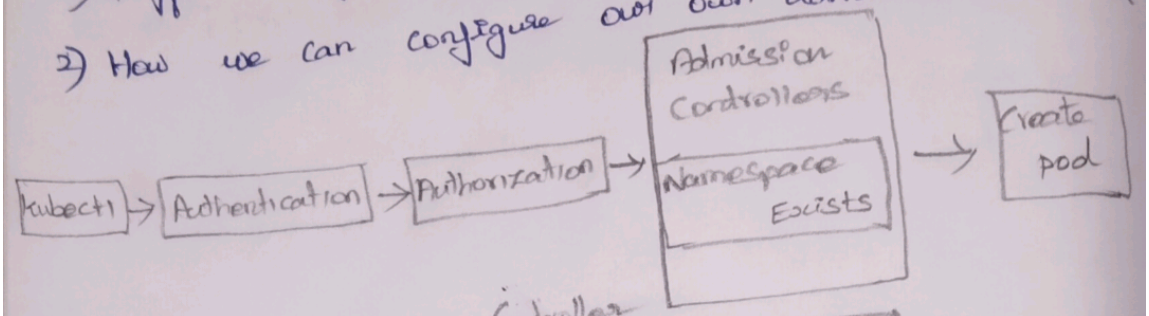
86 (2025 updates) Validating & Mutating Admission Controllers

0:27 One of the confusing topic Read it thoroughly

Topics to be discussed:-

1) Different types of admission controllers (3)

2) How we can configure our own admission controller



What is Validating Admission Controller

We looked at the "NamespaceExists" or "NamespaceLifecycle" admission controller, it can help validate if a namespace already exists & rejects the request if it doesn't exist. This is known as Validating admission controller

What is Mutating Admission Controller

DefaultStorageClass plugin:-

⇒ This is enabled by default

Eg:- we're submitting a request to create a PVC, the 0:47 request goes through authentication, authorization & finally the admission controller. 1:37

Mutating admission controller

PVC-definition.yaml

```
apiVersion : v1
kind: PersistentVolumeClaim
metadata :
    name: myclaim
spec:
    accessModes :
        - ReadWriteOnce
    resources:
        requests:
            storage : 500Mi
    storageClassName : default
```

Think like as we have not given this in manifest file

⇒ The DefaultStorageClass Admission Controller watch your a request to create a PVC & check it has a "storage class" mentioned in it, if not, it'll modify your request to add the Default Storage Class to your request

⇒ This could be whatever storage class is configured as the Default StorageClass in your cluster,

⇒ So, when the PVC is created & you inspect it you'll see that a StorageClass default is added to it even though you hadn't specified it during the creation.

```
kubectl describe pvc myclaim

Name:          myclaim
Namespace:     default
StorageClass:  default
Status:        Pending
Volume:
Labels:        <none>
Annotations:   <none>
```

⇒ This type of "admission controller" is known as mutating admission Controller

⇒ It can change or mutate the object itself before it is created.

Mutating admission Controller ⇒ Change the request

Validating admission Controller ⇒ Validate the request & allow (or) deny it

There are "admission controllers" which can do both "mutate a request" as well as "Validate a request"

Generally, "mutating admission controllers" that are invoked first followed by "Validating admission controllers"

Reason:-

Any change made by the mutating admission Controller can be considered during the validation process

2:13 3:03

In the example

The "NamespaceAutoProvisioning admission Controller", which is a "mutating admission controller" is run first followed by the Validating controller - NamespaceExists

Mutating - NamespaceAutoProvision

Validating - NamespaceExists

If it was run the otherway, then the NamespaceExists admission Controller would always reject the request for a namespace that doesnot exist & the NamespaceAutoProvisioning Controller would never be invoked to create the missing namespace

⇒ when a request goes through these admission Controllers, if any admission controller rejects the request the request is rejected & an error message is shown to the user

⇒ The admission Controllers like *AlwaysPullImages * DefaultStorageClass *EventRateLimit *NamespaceAutoProvision *NamespaceExists & many more these all are built-in admission controllers that are part of K8S source code & are compiled & shipped with K8S

2) What if we want our own admission Controllers with our own mutation & validation that has our own logic

Custom admission Controller

To support external admission Controllers, there are 2 special admission Controllers available

* Mutating Admission Webhook
* Validating Admission Webhook

4:35

3:27 ⇒ We can Configure these webhooks to point to a server that's hosted either within the K8S cluster (or) outside it, & our server will have our own admission webhook service running with our own code & logic.

⇒ After the request goes through all the built-in admission Controllers, it hits the webhook that's configured

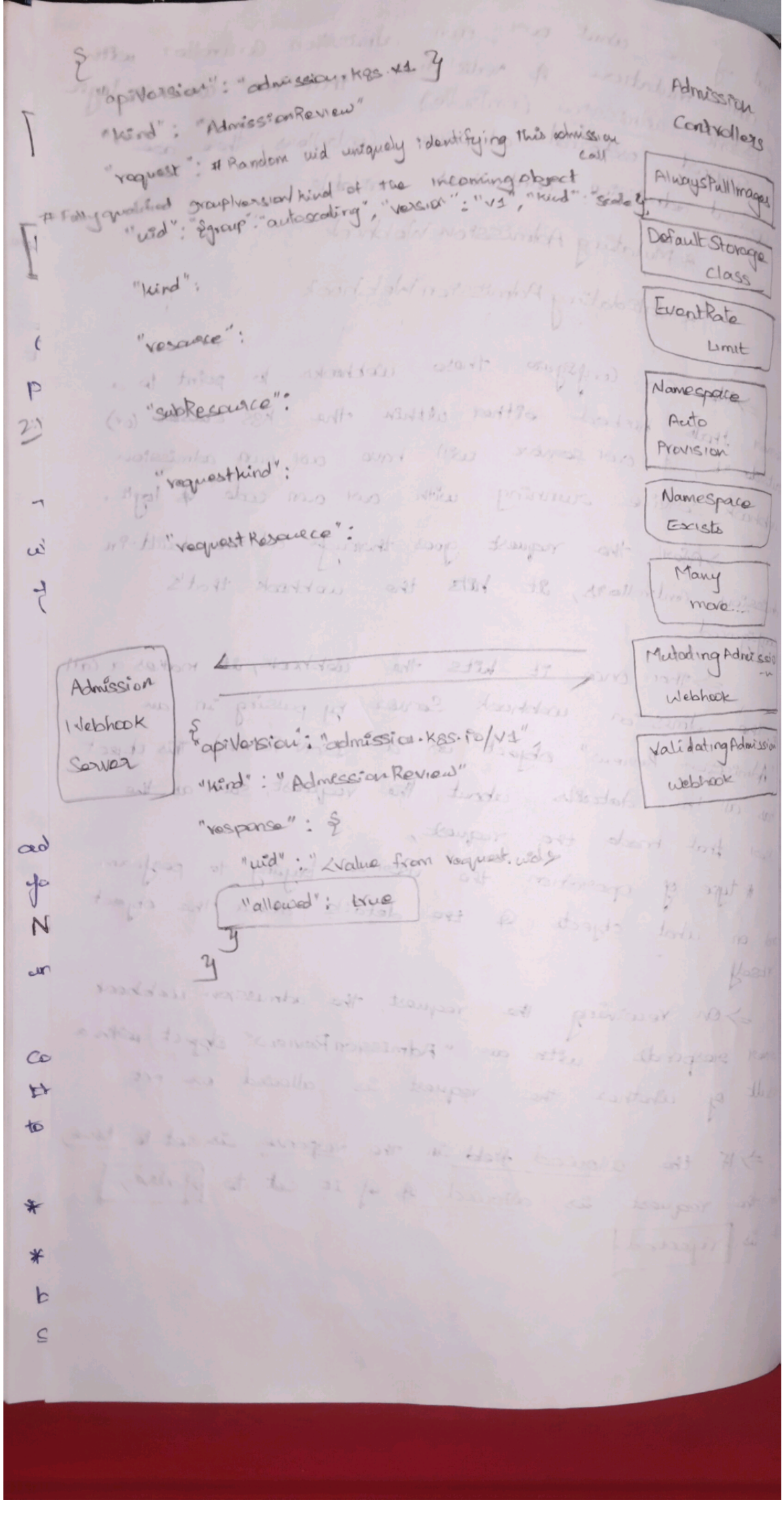
⇒ Then once it hits the webhook, it makes a call to the admission webhook server by passing in an "Admission Review" object in a JSON format. This object has all the details about the request, such as the user that made the request,

* type of operation the user is trying to perform

and on what objects & the details about the object itself

⇒ On receiving the request, the admission webhook server responds with an "Admission Review" object with a result of whether the request is allowed or not

⇒ If the allowed field in the response is set to true, then the request is allowed & if it set to false, it is rejected.


{
"apiVersion": "admission.k8s.v1"}
Admission Controllers
"kind": "AdmissionReview"
"request": # Random uid uniquely identifying this admission call
Fully qualified group/version/kind of the incoming object
"uid": "group": "autoscaling", "version": "v1", "kind": "Scale"}
"kind":
"resource":
"subResource":
"requestKind":
"requestResource":
AlwaysPullImages
Default Storage class
EventRate Limit
Namespace Auto Provision
Namespace Exists
Many more...
MutatingAdmissionWebhook
ValidatingAdmissionWebhook
Admission Webhook Server
{"apiVersion": "admission.k8s.io/v1"
"kind": "AdmissionReview"
"response": {
"uid": <value from request.uid>
"allowed": true
}
}

How to set this up

1st) Deploy webhook server
2nd) Configure the webhook

⇒ First, we must deploy our admission webhook server, which will have our own logic, & then we configure the webhook on K8s by creating a webhook configuration object

1) Deploy Webhook Server.

The first step is to deploy our own webhook server, which will have our own logic & then we configure the webhook on K8s by creating a webhook configuration object

Lets take a look at each of those steps next

⇒ So the first step is to deploy our own webhook server (admission webhook server) this could be an API server that could be built on any platform

An example code of a webhook server, written in Go can be found in K8s documentation page, It'll be written in Go programming language

You could develop your own server in other languages as well if required

⊗ The only requirement is that it must accept the mutate & validate APIs & respond with JSON object that the webserver expects

1:22
5:19

Here is the pseudo code of a Sample webhook server written in python.

Those are 2 calls

* a validate call
* * a mutate call

⇒ The Validate Call receives the Validation webhook request & in this example composes the name of the object & the name of the user who sends

the request & rejects the request if its the same name

⇒ Its a simple use case to show what we can do with the requests that come in.

sample webhook server written in python

validate call

```
@app.route("/validate", methods=["POST"])
def validate():
    object_name = request.json["request"]["object"]["metadata"]
                                                    ["name"]
    user_name = request.json["request"]["userInfo"]["name"]
    status = True
    if object_name == user_name:
        message = "You can't create object with your own name"
        status = False
    return jsonify(
    {
      "response": {
        "allowed": status,
        "uid": request.json["request"]["uid"],
      } , "status": {"message": message},
    })
```

mutate call

```
@app.route("/mutate", methods=[POST])
def mutate():
    user_name = request.json["request"]["userInfo"]["name"]
    patch=[{"op": "add", "path": "/metadata/labels/users", "value":
                                                    user_name}]
    return jsonify(
    {
        "response": {
            "allowed": True,
            "uid": request.json["request"]["uid"],
            "patch: base64.b64encode(patch),
            "patchtype": "JsonPatch", } } )
```

patch object

⇒ And if you look down, we'll see the [mutating webhook] which gets the username & responds with a JSON patch operation of adding the username as a label to any request that was raised by anyone

⇒ If we look at the "piece of code", the patch object is a list of patch operations, with each operation being add, remove, replace, move, copy or test & we then specify the path within the JSON object, that needs to be targeted your change, In this case it is /metadata/labels/ users & then the value that needs to be added, & if it is an add operation, so we get the username from the request. So thats going to be the value of that particular label.

This is then sent as a base64-encoded object as part of the response

Takeaway from the piece of code :-

The "admission webhook server" is a "server" that you deploy that contains the logic or the code to permit (or) reject a request & it must be able to receive & respond with the appropriate responses that the webhook expects

This is the example to show what kind of things you can do, (or) what kind of things that you can code & implement in the webhook server that you deploy

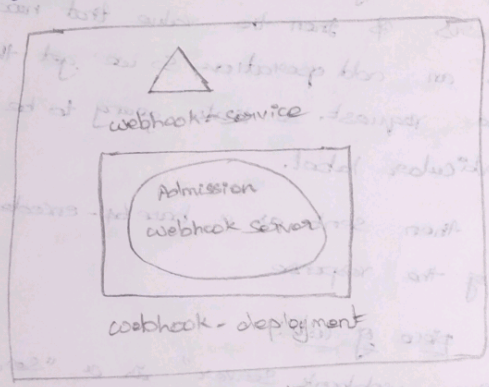
Once we developed our webhook server the next step is to host it

Deploy webhook server (& Host)

⇒ Either we run it as a **server** somewhere (or) containerize it & deploy it within k8s cluster itself as a **deployment**

⇒ If deployed as a **deployment** in a k8s cluster then it needs a **service** for it to be accessed.

⇒ So we have a service named **"webhook-service"** as well



1st part "installation" is done & now the 2nd part "configuration"

2) Configuring Admission Webhook

The next step is to configure our cluster to reach out to the service & validate or mutate the requests. For this, we'll create a validating webhook configuration object

Validating webhook Configuration Object

```
apiVersion: admissionregistration.k8s.io/v1
kind: ValidatingWebhookConfiguration
metadata:
  name: "pod-policy.example.com"
webhooks:
- name: "pod-policy.example.com"
  clientConfig:
    service:
      namespace: "webhook-namespace"
      name: "webhook-service"
    caBundle: "Ci0tLS0tLk..... tLS0K"
  rules:
    - apiGroups: [""]
      apiVersions: ["v1"]
      operations: ["CREATE"]
      resources: ["pods"]
      scope: "Namespaced"
```

⇒ Here, we are configuring "validate webhook" so we have mentioned the kind as "Validate", for mutate, the "kind" should be "mutating webhook Configuration"

⇒ Under webhooks, we configure different webhooks so, a webhook has a name, clientConfig and a set of rules, the name we set to "pod-policy.example.com" & the client Config is where we configure the location of our admission webhook server, if we deployed this server externally on our own that's not part of a deployment in K8S cluster, then we can simply provide a URL path to server like this url: https://external-server.example.com

⇒ Now, instead if we deployed the server as another service on our own cluster, you use the service configuration & provide the namespace & name of the service which in our case is "webhook-service"

```
apiVersion: admissionregistration.k8s.io/v1
kind: ValidatingWebhookConfiguration
metadata:
  name: "pod-policy.example.com"
webhooks:
- name: "pod-policy.example.com"
  clientConfig:
    service:
      namespace: "webhook-namespace"
      name: "webhook-service"
    caBundle: "CiBtLSOtLOk..... tr SOtK"
  rules:
  - apiGroups: [""]
    apiVersions: ["v1"]
    operations: ["CREATE"]
    resources: ["pods"]
    scope: "Namespaced"
```

Now, of course the communication b/w the API server & the webhook server has to be over TLS, so a certificate bundle should be configured, so the server has to be configured with pair of certificates. Then caBundle has to be created and passed into this client config caBundle