

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 emitere 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**.

Atenție! Cei care lucrează de pe stațiile din laborator, săriți direct la Secțiunea “Server RabbitMQ”.

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/2.5.4.RELEASE/spring-cloud-dataflow-server-
2.5.4.RELEASE.jar
```

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

<https://mvnrepository.com/artifact/org.springframework.cloud/spring-cloud-dataflow-server/2.5.4.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/2.5.4.RELEASE/spring-cloud-dataflow-shell-
2.5.4.RELEASE.jar
```

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

<https://mvnrepository.com/artifact/org.springframework.cloud/spring-cloud-dataflow-shell/2.5.4.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/2.4.3.RELEASE/spring-cloud-skipper-server-2.4.3.RELEASE.jar
```

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

<https://mvnrepository.com/artifact/org.springframework.cloud/spring-cloud-skipper-server/2.4.3.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/>.

ALTERNATIVĂ: 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-2.4.3.RELEASE.jar  
java -jar spring-cloud-dataflow-server-2.5.4.RELEASE.jar  
java -jar spring-cloud-dataflow-shell-2.5.4.RELEASE.jar
```

Atenție: execuți ultima comandă DOAR după ce server-ul Data Flow și componenta Skipper au pornit complet! Execuți primele 2 comenzi mai întâi, așteptați să pornească server-ul, apoi execuți-o pe ultima, ca shell-ul să se conecteze corect la server.

```

student@debian:~/DataFlow$ java -jar spring-cloud-skipper-server-2.4.3.RELEASE.jar
2021-11-21 15:24:43.007 INFO 55436 --- [           main] o.s.c.s.s.a.ProfileApplicationListener : Setting property 'spring.cloud.kubernetes.enabled' to false
2021-11-21 15:24:43.007 INFO 55436 --- [           main] c.e.c.ConfigServicePropertySourceLocator : Fetching config from server at : http://localhost:8888
2021-11-21 15:24:44.999 INFO 55436 --- [           main] c.e.c.ConfigServicePropertySourceLocator : Connect to EC2 instance on url: http://localhost:8888 will be trying the next url if available
2021-11-21 15:24:45.009 WARN 55436 --- [           main] c.e.c.ConfigServicePropertySourceLocator : Could not locate PropertySource: I/O error on GET request for "http://localhost:8888/local": Connection refused (Connection refused); nested exception is java.net.ConnectException: Connection refused (Connection refused)
2021-11-21 15:24:45.006 INFO 55436 --- [           main] o.s.c.s.s.app.SkipperServerApplication : The following ports are active: local
2021-11-21 15:24:50.717 INFO 55436 --- [           main] o.s.cloud.context.scope.GenericScope : Bean definition for id=a7424f66-5eca-3a1c-910f-f0c6662f28aa
2021-11-21 15:24:52.325 INFO 55436 --- [           main] o.s.w.embedded.tomcat.TomcatWebServer : Tomcat initialized with port(s): 7577 (http)
2021-11-21 15:24:52.351 INFO 55436 --- [           main] o.apache.catalina.core.StandardService : Starting service [Tomcat]
2021-11-21 15:24:52.351 INFO 55436 --- [           main] org.apache.catalina.core.StandardEngine : Starting Servlet engine: [Apache Tomcat/9.0.34]
2021-11-21 15:24:52.499 INFO 55436 --- [           main] o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring embedded WebApplicationContext
2021-11-21 15:24:56.379 INFO 55436 --- [           main] com.zaxxer.hikari.HikariDataSource : HikariPool-1 [           main] student@debian:~/DataFlow$0x26
student@debian:~/DataFlow$ java -jar spring-cloud-dataflow-shell-2.5.4.RELEASE.jar
2021-11-21 15:24:56.887 INFO 55457 --- [           main] o.s.c.d.a.l.ProfileApplicationListener : Setting property 'spring.cloud.kubernetes.enabled' to false
2021-11-21 15:25:00.179 INFO 55457 --- [           main] o.s.c.d.a.l.ProfileApplicationListener : Setting property 'spring.cloud.kubernetes.enabled' to false.

Spring Cloud Data Flow Server (v2.5.4.RELEASE)
2021-11-21 15:25:00.265 INFO 55457 --- [           main] c.e.c.ConfigServicePropertySourceLocator : Fetching config from server at : http://localhost:8888
2021-11-21 15:25:00.515 INFO 55457 --- [           main] c.e.c.ConfigServicePropertySourceLocator : Connect Timeout Exception on Url http://localhost:8888 Will be trying the next url if available
2021-11-21 15:25:00.515 WARN 55457 --- [           main] c.e.c.ConfigServicePropertySourceLocator : Could not locate PropertySource: I/O error on GET request for "http://localhost:8888/spring-cloud-dataflow-server/local": Connection refused (Connection refused); nested exception is java.net.ConnectException: Connection refused (Connection refused).

```

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>

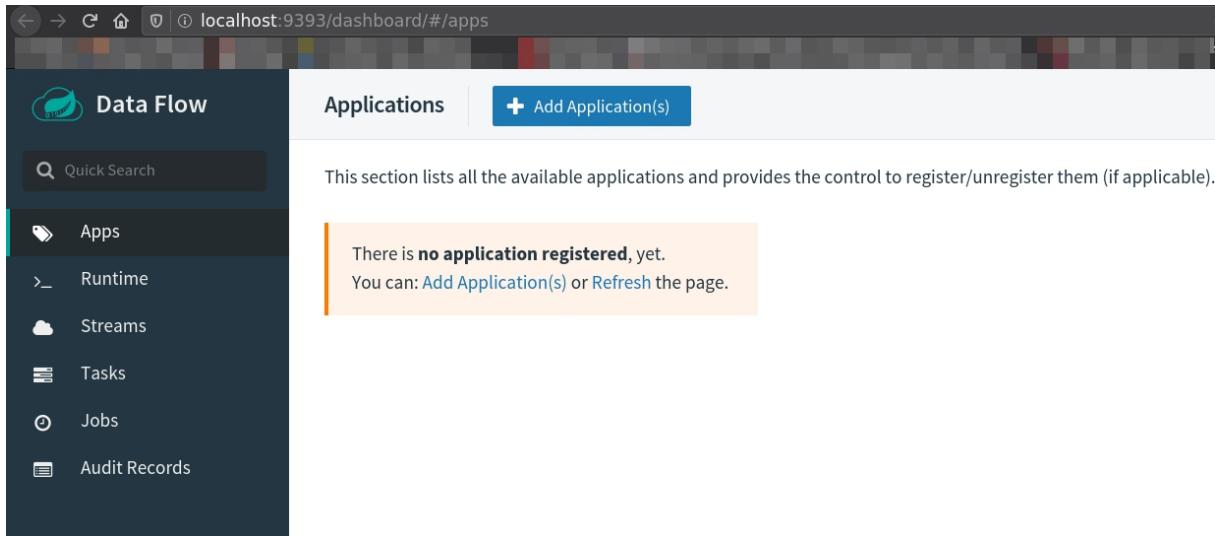


Figura 1 - Panoul de control Spring Data Flow

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.

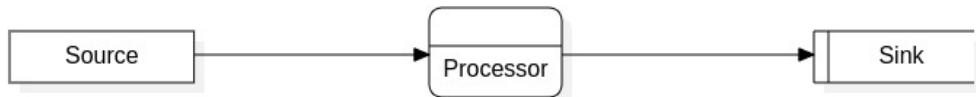


Figura 2 - Diagrama unui flux de date

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.

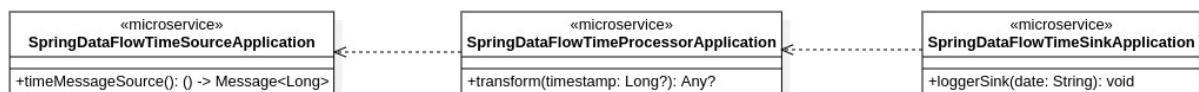


Figura 3 - Diagrama de clase

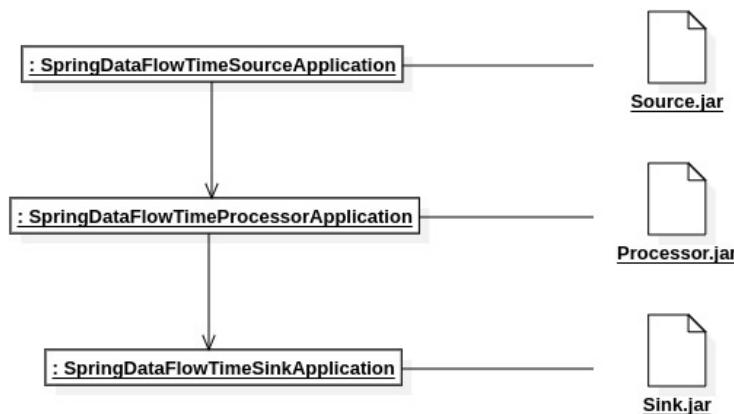


Figura 4 - Diagrama de obiecte

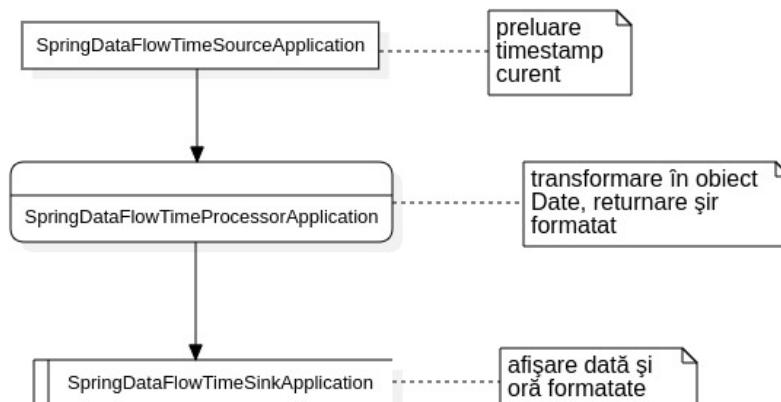


Figura 5 - Diagrama fluxului de date

Microserviciul sursă

Creați un proiect **Spring Boot** (folosind **Maven** sau **Gradle**), conform cu instrucțiunile din **laboratorul 3** de Sisteme Distribuite. **Nu aveți nevoie de dependența Spring Web (spring-boot-starter-web)**.

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.SpringApplication
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().time).build() }
    }
}

fun main(args: Array<String>) {
    SpringApplication.run(SpringDataFlowTimeSourceApplication::class.java, *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 anotată 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.SpringApplication
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 siruri 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.SpringApplication
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 sir 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 rezultante 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_APPLICATIE> --type <TIPO_APPLICATIE> --uri
file:///CALE_CĂTRE_ARTEFACTUL_JAR_AL_APPLICATIEI_SURSA
```

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

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

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

```
Terminal: Local (3) + ▾
student@debian:~/Documents/SD_Labs/Lab_9/SD_Laborator_09/exemplul 1/DataFlow Source/target$ realpath DataFlowSource-1.0-SNAPSHOT.jar
/home/student/Documents/SD_Labs/Lab_9/SD_Laborator_09/exemplul 1/DataFlow Source/target/DataFlowSource-1.0-SNAPSHOT.jar
student@debian:~/Documents/SD_Labs/Lab_9/SD_Laborator_09/exemplul 1/DataFlow Source/target$
```

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

```
app register --name time-source --type source --uri
file:///home/student/Documents/SD_Labs/Lab_9/SD_Laborator_09/exemplul %201/DataFlow%20Source/target/DataFlowSource-1.0-SNAPSHOT.jar
```

Procedați asemănător cu celelalte 2 microservicii procesor și *sink*:

```
app register --name time-processor --type processor --uri
file:///home/student/Documents/SD_Labs/Lab_9/SD_Laborator_09/exemplul %201/DataFlow%20Processor/target/DataFlowProcessor-1.0-SNAPSHOT.jar
```

```
app register --name logging-sink --type sink --uri  
file:///home/student/Documents/SD_Labs/Lab_9/SD_Laborator_09/exemplul  
%201/DataFlow%20Sink/target/DataFlowSink-1.0-SNAPSHOT.jar
```

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

```
student@debian:~/DataFlow$ java -jar spring-cloud-dataflow-shell-2.5.4.RELEASE.jar  
  
2.5.4.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:///home/student/Documents/SD_Labs/Lab_9/SD_Laborator_09/exemplul%201/DataFlow%20Source/target/DataFlowSource-1.0-SNAPSHOT.jar  
Successfully registered application 'source:time-source'  
dataflow:>app register --name time-processor --type processor --uri file:///home/student/Documents/SD_Labs/Lab_9/SD_Laborator_09/exemplul%201/DataFlow%20Processor/target/DataFlowProcessor-1.0-SNAPSHOT.jar  
Successfully registered application 'processor:time-processor'  
dataflow:>app register --name logging-sink --type sink --uri file:///home/student/Documents/SD_Labs/Lab_9/SD_Laborator_09/exemplul%201/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, aşa 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 de 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 emite date.

Verificarea funcționării fluxului de date

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

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

Name	Type	Uri	Versions
logging-sink	SINK	file:///home/student/Documents/SD_Labs/Lab_9/SD_Laborator_09/exemplul%201/DataFlow%20Sink/target/DataFlowSink-1.0-SNAPSHOT.jar	1.0-SNAPSHOT
time-processor	PROCESSOR	file:///home/student/Documents/SD_Labs/Lab_9/SD_Laborator_09/exemplul%201/DataFlow%20Processor/target/DataFlowProcessor-1.0-SNAPSHOT.jar	1.0-SNAPSHOT
time-source	SOURCE	file:///home/student/Documents/SD_Labs/Lab_9/SD_Laborator_09/exemplul%201/DataFlow%20Source/target/DataFlowSource-1.0-SNAPSHOT.jar	1.0-SNAPSHOT

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 logurile pentru eventualele probleme apărute în cod sau la instalare (se explică mai jos).

Component	Status	Instances
time-to-log.time-processor-v1	DEPLOYING	1 instance
time-to-log.time-source-v1	DEPLOYING	1 instance
time-to-log.logging-sink-v1	DEPLOYING	1 instance

Figura 9 - Verificarea stării de funcționare a aplicațiilor

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

The screenshot shows the Data Flow interface with the 'Streams' section selected. A single stream definition is listed: 'time-to-log'. The interface includes a search bar, filter options, and a status indicator showing 'DEPLOYING'.

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**:

The screenshot shows the 'Instances for app time-to-log.logging-sink-v1' screen. It displays configuration details for an instance named 'time-to-log.logging-sink-v1-0'. The 'stdout' field is highlighted with a red box.

guid	20923
pid	5166
port	20923
skipper.application.name	logging-sink
skipper.release.name	time-to-log
skipper.release.version	1
stderr	/tmp/1638826939712/time-to-log.logging-sink-v1/stderr_0.log
stdout	/tmp/1638826939712/time-to-log.logging-sink-v1/stdout_0.log
url	http://127.0.1.1:20923
working.dir	/tmp/1638826939712/time-to-log.logging-sink-v1

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).

```

/tmp/1638809589894/time-to-log.logging-sink-v1/stdout_0.log - Mousepad
File Edit Search View Document Help
:: Spring Boot :: (v2.3.12.RELEASE)

2021-12-06 18:53:13.631 INFO 4508 --- [main] com.sd.laborator.SinkKt : Starting SinkKt v1.0-SNAPSHOT on debian with PID 4508 (/home/student/Documents/SD_Labs/Lab...
2021-12-06 18:53:13.636 INFO 4508 --- [main] com.sd.laborator.SinkKt : No active profile set, falling back to default profile: default
2021-12-06 18:53:15.656 WARN 4508 --- [kground-preinit] o.s.h.c.j.JacksonObjectMapperBuilder : For Jackson Kotlin classes support please add "com.fasterxml.jackson.module:jackson-module...
2021-12-06 18:53:17.284 INFO 4508 --- [main] faultConfiguringBeanFactoryPostProcessor : No bean named 'errorChannel' has been explicitly defined. Therefore, a default PublishSubscribe...
2021-12-06 18:53:17.311 INFO 4508 --- [main] faultConfiguringBeanFactoryPostProcessor : No bean named 'taskScheduler' has been explicitly defined. Therefore, a default ThreadPoolTaskSchedul...
2021-12-06 18:53:17.341 INFO 4508 --- [main] faultConfiguringBeanFactoryPostProcessor : No bean named 'integrationHeaderChannelRegistry' has been explicitly defined. Therefore, a ...
2021-12-06 18:53:17.530 INFO 4508 --- [main] trationDelegatesBeanPostProcessorChecker : Bean 'integrationChannelResolver' of type [org.springframework.integration.support.channel...
2021-12-06 18:53:17.550 INFO 4508 --- [main] trationDelegatesBeanPostProcessorChecker : Bean 'integrationDisposableAutoCreatedBeans' of type [org.springframework.integration.config...
2021-12-06 18:53:17.600 INFO 4508 --- [main] trationDelegatesBeanPostProcessorChecker : Bean 'org.springframework.integration.config.IntegrationManagementConfiguration' of type [c...
2021-12-06 18:53:19.986 INFO 4508 --- [main] onConfigurationFunctionBindingRegistrar : Functional binding is disabled due to the presence of @EnableBinding annotation in your co...
2021-12-06 18:53:20.093 INFO 4508 --- [main] o.s.s.c.ThreadPoolTaskScheduler : Initializing ExecutorService 'taskScheduler'
2021-12-06 18:53:20.394 INFO 4508 --- [main] o.s.c.s.m.DirectWithAttributesChannel : Channel 'application_input' has 1 subscriber(s).
2021-12-06 18:53:20.398 INFO 4508 --- [main] o.s.i.endpoint.EventDrivenConsumer : Adding (logging-channel-adapter: org.springframework.integration.errorLogger) as a subscript...
2021-12-06 18:53:20.414 INFO 4508 --- [main] o.s.i.channel.PublishSubscribeChannel : Channel 'application_errorChannel' has 1 subscriber(s).
2021-12-06 18:53:20.414 INFO 4508 --- [main] o.s.i.endpoint.EventDrivenConsumer : started bean '_org.springframework.integration.errorLogger'
2021-12-06 18:53:21.533 INFO 4508 --- [main] c.s.b.r.p.RabbitExchangeQueueProvisioner : declaring queue for inbound time-to-log.time-processor.time-to-log, bound to: time-to-log...
2021-12-06 18:53:21.550 INFO 4508 --- [main] o.s.a.r.c.CachingConnectionFactory : Attempting to connect to: [localhost:5672]
2021-12-06 18:53:21.858 INFO 4508 --- [main] o.s.a.r.c.CachingConnectionFactory : Created new connection: rabbitConnectionFactory#459e9125:0/SimpleConnection@33990a0c [deleg...
2021-12-06 18:53:22.138 INFO 4508 --- [main] o.s.c.stream.binder.BinderErrorChannel : Channel 'time-to-log.time-processor.time-to-log.errors' has 1 subscriber(s).
2021-12-06 18:53:22.138 INFO 4508 --- [main] o.s.c.stream.binder.BinderErrorChannel : Channel 'time-to-log.time-processor.time-to-log.errors' has 2 subscriber(s).
2021-12-06 18:53:22.207 INFO 4508 --- [main] o.s.i.a.i.AmqpInboundChannelAdapter : started bean 'inbound.time-to-log.time-processor.time-to-log'
2021-12-06 18:53:22.260 INFO 4508 --- [main] com.sd.laborator.SinkKt : Started SinkKt in 11.035 seconds (JVM running for 12.248)

Am primit urmatorul mesaj: 2021/12/06 06:53:24
Am primit urmatorul mesaj: 2021/12/06 06:53:34
Am primit urmatorul mesaj: 2021/12/06 06:53:44
Am primit urmatorul mesaj: 2021/12/06 06:53:54
Am primit urmatorul mesaj: 2021/12/06 06:54:04
Am primit urmatorul mesaj: 2021/12/06 06:54:14
Am primit urmatorul mesaj: 2021/12/06 06:54:24
Am primit urmatorul mesaj: 2021/12/06 06:54:34
Am primit urmatorul mesaj: 2021/12/06 06:54:44
Am primit urmatorul mesaj: 2021/12/06 06:54:54
Am primit urmatorul mesaj: 2021/12/06 06:55:04
Am primit urmatorul mesaj: 2021/12/06 06:55:14
Am primit urmatorul mesaj: 2021/12/06 06:55:24
Am primit urmatorul mesaj: 2021/12/06 06:55:34
Am primit urmatorul mesaj: 2021/12/06 06:55:44
Am primit urmatorul mesaj: 2021/12/06 06:55:54
Am primit urmatorul mesaj: 2021/12/06 06:56:04
Am primit urmatorul mesaj: 2021/12/06 06:56:14
Am primit urmatorul mesaj: 2021/12/06 06:56:24
Am primit urmatorul mesaj: 2021/12/06 06:56:34
Am primit urmatorul mesaj: 2021/12/06 06:56:44

```

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.

Se recomandă consultarea următoarelor referințe:

- noțiunile despre DDD sumarizate în **laboratorul 6** (pag. 4) de la disciplina Sisteme Distribuite
- capitolul **Domain-Driven Design (DDD) Principles and Patterns** din carteia **Architectural Patterns: Uncover Essential Patterns in the Most Indispensable Realm of Enterprise Architecture**, de Pethuru Raj Chelliah

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 într-un 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

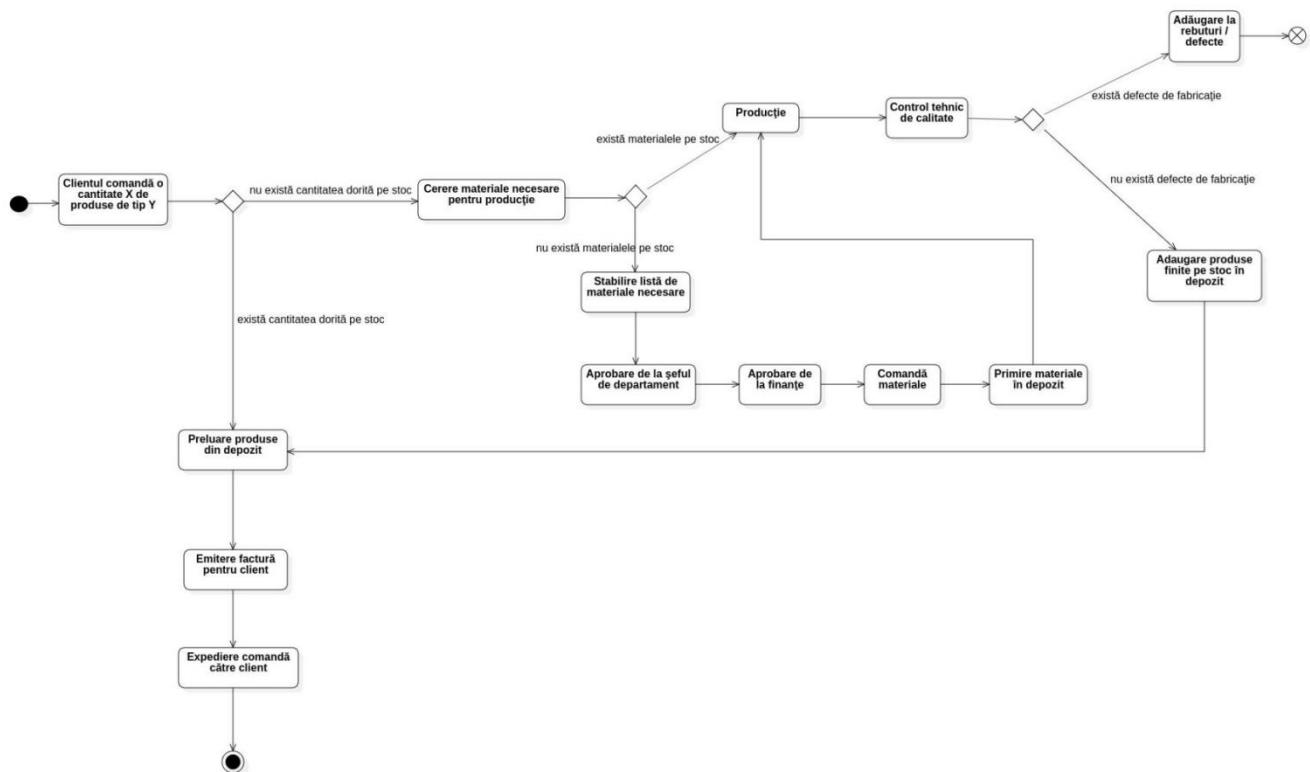


Figura 13 - Fluxul de afaceri

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.

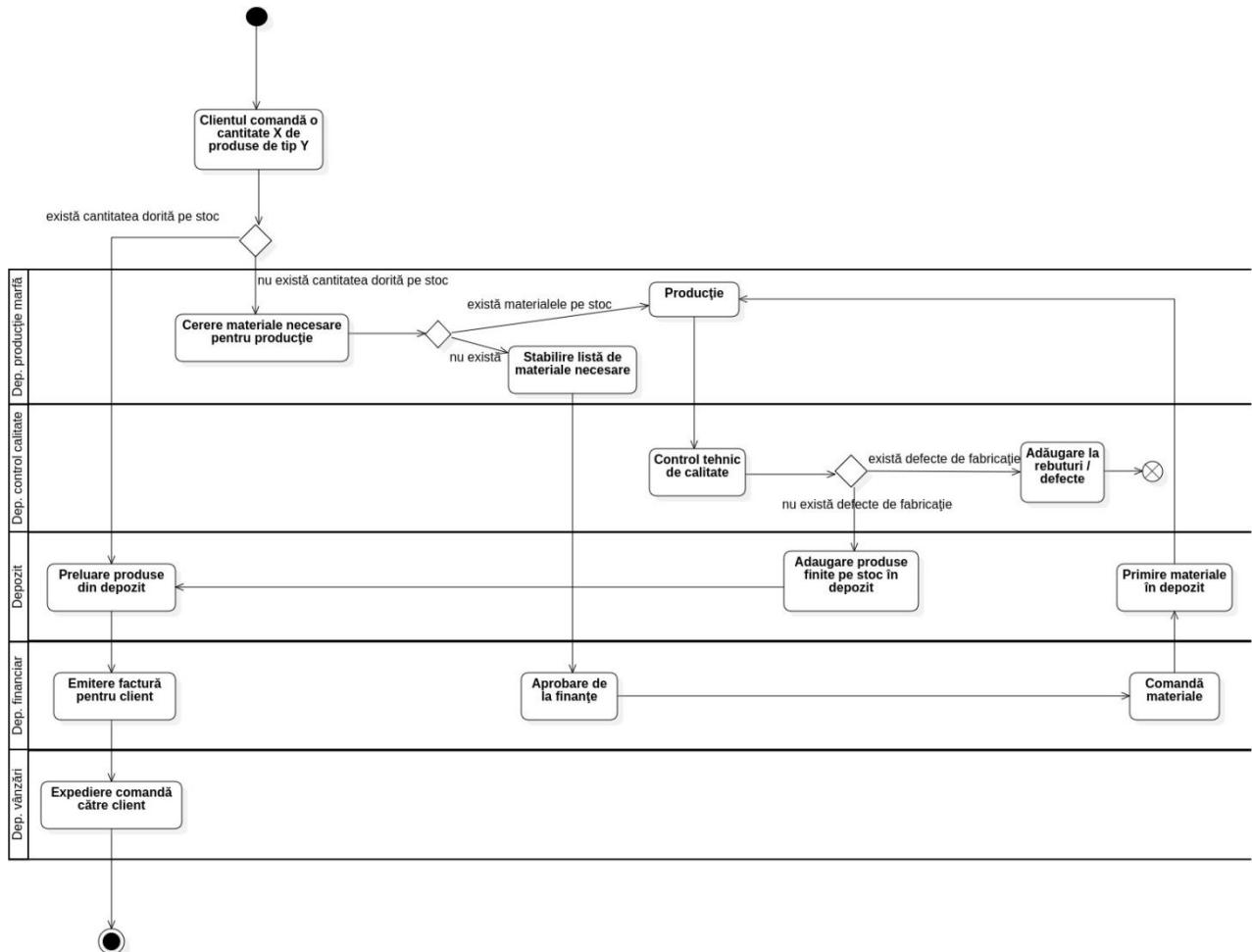


Figura 14 - Împărțirea pe departamente

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.

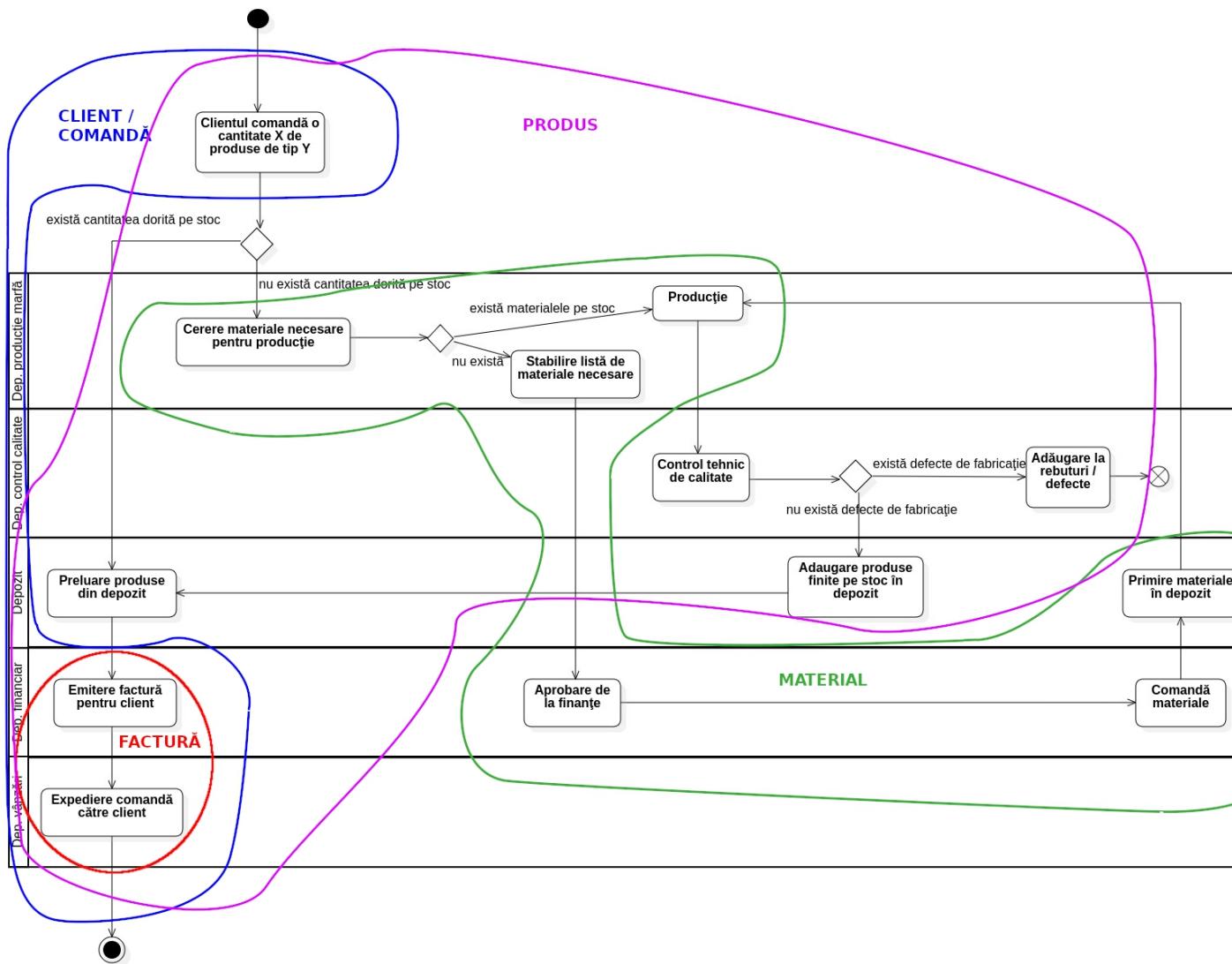


Figura 15 - Contextele mărginite

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 stabilită. Totuși, prețul, descrierea, și alte atrbute specifice pot fi schimbate de câte ori este nevoie.

Entitățile care reies, la o primă analiză, din fluxul de afacere modelat ar fi următoarele:

- CLIENT
- COMANDĂ
- PRODUS
- MATERIAL
- FACTURĂ

Pentru că avem nevoie și de stratul de persistență s-a ales utilizarea unui SGBD care suportă SQL. Proprietățile, respectiv relațiile între aceste entități se pot modela utilizând o **diagramă entitate-relație** (*Entity-Relationship Diagram*).

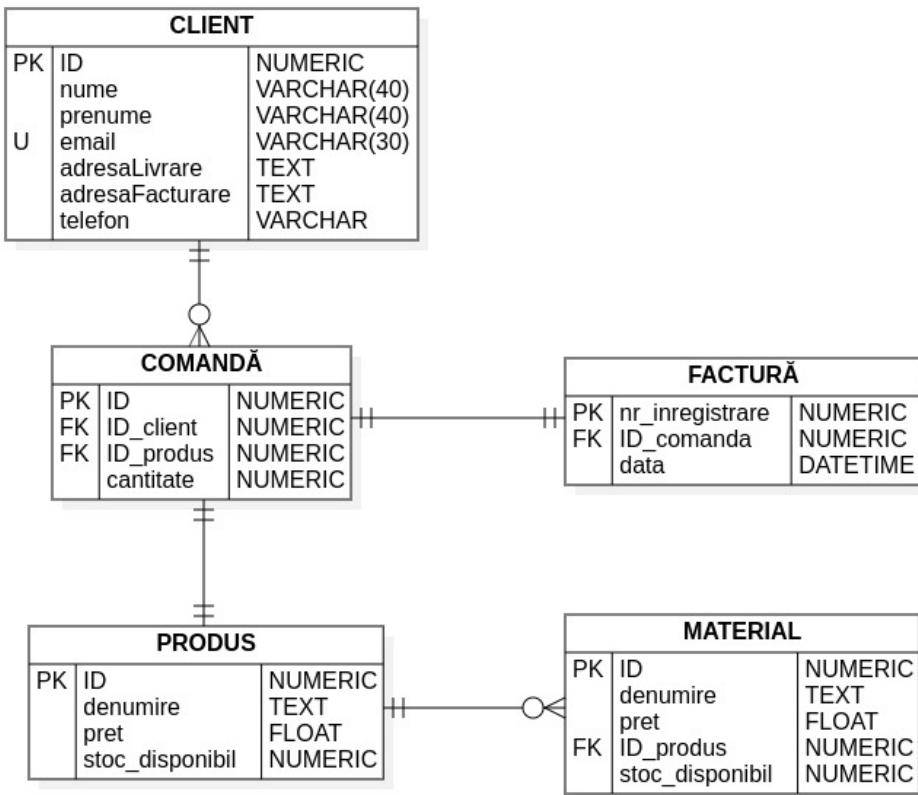


Figura 16 - Diagrama ERD

Identificarea serviciilor de domeniu (domain services)

În limbajul specific analizei aplicației (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 dar și concepte care nu pot fi modelate ca și entități sau obiecte de valoare (o cărpăceală de fapt din punct de vedere al proiectării pentru sisteme mari - acolo astfel de nedeterminări sunt semnale de alarmă ca analiza și proiectarea au scăpat mai multe aspecte din vedere). 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 descrise utilizând **limbajul specific asociat respectivelor contexte (de fapt în acest caz trivial se crează simple asocieri de operații prin care trebuie să treacă un produs (vezi rețele petri temporizate și colorate)**:

- **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 finanțiar
- **materialele necesare sunt comandate** de la furnizori de către departamentul finanțiar
- 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 **coresponde 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*). Abordarea este greșită din punct de vedere al proiectării cu microservicii unde trebuie să începem cu diagrama de microservicii și apoi să vedem descrierea fiecărui la un nivel mai fin de proiectare (de exemplu utilizând diagrama de clase). Cei buni ar trebui să treacă mai întâi prin această fază de proiectare. Totuși pentru simplitatea exemplului în acest caz s-a făcut rabat la aceste reguli și s-a trecut direct la diagrama de clase.

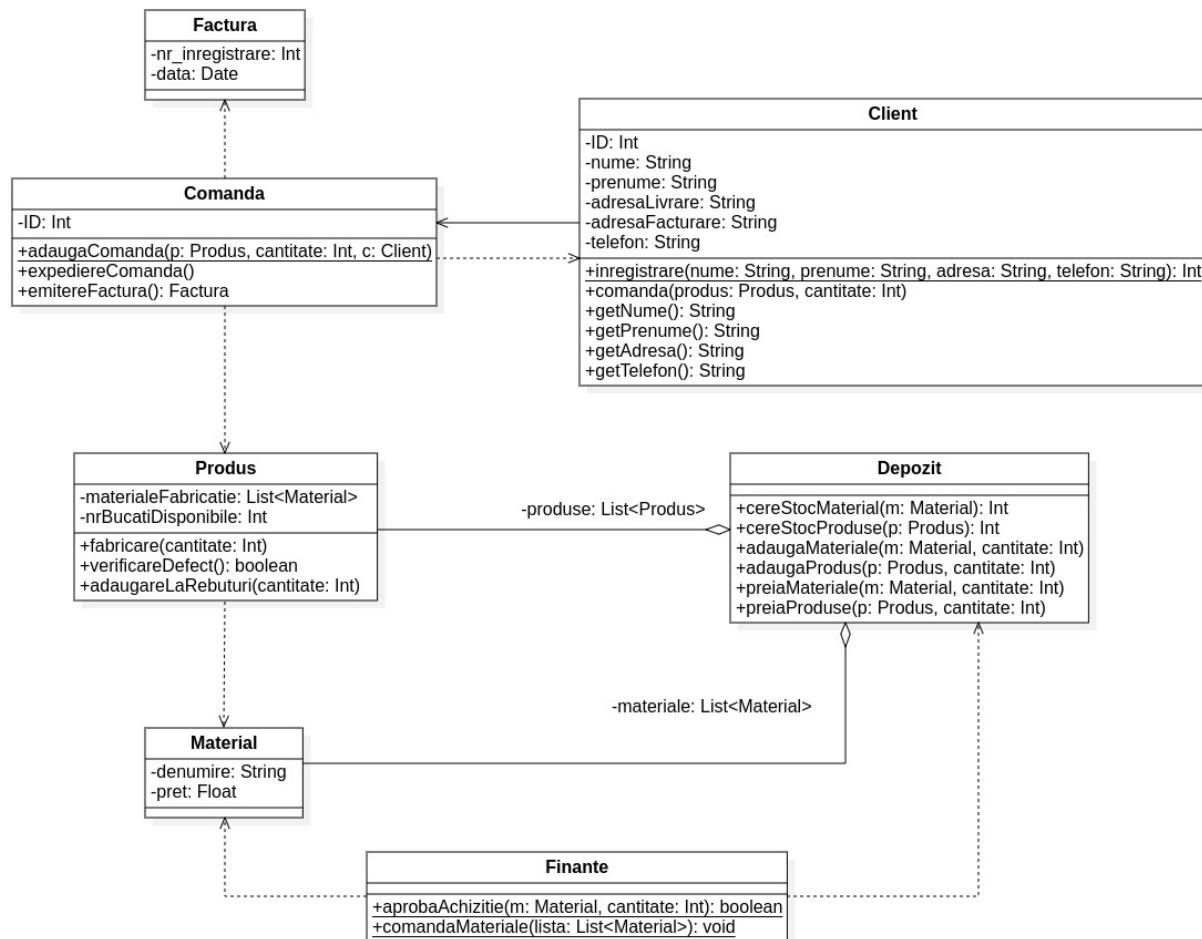


Figura 17 - Diagrama de clase

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 finanțier, 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:

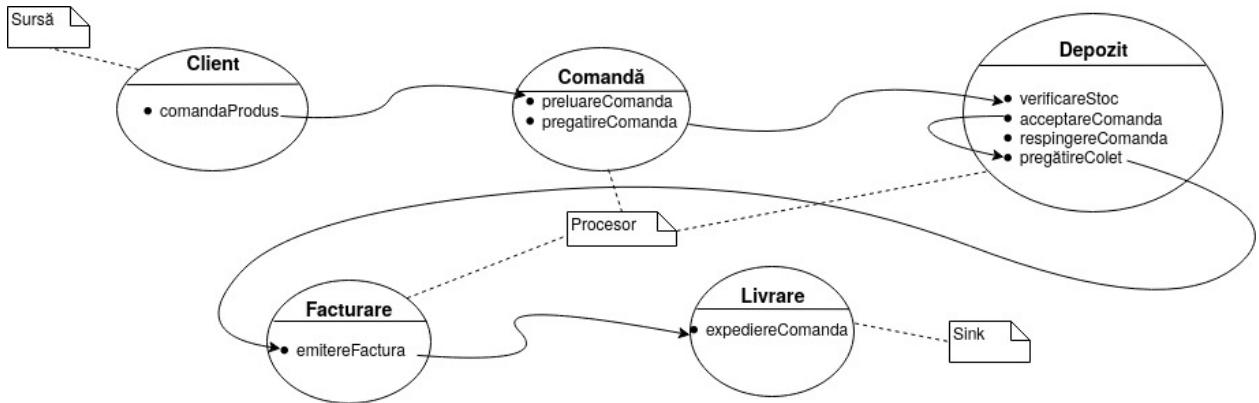


Figura 18 - Diagrama de microservicii

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.SpringApplication
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 listaProduse: 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 = listaProduse[0 until
    }
}
  
```

```

listaProduse.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.SpringApplication
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 pregeste 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 comenzi 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 comenzi
    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.SpringApplication
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 comenzi si atat
    fun emitereFactura(comanda: String?): String {
        val (identitateClient, produsComandat, cantitate,
adresaLivrare) = comanda!!.split("|")
        println("Emit factura pentru comanda $comanda...")
        val nrFactura = abs(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.SpringApplication
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 comenzi 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 fiecareia dintre ele. Apoi, configurați următorul pipeline:

client comanda depozit facturare livrare
--

```

2021-12-07 09:51:14.209 INFO 12377 --- [main] com.sd.laborator.LivrareKt           : Starting LivrareKt v1.0-SNAPSHOT on debian with PID 12377 (/home/student/Documents/SD_Labs/Lab_9/SD_Laborator_09)
2021-12-07 09:51:14.515 WARN 12377 --- [kground-preinit] o.s.n.c.j.JacksonObjectMapperBuilder   : No active profile set, falling back to default profiles: default
2021-12-07 09:51:15.519 INFO 12377 --- [main] o.s.n.c.j.JacksonObjectMapperBuilder   : For Jackson Kotlin classes support please add "com.fasterxml.jackson.module:jackson-module-kotlin" to the classpath
2021-12-07 09:51:17.699 INFO 12377 --- [main] faultConfiguringBeanFactoryPostProcessor: No bean named 'errorChannel' has been explicitly defined. Therefore, a default PublishSubscribeChannel will be created
2021-12-07 09:51:17.713 INFO 12377 --- [main] faultConfiguringBeanFactoryPostProcessor: No bean named 'taskScheduler' has been explicitly defined. Therefore, a default ThreadPoolTaskScheduler will be created
2021-12-07 09:51:17.722 INFO 12377 --- [main] faultConfiguringBeanFactoryPostProcessor: No bean named 'integrationHeaderChannelRegistry' has been explicitly defined. Therefore, a default DefaultHeaderChannelRegistry will be created
2021-12-07 09:51:17.964 INFO 12377 --- [main] trationDelegate$BeanPostProcessorChecker: Bean 'integrationChannelResolver' of type [org.springframework.integration.support.channel.BeanFactoryChannelResolver] is not eligible for getting processed by all BeanPostProcessors (due to its bypassed setting)
2021-12-07 09:51:17.988 INFO 12377 --- [main] trationDelegate$BeanPostProcessorChecker: Bean 'integrationDisposableAutoCreatedBeans' of type [org.springframework.integration.config.annotation.DisposableBeanPostProcessor] is not eligible for getting processed by all BeanPostProcessors (due to its bypassed setting)
2021-12-07 09:51:20.409 INFO 12377 --- [main] trationDelegate$BeanPostProcessorChecker: Bean 'org.springframework.integration.management.IntegrationManagementConfiguration' of type [org.springframework.integration.config.annotation.IntegrationManagementConfiguration] is not eligible for getting processed by all BeanPostProcessors (due to its bypassed setting)
2021-12-07 09:51:22.409 INFO 12377 --- [main] trationDelegate$BeanPostProcessorChecker: Bean 'org.springframework.integration.function.FunctionBindingRegistrar' of type [org.springframework.integration.function.FunctionBindingRegistrar] is not eligible for getting processed by all BeanPostProcessors (due to its bypassed setting)
2021-12-07 09:51:22.605 INFO 12377 --- [main] o.s.c.s.m.ThreadPoolTaskScheduler       : Initializing ExecutorService 'taskScheduler'
2021-12-07 09:51:23.235 INFO 12377 --- [main] o.s.o.r.c.CachingConnectionFactory      : Attempting to connect to: [localhost:5672]
2021-12-07 09:51:23.245 INFO 12377 --- [main] o.s.i.endpoint.EventDrivenConsumer       : Adding {logging-channel-adapter:[org.springframework.integration.errorLogger]} as a subscriber to the 'errorChannel' channel
2021-12-07 09:51:23.255 INFO 12377 --- [main] o.s.i.channel.PublishSubscribeChannel    : Channel 'application:errorChannel' has 1 subscriber(s).
2021-12-07 09:51:23.255 INFO 12377 --- [main] o.s.i.endpoint.EventDrivenConsumer       : started bean 'org.springframework.integration.errorLogger'
2021-12-07 09:51:24.043 INFO 12377 --- [main] c.s.b.r.p.RabbitExchangeQueueProvisioner: declaring queue for inbound: app:facturare.app, bound to: app:facturare
2021-12-07 09:51:24.059 INFO 12377 --- [main] o.s.a.r.c.CachingConnectionFactory      : Attempting to connect to: [localhost:5672]
2021-12-07 09:51:24.077 INFO 12377 --- [main] o.s.a.r.c.CachingConnectionFactory      : Create a connection to [localhost:5672]
2021-12-07 09:51:24.252 INFO 12377 --- [main] o.s.t.s.binder.BinderErrorHandler      : Channel 'app:facturare.app.errors' has 1 subscriber(s).
2021-12-07 09:51:24.252 INFO 12377 --- [main] o.s.t.s.binder.BinderErrorHandler      : Channel 'app:facturare.app.errors' has 2 subscriber(s).
2021-12-07 09:51:24.268 INFO 12377 --- [main] o.s.l.i.i.AmqpInboundChannelAdapter     : started bean 'inbound.app:facturare.app'
2021-12-07 09:51:24.295 INFO 12377 --- [main] com.sd.laborator.LivrareKt           : Started LivrareKt in 12.462 seconds (JVM running for 13.795)
5-a expediat urmatoarea comanda: Popescu Ion|Combinzon|75|Codrii Vlasiel nr 14
5-a expediat urmatoarea comanda: Popescu Ion|Vaccin anti-COVID-19|46|Codrii Vlasiel nr 14
5-a expediat urmatoarea comanda: Popescu Ion|Mască medicală|98|Codrii Vlasiel nr 14
5-a expediat urmatoarea comanda: Popescu Ion|Mască protecție|4|Codrii Vlasiel nr 14
5-a expediat urmatoarea comanda: Popescu Ion|Manusa chirurgicala|89|Codrii Vlasiel nr 14
5-a expediat urmatoarea comanda: Popescu Ion|Manusa chirurgicala|18|Codrii Vlasiel nr 14
5-a expediat urmatoarea comanda: Popescu Ion|Vaccin anti-COVID-19|70|Codrii Vlasiel nr 14
5-a expediat urmatoarea comanda: Popescu Ion|Mască protectie|85|Codrii Vlasiel nr 14
5-a expediat urmatoarea comanda: Popescu Ion|Mască protecție|8|Codrii Vlasiel nr 14
5-a expediat urmatoarea comanda: Popescu Ion|Combinzon|46|Codrii Vlasiel nr 14

```

Figura 19 - Verificarea log-ului microserviciului de 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

- Implementați o aplicație care să poată executa comenzi Linux sub formă de *pipeline*, fără a folosi *pipe*-urile *shell*-ului, ci folosind microservicii care interacționează conform unui automat de stare simulat prin interacțiuni simultane între mai multe *pipeline-uri de tip Spring Data Flow*.

Exemplu de comenzi conectate prin *pipeline*:

```
fortune | cowsay | lolcat
```

Exemplu de rezultat:

```

student@debian:~$ fortune | cowsay | lolcat
< Make a wish, it might come true. >
-----
 \ ^ ^
  (oo)\_____
  (__)\       )\/\
               ||----w |
               ||     ||

student@debian:~$ 

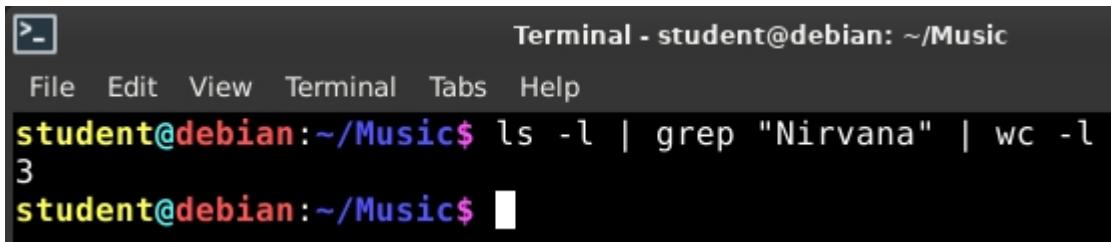
```

(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 Nirvana dintr-un folder:

```
ls -l | grep "Nirvana" | wc -l
```

Exemplu de rezultat:



```
Terminal - student@debian: ~/Music
File Edit View Terminal Tabs Help
student@debian:~/Music$ ls -l | grep "Nirvana" | wc -l
3
student@debian:~/Music$
```

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 „|“), stabilite la bun început apoi la analiza unui număr variabil de cazuri (să spunem 3-4) va rezulta acel automat de stare (care descrie procesarea).

- **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 comenzi care a mai rămas de executat către următorul procesor, și.a.m.d.

Exemplu:

```
ls -l | grep "Nirvana" | 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 "Nirvana" | wc -l
```

Al doilea procesor primește acel rezultat și comanda rămasă de mai sus, aplică **grep "Nirvana"** 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 comenziilor în pipeline
- 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, aşadar, în funcție de numărul de pipe-uri (simboluri „|“) din comandă, se vor instala (*deploy*) atâtea microservicii procesor căte 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 applications: [source.tcp, sink.twitter-update, source.http, sink.jdbc, sink.rabbit, source.rabbit, sink.mongodb, source.ftp, processor.object-detection, processor.image-recognition, sink.rsocket, processor.transform, processor.twitter-trend, source.sftp, processor.filter, source.file, sink.cassandra, sink.router, sink.twitter-message, source.twitter-message, source.mqtt, processor.splitter, processor.groovy, sink.redis, source.load-generator, sink.sftp, processor.semantic-segmentation, source.websocket, sink.file, source.cdc-bezium, processor.http-request, processor.header-enricher, source.time, sink.wavefront, processor.script, sink.mqtt, sink.tcp, source.jdbc, source.geode, sink.analytics, processor.bridge, source.s3, sink.ftp, sink.log, processor.aggregator, sink.elasticsearch, sink.throughput, source.zeromq, sink.s3, source.jms, sink.zeromq, sink.websocket, source.mongodb, source.mail, source.twitter-stream, sink.pgcopy, source.twitter-search, sink.geode, source.syslog]
dataflow:>
```

2. Să se modifice exemplul 2 din laborator astfel:

- a) realizarea unei interfețe GUI (puteți merge pe Flask, pyQt, Qt)
- b) utilizarea unei baze de date relaționale (de ex. SQLite) evident încapsulată ca un microserviciu
- 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 ulterioră (se modifică diagrama de uservicii, verific SOLID pentru userv si apoi descriu implementarea cu clase (de ex)).
- d) când se adună mai multe comenzi (minim 3) pentru același produs, depozitul să genereze o comandă pentru respectivul produs către producător (deci, un microserviciu separat pentru producător și o funcție dedicată în acest sens în microserviciul **Depozit**). Se va modifica descrierea ER astfel încât să se poată stabili diferențiat acest număr minim de produse. Se vor utiliza evenimente nor în implementare.

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