
KubeCon
CloudNativeCon
North America 2019

Sig-API-Machinery Deep Dive

Antoine Pelisse (Google), Stefan Schimanski (Red Hat)

Agenda

- CRDs
  - Immutability
  - Equality
  - x-kubernetes-list-type / x-kubernetes-map-type
- Server-Side Apply
- Priority & Fairness

WIP: Immutability


```yaml
type: object
properties:
  slice:
    type: array
    x-kubernetes-mutability: Immutable
    items:
      type: string
```

{“slice”: [“a”,”b”]}	→ {“slice”: [“a”,”b”]}	✓
{“slice”: [“a”,”b”]}	→ {“slice”: [“b”,”a”]}	×
{“slice”: []}	→ {“slice”: null}	?
{“slice”: []}	→ {}	?

```yaml
type: object
properties:
  slice:
    type: array
    items:
      type: string
      x-kubernetes-mutability: Immutable
      nullable: true
```

{“slice”: [“a”,”b”]}	→ {“slice”: [“a”,”b”]}	✓
{“slice”: [“a”,”b”]}	→ {“slice”: [“b”,”a”]}	×
{“slice”: [“a”,”b”]}	→ {“slice”: [”a”]}	✓ ?
{“slice”: [“a”]}	→ {“slice”: [”a”,”b”]}	✓ ?
{“slice”: []}	→ {“slice”: null}	✓ ?
{“slice”: []}	→ {}	✓ ?
{“slice”: [“”]}	→ {“slice”: [null]}	×

Defaulting


```
type: object
properties:
  slice:
    type: array
    x-kubernetes-mutability: Immutable
    items:
      type: string
```

Assume: {“slice”: []} → {} ✓

Is this a good behaviour?

Defaulting


```
type: object
properties:
  slice:
    type: array
    x-kubernetes-mutability: Immutable
    items:
      type: string
    default: [“a”]
```

defaulting is strict.

Assume: {“slice”: []} → {} ✓

Is this a good behaviour?

{“slice”: []} → {} → {“slice”:[“a”]} ✓?

defaulting

Validation


```
type: object
properties:
  slice:
    type: array
    x-kubernetes-mutability: Immutable
    items:
      type: string
required: [“slice”]
```

validation is strict.

Assume: {“slice”: []} → {} ✓

Is this a good behaviour?

valid → invalid

Equality

JSON

When are objects equal?

Rule: if object A **==** object B, then request on A **==** request on B.

in etcd	in etcd
in request	in response

Corollary:

With defaulting and validation being strict, equality must be strict (`reflect.DeepEqual`)

Equality

When are objects equal? reflect.DeepEqual

JSON

Is this what we want? Was this an accident?

Native types:

```
type Foo struct {
  Slice []string `json:"slice,omitempty"`
}
```

```
json.Unmarshal(`{"slice": []`, &Foo{}}  →  Foo{Slice: nil}
```

Native types (often) normalize, CRDs never do.

Protobuf

Protobuf

When are objects equal?

```
type Foo struct {
  Slice []string `protobuf:"bytes,2,rep,name=slice"`
}
```

`[]` → `nil` (even without omitempty)

`null` → `nil`

Protobuf normalizes even more.

Equality

JSON

When are objects equal? reflect.DeepEqual

Is this what we want? Was this an accident?

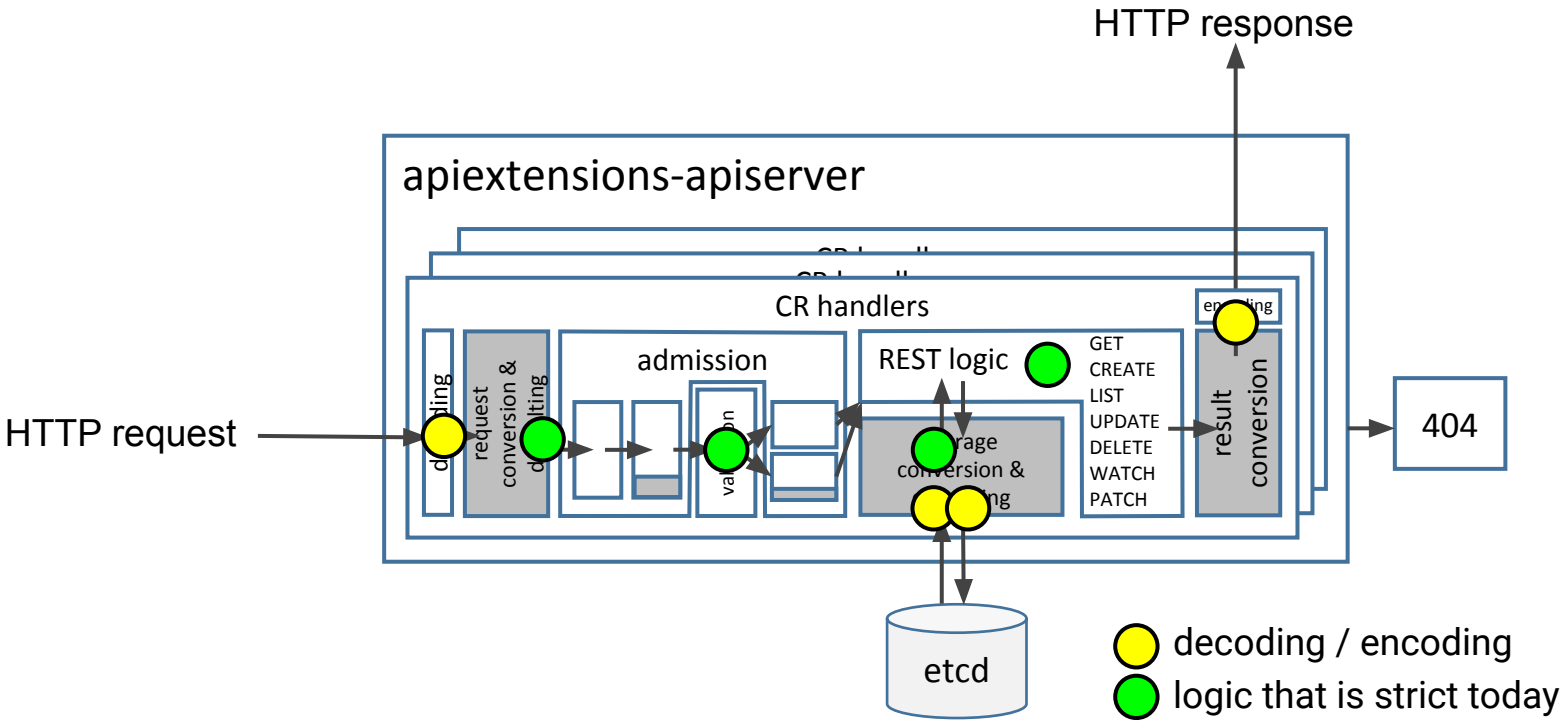
Native types:

```
type Foo struct {
  Slice []string `json:"slice,omitempty"`
}
```

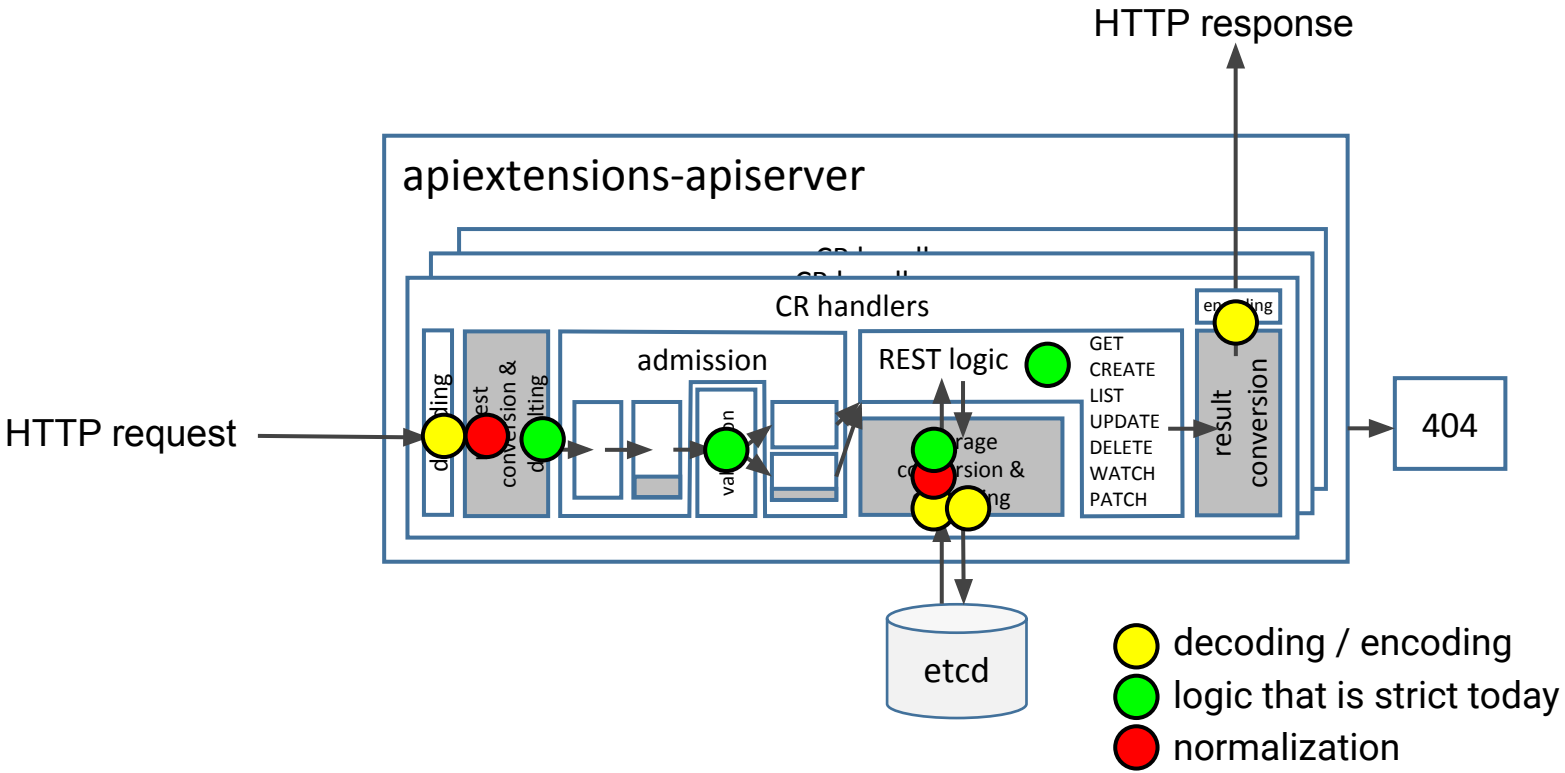
```
json.Unmarshal(`{"slice": []`, &Foo{}}  →  Foo{Slice: nil}
```

Native types (often) normalize, CRDs never do. Should they?

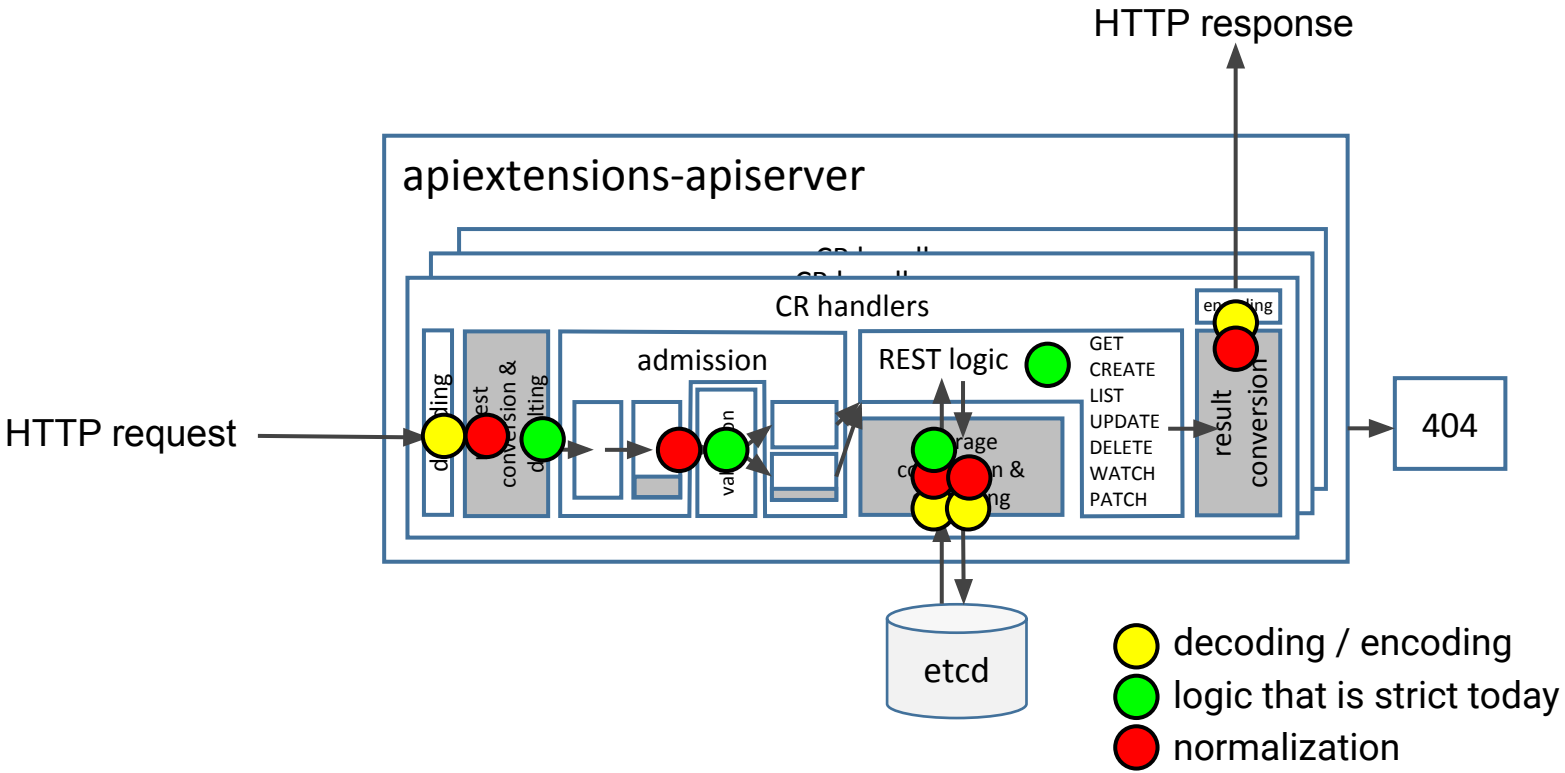
Request normalization



Request normalization



Request normalization



List-type / map-type

- **native types: strategic merge patch** defines merge strategy.
- CRDs never supported SMP. CRDs support server-side-apply.

New CRD OpenAPI extensions (since 1.16):

- `x-kubernetes-list-type: atomic (default) | set | map`
  `x-kubernetes-list-map-keys: ["name"]`

- `x-kubernetes-map-type: atomic | granular (default)`

Only with structural schemas.

Lists

- `x-kubernetes-list-type: atomic | set | map` (default)
  `x-kubernetes-list-map-keys: ["name"]`

keys fields must be scalar or atomic

unique keys

```
{"array": [
    {"name":"a", "value":42},
    {"name":"b", "value":1}
]}
```

map

unique items

```
{"array": [
    {"a":"x", "b":42},
    {"a":"y", "b":1},
    {"a":"y", "c":[1,2,3]}
]}
```

set

Maps

`x-kubernetes-map-type: atomic | granular` (default)

`{“map”:{“a”:”x”}} + {“map”:{“b”:42}} → {“map”:{“a”:”x”, “b”:42}}`

granular

`{“map”:{“a”:”x”}} + {“map”:{“b”:42}} → {“map”:{“b”:42}}`

atomic

Server-side Apply: Declarative

Kubernetes is about declarative “configurations”

Resources specify intent, and allow different actors to have different opinions.

`kubectl apply` allows declarative intents:

- No multiple actors
- No intent for controllers!

Client-side Apply: Limitations

Client-side apply uses “Strategic-Merge Patch”:

- Tedious update to protocol
- Requires coordinated client and server changes

Only has implementation in Go, or shell-out `kubectl`

It doesn’t support:

- multi-keys associative lists
- unions
- multiple appliers
- multiple versions
- so many other bugs

Server-side Apply: Overview

From very far away:

- Server-side Apply tracks which actors manage which fields for all operations
- Clients “apply” their intent, and only their intent
- Their intent is merged on the server

Field Management

Server-side apply manages everyone's intent.

Two ways to determine intent:

- Apply: Actor has an opinion about each fields specified in the configuration they send.

- Update: The intent is computed from the fields that have changed.

Apply and Update workflows

“Update” is triggered by the well-known existing flow:

- POST
- PUT
- PATCH (SMP, JSonPatch, JSon Merge patch)

“Apply” is triggered by sending a Yaml PATCH:

```
$ cat <<EOF | curl -XPATCH -d @- -H “Content-Type: application/apply-patch+yaml” \
    server/apis/apps/v1/namespaces/default/deployments/nginx

apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx
...
```

Fields Sets

- Set is a trie of fields owned:

```
"f:metadata":{"f:labels":{"f:sidecar_version": {}}},
"f:spec":{"f:template":{"f:spec":{"f:containers":{
  "k:{\"name\":\"sidecar\"}":{".": {},"f:image": {}}
}}}}
```

- One fields set per manager and per version
- Fields can be owned by multiple managers if they set the same value
- Changing value either takes over the ownership, or causes a conflict

Conflicts

- Update always grabs the ownership when a value is changed: all other managers lose that field.
- Apply has more cases:
  - If the value is the same, the ownership is shared (field is present in multiple sets)
  - If the value is different, a conflict is returned (e.g. “`spec.replicas is managed by hpa`”)
  - Conflicts can be forced, with the force query parameter to the request.

Merging

Merging is “simple”: Add all applied change on top of existing object

Fields that are not applied are left unchanged

We then remove list or map items that were formerly owned by that manager, and not owned by any other applyer.

What's missing?

There are a few things that we need to improve:

- Performance: tracking all fields of all objects takes time.
- Field set size: we'd love to find a more compact format for the field set
- Unions: SSA creates a single path for all resources, CRD included, so we can implement unions there.
- Ability to "declaratively remove" fields or list/map items.
- Tracking changes from mutating webhooks (but, performance …)

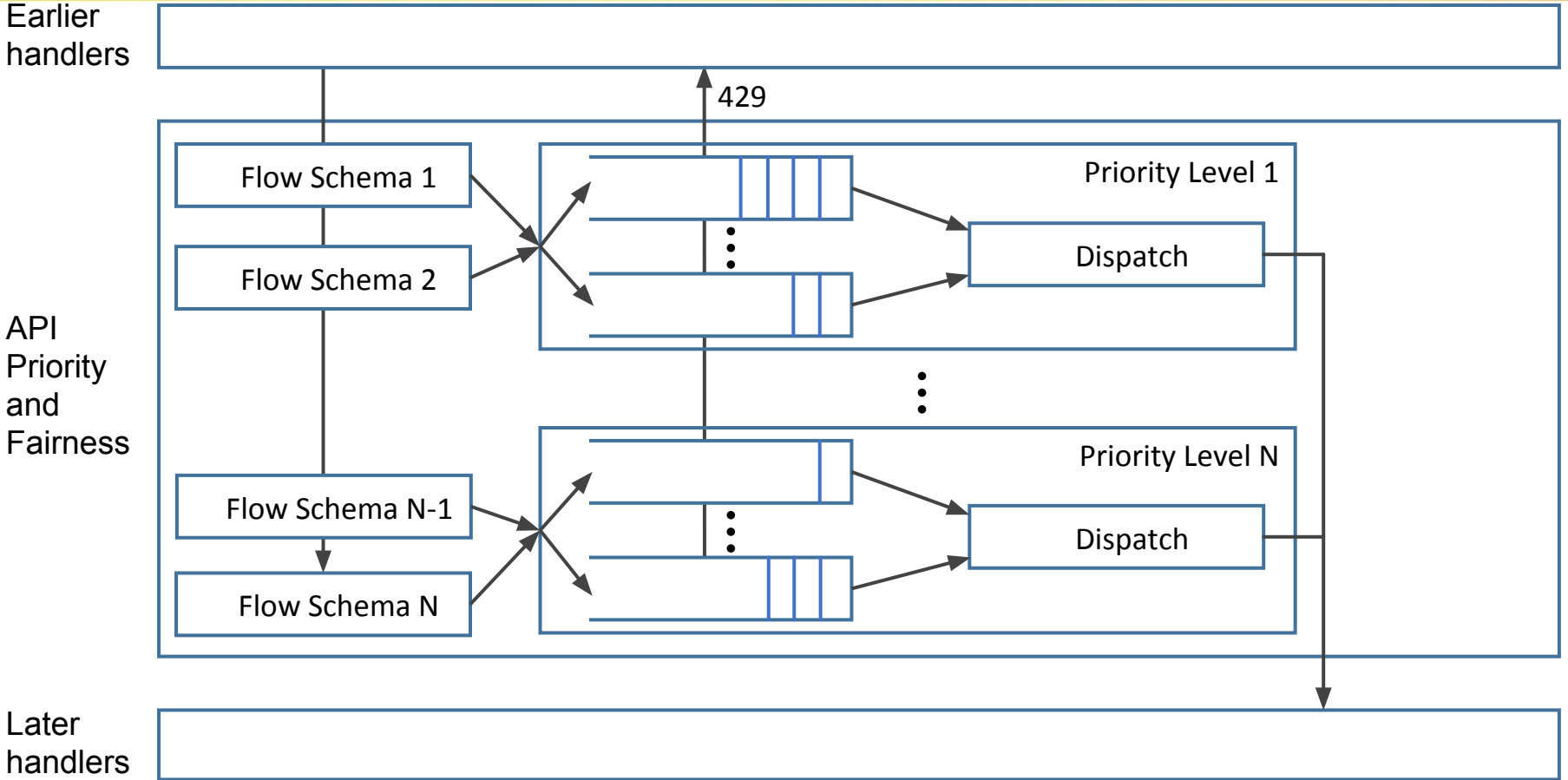
API Priority and Fairness

- Aaron Prindle, Google
- Bruce Ma, Ant Financial
- Daniel Smith, Google
- Mike Spreitzer, IBM
- Min Jin, Ant Financial
- Tony He, Ant Financial

API Priority and Fairness

- Goals:
  - Reserve capacity for self-maintenance
  - Protection against buggy controllers
  - Protection against buggy/greedy parts of workload
- What to regulate:
  - The product of dispatch rate X execution duration
  - … that is, the number executing
- Approach:
  - Divide server's capacity among priority levels
  - Concurrency limit and optionally queuing at each priority level
  - Classify request to flow, associate flow to priority level
- This is a more sophisticated version of the max-in-flight limit

API Priority and Fairness



API Priority and Fairness

- Example PriorityLevelConfiguration:

```
kind: PriorityLevelConfiguration
spec:
    type: Limited
    limited:
        assuredConcurrencyShares: 30
        limitResponse:
            type: Queue
            queuing:
                queues: 50
                handSize: 3
                queueLengthLimit: 10
```

- PriorityLevelConfiguration with no queuing:

```
kind: PriorityLevelConfiguration
spec:
    type: Limited
    limited:
        assuredConcurrencyShares: 30
        limitResponse:
            type: Reject
```

- PriorityLevelConfiguration with no concurrency limit:

```
kind: PriorityLevelConfiguration
spec:
    type: Exempt
```

API Priority and Fairness

- Example FlowSchema:

```
kind: FlowSchema
spec:
    priorityLevelConfiguration: {name: system-high}
    matchingPrecedence: 1500
    distinguisherMethod: {type: ByUser}
    rules:
      - subjects:
          - kind: Group
          - group: {name: "system:nodes"}
      - resourceRules:
          - verbs: [get, list]
            apiGroups: [""]
            resources: [pods, services, nodes/status]
            namespaces: ["*"]
      - nonResourceRules:
          - verbs: [get, list]
            nonResourceURLs: ["*"]
```

API Priority and Fairness

- Match request from system service account to read anything:

```
kind: FlowSchema
spec:
    priorityLevelConfiguration: {name: system-high}
    matchingPrecedence: 1500
    distinguisherMethod: {type: ByNamespace}
    rules:
      - subjects:
          - kind: ServiceAcount
          - serviceAccount: {namespace: kube-system, name: "*"}
      - resourceRules:
          - verbs: [get, list]
            apiGroups: ["*"]
            resources: ["*"]
            clusterScope: true
            namespaces: ["*"]
```