Using `client-go`

Writing Golang clients for talking to Kubernetes

Luca Sepe

Contents

1 Preface **1**

1.1 To get the most out of this notebook . . . 1
1.2 Who this notebook is for . . . 2
1.3 Download the example code files . . . 2

2 Setting up a local development environment **3**

2.1 Installing cURL . . . 3
2.2 Installing GNU Make . . . 3
2.2.1 Installing `make` on Linux . . . 4
2.2.2 Installing `make` on MacOs . . . 4
2.2.3 Installing `make` on Windows . . . 4
2.3 Installing Docker . . . 4
2.4 Installing `kubectl` . . . 5
2.5 Installing KinD . . . 5
2.6 Installing jq . . . 6

3 The Kubernetes API Server **8**

3.1 API conventions . . . 9
3.2 Resources . . . 10
3.3 Custom Resources . . . 13
3.3.1 (Hands-On) Creating custom resource definition . . . 13
3.3.2 (Hands-On) Creating custom objects . . . 16

4 Local Kubernetes development with KinD **17**

4.1 Using Gnu Make to automate your development workflow . . . 18
4.2 A quick summary about Makefiles and how they works . . . 19
4.2.1 Makefile Rules . . . 19
4.2.2 Makefile Variables . . . 20

5 Exploring the API server using cURL **21**
5.1 Role–based access control (RBAC) . . . 23
5.1.1 Role or ClusterRole . . . 23
5.1.2 RoleBinding and ClusterRoleBinding . . . 24

6 Introducing client-go **27**
6.1 Installing client-go . . . 27
6.1.1 Using the latest version . . . 27
6.1.2 Using a specific version . . . 28
6.2 Types of clients in client-go . . . 28
6.3 Initializing a client . . . 29
6.3.1 (Hands–On) Creating a rest.Config using default *kubeconfig* rules . . . 29
6.3.2 (Hands–On) Creating a rest.Config using flags to specify a custom *kubeconfig* file . . . 30

7 Using rest.RESTClient **31**
7.1 (Hands–On) Creating a deployment . . . 32
7.2 (Hands–On) Listing pods . . . 34
7.3 (Hands–On) Updating a deployment image . . . 36
7.4 (Hands–On) Deleting a deployment . . . 39

8 Using kubernetes.Clientset **41**
8.1 (Hands–On) Creating a deployment . . . 42
8.2 (Hands–On) Listing pods . . . 44
8.3 (Hands–On) Updating a deployment image . . . 46
8.4 (Hands–On) Deleting a deployment . . . 47

9 Using dynamic.Interface **49**
9.1 (Hands–On) Listing pods . . . 50
9.2 (Hands–On) Getting and updating a custom resource . . . 51

10 Using discovery.DiscoveryClient **55**
10.1 (Hands–On) Listing Kubernetes API resources . . . 55

11 Using labels and selectors **58**
11.1 (Hands–On) Creating labels and initializing selectors . . . 58
11.2 (Hands–On) List resources using label selectors . . . 60

12 (Hands-On) Display clients HTTP calls contents **63**

13 Watching for changes **67**
13.1 (Hands-On) Watching for changes (using `rest.RESTClient`) . . . 68
13.2 (Hands-On) Watching for changes (using `dynamic.Interface`) . . . 72
13.3 (Hands-On) Watching for changes (using `kubernetes.Clientset`) . . . 74

14 Using `RetryWatcher` **76**
14.1 (Hands-On) Watching for changes using the `RetryWatcher` . . . 77

15 Digging into `tools/cache` package **80**
15.1 `cache.ListerWatcher` . . . 80
15.2 `cache.Store` and `cache.Queue` . . . 80
15.3 `cache.Reflector` . . . 81
15.4 `cache.DeltaFIFO` . . . 81
15.5 `cache.Controller` . . . 82
15.6 `cache.Indexer` . . . 82
15.7 The *"informer"* concept . . . 83
15.8 `cache.SharedInformer` . . . 83
15.9 Recap on `Controller`, `Reflector`, `DeltaFIFO`, `SharedIndexInformer` . . . 84

16 Using informers **86**
16.1 (Hands-On) Watching for secrets using `SharedInformer` . . . 87

17 WorkQueues **90**
17.1 (Hands-On) Writing a controller using `RateLimitingInterface` and `SharedIndexInformer` . . . 92

18 Using code generators for Custom Resource Defintions (CRD) **100**
18.1 (Hands-On) Writing the type definition source code with code generator tags . . . 103
18.2 (Hands-On) Registering your types to the API server . . . 105
18.3 (Hands-On) Executing the code generation for your Custom Resource types . . . 106
18.4 (Hands-On) Using the generated clientset in your program . . . 107

1 Preface

Welcome to:

Using `client-go`

Writing Kubernetes Client applications using Go

and thanks for choosing to spend some time with me.

This is a Go programming notebook about Kubernetes `client-go` library; it will:

- cover the foundations and the core ideas
- inspect the packages showing structs and interfaces relations
- introduce you to the whole concepts preparatory to master custom controllers implementation
- show you how to create your custom resource and using generators to create clients, listers, informers etc.
- how to write an operator to reconcile your custom resource *(coming soon)*

1.1 To get the most out of this notebook

A basic knowledge of the Go language is assumed throughout this book.

If you are not yet familiar with this programming language, consider running through the online tutorial before you begin reading (go.dev/tour).

To run the examples, you will need:

- Go installed – examples were written using the 1.17 version
- GNU Make tool

- Docker required to make `kind` work
- KinD to run Kubernetes on your local computer
- kubectl to run commands against Kubernetes clusters
- jq to slice, filter, map and transform `kubectl` JSON output

I will step through the process of installing all the tools required throughout this notebook.

1.2 Who this notebook is for

You're a cloud-native developer or an SRE or are you just interested in writing client applications wanting to get the maximum out of Kubernetes.

1.3 Download the example code files

You can download the example code files for this notebook from GitHub at:

» https://github.com/lucasepe/using-client-go

In case there's an update to the code, it will be updated on the existing GitHub repository.

7 Using `rest.RESTClient`

`rest.RESTClient` provides rich APIs for various settings and a fluent interface to simplify Kubernetes API calls.

- has support for core and custom resources
- it is the base on which the other types of clients are built

The basic steps to perform one of the possible operations (i.e. get, delete, create, update etc...) using `rest.RESTClient` are:

1. define the type of resource to use and the related group, version and operation (get, create, list, delete, etc.)
2. load and configure the `rest.Client` configuration
3. once you get the configuration object set the necessary values for the APIs you need to call (such as the required path, group, version, serialization and deserialization tools, etc.)
4. create a `rest.RESTClient` instance, using the the configuration object as input parameter
5. using the fluent API on the `rest.RESTClient` instance, define all the parameters (namespace, resources, eventually the payload, the result object, etc.)

You will see how to apply these steps to:

- create a deployment
- list pods
- update a deployment image
- delete a deployment

Source code @ https://github.com/lucasepe/using-client-go/tree/main/using-rest-client.

For each example the equivalent `kubectl` command will be shown.

7.1 (Hands-On) Creating a deployment

» *This example emulate the command:* `kubectl create deployment nginx --image=nginx`.

The type of resource is a *Deployment* and the related operation is a *create* (POST); searching the Kubernetes API reference you can find path, group, version and required body:

```
package main

import (
    "context"
    "encoding/json"
    "fmt"

    appsv1 "k8s.io/api/apps/v1"
    corev1 "k8s.io/api/core/v1"
    metav1 "k8s.io/apimachinery/pkg/apis/meta/v1"
    "k8s.io/client-go/kubernetes/scheme"
    "k8s.io/client-go/rest"
    "k8s.io/client-go/tools/clientcmd"
)

func main() {
    configLoader := clientcmd.NewNonInteractiveDeferredLoadingClientConfig(
        clientcmd.NewDefaultClientConfigLoadingRules(),
        &clientcmd.ConfigOverrides{},
    )

    namespace, _, err := configLoader.Namespace()
    if err != nil {
        panic(err)
    }

    cfg, err := configLoader.ClientConfig()
    if err != nil {
        panic(err)
    }

    // POST /apis/apps/v1/namespaces/{namespace}/deployments

    // the base API path "/apis"
    cfg.APIPath = "apis"
    // the Deployment group and version "/apps/v1"
    cfg.GroupVersion = &appsv1.SchemeGroupVersion
    // specify the serializer
    cfg.NegotiatedSerializer = scheme.Codecs.WithoutConversion()
```

10 Using `discovery.DiscoveryClient`

While the clients seen so far have the main purpose of retrieving and managing Kubernetes objects, `discovery.DiscoveryClient` provides ways to discover server–supported API groups, versions and resources.

Let's see how to use it to implement a functionality similar to the `kubectl api-resources` command.

Source code @ https://github.com/lucasepe/using-client-go/tree/main/using-discovery-client.

10.1 (Hands-On) Listing Kubernetes API resources

```go
package main

import (
  "encoding/json"
  "fmt"

  "k8s.io/apimachinery/pkg/util/errors"
  "k8s.io/client-go/discovery"
  "k8s.io/client-go/tools/clientcmd"
)

func main() {
  configLoader := clientcmd.NewNonInteractiveDeferredLoadingClientConfig(
    clientcmd.NewDefaultClientConfigLoadingRules(),
    &clientcmd.ConfigOverrides{},
  )

  rc, err := configLoader.ClientConfig()
  if err != nil {
    panic(err)
  }

  // create a new DiscoveryClient using the given config
```

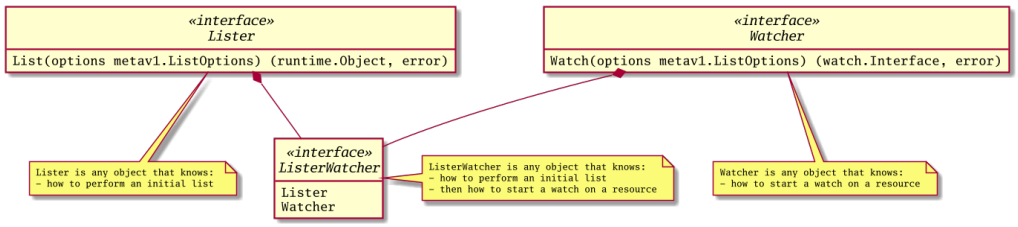
15 Digging into `tools/cache` package

In order to understand Informers let's dig more into `tools/cache` package.

15.1 `cache.ListerWatcher`

`ListWatcher` is something that list all resources of a specific kind (pods, deployments, namespaces, etc..) and then sets up watches on them.

» https://github.com/kubernetes/client-go/blob/master/tools/cache/listwatch.go

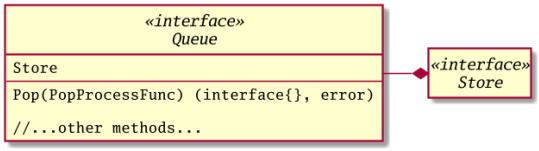


15.2 `cache.Store` and `cache.Queue`

`ListWatcher`, using a Kubernetes client, collects resources of a particular kind and some related events; then these things are saved in a generic object storage – the `Store`.

» https://github.com/kubernetes/client-go/blob/master/tools/cache/store.go

» https://github.com/kubernetes/client-go/blob/master/tools/cache/fifo.go



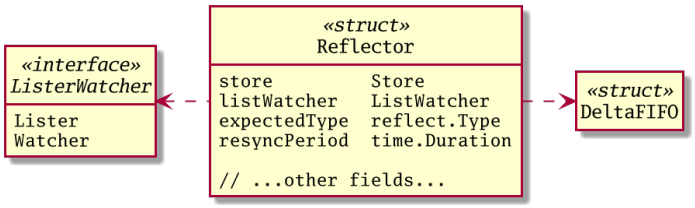
`Queue` is a `Store`, but with a `Pop()` function.

15.3 cache.Reflector

`Reflector` reflects the contents of a Kubernetes message channel into a cache.

- puts into the `Store` the results of the `ListerWatcher` `List(...)` function
- turns the incoming `WatchEvents` into updates, removals and additions of items in the `Store`

» https://github.com/kubernetes/client-go/blob/master/tools/cache/reflector.go

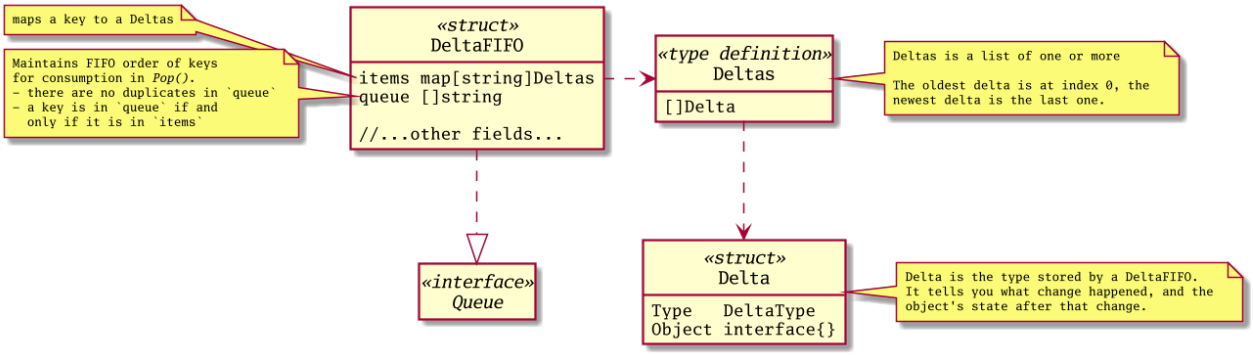


Then if you want to do something in your program with resources of a particular kind, you can look to the cache rather than to the API server itself.

15.4 cache.DeltaFIFO

Is the `Store` implementation used by `Reflector`.

» https://github.com/kubernetes/client-go/blob/master/tools/cache/delta_fifo.go



18 Using code generators for Custom Resource Defintions (CRD)

The Kubernetes API server is easily extendable by Custom Resource Defintions (CRD).

`code-generators` can be used to build native, versioned clients, informers and other helpers for your Custom Resource Definition (CRD). Infact, `client-go` requires that `runtime.Object` types (that must be implemented by your custom resources) must have `DeepCopy()` methods.

Besides many `code-generators` are available:

- `deepcopy-gen` creates a method `func (t* T) DeepCopy() *T` for each type `T`
- `client-gen` creates clientsets for your custom resource API groups
- `lister-gen` creates listers for your custom resurces
- `informer-gen` creates informers for your custom resurces

To show how these generators work let's create a custom resource definition to expose expressions to evaluate.

Then we will create a program that using the generated clientset, fetch the specified *expression* resource and evaluate it (using the spec data) saving the result in the resource status.

Before we start coding, we need to define the CRDs that the program will handle. As with any other API, Kubernetes allows you to define its custom API objects using OpenAPI specification.

```
apiVersion: apiextensions.k8s.io/v1
kind: CustomResourceDefinition
metadata:
  # name must match the spec fields below, and be in the form: <plural>.<group>
  name: expressions.example.org
spec:
  # group name to use for REST API: /apis/<group>/<version>
  group: example.org
  names:
    # kind is normally the CamelCased singular type. Your resource manifests use this.
```

```
    kind: Expression
    listKind: ExpressionList
    # plural name to be used in the URL: /apis/<group>/<version>/<plural>
    plural: expressions
    # singular name to be used as an alias on the CLI and for display
    singular: expression
    # shortNames allow shorter string to match your resource on the CLI
    shortNames:
    - exp
  # either Namespaced or Cluster
  scope: Namespaced
  # list of versions supported by this CustomResourceDefinition
  versions:
  - name: v1alpha1
    additionalPrinterColumns:
    - jsonPath: .spec.body
      description: The expression to evaluate
      name: Expression
      type: string
    - jsonPath: .status.result
      description: The evaluation result
      name: Result
      type: string
    schema:
      openAPIV3Schema:
        type: object
        properties:
          spec:
            properties:
              body:
                type: string
              data:
                type: string
            required:
            - body
            - data
            type: object
          status:
            properties:
              result:
                type: string
              error:
                type: string
            type: object
        type: object
    # Each version can be enabled/disabled by Served flag
    served: true
    # One and only one version must be marked as the storage version
    storage: true
```

```
    subresources:
      status: {}
status:
  acceptedNames:
    kind: ""
    plural: ""
  conditions: []
  storedVersions: []
```

Place it in the `manifests/crds` folder. Below the folder layout:

```
using-codegen
└── manifests
    ├── crds                        # contains Custom Resource Definition YAML files
    │   └── expression-crd.yaml
    └── examples                    # contains Custom Resource examples YAML files
        ├── demo1.yaml
        ├── demo2.yaml
        ├── demo3.yaml
        └── demo4.yaml
```

Here is how an expression appears (this is in the `manifests/examples` directory):

```
apiVersion: example.org/v1alpha1
kind: Expression
metadata:
  name: demo1
  namespace: default
spec:
  body: x + y + z
  data: |-
    {
      "x": 8,
      "y": 1,
      "z": 7
    }
```

Now, we can write the type definition with the tags to generate deepcopy functions and the clientset.