Defaulting and Pruning for Custom Resources

Authors:
- Dr. Stefan Schimanski, @sttts

Approvers:
- ?David Eads, @deads2k?
- ?Daniel Smith, @lavalamp? – substituted by ?Eric Tune @erictune? while Daniel is out

Top-level approver sponsor:
- ?Clayton Coleman, @smarterclayton?

Last Updated: 20th of July, 2018
Status: KEP Draft
Owning SIGs: @sig-api-machinery
Participating SIGs: @sig-architecture

Table of Contents

Table of Contents

Overview

Goals

Non-Goals

Motivation
- Pruning
- Defaulting
- Mixing of Schema and Value Validation

Formal Proposal – following pruning option 1
- Types and Formats
- Polymorphic Fields
- Defaulting

Excluding values from Pruning

Ratcheting (slightly related)

References

[OLD] Alternative: following pruning option 4

Overview

Native Golang based resources do not persist JSON fields which are not part of the Golang structs which are backing them in the API server memory. This artifact of the use of typed Golang structs inside of the REST implementation has turned into an API convention. Major parts of the Kubernetes depend on this to ensure consistency of data persisted to etcd and returned from the REST API.

Without consistency of data in etcd, objects can suddenly render unaccessible on version upgrade because unexpected data may break decoding. Even if persisted data has correct format and decodes correctly, having it not gone through validation and admission when it was stored can break cluster-wide invariants. For example assume the `privileged: true/false` field is added to a type in Kubernetes version X+1. In version X, there is no security check around this. So every user could set that flag if we didn't drop unknown fields. When the field is added in X+1, that user suddenly has escalated access (note: on read we do not run admission). This is a serious security risk.

CustomResources are persisted as JSON blobs today (with the exception of `ObjectMeta` which is pruned since Kubernetes 1.11), i.e. we do not drop unknown fields, with the described consequences. This proposal is about adding a decoding step named "pruning" into the decoder from JSON to `unstructured.Unstructured` inside the apiextensions-apiserver. This pruning step will drop fields not specified in the OpenAPI validation spec, leading to the same persistence semantics as for native types.

Algorithmically deeply related to validation and pruning is "defaulting", i.e. the application of the OpenAPI "default" values. Defaulting is done after decoding for native types, before conversion. This proposal suggests to add defaulting just after pruning to the CustomResource JSON decoder.

Goals

- Prune unknown fields from CustomResources silently. Unknown means not to be specified in the OpenAPI validation spec.
- Allow to opt-out of pruning via the OpenAPI validation spec for a subtree of the JSON objects.
- Apply OpenAPI validation schema defaults to CustomResources.
- Have simple semantics for pruning and defaulting.

Non-Goals

- Add a strict mode to the REST API which rejects objects with unknown fields.

Motivation

Pruning

Pruning of a JSON value means to remove “unknown fields”. A field (given as a JSON path) is unknown if it is not specified in the OpenAPI validation schema.

A JSON path `<JSON path>.x` is specified in an OpenAPI validation schema if `”x” is a key in a corresponding `properties` field in the schema.

Example 1: Assume the OpenAPI schema

`{properties: {“a”: {}, “b”: {type: “string”}, “c”: {not: {}}}}`.

Then a JSON object `{“a”:1, “c”: 3, “d”: 4}` is pruned to `{“a”:1, “c”:3}`.

Note that the pruned object does not validate.

Example 2: Assume the OpenAPI schema

`{anyOf: [{properties: {“a”: {}}}, {properties: {“b”: {}}}]`.

Then a JSON object `{“a”:1, “b”: 2, “c”: 3}` is pruned to `{“a”:1, “b”:2}` with the given semantics. Note that also `{“a”:1}`, `{“b”:2}` would be natural pruning results if we defined “specified” in a different way.

Example 3: Assume the OpenAPI schema

```
`{
    properties: {
        “a”: {},
        “b”: {type: “string”, properties: {“x”: {}}
    },
```

```
        anyOf: [
            { properties: {“c”:{}} },
            { properties: {“d”:{not:{}}}}
        ]
}`.
```

Then `a`, `b`, `b.x`, `c`, `d` are specified, `b.y` and `e` are not specified.

Question: is it natural semantics to assume `d` to be specified and not to prune it? Note that there is no object with `d` that validates against `”d”:{not:{}}`. But due to the `anyOf` there are objects which will validate against the complete schema.

Pruning Options: Motivated by the examples, we have different options to define the pruning semantics:

1. Use the described semantics of `specified`.
2. Only consider `properties` fields in the schema which actually successfully validate a given object (then `d` would be pruned in example 3).
3. Only consider `properties` fields outside of `anyOf`, `allOf`, `oneOf`, `not` during pruning.
4. Only consider `properties` fields outside of `anyOf`, `allOf`, `oneOf`, `not` during pruning, but enforce that every `properties` key inside `anyOf`, `allOf`, `oneOf`, `not` also appears outside all of those.

From these options:

1. leads to fields being kept from pruning although they only appear in branches of the OpenAPI validation schema which do not contribute to validation of an object.
2. is ambiguous if you have multiple branches of `anyOf` validating. Should we drop the fields of the first or the second branch?
3. leads to surprising pruning of fields that the user forgot to specify outside of propositional logic of `anyOf`, `allOf`, `oneOf`, `not`.
4. forces the user to re-specify all those properties outside of `anyOf`, `allOf`, `oneOf`, `not` which appear inside of them. The outcome will match that of 1, with the difference that it makes the skeleton explicit.

Here we propose **not to follow 2 and 3** due to ambiguity of 2 and the danger of user mistakes of 3. Both 1 and 4 lead to the same pruning behaviour and only differ in whether the user has to re-specify property keys or whether they are automatically derived.

Defaulting

The OpenAPI validation schema allows to define defaults via the `default` field. For example:

`{properties: {“a”: {default: {“x”: 42}}}}`

An empty object `{}` would be defaulted to `{a:{“x”:42}}`

Obviously, the default values should validate. But OpenAPI validation schemata do not enforce that. Also one default in a branch of `anyOf`, `allOf`, `oneOf` might force another validation run onto another branch of these quantors.

Even worse: forcing onto another branch might make more defaults applicable. In other words, we would need to run validation with defaulting until it converges. This is a hint that the semantics of defaulting in full OpenAPI schemata are not well defined either.

We propose to allow defaults only outside of `anyOf`, `allOf`, `oneOf`, `not`.

Without that, we would get ambiguous defaulting semantics::

Example 4: Assume `anyOf:[{default: 1}, {default: 2}]`. Which branch should be chosen while applying the defaults.

Example 5: Assume

```
`anyOf:[
        {properties: {“a”: {pattern “1”}, “b”: {type: “string”}, “d”: {default: 1}},
        {properties: {“a”: {pattern “2”}, “c”: {type: “string”}, “d”: {default: 2}}
]`.
```

Take an object `{“a”: “1”, “b”:”foo”, “c”:”bar”}`. Assuming we follow pruning option 1, it is pruned to `{“a”: “1”, “b”:”foo”, “c”:”bar”}` and to default to `{“a”:”1”, “b”: “foo”, “c”:”bar”, “d”: 1}`.

But what about the object `{“a”: “12”, “b”:”foo”, “c”:”bar”}`? Both patterns apply. With pruning option 1 we keep `”b”` and `”c”`. But what about the default? Should we default to `1` or `2`?

Clearly, we can make the algorithm deterministic by choosing one, e.g. the first matching branch. This choice is arbitrary, hard to understand, and gets even more confusing with even more nested `allOf`, `anyOf` or even `not`.

Mixing of Schema and Value Validation

The OpenAPI validation schema mixes

- the actual structural schema validation (which is usually done by the Golang struct JSON decoding for native types)
- and the value validation (usually done in the validation step of the API server handler pipeline for native types).

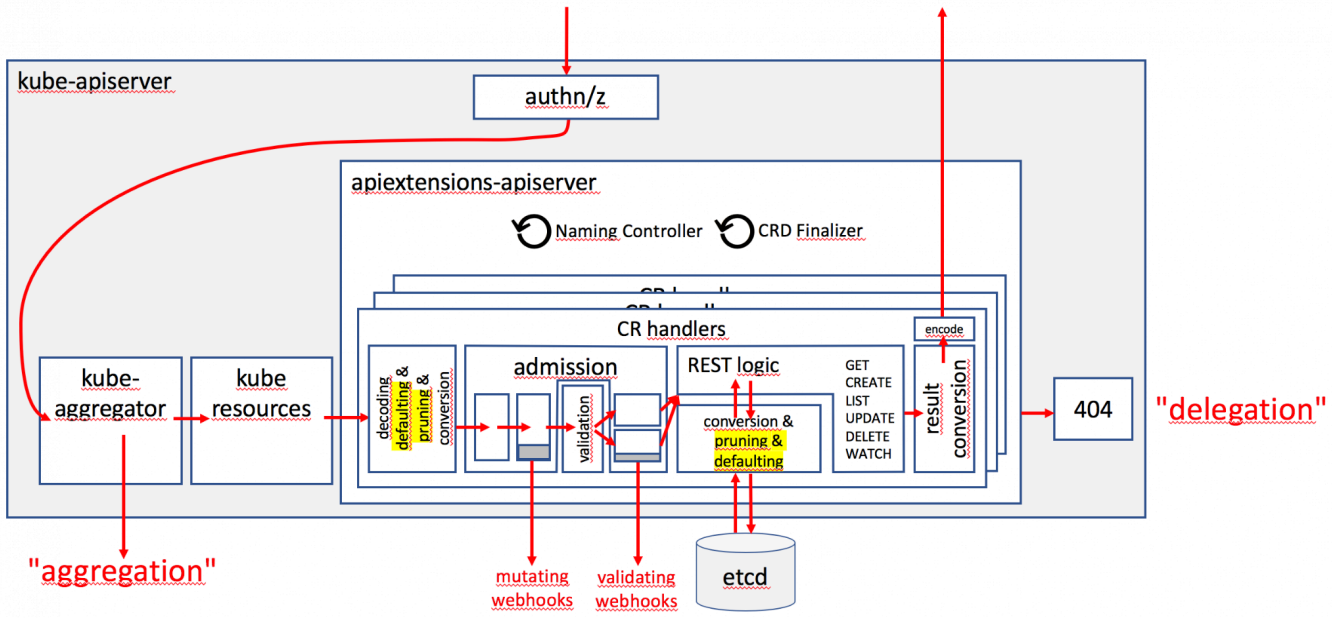
For CRDs we cannot distinguish both. This was first noticed by @lavalamp in https://github.com/kubernetes/kubernetes/pull/64907#issuecomment-397015030.

Example 6: Assume

```
`{properties: {
        “a”: {type: string, pattern: “<some-long-regex-for-ips>”, format: “ip”},
        “b”: {properties: {“x”: {}, “y”: {}}
}`
```

The type and properties are usually called schema validation, while regex patterns and formats are about values. The latter would be tested in the validation phase, the former would be checked during decoding for native types in the JSON decoder.

Remark: the line between type and format is blurry. Format is optional to be processed by tools, and the value is open ended. In Kube we have some formats (like `date`) which correspond to custom JSON unmarshallers in the native types. That format would be verified during decoding, although technically a date is just a string. So we might want to replicate that behaviour at some point.



For pruning and defaulting we have to apply the OpenAPI validation schema inside of the decoder step (the left box inside apiextensions-apiserver in the figure above). This is considerably earlier than for native types. This has a number of implications:

1. We have to apply full OpenAPI value validation (e.g. regular expressions, propositional evaluation) during decoding.
2. Our generic registry applies validation **after** defaulting. We need the other way around for CRDs.
3. Our generic registry expects defaulting to always succeed (no error result type). CRD validation needed by defaulting can fail.

To avoid these and to get sane semantics, we propose to split the CRD validation into two steps:

1. During decoding we validate using a skeleton schema which lacks value validations and propositional logic (`anyOf`, `allOf`, `oneOf`, `not`).
2. During defaulting we validate using the skeleton schema and apply the resulting defaulter.
3. During standard generic registry validation phase we validate using the full validation schema.

Because the first step succeeds, also the second step succeeds. So we don't need error handling during defaulting. The third step will catch wrong types of the used defaults.

Formal Proposal – following pruning option 1

The examples in the motivational section show that the semantics of full OpenAPI validation schemata are not trivial in respect to pruning and defaulting. To simplify the algorithms and to enforce "sane" schemata which allow to split schema and value validation, we propose to derive a skeleton schema from the full user-given OpenAPI validation schema

- which does not contain value validations
- which is complete enough for pruning and defaulting.

Remark: we have two options to define and implement pruning and defaulting:

1. directly on the full OpenAPI validation schema with two custom algorithms doing parallel recursion over the schema and the input object.
2. via an intermediate representation (the skeleton schema) and using go-openapi pruning and defaulting. Both algorithms are ten-liners based on the go-openapi/validate output. With the skeleton schema the go-openapi pruning algorithm coincides with our pruning option 1.

Both routes lead to the same algorithm: the intermediate representation of the skeleton schema makes merging of OpenAPI validation schema constraints explicit, while the custom algorithms would hide that in its recursion code.

Moreover, the main reason for the intermediate schema: the custom algorithm would have to replicate a lot of the validation logic of go-openapi, e.g. the semantics of `properties`, `additionalPropoerties`, `patternProperties` and the same for items of an array. With route 2 we get all of this for free.

Example 7: Assume

```
`{
        patternProperties: {
                "^foo-.*": { properties: {"a":{}}, "c": {default: 1} },
                "^bar-.*": { properties: {"b":{}}, "c": {default: 2} }
        }
}`.
```

An object `{"foo-42": {"a": 1, "b": 2}, "bar-42": {"a": 1", "b": 2}}` will be pruned to `{"foo-42": {"a": 1}, "bar-42": {"b": 2}}` and defaulted to `{"foo-42": {"a": 1, "c": 1}, "bar-42": {"b": 2, "c": 2}}`.

Note that the default and the pruning only depends on the object keys, not on values.

Definition: For a given OpenAPI validation schema `s` the *skeleton schema* `skel(s)` is derived by

- applying `skel` to all elements of `.allOf`, `.anyOf`, `.oneOf` and `.not` giving `s_1, …, s_n`,
- then dropping all fields from `s` other than
    - `type`,
    - `items`,
    - `additionalItems`,
    - `properties`,
    - `patternProperties`,
    - `additionalProperties`,
    - `default`,

  giving `drop(s)`,
- then merging `s_i` into it's containing object.

I.e.:

```
skel(s) := merge(drop(s), s_1, …, s_n)
```

with

```
merge(x_1, …, x_n) := {
```

type: t *if all x_i agree on t as type, undefined otherwise*[1]
items: [merge(x_i1, …, x_ik)] *where s_ij = s_j.items[i] if defined*,
additionalItems: merge(p_1, …, p_i),
for all x_i with defined additionalItems p_i
properties: { k_i: merge(v_i1, …, v_ik) *for keys appearing in* x_ij with values x_ij },
patternProperties: { k_i: merge(v_i1, …, v_ik) *for keys appearing in* x_ij with values x_ij },
additionalProperties: merge(p_1, …, p_i),
for all x_i with defined additionalProperties p_i
default: first default value of `x_i` or fail[2]
}.

The skeleton schema especially lacks:

- all value validations like `pattern`, `format`, `minValue`, `maxValue`, ...
- all propositional operators like `anyOf`, `allOf`, `oneOf`, `not`.

The skeleton schema `skel(s)`

- puts less or equal constraints on `type` than `s`.
- every field specified by a `properties` key in `s` is specified in `skel(s)`.
- if there are no default values under `allOf`, `anyOf`, `oneOf`, `not` in `s`, for each JSON path the default matches in `s` and `skel(s)`.

The computation of `skel(s)` is `O(size(s))` and `size(skel(s)) <= size(s)`.

Note that a field might be constrained by the `type` construct in the full OpenAPI validation schema, but not in its skeleton. This is fine because we have to support polymorphic fields like `IntOrString`, but avoid `allOf`, `anyOf`, `oneOf`, `not` in the skeleton.

Property: if the OpenAPI validation schema applies to an object, so does its skeleton schema.

Pruning and defaulting will be implemented based on the skeleton schema of the specified OpenAPI validation schema.

Optionally for debugging, the skeleton schema derived from the validation schema by the apiextensions-apiserver can be stored in the CRD status.

Example 8: Assume

---

[1] compare discussion about "types and formats": type could be omitted completely.
[2] if there is no default under `anyOf`, `andOf`, `oneOf`, `not` there will be at most one default, compare restriction 2.

```
`s := {
        anyOf: [
                {properties: {“a”: {type: “string”}}},
                {properties: {“a”: {type: “integer”}, “b”: {type: “string”}}}
        ],
        properties: {“c”: {}}
}`.
```

Then

```
`skel(s) = {
        properties: {
                “a”: {},
                “b”: {type: “string”},
                “c”: {}
        }
}`
```

The `type` of `a` does not match on all paths. Hence, it is omitted in the skeleton. In contrast, `b` has a unique `type` of `”string”`, so it stays in the skeleton.

Types and Formats

Note, that having less `type` constraints in the skeleton than in the whole OpenAPI validation schema means that admission and conversion will see possibly wrong types. The final validation in the registry validation phase will check for the complete OpenAPI validation schema and catch those type errors.

Question: we could extent the `skel` function to keep a list of `type` values. As long as all branches define the type, we can add `anyOf: [{type: “type1”, type: “type2, ….}]` to the skeleton. If one branch does not define the type though, this is still not possible.

In native types, we have custom unmarshallers for date / timestamps. In OpenAPI these would be rendered as `{type: “string”, format: “date”}`. With the proposed skeleton algorithm, we would not verify these fields before full OpenAPI validation in the registry. We could move non-contradicting `format` constraints into the skeleton as well, like we do for `type` already. Then admission would be protected from invalid format (if admission uses Golang decoding, this might be relevant to get proper error messages).

Polymorphic Fields

Example 9: Assume

```
`s := {
        anyOf: [
                {properties: {“a”: {type: “string”}}},
                {additionalProperties: {type: “integer”}}
        ],
}`.
```

Then

```
`skel(s) = {properties: {}, additionalProperties: {}}`.
```

In the CRD validation, we reject OpenAPI validation schemata like `skel(s)` with `properties` and one of `additionalProperties` or `patternProperties`[3] being defined at the same time. Having `additionalProperties` or `patternProperties` defined there means to have a `map[string]T` like field where we don’t want to prune the unknown “keys”.

The schema `s` above means to either have a `map[string]int64` or a `struct` with `a` of type string. This is a special case of polymorphism because for the same JSON path is typed with two different types in the Golang sense (struct and `map[string]T`), but same types in the JSON sense `JSON object).

Hence, we have to add the following restriction to OpenAPI validation schemata:

Restriction 1 on Object Polymorphism: reject CRD OpenAPI validation schemata `s` with `skel(s)` having `properties` and one of `additionalProperties` or `patternProperties` being defined for the same JSON path.

Defaulting

Restriction 2: Defaults may only be defined outside of `anyOf`, `allOf`, `oneOf` and `not`. All other defaults are rejected in CRD validation.

Consequence: defaults are not conditional, but static. I.e. they don’t depend on the validity of other parts of the input object.

Example 11: in a OpenAPI validation schema, one can write:

```
`anyOf: [
        { properties: {“a”: {pattern: “1”}, “b”: {default: “1”}}},
```

[3] We have a bug in validation not excluding `properties` and `patternProperties`, compare https://github.com/kubernetes/kubernetes/issues/66610

{ properties: {“a”: {pattern: “2”}, “b”: {default: “2”}}}
]`.

This says to default `b` to the same value as `a`. This cannot be expressed with a skeleton schema. In this sense, the defaults are static.

Pruning and defaulting will be applied after every unstructured JSON decoding and before conversion.

Excluding values from Pruning

There are cases where parts of an object are verbatim JSON, i.e. without any applied schema and especially without a complete specification which allows to apply pruning. E.g. the [`JSONSchemaProps` types in apiextensions-apiserver contain fields of type JSON](https://github.com/kubernetes/kubernetes/blob/master/staging/src/k8s.io/apiextensions-apiserver/pkg/apis/apiextensions/v1beta1/types_jsonschema.go#L28) for defaults, examples and enums. These should never be pruned (and they are not pruned while decoding using JSON marshalling).

Hence, we need a mechanism to express that in the OpenAPI validation schema (compare https://github.com/kubernetes/kubernetes/pull/64558#issuecomment-403564033).

Raw JSON Option 1: add a format `json`, e.g.:

`{properties: {“x”: {format: “json”}}}`

In this example “x” is excluded from pruning. Note that you can still use any kind of OpenAPI validation schema constructs to restrict `“x”` further.

Note that we lose expressiveness of the existing `format` strings: we either apply the format to `”json”` or any other pre-defined format. I.e. we cannot express that pruning should be disabled, but if it is an integer, it should be an int32.

Question: this feels like a reasonable loss of expressivity. Do we accept that?

Raw JSON Option 2: add an extension property, e.g.

`{properties: {“x”: {x-no-pruning: true}}}`.

This would not lead to a loss in expressivity. It is formulated negatively intentionally to have `false` as the default with `omitempty`.

As OpenAPI extensions are prefixed with `x-kubernetes` elsewhere, we probably should name it `x-kubernetes-no-pruning: true`.

We propose to follow option 2 because it is more explicit, does not reduce expressivity and does not mangle with the already very vague `format` field definition.

Nested x-no-pruning

Question: should we support nested `x-no-pruning`, i.e. disabling pruning for a sub-object, but re-enable it something deep inside of it? E.g.

```
`{properties: {
        “x”: {
                x-no-pruning: true,
                properties: {
                        “y”: {
                                x-no-pruning: false,
                                properties: { “z”: {} }
                        }
                },
        },
}}`.
```

If we do, the object

```
`{
        “a”: 1,
        “x”: {
                “b”: 2,
                “y”: {
                        “c”: 3,
                        “z”: 42
                }
        }
}`
```

is pruned to `{“x”:{“b”: 2, “y”:{“z”: 42}}}`.

Ratcheting (slightly related)

In https://github.com/kubernetes/kubernetes/pull/64907 a ratcheting mechanism during validation is introduced: if a specific proposition during validation is not fulfilled in an object stored in etcd, it is not required either during updates of that object.

Ratcheting does not work well with arbitrary propositional OpenAPI validation schemata because propositions in negative positions (under `not` or `oneOf`) that are weakened via ratcheting will make the whole validation stricter instead of weaker (compare https://github.com/kubernetes/kubernetes/pull/64907#issuecomment-396526377).

We have multiple options to restrict the use of ratcheting (which could be the property of certain OpenAPI validation schemata constructs):

1. Allow ratcheting only in the skeleton schema. Everything in the skeleton schema is positive.
2. Disallow ratcheting under `not` and under `oneOf`, but allow it everywhere else.
3. Disallow ratcheting in negative positions. This implies that it is disallowed under `oneOf`, but is allowed e.g. under `not: {not: ….}`.

The first is the most strict one, the last is the most relaxed one. On the other hand, 1 is the simplest to reason about.

Question: which of the options do we want?

References

- WIP implementation PR https://github.com/kubernetes/kubernetes/pull/64558

[OLD] Alternative: following pruning option 4

The examples in the motivational section show that the semantics of full OpenAPI validation schemata are not trivial in respect to pruning and defaulting. To simplify the algorithms and to enforce “sane” schemaschemata which allow to split schema and value validation, we propose a restriction of the OpenAPI validation schema such that

- we can derive a skeleton schema by dropping value validation constructs
- and that skeleton schema is complete enough for pruning and defaulting.

Definition: For a given OpenAPI validation schema the *skeleton schema* is derived by dropping all fields other than:

- `properties`
- `type`

- `items`
- `additionalItems`,
- `additionalProperties`
- `patternProperties`
- `default`.

The skeleton schema especially lacks:

- all value validations like `pattern`, `format`, `minValue`, `maxValue`, ...
- all propositional operators like `anyOf`, `allOf`, `oneOf`, `not`.

Note that a field might be constrained by the `type` construct in the full OpenAPI validation schema, but not in its skeleton. This is fine because we have to support polymorphic fields like `IntOrString`.

Property: if the OpenAPI validation schema applies to an object, so does its skeleton schema.

Claim: the validation during Golang struct type JSON decoding without custom decoders is expressible via a skeleton schema.

Pruning and defaulting will be implemented based on the skeleton schema of the specified OpenAPI validation schema.

Optionally for debugging, the skeleton schema is derived from the validation schema by the apiextensions-apiserver and stored in the CRD status.

Invariant A: every `properties` key appearing in non-skeleton parts of the OpenAPI validation schema, must appear in the skeleton schema part for the same JSON path.

We can assign a JSON path to each `properties` key in the OpenAPI validation schema. This way we can identify those inside the skeleton and those which are dropped when deriving the skeleton. We can verify invariant A in O(size(OpenAPI validation schema)).

Note again, that the skeleton might not specify the `type` where the full OpenAPI validation schema does (which is fine).

Example 7: Assume

```
{
    anyOf: [{properties: {“a”: {type: “string”}}}, {properties: {“b”: {}}}],
    properties: {“a”:{}, “b”:{}}
}
```.

This fulfills invariant A because `a` and `b` inside of the `anyOf` appear also appear in the skeleton `{properties: {“a”:{}, “b”:{}}}`.

We can verify invariant A in O(size(OpenAPI validation schema)) during CRD validation.

An OpenAPI validation schema with `properties` and at least one of `addtionalProperties` or `patternProperties` for the same JSON path is called *object polymorphic*.

Property: every OpenAPI validation schema not object polymorphic can (easily, by adding missing `properties` keys) be extended to an equivalent (in the sense it validates the same objects) OpenAPI validation schema which fulfills invariant A.

Note, that this requires potentially that the skeleton schema does not specify all types that the whole schema does. The user is adviced to add as many `type` constraints as possible to the skeleton part of the OpenAPI validation schema for maximal validation, e.g. before admission webhooks are called.

Restriction 1: OpenAPI validation schemata contradicting invariant A, will be rejected during CRD validation.

Restriction 2: Defaults may only be defined inside the skeleton schema parts of the OpenAPI validation schema. All other defaults are rejected.

Consequence: defaults are not conditional, but static. I.e. they don’t depend on the validity of other parts of the input object.

Example 8: in a OpenAPI validation schema, one can write:

```
`anyOf: [
    { properties: {“a”: {pattern: “1”}, “b”: {default: “1”}}},
    { properties: {“a”: {pattern: “2”}, “b”: {default: “2”}}}
]`.
```

This says to default `b` to the same value as `a`. This cannot be expressed with a skeleton schema. In this sense, the defaults are static.

Pruning and defaulting will be applied after every unstructured JSON decoding and before conversion.