Understanding the APIService object

Learn about the APIService object and its behind-the-scenes role.

We'll cover the following

Overview

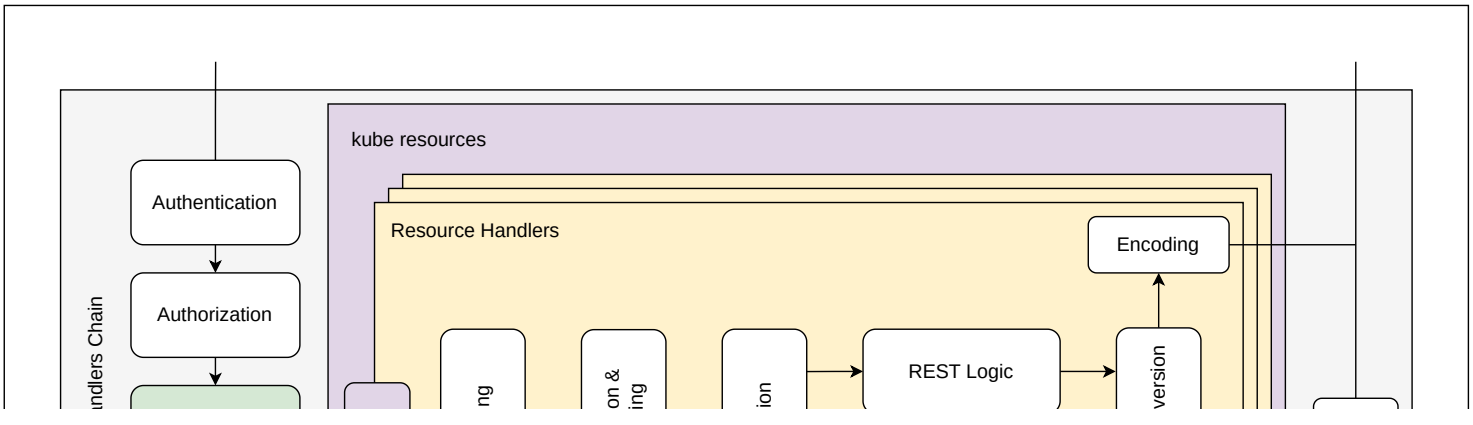
Before diving into the complexities of how the aggregated apiserver works, let's take a look at how to register our aggregated apiserver.

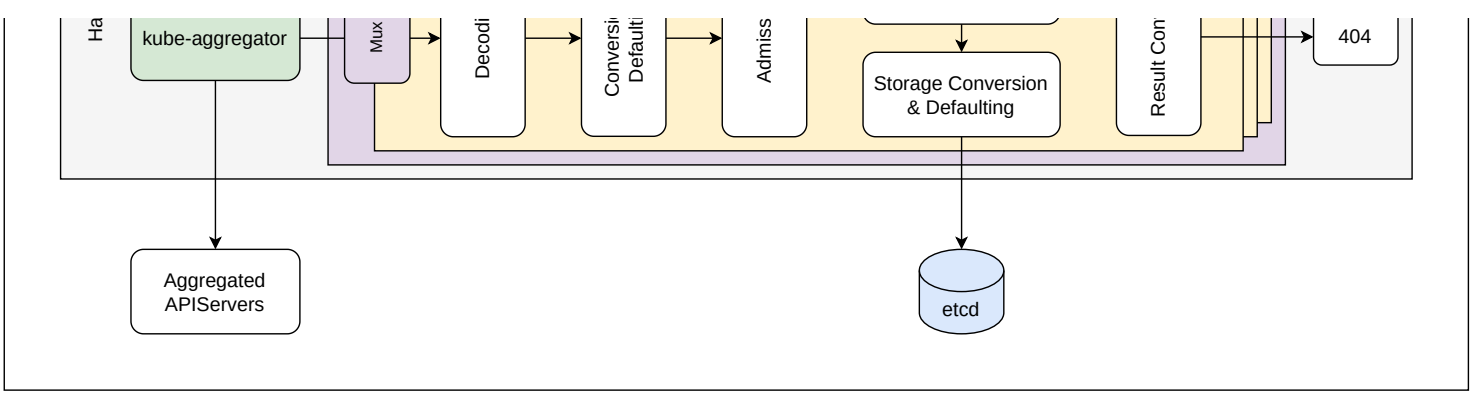
The aggregator layer

We can have multiple aggregated apiservers. They run separately alongside the `kube-apiserver` to provide REST APIs. They could run outside of the Kubernetes cluster, or inside of a cluster as Pods. The `kube-apiserver` provides core Kubernetes APIs, while aggregated apiservers serve our custom APIs.

API groups served by an aggregated apiserver are automatically proxied by the `kube-apiserver` to the aggregated apiserver. All those APIs are detectable in the `kube-apiserver`. In other words, the `kube-apiserver` aggregates these APIs and exposes them uniformly for external access, such as `kubectl` and other API clients. This brings a consistent user experience, because users do not need to care about where the APIs are actually served. All the users need to know is the address of the `kube-apiserver`. This can also help protect the security of aggregated apiservers.

In the `kube-apiserver`, there is a component called the `kube-aggregator` to handle this proxying work. Below, the diagram illustrates the whole process of how the `kube-apiserver` serves the REST APIs.





The kube-apiserver with an integrated kube-aggregator

1. The `kube-apiserver` automatically discovers the API groups by visiting `/apis` and `/apis/<group-name>` of all the aggregated apiservers.
2. All the APIs are exposed at the `kube-apiserver`.
3. The `kube-apiserver` receives all the requests and passes them to a handler chain consisting of authentication, audit logging, impersonation, in-flight throttle, authorization, etc.
4. The `kube-apiserver` knows self-provided APIs (Kubernetes APIs and custom resource APIs defined by CRDs) and aggregated APIs clearly. It passes all the self-provided APIs to built-in resource handlers for post processing, such as decoding, conversion, defaulting, version conversion, etc.
5. As the Kubernetes API server recognizes the aggregated APIs, it intercepts them and forwards them to the aggregated apiservers. This is done by the `kube-aggregator`. The `kube-aggregator` does not have to know all the resources served in the API group `/api/<group-name>/<version>`. It just does lazy proxying. All the requests under the matching HTTP path will be proxied.

But how does the `kube-apiserver` know exactly where and how to talk to aggregated apiservers? Let's look at this process in detail.

The `APIService` object

An `APIService` object must be created to register an aggregated apiserver to the `kube-apiserver`. This object must be created whenever we want to use APIs served by the aggregated apiserver. When we delete this object, the `kube-apiserver` will stop exposing the APIs served by the aggregated apiserver.

In this object, we specify a group and version of the serving APIs by this aggregated apiserver. This does *not* mean we can only serve APIs with one group and one version in our aggregated apiserver—we could serve multiple groups and versions. We only need to declare multiple `APIService` objects to indicate all the serving groups and versions.

Normally, we'd run aggregated apiservers with multiple replicas for high availability. If we are running aggregated apiservers inside of a Kubernetes cluster, we can expose them with a service and specify the service information in the `APIService` object, as shown below:

```
1  apiVersion: apiregistration.k8s.io/v1
2  kind: APIService
3  metadata:
```

```
  name: v1alpha1.pwk.educative.io
spec:
  group: pwk.educative.io
  version: v1alpha1
  service:
    name: my-pwk-demo
    namespace: pwk-system
  # caBundle: base64-encoded-caBundle
  insecureSkipTLSVerify: true
  groupPriorityMinimum: 1000
  versionPriority: 100
```

A sample APIService object

The service could be a normal `ClusterIP` service in the cluster. It can also be an `ExternalName` service if we run it outside of a Kubernetes cluster. Here, the serving port of the service must be `443`, but the Pods could use other ports instead. Our aggregated apiservers are running securely with certificates. We do need to provide a certificate authority (CA) for the `kube-apiserver` to trust our exposing service. In the example above, we set `insecureSkipTLSVerify` to `true`, which disables the TLS certificate verification when the `kube-apiserver` communicates with this server. However, this is not strongly encouraged for any production environment, which may get man-in-the-middle-attacks. Instead, we use `caBundle` to specify a PEM encoded CA bundle. This is very important for any production clusters.

At the end of the example, there are two priorities, `groupPriorityMinimum` and `versionPriority`.

The `groupPriorityMinimum` field determines the least possible priority that this group should have. In Kubernetes, there are many API groups. All these groups are sorted primarily based on `groupPriorityMinimum`. Below is a list of preset priority values for native Kubernetes API groups.

```
// The proper way to resolve  letting the aggregator know the desired group and version-within-group or
// is to refactor the genericapiserver.DelegationTarget to include a list of priorities based on which A
// This requires the APIGroupInfo struct to evolve and include the concept of priorities and to avoid m
// That ripples out every bit as far as you'd expect, so for 1.7 we'll include the list here instead of
var apiVersionPriorities = map[schema.GroupVersion]priority{
   {Group: "", Version: "v1"}: {group: 18000, version: 1},
   // to my knowledge, nothing below here collides
   {Group: "apps", Version: "v1"}:                              {group: 17800, version: 15},
   {Group: "events.k8s.io", Version: "v1"}:                     {group: 17750, version: 15},
   {Group: "events.k8s.io", Version: "v1beta1"}:                {group: 17750, version: 5},
   {Group: "authentication.k8s.io", Version: "v1"}:             {group: 17700, version: 15},
   {Group: "authorization.k8s.io", Version: "v1"}:              {group: 17600, version: 15},
   {Group: "autoscaling", Version: "v1"}:                       {group: 17500, version: 15},
   {Group: "autoscaling", Version: "v2"}:                       {group: 17500, version: 30},
   {Group: "autoscaling", Version: "v2beta1"}:                  {group: 17500, version: 9},
   {Group: "autoscaling", Version: "v2beta2"}:                  {group: 17500, version: 1},
   {Group: "batch", Version: "v1"}:                             {group: 17400, version: 15},
   {Group: "batch", Version: "v1beta1"}:                        {group: 17400, version: 9},
   {Group: "batch", Version: "v2alpha1"}:                       {group: 17400, version: 9},
   {Group: "certificates.k8s.io", Version: "v1"}:               {group: 17300, version: 15},
   {Group: "networking.k8s.io", Version: "v1"}:                 {group: 17200, version: 15},
   {Group: "networking.k8s.io", Version: "v1alpha1"}:           {group: 17200, version: 1},
   {Group: "policy", Version: "v1"}:                            {group: 17100, version: 15},
   {Group: "policy", Version: "v1beta1"}:                       {group: 17100, version: 9},
   {Group: "rbac.authorization.k8s.io", Version: "v1"}:         {group: 17000, version: 15},
```

```
26    {Group: "storage.k8s.io", Version: "v1"}:                 {group: 16800, version: 15},
27    {Group: "storage.k8s.io", Version: "v1beta1"}:            {group: 16800, version: 9},
28    {Group: "storage.k8s.io", Version: "v1alpha1"}:           {group: 16800, version: 1},
29    {Group: "apiextensions.k8s.io", Version: "v1"}:           {group: 16700, version: 15},
30    {Group: "admissionregistration.k8s.io", Version: "v1"}:   {group: 16700, version: 15},
```

Preset priority values for Kubernetes built-in groups

But what is the field `groupPriorityMinimum` and what role does it play?

The higher the value of `groupPriorityMinimum`, the higher the priority, and that group is preferred by clients over lower ones. If there are conflicting resource kind names, resource names, or shot names, the group with the highest `groupPriorityMinimum` value wins. This is how `kubectl` searches the resource and finds the matching REST API to send requests to.

The second priority, `versionPriority`, is easier to understand. It just orders all the versions that belong to this group. The version with the highest `versionPriority` value is preferred by dynamic clients.

Conclusion

The `APIService` object provides us a way to register our aggregated apiservers to the `kube-apiserver` whenever we want. We can delete our custom APIs by simply deleting the corresponding `APIService` object. It's quite handy!

Back

Introduction to Aggregated Apiserver

Next

How the Aggregated Apiserver Works