Declarative API Validation

FEATURE STATE: `Kubernetes v1.33 [beta]`

Kubernetes 1.34 includes optional *declarative validation* for APIs. When enabled, the Kubernetes API server can use this mechanism rather than the legacy approach that relies on hand-written Go code (`validation.go` files) to ensure that requests against the API are valid. Kubernetes developers, and people extending the Kubernetes API, can define validation rules directly alongside the API type definitions (`types.go` files). Code authors define special comment tags (e.g., `+k8s:minimum=0`). A code generator (`validation-gen`) then uses these tags to produce optimized Go code for API validation.

While primarily a feature impacting Kubernetes contributors and potentially developers of extension API servers, cluster administrators should understand its behavior, especially during its rollout phases.

Declarative validation is being rolled out gradually. In Kubernetes 1.34, the APIs that use declarative validation include:

- ReplicationController

Note:

For the beta of this feature, Kubernetes is intentionally using a superseded API as a test bed for the change. Future Kubernetes releases may roll this out to more APIs.

- `DeclarativeValidation`: (Beta, Default: `true`) When enabled, the API server runs *both* the new declarative validation and the old hand-written validation for migrated types/fields. The results are compared internally.
- `DeclarativeValidationTakeover`: (Beta, Default: `false`) This gate determines which validation result is *authoritative* (i.e., returned to the user and used for admission decisions).

Default Behavior (Kubernetes 1.34):

- With `DeclarativeValidation=true` and `DeclarativeValidationTakeover=false` (the default values for the gates), both validation systems run.
- **The results of the *hand-written* validation are used.** The declarative validation runs in a mismatch mode for comparison.
- Mismatches between the two validation systems are logged by the API server and increment the `declarative_validation_mismatch_total` metric. This helps developers identify and fix discrepancies during the Beta phase.
- **Cluster upgrades should be safe** regarding this feature, as the authoritative validation logic doesn't change by default.

Administrators can choose to explicitly enable `DeclarativeValidationTakeover=true` to make the *declarative* validation authoritative for migrated fields, typically after verifying stability in their environment (e.g., by monitoring the mismatch metric).

Disabling declarative validation

As a cluster administrator, you might consider disabling declarative validation whilst it is still beta, under specific circumstances:

- **Unexpected Validation Behavior:** If enabling `DeclarativeValidationTakeover` leads to unexpected validation errors or allows objects that were previously invalid.
- **Performance Regressions:** If monitoring indicates significant latency increases (e.g., in `apiserver_request_duration_seconds`) correlated with the feature's enablement.
- **High Mismatch Rate:** If the `declarative_validation_mismatch_total` metric shows frequent mismatches, suggesting potential bugs in the declarative rules affecting the cluster's workloads, even if `DeclarativeValidationTakeover` is false.

To revert to only using hand-written validation (as used before Kubernetes v1.33), disable the `DeclarativeValidation` feature gate, for example via command-line arguments: (`--feature-gates=DeclarativeValidation=false`). This also implicitly disables the effect of `DeclarativeValidationTakeover`.

Considerations for downgrade and rollback

Disabling the feature acts as a safety mechanism. However, be aware of a potential edge case (considered unlikely due to extensive testing): If a bug in declarative validation (when `DeclarativeValidationTakeover=true`) *incorrectly allowed* an invalid object to be persisted, disabling the feature gates might then cause subsequent updates to that specific object to be blocked by the now-authoritative (and correct) hand-written validation. Resolving this might require manual correction of the stored object, potentially via direct etcd modification in rare cases.

For details on managing feature gates, see Feature Gates.

Declarative validation tag reference

This document provides a comprehensive reference for all available declarative validation tags.

Tag catalog

Tag	Description
`+k8s:eachKey`	Declares a validation for each key in a map.
`+k8s:eachVal`	Declares a validation for each value in a map or list.
`+k8s:enum`	Indicates that a string type is an enum.
`+k8s:forbidden`	Indicates that a field may not be specified.
`+k8s:format`	Indicates that a string field has a particular format.
`+k8s:ifDisabled`	Declares a validation that only applies when an option is disabled.

Tag	Description
`+k8s:ifEnabled`	Declares a validation that only applies when an option is enabled.
`+k8s:isSubresource`	Specifies that validations in a package only apply to a specific subresource.
`+k8s:item`	Declares a validation for an item of a slice declared as a `+k8s:listType=map`.
`+k8s:listMapKey`	Declares a named sub-field of a list's value-type to be part of the list-map key.
`+k8s:listType`	Declares a list field's semantic type.
`+k8s:maxItems`	Indicates that a list field has a limit on its size.
`+k8s:maxLength`	Indicates that a string field has a limit on its length.
`+k8s:minimum`	Indicates that a numeric field has a minimum value.
`+k8s:neq`	Verifies the field's value is not equal to a specific disallowed value.
`+k8s:opaqueType`	Indicates that any validations declared on the referenced type will be ignored.
`+k8s:optional`	Indicates that a field is optional to clients.
`+k8s:required`	Indicates that a field must be specified by clients.
`+k8s:subfield`	Declares a validation for a subfield of a struct.
`+k8s:supportsSubresource`	Declares a supported subresource for the types within a package.
`+k8s:unionDiscriminator`	Indicates that this field is the discriminator for a union.
`+k8s:unionMember`	Indicates that this field is a member of a union group.
`+k8s:zeroOrOneOfMember`	Indicates that this field is a member of a zero-or-one-of group.

Tag Reference

`+k8s:eachKey`

Description:

Declares a validation for each key in a map.

Payload:

- `<validation-tag>`: The tag to evaluate for each key.

Usage Example:

```
type MyStruct struct {
    // +k8s:eachKey=+k8s:minimum=1
    MyMap map[int]string `json:"myMap"`}
```

In this example, `eachKey` is used to specify that the `+k8s:minimum` tag should be applied to each `int` key in `MyMap`. This means that all keys in the map must be >= 1.

`+k8s:eachVal`

Description:

Declares a validation for each value in a map or list.

Payload:

- `<validation-tag>`: The tag to evaluate for each value.

Usage Example:

```
type MyStruct struct {
    // +k8s:eachVal=+k8s:minimum=1
    MyMap map[string]int `json:"myMap"`}
```

In this example, `eachVal` is used to specify that the `+k8s:minimum` tag should be applied to each element in `MyList`. This means that all fields in `MyStruct` must be >= 1.

`+k8s:enum`

Description:

Indicates that a string type is an enum. All const values of this type are considered values in the enum.

Usage Example:

First, define a new string type and some constants of that type:

```
// +k8s:enum
type MyEnum stringconst (    MyEnumA MyEnum = "A"    MyEnumB MyEnum = "B")
```

Then, use this type in another struct:

```
type MyStruct struct {
    MyField MyEnum `json:"myField"`
}
```

The validation logic will ensure that `MyField` is one of the defined enum values (`"A"` or `"B"`).

`+k8s:forbidden`

Description:

Indicates that a field may not be specified.

Usage Example:

```
type MyStruct struct {
    // +k8s:forbidden
    MyField string `json:"myField"`}
```

In this example, `MyField` cannot be provided (it is forbidden) when creating or updating `MyStruct`.

`+k8s:format`

Description:

Indicates that a string field has a particular format.

Payloads:

- `k8s-ip`: This field holds an IPv4 or IPv6 address value. IPv4 octets may have leading zeros.
- `k8s-long-name`: This field holds a Kubernetes "long name", aka a "DNS subdomain" value.
- `k8s-short-name`: This field holds a Kubernetes "short name", aka a "DNS label" value.

Usage Example:

```
type MyStruct struct {
    // +k8s:format=k8s-ip
    IPAddress string `json:"ipAddress"`    // +k8s:format=k8s-long-name    Subdomain string `json:"subdomain"`    // +k8s:format=k
```

In this example:

- `IPAddress` must be a valid IP address.
- `Subdomain` must be a valid DNS subdomain.
- `Label` must be a valid DNS label.

`+k8s:ifDisabled`

Description:

Declares a validation that only applies when an option is disabled.

Arguments:

- `<option>` (string, required): The name of the option.

Payload:

- `<validation-tag>`: This validation tag will be evaluated only if the validation option is disabled.

Usage Example:

```
type MyStruct struct {
    // +k8s:ifDisabled("my-feature")=+k8s:required
    MyField string `json:"myField"`}
```

In this example, `MyField` is required only if the "my-feature" option is disabled.

`+k8s:ifEnabled`

Description:

Declares a validation that only applies when an option is enabled.

Arguments:

- `<option>` (string, required): The name of the option.

Payload:

- `<validation-tag>`: This validation tag will be evaluated only if the validation option is enabled.

Usage Example:

```
type MyStruct struct {
    // +k8s:ifEnabled("my-feature")=+k8s:required
    MyField string `json:"myField"`}
```

In this example, `MyField` is required only if the "my-feature" option is enabled.

`+k8s:isSubresource`

Description:

The `+k8s:isSubresource` tag is a package-level comment that **scopes the validation rules within that package to a specific subresource**. It essentially tells the code generator, "The validation logic defined here is the specific implementation for this subresource and should not be applied to the root object or any other subresource."

CRITICAL DEPENDENCY:

This tag is **dependent** on a corresponding `+k8s:supportsSubresource` tag being present in the package where the main API type is defined.

- `+k8s:supportsSubresource` opens the door by telling the dispatcher that a subresource is valid.
- `+k8s:isSubresource` provides the specialized validation logic that runs when a request comes through that door.

If you use `+k8s:isSubresource` without the corresponding `+k8s:supportsSubresource` declaration on the main type, the specialized validation code will be generated but will be **unreachable**. The main dispatcher will not recognize the subresource path and will reject the request before it can be routed to your specific validation logic.

This dependency allows for powerful organization, such as placing your main API types in one package and defining their subresource-specific validations in separate, dedicated packages.

Scope: Package

Payload:

- `<subresource-path>`: The path of the subresource to which the validations in this package should apply (e.g., `"/status"`, `"/scale"`).

Usage Example:

This two-part example demonstrates the intended use case of separating concerns.

1. Declare Support in the Main API Package: First, declare that the `Deployment` type supports `/scale` validation in its primary package.

File: `staging/src/k8s.io/api/apps/v1/doc.go`

```
// This enables the validation dispatcher to handle requests for "/scale".
// +k8s:supportsSubresource="/scale"
package v1// ... includes the definition for the Deployment type
```

2. Scope Validation Logic in a Separate Package: Next, create a separate package for the validation rules that are specific *only* to the `/scale` subresource.

File: `staging/src/k8s.io/api/apps/v1/validations/scale/doc.go`

```
// This ensures the rules in this package ONLY run for the "/scale" subresource.
// +k8s:isSubresource="/scale"
package scaleimport "k8s.io/api/apps/v1"// Validation code in this package would reference types from package v1 (e.g., v1.Scale)..
```

`+k8s:item`

Description:

Declares a validation for an item of a slice declared as a `+k8s:listType=map`. The item to match is declared by providing field-value pair arguments where the field is a `listMapKey`. All `listMapKey` key fields must be specified.

Usage:

`+k8s:item(<listMapKey-JSON-field-name>: <value>,...)=<validation-tag>`

`+k8s:item(stringKey: "value", intKey: 42, boolKey: true)=<validation-tag>`

Arguments must be named with the JSON names of the list-map key fields. Values can be strings, integers, or booleans.

Payload:

- `<validation-tag>`: The tag to evaluate for the matching list item.

Usage Example:

```
type MyStruct struct {
    // +k8s:listType=map
    // +k8s:listMapKey=type    // +k8s:item(type: "Approved")=+k8s:zeroOrOneOfMember    // +k8s:item(type: "Denied")=+k8s:zeroOrOn
```

In this example:

- The condition with `type` "Approved" is part of a zero-or-one-of group.
- The condition with `type` "Denied" is part of a zero-or-one-of group.

`+k8s:listMapKey`

Description:

Declares a named sub-field of a list's value-type to be part of the list-map key. This tag is required when `+k8s:listType=map` is used. Multiple `+k8s:listMapKey` tags can be used on a list-map to specify that it is keyed off of multiple fields.

Payload:

- `<field-json-name>`: The JSON name of the field to be used as the key.

Usage Example:

```
// +k8s:listType=map
// +k8s:listMapKey=keyFieldOne
// +k8s:listMapKey=keyFieldTwo
type MyList []MyStructtype MyStruct struct {    keyFieldOne string `json:"keyFieldOne"`    keyFieldTwo string `json:"keyFieldTwo"`
```

In this example, `listMapKey` is used to specify that the `keyField` of `MyStruct` should be used as the key for the list-map.

`+k8s:listType`

Description:

Declares a list field's semantic type. This tag is used to specify how a list should be treated, for example, as a map or a set.

Payload:

- `atomic`: The list is treated as a single atomic value.
- `map`: The list is treated as a map, where each element has a unique key. Requires the use of `+k8s:listMapKey`.
- `set`: The list is treated as a set, where each element is unique.

Usage Example:

```
// +k8s:listType=map
// +k8s:listMapKey=keyField
type MyList []MyStructtype MyStruct struct {    keyField string `json:"keyField"`    valueField string `json:"valueField"`}
```

In this example, `MyList` is declared as a list of type `map`, with `keyField` as the key. This means that the validation logic will ensure that each element in the list has a unique `keyField`.

`+k8s:maxItems`

Description:

Indicates that a list field has a limit on its size.

Payload:

- `<non-negative integer>`: This field must be no more than X items long.

Usage Example:

```
type MyStruct struct {
    // +k8s:maxItems=5
    MyList []string `json:"myList"`}
```

In this example, `MyList` cannot contain more than 5 items.

`+k8s:maxLength`

Description:

Indicates that a string field has a limit on its length.

Payload:

- `<non-negative integer>`: This field must be no more than X characters long.

Usage Example:

```
type MyStruct struct {
    // +k8s:maxLength=10
    MyString string `json:"myString"`}
```

In this example, `MyString` cannot be longer than 10 characters.

`+k8s:minimum`

Description:

Indicates that a numeric field has a minimum value.

Payload:

- `<integer>`: This field must be greater than or equal to `x`.

Usage Example:

```
type MyStruct struct {
    // +k8s:minimum=0
    MyInt int `json:"myInt"`}
```

In this example, `MyInt` must be greater than or equal to 0.

`+k8s:neq`

Description:

Verifies the field's value is not equal to a specific disallowed value. Supports string, integer, and boolean types.

Payload:

- `<value>`: The disallowed value. The parser will infer the type (string, int, bool).

Usage Example:

```
type MyStruct struct {
    // +k8s:neq="disallowed"
    MyString string `json:"myString"`    // +k8s:neq=0    MyInt int `json:"myInt"`    // +k8s:neq=true    MyBool bool `json:"myBoo
```

In this example:

- `MyString` cannot be equal to `"disallowed"`.
- `MyInt` cannot be equal to `0`.
- `MyBool` cannot be equal to `true`.

`+k8s:opaqueType`

Description:

Indicates that any validations declared on the referenced type will be ignored. If a referenced type's package is not included in the generator's current flags, this tag must be set, or code generation will fail (preventing silent mistakes). If the validations should not be ignored, add the type's package to the generator using the `--readonly-pkg` flag.

Usage Example:

```
import "some/external/package"

type MyStruct struct {
    // +k8s:opaqueType
    ExternalField package.ExternalType `json:"externalField"`}
```

In this example, any validation tags on `package.ExternalType` will be ignored.

`+k8s:optional`

Description:

Indicates that a field is optional to clients.

Usage Example:

```
type MyStruct struct {
    // +k8s:optional
    MyField string `json:"myField"`}
```

In this example, `MyField` is not required to be provided when creating or updating `MyStruct`.

`+k8s:required`

Description:

Indicates that a field must be specified by clients.

Usage Example:

```
type MyStruct struct {
    // +k8s:required
    MyField string `json:"myField"`}
```

In this example, `MyField` must be provided when creating or updating `MyStruct`.

`+k8s:subfield`

Description:

Declares a validation for a subfield of a struct.

Arguments:

- `<field-json-name>` (string, required): The JSON name of the subfield.

Payload:

- `<validation-tag>`: The tag to evaluate for the subfield.

Usage Example:

```
type MyStruct struct {
    // +k8s:subfield("mySubfield")=+k8s:required
    MyStruct MyStruct `json:"MyStruct"`}type MyStruct struct {    MySubfield string `json:"mySubfield"`}
```

In this example, `MySubfield` within `MyStruct` is required.

`+k8s:supportsSubresource`

Description:

The `+k8s:supportsSubresource` tag is a package-level comment tag that **declares which subresources are valid targets for validation** for the types within that package. Think of this tag as registering an endpoint; it tells the validation framework that a specific subresource path is recognized and should not be immediately rejected.

When the validation code is generated, this tag adds the specified subresource path to the main dispatch function for a type. This allows incoming requests for that subresource to be routed to a validation implementation.

Multiple tags can be used to declare support for several subresources. If no `+k8s:supportsSubresource` tags are present in a package, validation is only enabled for the root resource (e.g., `.../myresources/myobject`), and any requests to subresources will fail with a "no validation found" error.

Standalone Usage:

If you use `+k8s:supportsSubresource` without a corresponding `+k8s:isSubresource` tag for a specific validation, the validation rules for the root object will be applied to the subresource by default.

Scope: Package

Payload:

- `<subresource-path>`: The path of the subresource to support (e.g., `"/status"`, `"/scale"`).

Usage Example:

By adding these tags, you are enabling the validation system to handle requests for the `/status` and `/scale` subresources for the types defined in package `v1`.

File: `staging/src/k8s.io/api/core/v1/doc.go`

```
// +k8s:supportsSubresource="/status"
// +k8s:supportsSubresource="/scale"
package v1
```

`+k8s:unionDiscriminator`

Description:

Indicates that this field is the discriminator for a union.

Arguments:

- `union` (string, optional): The name of the union, if more than one exists.

Usage Example:

```
type MyStruct struct {
    TypeMeta int

    // +k8s:unionDiscriminator
    D D `json:"d"`    // +k8s:unionMember    // +k8s:optional    M1 *M1 `json:"m1"`    // +k8s:unionMember    // +k8s:optional
```

In this example, the `Type` field is the discriminator for the union. The value of `Type` will determine which of the union members (`M1` or `M2`) is expected to be present.

`+k8s:unionMember`

Description:

Indicates that this field is a member of a union.

Arguments:

- `union` (string, optional): The name of the union, if more than one exists.
- `memberName` (string, optional): The discriminator value for this member. Defaults to the field's name.

Usage Example:

```
type MyStruct struct {
    // +k8s:unionMember(union: "union1")
    // +k8s:optional    M1 *M1 `json:"u1m1"`    // +k8s:unionMember(union: "union1")    // +k8s:optional    M2 *M2 `json:"u1m2"`}t
```

In this example, `M1` and `M2` are members of the named union `union1`.

`+k8s:zeroOrOneOfMember`

Description:

Indicates that this field is a member of a zero-or-one-of union. A zero-or-one-of union allows at most one member to be set. Unlike regular unions, having no members set is valid.

Arguments:

- `union` (string, optional): The name of the union, if more than one exists.

- `memberName` (string, optional): The custom member name for this member. Defaults to the field's name.

Usage Example:

```
type MyStruct struct {
    // +k8s:zeroOrOneOfMember
    // +k8s:optional    M1 *M1 `json:"m1"`    // +k8s:zeroOrOneOfMember    // +k8s:optional    M2 *M2 `json:"m2"`}type M1 struct{}
```

In this example, at most one of `A` or `B` can be set. It is also valid for neither to be set.

Deprecated API Migration Guide

As the Kubernetes API evolves, APIs are periodically reorganized or upgraded. When APIs evolve, the old API is deprecated and eventually removed. This page contains information you need to know when migrating from deprecated API versions to newer and more stable API versions.

Removed APIs by release

v1.32

The **v1.32** release stopped serving the following deprecated API versions:

Flow control resources

The **flowcontrol.apiserver.k8s.io/v1beta3** API version of FlowSchema and PriorityLevelConfiguration is no longer served as of v1.32.

- Migrate manifests and API clients to use the **flowcontrol.apiserver.k8s.io/v1** API version, available since v1.29.
- All existing persisted objects are accessible via the new API
- Notable changes in **flowcontrol.apiserver.k8s.io/v1**:
    - The PriorityLevelConfiguration `spec.limited.nominalConcurrencyShares` field only defaults to 30 when unspecified, and an explicit value of 0 is not changed to 30.

v1.29

The **v1.29** release stopped serving the following deprecated API versions:

Flow control resources

The **flowcontrol.apiserver.k8s.io/v1beta2** API version of FlowSchema and PriorityLevelConfiguration is no longer served as of v1.29.

- Migrate manifests and API clients to use the **flowcontrol.apiserver.k8s.io/v1** API version, available since v1.29, or the **flowcontrol.apiserver.k8s.io/v1beta3** API version, available since v1.26.
- All existing persisted objects are accessible via the new API
- Notable changes in **flowcontrol.apiserver.k8s.io/v1**:
    - The PriorityLevelConfiguration `spec.limited.assuredConcurrencyShares` field is renamed to `spec.limited.nominalConcurrencyShares` and only defaults to 30 when unspecified, and an explicit value of 0 is not changed to 30.
- Notable changes in **flowcontrol.apiserver.k8s.io/v1beta3**:
    - The PriorityLevelConfiguration `spec.limited.assuredConcurrencyShares` field is renamed to `spec.limited.nominalConcurrencyShares`

v1.27

The **v1.27** release stopped serving the following deprecated API versions:

CSIStorageCapacity

The **storage.k8s.io/v1beta1** API version of CSIStorageCapacity is no longer served as of v1.27.

- Migrate manifests and API clients to use the **storage.k8s.io/v1** API version, available since v1.24.
- All existing persisted objects are accessible via the new API
- No notable changes

v1.26

The **v1.26** release stopped serving the following deprecated API versions:

Flow control resources

The **flowcontrol.apiserver.k8s.io/v1beta1** API version of FlowSchema and PriorityLevelConfiguration is no longer served as of v1.26.

- Migrate manifests and API clients to use the **flowcontrol.apiserver.k8s.io/v1beta2** API version.
- All existing persisted objects are accessible via the new API
- No notable changes

HorizontalPodAutoscaler

The **autoscaling/v2beta2** API version of HorizontalPodAutoscaler is no longer served as of v1.26.

- Migrate manifests and API clients to use the **autoscaling/v2** API version, available since v1.23.
- All existing persisted objects are accessible via the new API
- Notable changes:
    - `targetAverageUtilization` is replaced with `target.averageUtilization` and `target.type: Utilization`. See Autoscaling on multiple metrics and custom metrics.

v1.25

The **v1.25** release stopped serving the following deprecated API versions:

CronJob

The **batch/v1beta1** API version of CronJob is no longer served as of v1.25.

- Migrate manifests and API clients to use the **batch/v1** API version, available since v1.21.
- All existing persisted objects are accessible via the new API
- No notable changes

EndpointSlice

The **discovery.k8s.io/v1beta1** API version of EndpointSlice is no longer served as of v1.25.

- Migrate manifests and API clients to use the **discovery.k8s.io/v1** API version, available since v1.21.
- All existing persisted objects are accessible via the new API
- Notable changes in **discovery.k8s.io/v1**:
    - use per Endpoint `nodeName` field instead of deprecated `topology["kubernetes.io/hostname"]` field
    - use per Endpoint `zone` field instead of deprecated `topology["topology.kubernetes.io/zone"]` field
    - `topology` is replaced with the `deprecatedTopology` field which is not writable in v1

Event

The **events.k8s.io/v1beta1** API version of Event is no longer served as of v1.25.

- Migrate manifests and API clients to use the **events.k8s.io/v1** API version, available since v1.19.
- All existing persisted objects are accessible via the new API
- Notable changes in **events.k8s.io/v1**:
    - `type` is limited to `Normal` and `Warning`
    - `involvedObject` is renamed to `regarding`
    - `action`, `reason`, `reportingController`, and `reportingInstance` are required when creating new **events.k8s.io/v1** Events
    - use `eventTime` instead of the deprecated `firstTimestamp` field (which is renamed to `deprecatedFirstTimestamp` and not permitted in new **events.k8s.io/v1** Events)
    - use `series.lastObservedTime` instead of the deprecated `lastTimestamp` field (which is renamed to `deprecatedLastTimestamp` and not permitted in new **events.k8s.io/v1** Events)
    - use `series.count` instead of the deprecated `count` field (which is renamed to `deprecatedCount` and not permitted in new **events.k8s.io/v1** Events)
    - use `reportingController` instead of the deprecated `source.component` field (which is renamed to `deprecatedSource.component` and not permitted in new **events.k8s.io/v1** Events)
    - use `reportingInstance` instead of the deprecated `source.host` field (which is renamed to `deprecatedSource.host` and not permitted in new **events.k8s.io/v1** Events)

HorizontalPodAutoscaler

The **autoscaling/v2beta1** API version of HorizontalPodAutoscaler is no longer served as of v1.25.

- Migrate manifests and API clients to use the **autoscaling/v2** API version, available since v1.23.
- All existing persisted objects are accessible via the new API
- Notable changes:
    - `targetAverageUtilization` is replaced with `target.averageUtilization` and `target.type: Utilization`. See Autoscaling on multiple metrics and custom metrics.

PodDisruptionBudget

The **policy/v1beta1** API version of PodDisruptionBudget is no longer served as of v1.25.

- Migrate manifests and API clients to use the **policy/v1** API version, available since v1.21.
- All existing persisted objects are accessible via the new API
- Notable changes in **policy/v1**:
    - an empty `spec.selector` (`{}`) written to a `policy/v1` PodDisruptionBudget selects all pods in the namespace (in `policy/v1beta1` an empty `spec.selector` selected no pods). An unset `spec.selector` selects no pods in either API version.

PodSecurityPolicy

PodSecurityPolicy in the **policy/v1beta1** API version is no longer served as of v1.25, and the PodSecurityPolicy admission controller will be removed.

Migrate to Pod Security Admission or a 3rd party admission webhook. For a migration guide, see Migrate from PodSecurityPolicy to the Built-In PodSecurity Admission Controller. For more information on the deprecation, see PodSecurityPolicy Deprecation: Past, Present, and Future.

RuntimeClass

RuntimeClass in the **node.k8s.io/v1beta1** API version is no longer served as of v1.25.

- Migrate manifests and API clients to use the **node.k8s.io/v1** API version, available since v1.20.
- All existing persisted objects are accessible via the new API
- No notable changes

v1.22

The **v1.22** release stopped serving the following deprecated API versions:

Webhook resources

The **admissionregistration.k8s.io/v1beta1** API version of MutatingWebhookConfiguration and ValidatingWebhookConfiguration is no longer served as of v1.22.

- Migrate manifests and API clients to use the **admissionregistration.k8s.io/v1** API version, available since v1.16.
- All existing persisted objects are accessible via the new APIs
- Notable changes:
    - `webhooks[*].failurePolicy` default changed from `Ignore` to `Fail` for v1
    - `webhooks[*].matchPolicy` default changed from `Exact` to `Equivalent` for v1
    - `webhooks[*].timeoutSeconds` default changed from `30s` to `10s` for v1
    - `webhooks[*].sideEffects` default value is removed, and the field made required, and only `None` and `NoneOnDryRun` are permitted for v1
    - `webhooks[*].admissionReviewVersions` default value is removed and the field made required for v1 (supported versions for AdmissionReview are `v1` and `v1beta1`)
    - `webhooks[*].name` must be unique in the list for objects created via `admissionregistration.k8s.io/v1`

CustomResourceDefinition

The **apiextensions.k8s.io/v1beta1** API version of CustomResourceDefinition is no longer served as of v1.22.

- Migrate manifests and API clients to use the **apiextensions.k8s.io/v1** API version, available since v1.16.
- All existing persisted objects are accessible via the new API
- Notable changes:
    - `spec.scope` is no longer defaulted to `Namespaced` and must be explicitly specified
    - `spec.version` is removed in v1; use `spec.versions` instead
    - `spec.validation` is removed in v1; use `spec.versions[*].schema` instead
    - `spec.subresources` is removed in v1; use `spec.versions[*].subresources` instead
    - `spec.additionalPrinterColumns` is removed in v1; use `spec.versions[*].additionalPrinterColumns` instead
    - `spec.conversion.webhookClientConfig` is moved to `spec.conversion.webhook.clientConfig` in v1
    - `spec.conversion.conversionReviewVersions` is moved to `spec.conversion.webhook.conversionReviewVersions` in v1
    - `spec.versions[*].schema.openAPIV3Schema` is now required when creating v1 CustomResourceDefinition objects, and must be a structural schema
    - `spec.preserveUnknownFields: true` is disallowed when creating v1 CustomResourceDefinition objects; it must be specified within schema definitions as `x-kubernetes-preserve-unknown-fields: true`
    - In `additionalPrinterColumns` items, the `JSONPath` field was renamed to `jsonPath` in v1 (fixes #66531)

APIService

The **apiregistration.k8s.io/v1beta1** API version of APIService is no longer served as of v1.22.

- Migrate manifests and API clients to use the **apiregistration.k8s.io/v1** API version, available since v1.10.
- All existing persisted objects are accessible via the new API
- No notable changes

TokenReview

The **authentication.k8s.io/v1beta1** API version of TokenReview is no longer served as of v1.22.

- Migrate manifests and API clients to use the **authentication.k8s.io/v1** API version, available since v1.6.
- No notable changes

SubjectAccessReview resources

The **authorization.k8s.io/v1beta1** API version of LocalSubjectAccessReview, SelfSubjectAccessReview, SubjectAccessReview, and SelfSubjectRulesReview is no longer served as of v1.22.

- Migrate manifests and API clients to use the **authorization.k8s.io/v1** API version, available since v1.6.
- Notable changes:
    - `spec.group` was renamed to `spec.groups` in v1 (fixes #32709)

CertificateSigningRequest

The **certificates.k8s.io/v1beta1** API version of CertificateSigningRequest is no longer served as of v1.22.

- Migrate manifests and API clients to use the **certificates.k8s.io/v1** API version, available since v1.19.
- All existing persisted objects are accessible via the new API
- Notable changes in `certificates.k8s.io/v1`:
    - For API clients requesting certificates:

        - `spec.signerName` is now required (see [known Kubernetes signers]()), and requests for `kubernetes.io/legacy-unknown` are not allowed to be created via the `certificates.k8s.io/v1` API
        - `spec.usages` is now required, may not contain duplicate values, and must only contain known usages
    - For API clients approving or signing certificates:
        - `status.conditions` may not contain duplicate types
        - `status.conditions[*].status` is now required
        - `status.certificate` must be PEM-encoded, and contain only `CERTIFICATE` blocks

Lease

The **coordination.k8s.io/v1beta1** API version of Lease is no longer served as of v1.22.

- Migrate manifests and API clients to use the **coordination.k8s.io/v1** API version, available since v1.14.
- All existing persisted objects are accessible via the new API
- No notable changes

Ingress

The **extensions/v1beta1** and **networking.k8s.io/v1beta1** API versions of Ingress is no longer served as of v1.22.

- Migrate manifests and API clients to use the **networking.k8s.io/v1** API version, available since v1.19.
- All existing persisted objects are accessible via the new API
- Notable changes:
    - `spec.backend` is renamed to `spec.defaultBackend`
    - The backend `serviceName` field is renamed to `service.name`
    - Numeric backend `servicePort` fields are renamed to `service.port.number`
    - String backend `servicePort` fields are renamed to `service.port.name`
    - `pathType` is now required for each specified path. Options are `Prefix`, `Exact`, and `ImplementationSpecific`. To match the undefined `v1beta1` behavior, use `ImplementationSpecific`.

IngressClass

The **networking.k8s.io/v1beta1** API version of IngressClass is no longer served as of v1.22.

- Migrate manifests and API clients to use the **networking.k8s.io/v1** API version, available since v1.19.
- All existing persisted objects are accessible via the new API
- No notable changes

RBAC resources

The **rbac.authorization.k8s.io/v1beta1** API version of ClusterRole, ClusterRoleBinding, Role, and RoleBinding is no longer served as of v1.22.

- Migrate manifests and API clients to use the **rbac.authorization.k8s.io/v1** API version, available since v1.8.
- All existing persisted objects are accessible via the new APIs
- No notable changes

PriorityClass

The **scheduling.k8s.io/v1beta1** API version of PriorityClass is no longer served as of v1.22.

- Migrate manifests and API clients to use the **scheduling.k8s.io/v1** API version, available since v1.14.
- All existing persisted objects are accessible via the new API
- No notable changes

Storage resources

The **storage.k8s.io/v1beta1** API version of CSIDriver, CSINode, StorageClass, and VolumeAttachment is no longer served as of v1.22.

- Migrate manifests and API clients to use the **storage.k8s.io/v1** API version
    - CSIDriver is available in **storage.k8s.io/v1** since v1.19.
    - CSINode is available in **storage.k8s.io/v1** since v1.17
    - StorageClass is available in **storage.k8s.io/v1** since v1.6
    - VolumeAttachment is available in **storage.k8s.io/v1** v1.13
- All existing persisted objects are accessible via the new APIs
- No notable changes

v1.16

The **v1.16** release stopped serving the following deprecated API versions:

NetworkPolicy

The **extensions/v1beta1** API version of NetworkPolicy is no longer served as of v1.16.

- Migrate manifests and API clients to use the **networking.k8s.io/v1** API version, available since v1.8.
- All existing persisted objects are accessible via the new API

DaemonSet

The **extensions/v1beta1** and **apps/v1beta2** API versions of DaemonSet are no longer served as of v1.16.

- Migrate manifests and API clients to use the **apps/v1** API version, available since v1.9.
- All existing persisted objects are accessible via the new API
- Notable changes:
    - `spec.templateGeneration` is removed
    - `spec.selector` is now required and immutable after creation; use the existing template labels as the selector for seamless upgrades
    - `spec.updateStrategy.type` now defaults to `RollingUpdate` (the default in `extensions/v1beta1` was `OnDelete`)

Deployment

The **extensions/v1beta1**, **apps/v1beta1**, and **apps/v1beta2** API versions of Deployment are no longer served as of v1.16.

- Migrate manifests and API clients to use the **apps/v1** API version, available since v1.9.
- All existing persisted objects are accessible via the new API
- Notable changes:
    - `spec.rollbackTo` is removed
    - `spec.selector` is now required and immutable after creation; use the existing template labels as the selector for seamless upgrades
    - `spec.progressDeadlineSeconds` now defaults to `600` seconds (the default in `extensions/v1beta1` was no deadline)
    - `spec.revisionHistoryLimit` now defaults to `10` (the default in `apps/v1beta1` was `2`, the default in `extensions/v1beta1` was to retain all)
    - `maxSurge` and `maxUnavailable` now default to `25%` (the default in `extensions/v1beta1` was `1`)

StatefulSet

The **apps/v1beta1** and **apps/v1beta2** API versions of StatefulSet are no longer served as of v1.16.

- Migrate manifests and API clients to use the **apps/v1** API version, available since v1.9.
- All existing persisted objects are accessible via the new API
- Notable changes:
    - `spec.selector` is now required and immutable after creation; use the existing template labels as the selector for seamless upgrades
    - `spec.updateStrategy.type` now defaults to `RollingUpdate` (the default in `apps/v1beta1` was `OnDelete`)

ReplicaSet

The **extensions/v1beta1**, **apps/v1beta1**, and **apps/v1beta2** API versions of ReplicaSet are no longer served as of v1.16.

- Migrate manifests and API clients to use the **apps/v1** API version, available since v1.9.
- All existing persisted objects are accessible via the new API
- Notable changes:
    - `spec.selector` is now required and immutable after creation; use the existing template labels as the selector for seamless upgrades

PodSecurityPolicy

The **extensions/v1beta1** API version of PodSecurityPolicy is no longer served as of v1.16.

- Migrate manifests and API client to use the **policy/v1beta1** API version, available since v1.10.
- Note that the **policy/v1beta1** API version of PodSecurityPolicy will be removed in v1.25.

What to do

Test with deprecated APIs disabled

You can test your clusters by starting an API server with specific API versions disabled to simulate upcoming removals. Add the following flag to the API server startup arguments:

```
--runtime-config=<group>/<version>=false
```

For example:

```
--runtime-config=admissionregistration.k8s.io/v1beta1=false,apiextensions.k8s.io/v1beta1,...
```

Locate use of deprecated APIs

Use client warnings, metrics, and audit information available in 1.19+ to locate use of deprecated APIs.

Migrate to non-deprecated APIs

- Update custom integrations and controllers to call the non-deprecated APIs
- Change YAML files to reference the non-deprecated APIs

  You can use the `kubectl convert` command to automatically convert an existing object:

  ```
  kubectl convert -f <file> --output-version <group>/<version>.
  ```

  For example, to convert an older Deployment to `apps/v1`, you can run:

  ```
  kubectl convert -f ./my-deployment.yaml --output-version apps/v1
  ```

  This conversion may use non-ideal default values. To learn more about a specific resource, check the Kubernetes API reference.

Note:

The `kubectl convert` tool is not installed by default, although in fact it once was part of `kubectl` itself. For more details, you can read the deprecation and removal issue for the built-in subcommand.

To learn how to set up `kubectl convert` on your computer, visit the page that is right for your operating system: Linux, macOS, or Windows.

Common Expression Language in Kubernetes

The Common Expression Language (CEL) is used in the Kubernetes API to declare validation rules, policy rules, and other constraints or conditions.

CEL expressions are evaluated directly in the API server, making CEL a convenient alternative to out-of-process mechanisms, such as webhooks, for many extensibility use cases. Your CEL expressions continue to execute so long as the control plane's API server component remains available.

Language overview

The CEL language has a straightforward syntax that is similar to the expressions in C, C++, Java, JavaScript and Go.

CEL was designed to be embedded into applications. Each CEL "program" is a single expression that evaluates to a single value. CEL expressions are typically short "one-liners" that inline well into the string fields of Kubernetes API resources.

Inputs to a CEL program are "variables". Each Kubernetes API field that contains CEL declares in the API documentation which variables are available to use for that field. For example, in the `x-kubernetes-validations[i].rules` field of CustomResourceDefinitions, the `self` and `oldSelf` variables are available and refer to the previous and current state of the custom resource data to be validated by the CEL expression. Other Kubernetes API fields may declare different variables. See the API documentation of the API fields to learn which variables are available for that field.

Example CEL expressions:

Examples of CEL expressions and the purpose of each

Rule	Purpose
`self.minReplicas <= self.replicas && self.replicas <= self.maxReplicas`	Validate that the three fields defining replicas are ordered appropriately
`'Available' in self.stateCounts`	Validate that an entry with the 'Available' key exists in a map
`(self.list1.size() == 0) != (self.list2.size() == 0)`	Validate that one of two lists is non-empty, but not both
`self.envars.filter(e, e.name = 'MY_ENV').all(e, e.value.matches('^[a-zA-Z]*$'))`	Validate the 'value' field of a listMap entry where key field 'name' is 'MY_ENV'
`has(self.expired) && self.created + self.ttl < self.expired`	Validate that 'expired' date is after a 'create' date plus a 'ttl' duration
`self.health.startsWith('ok')`	Validate a 'health' string field has the prefix 'ok'
`self.widgets.exists(w, w.key == 'x' && w.foo < 10)`	Validate that the 'foo' property of a listMap item with a key 'x' is less than 10
`type(self) == string ? self == '99%' : self == 42`	Validate an int-or-string field for both the int and string cases
`self.metadata.name == 'singleton'`	Validate that an object's name matches a specific value (making it a singleton)
`self.set1.all(e, !(e in self.set2))`	Validate that two listSets are disjoint
`self.names.size() == self.details.size() && self.names.all(n, n in self.details)`	Validate the 'details' map is keyed by the items in the 'names' listSet
`self.details.all(key, key.matches('^[a-zA-Z]*$'))`	Validate the keys of the 'details' map
`self.details.all(key, self.details[key].matches('^[a-zA-Z]*$'))`	Validate the values of the 'details' map

CEL options, language features, and libraries

CEL is configured with the following options, libraries and language features, introduced at the specified Kubernetes versions:

CEL option, library or language feature	Included	Availability
Standard macros	`has`, `all`, `exists`, `exists_one`, `map`, `filter`	All Kubernetes versions
Standard functions	See official list of standard definitions	All Kubernetes versions
Homogeneous Aggregate Literals	-	All Kubernetes versions
Default UTC Time Zone	-	All Kubernetes versions
Eagerly Validate Declarations	-	All Kubernetes versions
Extended strings library, Version 1	`charAt`, `indexOf`, `lastIndexOf`, `lowerAscii`, `upperAscii`, `replace`, `split`, `join`, `substring`, `trim`	Kubernetes versions between 1.25 and 1.30
Extended strings library, Version 2	`charAt`, `indexOf`, `lastIndexOf`, `lowerAscii`, `upperAscii`, `replace`, `split`, `join`, `substring`, `trim`	Kubernetes versions 1.30+
Kubernetes list library	See Kubernetes list library	All Kubernetes versions
Kubernetes regex library	See Kubernetes regex library	All Kubernetes versions
Kubernetes URL library	See Kubernetes URL library	All Kubernetes versions
Kubernetes IP address library	See Kubernetes IP address library	Kubernetes versions 1.31+
Kubernetes CIDR library	See Kubernetes CIDR library	Kubernetes versions 1.31+
Kubernetes authorizer library	See Kubernetes authorizer library	All Kubernetes versions
Kubernetes quantity library	See Kubernetes quantity library	Kubernetes versions 1.29+

CEL option, library or language feature	Included	Availability
Kubernetes semver library	See Kubernetes semver library	Kubernetes versions 1.34+
Kubernetes format library	See Kubernetes format library	Kubernetes versions 1.32+
CEL optional types	See CEL optional types	Kubernetes versions 1.29+
CEL CrossTypeNumericComparisons	See CEL CrossTypeNumericComparisons	Kubernetes versions 1.29+
CEL TwoVarComprehensions	See CEL TwoVarComprehensions	Kubernetes versions 1.33+

CEL functions, features and language settings support Kubernetes control plane rollbacks. For example, *CEL Optional Values* was introduced at Kubernetes 1.29 and so only API servers at that version or newer will accept write requests to CEL expressions that use *CEL Optional Values*. However, when a cluster is rolled back to Kubernetes 1.28 CEL expressions using "CEL Optional Values" that are already stored in API resources will continue to evaluate correctly.

Kubernetes CEL libraries

In additional to the CEL community libraries, Kubernetes includes CEL libraries that are available everywhere CEL is used in Kubernetes.

Kubernetes list library

The list library includes `indexOf` and `lastIndexOf`, which work similar to the strings functions of the same names. These functions either the first or last positional index of the provided element in the list.

The list library also includes `min`, `max` and `sum`. Sum is supported on all number types as well as the duration type. Min and max are supported on all comparable types.

`isSorted` is also provided as a convenience function and is supported on all comparable types.

Examples:

Examples of CEL expressions using list library functions

CEL Expression	Purpose
`names.isSorted()`	Verify that a list of names is kept in alphabetical order
`items.map(x, x.weight).sum() == 1.0`	Verify that the "weights" of a list of objects sum to 1.0
`lowPriorities.map(x, x.priority).max() < highPriorities.map(x, x.priority).min()`	Verify that two sets of priorities do not overlap
`names.indexOf('should-be-first') == 1`	Require that the first name in a list if a specific value

See the Kubernetes List Library godoc for more information.

Kubernetes regex library

In addition to the `matches` function provided by the CEL standard library, the regex library provides `find` and `findAll`, enabling a much wider range of regex operations.

Examples:

Examples of CEL expressions using regex library functions

CEL Expression	Purpose
`"abc 123".find('[0-9]+')`	Find the first number in a string
`"1, 2, 3, 4".findAll('[0-9]+').map(x, int(x)).sum() < 100`	Verify that the numbers in a string sum to less than 100

See the Kubernetes regex library godoc for more information.

Kubernetes URL library

To make it easier and safer to process URLs, the following functions have been added:

- `isURL(string)` checks if a string is a valid URL according to the Go's net/url package. The string must be an absolute URL.
- `url(string) URL` converts a string to a URL or results in an error if the string is not a valid URL.

Once parsed via the `url` function, the resulting URL object has `getScheme`, `getHost`, `getHostname`, `getPort`, `getEscapedPath` and `getQuery` accessors.

Examples:

Examples of CEL expressions using URL library functions

CEL Expression	Purpose
`url('https://example.com:80/').getHost()`	Gets the 'example.com:80' host part of the URL
`url('https://example.com/path with spaces/').getEscapedPath()`	Returns '/path%20with%20spaces/'

See the Kubernetes URL library godoc for more information.

Kubernetes IP address library

To make it easier and safer to process IP addresses, the following functions have been added:

- `isIP(string)` checks if a string is a valid IP address.
- `ip(string) IP` converts a string to an IP address object or results in an error if the string is not a valid IP address.

For both functions, the IP address must be an IPv4 or IPv6 address. IPv4-mapped IPv6 addresses (e.g. `::ffff:1.2.3.4`) are not allowed. IP addresses with zones (e.g. `fe80::1%eth0`) are not allowed. Leading zeros in IPv4 address octets are not allowed.

Once parsed via the `ip` function, the resulting IP object has the following library of member functions:

Available member functions of an IP address object

Member Function	CEL Return Value	Description
`isCanonical()`	bool	Returns true if the IP address is in its canonical form. There is exactly one canonical form for every IP address, so fields containing IPs in canonical form can just be treated as strings when checking for equality or uniqueness.
`family()`	int	Returns the IP address family, `4` for IPv4 and `6` for IPv6.
`isUnspecified()`	bool	Returns true if the IP address is the unspecified address. Either the IPv4 address "0.0.0.0" or the IPv6 address "::".
`isLoopback()`	bool	Returns true if the IP address is the loopback address. Either an IPv4 address with a value of 127.x.x.x or an IPv6 address with a value of ::1.
`isLinkLocalMulticast()`	bool	Returns true if the IP address is a link-local multicast address. Either an IPv4 address with a value of 224.0.0.x or an IPv6 address in the network ff00::/8.
`isLinkLocalUnicast()`	bool	Returns true if the IP address is a link-local unicast address. Either an IPv4 address with a value of 169.254.x.x or an IPv6 address in the network fe80::/10.
`isGlobalUnicast()`	bool	Returns true if the IP address is a global unicast address. Either an IPv4 address that is not zero or 255.255.255.255 or an IPv6 address that is not a link-local unicast, loopback or multicast address.

Examples:

Examples of CEL expressions using IP address library functions

CEL Expression	Purpose
`isIP('127.0.0.1')`	Returns true for a valid IP.
`ip('2001:db8::abcd').isCanonical()`	Returns true for a canonical IPv6.
`ip('2001:DB8::ABCD').isCanonical()`	Returns false because the canonical form is lowercase.
`ip('127.0.0.1').family() == 4`	Check the address family of an IP.
`ip('::1').isLoopback()`	Check if an IP is a loopback address.
`ip('192.168.0.1').isGlobalUnicast()`	Check if an IP is a global unicast address.

See the Kubernetes IP address library godoc for more information.

Kubernetes CIDR library

CIDR provides a CEL function library extension of CIDR notation parsing functions.

`cidr`

Converts a string in CIDR notation to a network address representation or results in an error if the string is not a valid CIDR notation. The CIDR must be an IPv4 or IPv6 subnet address with a mask. Leading zeros in IPv4 address octets are not allowed. IPv4-mapped IPv6 addresses (e.g. `::ffff:1.2.3.4/24`) are not allowed.

`cidr(<string>) <CIDR>`

Examples:

`cidr('192.168.0.0/16')` // returns an IPv4 address with a CIDR mask `cidr('::1/128')` // returns an IPv6 address with a CIDR mask `cidr('192.168.0.0/33')` // error `cidr('::1/129')` // error `cidr('192.168.0.1/16')` // error, because there are non-0 bits after the prefix

`isCIDR`

Returns true if a string is a valid CIDR notation representation of a subnet with mask. The CIDR must be an IPv4 or IPv6 subnet address with a mask. Leading zeros in IPv4 address octets are not allowed. IPv4-mapped IPv6 addresses (e.g. `::ffff:1.2.3.4/24`) are not allowed.

`isCIDR(<string>) <bool>`

Examples:

`isCIDR('192.168.0.0/16')` // returns true `isCIDR('::1/128')` // returns true `isCIDR('192.168.0.0/33')` // returns false `isCIDR('::1/129')` // returns false

`containsIP` / `containsCIDR` / `ip` / `masked` / `prefixLength`

- `containsIP`: Returns true if a the CIDR contains the given IP address. The IP address must be an IPv4 or IPv6 address. May take either a string or IP address as an argument.
- `containsCIDR`: Returns true if a the CIDR contains the given CIDR. The CIDR must be an IPv4 or IPv6 subnet address with a mask. May take either a string or CIDR as an argument.
- `ip`: Returns the IP address representation of the CIDR.
- `masked`: Returns the CIDR representation of the network address with a masked prefix. This can be used to return the canonical form of the CIDR network.
- `prefixLength`: Returns the prefix length of the CIDR in bits. This is the number of bits in the mask.

Examples:

Examples of CEL expressions using CIDR library functions

CEL Expression	Purpose
`cidr('192.168.0.0/24').containsIP(ip('192.168.0.1'))`	Checks if a CIDR contains a given IP address (IP object).
`cidr('192.168.0.0/24').containsIP(ip('192.168.1.1'))`	Checks if a CIDR contains a given IP address (IP object).
`cidr('192.168.0.0/24').containsIP('192.168.0.1')`	Checks if a CIDR contains a given IP address (string).
`cidr('192.168.0.0/24').containsIP('192.168.1.1')`	Checks if a CIDR contains a given IP address (string).
`cidr('192.168.0.0/16').containsCIDR(cidr('192.168.10.0/24'))`	Checks if a CIDR contains another given CIDR (CIDR object).
`cidr('192.168.1.0/24').containsCIDR(cidr('192.168.2.0/24'))`	Checks if a CIDR contains another given CIDR (CIDR object).
`cidr('192.168.0.0/16').containsCIDR('192.168.10.0/24')`	Checks if a CIDR contains another given CIDR (string).
`cidr('192.168.1.0/24').containsCIDR('192.168.2.0/24')`	Checks if a CIDR contains another given CIDR (string).
`cidr('192.168.0.1/24').ip()`	Returns the IP address part of a CIDR.
`cidr('192.168.0.1/24').ip().family()`	Returns the family of the IP address part of a CIDR.
`cidr('::1/128').ip()`	Returns the IP address part of an IPv6 CIDR.
`cidr('::1/128').ip().family()`	Returns the family of the IP address part of an IPv6 CIDR.
`cidr('192.168.0.0/24').masked()`	Returns the canonical form of a CIDR network.
`cidr('192.168.0.1/24').masked()`	Returns the canonical form of a CIDR network, masking non-prefix bits.
`cidr('192.168.0.0/24') == cidr('192.168.0.0/24').masked()`	Compares a CIDR to its canonical form (already canonical).
`cidr('192.168.0.1/24') == cidr('192.168.0.1/24').masked()`	Compares a CIDR to its canonical form (not canonical).
`cidr('192.168.0.0/16').prefixLength()`	Returns the prefix length of an IPv4 CIDR.
`cidr('::1/128').prefixLength()`	Returns the prefix length of an IPv6 CIDR.

See the Kubernetes CIDR library godoc for more information.

Kubernetes authorizer library

For CEL expressions in the API where a variable of type `Authorizer` is available, the authorizer may be used to perform authorization checks for the principal (authenticated user) of the request.

API resource checks are performed as follows:

1. Specify the group and resource to check: `Authorizer.group(string).resource(string) ResourceCheck`
2. Optionally call any combination of the following builder functions to further narrow the authorization check. Note that these functions return the receiver type and can be chained:
   - `ResourceCheck.subresource(string) ResourceCheck`
   - `ResourceCheck.namespace(string) ResourceCheck`
   - `ResourceCheck.name(string) ResourceCheck`
3. Call `ResourceCheck.check(verb string) Decision` to perform the authorization check.
4. Call `allowed() bool` or `reason() string` to inspect the result of the authorization check.

Non-resource authorization performed are used as follows:

1. Specify only a path: `Authorizer.path(string) PathCheck`
2. Call `PathCheck.check(httpVerb string) Decision` to perform the authorization check.
3. Call `allowed() bool` or `reason() string` to inspect the result of the authorization check.

To perform an authorization check for a service account:

- `Authorizer.serviceAccount(namespace string, name string) Authorizer`

Examples of CEL expressions using URL library functions

CEL Expression	Purpose
`authorizer.group('').resource('pods').namespace('default').check('create').allowed()`	Returns true if the principal (user or service account) is allowed create pods in the 'default' namespace.
`authorizer.path('/healthz').check('get').allowed()`	Checks if the principal (user or service account) is authorized to make HTTP GET requests to the /healthz API path.
`authorizer.serviceAccount('default', 'myserviceaccount').resource('deployments').check('delete').allowed()`	Checks if the service account is authorized to delete deployments.

FEATURE STATE: `Kubernetes v1.34 [stable]` (enabled by default: true)

For CEL expressions in the API where a variable of type `Authorizer` is available, field and label selectors can be included in authorization checks.

Examples of CEL expressions using selector authorization functions

CEL Expression	Purpose
`authorizer.group('').resource('pods').fieldSelector('spec.nodeName=mynode').check('list').allowed()`	Returns true if the principal (or service account) is allowe list pods with the field select `spec.nodeName=mynode`.
`authorizer.group('').resource('pods').labelSelector('example.com/mylabel=myvalue').check('list').allowed()`	Returns true if the principal (or service account) is allowe list pods with the label select `example.com/mylabel=myva`

See the Kubernetes Authz library and Kubernetes AuthzSelectors library godoc for more information.

Kubernetes format library

The `format` library provides functions for validating common Kubernetes string formats. This can be useful in the `messageExpression` of validation rules to provide more specific error messages.

The library provides `format()` functions for each named format, and a generic `format.named()` function.

- `format.named(string)` → `?Format`: Returns the `Format` object for the given format name, if it exists. Otherwise, returns `optional.none`.
- `format.<formatName>() -> Format`: Convenience functions for all the named formats are also available. For example, `format.dns1123Label()` returns the `Format` object for DNS-1123 labels.
- `<Format>.validate(string) -> list<string>?`: Validates the given string against the format. Returns `optional.none` if the string is valid, otherwise an optional containing a list of validation error strings.

Available Formats:

The following format names are supported:

Available formats for the format library

Format Name	Description
`dns1123Label`	Validates if the string is a valid DNS-1123 label.
`dns1123Subdomain`	Validates if the string is a valid DNS-1123 subdomain.
`dns1035Label`	Validates if the string is a valid DNS-1035 label.
`qualifiedName`	Validates if the string is a valid qualified name.
`dns1123LabelPrefix`	Validates if the string is a valid DNS-1123 label prefix.
`dns1123SubdomainPrefix`	Validates if the string is a valid DNS-1123 subdomain prefix.
`dns1035LabelPrefix`	Validates if the string is a valid DNS-1035 label prefix.
`labelValue`	Validates if the string is a valid label value.
`uri`	Validates if the string is a valid URI. Uses the same pattern as `isURL`, but returns an error list.
`uuid`	Validates if the string is a valid UUID.
`byte`	Validates if the string is a valid base64 encoded string.
`date`	Validates if the string is a valid date in `YYYY-MM-DD` format.
`datetime`	Validates if the string is a valid datetime in RFC3339 format.

Examples:

Examples of CEL expressions using format library functions

CEL Expression	Purpose
`!format.dns1123Label().validate(self.metadata.name).hasValue()`	A validation rule that checks if an object's name is a valid DNS-1123 label.
`format.dns1123Label().validate(self.metadata.name).orValue([]).join("\\n")`	A `messageExpression` that returns specific validation errors for a field. If the field is valid, `validate` returns `optional.none`, and `orValue` provides an empty list, resulting in an empty string.

See the Kubernetes Format library godoc for more information.

Kubernetes quantity library

Kubernetes 1.28 adds support for manipulating quantity strings (ex 1.5G, 512k, 20Mi)

- `isQuantity(string)` checks if a string is a valid Quantity according to Kubernetes' resource.Quantity.
- `quantity(string) Quantity` converts a string to a Quantity or results in an error if the string is not a valid quantity.

Once parsed via the `quantity` function, the resulting Quantity object has the following library of member functions:

Available member functions of a Quantity

Member Function	CEL Return Value	Description
`isInteger()`	bool	Returns true if and only if asInteger is safe to call without an error
`asInteger()`	int	Returns a representation of the current value as an `int64` if possible or results in an error if conversion would result in overflowor loss of precision.
`asApproximateFloat()`	float	Returns a `float64` representation of the quantity which may lose precision. If the value of the quantity is outside the range of a `float64`, `+Inf/-Inf` will be returned.
`sign()`	int	Returns `1` if the quantity is positive, `-1` if it is negative. `0` if it is zero.
`add(<Quantity>)`	Quantity	Returns sum of two quantities
`add(<int>)`	Quantity	Returns sum of quantity and an integer
`sub(<Quantity>)`	Quantity	Returns difference between two quantities
`sub(<int>)`	Quantity	Returns difference between a quantity and an integer
`isLessThan(<Quantity>)`	bool	Returns true if and only if the receiver is less than the operand
`isGreaterThan(<Quantity>)`	bool	Returns true if and only if the receiver is greater than the operand
`compareTo(<Quantity>)`	int	Compares receiver to operand and returns 0 if they are equal, 1 if the receiver is greater, or -1 if the receiver is less than the operand

Examples:

Examples of CEL expressions using URL library functions

CEL Expression	Purpose
`quantity("500000G").isInteger()`	Test if conversion to integer would throw an error
`quantity("50k").asInteger()`	Precise conversion to integer
`quantity("9999999999999999999999999999999999999G").asApproximateFloat()`	Lossy conversion to float
`quantity("50k").add(quantity("20k"))`	Add two quantities
`quantity("50k").sub(20000)`	Subtract an integer from a quantity
`quantity("50k").add(20).sub(quantity("100k")).sub(-50000)`	Chain adding and subtracting integers and quantities
`quantity("200M").compareTo(quantity("0.2G"))`	Compare two quantities
`quantity("150Mi").isGreaterThan(quantity("100Mi"))`	Test if a quantity is greater than the receiver
`quantity("50M").isLessThan(quantity("100M"))`	Test if a quantity is less than the receiver

Kubernetes semver library

Kubernetes v1.34 adds support for parsing and comparing strings that follow the Semantic Versioning 2.0.0 specification. Refer to the semver.org documentation for information on accepted patterns.

- `isSemver(string)` checks if a string is a valid semantic version.
- `semver(string)` converts a string to a Semver object or results in an error.

An optional boolean `normalize` argument can be passed to `isSemver` and `semver`. If `true`, normalization removes any "v" prefix, adds a 0 minor and patch numbers to versions with only major or major.minor components specified, and removes any leading 0s.

Once parsed via the `semver` function, the resulting Semver object has the following library of member functions:

Available member functions of a Semver object

Member Function	CEL Return Value	Description
`major()`	int	Returns the major version number.
`minor()`	int	Returns the minor version number.
`patch()`	int	Returns the patch version number.
`isLessThan(<Semver>)`	bool	Returns true if and only if the receiver is less than the operand.
`isGreaterThan(<Semver>)`	bool	Returns true if and only if the receiver is greater than the operand.
`compareTo(<Semver>)`	int	Compares receiver to operand and returns 0 if they are equal, 1 if the receiver is greater, or -1 if the receiver is less than the operand.

Examples:

Examples of CEL expressions using semver library functions

CEL Expression	Purpose
`isSemver('1.0.0')`	Returns true for a valid Semver string.
`isSemver('v1.0', true)`	Returns true for a normalizable Semver string.
`semver('1.2.3').major()`	Returns the major version of a Semver.
`semver('1.2.3').compareTo(semver('2.0.0')) < 0`	Compare two Semver strings.

See the Kubernetes Semver library godoc for more information.

Type checking

CEL is a gradually typed language.

Some Kubernetes API fields contain fully type checked CEL expressions. For example, CustomResourceDefinitions Validation Rules are fully type checked.

Some Kubernetes API fields contain partially type checked CEL expressions. A partially type checked expression is an expressions where some of the variables are statically typed but others are dynamically typed. For example, in the CEL expressions of ValidatingAdmissionPolicies the `request` variable is typed, but the `object` variable is dynamically typed. As a result, an expression containing `request.namex` would fail type checking because the `namex` field is not defined. However, `object.namex` would pass type checking even when the `namex` field is not defined for the resource kinds that `object` refers to, because `object` is dynamically typed.

The `has()` macro in CEL may be used in CEL expressions to check if a field of a dynamically typed variable is accessible before attempting to access the field's value. For example:

```
has(object.namex) ? object.namex == 'special' : request.name == 'special'
```

Type system integration

Table showing the relationship between OpenAPIv3 types and CEL types

OpenAPIv3 type	CEL type
'object' with Properties	object / "message type" (`type(<object>)` evaluates to `selfType<uniqueNumber>.path.to.object.from.self`)
'object' with `additionalProperties`	map
'object' with `x-kubernetes-embedded-type`	object / "message type", 'apiVersion', 'kind', 'metadata.name' and 'metadata.generateName' are implicitly included in schema

OpenAPIv3 type	CEL type
'object' with x-kubernetes-preserve-unknown-fields	object / "message type", unknown fields are NOT accessible in CEL expression
`x-kubernetes-int-or-string`	Union of `int` or `string`, `self.intOrString < 100 | self.intOrString == '50%'` evaluates to true for both `50` and `"50%"`
'array'	list
'array' with `x-kubernetes-list-type=map`	list with map based Equality & unique key guarantees
'array' with `x-kubernetes-list-type=set`	list with set based Equality & unique entry guarantees
'boolean'	boolean
'number' (all formats)	double
'integer' (all formats)	int (64)
no equivalent	uint (64)
'null'	null_type
'string'	string
'string' with format=byte (base64 encoded)	bytes
'string' with format=date	timestamp (`google.protobuf.Timestamp`)
'string' with format=datetime	timestamp (`google.protobuf.Timestamp`)
'string' with format=duration	duration (`google.protobuf.Duration`)

Also see: CEL types, OpenAPI types, Kubernetes Structural Schemas.

Equality comparison for arrays with `x-kubernetes-list-type` of `set` or `map` ignores element order. For example `[1, 2] == [2, 1]` if the arrays represent Kubernetes `set` values.

Concatenation on arrays with `x-kubernetes-list-type` use the semantics of the list type:

`set`
: `X + Y` performs a union where the array positions of all elements in `X` are preserved and non-intersecting elements in `Y` are appended, retaining their partial order.

`map`
: `X + Y` performs a merge where the array positions of all keys in `X` are preserved but the values are overwritten by values in `Y` when the key sets of `X` and `Y` intersect. Elements in `Y` with non-intersecting keys are appended, retaining their partial order.

Escaping

Only Kubernetes resource property names of the form `[a-zA-Z_.-/][a-zA-Z0-9_.-/]*` are accessible from CEL. Accessible property names are escaped according to the following rules when accessed in the expression:

Table of CEL identifier escaping rules

escape sequence	property name equivalent
`__underscores__`	`__`
`__dot__`	`.`
`__dash__`	`-`
`__slash__`	`/`
`__{keyword}__`	CEL **RESERVED** keyword

When you escape any of CEL's **RESERVED** keywords you need to match the exact property name use the underscore escaping (for example, `int` in the word `sprint` would not be escaped and nor would it need to be).

Examples on escaping:

Examples escaped CEL identifiers

property name	rule with escaped property name
`namespace`	`self.__namespace__ > 0`
`x-prop`	`self.x__dash__prop > 0`
`redact_d`	`self.redact__underscores__d > 0`
`string`	`self.startsWith('kube')`

Resource constraints

CEL is non-Turing complete and offers a variety of production safety controls to limit execution time. CEL's *resource constraint* features provide feedback to developers about expression complexity and help protect the API server from excessive resource consumption during evaluation. CEL's resource constraint features are used to prevent CEL evaluation from consuming excessive API server resources.

A key element of the resource constraint features is a *cost unit* that CEL defines as a way of tracking CPU utilization. Cost units are independent of system load and hardware. Cost units are also deterministic; for any given CEL expression and input data, evaluation of the expression by the CEL interpreter will always result in the same cost.

Many of CEL's core operations have fixed costs. The simplest operations, such as comparisons (e.g. <) have a cost of 1. Some have a higher fixed cost, for example list literal declarations have a fixed base cost of 40 cost units.

Calls to functions implemented in native code approximate cost based on the time complexity of the operation. For example: operations that use regular expressions, such as `match` and `find`, are estimated using an approximated cost of `length(regexString)*length(inputString)`. The approximated cost reflects the worst case time complexity of Go's RE2 implementation.

Runtime cost budget

All CEL expressions evaluated by Kubernetes are constrained by a runtime cost budget. The runtime cost budget is an estimate of actual CPU utilization computed by incrementing a cost unit counter while interpreting a CEL expression. If the CEL interpreter executes too many instructions, the runtime cost budget will be exceeded, execution of the expressions will be halted, and an error will result.

Some Kubernetes resources define an additional runtime cost budget that bounds the execution of multiple expressions. If the sum total of the cost of expressions exceed the budget, execution of the expressions will be halted, and an error will result. For example the validation of a custom resource has a *per-validation* runtime cost budget for all Validation Rules evaluated to validate the custom resource.

Estimated cost limits

For some Kubernetes resources, the API server may also check if worst case estimated running time of CEL expressions would be prohibitively expensive to execute. If so, the API server prevent the CEL expression from being written to API resources by rejecting create or update operations containing the CEL expression to the API resources. This feature offers a stronger assurance that CEL expressions written to the API resource will be evaluated at runtime without exceeding the runtime cost budget.

Kubernetes Deprecation Policy

This document details the deprecation policy for various facets of the system.

Kubernetes is a large system with many components and many contributors. As with any such software, the feature set naturally evolves over time, and sometimes a feature may need to be removed. This could include an API, a flag, or even an entire feature. To avoid breaking existing users, Kubernetes follows a deprecation policy for aspects of the system that are slated to be removed.

Deprecating parts of the API

Since Kubernetes is an API-driven system, the API has evolved over time to reflect the evolving understanding of the problem space. The Kubernetes API is actually a set of APIs, called "API groups", and each API group is independently versioned. API versions fall into 3 main tracks, each of which has different policies for deprecation:

Example	Track
v1	GA (generally available, stable)
v1beta1	Beta (pre-release)
v1alpha1	Alpha (experimental)

A given release of Kubernetes can support any number of API groups and any number of versions of each.

The following rules govern the deprecation of elements of the API. This includes:

- REST resources (aka API objects)
- Fields of REST resources
- Annotations on REST resources, including "beta" annotations but not including "alpha" annotations.
- Enumerated or constant values
- Component config structures

These rules are enforced between official releases, not between arbitrary commits to master or release branches.

Rule #1: API elements may only be removed by incrementing the version of the API group.

Once an API element has been added to an API group at a particular version, it can not be removed from that version or have its behavior significantly changed, regardless of track.

Note:

For historical reasons, there are 2 "monolithic" API groups - "core" (no group name) and "extensions". Resources will incrementally be moved from these legacy API groups into more domain-specific API groups.

Rule #2: API objects must be able to round-trip between API versions in a given release without information loss, with the exception of whole REST resources that do not exist in some versions.

For example, an object can be written as v1 and then read back as v2 and converted to v1, and the resulting v1 resource will be identical to the original. The representation in v2 might be different from v1, but the system knows how to convert between them in both directions. Additionally, any new field added in v2 must be able to round-trip to v1 and back, which means v1 might have to add an equivalent field or represent it as an annotation.

Rule #3: An API version in a given track may not be deprecated in favor of a less stable API version.

- GA API versions can replace beta and alpha API versions.
- Beta API versions can replace earlier beta and alpha API versions, but *may not* replace GA API versions.
- Alpha API versions can replace earlier alpha API versions, but *may not* replace GA or beta API versions.

Rule #4a: API lifetime is determined by the API stability level

- GA API versions may be marked as deprecated, but must not be removed within a major version of Kubernetes
- Beta API versions are deprecated no more than 9 months or 3 minor releases after introduction (whichever is longer), and are no longer served 9 months or 3 minor releases after deprecation (whichever is longer)
- Alpha API versions may be removed in any release without prior deprecation notice

This ensures beta API support covers the maximum supported version skew of 2 releases, and that APIs don't stagnate on unstable beta versions, accumulating production usage that will be disrupted when support for the beta API ends.

Note:

There are no current plans for a major version revision of Kubernetes that removes GA APIs.

Note:

Until #52185 is resolved, no API versions that have been persisted to storage may be removed. Serving REST endpoints for those versions may be disabled (subject to the deprecation timelines in this document), but the API server must remain capable of decoding/converting previously persisted data from storage.

Rule #4b: The "preferred" API version and the "storage version" for a given group may not advance until after a release has been made that supports both the new version and the previous version

Users must be able to upgrade to a new release of Kubernetes and then roll back to a previous release, without converting anything to the new API version or suffering breakages (unless they explicitly used features only available in the newer version). This is particularly evident in the stored representation of objects.

All of this is best illustrated by examples. Imagine a Kubernetes release, version X, which introduces a new API group. A new Kubernetes release is made every approximately 4 months (3 per year). The following table describes which API versions are supported in a series of subsequent releases.

Release	API Versions	Preferred/Storage Version	Notes
X	v1alpha1	v1alpha1	
X+1	v1alpha2	v1alpha2	• v1alpha1 is removed. See release notes for required actions.
X+2	v1beta1	v1beta1	• v1alpha2 is removed. See release notes for required actions.
X+3	v1beta2, v1beta1 (deprecated)	v1beta1	• v1beta1 is deprecated. See release notes for required actions.
X+4	v1beta2, v1beta1 (deprecated)	v1beta2	
X+5	v1, v1beta1 (deprecated), v1beta2 (deprecated)	v1beta2	• v1beta2 is deprecated. See release notes for required actions.
X+6	v1, v1beta2 (deprecated)	v1	• v1beta1 is removed. See release notes for required actions.
X+7	v1, v1beta2 (deprecated)	v1	
X+8	v2alpha1, v1	v1	• v1beta2 is removed. See release notes for required actions.
X+9	v2alpha2, v1	v1	• v2alpha1 is removed. See release notes for required actions.
X+10	v2beta1, v1	v1	• v2alpha2 is removed. See release notes for required actions.
X+11	v2beta2, v2beta1 (deprecated), v1	v1	• v2beta1 is deprecated. See release notes for required actions.
X+12	v2, v2beta2 (deprecated), v2beta1 (deprecated), v1 (deprecated)	v1	• v2beta2 is deprecated. See release notes for required actions. • v1 is deprecated in favor of v2, but will not be removed
X+13	v2, v2beta1 (deprecated), v2beta2 (deprecated), v1 (deprecated)	v2	
X+14	v2, v2beta2 (deprecated), v1 (deprecated)	v2	• v2beta1 is removed. See release notes for required actions.
X+15	v2, v1 (deprecated)	v2	• v2beta2 is removed. See release notes for required actions.

REST resources (aka API objects)

Consider a hypothetical REST resource named Widget, which was present in API v1 in the above timeline, and which needs to be deprecated. We document and announce the deprecation in sync with release X+1. The Widget resource still exists in API version v1 (deprecated) but not in v2alpha1. The Widget resource continues to exist and function in releases up to and including X+8. Only in release X+9, when API v1 has aged out, does the Widget resource cease to exist, and the behavior get removed.

Starting in Kubernetes v1.19, making an API request to a deprecated REST API endpoint:

1. Returns a `Warning` header (as defined in RFC7234, Section 5.5) in the API response.
2. Adds a `"k8s.io/deprecated":"true"` annotation to the audit event recorded for the request.

3. Sets an `apiserver_requested_deprecated_apis` gauge metric to `1` in the `kube-apiserver` process. The metric has labels for `group`, `version`, `resource`, `subresource` that can be joined to the `apiserver_request_total` metric, and a `removed_release` label that indicates the Kubernetes release in which the API will no longer be served. The following Prometheus query returns information about requests made to deprecated APIs which will be removed in v1.22:

   ```
   apiserver_requested_deprecated_apis{removed_release="1.22"} * on(group,version,resource,subresource) group_right() apiserver_
   ```

Fields of REST resources

As with whole REST resources, an individual field which was present in API v1 must exist and function until API v1 is removed. Unlike whole resources, the v2 APIs may choose a different representation for the field, as long as it can be round-tripped. For example a v1 field named "magnitude" which was deprecated might be named "deprecatedMagnitude" in API v2. When v1 is eventually removed, the deprecated field can be removed from v2.

Enumerated or constant values

As with whole REST resources and fields thereof, a constant value which was supported in API v1 must exist and function until API v1 is removed.

Component config structures

Component configs are versioned and managed similar to REST resources.

Future work

Over time, Kubernetes will introduce more fine-grained API versions, at which point these rules will be adjusted as needed.

Deprecating a flag or CLI

The Kubernetes system is comprised of several different programs cooperating. Sometimes, a Kubernetes release might remove flags or CLI commands (collectively "CLI elements") in these programs. The individual programs naturally sort into two main groups - user-facing and admin-facing programs, which vary slightly in their deprecation policies. Unless a flag is explicitly prefixed or documented as "alpha" or "beta", it is considered GA.

CLI elements are effectively part of the API to the system, but since they are not versioned in the same way as the REST API, the rules for deprecation are as follows:

Rule #5a: CLI elements of user-facing components (e.g. kubectl) must function after their announced deprecation for no less than:

- **GA: 12 months or 2 releases (whichever is longer)**
- **Beta: 3 months or 1 release (whichever is longer)**
- **Alpha: 0 releases**

Rule #5b: CLI elements of admin-facing components (e.g. kubelet) must function after their announced deprecation for no less than:

- **GA: 6 months or 1 release (whichever is longer)**
- **Beta: 3 months or 1 release (whichever is longer)**
- **Alpha: 0 releases**

Rule #5c: Command line interface (CLI) elements cannot be deprecated in favor of less stable CLI elements

Similar to the Rule #3 for APIs, if an element of a command line interface is being replaced with an alternative implementation, such as by renaming an existing element, or by switching to use configuration sourced from a file instead of a command line argument, that recommended alternative must be of the same or higher stability level.

Rule #6: Deprecated CLI elements must emit warnings (optionally disable) when used.

Deprecating a feature or behavior

Occasionally a Kubernetes release needs to deprecate some feature or behavior of the system that is not controlled by the API or CLI. In this case, the rules for deprecation are as follows:

Rule #7: Deprecated behaviors must function for no less than 1 year after their announced deprecation.

If the feature or behavior is being replaced with an alternative implementation that requires work to adopt the change, there should be an effort to simplify the transition whenever possible. If an alternative implementation is under Kubernetes organization control, the following rules apply:

Rule #8: The feature of behavior must not be deprecated in favor of an alternative implementation that is less stable

For example, a generally available feature cannot be deprecated in favor of a Beta replacement. The Kubernetes project does, however, encourage users to adopt and transitions to alternative implementations even before they reach the same maturity level. This is particularly important for exploring new use cases of a feature or getting an early feedback on the replacement.

Alternative implementations may sometimes be external tools or products, for example a feature may move from the kubelet to container runtime that is not under Kubernetes project control. In such cases, the rule cannot be applied, but there must be an effort to ensure that there is a transition path that does not compromise on components' maturity levels. In the example with container runtimes, the effort may involve trying to ensure that popular container runtimes have versions that offer the same level of stability while implementing that replacement behavior.

Deprecation rules for features and behaviors do not imply that all changes to the system are governed by this policy. These rules apply only to significant, user-visible behaviors which impact the correctness of applications running on Kubernetes or that impact the administration of Kubernetes clusters, and which are being removed entirely.

An exception to the above rule is *feature gates*. Feature gates are key=value pairs that allow for users to enable/disable experimental features.

Feature gates are intended to cover the development life cycle of a feature - they are not intended to be long-term APIs. As such, they are expected to be deprecated and removed after a feature becomes GA or is dropped.

As a feature moves through the stages, the associated feature gate evolves. The feature life cycle matched to its corresponding feature gate is:

- Alpha: the feature gate is disabled by default and can be enabled by the user.
- Beta: the feature gate is enabled by default and can be disabled by the user.
- GA: the feature gate is deprecated (see "Deprecation") and becomes non-operational.
- GA, deprecation window complete: the feature gate is removed and calls to it are no longer accepted.

Deprecation

Features can be removed at any point in the life cycle prior to GA. When features are removed prior to GA, their associated feature gates are also deprecated.

When an invocation tries to disable a non-operational feature gate, the call fails in order to avoid unsupported scenarios that might otherwise run silently.

In some cases, removing pre-GA features requires considerable time. Feature gates can remain operational until their associated feature is fully removed, at which point the feature gate itself can be deprecated.

When removing a feature gate for a GA feature also requires considerable time, calls to feature gates may remain operational if the feature gate has no effect on the feature, and if the feature gate causes no errors.

Features intended to be disabled by users should include a mechanism for disabling the feature in the associated feature gate.

Versioning for feature gates is different from the previously discussed components, therefore the rules for deprecation are as follows:

Rule #9: Feature gates must be deprecated when the corresponding feature they control transitions a lifecycle stage as follows. Feature gates must function for no less than:

- **Beta feature to GA: 6 months or 2 releases (whichever is longer)**
- **Beta feature to EOL: 3 months or 1 release (whichever is longer)**
- **Alpha feature to EOL: 0 releases**

Rule #10: Deprecated feature gates must respond with a warning when used. When a feature gate is deprecated it must be documented in both in the release notes and the corresponding CLI help. Both warnings and documentation must indicate whether a feature gate is non-operational.

Deprecating a metric

Each component of the Kubernetes control-plane exposes metrics (usually the `/metrics` endpoint), which are typically ingested by cluster administrators. Not all metrics are the same: some metrics are commonly used as SLIs or used to determine SLOs, these tend to have greater import. Other metrics are more experimental in nature or are used primarily in the Kubernetes development process.

Accordingly, metrics fall under three stability classes (`ALPHA`, `BETA` `STABLE`); this impacts removal of a metric during a Kubernetes release. These classes are determined by the perceived importance of the metric. The rules for deprecating and removing a metric are as follows:

Rule #11a: Metrics, for the corresponding stability class, must function for no less than:

- **STABLE: 4 releases or 12 months (whichever is longer)**
- **BETA: 2 releases or 8 months (whichever is longer)**
- **ALPHA: 0 releases**

Rule #11b: Metrics, after their *announced deprecation*, must function for no less than:

- **STABLE: 3 releases or 9 months (whichever is longer)**
- **BETA: 1 releases or 4 months (whichever is longer)**
- **ALPHA: 0 releases**

Deprecated metrics will have their description text prefixed with a deprecation notice string '(Deprecated from x.y)' and a warning log will be emitted during metric registration. Like their stable undeprecated counterparts, deprecated metrics will be automatically registered to the metrics endpoint and therefore visible.

On a subsequent release (when the metric's `deprecatedVersion` is equal to *current_kubernetes_version - 3*), a deprecated metric will become a *hidden* metric. ***Unlike*** their deprecated counterparts, hidden metrics will *no longer* be automatically registered to the metrics endpoint (hence hidden). However, they can be explicitly enabled through a command line flag on the binary (`--show-hidden-metrics-for-version=`). This provides cluster admins an escape hatch to properly migrate off of a deprecated metric, if they were not able to react to the earlier deprecation warnings. Hidden metrics should be deleted after one release.

Exceptions

No policy can cover every possible situation. This policy is a living document, and will evolve over time. In practice, there will be situations that do not fit neatly into this policy, or for which this policy becomes a serious impediment. Such situations should be discussed with SIGs and project leaders to find the best solutions for those specific cases, always bearing in mind that Kubernetes is committed to being a stable system that, as much as possible, never breaks users. Exceptions will always be announced in all relevant release notes.

API Overview

This section provides reference information for the Kubernetes API.

The REST API is the fundamental fabric of Kubernetes. All operations and communications between components, and external user commands are REST API calls that the API Server handles. Consequently, everything in the Kubernetes platform is treated as an API object and has a corresponding entry in the API.

The Kubernetes API reference lists the API for Kubernetes version v1.34.

For general background information, read The Kubernetes API. Controlling Access to the Kubernetes API describes how clients can authenticate to the Kubernetes API server, and how their requests are authorized.

API versioning

The JSON and Protobuf serialization schemas follow the same guidelines for schema changes. The following descriptions cover both formats.

The API versioning and software versioning are indirectly related. The API and release versioning proposal describes the relationship between API versioning and software versioning.

Different API versions indicate different levels of stability and support. You can find more information about the criteria for each level in the API Changes documentation.

Here's a summary of each level:

- Alpha:
  - The version names contain `alpha` (for example, `v1alpha1`).
  - Built-in alpha API versions are disabled by default and must be explicitly enabled in the `kube-apiserver` configuration to be used.
  - The software may contain bugs. Enabling a feature may expose bugs.
  - Support for an alpha API may be dropped at any time without notice.
  - The API may change in incompatible ways in a later software release without notice.
  - The software is recommended for use only in short-lived testing clusters, due to increased risk of bugs and lack of long-term support.
- Beta:
  - The version names contain `beta` (for example, `v2beta3`).
  - Built-in beta API versions are disabled by default and must be explicitly enabled in the `kube-apiserver` configuration to be used (**except** for beta versions of APIs introduced prior to Kubernetes 1.22, which were enabled by default).
  - Built-in beta API versions have a maximum lifetime of 9 months or 3 minor releases (whichever is longer) from introduction to deprecation, and 9 months or 3 minor releases (whichever is longer) from deprecation to removal.
  - The software is well tested. Enabling a feature is considered safe.
  - The support for a feature will not be dropped, though the details may change.
  - The schema and/or semantics of objects may change in incompatible ways in a subsequent beta or stable API version. When this happens, migration instructions are provided. Adapting to a subsequent beta or stable API version may require editing or re-creating API objects, and may not be straightforward. The migration may require downtime for applications that rely on the feature.
  - The software is not recommended for production uses. Subsequent releases may introduce incompatible changes. Use of beta API versions is required to transition to subsequent beta or stable API versions once the beta API version is deprecated and no longer served.

  Note:

  Please try beta features and provide feedback. After the features exit beta, it may not be practical to make more changes.
- Stable:
  - The version name is `vX` where `X` is an integer.
  - Stable API versions remain available for all future releases within a Kubernetes major version, and there are no current plans for a major version revision of Kubernetes that removes stable APIs.

API groups

API groups make it easier to extend the Kubernetes API. The API group is specified in a REST path and in the `apiVersion` field of a serialized object.

There are several API groups in Kubernetes:

- The *core* (also called *legacy*) group is found at REST path `/api/v1`. The core group is not specified as part of the `apiVersion` field, for example, `apiVersion: v1`.
- The named groups are at REST path `/apis/$GROUP_NAME/$VERSION` and use `apiVersion: $GROUP_NAME/$VERSION` (for example, `apiVersion: batch/v1`). You can find the full list of supported API groups in Kubernetes API reference.

Enabling or disabling API groups

Certain resources and API groups are enabled by default. You can enable or disable them by setting `--runtime-config` on the API server. The `--runtime-config` flag accepts comma separated `<key>[=<value>]` pairs describing the runtime configuration of the API server. If the `=<value>` part is omitted, it is treated as if `=true` is specified. For example:

- to disable `batch/v1`, set `--runtime-config=batch/v1=false`
- to enable `batch/v2alpha1`, set `--runtime-config=batch/v2alpha1`

- to enable a specific version of an API, such as `storage.k8s.io/v1beta1/csistoragecapacities`, set `--runtime-config=storage.k8s.io/v1beta1/csistoragecapacities`

Note:

When you enable or disable groups or resources, you need to restart the API server and controller manager to pick up the `--runtime-config` changes.

Persistence

Kubernetes stores its serialized state in terms of the API resources by writing them into etcd.

What's next

- Learn more about API conventions
- Read the design documentation for aggregator
- Learn about Declarative API Validation.

Server-Side Apply

FEATURE STATE: `Kubernetes v1.22 [stable]` (enabled by default: true)

Kubernetes supports multiple appliers collaborating to manage the fields of a single object.

Server-Side Apply provides an optional mechanism for your cluster's control plane to track changes to an object's fields. At the level of a specific resource, Server-Side Apply records and tracks information about control over the fields of that object.

Server-Side Apply helps users and controllers manage their resources through declarative configuration. Clients can create and modify objects declaratively by submitting their *fully specified intent*.

A fully specified intent is a partial object that only includes the fields and values for which the user has an opinion. That intent either creates a new object (using default values for unspecified fields), or is combined, by the API server, with the existing object.

Comparison with Client-Side Apply explains how Server-Side Apply differs from the original, client-side `kubectl apply` implementation.

Field management

The Kubernetes API server tracks *managed fields* for all newly created objects.

When trying to apply an object, fields that have a different value and are owned by another manager will result in a conflict. This is done in order to signal that the operation might undo another collaborator's changes. Writes to objects with managed fields can be forced, in which case the value of any conflicted field will be overridden, and the ownership will be transferred.

Whenever a field's value does change, ownership moves from its current manager to the manager making the change.

Apply checks if there are any other field managers that also own the field. If the field is not owned by any other field managers, that field is set to its default value (if there is one), or otherwise is deleted from the object. The same rule applies to fields that are lists, associative lists, or maps.

For a user to manage a field, in the Server-Side Apply sense, means that the user relies on and expects the value of the field not to change. The user who last made an assertion about the value of a field will be recorded as the current field manager. This can be done by changing the field manager details explicitly using HTTP `POST` (**create**), `PUT` (**update**), or non-apply `PATCH` (**patch**). You can also declare and record a field manager by including a value for that field in a Server-Side Apply operation.

A Server-Side Apply **patch** request requires the client to provide its identity as a field manager. When using Server-Side Apply, trying to change a field that is controlled by a different manager results in a rejected request unless the client forces an override. For details of overrides, see Conflicts.

When two or more appliers set a field to the same value, they share ownership of that field. Any subsequent attempt to change the value of the shared field, by any of the appliers, results in a conflict. Shared field owners may give up ownership of a field by making a Server-Side Apply **patch** request that doesn't include that field.

Field management details are stored in a `managedFields` field that is part of an object's `metadata`.

If you remove a field from a manifest and apply that manifest, Server-Side Apply checks if there are any other field managers that also own the field. If the field is not owned by any other field managers, it is either deleted from the live object or reset to its default value, if it has one. The same rule applies to associative list or map items.

Compared to the (legacy) `kubectl.kubernetes.io/last-applied-configuration` annotation managed by `kubectl`, Server-Side Apply uses a more declarative approach, that tracks a user's (or client's) field management, rather than a user's last applied state. As a side effect of using Server-Side Apply, information about which field manager manages each field in an object also becomes available.

Example

A simple example of an object created using Server-Side Apply could look like this:

Note:

`kubectl get` omits managed fields by default. Add `--show-managed-fields` to show `managedFields` when the output format is either `json` or `yaml`.

```
---
apiVersion: v1kind: ConfigMapmetadata:  name: test-cm  namespace: default  labels:    test-label: test  managedFields:  - manager:
```

That example ConfigMap object contains a single field management record in `.metadata.managedFields`. The field management record consists of basic information about the managing entity itself, plus details about the fields being managed and the relevant operation (`Apply` or `Update`). If the request that last changed that field was a Server-Side Apply **patch** then the value of `operation` is `Apply`; otherwise, it is `Update`.

There is another possible outcome. A client could submit an invalid request body. If the fully specified intent does not produce a valid object, the request fails.

It is however possible to change `.metadata.managedFields` through an **update**, or through a **patch** operation that does not use Server-Side Apply. Doing so is highly discouraged, but might be a reasonable option to try if, for example, the `.metadata.managedFields` get into an inconsistent state (which should not happen in normal operations).

The format of `managedFields` is described in the Kubernetes API reference.

Caution:

The `.metadata.managedFields` field is managed by the API server. You should avoid updating it manually.

Conflicts

A *conflict* is a special status error that occurs when an `Apply` operation tries to change a field that another manager also claims to manage. This prevents an applier from unintentionally overwriting the value set by another user. When this occurs, the applier has 3 options to resolve the conflicts:

- **Overwrite value, become sole manager:** If overwriting the value was intentional (or if the applier is an automated process like a controller) the applier should set the `force` query parameter to true (for `kubectl apply`, you use the `--force-conflicts` command line parameter), and make the request again. This forces the operation to succeed, changes the value of the field, and removes the field from all other managers' entries in `managedFields`.
- **Don't overwrite value, give up management claim:** If the applier doesn't care about the value of the field any more, the applier can remove it from their local model of the resource, and make a new request with that particular field omitted. This leaves the value unchanged, and causes the field to be removed from the applier's entry in `managedFields`.
- **Don't overwrite value, become shared manager:** If the applier still cares about the value of a field, but doesn't want to overwrite it, they can change the value of that field in their local model of the resource so as to match the value of the object on the server, and then make a new request that takes into account that local update. Doing so leaves the value unchanged, and causes that field's management to be shared by the applier along with all other field managers that already claimed to manage it.

Field managers

Managers identify distinct workflows that are modifying the object (especially useful on conflicts!), and can be specified through the `fieldManager` query parameter as part of a modifying request. When you Apply to a resource, the `fieldManager` parameter is required. For other updates, the API server infers a field manager identity from the "User-Agent:" HTTP header (if present).

When you use the `kubectl` tool to perform a Server-Side Apply operation, `kubectl` sets the manager identity to `"kubectl"` by default.

Serialization

At the protocol level, Kubernetes represents Server-Side Apply message bodies as YAML, with the media type `application/apply-patch+yaml`.

Note:

Whether you are submitting JSON data or YAML data, use `application/apply-patch+yaml` as the `Content-Type` header value.

All JSON documents are valid YAML. However, Kubernetes has a bug where it uses a YAML parser that does not fully implement the YAML specification. Some JSON escapes may not be recognized.

The serialization is the same as for Kubernetes objects, with the exception that clients are not required to send a complete object.

Here's an example of a Server-Side Apply message body (fully specified intent):

```
{
  "apiVersion": "v1",
  "kind": "ConfigMap"
}
```

(this would make a no-change update, provided that it was sent as the body of a **patch** request to a valid `v1/configmaps` resource, and with the appropriate request `Content-Type`).

Operations in scope for field management

The Kubernetes API operations where field management is considered are:

1. Server-Side Apply (HTTP `PATCH`, with content type `application/apply-patch+yaml`)
2. Replacing an existing object (**update** to Kubernetes; `PUT` at the HTTP level)

Both operations update `.metadata.managedFields`, but behave a little differently.

Unless you specify a forced override, an apply operation that encounters field-level conflicts always fails; by contrast, if you make a change using **update** that would affect a managed field, a conflict never provokes failure of the operation.

All Server-Side Apply **patch** requests are required to identify themselves by providing a `fieldManager` query parameter, while the query parameter is optional for **update** operations. Finally, when using the `Apply` operation you cannot define `managedFields` in the body of the request that you submit.

An example object with multiple managers could look like this:

```
---
apiVersion: v1kind: ConfigMapmetadata:  name: test-cm  namespace: default  labels:    test-label: test  managedFields:  - manager:
```

In this example, a second operation was run as an **update** by the manager called `kube-controller-manager`. The update request succeeded and changed a value in the data field, which caused that field's management to change to the `kube-controller-manager`.

If this update has instead been attempted using Server-Side Apply, the request would have failed due to conflicting ownership.

Merge strategy

The merging strategy, implemented with Server-Side Apply, provides a generally more stable object lifecycle. Server-Side Apply tries to merge fields based on the actor who manages them instead of overruling based on values. This way multiple actors can update the same object without causing unexpected interference.

When a user sends a *fully-specified intent* object to the Server-Side Apply endpoint, the server merges it with the live object favoring the value from the request body if it is specified in both places. If the set of items present in the applied config is not a superset of the items applied by the same user last time, each missing item not managed by any other appliers is removed. For more information about how an object's schema is used to make decisions when merging, see sigs.k8s.io/structured-merge-diff.

The Kubernetes API (and the Go code that implements that API for Kubernetes) allows defining *merge strategy markers*. These markers describe the merge strategy supported for fields within Kubernetes objects. For a CustomResourceDefinition, you can set these markers when you define the custom resource.

Golang marker	OpenAPI extension	Possible values	Description
`//+listType`	`x-kubernetes-list-type`	`atomic`/`set`/`map`	Applicable to lists. `set` applies to lists that include only scalar elements. These elements must be unique. `map` applies to lists of nested types only. The key values (see `listMapKey`) must be unique in the list. `atomic` can apply to any list. If configured as `atomic`, the entire list is replaced during merge. At any point in time, a single manager owns the list. If `set` or `map`, different managers can manage entries separately.
`//+listMapKey`	`x-kubernetes-list-map-keys`	List of field names, e.g. `["port", "protocol"]`	Only applicable when `+listType=map`. A list of field names whose values uniquely identify entries in the list. While there can be multiple keys, `listMapKey` is singular because keys need to be specified individually in the Go type. The key fields must be scalars.
`//+mapType`	`x-kubernetes-map-type`	`atomic`/`granular`	Applicable to maps. `atomic` means that the map can only be entirely replaced by a single manager. `granular` means that the map supports separate managers updating individual fields.
`//+structType`	`x-kubernetes-map-type`	`atomic`/`granular`	Applicable to structs; otherwise same usage and OpenAPI annotation as `//+mapType`.

If `listType` is missing, the API server interprets a `patchStrategy=merge` marker as a `listType=map` and the corresponding `patchMergeKey` marker as a `listMapKey`.

The `atomic` list type is recursive.

(In the Go code for Kubernetes, these markers are specified as comments and code authors need not repeat them as field tags).

Custom resources and Server-Side Apply

By default, Server-Side Apply treats custom resources as unstructured data. All keys are treated the same as struct fields, and all lists are considered atomic.

If the CustomResourceDefinition defines a schema that contains annotations as defined in the previous Merge Strategy section, these annotations will be used when merging objects of this type.

Compatibility across topology changes

On rare occurrences, the author for a CustomResourceDefinition (CRD) or built-in may want to change the specific topology of a field in their resource, without incrementing its API version. Changing the topology of types, by upgrading the cluster or updating the CRD, has different consequences when updating existing objects. There are two categories of changes: when a field goes from `map`/`set`/`granular` to `atomic`, and the other way around.

When the `listType`, `mapType`, or `structType` changes from `map`/`set`/`granular` to `atomic`, the whole list, map, or struct of existing objects will end-up being owned by actors who owned an element of these types. This means that any further change to these objects would cause a conflict.

When a `listType`, `mapType`, or `structType` changes from `atomic` to `map`/`set`/`granular`, the API server is unable to infer the new ownership of these fields. Because of that, no conflict will be produced when objects have these fields updated. For that reason, it is not recommended to change a type from `atomic` to `map`/`set`/`granular`.

Take for example, the custom resource:

```
---
apiVersion: example.com/v1kind: Foometadata:  name: foo-sample  managedFields:  - manager: "manager-one"    operation: Apply    ap
```

Before `spec.data` gets changed from `atomic` to `granular`, `manager-one` owns the field `spec.data`, and all the fields within it (`key1` and `key2`). When the CRD gets changed to make `spec.data` `granular`, `manager-one` continues to own the top-level field `spec.data` (meaning no other managers can delete the map called `data` without a conflict), but it no longer owns `key1` and `key2`, so another manager can then modify or delete those fields without conflict.

Using Server-Side Apply in a controller

As a developer of a controller, you can use Server-Side Apply as a way to simplify the update logic of your controller. The main differences with a read-modify-write and/or patch are the following:

- the applied object must contain all the fields that the controller cares about.
- there is no way to remove fields that haven't been applied by the controller before (controller can still send a **patch** or **update** for these use-cases).
- the object doesn't have to be read beforehand; `resourceVersion` doesn't have to be specified.

It is strongly recommended for controllers to always force conflicts on objects that they own and manage, since they might not be able to resolve or act on these conflicts.

Transferring ownership

In addition to the concurrency controls provided by conflict resolution, Server-Side Apply provides ways to perform coordinated field ownership transfers from users to controllers.

This is best explained by example. Let's look at how to safely transfer ownership of the `replicas` field from a user to a controller while enabling automatic horizontal scaling for a Deployment, using the HorizontalPodAutoscaler resource and its accompanying controller.

Say a user has defined Deployment with `replicas` set to the desired value:

`application/ssa/nginx-deployment.yaml` Copy application/ssa/nginx-deployment.yaml to clipboard

```
apiVersion: apps/v1
kind: Deploymentmetadata:  name: nginx-deployment  labels:    app: nginxspec:  replicas: 3  selector:    matchLabels:      app: ng
```

And the user has created the Deployment using Server-Side Apply, like so:

```
kubectl apply -f https://k8s.io/examples/application/ssa/nginx-deployment.yaml --server-side
```

Then later, automatic scaling is enabled for the Deployment; for example:

```
kubectl autoscale deployment nginx-deployment --cpu-percent=50 --min=1 --max=10
```

Now, the user would like to remove `replicas` from their configuration, so they don't accidentally fight with the HorizontalPodAutoscaler (HPA) and its controller. However, there is a race: it might take some time before the HPA feels the need to adjust `.spec.replicas`; if the user removes `.spec.replicas` before the HPA writes to the field and becomes its owner, then the API server would set `.spec.replicas` to 1 (the default replica count for Deployment). This is not what the user wants to happen, even temporarily - it might well degrade a running workload.

There are two solutions:

- (basic) Leave `replicas` in the configuration; when the HPA eventually writes to that field, the system gives the user a conflict over it. At that point, it is safe to remove from the configuration.

- (more advanced) If, however, the user doesn't want to wait, for example because they want to keep the cluster legible to their colleagues, then they can take the following steps to make it safe to remove `replicas` from their configuration:

First, the user defines a new manifest containing only the `replicas` field:

```
# Save this file as 'nginx-deployment-replicas-only.yaml'.
apiVersion: apps/v1kind: Deploymentmetadata:  name: nginx-deploymentspec:  replicas: 3
```

Note:

The YAML file for SSA in this case only contains the fields you want to change. You are not supposed to provide a fully compliant Deployment manifest if you only want to modify the `spec.replicas` field using SSA.

The user applies that manifest using a private field manager name. In this example, the user picked `handover-to-hpa`:

```
kubectl apply -f nginx-deployment-replicas-only.yaml \
  --server-side --field-manager=handover-to-hpa \  --validate=false
```

If the apply results in a conflict with the HPA controller, then do nothing. The conflict indicates the controller has claimed the field earlier in the process than it sometimes does.

At this point the user may remove the `replicas` field from their manifest:

`application/ssa/nginx-deployment-no-replicas.yaml` Copy application/ssa/nginx-deployment-no-replicas.yaml to clipboard

```
apiVersion: apps/v1
kind: Deploymentmetadata:  name: nginx-deployment  labels:    app: nginxspec:  selector:    matchLabels:      app: nginx  template
```

Note that whenever the HPA controller sets the `replicas` field to a new value, the temporary field manager will no longer own any fields and will be automatically deleted. No further clean up is required.

Transferring ownership between managers

Field managers can transfer ownership of a field between each other by setting the field to the same value in both of their applied configurations, causing them to share ownership of the field. Once the managers share ownership of the field, one of them can remove the field from their applied configuration to give up ownership and complete the transfer to the other field manager.

Comparison with Client-Side Apply

Server-Side Apply is meant both as a replacement for the original client-side implementation of the `kubectl apply` subcommand, and as simple and effective mechanism for controllers to enact their changes.

Compared to the `last-applied` annotation managed by `kubectl`, Server-Side Apply uses a more declarative approach, which tracks an object's field management, rather than a user's last applied state. This means that as a side effect of using Server-Side Apply, information about which field manager manages each field in an object also becomes available.

A consequence of the conflict detection and resolution implemented by Server-Side Apply is that an applier always has up to date field values in their local state. If they don't, they get a conflict the next time they apply. Any of the three options to resolve conflicts results in the applied configuration being an up to date subset of the object on the server's fields.

This is different from Client-Side Apply, where outdated values which have been overwritten by other users are left in an applier's local config. These values only become accurate when the user updates that specific field, if ever, and an applier has no way of knowing whether their next apply will overwrite other users' changes.

Another difference is that an applier using Client-Side Apply is unable to change the API version they are using, but Server-Side Apply supports this use case.

Migration between client-side and server-side apply

Upgrading from client-side apply to server-side apply

Client-side apply users who manage a resource with `kubectl apply` can start using server-side apply with the following flag.

```
kubectl apply --server-side [--dry-run=server]
```

By default, field management of the object transfers from client-side apply to kubectl server-side apply, without encountering conflicts.

Caution:

Keep the `last-applied-configuration` annotation up to date. The annotation infers client-side applies managed fields. Any fields not managed by client-side apply raise conflicts.

For example, if you used `kubectl scale` to update the replicas field after client-side apply, then this field is not owned by client-side apply and creates conflicts on `kubectl apply --server-side`.

This behavior applies to server-side apply with the `kubectl` field manager. As an exception, you can opt-out of this behavior by specifying a different, non-default field manager, as seen in the following example. The default field manager for kubectl server-side apply is `kubectl`.

```
kubectl apply --server-side --field-manager=my-manager [--dry-run=server]
```

Downgrading from server-side apply to client-side apply

If you manage a resource with `kubectl apply --server-side`, you can downgrade to client-side apply directly with `kubectl apply`.

Downgrading works because kubectl Server-Side Apply keeps the `last-applied-configuration` annotation up-to-date if you use `kubectl apply`.

This behavior applies to Server-Side Apply with the `kubectl` field manager. As an exception, you can opt-out of this behavior by specifying a different, non-default field manager, as seen in the following example. The default field manager for kubectl server-side apply is `kubectl`.

```
kubectl apply --server-side --field-manager=my-manager [--dry-run=server]
```

API implementation

The `PATCH` verb (for an object that supports Server-Side Apply) accepts the unofficial `application/apply-patch+yaml` content type. Users of Server-Side Apply can send partially specified objects as YAML as the body of a `PATCH` request to the URI of a resource. When applying a configuration, you should always include all the fields that are important to the outcome (such as a desired state) that you want to define.

All JSON messages are valid YAML. Therefore, in addition to using YAML request bodies for Server-Side Apply requests, you can also use JSON request bodies, as they are also valid YAML. In either case, use the media type `application/apply-patch+yaml` for the HTTP request.

Access control and permissions

Since Server-Side Apply is a type of `PATCH`, a principal (such as a Role for Kubernetes RBAC) requires the **patch** permission to edit existing resources, and also needs the **create** verb permission in order to create new resources with Server-Side Apply.

Clearing `managedFields`

It is possible to strip all `managedFields` from an object by overwriting them using a **patch** (JSON Merge Patch, Strategic Merge Patch, JSON Patch), or through an **update** (HTTP `PUT`); in other words, through every write operation other than **apply**. This can be done by overwriting the `managedFields` field with an empty entry. Two examples are:

```
PATCH /api/v1/namespaces/default/configmaps/example-cm
Accept: application/json
Content-Type: application/merge-patch+json
{  "metadata": {    "managedFields": [      {}    ]  }}
```

```
PATCH /api/v1/namespaces/default/configmaps/example-cm
Accept: application/json
Content-Type: application/json-patch+json
If-Match: 1234567890123456789
```

```
[{"op": "replace", "path": "/metadata/managedFields", "value": [{}]}]
```

This will overwrite the `managedFields` with a list containing a single empty entry that then results in the `managedFields` being stripped entirely from the object. Note that setting the `managedFields` to an empty list will not reset the field. This is on purpose, so `managedFields` never get stripped by clients not aware of the field.

In cases where the reset operation is combined with changes to other fields than the `managedFields`, this will result in the `managedFields` being reset first and the other changes being processed afterwards. As a result the applier takes ownership of any fields updated in the same request.

Note:

Server-Side Apply does not correctly track ownership on sub-resources that don't receive the resource object type. If you are using Server-Side Apply with such a sub-resource, the changed fields may not be tracked.

What's next

You can read about `managedFields` within the Kubernetes API reference for the `metadata` top level field.

Client Libraries

This page contains an overview of the client libraries for using the Kubernetes API from various programming languages.

To write applications using the Kubernetes REST API, you do not need to implement the API calls and request/response types yourself. You can use a client library for the programming language you are using.

Client libraries often handle common tasks such as authentication for you. Most client libraries can discover and use the Kubernetes Service Account to authenticate if the API client is running inside the Kubernetes cluster, or can understand the kubeconfig file format to read the credentials and the API Server address.

Officially-supported Kubernetes client libraries

The following client libraries are officially maintained by Kubernetes SIG API Machinery.

Language	Client Library	Sample Programs
C	github.com/kubernetes-client/c	browse
dotnet	github.com/kubernetes-client/csharp	browse
Go	github.com/kubernetes/client-go/	browse
Haskell	github.com/kubernetes-client/haskell	browse
Java	github.com/kubernetes-client/java	browse
JavaScript	github.com/kubernetes-client/javascript	browse
Perl	github.com/kubernetes-client/perl/	browse
Python	github.com/kubernetes-client/python/	browse
Ruby	github.com/kubernetes-client/ruby/	browse

Community-maintained client libraries

Note: This section links to third party projects that provide functionality required by Kubernetes. The Kubernetes project authors aren't responsible for these projects, which are listed alphabetically. To add a project to this list, read the content guide before submitting a change. More information.

The following Kubernetes API client libraries are provided and maintained by their authors, not the Kubernetes team.

Language	Client Library
Clojure	github.com/yanatan16/clj-kubernetes-api
DotNet	github.com/tonnyeremin/kubernetes_gen
DotNet (RestSharp)	github.com/masroorhasan/Kubernetes.DotNet
Elixir	github.com/obmarg/kazan
Elixir	github.com/coryodaniel/k8s
Java (OSGi)	bitbucket.org/amdatulabs/amdatu-kubernetes
Java (Fabric8, OSGi)	github.com/fabric8io/kubernetes-client
Java	github.com/manusa/yakc
Lisp	github.com/brendandburns/cl-k8s
Lisp	github.com/xh4/cube
Node.js (TypeScript)	github.com/Goyoo/node-k8s-client
Node.js	github.com/ajpauwels/easy-k8s
Node.js	github.com/godaddy/kubernetes-client
Node.js	github.com/tenxcloud/node-kubernetes-client
Perl	metacpan.org/pod/Net::Kubernetes
PHP	github.com/allansun/kubernetes-php-client
PHP	github.com/maclof/kubernetes-client
PHP	github.com/travisghansen/kubernetes-client-php

Language	Client Library
PHP	github.com/renoki-co/php-k8s
Python	github.com/cloudcoil/cloudcoil
Python	github.com/fiaas/k8s
Python	github.com/gtsystem/lightkube
Python	github.com/kr8s-org/kr8s
Python	github.com/mnubo/kubernetes-py
Python	github.com/tomplus/kubernetes_asyncio
Python	github.com/Frankkkkk/pykorm
Ruby	github.com/abonas/kubeclient
Ruby	github.com/k8s-ruby/k8s-ruby
Ruby	github.com/kontena/k8s-client
Rust	github.com/kube-rs/kube
Rust	github.com/ynqa/kubernetes-rust
Scala	github.com/hagay3/skuber
Scala	github.com/hnaderi/scala-k8s
Scala	github.com/joan38/kubernetes-client
Swift	github.com/swiftkube/client

Kubernetes API health endpoints

The Kubernetes API server provides API endpoints to indicate the current status of the API server. This page describes these API endpoints and explains how you can use them.

API endpoints for health

The Kubernetes API server provides 3 API endpoints (`healthz`, `livez` and `readyz`) to indicate the current status of the API server. The `healthz` endpoint is deprecated (since Kubernetes v1.16), and you should use the more specific `livez` and `readyz` endpoints instead. The `livez` endpoint can be used with the `--livez-grace-period` flag to specify the startup duration. For a graceful shutdown you can specify the `--shutdown-delay-duration` flag with the `/readyz` endpoint. Machines that check the `healthz`/`livez`/`readyz` of the API server should rely on the HTTP status code. A status code `200` indicates the API server is `healthy`/`live`/`ready`, depending on the called endpoint. The more verbose options shown below are intended to be used by human operators to debug their cluster or understand the state of the API server.

The following examples will show how you can interact with the health API endpoints.

For all endpoints, you can use the `verbose` parameter to print out the checks and their status. This can be useful for a human operator to debug the current status of the API server, it is not intended to be consumed by a machine:

```
curl -k https://localhost:6443/livez?verbose
```

or from a remote host with authentication:

```
kubectl get --raw='/readyz?verbose'
```

The output will look like this:

```
[+]ping ok
[+]log ok
[+]etcd ok
[+]poststarthook/start-kube-apiserver-admission-initializer ok
[+]poststarthook/generic-apiserver-start-informers ok
[+]poststarthook/start-apiextensions-informers ok
[+]poststarthook/start-apiextensions-controllers ok
[+]poststarthook/crd-informer-synced ok
[+]poststarthook/bootstrap-controller ok
[+]poststarthook/rbac/bootstrap-roles ok
[+]poststarthook/scheduling/bootstrap-system-priority-classes ok
[+]poststarthook/start-cluster-authentication-info-controller ok
[+]poststarthook/start-kube-aggregator-informers ok
[+]poststarthook/apiservice-registration-controller ok
[+]poststarthook/apiservice-status-available-controller ok
[+]poststarthook/kube-apiserver-autoregistration ok
[+]autoregister-completion ok
[+]poststarthook/apiservice-openapi-controller ok
healthz check passed
```

The Kubernetes API server also supports to exclude specific checks. The query parameters can also be combined like in this example:

```
curl -k 'https://localhost:6443/readyz?verbose&exclude=etcd'
```

The output show that the `etcd` check is excluded:

```
[+]ping ok
[+]log ok
[+]etcd excluded: ok
[+]poststarthook/start-kube-apiserver-admission-initializer ok
[+]poststarthook/generic-apiserver-start-informers ok
[+]poststarthook/start-apiextensions-informers ok
[+]poststarthook/start-apiextensions-controllers ok
[+]poststarthook/crd-informer-synced ok
```

```
[+]poststarthook/bootstrap-controller ok
[+]poststarthook/rbac/bootstrap-roles ok
[+]poststarthook/scheduling/bootstrap-system-priority-classes ok
[+]poststarthook/start-cluster-authentication-info-controller ok
[+]poststarthook/start-kube-aggregator-informers ok
[+]poststarthook/apiservice-registration-controller ok
[+]poststarthook/apiservice-status-available-controller ok
[+]poststarthook/kube-apiserver-autoregistration ok
[+]autoregister-completion ok
[+]poststarthook/apiservice-openapi-controller ok
[+]shutdown ok
healthz check passed
```

Individual health checks

FEATURE STATE: `Kubernetes v1.34 [alpha]`

Each individual health check exposes an HTTP endpoint and can be checked individually. The schema for the individual health checks is `/livez/<healthcheck-name>` or `/readyz/<healthcheck-name>`, where `livez` and `readyz` can be used to indicate if you want to check the liveness or the readiness of the API server, respectively. The `<healthcheck-name>` path can be discovered using the `verbose` flag from above and take the path between `[+]` and `ok`. These individual health checks should not be consumed by machines but can be helpful for a human operator to debug a system:

```
curl -k https://localhost:6443/livez/etcd
```

Kubernetes API Concepts

The Kubernetes API is a resource-based (RESTful) programmatic interface provided via HTTP. It supports retrieving, creating, updating, and deleting primary resources via the standard HTTP verbs (POST, PUT, PATCH, DELETE, GET).

For some resources, the API includes additional subresources that allow fine-grained authorization (such as separate views for Pod details and log retrievals), and can accept and serve those resources in different representations for convenience or efficiency.

Kubernetes supports efficient change notifications on resources via *watches*:

in the Kubernetes API, watch is a verb that is used to track changes to an object in Kubernetes as a stream. It is used for the efficient detection of changes.

Kubernetes also provides consistent list operations so that API clients can effectively cache, track, and synchronize the state of resources.

You can view the API reference online, or read on to learn about the API in general.

Kubernetes API terminology

Kubernetes generally leverages common RESTful terminology to describe the API concepts:

- A *resource type* is the name used in the URL (`pods`, `namespaces`, `services`)
- All resource types have a concrete representation (their object schema) which is called a *kind*
- A list of instances of a resource type is known as a *collection*
- A single instance of a resource type is called a *resource*, and also usually represents an *object*
- For some resource types, the API includes one or more *sub-resources*, which are represented as URI paths below the resource

Most Kubernetes API resource types are objects – they represent a concrete instance of a concept on the cluster, like a pod or namespace. A smaller number of API resource types are *virtual* in that they often represent operations on objects, rather than objects, such as a permission check (use a POST with a JSON-encoded body of `SubjectAccessReview` to the `subjectaccessreviews` resource), or the `eviction` sub-resource of a Pod (used to trigger API-initiated eviction).

Object names

All objects you can create via the API have a unique object name to allow idempotent creation and retrieval, except that virtual resource types may not have unique names if they are not retrievable, or do not rely on idempotency. Within a namespace, only one object of a given kind can have a given name at a time. However, if you delete the object, you can make a new object with the same name. Some objects are not namespaced (for example: Nodes), and so their names must be unique across the whole cluster.

API verbs

Almost all object resource types support the standard HTTP verbs - GET, POST, PUT, PATCH, and DELETE. Kubernetes also uses its own verbs, which are often written in lowercase to distinguish them from HTTP verbs.

Kubernetes uses the term **list** to describe the action of returning a collection of resources, to distinguish it from retrieving a single resource which is usually called a **get**. If you sent an HTTP GET request with the `?watch` query parameter, Kubernetes calls this a **watch** and not a **get** (see Efficient detection of changes for more details).

For PUT requests, Kubernetes internally classifies these as either **create** or **update** based on the state of the existing object. An **update** is different from a **patch**; the HTTP verb for a **patch** is PATCH.

Resource URIs

All resource types are either scoped by the cluster (`/apis/GROUP/VERSION/*`) or to a namespace (`/apis/GROUP/VERSION/namespaces/NAMESPACE/*`). A namespace-scoped resource type will be deleted when its namespace is deleted and access to that resource type is controlled by authorization checks on the namespace scope.

Note: core resources use `/api` instead of `/apis` and omit the GROUP path segment.

Examples:

- `/api/v1/namespaces`
- `/api/v1/pods`
- `/api/v1/namespaces/my-namespace/pods`
- `/apis/apps/v1/deployments`
- `/apis/apps/v1/namespaces/my-namespace/deployments`
- `/apis/apps/v1/namespaces/my-namespace/deployments/my-deployment`

You can also access collections of resources (for example: listing all Nodes). The following paths are used to retrieve collections and resources:

- Cluster-scoped resources:
  - `GET /apis/GROUP/VERSION/RESOURCETYPE` - return the collection of resources of the resource type
  - `GET /apis/GROUP/VERSION/RESOURCETYPE/NAME` - return the resource with NAME under the resource type
- Namespace-scoped resources:
  - `GET /apis/GROUP/VERSION/RESOURCETYPE` - return the collection of all instances of the resource type across all namespaces
  - `GET /apis/GROUP/VERSION/namespaces/NAMESPACE/RESOURCETYPE` - return collection of all instances of the resource type in NAMESPACE
  - `GET /apis/GROUP/VERSION/namespaces/NAMESPACE/RESOURCETYPE/NAME` - return the instance of the resource type with NAME in NAMESPACE

Since a namespace is a cluster-scoped resource type, you can retrieve the list ("collection") of all namespaces with `GET /api/v1/namespaces` and details about a particular namespace with `GET /api/v1/namespaces/NAME`.

- Cluster-scoped subresource: `GET /apis/GROUP/VERSION/RESOURCETYPE/NAME/SUBRESOURCE`
- Namespace-scoped subresource: `GET /apis/GROUP/VERSION/namespaces/NAMESPACE/RESOURCETYPE/NAME/SUBRESOURCE`

The verbs supported for each subresource will differ depending on the object - see the API reference for more information. It is not possible to access sub-resources across multiple resources - generally a new virtual resource type would be used if that becomes necessary.

HTTP media types

Over HTTP, Kubernetes supports JSON, YAML, CBOR and Protobuf wire encodings.

By default, Kubernetes returns objects in JSON serialization, using the `application/json` media type. Although JSON is the default, clients may request a response in YAML, or use the more efficient binary Protobuf representation for better performance at scale.

The Kubernetes API implements standard HTTP content type negotiation: passing an `Accept` header with a `GET` call will request that the server tries to return a response in your preferred media type. If you want to send an object in Protobuf to the server for a `PUT` or `POST` request, you must set the `Content-Type` request header appropriately.

If you request an available media type, the API server returns a response with a suitable `Content-Type`; if none of the media types you request are supported, the API server returns a `406 Not acceptable` error message. All built-in resource types support the `application/json` media type.

Chunked encoding of collections

For JSON and Protobuf encoding, Kubernetes implements custom encoders that write item, by item. The feature doesn't change the output, but allows API server to avoid loading whole LIST response into memory. Using other types of encoding (including pretty representation of JSON) should be avoided for large collections of resources (>100MB) as it can have negative performance impact.

JSON resource encoding

The Kubernetes API defaults to using JSON for encoding HTTP message bodies.

For example:

1. List all of the pods on a cluster, without specifying a preferred format

   ```
   GET /api/v1/pods

   200 OK
   Content-Type: application/json

   … JSON encoded collection of Pods (PodList object)
   ```

2. Create a pod by sending JSON to the server, requesting a JSON response.

   ```
   POST /api/v1/namespaces/test/pods
   Content-Type: application/json
   Accept: application/json
   … JSON encoded Pod object

   200 OK
   Content-Type: application/json

   {
     "kind": "Pod",
     "apiVersion": "v1",
     …
   }
   ```

You can also request table and metadata-only representations of this encoding.

YAML resource encoding

Kubernetes also supports the `application/yaml` media type for both requests and responses. `YAML` can be used for defining Kubernetes manifests and API interactions.

For example:

1. List all of the pods on a cluster in YAML format

   ```
   GET /api/v1/pods
   Accept: application/yaml

   200 OK
   Content-Type: application/yaml

   … YAML encoded collection of Pods (PodList object)
   ```

2. Create a pod by sending YAML-encoded data to the server, requesting a YAML response:

   ```
   POST /api/v1/namespaces/test/pods
   Content-Type: application/yaml
   Accept: application/yaml
   … YAML encoded Pod object

   200 OK
   Content-Type: application/yaml

   apiVersion: v1
   kind: Pod
   metadata:
     name: my-pod
     …
   ```

You can also request table and metadata-only representations of this encoding.

Kubernetes Protobuf encoding

Kubernetes uses an envelope wrapper to encode Protobuf responses. That wrapper starts with a 4 byte magic number to help identify content in disk or in etcd as Protobuf (as opposed to JSON). The 4 byte magic number data is followed by a Protobuf encoded wrapper message, which describes the encoding and type of the underlying object. Within the Protobuf wrapper message, the inner object data is recorded using the `raw` field of Unknown (see the IDL for more detail).

For example:

1. List all of the pods on a cluster in Protobuf format.

   ```
   GET /api/v1/pods
   Accept: application/vnd.kubernetes.protobuf

   200 OK
   Content-Type: application/vnd.kubernetes.protobuf

   … binary encoded collection of Pods (PodList object)
   ```

2. Create a pod by sending Protobuf encoded data to the server, but request a response in JSON.

   ```
   POST /api/v1/namespaces/test/pods
   Content-Type: application/vnd.kubernetes.protobuf
   Accept: application/json
   … binary encoded Pod object

   200 OK
   Content-Type: application/json

   {
     "kind": "Pod",
     "apiVersion": "v1",
     ...
   }
   ```

You can use both techniques together and use Kubernetes' Protobuf encoding to interact with any API that supports it, for both reads and writes. Only some API resource types are compatible with Protobuf.

The wrapper format is:

```
A four byte magic number prefix:
  Bytes 0-3: "k8s\x00" [0x6b, 0x38, 0x73, 0x00]

An encoded Protobuf message with the following IDL:
  message Unknown {
    // typeMeta should have the string values for "kind" and "apiVersion" as set on the JSON object
    optional TypeMeta typeMeta = 1;

    // raw will hold the complete serialized object in protobuf. See the protobuf definitions in the client libraries for a given
    optional bytes raw = 2;

    // contentEncoding is encoding used for the raw data. Unspecified means no encoding.
    optional string contentEncoding = 3;
```

```
    // contentType is the serialization method used to serialize 'raw'. Unspecified means application/vnd.kubernetes.protobuf and
    // omitted.
    optional string contentType = 4;
  }

  message TypeMeta {
    // apiVersion is the group/version for this type
    optional string apiVersion = 1;
    // kind is the name of the object schema. A protobuf definition should exist for this object.
    optional string kind = 2;
  }
```

Note:

Clients that receive a response in `application/vnd.kubernetes.protobuf` that does not match the expected prefix should reject the response, as future versions may need to alter the serialization format in an incompatible way and will do so by changing the prefix.

You can also request table and metadata-only representations of this encoding.

Compatibility with Kubernetes Protobuf

Not all API resource types support Kubernetes' Protobuf encoding; specifically, Protobuf isn't available for resources that are defined as CustomResourceDefinitions or are served via the aggregation layer.

As a client, if you might need to work with extension types you should specify multiple content types in the request `Accept` header to support fallback to JSON. For example:

```
Accept: application/vnd.kubernetes.protobuf, application/json
```

CBOR resource encoding

FEATURE STATE: `Kubernetes v1.32 [alpha]` (enabled by default: false)

With the `CBORServingAndStorage` feature gate enabled, request and response bodies for all built-in resource types and all resources defined by a CustomResourceDefinition may be encoded to the CBOR binary data format. CBOR is also supported at the aggregation layer if it is enabled in individual aggregated API servers.

Clients should indicate the IANA media type `application/cbor` in the `Content-Type` HTTP request header when the request body contains a single CBOR encoded data item, and in the `Accept` HTTP request header when prepared to accept a CBOR encoded data item in the response. API servers will use `application/cbor` in the `Content-Type` HTTP response header when the response body contains a CBOR-encoded object.

If an API server encodes its response to a watch request using CBOR, the response body will be a CBOR Sequence and the `Content-Type` HTTP response header will use the IANA media type `application/cbor-seq`. Each entry of the sequence (if any) is a single CBOR-encoded watch event.

In addition to the existing `application/apply-patch+yaml` media type for YAML-encoded server-side apply configurations, API servers that enable CBOR will accept the `application/apply-patch+cbor` media type for CBOR-encoded server-side apply configurations. There is no supported CBOR equivalent for `application/json-patch+json` or `application/merge-patch+json`, or `application/strategic-merge-patch+json`.

You can also request table and metadata-only representations of this encoding.

Efficient detection of changes

The Kubernetes API allows clients to make an initial request for an object or a collection, and then to track changes since that initial request: a **watch**. Clients can send a **list** or a **get** and then make a follow-up **watch** request.

To make this change tracking possible, every Kubernetes object has a `resourceVersion` field representing the version of that resource as stored in the underlying persistence layer. When retrieving a collection of resources (either namespace or cluster scoped), the response from the API server contains a `resourceVersion` value. The client can use that `resourceVersion` to initiate a **watch** against the API server.

When you send a **watch** request, the API server responds with a stream of changes. These changes itemize the outcome of operations (such as **create**, **delete**, and **update**) that occurred after the `resourceVersion` you specified as a parameter to the **watch** request. The overall **watch** mechanism allows a client to fetch the current state and then subscribe to subsequent changes, without missing any events.

If a client **watch** is disconnected then that client can start a new **watch** from the last returned `resourceVersion`; the client could also perform a fresh **get** / **list** request and begin again. See Resource Version Semantics for more detail.

For example:

1. List all of the pods in a given namespace.

   ```
   GET /api/v1/namespaces/test/pods
   ---
   200 OK
   Content-Type: application/json

   {
     "kind": "PodList",
     "apiVersion": "v1",
     "metadata": {"resourceVersion":"10245"},
     "items": [...]
   }
   ```

2. Starting from resource version 10245, receive notifications of any API operations (such as **create**, **delete**, **patch** or **update**) that affect Pods in the *test* namespace. Each change notification is a JSON document. The HTTP response body (served as `application/json`) consists a series of JSON

documents.

```
GET /api/v1/namespaces/test/pods?watch=1&resourceVersion=10245
---
200 OK
Transfer-Encoding: chunked
Content-Type: application/json

{
  "type": "ADDED",
  "object": {"kind": "Pod", "apiVersion": "v1", "metadata": {"resourceVersion": "10596", ...}, ...}
}
{
  "type": "MODIFIED",
  "object": {"kind": "Pod", "apiVersion": "v1", "metadata": {"resourceVersion": "11020", ...}, ...}
}
...
```

A given Kubernetes server will only preserve a historical record of changes for a limited time. Clusters using etcd 3 preserve changes in the last 5 minutes by default. When the requested **watch** operations fail because the historical version of that resource is not available, clients must handle the case by recognizing the status code `410 Gone`, clearing their local cache, performing a new **get** or **list** operation, and starting the **watch** from the `resourceVersion` that was returned.

For subscribing to collections, Kubernetes client libraries typically offer some form of standard tool for this **list**-then-**watch** logic. (In the Go client library, this is called a `Reflector` and is located in the `k8s.io/client-go/tools/cache` package.)

Watch bookmarks

To mitigate the impact of short history window, the Kubernetes API provides a watch event named `BOOKMARK`. It is a special kind of event to mark that all changes up to a given `resourceVersion` the client is requesting have already been sent. The document representing the `BOOKMARK` event is of the type requested by the request, but only includes a `.metadata.resourceVersion` field. For example:

```
GET /api/v1/namespaces/test/pods?watch=1&resourceVersion=10245&allowWatchBookmarks=true
---
200 OK
Transfer-Encoding: chunked
Content-Type: application/json

{
  "type": "ADDED",
  "object": {"kind": "Pod", "apiVersion": "v1", "metadata": {"resourceVersion": "10596", ...}, ...}
}
...
{
  "type": "BOOKMARK",
  "object": {"kind": "Pod", "apiVersion": "v1", "metadata": {"resourceVersion": "12746"} }
}
```

As a client, you can request `BOOKMARK` events by setting the `allowWatchBookmarks=true` query parameter to a **watch** request, but you shouldn't assume bookmarks are returned at any specific interval, nor can clients assume that the API server will send any `BOOKMARK` event even when requested.

Streaming lists

FEATURE STATE: `Kubernetes v1.34 [beta]` (enabled by default: true)

On large clusters, retrieving the collection of some resource types may result in a significant increase of resource usage (primarily RAM) on the control plane. To alleviate the impact and simplify the user experience of the **list** + **watch** pattern, Kubernetes v1.32 promotes to beta the feature that allows requesting the initial state (previously requested via the **list** request) as part of the **watch** request.

On the client-side the initial state can be requested by specifying `sendInitialEvents=true` as query string parameter in a **watch** request. If set, the API server starts the watch stream with synthetic init events (of type `ADDED`) to build the whole state of all existing objects followed by a `BOOKMARK` event (if requested via `allowWatchBookmarks=true` option). The bookmark event includes the resource version to which is synced. After sending the bookmark event, the API server continues as for any other **watch** request.

When you set `sendInitialEvents=true` in the query string, Kubernetes also requires that you set `resourceVersionMatch` to `NotOlderThan` value. If you provided `resourceVersion` in the query string without providing a value or don't provide it at all, this is interpreted as a request for *consistent read*; the bookmark event is sent when the state is synced at least to the moment of a consistent read from when the request started to be processed. If you specify `resourceVersion` (in the query string), the bookmark event is sent when the state is synced at least to the provided resource version.

Example

An example: you want to watch a collection of Pods. For that collection, the current resource version is 10245 and there are two pods: `foo` and `bar`. Then sending the following request (explicitly requesting *consistent read* by setting empty resource version using `resourceVersion=`) could result in the following sequence of events:

```
GET /api/v1/namespaces/test/pods?watch=1&sendInitialEvents=true&allowWatchBookmarks=true&resourceVersion=&resourceVersionMatch=Not
---
200 OK
Transfer-Encoding: chunked
Content-Type: application/json

{
  "type": "ADDED",
  "object": {"kind": "Pod", "apiVersion": "v1", "metadata": {"resourceVersion": "8467", "name": "foo"}, ...}
}
{
  "type": "ADDED",
  "object": {"kind": "Pod", "apiVersion": "v1", "metadata": {"resourceVersion": "5726", "name": "bar"}, ...}
}
{
  "type": "BOOKMARK",
  "object": {"kind": "Pod", "apiVersion": "v1", "metadata": {"resourceVersion": "10245"} }
}
...
<followed by regular watch stream starting from resourceVersion="10245">
```

Response compression

FEATURE STATE: `Kubernetes v1.16 [beta]` (enabled by default: true)

`APIResponseCompression` is an option that allows the API server to compress the responses for **get** and **list** requests, reducing the network bandwidth and improving the performance of large-scale clusters. It is enabled by default since Kubernetes 1.16 and it can be disabled by including `APIResponseCompression=false` in the `--feature-gates` flag on the API server.

API response compression can significantly reduce the size of the response, especially for large resources or collections. For example, a **list** request for pods can return hundreds of kilobytes or even megabytes of data, depending on the number of pods and their attributes. By compressing the response, the network bandwidth can be saved and the latency can be reduced.

To verify if `APIResponseCompression` is working, you can send a **get** or **list** request to the API server with an `Accept-Encoding` header, and check the response size and headers. For example:

```
GET /api/v1/pods
Accept-Encoding: gzip
---
200 OK
Content-Type: application/json
content-encoding: gzip
...
```

The `content-encoding` header indicates that the response is compressed with `gzip`.

Retrieving large results sets in chunks

FEATURE STATE: `Kubernetes v1.29 [stable]` (enabled by default: true)

On large clusters, retrieving the collection of some resource types may result in very large responses that can impact the server and client. For instance, a cluster may have tens of thousands of Pods, each of which is equivalent to roughly 2 KiB of encoded JSON. Retrieving all pods across all namespaces may result in a very large response (10-20MB) and consume a large amount of server resources.

The Kubernetes API server supports the ability to break a single large collection request into many smaller chunks while preserving the consistency of the total request. Each chunk can be returned sequentially which reduces both the total size of the request and allows user-oriented clients to display results incrementally to improve responsiveness.

You can request that the API server handles a **list** by serving single collection using pages (which Kubernetes calls *chunks*). To retrieve a single collection in chunks, two query parameters `limit` and `continue` are supported on requests against collections, and a response field `continue` is returned from all **list** operations in the collection's `metadata` field. A client should specify the maximum results they wish to receive in each chunk with `limit` and the server will return up to `limit` resources in the result and include a `continue` value if there are more resources in the collection.

As an API client, you can then pass this `continue` value to the API server on the next request, to instruct the server to return the next page (*chunk*) of results. By continuing until the server returns an empty `continue` value, you can retrieve the entire collection.

Like a **watch** operation, a `continue` token will expire after a short amount of time (by default 5 minutes) and return a `410 Gone` if more results cannot be returned. In this case, the client will need to start from the beginning or omit the `limit` parameter.

For example, if there are 1,253 pods on the cluster and you want to receive chunks of 500 pods at a time, request those chunks as follows:

1. List all of the pods on a cluster, retrieving up to 500 pods each time.

   ```
   GET /api/v1/pods?limit=500
   ---
   200 OK
   Content-Type: application/json

   {
     "kind": "PodList",
     "apiVersion": "v1",
     "metadata": {
       "resourceVersion":"10245",
       "continue": "ENCODED_CONTINUE_TOKEN",
       "remainingItemCount": 753,
       ...
     },
     "items": [...] // returns pods 1-500
   }
   ```

2. Continue the previous call, retrieving the next set of 500 pods.

   ```
   GET /api/v1/pods?limit=500&continue=ENCODED_CONTINUE_TOKEN
   ---
   200 OK
   Content-Type: application/json

   {
     "kind": "PodList",
     "apiVersion": "v1",
   ```

```
  "metadata": {
    "resourceVersion":"10245",
    "continue": "ENCODED_CONTINUE_TOKEN_2",
    "remainingItemCount": 253,
    ...
  },
  "items": [...] // returns pods 501-1000
}
```

3. Continue the previous call, retrieving the last 253 pods.

```
GET /api/v1/pods?limit=500&continue=ENCODED_CONTINUE_TOKEN_2
---
200 OK
Content-Type: application/json

{
  "kind": "PodList",
  "apiVersion": "v1",
  "metadata": {
    "resourceVersion":"10245",
    "continue": "", // continue token is empty because we have reached the end of the list
    ...
  },
  "items": [...] // returns pods 1001-1253
}
```

Notice that the `resourceVersion` of the collection remains constant across each request, indicating the server is showing you a consistent snapshot of the pods. Pods that are created, updated, or deleted after version `10245` would not be shown unless you make a separate **list** request without the `continue` token. This allows you to break large requests into smaller chunks and then perform a **watch** operation on the full set without missing any updates.

`remainingItemCount` is the number of subsequent items in the collection that are not included in this response. If the **list** request contained label or field selectors then the number of remaining items is unknown and the API server does not include a `remainingItemCount` field in its response. If the **list** is complete (either because it is not chunking, or because this is the last chunk), then there are no more remaining items and the API server does not include a `remainingItemCount` field in its response. The intended use of the `remainingItemCount` is estimating the size of a collection.

Collections

In Kubernetes terminology, the response you get from a **list** is a *collection*. However, Kubernetes defines concrete kinds for collections of different types of resource. Collections have a kind named for the resource kind, with `List` appended.

When you query the API for a particular type, all items returned by that query are of that type. For example, when you **list** Services, the collection response has `kind` set to `ServiceList`; each item in that collection represents a single Service. For example:

```
GET /api/v1/services
```

```
{
  "kind": "ServiceList",
  "apiVersion": "v1",
  "metadata": {
    "resourceVersion": "2947301"
  },
  "items": [
    {
      "metadata": {
        "name": "kubernetes",
        "namespace": "default",
...      "metadata": {        "name": "kube-dns",        "namespace": "kube-system",...
```

There are dozens of collection types (such as `PodList`, `ServiceList`, and `NodeList`) defined in the Kubernetes API. You can get more information about each collection type from the Kubernetes API documentation.

Some tools, such as `kubectl`, represent the Kubernetes collection mechanism slightly differently from the Kubernetes API itself. Because the output of `kubectl` might include the response from multiple **list** operations at the API level, `kubectl` represents a list of items using `kind: List`. For example:

```
kubectl get services -A -o yaml
```

```
apiVersion: v1
kind: Listmetadata:  resourceVersion: ""  selfLink: ""items:- apiVersion: v1  kind: Service  metadata:    creationTimestamp: "2021
```

Note:

Keep in mind that the Kubernetes API does not have a `kind` named `List`.

`kind: List` is a client-side, internal implementation detail for processing collections that might be of different kinds of object. Avoid depending on `kind: List` in automation or other code.

Table fetches

When you run `kubectl get`, the default output format is a simple tabular representation of one or more instances of a particular resource type. In the past, clients were required to reproduce the tabular and describe output implemented in `kubectl` to perform simple lists of objects. A few limitations of that approach include non-trivial logic when dealing with certain objects. Additionally, types provided by API aggregation or third party resources are not known at compile time. This means that generic implementations had to be in place for types unrecognized by a client.

In order to avoid potential limitations as described above, clients may request the Table representation of objects, delegating specific details of printing to the server. The Kubernetes API implements standard HTTP content type negotiation: passing an `Accept` header containing a value of `application/json;as=Table;g=meta.k8s.io;v=v1` with a `GET` call will request that the server return objects in the Table content type.

For example, list all of the pods on a cluster in the Table format.

```
GET /api/v1/pods
Accept: application/json;as=Table;g=meta.k8s.io;v=v1
---
200 OK
Content-Type: application/json

{
    "kind": "Table",
    "apiVersion": "meta.k8s.io/v1",
    ...
    "columnDefinitions": [
        ...
    ]
}
```

For API resource types that do not have a custom Table definition known to the control plane, the API server returns a default Table response that consists of the resource's `name` and `creationTimestamp` fields.

```
GET /apis/crd.example.com/v1alpha1/namespaces/default/resources
---
200 OK
Content-Type: application/json
...

{
    "kind": "Table",
    "apiVersion": "meta.k8s.io/v1",
    ...
    "columnDefinitions": [
        {
            "name": "Name",
            "type": "string",
            ...
        },
        {
            "name": "Created At",
            "type": "date",
            ...
        }
    ]
}
```

Not all API resource types support a Table response; for example, a CustomResourceDefinitions might not define field-to-table mappings, and an APIService that extends the core Kubernetes API might not serve Table responses at all. If you are implementing a client that uses the Table information and must work against all resource types, including extensions, you should make requests that specify multiple content types in the `Accept` header. For example:

```
Accept: application/json;as=Table;g=meta.k8s.io;v=v1, application/json
```

If the client indicates it only accepts `...;as=Table;g=meta.k8s.io;v=v1`, servers that don't support table responses will return a 406 error code.

If falling back to full objects in that case is desired, clients can add `,application/json` (or any other supported encoding) to their Accept header, and handle either table or full objects in the response:

```
Accept: application/json;as=Table;g=meta.k8s.io;v=v1,application/json`
```

For more information on content type negotiation, see the MDN Content Negotiation.

Metadata-only fetches

To request partial object metadata, you can request metadata only responses in the `Accept` header. The Kubernetes API implements a variation on HTTP content type negotiation. As a client, you can provide an `Accept` header with the desired media type, along with parameters that indicate you want only metadata. For example: `Accept: application/json;as=PartialObjectMetadata;g=meta.k8s.io;v=v1` for JSON.

For example, to list all of the pods in a cluster, across all namespaces, but returning only the metadata for each pod:

```
GET /api/v1/pods
Accept: application/json;as=PartialObjectMetadata;g=meta.k8s.io;v=v1
---
200 OK
Content-Type: application/json

{
    "kind": "PartialObjectMetadataList",
    "apiVersion": "meta.k8s.io/v1",
    "metadata": {
        "resourceVersion": "...",
    },
    "items": [
        {
            "apiVersion": "meta.k8s.io/v1",
            "kind": "PartialObjectMetadata",
            "metadata": {
                "name": "pod-1",
                ...
            }
        },
        {
            "apiVersion": "meta.k8s.io/v1",
            "kind": "PartialObjectMetadata",
            "metadata": {
                "name": "pod-2",
                ...
            }
        }
    ]
}
```

For a request for a collection, the API server returns a PartialObjectMetadataList. For a request for a single object, the API server returns a PartialObjectMetadata representation of the object. In both cases, the returned objects only contain the `metadata` field. The `spec` and `status` fields are omitted.

This feature is useful for clients that only need to check for the existence of an object, or that only need to read its metadata. It can significantly reduce the size of the response from the API server.

You can request a metadata-only fetch for all available media types (JSON, YAML, CBOR and Kubernetes Protobuf). For Protobuf, the `Accept` header would be `application/vnd.kubernetes.protobuf;as=PartialObjectMetadata;g=meta.k8s.io;v=v1`.

The Kubernetes API server supports partial fetching for nearly all of its built-in APIs. However, you can use Kubernetes to access other API servers via the aggregation layer, and those APIs may not support partial fetches.

If a client uses the `Accept` header to **only** request a response `...;as=PartialObjectMetadata;g=meta.k8s.io;v=v1`, and accesses an API that doesn't support partial responses, Kubernetes responds with a 406 HTTP error.

If falling back to full objects in that case is desired, clients can add `,application/json` (or any other supported encoding) to their Accept header, and handle either PartialObjectMetadata or full objects in the response. It's a good idea to specify that a partial response is preferred, using the `q` (*quality*) parameter. For example:

```
Accept: application/json;as=PartialObjectMetadata;g=meta.k8s.io;v=v1, application/json;q=0.9
```

For more information on content type negotiation, see the MDN Content Negotiation.

Resource deletion

When you **delete** a resource this takes place in two phases.

1. *finalization*
2. removal

```
{
  "kind": "ConfigMap",
  "apiVersion": "v1",
  "metadata": {
    "finalizers": ["url.io/neat-finalization", "other-url.io/my-finalizer"],
    "deletionTimestamp": nil,
  }
}
```

When a client first sends a **delete** to request the removal of a resource, the `.metadata.deletionTimestamp` is set to the current time. Once the `.metadata.deletionTimestamp` is set, external controllers that act on finalizers may start performing their cleanup work at any time, in any order.

Order is **not** enforced between finalizers because it would introduce significant risk of stuck `.metadata.finalizers`.

The `.metadata.finalizers` field is shared: any actor with permission can reorder it. If the finalizer list were processed in order, then this might lead to a situation in which the component responsible for the first finalizer in the list is waiting for some signal (field value, external system, or other) produced by a component responsible for a finalizer later in the list, resulting in a deadlock.

Without enforced ordering, finalizers are free to order amongst themselves and are not vulnerable to ordering changes in the list.

Once the last finalizer is removed, the resource is actually removed from etcd.

Force deletion

FEATURE STATE: `Kubernetes v1.32 [alpha]` (enabled by default: false)

Caution:

This may break the workload associated with the resource being force deleted, if it relies on the normal deletion flow, so cluster breaking consequences may apply.

By enabling the delete option `ignoreStoreReadErrorWithClusterBreakingPotential`, the user can perform an unsafe force **delete** operation of an undecryptable/corrupt resource. This option is behind an ALPHA feature gate, and it is disabled by default. In order to use this option, the cluster operator must enable the feature by setting the command line option `--feature-gates=AllowUnsafeMalformedObjectDeletion=true`.

Note:

The user performing the force **delete** operation must have the privileges to do both the **delete** and **unsafe-delete-ignore-read-errors** verbs on the given resource.

A resource is considered corrupt if it can not be successfully retrieved from the storage due to:

- transformation error (for example: decryption failure), or
- the object failed to decode.

The API server first attempts a normal deletion, and if it fails with a *corrupt resource* error then it triggers the force delete. A force **delete** operation is unsafe because it ignores finalizer constraints, and skips precondition checks.

The default value for this option is `false`, this maintains backward compatibility. For a **delete** request with `ignoreStoreReadErrorWithClusterBreakingPotential` set to `true`, the fields `dryRun`, `gracePeriodSeconds`, `orphanDependents`, `preconditions`, and `propagationPolicy` must be left unset.

Note:

If the user issues a **delete** request with `ignoreStoreReadErrorWithClusterBreakingPotential` set to `true` on an otherwise readable resource, the API server aborts the request with an error.

Single resource API

The Kubernetes API verbs **get**, **create**, **update**, **patch**, **delete** and **proxy** support single resources only. These verbs with single resource support have no support for submitting multiple resources together in an ordered or unordered list or transaction.

When clients (including kubectl) act on a set of resources, the client makes a series of single-resource API requests, then aggregates the responses if needed.

By contrast, the Kubernetes API verbs **list** and **watch** allow getting multiple resources, and **deletecollection** allows deleting multiple resources.

Field validation

Kubernetes always validates the type of fields. For example, if a field in the API is defined as a number, you cannot set the field to a text value. If a field is defined as an array of strings, you can only provide an array. Some fields allow you to omit them, other fields are required. Omitting a required field from an API request is an error.

If you make a request with an extra field, one that the cluster's control plane does not recognize, then the behavior of the API server is more complicated.

By default, the API server drops fields that it does not recognize from an input that it receives (for example, the JSON body of a `PUT` request).

There are two situations where the API server drops fields that you supplied in an HTTP request.

These situations are:

1. The field is unrecognized because it is not in the resource's OpenAPI schema. (One exception to this is for CRDs that explicitly choose not to prune unknown fields via `x-kubernetes-preserve-unknown-fields`).
2. The field is duplicated in the object.

Validation for unrecognized or duplicate fields

FEATURE STATE: `Kubernetes v1.27 [stable]` (enabled by default: true)

From 1.25 onward, unrecognized or duplicate fields in an object are detected via validation on the server when you use HTTP verbs that can submit data (`POST`, `PUT`, and `PATCH`). Possible levels of validation are `Ignore`, `Warn` (default), and `Strict`.

`Ignore`
: The API server succeeds in handling the request as it would without the erroneous fields being set, dropping all unknown and duplicate fields and giving no indication it has done so.

`Warn`
: (Default) The API server succeeds in handling the request, and reports a warning to the client. The warning is sent using the `Warning:` response header, adding one warning item for each unknown or duplicate field. For more information about warnings and the Kubernetes API, see the blog article Warning: Helpful Warnings Ahead.

`Strict`
: The API server rejects the request with a 400 Bad Request error when it detects any unknown or duplicate fields. The response message from the API server specifies all the unknown or duplicate fields that the API server has detected.

The field validation level is set by the `fieldValidation` query parameter.

Note:

If you submit a request that specifies an unrecognized field, and that is also invalid for a different reason (for example, the request provides a string value where the API expects an integer for a known field), then the API server responds with a 400 Bad Request error, but will not provide any information on unknown or duplicate fields (only which fatal error it encountered first).

You always receive an error response in this case, no matter what field validation level you requested.

Tools that submit requests to the server (such as `kubectl`), might set their own defaults that are different from the `Warn` validation level that the API server uses by default.

The `kubectl` tool uses the `--validate` flag to set the level of field validation. It accepts the values `ignore`, `warn`, and `strict` while also accepting the values `true` (equivalent to `strict`) and `false` (equivalent to `ignore`). The default validation setting for kubectl is `--validate=true`, which means strict server-side field validation.

When kubectl cannot connect to an API server with field validation (API servers prior to Kubernetes 1.27), it will fall back to using client-side validation. Client-side validation will be removed entirely in a future version of kubectl.

Note:

Prior to Kubernetes 1.25, `kubectl --validate` was used to toggle client-side validation on or off as a boolean flag.

Starting from v1.33, Kubernetes (including v1.34) offers a way to define field validations using *declarative tags*. This is useful for people contributing to Kubernetes itself, and it's also relevant if you're writing your own API using Kubernetes libraries. To learn more, see Declarative API Validation.

Dry-run

FEATURE STATE: `Kubernetes v1.19 [stable]` (enabled by default: true)

When you use HTTP verbs that can modify resources (`POST`, `PUT`, `PATCH`, and `DELETE`), you can submit your request in a *dry run* mode. Dry run mode helps to evaluate a request through the typical request stages (admission chain, validation, merge conflicts) up until persisting objects to storage. The response body for the request is as close as possible to a non-dry-run response. Kubernetes guarantees that dry-run requests will not be persisted in storage or have any other side effects.

Make a dry-run request

Dry-run is triggered by setting the `dryRun` query parameter. This parameter is a string, working as an enum, and the only accepted values are:

[no value set]
: Allow side effects. You request this with a query string such as `?dryRun` or `?dryRun&pretty=true`. The response is the final object that would have been persisted, or an error if the request could not be fulfilled.

`All`
: Every stage runs as normal, except for the final storage stage where side effects are prevented.

When you set `?dryRun=All`, any relevant admission controllers are run, validating admission controllers check the request post-mutation, merge is performed on `PATCH`, fields are defaulted, and schema validation occurs. The changes are not persisted to the underlying storage, but the final object which would have been persisted is still returned to the user, along with the normal status code.

If the non-dry-run version of a request would trigger an admission controller that has side effects, the request will be failed rather than risk an unwanted side effect. All built in admission control plugins support dry-run. Additionally, admission webhooks can declare in their configuration object that they do not have side effects, by setting their `sideEffects` field to `None`.

Note:

If a webhook actually does have side effects, then the `sideEffects` field should be set to "NoneOnDryRun". That change is appropriate provided that the webhook is also be modified to understand the `DryRun` field in AdmissionReview, and to prevent side effects on any request marked as dry runs.

Here is an example dry-run request that uses `?dryRun=All`:

```
POST /api/v1/namespaces/test/pods?dryRun=All
Content-Type: application/json
Accept: application/json
```

The response would look the same as for non-dry-run request, but the values of some generated fields may differ.

Generated values

Some values of an object are typically generated before the object is persisted. It is important not to rely upon the values of these fields set by a dry-run request, since these values will likely be different in dry-run mode from when the real request is made. Some of these fields are:

- `name`: if `generateName` is set, `name` will have a unique random name
- `creationTimestamp` / `deletionTimestamp`: records the time of creation/deletion
- `UID`: uniquely identifies the object and is randomly generated (non-deterministic)
- `resourceVersion`: tracks the persisted version of the object
- Any field set by a mutating admission controller
- For the `Service` resource: Ports or IP addresses that the kube-apiserver assigns to Service objects

Dry-run authorization

Authorization for dry-run and non-dry-run requests is identical. Thus, to make a dry-run request, you must be authorized to make the non-dry-run request.

For example, to run a dry-run **patch** for a Deployment, you must be authorized to perform that **patch**. Here is an example of a rule for Kubernetes RBAC that allows patching Deployments:

```yaml
rules:
- apiGroups: ["apps"]  resources: ["deployments"]  verbs: ["patch"]
```

See Authorization Overview.

Updates to existing resources

Kubernetes provides several ways to update existing objects. You can read choosing an update mechanism to learn about which approach might be best for your use case.

You can overwrite (**update**) an existing resource - for example, a ConfigMap - using an HTTP PUT. For a PUT request, it is the client's responsibility to specify the `resourceVersion` (taking this from the object being updated). Kubernetes uses that `resourceVersion` information so that the API server can detect lost updates and reject requests made by a client that is out of date with the cluster. In the event that the resource has changed (the `resourceVersion` the client provided is stale), the API server returns a `409 Conflict` error response.

Instead of sending a PUT request, the client can send an instruction to the API server to **patch** an existing resource. A **patch** is typically appropriate if the change that the client wants to make isn't conditional on the existing data. Clients that need effective detection of lost updates should consider making their request conditional on the existing `resourceVersion` (either HTTP PUT or HTTP PATCH), and then handle any retries that are needed in case there is a conflict.

The Kubernetes API supports four different PATCH operations, determined by their corresponding HTTP `Content-Type` header:

`application/apply-patch+yaml`
: Server Side Apply YAML (a Kubernetes-specific extension, based on YAML). All JSON documents are valid YAML, so you can also submit JSON using this media type. See Server Side Apply serialization for more details. To Kubernetes, this is a **create** operation if the object does not exist, or a **patch** operation if the object already exists.

`application/json-patch+json`
: JSON Patch, as defined in RFC6902. A JSON patch is a sequence of operations that are executed on the resource; for example `{"op": "add", "path": "/a/b/c", "value": [ "foo", "bar" ]}`. To Kubernetes, this is a **patch** operation.

  A **patch** using `application/json-patch+json` can include conditions to validate consistency, allowing the operation to fail if those conditions are not met (for example, to avoid a lost update).

`application/merge-patch+json`
: JSON Merge Patch, as defined in RFC7386. A JSON Merge Patch is essentially a partial representation of the resource. The submitted JSON is combined with the current resource to create a new one, then the new one is saved. To Kubernetes, this is a **patch** operation.

`application/strategic-merge-patch+json`
: Strategic Merge Patch (a Kubernetes-specific extension based on JSON). Strategic Merge Patch is a custom implementation of JSON Merge Patch. You can only use Strategic Merge Patch with built-in APIs, or with aggregated API servers that have special support for it. You cannot use `application/strategic-merge-patch+json` with any API defined using a CustomResourceDefinition.

  Note:

  The Kubernetes *server side apply* mechanism has superseded Strategic Merge Patch.

Kubernetes' Server Side Apply feature allows the control plane to track managed fields for newly created objects. Server Side Apply provides a clear pattern for managing field conflicts, offers server-side **apply** and **update** operations, and replaces the client-side functionality of `kubectl apply`.

For Server-Side Apply, Kubernetes treats the request as a **create** if the object does not yet exist, and a **patch** otherwise. For other requests that use PATCH at the HTTP level, the logical Kubernetes operation is always **patch**.

See Server Side Apply for more details.

Choosing an update mechanism

HTTP PUT to replace existing resource

The **update** (HTTP `PUT`) operation is simple to implement and flexible, but has drawbacks:

- You need to handle conflicts where the `resourceVersion` of the object changes between your client reading it and trying to write it back. Kubernetes always detects the conflict, but you as the client author need to implement retries.
- You might accidentally drop fields if you decode an object locally (for example, using client-go, you could receive fields that your client does not know how to handle - and then drop them as part of your update.
- If there's a lot of contention on the object (even on a field, or set of fields, that you're not trying to edit), you might have trouble sending the update. The problem is worse for larger objects and for objects with many fields.

HTTP PATCH using JSON Patch

A **patch** update is helpful, because:

- As you're only sending differences, you have less data to send in the `PATCH` request.
- You can make changes that rely on existing values, such as copying the value of a particular field into an annotation.
- Unlike with an **update** (HTTP `PUT`), making your change can happen right away even if there are frequent changes to unrelated fields): you usually would not need to retry.
  - You might still need to specify the `resourceVersion` (to match an existing object) if you want to be extra careful to avoid lost updates
  - It's still good practice to write in some retry logic in case of errors.
- You can use test conditions to careful craft specific update conditions. For example, you can increment a counter without reading it if the existing value matches what you expect. You can do this with no lost update risk, even if the object has changed in other ways since you last wrote to it. (If the test condition fails, you can fall back to reading the current value and then write back the changed number).

However:

- You need more local (client) logic to build the patch; it helps a lot if you have a library implementation of JSON Patch, or even for making a JSON Patch specifically against Kubernetes.
- As the author of client software, you need to be careful when building the patch (the HTTP request body) not to drop fields (the order of operations matters).

HTTP PATCH using Server-Side Apply

Server-Side Apply has some clear benefits:

- A single round trip: it rarely requires making a `GET` request first.
  - and you can still detect conflicts for unexpected changes
  - you have the option to force override a conflict, if appropriate
- Client implementations are easy to make.
- You get an atomic create-or-update operation without extra effort (similar to `UPSERT` in some SQL dialects).

However:

- Server-Side Apply does not work at all for field changes that depend on a current value of the object.
- You can only apply updates to objects. Some resources in the Kubernetes HTTP API are not objects (they do not have a `.metadata` field), and Server-Side Apply is only relevant for Kubernetes objects.

Resource versions

Resource versions are strings that identify the server's internal version of an object. Resource versions can be used by clients to determine when objects have changed, or to express data consistency requirements when getting, listing and watching resources. Resource versions must be treated as opaque by clients and passed unmodified back to the server.

You must not assume resource versions are numeric or collatable. API clients may only compare two resource versions for equality (this means that you must not compare resource versions for greater-than or less-than relationships).

`resourceVersion` fields in metadata

Clients find resource versions in resources, including the resources from the response stream for a **watch**, or when using **list** to enumerate resources.

v1.meta/ObjectMeta - The `metadata.resourceVersion` of a resource instance identifies the resource version the instance was last modified at.

v1.meta/ListMeta - The `metadata.resourceVersion` of a resource collection (the response to a **list**) identifies the resource version at which the collection was constructed.

`resourceVersion` parameters in query strings

The **get**, **list**, and **watch** operations support the `resourceVersion` parameter. From version v1.19, Kubernetes API servers also support the `resourceVersionMatch` parameter on *list* requests.

The API server interprets the `resourceVersion` parameter differently depending on the operation you request, and on the value of `resourceVersion`. If you set `resourceVersionMatch` then this also affects the way matching happens.

Semantics for get and list

For **get** and **list**, the semantics of `resourceVersion` are:

get:

resourceVersion unset	resourceVersion="0"	resourceVersion="{value other than 0}"
Most Recent	Any	Not older than

list:

From version v1.19, Kubernetes API servers support the `resourceVersionMatch` parameter on *list* requests. If you set both `resourceVersion` and `resourceVersionMatch`, the `resourceVersionMatch` parameter determines how the API server interprets `resourceVersion`.

You should always set the `resourceVersionMatch` parameter when setting `resourceVersion` on a **list** request. However, be prepared to handle the case where the API server that responds is unaware of `resourceVersionMatch` and ignores it.

Unless you have strong consistency requirements, using `resourceVersionMatch=NotOlderThan` and a known `resourceVersion` is preferable since it can achieve better performance and scalability of your cluster than leaving `resourceVersion` and `resourceVersionMatch` unset, which requires quorum read to be served.

Setting the `resourceVersionMatch` parameter without setting `resourceVersion` is not valid.

This table explains the behavior of **list** requests with various combinations of `resourceVersion` and `resourceVersionMatch`:

resourceVersionMatch param	paging params	resourceVersion not set	resourceVersion="0"	resourceVersion="{value other than 0}"
unset	*limit unset*	Most Recent	Any	Not older than
unset	limit=<n>, *continue unset*	Most Recent	Any	Exact
unset	limit=<n>, continue=<token>	Continuation	Continuation	Invalid, HTTP `400 Bad Request`
`resourceVersionMatch=Exact`	*limit unset*	Invalid	Invalid	Exact
`resourceVersionMatch=Exact`	limit=<n>, *continue unset*	Invalid	Invalid	Exact
`resourceVersionMatch=NotOlderThan`	*limit unset*	Invalid	Any	Not older than
`resourceVersionMatch=NotOlderThan`	limit=<n>, *continue unset*	Invalid	Any	Not older than

Note:

If your cluster's API server does not honor the `resourceVersionMatch` parameter, the behavior is the same as if you did not set it.

The meaning of the **get** and **list** semantics are:

Any
: Return data at any resource version. The newest available resource version is preferred, but strong consistency is not required; data at any resource version may be served. It is possible for the request to return data at a much older resource version that the client has previously observed, particularly in high availability configurations, due to partitions or stale caches. Clients that cannot tolerate this should not use this semantic. Always served from *watch cache*, improving performance and reducing etcd load.

Most recent

Return data at the most recent resource version. The returned data must be consistent (in detail: served from etcd via a quorum read). For etcd v3.4.31+ and v3.5.13+, Kubernetes 1.34 serves “most recent” reads from the *watch cache*: an internal, in-memory store within the API server that caches and mirrors the state of data persisted into etcd. Kubernetes requests progress notification to maintain cache consistency against the etcd persistence layer. Kubernetes v1.28 through to v1.30 also supported this feature, although as Alpha it was not recommended for production nor enabled by default until the v1.31 release.

Not older than
: Return data at least as new as the provided `resourceVersion`. The newest available data is preferred, but any data not older than the provided `resourceVersion` may be served. For **list** requests to servers that honor the `resourceVersionMatch` parameter, this guarantees that the collection's `.metadata.resourceVersion` is not older than the requested `resourceVersion`, but does not make any guarantee about the `.metadata.resourceVersion` of any of the items in that collection. Always served from *watch cache*, improving performance and reducing etcd load.

Exact
: Return data at the exact resource version provided. If the provided `resourceVersion` is unavailable, the server responds with HTTP `410 Gone`. For **list** requests to servers that honor the `resourceVersionMatch` parameter, this guarantees that the collection's `.metadata.resourceVersion` is the same as the `resourceVersion` you requested in the query string. That guarantee does not apply to the `.metadata.resourceVersion` of any items within that collection. With the `ListFromCacheSnapshot` feature gate enabled by default, API server will attempt to serve the response from snapshots if one is available with `resourceVersion` older than requested. This improves performance and reduces etcd load. API server starts with no snapshots, creates a new snapshot on every watch event and keeps them until it detects etcd is compacted or if cache is full with events older than 75 seconds. If the provided `resourceVersion` is unavailable, the server will fallback to etcd.

Continuation
: Return the next page of data for a paginated list request, ensuring consistency with the exact `resourceVersion` established by the initial request in the sequence. Response to **list** requests with limit include *continue token*, that encodes the `resourceVersion` and last observed position from which to resume the list. If the `resourceVersion` in the provided *continue token* is unavailable, the server responds with HTTP `410 Gone`. With the `ListFromCacheSnapshot` feature gate enabled by default, API server will attempt to serve the response from snapshots if one is available with `resourceVersion` older than requested. This improves performance and reduces etcd load. API server starts with no snapshots, creates a new snapshot on every watch event and keeps them until it detects etcd is compacted or if cache is full with events older than 75 seconds. If the `resourceVersion` in provided *continue token* is unavailable, the server will fallback to etcd.

Note:

When you **list** resources and receive a collection response, the response includes the list metadata of the collection as well as object metadata for each item in that collection. For individual objects found within a collection response, `.metadata.resourceVersion` tracks when that object was last updated, and not how up-to-date the object is when served.

When using `resourceVersionMatch=NotOlderThan` and limit is set, clients must handle HTTP `410 Gone` responses. For example, the client might retry with a newer `resourceVersion` or fall back to `resourceVersion=""`.

When using `resourceVersionMatch=Exact` and `limit` is unset, clients must verify that the collection's `.metadata.resourceVersion` matches the requested `resourceVersion`, and handle the case where it does not. For example, the client might fall back to a request with `limit` set.

Semantics for watch

For **watch**, the semantics of resource version are:

watch:

resourceVersion unset	resourceVersion="0"	resourceVersion="{value other than 0}"
Get State and Start at Most Recent	Get State and Start at Any	Start at Exact

The meaning of those **watch** semantics are:

Get State and Start at Any
: Start a **watch** at any resource version; the most recent resource version available is preferred, but not required. Any starting resource version is allowed. It is possible for the **watch** to start at a much older resource version that the client has previously observed, particularly in high availability configurations, due to partitions or stale caches. Clients that cannot tolerate this apparent rewinding should not start a **watch** with this semantic. To establish initial state, the **watch** begins with synthetic "Added" events for all resource instances that exist at the starting resource version. All following watch events are for all changes that occurred after the resource version the **watch** started at.

  Caution:

  watches initialized this way may return arbitrarily stale data. Please review this semantic before using it, and favor the other semantics where possible.

Get State and Start at Most Recent
: Start a **watch** at the most recent resource version, which must be consistent (in detail: served from etcd via a quorum read). To establish initial state, the **watch** begins with synthetic "Added" events of all resources instances that exist at the starting resource version. All following watch events are for all changes that occurred after the resource version the **watch** started at.

Start at Exact
: Start a **watch** at an exact resource version. The watch events are for all changes after the provided resource version. Unlike "Get State and Start at Most Recent" and "Get State and Start at Any", the **watch** is not started with synthetic "Added" events for the provided resource version. The client is assumed to already have the initial state at the starting resource version since the client provided the resource version.

"410 Gone" responses

Servers are not required to serve all older resource versions and may return a HTTP `410 (Gone)` status code if a client requests a `resourceVersion` older than the server has retained. Clients must be able to tolerate `410 (Gone)` responses. See Efficient detection of changes for details on how to handle `410 (Gone)` responses when watching resources.

If you request a `resourceVersion` outside the applicable limit then, depending on whether a request is served from cache or not, the API server may reply with a `410 Gone` HTTP response.

Unavailable resource versions

Servers are not required to serve unrecognized resource versions. If you request **list** or **get** for a resource version that the API server does not recognize, then the API server may either:

- wait briefly for the resource version to become available, then timeout with a `504 (Gateway Timeout)` if the provided resource versions does not become available in a reasonable amount of time;
- respond with a `Retry-After` response header indicating how many seconds a client should wait before retrying the request.

If you request a resource version that an API server does not recognize, the kube-apiserver additionally identifies its error responses with a message `Too large resource version`.

If you make a **watch** request for an unrecognized resource version, the API server may wait indefinitely (until the request timeout) for the resource version to become available.