



Kubernetes API Server

Kubernetes Meetup
Irvi Aini (@irvifa)

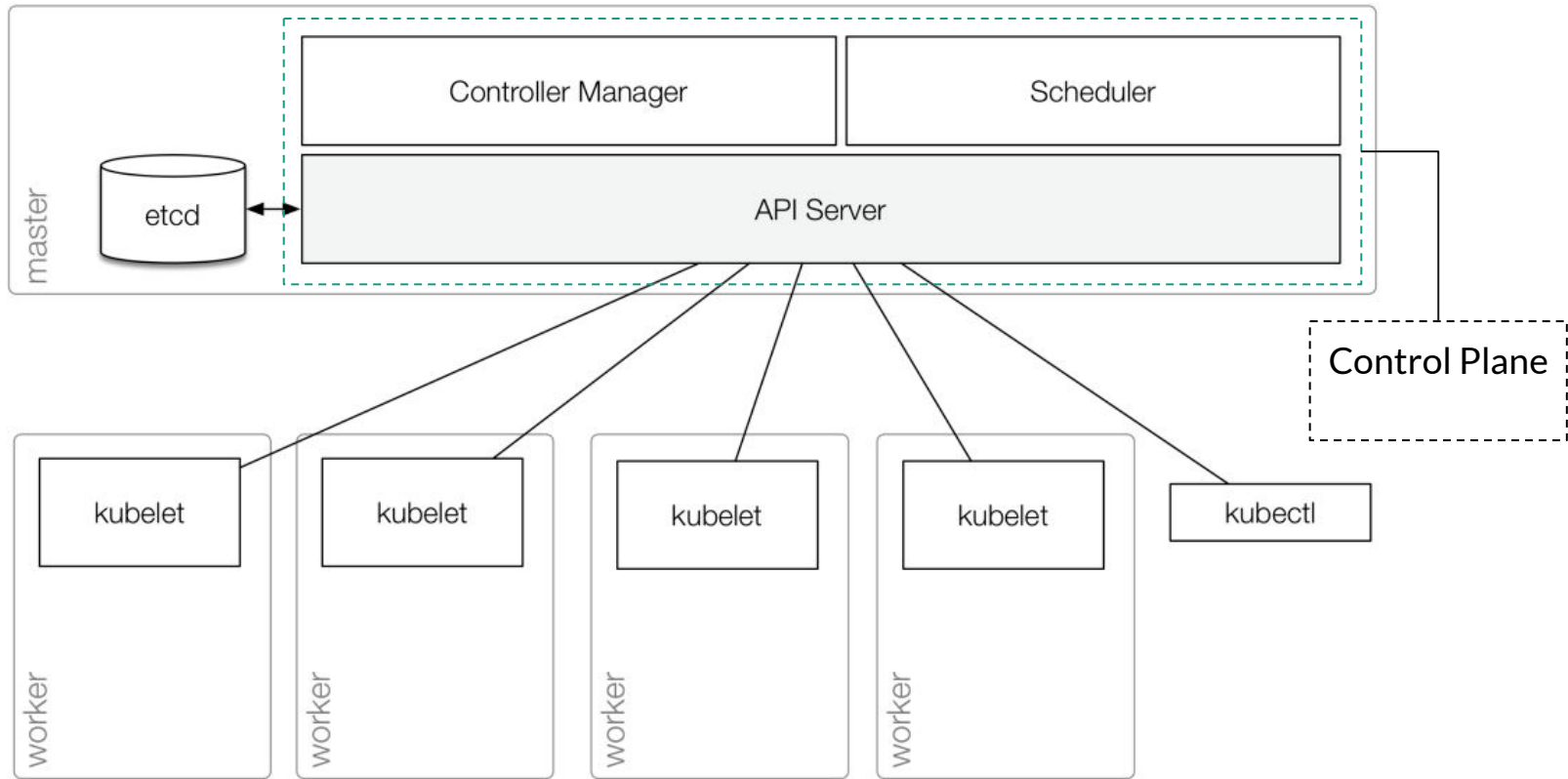


\$(whoami)

Software Engineer - Data @ Traveloka

~1 year-ish

Previously: Software Engineer Intern in Cermati and Bukalapak



Kubernetes Conceptual Overview

Introducing API Server

Central management entity and the only component that have direct access to etcd, implemented as RESTful API over HTTP, through which all other components interact...



Pieces of API Server

- API management
 - The process by which APIs are exposed and managed by the server
- Request processing
 - Set of functionality that processes individual API requests from a client
- Internal control loops
 - Responsible for background operations necessary to the successful operation of the API server

How client can make an API request?

Since it's a HTTP request, how the client and server communicate?



Terminology

- **Kind**, the type of an entity, there are three type of **Kind**:
 - Object, represent a persistent entity in the system. An object may have multiple resources that clients can use to perform specific actions. Examples: Pod and Namespace.
 - Lists, are collections of resources of one or more kinds of entities. It have limited amount of metadata, ie: PodList, NodeList.
 - Special purpose kinds, used to perform specific actions on objects or non persistent entities., such as /status

```
apiVersion: v1
kind: Pod
metadata:
  name: webserver
spec:
  containers:
    - name: nginx
      image: nginx:1.9
      ports:
        - containerPort: 80
```



Terminology (Con't)

- **API Group**, a collection of Kinds that are logically related
- **Version**, Each API Group can exist in multiple versions. The API server does lossless conversion to return objects in the requested version.
- **Resource** is the representation of a system entity sent or retrieved as JSON via HTTP; ie:
 - exposed as an individual resource (such as .../namespaces/default)
 - collections of resources (like .../jobs)



API Paths

All kubernetes requests begin with prefix `/api/` (**core API**) or `/apis/` (**APIs grouped by API group**).

Why differentiate it into two prefix? 丿 ('~`;) ㄱ

At the first time, API group doesn't exist, so **original core object** such as **Service** and **Pod** are maintained inside `/api/`. Meanwhile newer object generally added within **API group**, such as **Job** object maintained under `/apis/batch/v1`



Multiple API Version Support

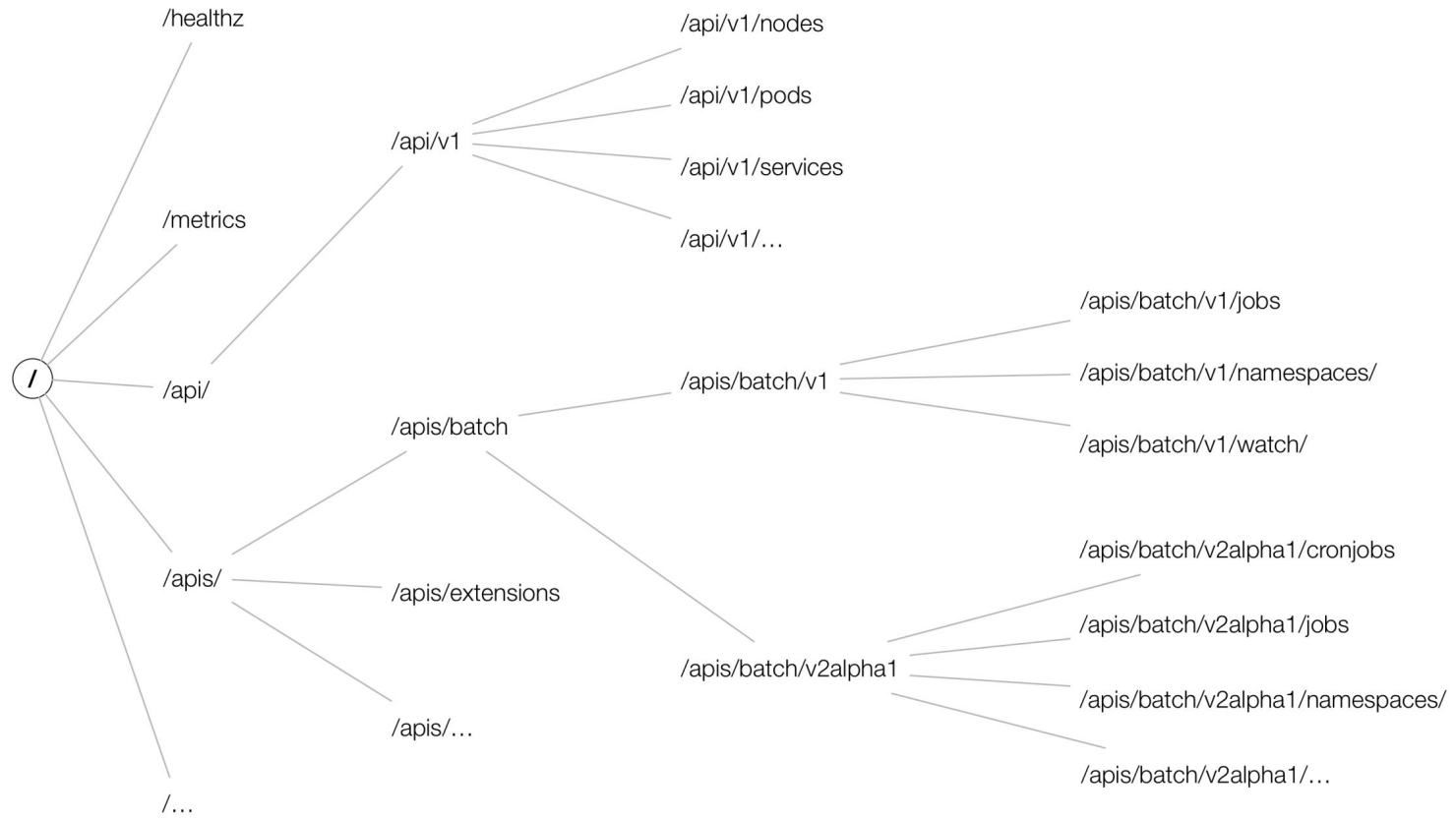
- Different API versions, implied different level of stability and support, constituted of:
 - Stable (GA): ie, **v1**, will appear in subsequent versions
 - Beta: ie, **v1beta1** is enabled by default and is already well tested, but semantics of object may change in incompatible ways in subsequent beta or stable release
 - Alpha: ie: **v1alpha1** is disabled by default, support for a feature that may be dropped in subsequent versions



Multiple API Version Support (Con't)

Because of the aforementioned reasons, API server has three different representations of API of all times:

- External representation, representation which come in via API request
- Internal representation, in memory representations of objects used within API server for processing
- Storage representation, recorded into the storage layer to persist the API objects



Part of HTTP API Space



Namespaced and Non-namespaced Object

Namespace added layer of grouping objects, namespaced object only can be created inside a namespace, and the name of the namespace will be included in the HTTP path

Namespaced resource path:

- `/api/v1/namespaces/<namespace-name>/<resource-type-name>/<resource-name>`
- `/apis/<api-group>/<api-version>/namespaces/<namespace-name>/<resource-type-name>/<resource-name>`

Non namespaced resource path, obvious example (Namespace itself and Node):

- `/api/v1/<resource-type-name>/<resource-name>`
- `/apis/<api-group>/<api-version>/<resource-type-name>/<resource-name>`




Note that since Job is non cluster wide object, it's located within a namespace as contrasted with Node



API Discovery

- Use the kubectl proxy command.
 - Provide authentication that necessary to access our cluster
 - Create a simple proxy running on port 8001 on our local machine.
- API server provide a way to know the HTTP path and payload that need to be sent using OpenAPI specifications at the following path:
 - /swaggerapi Before kubernetes 1.10, server Swagger 1.2
 - /openapi/v2 1.10 and beyond, serves OpenAPI (Swagger 2.0)



Swagger/OpenAPI Specifications from Kubernetes Object

```
{
  "name": "pods",
  "singularName": "",
  "namespaced": true,
  "kind": "Pod",
  "verbs": [
    "create",
    "delete",
    "deletecollection",
    "get",
    "list",
    "patch",
    "proxy",
    "update",
    "watch"
  ],
  "shortNames": [
    "po"
  ],
  "categories": [
    "all"
  ]
},
{
  "name": "pods/attach",
  "singularName": "",
  "namespaced": true,
  "kind": "Pod",
  "verbs": []
}
```




How We can Do That?

Code within it that knows how to perform the various translations between all of these representations.

An API object may be submitted as a v1alpha1 version, stored as a v1 object, and subsequently retrieved as a v1beta1 object or any other arbitrary supported version. Explained better in sig-architecture (•_•)

These transformations are achieved with reasonable performance using machine-generated *deep-copy* libraries, which perform the appropriate translations.

Request Management



Types of Requests

GET */api/v1/namespaces/default/pods/foo*, get information of object

LIST */api/v1/namespaces/default/pods*, list collection of object

POST */api/v1/namespaces/default/pods*, create a new object

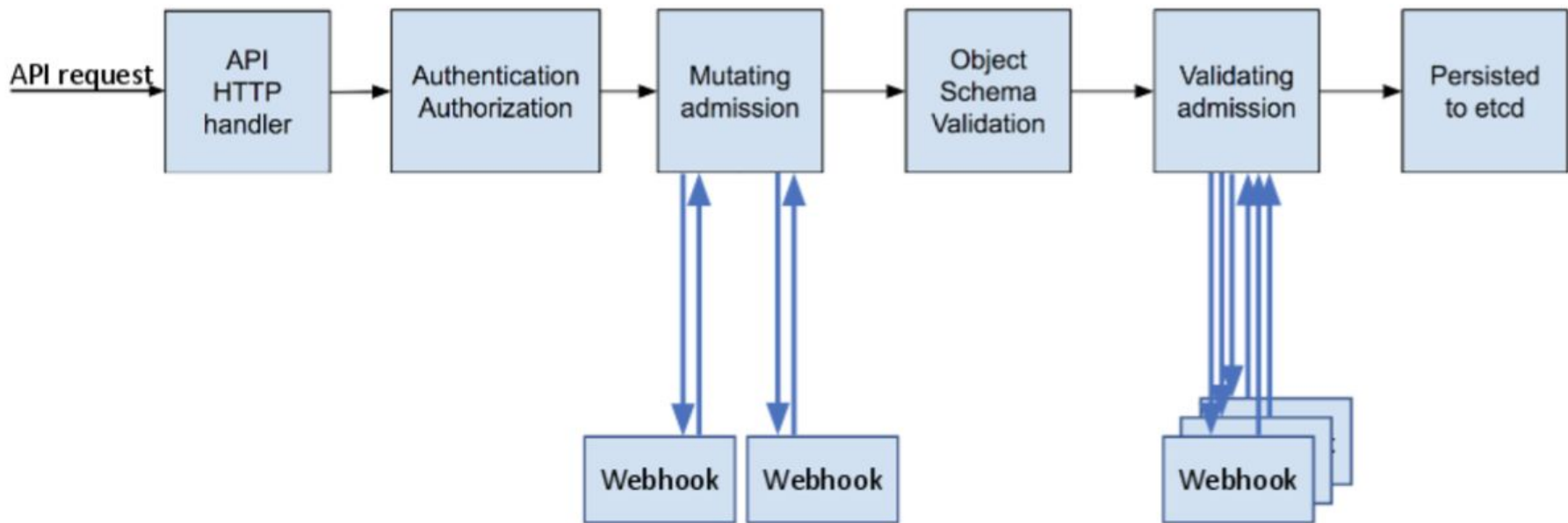
PUT */api/v1/namespaces/default/pods/foo*, update existing object

DELETE */api/v1/namespaces/default/pods/foo*, delete existing object



Life of a Request

- Authentication, established identity associated with the request. Built in authentication:
 - Bearer token
 - Client certificates
 - HTTP basic auth -> discouraged
- RBAC/Authorization
 - Identity must have appropriate role associated with the request.
 - Success -> appropriate roles
 - 403 -> otherwise
- Admission control
 - This will determine if the request is well formed and potentially needed modification before it's being proceed. If there's an error the request is rejected. This part is pluggable, ie: Istio sidecar



Admission Controller



Life of a Request (Con't)

- Validation
 - Only performed on single object, if it requires broader knowledge of the cluster state, it need to be implemented as a admission controller.
- Specialized request, consisted of /proxy, /logs, /attach, and /exec
 - /proxy : Open a long time running connection to the API server, these request provide streaming data instead of immediate response
 - /logs : Make a request to get the logs for a Pod by appending /logs to the end of the path for a particular Pod (e.g., /api/v1/namespaces/default/pods/some-pod/logs) and then specifying the container name as an HTTP query parameter and an HTTP GET request. Given a default request, the API server returns all of the logs up to the current time, as plain text, and then closes the HTTP request, unless a --follow query is specified.
 - /attach and /exec will use WebSocket to provide bidirectional data streaming proccess.



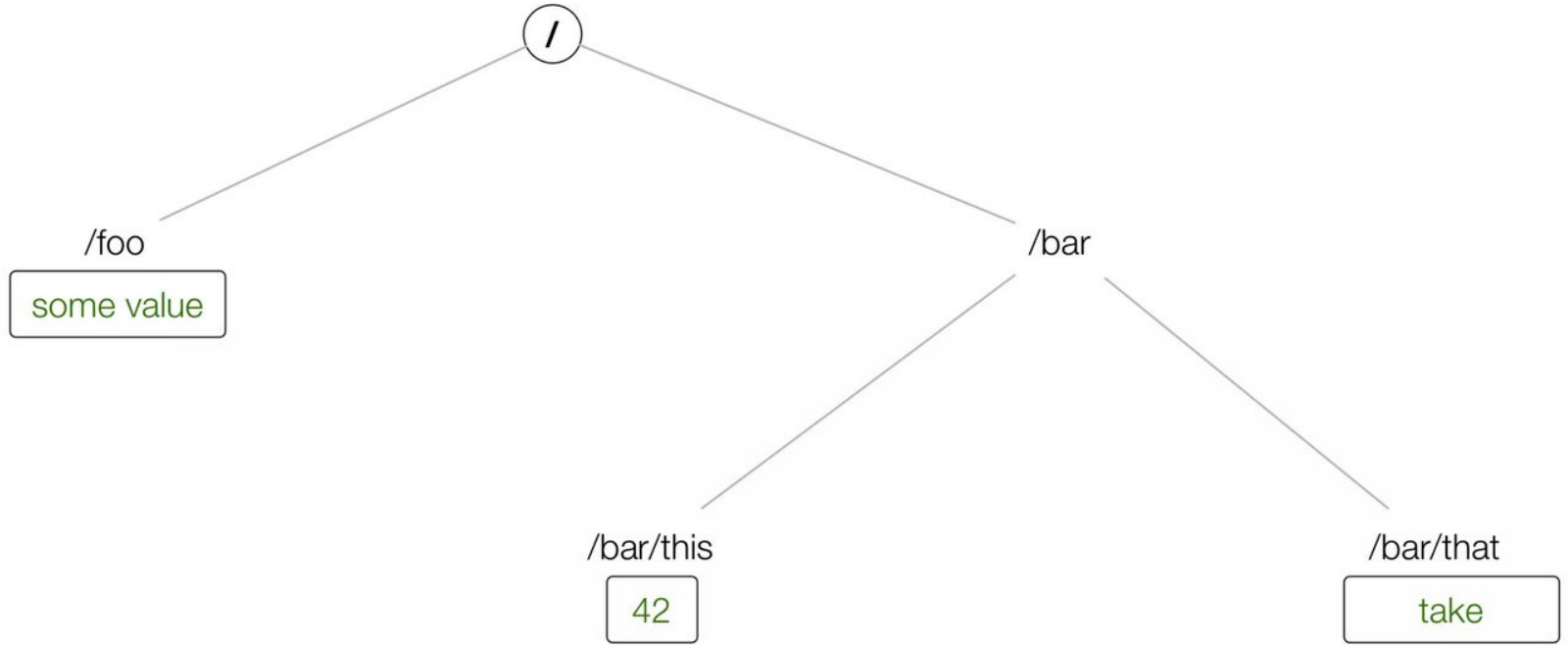
Life of a Request (Con't)

- Watch, monitors a path for changes. Instead of using a polling mechanism that may cause an additional load (due to fast poll) or extra latency (because of slow poll). When user add query ?watch=true to API server request, the API server switch to watch mode, thus leaving the connection between client and server open. Int this mode, the data returned is not the object, but also the state of the object itself.
- Optimistically Concurrent Updates



Quick Intro to Etcd

- Etcd name is derived from the fact that /etc path in *nix is used to store data and config with the addition of d (distributed).
- Etcd using Raft protocol
- The data model formed in the form of hierarchy of key and value pair, this was replaced with flat model in the etcd3

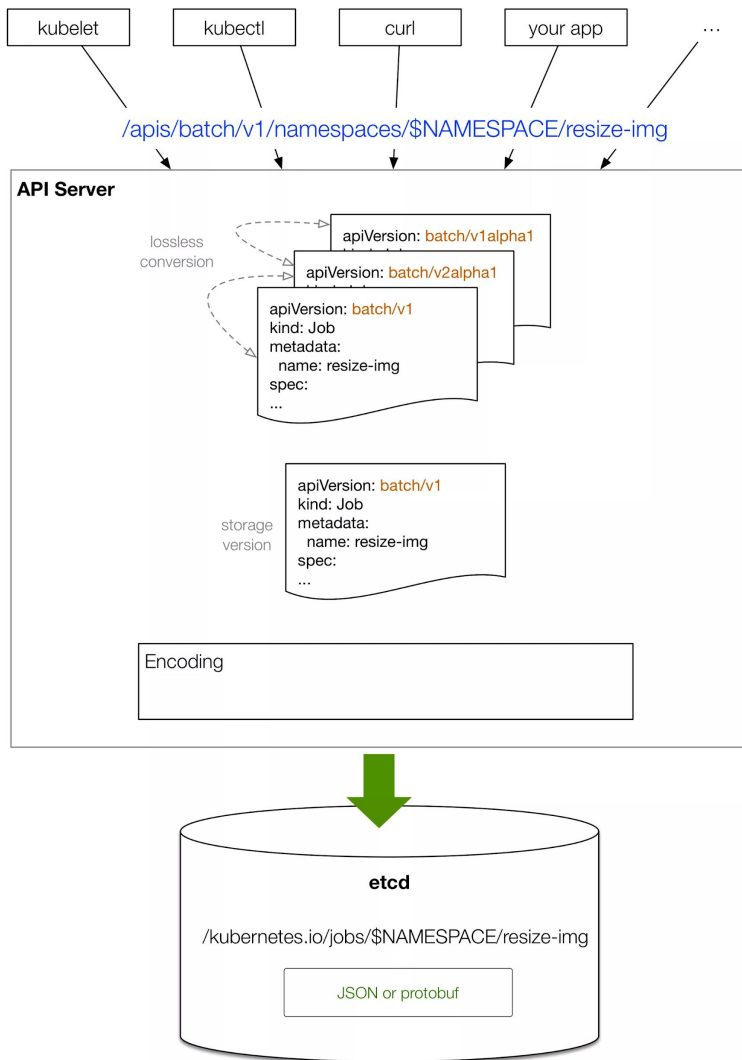


Hierarchy of data model in etcd (key-value store)




Cluster state in etcd

- Etcd is an independent component of the control plane.
- Version used: etcd2 in 1.5.x and etcd3(flat data model) forward
- The way of API server using the etcd can be influenced during start-time



This is what actually happened when we perform `kubectl create -f pod.yaml`

- 
- A client such as kubectl provides an desired object state, for example, YAML in version v1.
 - kubectl converts the YAML into JSON to send it over the wire.
 - Between different versions of the same kind, the API server can perform a lossless conversion leveraging annotations to store information that cannot be expressed in older API versions.
 - The API Server turns the input object state into a canonical storage version, depending on the API Server version itself, usually the newest stable one, for example, v1.
 - Actual storage process in etcd, at a certain key, into a value with the encoding to JSON or protobuf.



Serialization of Stateflow in Details

- The API Server store all known Kubernetes object into Go [type registry](#) called Scheme.
 - Each versions of kinds are defined along with how they can be converted, how new objects are created, and how an object can be encoded and decoded to JSON and Protobuf

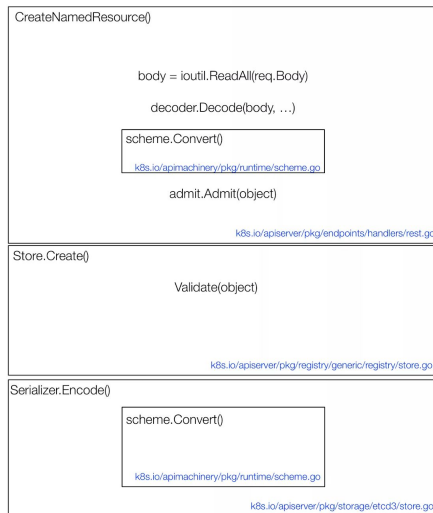
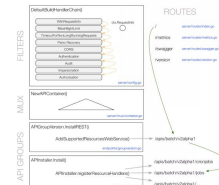


- API server receive an object, it will know from the HTTP path which version to expect and create a matching empty object using Scheme in right version.
- It will convert the HTTP payload using JSON/Protobuf decoder to created object
- The object is now in one of supported versions of given type.

POST /apis/batch/v2alpha1/namespaces/\$NAMESPACE/jobs/resize-img



HTTP API



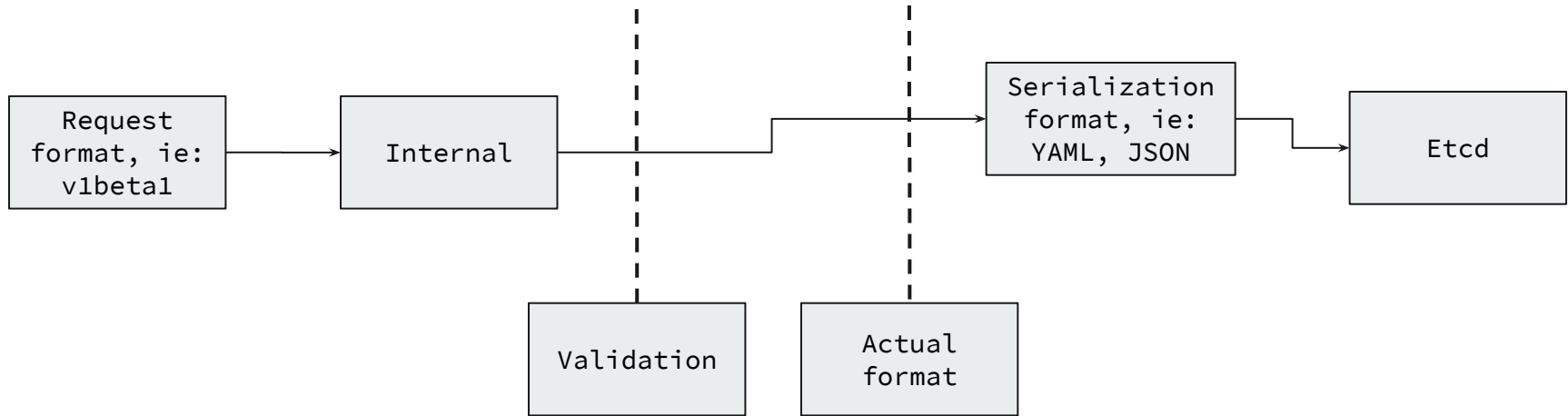


Lossless Conversions

- How to handle conversions problem?
 - Lossless conversion!
 - API Server should know how to handle those problem between each pair of version conversions.. ie. $v1 \Leftrightarrow v1alpha1$, $v1 \Leftrightarrow v1beta1$, $v1beta1 \Leftrightarrow v1alpha1$
 - API Server use a special internal version for each type, which is actually a superset of all supported versions for the type with all of its feature.
 - The flow will be convert the incoming object to internal version and then storage version..

$v1beta1 \Rightarrow \text{internal} \Rightarrow v1$

Admission and Validation





Admission and Validation (Cont'd)

- Admission, check that an object can be created or updated by verifying global constraints and might set default, depends on cluster config, there's a number of them, including (many more available in multi-tenant k8s):
 - **NamespaceLifecycle** - rejects all incoming requests in a namespace context if the namespace does not exist.
 - **LimitRanger** - enforces usage limit per resource basis in namespace
 - **ServiceAccount** - create svc account for pod
 - **DefaultStorageClass** - sets the default value of PersistentVolumeClaim's storage class, in case a user don't provide its value
 - **ResourceQuota** - enforces quota constraints for the current user on the cluster and might reject requests if the quota is not enough.



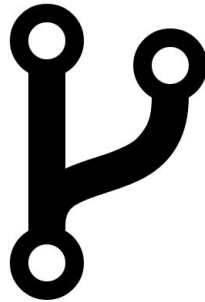
Admission and Validation (Cont'd)

- Validation, checks that an incoming object (during creation and updates) is well-formed in the sense that it only has valid values, for example:
 - checks that all mandatory fields are set.
 - checks that all strings have a valid format (for example, only include lowercase characters).
 - checks that no contradicting fields are set (for example, two containers with the same name).

Declaration and Creation of CRDs: Intro



How to integrate
owned resources
to k8s API



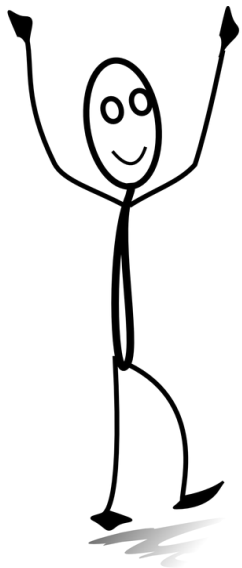
Fork and patch
kube-apiserver source code
+ integrate owned resources



Core API grow over time ->
bloated API



Discovering and Using CRDs



1. Custom Resource Definition (CRDs) which formerly were called Third Party Resources (TPRs).
 - + Simple, yet flexible way to define your own object kinds and let the API Server handle the entire lifecycle.
2. User API Servers (UAS) that run in parallel to the main API Server.
 - These are more involved in terms of development and require you to invest more up-front
 - + Give you much more fine-grained control over what is going on with the objects.



References

1. Tracey, Craig and Burns, Brendan. Managing Kubernetes. O'Reilly Media, Inc.
2. <https://blog.openshift.com/kubernetes-deep-dive-api-server-part-1/>
3. <https://blog.openshift.com/kubernetes-deep-dive-api-server-part-2/>
4. <https://blog.openshift.com/kubernetes-deep-dive-api-server-part-3a/>
5. <https://blog.openshift.com/kubernetes-deep-dive-code-generation-customresources/>

Q & A