Introduction to Aggregated Apiserver

Get introduced to the aggregated apiserver in Kubernetes.

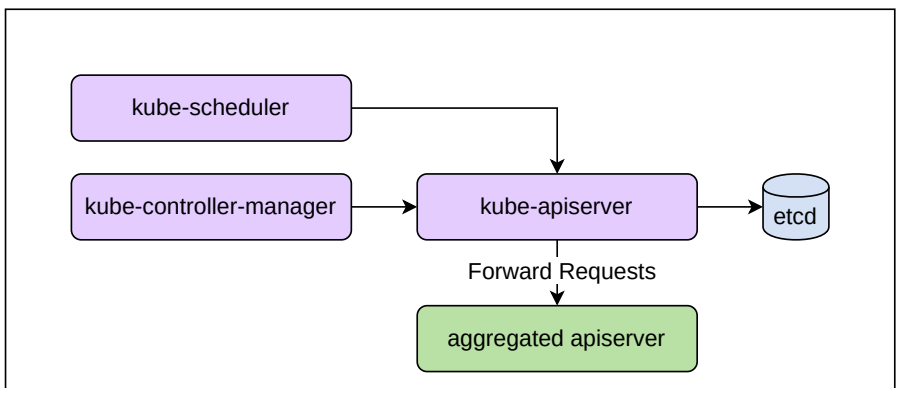
We'll cover the following

Overview

The main reason for extending Kubernetes' core APIs would be those APIs not meeting our business requirements any more. Kubernetes has introduced two primary extension mechanisms to extend Kubernetes APIs— CustomResourceDefinition (CRD) and the aggregated apiserver. They both provide RESTful APIs and work with `kubectl`. We can use `kubectl get` to get or list the resources, just like how we list Pods.

What is the aggregated apiserver?

The **aggregated apiserver** (also known as the **custom apiserver**) is a standalone HTTP server that provides REST APIs. However, it works totally differently from CRD. In the aggregated apiserver, we can develop fully featured Kubernetes APIs. In straightforward terms, the aggregated apiserver works like another `kube-apiserver`. It cascades to the `kube-apiserver` to provide additional APIs that are not offered by the core Kubernetes APIs. This is why we use the word aggregated to describe it. The aggregated apiserver sits behind the `kube-apiserver`. This kind of arrangement is also called **API Aggregation (AA)**.



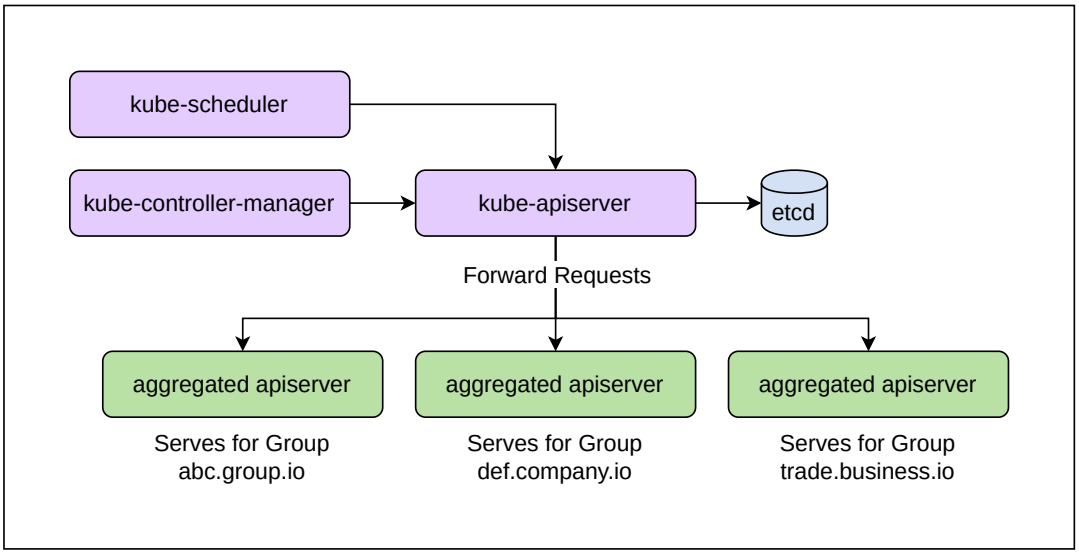
What is the aggregated apiserver?

We can dynamically register our aggregated apiservers to the `kube-apiserver` by creating an `APIService` object below. The `kube-apiserver` determines where a request should be sent and knows how to contact it. With this object, we claim the URL path `/apis/<group>/<version>/...` in the `kube-apiserver`. If we manually create an object for registration, the aggregation layer will do nothing. Normally, we run our aggregated apiservers in Pods that run in our cluster. Moreover, we expose a service for aggregated apiservers, which will be referred to below the `APIService` object. To register multiple APIs with groups and versions, we just need to create multiple corresponding `APIService` objects. When we want to unregister our APIs, we just need to delete the corresponding `APIService` object from Kubernetes. We can register again anytime we want. Below is a sample YAML to register our APIs via the `APIService` object:

```
apiVersion: apiregistration.k8s.io/v1
kind: APIService
metadata:
  name: <name of the registration object>
spec:
  group: <API group name this extension apiserver hosts>
  version: <API version this extension apiserver hosts>
  groupPriorityMinimum: <priority this APIService for this group, see API documentation>
  versionPriority: <prioritizes ordering of this version within a group, see API documentation>
  service:
    namespace: <namespace of the extension apiserver service>
    name: <name of the extension apiserver service>
  caBundle: <pem encoded ca cert that signs the server cert used by the webhook>
```

Register an aggregated apiserver

We could have multiple aggregate apiservers running inside a Kubernetes cluster. These apiservers may be developed for different API groups and maintained by different teams. We can install them separately as stand-alone components because they're running independently. If one of these aggregate apiservers crashes, the others are not affected.



Multiple aggregated apiservers

There are hardly any limitations to what we can do with an aggregated apiserver. It offers terrific flexibility and can do many things that CRDs cannot. However, this does not mean that the aggregated apiserver should always be our first choice when we want to extend Kubernetes APIs. The CRD is actually a better first choice. We should only use the aggregate apiserver when the CRD can't satisfy our business needs due to the complexity of the aggregate apiserver's development and operations. The `kube-apiserver` is the most ingenious component in Kubernetes, but it's also the most complicated one. The same goes for the aggregated apiserver, which has a steep learning curve.

When to use the aggregated apiserver

As mentioned above, the aggregated apiserver can be used in place of CRDs, and it can do everything that CRDs can and cannot do.

The limitations of using CRDs

Let's first look at the limitations of using CRDs to extend our APIs.

- CRDs use the same storage medium as the `kube-apiserver`. The `kube-apiserver` stores all the custom resources directly to `etcd`.
- CRDs only use `JSON` for API serializations. Protobuf is not supported by CRDs. So, when protobuf is a requirement for our APIs, we have to use the aggregated apiserver.
- CRDs do not support long-running subresources/endpoints, like `websocket` and `proxy`, for our own resources—they are out of the CRDs' scope. However, CRDs can handle the subresources `/status` and `/scale` well.
- CRDs do validations in a confined way. Most validations are supported in the CRD using OpenAPI v3.0. We can check the value type, value range, and enumerated values. A more complicated way is using regex for single field checking. All those validations are done in a generic way. For other validations, we need to build a validation webhook to add our custom rules, such as validating the relationships of field A and field B.

- CRDs can't integrate with other external systems. There's no way to intervene all the API requests. All the APIs that are serving are hosted solely by the `kube-apiserver`. We can't interact with our own system or tweak the requests or responses.
- CRDs only support standard CRUD semantics for all API endpoints.

What does the aggregated apiserver solve?

An aggregated apiserver does not have these restrictions; it offers nearly infinite flexibility. With aggregated apiservers, we can:

- Use any storage medium as we want. We can store data in memory for maximum performance. We can also write data to a time series database instead of a key-value store. We can also pass the data to other external systems. To better secure the data, we can also encrypt sensitive information.
- Provide all kinds of scheme support, including `JSON` and `Protobuf`. So, our custom resource clients can use Protocol Buffers for communication.
- Provide any custom subresources, such as `/exec`, `/logs`, and `/proxy`. We can also use custom protocols like `WebSocket` and `HTTP/2`.
- Implement custom operations for validation, admission, and conversion, as well as mutating and defaulting, in any language we are familiar with. For CRDs, we can implement them with a roundtrip through webhooks, but we have to add latency. We may also need to take efforts to maintain multiple webhooks at the same time.
- Implement custom semantics, such as triggering other systems to create corresponding resources.

We can see that the aggregated apiserver provides great flexibility and therefore serves as an alternative for scenarios where CRDs are still limited.

Response latency of aggregated apiservers

Using the aggregated apiserver could introduce response latency as well. Every request on the extending APIs is forwarded to the aggregated apiserver by `kube-apiserver`. The network may be high latency and the aggregated apiserver takes extra steps (such as authentication, authorization, storage processing, etc.) to handle every request. It then returns the response to the `kube-apiserver`.

Thus, if our aggregate apiserver cannot meet that latency requirement, we need to consider making changes to mitigate the problem.

Conclusion

The aggregated apiserver empowers us to extend Kubernetes APIs with the aggregation layer. It works closely with the `kube-apiserver` and brings consistent user experiences like other core Kubernetes APIs.