cardlay®

ZIPFILES ON THE FLY WITH MICRONAUT

Jacob Aae Mikkelsen

Greach

AGENDA

- The Problem
- Simple Solution
- Backpressure and Reactive Programming
- Zipfiles in Java
- Testing the solution

Examples are in Micronaut

JACOB AAE MIKKELSEN

- Senior Software Architect at Cardlay A/S
- GR8Conf EU - Organizing Team Member
- External Associate Professor - University of Southern Denmark
- Groovy Ecosystem Nerd
- @JacobAae

THE PROBLEM

Download a selection of images in a zipfile

- Storage is expensive
  - Wish to not have (unused) zipfiles laying around

FIRST ATTEMPT

CREATING A ZIP FILE

```
ByteArrayOutputStream createZipOutput(List<String> filenames) {
  ByteArrayOutputStream baos = new ByteArrayOutputStream()
  ZipOutputStream myzipFile = new ZipOutputStream(baos)

  filenames.each { String filename ->
    myzipFile.putNextEntry(new ZipEntry(filename))
    URL resource = this.class.classLoader.getResource(filename)
    myzipFile.write(resource.openStream().bytes)
    myzipFile.closeEntry()
  }
  myzipFile.finish()
  myzipFile.close()
  baos
}
```

STREAMEDFILE FROM SERVICE

```
StreamedFile getStreamedFileZipfile() {
  log.debug("Producing zipfile as StreamedFile")

  ByteArrayOutputStream baos = createZipOutput(
        fileInfoRepository.filenames)
  def inputstream = new ByteArrayInputStream(
        baos.toByteArray())

  new StreamedFile(inputstream, "download.zip")
}
```

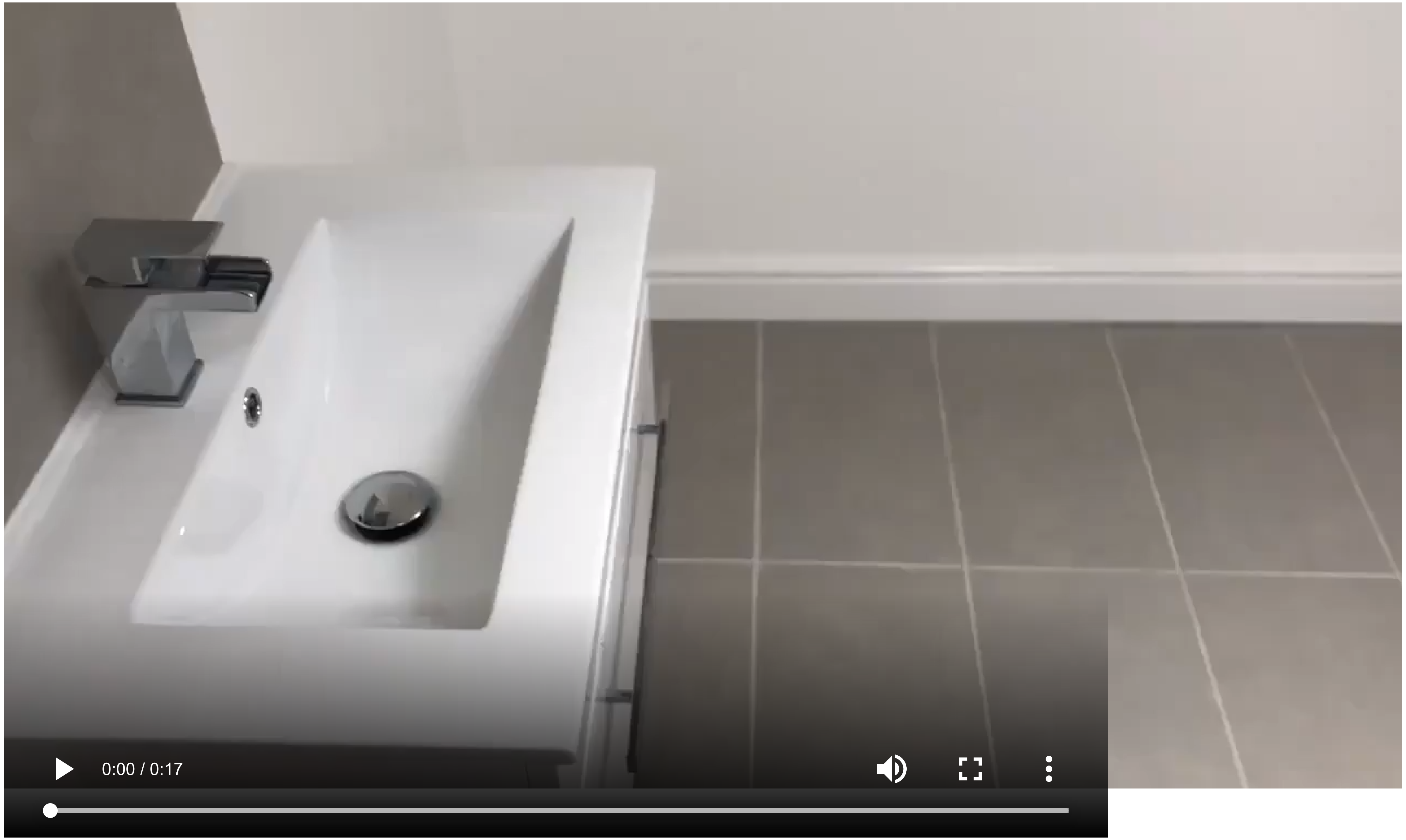
CONTROLLER

```
@Get("/plain")
StreamedFile plain() {
  log.debug("Download plain StreamedFile zip")
  zipService.streamedFileZipfile
}
```

USAGE

```
$ curl localhost:8080/zip/plain > out.zip
  % Total    % Received % Xferd  Average Speed   Time    Time
                                 Dload  Upload   Total   Spent
100 40.2M    0 40.2M    0     0  26.5M      0 --:--:--  0:00:01
```

WIN?



CAVEAT

- This builds entire zipfile in memory before starting to serve it
- What if multiple clients requests simultaneously
  - And what if they are slow to consume it (slow internet)

💡 *Remember non-functional requirements*

SIMULTANEOUS PROCESSES

parallel_commands.sh

```
#!/bin/bash

for cmd in "$@"; do {
  echo "Process \"$cmd\" started";
  $cmd & pid=$!
  PID_LIST+=" $pid";
} done

trap "kill $PID_LIST" SIGINT
echo "Parallel processes have started";

wait $PID_LIST
echo
echo "All processes have completed";
```

TEST SCENARIO

- Micronaut started with `-Xmx512m`
- 8 Parallel Processes
- Each limited to maximum transfer rate 500 Kb/sec

```
./parallel_commands.sh \
"curl -v --limit-rate 500K localhost:8080/zip/plain -o o-0.zip" \
"curl -v --limit-rate 500K localhost:8080/zip/plain -o o-1.zip" \
"curl -v --limit-rate 500K localhost:8080/zip/plain -o o-2.zip" \
"curl -v --limit-rate 500K localhost:8080/zip/plain -o o-3.zip" \
"curl -v --limit-rate 500K localhost:8080/zip/plain -o o-4.zip" \
"curl -v --limit-rate 500K localhost:8080/zip/plain -o o-5.zip" \
"curl -v --limit-rate 500K localhost:8080/zip/plain -o o-6.zip" \
"curl -v --limit-rate 500K localhost:8080/zip/plain -o o-7.zip"
```

TEST RESULT

```
ERROR i.m.h.s.netty.RoutingInBoundHandler -
  Unexpected error occurred: Java heap space
```

SOLUTION

We should produce the zip in parts and serve those when the consumer is ready

Backpressure

DIAGRAM - NAIVE WAY

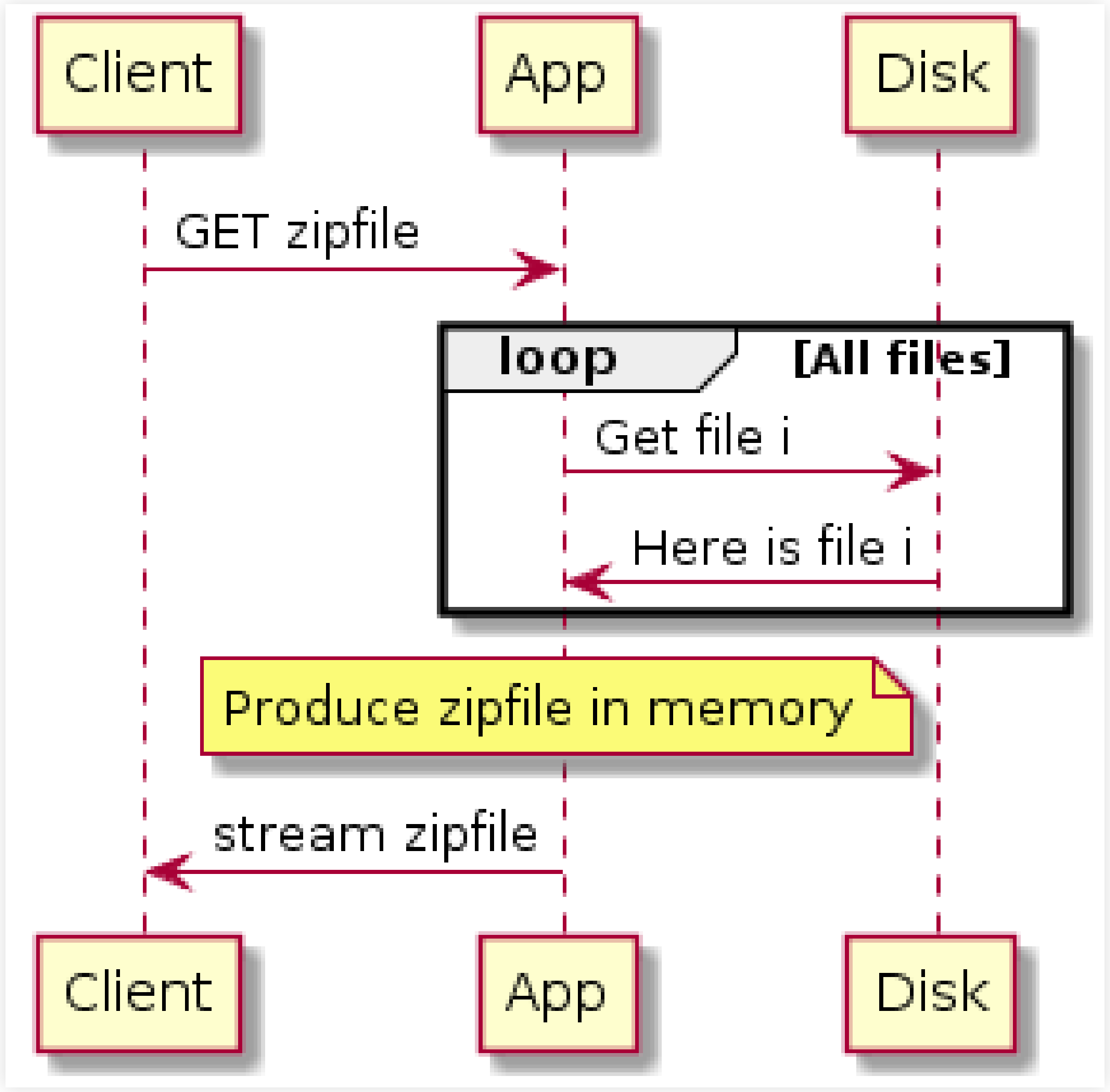
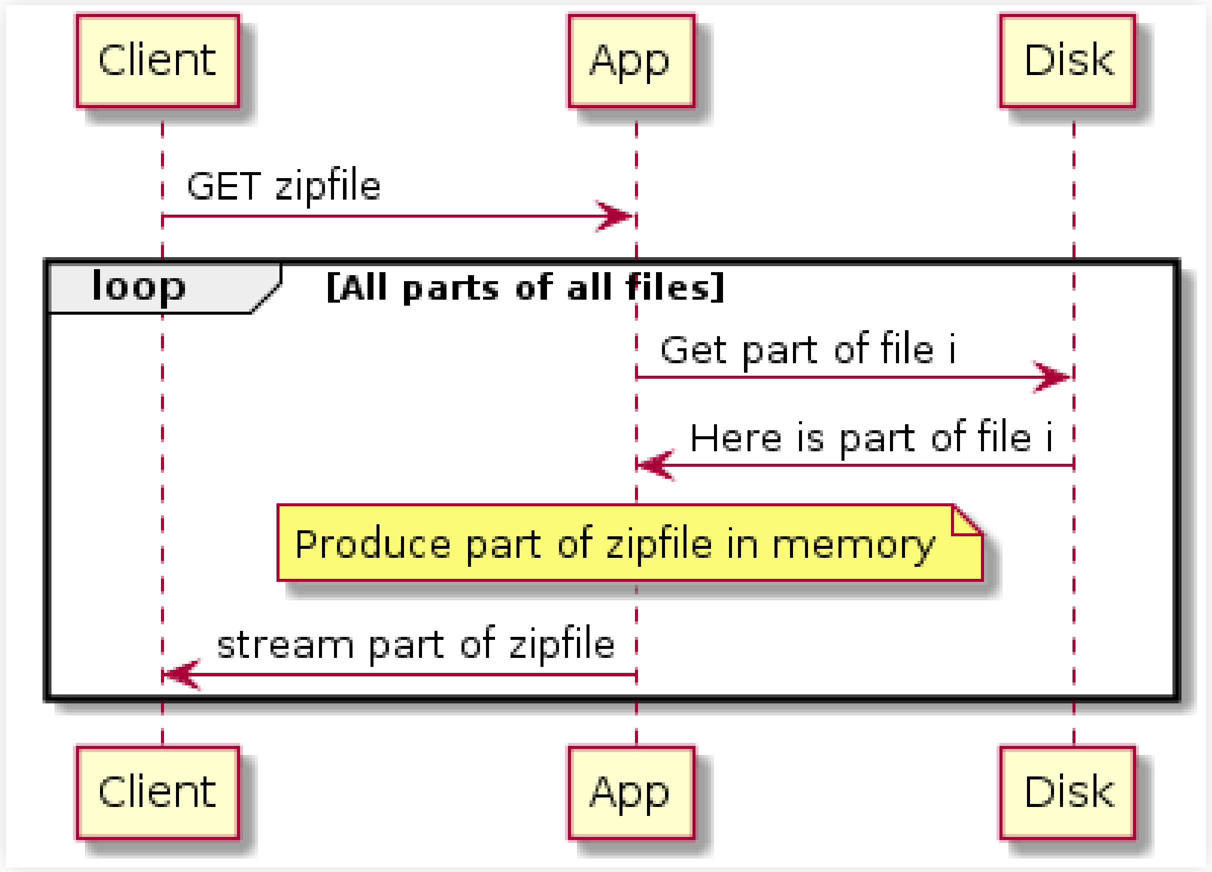


DIAGRAM - REACTIVE WAY



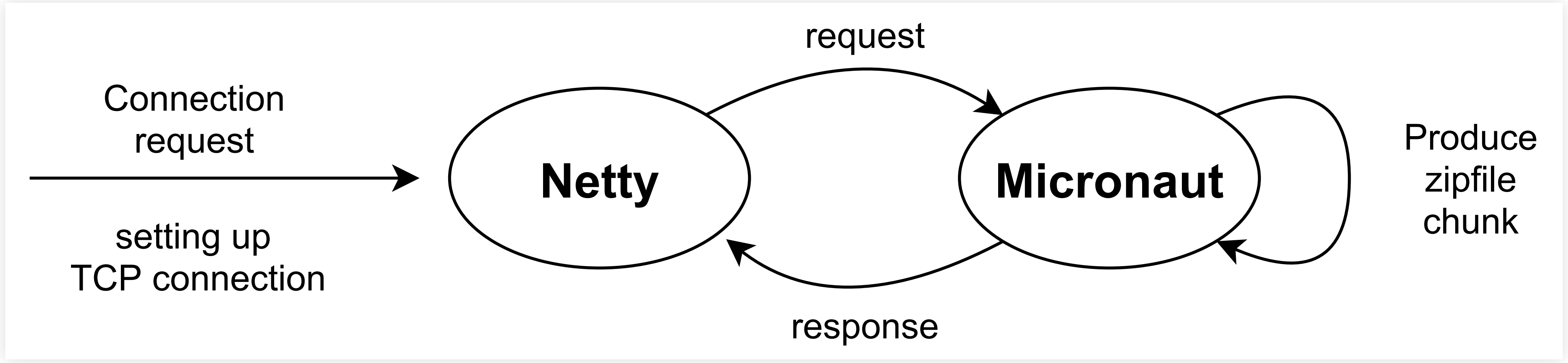
REACTIVE PROGRAMMING AND RXJAVA2

REACTIVE PROGRAMMING 101

Reactive Programming Is Programming With Asynchronous Data Streams - you react when an event happens in the stream

Reactive Programming != Reactive System

EVENT LOOP (SIMPLIFIED)



RXJAVA2

RxJava is a Reactive Extensions Java implementation that allows us to write event-driven, and asynchronous applications

In RxJava a stream is called Observable

We would like to generate an Observable that Micronaut (and Netty) can subscribe to

HOT OR COLD OBSERVABLE

Observable can be **hot** or **cold**

A hot Observable begins generating items and emits them immediately when they are created. Hot Observable emits items at its own pace, and it is up to its observers to keep up.

It is contrary to a Cold Observables pull model of processing.

Hot observables need special strategy for handling backpressure (error, drop latest/oldest)

BACKPRESSURE AWARE

In RxJava2 clear distinction between backpressure-aware and non-backpressure-aware sources

- **Observable** sources don’t support backpressure.
  - We should use it for sources that we merely consume and can’t influence.
- **Flowable** backpressure aware source

FLOWABLE<T>

*`Flowable<T>`: Emits 0 or n items and terminates with an success or an error event. Supports **backpressure**, which allows to control how fast a source emits items.*

FLOWABLE

```
Flowable.generate(
    Callable<S> initialState,
    final BiConsumer<S, Emitter<T>> generator
)
```

> *Returns a cold, synchronous, stateful and backpressure-aware generator of values.*
>
> — From the Javadoc

FLOWABLE

- `initialState` Setup relevant state for use when producing elements of the reactive stream
  - *Here:* Keep track of the files to include, the current file, and where in the current file we have read and included
- `generator` Produce and emit the generated elements and report when done or error.
  - *Here:* Read the next part of the current file and include in the zip

ZIPFILES IN JAVA/GROOVY LAND

ZIP METHODS

Files included when produced with ZipOutputStream is using one of two methods: **DEFLATED** and **STORED**. Big difference when writing!

```
ZipEntry entry = current.entry;
switch (entry.method) {
case DEFLATED:
    super.write(b, off, len);
    break;
case STORED:
    written += len;
    if (written - locoff > entry.size) {
        throw new ZipException(
            "attempt to write past end of STORED entry");
    }
    out.write(b, off, len);
    break;
```

DEFLATED

- Compression used
- No outout produced before a full file is included
- Dangerous (memory wise) if large files are included

STORED

- No compression used
- Output streamed right through
- **BUT** Requires CRC code and file size when file is added (not after)
    - These could be stored in database

METHOD

We will use STORED mode, since images are already compressed and due to memory concerns

THE MICRONAUT WAY

with RxJava 2

CONTROLLER

```
@Get("/chunked" )
@Produces('application/octet-stream')
@Header(name="Content-Disposition",
    value='attachment; filename="filename.zip"')
Flowable<byte[]> chunked()  {
    log.debug("Download chunked/reactive zip")
    zipService.zipInChunks
}
```

SERVICE

```
Flowable<byte[]> getZipInChunks() {
  Flowable.generate ( { -> // Initial state
       new ZipProducerState(repository.filenames, repository)
    },
    { ZipProducerState state, emitter -> // Generator
      state.produceNext()
      state.hasNext() ? emitter.onNext( state.getNext() ) :
            emitter.onComplete()
      state
    } as io.reactivex.functions.BiConsumer
  )
}
```

ZIPPRODUCERSTATE - CONSTRUCTOR

```
ZipProducerState(List<String> files,
       FileInfoRepository fileInfoRepository) {
  this.files = files
  this.fileInfoRepository = fileInfoRepository
  currentFileIndex = 0
  outputStream = new ChunkedByteoutputStream(bufferCapacity)
  zipStream = new ZipOutputStream(outputStream)

  gotoNextFile()
}
```

CHUNKEDBYTEOUTPUTSTREAM

- Output stream that buffers into fair sized byte[]
- Buffer size set at creation time fx. 4Kb

ZIPPRODUCERSTATE - GOTONEXTFILE 1

```
private gotoNextFile() {
  if( currentFileIndex < files.size() ) {
    String filename = files[currentFileIndex]
    URL resource = this.class.classLoader.getResource(filename)
    currentFileStream = resource.openStream()

    ZipEntry zipEntry = new ZipEntry(filename)
    zipEntry.setMethod(ZipEntry.STORED)
    zipEntry.crc = fileInfoRepository.getCrc(filename)
    zipEntry.size = fileInfoRepository.getSize(filename)

    zipStream.putNextEntry(zipEntry)

    currentFileIndex++
```

ZIPPRODUCERSTATE - GOTONEXTFILE 2

```
    } else {
        zipStream.closeEntry()
        zipStream.finish()
        zipStream.close()

        outputStream.finishedWithInput()
        done = true
    }
}
```

ZIPPRODUCERSTATE - PRODUCENEXTCHUNK()

```
private void produceNextChunk() {
  if( done ) { return }
  byte[] tempBuffer = new byte[bufferCapacity]
  int read = currentFileStream.read(tempBuffer)
  if( read > 0) {
    if( read < bufferCapacity) {
      zipStream.write(Arrays.copyOf(tempBuffer, read))
    } else {
      zipStream.write(tempBuffer)
    }
  } else {
    gotoNextFile()
  }
}
```

This can produce small chunks if images are small

ZIPPRODUCERSTATE - UTILITIES

```
void produceNext() {
  while( !done && !outputStream.hasNext() ) {
    produceNextChunk()
  }
}

boolean hasNext() {
  outputStream.hasNext()
}

byte[] getNext() {
  outputStream.getNext()
}
```

TESTING

TEST SCENARIO

- Micronaut started with `-Xmx100m` (1/5 of before)
- 30 Parallel Processes
- Each limited to maximum transfer rate 500 Kb/sec

```
./parallel_commands.sh \
"curl -v --limit-rate 500K localhost:8080/zip/chunked -o 0.zip" \
"curl -v --limit-rate 500K localhost:8080/zip/chunked -o 1.zip" \
"curl -v --limit-rate 500K localhost:8080/zip/chunked -o 2.zip" \

...

"curl -v --limit-rate 500K localhost:8080/zip/chunked -o 28.zip"\
"curl -v --limit-rate 500K localhost:8080/zip/chunked -o 29.zip"
```

TEST RESULT

```
All processes have completed
```

And all zipfiles downloaded

KEY LEARNINGS

- Remember non-functional requirements
- ZipOutputStream has **two** methods: DEFLATED and STORED
- If storage and time is not a problem - consider sending a link once zip is generated and use a queue
  - If it is - go reactive with backpressure

THANKS

Code and slides: https://github.com/JacobAae/zipstreamer

QUESTIONS

Feedback: https://greach.contestia.es