# How the Aggregated Apiserver Works

Learn how the aggregated apiserver works in Kubernetes.

We'll cover the following



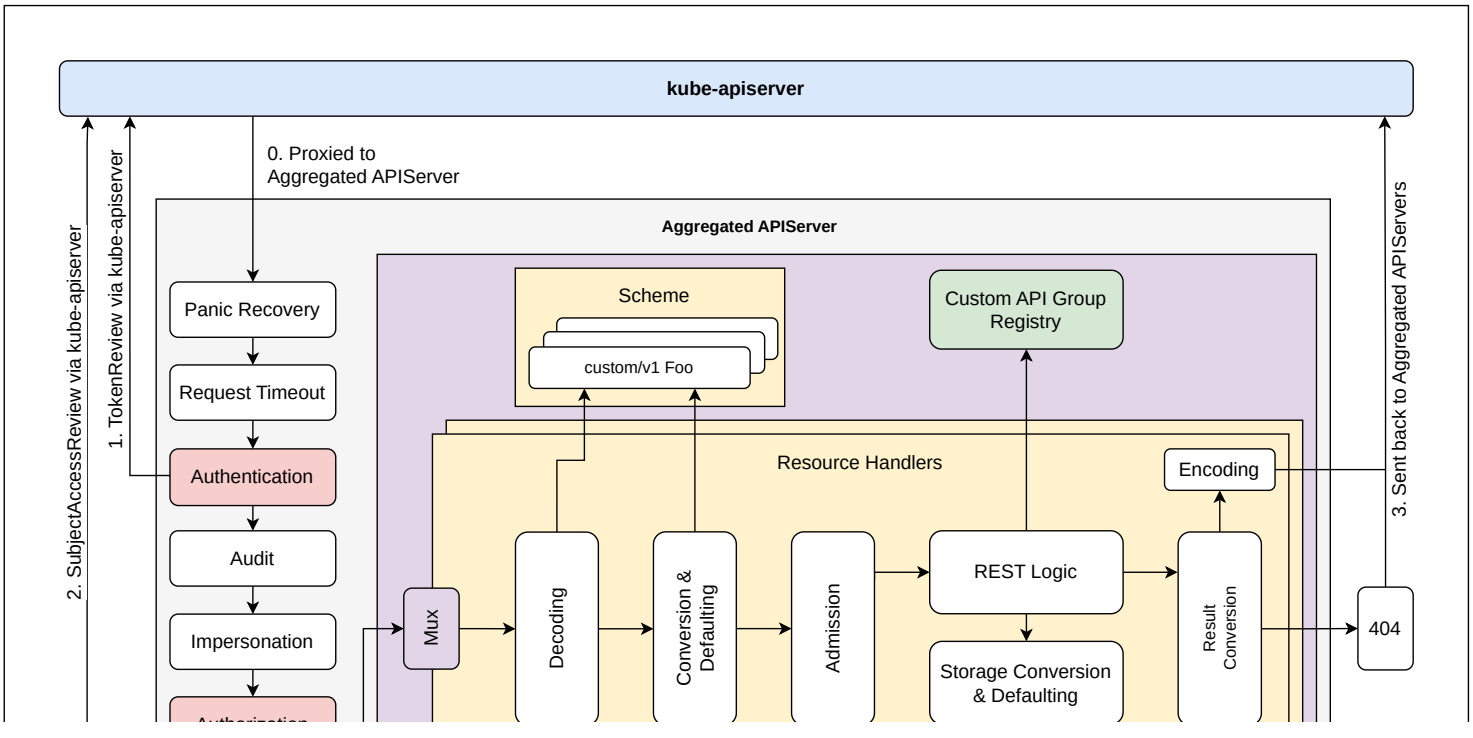
## Overview

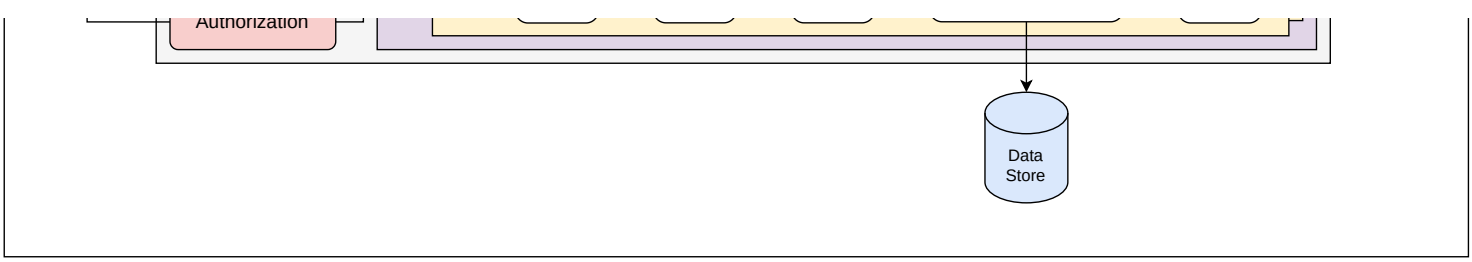
Before diving into the implementation of the aggregated apiserver, let's take a tour of how the aggregated apiserver works in a Kubernetes cluster.

We'll start with a high-level view on the aggregated apiserver. What is it made of?

## A high-level view

An aggregated apiserver works in a similar way to the `kube-apiserver`. It resembles most parts of the `kube-apiserver`. This leads to nearly the same architecture (as shown below). However, the aggregated apiserver does not have an embedded `kube-aggregator` and an embedded `apiextension-apiserver` (which serves CRDs).






The inner architecture of the aggregated apiserver

From the image above, we can see that the aggregated apiserver has the same basic internal structure as the `kube-apiserver`:

- It has its own handler chain, including authentication, auditing, impersonation, and authorization. While the authentication and authorization here is not all self implemented, we may interact with the `kube-apiserver` to validate the auth info. It runs standalone, but actually works as a subsidiary. We will go over the auth part in detail a little bit later.
- It has its own resource handlers for every custom API group. Every handler pipeline consists of object decoding, conversion, defaulting, admission, REST mapping, object storage, and encoding for the response result. This is exactly the same as the `kube-apiserver`. During the admission phase, we can call admission webhooks as well.
- When implementing an aggregated apiserver, using `etcd` as the back-end storage is not a prerequisite. There are no restrictions on what kind of storage we can use in aggregated apiservers. Admittedly, we could still write objects to `etcd`. We could even share the `etcd` cluster with the `kube-apiserver`. We can choose whatever we like as the back-end storage.
- In aggregated apiservers, we have our own scheme and registry implementations for custom API groups and versions. This is quite different from CRDs, where the `kube-apiserver` helps serve the APIs. The implementations here can be tweaked to any degree.
- Since aggregated apiservers run standalone, they do need authentication for security. Normally, the authentication methods are TLS certificates and token-based authentication. Aggregated apiservers call back to the `kube-apiserver` with a `TokenAccessReview` request to check the user validity.
- Object-level auditing is done in aggregated apiservers, while the `kube-apiserver` only audits on the meta level. For authorization, aggregated apiservers use RBAC rules as well.
- RBAC rules are also applied to APIs served by aggregated apiservers. Thus, aggregated apiservers will send `SubjectAccessReview` requests to the `kube-apiserver` for authorization decisions.

Now, let's talk about the delegated authentification parts of the architecture in detail.

## Delegated authentication and trust

Aggregated apiservers sit behind the `kube-apiserver` and receive the proxied requests from it. The `kube-apiserver` already handles the authentication part, and aggregated apiservers can reuse the

results of that authentication (i.e., the username and group). So, the `kube-apiserver` inserts this authentication information into HTTP headers when proxying requests to aggregated apiservers. We usually use the headers `X-Remote-User`, `X-Remote-Group`, and `X-Remote-Extra-`. These headers can be configured to other names with flags `--requestheader-username-headers`, `--requestheader-group-headers`, and `--requestheader-extra-headers-prefix`.

But this brings up another question—how do aggregated apiservers trust the requests with these headers? Any other callers can mock the requests and claim to have done the authentication. This problem is solved with mutual TLS certificates. They are stored in the ConfigMap `kube-system/extension-apiserver-authentication`, which is automatically created by the `kube-apiserver` upon the bootstrap's starting. Below we have the manifest:

```
apiVersion: v1
kind: ConfigMap
metadata:
  creationTimestamp: "2021-01-28T14:11:51Z"
  name: extension-apiserver-authentication
  namespace: kube-system
  resourceVersion: "42"
  uid: 44f6317e-1602-42b0-8c2e-4bca06b72a14
data:
  client-ca-file: |
    -----BEGIN CERTIFICATE-----
    MIIC5zCCAc+gAwIBAgIBADANBgkqhkiG9w0BAQsFADAVMRMwEQYDVQQDEwprdWJl
    cm5ldGVzMB4XDTIxMDEyODE0MTE0MFoXDTMxMDEyNjE0MTE0MFowFTETMBEGA1UE
    AxMKa3ViZXJuZXRlczCCASIwDQYJKoZIhvcNAQEBBQADggEPADCCAQoCggEBAL98
    lS6hdfrymRrSxPDnynW1M5tKrx+sF+8fsvdmO0VXGxxPilXmtDJYSQjYSTMkZQBs
    VtTZt25mNj1jgCGZ/lmmeXvyyKMxmYOMGbhqKZ7tb0S0yyP+FrP8FDfTwY2ubzV0
    o8nYNSM4NblzRGSkpEEhiXPtYBAwOClNKZiD62itODXdYaLd40Eya88xUb0lWcLj
    KKUywNngwRLzhLDJWXyJ+T6kjOV23LR/7eU0JUNLG8P9LiVMdNDmPRWcqfZQf6l8
    KOw7f87oM1hnBqTlwnZaqeocq7TMQzfby7uS7pHOpsgQ7mmRhg6z/ex9VxV2DJJE
    TznpUqvMYxB24Y/V5QMCAwEAAaNCMEAwDgYDVR0PAQH/BAQDAgKkMA8GA1UdEwEB
    /wQFMAMBAf8wHQYDVR0OBBYEFJOgErD2yGdBjCEEfxNc6XvaiOEbMA0GCSqGSIb3
    DQEBCwUAA4IBAQAjV664c5vYX+On3IAzcEzhAftJ+vyWuaS7pJSD+dE9TwElijO3
    544/iMOG7Ejaiyf0wGncCqiKE0VgNDYCdAS/s5/rWz7u78IAD34/VE2C+hVRsTaP
    Paa6SWketpMN7ZYbWeG7xJzctBDcs3Wxe3NdujcP5FdV/Ru5OTL0vP0MlIRcdEID
    x5T4pDhf5Q1j/hlEcMJrnTE1YcfYaSTJlEZeWm0iyoIMmzilHa7guj3+vRbr5Ba9
    MAo6OSIW13PtNE98d3olabr1csKHiDvmxBsJLbN/Lq6HnbfVBOZB+Yg6C/r5PCE5
    Xvy422CK0070CAgsi6F/LUU6yrNe4pE9Ncmm
    -----END CERTIFICATE-----
  requestheader-allowed-names: '["front-proxy-client"]'
  requestheader-client-ca-file: |
    -----BEGIN CERTIFICATE
```

The ConfigMap of extension-apiserver-authentication

With the information above, the aggregated apiservers know how to authenticate incoming requests. This helps make the communications between the `kube-apiserver` and aggregated apiservers secure. They will authenticate:

1. Clients using the client certificates signed by the CA, specified in the `client-ca-file`.
2. Clients pre-authenticated by the `kube-apiserver`, whose requests are proxied using the given CA `requestheader-client-ca-file` and inserted with auth information (username and group) by the HTTP headers `X-Remote-Group` and `X-Remote-User`.

If the requests are not using client certificates, tokens in the HTTP header `Authorization: bearer token` can be used instead to identify their validity. Aggregated apiservers will call back the `kube-apiserver` for decisions by sending a `TokenAccessReview` object containing the token.

Now, the authentication mechanism is mostly done automatically by the `k8s.io/apiserver` library; we only need to know how to provide the right parameters for configuration, but knowing what is going on behind the hood is really important.

## Delegated authorization

Every request must get authorized after being authenticated. Aggregated apiservers use RBAC for authorization by default. All the RBAC rules are enforced and validated in the `kube-apiserver`. Aggregated apiservers will delegate the authorization via `SubjectAccessReview`.

Below, we have an example:

```
apiVersion: authorization.k8s.io/v1
kind: SubjectAccessReview
spec:
  groups:
  - groupValue
  resourceAttributes:
    group: custom
    name: example
    namespace: default
    resource: foos
    verb: create
    version: v1beta1
  uid: uidValue
  user: bobby
```

An example SubjectAccessReview object

The user `bobby` wants to create an object in the group version `custom/v1beta1` named `example` in the namespace `default`. The `kube-apiserver` receives this review request, evaluates the RBAC rules, and makes a decision. The decision will be appended to this review request's status, as shown below.

```
apiVersion: authorization.k8s.io/v1
kind: SubjectAccessReview
spec:
  ...
status:
  allowed: true
  denied: false
  reason: "rule xyz allowed this request"
```

The status of the SubjectAccessReview object

For performance reasons, the delegated authorization mechanism keeps a local cache in each aggregated apiserver.

## In conclusion

In the aggregated apiserver, both authentication and authorization are mostly done automatically by the library `k8s.io/apiserver`. Understanding the underlying principles of how the aggregated apiserver works is valuable, because we can then know exactly how to implement an aggregated apiserver.

Back

Understanding the APIService object

Next

Play with the Sample API Server