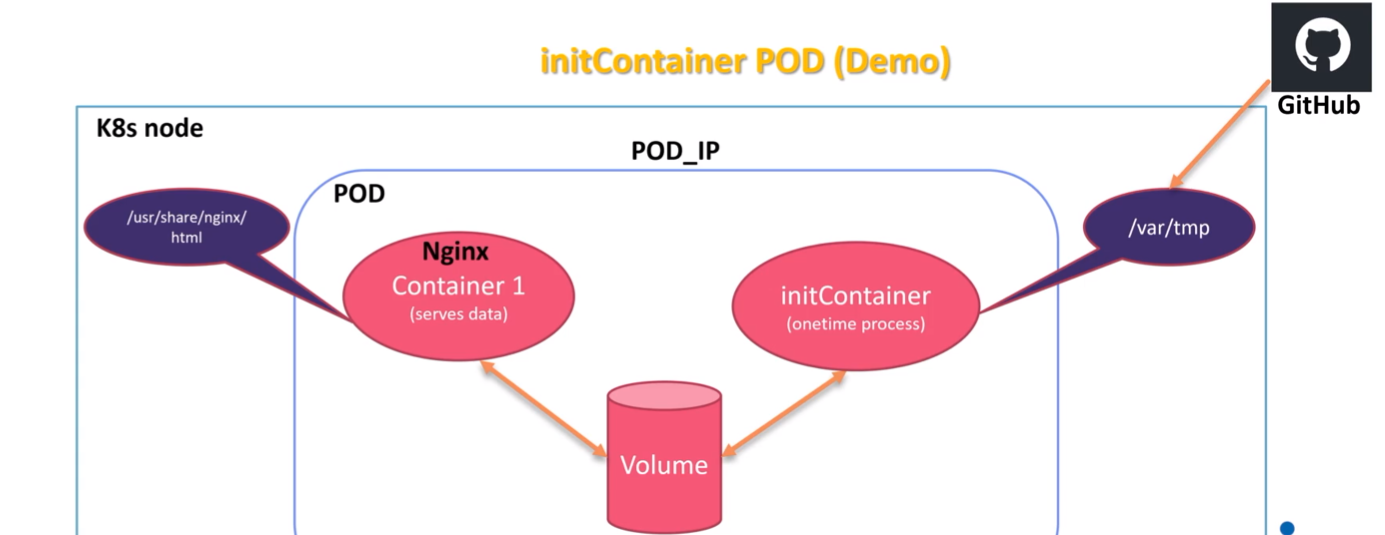
**INIT CONTAINERS:**

* Init containers run before the app container defined under the app section. In a pod yaml file under spec, we will define init containers in the container section. When we spin up the containers first init containers are initialized then other containers will get started. Hence these containers are named as init containers as they initialize first.
* Init containers will not be in running state they will be in completed state.
* It works similar to the actual containers but each init container must complete successfully before the next one starts. Even Though we create the multiple init containers inside the section, all these containers will be created one after the other in a series order. Later other containers will get created.
* If the init container fails, kubelet will restart the init container till it succeeds. This will happen only when the restart policy is not set to Never.
* Init containers do not support probes and lifecycle.
* In the below use cases we can go for init containers
* To do initial setup before actual applications come up. For example, we are trying to deploy an nginx application. Before this comes up some parameter modifications need to be done in the kernel. In such conditions this particular task is assigned to the init containers to do the job because before actual containers come up, init containers will get created first.
* To run some code/script as part of initialization before the app gets created. Lets say, we are writing a python/django application. Before it comes up, there is a need to execute python/django scripts wherein these are not supposed to be executed as part of my regular application. This has to be separated as one job and kept in the init container section to make it ready before the actual container comes up.
* To do some kernel changes specifically for an application. For suppose, If you want to do some kernel changes before the actual application is running, we go for the init containers. There is no limit for defining init containers.



In the above scenario, we are trying to create a pod in the node which communicates through the pod ip address to your nginx application listening on 80 port to serve the data. If someone wants to access the application from the particular node they will try to use the pod ip address and port number. If they try to access from outside of the application then a hostport is used.

The job of the init container is to contact the github and download the index.html file and store it in the /var/tmp location inside the container which is mounted to the volume. The data in the /var/tmp is visible to the volume(/var/lib/kubelet/<pod\_uid>/volumes) as it is mounted and the same data is used by the nginx application. When the user tries to access the application the content inside index.html is made visible. Once the job inside the init container is done successfully it will reach the completed state but the pod and the nginx container will be in running state.

Let us see the practical implementation of the init containers in a pod.

Create the yaml file for creating init containers through vim file editor.



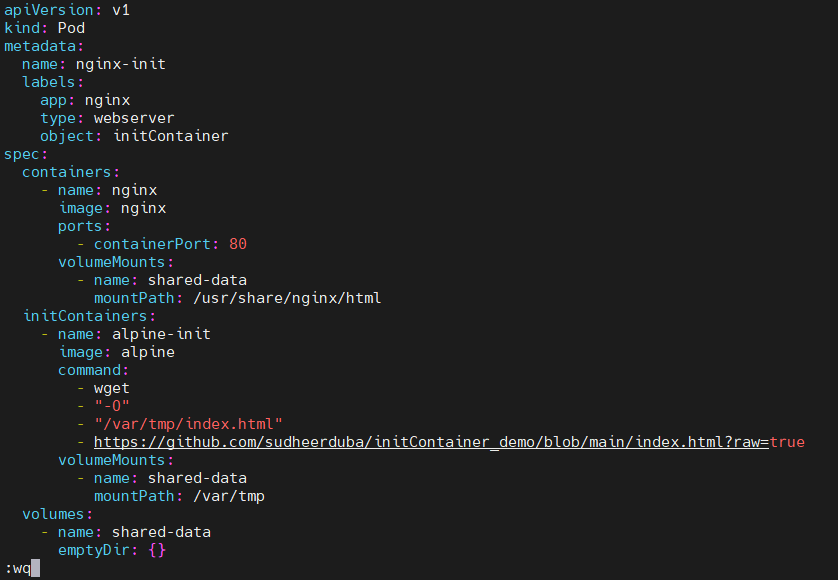
In this file, apiversion for pod is v1, kind is pod as we are trying to create a pod object and then name the pod under metadata, also label it(optional) with app as nginx, type as web server and object as initContainer.

Under specifications create a container with the name as nginx-init which is created with the nginx image meant to serve the data and the container port is 80, while mounting the path /usr/share/nginx/html directory to the shared volumes which is defined under volume section along with emptyDir.

Then create one init container by providing the name(alpine-init) using alpine image to run some basic commands as wget to download the index.html file stored in the below mentioned github url. -O is to store the output into the given location /var/html/index.html.

Volume mounts section is to mount the shared volume to the above container at the mount path /var/tmp.Finally save the file.

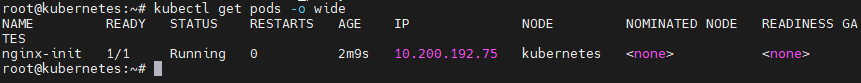
Here the first init container is created and does the task to download the index.html file from the github url and places it in the /var/tmp location then init container will reach the completed state. As init container is mounted to the shared volume, index.html is accessed by the nginx application running in another container which is also mounted to the same shared volume and this container will be in running state until pod is running.



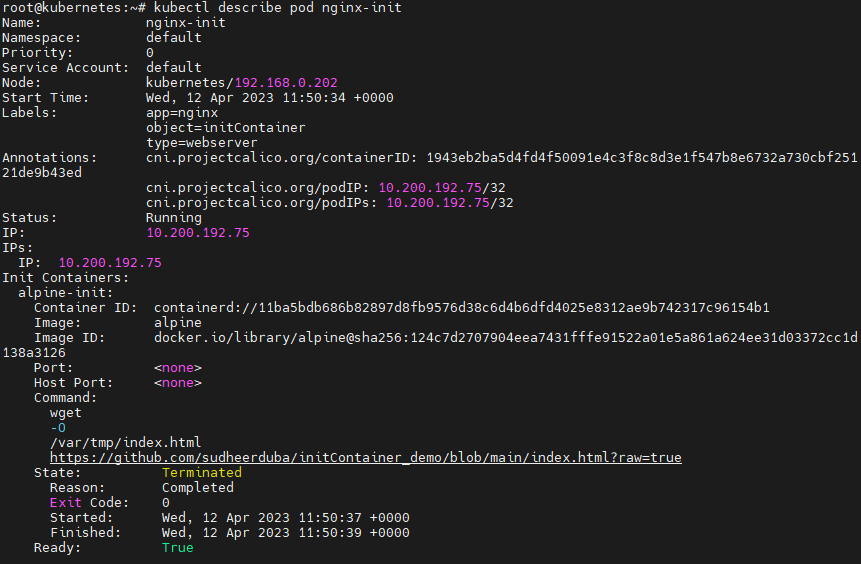
Kubectl create -f file\_name command will create the pod according to the specified file. Hence the pod is created.

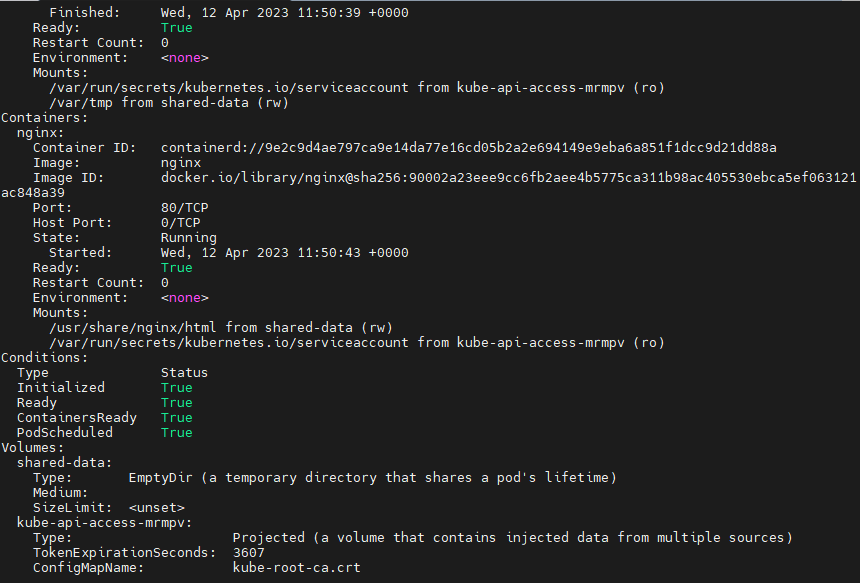


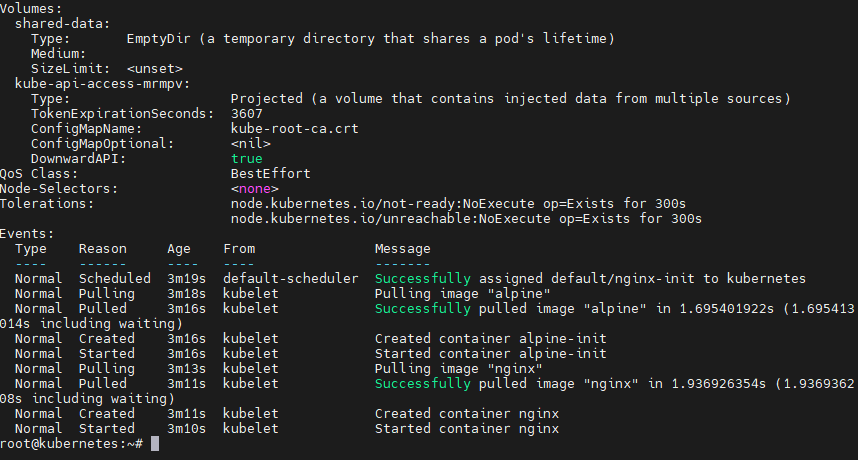
Kubectl get pods -o wide command will give the pod details like status, age, pod ip address and on which node that particular pod is created.



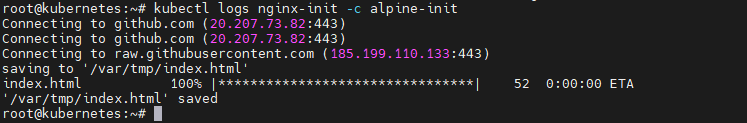
To know the complete information about the pod, run the command kubectl describe pod pod\_name. Here under init containers exit code is zero and state is shown as terminated as the reason is completed since whatever the task defined inside it is finished and cannot login to the container again as it is init container. And other containers will be running as usual till pod runs.



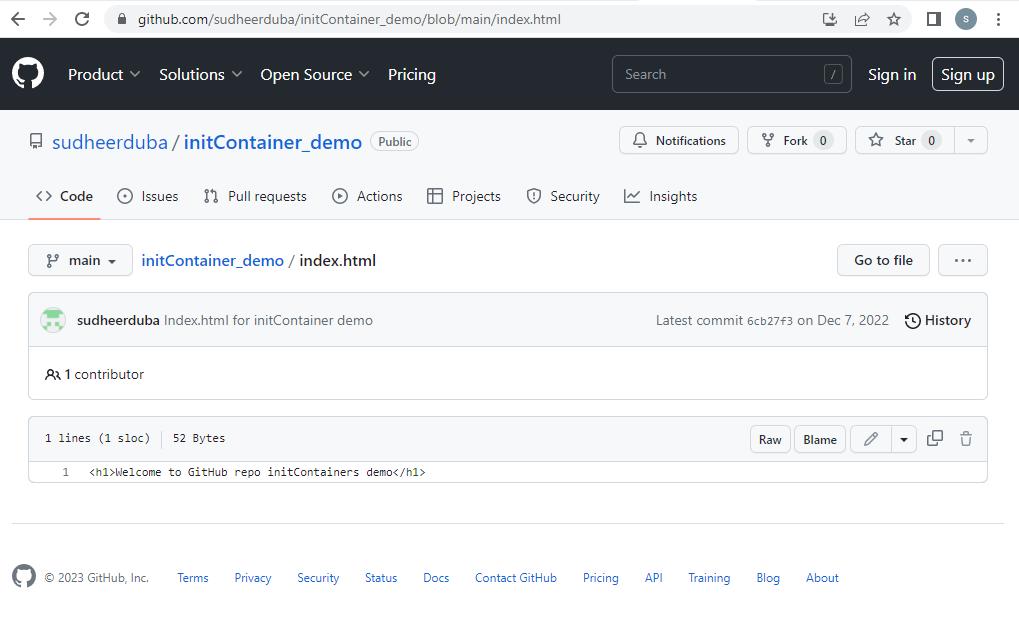




Kubectl logs pod\_name -c container\_name command is to show the logs of the specified container. This command gives clear idea about the task done inside the init container.



This is the data written inside index.html on the github.



When we curl the container ip address where nginx application is running, the content present in the index.html just as shown above has to be accessed.



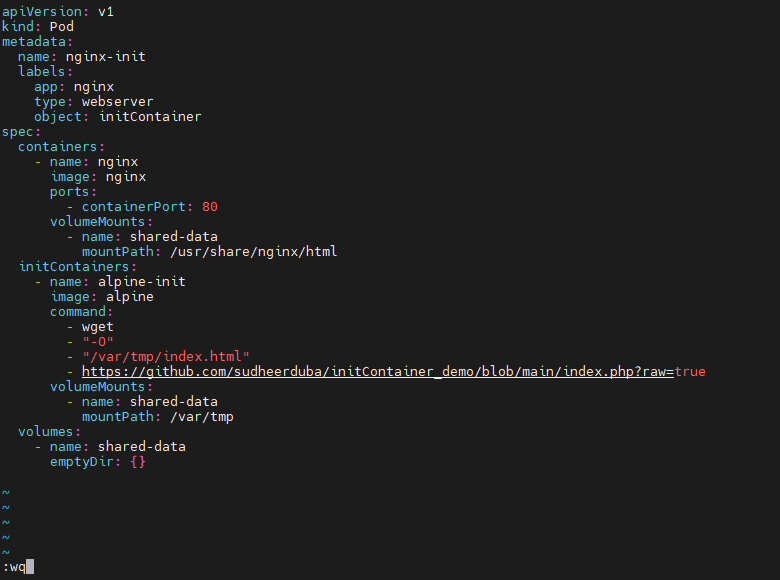
Let us now delete the init container pod through the command kubectl delete -f file\_name.



Meanwhile open the init container pod file through the vim file editor.



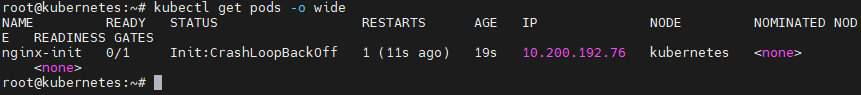
Modify the github url with php file extension which is not available in the github account to see the variance of init container working mechanism. Then save the file.



Kubectl create -f file\_name command will create the pod according to the specified file. Hence the pod is created.



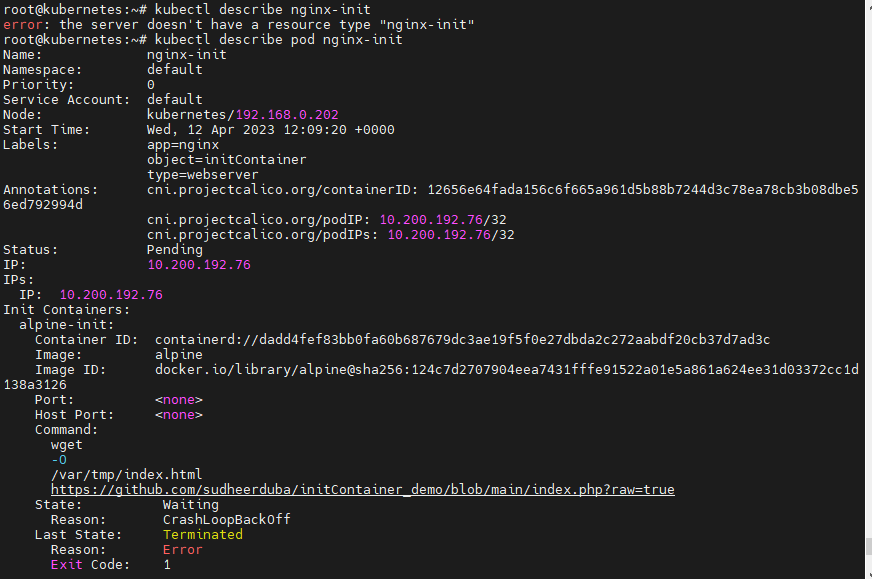
Kubectl get pods -o wide command will give the pod details like status, age, pod ip address and on which node that particular pod is created. Below it is shown that init container is exited with CrashLoopBackOff error as the php file mentioned in the url is not available in the github.

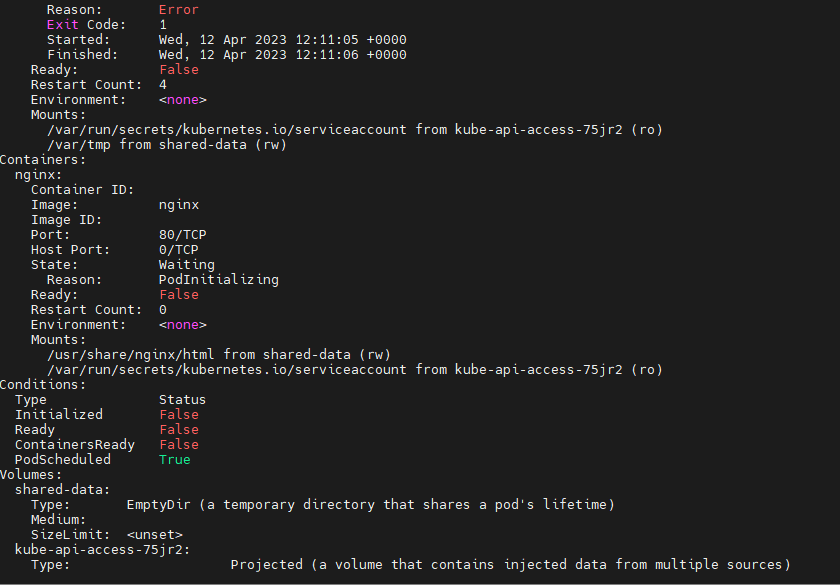


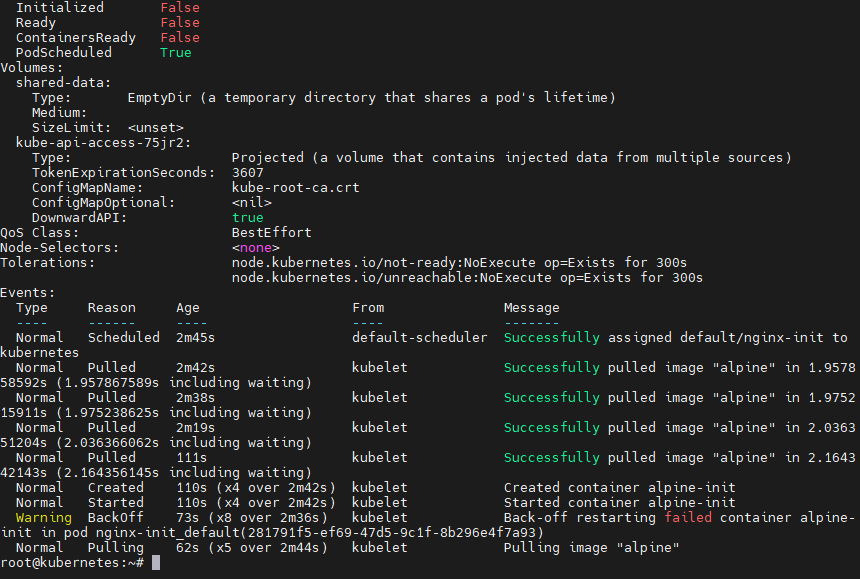
If we try to access the pod ip address also it will fail.



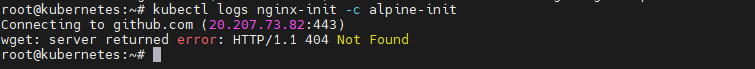
When we describe the pod, here under init containers exit code is one and state is shown as waiting as the reason is CrashLoopBack error since whatever the task defined is not finished due to lack of php file in the github account and other containers will also be in waiting as pod is in still in initializing state.





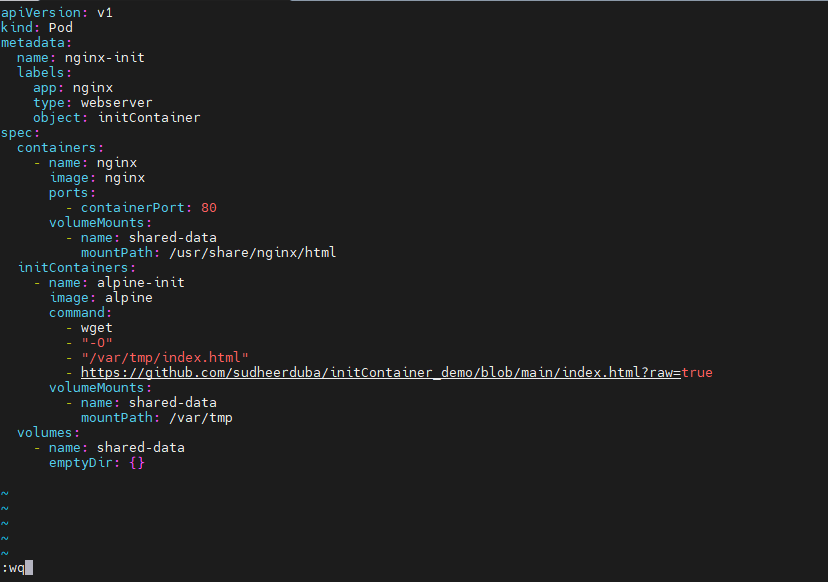


While checking the logs of init containers we can encounter the exact issue occurred during execution.

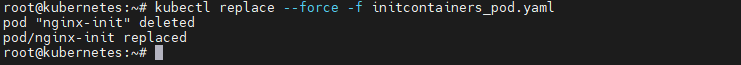


Again make changes in the init container pod file with index.html file in the url and save the file.

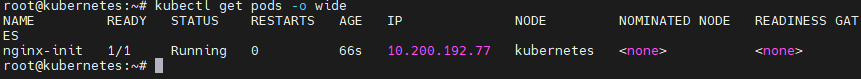




Now forcefully replace the modified file with the command kubectl replace –force -f file\_name.



Thus pod will be in running state as index.html is present in the github and it will continue with the task assigned to the init container.



Therefore we can access the content of the index.html file with the pod ip address through the curl command.

