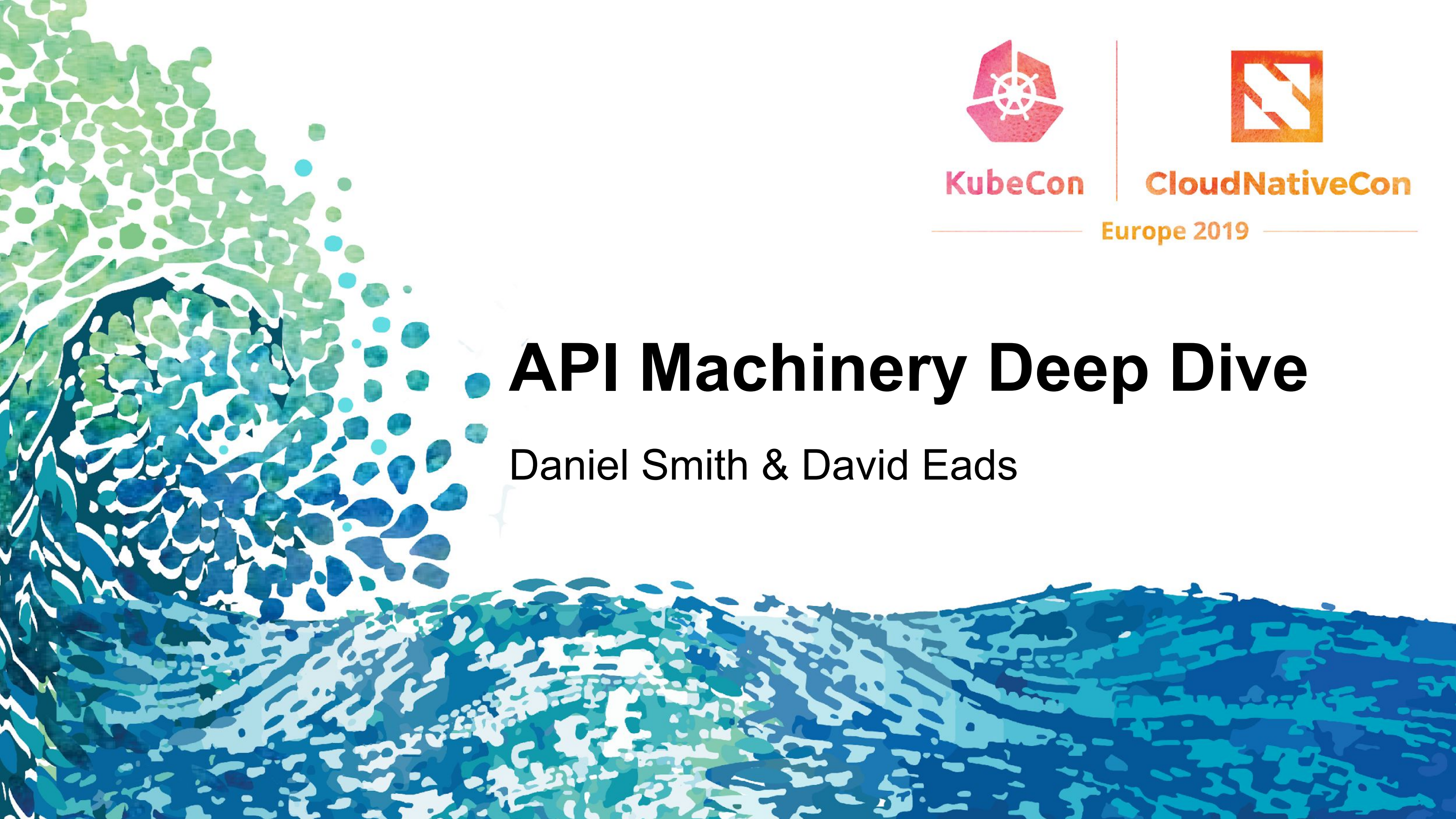


KubeCon | CloudNativeCon

Europe 2019

API Machinery Deep Dive

Daniel Smith & David Eads

Agenda

- Apply - path to beta
- Control Plane Fairness - path to alpha
- Certificate Management - explain where we are
- CRDs - path to GA (if we have time)

Apply to Beta

- Replace `kubectl apply` (built on SMP) with a new endpoint
- Apply endpoint is actually a PATCH with Content-Type: application/apply-patch+yaml
- Accepts partially specified objects (PSO)
- Knows the structure of the resource from openapi, but has its own format
- Some apply code run on every modifying endpoint

Apply - single user

```
# how apply used to work
metadata:
  annotations:
    kubectl.kubernetes.io/last-applied-configuration: “{entire object sized thing}”
...
```

- Apply needs access to some kind of historical information
- This was the easiest choice at the time

Apply - multiple users

```
# how apply might be extended to support multiple users
metadata:
  lastAppliedConfigurations:
    user1: “{entire object sized thing}”
    user2: “{entire object sized thing}”
    ...
    userN: “{entire object sized thing}”
```

- 1 user, “just” doubles the object size
- N users, (N+1) times the object size

Apply - multiple users, better

```
# what would be an acceptable size for each entry?
metadata:
  hypotheticalNewApplyField:
    user1: # some ~(object_size)/N sized thing
    user2: # some ~(object_size)/N sized thing
    ...
    userN: # some ~(object_size)/N sized thing
```

- 1 user, doubles the object size
- N users, also doubles the object size

Apply - how can we save space?

```
# last-applied-configuration
{
  “field1”: “value1”,
  “field2”: “value2”,
  “field3”: “value3”
}
```

```
# our new apply field:
{
  “field1”: “value1”,
  “field2”: “value2”,
  “field3”: “value3”
}, {
  “field1”: “value1”,
  “field2”: “value2”,
  “field3”: “value3”
}
```

Apply - how can we save space?

```
# last-applied-configuration
{
  “field1”: “value1”,
  “field2”: “value2”,
  “field3”: “value3”
}
```

```
# our new apply field:
{
  “field1”: “value1”,
  “field2”: “value2”
}, {
  “field3”: “value3”
}
```

- Each field should only show up once

Apply - how can we save space?

```
# last-applied-configuration
{
  “field1”: “value1”,
  “field2”: “value2”,
  “field3”: “value3”
}
```

```
# our new apply field:
{
  “field1”,
  “field2”
}, {
  “field3”
}
```

- Each field should only show up once
- We can also get rid of the values

Apply - field sets

- ToFieldset(Object) -> Fieldset
- Compare(Object, Object) -> Fieldset*

- Difference(Fieldset, Fieldset) -> Fieldset
- Intersection(Fieldset, Fieldset) -> Fieldset
- Union(Fieldset, Fieldset) -> Fieldset

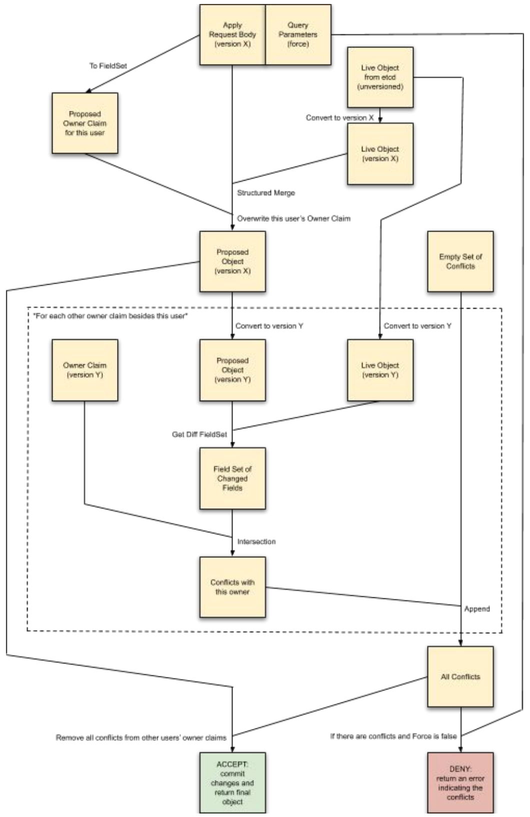
*actually 3 fieldsets: added, modified, removed

Apply - multiple versions

- Only fully specified objects can be converted
  - No PSO
  - No fieldsets

- Still doable! (but much more complicated)

```
# v1beta1
letter: A
additionalLetters: [B, C, ...]
# v1
letters: [A, B, C, ...]
```


Apply Request Body (version X)
Query Parameters (force)
To FieldSet
Live Object from etcd (unversioned)
Proposed Owner Claim for this user
Convert to version X
Live Object (version X)
Structured Merge
Overwrite this user's Owner Claim
Proposed Object (version X)
Empty Set of Conflicts
"For each other owner claim besides this user"
Convert to version Y
Convert to version Y
Owner Claim (version Y)
Proposed Object (version Y)
Live Object (version Y)
Get Diff FieldSet
Field Set of Changed Fields
Intersection
Conflicts with this owner
Append
All Conflicts
Remove all conflicts from other users' owner claims
If there are conflicts and Force is false
ACCEPT: commit changes and return final object
DENY: return an error indicating the conflicts

Flow Control

Problem: if you load apiserver too much, your cluster falls over!

Flow Control

Goals:

- Prioritization: different traffic classes get different amounts of apiserver throughput
  - system requests vs user requests
  - also: leader election requests vs event creation
- Fairness: in the same priority, requests from some actor don't starve out requests from other actors
- CPU use protection
- RAM use protection (while actively processing a request)

Flow Control

Non-goals:

- Hostile DOS prevention
- Pre-processing steps:
  - RAM from queueing requests
  - CPU/RAM for SSL handshakes
- Load balancing - every apiserver is independent

Flow Control

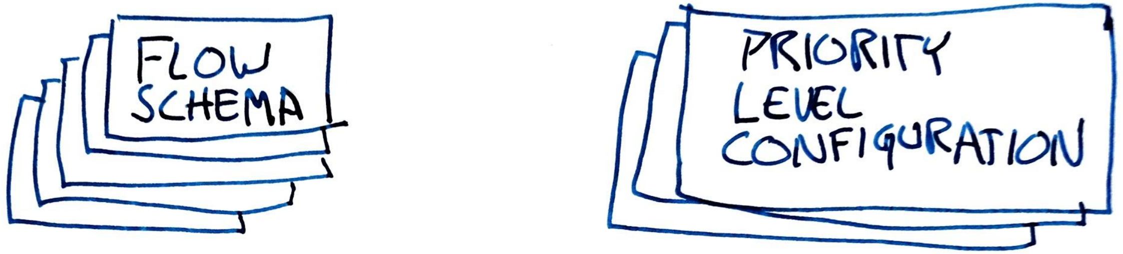
Future goals:

- Automatically tune the total system concurrency

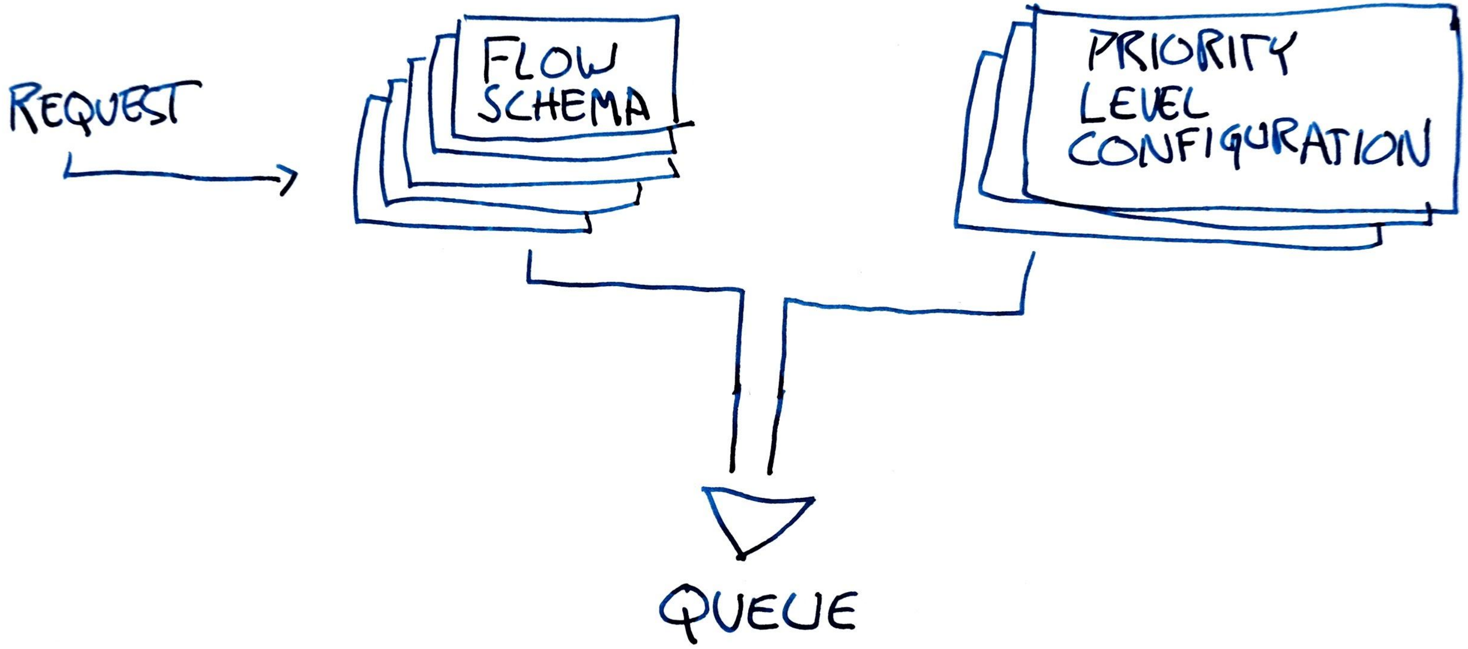
Flow Control

How does it work?

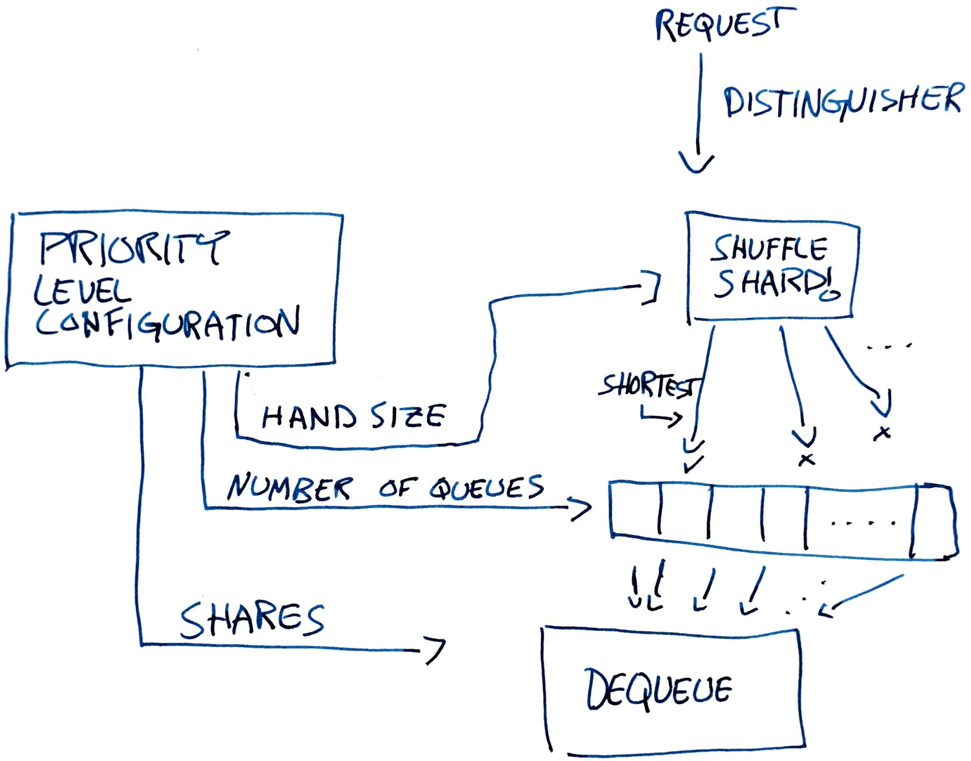
Flow Control



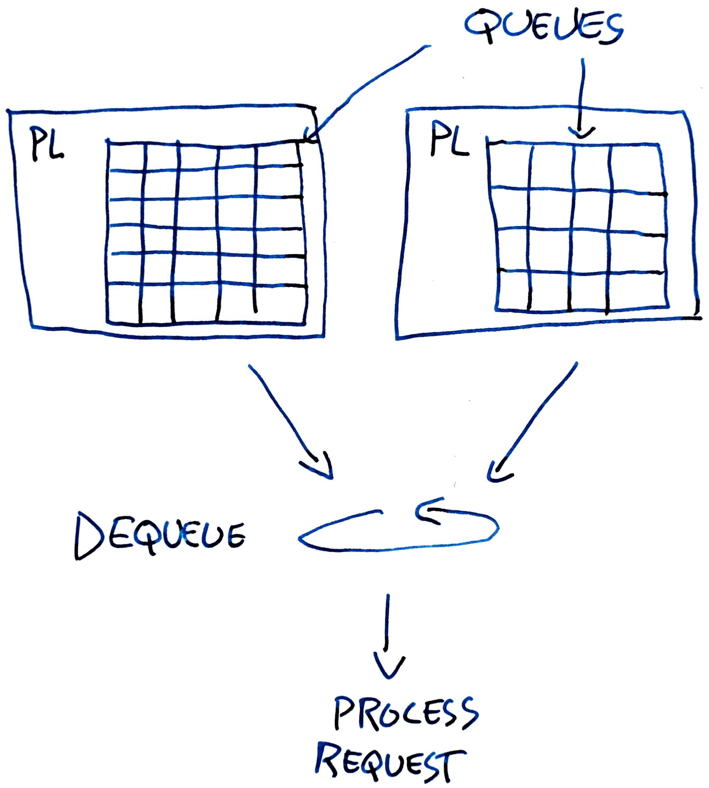
Flow Control



Flow Control



Flow Control



certificate settings today

--client-ca-file	CA bundle used to verify client certificate connections from clients and identify users. (I am Bob). Must be able to verify `kube-controller-manager --cluster-signing-cert-file` or `kubelet --rotate-certificates` will fail.
--requestheader-client-ca-file	CA bundle used to verify client certificate connections from front proxies that are asserting the identity of user. (This request is from Bob). Must be able to verify `kube-apiserver --proxy-client-cert-file` or aggregation in the cluster will fail by default.
--kubelet-certificate-authority	CA bundle used to verify kubelets for connections from KAS to kubelet. (Think logs,exec,etc). Must be able to verify `kubelet --tls-cert-file`. Must be able to verify `kube-controller-manager --cluster-signing-cert-file` or `kubelet --rotate-server-certificates` will fail.
--kubelet-client-certificate	Client cert used to identify KAS to the kubelets. Must be verifiable by `kubelet --client-ca-file`.
--kubelet-client-key	Client key used to identify KAS to the kubelets
--proxy-client-cert-file	Client cert used to identify KAS to aggregated API servers as a front proxy. Must be verifiable by `kube-apiserver --requestheader-client-ca-file` or aggregation in the cluster will fail by default
--proxy-client-key-file	Client key used to identify KAS to aggregated API servers as a front proxy
--service-account-key-file	RSA keys used to verify ServiceAccount tokens. Must be able to verify `kube-controller-manager --service-account-private-key-file` for all keys you want to continue working.
kube-controller-manager	**What's it for**
--client-ca-file	CA bundle used to verify client certificate connections from clients and identify users. (I am Bob)
--tls-cert-file	Serving cert used to serve requests
--tls-private-key-file	Serving key used to serve requests

certificate settings today

--tls-private-key-file	Serving key used to serve requests not matching SNI
--tls-sni-cert-key	Special flag format to specify hostname-pattern,cert,key tuples to serve matching SNI requests. If used for kubernetes.default.service, must be verifiable with `kube-controller-manager --root-ca-file`.
--client-ca-file	CA bundle used to verify client certificate connections from clients and identify users. (I am Bob). Must be able to verify `kube-controller-manager --cluster-signing-cert-file` or `kubelet --rotate-certificates` will fail.
--requestheader-client-ca-file	CA bundle used to verify client certificate connections from front proxies that are asserting the identity of user. (This request is from Bob). Must be able to verify `kube-apiserver --proxy-client-cert-file` or aggregation in the cluster will fail by default.
--kubelet-certificate-authority	CA bundle used to verify kubelets for connections from KAS to kubelet. (Think logs,exec,etc). Must be able to verify `kubelet --tls-cert-file`. Must be able to verify `kube-controller-manager --cluster-signing-cert-file` or `kubelet --rotate-server-certificates` will fail.
--kubelet-client-certificate	Client cert used to identify KAS to the kubelets. Must be verifiable by `kubelet --client-ca-file`.
--kubelet-client-key	Client key used to identify KAS to the kubelets
--proxy-client-cert-file	Client cert used to identify KAS to aggregated API servers as a front proxy. Must be verifiable by `kube-apiserver --requestheader-client-ca-file` or aggregation in the cluster will fail by default
--proxy-client-key-file	Client key used to identify KAS to aggregated API servers as a front proxy
--service-account-key-file	RSA keys used to verify ServiceAccount tokens. Must be able to verify `kube-controller-manager --service-account-private-key-file` for all keys you want to continue working.

kube-controller-manager	**What's it for**
--client-ca-file	CA bundle used to verify client certificate connections from clients and identify users. (I am Bob)
--tls-cert-file	Serving cert used to serve requests
--tls-private-key-file	Serving key used to serve requests
--cluster-signing-cert-file	Signing cert used to issue approved CSR requests. Must be verifiable with `kube-apiserver --kubelet-client-certificate` and `kube-apiserver --client-ca-file` or `kubelet --rotate-certificates` will fail.
--cluster-signing-key-file	Signing key used to issue approved CSR requests
--requestheader-client-ca-file	CA bundle used to verify client certificate connections from front proxies that are asserting the identity of user. (This request is from Bob)
--root-ca-file	CA bundle injected into ServiceAccount token secrets. It is **only** intended to be used to verify a connection to the kube-apiserver on the service network. All other uses are either wrong or coincidence. Must be able to verify `kube-apiserver --tls-cert-file`
--service-account-private-key-file	RSA key used to sign ServiceAccount tokens. Must be verifiable by `kube-apiserver --service-account-key-file` or ServiceAccounts will not be able to

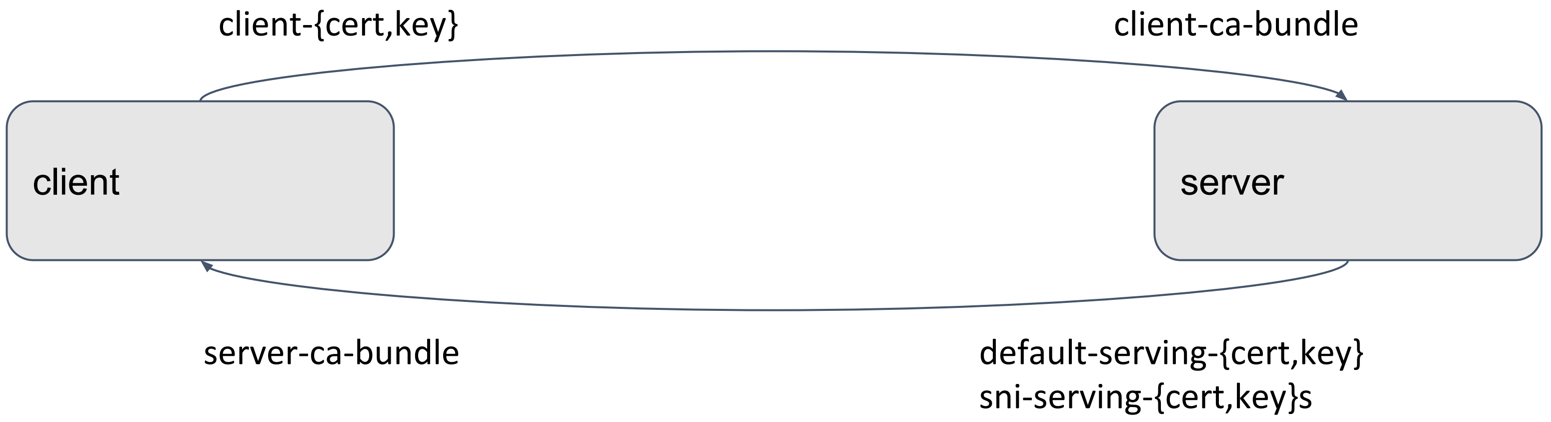
certificate settings today

kube-apiserver	**What's it for**
–etcd-cafile	CA bundle used to verify the etcd server really is the etcd server
–etcd-certfile	Client cert used to identify KAS to the etcd server
–etcd-keyfile	Client key used to identify KAS to the etcd server
–tls-cert-file	Serving cert used to serve requests not matching SNI. Must be verifiable with `kube-controller-manager --root-ca-file` or service must be handled with SNI.
–tls-private-key-file	Serving key used to serve requests not matching SNI
–tls-sni-cert-key	Special flag format to specify hostname-pattern,cert,key tuples to serve matching SNI requests. If used for kubernetes.default.service, must be verifiable with `kube-controller-manager --root-ca-file`..
–client-ca-file	CA bundle used to verify client certificate connections from clients and identify users. (I am Bob). Must be able to verify `kube-controller-manager --cluster-signing-cert-file` or `kubelet --rotate-certificates` will fail.
–requestheader-client-ca-file	CA bundle used to verify client certificate connections from front proxies that are asserting the identity of user. (This request is from Bob). Must be able to verify `kube-apiserver --proxy-client-cert-file` or aggregation in the cluster will fail by default.
–kubelet-certificate-authority	CA bundle used to verify kubelets for connections from KAS to kubelet. (Think logs,exec,etc). Must be able to verify `kubelet --tls-cert-file`. Must be able to verify `kube-controller-manager --cluster-signing-cert-file` or `kubelet --rotate-server-certificates` will fail.
–kubelet-client-certificate	Client cert used to identify KAS to the kubelets. Must be verifiable by `kubelet --client-ca-file`.
–kubelet-client-key	Client key used to identify KAS to the kubelets
–proxy-client-cert-file	Client cert used to identify KAS to aggregated API servers as a front proxy. Must be verifiable by `kube-apiserver --requestheader-client-ca-file` or aggregation in the cluster will fail by default
–proxy-client-key-file	Client key used to identify KAS to aggregated API servers as a front proxy
–service-account-key-file	RSA keys used to verify ServiceAccount tokens. Must be able to verify `kube-controller-manager --service-account-private-key-file` for all keys you want to continue working.

kube-controller-manager	**What's it for**
–client-ca-file	CA bundle used to verify client certificate connections from clients and identify users. (I am Bob)
–tls-cert-file	Serving cert used to serve requests
–tls-private-key-file	Serving key used to serve requests
–cluster-signing-cert-file	Signing cert used to issue approved CSR requests. Must be verifiable with `kube-apiserver --kubelet-client-certificate` and `kube-apiserver --client-ca-file` or `kubelet --rotate-certificates` will fail.
–cluster-signing-key-file	Signing key used to issue approved CSR requests
–requestheader-client-ca-file	CA bundle used to verify client certificate connections from front proxies that are asserting the identity of user. (This request is from Bob)
–root-ca-file	CA bundle injected into ServiceAccount token secrets. It is **only** intended to be used to verify a connection to the kube-apiserver on the service network. All other uses are either wrong or coincidence. Must be able to verify `kube-apiserver --tls-cert-file`
–service-account-private-key-file	RSA key used to sign ServiceAccount tokens. Must be verifiable by `kube-apiserver --service-account-key-file` or ServiceAccounts will not be able to authenticate

kubelet	**What's it for**
–client-ca-file	CA bundle used to verify client certificate connections from clients and identify users. (I am Bob). Must be able to verify `kube-apiserver --kubelet-client-certificate` or some endpoints will fail.
–tls-cert-file	Serving cert used to serve requests. Must be verifiable by `kube-apiserver--kubelet-certificate-authority` or some endpoints will fail.
–tls-private-key-file	Serving key used to serve requests
–rotate-certificates	If true, get client cert/key pairs for authentication to kube-apiserver from the CSR API. This interacts with `kube-apiserver --client-ca-file` and `kube-controller-manager --cluster-signing-cert-file`.
–rotate-server-certificates	If true, get serving cert/key pairs (--tls-cert-file/--tls-private-key-file) from the CSR API. This interacts with `kube-apiserver--kubelet-certificate-authority` and `kube-controller-manager --cluster-signing-cert-file`.

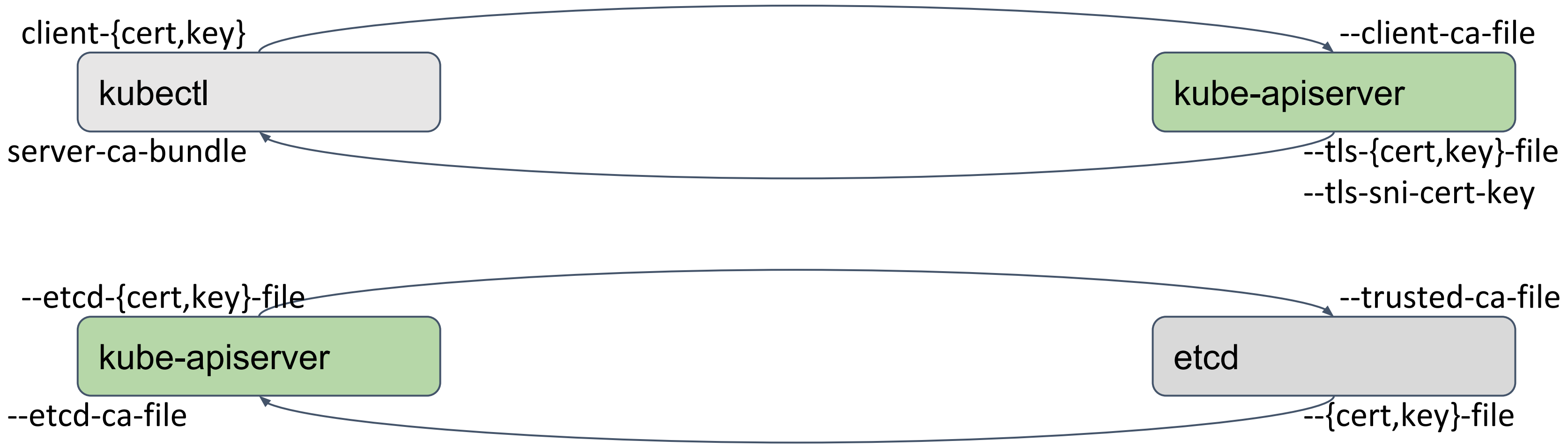
Certs, super basic mTLS



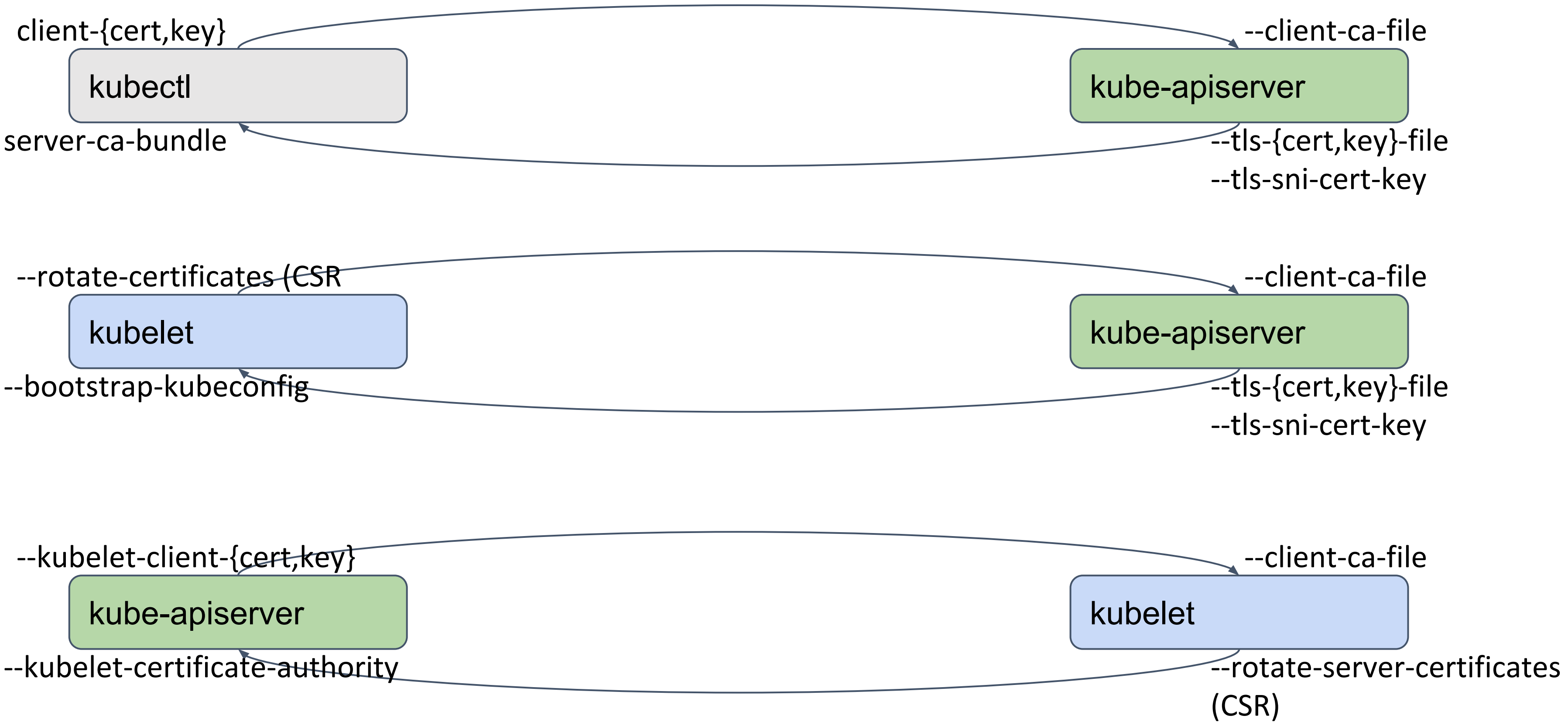
Reminders

- Certificates are signed by issuers
- CA bundles contain every valid issuer and possibly its chain
- Rotate by expanding trust first

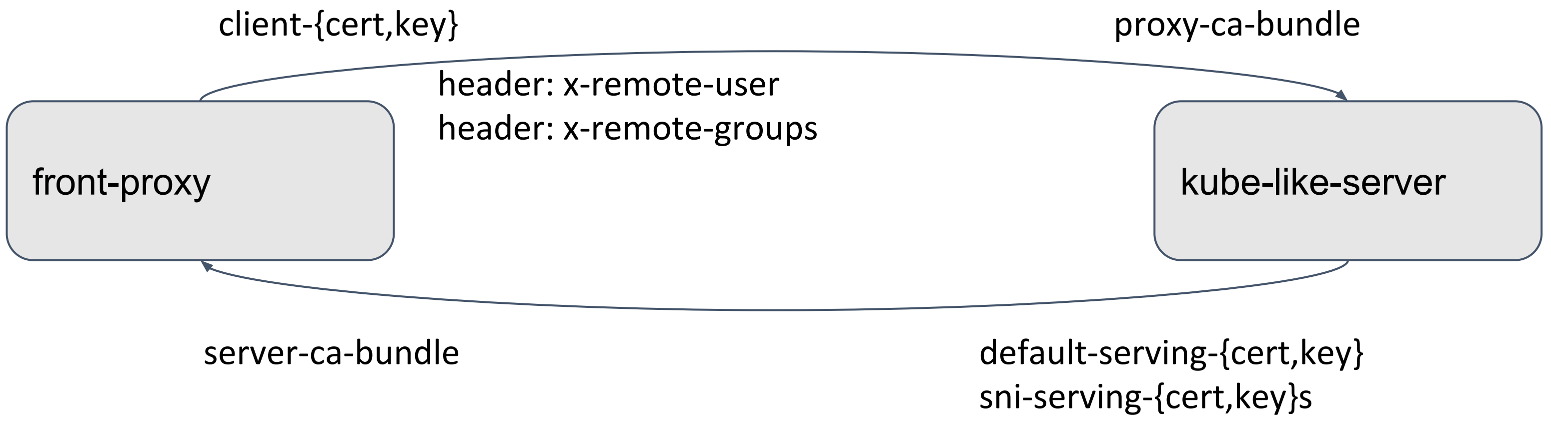
Certs, simple chains



Certs, simple chains



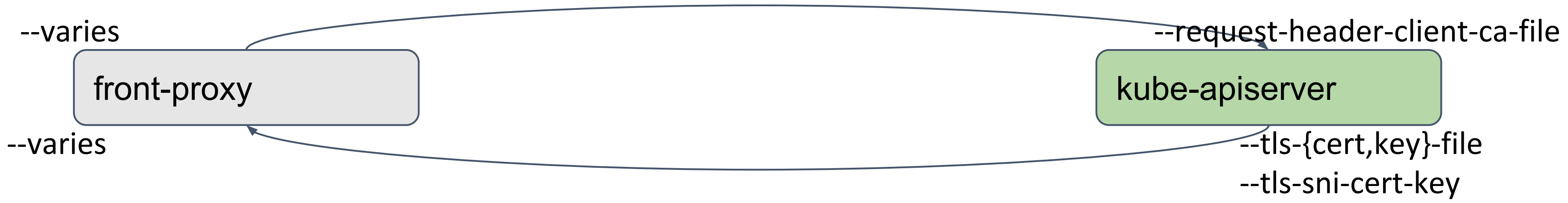
Certs, front proxy - kube-specific-ish

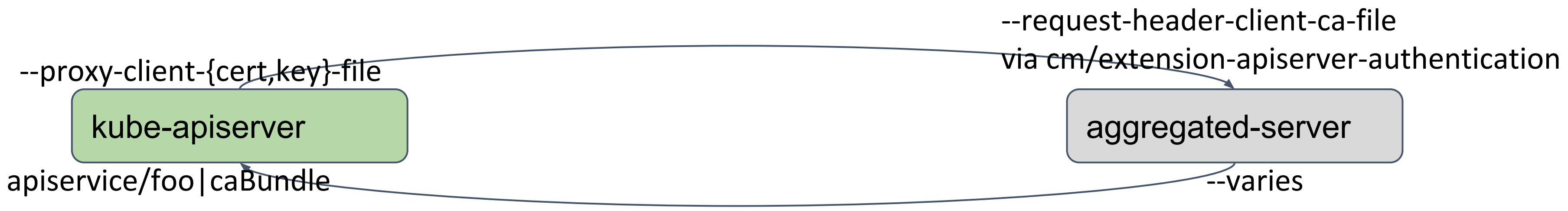


Details

- front-proxy decides the identity of the client
- proxy-ca-bundle verifies the identity of the front-proxy
- headers assert identity of the user

Certs, front-proxies and aggregator




--proxy-client-{cert,key}-file
kube-apiserver
apiservice/foo|caBundle
--request-header-client-ca-file
via cm/extension-apiserver-authentication
aggregated-server
--varies

Certs, CSR and kubelets

- Relationships are hard
  - kubelet --rotate-certificates --rotate-server-certificates
  - kube-controller-manager --cluster-signing-cert-file --cluster-signing-key-file
  - kube-apiserver --kubelet-certificate-authority --client-ca-file
- All must agree

CRDs - path to GA

- OpenAPI publishing
- Pruning
- Defaulting
- Arbitrary subresources?
- Discovery priority?

CRDs - Pruning, what “real” resources do

kubectl create -f …

```
kind: Pod
metadata:
  name: cool-pod
non-existing-field: value!
```

kubectl get pod/cool-pod…

```
kind: Pod
metadata:
  name: cool-pod
```

CRDs - Pruning, the downside

kubectl create -f ...

```
kind: Pod
metadata:
  name: cool-pod
nifty-new-feature: value!
```

Your pod got created, but it doesn't have your new feature.

This is the frustrating case.

kubectl client-side validation
to the rescue

CRDs - Pruning, security and consistency

kubectl create -f ...

```
kind: Pod
metadata:
  name: cool-pod
spec:
  superPrivileged: value!
```

Update node restarts with `superPrivileged` support.

Security issue because superPrivileged was never checked security-wise.

Know what can be in etcd.

CRDs - Defaulting, simple scenario

kubectl create -f ...

```
kind: Pod
metadata:
  name: cool-pod
```

Meanwhile in etcd...

```
kind: Pod
metadata:
  name: cool-pod
neat-defaulted-field: "defaulted"
```

CRDs - Defaulting, simple scenario

Inside etcd...

```
kind: Pod
metadata:
  name: cool-pod
neat-defaulted-field: "defaulted"
```

kubectl get pod/cool-pod...

```
kind: Pod
metadata:
  name: cool-pod
neat-defaulted-field: "defaulted"
```

CRDs - Defaulting, versioned scenario

kubectl apply -f crd.yaml

```
kind: CRD
validation:
 openAPIV3Schema:
   properties:
     neat-defaulted-field:
       default: "defaulted"
```

Inside etcd...

```
kind: Pod
metadata:
  name: cool-pod
```

kubectl get pod/cool-pod...

```
kind: Pod
metadata:
  name: cool-pod
neat-defaulted-field: "defaulted"
```

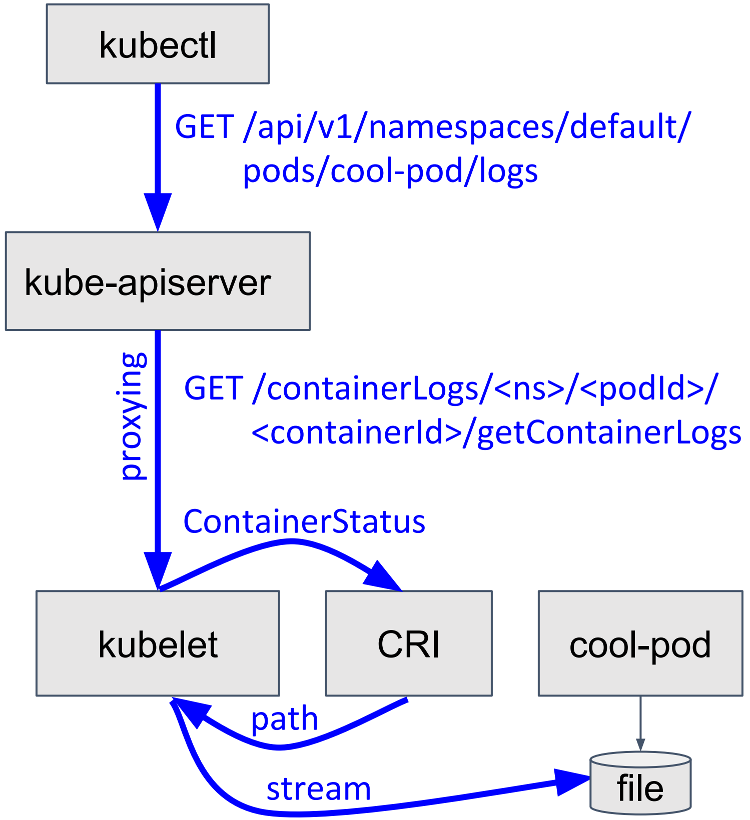
CRDs - Arbitrary subresources ?

kubectl logs pod/cool-pod

kubectl logs vm/cool-vm

kubectl exec vm/cool-vm

kubectl port-forward vm/cool-vm


kubectl
GET /api/v1/namespaces/default/ pods/cool-pod/logs
kube-apiserver
proxying
GET /containerLogs/<ns>/<podId>/ <containerId>/getContainerLogs
ContainerStatus
kubelet
CRI
cool-pod
path
stream
file

CRDs - Priority ?

```
kind: CustomResourceDefinition
metadata:
  name: foo.one.com
```

```
kind: CustomResourceDefinition
metadata:
  name: foo.two.com
```

kubectl get foo

Which foo do you want?