scality-utapi

Release 7.0.0

Aug 28, 2017

Documentation

Service Utilization API for tracking resource usage and metrics reporting

CHAPTER 1

Design

Please refer to the design for more information.

CHAPTER 2

Client

The module exposes a client, named UtapiClient. Projects can use this client to push metrics directly to the underlying datastore (Redis) without the need of an extra HTTP request to Utapi.

```
const { UtapiClient } = require('utapi');

const config = {
    redis: {
        host: '127.0.0.1',
        port: 6379
    },
    localCache: {
        host: '127.0.0.1',
        port: 6379
    }
}
const c = new UtapiClient(config);

// The second argument to `pushMetric` is a hexadecimal string Request Unique
// Identifier used for logging.
c.pushMetric('createBucket', '3d534b1511e5630e68f0', { bucket: 'demo' });

c.pushMetric('putObject', '3d534b1511e5630e68f0', {
    bucket: 'demo',
    newByteLength: 1024,
    oldByteLength: null,
});

c.pushMetric('putObject', '3d534b1511e5630e68f0', {
    bucket: 'demo',
    newByteLength: 1024,
    oldByteLength: 256,
});

c.pushMetric('multiObjectDelete', '3d534b1511e5630e68f0', {
    bucket: 'demo',
```

```
    byteLength: 1024,
    numberOfObjects: 999,
});
```

If an error occurs during a `pushMetric` call and the client is unable to record metrics in the underlying datastore, metric data is instead stored in a local Redis cache. Utapi attempts to push these cached metrics (every five minutes, by default) using a component named UtapiReplay. If the `pushMetric` call initiated by UtapiReplay fails, the metric is reinserted into the local Redis cache. The particularities of this behavior are configurable. For further information, see design.

CHAPTER 3

Listing Metrics with Utapi

To make a successful request to Utapi you would need

1. *IAM user with a policy giving access to Utapi*
2. *Sign request with Auth V4*

IAM user with a policy giving access to Utapi

Note: The examples here use AWS CLI but any AWS SDK is capable of these actions.

endpoint-url: This would be `https://<host>:<port>` where your Identity(IAM) Server is running.

1. Create an IAM user

```
aws iam --endpoint-url <endpoint> create-user --user-name utapiuser
```

2. Create access key for the user

```
aws iam --endpoint-url <endpoint> create-access-key --user-name utapiuser
```

3. Define a managed IAM policy

   sample utapi policy

```
cat - > utapipolicy.json <<EOF
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "utapiMetrics",
            "Action": [ "utapi:ListMetrics" ],
            "Effect": "Allow",
            "Resource": [
                "arn:scality:utapi::012345678901:buckets/*",
```

```
                "arn:scality:utapi::012345678901:accounts/*",
                "arn:scality:utapi::012345678901:users/*",
            ]
        }
    ]
}
EOF
```

In the above sample, the `Resource` property includes a series of Amazon Resource Names (ARNs) used to define which resources the policy applies to. Thus the sample policy applies to a user with an account ID '012345678901', and grants access to metrics at the levels 'buckets', 'accounts', and 'users'.

The account ID of the ARN can also be omitted, allowing any account to access metrics for those resources. As an example, we can extend the above sample policy to allow any account to access metrics at the level 'service':

```
...
"Resource": [
    "arn:scality:utapi::012345678901:buckets/*",
    "arn:scality:utapi::012345678901:accounts/*",
    "arn:scality:utapi::012345678901:users/*",
    "arn:scality:utapi:::service/*",
]
...
```

The omission of a metric level denies a user access to all resources at that level. For example, we can allow access to metrics only at the level 'buckets':

```
...
"Resource": ["arn:scality:utapi::012345678901:buckets/*"]
...
```

Further, access may be limited to specific resources within a metric level. For example, we can allow access to metrics only for a bucket 'foo':

```
...
"Resource": ["arn:scality:utapi::012345678901:buckets/foo"]
...
```

Or allow access to metrics for the bucket 'foo' for any user:

```
...
"Resource": ["arn:scality:utapi:::buckets/foo"]
...
```

4. Create a managed IAM policy

   Once your IAM policy is defined, create the policy using the following command.

```
aws iam --endpoint-url <endpoint> create-policy --policy-name utapipolicy \
 --policy-document file://utapipolicy.json
```

   A sample output of the above command would look like

```
{
    "Policy": {
        "PolicyName": "utapipolicy",
        "CreateDate": "2017-06-01T19:31:18.620Z",
        "AttachmentCount": 0,
```

```
        "IsAttachable": true,
        "PolicyId": "ZXR6A36LTYANPAI7NJ5UV",
        "DefaultVersionId": "v1",
        "Path": "/",
        "Arn": "arn:aws:iam::0123456789012:policy/utapipolicy",
        "UpdateDate": "2017-06-01T19:31:18.620Z"
    }
}
```

The arn property of the response, which we call `<policy arn>`, will be used in the next step to attach the policy to the user.

5. Attach user to the managed policy

```
aws --endpoint-url <endpoint> iam  attach-user-policy --user-name utapiuser
--policy-arn <policy arn>
```

Now the user `utapiuser` has access to ListMetrics request in Utapi on all buckets.

Signing request with Auth V4

There are two options here.

You can generate V4 signature using AWS SDKs or the node module aws4. See the following urls for reference.

- http://docs.aws.amazon.com/general/latest/gr/sigv4_signing.html
- http://docs.aws.amazon.com/general/latest/gr/sigv4-signed-request-examples.html
- https://github.com/mhart/aws4

You may also view examples making a request with Auth V4 using various languages and AWS SDKs here.

Alternatively, you can use a nifty command line tool available in Scality's S3.

You can git clone S3 repo from here https://github.com/scality/S3.git and follow the instructions in README to install the dependencies.

If you have S3 running inside a docker container you can docker exec into the S3 container as

```
docker exec -it <container id> bash
```

and then run the command

```
node bin/list_metrics
```

It will generate the following output listing available options.

```
Usage: list_metrics [options]

  Options:

    -h, --help                    output usage information
    -V, --version                 output the version number
    -a, --access-key <accessKey>  Access key id
    -k, --secret-key <secretKey>  Secret access key
    -m, --metric <metric>         Metric type
    --buckets <buckets>           Name of bucket(s) with a comma separator if
                                  more than one
```

```
    --accounts <accounts>          Account ID(s) with a comma separator if more
                                   than one
    --users <users>                User ID(s) with a comma separator if more than
                                   one
    --service <service>            Name of service
    -s, --start <start>            Start of time range
    -r, --recent                   List metrics including the previous and
                                   current 15 minute interval
    -e --end <end>                 End of time range
    -h, --host <host>              Host of the server
    -p, --port <port>              Port of the server
    --ssl                          Enable ssl
    -v, --verbose
```

A typical call to list metrics for a bucket `demo` to Utapi in a https enabled deployment would be

```
node bin/list_metrics --metric buckets --buckets demo --start 1476231300000
--end 1476233099999 -a myAccessKey -k mySecretKey -h 127.0.0.1 -p 8100 --ssl
```

Both start and end times are time expressed as UNIX epoch timestamps **expressed in milliseconds**.

Keep in mind, since Utapi metrics are normalized to the nearest 15 min. interval, so start time and end time need to be in specific format as follows.

Start time

Start time needs to be normalized to the nearest 15 minute interval with seconds and milliseconds set to 0. So valid start timestamps would look something like `09:00:00:000`, `09:15:00:000`, `09:30:00:000` and `09:45:00:000`.

For example

Date: Tue Oct 11 2016 17:35:25 GMT-0700 (PDT)

Unix timestamp (milliseconds): 1476232525320

Here's a typical JS method to get start timestamp

```
function getStartTimestamp(t) {
    const time = new Date(t);
    const minutes = time.getMinutes();
    const timestamp = time.setMinutes((minutes - minutes % 15), 0, 0);
    return timestamp;
}
```

This would format the start time timestamp to `1476231300000`

End time

End time needs to be normalized to the nearest 15 minute end interval with seconds and milliseconds set to 59 and 999 respectively. So valid end timestamps would look something like `09:14:59:999`, `09:29:59:999`, `09:44:59:999` and `09:59:59:999`.

Here's a typical JS method to get end timestamp

```
function getEndTimestamp(t) {
    const time = new Date(t);
    const minutes = time.getMinutes();
    const timestamp = time.setMinutes((minutes - minutes % 15) + 15, 0, -1);
    return timestamp;
}
```

This would format the end time timestamp to `1476233099999`

CHAPTER 4

Guidelines

Please read our coding and workflow guidelines at scality/Guidelines.

Contributing

In order to contribute, please follow the Contributing Guidelines.

CHAPTER 5

UTAPI CloudServer Integration

Instructions for setup

1. Required packages:

- git

```
apt-get install git
```

- redis server (stable version)

```
mkdir ~/cloudServer_scripts && \
cd ~/cloudServer_scripts && \
wget http://download.redis.io/releases/redis-4.0.1.tar.gz && \
tar xzf redis-4.0.1.tar.gz && \
cd redis-4.0.1 && \
make
```

{{ADD COMMANDS FOR EACH INSTALL}}

2. Setting up Zenko Cloud Server (formerly S3 server):

- Clone the cloud server:

```
cd ~/ && \
git clone https://github.com/scality/S3.git && \
cd S3 && \
git checkout 42-hackathon-utapi &&\
//npm i
```

- Edit config.json file of Cloud Server and add the following lines:

```
"localCache": {
    "host": "{endpoint-url}",
    "port": 6379
},
"utapi": {
    "workers": 1,
    "redis": {
        "host": "{endpoint-url}",
        "port": 6379
    }
}
```

Note: {endpoint-url}: This would be https:// where your cloud server is running(usually 127.0.0.1). The host should be the same as the cloudserver host.

- Make an image from the Dockerfile in the S3 folder:

```
docker build -t cloudserver . && \
docker images
```

Upon successful built of the image, the output of docker images should have the cloudserver image along with your other images:

```
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
cloudserver         latest              ecf4fc3a4850        14 seconds ago      296MB
node                6-slim              57264212e5c2        9 days ago          215MB
hello-world         latest              1815c82652c0        1 months ago        1.84kB
```

{{PLEASE REWRITE THIS CHUNK TO MAKE IT WORK WITH DOCKER CONTAINERS}}

3. Run a docker container from the image:

```
docker run -d --name cloudServer -v ~/cloudServer_scripts:/usr/src/app/cloudServer_
↪scripts  cloudserver
```

For more options to add to the container, please refer to this documentation link.

{{PLEASE REWRITE THIS CHUNK TO MAKE IT WORK WITH DOCKER CONTAINERS}}

4. Configure the container:

Open a new terminal open a bash shell to the container

```
docker exec -it cloudServer bash
```

5. Start the redis :

```
cloudServer_scripts/redis-4.0.1/src/redis-server --daemonize yes
```

The server will run as a daemon.

6. Install awscli in the docker container:

```
apt-get update && \
apt-get install apt-file && \
apt-file update && \
apt-get install awscli
```

7. Configure access keys for the utapiuser:

```
aws configure --profile utapiuser
```

- Example of configuration:

```
AWS Access Key ID [None]: accessKey1
AWS Secret Access Key [None]: verySecretKey1
Default region name [None]:
Default output format [None]:
```

For more information about User Access Configuration, please refer to this documentation

{{PLEASE LINK TO S3 AND UTAPI DOC SECTIONS ABOUT USER ACCESS CONFIGURATION}}

8. Start the UTAPI server:

```
npm run start_utapi
```

- By default the UTAPI server runs at http://localhost:8100

8. Testing Cloud Server (s3 server):

- Open a new terminal and open interative mode to the sdocker machine:

```
docker exec -it cloudServer bash
```

- Create a bucket named 'utapi-bucket'

```
aws s3api create-bucket --bucket utapi-bucket --endpoint http://localhost:8000 --
↪profile utapiuser
```

- Expected Output:

```
{
    "Location": "/utapi-bucket"
}
```

9. Copy a test file in 'utapi-bucket':

```
fallocate -l 100M file.out && \
aws s3api put-object --bucket utapi-bucket --key utapi-object --body ./file.out --
↪endpoint http://localhost:8000 --profile utapiuser
```

- Expected Output:

```
{
    "ETag": "\"2f282b84e7e608d5852449ed940bfc51\""
}
```

10. Getting Storage Utilization of the UTAPI bucket:

- Create a script in the host machine in folder that has been mounted as a volume in the container.

```
cd ~/cloudServer_scripts
```

The JS way:

- Create a script (metrics.js) with the following code (change accessKeyID, secretAccessKey and bucketName if needed):

```
const http = require('http');
const aws4 = require('aws4');

// Input AWS access key, secret key, and session token.
const accessKeyId = 'accessKey1';
const secretAccessKey = 'verySecretKey1';
const token = '';
const bucketName = 'utapi-bucket';
// Get the start and end times for a range of one month.
const startTime = new Date(2017, 7, 1, 0, 0, 0, 0).getTime();
const endTime = new Date(2017, 10, 1, 0, 0, 0, 0).getTime() - 1;
const requestBody = JSON.stringify({
    buckets: [bucketName],
    timeRange: [startTime, endTime],
});
const header = {
    host: 'localhost',
    port: 8100,
    method: 'POST',
    service: 's3',
    path: '/buckets?Action=ListMetrics',
    signQuery: false,
    body: requestBody,
};
const credentials = { accessKeyId, secretAccessKey, token };
const options = aws4.sign(header, credentials);
const request = http.request(options, response => {
    const body = [];
    response.on('data', chunk => body.push(chunk));
    response.on('end', () => process.stdout.write(`${body.join('')}\n`));
});
request.on('error', e => process.stdout.write(`error: ${e.message}\n`));
request.write(requestBody);
request.end();
```

- Run the script from the docker container:

```
node cloudServer_scripts/metrics.js
```

- Example of output:

```
[{"timeRange":[1501545600000,1509494399999],"storageUtilized":[0,104857600],
↪"incomingBytes":104857600,"outgoingBytes":0,"numberOfObjects":[0,1],"operations":{
↪"s3:DeleteBucket":0,"s3:DeleteBucketCors":0,"s3:DeleteBucketWebsite":0,
↪"s3:DeleteObjectTagging":0,"s3:ListBucket":0,"s3:GetBucketAcl":0,"s3:GetBucketCors
↪":0,"s3:GetBucketWebsite":0,"s3:GetBucketLocation":0,"s3:CreateBucket":2,
↪"s3:PutBucketAcl":0,"s3:PutBucketCors":0,"s3:PutBucketWebsite":0,"s3:PutObject":2,
↪"s3:CopyObject":0,"s3:UploadPart":0,"s3:ListBucketMultipartUploads":0,
↪"s3:ListMultipartUploadParts":0,"s3:InitiateMultipartUpload":0,
↪"s3:CompleteMultipartUpload":0,"s3:AbortMultipartUpload":0,"s3:DeleteObject":0,
↪"s3:MultiObjectDelete":0,"s3:GetObject":0,"s3:GetObjectAcl":0,"s3:GetObjectTagging
↪":0,"s3:PutObjectAcl":0,"s3:PutObjectTagging":0,"s3:HeadBucket":0,"s3:HeadObject":0,
↪"s3:PutBucketVersioning":0,"s3:GetBucketVersioning":0,"s3:PutBucketReplication":0,
↪"s3:GetBucketReplication":0,"s3:DeleteBucketReplication":0},"bucketName":"utapi-
↪bucket"}]
```

The Pythonian way(to get storage utilized):

- Create a .py file with the following code:

```
import sys, os, base64, datetime, hashlib, hmac, datetime, calendar, json
import requests # pip install requests

access_key = 'accessKey1'
secret_key = 'verySecretKey1'

method = 'POST'
service = 's3'
host = 'localhost:8100'
region = 'us-east-1'
canonical_uri = '/buckets'
canonical_querystring = 'Action=ListMetrics&Version=20160815'
content_type = 'application/x-amz-json-1.0'
algorithm = 'AWS4-HMAC-SHA256'

t = datetime.datetime.utcnow()
amz_date = t.strftime('%Y%m%dT%H%M%SZ')
date_stamp = t.strftime('%Y%m%d')

# Key derivation functions. See:
# http://docs.aws.amazon.com/general/latest/gr/signature-v4-examples.html#signature-
↪v4-examples-python
def sign(key, msg):
    return hmac.new(key, msg.encode("utf-8"), hashlib.sha256).digest()

def getSignatureKey(key, date_stamp, regionName, serviceName):
    kDate = sign(('AWS4' + key).encode('utf-8'), date_stamp)
    kRegion = sign(kDate, regionName)
    kService = sign(kRegion, serviceName)
    kSigning = sign(kService, 'aws4_request')
    return kSigning

def get_start_time(t):
```

```
    start = t.replace(minute=t.minute - t.minute % 15, second=0, microsecond=0)
    return calendar.timegm(start.utctimetuple()) * 1000;

def get_end_time(t):
    end = t.replace(minute=t.minute - t.minute % 15, second=0, microsecond=0)
    return calendar.timegm(end.utctimetuple()) * 1000 - 1;

start_time = get_start_time(datetime.datetime(2017, 6, 1, 0, 0, 0, 0))
end_time = get_end_time(datetime.datetime(2017, 9, 1, 0, 0, 0, 0))

# Request parameters for listing Utapi bucket metrics--passed in a JSON block.
bucketListing = {
    'buckets': [ 'utapi-bucket' ],
    'timeRange': [ start_time, end_time ],
}

request_parameters = json.dumps(bucketListing)

payload_hash = hashlib.sha256(request_parameters).hexdigest()

canonical_headers = \
    'content-type:{0}\nhost:{1}\nx-amz-content-sha256:{2}\nx-amz-date:{3}\n' \
    .format(content_type, host, payload_hash, amz_date)

signed_headers = 'content-type;host;x-amz-content-sha256;x-amz-date'

canonical_request = '{0}\n{1}\n{2}\n{3}\n{4}\n{5}' \
    .format(method, canonical_uri, canonical_querystring, canonical_headers,
            signed_headers, payload_hash)

credential_scope = '{0}/{1}/{2}/aws4_request' \
    .format(date_stamp, region, service)

string_to_sign = '{0}\n{1}\n{2}\n{3}' \
    .format(algorithm, amz_date, credential_scope,
            hashlib.sha256(canonical_request).hexdigest())

signing_key = getSignatureKey(secret_key, date_stamp, region, service)

signature = hmac.new(signing_key, (string_to_sign).encode('utf-8'),
                     hashlib.sha256).hexdigest()

authorization_header = \
    '{0} Credential={1}/{2}, SignedHeaders={3}, Signature={4}' \
    .format(algorithm, access_key, credential_scope, signed_headers, signature)

# The 'host' header is added automatically by the Python 'requests' library.
headers = {
    'Content-Type': content_type,
    'X-Amz-Content-Sha256': payload_hash,
    'X-Amz-Date': amz_date,
    'Authorization': authorization_header
}

endpoint = 'http://' + host + canonical_uri + '?' + canonical_querystring;

r = requests.post(endpoint, data=request_parameters, headers=headers)
```

```
s = r.text
split1 = s.rsplit('storageUtilized', 500)[1]
split2 = split1[5:].rsplit(']', 100)[0]
print (split2)
```

- Run the python file

```
python <path-to-*.py>
```

- The output is the total utilization of the bucket in bytes.