Client-Go: Setup and Read

can you teach me client-go in a series of 4-6 parts? also take some reference from this transcript, otherwise dont if ou dont want, give the first part now, ive only started learnig go , so keep in mind to keep things super detailed with codes and take things from basic to advanced

Intro
0:00
Okay cool so you have learned the C fundamentals from either watching this or this and now you're wondering what
0:05
next what can we do with this particular knowledge The Next Step would be to creating applications using cuberes
0:11
right understanding the cuber fundamentals and The Core Concepts is one thing and using them to create
0:17
applications on top of cuties is another thing and the concepts which are involved in creating cuties native
0:23
applications can get a little bit overwhelming especially for beginners now in order for us to create
0:28
applications on cuberes we need to use its API itself so Cub is the tool that we use under the hood it's an API now if
0:35
you knew about this that is great but if you didn't I have a Blog in the description box down below that will teach you the basic fundamentals of
0:41
using a cuties API so essentially the central component we need in order to develop applications on cities is the
0:47
API itself so if you're wondering how can I use the communties API to create applications this is the video that
0:53
you're looking for hey everyone my name is Kunal Verma and I'm working as a devil at CU simplify in this particular
0:58
video we are going to learn how you can perform basic crud operations on a cuberes cluster that is create read
1:04
update and delete and yeah not using Cube CTL not using any curl command or
1:10
any https or any yl file we'll basically be writing code and we'll basically use something that the communties community
1:16
already provides for you to create applications using kubernetes if this sounds interesting hit the like button
1:21
and let's get started okay so our aim is basically to interact with the kubernetes cluster right at the end of
Intro to Client Libraries
1:27
the day we are going to perform the crud operations on a kubernetes this cluster right so for that purpose what we're
1:32
going to do is we are going to use something called as client library and client Library you can consider them as
1:37
sdks but they basically help us to interact with the cuber API using any
1:43
programming language right so as you can see the officially supported cities client libraries or the sdks programming
1:49
languages that are supported right now are listed down below so we have C goang Java JavaScript all these popular

1:56

languages even python so if you know any of these languages you can use your particular programming language and

2:02

interact with the cuberes cluster right so essentially with client libraries what we do is we don't interact with the

2:08

cuberes API directly but the library or the SDK basically provides an layer of

2:14

abstraction in between so any client library or the SDK that you use has all the functions methods and data

2:21

structures that you need in order to interact with the cuberes cluster you don't need to directly interact with the communties API using https or you know

2:29

using the Cur command you can use these pre-built data structures and the methods to interact with the communties

2:35

API right and that makes things a whole lot easier to develop applications with the help of cuties right so today what

2:42

we're going to do is we are going to use the client library for goang which is the go client and we'll basically be

2:48

using this particular library to write the code for our cred operations you can also use any language from this

2:54

particular list if you know python you can use Python if you know JavaScript you can opt for using JavaScript and

2:59

understand how how the client library is structured but in this particular one we are going to use goang because mostly

3:04

when we work with kubernetes or any Cloud native tool we often use go line right so that is what we're going to do Setting up the environment

3:10

today okay so this is what's going to happen right we are going to use the client go library right to interact with

3:18

the cuberes Clusters and there are a few steps that we need to follow in order to

3:23

basically first configure the client go library and to set up our environment and then go ahead and perform all the

3:29

scud operations so let us first set up our go project right let us do some initial configurations and then we'll

3:34

come back to how we need to work and how we need to basically configure our client code so before we jump onto this

3:40

make sure you already have a cuberes cluster up and running that we'll be using for testing purposes right so I've

3:45

already gone ahead and do that so if I do Cube CTL get nodes you'll see that I have a mini Cube cluster up and running

3:51

it's a single node cluster right you can either use kind or k3s or any other tool out there to quickly bootstrap a test

3:58

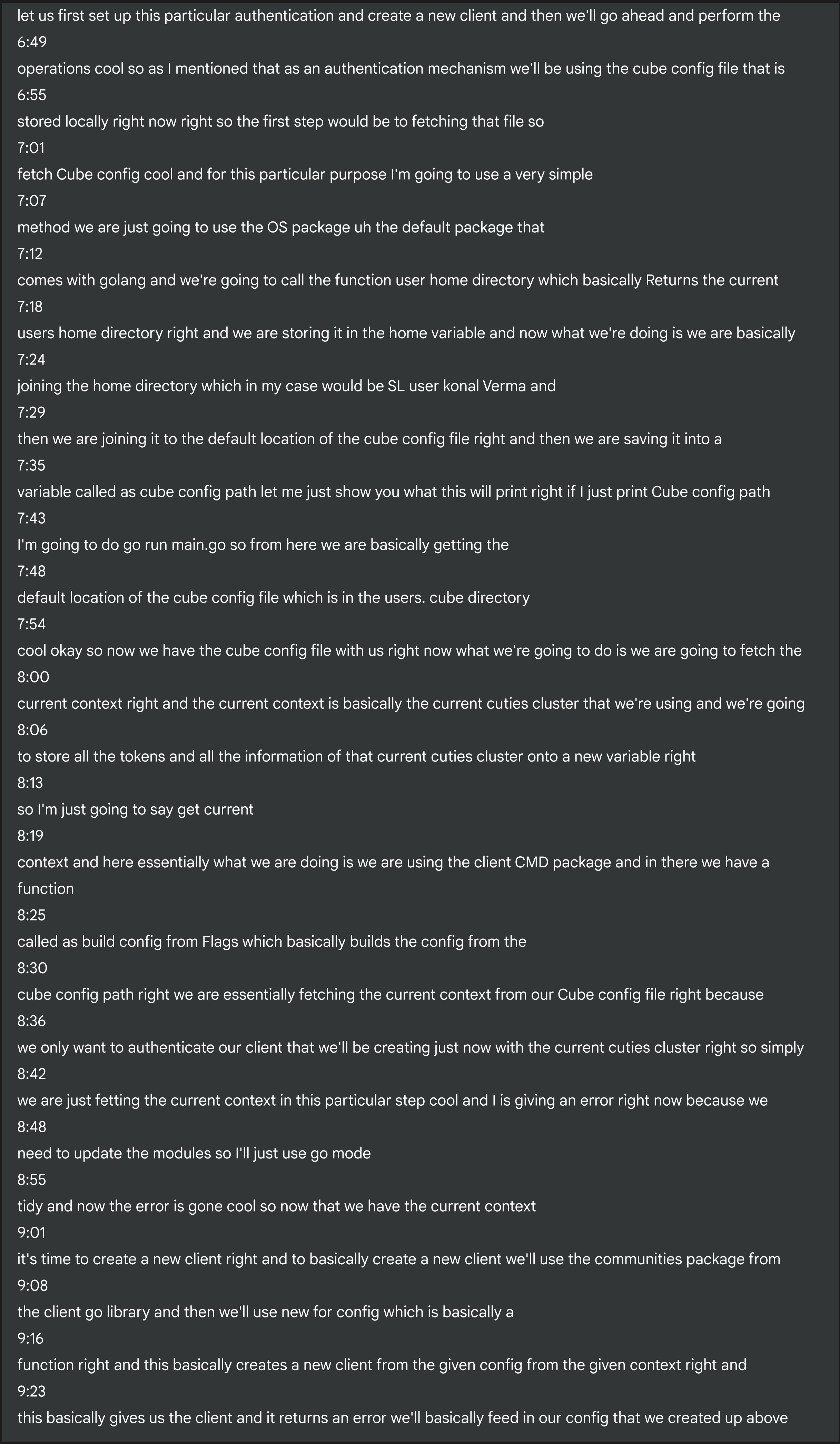
cluster and it doesn't need to have more than one node one node will be fine for our purpose cool so what I'm going to do

4:05

is I'm going to do everything from scratch if this is a first time working with goang I'll recommend first uh pause this particular video right now and go

4:11

ahead and check out some of the goang resources and get familiar with the basic syntax so that you can come back
4:16
and you can follow along with me so let us start by setting up our project and I'm going to create a new directory here
4:22
call it demo C right and then I'm going to open it into my vs code cool you can
4:28
use any editor of your choice I'm going to use vs code and then what we're going to do is
4:34
we are going to create a go. mod file which basically will help us manage all
4:40
of our dependencies I'm going to just go with example right now and let us name
4:47
that's it so the go. mod file is created I create a main. go as well I'm going to just increase the font and then package
4:55
Main Funk main all the normal boiler plate
5:02
stuff for goang okay so now our boilerplate project is set up now in order for us to use the client go
5:08
Library we need to fetch it right and in the documentation you'll get everything this is just a simple command so go get
5:15
k. io/ client go and we'll fetch the latest version so we'll go ahead and
5:21
paste the command here so the client go library is added and the go dood file has been updated okay so before we even
Creating a new Client
5:28
move on to performing the cred operations there are a few things that we need to do right so the number one
5:33
thing we need to do is we need to authenticate right because as you all
5:39
know that in order for us to connect with the communties API we need to use
5:44
an authenticated mechanism right and in this particular case what
5:49
we're going to do is we're going to use the cube config file right so the client
5:55
go or any client library that you may use in the future they use the cube
6:01
config file as a way of authentication and then from the cube config file what
6:06
we'll do is we'll create a new client right because essentially
6:12
communities is an API right and this basically involves the client server
6:17
communication where the client is ourselves right the developers so we basically need two things right one is
6:23
an authentication mechanism and in this case we are going to use the cube config file and then using this Cube config
6:28
file we'll create a new client right from the client go library and this particular client will help us to
6:35
perform all of our crowd operations right that is the process so
6:41



let us first set up this particular authentication and create a new client and then we'll go ahead and perform the
6:49
operations cool so as I mentioned that as an authentication mechanism we'll be using the cube config file that is
6:55
stored locally right now right so the first step would be to fetching that file so
7:01
fetch Cube config cool and for this particular purpose I'm going to use a very simple
7:07
method we are just going to use the OS package uh the default package that
7:12
comes with golang and we're going to call the function user home directory which basically Returns the current
7:18
users home directory right and we are storing it in the home variable and now what we're doing is we are basically
7:24
joining the home directory which in my case would be SL user konal Verma and
7:29
then we are joining it to the default location of the cube config file right and then we are saving it into a
7:35
variable called as cube config path let me just show you what this will print right if I just print Cube config path
7:43
I'm going to do go run main.go so from here we are basically getting the
7:48
default location of the cube config file which is in the users. cube directory
7:54
cool okay so now we have the cube config file with us right now what we're going to do is we are going to fetch the
8:00
current context right and the current context is basically the current cuties cluster that we're using and we're going
8:06
to store all the tokens and all the information of that current cuties cluster onto a new variable right
8:13
so I'm just going to say get current
8:19
context and here essentially what we are doing is we are using the client CMD package and in there we have a function
8:25
called as build config from Flags which basically builds the config from the
8:30
cube config path right we are essentially fetching the current context from our Cube config file right because
8:36
we only want to authenticate our client that we'll be creating just now with the current cuties cluster right so simply
8:42
we are just fetting the current context in this particular step cool and I is giving an error right now because we
8:48
need to update the modules so I'll just use go mode
8:55
tidy and now the error is gone cool so now that we have the current context
9:01
it's time to create a new client right and to basically create a new client we'll use the communities package from
9:08
the client go library and then we'll use new for config which is basically a
9:16
function right and this basically creates a new client from the given config from the given context right and
9:23
this basically gives us the client and it returns an error we'll basically feed in our config that we created up above

9:30

and then basically it returns two things one is the client so we'll create a new variable called as client and then we'll

9:36

also return an error because we need that as well it returns two things and

9:41

we'll also deal with the error right now nil and then I don't know

9:49

Panic error dot error cool so now we

9:55

have created a new client right this is an authenticated client which we can use to contact our cuties cluster now

10:02

interestingly there is actually another method we can use from the cuties package to create a new client and which

10:08

is a little bit shorter from this so if I just remove this right now what we can do is we can use

10:15

cuberes do new config or die and this basically is similar to the previous one

10:21

that we used but the only difference is that it handles the errors automatically right so we feed in the config that we

10:28

created in the above step and we don't need to return any error here right it only returns one thing that is the

10:34

client let us use this one just for a little bit Simplicity you can use any of them and both of them works just fine

Read all Pods

10:40

okay so what we have till now we have an authenticated client with

10:46

us authenticated client which we can use to talk to our

10:53

culties cluster right so the initial configuration is done now now we can go

11:00

ahead and perform our crud operations right so basically we'll perform

11:09

create read

11:15

update and delete uh sorry for the

11:21

spelling and on what we are going to perform this so we'll perform this operations on a

11:27

pod right as simple as that so this means that we'll create a pod we'll read all the pods that are

11:33

currently running in our cluster we'll update a specific pod that we need to do and then we'll delete a specific F right

11:40

these are the most basic operations and through these you'll basically understand how a client library is used

11:45

to perform operations right okay cool so let us start with the most basic operations from this is reading all the

11:52

existing pods in the cluster now for this purpose you definitely need some of the pods already running right so that

11:58

you have program basically gives you an output so if I do cctl get pods oops

12:05

cctl get pods I already have a few pods up and running and when we'll execute our program it should return all of

12:12

these particular pods right if we build it correctly okay so for our first task we are going to do is

12:18

read existing FS cool okay so before I move on I just wanted to know if you're

12:23

a beginner in goang right if you just have the knowledge of the basic syntax I would highly recommend to focus on two

12:29

topics one is structs to create your own types and other is pointers right

12:34

because these are the two things that we are going to extensively use when we are dealing with a library such as client go

12:40

or if you see any other open source projects out there you'll definitely find that these two particular concepts

12:46

are heavily used there right so having a knowledge of pointers and structs and types will definitely come in handy I'm

12:53

going to try my best to explain everything in detail we are going to call the client package and now we have a bunch of

13:00

structs here right so these are specific types so what we basically doing is as a first step we want the cun cluster

13:07

because client is now directly contacting with the cuties cluster right so as a first step we want is to

13:14

indicate the cuties cluster of the particular resource that we are concerned with right and in this case we

13:19

are concerned with the parts now if you are familiar with the cities API Basics if you're not I have a Blog on that

13:26

already written so you can check out the description box down below to to basically know about the common terminologies of the cuties API and how

13:33

it is structured but the Pod resource is basically found under the core V1 API

13:38

Group right so if I just type core V1 so we have a function core V1 which is

13:45

basically returning an interface so if you want to check out the interface you can just click on command and click and

13:51

this interface is essentially returning other interfaces but the bottom line is under this particular interface you'll

13:57

fil all the CU resources that are found under the core V1 API Group right so we

14:04

have notes we have persistent volumes we have replication controllers right we

14:10

have Services secrets so these are and config Maps so these are all the cuberes

14:15

resources or cuberes objects that are found under the core V1 API Group and now what we are concerned with is parts

14:22

right so here you'll see a pods getter interface right so pods getter is an

14:28

inter interace which basically is returning a pod function which is essentially returning all the pods right

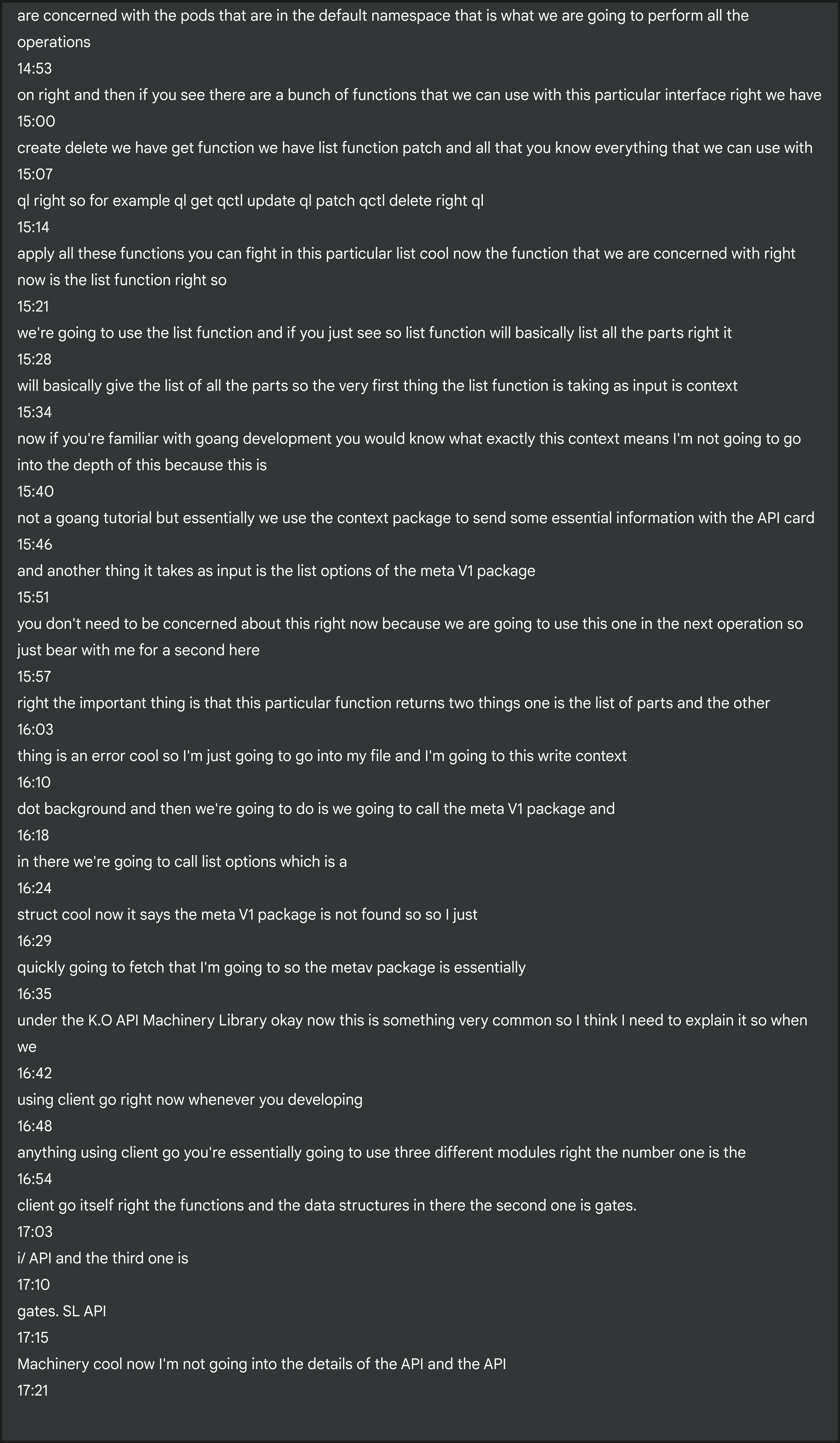
14:35

so what we're going to do is we're going to go here and we're going to just say pods right and this particular function

14:41

takes an input the name space that we are concerned with so I'll just go ahead and type in default right now because we

14:48



are concerned with the pods that are in the default namespace that is what we are going to perform all the operations

14:53

on right and then if you see there are a bunch of functions that we can use with this particular interface right we have

15:00

create delete we have get function we have list function patch and all that you know everything that we can use with

15:07

ql right so for example ql get qctl update ql patch qctl delete right ql

15:14

apply all these functions you can fight in this particular list cool now the function that we are concerned with right now is the list function right so

15:21

we're going to use the list function and if you just see so list function will basically list all the parts right it

15:28

will basically give the list of all the parts so the very first thing the list function is taking as input is context

15:34

now if you're familiar with goang development you would know what exactly this context means I'm not going to go into the depth of this because this is

15:40

not a goang tutorial but essentially we use the context package to send some essential information with the API card

15:46

and another thing it takes as input is the list options of the meta V1 package

15:51

you don't need to be concerned about this right now because we are going to use this one in the next operation so just bear with me for a second here

15:57

right the important thing is that this particular function returns two things one is the list of parts and the other

16:03

thing is an error cool so I'm just going to go into my file and I'm going to this write context

16:10

dot background and then we're going to do is we going to call the meta V1 package and

16:18

in there we're going to call list options which is a

16:24

struct cool now it says the meta V1 package is not found so so I just

16:29

quickly going to fetch that I'm going to so the metav package is essentially

16:35

under the K.O API Machinery Library okay now this is something very common so I think I need to explain it so when we

16:42

using client go right now whenever you developing

16:48

anything using client go you're essentially going to use three different modules right the number one is the

16:54

client go itself right the functions and the data structures in there the second one is gates.

17:03

i/ API and the third one is

17:10

gates. SL API

17:15

Machinery cool now I'm not going into the details of the API and the API

17:21

Machinery library right uh I link down a Blog in the description box down below so you can check them out and know more

17:27

but essentially API machery and the API libraries or the modules they contain some helper functions and helper data

17:34

structures that the client go actually uses right and in the end the user directly deals with the client go

17:40

library right but this is also essential to know that the client go library is dependent on these two modules now if

17:46

you're talking about developing Advanced applications with kubernetes right you're definitely going to need all of

17:51

these three packages and you definitely need to understand what is the/ API and

17:57

what is the/ API machinery and what are the different methods types and structures that these two contain right

18:03

but in this particular one I'm not going to go into the details of this I link a Blog in the description box so you can check it out cool so we're using the

18:09

meta V1 package from the API machine library and we are just going to fetch it here cool now as I mentioned

18:15

previously the list function returns two things right one is basically a pod list

18:21

which is a struct which is the information of all the parts right and this is what we're going to use to

18:26

basically get the names of all the parts and other thing is the error right I'm just going to write here pods

18:35

error and this is what it's returning and then we'll handle the error as

18:42

usual I'm just going to write Panic error. error cool and now this is what

18:49

we are dealing with here right so we have pods variable which is of type V1

18:54

pod list cool now in order to see what exactly does this pod

19:01

list type returns this is the field that we are interested in items field and

19:06

items field is basically a list of all the parts it's a slice of ports right so

19:11

in order for us to get the names of all these ports we need to access this particular items field right so what I'm

19:18

going to do is I'm going to create a loop right and I'm going to range over

19:24

the pods do item right because it's a slice and then

19:29

basically we're going to print so essentially I'm just printing the name of all the parts right so I think this

19:35

should be done right and let us execute our particular program so go run main.go and if

19:43

everything works it will print the names of all the pods and we have the names of all the pods if I just zoom in a bit so

19:50

you see these are all the pods that are currently running in my cuberes cluster we can also you know double check I'm

19:57

not telling lies so you can compare you know these are all the pods that are running and the program has successfully

20:03

worked cool so now we have listed down all the

20:08

pods okay cool let us move on to the create operation now right and see how

Create a Pod

20:14

the create operation would work so the read operation is pretty simple and now we understand a little

20:20

bit about what are the things that we need and what are the things that we need to configure now all the other operations are done in a very similar

20:26

fashion there are a few things that we need to configure here right so if we talk about the create operation there

20:32

are two things that we need to configure number one is the Pod

20:37

definition and please pardon my handwriting but I hope you get the point and in the p definition what essentially

20:43

we need is the P name right we need the container

20:50

name and we need the very very important one the container image that we're going

20:57

to use in that that particular pod right and after that we'll use this particular

21:03

pod definition file to create our

21:08

pod cool so these are the two steps let us do them one by one so what I'm going to do is I'm just

21:15

going to comment these lines because we don't need them right now and we're going to create a new section here which

21:22

is create a new pod cool and as a first step we are going to create a new pod

21:31

definition and how do we do it this is basically the way that we Define a part

21:37

definition when we are using the client go Library uh don't get overwhelmed I'm just going to explain the most important

21:42

parts but if you're familiar with the goang basics you would get an idea of what I'm doing here right but just to

21:48

explain and before I explain this I'm going to just go ahead and do go more

21:53

tidy so that all of our modules get surrendered and we don't have any red

21:58

Marks here cool so essentially what we're doing here is we are accessing the

22:04

Pod struct from directly from the V1 package right as I mentioned that the Pod resource is found under the core V1

22:10

API Group right so we have the V1 package under the k. i/ API module and

22:16

we are directly using the Pod struct from there right and in the Pod structs we have the following Fields right so we

22:23

have we have type meta we have object meta if you have worked with cubes right if you have created a a CU zml file you

22:29

would know what this field means so if you want to understand this more clearly right compare these particular fields to

22:35

what we use with the city zml files right so in under objects meta we Define

22:41

the name right then we have what do we have here we have the resource version

22:47

we have you know the create time stamp delete time stamp we have the annotations everything that you'll find

22:52

in a typical Community zaml file you'll find it in this particular struct so that's how under the old the API is

22:59

working cool so what we are doing is basically we are calling the object meta

23:04

struct right from the metav V1 package again right and then we are accessing two Fields the number one is generate

23:10

name so the generate name field is basically giving a custom name to our pod and in this case I'm going to change

23:16

the name of the Pod to let us say konal crud cool and then the name space

23:25

we are giving it as default right so it will create a pod with the name konal cud and then in what name space the

23:32

default name space the next thing we are defining is the spec itself right so if you are familiar with comml file this is

23:38

the spec field in which we are defining the containers that we are going to use right the name of the container is the

23:45

enginex container and the image we are using is enginex latest image cool just to show you what exactly does the part

23:51

spec struct contains it basically contains a slice of volume so you can Define the volumes you can Define in it

23:58

contain containers you can Define containers and that is what we are using right now so the containers field is

24:03

basically a slice right and in which you can basically add multiple containers but right now we are just adding one

24:09

single container engx container and which is using the enginex latest image so this is how we Define a p definition

24:16

right now the next step would be the second step is to use the p

24:23

definition right and we are going to create a new P now it's pretty similar

24:28

to how we read a particular pod so if you go back here what we did was we called the client we then called the C1

24:36

package we then called the pods function and we basically defined the name space here and then from the Pod interface we

24:43

called the list function right now in this particular case what we're going to do is we're going to call the create

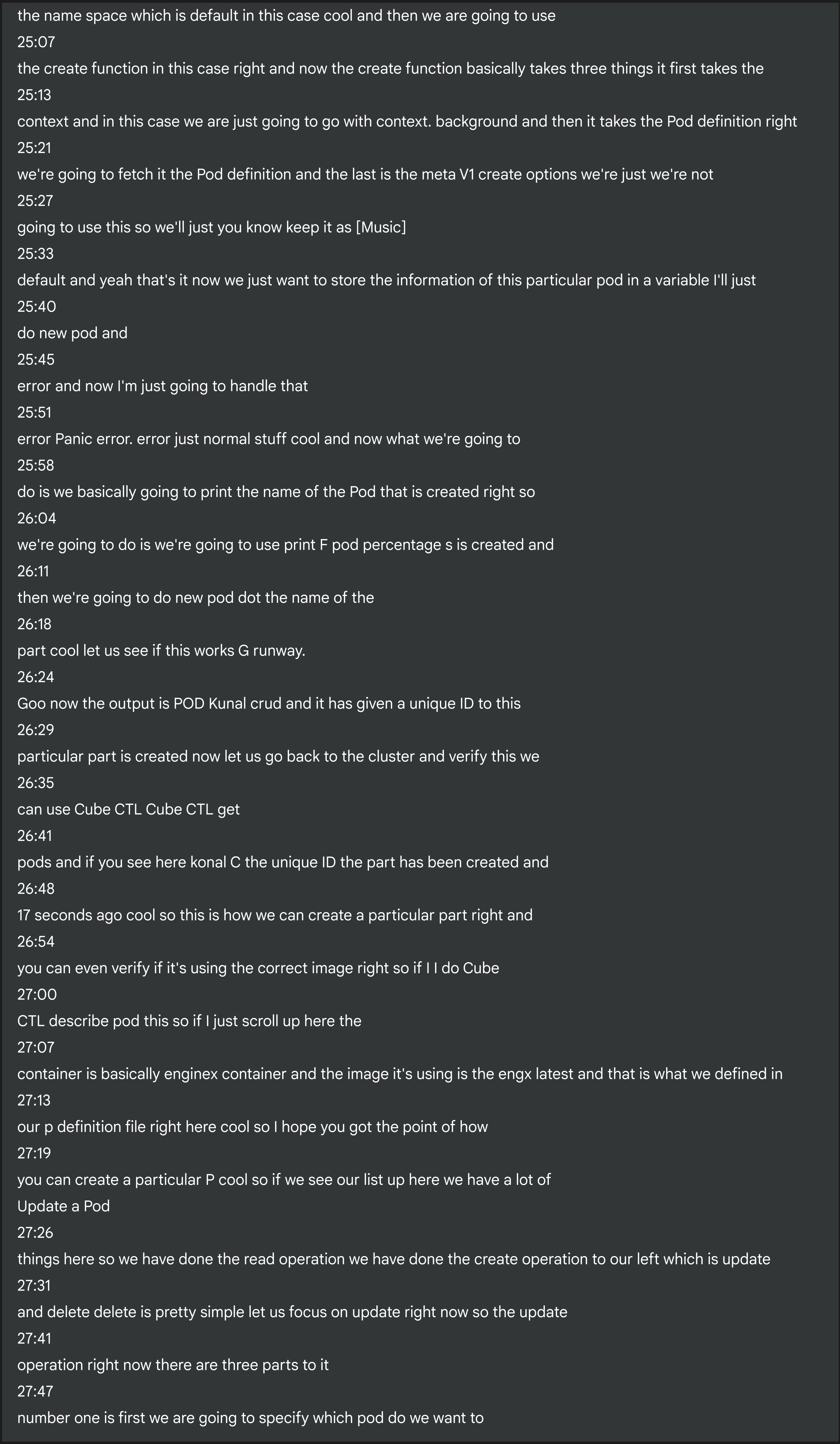
24:48

function cool so if I do

24:54

client doc V1 dop and then we'll basically fetch it with

25:00



the name space which is default in this case cool and then we are going to use

25:07

the create function in this case right and now the create function basically takes three things it first takes the

25:13

context and in this case we are just going to go with context. background and then it takes the Pod definition right

25:21

we're going to fetch it the Pod definition and the last is the meta V1 create options we're just we're not

25:27

going to use this so we'll just you know keep it as [Music]

25:33

default and yeah that's it now we just want to store the information of this particular pod in a variable I'll just

25:40

do new pod and

25:45

error and now I'm just going to handle that

25:51

error Panic error. error just normal stuff cool and now what we're going to

25:58

do is we basically going to print the name of the Pod that is created right so

26:04

we're going to do is we're going to use print F pod percentage s is created and

26:11

then we're going to do new pod dot the name of the

26:18

part cool let us see if this works G runway.

26:24

Goo now the output is POD Kunal crud and it has given a unique ID to this

26:29

particular part is created now let us go back to the cluster and verify this we

26:35

can use Cube CTL Cube CTL get

26:41

pods and if you see here konal C the unique ID the part has been created and

26:48

17 seconds ago cool so this is how we can create a particular part right and

26:54

you can even verify if it's using the correct image right so if I I do Cube

27:00

CTL describe pod this so if I just scroll up here the

27:07

container is basically enginex container and the image it's using is the engx latest and that is what we defined in

27:13

our p definition file right here cool so I hope you got the point of how

27:19

you can create a particular P cool so if we see our list up here we have a lot of

Update a Pod

27:26

things here so we have done the read operation we have done the create operation to our left which is update

27:31

and delete delete is pretty simple let us focus on update right now so the update

27:41

operation right now there are three parts to it

27:47

number one is first we are going to specify which pod do we want to

27:52

update specifying the Pod and of course the Pod should already exist previously

27:59

right the second is we going to Define what we need to

28:06

update so what we need to update in our particular case we're

28:11

going to keep this simple and we're going to just change the image version right the container image version so

28:17

right now we are using engine X latest right in our pod that we just created right now and we're going to

28:22

change the version to 1. 125.4 or the latest version any version that you want right

28:28

and then the last step would be to use the update function from the Pod

28:35

interface right so we first use the list function right from the interface we then use the create function and now

28:41

we're going to use the update function and while we are actually implementing this you'll understand more about what exactly does the update function takes

28:48

input as cool so let us first clean things up here I'm just going

28:54

to comment all of this because we don't need it right and our aim is to

29:00

basically update a f okay so this is basically how we are performing the

29:05

update operation right I know this is a lot of things I'm going to explain the very important parts of this and I'm

29:10

also going to leave some resources for you in the description box so that you can understand what we're doing here cool so what we're doing essentially is

29:17

first we are retrieving the latest pod or the Pod that we want to update cool and for that what we are doing is We are

29:23

following the same syntax and we are using the get function right and the get fun function basically takes three input

29:29

one is the context one is the name of the part that we are concerned with and the last one is the metav V1 doget

29:36

options right the part that we are concerned with right now is the konal c p that we created in the previous step

29:42

cool and I'm storing the information in the current pod variable cool so the

29:47

first step is we retriev the latest pod now what we want to do is we want to update the container image of this

29:54

particular P right so what we are doing is we are calling the current P variable we are accessing the spec field then we

30:00

are accessing the container at the zeroth index right because the container

30:06

is a slice right and we only have one image so if you have multiple containers so you can access it using you know

30:12

index numbers is for example if you have two containers and if you want to change the image of the second container you

30:17

can use you can use one here and this would basically access the information of the second container Cool right now

30:25
we are only using one container I'm just going to go with z and then we are accessing the image field of that particular container and
30:32
then we are giving it a new image version so here I'm using enginex
30:38
125.4 you can use any version just to show you all so we are basically changing from enginex
30:46
latest to enginex 1. 125.4 oops part four cool and that is
30:54
what we're doing here this is the second step right so we specified the Pod we
30:59
basically made the change right we basically updated our particular pod and now what we're doing is we are calling
31:04
the update function right and in here we are giving it the context we are giving it the current pod information right
31:11
because the current pod things have been updated right now right and then we are giving it with the meta 1 update options
31:18
right pretty similar stuff again we are storing this in the updated pod variable and then we are printing a line saying
31:24
that the Pod with the name has been updated cool now the interesting part
31:31
here is all this particular operation is been enclosed under the retry on
31:36
conflict function now what is a retry on conflict function or what is the retry
31:42
package right
31:47
retry so imagine this particular situation that you have multiple clients right and let us say we have client 1
31:54
client two and client three right now imagine a situation that the client one
32:00
request the cuties API to update a specific F let us say
32:05
pod one right now at the same time client 2
32:11
also requests the cuties API to update pod one and client 3 also requirs the
32:16
cuties API to update pod [Music] one now you can imagine the situation
32:23
that there would be a conflict right and essentially we need to handle this particular conflict right we need
32:30
to tell the communties API how to handle when multiple clients are requesting or
32:36
are updating the same resource right and that is where we use the retry package
32:41
and specifically the retry on conflict function now what this particular function would do is let us say we are
32:48
using the retry function and entire update operation is under this retry function now let us say at time T1
32:55
client 1 and client 2 tries to update cre the same resource P one cool so what
33:02
this particular function will do is it will perform a retry

33:08
mechanism that is it will first fail and then retry the operation

33:15
again cool so imagine this situation that client one requests a pod and client to request a pod again this

33:21
particular operation would fail because both the clients are requesting simultaneously once the particular operations fail right the retry

33:29
mechanism with will retry the operation again it will first randomly give the priority to a certain client let us say

33:35
client one so the client one can perform its operation right and then it will give priority to client two so then

33:41
client two will perform the operations so essentially we are handling a situation where conflicts between

33:47
multiple clients can happen right now I basically gave you a very bird's eye overview of how this retra mechanism

33:53
works but if you want to know a little bit more in detail I'm going to link resource in the description box down

33:59
below so you can check it out and you can understand how this particular retra mechanism is working in detail

34:05
right but just for now understand that we are in a way handling error that could occur while updating our P

34:12
resource cool so let us execute this particular program we'll go run main.go

34:18
and it said the Pod has been updated let us verify this so if we go Cube CTL

34:26
we'll go describe right and the image version is changed

34:31
to enginex 1.254 cool so the engx image version

34:37
First was latest and now we changed it to 1. 125.4 cool so that is how we update a

34:44
particular pod cool so now we did the create operation we did the update operation we also did the read operation

Delete a Pod

34:50
right and the last but not the least is the delete operation which is pretty pretty simple it is very similar to the

34:56
read operation because in the operation we were using the list function right

35:02
and in the delete operation we are going to directly use the delete function from

35:08
the P interface right if you remember that so essentially what we are doing is we

35:16
are calling the client again and then we are calling core V1 the pods function from there we are

35:24
fetching the pods which variable one thing you'll definitely notice is that I have been

35:30
calling this particular chain again and again to simplify this particular process you can save it in a variable

35:35
for example I save it in let us say p

35:44
client right and then it's just simply we'll directly call the P client and we

35:50

can access all the functions of a particular part this is something that you can Define at the start of your program and you can use it again and

35:56

again cool now as I mentioned we'll be using the delete function and again we just going to give it the

36:02

context and we'll specify the Pod that we want to delete right I'm just going to fetch K get

36:10

pods I'm just going to you know you can select any part from here let us you know delete this particular pod right

36:17

I'm just going to delete this one and then we are fetching The Meta V1 delete

36:23

options from here cool now the delete function only

36:28

Returns the error right so I'm just going to write delete

36:34

error right and I'm then just going to handle the delete

36:43

error I don't know let us say delete error. error yeah and we can you know put this

36:50

inside a panic function cool so yeah this is how we can basically perform the delete operation let us just execute the

36:57

program go and main.go it has successfully executed and if we list

37:03

down the pods again so you can see that the demo what

37:08

was the name again yeah the demo crud 55 uh pod has been deleted successfully

37:15

cool so now we have performed all the basic you know cred operations right we

37:23

performed the create operation read operation update operation and the delete operation and I hope that you got

Outro

37:29

a very basic idea of how is it working directly with the communties API using a

37:35

client library right in this particular one we use the Cent go but essentially you can use any other Library depending

37:40

on the language that you're using right most of the code and when we are dealing with cuberes or any other Cloud native

37:46

tool we often talk in terms of goang but the cuberes community has created other

37:51

client libraries as well right so you don't need to worry about that yeah I think that is it from this particular

37:57

video I hope you at least learn one thing new right that is all my aim with every video that I create and I'll also

38:03

leave a few links in the description box down below so make sure to check them out yeah and if you want more videos

38:09

like this more Hands-On stuff and you want me to create a full-fledged application using goang with kubernetes

38:15

you can definitely leave a comment down below and I'll try my best to make a video on that particular topic but yeah that's it for today and see you in the

38:21

next one

Hey there! It's awesome that you're diving into **client-go**. Since you're a 2nd-year student moving into DevOps and MLOps, mastering this library is a massive "level-up." It's the difference between just *using* Kubernetes and *building* the tools that manage it.

Since you're new to Go, I'll break down the syntax as we go. We will follow the transcript's flow but add "god-level" detail to ensure you understand every line.

🌐 Part 1: The Foundation – Setup & Authentication

Before we can tell Kubernetes to do anything, we need to speak its language. Kubernetes is essentially a giant **REST API**. **Client-go** is the official Go SDK that wraps those API calls into easy-to-use Go functions.

🛠️ Step 1: Environment Readiness

You need two things running before we write a single line of code:

1. **A Kubernetes Cluster:** Use **Minikube** or **Kind** (transcript recommends Minikube).
2. **Go Installed:** Ensure you have Go 1.20+ installed.

📂 Step 2: Project Initialization

Open your terminal and create a new directory for this series.

```Bash
mkdir k8s-client-demo && cd k8s-client-demo
# Initialize the module (replaces 'example' with your own path)
go mod init github.com/yourusername/client-demo
```

Now, we need to install the **Client-go** library and its "machinery" (helper tools).

```Bash
go get k8s.io/client-go@latest
go get k8s.io/apimachinery@latest
go get k8s.io/api@latest
```

🔐 Step 3: The "Magic" Boilerplate (Authentication)

To talk to the cluster, your code needs "credentials." Usually, these are stored in your `~/.kube/config` file.

Create a file named `main.go`. We will write the code to find that file and create a "Clientset."

```Go
package main

import (
        "context"
        "fmt"
        "os"
        "path/filepath"

        metav1 "k8s.io/apimachinery/pkg/apis/meta/v1"
        "k8s.io/client-go/kubernetes"
        "k8s.io/client-go/tools/clientcmd"
)

func main() {
        // 1. Find the Kubeconfig file
        // We use the filepath package to handle different OS path separators
```

```
	userHomeDir, _ := os.UserHomeDir()
	kubeconfigPath := filepath.Join(userHomeDir, ".kube", "config")

	// 2. Build the config object from the file
	// clientcmd.BuildConfigFromFlags(masterUrl, kubeconfigPath)
	config, err := clientcmd.BuildConfigFromFlags("", kubeconfigPath)
	if err != nil {
		fmt.Printf("Error building config: %s\n", err.Error())
		os.Exit(1)
	}

	// 3. Create the Clientset
	// The clientset is the entry point for all K8s resources (Pods, Nodes, etc.)
	clientset, err := kubernetes.NewForConfig(config)
	if err != nil {
		panic(err.Error())
	}

	// We are now authenticated! Let's do something with it.
	listPods(clientset)
}
```

> [!NOTE] **What is a Clientset?** Think of it as a "Master Controller." It contains sub-clients for every API group (like `CoreV1` for Pods or `AppsV1` for Deployments).

📄 Step 4: Your First Operation – Listing Pods

Now that we have the `clientset`, let's list all Pods in the `default` namespace. This is the **Read** part of CRUD.

Add this function below your `main` function:

Go

```
func listPods(clientset *kubernetes.Clientset) {
	fmt.Println("🚀 Fetching Pods...")

	// podsClient targets the CoreV1 API group, specifically Pods in the 'default' namespace
	podsClient := clientset.CoreV1().Pods("default")

	// List returns a PodList struct
	// context.Background() is a way to handle timeouts/cancellation
	// metav1.ListOptions{} allows us to filter or label-select (leaving empty for now)
	pods, err := podsClient.List(context.Background(), metav1.ListOptions{})
	if err != nil {
		panic(err.Error())
	}

	fmt.Printf("There are %d pods in the default namespace\n", len(pods.Items))

	// Iterate through the slice of Pods
	for _, pod := range pods.Items {
		fmt.Printf("- Pod Name: %s | Status: %s\n", pod.Name, pod.Status.Phase)
	}
}
```

🧬 Breaking Down the Go Syntax for Beginners

Since you're starting with Go, here are three things happening in the code above that are crucial:

1. **Pointers (`*`):** Notice `clientset *kubernetes.Clientset`. We pass a pointer to the clientset so the function doesn't copy the whole massive object into memory. It just looks at the address.
2. **Error Handling:** In Go, functions return the result **and** an error. We check `if err != nil` constantly. This is the "Go way" of being safe.
3. **Context:** `context.Background()` is passed to almost every network call in Go. It's like a "signal" that tells the program, "Keep trying until this is done or the program stops."

🚀 Running Your Code

1. Start your cluster: `minikube start`
2. Create a dummy pod so you have something to see: `kubectl run nginx --image=nginx`
3. Run your Go program:

Bash

```bash
go run main.go
```

Output should look like:

Plaintext

```
🚀 Fetching Pods...
There are 1 pods in the default namespace
- Pod Name: nginx | Status: Running
```

📝 Summary of Part 1

- **Authentication:** We used `clientcmd` to load our local kubeconfig.
- **Clientset:** We initialized the main hub for API communication.
- **API Groups:** We learned that Pods live under `CoreV1()`.
- **List:** We used the `List` method to fetch resources into a Go slice.

In Part 2, we will move to "Create" and "Delete" operations, where you'll learn how to define a Pod's YAML-like structure directly in Go code!

Would you like me to move on to Part 2 (Creating resources) now, or do you have any questions about the authentication logic?

next

Awesome! Now that we can talk to the cluster and list what's inside, let's learn how to **Create** and **Delete** resources.

In this part, you'll see how the YAML files you usually write for Kubernetes translate directly into Go **structs**.

🏗️ Part 2: Creation & Deletion – Building and Destroying Resources

When we use `kubectl apply -f pod.yaml`, Kubernetes takes that YAML and converts it to JSON. When using **client-go**, we skip the YAML and build a Go "Object" directly.

🧩 Step 1: Understanding the Pod "Struct"

In Go, a Pod is defined by a massive `struct` (a collection of fields). The two most important parts are:

1. **ObjectMeta**: The "Metadata" (Name, Namespace, Labels).
2. **Spec**: The "Specification" (What containers? What images?).

🚀 Step 2: Coding the Creation

Open your `main.go` and let's add a new function called `createPod`.

[!TIP] **Colors & Emojis Checklist:** > 🟢 **Green** for success paths. 🔴 **Red** for error handling. 🔵 **Blue** for definitions.

Go

```go
import (
        // Add this to your imports
        corev1 "k8s.io/api/core/v1"
)

func createPod(clientset *kubernetes.Clientset) {
        fmt.Println("🏗️ Creating a new Pod...")

        // 🔵 Define the Pod Object
        newPod := &corev1.Pod{
                ObjectMeta: metav1.ObjectMeta{
                        Name: "client-go-pod", // The name of your pod
                        Labels: map[string]string{
                                "app": "demo",
                        },
                },
                Spec: corev1.PodSpec{
                        Containers: []corev1.PodContainer{
                                {
                                        Name:  "web-container",
                                        Image: "nginx:latest",
                                        Ports: []corev1.ContainerPort{
                                                {ContainerPort: 80},
                                        },
                                },
                        },
                },
        }

        // 🚀 Execute the Creation
        // We use the same podsClient logic as before
        podsClient := clientset.CoreV1().Pods("default")

        result, err := podsClient.Create(context.Background(), newPod, metav1.CreateOptions{})

        if err != nil {
                // 🔴 Error Handling
                fmt.Printf("❌ Failed to create Pod: %v\n", err)
                return
        }

        // 🟢 Success!
        fmt.Printf("✅ Pod created successfully: %s\n", result.Name)
}
```

🗑️ Step 3: Coding the Deletion

Deleting is much simpler because you only need the **Name** of the resource.

Go

```go
func deletePod(clientset *kubernetes.Clientset, podName string) {
        fmt.Printf("🗑️ Deleting Pod: %s...\n", podName)

        podsClient := clientset.CoreV1().Pods("default")

        // metav1.DeleteOptions allows you to set "GracePeriodSeconds" (e.g., force delete)
        err := podsClient.Delete(context.Background(), podName, metav1.DeleteOptions{})

        if err != nil {
                // 🔴 Error Handling
                fmt.Printf("❌ Failed to delete Pod: %v\n", err)
                return
        }

        // 🟢 Success!
        fmt.Printf("✅ Pod %s deleted!\n", podName)
}
```

🧬 "Beginner-Go" Concept: Nested Structs

If you look at the `newPod` definition above, it looks like a "Tree."

- `corev1.Pod` is the root.
- Inside it, `Spec` is another struct.
- Inside `Spec`, `Containers` is a **Slice** (like an array) of `PodContainer` structs.

Why is it a slice? Because a Pod can have more than one container (Sidecars)! Even if you only want one, you must put it inside `[]corev1.Container{ ... }`.

🛠️ Updating your `main()` to test:

Update your `main` function to call these in order:

Go

```go
func main() {
        // ... (Your authentication code from Part 1) ...

        // 1. Create it
        createPod(clientset)

        // 2. Wait a second so you can see it in terminal
        fmt.Println("Waiting 5 seconds for you to check 'kubectl get pods'...")
        // (You can add time.Sleep(5 * time.Second) here if you want)

        // 3. Delete it
        deletePod(clientset, "client-go-pod")
}
```

📝 Summary of Part 2

- **Object Definition**: We learned that K8s resources are just deeply nested Go structs.
- **Creation**: Use `.Create()` with a pointer to your struct (`&corev1.Pod`).
- **Deletion**: Use `.Delete()` using just the string name.
- **Namespaces**: Notice how we keep using `.Pods("default")`. If you change that to `"kube-system"`, you'd be targeting a different part of the cluster!

Coming up in Part 3: We will tackle **Updating** resources. This is the "Advanced" part of basic CRUD because of something called **Conflicts** (what happens when two people try to update the same pod at once?).

Ready for the "Conflict/Retry" logic in Part 3?

next

This is where things get a bit more "real world." In Kubernetes, you rarely just create and delete things; you **update** them—changing a container image, adding a label, or scaling a deployment.

🔄 Part 3: Updating Resources & The "Conflict" Problem

In Kubernetes, updates follow a **"Read-Modify-Write"** pattern. You can't just send a command saying "Change image to X." Instead, you must:

1. **GET** the current version of the object from the cluster.
2. **MODIFY** the local copy in your Go code.
3. **UPDATE** (Write) it back to the cluster.

⚠️ The "Conflict" Trap (Optimistic Concurrency)

Kubernetes uses **ResourceVersions**. If you GET a pod (v1), and while you are modifying it, someone else updates it (v2), your UPDATE call will fail because your local copy is outdated.

To solve this, we use the `retry` package from `client-go`.

🛠️ Step 1: Installing the Retry Package

You'll need a specific utility package for this. Run this in your terminal:

Bash

```bash
go get k8s.io/client-go/util/retry
```

🚀 Step 2: Coding the Update (The Pro Way)

We are going to update the Pod we created in Part 2. We'll change its image from `nginx:latest` to `nginx:1.21.0`.

Go

```go
import (
        // Add these to your imports
        "k8s.io/client-go/util/retry"
    "fmt"
)

func updatePodImage(clientset *kubernetes.Clientset, podName string) {
        fmt.Printf("🔄 Attempting to update image for Pod: %s...\n", podName)

        podsClient := clientset.CoreV1().Pods("default")

        // 🔵 The Retry Loop
        // RetryOnConflict will re-run the code inside if the cluster says "version conflict"
        err := retry.RetryOnConflict(retry.DefaultRetry, func() error {

                // 1. GET the latest version of the pod
                result, getErr := podsClient.Get(context.Background(), podName, metav1.GetOptions{})
                if getErr != nil {
                        return getErr // If we can't find it, stop retrying
                }

                // 2. MODIFY the local copy
                // We access the first container (index 0) and change its image
                result.Spec.Containers[0].Image = "nginx:1.21.0"

                // 3. UPDATE the pod in the cluster
                _, updateErr := podsClient.Update(context.Background(), result, metav1.UpdateOptions{})
                return updateErr
        })

        if err != nil {
                // 🔴 Error Handling
                fmt.Printf("❌ Update failed: %v\n", err)
                return
        }

        // 🟢 Success!
        fmt.Println("✅ Pod image updated to nginx:1.21.0 successfully!")
}
```

🧬 Beginner-Go Concept: Anonymous Functions (Closures)

Notice this syntax: `func() error { ... }` inside the `RetryOnConflict` call. This is an **Anonymous Function**.

- In Go, functions are "First Class," meaning you can pass a function into another function as an argument.
- The `RetryOnConflict` function takes our "instructions" and runs them. If it fails due to a conflict, it simply runs our function again.

🔍 Step 3: Verifying the Change

After running your code, you can verify it in your terminal:

Bash

```bash
kubectl describe pod client-go-pod | grep Image:
```

You should see it changed from `latest` to `1.21.0`.

📝 Summary of Part 3

- **Optimistic Concurrency**: Kubernetes uses `resourceVersion` to prevent users from overwriting each other's changes.
- **Get-Modify-Update**: This is the standard loop for making changes safely.
- **Retry Utility**: `retry.RetryOnConflict` is your best friend. It automates the "oops, someone else changed it, let me try again" logic.

💡 DevOps Tip

In a real production environment (like MLOps pipelines), many controllers might be touching the same Pod at once (Logging agents, Sidecar injectors, etc.). Without this `RetryOnConflict` logic, your automation scripts will crash constantly!

Part 4 is where we get into the "Big Leagues": In-Cluster Authentication. Right now, our code only works on your laptop because it looks for a `.kube/config` file. But how does code running **inside** a Kubernetes Pod talk to the API?

Ready to move your code from your laptop into the cluster?

next

This is a crucial turning point for any DevOps or MLOps engineer. So far, your Go code has been running on your **local machine** (Laptop), using your personal `kubeconfig` to talk to the cluster.

But in the real world, your AI models, scrapers, or automation tools will run **inside a Pod** within the cluster. A Pod doesn't have a `~/.kube/config` file. Instead, it uses a **ServiceAccount**.

🛸 Part 4: In-Cluster Config & Service Accounts

When your code runs inside Kubernetes, it retrieves its "identity" from a token located at a specific file path: `/var/run/secrets/kubernetes.io/serviceaccount/token`.

`client-go` makes this transition incredibly easy with a function called `InClusterConfig`.

🛠️ Step 1: The "Smart" Authentication Code

We want our code to be "smart"—if it's running on your laptop, it should use the local file; if it's in a Pod, it should use the internal token.

Update your `main.go` authentication logic:

```go
func getClient() (*kubernetes.Clientset, error) {
        // 1. Try to get In-Cluster config first (running inside a Pod)
        config, err := rest.InClusterConfig()
        if err != nil {
                fmt.Println("🌐 Not running inside cluster, falling back to kubeconfig...")

                // 2. Fallback to Local Kubeconfig (running on Laptop)
                userHomeDir, _ := os.UserHomeDir()
                kubeconfigPath := filepath.Join(userHomeDir, ".kube", "config")
                config, err = clientcmd.BuildConfigFromFlags("", kubeconfigPath)
                if err != nil {
                        return nil, err
                }
        }

        // 🔵 Create the Clientset from whichever config worked
        return kubernetes.NewForConfig(config)
}
```

📦 Step 2: Preparing for Deployment (Dockerizing)

Since you are interested in MLOps, you know that to run this in K8s, we need to containerize it.

1. **Build the Go Binary:**

   ```bash
   # We use CGO_ENABLED=0 to ensure the binary is static and works in tiny images
   GOOS=linux GOARCH=amd64 CGO_ENABLED=0 go build -o k8s-app main.go
   ```

2. **Create a `Dockerfile`:**

   ```dockerfile
   FROM alpine:latest
   COPY k8s-app /k8s-app
   ENTRYPOINT ["/k8s-app"]
   ```

3. **Build and Push:** (Assume you are using Minikube's local registry or Docker Hub).

📜 Step 3: RBAC (The "Permissions" Wall)

Even if your code is running inside a Pod, Kubernetes will **block** it by default. You must give your Pod "Permission" to see other Pods. This is called **RBAC** (Role-Based Access Control).

You need to apply this YAML to your cluster before your Go app will work:

```yaml
# 🔵 1. The Identity
apiVersion: v1
kind: ServiceAccount
metadata:
  name: pod-reader-sa

---
```

```yaml
# 🔴 2. The Permission (The "What")
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: pod-reader-role
rules:
- apiGroups: [""] # "" indicates the core API group (where Pods live)
  resources: ["pods"]
  verbs: ["get", "list", "watch"] # Permissions needed for our code

---
# 🟢 3. The Connection (The "Who")
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: read-pods-binding
subjects:
- kind: ServiceAccount
  name: pod-reader-sa
roleRef:
  kind: Role
  name: pod-reader-role
  apiGroup: rbac.authorization.k8s.io
```

🧬 Beginner-Go Concept: Error Interfaces

In the `getClient` function, notice how we handle the error from `rest.InClusterConfig()`. In Go, an error is just an interface. If it's `nil`, the call succeeded. If it's not `nil`, it tells us *why* it failed (in this case, "Environment variables not found"). This allows us to "try one way, then try another" gracefully.

📝 Summary of Part 4

- `InClusterConfig()`: Allows code to authenticate using the Pod's internal ServiceAccount token.
- **Portability**: Writing "Smart Authentication" allows the same binary to work on your dev machine and in production.
- **RBAC**: Without a `Role` and `RoleBinding`, your `client-go` code will get a "403 Forbidden" error from the API server.

🚀 Next Step: The "God-Level" Feature

Up until now, our code is "Request-Response." We ask for pods, we get them, then the program ends. But real DevOps tools (like ArgoCD or Prometheus) **watch** the cluster in real-time.

In Part 5, we will learn about "Watch" and "Informers"—the secret sauce behind how Kubernetes controllers react to changes instantly.

Ready to build a real-time monitor?

next

This is where you transition from writing simple scripts to building actual **Kubernetes Controllers**.

In the previous parts, we used `List()`. The problem with `List()` is that it's a "snapshot." If a Pod dies a second after you list it, your data is stale. To build high-performance MLOps tools, you need to react the **instant** something changes.

🕵️ Part 5: Informers & The "Watch" Mechanism

Instead of constantly polling the API (which is heavy and slow), `client-go` provides **Informers**. An Informer maintains a local, lightning-fast cache of the cluster state and notifies you via "Handlers" whenever something happens.

🛠️ Step 1: The Informer Pattern

To use an Informer, we use a **SharedInformerFactory**. This is efficient because if you have multiple parts of your code watching Pods, they all share one single connection to the API.

```go
import (
	"fmt"
	"time"

	"k8s.io/client-go/informers"
	"k8s.io/client-go/tools/cache"
)

func startPodWatch(clientset *kubernetes.Clientset) {
	// 1. Create a factory that refreshes every 30 seconds
	factory := informers.NewSharedInformerFactory(clientset, time.Second*30)

	// 2. Get the informer for Pods
	podInformer := factory.Core().V1().Pods().Informer()

	// 3. Define the "Event Handlers" (The logic for what to do on changes)
	podInformer.AddEventHandler(cache.ResourceEventHandlerFuncs{
		// 🟢 Triggered when a new Pod is created
		AddFunc: func(obj interface{}) {
			pod := obj.(*corev1.Pod)
			fmt.Printf("🆕 [EVENT] Pod Added: %s\n", pod.Name)
		},
		// 🟡 Triggered when a Pod is updated (e.g., image change, status change)
		UpdateFunc: func(oldObj, newObj interface{}) {
			newPod := newObj.(*corev1.Pod)
			oldPod := oldObj.(*corev1.Pod)
			fmt.Printf("🔄 [EVENT] Pod Updated: %s (Status: %s -> %s)\n",
				newPod.Name, oldPod.Status.Phase, newPod.Status.Phase)
		},
		// 🔴 Triggered when a Pod is deleted
		DeleteFunc: func(obj interface{}) {
			pod := obj.(*corev1.Pod)
			fmt.Printf("❌ [EVENT] Pod Deleted: %s\n", pod.Name)
		},
	})

	// 4. Start the informer
	stopper := make(chan struct{})
	defer close(stopper)

	fmt.Println("🛰️ Real-time Pod Monitor Started...")
	factory.Start(stopper)

	// Keep the function running so it keeps watching
	<-stopper
}
```

🧬 Beginner-Go Concept: Type Assertion

In the handlers above, you see `obj.(*corev1.Pod)`.

- The Informer is generic; it sends data as an `interface{}` (which means "it could be anything").
- Since we know we are watching Pods, we use **Type Assertion** to tell Go: "Treat this object as a Pod struct so I can access its fields like `.Name`."

🚀 Step 2: Why this is "God-Level" for MLOps

Imagine you are building an **Auto-Scaling system for GPU jobs**:

1. Your Informer watches for new Pods with a specific label like `task: training`.

2. The `AddFunc` triggers.
3. Your Go code checks if there are enough GPUs.
4. If not, your code calls the Cloud Provider API to spin up a new node.

This is exactly how tools like **Karpenter** or **KubeFlow** work under the hood!

📝 Summary of Part 5

- **Poll vs. Watch**: Polling is "Are we there yet?"; Watching is "Tell me when we arrive."
- **Informer Factory**: Manages the lifecycle of multiple informers.
- **Handlers**: The specific functions that execute your business logic on `Add`, `Update`, or `Delete`.
- **Shared Cache**: Informers reduce the load on the K8s API server by keeping a local copy of data.

🏁 Final Part Preview: Part 6 – Labels, Selectors, and Filters

In a real cluster, there are thousands of Pods. You don't want to watch *everything*.

In the final part, we will learn how to use Label Selectors to filter exactly which resources your client-go code cares about (e.g., "Only show me Pods that belong to my MLOps project").

Would you like to wrap up with the final part on Filtering?

next

We've reached the final piece of the puzzle! In a real Kubernetes cluster (especially in DevOps/MLOps), you might have thousands of Pods running. Your application shouldn't be "listening" to every single one of them. That would be noisy and waste memory.

Instead, you need to use **Label Selectors** to filter the noise and focus only on the resources that belong to your project.

🎯 Part 6: Filtering with Label Selectors

In this final part, we will combine everything we've learned to create a "Scoping" mechanism. We will tell `client-go` to only interact with Pods that have a specific label, like `env: production` or `team: mlops`.

🛠️ Step 1: List with Selectors

When using the `.List()` or `.Watch()` methods, we've been passing an empty `metav1.ListOptions{}`. Now, we will fill that struct with a **LabelSelector**.

Go

```go
func listFilteredPods(clientset *kubernetes.Clientset) {
        // 🔵 1. Define the Selector
        // This is the Go equivalent of: kubectl get pods -l team=mlops
        labelSelector := "team=mlops"

        listOptions := metav1.ListOptions{
                LabelSelector: labelSelector,
        }

        fmt.Printf("🔍 Searching for Pods with label: %s...\n", labelSelector)

        // 🚀 2. Execute the List with the filter
        pods, err := clientset.CoreV1().Pods("default").List(context.Background(), listOptions)
        if err != nil {
                fmt.Printf("❌ Error: %v\n", err)
                return
        }

        for _, pod := range pods.Items {
                fmt.Printf("✅ Found Project Pod: %s\n", pod.Name)
        }
}
```

🧪 Step 2: Advanced Filtering in Informers

If you are building a controller (like we did in Part 5), you can bake the filter directly into the **Informer Factory**. This is much more efficient because the API server will only send updates for Pods that match your criteria.

```Go
func startScopedInformer(clientset *kubernetes.Clientset) {
        // 🔵 Use TweakListOptions to inject a global filter into the Informer
        factory := informers.NewSharedInformerFactoryWithOptions(
                clientset,
                time.Second*30,
                informers.WithNamespace("default"),
                informers.WithTweakListOptions(func(options *metav1.ListOptions) {
                        options.LabelSelector = "app=echo-learn" // Specific to your project!
                }),
        )

        // Now this informer ONLY sees 'echo-learn' pods
        podInformer := factory.Core().V1().Pods().Informer()

    // ... rest of the handler logic from Part 5 ...
}
```

🧬 Beginner-Go Concept: Variadic Functions

Notice `informers.NewSharedInformerFactoryWithOptions(...)`. The `...` in the documentation for this function means it is **Variadic**.

- You can pass zero, one, or ten "Option" functions to it.
- This is a common Go pattern called **Functional Options**. It makes your code very readable and easy to extend without breaking older parts of your application.

🎓 Summary of the 6-Part Series

Part	Concept	Key Takeaway
1	**Auth & List**	`clientcmd` gets you in; `Clientset` is your master hub.
2	**Create & Delete**	K8s resources are just Go `structs`.
3	**Update & Conflict**	Always use `RetryOnConflict` for safe updates.
4	**In-Cluster Auth**	`rest.InClusterConfig()` is for code running inside Pods.
5	**Informers**	Real-time monitoring is faster than polling.
6	**Selectors**	Use `LabelSelectors` to keep your tools efficient.

🏆 Your Next Steps as a DevOps/MLOps Developer

Since you're building **"EchoLearn"** (the AI assistant for deaf students) and **"K8s-Genesis"**, here is how you can apply this immediately:

1. **Dynamic Config:** Use `client-go` to watch a `ConfigMap`. When you update the ConfigMap with new AI model parameters, your Go app can detect it via an **Informer** and reload the model without restarting the Pod!
2. **Job Monitoring:** If you're running heavy ML training jobs, use a `client-go` tool to watch the `Status` of your Job pods and send an alert if they fail.

Would you like me to help you write a specific "Controller" script for one of your projects, like EchoLearn or K8s-Genesis?

when we get pod details , how does the output look like? can you also tell how to get specific things like ip, ports and other things

Since you are starting with Go, it’s important to understand that `client-go` doesn't return a "string" or "text"—it returns a complex **Go Object (Struct)**.

When you call `pods, err := clientset.CoreV1().Pods("default").Get(...)`, you get back a pointer to a `corev1.Pod` object.

📦 1. What does the "Raw" Pod Object look like?

If you were to print the entire object using Go's "plus-v" formatting (`fmt.Printf("%+v", pod)`), it would look like a giant tree of data. Here is the conceptual structure:

Go

```go
Pod {
    TypeMeta:   {Kind: "Pod", APIVersion: "v1"},
    ObjectMeta: {Name: "my-pod", Namespace: "default", Labels: {...}, ...},
    Spec:       {
        Containers: [
            {Name: "nginx", Image: "nginx:latest", Ports: [...], ...}
        ],
        NodeName: "minikube",
        ...
    },
    Status:     {
        Phase: "Running",
        PodIP: "10.244.0.5",
        HostIP: "192.168.49.2",
        ContainerStatuses: [...],
        ...
    },
}
```

🔍 2. How to extract specific details?

In Go, you access these nested fields using the **dot notation**. Since many fields are **slices** (arrays) or **pointers**, you have to be careful about how you access them.

Here is the "God-level" breakdown of how to get the most common details:

🌐 Pod IP and Node Name

These live inside the `Status` and `Spec` sections respectively.

Go

```go
// The Pod's internal IP address
podIP := pod.Status.PodIP

// The name of the Worker Node the pod is running on
```

```
nodeName := pod.Spec.NodeName

fmt.Printf("📍 IP: %s | Node: %s\n", podIP, nodeName)
```

🔌 Ports and Images

Since a Pod can have multiple containers, you usually need to loop through the `Spec.Containers` slice.

```go
for _, container := range pod.Spec.Containers {
    fmt.Printf("📦 Container: %s\n", container.Name)
    fmt.Printf("🖼️  Image: %s\n", container.Image)

    // Ports are also a slice (a container can have multiple ports)
    for _, port := range container.Ports {
        fmt.Printf("⚓ Port: %d (%s)\n", port.ContainerPort, port.Protocol)
    }
}
```

🏷️ Labels and Annotations

Labels are stored as a `map[string]string`. You can look for a specific key.

```go
// Check for a specific label
appLabel, exists := pod.Labels["app"]
if exists {
    fmt.Printf("🏷️ App Label: %s\n", appLabel)
}

// Iterate through all labels
for key, value := range pod.Labels {
    fmt.Printf("Label: %s = %s\n", key, value)
}
```

🛠️ 3. Complete "Detail Fetcher" Example

Here is a ready-to-use function that extracts the most important "DevOps" info from a Pod:

```go
func printDetailedPodInfo(pod *corev1.Pod) {
        fmt.Println("-----------------------------------------")
        fmt.Printf("🆔 NAME:      %s\n", pod.Name)
        fmt.Printf("🟢 STATUS:    %s\n", pod.Status.Phase)
        fmt.Printf("🌐 POD IP:    %s\n", pod.Status.PodIP)
        fmt.Printf("🖥️  NODE:      %s\n", pod.Spec.NodeName)

        // Accessing first container image safely
        if len(pod.Spec.Containers) > 0 {
                fmt.Printf("🖼️  IMAGE:     %s\n", pod.Spec.Containers[0].Image)
        }

        // Calculating Restart Count
        // This lives in Status -> ContainerStatuses
        if len(pod.Status.ContainerStatuses) > 0 {
                restarts := pod.Status.ContainerStatuses[0].RestartCount
                fmt.Printf("🔄 RESTARTS:  %d\n", restarts)
        }
        fmt.Println("-----------------------------------------")
}
```

⚠️ Pro-Tip: Dealing with "Nil" Pointers

When you are a beginner in Go, you will run into "Panic" errors if you try to access a field that doesn't exist yet.

> **Example:** If a Pod is still in `Pending` state, it might not have a `pod.Status.PodIP` yet. Always check if the pod is actually `Running` before you rely on the IP address!

```go
if pod.Status.Phase == corev1.PodRunning {
    fmt.Println("IP is:", pod.Status.PodIP)
} else {
    fmt.Println("Pod is not running yet, IP not assigned.")
}
```

🧠 DevOps Context: Why does this matter?

In your **K8s-Genesis** project or any MLOps pipeline, you might need to:

1. **MLOps:** Get the Pod IP to send a "Start Training" signal to a specific worker.
2. **DevOps:** Check the `RestartCount`. If it's `> 5`, trigger a notification to Slack because the app is in a **CrashLoopBackOff**.

Would you like me to show you how to filter Pods by their status (e.g., only show Pods that have more than 0 restarts)?