MinIO

1. Overview:

MinIO can serve the object storage similar to S3 bucket.

MinIO Deployed Location:

```
http://10.3.9.246:9000/
credential:
indoc-minio:****
```

2. Setup:

MinIO support standalone deployment and distributed deployment. Versioning is **ONLY** available under distributed deployment. So each mode has its own behaviours in terms of file operations(eg. upload, download and deletion) if versioning is enabled. Please refer to versioning section underneath.

2.1 Standalone Setup

Standalone means it will run the Minio on a single server. If there are some existing files or folders in the mounted directory, MinIO will also shows them after starts. But they will not have any metadata. Also, under the standalone mode, the versioning feature is not available.

To run the MinIO in docker version(document):

```
docker run -p 9000:9000 \
  -e "MINIO_ACCESS_KEY=AKIAIOSFODNN7EXAMPLE" \
  -e "MINIO_SECRET_KEY=**** \
  minio/minio server /data
```

To run the MinIO in docker with OpenID IDP (KeyClock), set NFS mounted path “/data“ as persistence storage location(document):

```
docker run -p 9000:9000 \
  --name minio_indoc -d \
  -v /data:/data \
  -e "MINIO_ACCESS_KEY=indoc-minio" \
  -e "MINIO_SECRET_KEY=**** \
  -e "MINIO_IDENTITY_OPENID_CONFIG_URL=<host>/vre/auth/realms/vre/.well-
known/openid-configuration" \
  -e "MINIO_IDENTITY_OPENID_CLIENT_ID=minio" \
  -e "MINIO_IDENTITY_OPENID_CLAIM_NAME=policy" \
  -e "MINIO_IDENTITY_OPENID_SCOPES=email,openid,profile,roles,web-
origins"
  minio/minio server /data
```

Or to change service credentials:

```
docker run -p 9000:9000 \
  --name vre_minio \
  -v /data:/data \
  -e "MINIO_ACCESS_KEY=indoc-minio" \
  -e "MINIO_SECRET_KEY=**** \
  -e "MINIO_ACCESS_KEY_OLD=AKIAIOSFODNN7EXAMPLE" \
  -e "MINIO_SECRET_KEY_OLD=**** \
  minio/minio server /data
```

2.2 Distribution Setup

Distribution mode allow MinIO to mount several disk point. But it will require to format the disk(folder) at beginning. If there is some file before MinIO kicks in, MinIO will raise error. Afterward, if you shutdown the MinIO, it will detect formatted folder and run again. The versioning feature is **ONLY** available in this setup. If you want to add a new mounted data point, you can add a new parameter as `-v /<new_data_point_at_host>:/<mapping_to_docker>` (the new point **should not** contain any other file outside MinIO)

Below are the command amount 3 different data point to MinIO. Please note the line 11 will require {1…4}

```
docker run -p 9001:9000  \
-v /data1:/data1  \
-v /data2:/data2  \
-v /data3:/data3  \
-e "MINIO_ACCESS_KEY=indoc-minio"  \
-e "MINIO_SECRET_KEY=****"  \
-e "MINIO_IDENTITY_OPENID_CONFIG_URL=<host>/vre/auth/realms/vre/.well-known/openid-configuration" \
-e "MINIO_IDENTITY_OPENID_CLIENT_ID=minio"  \
-e "MINIO_IDENTITY_OPENID_CLAIM_NAME=policy"  \
-e "MINIO_IDENTITY_OPENID_SCOPES=email,openid,profile,roles,web-origins"  \
minio/minio server /data{1...4}
```

2.3 (Optional)Encryption Setup

If we want to enable the server side encryption, we need setup a Key Management Service(KMS) https://github.com/minio/kes#kes . We can setup up a new instance by docker or use the dev environment by MinIO group: `https://play.min.io:7373`. You can download the key and certificate by following:

```
curl -sSL --tlsv1.2 \
   -O 'https://raw.githubusercontent.com/minio/kes/master/root.key' \
   -O 'https://raw.githubusercontent.com/minio/kes/master/root.cert'
```

Afterwards, we can configure MinIO and connect to KMS. **But DONT use setup from MinIO group for production environment**. Note we need to bind the cert into docker image:

```
docker run -p 9001:9000 \
  -v /data:/data \
  -v /home/zzhan/minio/root.cert:/var/lib/docker/root.cert \
  -v /home/zzhan/minio/root.key:/var/lib/docker/root.key \
  -e "MINIO_ACCESS_KEY=indoc-minio" \
  -e "MINIO_SECRET_KEY=****" \
  -e "MINIO_ACCESS_KEY_OLD=AKIAIOSFODNN7EXAMPLE" \
  -e "MINIO_SECRET_KEY_OLD=****" \
  -e "MINIO_KMS_KES_ENDPOINT=https://play.min.io:7373" \
  -e "MINIO_KMS_KES_KEY_FILE=/var/lib/docker/root.key" \
  -e "MINIO_KMS_KES_CERT_FILE=/var/lib/docker/root.cert" \
  -e "MINIO_KMS_KES_KEY_NAME=my-key" \
  minio/minio server /data
```

3. Integration with Keycloak(OpenID):

Step 1: Get token from keycloak autentication URL:

```
<host>/vre/auth/realms/vre/protocol/openid-connect/token
## parameters (x-www-form-urlencoded)
grant_type:password
username:****
password:****
# client_<parameter> are from keycloak
client_id:minio
client_secret:****
```

This request will require the `POST` action and embeds the payload as parameters:

```
curl -d
"grant_type=password&username=****&password=****&client_id=minio&client_
secret=****" \
-H "Content-Type: application/x-www-form-urlencoded" \
-X POST <host>/vre/auth/realms/vre/protocol/openid-connect/token
```

Step 2: Grants Role for the Client by Token

Use the token in step 1, we can retrieve the tokens from Minio. Please note that we use `ClientGrantProvider`. It has `retrieve()` function to fetch the credentials underline. If the token expired, SDK will call `retrieve()` again to fetch a new temp tokens.

```
import requests
import xmltodict
from minio import Minio
from progress import Progress
from minio.commonconfig import Tags
import os
import time
import datetime

from minio.credentials.providers import ClientGrantsProvider

def get_jwt():
    print("fetching from keycloak")

    # first login with keycloak
    # pass the username and password here
    username = "admin"
    password = "****"
    payload = {
        "grant_type":"password",
        "username":username,
        "password":password,
        "client_id":"react-app",
        # "client_secret":"****",
```

```
    }
    headers = {
        "Content-Type": "application/x-www-form-urlencoded"
    }

    result = requests.post("<host>/vre/auth/realms/vre/protocol/openid-
connect/token", data=payload, headers=headers)
    keycloak_access_token = result.json().get("access_token")
    expire_time = result.json().get("expires_in")
    # print(result.json())
    return result.json()

# get_jwt()

provider = ClientGrantsProvider(
    get_jwt,
    "https://vre-staging-minio.indocresearch.org",
    duration_seconds = 1000
)
# print(provider.retrieve())
c = provider.retrieve()

client = Minio(
    "vre-staging-minio.indocresearch.org",
    credentials=provider,
    secure=True)

buckets = client.list_buckets()
print(buckets)
for bucket in buckets:
    print(bucket.name, bucket.creation_date)
```

(optional) Or if you want to use the refresh the token, you only need to `_get_jwt()` function

```
class Minio_Client():
    def __init__(self, access_token, refresh_token):
        # preset the tokens for refreshing
        self.access_token = access_token
        self.refresh_token = refresh_token

        # retrieve credential provide with tokens
        c = self.get_provider()

        self.client = Minio(
            ConfigClass.MINIO_ENDPOINT,
            credentials=c,
            secure=ConfigClass.MINIO_HTTPS)
```

```
    # function helps to get new token/refresh the token
    def _get_jwt(self):
        print("refresh token")
        # here use the refresh token type
        payload = {
            "grant_type" : "refresh_token",
            "refresh_token": self.refresh_token,
            "client_id":ConfigClass.MINIO_OPENID_CLIENT,
        }
        headers = {
            "Content-Type": "application/x-www-form-urlencoded"
        }

        # use http request to fetch from keycloak
        result = requests.post(ConfigClass.KEYCLOAK_URL+"/vre/auth
/realms/vre/protocol/openid-connect/token", data=payload,
headers=headers)
        keycloak_access_token = result.json().get("access_token")

        self.access_token = result.json().get("access_token")
        self.refresh_token = result.json().get("refresh_token")
        print(result.json())
        print(self.access_token)
        print(self.refresh_token)

        return result.json()

    # use the function above to create a credential object in minio
    # it will use the jwt function to refresh token if token expired
    def get_provider(self):
        minio_http = ("https://" if ConfigClass.MINIO_HTTPS else
"http://") + ConfigClass.MINIO_ENDPOINT
        # print(minio_http)
        provider = ClientGrantsProvider(
            self._get_jwt,
            minio_http,
        )

        return provider
```

4. Python SDK:

Install MinIO through pip3(document)

```
pip3 install minio
```

See the python example:

```
# Import MinIO library.
from minio import Minio
from minio.error import (ResponseError, BucketAlreadyOwnedByYou,
                         BucketAlreadyExists)

# Initialize minioClient with an endpoint and access/secret keys.
minioClient = Minio('play.min.io',
                    access_key='AKIAIOSFODNN7EXAMPLE',
                    secret_key='****',
                    secure=True)

# Make a bucket with the make_bucket API call.
try:
       minioClient.make_bucket("maylogs", location="us-east-1")
except BucketAlreadyOwnedByYou as err:
       pass
except BucketAlreadyExists as err:
       pass
except ResponseError as err:
       raise

# Put an object 'pumaserver_debug.log' with contents from
'pumaserver_debug.log'.
try:
       minioClient.fput_object('maylogs', 'pumaserver_debug.log', '/tmp
/pumaserver_debug.log')
except ResponseError as err:
       print(err)
```

Due to poor API document, please check the examples on their git.

File Download:

Minio provides two way to download the file(Minio Docs):

1. use `fget(bucket_name, object_name, file_path)`: This will directly download into machine
2. use `presigned_get_object(bucket_name, object_name)`: Using presigned URL is more preferable way. The function will return the URL that expires in 7 days. By clicking the url, the browser will start to download the file. Note: the url is **not** one time used. It is public to whoever get the link. If the link expired during the download then it will not interrupt the downloading. **So it is better to keep the expiry time very short(~1min), since the expiration will not stop the ongoing download action.** If you want to download the file under the sub-folder, you can add the sub-folder in object as `<sub-folder>/<your_file>`. If you only give `<sub_folder>/` without any file there, the web browser will give an error.

Presigned Download URL example:

```
http://10.3.9.246:9000/test/models.zip?X-Amz-Security-
Token=eyJhbGciOiJIUzUxMiIsInR5cCI6IkpXVCJ9.
eyJhY2Nlc3NLZXkiOiJaVEQ2SjU5NUpPNk83SkxKRkxYVSIsImFjciI6IjEiLCJhbGxvd2Vk
LW9yaWdpbnMiOlsiaHR0cDovLzEwLjMuOS4yNDY6OTAwMCJdLCJhdWQiOiJhY2NvdW50Iiwi
YXpwIjoibWluaW8iLCJlbWFpbCI6InpoYW5nemhpcWluN0BnbWFpbC5jb20iLCJlbWFpbF92
ZXJpZmllZCI6ZmFsc2UsImV4cCI6IjE2MTg1MjA3NDIiLCJmYW1pbHlfbmFtZSI6InNhbWFu
dGhhIiwiZ2l2ZW5fbmFtZSI6InpoYW5nIiwiaWF0IjoxNjE4NTIwNDQyLCJpc3MiOiJodHRw
Oi8vMTAuMy43LjIyMC92cmUvYXV0aC9yZWFsbXMvdnJlIiwianRpIjoiY2M5YzM2NmUtNjg2
Yi00MTIwLWFiNGMtNjI3Zjg5OWE0MmM2IiwibmFtZSI6InpoYW5nIHNhbWFudGhhIiwicG9s
aWN5IjoicmVhZHdyaXRlIiwicHJlZmVycmVkX3VzZXJuYW1lIjoic2FtYW50aGEiLCJyZWFs
bV9hY2Nlc3MiOnsicm9sZXMiOlsiY29udHJpYnV0b3IiLCJvZmZsaW5lX2FjY2VzcyIsInVt
YV9hdXRob3JpemF0aW9uIl19LCJyZXNvdXJjZV9hY2Nlc3MiOnsiYWNjb3VudCI6eyJyb2xl
cyI6WyJtYW5hZ2UtYWNjb3VudCIsIm1hbmFnZS1hY2NvdW50LWxpbmtzIiwidmlldy1wcm9m
aWxlIl19fSwic2NvcGUiOiJwcm9maWxlIGVtYWlsIiwic2Vzc2lvbl9zdGF0ZSI6Ijg2ZDkz
ODM5LTNkYzktNGUwMC04ZTdhLTFjNzcwNTdhMTY0ZCIsInN1YiI6IjlkNjZiYmU3LTQ1NDIt
NDM3My1iY2FiLTkxYjgyNWYxNjg0NyIsInR5cCI6IkJlYXJlciJ9.
qLfQt8VVRddZ0fsS6XPrz-L-
BIe80xaFDh99EVfpqsWThbLYj7AQFbzzIUzKMAuBupL6p9c4fw0xgyobCfZOmA&X-Amz-
Algorithm=AWS4-HMAC-SHA256&X-Amz-Credential=ZTD6J595JO6O7JLJFLXU%
2F20210415%2Fus-east-1%2Fs3%2Faws4_request&X-Amz-
Date=20210415T210042Z&X-Amz-Expires=604800&X-Amz-SignedHeaders=host&X-
Amz-
Signature=f5e64a6dc718630485932dcccd68fe349b30272232dd33ec0634bf06392edf
08
```


File Upload:

Minio provides three ways to upload the file:

1. `put_object(bucket_name, object_name, data, length)` MinIO | put_object : This is the traditional method to upload a file stream to target bucket/name.
2. `fput_object(bucket_name, object_name, file_path)` MinIO | fput_object : `fgut_object` support to upload a file in the directory. The underlining code is use the file read to wrap up the method `put_object`.
3. `presigned_put_object(bucket_name, object_name, expires)` MinIO | presigned_put_object : `preasigned` is a new method to upload the file. It provide some kind of the destination url for clients. The client will only need to read the file as file stream and send it as `http PUT request`(for all the programming language). Note: it has same behaviour as `presigned_get_object`. The url is **not** one time used. It is public to whoever get the link. If the link expired during the download then it will not interrupt the uploading. **So it is better to keep the expiry time very short(~10s), since the expiration will not stop the ongoing upload action.** If, later on, we expose the link directly to user, then we can increase to ~1min.

   If you want to upload into sub-folder, change the object into path as `<sub-folder>/<your_file>`. There is an interesting behavior that if you only give `<sub_folder>/` without any file there, the MinIO will create an empty folder there and return http 409 error. One thing to mention that we can use this API to create a empty sub-folders. To avoid the abuse of this API, please remember to check there is no ending slash.

Presigned Download URL example:

```
http://10.3.9.246:9000/test/presigned_obj_2.img?X-Amz-Security-
Token=eyJhbGciOiJIUzUxMiIsInR5cCI6IkpXVCJ9.
eyJhY2Nlc3NLZXkiOiJYWjdaWEpJUzZNNEhRTkZFNFNOMiIsImFjciI6IjEiLCJhbGxvd2Vk
LW9yaWdpbnMiOlsiaHR0cDovLzEwLjMuOS4yNDY6OTAwMCJdLCJhdWQiOiJhY2NvdW50Iiwi
YXpwIjoibWluaW8iLCJlbWFpbCI6InpoYW5nemhpcWluN0BnbWFpbC5jb20iLCJlbWFpbF92
ZXJpZmllZCI6ZmFsc2UsImV4cCI6IjE2MjAwNzI0MTkiLCJmYW1pbHlfbmFtZSI6InNhbWFu
dGhhIiwiZ2l2ZW5fbmFtZSI6InpoYW5nIiwiaWF0IjoxNjIwMDcyMTE5LCJpc3MiOiJodHRw
Oi8vMTAuMy43LjIyMC92cmUvYXV0aC9yZWFsbXMvdnJlIiwianRpIjoiZmE5ZTMyOWMtMjhi
ZS00MTQ1LTk2M2MtYmRlNDk4M2I5NDI0IiwibmFtZSI6InpoYW5nIHNhbWFudGhhIiwicG9s
aWN5IjoicmVhZHdyaXRlIiwicHJlZmVycmVkX3VzZXJuYW1lIjoic2FtYW50aGEiLCJyZWFs
bV9hY2Nlc3MiOnsicm9sZXMiOlsidHZiY2xvdWQtYWRtaW4iLCJpbmRvY3Rlc3Rwcm9qZWN0
LWFkbWluIiwib2ZmbGluZV9hY2Nlc3MiLCJnZW5lcmF0ZS1hZG1pbiIsInVtYV9hdXRob3Jp
emF0aW9uIl19LCJyZXNvdXJjZV9hY2Nlc3MiOnsiYWNjb3VudCI6eyJyb2xlcyI6WyJtYW5h
Z2UtYWNjb3VudCIsIm1hbmFnZS1hY2NvdW50LWxpbmtzIiwidmlldy1wcm9maWxlIl19fSwi
c2NvcGUiOiJwcm9maWxlIGVtYWlsIiwic2Vzc2lvbl9zdGF0ZSI6ImYzYjdhNzgzLWFlNDMt
NDRkMC05ZmIwLTMxNTQ0MmZhZmZmYiIsInN1YiI6IjlkNjZiYmU3LTQ1NDItNDM3My1iY2Fi
LTkxYjgyNWYxNjg0NyIsInR5cCI6IkJlYXJlciJ9.FheMe-GKdxwoKPljqIfc-w-
hvtVJIodJzz0VaFanAJ4639ZzMUxQxLZTVnXdbR6Yl0og-APSdoxuP9Hu6qAjFw&X-Amz-
Algorithm=AWS4-HMAC-SHA256&X-Amz-Credential=XZ7ZXJIS6M4HQNFE4SN2%
2F20210503%2Fus-east-1%2Fs3%2Faws4_request&X-Amz-
Date=20210503T200204Z&X-Amz-Expires=60&X-Amz-SignedHeaders=host&X-Amz-
Signature=c84d1f998f1518fd15d119cb580a53e3d4048e3af892c3780a0ee9f2463597
ef
```


Code Example:

```
#########################
# fput_object
#########################

# setup metadata.
upload_file = "400M.img"
upload_size = os.path.getsize(upload_file)
# print(upload_size)
p = Progress()
p.set_meta(total_length=upload_size, object_name=upload_file)

# upload with progress bar
result = client.fput_object(
    "test", "test_folder_1", upload_file,
    metadata={"My-Project": "one"},
    progress=Progress(),
)
print(
    "\ncreated {0} object; etag: {1}, version-id: {2}".format(
        result.object_name, result.etag, result.version_id,
    ),
)


#########################
# presigned_put_object
#########################

presigned_upload_url = client.presigned_put_object("test",
"presigned_obj_2.img", expires=datetime.timedelta(seconds=60))
headers = {'Content-type': 'application/octet-stream'}
r = requests.put(presigned_upload_url, data=open('/home/color/indoc
/minio/400M.img', 'rb'), headers=headers)
print(r)
```

5. Connection with Rabbitmq:

Connect is base on the official document. with following step to connect to the rabbitmq.

Start Rabbitmq:

Following command will start Rabbitmq at port 5672. The default credential is user:****.(check docs):

```
docker run -p 5672:5672 -e RABBITMQ_DEFAULT_USER=user -e
RABBITMQ_DEFAULT_PASS=**** bitnami/rabbitmq:latest
```

Add queue configuration

The following command will connect the MinIO with Rabbitmq. by running the command, it will also return a ARN for the queue. Please use it in next step. eg. `SQS ARNs: arn:minio:sqs::_:amqp`

- `MINIO_NOTIFY_AMQP_ENABLE` and turn on the connection setting.
- `MINIO_NOTIFY_AMQP_URL` is the represent the url of queue. please remember to add `amqp://` prefix
- `MINIO_NOTIFY_AMQP_EXCHANGE`, `MINIO_NOTIFY_AMQP_EXCHANGE_TYPE` and `MINIO_NOTIFY_AMQP_ROUTING_KEY` are for the channel setup please make sure receiver and sender have same setting.

```
docker run -p 9000:9000 \
  -v /data:/data \
  -e "MINIO_ACCESS_KEY=indoc-minio" \
  -e "MINIO_SECRET_KEY=****" \
  -e "MINIO_ACCESS_KEY_OLD=AKIAIOSFODNN7EXAMPLE" \
  -e "MINIO_SECRET_KEY_OLD=****" \
  -e "MINIO_NOTIFY_AMQP_ENABLE=on" \
  -e "MINIO_NOTIFY_AMQP_URL=amqp://user:password@10.3.9.246:5672" \
  -e "MINIO_NOTIFY_AMQP_EXCHANGE=direct_logs" \
  -e "MINIO_NOTIFY_AMQP_EXCHANGE_TYPE=direct" \
  -e "MINIO_NOTIFY_AMQP_ROUTING_KEY=minio" \
  minio/minio server /data
```

Add the notification to bucket

same as S3 we need to add the notification policy that state which event is on. check the document. You need to specify which queue bucket should send to. eg. `SQS ARNs: arn:minio:sqs::_:amqp`

```
from minio.notificationconfig import (NotificationConfig,
PrefixFilterRule,
                                      QueueConfig)

res = client.get_bucket_notification('test')
print(res)

config = NotificationConfig(
    queue_config_list=[
        QueueConfig(
            "arn:minio:sqs::_:amqp",
            ["s3:ObjectCreated:*", "s3:ObjectRemoved:*", "s3:
ObjectAccessed:*"],
            config_id="1",
        ),
    ],
)

# set the policy above to target bucket
res = client.set_bucket_notification('test', config)
print(res)
```

Listening on the notification

With this example, we use the `pika` to connect with Rabbitmq. Remember to update following if needed:

- `line 6`: correct credential for Rabbitmq user.
- `line 7`: correct credential for Rabbitmq endpoint and port.
- `line 13`: correct exchange and exchange_type.
- `line 23`: correct routing key.

```
import pika
import time
import sys

# channel
credentials = pika.PlainCredentials('user', 'password')
connection = pika.BlockingConnection(pika.ConnectionParameters
('10.3.9.246', 5672, '/', credentials))
channel = connection.channel()

##########################################################################
####################
# please make sure the exchange, exchange type and routing_key are the
same with sender   #
##########################################################################
####################
channel.exchange_declare(exchange='direct_logs', exchange_type='direct')
```

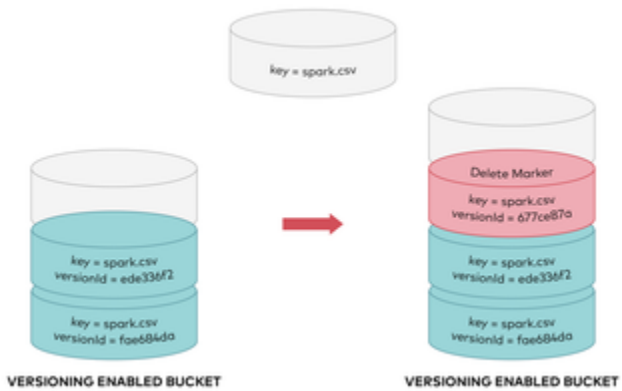
```
# queue
# we need to make sure that the queue will survive a RabbitMQ node
restart(durable)
# Secondly, once the consumer connection is closed, the queue should be
deleted. There's an exclusive flag for that:
result = channel.queue_declare(queue='', exclusive=True, durable=True)
queue_name = result.method.queue

# get the severity
# severities = ['info', 'warning', 'error']
route = ['minio']

# routing key
for severity in route:
    channel.queue_bind(exchange='direct_logs', queue=queue_name,
routing_key=severity)
print(' [*] Waiting for logs. To exit press CTRL+C')

def callback(ch, method, properties, body):
    print(" [x] Received %r" % body)
    time.sleep(body.count(b'.'))
    print(" [x] Done")
    ch.basic_ack(delivery_tag = method.delivery_tag)

# we can use the Channel#basic_qos channel method with the
prefetch_count=1 setting.
# This uses the basic.qos protocol method to tell RabbitMQ not to give
more than one message to a worker at a time.
# Or, in other words, don't dispatch a new message to a worker until it
has processed and acknowledged the previous one.
#  Instead, it will dispatch it to the next worker that is not still
busy
channel.basic_qos(prefetch_count=1)

# Manual message acknowledgments are turned on by default.
# In previous examples we explicitly turned them off via the
auto_ack=True flag.
# It's time to remove this flag and send a proper acknowledgment from
the worker,
# once we're done with a task.
channel.basic_consume(queue=queue_name, on_message_callback=callback)
channel.start_consuming()
```

6. Versioning

The versioning is **ONLY** available in the distributed mode. Referring the image below, the versioning act like the stack which versions place on top of each other. If there is no version id specified, the get object will always get the latest object. If there is a delete action on object, the latest will be marked as deleted. The get will still return the object is not exist. To permanently delete the object, you need to specify the version id. Below are some special operation:

- If a file is mark as delete, you can still delete the version with "delete marker". The file will appear again.
- If all version has been removed, the file will still appear in the file system.
- If you want to list all historical version of a file, you have to user MinIO admin as `mc ls --versions <alias>/<PATH>` https://docs.min.io/minio/baremetal/reference/minio-cli/minio-mc/mc-ls.html
- Also you can undo the operation via minio client as `./mc undo <alias>/<bucket>/<object> --last <number_of_version>`
- Please note that `presigned_get_url` also support different version_ids.
- Metadata update will not create a new version.

For more detailed infomation, please refer to the MinIO docs: https://docs.min.io/docs/minio-bucket-versioning-guide.html



Sample code:

```
# enable versioning
client.make_bucket("test")
client.set_bucket_versioning("test", VersioningConfig(ENABLED))

# upload with progress bar
result = client.fput_object(
    "test", "test.py", "test.py",
    metadata={"My-Project": "one"},
)
print(
    "\ncreated {0} object; etag: {1}, version-id: {2}".format(
        result.object_name, result.etag, result.version_id,
    ),
)

# then upload another version and try to delete it
# the object will be marked as the delete BUT the file will be
# in the file system
result = client.fput_object(
    "test", "test.py", "test.py",
    metadata={"My-Project": "one"},
)
print(
    "\ncreated {0} object; etag: {1}, version-id: {2}".format(
        result.object_name, result.etag, result.version_id,
```

```
    ),
)
client.remove_object("test", "test.py")
# try to get the infomation of removed object with version id
result = client.stat_object("test", "test.py", version_id=result.
version_id)
print(result)
```

7. Minio Policy

Minio has two type of policies:

1. user/user group policy: this type of policy is to restrict the user access or group access with respect to each bucket.
2. bucket policy: this type of policy is for buckets to have some access control at high level.

7.1 User/User Group Policy:

7.1.1 Add policy:

Please note that the policy can **only be added** by admin client. so first we need to setup the admin client to create a policy https://docs.min.io/docs/minio-multi-user-quickstart-guide.html . One limitation is the admin client **ONLY** support go language, but we still can use python to run the shell command:

1. install admin console: https://docs.min.io/docs/minio-client-quickstart-guide.html (use linux version)
2. set the alias for the host `mc alias set minio <host> <access_key> <screte_key>`
3. create a policy file at local machine. It will allow the user to **ONLY** access the bucket `my-bucketname`:

```
{
  "Version": "2021-5-5",
  "Statement": [
    {
      "Action": [
        "s3:GetObject",
        "s3:ListBucket"
      ],
      "Effect": "Allow",
      "Resource": [
        "arn:aws:s3:::my-bucketname/*"
      ],
      "Sid": ""
    }
  ]
}
```

4. add the policy to minio server:

```
#./mc admin policy add <alias> <policy_name> <local_policy_file>
./mc admin policy add minio getonly getonly.json
```

5. check the policy:

```
#./mc admin policy info <alias> <policy_name>
./mc admin policy info minio getonly
```

7.1.2 Grant Policy to User without OpenID:

once we create a policy, we can add it with user/user group. Then the user will be granted with specific set of permission.

1. create a policy file at local machine. It will allow the user to **ONLY** access the bucket `my-bucketname`. (same as before. If you already made one please skip this):

```
{
  "Version": "2021-5-5",
  "Statement": [
    {
      "Action": [
        "s3:GetObject",
        "s3:ListBucket"
      ],
      "Effect": "Allow",
      "Resource": [
        "arn:aws:s3:::my-bucketname/*"
      ],
      "Sid": ""
    }
  ]
}
```

2. add the policy to minio server(same as before. If you already made one please skip this):

```
#./mc admin policy add <alias> <policy_name> <local_policy_file>
./mc admin policy add minio getonly getonly.json
```

3. add a user within minio. Note: this user is only recognized by minio:

```
#./mc admin user add <alias> <username> <password>
./mc admin user add minio newuser newuser123
```

4. grant user with the policy:

```
#./mc admin policy set <alias> <policy> user=<username>
./mc admin policy set minio getonly user=newuser
```

7.1.3 Grant Policy to User with KeyCloak OpenID:

we can also assign the minio policy to keyCloak user or group. We can add the policy in MinIO and integrate with KeyCloak user/groups attributes:

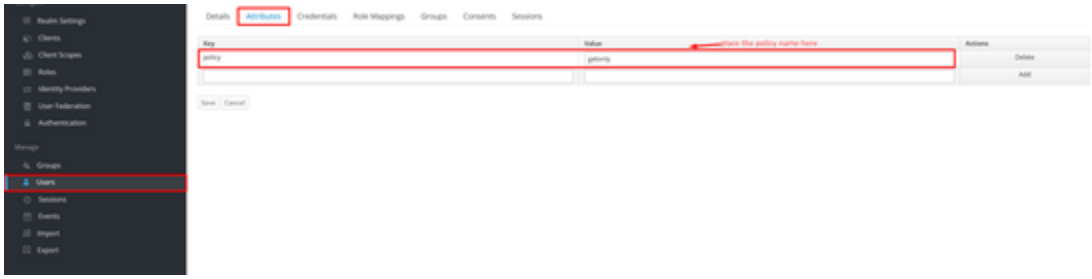
1. create a policy file at local machine. It will allow the user to **ONLY** access the bucket `my-bucketname`. (same as before. If you already made one please skip this):

```
{
  "Version": "2021-5-5",
  "Statement": [
    {
      "Action": [
        "s3:GetObject",
        "s3:ListBucket"
      ],
      "Effect": "Allow",
      "Resource": [
        "arn:aws:s3:::my-bucketname/*"
      ],
      "Sid": ""
    }
  ]
}
```

2. add the policy to minio server(same as before. If you already made one please skip this):

```
#./mc admin policy add <alias> <policy_name> <local_policy_file>
./mc admin policy add minio getonly getonly.json
```

3. go to KeyCloak and find the target user. In the attribute tab, add a pair as `policy:<policy_name>`. If there are multiple policies, it should be separated by comma as `policy:<policy_name_1>,<policy_name_2>`



4. Alternatively, we can also assign the policy a group. Afterwards, all users under the group will be assigned the same policy.

Note: the user policy will overwrite the group policy which means MinIO will apply user policy if he/she has one. Then if user does not have any policy specified, MinIO will apply the group policy.

7.1.4 Special Policy

We can setup the policy by user property in the minio https://docs.min.io/docs/minio-multi-user-quickstart-guide.html . For example, we can add policy for user that he can only access his name folder under the project.

```
    {
        "Version": "2012-10-17",
        "Statement": [
            {
            "Action": ["s3:ListBucket"],
            "Effect": "Allow",
            "Resource": ["arn:aws:s3:::gr-tes1123", "arn:aws:s3:::core-
tes1123"]
            },
            {
            "Action": [
                "s3:GetObject",
                "s3:PutObject"
            ],
            "Effect": "Allow",
            "Resource": ["arn:aws:s3:::gr-tes1123/${jwt:
preferred_username}/*", "arn:aws:s3:::core-tes1123/${jwt:
preferred_username}/*"]
            }
        ]
    }
```

7.2 Policy Inherent

The policy will inherent from parent folder. That means if a user is assigned to a policy in parent the folder, he/she will have **same permission** in the underlining folder. To make it clear, please check following example:

Example Bucket Structure

```
- test (Bucket)
|----- lv2
(subfolder)
```

Example Policy: Grant user to have `GetObject` and `ListBucket` for test bukcet.

```
{
  "Version":
"2012-10-17",
  "Statement":
[
    {

"Action": [
        "s3:
GetObject",
        "s3:
```

Example Policy: But deny all access to the `test/lv2` underneath.

```
{
  "Version":
"2012-10-17",
  "Statement":
[
    {

"Action": [
        "s3:
GetObject",
        "s3:
```

```
ListBucket"
      ],

"Effect":
"Allow", #
allow the get
and list

"Resource": [
        "arn:
aws:s3:::test
/*"
      ],
      "Sid": ""
    }
  ]
}
```

```
ListBucket"
      ],

"Effect":
"Deny", # here
block all
access

"Resource": [
        "arn:
aws:s3:::test
/lv2/*"
      ],
      "Sid": ""
    }
  ]
}
```

Result: From the testing, the parent policy will overwrite the child policy. In this case, the user will still have access to `test/lv2` , even if we set a policy to block it

8. Encryption

Minio supports the server side encryption https://docs.min.io/minio/baremetal/security/encryption/encryption-key-management.html#minio-encryption-sse-encryption-process . Below are some keys will be used during the encryption:

1. Customer Master Key: This this the key inside KMS or KES. It is specified by `MINIO_KMS_KES_KEY_NAME` when we start the MinIO
2. External Key(EK): EK is also called Data Encrytion Key. EK is generate per upload. MinIO will fetch a new key **for object every uploading**.
3. Key Encryption Key: The key is used to encryption the `OEK` in the metadata. This key is not stored anywhere. It will generate whenever the de/encryption operates by `KEK := PRF(EK, IV, context_values)`.
   a. `EK` is the external key.
   b. `IV` is the the random initial value per object in metadata.
   c. `context_values` is the additional parameter related to bucket or bound the range of `KEK`.
4. Object Encryption Key(OEK): `OEK` is random generated per object per upload. It will store as the object-wised metadata. You can treat it as hashing key.

8.1 Encryption Detail

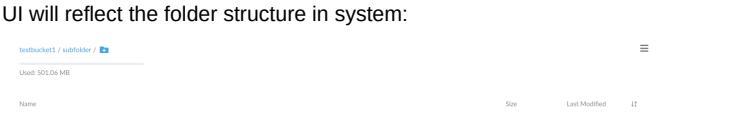
- We can use the api to set the encryption for specific bucket: `client.set_bucket_encryption("encrytion", SSEConfig(Rule.new_sse_s3_rule())`
- The key is generate per upload, meaning each upload will paired with a unique OEK. If versioning is enabled, then each version will paired with a unique OEK
- Afterwards, the file upload to server will be encrypted. If directly open the file, the contents are unreadable.
- If download operation triggers, the MinIO will decrypt the file on server side. And send back to clients.

---

9. Others:

- Storage mapping:

from the setup before we mount the `/data` folder as the default path with MinIO. by this all the action(eg. upload file, create bucket) will be submit into this folder. Take a closed look with the structure. Unlike the AWS S3, the MinIO use the tree structure. That means the level show in UI will reflect the folder structure in system:




```
root
  |---testbucket1
  |       |--------subfolder
  |--- ...
```

Limitation:

Throughout the test, here are some of the limitation of MinIO:

Setup Limitation:

1. When integrate with keycloak and try to use the Oauth2.0 to login, the dashboard shows the error of redirect url. This maybe the configure of the keycloak.
2. When we use the python SDK we need to use the long-term credential to invoke apis. There is a way to retrieve a temporary credential but it will require to run a `.go` script and open a web page. This would cost an extra effort when get the temporary credentials.

Bucket Limitation:

1. under the bucket, we cannot create empty folder unless there is an object
2. bucket has contraint same as S3 bucket that cannot have space, `.`

Tag Limitation(code):

1. number of tags has limit:
   a. bucket can have 50 tags.
   b. object can have 10 tags.
2. the key length cannot be greater than 128 chars
3. the value of tags must be string and length cannot be greater than 256 chars

Upload Limitation:

1. there is a default maximum upload size (5G) if the size greater than it, the api will trigger the multi-part upload.

SDK Limitation:

1. unable to configure ACL based on user groups
2. unable to manage user groups
3. unable to list folders on tree structure, without listing all the files in the folder.
4. No file list pagination