PROCESSING DATA LAKE WITH NODE.JS IN SERVERLESS ARCHITECTURE
Well, as per usual I don't know which talk to choose...

There's a session on the future of web hosting which sounds good... or maybe the one about virtual assistants...

Don't overcomplicate it, I've found the perfect talk for us.

Perfect!

HOW TO BUILD A CHATBOT FOR BLOCKCHAIN SMART CONTRACTS USING GRAPHQL WITH SERVERLESS APIs SUPPORTED BY A SERVICE MESH
Nikolay Matvienko
Software Engineer at Grid Dynamics
You can find me at twitter.com/matvi3nko
github.com/matvi3nko
DATA LAKE

ETL / ELT

STREAMING

BATCH

BIG DATA

REAL-TIME

SERVERLESS

LAMBDA

NODE.JS

PYTHON

GO

EC2

FASTER

HOW TO
THE GOAL

NEW RAW DATA

S3 Standard
Frequently accessed structured data

S3 + Hadoop
Raw and structured data

S3 infrequent access
Historical data

Amazon Glacier
Data archive

DATA LAKE

HOT DATA

COLD DATA
CASE STUDY

INTELLIGENCE SUPPLY CHAIN

- Hundreds GB RAW data and 60M new messages daily
- 8M UNIQUE ITEMS over the world
TECHNOLOGIES

UI

SERVERLESS BACKEND API
- AWS Lambda
- Node.js / GraphQL / Databases
- TS

DATABASES

DATA LAKE (BIG DATA)
- AWS S3
- Hadoop Cloudera
- pySpark

@matvi3nko
TECHNOLOGIES

UI

SERVERLESS BACKEND API
AWS Lambda
Node.js / GraphQL / Databases
TS

DATABASES
AWS Lambda
Node.js / Databases
TS

SERVERLESS DATA PROCESSING
AWS S3
Hadoop Cloudera
pySpark

DATA LAKE (BIG DATA)
export const handler = async (event) => {
    const data = event.Records[0].body;

    // - TRANSFORM data  
    // - WRITE to DB or 
    // - PUT TO QUEUE/STREAM/TOPIC

    return 'success';
};
NODE.JS POPULARITY

2014 AWS LAMBDA WITH NODE.JS
- PURE ASYNCHRONOUS
- MINIMALISTIC CORE
- FAST STARTUP WITH HIGH PERFORMANCE

2018 SERVERLESS.COM

51% Node.js 8.10
16% Python 2.7
13% Node.js 6.10
9% Python 3.6
3% Java 8
2% Go
NODE.JS FUNCTION VS SERVER

EVENT LOOP

MANY REQUESTS

ONE REQUEST

With one or batch of records

ASYNC CALLS

ASYNC RESULT

BUSY BY CPU HEAVY TASKS

@matvi3nko
I/O – PERFORMANCE BOTTLENECK

@matvi3nko
SUMMARY

CPU HEAVY TASKS
Get request with single record

I/O INTENSIVE TASKS
Get request with batch of records (like a server)

REQUEST → SINGE record → EVENT LOOP → I/O CALL

REQUEST → BATCH of records → EVENT LOOP → I/O TASKS ASYNC CALLS
SERVERLESS COMPUTE SERVICE

DATA LAKE S3  AWS LAMBDA  DATA LAKE S3  LAMBDA FUNCTION

Symbols in the presentation.

THIS IS MY ARCHITECTURE – 177 VIDEOS
https://aws.amazon.com/ru/this-is-my-architecture/

@matvi3nko
EXAMPLE

SPEAKER

DISPLAY

CAMERA

... + 1000 MICROCHIPS

FACTORIES

CASE

MICROPHONE

@matvi3nko
BATCH PROCESSING

DATA FROM FACTORIES

SNAPSHOT:
/yyyy=2019/mm=05/dd=25/category

@matvi3nko
BATCH DATA PROCESSING

SNAPSHOT:
/yyy=2019/mm=05/dd=25/

FUNCTION
SCAN
GET FILES FOR EACH FACTORY

FUNCTION
READ FILES
MERGE RECORDS

FUNCTION
MERGED RECORD
from tables/files

FUNCTION
PROCESS/ANALYZE
MODEL

DB

DONE!
RELOCATION OF PEOPLE TO A NEW BUILDING

SNAPSHOT: /yyyy=2019/mm=05/dd=25/

LIST

QUEUE

QUEUE

DOORS

ELEVATORS

STREAMS

@matvi3nko
BATCH PROCESSING ARCHITECTURE

SNAPSHOT:
/yyyy=2019/mm=05/dd=25/

DATABASE

TABLES/FILES for each factory

FUNCTION

READ FILES MERGE RECORDS

FUNCTION

PROCESS/ANALYZE MODEL

FUNCTION

DATA STREAMS

FUNCTION

SCAN GET FILES FOR EACH FACTORY

QUEUE

NOTIFICATION TOPIC

QUEU
REAL-TIME PROCESSING ARCHITECTURE

SNAPSHOT: 
/yyy=2019/mm=05/dd=25/

NOTIFICATION TOPIC

QUEUE

FUNCTION

TABLES/FILES for each factory

QUEUE

FUNCTION

READ FILES

MERGED RECORD from tables/files

DATA STREAMS

FUNCTION

PROCESS/ANALYZE MODEL

DB
CONCLUSION

1. FROM BIG DATA TO A LARGE NUMBER OF MESSAGES
2. THE MORE MESSAGES THE FUNCTION ACCEPTS, THE MORE IT NEEDS TO BE PARALLELIZED
3. USE THE QUEUE FOR MESSAGES, AND DATA STREAMS TO TRANSFER MODELS / LARGE COLLECTION
4. INCREASE THE NUMBER OF STREAM SHARDS. 1 SHARD = 1 LAMBDA FUNCTION
5. PREPARE TO STREAMING / REAL-TIME PROCESSING
PROGRESS

SOLUTION STRUCTURE AND FUNCTION

BASE ARCHITECTURE DESIGN
import AWS from 'as-sdk';
const s3Client = new AWS.S3({region});

export const handler = (event) => {
  const [message] = event.Records;

  return new Promise((resolve, reject) => {
    this.s3Client.selectObjectContent({ Key: message.path }, (err, data) => {
      if (err) {
        reject(err);
      }
      resolve(data);
    });
  });
};
SERVERLESS PROJECT STRUCTURE

//transform
- serverless.yml
- handler.ts

//analyze
- serverless.yml
- handler.ts

/node_modules
serverless.yml
package.json

DISADVANTAGES

1. NODE_MODULES contains dependencies of all functions
   Have to control and split them in SERVERLESS.YML
2. Lack of function isolation
3. Lack of independent install / build / test
4. Becomes monolith project
MONOREPO SERVERLESS PROJECTS STRUCTURE

/lib
/node_modules
/errors
/factories
/models
/providers
– package.json
/node_modules
– functionA1.ts
– package.json
– serverless.yml
/node_modules
– functionA2.ts
– package.json
– serverless.yml

ADVANTAGES

1. LIB contains all common infrastructure, domain logic and cloud provider’s SDK
2. Functions became isolated projects with flexible splitting and contains only business logic
3. LIB and PROJECTS versioning
4. NPM resolves NODE_MODULES dependencies automatically
5. Independent install / build / test / deploy / troubleshooting

@matvi3nko
import { Model } from '@holyjs/models'
INITIALIZE ONLY ONCE

FIRST EXECUTION
ON COLD START

ALL NEXT
ON WARM START

ENCAPSULATE IN LIB
BOILERPLATE CODE
USE FACTORIES / IoC

```javascript
import { createCsvS3Provider } from '@holyjs/lib';
const service = new Service(createCsvS3Provider());

export const handler = async (event) => {
  // Logic
};
```
import { createCsvS3Provider, createQueueProvider } from '@holyjs/lib';
const service = new Service(createCsvS3Provider());

export const handler = async (event) => {
    const source = service.getObject(event)
        .pipe(flatMap(service.transform))
        .pipe(bufferCount(10))
        .pipe(flatMap(message => createQueueProvider().putBatch(message)));

    return new Promise((resolve, reject) => {
        source.subscribe(() => { /* handle */},
            err => {
                err instanceof InfrastructureError && reject(err);
                err instanceof DomainError && reject(err);
            },
            resolve);
    });
}
import { createCsvS3Provider, createQueueProvider } from '@holyjs/lib';
const service = new Service(createCsvS3Provider());

export const handler = async (event) => {
  const source = service.getObject(event)
    .pipe(flatMap(service.transform))
    .pipe(bufferCount(10))
    .pipe(flatMap(message => createQueueProvider().putBatch(message)));

  return new Promise((resolve, reject) => {
    source.subscribe(() => {
      /* handle */
    },
    err => {
      err instanceof InfrastructureError && reject(err);
      err instanceof DomainError && reject(err);
    },
    resolve);
  });
};
4 COMPONENTS OF THE FUNCTION

1. INCOMING DATA
2. TYPE OF TRANSFORMATION
3. DESTINATION
4. MAIN ERROR HANDLER
PROGRESS

BASE ARCHITECTURE DESIGN

SOLUTION STRUCTURE AND FUNCTION

ERROR HANDLING
1. Request XX-YY: “Process exited before completing request”
2. Function completed on its timeout (up to limit)
HOW TO HANDLE

1. FUNCTION HANGS
   Don’t do that: `context.callbackWaitsForEmptyEventLoop = false;`
   or close Sequelize connection to fix that
   Use callback cb(), rewrite to async/await (Promises)

2. FUNCTION DOES NOT PERFORM PART OF THE LOGIC
   You added async or return value.
   Find missed await / return Promise
WAIT FOR RESPONSES FROM THE SERVICES

```cpp
// CODE
SEND(MESSAGE);
// CODE
RETURN SEND(MESSAGE);
// CODE
AWAIT SEND(MESSAGE);
// CODE
SEND;
```

SEND

SAVE 1$ → LOSE MILLIONS $
RETRY STRATEGY

RETRIES 3 TIMES (BY DEFAULT)

DATA LAKE → QUEUE
SCAN ALL OBJECTS FOR EACH FOLDER
OBJECTS PATHS
QUEUE
DISPATCH MERGE BY FACTORY
PHONE WITH ALL INFO
QUEUE
TRANSFORM
QUEUE
ANALYZE
DATABASE

GET ALL RECORDS AND AGGREGATE EACH WITH DETAILS

New SNAPSHOT DATE

@matvi3nko
DEAD LETTER QUEUE

RETries 3 TIMES

Transform

returns Error

DLQ
STORES FAILED MESSAGES

"reprocess_[type]_errors"

REPROCESS
PULLS MESSAGES FROM DLQ BY THE REQUEST

REPROCESSING TOPIC

ACTOR

TOPIC

1

2

3

4

5

6

7
DLQ FOR THE QUEUE

DATA LAKE 🔄 New SNAPSHOT DATE
QUEUE 🔄 DISPATCH
MENT BY FACTORY
DATA STREAMS 🔄 ANALYZE
DATABASE 🔄

DATA LAKE 🔄 New SNAPSHOT DATE
QUEUE 🔄 SCAN ALL OBJECTS
FOR EACH FOLDER
OBJECTS PATHS 🔄 ANALYZE
GET ALL RECORDS AND
AGGREGATE EACH
WITH DETAILS

QUEUE 🔄 TRANSFORM
RECORDS 🔄 DATABASE

ACTOR 🔄 REPROCESSING TOPIC
"reprocess_messages"
DLQ 🔄 REPROCESS
PULLS MESSAGES
FROM DLQ BY THE REQUEST

1. EASY TO TROUBLESHOOT
2. NO CORE DUMPS
3. NO NEED TO REPROCESS
   GIGABYTES OF DATA AGAIN
KINESIS ERROR HANDLING

DATA LAKE
New SNAPSHOT DATE

QUEUE

DISPATCH
MERGE BY FACTORY

TOPIC

OBJECTS PATHS

SCAN ALL OBJECTS
FOR EACH FOLDER

QUEUE

GET ALL RECORDS AND
AGGREGATE EACH
WITH DETAILS

RECORDS

DATA STREAM

DATABASE

ACTOR

"reprocess_messages"

REPROCESSING TOPIC

REPROCESS
PULLS MESSAGES
FROM DLQ BY THE REQUEST

DLQ

@matvi3nko
ERROR HANDLING STRATEGY

1. PUT TO STREAM (TRANSFORM)
   a. Handle partially successful response
      FailedRecordCount : Number

2. READ FROM (ANALYZE)
   a. Reduce retry times
      customBackoff: (retryCount) => {..}
   b. Use Dead Letter Queue
   c. Handle duplicate Records

UP TO 7 days each 100 ms
EXACTLY ONCE PROCESSING

1. REDUCE THE RISK OF FAILURE
   
   ```javascript
   const response = await putToStream(record);
   // do something with response -> RISK
   ```

2. USE REDIS CACHE TO STORE KEYS OF RECORDS
   
   key: [date]-[shard-id]-[sequence-number]

3. FILTER DUPLICATES
KINESIS ERROR HANDLING

RETries N TIMES
24h - 7d

InfraStructure Errors
reject(error)

BUSiness Errors
INTERNAL ERRORS

REPROCESS
PULLS MESSAGES
FROM DLQ BY THE REQUEST

DLQ
STORES
FAILED RECORDS

NOTIFICATION TOPIC
FOR DLQ

Redis
resolve()
SEND RECORD
TO SNS TOPIC

ANALYZE
FUNCTION

Pull by error type
ARCHITECTURE WITH REPROCESSING BLOCK

DATA LAKE → New SNAPSHOT DATE → QUEUE → λ  
SCAN ALL OBJECTS FOR EACH FOLDER → OBJECTS PATHS → QUEUE → λ  
GET ALL RECORDS → PHONE WITH ALL INFO → λ  
DATA STREAM → ANALYZE

Redis

DATABASE

"reprocess_messages"

ACTOR → REPROCESSING TOPIC → λ  
REPROCESS → DLQ → λ  
REPROCESS TOPIC → λ  
REPROCESS → DLQ → λ  
REPROCESS FOR DLQ
CONCLUSION

1. CREATE CUSTOM TYPES OF ERRORS
2. REPROCESS ONLY FAILED MESSAGE AND NOT GIGA/TERABYTES OF DATA
3. DON'T LOSE MESSAGES
4. USE DLQ WITH FILTERING
5. PROCESS EXACTLY ONCE
PROGRESS

@matvi3nko
AWS S3: SIMPLE STORAGE SERVICE

1. S3 SELECT REQUEST

```javascript
const params = {
  Bucket: 'bucket_name',
  Key: 'key_name',
  ExpressionType: 'SQL',
  Expression: `SELECT s.name, s.year FROM S3Object s WHERE s.name = 'HolyJS'`,
},
OutputSerialization: {
  JSON: { RecordDelimiter: '\n' },
};
```

2. QUERY RESULT

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
<th>year</th>
<th>details</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

UP TO 4x FASTER

JSON | CSV | Parquet
DATA FORMATS

RAW OR COMPRESSED?

80MB
PARQUET
BINARY

1.4GB
CSV
TEXT. ONLY VALUES

~X20 bigger size

6.5GB
JSON
TEXT. FIELDS + VALUES

~X80 bigger size
75% - fields names
EXTRACTING BIG OBJECTS/FILES

DATA LAKE

New SNAPSHOT DATE

QUEUE

DATA STREAM

PHONE WITH ALL INFO

REDIS

DATABASE

OBJECTS PATHS

SCAN ALL OBJECTS FOR EACH FOLDER

GET ALL RECORDS

PHONE WITH ALL INFO

TOPIC

TRANSFORM

REPROCESS

TOPIC for DLQ

ALGORITHM FOR DLQ

DLQ

REPROCESS

REPROCESSING	TOPIC

REPROCESSING TOPIC

"reprocess_messages"

ACTOR

@matvi3nko
EXTRACT DATA

S3 OBJECT PATH:

s3/buckets/bucket-name/entity/yyyy=2019/mm=05/dd=25/partition-by-category/key.parquet
DATA TRANSFORMATIONS

Component | Details | Where used
---|---|---

**ELEMENT-WISE**

**AGGREGATION**

**COMPOSITION**
BIG FILES PROCESSING

1. GET ALL IDs/KEYs

2. MERGE BY ID/KEY

3. START EXTRACTING DATA FROM S3 USING SELECT IN PARALLEL REQUEST
   `SELECT * FROM S3Object s WHERE s.id IN (1,2,3)`

4. TRANSFORM DATA STREAM AND SEND TO KINESIS
   3GB CSV
   12GB JSON
   ~1.5 minute
SENDING BIG OBJECTS

Component: S3 OBJECTS
Details: JSON
Where used: BIG SIZE

QUEUE → TRANSFORM → SENDING IN BATCH → KINESIS
DECOUPLING

NNN Records

S3 OBJECTS
Component Details

WHERE USED

S3 OBJECT

TRANSFORM

QUEUE

TRANSFORM

KINESSES
AHEAD OF THE QUEUE

DATA LAKE → New SNAPSHOT DATE

DATA LAKE

Queue

Lambda

Scan all objects for each folder

Lambda

Objects paths

Lambda

Get all records

Lambda

Transform

Lambda

Data stream

Lambda

Analyze

Redis

DATABASE

"reprocess_messages"

Actor

Reprocessing topic

Reprocess

Reprocess

Topic

Reprocess

DLQ

DLQ

DLQ
ON-DEMAND REPROCESSING

DATA LAKE

New SNAPSHOT DATE

SCAN ALL OBJECTS FOR EACH FOLDER

OBJECTS PATHS

QUEUE

PRIORITY QUEUE

DISPATCH

TOPIC

QUEUE

TOPIC

JOBS

API SERVICE

CREATE JOB

JOBS TABLE

REPROCESS DATE

@matvi3nko
CONCLUSION

1. USE DATA COMPRESSION
2. GZIP RECORDS BEFORE PUT TO KINESIS STREAM
3. INCREASE MEMORY (CPU AUTO-LY) TO WORK WITH BIG OBJECTS/FILES
4. BUT DON’T BUFFER RESPONSE, WORK WITH STREAMS
PROGRESS

- BASE ARCHITECTURE DESIGN
- SOLUTION STRUCTURE AND FUNCTION
- ERROR HANDLING
- READ AND PROCESS DATA
- SCALABILITY
ARCHITECTURE

S3 -> TASK-DISPATCHER -> REPROCESSOR -> DLQ

ACTOR

REPROCESSING TOPIC

TRANSFORM

ANALYZE

DATABASE -> Redis

ADD LAMBDA FUNCTIONS FOR EACH COMPONENT

REPROCESSOR

DLQ
PROBLEM

SHARDS OF THE STREAM

1

2

3

DATABASE

ANALYZE

DATABASE
DECOUPLING

1. READ IN BATCH
2A. READ ONLY 1
2B. DECOUPLE LONG I/O READ IN BATCH

INFRASTRUCTURE ERRORS

MODEL A STREAM
WRITE
NOSQL DB

MODEL B STREAM
WRITE
SQL DB

PROCESSING ERRORS

KINESIS
ANALYZE
PROGRESS

- BASE ARCHITECTURE DESIGN
- SOLUTION STRUCTURE AND FUNCTION
- ERROR HANDLING
- READ AND PROCESS DATA
- SCALABILITY
1. 25% to develop – 75% spend on integration, but more flexible for changes in a result.
2. 5K, 7K, 10K lambda functions in parallel
3. Terabytes of data
4. Serverless vs EC2 cost
SUMMARY

1. PERFORMANCE
2. TROUBLESHOOTING
3. MEMORY LEAKS
4. PROVIDER LOCK
5. ARCHITECTURE
THANKS!

Nikolay Matvienko
matvi3nko@gmail.com
Twitter.com/matvi3nko
github.com/matvi3nko