PROCESSING (BIG) DATA THE MICROSERVICE WAY

DATASERVICES

Dr. Josef Adersberger (@adersberger), QAware GmbH
ENTERPRISE

BIG DATA

PROCESSING

http://www.datasciencecentral.com

http://www.cardinalface.com/misc/companies_list.html
All things distributed:
- distributed processing
- distributed databases

Data to information:
- machine (deep) learning
- advanced statistics
- natural language processing
- semantic web

Low latency and high throughput:
- stream processing
- messaging
- event-driven
DATA PROCESSING

SYSTEM INTEGRATION

APIS

UIS

data -> information

information -> systems

information -> user

information -> blended information
SOLUTIONS

Beer is the cause and solution to all of life’s problems.

(Homer Simpson)
THE {BIG, SMART, FAST} DATA SWISS ARMY KNIVES

Spark

Apache Flink

Hadoop MapReduce
Distributed Data

Driver

Distributed Processing

Distributed Data

icon credits to Nimal Raj (database), Arthur Shlain (console) and alvarobueno (takslist)
DATA SERVICES

{BIG, FAST, SMART} DATA

MICRO-SERVICE
BASIC IDEA: DATA PROCESSING WITHIN A GRAPH OF MICROSERVICES

DIRECTED GRAPH OF MICROSERVICES EXCHANGING DATA VIA MESSAGING
BASIC IDEA: COHERENT PLATFORM FOR MICRO- AND DATASERVICES
OPEN SOURCE DATASERVICE PLATFORMS

- Open source project based on the Spring stack
  - Microservices: Spring Boot
  - Messaging: Kafka 0.9, Kafka 0.10, RabbitMQ

- Open source by Lightbend (part. commercialised & proprietary)
  - Microservices: Lagom, Play
  - Messaging: akka

- Standardized API with several open source implementations
  - Microservices: JavaEE micro container
  - Messaging: JMS
THE BIG PICTURE

TARGET RUNTIME

SPRING CLOUD DATA FLOW SERVER (SCDF SERVER)

CHANNELS (input/output)

SPRING CLOUD STREAM

SPRING BOOT

SPRING INTEGRATION

SPRING FRAMEWORK

MESSAGE BROKER

APP

BINDER

APP

APP

APP

kafka

RabbitMQ
THE VEINS: SCALABLE DATA LOGISTICS WITH MESSAGING

STREAM PARTITIONING: TO BE ABLE TO SCALE MICROSERVICES
BACK PRESSURE HANDLING: TO BE ABLE TO COPE WITH PEEKS
STREAM PARTITIONING

PARTITION KEY -> PARTITION SELECTOR -> PARTITION INDEX

\[ f(\text{message}) \rightarrow \text{field} \quad f(\text{field}) \rightarrow \text{index} \quad f(\text{index}) \rightarrow \text{pindex} \]

\[ \text{pindex} = \text{index} \mod \text{output instances} \]
BACK PRESSURE HANDLING

1. Signals if (message) pressure is too high
2. Regulates inbound (message) flow
3. (Data) retention lake
DISCLAIMER: THERE IS ALSO A TASK EXECUTION MODEL (WE WILL IGNORE)

- short-living
- finite data set
- programming model = Spring Cloud Task
- starters available for JDBC and Spark as data source/sink
CONNECTED CAR PLATFORM

EDGE SERVICE
- MQTT Broker (apigee Link)
- MQTT Source
- Data Cleansing
- Masterdata Blending
- Realtime traffic analytics

KPI ANALYTICS
- Spark
- Presto
- react-vis

DASHBOARD

ESB
- Camel
- gPRC

Kafka

Kafka
TALK CHEAP
SHOW ME THE CODE

DATA SERVICES

DEVELOPERS'S VIEW - ON SPRING CLOUD DATA FLOW
ASSEMBLING A STREAM

- App starters: A set of pre-built apps aka dataservices
- Composition of apps with linux-style pipe syntax:

http | magichappenshere | log

<table>
<thead>
<tr>
<th>Source</th>
<th>Processor</th>
<th>Sink</th>
</tr>
</thead>
<tbody>
<tr>
<td>file</td>
<td>aggregator</td>
<td>aggregate-counter</td>
</tr>
<tr>
<td>ftp</td>
<td>bridge</td>
<td>cassandra</td>
</tr>
<tr>
<td>gemfire</td>
<td>filter</td>
<td>counter</td>
</tr>
<tr>
<td>gemfire-cq</td>
<td>groovy-filter</td>
<td>field-value-counter</td>
</tr>
<tr>
<td>http</td>
<td>groovy-transform</td>
<td>file</td>
</tr>
<tr>
<td>jdbc</td>
<td>httpclient</td>
<td>ftp</td>
</tr>
<tr>
<td>jms</td>
<td>pmml</td>
<td>gemfire</td>
</tr>
<tr>
<td>load-generator</td>
<td>scriptable-transform</td>
<td>gpfdlist</td>
</tr>
<tr>
<td>loggregator</td>
<td>splitter</td>
<td>hdfs</td>
</tr>
<tr>
<td>mail</td>
<td>tasklaunchrequest-transform</td>
<td>hdfs-dataset</td>
</tr>
<tr>
<td>mongodb</td>
<td>tcp-client</td>
<td>jdbc</td>
</tr>
<tr>
<td>rabbit</td>
<td>transform</td>
<td>log</td>
</tr>
<tr>
<td>s3</td>
<td>mongodb</td>
<td></td>
</tr>
<tr>
<td>sftp</td>
<td>pgcopy</td>
<td></td>
</tr>
<tr>
<td>syslog</td>
<td>rabbit</td>
<td></td>
</tr>
<tr>
<td>tcp</td>
<td>redis-pubsub</td>
<td></td>
</tr>
<tr>
<td>tcp-client</td>
<td>router</td>
<td></td>
</tr>
<tr>
<td>time</td>
<td>s3</td>
<td></td>
</tr>
<tr>
<td>trigger</td>
<td>sftp</td>
<td></td>
</tr>
<tr>
<td>trigger_task</td>
<td>task-launcher-cloudfoundry</td>
<td></td>
</tr>
<tr>
<td>twitterstream</td>
<td>task-launcher-local</td>
<td></td>
</tr>
<tr>
<td></td>
<td>task-launcher-yarn</td>
<td></td>
</tr>
<tr>
<td></td>
<td>tcp</td>
<td></td>
</tr>
<tr>
<td></td>
<td>throughput</td>
<td></td>
</tr>
<tr>
<td></td>
<td>websocket</td>
<td></td>
</tr>
</tbody>
</table>
with parameters:

```bash
twitterstream
  --consumerKey=<CONSUMER_KEY>
  --consumerSecret=<CONSUMER_SECRET>
  --accessToken=<ACCESS_TOKEN>
  --accessTokenSecret=<ACCESS_TOKEN_SECRET>
| log
```

with explicit input channel & analytics:

```bash
:tweets.twitterstream >
field-value-counter
  --fieldName=lang --name=language
```

with SpEL expression and explicit output type

```bash
:tweets.twitterstream >
filter
  --expression=$.lang=='en'
  --outputType=application/json
```
OUR SAMPLE APPLICATION: WORLD MOOD

https://github.com/adersberger/spring-cloud-dataflow-samples
DEVELOPING CUSTOM APPS: THE VERY BEGINNING

https://start.spring.io

SPRING INITIALIZR bootstrap your application now

Generate a Gradle Project with Spring Boot 1.5.3

Project Metadata
Artifact coordinates
Group
dedu.qaware
Artifact
tweetalyzer

Dependencies
Add Spring Boot Starters and dependencies to your application
Search for dependencies
Web, Security, JPA, Actuator, Devtools...

Selected Dependencies
Stream Kafka

Generate Project
@SpringBootApplication
@EnableBinding(Source.class)
public class TwitterIngester {

    private Iterator<String> lines;

    @Bean
    @InboundChannelAdapter(value = Source.OUTPUT,
        poller = @Poller(fixedDelay = "200", maxMessagesPerPoll = "1"))
    public MessageSource<String> twitterMessageSource() {
        return () -> new GenericMessage<>(emitTweet());
    }

    private String emitTweet() {
        if (lines == null || !lines.hasNext()) lines = readTweets();
        return lines.next();
    }

    private Iterator<String> readTweets() {
        //...
    }
}
@RunWith(SpringRunner.class)
@SpringBootTest(webEnvironment= SpringBootTest.WebEnvironment.RANDOM_PORT)
public class TwitterIngesterTest {

    @Autowired
    private Source source;
    @Autowired
    private MessageCollector collector;

    @Test
    public void tweetIngestionTest() throws InterruptedException {
        for (int i = 0; i < 100; i++) {
            Message<String> message = (Message<String>) collector.forChannel(source.output()).take();
            assert (message.getPayload().length() > 0);
        }
    }
}
@SpringBootApplication
@EnableBinding(Processor.class)
public class TweetSentimentProcessor {
    @Autowired
    StanfordNLP nlp;

    @StreamListener(Processor.INPUT) //input channel with default name
    @SendTo(Processor.OUTPUT) //output channel with default name
    public int analyzeSentiment(String tweet){
        return TupleBuilder.tuple().of("mood", findSentiment(tweet));
    }

    public int findSentiment(String tweet) {
        int mainSentiment = 0;
        if (tweet != null && tweet.length() > 0) {
            int longest = 0;
            Annotation annotation = nlp.process(tweet);
            for (CoreMap sentence : annotation.get(CoreAnnotations.SentencesAnnotation.class)) {
                Tree tree = sentence.get(SentimentCoreAnnotations.SentimentAnnotatedTree.class);
                int sentiment = RNNCoreAnnotations.getPredictedClass(tree);
                String partText = sentence.toString();
                if (partText.length() > longest) {
                    mainSentiment = sentiment;
                    longest = partText.length();
                }
            }
            return mainSentiment;
        }
        return mainSentiment;
    }
}
@RunWith(SpringRunner.class)
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
public class TweetSentimentProcessorTest {

    @Autowired
    private Processor processor;

    @Autowired
    private MessageCollector collector;

    @Autowired
    private TweetSentimentProcessor sentimentProcessor;

    @Test
    public void testAnalysis() {
        checkFor("I hate everybody around me!");
        checkFor("The world is lovely");
        checkFor("I f***ing hate everybody around me. They're from hell");
        checkFor("Sunny day today!");
    }

    private void checkFor(String msg) {
        processor.input().send(new GenericMessage<>(msg));
        assertThat(
            collector.forChannel(processor.output()),
            receivesPayloadThat(
                equalTo(TupleBuilder.tuple().of("mood", sentimentProcessor.findSentiment(msg))));
    }
}
DEVELOPING THE STREAM DEFINITIONS WITH FLO

Streams
Create a stream using text based input or the visual editor.

```
1. main.time | transform --expression=payload.toUpperCase() | log
2. tap1=main.time > filter --expression=payload.contains('abc') | log
3. tap2=main.transform > scriptable-transform --script="payload":"payload" --language=groovy | log
4. tap3=tap2.scriptable-transform > filter --expression=payload.contains('special') | log
```
RUNNING IT LOCAL

```
$ redis-server &
$ zookeeper-server-start.sh ./config/zookeeper.properties &
$ kafka-server-start.sh ./config/server.properties &
$ java -jar spring-cloud-dataflow-server-local-1.2.0.RELEASE.jar &
$ java -jar ../spring-cloud-dataflow-shell-1.2.0.RELEASE.jar

dataflow:> app import -uri [1]

dataflow:> app register --name tweetsentimentalyzer --type processor --uri file:///libs/worldmoodindex-0.0.2-SNAPSHOT.jar

dataflow:> stream create tweets-ingestion --definition "twitterstream --consumerKey=A --consumerSecret=B --accessToken=C --accessTokenSecret=D | filter —expression=#jsonPath(payload,'$.lang')=='en' | log" —deploy

dataflow:> stream create tweets-analyzer --definition "tweets-ingestion.filter > tweetsentimentalyzer | field-value-counter --fieldName=mood -name=Mood"

dataflow:> stream deploy tweets-analyzer —properties
"deployer.tweetsentimentalyzer.memory=1024m,deployer.tweetsentimentalyzer.count=8,
app.transform.producer.partitionKeyExpression=payload.id"

```
SCALE
LIKE A BOSS
$ git clone https://github.com/spring-cloud/spring-cloud-dataflow-server-kubernetes

$ kubectl create -f src/etc/kubernetes/kafka-zk-controller.yml
$ kubectl create -f src/etc/kubernetes/kafka-zk-service.yml
$ kubectl create -f src/etc/kubernetes/kafka-controller.yml
$ kubectl create -f src/etc/kubernetes/mysql-controller.yml
$ kubectl create -f src/etc/kubernetes/mysql-service.yml
$ kubectl create -f src/etc/kubernetes/kafka-service.yml
$ kubectl create -f src/etc/kubernetes/redis-controller.yml
$ kubectl create -f src/etc/kubernetes/redis-service.yml
$ kubectl create -f src/etc/kubernetes/scdf-config-kafka.yml
$ kubectl create -f src/etc/kubernetes/scdf-secrets.yml
$ kubectl create -f src/etc/kubernetes/scdf-service.yml
$ kubectl create -f src/etc/kubernetes/scdf-controller.yml
$ kubectl get svc #lookup external ip “scdf” <IP>

$ java -jar ../spring-cloud-dataflow-shell-1.2.0.RELEASE.jar

dataflow:> dataflow config server --uri http://<IP>:9393

dataflow:> app import --uri [2]

dataflow:> app register --type processor --name tweetsentimentalyzer --uri docker:qaware/tweetsentimentalyzer-processor:latest

dataflow:> ...

LESSONS LEARNED
PRO
- specialized programming model -> efficient
- specialized execution environment -> efficient
- support for all types of data (big, fast, smart)

CON
- disjoint programming model (data processing <-> services)
- maybe a disjoint execution environment (data stack <-> service stack)

BEST USED
- further on: as default for {big,fast,smart} data processing
<table>
<thead>
<tr>
<th>PRO</th>
<th>CON</th>
</tr>
</thead>
<tbody>
<tr>
<td>coherent execution environment (runs on microservice stack)</td>
<td>has limitations on throughput (big &amp; fast data) due to less optimization (like data affinity, query optimizer, ...) and message-wise processing</td>
</tr>
<tr>
<td>coherent programming model with emphasis on separation of concerns</td>
<td>technology immature in certain parts (e.g. diagnosability)</td>
</tr>
<tr>
<td>bascialy supports all types of data (big, fast, smart)</td>
<td></td>
</tr>
</tbody>
</table>

**BEST USED FOR**
- hybrid applications of data processing, system integration, API, UI
- moderate throughput data applications with existing dev team
- Message by message processing
Thank you!

Questions?

josef.adersberger@qaware.de
@adersberger
https://github.com/adersberger/spring-cloud-dataflow-samples
BONUS SLIDES
MORE...

- Reactive programming
  ```java
  public Flux<String> transform(@Input("input") Flux<String> input) {
    return input.map(s -> s.toUpperCase());
  }
  ```
- Diagnosability
@EnableBinding(Sink::class)
@EnableConfigurationProperties(PostgresSinkProperties::class)
class PostgresSink {

    @Autowired
    lateinit var props: PostgresSinkProperties

    @StreamListener(Sink.INPUT)
    fun processTweet(message: String) {
        Database.connect(props.url, user = props.user, password = props.password, driver = "org.postgresql.Driver")
        transaction {
            SchemaUtils.create(Messages)
            Messages.insert {
                it[Messages.message] = message
            }
        }
    }

    object Messages : Table() {
        val id = integer("id").autoIncrement().primaryKey()
        val message = text("message")
    }
}
MICRO ANALYTICS SERVICES

Dashboard

Microservice

Microservice

...
BLUEPRINT ARCHITECTURE

DATASERVICES

Reactive Streams

Ingestion
Exposition

MESSAGING

Stream Processing

Digestion

DATA LAKE
THE SECRET OF BIG DATA PERFORMANCE

Rule 1: Be as close to the data as possible! (CPU cache > memory > local disk > network)

Rule 2: Reduce data volume as early as possible! (as long as you don’t sacrifice parallelization)

Rule 3: Parallelize as much as possible!

Rule 4: Premature diagnosability and optimization
### The Big Picture

![Diagram](http://cloud.spring.io/spring-cloud-dataflow)

<table>
<thead>
<tr>
<th>REST-APIs / Shell / DSL</th>
<th>Dashboard</th>
<th>Flo for Spring Cloud Data Flow</th>
<th>Spring Flo</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Spring Cloud Data Flow - Core</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Uses**

| **Spring Cloud Deployer - Service Provider Interface (SPI)** |

**Implements**

<table>
<thead>
<tr>
<th><strong>Spring Cloud Deployer Local</strong></th>
<th><strong>Spring Cloud Deployer Cloud Foundry</strong></th>
<th><strong>Spring Cloud Deployer Yarn</strong></th>
<th><strong>Spring Cloud Deployer Kubernetes</strong></th>
<th><strong>Spring Cloud Deployer Mesos</strong></th>
</tr>
</thead>
</table>

**Deploys**

| **Spring Cloud Stream App Starters** | **Spring Cloud Task App Starters** |

**Uses**

| **Spring Cloud Stream** | **Spring Cloud Task** |

| **Spring Integration** | **Spring Boot** | **Spring Batch** |

[http://cloud.spring.io/spring-cloud-dataflow](http://cloud.spring.io/spring-cloud-dataflow)
BASIC IDEA: BI-MODAL SOURCES AND SINKS

Sources

Processors

Sinks

READ FROM / WRITE TO: FILE, DATABASE, URL, ...

INGEST FROM / DIGEST TO: TWITTER, MQ, LOG, ...