The Heart of Kubernetes: The kube-apiserver

Learn about the kube-apiserver.

We'll cover the following

The `kube-apiserver`

The `kube-apiserver` is the most essential and complicated component in Kubernetes.

It sits in the center of Kubernetes as the primary management point, receiving and handling all RESTful requests, such as creating/updating/deleting a `Pod`, querying a group of resources, etc. It accepts various kinds of information from requests and disseminates them to all clients. The `kubectl` client is one of the most commonly used clients for interacting with the `kube-apiserver`. We can think of the `kube-apiserver` as the front-end server for the control plane. This is why we regard the `kube-apiserver` as the "heart" of Kubernetes.

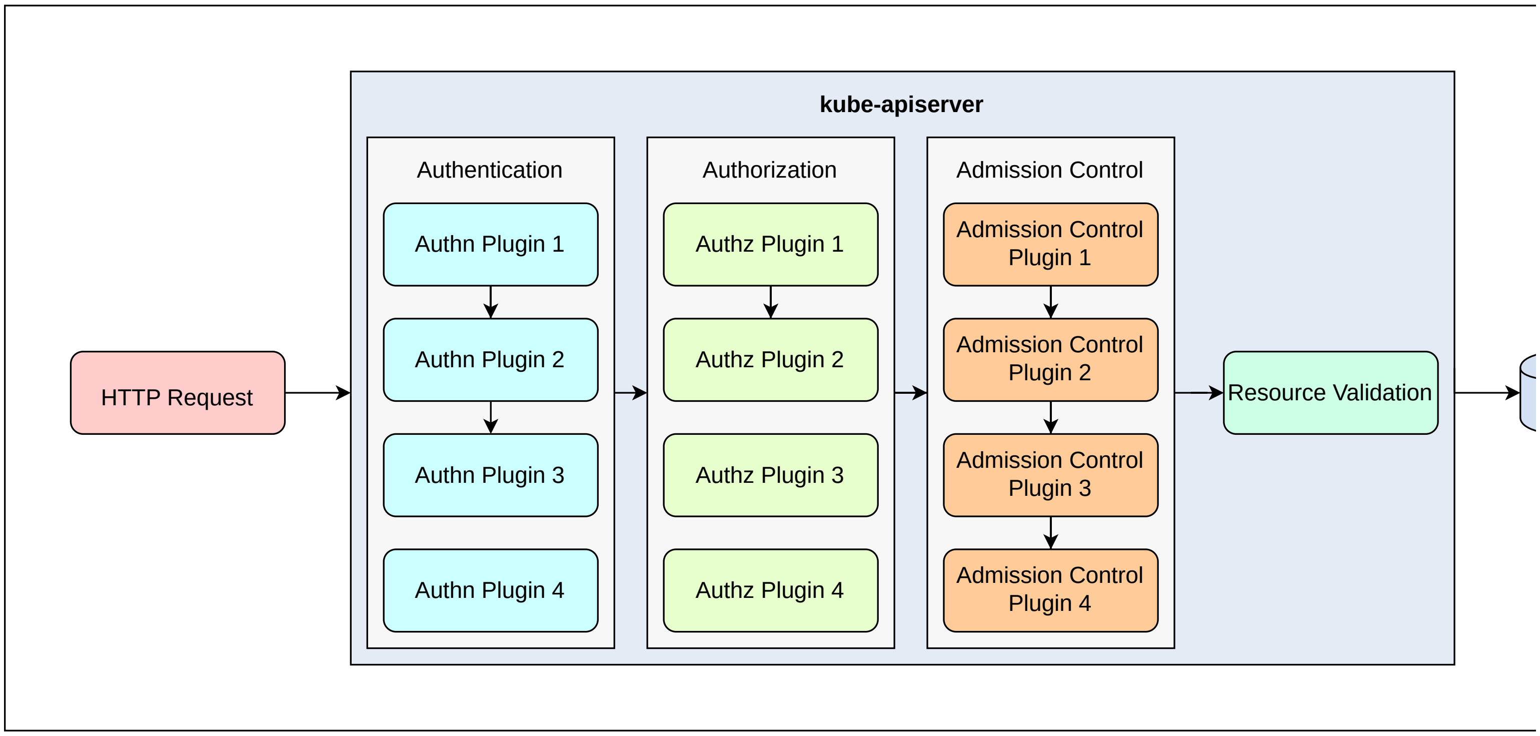
It's the only component allowed to communicate with the back-end `etcd` storage. This is an excellent way to decouple applications with data, which makes the `kube-apiserver` stateless and horizontally scalable. The `kube-apiserver` is so essential that it should be deployed with multiple replicas to provide high availability service for the entire cluster. Additionally, security must be taken into account, which is crucial for such a distributed system.

The `kube-apiserver` provides authentications and authorizations for all requests, including those from other Kubernetes components, such as `kube-scheduler`, `kube-controller-manager`, etc.

The `kube-apiserver` will also create and maintain RESTful APIs for every CRD (Custom Resource Definition).

Inside the `kube-apiserver`

Now, let's take a closer look at the `kube-apiserver`. The graph below shows what happens inside the `kube-apiserver` when receiving a request.



Inside the kube-apiserver

Stage 1: Authentication

Security is vital. First, the `kube-apiserver` needs to **authenticate** the client sending the request, because the safety of the `kube-apiserver` concerns the entire cluster. The `kube-apiserver` does allow requests from anonymous clients, but it isn't suggested in any production environments. In Kubernetes, we have various objects, like `Deployment`, `Pod`, `Service`, etc., but there is no such object called `User`. So, we can't create a `User` object via an API call. Kubernetes `User` objects are created in the following ways:

- **mTLS (mutual TLS)** for authentication. Any request that presents a valid certificate signed by the CA (certificate authority) of our cluster is considered to be valid. (In an mTLS, both clients and servers are using valid certificates signed by a common CA.) The username derives from the common name (`CN`) field in the subject of the client certificate (e.g., `/CN=Shelton`").
- `Bootstrap` **and** `ServiceAccount`: These are tokens managed by Kubernetes, which can be used for authentication as well. These two kinds of built-in tokens in Kubernetes are used widely, especially the `ServiceAccount` token. It can be associated with a `Pod` running in the Kubernetes cluster. The `ServiceAccount` token can be mounted into a `Pod` at the fixed location `/run/secrets/kubernetes.io/serviceaccount/token`, so that an in-cluster process can read and use it to talk to the `kube-apiserver`. The `ServiceAccount` token can be explicitly associated with a `Pod` by the field `podSpec.serviceAccountName`. The long tokens are recognized and interpreted to

the `User` in another form. Such a long `ServiceAccount` token will be construed with the user name `system:serviceaccount:(NAMESPACE):(SERVICEACCOUNT)`.

- **Static token**: We may also provide a static file that contains a group of usernames and tokens/passwords by the flag `--token-auth-file=SOMEFILE`. However, this is not advisable in any production environment. This is a static configuration that the `kube-apiserver` will not reload unless it restarts.
- **OpenID Connect tokens:** Kubernetes also supports using external OAuth2 providers.
- **Webhook tokens:** To better integrate with other authentication systems, Kubernetes provides webhooks for customization. It's easy to write our implementations to verify the identity. We'll dive into this in-depth and show a completed tutorial on this in a later lesson.

The `kube-apiserver` builds up a union authentication chain that connects these authentication plugins. The plugins will be called one by one until one of them confirms the identity that sends the request. If none can identify the user, an unauthorized response with a status code of `401` will be returned. The user name may come from a `ServiceAccount`, user ID, `CN` name, or group name that belongs to the user. This user name will be used in the next stage—authorization.

Stage 2: Authorization

In **authorization**, the main task is to determine user privileges. For example, Alice can delete a `Pod` in Kubernetes, while Bob is only allowed to create a `Pod`.

During the authorization stage, the `kube-apiserver` can also be configured to use one or more authorization plugins, like in the authentication stage. Identical to that, the request gets authorized if one of the plugins admits it.

> **Note:** The flag `--authorization-mode` strings specifies an ordered and comma-delimited list of `AlwaysAllow,AlwaysDeny,ABAC,Webhook,RBAC,Node`. The default value is `AlwaysAllow`. Typically, we explicitly set it to `--authorization-mode=Node,RBAC`.

- ABAC: This stands for "attribute-based access control" and is the same as the static token above. It's configured with a static file that combines users and policies.
- RBAC: This stands for "role-based access control" and is the most commonly used method for authorization in Kubernetes. Other external authorization systems can also be integrated into Kubernetes as authorization plugins.

As soon as the request passes authorization, it comes to the following stage.

Stage 3: Admission control

Only requests trying to create, modify, or delete a resource will be sent to this stage. In this stage, the `kube-apiserver` also builds a chain for a group of admission control plugins. These plugins can intercept requests or inject some default data, such as initializing missing fields in the resource specifications, overriding some values, or even rejecting a request.

The `kube-apiserver` does provide a list of enabled plugins.

> **Note:** The flag `--enable-admission-plugins` strings specifies a group of comma-delimited admission plugins that should be enabled, where the order doesn't matter.

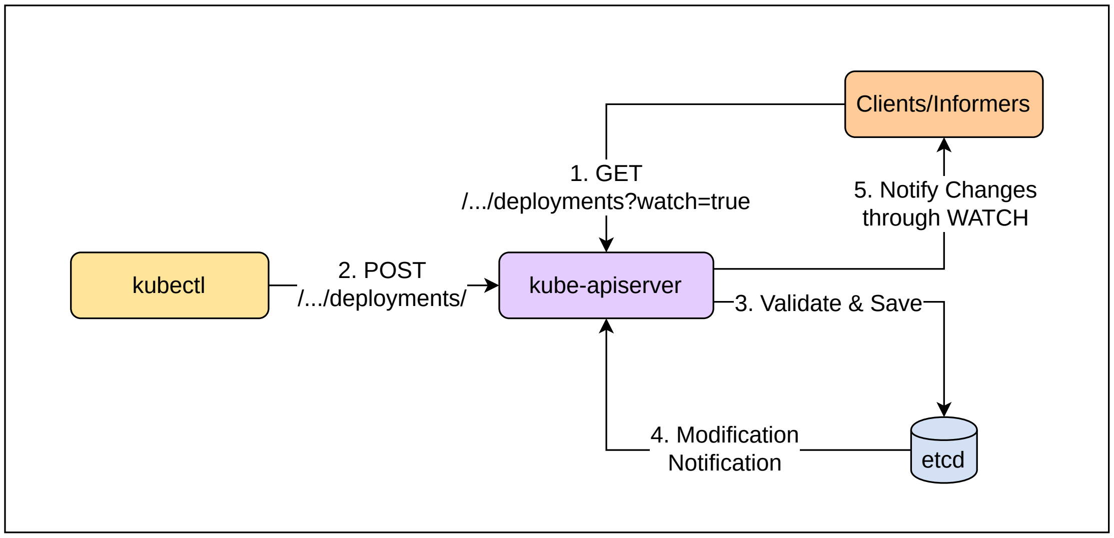
Different from the two stages above, the request must pass through all the configured admission control plugins.

Stage 4: Validating and storing it persistently

After the request passes through the above stage, the `kube-apiserver` can validate the object, such as port range, string length, and qualified domain name, and then store it in `etcd` and return a response.

How to interact with Kubernetes

The `kube-apiserver` is the heart of Kubernetes that all the other components talk to and fetch needed data from to process. The `kube-apiserver` provides a `WATCH` mechanism, which is the most charming feature design. The `kube-apiserver` doesn't tell all the other components/controllers what to do. All it does is enable those components/controllers to observe changes in their interested objects. As mentioned in the previous lesson, all the components are running in their own ways, which is one of the golden rules for designing and writing a good controller/operator.


Clients/Informers
1. GET /.../deployments?watch=true
5. Notify Changes through WATCH
kubectl
2. POST /.../deployments/
kube-apiserver
3. Validate & Save
4. Modification Notification
etcd

How to interact with the kube-apiserver

Clients watch for changes by opening an HTTP connection (with the query parameter `watch=true`) to the `kube-apiserver`. With this connection, a stream of notifications on the watched objects will be received by all connected clients that are watching these objects. Every time an object is created, updated, and deleted, the `kube-apiserver` will send out a new version of the objects to all connected clients.

It's the `WATCH` mechanism in the `kube-apiserver` that brings these loosely coupled components to work in perfect harmony. Using `WATCH`, it's more efficient to receive the changes, instead of polling/re-listing periodically. Additionally, it greatly helps reduce the body sizes of the request payloads and mitigate the burden of itself, especially when the cluster grows bigger with more objects served. This is illustrative of how to design and write a good controller/operator. This golden rule uses `WATCH` to get notified of changes instead of re-listing.

It's simple to use `kubectl` to watch resources. For example, when deploying a `Pod`, we don't need to get or list the `Pod` objects repeatedly with `kubectl get pod`. Instead, we can append the flag `-w` (or `--watch`) at the end of our command and get notified of each change (creating, updating, and deleting) made to a `Pod`.

Terminal 1

Terminal

Click to Connect...

```
kubectl get pod -n kube-system -w
```

Watch Pods in the kube-system namespace

The output will be similar to the following:

```
NAME                                                  READY   STATUS              RESTARTS   AGE
coredns-6d4b75cb6d-g9n9h                              1/1     Running             0          52s
```

```
coredns-6d4b75cb6d-zv6bt                            1/1     Running             0          52s
etcd-educative-demo-control-plane                   1/1     Running             0          65s
kindnet-8wphg                                       0/1     ContainerCreating   0          36s
kindnet-l9lw9                                       1/1     Running             0          52s
kindnet-x22kd                                       0/1     ContainerCreating   0          35s
kube-apiserver-educative-demo-control-plane         1/1     Running             0          61s
kube-controller-manager-educative-demo-control-plane 1/1     Running             0          61s
kube-proxy-mth29                                    0/1     ContainerCreating   0          35s
kube-proxy-rzzwd                                    1/1     Running             0          52s
kube-proxy-sh7qn                                    0/1     ContainerCreating   0          36s
kube-scheduler-educative-demo-control-plane         1/1     Running             0          61s
kube-proxy-sh7qn                                    1/1     Running             0          37s
kindnet-8wphg                                       1/1     Running             0          40s
kube-proxy-mth29                                    1/1     Running             0          39s
kindnet-x22kd                                       1/1     Running             0          40s
```

Watch the lifecycle of Pods in the kube-system namespace

The `client-go` library

Kubernetes also provides a native programmable library `client-go` that can be used to watch resources. We'll demonstrate this in later lessons.

Back

What is Kubernetes?

Next

The Conductor of Kubernetes: The kube-scheduler