we use an unannotated ConfigMap for the application configuration.

Although our operator is quite functional, it is also clear that our shell script-based operator is still quite simple and doesn't cover edge or error cases. You can find many more interesting, production-grade examples in the wild.

The canonical place to find real-world operators is Operator Hub. The operators in this catalog are all based on the concepts covered in this chapter. We have already seen how a Prometheus operator can manage Prometheus installations. Another