Configuring Your Kubernetes Cluster on the Next Level

Lucas Käldström
14th of November 2018 - KubeCon Shanghai
Lucas Käldström, Upper Secondary School Student, 19 years old

CNCF Ambassador, Certified Kubernetes Administrator and Kubernetes SIG Lead

Speaker at KubeCon in Berlin, Austin & Copenhagen

Kubernetes approver and subproject owner, active in the community for ~3 years

Driving luxas labs which currently performs contracting for Weaveworks

A guy that has never attended a computing class
Agenda

1. Where are we at today?
   a. Current status & limitations

2. Where do we want to go?
   a. ComponentConfig hallway pitch
   b. Solutions and conventions
   c. Technical deep-dive

3. Let’s write a ComponentConfig Go application!
   a. Walkthrough of a demo Go app (https://github.com/luxas/sample-config)

4. The future
   a. Roadmap & how you can help
I. Current status
Current status

1. Kubernetes releases at a fast pace -- every three months, hard to keep up

2. As the software is quickly evolving, the configuration knobs are too
   a. => backwards-incompatible changes in configuration needs to be made

3. The admin interface when configuring Kubernetes components is flags
   a. If you try to use a flag removed in new version => exit 1
   b. Need to build complex version-dependent flag set logic

4. Flag names and functionality is a bit inconsistent between components
   a. (Currently also many inconsistencies in the underlying config structs)
1. External ComponentConfig structs are available for components:
   a. k8s.io/kubelet (v1beta1)
   b. k8s.io/kube-proxy (v1alpha1)
   c. k8s.io/kube-scheduler (v1alpha1)
   d. k8s.io/kube-controller-manager (v1alpha1)

2. The kubelet, kube-proxy and kube-scheduler can read ComponentConfig from a file using the `--config` flag
   a. Only the kubelet has an API version that is beta or higher, so only for the kubelet it is recommended to use ComponentConfig in production

3. Shared types:
   a. LeaderElectionConfiguration and DebuggingConfiguration are hosted in k8s.io/apiserver
   b. ClientConnectionConfiguration is hosted in k8s.io/apimachinery
II. Enter ComponentConfig
End goal:

$ kube-controller-manager --config config.yaml

```yaml
apiVersion: kubecontrollermanager.config.k8s.io/v1
kind: KubeControllerManagerConfiguration
controllers:
  csrSigning:
    clusterSigningCertFile: /some/path
namespace:
  concurrentNamespaceSyncs: 5
nodeLifecycle:
  enableTaintManager: true
```
Hallway pitch

1. All components read a config file that follows Kubernetes API conventions*
2. When upgrading, the new binary can still read the older API structure
3. Higher-level systems can easily access the config structs via vending
4. Common configuration types are shared between components
5. The UX for integrating with any Kubernetes-like component gets consistent

*To a reasonable extent. ComponentConfigs aren’t REST resources, so some of the guidelines aren’t applicable
ComponentConfig conventions

API group name: `{samplecomponent}.config.k8s.io`

API version: Follows K8s standard, e.g. v1alpha1, v1alpha2, v1beta1, v1, v2beta1, v2

Kind name: `{SampleComponent}Configuration`

Code structure as of v1.12:

apiVersion: kubecontrollermanager.config.k8s.io/v1beta1
kind: KubeControllerManagerConfiguration

controllers:
  csrSigning:
    clusterSigningCertFile: /some/path

namespace:
  concurrentNamespaceSyncs: 5

nodeLifecycle:
  enableTaintManager: true
Common / shared config reuse

One common “problem” so far is that the components have been using nearly the same types of fields, but using custom and different schemas. Real-world example on the right.

To mitigate this, we’ve introduced “shared config types” packages in:
- k8s.io/apimachinery/pkg/apis/config/v1alpha1
- k8s.io/apiserver/pkgs/apis/config/v1alpha1

These packages host structs like `LeaderElectionConfiguration` and `ClientConnectionConfiguration`. This shared set will grow over time.

Real-world example:

```
apiVersion: kubeproxy.config.k8s.io/v1alpha1
kind: KubeProxyConfiguration
clientConnection:
  burst: 10
  contentType: application/vnd.kubernetes.protobuf
  kubeconfig: /var/lib/kube-proxy/kubeconfig.conf
  qps: 5

apiVersion: kubelet.config.k8s.io/v1beta1
kind: KubeletConfiguration
contentType: application/vnd.kubernetes.protobuf
kubeAPIBurst: 10
kubeAPIQPS: 5
kubeConfig: /etc/kubernetes/kubelet.conf
```
External and internal versions

- An **external type** is a struct that can be encoded and decoded.
  - It **must** have JSON tags set on all its fields

- An **internal type** is a struct that the program uses internally in all code
  - It should not have JSON tag set.
  - The internal type’s schema should equal the schema of the latest stable external type.

- **External types are defaulted**
  - After an external type has been decoded from a file, it is automatically defaulted

- **External types register conversions** to the internal type
  - This in order to make it possible for the program to use the config post-decoding

- **Auto-generated code** is created to minimize what needs to be written
  - Auto-generated code is used for conversion, nested defaulting and implementing interfaces
III. Sample Go application
Guide: Use ComponentConfig for your app

Let’s say you’re building an addon to Kubernetes, and want it to have the same “look and feel” as the other k8s components. You don’t like to reinvent the wheel and reimplement all the API machinery features, but re-use the Kubernetes API machinery framework.

Here’s a walkthrough of the files to be written for it to work:

- **types.go** -- Contains the types for the API group
- **register.go** -- Registers the types in a SchemeBuilder.
- **doc.go** -- Includes meta tags that act as parameters to code generators
- **auto-generated code for defaulting, conversions and deepcopy interfaces**
- **For external packages only:**
  - **conversion.go** -- Conversion code from external to internal API.
  - **defaults.go** -- Contains defaulting functions for the structs.
Guide: Directory structure

Here’s an example directory structure.

A couple of things to note:

- The internal types are in `pkg/apis/config`
  - It does not register any defaulting or conversions.
- The external types are versioned and defaulted
  - (Ideally) lossless conversions exist between every external type and the internal type
- `scheme.go` is in a dedicated package
  - This allows for easy imports and encoding/decoding
- The internal types should equal the latest stable types
  - The latest stable types’ conversions to internal == no-op
Guide: What we’ll do

A Go program that reads a file with either v1 or v1beta1 config, if `--config` is set, or outputs the default v1 config. How we’ll do this:

a. Create an internal package in `pkg/apis/config` with the internal `MyAppConfiguration` struct.
b. Create two external packages in `pkg/apis/config/{v1,v1beta1}` with external structs
c. Create defaulting logic for the external types in `defaults.go` files
d. Create custom conversion logic from v1beta1 -> internal type (== v1)
e. Create a scheme with the internal and external packages linked
f. Create a small program in `cmd/sample-config` that reads a file into the internal config, or uses the v1 defaults
Guide: What we’ll do

$ bin/sample-config --config [config-file]

If [config-file] is set:

```
cfg.yaml (v1|v1beta1) → Read → Decode → Default → Convert
```

If [config-file] is not set:

```
v1.MyAppConfiguration{} → Flags → Default → Convert → Convert to v1 → Encode → stdout
```

Internal type for use in program

Default

Convert to v1

Encode

stdout
Any serializable type must implement runtime.Object and embed TypeMeta!

Internal, has same schema as v1, but no json tags

```go
// +k8s:deepcopy-gen:interfaces=k8s.io/apimachinery/pkg/runtime.Object
type MyAppConfiguration struct {
    metav1.TypeMeta
    // ClientConnection configures the connection to Kubernetes
    ClientConnection apimachineryconfig.ClientConnectionConfiguration
    // LeaderElection configures so the component can be HA-deployed
    LeaderElection apiserverconfig.LeaderElectionConfiguration
    // Server holds configuration settings for the HTTPS server
    ServerServerConfiguration
}
type ServerConfiguration struct {
    // Default: "0.0.0.0"
    Address string
    // Default: 10250
    Port uint32
    // TLSConfig holds settings for the TLS configuration
    TLSConfig TLSConfig
}

type TLSConfig struct {
    // TLSCertFile string
    // TLSPrivateKeyFile string
}
```

External v1beta1 with json tags

```go
// +k8s:deepcopy-gen:interfaces=k8s.io/apimachinery/pkg/runtime.Object
type MyAppConfiguration struct {
    metav1.TypeMeta `json:"",inline"
    // ClientConnection configures the connection to Kubernetes
    ClientConnection apimcfgv1.ClientConnectionConfiguration `json:"clientConnection"
    // LeaderElection configures so the component can be deployed in HA mode on k8s
    LeaderElection apiscfgv1.LeaderElectionConfiguration `json:"leaderElection"
    // Default: "0.0.0.0"
    ServerAddress string `json:"serverAddress"`
    // Default: 10250
    HTTPSPort uint32 `json:"port"
    // TLSConfig holds settings for the TLS configuration
    TLSConfig TLSConfig `json:"tlsConfig"
}
type TLSConfig struct {
    // TLSCertFile string
    // TLSPrivateKeyFile string
}
```
package config

const GroupName = "config.luxaslabs.com"

var (
    SchemeBuilder = runtime.NewSchemeBuilder(
        addKnownTypes,
    )
    localSchemeBuilder = &SchemeBuilder
    AddToScheme = localSchemeBuilder.AddToScheme
    // SchemeGroupVersion is the group & version for this scheme
    SchemeGroupVersion = schema.GroupVersion{
        Group: GroupName,
        Version: runtime.APIVersionInternal,
    }
}

// Adds the list of known types to the given scheme.
func addKnownTypes(scheme *runtime.Scheme) error {
    scheme.AddKnownTypes(SchemeGroupVersion,
        &MyAppConfiguration{},
    )
    return nil
}

package v1beta1

const GroupName = "config.luxaslabs.com"

var (
    SchemeBuilder = runtime.NewSchemeBuilder(
        addKnownTypes,
        addDefaultingFuncs,
    )
    localSchemeBuilder = &SchemeBuilder
    AddToScheme = localSchemeBuilder.AddToScheme
    // SchemeGroupVersion is the group & version for this scheme
    SchemeGroupVersion = schema.GroupVersion{
        Group: GroupName,
        Version: "v1beta1",
    }
}

func addKnownTypes(scheme *runtime.Scheme) error {
    scheme.AddKnownTypes(SchemeGroupVersion,
        &MyAppConfiguration{},
    )
    return nil
}
Guide 3/7: The doc.go file

In doc.go, it is specified how the autogenerated code should be generated.

- `+k8s:deepcopy-gen` should always be `package` for ComponentConfig pkgs.
  - deepcopy-gen makes serializable configs implement runtime.Object.
- `+k8s:conversion-gen` points to the internal package and optional shared pkgs
  - conversion-gen creates Convert_v1_To_v2 functions and registers those conversions

External sample:
```go
// +k8s:deepcopy-gen=package
// +k8s:conversion-gen=github.com/company/my-app/pkg/apis/config
// +k8s:conversion-gen=k8s.io/apimachinery/pkg/apis/config/v1alpha1
// +k8s:conversion-gen=k8s.io/apiserver/pkg/apis/config/v1alpha1
// +k8s:defaulter-gen=TypeMeta
package v1beta1
```

Internal sample:
```
// +k8s:deepcopy-gen=package
package config
```

- `+k8s:defaulter-gen` should always be `TypeMeta` for ComponentConfig pkgs.
  - defaulter-gen registers defaulting funcs for all types that embed the parameter (TypeMeta).
This file must include the `addDefaultingFuncs(scheme)` function that in turn calls the auto-generated `RegisterDefaults(scheme)` function. The format of the defaulting function is `SetDefaults_{SampleStruct}`. Defaulting fns from shared pkgs are optional and can be called via `RecommendedDefault{SampleStruct}`.

```go
class v1beta1

func addDefaultingFuncs(scheme *runtime.Scheme) error {
    return RegisterDefaults(scheme)
}

func SetDefaults_MyAppConfiguration(obj *MyAppConfiguration) {
    if len(obj.ServerAddress) == 0 {
        obj.ServerAddress = "0.0.0.0"
    }
    if obj.HTTPSPort == 0 {
        obj.HTTPSPort = 9090
    }

    apimachineryconfigv1.RecommendedDefaultClientConnectionConfiguration(&obj.ClientConnection)
    apiserverconfigv1.RecommendedDefaultLeaderElectionConfiguration(&obj.LeaderElection)
}
```
Guide 5/7: The conversions.go file

For straightforward conversions between structs, where the field name and type match, autogenerated code is created automatically. But when more complex changes are made to a new API, you must write the `Convert_{v1}_{SampleStruct}_To_{internal}_{SampleStruct}` function yourself.

Remember to first call the (partial) autogenerated portion of the conversion!

```go
code snippet
package v1beta1

func Convert_v1beta1_MyAppConfiguration_To_config_MyAppConfiguration(in *MyAppConfiguration, out *config.MyAppConfiguration, s conversion.Scope) error {
    if err := autoConvert_v1beta1_MyAppConfiguration_To_config_MyAppConfiguration(in, out, s); err != nil {
        return err
    }
    out.Server.Address = in.ServerAddress
    out.Server.Port = in.HTTPSPort
    out.Server.TLSCertFile = in.TLSConfig.TLSCertFile
    out.Server.TLSPrivateKeyFile = in.TLSConfig.TLSPrivateKeyFile
    return nil
}
```
A scheme holds all registered types, defaulting and conversion functions centralized in one place. Use the `AddToScheme` functions to register a specific package.

In the scheme package, all known versions are aggregated and registered in the global Scheme variable.

Codecs provide encoding/decoding functions for the types in the scheme.
Guide 7/7: The hard work pays off!

Now, we can read any file with an external version directly into the internal version the program may use using `runtime.DecodeInto()`.

The internal version can also be populated from defaults in an external version using Scheme.Default() & .Convert().

Finally, it’s easy to marshal any object into whatever form using a generic encoder for a specific external version, and `runtime.Encode()`.

```go
// decodeFileInto reads a file and decodes the it into an internal type
func decodeFileInto(filePath string, obj runtime.Object) error {
    content, err := ioutil.ReadFile(filePath)
    if err != nil { return err }
    // Regardless of if the bytes are of the v1 or v1beta1 version,
    // it will be read successfully and converted into the internal version
    return runtime.DecodeInto(scheme.Codecs.UniversalDecoder(), content, obj)
}

// populateV1Defaults populates cfg based on v1 defaults
func populateV1Defaults(cfg *config.MyAppConfiguration) error {
    // Create a new config of some external version,
    // default it, convert it into the internal version
    v1cfg := &v1.MyAppConfiguration{}
    scheme.Scheme.Default(v1cfg)
    return scheme.Scheme.Convert(v1cfg, cfg, nil)
}

// marshalYAML marshals any ComponentConfig object registered in the scheme for the specific version
func marshalYAML(obj runtime.Object, groupVersion schema.GroupVersion) ([]byte, error) {
    // yamLEncoder is a generic-purpose encoder to YAML for this scheme
    // versionSpecificEncoder writes out YAML bytes for exactly this v1beta1 version
    versionSpecificEncoder := scheme.Codecs.EncoderForVersion(yamLEncoder, groupVersion)
    return runtime.Encode(versionSpecificEncoder, obj)
}
```
Roadmap

- v1.11
  - [The KEP for this feature](#) was designed, reviewed and approved
  - Internal refactoring for kubelet, kube-proxy, kube-controller-manager and kube-scheduler
  - The k8s.io/{component} repos for the mentioned components were created

- v1.13
  - Improve unit, API roundtrip, defaulting and validation testing coverage
  - Start the work of graduating the schemas of kube-proxy, kube-controller-manager and kube-scheduler from v1alpha1 to v1beta1

- v1.14
  - Refactor the API server’s codebase to support reading configuration from a file
  - (Hopefully by this time) Have all components’ API versions be beta or higher
Recap

1. ComponentConfig is the pattern of storing the config in a file
   a. This allows for “GitOps”-style deployments of Kubernetes components

2. This effort aims to provide an unified UX for configuring Kubernetes
   a. But you can also use these patterns in your Kubernetes extension or any application!

3. With the k8s ComponentConfig types in `k8s.io/*` repos, it’s possible to create higher-level tools that configure Kubernetes declaratively

4. Going forward we need component-focused contributors to design a stable (v1beta1 or higher) schema for all Kubernetes components
Thank you!

Get in touch:
@luxas on Github
@kubernetesonarm on Twitter
lucas@luxaslabs.com