scality-zenko-cloudserver

Release 7.0.0

Aug 15, 2023

Documentation

CHAPTER 1

Contributing

1.1 Need help?

We're always glad to help out. Simply open a GitHub issue and we'll give you insight. If what you want is not available, and if you're willing to help us out, we'll be happy to welcome you in the team, whether for a small fix or for a larger feature development. Thanks for your interest!

1.2 Got an idea? Get started!

In order to contribute, please follow the Contributing Guidelines. If anything is unclear to you, reach out to us on forum or via a GitHub issue.

1.3 Don't write code? There are other ways to help!

We're always eager to learn about our users' stories. If you can't contribute code, but would love to help us, please shoot us an email at zenko@scality.com, and tell us what our software enables you to do! Thanks for your time!

CHAPTER 2

Getting Started

2.1 Dependencies

Building and running the Scality Zenko CloudServer requires node.js 10.x and yarn v1.17.x. Up-to-date versions can be found at Nodesource.

2.2 Installation

1. Clone the source code

```
$ git clone https://github.com/scality/cloudserver.git
```

2. Go to the cloudserver directory and use yarn to install the js dependencies.

```
$ cd cloudserver
$ yarn install
```

2.3 Running CloudServer with a File Backend

```
$ yarn start
```

This starts a Zenko CloudServer on port 8000. Two additional ports, 9990 and 9991, are also open locally for internal transfer of metadata and data, respectively.

The default access key is accessKey1. The secret key is verySecretKey1.

By default, metadata files are saved in the localMetadata directory and data files are saved in the localData directory in the local ./cloudserver directory. These directories are pre-created within the repository. To save data or metadata in different locations, you must specify them using absolute paths. Thus, when starting the server:

```
$ mkdir -m 700 $(pwd)/myFavoriteDataPath
$ mkdir -m 700 $(pwd)/myFavoriteMetadataPath
$ export S3DATAPATH="$(pwd)/myFavoriteDataPath"
$ export S3METADATAPATH="$(pwd)/myFavoriteMetadataPath"
$ yarn start
```

2.4 Running CloudServer with Multiple Data Backends

```
$ export S3DATA='multiple'
$ yarn start
```

This starts a Zenko CloudServer on port 8000.

The default access key is accessKey1. The secret key is verySecretKey1.

With multiple backends, you can choose where each object is saved by setting the following header with a location constraint in a PUT request:

```
'x-amz-meta-scal-location-constraint':'myLocationConstraint'
```

If no header is sent with a PUT object request, the bucket's location constraint determines where the data is saved. If the bucket has no location constraint, the endpoint of the PUT request determines location.

See the *Configuration* section to set location constraints.

2.5 Run CloudServer with an In-Memory Backend

```
$ yarn run mem_backend
```

This starts a Zenko CloudServer on port 8000.

The default access key is accessKey1. The secret key is verySecretKey1.

2.6 Run CloudServer with Vault User Management

```
export S3VAULT=vault
yarn start
```

Note: Vault is proprietary and must be accessed separately. This starts a Zenko CloudServer using Vault for user management.

2.7 Run CloudServer for Continuous Integration Testing or in Production with Docker

Run Cloudserver with DOCKER

2.7.1 Testing

Run unit tests with the command:

```
$ yarn test
```

Run multiple-backend unit tests with:

```
$ CI=true S3DATA=multiple yarn start
$ yarn run multiple_backend_test
```

Run the linter with:

```
$ yarn run lint
```

2.8 Running Functional Tests Locally

To pass AWS and Azure backend tests locally, modify tests/locationConfig/locationConfigTests.json so that `awsbackend` specifies the bucketname of a bucket you have access to based on your credentials, and modify `azurebackend` with details for your Azure account.

The test suite requires additional tools, **s3cmd** and **Redis** installed in the environment the tests are running in.

1. Install s3cmd
2. Install redis and start Redis.
3. Add localCache section to `config.json`:

   ```
   "localCache": {
     "host": REDIS_HOST,
     "port": REDIS_PORT
     }
   ```

   where `REDIS_HOST` is the Redis instance IP address (`"127.0.0.1"` if Redis is running locally) and `REDIS_PORT` is the Redis instance port (`6379` by default)

4. Add the following to the local etc/hosts file:

   ```
   127.0.0.1 bucketwebsitetester.s3-website-us-east-1.amazonaws.com
   ```

5. Start Zenko CloudServer in memory and run the functional tests:

   ```
   $ CI=true yarn run mem_backend
   $ CI=true yarn run ft_test
   ```

2.9 Configuration

There are three configuration files for Zenko CloudServer:

- `conf/authdata.json`, for authentication.
- `locationConfig.json`, to configure where data is saved.
- `config.json`, for general configuration options.

2.9.1 Location Configuration

You must specify at least one locationConstraint in locationConfig.json (or leave it as pre-configured).

You must also specify 'us-east-1' as a locationConstraint. If you put a bucket to an unknown endpoint and do not specify a locationConstraint in the PUT bucket call, us-east-1 is used.

For instance, the following locationConstraint saves data sent to `myLocationConstraint` to the file backend:

```
"myLocationConstraint": {
    "type": "file",
    "legacyAwsBehavior": false,
    "details": {}
},
```

Each locationConstraint must include the `type`, `legacyAwsBehavior`, and `details` keys. `type` indicates which backend is used for that region. Supported backends are mem, file, and scality.''legacyAwsBehavior'' indicates whether the region behaves the same as the AWS S3 'us-east-1' region. If the locationConstraint type is `scality`, `details` must contain connector information for sproxyd. If the locationConstraint type is `mem` or `file`, `details` must be empty.

Once locationConstraints is set in locationConfig.json, specify a default locationConstraint for each endpoint.

For instance, the following sets the `localhost` endpoint to the `myLocationConstraint` data backend defined above:

```
"restEndpoints": {
     "localhost": "myLocationConstraint"
},
```

To use an endpoint other than localhost for Zenko CloudServer, the endpoint must be listed in `restEndpoints`. Otherwise, if the server is running with a:

- **file backend**: The default location constraint is `file`
- **memory backend**: The default location constraint is `mem`

2.9.2 Endpoints

The Zenko CloudServer supports endpoints that are rendered in either:

- path style: http://myhostname.com/mybucket or
- hosted style: http://mybucket.myhostname.com

However, if an IP address is specified for the host, hosted-style requests cannot reach the server. Use path-style requests in that case. For example, if you are using the AWS SDK for JavaScript, instantiate your client like this:

```
const s3 = new aws.S3({
   endpoint: 'http://127.0.0.1:8000',
   s3ForcePathStyle: true,
});
```

2.9.3 Setting Your Own Access and Secret Key Pairs

Credentials can be set for many accounts by editing `conf/authdata.json`, but use the `SCALITY_ACCESS_KEY_ID` and `SCALITY_SECRET_ACCESS_KEY` environment variables to specify your own credentials.

scality-access-key-id-and-scality-secret-access-key

SCALITY_ACCESS_KEY_ID and SCALITY_SECRET_ACCESS_KEY

These variables specify authentication credentials for an account named "CustomAccount".

Note: Anything in the `authdata.json` file is ignored.

```
$ SCALITY_ACCESS_KEY_ID=newAccessKey SCALITY_SECRET_ACCESS_KEY=newSecretKey yarn start
```

2.9.4 Using SSL

To use https with your local CloudServer, you must set up SSL certificates.

1. Deploy CloudServer using our DockerHub page (run it with a file backend).

   Note: If Docker is not installed locally, follow the instructions to install it for your distribution

2. Update the CloudServer container's config

   Add your certificates to your container. To do this, #. exec inside the CloudServer container.

   1. Run `$> docker ps` to find the container's ID (the corresponding image name is `scality/cloudserver`.
   2. Copy the corresponding container ID (`894aee038c5e` in the present example), and run:

      ```
      $> docker exec -it 894aee038c5e bash
      ```

      This puts you inside your container, using an interactive terminal.

3. Generate the SSL key and certificates. The paths where the different files are stored are defined after the `-out` option in each of the following commands.

   1. Generate a private key for your certificate signing request (CSR):

      ```
      $> openssl genrsa -out ca.key 2048
      ```

   2. Generate a self-signed certificate for your local certificate authority (CA):

```
$> openssl req -new -x509 -extensions v3_ca -key ca.key -out ca.crt -days␣
↪99999  -subj "/C=US/ST=Country/L=City/O=Organization/CN=scality.test"
```

3. Generate a key for the CloudServer:

```
$> openssl genrsa -out test.key 2048
```

4. Generate a CSR for CloudServer:

```
$> openssl req -new -key test.key -out test.csr -subj "/C=US/ST=Country/
↪L=City/O=Organization/CN=*.scality.test"
```

5. Generate a certificate for CloudServer signed by the local CA:

```
$> openssl x509 -req -in test.csr -CA ca.crt -CAkey ca.key -CAcreateserial -
↪out test.crt -days 99999 -sha256
```

4. Update Zenko CloudServer `config.json`. Add a `certFilePaths` section to `./config.json` with appropriate paths:

```
"certFilePaths": {
    "key": "./test.key",
    "cert": "./test.crt",
    "ca": "./ca.crt"
}
```

5. Run your container with the new config.
   1. Exit the container by running `$> exit`.
   2. Restart the container with `$> docker restart cloudserver`.
6. Update the host configuration by adding s3.scality.test to /etc/hosts:

```
127.0.0.1       localhost s3.scality.test
```

7. Copy the local certificate authority (ca.crt in step 4) from your container. Choose the path to save this file to (in the present example, `/root/ca.crt`), and run:

```
$> docker cp 894aee038c5e:/usr/src/app/ca.crt /root/ca.crt
```

Note: Your container ID will be different, and your path to ca.crt may be different.

Test the Config

If aws-sdk is not installed, run `$> yarn install aws-sdk`.

Paste the following script into a file named "test.js":

```
const AWS = require('aws-sdk');
const fs = require('fs');
const https = require('https');

const httpOptions = {
    agent: new https.Agent({
```

(continues on next page)

(continued from previous page)

```
        // path on your host of the self-signed certificate
        ca: fs.readFileSync('./ca.crt', 'ascii'),
    }),
};

const s3 = new AWS.S3({
    httpOptions,
    accessKeyId: 'accessKey1',
    secretAccessKey: 'verySecretKey1',
    // The endpoint must be s3.scality.test, else SSL will not work
    endpoint: 'https://s3.scality.test:8000',
    sslEnabled: true,
    // With this setup, you must use path-style bucket access
    s3ForcePathStyle: true,
});

const bucket = 'cocoriko';

s3.createBucket({ Bucket: bucket }, err => {
    if (err) {
        return console.log('err createBucket', err);
    }
    return s3.deleteBucket({ Bucket: bucket }, err => {
        if (err) {
            return console.log('err deleteBucket', err);
        }
        return console.log('SSL is cool!');
    });
});
```

Now run this script with:

```
$> nodejs test.js
```

On success, the script outputs `SSL is cool!`.

CHAPTER 3

Using Public Clouds as data backends

3.1 Introduction

As stated in our GETTING STARTED guide, new data backends can be added by creating a region (also called location constraint) with the right endpoint and credentials. This section of the documentation shows you how to set up our currently supported public cloud backends:

- *Amazon S3* ;
- *Microsoft Azure* .

For each public cloud backend, you will have to edit your CloudServer `locationConfig.json` and do a few setup steps on the applicable public cloud backend.

3.2 AWS S3 as a data backend

3.2.1 From the AWS S3 Console (or any AWS S3 CLI tool)

Create a bucket where you will host your data for this new location constraint. This bucket must have versioning enabled:

- This is an option you may choose to activate at step 2 of Bucket Creation in the Console;
- With AWS CLI, use `put-bucket-versioning` from the `s3api` commands on your bucket of choice;
- Using other tools, please refer to your tool's documentation.

In this example, our bucket will be named `zenkobucket` and has versioning enabled.

3.2.2 From the CloudServer repository

locationConfig.json

Edit this file to add a new location constraint. This location constraint will contain the information for the AWS S3 bucket to which you will be writing your data whenever you create a CloudServer bucket in this location. There are a few configurable options here:

- `type` : set to `aws_s3` to indicate this location constraint is writing data to AWS S3;
- `legacyAwsBehavior` : set to `true` to indicate this region should behave like AWS S3 `us-east-1` region, set to `false` to indicate this region should behave like any other AWS S3 region;
- `bucketName` : set to an *existing bucket* in your AWS S3 Account; this is the bucket in which your data will be stored for this location constraint;
- `awsEndpoint` : set to your bucket's endpoint, usually `s3.amazonaws.com`;
- `bucketMatch` : set to `true` if you want your object name to be the same in your local bucket and your AWS S3 bucket; set to `false` if you want your object name to be of the form `{{localBucketName}}/{{objectname}}` in your AWS S3 hosted bucket;
- `credentialsProfile` and `credentials` are two ways to provide your AWS S3 credentials for that bucket, *use only one of them* :
  - `credentialsProfile` : set to the profile name allowing you to access your AWS S3 bucket from your `~/.aws/credentials` file;
  - `credentials` : set the two fields inside the object (`accessKey` and `secretKey`) to their respective values from your AWS credentials.

```
(...)
"aws-test": {
    "type": "aws_s3",
    "legacyAwsBehavior": true,
    "details": {
        "awsEndpoint": "s3.amazonaws.com",
        "bucketName": "zenkobucket",
        "bucketMatch": true,
        "credentialsProfile": "zenko"
    }
},
(...)
```

```
(...)
"aws-test": {
    "type": "aws_s3",
    "legacyAwsBehavior": true,
    "details": {
        "awsEndpoint": "s3.amazonaws.com",
        "bucketName": "zenkobucket",
        "bucketMatch": true,
        "credentials": {
            "accessKey": "WHDBFKILOSDDVF78NPMQ",
            "secretKey": "87hdfGCvDS+YYzefKLnjjZEYstOIuIjs/2X72eET"
        }
    }
},
(...)
```

Warning: If you set `bucketMatch` to `true`, we strongly advise that you only have one local bucket per AWS S3 location. Without `bucketMatch` set to `false`, your object names in your AWS S3 bucket will not be prefixed with your Cloud Server bucket name. This means that if you put an object `foo` to your CloudServer bucket `zenko1` and you then put a different `foo` to your CloudServer bucket `zenko2` and both `zenko1` and `zenko2` point to the same AWS bucket, the second `foo` will overwrite the first `foo`.

~/.aws/credentials

Tip: If you explicitly set your `accessKey` and `secretKey` in the `credentials` object of your `aws_s3` location in your `locationConfig.json` file, you may skip this section

Make sure your `~/.aws/credentials` file has a profile matching the one defined in your `locationConfig.json`. Following our previous example, it would look like:

```
[zenko]
aws_access_key_id=WHDBFKILOSDDVF78NPMQ
aws_secret_access_key=87hdfGCvDS+YYzefKLnjjZEYstOIuIjs/2X72eET
```

3.2.3 Start the server with the ability to write to AWS S3

Inside the repository, once all the files have been edited, you should be able to start the server and start writing data to AWS S3 through CloudServer.

```
# Start the server locally
$> S3DATA=multiple yarn start
```

3.2.4 Run the server as a docker container with the ability to write to AWS S3

Tip: If you set the `credentials` object in your `locationConfig.json` file, you don't need to mount your `.aws/credentials` file

Mount all the files that have been edited to override defaults, and do a standard Docker run; then you can start writing data to AWS S3 through CloudServer.

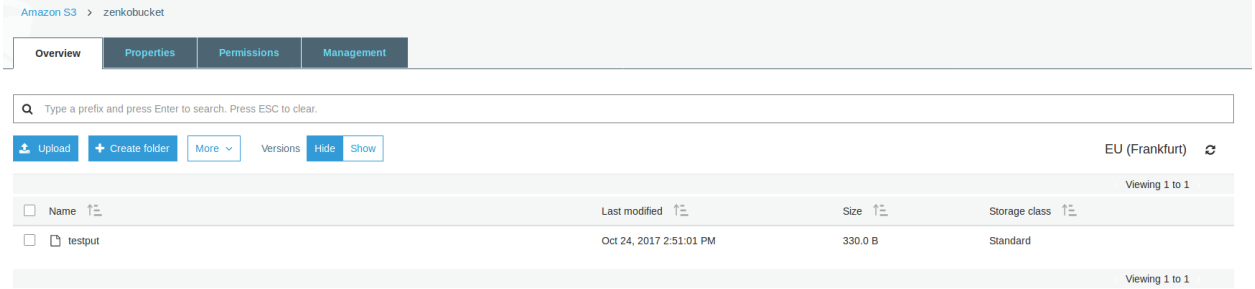
```
# Start the server in a Docker container
$> sudo docker run -d --name CloudServer \
-v $(pwd)/data:/usr/src/app/localData \
-v $(pwd)/metadata:/usr/src/app/localMetadata \
-v $(pwd)/locationConfig.json:/usr/src/app/locationConfig.json \
-v $(pwd)/conf/authdata.json:/usr/src/app/conf/authdata.json \
-v ~/.aws/credentials:/root/.aws/credentials \
-e S3DATA=multiple -e ENDPOINT=http://localhost -p 8000:8000 \
-d scality/cloudserver
```

3.2.5 Testing: put an object to AWS S3 using CloudServer

In order to start testing pushing to AWS S3, you will need to create a local bucket in the AWS S3 location constraint - this local bucket will only store the metadata locally, while both the data and any user metadata (`x-amz-meta` headers sent with a PUT object, and tags) will be stored on AWS S3. This example is based on all our previous steps.

```
# Create a local bucket storing data in AWS S3
$> s3cmd --host=127.0.0.1:8000 mb s3://zenkobucket --region=aws-test
# Put an object to AWS S3, and store the metadata locally
$> s3cmd --host=127.0.0.1:8000 put /etc/hosts s3://zenkobucket/testput
 upload: '/etc/hosts' -> 's3://zenkobucket/testput'  [1 of 1]
  330 of 330   100% in    0s   380.87 B/s  done
# List locally to check you have the metadata
$> s3cmd --host=127.0.0.1:8000 ls s3://zenkobucket
 2017-10-23 10:26       330   s3://zenkobucket/testput
```

Then, from the AWS Console, if you go into your bucket, you should see your newly uploaded object:



3.2.6 Troubleshooting

Make sure your `~/.s3cfg` file has credentials matching your local CloudServer credentials defined in `conf/authdata.json`. By default, the access key is `accessKey1` and the secret key is `verySecretKey1`. For more informations, refer to our template ~/.s3cfg .

Pre-existing objects in your AWS S3 hosted bucket can unfortunately not be accessed by CloudServer at this time.

Make sure versioning is enabled in your remote AWS S3 hosted bucket. To check, using the AWS Console, click on your bucket name, then on "Properties" at the top, and then you should see something like this:

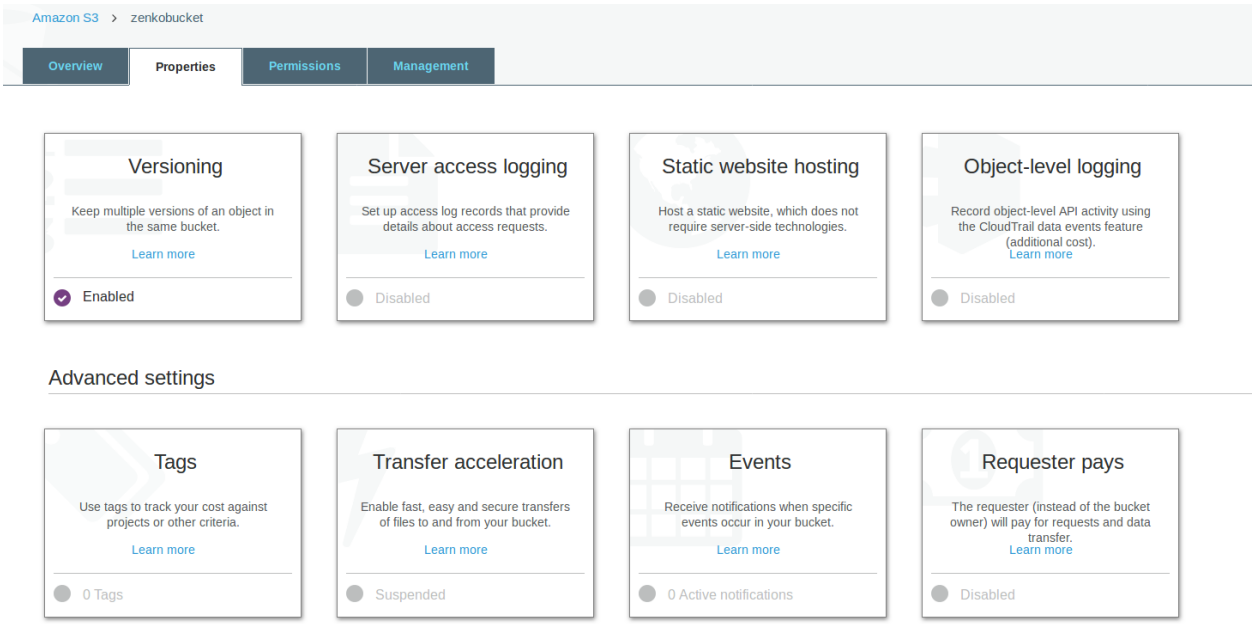
3.3 Microsoft Azure as a data backend

3.3.1 From the MS Azure Console

From your Storage Account dashboard, create a container where you will host your data for this new location constraint.

You will also need to get one of your Storage Account Access Keys, and to provide it to CloudServer. This can be found from your Storage Account dashboard, under "Settings, then "Access keys".

In this example, our container will be named `zenkontainer`, and will belong to the `zenkomeetups` Storage Account.



3.3.2 From the CloudServer repository

locationConfig.json

Edit this file to add a new location constraint. This location constraint will contain the information for the MS Azure container to which you will be writing your data whenever you create a CloudServer bucket in this location. There are a few configurable options here:

- `type` : set to `azure` to indicate this location constraint is writing data to MS Azure;
- `legacyAwsBehavior` : set to `true` to indicate this region should behave like AWS S3 `us-east-1` region, set to `false` to indicate this region should behave like any other AWS S3 region (in the case of MS Azure hosted data, this is mostly relevant for the format of errors);
- `azureStorageEndpoint` : set to your storage account's endpoint, usually `https://{{storageAccountName}}.blob.core.windows.net`;
- `azureContainerName` : set to an *existing container* in your MS Azure storage account; this is the container in which your data will be stored for this location constraint;
- `bucketMatch` : set to `true` if you want your object name to be the same in your local bucket and your MS Azure container; set to `false` if you want your object name to be of the form `{{localBucketName}}/{{objectname}}` in your MS Azure container ;
- `azureStorageAccountName` : the MS Azure Storage Account to which your container belongs;
- `azureStorageAccessKey` : one of the Access Keys associated to the above defined MS Azure Storage Account.

```
(...)
"azure-test": {
    "type": "azure",
    "legacyAwsBehavior": false,
    "details": {
```

(continues on next page)

(continued from previous page)

```
      "azureStorageEndpoint": "https://zenkomeetups.blob.core.windows.net/",
      "bucketMatch": true,
      "azureContainerName": "zenkontainer",
      "azureStorageAccountName": "zenkomeetups",
      "azureStorageAccessKey":
↪"auhyDo8izbuU4aZGdhxnWh0ODKFP3IWjsN1UfFaoqFbnYzPj9bxeCVAzTIcgzdgqomDKx6QS+8ov8PYCON0Nxw==
↪"
    }
},
(...)
```

Warning: If you set `bucketMatch` to `true`, we strongly advise that you only have one local bucket per MS Azure location. Without `bucketMatch` set to `false`, your object names in your MS Azure container will not be prefixed with your Cloud Server bucket name. This means that if you put an object `foo` to your CloudServer bucket `zenko1` and you then put a different `foo` to your CloudServer bucket `zenko2` and both `zenko1` and `zenko2` point to the same MS Azure container, the second `foo` will overwrite the first `foo`.

Tip: You may export environment variables to **override** some of your `locationConfig.json` variable ; the syntax for them is `{{region-name}}_{{ENV_VAR_NAME}}`; currently, the available variables are those shown below, with the values used in the current example:

```
$> export azure-test_AZURE_STORAGE_ACCOUNT_NAME="zenkomeetups"
$> export azure-test_AZURE_STORAGE_ACCESS_KEY=
↪"auhyDo8izbuU4aZGdhxnWh0ODKFP3IWjsN1UfFaoqFbnYzPj9bxeCVAzTIcgzdgqomDKx6QS+8ov8PYCON0Nxw==
↪"
$> export azure-test_AZURE_STORAGE_ENDPOINT="https://zenkomeetups.blob.core.windows.
↪net/"
```

3.3.3 Start the server with the ability to write to MS Azure

Inside the repository, once all the files have been edited, you should be able to start the server and start writing data to MS Azure through CloudServer.

```
# Start the server locally
$> S3DATA=multiple yarn start
```

3.3.4 Run the server as a docker container with the ability to write to MS Azure

Mount all the files that have been edited to override defaults, and do a standard Docker run; then you can start writing data to MS Azure through CloudServer.

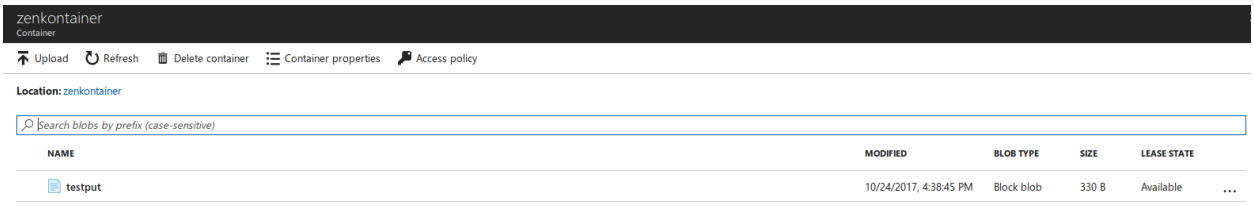
```
# Start the server in a Docker container
$> sudo docker run -d --name CloudServer \
-v $(pwd)/data:/usr/src/app/localData \
-v $(pwd)/metadata:/usr/src/app/localMetadata \
-v $(pwd)/locationConfig.json:/usr/src/app/locationConfig.json \
-v $(pwd)/conf/authdata.json:/usr/src/app/conf/authdata.json \
-e S3DATA=multiple -e ENDPOINT=http://localhost -p 8000:8000
-d scality/cloudserver
```

3.3.5 Testing: put an object to MS Azure using CloudServer

In order to start testing pushing to MS Azure, you will need to create a local bucket in the MS Azure region - this local bucket will only store the metadata locally, while both the data and any user metadata (`x-amz-meta` headers sent with a PUT object, and tags) will be stored on MS Azure. This example is based on all our previous steps.

```
# Create a local bucket storing data in MS Azure
$> s3cmd --host=127.0.0.1:8000 mb s3://zenkontainer --region=azure-test
# Put an object to MS Azure, and store the metadata locally
$> s3cmd --host=127.0.0.1:8000 put /etc/hosts s3://zenkontainer/testput
 upload: '/etc/hosts' -> 's3://zenkontainer/testput'  [1 of 1]
  330 of 330   100% in    0s   380.87 B/s  done
# List locally to check you have the metadata
$> s3cmd --host=127.0.0.1:8000 ls s3://zenkobucket
 2017-10-24 14:38       330   s3://zenkontainer/testput
```

Then, from the MS Azure Console, if you go into your container, you should see your newly uploaded object:



3.3.6 Troubleshooting

Make sure your `~/.s3cfg` file has credentials matching your local CloudServer credentials defined in `conf/authdata.json`. By default, the access key is `accessKey1` and the secret key is `verySecretKey1`. For more informations, refer to our template ~/.s3cfg .

Pre-existing objects in your MS Azure container can unfortunately not be accessed by CloudServer at this time.

3.4 For any data backend

3.4.1 From the CloudServer repository

config.json

Important: You only need to follow this section if you want to define a given location as the default for a specific endpoint

Edit the `restEndpoint` section of your `config.json` file to add an endpoint definition matching the location you want to use as a default for an endpoint to this specific endpoint. In this example, we'll make `custom-location` our default location for the endpoint `zenkotos3.com`:

```
(...)
"restEndpoints": {
    "localhost": "us-east-1",
    "127.0.0.1": "us-east-1",
```

(continues on next page)

(continued from previous page)

```
    "cloudserver-front": "us-east-1",
    "s3.docker.test": "us-east-1",
    "127.0.0.2": "us-east-1",
    "zenkotos3.com": "custom-location"
},
(...)
```

CHAPTER 4

Clients

List of applications that have been tested with Zenko CloudServer.

4.1 GUI

4.1.1 Cyberduck

- https://www.youtube.com/watch?v=-n2MCt4ukUg
- https://www.youtube.com/watch?v=IyXHcu4uqgU

4.1.2 Cloud Explorer

- https://www.youtube.com/watch?v=2hhtBtmBSxE

4.1.3 CloudBerry Lab

- https://youtu.be/IjIx8g_o0gY

4.2 Command Line Tools

4.2.1 s3curl

https://github.com/scality/S3/blob/master/tests/functional/s3curl/s3curl.pl

4.2.2 aws-cli

`~/.aws/credentials` on Linux, OS X, or Unix or `C:\Users\USERNAME\.aws\credentials` on Windows

```
[default]
aws_access_key_id = accessKey1
aws_secret_access_key = verySecretKey1
```

`~/.aws/config` on Linux, OS X, or Unix or `C:\Users\USERNAME\.aws\config` on Windows

```
[default]
region = us-east-1
```

Note: `us-east-1` is the default region, but you can specify any region.

See all buckets:

```
aws s3 ls --endpoint-url=http://localhost:8000
```

Create bucket:

```
aws --endpoint-url=http://localhost:8000 s3 mb s3://mybucket
```

4.2.3 s3cmd

If using s3cmd as a client to S3 be aware that v4 signature format is buggy in s3cmd versions < 1.6.1.

`~/.s3cfg` on Linux, OS X, or Unix or `C:\Users\USERNAME\.s3cfg` on Windows

```
[default]
access_key = accessKey1
secret_key = verySecretKey1
host_base = localhost:8000
host_bucket = %(bucket).localhost:8000
signature_v2 = False
use_https = False
```

See all buckets:

```
s3cmd ls
```

4.2.4 rclone

`~/.rclone.conf` on Linux, OS X, or Unix or `C:\Users\USERNAME\.rclone.conf` on Windows

```
[remote]
type = s3
env_auth = false
access_key_id = accessKey1
secret_access_key = verySecretKey1
region = other-v2-signature
endpoint = http://localhost:8000
location_constraint =
acl = private
```

(continues on next page)

(continued from previous page)

```
server_side_encryption =
storage_class =
```

See all buckets:

```
rclone lsd remote:
```

4.3 JavaScript

4.3.1 AWS JavaScript SDK

```
const AWS = require('aws-sdk');

const s3 = new AWS.S3({
    accessKeyId: 'accessKey1',
    secretAccessKey: 'verySecretKey1',
    endpoint: 'localhost:8000',
    sslEnabled: false,
    s3ForcePathStyle: true,
});
```

4.4 JAVA

4.4.1 AWS JAVA SDK

```
import com.amazonaws.auth.AWSCredentials;
import com.amazonaws.auth.BasicAWSCredentials;
import com.amazonaws.services.s3.AmazonS3;
import com.amazonaws.services.s3.AmazonS3Client;
import com.amazonaws.services.s3.S3ClientOptions;
import com.amazonaws.services.s3.model.Bucket;

public class S3 {

    public static void main(String[] args) {

        AWSCredentials credentials = new BasicAWSCredentials("accessKey1",
        "verySecretKey1");

        // Create a client connection based on credentials
        AmazonS3 s3client = new AmazonS3Client(credentials);
        s3client.setEndpoint("http://localhost:8000");
        // Using path-style requests
        // (deprecated) s3client.setS3ClientOptions(new S3ClientOptions().
↪withPathStyleAccess(true));
        s3client.setS3ClientOptions(S3ClientOptions.builder().
↪setPathStyleAccess(true).build());

        // Create bucket
        String bucketName = "javabucket";
```

(continues on next page)

(continued from previous page)

```
        s3client.createBucket(bucketName);

        // List off all buckets
        for (Bucket bucket : s3client.listBuckets()) {
            System.out.println(" - " + bucket.getName());
        }
    }
}
```

4.5 Ruby

4.5.1 AWS SDK for Ruby - Version 2

```
require 'aws-sdk'

s3 = Aws::S3::Client.new(
  :access_key_id => 'accessKey1',
  :secret_access_key => 'verySecretKey1',
  :endpoint => 'http://localhost:8000',
  :force_path_style => true
)

resp = s3.list_buckets
```

4.5.2 fog

```
require "fog"

connection = Fog::Storage.new(
{
    :provider => "AWS",
    :aws_access_key_id => 'accessKey1',
    :aws_secret_access_key => 'verySecretKey1',
    :endpoint => 'http://localhost:8000',
    :path_style => true,
    :scheme => 'http',
})
```

4.6 Python

4.6.1 boto2

```
import boto
from boto.s3.connection import S3Connection, OrdinaryCallingFormat


connection = S3Connection(
    aws_access_key_id='accessKey1',
```

(continues on next page)

(continued from previous page)

```
    aws_secret_access_key='verySecretKey1',
    is_secure=False,
    port=8000,
    calling_format=OrdinaryCallingFormat(),
    host='localhost'
)

connection.create_bucket('mybucket')
```

4.6.2 boto3

Client integration

```
import boto3

client = boto3.client(
    's3',
    aws_access_key_id='accessKey1',
    aws_secret_access_key='verySecretKey1',
    endpoint_url='http://localhost:8000'
)

lists = client.list_buckets()
```

Full integration (with object mapping)

```
import os

from botocore.utils import fix_s3_host
import boto3

os.environ['AWS_ACCESS_KEY_ID'] = "accessKey1"
os.environ['AWS_SECRET_ACCESS_KEY'] = "verySecretKey1"

s3 = boto3.resource(service_name='s3', endpoint_url='http://localhost:8000')
s3.meta.client.meta.events.unregister('before-sign.s3', fix_s3_host)

for bucket in s3.buckets.all():
    print(bucket.name)
```

4.7 PHP

Should force path-style requests even though v3 advertises it does by default.

4.7.1 AWS PHP SDK v3

```
use Aws\S3\S3Client;

$client = S3Client::factory([
    'region'  => 'us-east-1',
    'version'   => 'latest',
```

(continues on next page)

(continued from previous page)

```
    'endpoint' => 'http://localhost:8000',
    'use_path_style_endpoint' => true,
    'credentials' => [
        'key'    => 'accessKey1',
        'secret' => 'verySecretKey1'
    ]
]);

$client->createBucket(array(
    'Bucket' => 'bucketphp',
));
```

4.8 Go

4.8.1 AWS Go SDK

```
package main

import (
   "context"
   "fmt"
   "log"
   "os"
   "time"

   "github.com/aws/aws-sdk-go/aws"
   "github.com/aws/aws-sdk-go/aws/endpoints"
   "github.com/aws/aws-sdk-go/aws/session"
   "github.com/aws/aws-sdk-go/service/s3"
)

func main() {
   os.Setenv("AWS_ACCESS_KEY_ID", "accessKey1")
   os.Setenv("AWS_SECRET_ACCESS_KEY", "verySecretKey1")
   endpoint := "http://localhost:8000"
   timeout := time.Duration(10) * time.Second
   sess := session.Must(session.NewSession())

   // Create a context with a timeout that will abort the upload if it takes
   // more than the passed in timeout.
   ctx, cancel := context.WithTimeout(context.Background(), timeout)
   defer cancel()

   svc := s3.New(sess, &aws.Config{
      Region:   aws.String(endpoints.UsEast1RegionID),
      Endpoint: &endpoint,
   })

   out, err := svc.ListBucketsWithContext(ctx, &s3.ListBucketsInput{})
   if err != nil {
      log.Fatal(err)
   } else {
      fmt.Println(out)
```

(continues on next page)

(continued from previous page)

```
    }
}
```

CHAPTER 5

Docker

5.1 Environment Variables

5.1.1 S3DATA

S3DATA=multiple

This variable enables running CloudServer with multiple data backends, defined as regions.

For multiple data backends, a custom locationConfig.json file is required. This file enables you to set custom regions. You must provide associated rest_endpoints for each custom region in config.json.

Learn more about multiple-backend configurations

If you are using Scality RING endpoints, refer to your customer documentation.

Running CloudServer with an AWS S3-Hosted Backend

To run CloudServer with an S3 AWS backend, add a new section to the `locationConfig.json` file with the `aws_s3` location type:

```
(...)
"awsbackend": {
    "type": "aws_s3",
    "details": {
        "awsEndpoint": "s3.amazonaws.com",
        "bucketName": "yourawss3bucket",
        "bucketMatch": true,
        "credentialsProfile": "aws_hosted_profile"
    }
}
(...)
```

Edit your AWS credentials file to enable your preferred command-line tool. This file must mention credentials for all backends in use. You can use several profiles if multiple profiles are configured.

```
[default]
aws_access_key_id=accessKey1
aws_secret_access_key=verySecretKey1
[aws_hosted_profile]
aws_access_key_id={{YOUR_ACCESS_KEY}}
aws_secret_access_key={{YOUR_SECRET_KEY}}
```

As with locationConfig.json, the AWS credentials file must be mounted at run time: `-v ~/.aws/credentials:/root/.aws/credentials` on Unix-like systems (Linux, OS X, etc.), or `-v C:\Users\USERNAME\.aws\credential:/root/.aws/credentials` on Windows

Note: One account cannot copy to another account with a source and destination on real AWS unless the account associated with the accessKey/secretKey pairs used for the destination bucket has source bucket access privileges. To enable this, update ACLs directly on AWS.

5.1.2 S3BACKEND

S3BACKEND=file

For stored file data to persist, you must mount Docker volumes for both data and metadata. See *In Production with a Docker-Hosted CloudServer*

S3BACKEND=mem

This is ideal for testing: no data remains after the container is shut down.

5.1.3 ENDPOINT

This variable specifies the endpoint. To direct CloudServer requests to new.host.com, for example, specify the endpoint with:

```
$ docker run -d --name cloudserver -p 8000:8000 -e ENDPOINT=new.host.com zenko/
↪cloudserver
```

Note: On Unix-like systems (Linux, OS X, etc.) edit /etc/hosts to associate 127.0.0.1 with new.host.com.

5.1.4 REMOTE_MANAGEMENT_DISABLE

CloudServer is a part of Zenko. When you run CloudServer standalone it will still try to connect to Orbit by default (browser-based graphical user interface for Zenko).

Setting this variable to true(1) will default to accessKey1 and verySecretKey1 for credentials and disable the automatic Orbit management:

```
$ docker run -d --name cloudserver -p 8000:8000 -e REMOTE_MANAGEMENT_DISABLE=1 zenko/
↪cloudserver
```

5.1.5 SCALITY_ACCESS_KEY_ID and SCALITY_SECRET_ACCESS_KEY

These variables specify authentication credentials for an account named "CustomAccount".

Set account credentials for multiple accounts by editing conf/authdata.json (see below for further details). To specify one set for personal use, set these environment variables:

```
$ docker run -d --name cloudserver -p 8000:8000 -e SCALITY_ACCESS_KEY_ID=newAccessKey␣
↪\
-e SCALITY_SECRET_ACCESS_KEY=newSecretKey zenko/cloudserver
```

Note: This takes precedence over the contents of the authdata.json file. The authdata.json file is ignored.

Note: The ACCESS_KEY and SECRET_KEY environment variables are deprecated.

5.1.6 LOG_LEVEL

This variable changes the log level. There are three levels: info, debug, and trace. The default is info. Debug provides more detailed logs, and trace provides the most detailed logs.

```
$ docker run -d --name cloudserver -p 8000:8000 -e LOG_LEVEL=trace zenko/cloudserver
```

5.1.7 SSL

Set true, this variable runs CloudServer with SSL.

If SSL is set true:

- The ENDPOINT environment variable must also be specified.
- On Unix-like systems (Linux, OS X, etc.), 127.0.0.1 must be associated with <YOUR_ENDPOINT> in /etc/hosts.

> **Warning:** Self-signed certs with a CA generated within the container are suitable for testing purposes only. Clients cannot trust them, and they may disappear altogether on a container upgrade. The best security practice for production environments is to use an extra container, such as haproxy/nginx/stunnel, for SSL/TLS termination and to pull certificates from a mounted volume, limiting what an exploit on either component can expose.

```
   $ docker run -d --name cloudserver -p 8000:8000 -e SSL=TRUE -e ENDPOINT=<YOUR_
↪ENDPOINT> \
   zenko/cloudserver

For more information about using ClousdServer with SSL, see `Using SSL <GETTING_
↪STARTED.html#Using SSL>`__
```

5.1.8 LISTEN_ADDR

This variable causes CloudServer and its data and metadata components to listen on the specified address. This allows starting the data or metadata servers as standalone services, for example.

```
docker run -d --name s3server-data -p 9991:9991 -e LISTEN_ADDR=0.0.0.0
scality/s3server yarn run start_dataserver
```

5.1.9 DATA_HOST and METADATA_HOST

These variables configure the data and metadata servers to use, usually when they are running on another host and only starting the stateless Zenko CloudServer.

```
$ docker run -d --name cloudserver -e DATA_HOST=cloudserver-data \
-e METADATA_HOST=cloudserver-metadata zenko/cloudserver yarn run start_s3server
```

5.1.10 REDIS_HOST

Use this variable to connect to the redis cache server on another host than localhost.

```
$ docker run -d --name cloudserver -p 8000:8000 \
-e REDIS_HOST=my-redis-server.example.com zenko/cloudserver
```

5.1.11 REDIS_PORT

Use this variable to connect to the Redis cache server on a port other than the default 6379.

```
$ docker run -d --name cloudserver -p 8000:8000 \
-e REDIS_PORT=6379 zenko/cloudserver
```

5.2 Tunables and Setup Tips

5.2.1 Using Docker Volumes

CloudServer runs with a file backend by default, meaning that data is stored inside the CloudServer's Docker container.

For data and metadata to persist, data and metadata must be hosted in Docker volumes outside the CloudServer's Docker container. Otherwise, the data and metadata are destroyed when the container is erased.

```
$ docker run -v $(pwd)/data:/usr/src/app/localData -v $(pwd)/metadata:/usr/src/app/
↪localMetadata \
-p 8000:8000 -d zenko/cloudserver
```

This command mounts the ./data host directory to the container at /usr/src/app/localData and the ./metadata host directory to the container at /usr/src/app/localMetaData.

Tip: These host directories can be mounted to any accessible mount point, such as /mnt/data and /mnt/metadata, for example.

5.2.2 Adding, Modifying, or Deleting Accounts or Credentials

1. Create a customized authdata.json file locally based on /conf/authdata.json.
2. Use Docker volumes to override the default `authdata.json` through a Docker file mapping.

For example:

```
$ docker run -v $(pwd)/authdata.json:/usr/src/app/conf/authdata.json -p 8000:8000 -d \
zenko/cloudserver
```

5.2.3 Specifying a Host Name

To specify a host name (for example, s3.domain.name), provide your own config.json file using Docker volumes.

First, add a new key-value pair to the restEndpoints section of your config.json. Make the key the host name you want, and the value the default location_constraint for this endpoint.

For example, `s3.example.com` is mapped to `us-east-1` which is one of the `location_constraints` listed in your locationConfig.json file here.

For more information about location configuration, see: GETTING STARTED

```
"restEndpoints": {
    "localhost": "file",
    "127.0.0.1": "file",
    ...
    "cloudserver.example.com": "us-east-1"
},
```

Next, run CloudServer using a Docker volume:

```
$ docker run -v $(pwd)/config.json:/usr/src/app/config.json -p 8000:8000 -d zenko/
↪cloudserver
```

The local `config.json` file overrides the default one through a Docker file mapping.

5.2.4 Running as an Unprivileged User

CloudServer runs as root by default.

To change this, modify the dockerfile and specify a user before the entry point.

The user must exist within the container, and must own the /usr/src/app directory for CloudServer to run.

For example, the following dockerfile lines can be modified:

```
...
&& groupadd -r -g 1001 scality \
&& useradd -u 1001 -g 1001 -d /usr/src/app -r scality \
&& chown -R scality:scality /usr/src/app

...

USER scality
ENTRYPOINT ["/usr/src/app/docker-entrypoint.sh"]
```

5.3 Continuous Integration with a Docker-Hosted CloudServer

When you start the Docker CloudServer image, you can adjust the configuration of the CloudServer instance by passing one or more environment variables on the `docker run` command line.

To run CloudServer for CI with custom locations (one in-memory, one hosted on AWS), and custom credentials mounted:

```
$ docker run --name CloudServer -p 8000:8000 \
-v $(pwd)/locationConfig.json:/usr/src/app/locationConfig.json \
-v $(pwd)/authdata.json:/usr/src/app/conf/authdata.json \
-v ~/.aws/credentials:/root/.aws/credentials \
-e S3DATA=multiple -e S3BACKEND=mem zenko/cloudserver
```

To run CloudServer for CI with custom locations, (one in-memory, one hosted on AWS, and one file), and custom credentials set as environment variables):

```
$ docker run --name CloudServer -p 8000:8000 \
-v $(pwd)/locationConfig.json:/usr/src/app/locationConfig.json \
-v ~/.aws/credentials:/root/.aws/credentials \
-v $(pwd)/data:/usr/src/app/localData -v $(pwd)/metadata:/usr/src/app/localMetadata \
-e SCALITY_ACCESS_KEY_ID=accessKey1 \
-e SCALITY_SECRET_ACCESS_KEY=verySecretKey1 \
-e S3DATA=multiple -e S3BACKEND=mem zenko/cloudserver
```

5.4 In Production with a Docker-Hosted CloudServer

Because data must persist in production settings, CloudServer offers multiple-backend capabilities. This requires a custom endpoint and custom credentials for local storage.

Customize these with:

```
$ docker run -d --name CloudServer \
-v $(pwd)/data:/usr/src/app/localData -v $(pwd)/metadata:/usr/src/app/localMetadata \
-v $(pwd)/locationConfig.json:/usr/src/app/locationConfig.json \
-v $(pwd)/authdata.json:/usr/src/app/conf/authdata.json \
-v ~/.aws/credentials:/root/.aws/credentials -e S3DATA=multiple \
-e ENDPOINT=custom.endpoint.com \
-p 8000:8000 -d zenko/cloudserver \
```

CHAPTER 6

Integrations

6.1 High Availability

Docker Swarm is a clustering tool developed by Docker for use with its containers. It can be used to start services, which we define to ensure CloudServer's continuous availability to end users. A swarm defines a manager and n workers among $n + 1$ servers.

This tutorial shows how to perform a basic setup with three servers, which provides strong service resiliency, while remaining easy to use and maintain. We will use NFS through Docker to share data and metadata between the different servers.

Sections are labeled **On Server**, **On Clients**, or **On All Machines**, referring respectively to NFS server, NFS clients, or NFS server and clients. In the present example, the server's IP address is **10.200.15.113** and the client IP addresses are **10.200.15.96** and **10.200.15.97**

1. Install Docker (on All Machines)

   Docker 17.03.0-ce is used for this tutorial. Docker 1.12.6 and later will likely work, but is not tested.

   - On Ubuntu 14.04 Install Docker CE for Ubuntu as documented at Docker. Install the aufs dependency as recommended by Docker. The required commands are:

     ```
     $> sudo apt-get update
     $> sudo apt-get install linux-image-extra-$(uname -r) linux-image-extra-
     ↪virtual
     $> sudo apt-get install apt-transport-https ca-certificates curl software-
     ↪properties-common
     $> curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add␣
     ↪-
     $> sudo add-apt-repository "deb [arch=amd64] https://download.docker.com/
     ↪linux/ubuntu $(lsb_release -cs) stable"
     $> sudo apt-get update
     $> sudo apt-get install docker-ce
     ```

   - On CentOS 7 Install Docker CE as documented at Docker. The required commands are:

```
$> sudo yum install -y yum-utils
$> sudo yum-config-manager --add-repo https://download.docker.com/linux/
↪centos/docker-ce.repo
$> sudo yum makecache fast
$> sudo yum install docker-ce
$> sudo systemctl start docker
```

2. Install NFS on Client(s)

   NFS clients mount Docker volumes over the NFS server's shared folders. If the NFS commons are installed, manual mounts are no longer needed.

   - On Ubuntu 14.04

     Install the NFS commons with apt-get:

```
$> sudo apt-get install nfs-common
```

   - On CentOS 7

     Install the NFS utils; then start required services:

```
$> yum install nfs-utils
$> sudo systemctl enable rpcbind
$> sudo systemctl enable nfs-server
$> sudo systemctl enable nfs-lock
$> sudo systemctl enable nfs-idmap
$> sudo systemctl start rpcbind
$> sudo systemctl start nfs-server
$> sudo systemctl start nfs-lock
$> sudo systemctl start nfs-idmap
```

3. Install NFS (on Server)

   The NFS server hosts the data and metadata. The package(s) to install on it differs from the package installed on the clients.

   - On Ubuntu 14.04

     Install the NFS server-specific package and the NFS commons:

```
$> sudo apt-get install nfs-kernel-server nfs-common
```

   - On CentOS 7

     Install the NFS utils and start the required services:

```
$> yum install nfs-utils
$> sudo systemctl enable rpcbind
$> sudo systemctl enable nfs-server
$> sudo systemctl enable nfs-lock
$> sudo systemctl enable nfs-idmap
$> sudo systemctl start rpcbind
$> sudo systemctl start nfs-server
$> sudo systemctl start nfs-lock
$> sudo systemctl start nfs-idmap
```

   For both distributions:

   1. Choose where shared data and metadata from the local CloudServer shall be stored (The present example uses /var/nfs/data and /var/nfs/metadata). Set permissions for these folders for sharing over NFS:

```
$> mkdir -p /var/nfs/data /var/nfs/metadata
$> chmod -R 777 /var/nfs/
```

2. The /etc/exports file configures network permissions and r-w-x permissions for NFS access. Edit /etc/exports, adding the following lines:

```
/var/nfs/data        10.200.15.96(rw,sync,no_root_squash) 10.200.15.97(rw,
↪sync,no_root_squash)
/var/nfs/metadata    10.200.15.96(rw,sync,no_root_squash) 10.200.15.97(rw,
↪sync,no_root_squash)
```

   Ubuntu applies the no_subtree_check option by default, so both folders are declared with the same permissions, even though they're in the same tree.

3. Export this new NFS table:

```
$> sudo exportfs -a
```

4. Edit the `MountFlags` option in the Docker config in /lib/systemd/system/docker.service to enable NFS mount from Docker volumes on other machines:

```
MountFlags=shared
```

5. Restart the NFS server and Docker daemons to apply these changes.

   - On Ubuntu 14.04

   ```
   $> sudo service nfs-kernel-server restart
   $> sudo service docker restart
   ```

   - On CentOS 7

   ```
   $> sudo systemctl restart nfs-server
   $> sudo systemctl daemon-reload
   $> sudo systemctl restart docker
   ```

4. Set Up a Docker Swarm

- On all machines and distributions:

  Set up the Docker volumes to be mounted to the NFS server for CloudServer's data and metadata storage. The following commands must be replicated on all machines:

```
$> docker volume create --driver local --opt type=nfs --opt o=addr=10.200.15.113,
↪rw --opt device=:/var/nfs/data --name data
$> docker volume create --driver local --opt type=nfs --opt o=addr=10.200.15.113,
↪rw --opt device=:/var/nfs/metadata --name metadata
```

  There is no need to `docker exec` these volumes to mount them: the Docker Swarm manager does this when the Docker service is started.

- On a server:

  To start a Docker service on a Docker Swarm cluster, initialize the cluster (that is, define a manager), prompt workers/nodes to join in, and then start the service.

  Initialize the swarm cluster, and review its response:

```
$> docker swarm init --advertise-addr 10.200.15.113

Swarm initialized: current node (db2aqfu3bzfzzs9b1kfeaglmq) is now a manager.

To add a worker to this swarm, run the following command:

docker swarm join \
--token SWMTKN-1-5yxxencrdoelr7mpltljn325uz4v6fe1gojl14lzceij3nujzu-
↪2vfs9u6ipgcq35r90xws3stka \
10.200.15.113:2377

To add a manager to this swarm, run 'docker swarm join-token manager' and follow␣
↪the instructions.
```

- On clients:

  Copy and paste the command provided by your Docker Swarm init. A successful request/response will resemble:

```
$> docker swarm join --token SWMTKN-1-
↪5yxxencrdoelr7mpltljn325uz4v6fe1gojl14lzceij3nujzu-2vfs9u6ipgcq35r90xws3stka 10.
↪200.15.113:2377

This node joined a swarm as a worker.
```

6.1.1 Set Up Docker Swarm on Clients on a Server

Start the service on the Swarm cluster.

```
$> docker service create --name s3 --replicas 1 --mount type=volume,source=data,
↪target=/usr/src/app/localData --mount type=volume,source=metadata,target=/usr/src/
↪app/localMetadata -p 8000:8000 scality/cloudserver
```

On a successful installation, `docker service ls` returns the following output:

```
$> docker service ls
ID            NAME  MODE        REPLICAS  IMAGE
ocmggza412ft  s3    replicated  1/1       scality/cloudserver:latest
```

If the service does not start, consider disabling apparmor/SELinux.

6.1.2 Testing the High-Availability CloudServer

On all machines (client/server) and distributions (Ubuntu and CentOS), determine where CloudServer is running using `docker ps`. CloudServer can operate on any node of the Swarm cluster, manager or worker. When you find it, you can kill it with `docker stop <container id>`. It will respawn on a different node. Now, if one server falls, or if Docker stops unexpectedly, the end user will still be able to access your the local CloudServer.

6.1.3 Troubleshooting

To troubleshoot the service, run:

```
$> docker service ps s3docker service ps s3
ID                          NAME      IMAGE             NODE
↪    DESIRED STATE  CURRENT STATE       ERROR
0ar81cw4lvv8chafm8pw48wbc  s3.1      scality/cloudserver  localhost.localdomain.
↪localdomain  Running        Running 7 days ago
cvmf3j3bz8w6r4h0lf3pxo6eu   \_ s3.1  scality/cloudserver  localhost.localdomain.
↪localdomain  Shutdown       Failed 7 days ago   "task: non-zero exit (137)"
```

If the error is truncated, view the error in detail by inspecting the Docker task ID:

```
$> docker inspect cvmf3j3bz8w6r4h0lf3pxo6eu
```

6.1.4 Off you go!

Let us know how you use this and if you'd like any specific developments around it. Even better: come and contribute to our Github repository! We look forward to meeting you!

6.2 S3FS

You can export buckets as a filesystem with s3fs on CloudServer.

s3fs is an open source tool, available both on Debian and RedHat distributions, that enables you to mount an S3 bucket on a filesystem-like backend. This tutorial uses an Ubuntu 14.04 host to deploy and use s3fs over CloudServer.

6.2.1 Deploying Zenko CloudServer with SSL

First, deploy CloudServer with a file backend using our DockerHub page.

Note: If Docker is not installed on your machine, follow these instructions to install it for your distribution.

You must also set up SSL with CloudServer to use s3fs. See Using SSL for instructions.

6.2.2 s3fs Setup

Installing s3fs

Follow the instructions in the s3fs README,

Check that s3fs is properly installed. A version check should return a response resembling:

```
$> s3fs --version

Amazon Simple Storage Service File System V1.80(commit:d40da2c) with OpenSSL
Copyright (C) 2010 Randy Rizun <rrizun@gmail.com>
License GPL2: GNU GPL version 2 <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
```

Configuring s3fs

s3fs expects you to provide it with a password file. Our file is `/etc/passwd-s3fs`. The structure for this file is `ACCESSKEYID:SECRETKEYID`, so, for CloudServer, you can run:

```
$> echo 'accessKey1:verySecretKey1' > /etc/passwd-s3fs
$> chmod 600 /etc/passwd-s3fs
```

Using CloudServer with s3fs

1. Use /mnt/tests3fs as a mount point.

   ```
   $> mkdir /mnt/tests3fs
   ```

2. Create a bucket on your local CloudServer. In the present example it is named "tests3fs".

   ```
   $> s3cmd mb s3://tests3fs
   ```

3. Mount the bucket to your mount point with s3fs:

   ```
   $> s3fs tests3fs /mnt/tests3fs -o passwd_file=/etc/passwd-s3fs -o url="https://s3.
   ↪scality.test:8000/" -o use_path_request_style
   ```

   The structure of this command is: `s3fs BUCKET_NAME PATH/TO/MOUNTPOINT -o OPTIONS`. Of these mandatory options:

   - `passwd_file` specifies the path to the password file.
   - `url` specifies the host name used by your SSL provider.
   - **`use_path_request_style` forces the path style (by default,** s3fs uses DNS-style subdomains).

Once the bucket is mounted, files added to the mount point or objects added to the bucket will appear in both locations.

Example

Create two files, and then a directory with a file in our mount point:

```
$> touch /mnt/tests3fs/file1 /mnt/tests3fs/file2
$> mkdir /mnt/tests3fs/dir1
$> touch /mnt/tests3fs/dir1/file3
```

Now, use s3cmd to show what is in CloudServer:

```
$> s3cmd ls -r s3://tests3fs

2017-02-28 17:28         0   s3://tests3fs/dir1/
2017-02-28 17:29         0   s3://tests3fs/dir1/file3
2017-02-28 17:28         0   s3://tests3fs/file1
2017-02-28 17:28         0   s3://tests3fs/file2
```

Now you can enjoy a filesystem view on your local CloudServer.

6.3 Duplicity

How to back up your files with CloudServer.

6.3.1 Installing Duplicity and its Dependencies

To install Duplicity, go to this site. Download the latest tarball. Decompress it and follow the instructions in the README.

```
$> tar zxvf duplicity-0.7.11.tar.gz
$> cd duplicity-0.7.11
$> python setup.py install
```

You may receive error messages indicating the need to install some or all of the following dependencies:

```
$> apt-get install librsync-dev gnupg
$> apt-get install python-dev python-pip python-lockfile
$> pip install -U boto
```

Testing the Installation

1. Check that CloudServer is running. Run `$> docker ps`. You should see one container named `scality/cloudserver`. If you do not, run `$> docker start cloudserver` and check again.
2. Duplicity uses a module called "Boto" to send requests to S3. Boto requires a configuration file located in `/etc/boto.cfg` to store your credentials and preferences. A minimal configuration you can fine tune following these instructions is shown here:

```
   [Credentials]
   aws_access_key_id = accessKey1
   aws_secret_access_key = verySecretKey1

   [Boto]
   # If using SSL, set to True
   is_secure = False
   # If using SSL, unmute and provide absolute path to local CA certificate
   # ca_certificates_file = /absolute/path/to/ca.crt

.. note:: To set up SSL with CloudServer, check out our `Using SSL
          <./GETTING_STARTED#Using_SSL>`__ in GETTING STARTED.
```

3. At this point all requirements to run CloudServer as a backend to Duplicity have been met. A local folder/file should back up to the local S3. Try it with the decompressed Duplicity folder:

```
$> duplicity duplicity-0.7.11 "s3://127.0.0.1:8000/testbucket/"
```

Note:

Duplicity will prompt for a symmetric encryption passphrase. Save it carefully, as you will need it to recover your data. Alternatively, you can add the `--no-encryption` flag and the data will be stored plain.

If this command is successful, you will receive an output resembling:

```
--------------[ Backup Statistics ]--------------
StartTime 1486486547.13 (Tue Feb  7 16:55:47 2017)
EndTime 1486486547.40 (Tue Feb  7 16:55:47 2017)
ElapsedTime 0.27 (0.27 seconds)
SourceFiles 388
SourceFileSize 6634529 (6.33 MB)
NewFiles 388
NewFileSize 6634529 (6.33 MB)
DeletedFiles 0
ChangedFiles 0
ChangedFileSize 0 (0 bytes)
ChangedDeltaSize 0 (0 bytes)
DeltaEntries 388
RawDeltaSize 6392865 (6.10 MB)
TotalDestinationSizeChange 2003677 (1.91 MB)
Errors 0
-------------------------------------------------
```

Congratulations! You can now back up to your local S3 through Duplicity.

Automating Backups

The easiest way to back up files periodically is to write a bash script and add it to your crontab. A suggested script follows.

```
#!/bin/bash

# Export your passphrase so you don't have to type anything
export PASSPHRASE="mypassphrase"

# To use a GPG key, put it here and uncomment the line below
#GPG_KEY=

# Define your backup bucket, with localhost specified
DEST="s3://127.0.0.1:8000/testbucketcloudserver/"

# Define the absolute path to the folder to back up
SOURCE=/root/testfolder

# Set to "full" for full backups, and "incremental" for incremental backups
# Warning: you must perform one full backup befor you can perform
# incremental ones on top of it
FULL=incremental

# How long to keep backups. If you don't want to delete old backups, keep
# this value empty; otherwise, the syntax is "1Y" for one year, "1M" for
# one month, "1D" for one day.
OLDER_THAN="1Y"

# is_running checks whether Duplicity is currently completing a task
is_running=$(ps -ef | grep duplicity  | grep python | wc -l)

# If Duplicity is already completing a task, this will not run
if [ $is_running -eq 0 ]; then
    echo "Backup for ${SOURCE} started"
```

(continues on next page)

(continued from previous page)

```
    # To delete backups older than a certain time, do it here
    if [ "$OLDER_THAN" != "" ]; then
        echo "Removing backups older than ${OLDER_THAN}"
        duplicity remove-older-than ${OLDER_THAN} ${DEST}
    fi

    # This is where the actual backup takes place
    echo "Backing up ${SOURCE}..."
    duplicity ${FULL} \
        ${SOURCE} ${DEST}
        # If you're using GPG, paste this in the command above
        # --encrypt-key=${GPG_KEY} --sign-key=${GPG_KEY} \
        # If you want to exclude a subfolder/file, put it below and
        # paste this
        # in the command above
        # --exclude=/${SOURCE}/path_to_exclude \

    echo "Backup for ${SOURCE} complete"
    echo "------------------------------------"
fi
# Forget the passphrase...
unset PASSPHRASE
```

Put this file in `/usr/local/sbin/backup.sh`. Run `crontab -e` and paste your configuration into the file that opens. If you're unfamiliar with Cron, here is a good HowTo. If the folder being backed up is a folder to be modified permanently during the work day, we can set incremental backups every 5 minutes from 8 AM to 9 PM Monday through Friday by pasting the following line into crontab:

```
*/5 8-20 * * 1-5 /usr/local/sbin/backup.sh
```

Adding or removing files from the folder being backed up will result in incremental backups in the bucket.

CHAPTER 7

Architecture

7.1 Versioning

This document describes Zenko CloudServer's support for the AWS S3 Bucket Versioning feature.

7.1.1 AWS S3 Bucket Versioning

See AWS documentation for a description of the Bucket Versioning feature:

- Bucket Versioning
- Object Versioning

This document assumes familiarity with the details of Bucket Versioning, including null versions and delete markers, described in the above links.

7.1.2 Implementation of Bucket Versioning in Zenko CloudServer

Overview of Metadata and API Component Roles

Each version of an object is stored as a separate key in metadata. The S3 API interacts with the metadata backend to store, retrieve, and delete version metadata.

The implementation of versioning within the metadata backend is naive. The metadata backend does not evaluate any information about bucket or version state (whether versioning is enabled or suspended, and whether a version is a null version or delete marker). The S3 front-end API manages the logic regarding versioning information, and sends instructions to metadata to handle the basic CRUD operations for version metadata.

The role of the S3 API can be broken down into the following:

- put and delete version data
- store extra information about a version, such as whether it is a delete marker or null version, in the object's metadata

- send instructions to metadata backend to store, retrieve, update and delete version metadata based on bucket versioning state and version metadata
- encode version ID information to return in responses to requests, and decode version IDs sent in requests

The implementation of Bucket Versioning in S3 is described in this document in two main parts. The first section, *"Implementation of Bucket Versioning in Metadata"*, describes the way versions are stored in metadata, and the metadata options for manipulating version metadata.

The second section, *"Implementation of Bucket Versioning in API"*, describes the way the metadata options are used in the API within S3 actions to create new versions, update their metadata, and delete them. The management of null versions and creation of delete markers is also described in this section.

7.1.3 Implementation of Bucket Versioning in Metadata

As mentioned above, each version of an object is stored as a separate key in metadata. We use version identifiers as the suffix for the keys of the object versions, and a special version (the *"Master Version"*) to represent the latest version.

An example of what the metadata keys might look like for an object `foo/bar` with three versions (with . representing a null character):

key
foo/bar
foo/bar.098506163554375999999PARIS 0.a430a1f85c6ec
foo/bar.098506163554373999999PARIS 0.41b510cd0fdf8
foo/bar.098506163554373999998PARIS 0.f9b82c166f695

The most recent version created is represented above in the key `foo/bar` and is the master version. This special version is described further in the section *"Master Version"*.

Version ID and Metadata Key Format

The version ID is generated by the metadata backend, and encoded in a hexadecimal string format by S3 before sending a response to a request. S3 also decodes the hexadecimal string received from a request before sending to metadata to retrieve a particular version.

The format of a `version_id` is: `ts rep_group_id seq_id` where:

- `ts`: is the combination of epoch and an increasing number
- `rep_group_id`: is the name of deployment(s) considered one unit used for replication
- `seq_id`: is a unique value based on metadata information.

The format of a key in metadata for a version is:

`object_name separator version_id` where:

- `object_name`: is the key of the object in metadata
- `separator`: we use the `null` character (`0x00` or `\0`) as the separator between the `object_name` and the `version_id` of a key
- `version_id`: is the version identifier; this encodes the ordering information in the format described above as metadata orders keys alphabetically

An example of a key in metadata: `foo\0123456789000077PARIS 1234.123456` indicating that this specific version of `foo` was the `000777`th entry created during the epoch `1234567890` in the replication group `PARIS` with `1234.123456` as `seq_id`.

Master Version

We store a copy of the latest version of an object's metadata using `object_name` as the key; this version is called the master version. The master version of each object facilitates the standard GET operation, which would otherwise need to scan among the list of versions of an object for its latest version.

The following table shows the layout of all versions of `foo` in the first example stored in the metadata (with dot `.` representing the null separator):

key	value
foo	B
foo.v2	B
foo.v1	A

Metadata Versioning Options

Zenko CloudServer sends instructions to the metadata engine about whether to create a new version or overwrite, retrieve, or delete a specific version by sending values for special options in PUT, GET, or DELETE calls to metadata. The metadata engine can also list versions in the database, which is used by Zenko CloudServer to list object versions.

These only describe the basic CRUD operations that the metadata engine can handle. How these options are used by the S3 API to generate and update versions is described more comprehensively in *"Implementation of Bucket Versioning in API"*.

Note: all operations (PUT and DELETE) that generate a new version of an object will return the `version_id` of the new version to the API.

PUT

- no options: original PUT operation, will update the master version
- `versioning: true` create a new version of the object, then update the master version with this version.
- `versionId: <versionId>` create or update a specific version (for updating version's ACL or tags, or remote updates in geo-replication)
  - if the version identified by `versionId` happens to be the latest version, the master version will be updated as well
  - if the master version is not as recent as the version identified by `versionId`, as may happen with cross-region replication, the master will be updated as well
  - note that with `versionId` set to an empty string `''`, it will overwrite the master version only (same as no options, but the master version will have a `versionId` property set in its metadata like any other version). The `versionId` will never be exposed to an external user, but setting this internal-only `versionID` enables Zenko CloudServer to find this version later if it is no longer the master. This option of `versionId` set to `''` is used for creating null versions once versioning has been suspended, which is discussed in *"Null Version Management"*.

In general, only one option is used at a time. When `versionId` and `versioning` are both set, only the `versionId` option will have an effect.

DELETE

- no options: original DELETE operation, will delete the master version

- `versionId: <versionId>` delete a specific version

A deletion targeting the latest version of an object has to:

- delete the specified version identified by `versionId`
- replace the master version with a version that is a placeholder for deletion
  - **this version contains a special keyword, 'isPHD', to indicate the** master version was deleted and needs to be updated
- initiate a repair operation to update the value of the master version:
  - involves listing the versions of the object and get the latest version to replace the placeholder delete version
  - if no more versions exist, metadata deletes the master version, removing the key from metadata

Note: all of this happens in metadata before responding to the front-end api, and only when the metadata engine is instructed by Zenko CloudServer to delete a specific version or the master version. See section *"Delete Markers"* for a description of what happens when a Delete Object request is sent to the S3 API.

GET

- no options: original GET operation, will get the master version
- `versionId: <versionId>` retrieve a specific version

The implementation of a GET operation does not change compared to the standard version. A standard GET without versioning information would get the master version of a key. A version-specific GET would retrieve the specific version identified by the key for that version.

LIST

For a standard LIST on a bucket, metadata iterates through the keys by using the separator (`\0`, represented by `.` in examples) as an extra delimiter. For a listing of all versions of a bucket, there is no change compared to the original listing function. Instead, the API component returns all the keys in a List Objects call and filters for just the keys of the master versions in a List Object Versions call.

For example, a standard LIST operation against the keys in a table below would return from metadata the list of `[ foo/bar, bar, qux/quz, quz ]`.

key
foo/bar
foo/bar.v2
foo/bar.v1
bar
qux/quz
qux/quz.v2
qux/quz.v1
quz
quz.v2
quz.v1

7.1.4 Implementation of Bucket Versioning in API

Object Metadata Versioning Attributes

To access all the information needed to properly handle all cases that may exist in versioned operations, the API stores certain versioning-related information in the metadata attributes of each version's object metadata.

These are the versioning-related metadata properties:

- `isNull`: whether the version being stored is a null version.
- `nullVersionId`: the unencoded version ID of the latest null version that existed before storing a non-null version.
- `isDeleteMarker`: whether the version being stored is a delete marker.

The metadata engine also sets one additional metadata property when creating the version.

- `versionId`: the unencoded version ID of the version being stored.

Null versions and delete markers are described in further detail in their own subsections.

Creation of New Versions

When versioning is enabled in a bucket, APIs which normally result in the creation of objects, such as Put Object, Complete Multipart Upload and Copy Object, will generate new versions of objects.

Zenko CloudServer creates a new version and updates the master version using the `versioning: true` option in PUT calls to the metadata engine. As an example, when two consecutive Put Object requests are sent to the Zenko CloudServer for a versioning-enabled bucket with the same key names, there are two corresponding metadata PUT calls with the `versioning` option set to true.

The PUT calls to metadata and resulting keys are shown below:

(1) PUT foo (first put), versioning: `true`

key	value
foo	A
foo.v1	A

(2) PUT foo (second put), versioning: `true`

key	value
foo	B
foo.v2	B
foo.v1	A

Null Version Management

In a bucket without versioning, or when versioning is suspended, putting an object with the same name twice should result in the previous object being overwritten. This is managed with null versions.

Only one null version should exist at any given time, and it is identified in Zenko CloudServer requests and responses with the version id "null".

Case 1: Putting Null Versions

With respect to metadata, since the null version is overwritten by subsequent null versions, the null version is initially stored in the master key alone, as opposed to being stored in the master key and a new version. Zenko CloudServer checks if versioning is suspended or has never been configured, and sets the `versionId` option to `''` in PUT calls to the metadata engine when creating a new null version.

If the master version is a null version, Zenko CloudServer also sends a DELETE call to metadata prior to the PUT, in order to clean up any pre-existing null versions which may, in certain edge cases, have been stored as a separate version.[1]

The tables below summarize the calls to metadata and the resulting keys if we put an object 'foo' twice, when versioning has not been enabled or is suspended.

(1) PUT foo (first put), versionId: `''`

key	value
foo (null)	A

(2A) DELETE foo (clean-up delete before second put), versionId: `<version id of master version>`

key	value

(2B) PUT foo (second put), versionId: `''`

key	value
foo (null)	B

The S3 API also sets the `isNull` attribute to `true` in the version metadata before storing the metadata for these null versions.

Case 2: Preserving Existing Null Versions in Versioning-Enabled Bucket

Null versions are preserved when new non-null versions are created after versioning has been enabled or re-enabled.

If the master version is the null version, the S3 API preserves the current null version by storing it as a new key `(3A)` in a separate PUT call to metadata, prior to overwriting the master version `(3B)`. This implies the null version may not necessarily be the latest or master version.

To determine whether the master version is a null version, the S3 API checks if the master version's `isNull` property is set to `true`, or if the `versionId` attribute of the master version is undefined (indicating it is a null version that was put before bucket versioning was configured).

Continuing the example from Case 1, if we enabled versioning and put another object, the calls to metadata and resulting keys would resemble the following:

(3A) PUT foo, versionId: `<versionId of master version>` if defined or `<non-versioned object id>`

[1] Some examples of these cases are: (1) when there is a null version that is the second-to-latest version, and the latest version has been deleted, causing metadata to repair the master value with the value of the null version and (2) when putting object tag or ACL on a null version that is the master version, as explained in *"Behavior of Object-Targeting APIs"*.

key	value
foo	B
foo.v1 (null)	B

(3B) PUT foo, versioning: `true`

key	value
foo	C
foo.v2	C
foo.v1 (null)	B

To prevent issues with concurrent requests, Zenko CloudServer ensures the null version is stored with the same version ID by using `versionId` option. Zenko CloudServer sets the `versionId` option to the master version's `versionId` metadata attribute value during the PUT. This creates a new version with the same version ID of the existing null master version.

The null version's `versionId` attribute may be undefined because it was generated before the bucket versioning was configured. In that case, a version ID is generated using the max epoch and sequence values possible so that the null version will be properly ordered as the last entry in a metadata listing. This value ("non-versioned object id") is used in the PUT call with the `versionId` option.

Case 3: Overwriting a Null Version That is Not Latest Version

Normally when versioning is suspended, Zenko CloudServer uses the `versionId: ''` option in a PUT to metadata to create a null version. This also overwrites an existing null version if it is the master version.

However, if there is a null version that is not the latest version, Zenko CloudServer cannot rely on the `versionId: ''` option will not overwrite the existing null version. Instead, before creating a new null version, the Zenko CloudServer API must send a separate DELETE call to metadata specifying the version id of the current null version for delete.

To do this, when storing a null version (3A above) before storing a new non-null version, Zenko CloudServer records the version's ID in the `nullVersionId` attribute of the non-null version. For steps 3A and 3B above, these are the values stored in the `nullVersionId` of each version's metadata:

(3A) PUT foo, versioning: `true`

key	value	value.nullVersionId
foo	B	undefined
foo.v1 (null)	B	undefined

(3B) PUT foo, versioning: `true`

key	value	value.nullVersionId
foo	C	v1
foo.v2	C	v1
foo.v1 (null)	B	undefined

If defined, the `nullVersionId` of the master version is used with the `versionId` option in a DELETE call to metadata if a Put Object request is received when versioning is suspended in a bucket.

(4A) DELETE foo, versionId: `<nullVersionId of master version>` (v1)

key	value
foo	C
foo.v2	C

Then the master version is overwritten with the new null version:

(4B) PUT foo, versionId: `''`

key	value
foo (null)	D
foo.v2	C

The `nullVersionId` attribute is also used to retrieve the correct version when the version ID "null" is specified in certain object-level APIs, described further in the section *"Null Version Mapping"*.

Specifying Versions in APIs for Putting Versions

Since Zenko CloudServer does not allow an overwrite of existing version data, Put Object, Complete Multipart Upload and Copy Object return `400 InvalidArgument` if a specific version ID is specified in the request query, e.g. for a `PUT /foo?versionId=v1` request.

PUT Example

When Zenko CloudServer receives a request to PUT an object:

- It checks first if versioning has been configured
- If it has not been configured, Zenko CloudServer proceeds to puts the new data, puts the metadata by overwriting the master version, and proceeds to delete any pre-existing data

If versioning has been configured, Zenko CloudServer checks the following:

Versioning Enabled

If versioning is enabled and there is existing object metadata:

- If the master version is a null version (`isNull: true`) or has no version ID (put before versioning was configured):
  - store the null version metadata as a new version
  - create a new version and overwrite the master version
    - set `nullVersionId`: version ID of the null version that was stored

If versioning is enabled and the master version is not null; or there is no existing object metadata:

- create a new version and store it, and overwrite the master version

Versioning Suspended

If versioning is suspended and there is existing object metadata:

- If the master version has no version ID:

  - overwrite the master version with the new metadata (PUT `versionId: ''`)
  - delete previous object data
- If the master version is a null version:
  - delete the null version using the *versionId* metadata attribute of the master version (PUT `versionId: <versionId of master object MD>`)
  - put a new null version (PUT `versionId: ''`)
- If master is not a null version and `nullVersionId` is defined in the object's metadata:
  - delete the current null version metadata and data
  - overwrite the master version with the new metadata

If there is no existing object metadata, create the new null version as the master version.

In each of the above cases, set `isNull` metadata attribute to true when creating the new null version.

Behavior of Object-Targeting APIs

API methods which can target existing objects or versions, such as Get Object, Head Object, Get Object ACL, Put Object ACL, Copy Object and Copy Part, will perform the action on the latest version of an object if no version ID is specified in the request query or relevant request header (`x-amz-copy-source-version-id` for Copy Object and Copy Part APIs).

Two exceptions are the Delete Object and Multi-Object Delete APIs, which will instead attempt to create delete markers, described in the following section, if no version ID is specified.

No versioning options are necessary to retrieve the latest version from metadata, since the master version is stored in a key with the name of the object. However, when updating the latest version, such as with the Put Object ACL API, Zenko CloudServer sets the `versionId` option in the PUT call to metadata to the value stored in the object metadata's `versionId` attribute. This is done in order to update the metadata both in the master version and the version itself, if it is not a null version.[2]

When a version id is specified in the request query for these APIs, e.g. `GET /foo?versionId=v1`, Zenko CloudServer will attempt to decode the version ID and perform the action on the appropriate version. To do so, the API sets the value of the `versionId` option to the decoded version ID in the metadata call.

Delete Markers

If versioning has not been configured for a bucket, the Delete Object and Multi-Object Delete APIs behave as their standard APIs.

If versioning has been configured, Zenko CloudServer deletes object or version data only if a specific version ID is provided in the request query, e.g. `DELETE /foo?versionId=v1`.

If no version ID is provided, S3 creates a delete marker by creating a 0-byte version with the metadata attribute `isDeleteMarker: true`. The S3 API will return a `404 NoSuchKey` error in response to requests getting or heading an object whose latest version is a delete maker.

To restore a previous version as the latest version of an object, the delete marker must be deleted, by the same process as deleting any other version.

[2] If it is a null version, this call will overwrite the null version if it is stored in its own key (`foo\0<versionId>`). If the null version is stored only in the master version, this call will both overwrite the master version *and* create a new key (`foo\0<versionId>`), resulting in the edge case referred to by the previous footnote[1].

The response varies when targeting an object whose latest version is a delete marker for other object-level APIs that can target existing objects and versions, without specifying the version ID.

- Get Object, Head Object, Get Object ACL, Object Copy and Copy Part return `404 NoSuchKey`.
- Put Object ACL and Put Object Tagging return `405 MethodNotAllowed`.

These APIs respond to requests specifying the version ID of a delete marker with the error `405 MethodNotAllowed`, in general. Copy Part and Copy Object respond with `400 Invalid Request`.

See section *"Delete Example"* for a summary.

Null Version Mapping

When the null version is specified in a request with the version ID "null", the S3 API must use the `nullVersionId` stored in the latest version to retrieve the current null version, if the null version is not the latest version.

Thus, getting the null version is a two step process:

1. Get the latest version of the object from metadata. If the latest version's `isNull` property is `true`, then use the latest version's metadata. Otherwise,
2. Get the null version of the object from metadata, using the internal version ID of the current null version stored in the latest version's `nullVersionId` metadata attribute.

DELETE Example

The following steps are used in the delete logic for delete marker creation:

- If versioning has not been configured: attempt to delete the object
- If request is version-specific delete request: attempt to delete the version
- otherwise, if not a version-specific delete request and versioning has been configured:
  - create a new 0-byte content-length version
  - in version's metadata, set a 'isDeleteMarker' property to true
- Return the version ID of any version deleted or any delete marker created
- Set response header `x-amz-delete-marker` to true if a delete marker was deleted or created

The Multi-Object Delete API follows the same logic for each of the objects or versions listed in an xml request. Note that a delete request can result in the creation of a deletion marker even if the object requested to delete does not exist in the first place.

Object-level APIs which can target existing objects and versions perform the following checks regarding delete markers:

- If not a version-specific request and versioning has been configured, check the metadata of the latest version
- If the 'isDeleteMarker' property is set to true, return `404 NoSuchKey` or `405 MethodNotAllowed`
- If it is a version-specific request, check the object metadata of the requested version
- If the `isDeleteMarker` property is set to true, return `405 MethodNotAllowed` or `400 InvalidRequest`

7.2 Data-metadata daemon Architecture and Operational guide

This document presents the architecture of the data-metadata daemon (dmd) used for the community edition of Zenko CloudServer. It also provides a guide on how to operate it.

The dmd is responsible for storing and retrieving Zenko CloudServer data and metadata, and is accessed by Zenko CloudServer connectors through socket.io (metadata) and REST (data) APIs.

It has been designed such that more than one Zenko CloudServer connector can access the same buckets by communicating with the dmd. It also means that the dmd can be hosted on a separate container or machine.

7.2.1 Operation

Startup

The simplest deployment is still to launch with yarn start, this will start one instance of the Zenko CloudServer connector and will listen on the locally bound dmd ports 9990 and 9991 (by default, see below).

The dmd can be started independently from the Zenko CloudServer by running this command in the Zenko CloudServer directory:

```
yarn run start_dmd
```

This will open two ports:

- one is based on socket.io and is used for metadata transfers (9990 by default)
- the other is a REST interface used for data transfers (9991 by default)

Then, one or more instances of Zenko CloudServer without the dmd can be started elsewhere with:

```
yarn run start_s3server
```

Configuration

Most configuration happens in `config.json` for Zenko CloudServer, local storage paths can be changed where the dmd is started using environment variables, like before: `S3DATAPATH` and `S3METADATAPATH`.

In `config.json`, the following sections are used to configure access to the dmd through separate configuration of the data and metadata access:

```
"metadataClient": {
    "host": "localhost",
    "port": 9990
},
"dataClient": {
    "host": "localhost",
    "port": 9991
},
```

To run a remote dmd, you have to do the following:

- change both `"host"` attributes to the IP or host name where the dmd is run.
- Modify the `"bindAddress"` attributes in `"metadataDaemon"` and `"dataDaemon"` sections where the dmd is run to accept remote connections (e.g. `"::"`)

7.2.2 Architecture

This section gives a bit more insight on how it works internally.

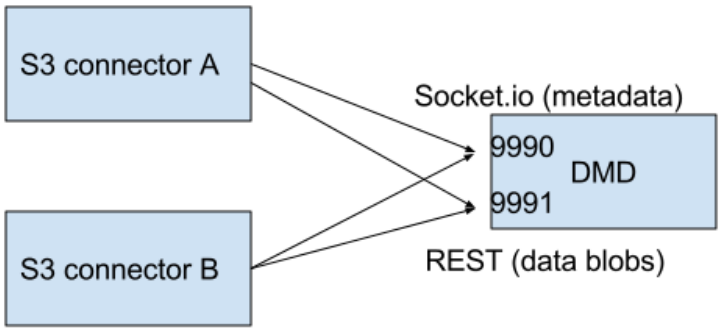


Fig. 1: ./images/data_metadata_daemon_arch.png

Metadata on socket.io

This communication is based on an RPC system based on socket.io events sent by Zenko CloudServerconnectors, received by the DMD and acknowledged back to the Zenko CloudServer connector.

The actual payload sent through socket.io is a JSON-serialized form of the RPC call name and parameters, along with some additional information like the request UIDs, and the sub-level information, sent as object attributes in the JSON request.

With introduction of versioning support, the updates are now gathered in the dmd for some number of milliseconds max, before being batched as a single write to the database. This is done server-side, so the API is meant to send individual updates.

Four RPC commands are available to clients: `put`, `get`, `del` and `createReadStream`. They more or less map the parameters accepted by the corresponding calls in the LevelUp implementation of LevelDB. They differ in the following:

- The `sync` option is ignored (under the hood, puts are gathered into batches which have their `sync` property enforced when they are committed to the storage)

- Some additional versioning-specific options are supported
- `createReadStream` becomes asynchronous, takes an additional callback argument and returns the stream in the second callback parameter

Debugging the socket.io exchanges can be achieved by running the daemon with `DEBUG='socket.io*'` environment variable set.

One parameter controls the timeout value after which RPC commands sent end with a timeout error, it can be changed either:

- via the `DEFAULT_CALL_TIMEOUT_MS` option in `lib/network/rpc/rpc.js`
- or in the constructor call of the `MetadataFileClient` object (in `lib/metadata/bucketfile/backend.js` as `callTimeoutMs`.

Default value is 30000.

A specific implementation deals with streams, currently used for listing a bucket. Streams emit `"stream-data"` events that pack one or more items in the listing, and a special `"stream-end"` event when done. Flow control is achieved by allowing a certain number of "in flight" packets that have not received an ack yet (5 by default). Two options can tune the behavior (for better throughput or getting it more robust on weak networks), they have to be set in `mdserver.js` file directly, as there is no support in `config.json` for now for those options:

- `streamMaxPendingAck`: max number of pending ack events not yet received (default is 5)
- `streamAckTimeoutMs`: timeout for receiving an ack after an output stream packet is sent to the client (default is 5000)

Data exchange through the REST data port

Data is read and written with REST semantic.

The web server recognizes a base path in the URL of `/DataFile` to be a request to the data storage service.

PUT

A PUT on `/DataFile` URL and contents passed in the request body will write a new object to the storage.

On success, a `201 Created` response is returned and the new URL to the object is returned via the `Location` header (e.g. `Location: /DataFile/50165db76eecea293abfd31103746dadb73a2074`). The raw key can then be extracted simply by removing the leading `/DataFile` service information from the returned URL.

GET

A GET is simply issued with REST semantic, e.g.:

```
GET /DataFile/50165db76eecea293abfd31103746dadb73a2074 HTTP/1.1
```

A GET request can ask for a specific range. Range support is complete except for multiple byte ranges.

DELETE

DELETE is similar to GET, except that a `204 No Content` response is returned on success.

7.3 Listing

7.3.1 Listing Types

We use three different types of metadata listing for various operations. Here are the scenarios we use each for:

- 'Delimiter' - when no versions are possible in the bucket since it is an internally-used only bucket which is not exposed to a user. Namely,

1. to list objects in the "user's bucket" to respond to a GET SERVICE request and
2. to do internal listings on an MPU shadow bucket to complete multipart upload operations.

- 'DelimiterVersion' - to list all versions in a bucket
- 'DelimiterMaster' - to list just the master versions of objects in a bucket

7.3.2 Algorithms

The algorithms for each listing type can be found in the open-source scality/Arsenal repository, in lib/algos/list.

7.4 Encryption

With CloudServer, there are two possible methods of at-rest encryption. (1) We offer bucket level encryption where Scality CloudServer itself handles at-rest encryption for any object that is in an 'encrypted' bucket, regardless of what the location-constraint for the data is and (2) If the location-constraint specified for the data is of type AWS, you can choose to use AWS server side encryption.

Note: bucket level encryption is not available on the standard AWS S3 protocol, so normal AWS S3 clients will not provide the option to send a header when creating a bucket. We have created a simple tool to enable you to easily create an encrypted bucket.

7.4.1 Example:

Creating encrypted bucket using our encrypted bucket tool in the bin directory

```
./create_encrypted_bucket.js -a accessKey1 -k verySecretKey1 -b bucketname -h␣
↪localhost -p 8000
```

7.4.2 AWS backend

With real AWS S3 as a location-constraint, you have to configure the location-constraint as follows

```
"awsbackend": {
    "type": "aws_s3",
    "legacyAwsBehavior": true,
    "details": {
        "serverSideEncryption": true,
        ...
    }
},
```

Then, every time an object is put to that data location, we pass the following header to AWS: `x-amz-server-side-encryption: AES256`

Note: due to these options, it is possible to configure encryption by both CloudServer and AWS S3 (if you put an object to a CloudServer bucket which has the encryption flag AND the location-constraint for the data is AWS S3 with serverSideEncryption set to true).

CHAPTER 8

Add New Backend Storage To Zenko CloudServer

This set of documents aims at bootstrapping developers with Zenko's CloudServer module, so they can then go on and contribute features.

8.1 Adding support for data backends not supporting the S3 API

These backends abstract the complexity of multiple APIs to let users work on a single common namespace across multiple clouds.

This documents aims at introducing you to the right files in CloudServer (the Zenko stack's subcomponent in charge of API translation, among other things) to add support to your own backend of choice.

8.1.1 General configuration

There are a number of constants and environment variables to define to support a new data backend; here is a list and where to find them:

`/constants.js`

- give your backend type a name, as part of the *externalBackends* object;
- specify whether versioning is implemented, as part of the *versioningNotImplemented* object;

`/lib/Config.js`

- this is where you should put common utility functions, like the ones to parse the location object from *locationConfig.json*;
- make sure you define environment variables (like *GCP_SERVICE_EMAIL* as we'll use those internally for the CI to test against the real remote backend;

/lib/data/external/{backendName}Client.js

- this file is where you'll instantiate your backend client; this should be a class with a constructor taking the config object built in */lib/Config.js* as parameter;
- over time, you may need some utility functions which we've defined in the folder */api/apiUtils*, and in the file */lib/data/external/utils*;

/lib/data/external/utils.js

- make sure to add options for *sourceLocationConstraintType* to be equal to the name you gave your backend in `/constants.js`;

/lib/data/external/{BackendName}_lib/

- this folder is where you'll put the functions needed for supporting your backend; keep your files as atomic as possible;

/tests/locationConfig/locationConfigTests.json

- this file is where you'll create location profiles to be used by your functional tests;

/lib/data/locationConstraintParser.js

- this is where you'll instantiate your client if the operation the end user sent effectively writes to your backend; everything happens inside the function *parseLC()*; you should add a condition that executes if *locationObj.type* is the name of your backend (that you defined in *constants.js*), and instantiates a client of yours. See pseudocode below, assuming location type name is *ztore*:

```
(...) //<1>
const ZtoreClient = require('./external/ZtoreClient');
const { config } = require('../Config'); //<1>

function parseLC(){ //<1>
(...) //<1>
    Object.keys(config.locationConstraints).forEach(location => { //<1>
        const locationObj = config.locationConstraints[location]; //<1>
        (...) //<1>
        if (locationObj.type === 'ztore' {
            const ztoreEndpoint = config.getZtoreEndpoint(location);
            const ztoreCredentials = config.getZtoreCredentials(location); //<2>
            clients[location] = new ZtoreClient({
                ztoreEndpoint,
                ztoreCredentials,
                ztoreBucketname: locationObj.details.ztoreBucketName,
                bucketMatch:  locationObj.details.BucketMatch,
                dataStoreName: location,
            }); //<3>
            clients[location].clientType = 'ztore';
        });
        (...) //<1>
    });
}
```

1. Code that is already there
2. You may need more utility functions depending on your backend specs
3. You may have more fields required in your constructor object depending on your backend specs

8.1.2 Operation of type PUT

PUT routes are usually where people get started, as it's the easiest to check! Simply go on your remote backend console and you'll be able to see whether your object actually went up in the cloud...

These are the files you'll need to edit:

`/lib/data/external/{BackendName}Client.js`

- the function that is going to call your *put()* function is also called *put()*, and it's defined in */lib/data/multipleBackendGateway.js*;
- define a function with signature like *put(stream, size, keyContext, reqUids, callback)*; this is worth exploring a bit more as these parameters are the same for all backends: //TODO: generate this from jsdoc
- *stream*: the stream of data you want to put in the cloud; if you're unfamiliar with node.js streams, we suggest you start training, as we use them a lot !
- *size*: the size of the object you're trying to put;
- *keyContext*: an object with metadata about the operation; common entries are *namespace*, *buckerName*, *owner*, *cipherBundle*, and *tagging*; if these are not sufficient for your integration, contact us to get architecture validation before adding new entries;
- *reqUids*: the request unique ID used for logging;
- *callback*: your function's callback (should handle errors);

`/lib/data/external/{backendName}_lib/`

- this is where you should put all utility functions for your PUT operation, and then import then in `/lib/data/external/{BackendName}Client.js`, to keep your code clean;

`tests/functional/aws-node-sdk/test/multipleBackend/put/put{BackendName}js`

- every contribution should come with thorough functional tests, showing nominal context gives expected behaviour, and error cases are handled in a way that is standard with the backend (including error messages and code);
- the ideal setup is if you simulate your backend locally, so as not to be subjected to network flakiness in the CI; however, we know there might not be mockups available for every client; if that is the case of your backend, you may test against the "real" endpoint of your data backend;

`tests/functional/aws-node-sdk/test/multipleBackend/utils.js`

- where you'll define a constant for your backend location matching your `/tests/locationConfig/locationConfigTests.json`

- depending on your backend, the sample *keys[]* and associated made up objects may not work for you (if your backend's key format is different, for example); if that is the case, you should add a custom *utils.get{{BackendName}}keys()* function returning ajusted *keys[]* to your tests.

8.1.3 Operation of type GET

GET routes are easy to test after PUT routes are implemented, hence why we're covering them second.

These are the files you'll need to edit:

`/lib/data/external/{BackendName}Client.js`

- the function that is going to call your *get()* function is also called *get()*, and it's defined in */lib/data/multipleBackendGateway.js*;
- define a function with signature like *get(objectGetInfo, range, reqUids, callback)*; this is worth exploring a bit more as these parameters are the same for all backends:

//TODO: generate this from jsdoc

- *objectGetInfo*: a dictionary with two entries: *key*, the object key in the data store, and *client*, the data store name;
- *range*: the range of bytes you will get, for "get-by-range" operations (we recommend you do simple GETs first, and then look at this);
- *reqUids*: the request unique ID used for logging;
- *callback*: your function's callback (should handle errors);

`/lib/data/external/{backendName}_lib/`

- this is where you should put all utility functions for your GET operation, and then import then in */lib/data/external/{{BackendName}}Client.js*, to keep your code clean;

`tests/functional/aws-node-sdk/test/multipleBackend/get/get{BackendName}js`

- every contribution should come with thorough functional tests, showing nominal context gives expected behaviour, and error cases are handled in a way that is standard with the backend (including error messages and code);
- the ideal setup is if you simulate your backend locally, so as not to be subjected to network flakiness in the CI; however, we know there might not be mockups available for every client; if that is the case of your backend, you may test against the "real" endpoint of your data backend;

`tests/functional/aws-node-sdk/test/multipleBackend/utils.js`

Note: You should need this section if you have followed the tutorial in order (that is, if you have covered the PUT operation already)

- where you'll define a constant for your backend location matching your `/tests/locationConfig/locationConfigTests.json`

- depending on your backend, the sample *keys[]* and associated made up objects may not work for you (if your backend's key format is different, for example); if that is the case, you should add a custom *utils.get{{BackendName}}keys()*

8.1.4 Operation of type DELETE

DELETE routes are easy to test after PUT routes are implemented, and they are similar to GET routes in our implementation, hence why we're covering them third.

These are the files you'll need to edit:

`/lib/data/external/{BackendName}Client.js`

- the function that is going to call your *delete()* function is also called *delete()*, and it's defined in `/lib/data/multipleBackendGateway.js`;
- define a function with signature like *delete(objectGetInfo, reqUids, callback)*; this is worth exploring a bit more as these parameters are the same for all backends:

//TODO: generate this from jsdoc

  - *objectGetInfo*: a dictionary with two entries: *key*, the object key in the data store, and *client*, the data store name;
  - *reqUids*: the request unique ID used for logging;
  - *callback*: your function's callback (should handle errors);

`/lib/data/external/{backendName}_lib/`

- this is where you should put all utility functions for your DELETE operation, and then import then in */lib/data/external/{{BackendName}}Client.js*, to keep your code clean;

`tests/functional/aws-node-sdk/test/multipleBackend/delete/delete{BackendName}js`

- every contribution should come with thorough functional tests, showing nominal context gives expected behaviour, and error cases are handled in a way that is standard with the backend (including error messages and code);
- the ideal setup is if you simulate your backend locally, so as not to be subjected to network flakiness in the CI; however, we know there might not be mockups available for every client; if that is the case of your backend, you may test against the "real" endpoint of your data backend;

`tests/functional/aws-node-sdk/test/multipleBackend/utils.js`

Note: You should need this section if you have followed the tutorial in order (that is, if you have covered the PUT operation already)

- where you'll define a constant for your backend location matching your `/tests/locationConfig/locationConfigTests.json`

- depending on your backend, the sample *keys[]* and associated made up objects may not work for you (if your backend's key format is different, for example); if that is the case, you should add a custom *utils.get{{BackendName}}keys()*

8.1.5 Operation of type HEAD

HEAD routes are very similar to DELETE routes in our implementation, hence why we're covering them fourth.

These are the files you'll need to edit:

`/lib/data/external/{BackendName}Client.js`

- the function that is going to call your *head()* function is also called *head()*, and it's defined in `/lib/data/multipleBackendGateway.js`;
- define a function with signature like *head(objectGetInfo, reqUids, callback)*; this is worth exploring a bit more as these parameters are the same for all backends:

// TODO:: generate this from jsdoc

- *objectGetInfo*: a dictionary with two entries: *key*, the object key in the data store, and *client*, the data store name;
- *reqUids*: the request unique ID used for logging;
- *callback*: your function's callback (should handle errors);

`/lib/data/external/{backendName}_lib/`

- this is where you should put all utility functions for your HEAD operation, and then import then in `/lib/data/external/{BackendName}Client.js`, to keep your code clean;

`tests/functional/aws-node-sdk/test/multipleBackend/get/get{BackendName}js`

- every contribution should come with thorough functional tests, showing nominal context gives expected behaviour, and error cases are handled in a way that is standard with the backend (including error messages and code);
- the ideal setup is if you simulate your backend locally, so as not to be subjected to network flakiness in the CI; however, we know there might not be mockups available for every client; if that is the case of your backend, you may test against the "real" endpoint of your data backend;

`tests/functional/aws-node-sdk/test/multipleBackend/utils.js`

Note: You should need this section if you have followed the tutorial in order (that is, if you have covered the PUT operation already)

- where you'll define a constant for your backend location matching your `/tests/locationConfig/locationConfigTests.json`
- depending on your backend, the sample *keys[]* and associated made up objects may not work for you (if your backend's key format is different, for example); if that is the case, you should add a custom *utils.get{{BackendName}}keys()*

8.1.6 Healthcheck

Healtchecks are used to make sure failure to write to a remote cloud is due to a problem on that remote cloud, an not on Zenko's side. This is usually done by trying to create a bucket that already exists, and making sure you get the expected answer.

These are the files you'll need to edit:

`/lib/data/external/{BackendName}Client.js`

- the function that is going to call your *healthcheck()* function is called *checkExternalBackend()* and it's defined in `/lib/data/multipleBackendGateway.js`; you will need to add your own;
- your healtcheck function should get *location* as a parameter, which is an object comprising:'
- *reqUids*: the request unique ID used for logging;
- *callback*: your function's callback (should handle errors);

`/lib/data/external/{backendName}_lib/{backendName}_create_bucket.js`

- this is where you should write the function performing the actual bucket creation;

`/lib/data/external/{backendName}_lib/utils.js`

- add an object named per your backend's name to the *backendHealth* dictionary, with proper *response* and *time* entries;

`lib/data/multipleBackendGateway.js`

- edit the *healthcheck* function to add your location's array, and call your healthcheck; see pseudocode below for a sample implementation, provided your backend name is *ztore*

```
(...) //<1>

    healthcheck: (flightCheckOnStartUp, log, callback) => { //<1>
        (...) //<1>
        const ztoreArray = []; //<2>
        async.each(Object.keys(clients), (location, cb) => { //<1>
            (...) //<1>
            } else if (client.clientType === 'ztore' {
                ztoreArray.push(location); //<3>
                return cb();
            }
        (...) //<1>
        multBackendResp[location] = { code: 200, message: 'OK' }; //<1>
        return cb();
    }, () => { //<1>
        async.parallel([
            (...) //<1>
            next => checkExternalBackend( //<4>
                clients, ztoreArray, 'ztore', flightCheckOnStartUp,
                externalBackendHealthCheckInterval, next),
        ] (...) //<1>
```

(continues on next page)

(continued from previous page)

```
            });
            (...) //<1>
        });
}
```

1. Code that is already there
2. The array that will store all locations of type 'ztore'
3. Where you add locations of type 'ztore' to the array
4. Where you actually call the healthcheck function on all 'ztore' locations

8.1.7 Multipart upload (MPU)

This is the final part to supporting a new backend! MPU is far from the easiest subject, but you've come so far it shouldn't be a problem.

These are the files you'll need to edit:

`/lib/data/external/{BackendName}Client.js`

You'll be creating four functions with template signatures:

- *createMPU(Key, metaHeaders, bucketName, websiteRedirectHeader, contentType, cacheControl, contentDisposition, contentEncoding, log, callback)* will initiate the multi part upload process; now, here, all parameters are metadata headers except for:
- *Key*, the key id for the final object (collection of all parts);
- *bucketName*, the name of the bucket to which we will do an MPU;
- *log*, the logger;
- ***uploadPart(request, streamingV4Params, stream, size, key, uploadId, partNumber, bucketName, log, callback)*** will be called for each part; the parameters can be explicited as follow:
- *request*, the request object for putting the part;
- *streamingV4Params*, parameters for auth V4 parameters against S3;
- *stream*, the node.js readable stream used to put the part;
- *size*, the size of the part;
- *key*, the key of the object;
- *uploadId*, multipart upload id string;
- *partNumber*, the number of the part in this MPU (ordered);
- *bucketName*, the name of the bucket to which we will do an MPU;
- *log*, the logger;
- ***completeMPU(jsonList, mdInfo, key, uploadId, bucketName, log, callback)*** **will** end the MPU process once all parts are uploaded; parameters can be explicited as follows:
- *jsonList*, user-sent list of parts to include in final mpu object;
- *mdInfo*, object containing 3 keys: storedParts, mpuOverviewKey, and splitter;

- *key*, the key of the object;
- *uploadId*, multipart upload id string;
- *bucketName*, name of bucket;
- *log*, logger instance:
- *abortMPU(key, uploadId, bucketName, log, callback)* will handle errors, and make sure that all parts that may have been uploaded will be deleted if the MPU ultimately fails; the parameters are:
- *key*, the key of the object;
- *uploadId*, multipart upload id string;
- *bucketName*, name of bucket;
- *log*, logger instance.

/lib/api/objectPutPart.js

- you'll need to add your backend type in appropriate sections (simply look for other backends already implemented).

/lib/data/external/{backendName}_lib/

- this is where you should put all utility functions for your MPU operations, and then import then in `/lib/data/external/{BackendName}Client.js`, to keep your code clean;

lib/data/multipleBackendGateway.js

- edit the *createMPU* function to add your location type, and call your *createMPU()*; see pseudocode below for a sample implementation, provided your backend name is *ztore*

```
(...) //<1>
    createMPU:(key, metaHeaders, bucketName, websiteRedirectHeader, //<1>
     location, contentType, cacheControl, contentDisposition,
     contentEncoding, log, cb) => {
        const client = clients[location]; //<1>
        if (client.clientType === 'aws_s3') { //<1>
            return client.createMPU(key, metaHeaders, bucketName,
            websiteRedirectHeader, contentType, cacheControl,
            contentDisposition, contentEncoding, log, cb);
        } else if (client.clientType === 'ztore') { //<2>
            return client.createMPU(key, metaHeaders, bucketName,
              websiteRedirectHeader, contentType, cacheControl,
              contentDisposition, contentEncoding, log, cb);
        }
        return cb();
    };
(...) //<1>
```

1. Code that is already there
2. Where the *createMPU()* of your client is actually called

8.1.8 Add functional tests

- `tests/functional/aws-node-sdk/test/multipleBackend/initMPU/{BackendName}InitMPU.js`
- `tests/functional/aws-node-sdk/test/multipleBackend/listParts/{BackendName}ListPart.js`
- `tests/functional/aws-node-sdk/test/multipleBackend/mpuAbort/{BackendName}AbortMPU.js`
- `tests/functional/aws-node-sdk/test/multipleBackend/mpuComplete/{BackendName}CompleteMPU.js`
- `tests/functional/aws-node-sdk/test/multipleBackend/mpuParts/{BackendName}UploadPart.js`

8.1.9 Adding support in Orbit, Zenko's UI for simplified Multi Cloud Management

This can only be done by our core developers' team. Once your backend integration is merged, you may open a feature request on the Zenko repository, and we will get back to you after we evaluate feasability and maintainability.

8.2 S3-Compatible Backends

8.2.1 Adding Support in CloudServer

This is the easiest case for backend support integration: there is nothing to do but configuration! Follow the steps described in our *Using Public Clouds as data backends* and make sure you:

- set `details.awsEndpoint` to your storage provider endpoint;
- use `details.credentials` and *not* `details.credentialsProfile` to set your credentials for that S3-compatible backend.

For example, if you're using a Wasabi bucket as a backend, then your region definition for that backend will look something like:

```
"wasabi-bucket-zenkobucket": {
"type": "aws_s3",
"legacyAwsBehavior": true,
"details": {
"awsEndpoint": "s3.wasabisys.com",
"bucketName": "zenkobucket",
"bucketMatch": true,
"credentials": {
"accessKey": "\\{YOUR_WASABI_ACCESS_KEY}",
"secretKey": "\\{YOUR_WASABI_SECRET_KEY}"
}
}
},
```

Adding Support in Zenko Orbit

This can only be done by our core developpers' team. If that's what you're after, open a feature request on the Zenko repository, and we will get back to you after we evaluate feasability and maintainability.

We always encourage our community to offer new extensions to Zenko, and new backend support is paramount to meeting more community needs. If that is something you want to contribute (or just do on your own version of the cloudserver image), this is the guid to read. Please make sure you follow our Contributing Guidelines/.

If you need help with anything, please search our forum for more information.

8.3 Add support for a new backend

Currently the main public cloud protocols are supported and more can be added. There are two main types of backend: those compatible with Amazon's S3 protocol and those not compatible.

Backend type	Supported	Active WIP	Not started
Private disk/fs	x		
AWS S3	x		
Microsoft Azure	x		
Backblaze B2		x	
Google Cloud	x		
Openstack Swift			x

Important: Should you want to request for a new backend to be supported, please do so by opening a Github issue, and filling out the "Feature Request" section of our template.

To add support for a new backend support to CloudServer official repository, please follow these steps:

- familiarize yourself with our Contributing Guidelines
- open a Github issue and fill out Feature Request form, and specify you would like to contribute it yourself;
- wait for our core team to get back to you with an answer on whether we are interested in taking that contribution in (and hence committing to maintaining it over time);
- once approved, fork the repository and start your development;
- use the forum with any question you may have during the development process;
- when you think it's ready, let us know so that we create a feature branch against which we'll compare and review your code;
- open a pull request with your changes against that dedicated feature branch;
- once that pull request gets merged, you're done.

Tip: While we do take care of the final rebase (when we merge your feature branch on the latest default branch), we do ask that you keep up to date with our latest default branch until then.

Important: If we do not approve your feature request, you may of course still work on supporting a new backend: all our "no" means is that we do not have the resources, as part of our core development team, to maintain this feature for the moment.

CHAPTER 9

Add A New Backend

Supporting all possible public cloud storage APIs is CloudServer's ultimate goal. As an open source project, contributions are welcome.

The first step is to get familiar with building a custom Docker image for CloudServer.

9.1 Build a Custom Docker Image

Clone Zenko's CloudServer, install all dependencies and start the service:

```
$ git clone https://github.com/scality/cloudserver
$ cd cloudserver
$ yarn install
$ yarn start
```

Tip: Some optional dependencies may fail, resulting in you seeing *yarn WARN* messages; these can safely be ignored. Refer to the User documentation for all available options.

Build the Docker image:

```
# docker build . -t
# {{YOUR_DOCKERHUB_ACCOUNT}}/cloudserver:{{OPTIONAL_VERSION_TAG}}
```

Push the newly created Docker image to your own hub:

```
# docker push
# {{YOUR_DOCKERHUB_ACCOUNT}}/cloudserver:{{OPTIONAL_VERSION_TAG}}
```

Note: To perform this last operation, you need to be authenticated with DockerHub

There are two main types of backend you could want Zenko to support:

== link:S3_COMPATIBLE_BACKENDS.adoc[S3 compatible data backends]

== link:NON_S3_COMPATIBLE_BACKENDS.adoc[Data backends using another protocol than the S3 protocol]