

Home Projects

Small or large home projects that usually involve setting up one system or another!

- Nextcloud 15 S3 Primary Storage using Minio and Docker
- Cloudflare Dynamic DNS

Nextcloud 15 S3 Primary Storage using Minio and Docker

Important note before we begin: Encryption will need to be enabled if you're using Nextcloud and S3 storage due to issues with files larger than 10mb. If you do not enable encryption you'll get an error similar to this which I have yet to find a workaround for:

```
A sha256 checksum could not be calculated for the provided upload body, because it was not seekable
```

Introduction

The aim of this project is to setup Nextcloud 15 inside docker using Minio as the primary storage.

For this project I'll be using my Manjaro based home computer but you can use whatever modern Linux system you have access to.

Before you start you'll obviously need `docker` but you'll also need `docker-compose` too. I won't go into details on how to set these up because there's a ton of guides out there which go over it in far more detail than I'm probably able for your specific system. You can also check out the docker documentation if you want to learn more.

After you've got docker and docker-compose setup and running we can start configuring our

containers.

Minio Container

I have a Minio container from a previous project so for this instance Minio will not be included in the docker-compose setup file and instead will be created as a separate instance and then added to our Nextcloud stack's network later.

I like to use bash scripts for each of my docker setups so that I can just easily run the commands already configured in the file which comes in extremely useful if you're moving between machines or if you don't want to have to remember how exactly you configured it. For Minio I use the following bash script:

```
#!/bin/bash
hport=9123
name="minio"
data="/opt/minio"
image="minio/minio"
function create {
    echo "Creating..."
    sudo mkdir -p $data
    sudo docker run -d -p 127.0.0.1:$hport:9000 \
        -v $data:/data \
        -e "MINIO_ACCESS_KEY=test" \
        -e "MINIO_SECRET_KEY=testtest" \
        --restart always \
        --name $name \
        $image server /data
    echo "Running $name on host port $hport"
}
function remove {
    echo "Removing..."
    sudo docker stop $name
    sudo docker rm $name
}
```

```

function update {
    echo "Pulling..."
    sudo docker pull $image
    remove
    create
}
case "$1" in
    update)
        update
        ;;

    remove)
        remove
        ;;

    create)
        create
        ;;

    *)
        echo $"Usage $0 {update|remove|create}"
        exit 1
esac

```

This script just sticks the docker commands into a nice little wrapper that we can run to simplify the creation and updating of the container. In this case I have set the Minio container to bind to host port `9123` and save its data in `/opt/minio`. I have also set the access and secret keys (which we'll need later) as simple strings for testing purposes. If you do not have access to your `/opt` folder you can simply change the directory to another path.

After you have copied and pasted the script to a file such as `minio.sh`, make the script executable with `chmod +x minio.sh` and run `./minio.sh create`. This will download all the required parts and run the container on the configured port (9123).

Minio does have a simple web frontend that you can visit by going to `http://localhost:9123` however it

does not have a lot of functionality and only really allows you to browse and manipulate buckets to a small degree. This is where Minio's shell client comes in super handy.

To obtain Minio's shell client visit the link: <https://docs.minio.io/docs/minio-client-complete-guide> and download the client using `wget` or similar then follow the instructions given. This will give you access to the `mc` client.

After you've got `mc` downloaded and working you can stick it in your path or use it as is. By sticking it in your path you can type `mc` to use it or you'll need to run it from wherever you placed it `/path/to/mc`.

Next we need to add our Minio server to our `mc` client list which can be done with the following command where `test` is the access key and `testtest` is the secret key as set in the Minio docker script above.

```
mc config host add minio http://localhost:9123 test testtest
```

Now that we've done that we can test it by typing `mc ls minio` which will list all buckets that our Minio server contains or nothing if it is a fresh install of Minio.

Obviously we don't want to use the admin account for our Nextcloud bucket so we'll have to create a new set of credentials.

Note: I won't go over how to set specific user-to-bucket policies so users will have access to all buckets on the Minio server. If you do not desire this then you'll need to set a policy for each user created.

To add a new user using `mc`, use the command:

```
mc admin user add minio nextcloud nextcloudnextcloud readwrite
```

This will add a user (access key) called `nextcloud` with the password (secret) `nextcloudnextcloud` with a global policy of `readwrite`.

We can now create a bucket for nextcloud to use: `mc mb minio/nextcloud`

Perfect, our Minio server is setup but we'll need to do one more thing later on to use it with our

Nextcloud containers.

Nextcloud Containers

Now we need to setup our Nextcloud containers using a docker-compose file. I have opted to use MySQL as the database just for cross-compatibility if I were to deploy this outside of my test environment.

Note: Nextcloud using a S3 service like Minio will save the metadata of the files in the configured database. This means that it is important to backup the database so that we do not lose file names and any other important information. Another huge factor is that we'll be running encryption with the keys being stored in the database too. This means that if you lose the database, you'll lose access to your files completely.

As stated before I like to use bash scripts to store my docker setups so the wrapper script for Nextcloud looks like this:

```
#!/bin/bash
name="nextcloud"
cdir="compose"
conf="nextcloud-docker-compose.yml"
function create {
    echo "Creating..."
    sudo docker-compose -f $cdir/$conf -p $name up -d
    echo "Running $name stack"
}
function remove {
    echo "Removing..."
    sudo docker-compose -f $cdir/$conf -p $name down
}
function update {
    echo "Pulling..."
    sudo docker-compose -f $cdir/$conf pull
    #remove
```

```

        create
    }
    case "$1" in
        update)
            update
            ;;

        remove)
            remove
            ;;

        create)
            create
            ;;

        *)
            echo $"Usage: $0 {update|remove|create}"
            exit 1
    esac

```

The script follows the same syntax as the Minio script above with slight differences due to using docker compose so we need to create the missing file next.

Create the directory `mkdir compose` and create the config file `compose/nextcloud-docker-compose.yml`.

```

version: '2'
volumes:
  nextcloud:
  db:
services:
  db:
    image: mariadb
    restart: always
    volumes:
      - db:/var/lib/mysql
    environment:

```

```
- MYSQL_ROOT_PASSWORD=wiggle
- MYSQL_PASSWORD=wiggle
- MYSQL_DATABASE=nextcloud
- MYSQL_USER=nextcloud

app:
  image: nextcloud
  ports:
    - 8483:80
  links:
    - db
  external_links:
    - minio
  volumes:
    - nextcloud:/var/www/html    restart: always
```

This will create the Nextcloud containers and make the web service available on port `8483`. You may wish to change the MySQL passwords to something more secure however this will be fine for testing purposes.

With that done we can now as similar to before run `chmod +x nextcloud.sh` and create the containers `./nextcloud.sh create`.

Putting It All Together

Perfect, that's done now however we have a couple of issues. Firstly networking wise our Minio container isn't connected to our Nextcloud network and secondly, the config that allows Nextcloud to use S3 object storage hasn't been created.

The first issue can be solved by connecting the Minio container to the Nextcloud stack's network by running the command: `docker network connect nextcloud_default minio`. This will solve the networking issue.

The second issue however is not as simple to deal with and will require a bit of tinkering with config files inside the docker volume. This means that we need to do a somewhat annoying workaround to

generate the required configs in the first place which means we'll need to setup Nextcloud twice.

First of all navigate to `http://localhost:8483` and follow the setup screen. Create a user of your choice and password (it doesn't matter what at this stage as it'll be deleted anyway) then click the `Storage & database` drop down and change it to the `Mysql/MariaDB` option. Fill the information with the user and password we configured earlier.

```
Database user: nextcloud
Database password: wiggle
Database name: nextcloudlocalhost: db
```

The important thing to note here is the field (database host) which we set as `db` in our docker compose file.

After the installation is complete we can move on to enabling object storage as the primary storage for Nextcloud.

When we ran the command to create the Nextcloud containers we made two containers, in my case those containers were called `nextcloud_app_1` and `nextcloud_db_1`. With their volumes called `nextcloud_db` and `nextcloud_nextcloud`. This means that on the container host system the volume we're after is by default under the `/var/lib/docker/volumes/nextcloud_nextcloud/_data` directory. This directory will contain the Nextcloud server files.

Inside the `config` folder will be a `config.php` file. Add the following lines to the array:

```
...
'objectstore' =>
array (
    'class' => 'OC\\Files\\ObjectStore\\S3',
    'arguments' =>
array (
    'bucket' => 'nextcloud',
    'autocreate' => false,
    'key' => 'nextcloud',
```

```
'secret' => 'nextcloudnextcloud',  
'hostname' => 'minio',  
'port' => 9000,  
'use_ssl' => false,  
'use_path_style' => true,  
,  
,  
); //End of the array
```

This tells Nextcloud to use Minio as primary storage for files. An important line is the `use_path_style` without which Nextcloud won't be able to connect to our Minio server without DNS configured. More information on this can be found in the Nextcloud documentation.

Now we need to recreate Nextcloud and reinstall it with using the new config file. Change the line `'installed' => true,` to `'installed' => false,` inside `config.php` and stop the Nextcloud containers `./nextcloud.sh remove` using the script we wrote earlier. Next remove the database volume `docker volume rm nextcloud_db` and then recreate the containers `./nextcloud.sh create`.

Navigate as previously to `http://localhost:8384` and fill in the information as before, install and with a bit of luck you'll reach a fresh Nextcloud instance without any error 500.

The final thing we need to do is set server side encryption to enabled so that we can upload files larger than 10 mbs. To do this using the Nextcloud web interface, go to `apps` and enable the `Default encryption module` then go to `settings` and under `Administration -> Security` tick the `Enable server-side-encryption checkbox`.

Nextcloud is now setup to use Minio as primary storage!

Cloudflare Dynamic DNS

Do you have your own domain on Cloudflare and want to update it like you would with a dynamic DNS service, but don't want to use said Dynamic DNS service?

Then this guide is for you!

What you'll need for these scripts to work for you:

- VPS or similar sitting in the cloud or at a remote location that is running a web server and PHP.
- Composer installed.
- Local VPS, server or anything that runs Linux with Cron and Curl.
- Already configured Cloudflare domain with a new sub domain setup for the dynamic DNS.
- Redis installed.

First of all we need to grab the required API so do a `composer require cloudflare/sdk`

Afterwards setup the following PHP script on your server along with the vendor folder you just downloaded.

```
<?php
$token = $_REQUEST["token"];
if ($token != "<some random string>")
    exit("Access denied.");
require_once __DIR__ . "/vendor/autoload.php";
$redis = new Redis();
$redis->connect("127.0.0.1", "6379");
$lastIP = $redis->get("dynamic:last-ip");
if (!$lastIP) $lastIP = "";
$m = "";
$skip = false;
```

```

$remoteIP = $_SERVER['REMOTE_ADDR'];
if (!filter_var($remoteIP, FILTER_VALIDATE_IP)) {
    $skip = true;
    $m .= "(Remote IP invalid)";
}
if (!$skip && $lastIP != "") {
    if ($remoteIP == $lastIP) $skip = true;
    $m .= "(LastIP and RemoteIP are the same)";
}
$redis->set("dynamic:last-ip", $remoteIP);
if (!$skip) {
    $key = new Cloudflare\API\Auth\APIKey('<email>', '<cloudflare API key>'); $adapter = new
    Cloudflare\API\Adapter\Guzzle($key);
    $zones = new \Cloudflare\API\Endpoints\Zones($adapter); $zoneID = $zones->getZoneID("<root domain
    name e.g example.com>");
    $dns = new \Cloudflare\API\Endpoints\DNS($adapter); $record = $dns->getRecordID($zoneID, "A",
    "<domain plus sub domain e.g: home.example.com>");
    $dns->updateRecordDetails($zoneID, $record, ["type" => "A", "name" => "<domain plus sub domain
    e.g: home.example.com>", "content" => $remoteIP]);
}
if (!$skip) {
    echo "done";
}else {
    echo "Skipped: ".$m;
}

```

You will see from the script that I'm using Redis to easily identify if I've already tried updating with the same IP address twice. This stops me flooding Cloudflare with useless requests but it can be removed if desired.

Next up on our home server we'll make a new Cron script to call the script we just made on the remote server. Do a `crontab -e` and add:

```

* * * * * curl -d "token=<your random string from above>" -X POST <url to the webserver running the

```

Now your DNS will be updated automagically every time it changes.

Enjoy!