

Sisteme Distribuite - Laborator 9

Microservicii cu Spring Cloud

Spring Cloud Data Flow

Data Flow reprezintă o componentă Spring Cloud utilizată pentru procesarea bazată pe microservicii a datelor în flux (eng. *stream*). Data Flow furnizează utilitare pentru crearea de topologii complexe de emiteri în flux (eng. *streaming*) și *pipeline*-uri de date.

Pipeline-urile de date constau în aplicații Spring Boot construite utilizând framework-uri diverse pentru microservicii, precum **Spring Cloud Stream** sau **Spring Cloud Task**.

Instalarea unui server local Data Flow

Pentru a putea utiliza Data Flow, trebuie descărcate și instalate câteva componente cu care se creează pipeline-uri de date. În continuare, veți executa comenzi din terminal și veți descărca artefacte JAR necesare pentru server-ul Data Flow. Se recomandă crearea în prealabil a unui folder separat, numit **DataFlow**, spre exemplu, pentru componentele descărcate, în care se vor executa comenzile „**wget** ...”:

```
mkdir DataFlow
cd DataFlow
```

Data Flow Server

Se preia artefactul gata împachetat din Maven Central, ce încapsulează server-ul pregătit pentru utilizare:

```
wget https://repo1.maven.org/maven2/org/springframework/cloud/spring-
cloud-dataflow-server/2.4.2.RELEASE/spring-cloud-dataflow-server-2.4.2
.RELEASE.jar
```

URL-ul către artefactul din Maven Central (ultima versiune disponibilă la momentul scrierii laboratorului):

<https://mvnrepository.com/artifact/org.springframework.cloud/spring-cloud-dataflow-server/2.4.2.RELEASE>

Data Flow Shell

Shell-ul Data Flow expune o interfață la linia de comandă cu care utilizatorul poate gestiona setările pipeline-urilor și fluxurilor de date. Ca și în cazul componentei anterioare, se descarcă artefactul gata de utilizare, astfel:

```
wget https://repo1.maven.org/maven2/org/springframework/cloud/spring-
cloud-dataflow-shell/2.4.2.RELEASE/spring-cloud-dataflow-shell-
2.4.2.RELEASE.jar
```

URL-ul către artefactul din Maven Central (ultima versiune disponibilă la momentul scrierii laboratorului):

<https://mvnrepository.com/artifact/org.springframework.cloud/spring-cloud-dataflow-shell/2.4.2.RELEASE>

Spring Cloud Skipper

Skipper este un utilitar ce facilitează descoperirea și gestionarea ciclului de viață al aplicațiilor Spring Boot. Se poate folosi ca utilitar de sine stătător, pe mașina locală, sau poate fi

utilizat pe o platformă *cloud* pentru a-l integra cu *pipeline*-uri de tip **CI-CD** (*Continuous Integration - Continuous Deployment*).

Descărcați această componentă asemănător:

```
wget https://repo1.maven.org/maven2/org/springframework/cloud/spring-cloud-skipper-server/2.3.2.RELEASE/spring-cloud-skipper-server-2.3.2.RELEASE.jar
```

URL-ul către artefactul din Maven Central (ultima versiune disponibilă la momentul scrierii laboratorului):

<https://mvnrepository.com/artifact/org.springframework.cloud/spring-cloud-skipper-server/2.3.2.RELEASE>

Server RabbitMQ

Pentru aplicațiile din laborator, este necesar și un *broker* de mesaje (eng. *message broker*). Aplicațiile implicate în *pipeline*-ul de mesaje au nevoie de o entitate de tip *middleware* pentru a comunica. În exemplele din laborator, veți folosi **RabbitMQ**.

Instalați și porniți un server local de RabbitMQ conform instrucțiunilor din **laboratorul 5** de Sisteme Distribuite. Dacă îl aveți deja instalat, verificați funcționarea acestuia navigând la URL-ul <http://localhost:15672/>.

O alternativă este să folosiți un server containerizat de RabbitMQ, astfel:

```
docker run -d --hostname rabbitmq --name rabbitmq -p 15672:15672 -p 5672:5672 rabbitmq
```

Comanda de mai sus execută un server RabbitMQ folosind Docker, astfel încât nu veți avea nevoie de instalarea unei instanțe locale.

Pornire server Data Flow

Server-ul local Data Flow se pornește odată cu celelalte 2 componente menționate anterior. Executați următoarele comenzi din folder-ul unde ați descărcat componentele Skipper, Data Flow Server și Data Flow Shell, **fiecare într-un terminal separat**:

```
java -jar spring-cloud-skipper-server-2.3.2.RELEASE.jar
java -jar spring-cloud-dataflow-server-2.4.2.RELEASE.jar
java -jar spring-cloud-dataflow-shell-2.4.2.RELEASE.jar
```

```

cosmin@debian-gl553v: /mnt/hdd/PREDARE/SD/laboratoare/laborator 9/DataFlow
File Edit View Terminal Tabs Help
teService : Setting up ReleaseStateUpdateService
2020-04-09 18:19:26.572 INFO 25914 --- [main] o.s.s.concurrent.ThreadPoolT
askExecutor : Initializing ExecutorService 'skipperThreadPoolTaskExecutor'
2020-04-09 18:19:26.705 WARN 25914 --- [main] JpaBaseConfiguration$JpaWebC
onfiguration : spring.jpa.open-in-view is enabled by default. Therefore, database quer
ies may be performed during view rendering. Explicitly configure spring.jpa.open-in-vi
ew to disable this warning
2020-04-09 18:19:26.934 INFO 25914 --- [main] o.s.s.concurrent.ThreadPoolT
askExecutor : Initializing ExecutorService 'applicationTaskExecutor'
2020-04-09 18:19:28.133 INFO 25914 --- [main] o.s.b.w.embedded.tomcat.Tomc
atWebServer : Tomcat started on port(s): 7577 (http) with context path ''
2020-04-09 18:19:28.136 INFO 25914 --- [main] o.s.c.s.s.app.SkipperServerA
pplication : Started SkipperServerApplication in 12.961 seconds (JVM running for 13.
73)
2020-04-09 18:19:28.308 INFO 25914 --- [main] .c.s.s.s.RepositoryInitializ
ationService : Initializing repository database for name local
2020-04-09 18:19:28.499 INFO 25914 --- [main] .s.c.s.s.s.DeployerInitializ
ationService : Added 'Local' platform account 'default' into deployer repository.

cosmin@debian-gl553v: /mnt/hdd/PREDARE/SD/laboratoare/laborator 9/DataFlow
File Edit View Terminal Tabs Help
2020-04-09 18:19:21.191 WARN 25889 --- [main] JpaBaseConfiguration$JpaWeb
Configuration : spring.jpa.open-in-view is enabled by default. Therefore, database qu
eries may be performed during view rendering. Explicitly configure spring.jpa.open-in
-view to disable this warning
2020-04-09 18:19:21.409 INFO 25889 --- [main] o.s.s.concurrent.ThreadPool
TaskExecutor : Initializing ExecutorService 'applicationTaskExecutor'
2020-04-09 18:19:22.474 INFO 25889 --- [main] .s.c.DataFlowControllerAuto
Configuration : Skipper URI [http://localhost:7577/api]
2020-04-09 18:19:22.697 INFO 25889 --- [main] o.a.coyote.http11.Http11Nio
Protocol : Starting ProtocolHandler ["http-nio-9393"]
2020-04-09 18:19:22.745 INFO 25889 --- [main] o.s.b.w.embedded.tomcat.Tom
catWebServer : Tomcat started on port(s): 9393 (http) with context path ''
2020-04-09 18:19:22.751 INFO 25889 --- [main] o.s.c.d.s.s.DataFlowServerA
pplication : Started DataFlowServerApplication in 11.887 seconds (JVM running for
12.377)
2020-04-09 18:19:23.027 INFO 25889 --- [main] .s.c.d.s.s.LauncherInitiali
zationService : Added 'Local' platform account 'default' into Task Launcher repositor
y.

- dataflow 2.4.2.RELEASE
File Edit View Terminal Tabs Help
d [95824175/95824175]

cosmin@debian-gl553v: /mnt/hdd/PREDARE/SD/laboratoare/laborator 9/DataFlow$ java -jar
spring-cloud-dataflow-shell-2.4.2.RELEASE.jar

Spring Cloud
DataFlow >>>>

2.4.2.RELEASE

Welcome to the Spring Cloud Data Flow shell. For assistance hit TAB or type "help".
Successfully targeted http://localhost:9393/
dataflow>

```

Păstrați cele 3 sesiuni de terminal pornite. Vor fi utilizate în aplicațiile din laborator.

Accesarea Data Flow Dashboard

Panoul de control al componentei Data Flow se poate accesa la următorul URL:
<http://localhost:9393/dashboard>

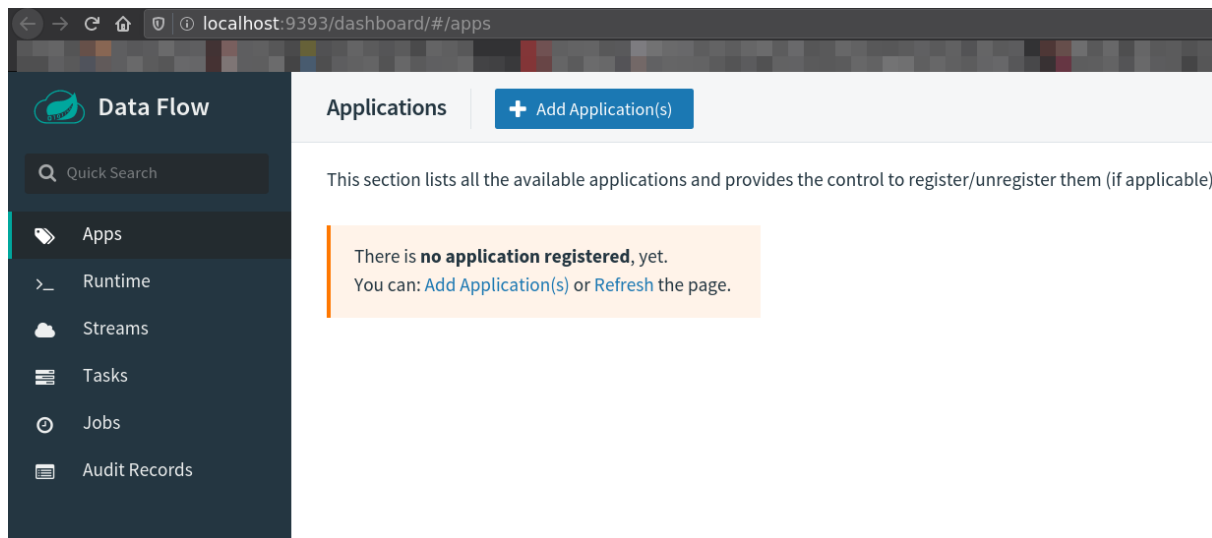


Figura 1 - Panoul de control Spring Data Flow

Veți avea nevoie de consola de administrare pentru vizualizarea fișierelor log și verificarea funcționării corecte a fluxurilor de date create.

Componentele unui pipeline de procesare a datelor în flux (stream processing pipeline)

Un *pipeline* de procesare a datelor în flux este alcătuit, conform specificațiilor Spring Data Flow, din 3 componente de bază:

- **Sursă** (eng. *Source*) - reprezintă generatorul de evenimente, sursa de date din pipeline, care produce date ce urmează a fi procesate
- **Procesor** (eng. *Processor*) - entitate care preia evenimente de la sursă și le procesează sub o anumită formă
- **Sink** - reprezintă destinația evenimentelor procesate; această entitate interceptează mesajele de la Procesor.



Figura 2 - Diagrama unui flux de date

Exemplul 1 - aplicație pipeline simplă cu Spring Cloud Data Flow

Pentru prima aplicație, se vor crea 3 microservicii Spring Boot corespunzătoare celor 3 tipuri de entități dintr-un pipeline Data Flow: o sursă de date, un procesor de date, și o entitate *sink*.

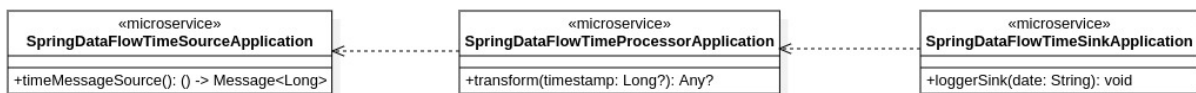


Figura 3 - Diagrama de clase

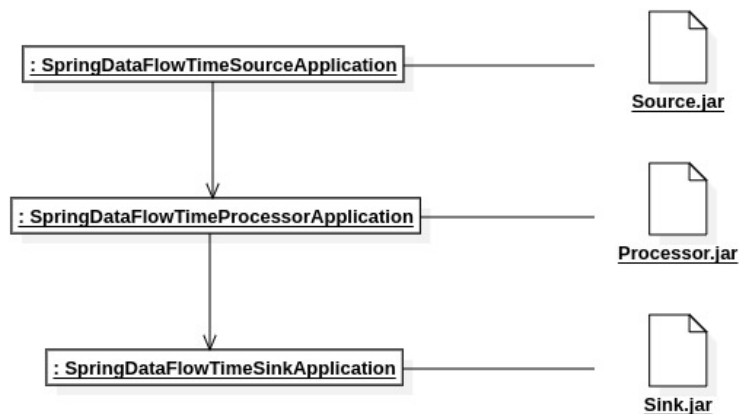


Figura 4 - Diagrama de obiecte

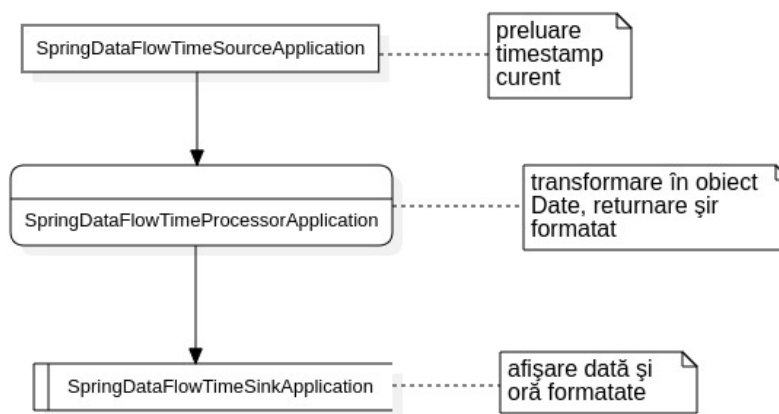


Figura 5 - Diagrama fluxului de date

Microserviciul sursă

Creați un proiect **Spring Boot** (folosind **Maven** sau **Gradle**), conform cu instrucțiunile din **laboratorul 3** de Sisteme Distribuite.

Adăugați dependența **Spring Cloud Starter Stream Rabbit** la proiect:

- **pentru Maven**: adăugați următoarea dependență în **pom.xml**:

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-stream-rabbit</artifactId>
  <version>3.0.3.RELEASE</version>
</dependency>
```

- **pentru Gradle**: adăugați următorul *compile group* în **build.gradle**:

```
compile group: 'org.springframework.cloud', name: 'spring-cloud-
starter-stream-rabbit', version: '3.0.3.RELEASE'
```

Creați un fișier sursă, denumit **Source.kt**, cu următorul conținut:

```
package com.sd.laborator

import org.springframework.boot.autoconfigure.SpringBootApplication
import org.springframework.boot.runApplication
import org.springframework.cloud.stream.annotation.EnableBinding
import org.springframework.cloud.stream.messaging.Source
import org.springframework.context.annotation.Bean
import
org.springframework.integration.annotation.InboundChannelAdapter
import org.springframework.integration.annotation.Poller
import org.springframework.messaging.Message
import org.springframework.messaging.support.MessageBuilder
import java.util.*

@EnableBinding(Source::class)
@SpringBootApplication
class SpringDataFlowTimeSourceApplication {
    @Bean
    @InboundChannelAdapter(value = Source.OUTPUT, poller =
[Poller(fixedDelay = "10000", maxMessagesPerPoll = "1")])
```

```

    fun timeMessageSource(): () -> Message<Long> {
        return { MessageBuilder.withPayload(Date().getTime()).build() }
    }
}

fun main(args: Array<String>) {
    runApplication<SpringDataFlowTimeSourceApplication>(*args)
}

```

Se observă că aplicația Spring Boot conține un *bean* care produce un mesaj o dată la 10 secunde. Mesajul conține un obiect *Timestamp* în corpul acestuia. Clasa de configurare a aplicației Spring este adnotată cu **@EnableBinding(Source::class)** pentru a fi legată de un *broker* (cel specificat între paranteze), și deci pentru a primi rolul de aplicație sursă în pipeline.

Microserviciul procesor

La fel ca în cazul microserviciului sursă, creați un alt proiect **Spring Boot** și adăugați dependența **Spring Cloud Starter Stream Rabbit** în maniera descrisă mai sus.

Adăugați un fișier sursă, denumit **Processor.kt**, cu următorul conținut:

```

package com.sd.laborator

import org.springframework.boot.autoconfigure.SpringBootApplication
import org.springframework.boot.runApplication
import org.springframework.cloud.stream.annotation.EnableBinding
import org.springframework.cloud.stream.messaging.Processor
import org.springframework.integration.annotation.Transformer
import java.text.DateFormat
import java.text.SimpleDateFormat

@EnableBinding(Processor::class)
@SpringBootApplication
class SpringDataFlowTimeProcessorApplication {
    @Transformer(inputChannel = Processor.INPUT, outputChannel = Processor.OUTPUT)
    fun transform(timestamp: Long?): Any? {
        val dateFormat: DateFormat = SimpleDateFormat("dd/MM/yyyy hh:mm:ss")
        return dateFormat.format(timestamp)
    }
}

fun main(args: Array<String>) {
    runApplication<SpringDataFlowTimeProcessorApplication>(*args)
}

```

Acest microserviciu are rolul de procesor de date (conform adnotării **@EnableBinding(Processor::class)**) și transformă valorile de tip *Timestamp* pe care le primește pe canalul de intrare în șiruri de caractere ce încapsulează data și ora indicată de *timestamp*.

Microserviciul sink

Creați și al 3-lea proiect Spring Boot în aceeași manieră, cu adăugarea dependenței

Spring Cloud Starter Stream Rabbit.

Adăugați un fișier sursă, denumit **Sink.kt**, cu următorul conținut:

```
package com.sd.laborator

import org.springframework.boot.autoconfigure.SpringBootApplication
import org.springframework.boot.runApplication
import org.springframework.cloud.stream.annotation.EnableBinding
import org.springframework.cloud.stream.annotation.StreamListener
import org.springframework.cloud.stream.messaging.Sink

@EnableBinding(Sink::class)
@SpringBootApplication
class SpringDataFlowTimeSinkApplication {
    @StreamListener(Sink.INPUT)
    fun loggerSink(date: String) {
        println("Am primit urmatorul mesaj: $date")
    }
}

fun main(args: Array<String>) {
    runApplication<SpringDataFlowTimeSinkApplication>(*args)
}
```

Conform adnotării **@EnableBinding(Sink::class)**, aplicate clasei de configurare Spring, rolul celui de-al 3-lea microserviciu este de entitate *sink*, care primește mesajele transformate de la procesor.

Bean-ul **loggerSink** „ascultă” pe canalul de intrare și primește rezultatul procesării din pipeline sub formă de șir de caractere și îl afișează la consolă.

Instalarea microserviciilor în server-ul Data Flow local

1. Împachetare microservicii sub formă de artefacte JAR

Folosiți *lifecycle*-ul **Maven package** sau *target*-ul **Gradle bootJar** pentru a împacheta cele 3 aplicații în artefacte JAR cu tot cu dependențele necesare.

Artefactele rezultate vor fi plasate în următoarele locații:

- pentru **Maven**: **target/<NUME_ARTEFACT>.jar**
- pentru **Gradle**: **build/libs/<NUME_ARTEFACT>.jar**

2. Înregistrarea microserviciilor sub formă de aplicații Data Flow

Reveniți la terminalul Spring Data Flow Shell deschis anterior.

Înregistrarea unui microserviciu ca aplicație Data Flow se face cu următoarea comandă:

```
app register --name <NUME_APLICAȚIE> --type <TIP_APLICAȚIE> --uri
file:///CALE_CĂTRE_ARTEFACTUL_JAR_AL_APLICAȚIEI_SURSĂ
```

Înlocuiți **CALE_CĂTRE_ARTEFACTUL_JAR_AL_APLICAȚIEI_SURSĂ** cu calea **absolută** către artefactul JAR generat în pasul de împachetare anterior.

Atenție: în cazul în care calea conține spații, înlocuiți-le cu șirul de caractere %20, întrucât calea absolută este trimisă ca parametru sub formă de URI.

Ca să preluați în mod facil calea completă a artefactului JAR, puteți deschide un terminal din IntelliJ în folder-ul ce conține acel artefact, apoi folosiți comanda **realpath**:

```
Terminal: Local x Local (2) x +
cosmin@debian-gl553v:/mnt/hdd/LUCRU/Spring Cloud/DataFlow Source/target$ realpath DataFlowSource-1.0-SNAPSHOT.jar
/mnt/hdd/LUCRU/Spring Cloud/DataFlow Source/target/DataFlowSource-1.0-SNAPSHOT.jar
```

Înlocuiți toate spațiile (dacă există) cu %20 și formați comanda completă de înregistrare pentru Data Flow Shell. Spre exemplu:

```
app register --name time-source --type source --uri
file:///mnt/hdd/LUCRU/Spring%20Cloud/DataFlow%20Source/target/Data-
FlowSource-1.0-SNAPSHOT.jar
```

Procedați asemănător cu celelalte 2 microservicii procesor și sink:

```
app register --name time-processor --type processor --uri
file:///mnt/hdd/LUCRU/Spring%20Cloud/DataFlow%20Processor/target/Da-
taFlowProcessor-1.0-SNAPSHOT.jar

app register --name logging-sink --type sink --uri
file:///mnt/hdd/LUCRU/Spring%20Cloud/DataFlow%20Sink/target/DataFlow-
Sink-1.0-SNAPSHOT.jar
```

Rezultatul comenzilor ar trebui să arate asemănător cu ce apare în captura următoare:

```
cosmin@debian-gl553v:/mnt/hdd/LUCRU/Spring Cloud/DataFlow$ java -jar spring-cloud-dataflow-
-shell-2.4.2.RELEASE.jar

Spring Cloud
DataFlow

2.4.2.RELEASE

Welcome to the Spring Cloud Data Flow shell. For assistance hit TAB or type "help".
Successfully targeted http://localhost:9393/
dataflow:>app register --name time-source --type source --uri file:///mnt/hdd/LUCRU/Sprin
g%20Cloud/DataFlow%20Source/target/DataFlowSource-1.0-SNAPSHOT.jar
Successfully registered application 'source:time-source'
dataflow:>app register --name time-processor --type processor --uri file:///mnt/hdd/LUCRU
/Spring%20Cloud/DataFlow%20Processor/target/DataFlowProcessor-1.0-SNAPSHOT.jar
Successfully registered application 'processor:time-processor'
dataflow:>app register --name logging-sink --type sink --uri file:///mnt/hdd/LUCRU/Spring
%20Cloud/DataFlow%20Sink/target/DataFlowSink-1.0-SNAPSHOT.jar
Successfully registered application 'sink:logging-sink'
dataflow:>
```

Figura 6 - Înregistrare cu succes a entităților din pipeline

Dacă este necesar, pentru a șterge o aplicație înregistrată pe serverul Data Flow, se poate folosi comanda:

```
app unregister --name <NUME_APLICAȚIE> --type <TIP_APLICAȚIE>
```

Crearea fluxului de date din entitățile înregistrate în Data Flow

Se creează fluxul de date conform diagramei din **Figura 5**, utilizând următoarea comandă în Data Flow Shell:


```
stream create --name time-to-log --definition 'time-source | time-processor | logging-sink'
```

Observați sintaxa specifică a definiției fluxului de date (parametrul `--definition`). Aceasta provine de la sintaxa *pipe*-urilor UNIX, spre exemplu o comandă de tipul:

```
ls -l | grep abc | wc -l
```

Așa cum în *shell*-ul Linux, comanda de sus creează un *pipeline* de 3 procese, conectând fluxul de ieșire al unuia la fluxul de intrare al următorului, așa se comportă și definiția fluxului de date din Data Flow:

```
time-source | time-processor | logging-sink
```

Canalul de ieșire al entității **time-source** este conectat la canalul de intrare al entității **time-processor**, respectiv canalul de ieșire al **time-processor** este conectat la canalul de intrare al **logging-sink**.

Odată ce fluxul de date a fost creat, acesta trebuie instalat (*deployed*) utilizând comanda următoare:

```
stream deploy --name time-to-log
```

```
dataflow:>stream create --name time-to-log --definition 'time-source | time-processor | logging-sink'
Created new stream 'time-to-log'
dataflow:>stream deploy --name time-to-log
Deployment request has been sent for stream 'time-to-log'
dataflow:>
```

Figura 7 - Instalarea fluxului de date

La nevoie, pentru a dezinstala un flux de date, folosiți comanda:

```
stream undeploy --name <NUME_STREAM>
```

În acest moment, aveți *pipeline*-ul instalat în server-ul local Data Flow, iar entitatea **Sursă** va începe imediat să emită date.

Verificarea funcționării fluxului de date

Pentru a verifica buna funcționare a fluxului de date configurat anterior, puteți folosi panoul grafic de configurare pus la dispoziție la URL-ul: <http://localhost:9393/dashboard>

Accesați secțiunea **Apps** din meniul din stânga și interfața vă afișează lista de aplicații instalate anterior. Veți observa cele 3 entități pe care le-ați instalat pe server:

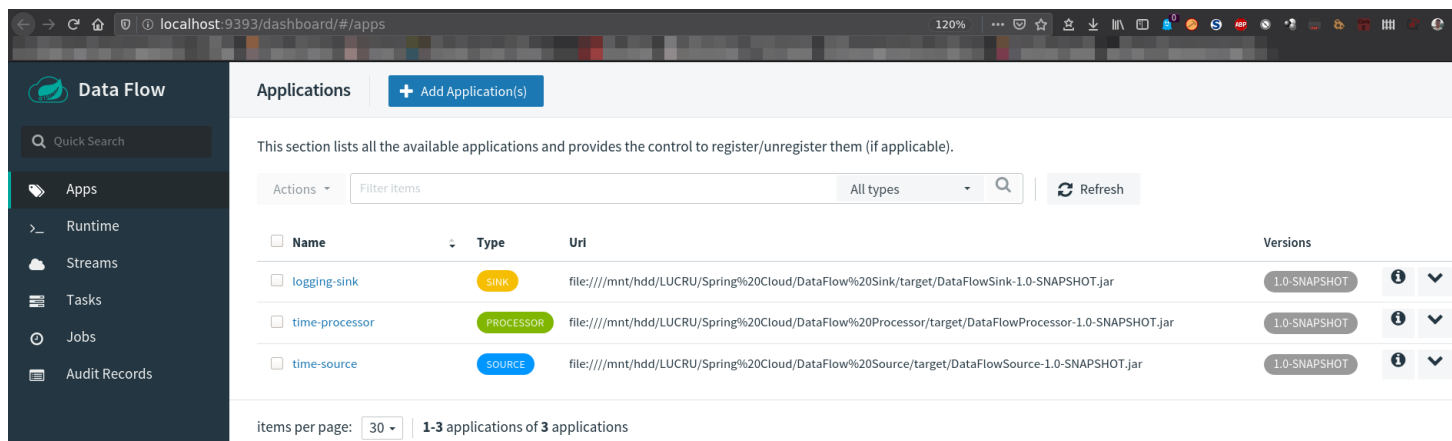


Figura 8 - Vizualizarea listei de aplicații Data Flow

Secțiunea „**Runtime**” vă afișează starea de funcționare a aplicațiilor. Dacă sunt marcate cu roșu, în starea „**Failed**”, atunci verificați log-urile pentru eventualele probleme apărute în cod sau la instalare (se explică mai jos).

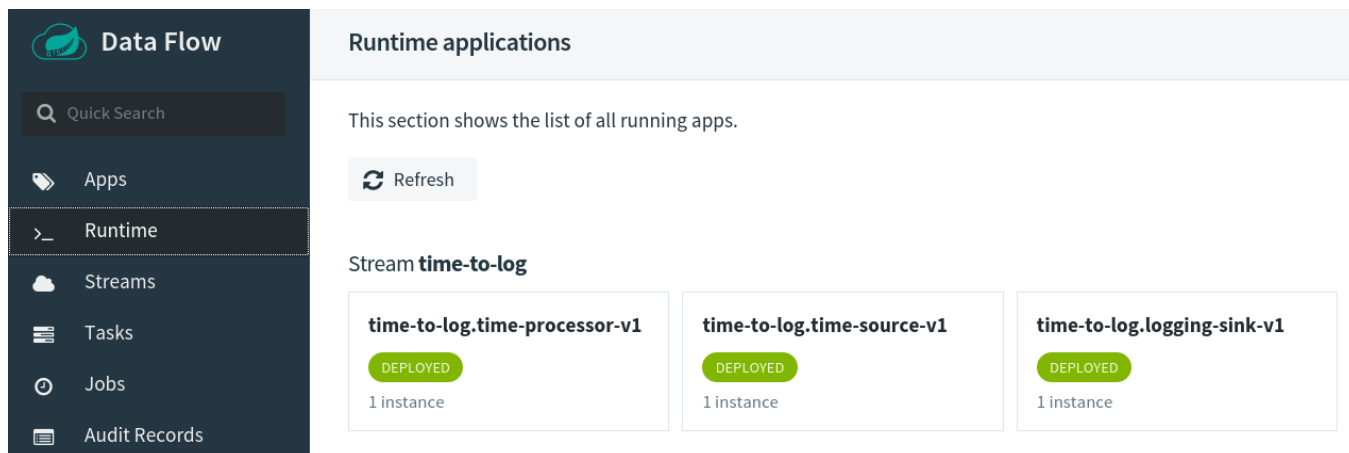


Figura 9 - Verificarea stării de funcționare a aplicațiilor

În secțiunea „**Streams**” puteți vizualiza care sunt fluxurile de date configurate și instalate.

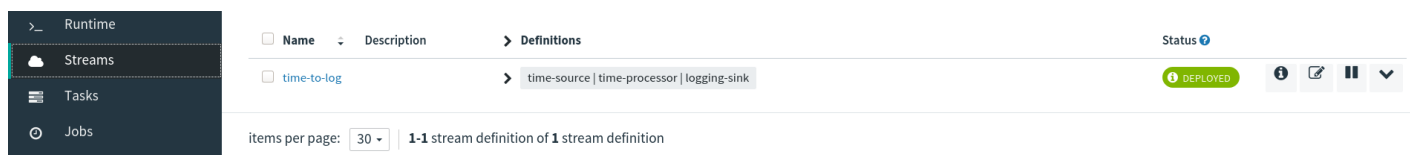
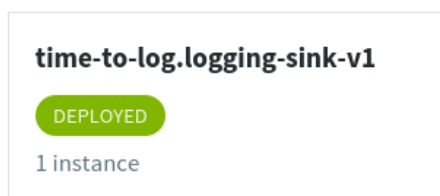


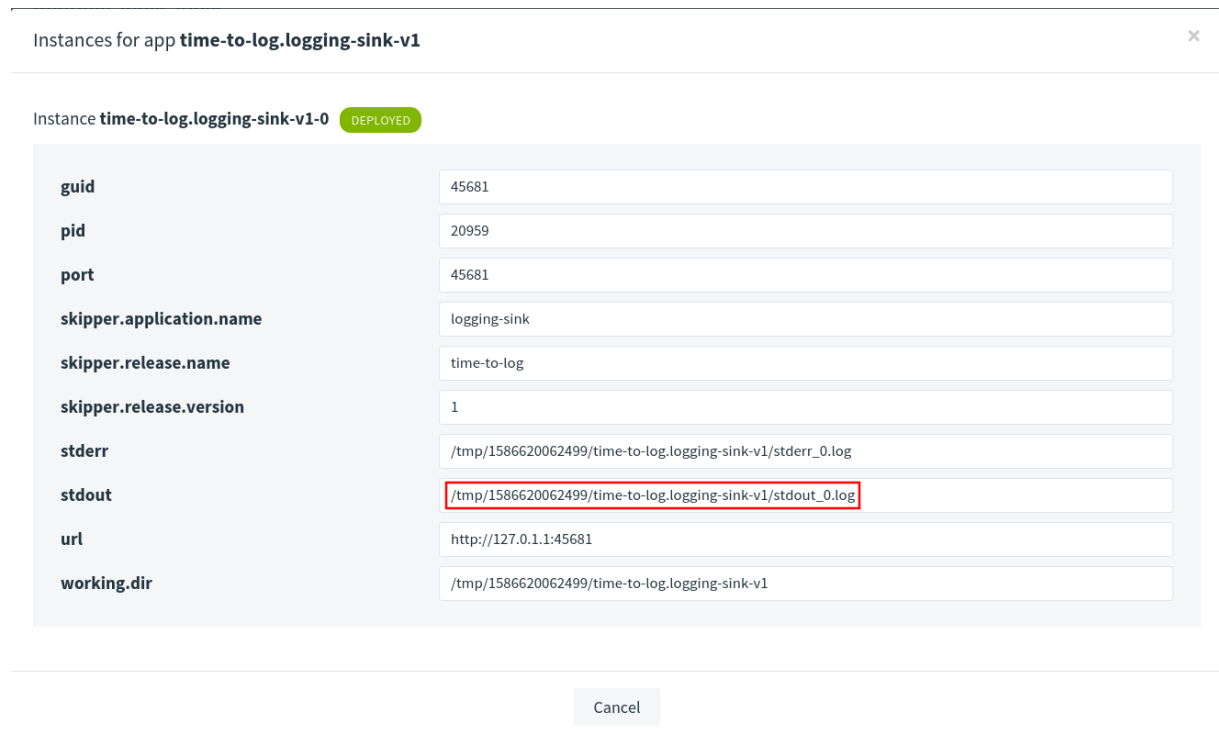
Figura 10 - Verificarea fluxurilor de date

Verificarea log-urilor

În secțiunea „**Runtime**”, apăsați pe entitatea de tip Sink (acolo ajung rezultatele procesării fluxului de date, deci acolo trebuie verificat *output*-ul aplicației).



Din panoul deschis deasupra interfeței, copiați calea către *log*-ul **stdout**:

Figura 11 - Preluarea *log*-ului unei aplicații Data Flow

Deschideți acel fișier log cu un editor de text și astfel puteți verifica ce anume a afișat aplicația la consolă (mai precis, ultima entitate din fluxul de date, *sink*-ul).

```

25 2020-04-11 19:15:04.555 INFO 24014 --- [main] o.s.s.c.ThreadPoolTaskScheduler : Initializing ExecutorService 'taskScheduler'
26 2020-04-11 19:15:04.819 INFO 24014 --- [main] o.s.c.s.m.DirectWithAttributesChannel : Channel 'application.input' has 1 subscriber(s).
27 2020-04-11 19:15:04.825 INFO 24014 --- [main] o.s.i.endpoint.EventDrivenConsumer : Adding
{logging-channel-adapter: org.springframework.integration.errorLogger} as a subscriber to the 'errorChannel' channel
28 2020-04-11 19:15:04.829 INFO 24014 --- [main] o.s.i.channel.PublishSubscribeChannel : Channel 'application.errorChannel' has 1 subscriber
(s).
29 2020-04-11 19:15:04.831 INFO 24014 --- [main] o.s.i.endpoint.EventDrivenConsumer : started bean
'org.springframework.integration.errorLogger'
30 2020-04-11 19:15:05.455 INFO 24014 --- [main] c.s.b.r.p.RabbitExchangeQueueProvisioner : declaring queue for inbound:
time-to-log.time-processor.time-to-log, bound to: time-to-log.time-processor
31 2020-04-11 19:15:05.462 INFO 24014 --- [main] o.s.a.r.c.CachingConnectionFactory : Attempting to connect to: [localhost:5672]
32 2020-04-11 19:15:05.613 INFO 24014 --- [main] o.s.a.r.c.CachingConnectionFactory : Created new connection:
rabbitConnectionFactory#2ddebff:0/SimpleConnection@5b7a7f33 [delegate=amqp://guest@127.0.1.1:5672/, localPort= 45188]
33 2020-04-11 19:15:05.765 INFO 24014 --- [main] o.s.c.stream.binder.BinderErrorChannel : Channel
'time-to-log.time-processor.time-to-log.errors' has 1 subscriber(s).
34 2020-04-11 19:15:05.765 INFO 24014 --- [main] o.s.c.stream.binder.BinderErrorChannel : Channel
'time-to-log.time-processor.time-to-log.errors' has 2 subscriber(s).
35 2020-04-11 19:15:05.839 INFO 24014 --- [main] o.s.i.a.i.AmqpInboundChannelAdapter : started bean
'inbound.time-to-log.time-processor.time-to-log'
36 2020-04-11 19:15:06.032 INFO 24014 --- [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port(s): 46760 (http) with
context path ''
37 2020-04-11 19:15:06.038 INFO 24014 --- [main] com.sd.laborator.SinkKt : Started SinkKt in 7.86 seconds (JVM running for
8.299)
38 Am primit urmatorul mesaj: 2020/04/11 07:15:20
39 Am primit urmatorul mesaj: 2020/04/11 07:15:20
40 Am primit urmatorul mesaj: 2020/04/11 07:15:20
41 Am primit urmatorul mesaj: 2020/04/11 07:15:20
42 Am primit urmatorul mesaj: 2020/04/11 07:15:20
43 Am primit urmatorul mesaj: 2020/04/11 07:15:20
44 Am primit urmatorul mesaj: 2020/04/11 07:16:20
45 Am primit urmatorul mesaj: 2020/04/11 07:16:20
46 Am primit urmatorul mesaj: 2020/04/11 07:16:20
47 Am primit urmatorul mesaj: 2020/04/11 07:16:20
48 Am primit urmatorul mesaj: 2020/04/11 07:16:20
49 Am primit urmatorul mesaj: 2020/04/11 07:16:20
50 Am primit urmatorul mesaj: 2020/04/11 07:17:20
51 Am primit urmatorul mesaj: 2020/04/11 07:17:20

```

Figura 12 - Verificarea log-ului entității *Sink*

Reîncărcarea aplicațiilor din fluxul Data Flow după modificări în cod

Dacă modificați codul microserviciilor componente din fluxul de date, reîmpachetați aplicația / aplicațiile modificate și, din consola Data Flow Shell, dezinstalați și reinstalați fluxul, astfel:

```
stream undeploy --name time-to-log
stream deploy --name time-to-log
```

```
dataflow:>stream undeploy --name time-to-log
Un-deployed stream 'time-to-log'
dataflow:>stream deploy --name time-to-log
Deployment request has been sent for stream 'time-to-log'
dataflow:>
```

Exemplul 2 - aplicație Data Flow cu proiectare DDD (Domain Driven Design)

Pentru exemplul 2, se va proiecta o aplicație conform principiilor *Domain Driven Design*. **Accentul se pune pe etapele de proiectare, implementarea efectivă va acoperi un set restrâns de cazuri de utilizare.**

Primul pas este identificarea fluxului de afaceri (eng. *business flow*), care modelează funcționarea unei companii: care sunt pașii necesari pentru a îndeplini o anumită cerință de afacere (eng. *business request*). Aplicația din laborator va modela în mod simplist funcționarea unei fabrici de produse Y, care primește comenzi de la clienți și livrează produsele acestora. În cazul în care cantitatea dorită de client nu este pe stoc, se cere fabricarea mai multor bucăți și aducerea lor pe stoc în depozit.

Diagrama de activitate - fluxul de afaceri

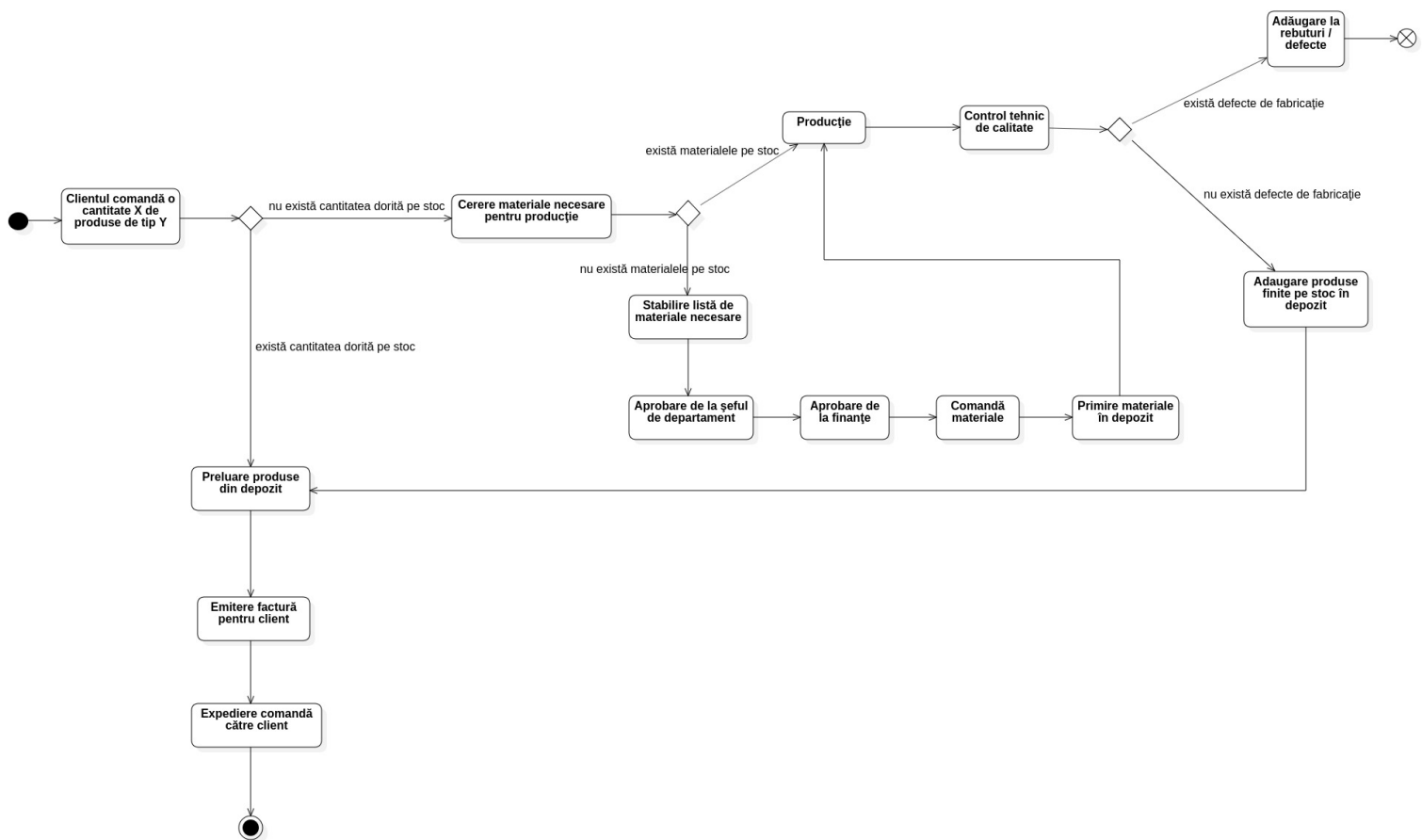


Figura 13 - Fluxul de afaceri

Diagrama de activitate - împărțirea pe departamente

O viziune mai clară asupra modului de funcționare a afacerii modelate se poate observa în diagrama următoare, cu activitățile împărțite în funcție de departamentul care le poate efectua.

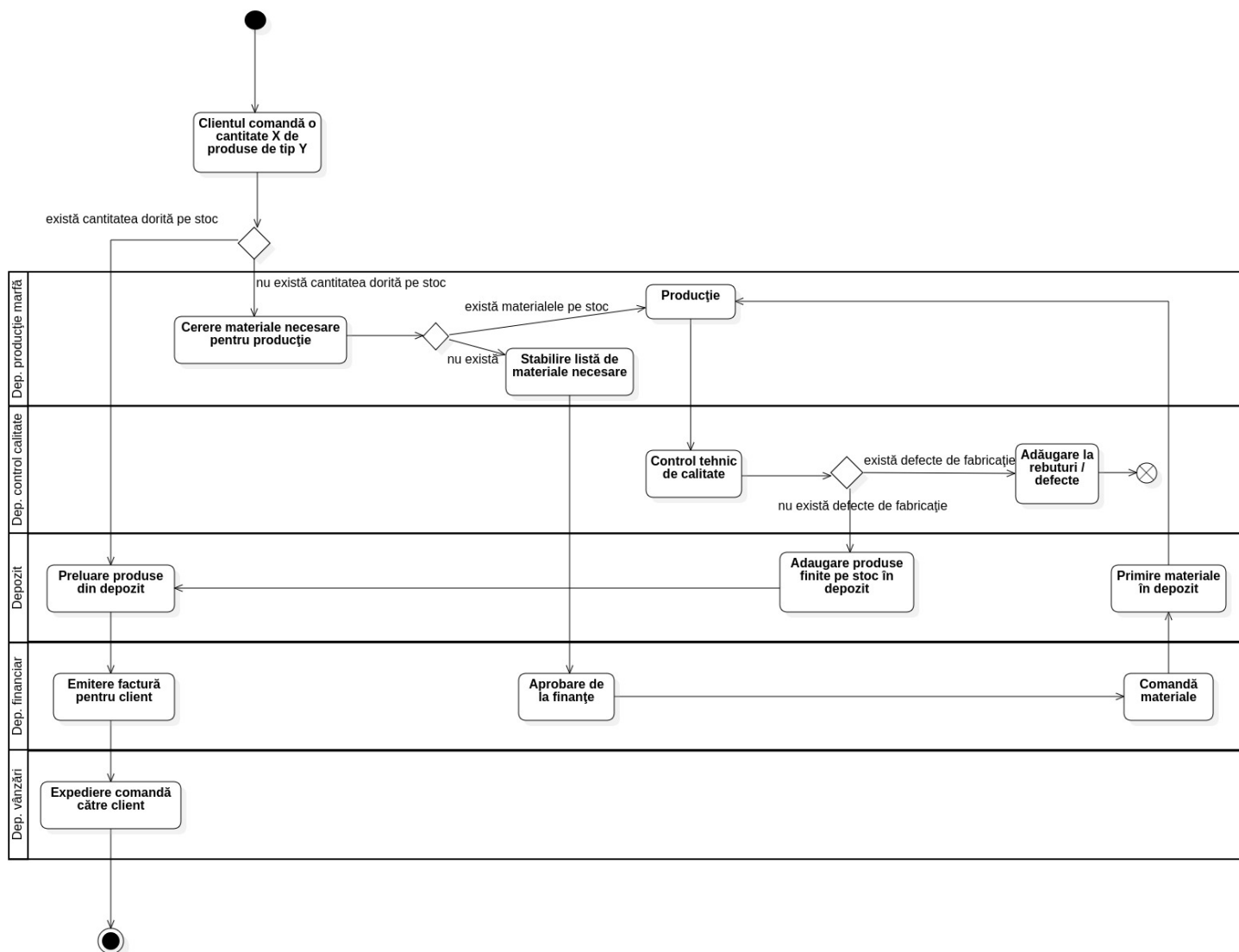


Figura 14 - Împărțirea pe departamente

Identificarea contextelor mărginite (bounded contexts)

Următorul pas este identificarea contextelor mărginite din fluxul de afacere. Acestea reprezintă linii conceptuale distinctive care definesc granițele ce separă contextele de alte componente ale sistemului.

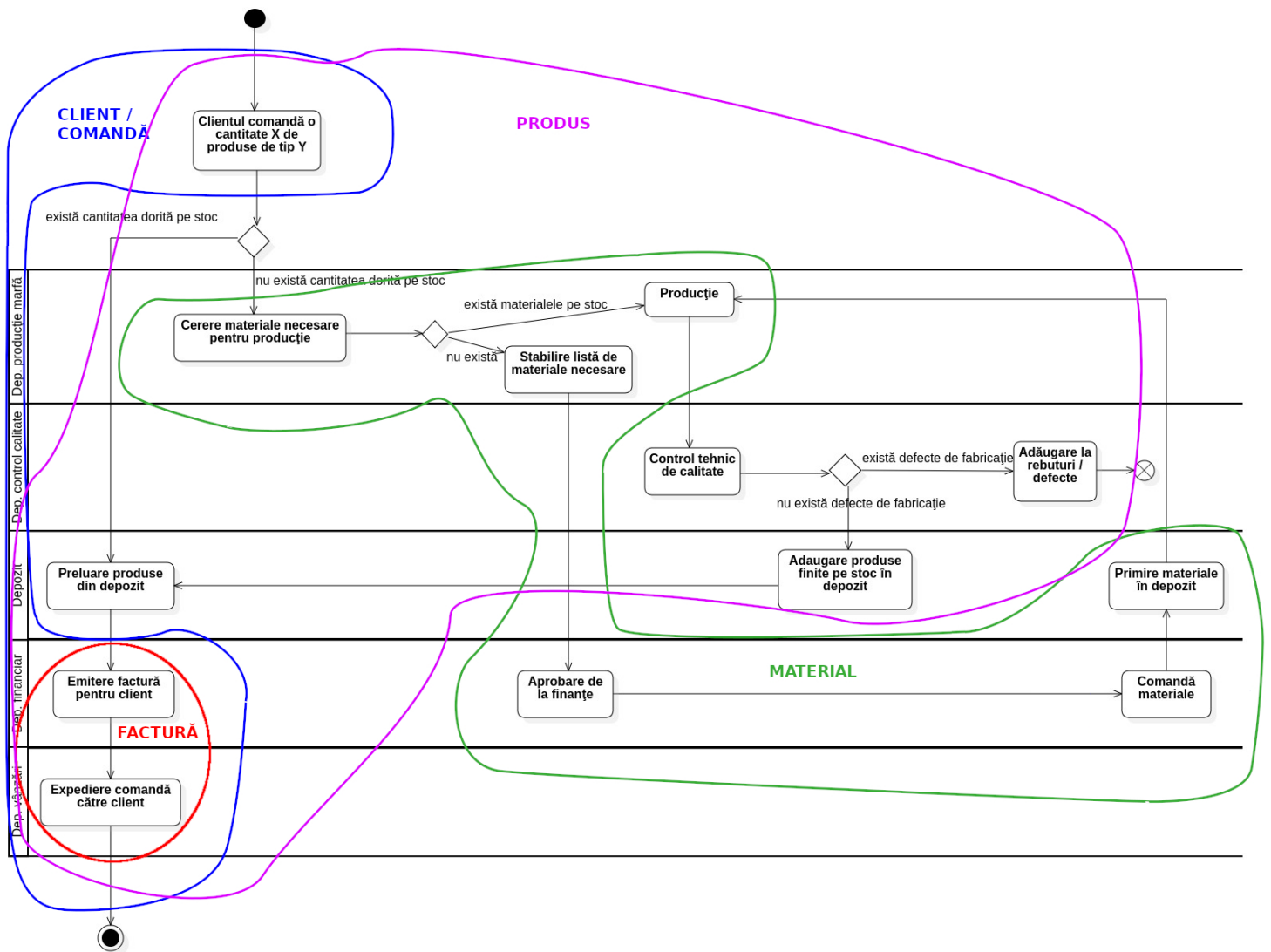


Figura 15 - Contextele mărginite

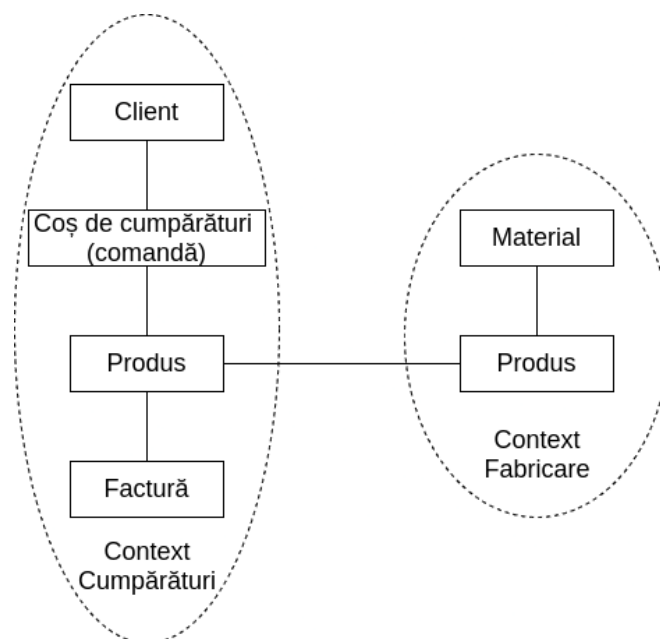


Figura 16: Maparea de context

Identificarea entităților

O entitate reprezintă un obiect mutabil: acesta își poate schimba proprietățile fără a-și modifica identitatea. De exemplu, un **Produs** este o entitate: produsul este unic și nu-și va schimba identitatea (ceea ce îl distinge în mod unic) odată ce aceasta este setată. Totuși, prețul, descrierea, și alte atribute specifice pot fi schimbate de câte ori este nevoie.

Entitățile care reies din fluxul de afacere modelat ar fi următoarele:

- CLIENT
- COMANDĂ
- PRODUS
- MATERIAL
- FACTURĂ

Proprietățile, respectiv relațiile între aceste entități se pot modela utilizând o **diagramă entitate-relație** (*Entity-Relationship Diagram*).

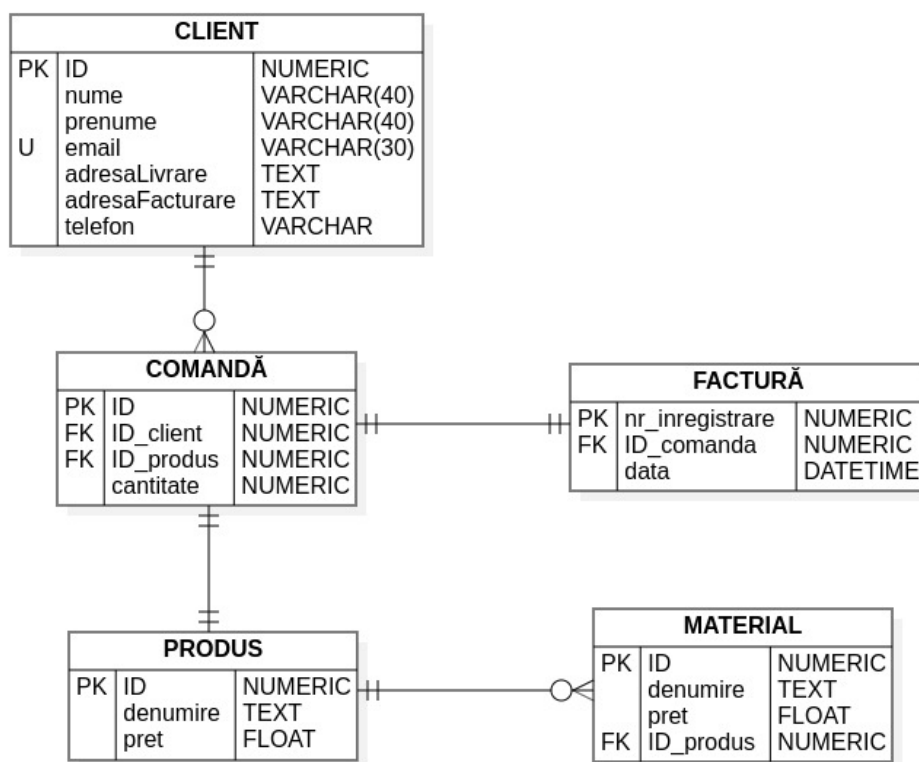


Figura 17 - Diagrama ERD

Identificarea serviciilor de domeniu (domain services)

În limbajul comun (eng. *ubiquitous language*), există situații în care acțiunile (cerințele de afacere) nu pot fi atribuite unei entități sau a unui obiect de valoare (eng. *value object*). Acele operațiuni pot fi asociate unui așa numit **serviciu de domeniu** (eng. *domain service*). Aceste servicii încapsulează logica de domeniu și concepte care nu pot fi modelate ca și entități sau obiecte de valoare. Serviciile de domeniu sunt responsabile pentru orchestrarea logicii de afacere (eng. *business logic*).

Pentru fiecare context mărginit, se specifică în continuare care sunt serviciile de domeniu identificate. Aceste servicii sunt specificate **în limbajul comun specific aceluia context** (*ubiquitous language*):

• PRODUS

- atunci când un client comandă un număr de produse, trebuie **cerut stocul** disponibil
- **identificarea materialelor** necesare fabricării unui produs
- **fabricarea** efectivă a produsului

- după ce un produs este fabricat, trebuie **verificat pentru** eventualele **defecte de fabricație**
- la detectarea unui defect, **produsul în cauză se adaugă la rebuturi**
- **adăugarea unui produs finit pe stoc**, dacă acesta nu are defecte de fabricație
- **emiterea facturii** pentru un anumit produs comandat de client
- după ce cantitatea de produse dorite de client este disponibilă, comanda **se expediază către adresa clientului**
- **MATERIAL**
 - **materialele sunt cerute de pe stoc** în cazul în care un produs nu există în cantitatea suficientă
 - dacă nu există suficiente materiale de producție, **se cere aprobarea achiziționării lor** de la departamentul financiar
 - **materialele necesare sunt comandate** de la furnizori de către departamentul financiar
 - odată ce comanda de **materiale** este onorată, acestea **se adaugă pe stoc** în depozit
 - **utilizarea materialelor de producție** pentru fabricarea unui produs
- **CLIENT**
 - clientul **se înregistrează în sistem** pentru a putea da comenzi
 - clientul poate **comanda o anumită cantitate de un anumit tip de produs**: „doresc X bucăți din produsul de tip Y”
 - clientului **îi este emisă o factură** pe comanda plasată, după ce produsele comandate sunt disponibile pentru livrare
 - clientului îi este **expediată comanda** pe care a plasat-o
- **COMANDĂ**
 - comanda este **înregistrată în sistem** după ce este plasată de client
 - pe baza unei comenzi înregistrate, **se emite o factură**, după ce produsele sunt pregătite pentru livrare
 - comanda **este expediată clientului** după ce este pregătită pentru livrare
- **FACTURĂ**
 - factura este emisă clientului după ce comanda sa este pregătită pentru livrare
 - comanda expediată **corespunde și include factura** aferentă pentru client

Proiectarea diagramei de clase

Conform serviciilor de domeniu identificate mai sus, se proiectează diagrama de clase care va scoate în evidență și modalitățile de implementare a cerințelor de afacere (*business requirements*).

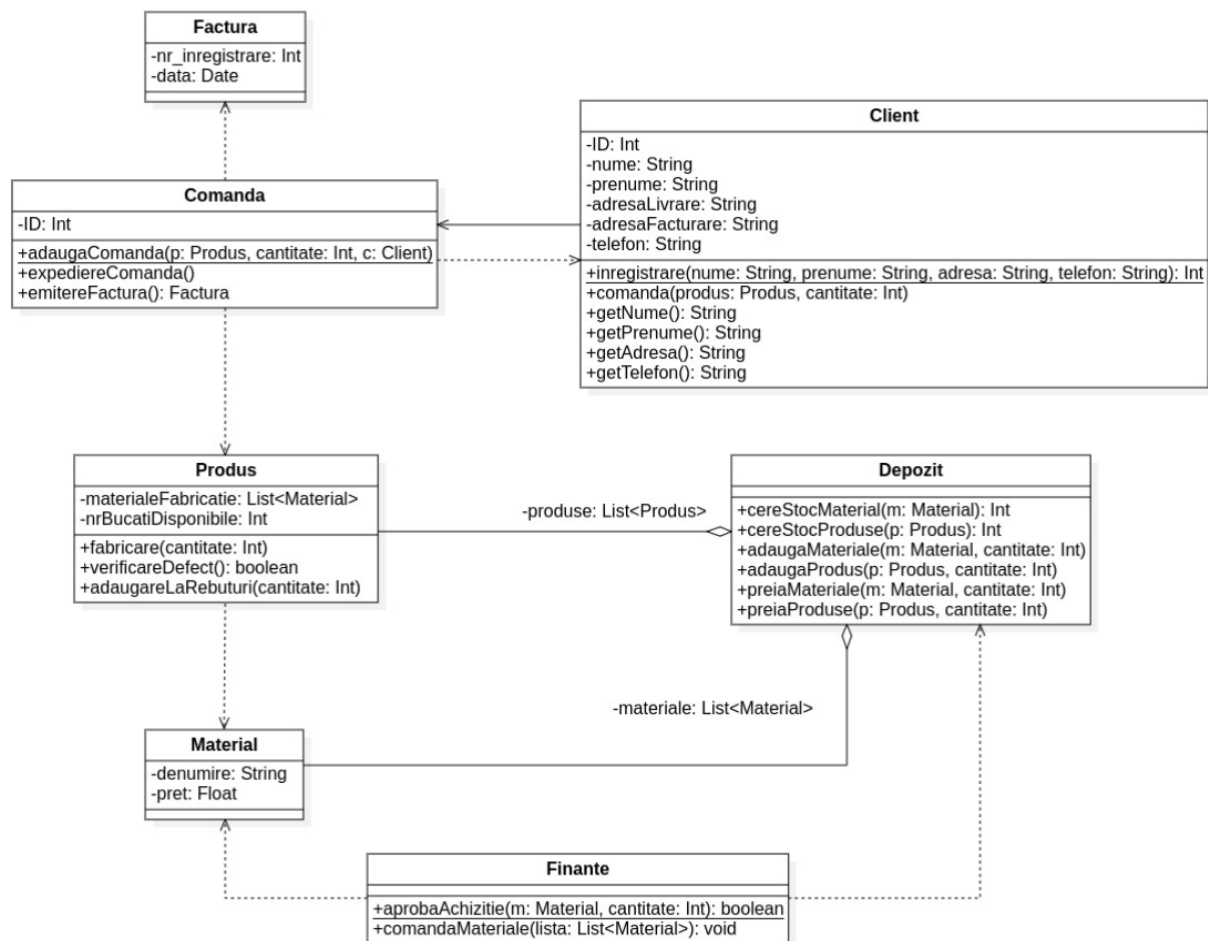


Figura 18 - Diagrama de clase

Reducerea cuplărilor și respectarea GDPR

Această etapă presupune asigurarea, pe cât posibil, a unei cuplări cât mai slabe între entitățile implicate în fluxul de afacere. De asemenea, din punct de vedere al **GDPR**, trebuie minimizată răspândirea informațiilor cu caracter personal prin fluxul de mesaje.

Spre exemplu, depozitul modelat în aplicația exemplu nu trebuie să știe pentru ce client pregătește și împachetează produsele pentru livrare. Pachetul va fi etichetat corespunzător înainte de a fi predat curierului, și abia atunci se poate accesa adresa de livrare și datele strict necesare ale clientului.

Asemănător, datele clientului nu au nicio relevanță și nu trebuie să ajungă în departamentul financiar, care doar comandă materiale necesare pentru producție, indiferent ce comandă a cărui client se datorează acestui fapt.

Din punct de vedere al identificării unui client sau a unei comenzi prin fluxul de mesaje, se pune problema încapsulării unui identificator unic, care este suficient pentru păstrarea consistenței informațiilor pe parcursul fluxului.

Proiectarea și implementarea microserviciilor

Microserviciile se proiectează și implementează corespunzător unui set restrâns de cazuri de utilizare, pentru simplitate. Așadar, implementarea din laborator se va conforma următorului graf de lucru:

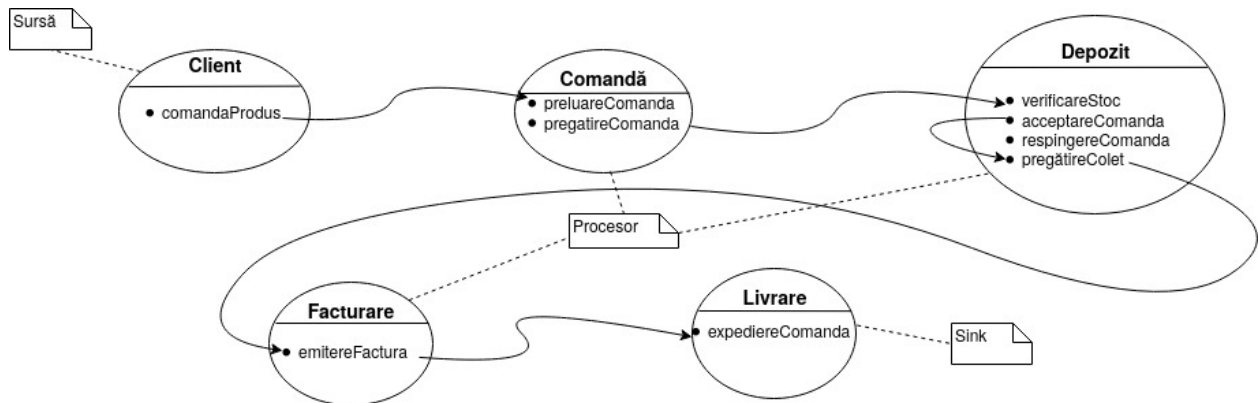


Figura 19 - Diagrama de microservicii

S-au marcat pe diagrama de mai sus și cele 3 tipuri de entități care intervin în *pipeline*-ul care se formează în consecință.

Fiecare microserviciu va fi creat într-un proiect Spring Boot separat, conform modelului din exemplul 1.

Client - microserviciu Sursă

```
package com.sd.laborator

import org.springframework.boot.autoconfigure.SpringBootApplication
import org.springframework.boot.runApplication
import org.springframework.cloud.stream.annotation.EnableBinding
import org.springframework.cloud.stream.messaging.Source
import org.springframework.context.annotation.Bean
import org.springframework.integration.annotation.InboundChannelAdapter
import org.springframework.integration.annotation.Poller
import org.springframework.messaging.Message
import org.springframework.messaging.support.MessageBuilder
import kotlin.random.Random

@EnableBinding(Source::class)
@SpringBootApplication
class ClientMicroservice {
    companion object {
        val listaProduce: List<String> = arrayListOf(
            "Masca protectie",
            "Vaccin anti-COVID-19",
            "Combinezon",
            "Manusa chirurgicala"
        )

        ///TODO - lista de produse sa fie preluata din baza de date /
        din fisier
    }

    @Bean
    @InboundChannelAdapter(value = Source.OUTPUT, poller =
    [Poller(fixedDelay = "10000", maxMessagesPerPoll = "1")])
    fun comandaProdus(): () -> Message<String> {
        return {
```

```

        val produsComandat = listaProduce[(0 until
listaProduce.size).shuffled()[0]]
        val cantitate: Int = Random.nextInt(1, 100)
        val identitateClient = "Popescu Ion"
        val adresaLivrare = "Codrii Vlasiei nr 14"

        ///TODO - datele clientului sa fie citite dinamic si
inregistrate in baza de date

        val mesaj =
"$identitateClient|$produsComandat|$cantitate|$adresaLivrare"
        MessageBuilder.withPayload(mesaj).build()
    }
}

fun main(args: Array<String>) {
    runApplication<ClientMicroservice>(*args)
}

```

Comandă - microserviciu Procesor

```

package com.sd.laborator

import org.springframework.boot.autoconfigure.SpringBootApplication
import org.springframework.boot.runApplication
import org.springframework.cloud.stream.annotation.EnableBinding
import org.springframework.cloud.stream.messaging.Processor
import org.springframework.integration.annotation.Transformer
import org.springframework.messaging.support.MessageBuilder
import java.text.DateFormat
import java.text.SimpleDateFormat
import kotlin.random.Random

@EnableBinding(Processor::class)
@SpringBootApplication
class ComandaMicroservice {
    private fun pregatireComanda(produs: String, cantitate: Int): Int
    {
        println("Se pregateste comanda $cantitate x \"$produs\"...")

        ///TODO - asignare numar de inregistrare
        ///TODO - inregistrare comanda in baza de date
        return Random.nextInt()
    }

    @Transformer(inputChannel = Processor.INPUT, outputChannel =
Processor.OUTPUT)
    fun preluareComanda(comanda: String?): String {
        val (identitateClient, produsComandat, cantitate,
adresaLivrare) = comanda!!.split("|")
        println("Am primit comanda urmatoare: ")
        println("$identitateClient | $produsComandat | $cantitate |

```

```

$adresaLivrare")

        val idComanda = pregatireComanda(produsComandat,
cantitate.toInt())

        ///TODO - in loc sa se trimita mesajul cu toate datele in
continuare, trebuie trimis doar ID-ul comenzii
        return "$comanda"
    }
}

fun main(args: Array<String>) {
    runApplication<ComandaMicroservice>(*args)
}

```

Depozit - microserviciu Procesor

```

package com.sd.laborator

import org.springframework.boot.autoconfigure.SpringBootApplication
import org.springframework.boot.runApplication
import org.springframework.cloud.stream.annotation.EnableBinding
import org.springframework.cloud.stream.messaging.Processor
import org.springframework.integration.annotation.Transformer
import org.springframework.messaging.support.MessageBuilder
import kotlin.random.Random

@EnableBinding(Processor::class)
@SpringBootApplication
class DepozitMicroservice {
    companion object {
        ///TODO - modelare stoc depozit (baza de date cu stocurile de
produse)
        val stocProduse: List<Pair<String, Int>> = mutableListOf(
            "Masca protectie" to 100,
            "Vaccin anti-COVID-19" to 20,
            "Combinezon" to 30,
            "Manusa chirurgicala" to 40
        )
    }

    private fun acceptareComanda(identificator: Int): String {
        println("Comanda cu identificatorul $identificator a fost
acceptata!")

        val produsDeExpediat = stocProduse[(0 until
stocProduse.size).shuffled()][0]]
        val cantitate = Random.nextInt(produsDeExpediat.second)

        return pregatireColet(produsDeExpediat.first, cantitate)
    }

    private fun respingereComanda(identificator: Int): String {

```

```

        println("Comanda cu identificatorul $identificator a fost
        respinsa! Stoc insuficient.")
        return "RESPINSA"
    }

    private fun verificareStoc(produs: String, cantitate: Int):
Boolean {
        ///TODO - verificare stoc produs
        return true
    }

    private fun pregatireColet(produs: String, cantitate: Int): String
{
        ///TODO - retragere produs de pe stoc in cantitatea
specificata
        println("Produsul $produs in cantitate de $cantitate buc. este
pregatit de livrare.")
        return "$produs|$cantitate"
    }

    @Transformer(inputChannel = Processor.INPUT, outputChannel =
Processor.OUTPUT)
    ///TODO - parametrul ar trebui sa fie doar numarul de inregistrare
al comenzii si atat
    fun procesareComanda(comanda: String?): String {
        val identificatorComanda = Random.nextInt()
        println("Procesez comanda cu identificatorul
$identificatorComanda...")

        ///TODO - procesare comanda in depozit
        val rezultatProcesareComanda: String = if (verificareStoc("",
100)) {
            acceptareComanda(identificatorComanda)
        } else {
            respingereComanda(identificatorComanda)
        }

        ///TODO - in loc sa se trimita mesajul cu toate datele in
continuare, trebuie trimis doar ID-ul comenzii
        return "$comanda"
    }
}

fun main(args: Array<String>) {
    runApplication<DepozitMicroservice>(*args)
}

```

Facturare - microserviciu Procesor

```

package com.sd.laborator

import org.springframework.boot.autoconfigure.SpringBootApplication
import org.springframework.boot.runApplication

```

```

import org.springframework.cloud.stream.annotation.EnableBinding
import org.springframework.cloud.stream.messaging.Processor
import org.springframework.integration.annotation.Transformer
import kotlin.random.Random

@EnableBinding(Processor::class)
@SpringBootApplication
class FacturareMicroservice {
    @Transformer(inputChannel = Processor.INPUT, outputChannel =
Processor.OUTPUT)
    ///TODO - parametrul ar trebui sa fie doar numarul de inregistrare
al comenzii si atat
    fun emitereFactura(comanda: String?): String {
        val (identitateClient, produsComandat, cantitate,
adresaLivrare) = comanda!!.split("|")
        println("Emit factura pentru comanda $comanda...")
        val nrFactura = Random.nextInt()
        println("S-a emis factura cu nr $nrFactura.")

        ///TODO - inregistrare factura in baza de date

        return "$comanda"
    }
}

fun main(args: Array<String>) {
    runApplication<FacturareMicroservice>(*args)
}

```

Livrare - microserviciu Sink

```

package com.sd.laborator

import org.springframework.boot.autoconfigure.SpringBootApplication
import org.springframework.boot.runApplication
import org.springframework.cloud.stream.annotation.EnableBinding
import org.springframework.cloud.stream.annotation.StreamListener
import org.springframework.cloud.stream.messaging.Sink

@EnableBinding(Sink::class)
@SpringBootApplication
class LivrareMicroservice {
    @StreamListener(Sink.INPUT)
    ///TODO - parametrul ar trebui sa fie doar numarul de inregistrare
al comenzii si atat
    fun expediereComanda(comanda: String) {
        println("S-a expediat urmatoarea comanda: $comanda")
    }
}

fun main(args: Array<String>) {
    runApplication<LivrareMicroservice>(*args)
}

```

Instalarea pipeline-ului în Data Flow

Înregistrați cele 5 aplicații în Data Flow conform tipului fiecăreia dintre ele. Apoi, configurați următorul pipeline:

```
client | comanda | depozit | facturare | livrare
```

Aplicații și teme

Temă de laborator

Completați aplicația a 2-a din laborator utilizând scheletul de cod dat. Pentru simplitate, la laborator folosiți fișiere text pe post de „bază de date”. Instalați aplicațiile și configurați *pipeline*-ul în Data Flow.

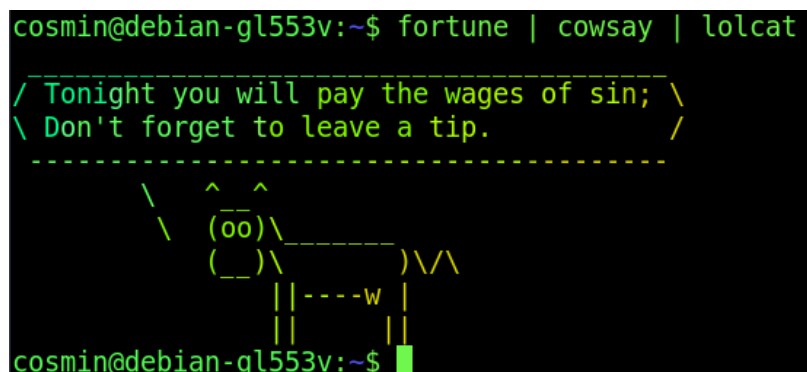
Teme pentru acasă

1. Implementați o aplicație care să poată executa comenzi Linux sub formă de *pipeline*, fără a folosi *pipe*-urile *shell*-ului, ci folosind microservicii conectate într-un *pipeline* Spring Data Flow.

Exemplu de comenzi conectate prin *pipeline*:

```
fortune | cowsay | lolcat
```

Exemplu de rezultat:



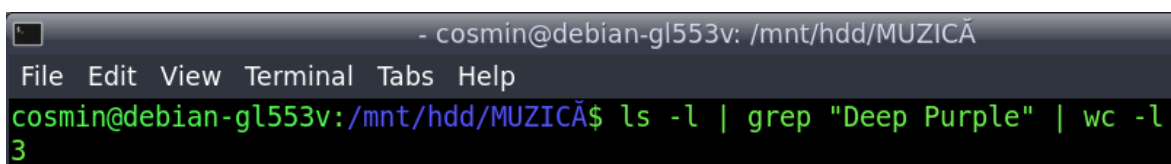
```
cosmin@debian-gl553v:~$ fortune | cowsay | lolcat
/ Tonight you will pay the wages of sin; \
\ Don't forget to leave a tip. /
-----
      \      ^__^
       (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
cosmin@debian-gl553v:~$
```

(Aceste comenzi sunt puse la dispoziție de pachetele corespunzătoare numelor lor, pe care le puteți instala utilizând comanda: **sudo apt install fortune cowsay lolcat**)

Un alt exemplu de pipeline Linux care afișează numărul de melodii de la formația Deep Purple dintr-un folder:

```
ls -l | grep "Deep Purple" | wc -l
```

Exemplu de rezultat:



```
- cosmin@debian-gl553v: /mnt/hdd/MUZICĂ
File Edit View Terminal Tabs Help
cosmin@debian-gl553v:/mnt/hdd/MUZICĂ$ ls -l | grep "Deep Purple" | wc -l
3
```

Sfaturi pentru rezolvare:

- microserviciul **sursă** poate citi o listă predefinită de comenzi scrise într-un fișier, câte una pe o linie, iar fiecare comandă va fi emisă în pipeline după un anumit timp.

Pentru simplitate, puteți considera comenzi care au **număr fix de pipe-uri** (simbolul „|”), stabilit de la bun început.

- **procesoarele de flux** preiau câte o parte din comanda Linux și execută partea „cea mai din stânga”, după care pasează mai departe rezultatul parțial, împreună cu restul comenzii care a mai rămas de executat către următorul procesor, ș.a.m.d.

Exemplu:

```
ls -l | grep "Deep Purple" | wc -l
```

Primul procesor din flux va executa `ls -l`, preia rezultatul parțial și trimite mai departe către următorul procesor acest rezultat, împreună cu ce a mai rămas din comandă:

```
grep "Deep Purple" | wc -l
```

Al doilea procesor primește acel rezultat și comanda rămasă de mai sus, aplică **grep "Deep Purple"** pe ce a primit, apoi rezultatul îl trimite mai departe către următorul procesor, împreună cu ce a mai rămas din comandă:

```
wc -l
```

Ultimul procesor încheie *pipeline*-ul, prin execuția ultimei părți (`wc -l`) pe rezultatul primit de la procesorul anterior.

- entitatea **sink** va fi consola (log-ul `stdout`), în care afișați rezultatele prelucrării comenzilor în pipeline

BONUS:

- luați în considerare cazul în care numărul de comenzi din *pipeline* este variabil. Deci, **procesoarele de flux** sunt în număr dinamic: nu se știe câte comenzi intermediare există în pipeline, așadar, în funcție de numărul de pipe-uri (simboluri „|”) din comandă, se vor instala (*deploy*) atâtea microservicii procesor câte sunt necesare (mai exact **numărul de pipe-uri** din comanda citită).
- procesoarele de flux să fie scalate orizontal folosind Docker. Puteți folosi aplicații predefinite Spring Data Flow dintr-un pachet Stream App Starter cu Docker: <https://cloud.spring.io/spring-cloud-stream-app-starters/>

Se poate importa un pachet de aplicații în Data Flow astfel:

Din Data Flow Shell, se folosește comanda:

```
app import --uri https://dataflow.spring.io/rabbitmq-docker-latest
```

```
dataflow:>app import --uri https://dataflow.spring.io/rabbitmq-docker-latest
Successfully registered 66 applications from [source.sftp, source.mqtt.metadata, source.cdc-debezium.metadata, sink.mqtt.metadata, source.file.metadata, processor.grpc.metadata, processor.tcp-client, source.s3.metadata, source.jms, source.ftp, processor.transform.metadata, source.time, sink.mqtt, sink.s3.metadata, processor.scriptable-transform, sink.log, source.load-generator, processor.transform, source.syslog, sink.websocket.metadata, source.loggregator.metadata, source.sftp-dataflow, source.s3, source.load-generator.metadata, processor.pmml.metadata, source.loggregator, source.tcp.metadata, processor.httpclient.metadata, sink.file.metadata, processor.object-detection.metadata, source.triggertask, source.twitterstream, source.gemfire-cq.metadata, processor.aggregator.metadata, sink.task-launcher-dataflow.metadata, source.mongodb, source.time.metadata, source.gemfire-cq, sink.counter.metadata, source.http, sink.tcp.metadata, sink.pgcopy.metadata, source.rabbit, source.jms.metadata, sink.gemfire.metadata, sink.cassandra.metadata, processor.tcp-client.metadata, processor.header-enricher, sink.throughput, processor.python-http, sink.mongodb, processor.twitter-sentiment, sink.log.metadata, processor.splitter, source.tcp, processor.python-jython.metadata, processor.image-recognition, source.trigger, source.mongodb.metadata, source.sftp-dataflow.metadata, processor.bridge, source.http.metadata, sink.ftp, source.rabbit.metadata, sink.jdbc, source.jdbc.metadata, source.mqtt, processor.pmml, sink.rabbit.metadata, processor.python-jython, sink.router.metadata, sink.cassandra, processor.filter.metadata, source.tcp-client.metadata, processor.header-enricher.metadata, processor.groovy-transform, source.ftp.metadata, sink.router, sink.redis-pubsub, source.tcp-client, processor.httpclient, sink.file, sink.websocket, source.syslog.metadata, sink.s3, source.cdc-debezium, sink.counter, sink.rabbit, processor.pose-estimation, processor.filter, source.trigger.metadata, source.mail.metadata, sink.pgcopy, processor.python-http.metadata, sink.jdbc.metadata, sink.k.ftp.metadata, processor.splitter.metadata, sink.sftp, processor.grpc, processor.groovy-filter.metadata, processor.twitter-sentiment.metadata, source.triggertask.metadata, sink.hdfs, sink.task-launcher-dataflow, processor.groovy-filter, sink.redis-pubsub.metadata, source.sftp.metadata, processor.image-recognition.metadata, processor.bridge.metadata, source.gemfire.metadata, processor.tensorflow, processor.counter, source.jdbc, processor.counter.metadata, processor.pose-estimation.metadata, sink.gemfire, source.gemfire, source.twitterstream.metadata, sink.hdfs.metadata, processor.tasklaunchrequest-transform, source.file, sink.mongodb.metadata, processor.tasklaunchrequest-transform.metadata, processor.scriptable-transform.metadata, processor.object-detection]
```

2. Să se modifice exemplul 2 din laborator astfel:

- a) realizarea unei interfețe GUI
- b) utilizarea unei baze de date relaționale (de ex. SQLite)
- c) să se trateze cazul în care o comandă nu poate fi onorată din cauza stocului lipsă: în acest caz, comanda va fi păstrată în contul clientului pentru reutilizare ulterioară.
- d) când se adună mai multe comenzi (minim 3) pentru același produs, depozitul să genereze o comandă pentru respectivul produs către producător (decă, un microserviciu

separat pentru producător și o funcție dedicată în acest sens în microserviciul **Depozit**).

Bibliografie

- [1]: Instalare Spring Data Flow - <https://dataflow.spring.io/docs/installation/local/manual/>
- [2]: Documentație Spring Data Flow - <https://dataflow.spring.io/docs/>
- [3]: Înregistrarea aplicațiilor flux în server-ul local de Spring Data Flow - <https://docs.spring.io/spring-cloud-dataflow-server-cloudfoundry/docs/1.0.0.M4/reference/html/spring-cloud-dataflow-register-apps.html>
- [4]: Procesarea fluxurilor de date utilizând Data Flow și RabbitMQ - <https://dataflow.spring.io/docs/stream-developer-guides/streams/data-flow-stream/>
- [5]: Fluxuri de date Spring Cloud Data Flow - <https://docs.spring.io/spring-cloud-dataflow/docs/2.5.0.BUILD-SNAPSHOT/reference/htmlsingle/#spring-cloud-dataflow-streams>
- [6]: Stream Pipeline DSL - <https://docs.spring.io/spring-cloud-dataflow/docs/current/reference/htmlsingle/#spring-cloud-dataflow-stream-intro-dsl>
- [7]: Architectural Patterns: Uncover Essential Patterns in the Most Indispensable Realm of Enterprise Architecture, Pethuru Raj Chelliah