

Laboratorul 1

Dezvoltarea unei aplicații simple pentru întreprindere (JEE)

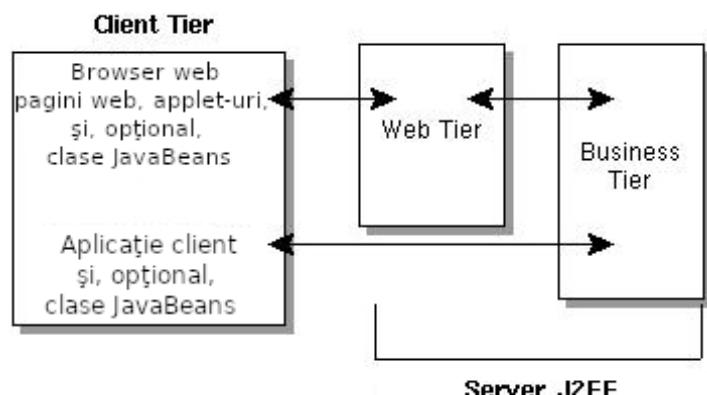
Arhitectura JEE

După cum am discutat la curs platforma **JEE (Java Enterprise Edition)** este proiectată pentru a-i ajuta pe dezvoltatori să creeze aplicații specifice întreprinderilor sau corporațiilor, multinivel - multistrat, scalabile și sigure.

1. Exemplu simplu de implementare în stilul JEE

Nivelul client constă într-o aplicație client care efectuează cereri către nivelul de afaceri care, în acest caz este alcătuit din două straturi cel pentru web și cel care conține logica de afaceri. Acest nivel tratează cererile venite de la clienți și procesează datele aplicației, pe care apoi le trimite către nivelul de persistență (detaliile au fost discutate la curs).

Datorită complexității mari a soluției Oracle pentru acest laborator s-a ales, utilizând GlassFish ca server de aplicații *enterprise* deoarece acesta este o variantă *open-source* de implementare a standardului JEE.



1.1. Instalare server GlassFish

Server-ul GlassFish este disponibil pentru descărcare aici:

<https://javaee.github.io/glassfish/download>

GlassFish
The Open Source Java EE Reference Implementation
 Java™
Enterprise Edition

Java EE 8 - GlassFish 5 Download

- [GlassFish 5.0 - Web Profile](#)
- [GlassFish 5.0 - Full Platform](#)

As an open source project, GlassFish is being developed in an open manner. Development versions of ongoing work for the next GlassFish iteration, i.e. GlassFish 5.0.1, will be made available shortly [here](#).

[Sources](#)
[Documentation](#)
[Download](#)
[Issue Tracker](#)

Descărcați ultima versiune de **GlassFish Full Platform** (versiunea **5.0** la momentul scrierii acestui laborator) de la URL-ul furnizat și dezarchivați conținutul. Pentru exemplul detaliat în continuare, vom considera server-ul dezarchivat în folder-ul `/home/student/opt/glassfish5`.

Atenție! GlassFish nu funcționează cu versiunile de Java ≥ 9 , aşadar trebuie să folosiți o versiune de Java SDK cel mult egală cu **8**. Verificați versiunea de Java cu următoarea

comandă:

```
java -version
```

În cazul în care sunt disponibile mai multe versiuni de Java pe stația de lucru (cu sistemul de operare Debian), iar acestea sunt înregistrate ca alternative, se poate schimba versiunea de Java la cerere, folosind comanda:

```
sudo update-alternatives --config java
```

Din lista de alternative, se alege indicele versiunii 8 (dacă este disponibilă! Dacă nu, trebuie instalată și înregistrată ca alternativă sau ca **JAVA_HOME** implicit în sistem). În acest caz, accesați următorul link:

<https://www.oracle.com/technetwork/java/javase/downloads/jdk11-downloads-5066655.html>

Și descărcați **jdk-11.0.5_linux-x64_bin.tar.gz**

Evident că, pentru o versiune mai veche de JDK, vă va cere un cont la ei (fie aveți, fie vi-l faceți). Apoi:

```
cd /home/nume_user/Downloads
tar -xvfz jdk-11.0.5_linux-x64_bin.tar.gz
```

Se va despacheta în: **/home/nume_user/Downloads/jdk-11.0.5/**

Această cale va fi utilizată în cadrul proiectelor IntelliJ din laborator pentru a indica JDK 11 atunci când acesta trebuie selecționat și nu există pe sistemul dumneavoastră deja instalat în calea implicită, adică **/usr/lib/jvm/.....**.

Sau, puteți să o despachetați acolo și eventual, dacă doriți să o mai folosiți și în alte medii:

```
sudo update-alternatives --install /usr/bin/java java
/usr/lib/jvm/jdk-11.0.5 1102
```

1.2. Creare și configurare proiect JEE minimal (Web Application) utilizând IntelliJ IDEA Community

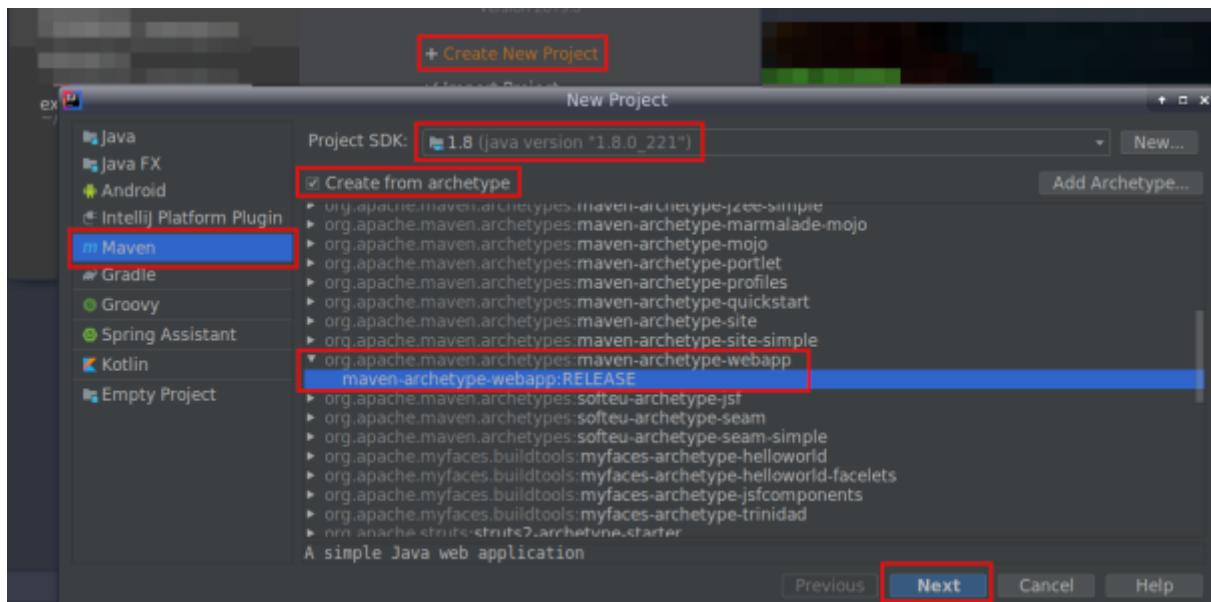
Mediul de dezvoltare IntelliJ IDEA Community nu dispune de posibilitatea de a crea în mod facil tipurile de proiecte Java Enterprise, aşa încât se vor urma pașii prezentați în continuare pentru a obține un proiect JEE **Web Application**.

1.2.1. Creare proiect IntelliJ

Deschideți IntelliJ IDEA Community, iar în meniul din partea dreaptă alegeti „Create new project”.

În fereastra de selecție a tipului de proiect, alegeti „**Maven**” în partea stângă, apoi selectați versiunea de **Java SDK 1.8**. Bifați „**Create from archetype**”, iar din lista de arhetipuri disponibile, expandați **org.apache.maven.archetypes:maven-archetype-webapp** și selectați **maven-archetype-webapp:RELEASE**. Apăsați pe „**Next**”.

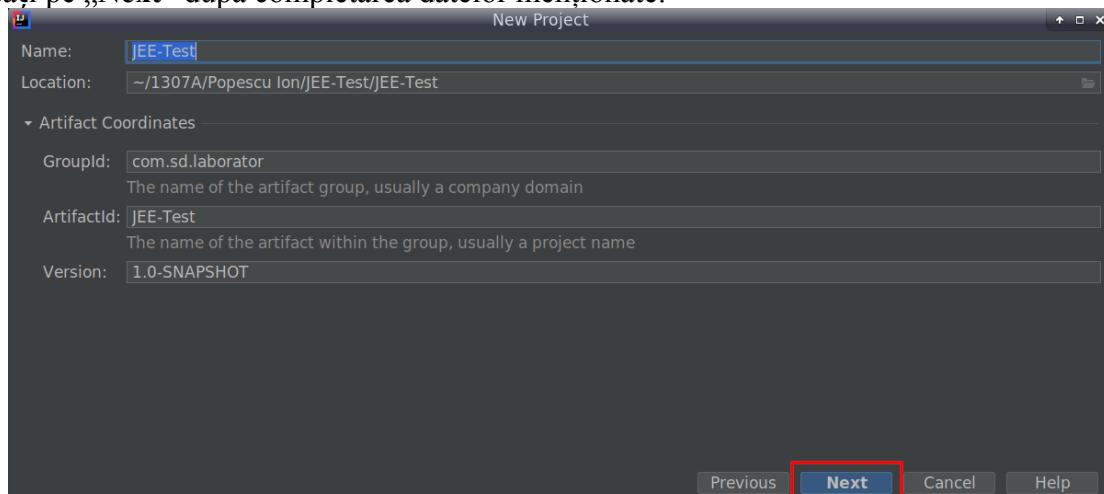
Dacă nu aveți Java 8 afișat în listă, apăsați pe butonul „**New**” din partea dreaptă a listei cu JDK-uri disponibile și selectați folder-ul unde este disponibilă versiunea 8 de Java SDK (de exemplu, **/usr/lib/jvm/oracle-java8-jdk-amd64**).



În continuare, se aleg numele și locația proiectului pe disc, precum și detaliile artefactului rezultat. Momentan, secțiunea „**Artifact Coordinates**” poate fi lăsată cu valorile implicate.

În acest exemplu, proiectul se va numi „**JEE-Test**”, iar locația va fi **~/1307A/Popescu Ion/JEE-Test**.

Apăsați pe „**Next**” după completarea datelor menționate.



În următoarea fereastră, se lasă totul neschimbă și se apasă „**Finish**”.

1.2.2. Configurare proiect Maven

Deschideți fișierul **pom.xml** (**Project Object Model**) și adăugați o primă dependență a proiectului: **JavaEE API** (<https://mvnrepository.com/artifact/javax/javaee-api/8.0>). Ca subordonat al *tag-ului <dependencies>*, adăugați următorul element:

```
<dependency>
<groupId>javax</groupId>
<artifactId>javaee-api</artifactId>
<version>8.0.1</version>
<scope>provided</scope>
</dependency>
```

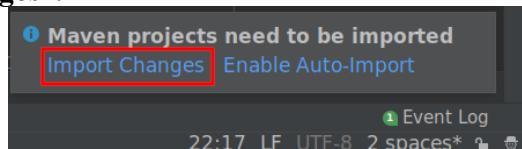
```

22   <dependencies>
23     <dependency>
24       <groupId>junit</groupId>
25       <artifactId>junit</artifactId>
26       <version>4.11</version>
27       <scope>test</scope>
28     </dependency>
29
30     <dependency>
31       <groupId>javax</groupId>
32       <artifactId>javaee-api</artifactId>
33       <version>8.0.1</version>
34       <scope>provided</scope>
35     </dependency>
36   </dependencies>
37
38   <build>
39     <finalName>JFF-Test</finalName>

```

project > dependencies > dependency

Pentru ca Maven să actualizeze proiectul conform cu schimbările făcute în fișierul **pom.xml**, atunci când IntelliJ afișează în partea din dreapta-jos un mesaj de avertizare în acest sens, alegeți „**Import changes**”.



Pentru a facilita execuția server-ului de aplicații *enterprise* și a încărca (a face *deploy*) automat la artefactele rezultate în urma compilării surselor, trebuie să adăugați plugin-ul **Cargo** pentru Maven, și să îl marcați ca dependență a proiectului. Așadar, adăugați următorul element ca subordonat al *tag-ului* **<plugins>**:

```

<plugin>
  <groupId>org.codehaus.cargo</groupId>
  <artifactId>cargo-maven2-plugin</artifactId>
  <version>1.7.9</version>
  <configuration>
    <container>
      <containerId>glassfish5x</containerId>
      <type>installed</type>
      <!-- Path to directory where glassfish is installed -->
      <home>/home/student/opt/glassfish5</home>
    </container>
    <configuration>
      <type>existing</type>
      <!-- Path to domains directory -->
      <home>/home/student/opt/glassfish5/glassfish/domains</home>
    <properties>
      <!-- Domain name where application will be deployed. -->
      <cargo.glassfish.domain.name>domain1</cargo.glassfish.domain.name>
      <!-- Glassfish user to authenticate -->
      <cargo.remote.username>admin</cargo.remote.username>
      <!-- Glassfish password to authenticate -->
      <cargo.remote.password></cargo.remote.password>
    </properties>
  </configuration>
</configuration>
</plugin>

```

Mai sus s-a evidențiat cu text îngroșat calea către server-ul GlassFish descărcat în pașii

anteriori. Înlocuiți această cale în mod corespunzător locației folder-ului **glassfish5** pe discul dvs.

Credențialele de autentificare la consola de administrare GlassFish au fost păstrate la valorile implicite (nume de utilizator: **admin**, parola: **<go1>**), cu scop demonstrativ. **Evident că nu se acceptă utilizarea unei parole vide în cazul real dintr-o firmă.**

```
73      <plugin>
74          <artifactId>maven-deploy-plugin</artifactId>
75          <version>2.8.2</version>
76      </plugin>
77      <plugin>
78          <groupId>org.codehaus.cargo</groupId>
79          <artifactId>cargo-maven2-plugin</artifactId>
80          <version>1.7.9</version>
81          <configuration>
82              <container>
83                  <containerId>glassfish5x</containerId>
84                  <type>installed</type>
85                  <!-- Path to directory where glassfish is installed -->
86                  <home>/home/cosmin/glassfish5</home>
87              </container>
88              <configuration>
89                  <type>existing</type>
90                  <!-- Path to domains directory -->
91                  <home>/home/cosmin/glassfish5/glassfish/domains</home>
92              <properties>
93                  <!-- Domain name where application will be deployed. -->
94                  <cargo.glassfish.domain.name>domain1</cargo.glassfish.domain.name>
95                  <!-- Glassfish user to authenticate -->
96                  <cargo.remote.username>admin</cargo.remote.username>
97                  <!-- Glassfish password to authenticate -->
98                  <cargo.remote.password></cargo.remote.password>
99              </properties>
100             </configuration>
101            </configuration>
102        </plugin>
103    </plugins>
104    </pluginManagement>
105  </build>
106 </project>
```

După adăugarea *plugin*-ului **Cargo**, marcați-l ca dependență a proiectului, adăugând următorul element ca subordonat al tag-ului **<dependencies>**:

```
<dependency>
    <groupId>org.codehaus.cargo</groupId>
    <artifactId>cargo-maven2-plugin</artifactId>
    <version>1.7.9</version>
</dependency>
```

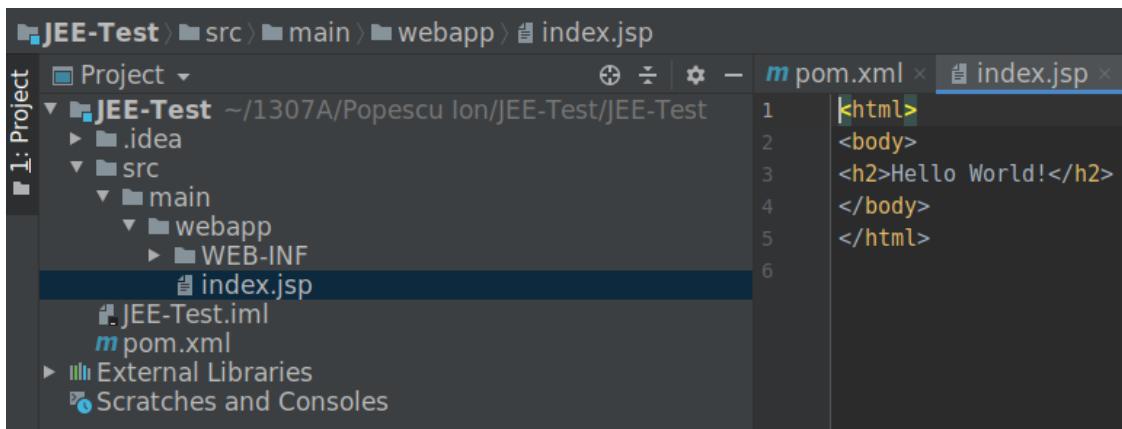
```
29
30      <dependency>
31          <groupId>javax</groupId>
32          <artifactId>javaee-api</artifactId>
33          <version>8.0.1</version>
34          <scope>provided</scope>
35      </dependency>
36
37      <dependency>
38          <groupId>org.codehaus.cargo</groupId>
39          <artifactId>cargo-maven2-plugin</artifactId>
40          <version>1.7.9</version>
41      </dependency>
42  </dependencies>
43
44  <build>
45      <finalName>JEE-Test</finalName>
46      <pluginManagement><!-- lock down plugins versions to avoid using Maven defaults (may be
```

Nu uitați să apăsați „Import changes” când IntelliJ detectează o modificare a fișierului **pom.xml specific Maven. Faceți acest lucru de fiecare dată când îl modificați**

(adăugați / ștergeți o dependență / un *plugin* sau modificați alte configurații).

1.2.3. Definirea organizării fișierelor sursă

Proiectul creat are structura din figura următoare:



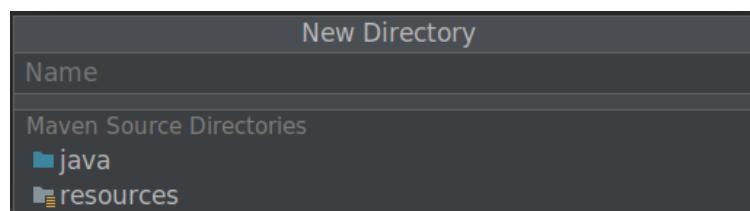
Folder-ul **src/main/webapp** este rădăcina componentei web a aplicației JEE de tip **Web Application**, ceea ce înseamnă că tot conținutul acesta va fi împachetat, la compilare, într-un fișier **WAR** (Web ARchive), ce va reprezenta artefactul încărcat (*deployed*) pe server-ul de aplicații *enterprise*.

Fișierul **index.jsp** este o pagină de tip **Java Server Page**, cu un conținut de test generat implicit.

Folder-ul **WEB-INF** conține fișiere / directoare inaccesibile public. Ele pot fi accesate doar de **servleti** sau alte pagini JSP, dar în niciun caz în mod direct. De obicei, aici se pun fișiere XML ce conțin descriptori care definesc comportamentul aplicației (de exemplu, în fișierul **WEB-INF/web.xml** se mapează servleții pe rutele de acces dorite - urmează în continuare).

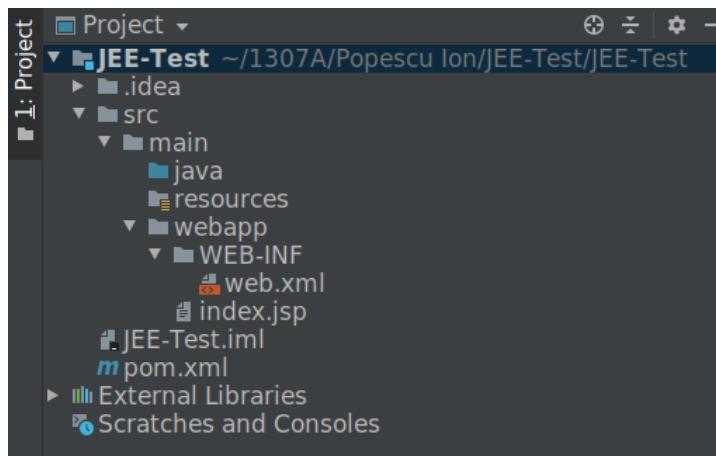
Din structura proiectului lipsesc două directoare, pe care le veți crea astfel:

- Apăsați dreapta pe folder-ul **main** → New → Directory.



- Selectați, pe rând, cele două directoare sugerate de IntelliJ, conform alegerii proiectul de tip Maven (dublu apăsați pe „**java**”, apoi repetați operațiunea de creare a unui folder nou și selectați apoi „**resources**”).

Folderul **java** va conține fișiere sursă Java (de ex. servleti), iar folderul **resources** va conține fișiere de tip resursă (de ex. fișiere de configurare XML, imagini, font-uri etc.). Așadar, structura finală de proiect arată astfel:



1.3. Adăugare servlet simplu

Un *servlet* este, în mod simplist spus, o componentă web care primește cereri și generează răspunsuri bazate pe cererile primite. Ca și tehnologie JEE, este plasat în stratul de *web*.

Adăugați un servlet simplu care va răspunde la o cerere HTTP de tip GET cu un mesaj „Hello from servlet”:

Apăsați dreapta pe folder-ul cu surse Java (**java**) → **New** → **Java Class**. Denumiți clasa ca „**HelloServlet**”. Introduceți următorul conținut:

```
public class HelloServlet extends HttpServlet {
    @Override
    protected void doGet(HttpServletRequest httpServletRequest,
    HttpServletResponse httpServletResponse)
            throws ServletException, IOException {
        httpServletResponse.getWriter().print("Hello from servlet");
    }
}
```

Servleții trebuie să se conformeze cu Java Servlet API, deci clasa creată extinde clasa de bază **HttpServlet**. Un servlet conține metode de tratare a fiecărui tip de cerere HTTP: pentru cerere GET → metoda **doGet()**, pentru cerere POST → metoda **doPost()**, și.a.m.d. Fiecare din aceste metode are la dispoziție ca parametri o variabilă care încapsulează cererea efectuată de client (**httpServletRequest**), respectiv una care încapsulează răspunsul ce va fi generat și trimis în metoda de tratare respectivă (**httpServletResponse**).

În cazul în care nu suprascrieți o anumită metodă de tratare a unui tip de cerere, aceasta va conține o implementare implicită, disponibilă în clasa de bază **HttpServlet**, mai exact un mesaj de eroare de tipul „**HTTP method TIP_METODĂ is not supported by this URL**”. Nu înseamnă că metoda nu există / nu este implementată!

Servletul creat mai sus pur și simplu scrie mesajul „Hello from servlet” în răspunsul trimis către client.

1.4. Maparea servletilor

Servleții sunt accesăți pe baza rutelor la care aceștia sunt mapăți în server-ul enterprise, dar pentru **HelloServlet** nu ați specificat nicăieri nici o cale de mapare, și deci nu l-ați putea accesa, dacă ar fi să testați aplicația aşa cum este acum.

Deschideți fișierul **webapp/WEB-INF/web.xml** și adăugați următorul conținut, ca un subordonat al elementului **<web-app>**:

```

<servlet>
    <servlet-name>Hello</servlet-name>
    <servlet-class>HelloServlet</servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name>Hello</servlet-name>
    <url-pattern>/hello</url-pattern>
</servlet-mapping>

```

```

m pom.xml × web.xml × index.jsp × HelloServlet.java ×
1   <!DOCTYPE web-app PUBLIC
2     "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
3     "http://java.sun.com/dtd/web-app_2_3.dtd" >
4
5   <web-app>
6     <display-name>Archetype Created Web Application</display-name>
7     <servlet>
8       <servlet-name>Hello</servlet-name>
9       <servlet-class>HelloServlet</servlet-class>
10    </servlet>
11   <servlet-mapping>
12     <servlet-name>Hello</servlet-name>
13     <url-pattern>/hello</url-pattern>
14   </servlet-mapping>
15 </web-app>
16

```

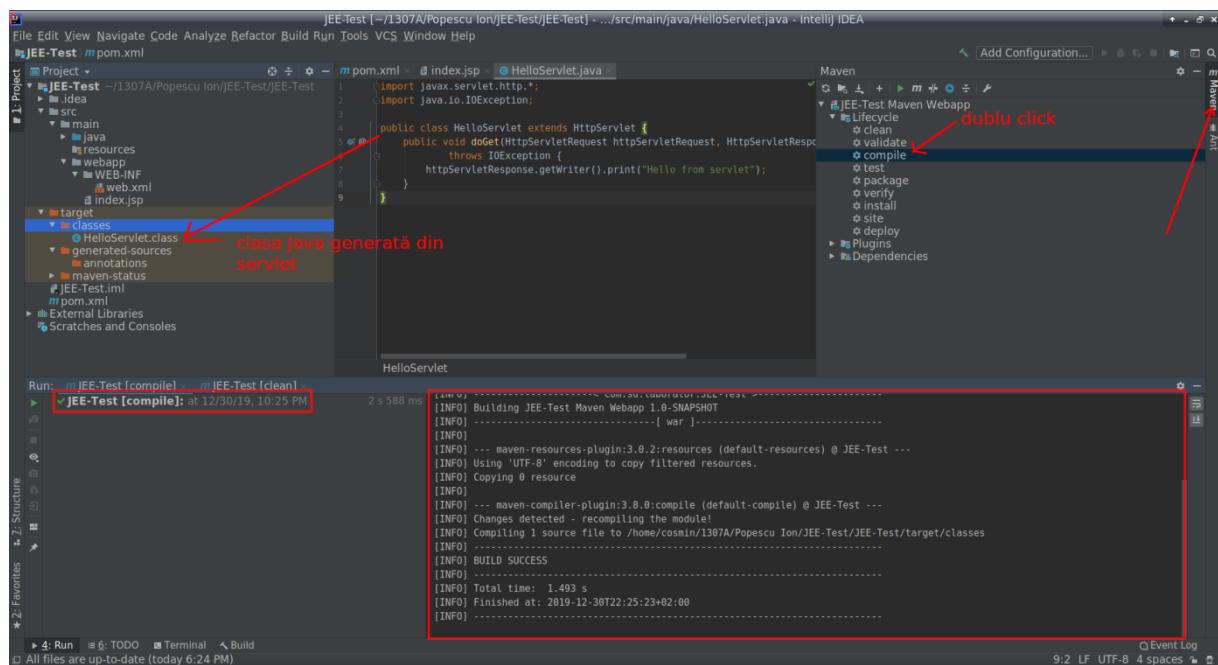
Cu ajutorul descriptorilor XML, pentru fiecare servlet în parte, descris prin numele clasei din care face parte (prefixat de numele pachetului, dacă este cazul), se poate specifica pe ce cale să fie mapat. Serverul *enterprise* folosește acest XML la încărcarea aplicației pentru a expune servleții corespunzător.

Calea prin care se poate accesa servlet-ul **HelloServlet** este, în acest caz, **/hello**. **Atenție, calea este relativă la rădăcina proiectului!** Adică, dacă pagina principală care face trimiter la rădăcina aplicației JEE este **/my-app**, atunci acest servlet se accesează folosind calea **/my-app/hello**, și nu simplu **/hello**!

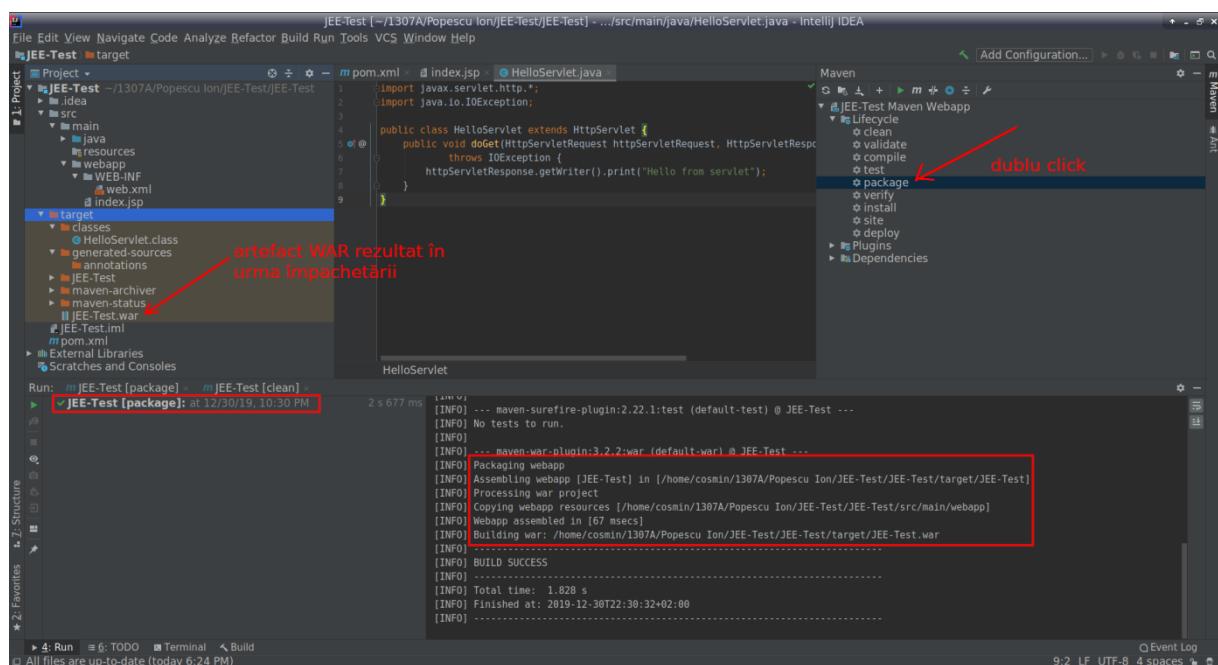
1.5. Compilare și împachetare aplicație JEE

Aplicația se compilează utilizând *lifecycle*-ul standard Maven numit „**compile**”, accesibil din panoul Maven din partea dreaptă a ferestrei IntelliJ. De asemenea, împachetarea aplicației într-un fișier **WAR**, pentru a o pregăti de încărcare pe server se face tot din același panou, folosind *lifecycle*-ul „**package**”.

Compilare:



Împachetare:



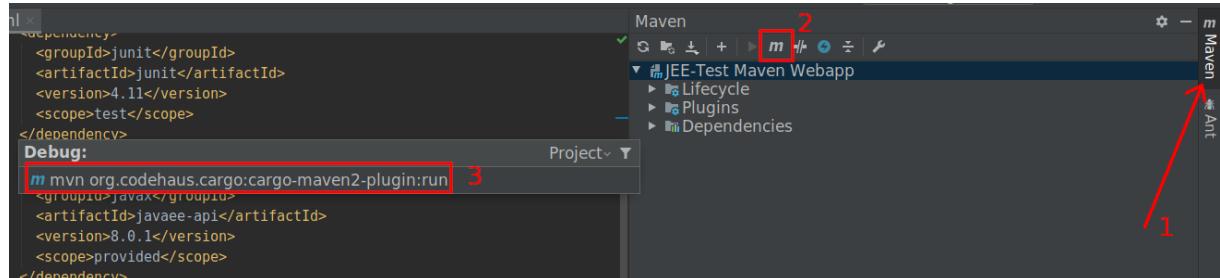
1.6. Încărcare artefact WAR și testare aplicație

După ce artefactul WAR este disponibil, trebuie încărcat (*deployed*) pe server-ul de

aplicații enterprise GlassFish (în acest caz). Acest lucru este făcut automat de *plugin-ul Cargo*, utilizând țelurile (*goal-urile*) Maven puse la dispoziție de acesta. Lista completă poate fi consultată în documentația Cargo: <https://codehaus-cargo.github.io/cargo/Maven2+plugin.html>

Goal-urile acestui *plugin* nu apar în lista de *Maven goals* din meniu standard IntelliJ (accesibilă din tab-ul **Maven** → secțiunea **Plugins**), așa încât vor trebui executate manual, astfel:

- Expandați tab-ul **Maven** din partea dreaptă a ferestrei IntelliJ, apoi apăsați pe pictograma în formă de „m” înclinat din partea de sus a panoului (**Execute Maven Goal**).

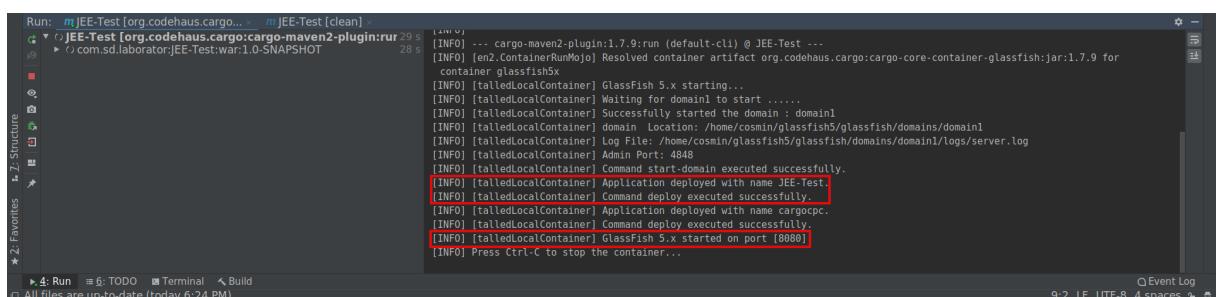


- Introduceți următoarea comandă și apăsați **ENTER**:

```
mvn org.codehaus.cargo:cargo-maven2-plugin:run
```

Atenție, prefixul **mvn** este adăugat automat de IntelliJ! Nu îl duplicați.

Acest goal Maven vă pornește server-ul GlassFish și, totodată, face și încărcarea artefactului WAR pe server.



Se poate observa că server-ul GlassFish a fost pornit cu succes, iar aplicația creată, împachetată sub formă de arhivă WAR a fost încărcată.

URL-ul de unde aplicația va fi disponibilă pentru testare este:

http://localhost:8080/<NUME_PROIECT>

În acest caz, dacă accesați <http://localhost:8080/JEE-Test> (ATENȚIE - este **case sensitive!**) dintr-un browser web, veți primi ca răspuns o pagină cu un mesaj „Hello World!”.



Hello World!

Ceea ce vedeți este rezultatul transformării într-un servlet a paginii de index din rădăcina folder-ului **webapp** (adică **index.jsp**). Paginile JSP sunt transformate automat în servleți de container-ul web!

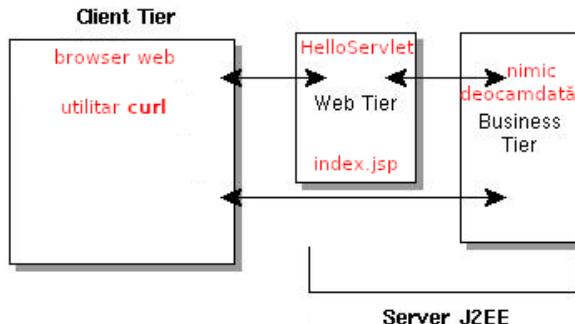
Prin accesarea rutei **/JEE-Test**, de fapt accesați rădăcina aplicației JEE create, deci server-ul vă va servi pagina implicită pe care o găsește disponibilă (dacă există). În acest caz există, și se numește **index.jsp**.

Pentru a accesa servlet-ul **Hello**, trebuie trimisă o cerere HTTP de tip GET către următorul URL: <http://localhost:8080/JEE-Test/hello>

Puteți face acest lucru pur și simplu navigând către acel URL dintr-un browser web, sau trimițând cererea manual folosind utilitarul **curl** din Linux:

```
$ curl -X GET http://localhost:8080/JEE-Test/hello
```

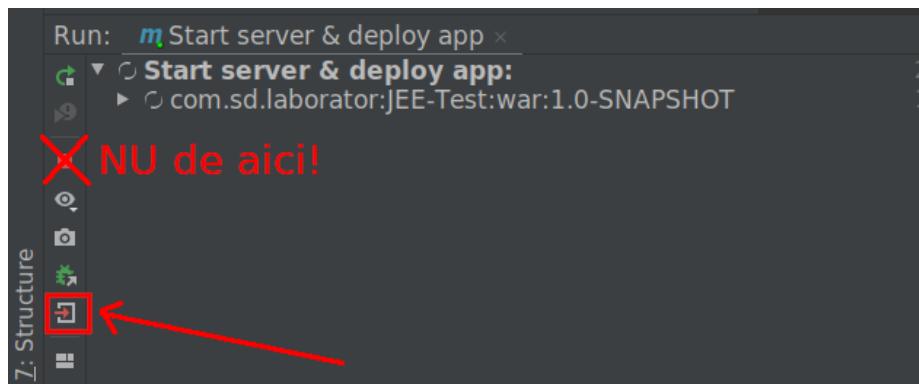
În acest moment, aplicația are următoarele componente, create în pașii efectuați până acum:



(structura aplicației create până acum)

1.7. Oprirea corectă a server-ului GlassFish

Server-ul GlassFish se oprește folosind butonul **Exit** din panoul de control al execuției din partea din stânga jos a ferestrei IntelliJ:

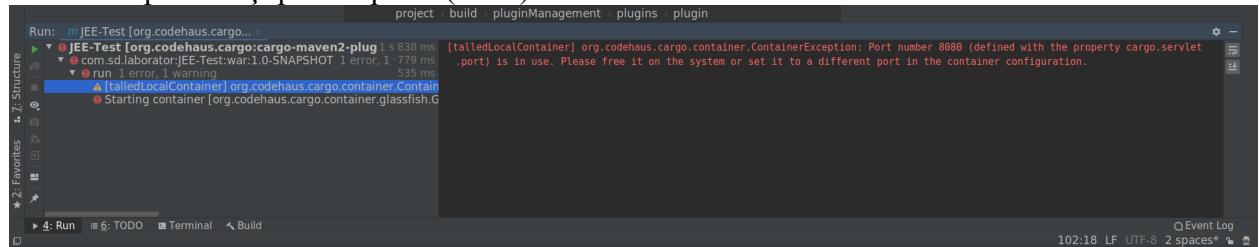


Veți ști că server-ul a fost oprit corect dacă veți următoarele mesaje în consola cu informații din partea de jos:

```
[INFO] [talledLocalContainer] Application deployed with name JEE-Test.  
[INFO] [talledLocalContainer] Command deploy executed successfully.  
[INFO] [talledLocalContainer] Application deployed with name cargopc.  
[INFO] [talledLocalContainer] Command deploy executed successfully.  
[INFO] [talledLocalContainer] GlassFish 5.x started on port [8080]  
[INFO] Press Ctrl-C to stop the container...  
[INFO] [talledLocalContainer] GlassFish 5.x is stopping...  
[INFO] -----  
[INFO] BUILD SUCCESS  
[INFO] -----  
[INFO] Total time: 49.247 s  
[INFO] Finished at: 2019-12-30T23:53:10+02:00  
[INFO] -----  
[INFO] [talledLocalContainer] Waiting for the domain to stop .  
[INFO] [talledLocalContainer] Command stop-domain executed successfully.  
[INFO] [talledLocalContainer] GlassFish 5.x is stopped
```

NU opriți server-ul folosind butonul Stop (pictograma pătrat roșu), deoarece procesul copil creat de *goal*-ul Maven care a pornit execuția GlassFish se va desprinde de procesul părinte controlat de IntelliJ și va rămâne în execuție în fundal, blocând portul 8080.

Dacă ați făcut din greșală acest lucru, veți primi o eroare dacă încercați să reporniți GlassFish pe același port implicit (8080):



Pentru a rezolva această problemă, executați următoarea comandă în terminal, ca să aflați PID-ul procesului care ocupă portul TCP 8080:

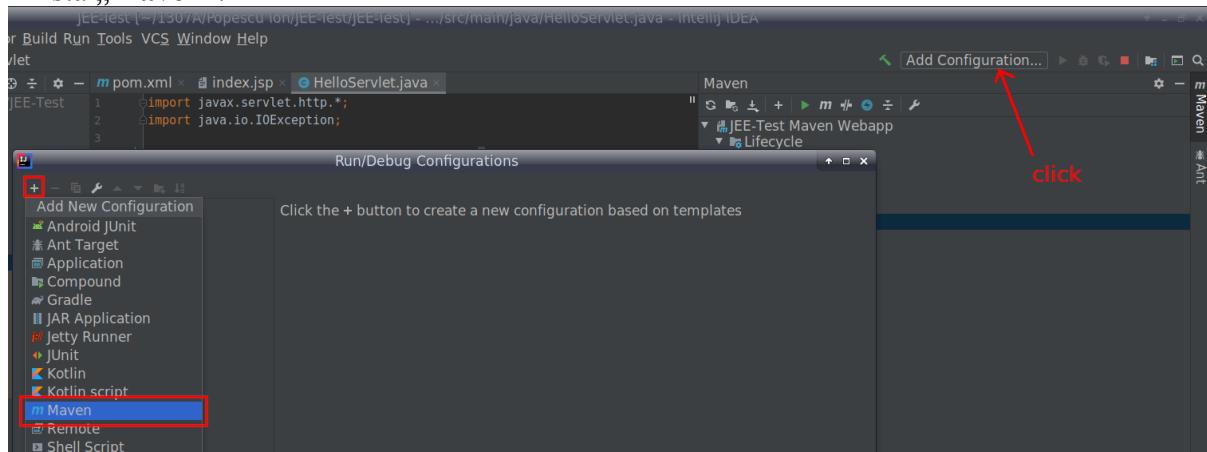
```
$ lsof -i tcp:8080
```

Folosiți PID-ul (de ex 3510) rezultat ca să închideți forțat acel proces:

```
$ kill -9 3510
```

1.8. Configurații de execuție

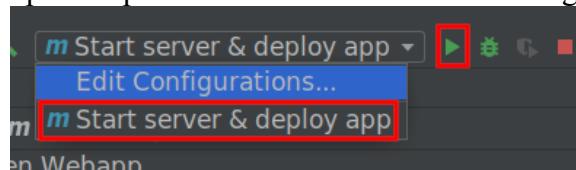
Pentru ușurință sporită, se pot crea configurații de execuție care execută Maven *goals* automat la apăsarea butonului „Run Configuration” din IntelliJ. În partea din dreapta-sus, apăsați pe „Add Configuration...”. În meniu afișat, apăsați pe semnul plus din stânga-sus, apoi alegeti din listă „Maven”.



Completați cu:

- Name: **Start server & deploy app**
- Command line: **org.codehaus.cargo:cargo-maven2-plugin:run**

Apoi apăsați pe OK, iar configurația de execuție va apărea în dreapta-sus a ferestrei IntelliJ. De acum, puteți porni server-ul GlassFish și încărca aplicația selectând configurația creată mai sus din listă, și apăsând pe butonul verde în formă de triunghi.



Apoi adăugați și o altă configurație des folosită, și anume reîncărcarea (*redeploy*) artefactelor:

- Name: **Redeploy**

- Command line: `org.codehaus.cargo:cargo-maven2-plugin:redeploy`

1.9. Reîncărcare după modificarea surselor

Dacă modificați fișierele sursă / adăugați fișiere noi / ștergeți fișiere și vreți să testați aplicația în versiunea nouă pe server, trebuie să urmați obligatoriu pașii:

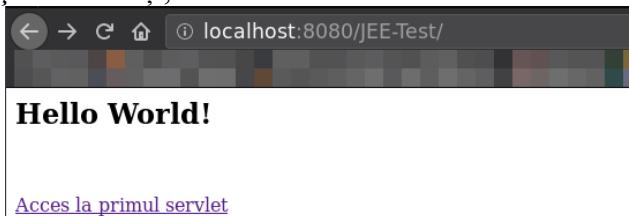
Compilare (compile) → Împachetare (package) → Reîncărcare (redeploy)

Cei trei pași au fost descriși în etapele anterioare.

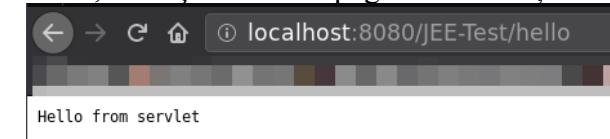
Spre exemplu, modificați pagina `index.jsp` pentru a include o ancoră HTML care redirecționează utilizatorul către calea prin care se accesează servlețul `HelloServlet`.

```
<html>
  <body>
    <h2>Hello World!</h2>
    <br />
    <p>
      <a href=".//hello">Acces la primul servlet</a>
    </p>
  </body>
</html>
```

Atenție la calea țintă, care este dată sub formă de cale relativă față de cea actuală: `.//hello`. Compilați, împachetați și reîncărcați, iar rezultatul va fi:



După apăsați pe ancoră, se afișează o altă pagină care conține răspunsul servlețului:

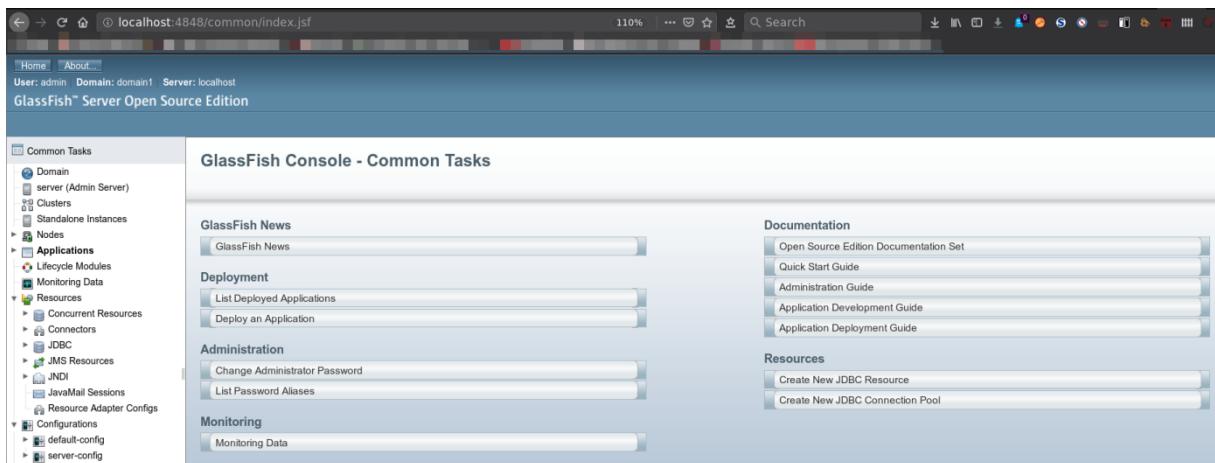


1.10. Accesul la consola de administrare GlassFish

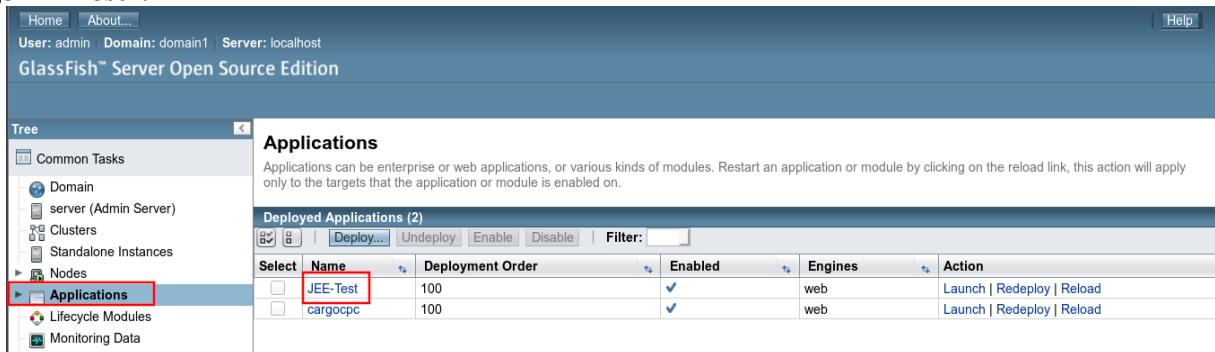
Având server-ul GlassFish pornit, se poate accesa consola de administrare ce este disponibilă în mod implicit pe portul 4848, după cum apare și în mesajele de la consola IntelliJ:

```
[INFO] --- cargo-maven2-plugin:1.7.9:run (default-cli) @ JEE-Test ...
[INFO] [en2.ContainerRunMojo] Resolved container artifact org.codehaus.cargo:cargo-core-container-glassfish:jar:1.7.9 for
container glassfish5x
[INFO] [talledLocalContainer] GlassFish 5.x starting...
[INFO] [talledLocalContainer] Waiting for domain1 to start .....
[INFO] [talledLocalContainer] Successfully started the domain : domain1
[INFO] [talledLocalContainer] domain Location: /home/cosmin/glassfish5/glassfish/domains/domain1
[INFO] [talledLocalContainer] Log File: /home/cosmin/glassfish5/glassfish/domains/domain1/logs/server.log
[INFO] [talledLocalContainer] Admin Port: 4848
[INFO] [talledLocalContainer] Command start-domain executed successfully.
[INFO] [talledLocalContainer] Application deployed with name JEE-Test.
[INFO] [talledLocalContainer] Command deploy executed successfully.
```

Accesați dintr-un browser web URL-ul <http://localhost:4848>, după ce v-ați asigurat că server-ul GlassFish a fost pornit cu succes.



Puteți, de exemplu, vizualiza aplicațiile enterprise încărcate pe server, din meniul din stânga → **Applications**. Acolo ar trebui să apară aplicația creată în pașii anteriori, numită „**JEE-Test**”.



După ce apăsați dreapta pe numele aplicației, se pot vedea componentele și tipul acestora într-o listă afișată în partea de jos. Puteți observa că apar liste servleți generați la compilare: cel implicit (**default**), pagina JSP `index.jsp` (**jsp**), și servlețul **Hello** pe care l-ați creat și mapat sub calea `/hello`.

De asemenea, tot de aici se poate găsi calea care identifică aplicația încărcată (către folder-ul rădăcină).

Edit Application

Modify an existing application or module.

Name:	JEE-Test
Status:	<input checked="" type="checkbox"/> Enabled
Virtual Servers:	server
Context Root:	/JEE-Test
Path relative to server's base URL. If empty, takes the default context path of a web application.	
Implicit CDI:	<input checked="" type="checkbox"/> Enabled
Implicit discovery of CDI beans	
Location:	\$com.sun.aas.instanceRootURI\$applications/JEE-Test/
Deployment Order:	100
A number that determines the loading order of the application at server startup. Lower numbers are loaded first. The default is 100.	
Libraries:	
Description:	

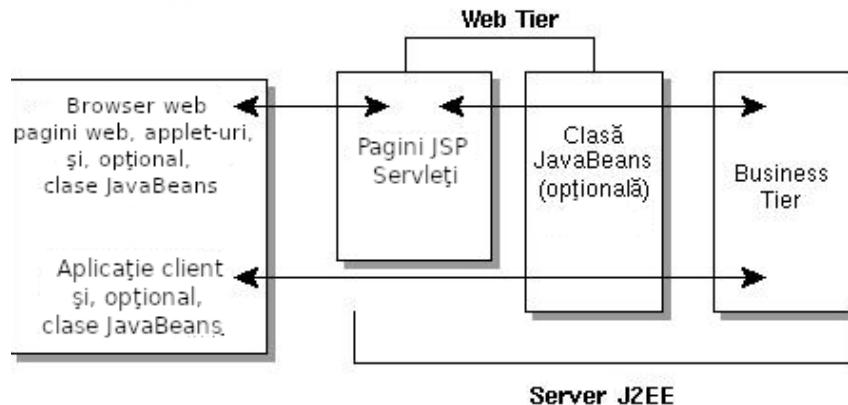
Modules and Components (4)

Module Name	Engines	Component Name	Type	Action
JEE-Test	[web]	-----	-----	Launch
JEE-Test		default	Serviet	
JEE-Test		Hello	Serviet	
JEE-Test		Jsp	Serviet	

ruta de bază (către folder-ul rădăcină) a aplicației

2. Utilizarea JavaBeans

Implementarea anterioară poate fi îmbunătățită în scopul creșterii flexibilității, și astfel se obține abordarea de mai jos:



Aici stratul de web poate conține clase de tip **JavaBeans** (tehnologia precursoare *bean*-urilor de tip *enterprise*). Acestea sunt clase Java ce respectă anumite convenții de proiectare și pot fi reutilizate și transmise între componentele aplicației pentru întreprindere.

Clasele **JavaBeans** încapsulează mai multe obiecte într-unul singur (numit simplu, *bean*). Scopul lor este de ușura reutilizarea componentelor software. Dezvoltatorii pot folosi astfel de componente scrise de alții fără a fi nevoiți să le înțeleagă funcționalitatea internă. Aceste clase au următoarele proprietăți:

- sunt **serializable**
- au un **constructor fără argument**
- proprietățile sunt private, iar accesul la acestea este permis doar prin intermediul **funcțiilor accesor (getters)** și **mutator (setters)**

În continuare, veți crea un formular web ce acceptă câteva date despre un student, le trimite unui servlet spre „procesare”, iar servlet-ul le încapsulează într-un **JavaBean** și le afișează într-o pagină JSP (stilul **Model-View-Controller**).

Creați o clasă Java denumită **StudentBean**, într-un nou pachet numit **beans**, în folderul cu surse **java**. Clasa are următoarea structură:

```
package beans;

public class StudentBean implements java.io.Serializable {
    private String nume = null;
    private String prenume = null;
    private int varsta = 0;

    public StudentBean() {
    }

    public String getNume() {
        return nume;
    }

    public void setNume(String nume) {
        this.nume = nume;
    }
}
```

```

public String getPrenume() {
    return prenume;
}

public void setPrenume(String prenume) {
    this.prenume = prenume;
}

public int getVarsta() {
    return varsta;
}

public void setVarsta(int varsta) {
    this.varsta = varsta;
}
}

```

Se observă că respectă toate caracteristicile unui *JavaBean*:

- Este serializabilă, deoarece implementează interfața **Serializable**
- are un constructor fără argumente
- conține proprietăți **private** accesibile prin metode **getter și setter**

Acum, creați o pagină JSP ce conține un formular prin care se cer datele unui student. Adăugați un fișier de tip „**file**” numit **formular.jsp** în folder-ul **webapp**:

```

<html xmlns:jsp="http://java.sun.com/JSP/Page">
    <head>
        <title>Formular student</title>
        <meta charset="UTF-8" />
    </head>
    <body>
        <h3>Formular student</h3>
        Introduceti datele despre student:
        <form action=".//process-student" method="post">
            Nume: <input type="text" name="nume" />
            <br />
            Prenume: <input type="text" name="prenume" />
            <br />
            Varsta: <input type="number" name="varsta" />
            <br />
            <br />
            <button type="submit" name="submit">Trimite</button>
        </form>
    </body>
</html>

```

Formularul conține trei câmpuri: nume, prenume și vîrstă. Calea spre care sunt trimise datele introduse este **./process-student**. Metoda HTTP prin care datele sunt trimise este **POST**, aşadar, logica de procesare a datelor va fi conținută în metoda **doPost()** a servlet-ului său.

În continuare, creați servlet-ul **ProcessStudentServlet** în folder-ul cu surse **java**. Introduceți codul Java:

```

import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

```

```

import java.io.IOException;
import java.time.Year;

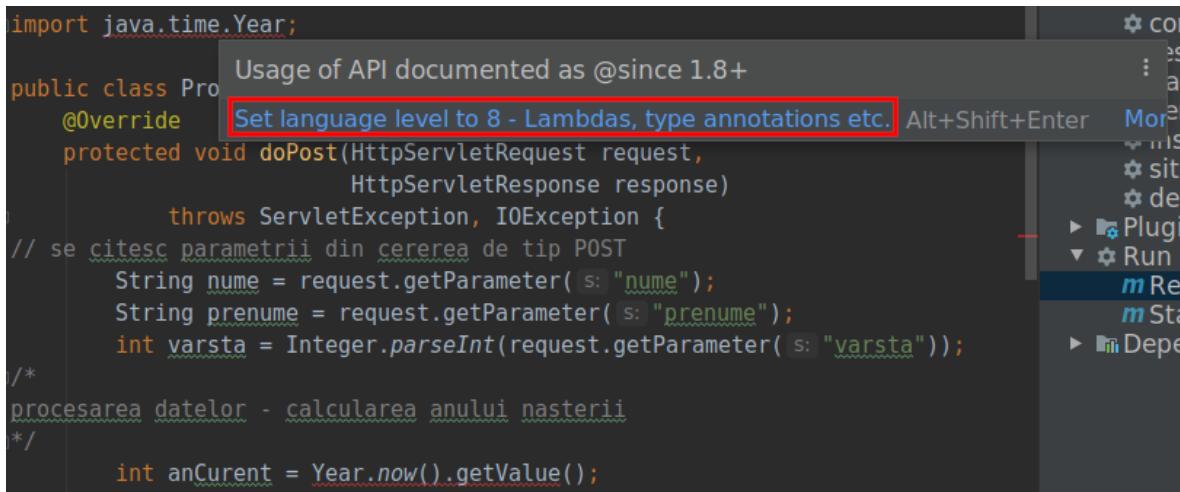
public class ProcessStudentServlet extends HttpServlet {
    @Override
    protected void doPost(HttpServletRequest request,
    HttpServletResponse response)
        throws ServletException, IOException {
        // se citesc parametrii din cererea de tip POST
        String nume = request.getParameter("nume");
        String prenume = request.getParameter("prenume");
        int varsta = Integer.parseInt(request.getParameter("varsta"));

        /*
        procesarea datelor - calcularea anului nasterii
        */
        int anCurent = Year.now().getValue();
        int anNastere = anCurent - varsta;

        // se trimit datele primite si anul nasterii catre o alta
pagina JSP pentru afisare
        request.setAttribute("nume", nume);
        request.setAttribute("prenume", prenume);
        request.setAttribute("varsta", varsta);
        request.setAttribute("anNastere", anNastere);
        request.getRequestDispatcher("./info-
student.jsp").forward(request, response);
    }
}

```

Dacă mediul de dezvoltare nu permite utilizarea `java.time.Year` (permis începând cu Java 1.8), dați click pe clasa importată `java.time.Year`, așteptați câteva secunde sau apăsați ALT+ENTER și selectați „Set language level to 8 ...”, apoi „Import changes” în dreapta-jos.



Servlet-ul preia pur și simplu datele din parametrii cererii HTTP, calculează anul (aproximativ) al nașterii studentului și redirecționează datele către o altă pagină JSP numită `info-student.jsp`.

Mapați servlet-ul corespunzător în `web.xml`:

```

...

```

```

<servlet>
    <servlet-name>ProcessStudent</servlet-name>
    <servlet-class>ProcessStudentServlet</servlet-class>
</servlet>
...
<servlet-mapping>
    <servlet-name>ProcessStudent</servlet-name>
    <url-pattern>/process-student</url-pattern>
</servlet-mapping>
...

```

Așadar, urmează pagina finală, unde se afișează datele redirecționate de servlet-ul intermediu **ProcessStudentServlet**. În folder-ul **webapp**, creați o altă pagină JSP numită **info-student.jsp** și introduceți următorul conținut:

```

<html xmlns:jsp="http://java.sun.com/JSP/Page">
    <head>
        <title>Informatii student</title>
    </head>
    <body>
        <h3>Informatii student</h3>

        <!-- populare bean cu informatii din cererea HTTP -->
        <jsp:useBean id="studentBean" class="beans.StudentBean" />
        <jsp:setProperty name="studentBean" property="nume" value='<%
request.getAttribute("nume") %>'/>
        <jsp:setProperty name="studentBean" property="prenume" value='<%
request.getAttribute("prenume") %>'/>
        <jsp:setProperty name="studentBean" property="varsta" value='<%
request.getAttribute("varsta") %>'/>

        <!-- folosirea bean-ului pentru afisarea informatiilor -->
        <p>Urmatoarele informatii au fost introduse:</p>
        <ul type="bullet">
            <li>Nume: <jsp:getProperty name="studentBean"
property="nume" /></li>
            <li>Prenume: <jsp:getProperty name="studentBean"
property="prenume" /></li>
            <li>Varsta: <jsp:getProperty name="studentBean"
property="varsta" /></li>
            <li>Anul nasterii: <%= request.getAttribute("anNastere")
%></li>
        </ul>
    </body>
</html>

```

Pagina JSP utilizează sintaxa specifică pentru preluarea unui *JavaBean* în contextul de execuție (**jsp:useBean**), setarea unei proprietăți din *bean* (adică apelarea unui *setter* - **jsp:setProperty**) și preluarea unei proprietăți din *bean* (adică apelarea unui *getter* - **jsp:getProperty**). Câmpul calculat în servlet (**anNastere**) nu face parte din *bean*, deci este preluat pur și simplu din cererea HTTP, ca atribut suplimentar.

Alternativ, pentru a popula mai ușor *bean*-ul cu toate datele conținute în cererea HTTP, se poate proceda astfel:

```

<jsp:useBean id="studentBean" class="beans.StudentBean"
scope="request">

```

```
<jsp:setProperty name="studentBean" property="*" />
</jsp:useBean>
```

În această variantă, proprietățile bean-ului vor fi populate automat dacă numele proprietăților încapsulate coincid cu numele atributelor primite în cererea HTTP. Alegeti varianta pe care o preferați.

Adăugați o nouă ancoră HTML în pagina `index.jsp` pentru a avea legătură către formularul creat anterior:

```
...
<p>
    <a href=".//formular.jsp">Formular student</a>
</p>
...
```

Compilați, împachetați și reîncărcați artefactul rezultat pe server-ul *enterprise*. Apoi, accesați următorul URL:

<http://localhost:8080/JEE-Test/>

Apăsați pe a doua ancoră HTML (**Formular student**):



Hello World!

[Acces la primul servlet](#)

[Formular student](#)

Introduceți câteva date în câmpurile de completat și apăsați „Trimite”.



Formular student

Introduceti datele despre student:

Nume: Popescu

Prenume: Ion

Varsta: 23

Trimite

Observați că datele au fost stocate într-un *JavaBean* și preluate spre afișare de o pagină JSP. Servlet-ul intermediar a calculat un al patrulea câmp, anul nașterii (pentru a simula o „procesare” făcută în stratul web).

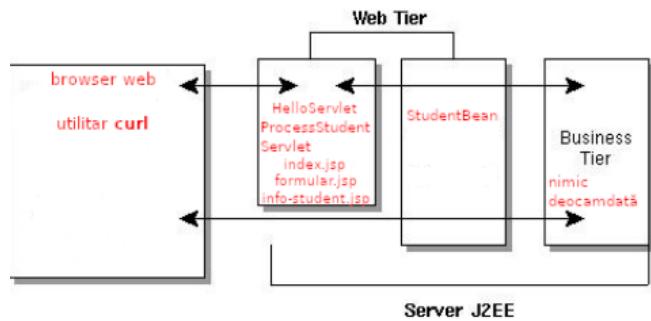


Informatii student

Urmatoarele informatii au fost introduse:

- Nume: Popescu
- Prenume: Ion
- Varsta: 23
- Anul nasterii: 1996

Adăugând funcționalitatea din pașii anteriori, aplicația dvs. are următoarea arhitectură în acest moment:



3. O variantă simplistă de persistență a datelor

Aplicația demonstrativă conține funcționalități de bază ce nu persistă datele care sunt implicate în fluxul de execuție: servlet-ul **ProcessStudentServlet**, spre exemplu, doar preia datele din formular și le trimitе unei pagini JSP pentru afișare.

Pentru finalul laboratorului, veți implementa o variantă simplă de persistență a datelor încapsulate în *JavaBean*-ul **StudentBean**. Mai precis, se dorește ca atunci când utilizatorul accesează formularul pentru introdus datele despre student, completează și apasă pe butonul „Trimite”, servlet-ul să nu doar preia datele și le redirecționează, ci le să salvează (**serializează**) într-un fișier XML. Pentru aceasta, veți folosi, spre exemplu, serializatorul **Jackson 2.x**.

(<https://mvnrepository.com/artifact/com.fasterxml.jackson.dataformat/jackson-dataformat-xml>).

Adăugați **Jackson Dataformat** ca dependență în proiectul Maven, în fișierul **pom.xml**, ca subordonat al tag-ului **<dependencies>**:

```
<dependency>
    <groupId>com.fasterxml.jackson.dataformat</groupId>
    <artifactId>jackson-dataformat-xml</artifactId>
    <version>2.10.1</version>
</dependency>
```

Apoi modificați codul servlet-ului **ProcessStudentServlet**, adăugând următorul bloc de cod înainte de trimiterea datelor către pagina de afișare JSP:

```
...
...

// initializare serializator Jackson
XmlMapper mapper = new XmlMapper();

// creare bean si populare cu date
StudentBean bean = new StudentBean();
bean.setNume(nume);
```

```
bean.setPrenume(prenume);
bean.setVarsta(varsta);

// serializare bean sub forma de string XML
mapper.writeValue(new File("/home/student/opt/1307A/Popescu
Ion/student.xml"), bean);

...
```

XmlMapper este clasa principală utilizată pentru operațiile de serializare / deserializare ale **Jackson 2.x**.

Pentru citirea datelor, creați un nou servlet numit **ReadStudentServlet** care, de această dată, va fi accesat direct din „meniul” de pe pagina principală **index.jsp**. Adăugați următorul conținut în corpul fișierului sursă:

```
import beans.StudentBean;
import com.fasterxml.jackson.dataformat.xml.XmlMapper;

import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.File;
import java.io.IOException;

public class ReadStudentServlet extends HttpServlet {
    @Override
    protected void doGet(HttpServletRequest request,
    HttpServletResponse response) throws ServletException, IOException {
        // deserializare student din fisierul XML de pe disc
        File file = new File("/home/student/opt/1307A/Popescu
Ion/student.xml");

        // se returneaza un raspuns HTTP de tip 404 in cazul in care
nu se gaseste fisierul cu date
        if (!file.exists()) {
            response.sendError(404, "Nu a fost gasit niciun student
serializat pe disc!");
            return;
        }

        XmlMapper xmlMapper = new XmlMapper();
        StudentBean bean = xmlMapper.readValue(file,
        StudentBean.class);

        request.setAttribute("nume", bean.getNume());
        request.setAttribute("prenume", bean.getPrenume());
        request.setAttribute("varsta", bean.getVarsta());

        // redirectionare date catre pagina de afisare a informatiilor
studentului
        request.getRequestDispatcher("./info-
student.jsp").forward(request, response);
    }
}
```

Acest servlet răspunde la cereri HTTP de tip GET, iar corpul metodei de tratare a cererilor face următoarele acțiuni: verifică dacă pe disc există un fișier numit **student.xml** la calea (de exemplu) **/home/student/opt/1307A/Popescu Ion/student.xml**.

Dacă nu există, înseamnă că niciun bean de tip StudentBean nu a fost serializat și atunci se returnează un răspuns de tip **404 Not Found** clientului apelant (cu un mesaj explicativ, bineînțeles).

Dacă fișierul există, se folosește metoda pentru deserializare a Jackson 2.x pentru a popula un obiect de tip StudentBean cu datele extrase din fișier. Având datele în memorie, acestea sunt setate ca atribute ale cererii HTTP ce este redirectionată în continuare către aceeași pagină JSP pentru afișare, **info-student.jsp** (deja existentă).

Nu uitați că servlet-ul trebuie mapat sub o anumită rută de acces în fișierul cu descriptori XML **web.xml**. Adăugați o mapare pentru acest servlet sub calea **/read-student**, astfel:

```
...
<servlet>
    <servlet-name>ReadStudent</servlet-name>
    <servlet-class>ReadStudentServlet</servlet-class>
</servlet>

...
<servlet-mapping>
    <servlet-name>ReadStudent</servlet-name>
    <url-pattern>/read-student</url-pattern>
</servlet-mapping>
```

Observați că acest servlet nu calculează acel câmp suplimentar cu anul nașterii studentului, aşadar ați putea trata acest caz în pagina JSP **info-student.jsp**, modificând modul de afișare al câmpului respectiv:

```
...
<!-- anul nasterii nu face parte din bean, il afisam separat (daca
exista) -->
<li>Anul nasterii: <%
    Object anNastere = request.getAttribute("anNastere");
    if (anNastere != null) {
        out.print(anNastere);
    } else {
        out.print("necunoscut");
    }
%></li>

...
```

În paginile JSP, între simbolurile **<%** și **%>** se poate încapsula cod Java (**scriptleti**). În domeniul paginii JSP este disponibil obiectul **out** ce reprezintă fluxul de ieșire cu care se pot afișa date în pagină.

Pentru preluarea anului nașterii s-a folosit clasa de bază **Object** pentru a nu restrângă generalitatea tipului de date primit ca atribut: se poate primi un număr întreg, un sir de caractere etc.

Nu în ultimul rând, adăugați în meniul principal (pagina **index.jsp**) o nouă ancoră care

face trimitere la noul servlet creat pentru citirea datelor unui student dintr-un fișier XML:

```
...  
<p>  
    <a href=".read-student">Vizualizare student</a>  
</p>  
...
```

Pentru a testa modificările aduse aplicației, urmați pașii: **compile → package → redeploy** (sau „Start server & deploy app” dacă server-ul GlassFish este oprit).

Accesați URL-ul <http://localhost:8080/JEE-Test> și adăugați un student nou prin completarea formularului „Formular student”.

Formular student

Introduceti datele despre student:

Nume: Popescu

Prenume: Ion

Varsta: 23

Trimite

După trimiterea formularului, veți fi redirecționat către pagina de afișare a informațiilor. Datele au fost serializate în fișierul XML (în acest exemplu) **/home/student/opt/1307A/Popescu_Ion/student.xml**.

Informatii student

Urmatoarele informatii au fost introduse:

- Nume: Popescu
- Prenume: Ion
- Varsta: 23
- Anul nasterii: 1997

Verificați acest lucru prin accesarea noii secțiuni „Vizualizare student” din meniul principal. Servlet-ul care vă oferă rezultatele citește datele din acel XML și le pasează paginii JSP, ce vi le afișează în format HTML.

Informatii student

Urmatoarele informatii au fost introduse:

- Nume: Popescu
- Prenume: Ion
- Varsta: 23
- Anul nasterii: necunoscut

Observați că anul nașterii este afișat ca necunoscut, deoarece nu a fost calculat în acest caz. De asemenea, se poate verifica direct, rapid, care date sunt păstrate pe disc în fișierul XML, folosind comanda:

```
$ cat /home/student/opt/1307A/Popescu\ Ion/student.xml
```

Atenție la **backslash spațiu!**

Teme de laborator

- Adăugați posibilitatea de a actualiza / șterge informațiile despre student din fișierul XML. De exemplu, puteți modifica pagina care vizualizează informațiile să completeze automat câmpuri HTML al unui alt formular, pe baza a ceea ce s-a citit din XML. Sub acel formular se poate adăuga un buton de „Actualizare” care va avea ca țintă un servlet sau o pagină JSP creată pentru acest scop.

Teme pe acasă

- Adăugați posibilitatea de a introduce mai multe seturi de informații despre studenți și, de asemenea, de a căuta un anumit student după un criteriu stabilit (de ex. după nume sau prenume), cu următorii pași:
 1. Căutarea studentului în fișierul XML care conține colecția de studenți
 2. Încărcarea datelor care coincid rezultatului căutării în memorie și afișarea lor ca răspuns al unui servlet sau într-o pagină JSP (la alegere).

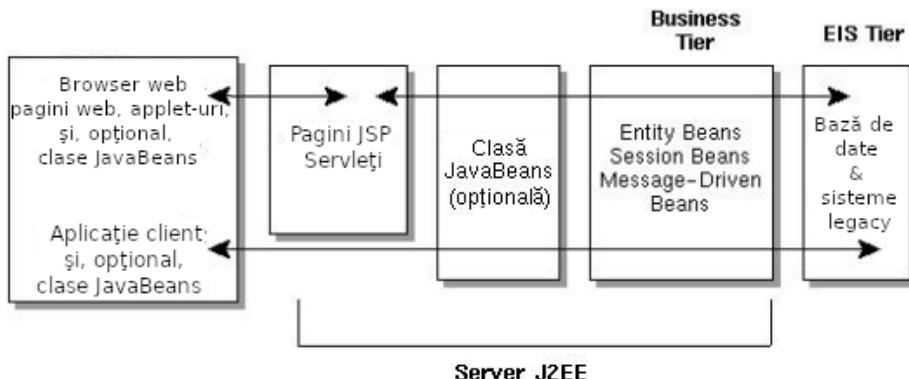
Bibliografie

- [1] Overview JEE - <https://javaee.github.io/firstcup/java-ee001.html>
- [2] Documentație JEE 8 - <https://javaee.github.io/glassfish/documentation>
- [3] Tutorial JEE - <https://javaee.github.io/tutorial/toc.html>
- [4] Documentație Cargo Plugin (+ listă de Maven *goals* disponibile) - <https://codehaus-cargo.github.io/cargo/Maven2+plugin.html>
- [5] Tutorial Oracle JavaBeans - <https://docs.oracle.com/javase/tutorial/javabeans/>
- [6] Java Servlet Technology - <https://docs.oracle.com/javaee/7/tutorial/servlets.htm>
- [7] Java Server Pages Technology - <https://www.oracle.com/technetwork/java/whitepaper-142163.html>

Laboratorul 2

Dezvoltarea componentelor de *business* într-o aplicație pentru întreprindere (JEE)

Introducere și arhitectura JEE



În laboratorul 1 s-a pus accent pe o arhitectură simplificată JEE, ce conține componente ce fac parte din **web tier**, respectiv componente **client** (browser-ul web sau utilitar capabil de a trimite cereri HTTP - `curl`).

În acest laborator, veți lucra la nivel de **business tier**, punând astfel accent pe tipurile de componente **business** din platforma JEE:

- **session beans** - de 3 tipuri:
 - **stateless session beans** → nu mențin o legătură **client ↔ bean**
 - **stateful session beans** → mențin legătura cu clientul apelant (fiecare client cu *stateful session bean*-ul lui)
 - **singleton session beans** → o singură instanță disponibilă la nivel de server *enterprise*
- **entity beans** → încapsulează funcționalitate, respectiv maparea datelor dintr-o bază de date sub formă de obiecte (învechite începând de la EJB 3.0, înlocuite de entitățile de tip **JPA entity** din Java Persistence API)
- **message-driven beans** → permit procesarea mesajelor în mod asincron

Aplicație JEE completă

În laboratorul 1, ați creat un proiect JEE minimal, de tip **Web Application**, cu împachetare **WAR** (Web ARchive). În acest laborator, deoarece urmează să utilizați componentele de **business tier**, veți învăța cum se creează un proiect JEE complet, de tip **Enterprise Application**, cu împachetare **EAR** (Enterprise ARchive) din IntelliJ IDEA Community și cum se adaugă dependențele necesare.

După terminarea orei de laborator, deschideți consola Glassfish: <http://localhost:4848>, dați click pe Applications în partea stângă, bifăți tot în afară de cargoCPC și apăsați pe butonul Undeploy.

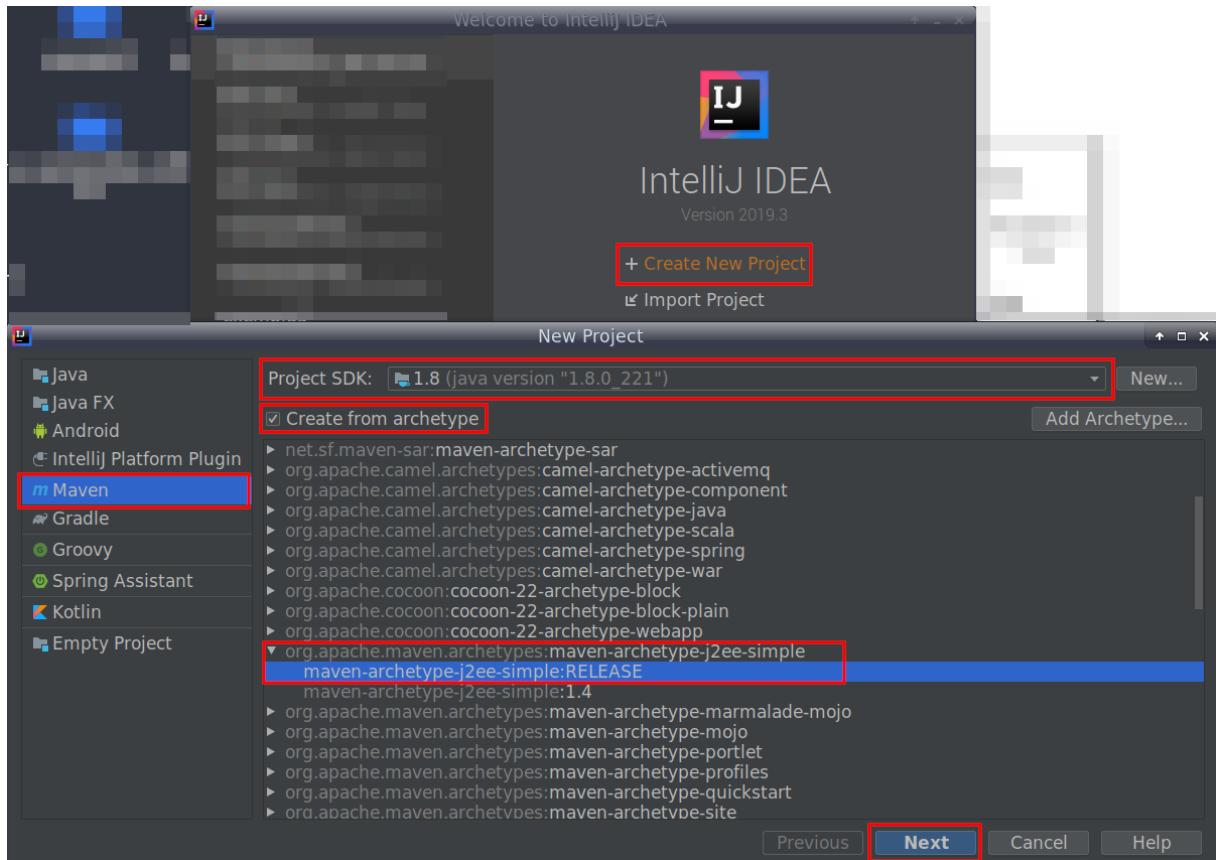
1.1. Creare și configurare proiect JEE complet folosind IntelliJ IDEA Community

1.1.1. Creare proiect IntelliJ

Sisteme Distribuite Laboratorul 2

Deschideți IntelliJ IDEA Community, iar în meniul din partea dreaptă alegeti „Create new project”.

În fereastra de selecție a tipului de proiect, alegeti „Maven” în partea stângă, apoi selectați versiunea de **Java SDK 1.8**. Bifați „Create from archetype”, iar din lista de arhetipuri disponibile, expandați `org.apache.maven.archetypes:maven-archetype-j2ee-simple` și selectați `maven-archetype-j2ee-simple:RELEASE`. Click pe „Next”.

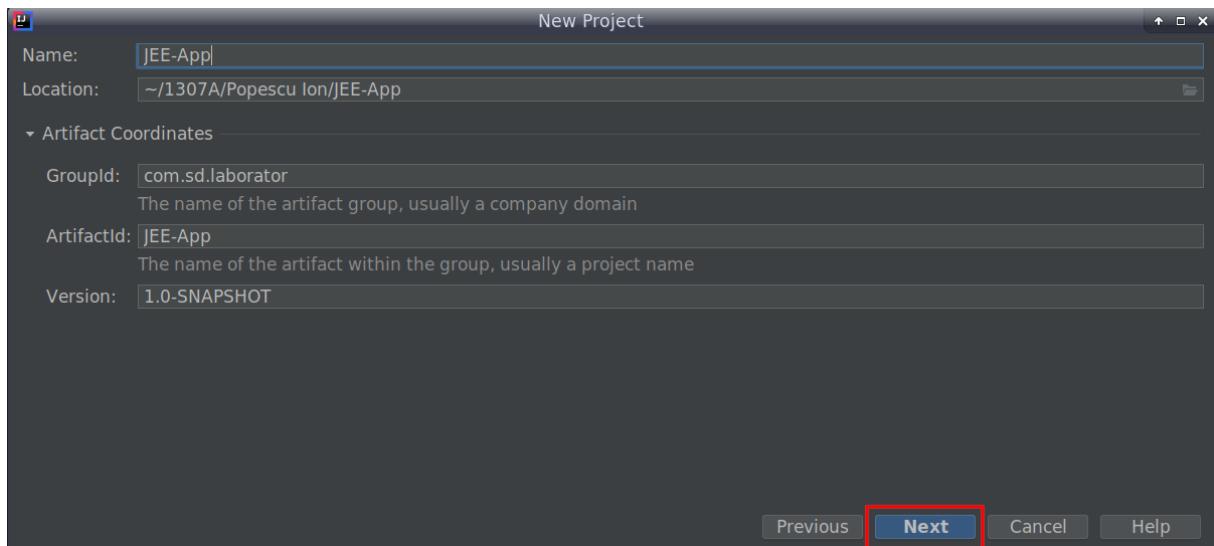


În continuare, se aleg numele și locația proiectului pe disc, precum și detaliile artefactului EAR rezultat. Secțiunea „**Artifact Coordinates**” poate fi lăsată cu valorile implicate, sau puteți completa, dacă doriți, **GroupId**-ul cu o valoare personalizată, cum ar fi **com.sd.laborator**.

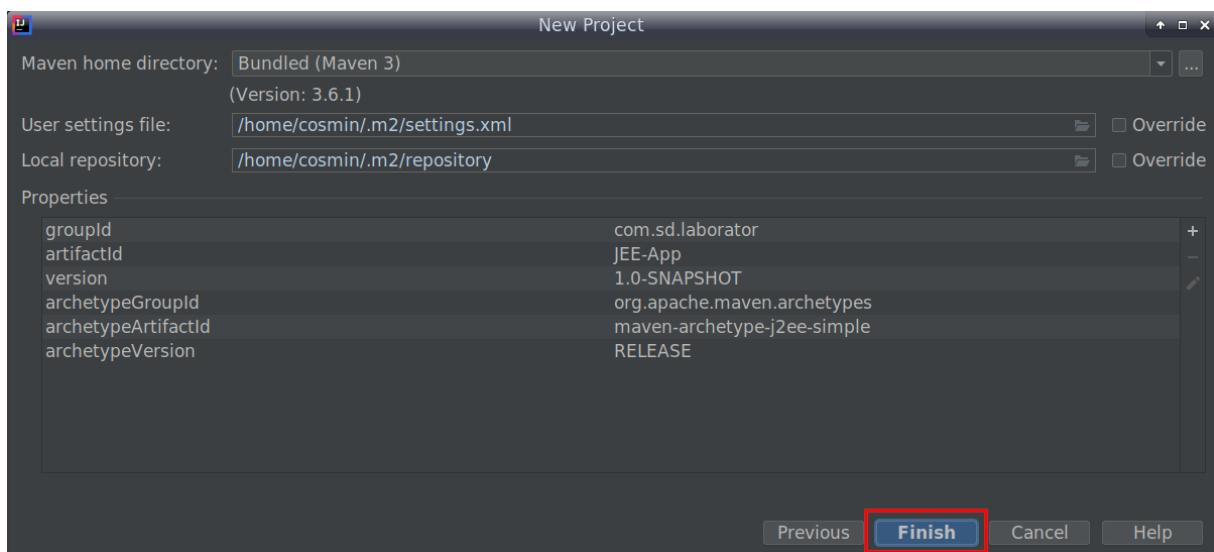
În acest exemplu, proiectul se va numi „**JEE-App**”, iar locația va fi **~/1307A/Popescu Ion/JEE-App**.

Click pe „Next” după completarea datelor menționate.

Sisteme Distribuite Laboratorul 2



În următoarea fereastră, se lasă totul neschimbă și se apasă „Finish”.



Așteptați ca Maven să termine de generat structura proiectului și să aducă primele dependențe definite în configurația arhetipului.

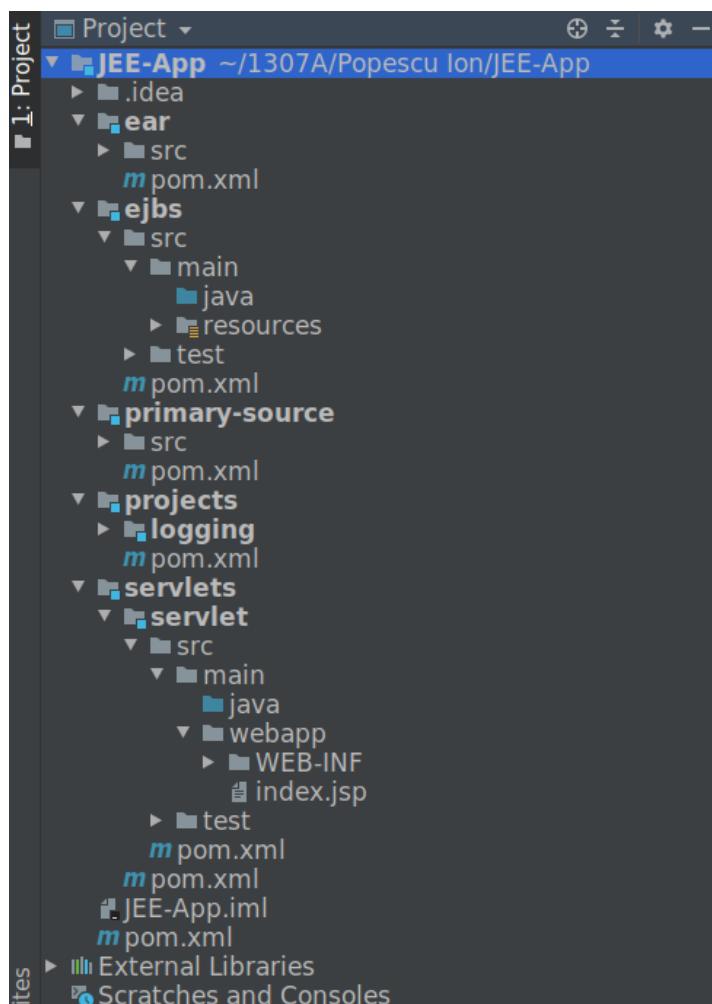
1.1.2. Configurare proiect Maven

Proiectul Maven generat de arhetipul pentru JEE este organizat în module, fiecare modul conținând anumite tipuri de componente, după cum urmează:

- **ear** → generează artefactul EAR ce încapsulează toate celelalte module ale proiectului sub formă de arhivă Enterprise ARchive.
- **ejb** → conține codul de *business* al claselor *Enterprise Java Beans*. Aici veți pune clasele EJB, organizate eventual în pachete.
- **primary-source** → conține clase adiționale utilizate în proiect (clase care nu reprezintă neapărat componente specifice JEE)
- **projects** → conține eventuale subproiecte ale proiectului JEE de bază. Are deja adăugat un schelet de subproiect denumit **logging**.
- **servlets** → conține un submodul numit **servlet**, care reprezintă punctul de intrare al componentei web (nivelul *web tier*). Aici puteți adăuga clase servlet, pagini JSP, etc.

Calea implicită de acces a folder-ului rădăcină a componentei web este /servlet.
Așadar, pentru a vizualiza pagina index.jsp creată automat, după încărcarea aplicației, se accesează URL-ul: <http://localhost:8080/servlet/>.

Structura de proiect arată ca în figură:



Observați că fiecare modul are fișierul său de configurare **pom.xml** (**Project Object Model**), deoarece se pot declara dependențe între modulele componente, respectiv, pentru fiecare modul în parte se pot face configurații personalizate, se pot declara dependențe și folosi *plugin-uri* personalizate etc.

Adăugați dependența **JavaEE API** (<https://mvnrepository.com/artifact/javax/javaee-api/8.0>) în fiecare fișier **pom.xml** al modulelor **ejbs**, **primary-source** și **servlet** (**atenție, nu servlets!**). Așadar, ca subordonat al *tag-ului <dependencies>* din fișierele **pom.xml**, adăugați următorul element:

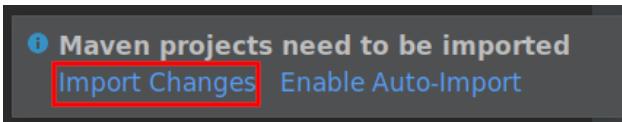
```
<dependency>
  <groupId>javax</groupId>
  <artifactId>javaee-api</artifactId>
  <version>8.0.1</version>
  <scope>provided</scope>
</dependency>
```

Sisteme Distribuite Laboratorul 2

The screenshot shows the IntelliJ IDEA interface with the project structure on the left and the pom.xml file content on the right. The project structure includes modules like JEE-App, ejbs, servlets, and primary-source, each with its own pom.xml file highlighted with a red box. The pom.xml file content on the right shows dependencies for javax.api and javaee-api, with the latter dependency highlighted by a red box.

```
4 xsi:schemaLocation="http://maven.apache.org/PO
5 <modelVersion>4.0.0</modelVersion>
6
7 <parent>
8   <groupId>com.sd.laborator</groupId>
9   <artifactId>servlets</artifactId>
10  <version>1.0-SNAPSHOT</version>
11 </parent>
12
13 <artifactId>servlet</artifactId>
14 <packaging>war</packaging>
15
16 <name>servlet</name>
17
18 <dependencies>
19   <dependency>
20     <groupId>com.sd.laborator</groupId>
21     <artifactId>primary-source</artifactId>
22     <scope>provided</scope>
23   </dependency>
24   <dependency>
25     <groupId>javax</groupId>
26     <artifactId>javaee-api</artifactId>
27     <version>8.0.1</version>
28     <scope>provided</scope>
29   </dependency>
30 </dependencies>
31
32 </project>
```

După modificările făcute în fișierele **pom.xml**, nu uitați să sincronizați modificările, apăsând „**Import changes**” când IntelliJ vă cere acest lucru:



1.1.3. Completare descriptori XML lipsă

Arhetipul din care proiectul Maven a fost creat nu adaugă conținut în descriptorii XML ai componentelor web, respectiv EJB. Aceștia trebuie adăugați manual, astfel:

Dechideți **servlets/servlet/src/main/webapp/WEB-INF/web.xml** și adăugați următorul conținut:

```
<!DOCTYPE web-app PUBLIC
"-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
"http://java.sun.com/dtd/web-app_2_3.dtd" >

<web-app>
  <display-name>Web Component App</display-name>
</web-app>
```

Apoi deschideți **ejbs/src/main/resources/META-INF/ejb-jar.xml** și adăugați următorul conținut:

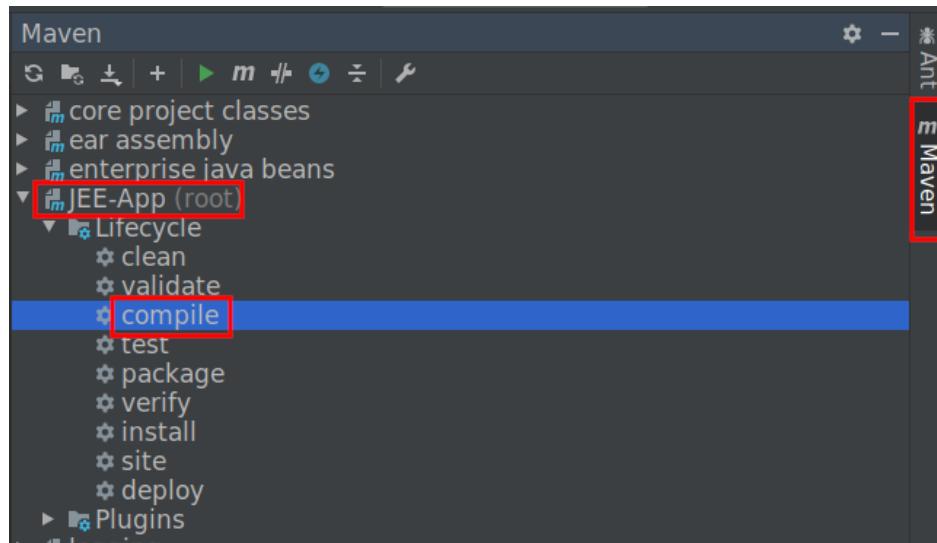
```
<ejb-jar xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
  http://java.sun.com/xml/ns/javaee/ejb-jar_3_0.xsd"
```

```
version="3.0">  
</ejb-jar>
```

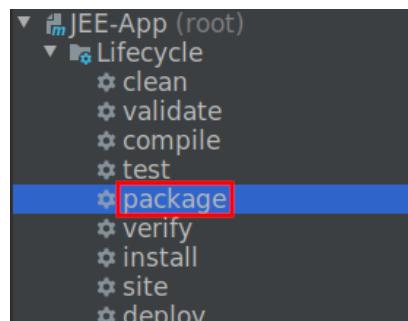
1.2. Compilarea și împachetarea aplicației JEE

Operațiile care pot fi făcute asupra proiectului sunt disponibile sub formă de *Maven lifecycles* în partea dreaptă a ferestrei IntelliJ, în conținutul panoului **Maven**.

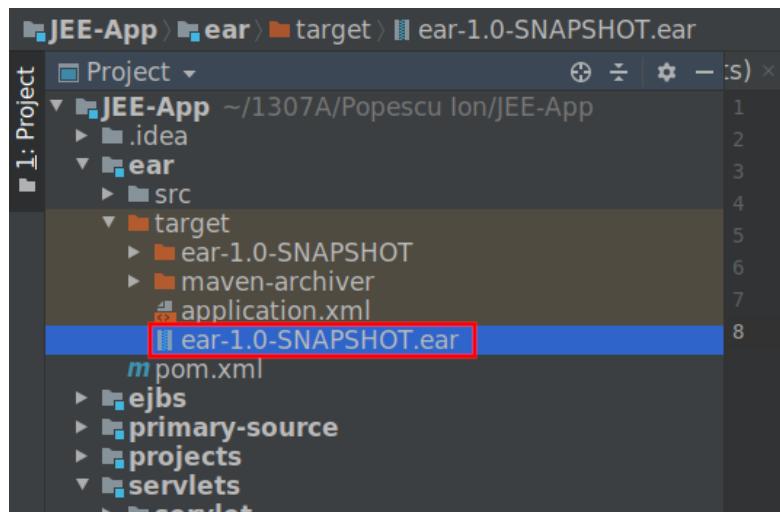
Pentru a compila proiectul, se expandează panoul Maven și se utilizează *lifecycle-ul compile* al modulului Maven care cuprinde întregul proiect, adică cel denumit exact ca și proiectul IntelliJ: **<nume_proiect> (root)**.



Tot din același modul Maven se poate împacheta structura aplicației rezultate în urma compilării (conținutul folder-ului **target**) folosind *lifecycle-ul package*.

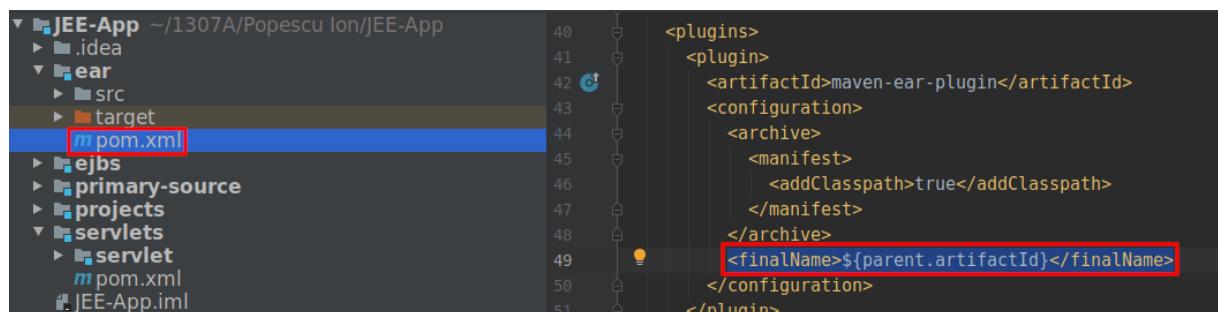


Acest pas va genera un fișier **EAR** (Enterprise ARchive) ca subordonat al folder-ului **target** din modulul **ear**, vizibil în structura de proiect din partea stângă a ferestrei IntelliJ. Artefactul va fi denumit, în mod implicit, sub forma **ear-<versiune>.ear**.



Numele artefactului rezultat este stabilit în mod implicit conform regulii: **<nume_modul_ear>-<versiune>.ear**. Pentru a vă ușura munca atunci când veți încărca aplicația pe server, configurați modulul ear astfel încât să exporte artefactul EAR sub un nume mai „prietenos”, și anume cel al proiectului părinte. Deschideți **ear/pom.xml** și adăugați următoarea configurație ca și subordonat al *tag-ului <configuration>* pentru *plugin-ul maven-ear-plugin*:

```
<finalName>${parent.artifactId}</finalName>
```



După ce împachetați din nou aplicația (cu *lifecycle-ul package*), veți observa că rezultă un artefact EAR sub numele de **JEE-App.ear** (mult mai sugestiv ca cel vechi, implicit). Cel vechi poate fi șters.

1.3. Curățarea proiectului

Tot din panoul Maven se poate curăța folder-ul cu fișiere compilate prin utilizarea *lifecycle-ului clean*. Aceasta va șterge tot conținutul folder-elor **target** al modulelor componente. Dacă se dorește curățarea doar a anumitor module, se execută *lifecycle-ul clean* din secțiunea corespunzătoare acestora din panoul Maven.

1.4. Pornirea server-ului GlassFish

Dintr-o sesiune de terminal, executați următoarea comandă:

```
<LOCATIE_SERVER_GLASSFISH>/bin/asadmin start-domain
```

De exemplu, în acest caz, server-ul fiind localizat în **/home/student/opt/glassfish5**, comanda este:

```
~/opt/glassfish5/bin/asadmin start-domain
```

```
- cosmin@debian-gl553v: ~/glassfish5/bin
File Edit View Terminal Tabs Help
cosmin@debian-gl553v:~/glassfish5/bin$ ./asadmin start-domain
Waiting for domain1 to start .....
Successfully started the domain : domain1
domain Location: /home/cosmin/glassfish5/glassfish/domains/domain1
Log File: /home/cosmin/glassfish5/glassfish/domains/domain1/logs/server.log
Admin Port: 4848
Command start-domain executed successfully.
cosmin@debian-gl553v:~/glassfish5/bin$
```

1.5. Oprirea server-ului GlassFish

Într-o sesiune de terminal, se execută comanda:

```
<LOCATIE_SERVER_GLASSFISH>/bin/asadmin stop-domain
```

În acest caz, ar fi:

```
~/opt/glassfish5/bin/asadmin stop-domain
```

1.6. Încărcarea proiectului (deploy) pe server-ul GlassFish

Deoarece s-a utilizat un arhetip de proiect JEE modular, încărcarea artefactului EAR pe server-ul de aplicații *enterprise* se poate face mai facil din linia de comandă, decât cu *plugin*-ul Cargo, utilizat în laboratorul 1.

Atenție: nu puteți încărca o aplicație în format EAR pe server-ul GlassFish dacă nu există cel puțin un tip de *Enterprise Bean* creat și descris în `ejb-jar.xml`. Deci, nu veți putea utiliza efectiv comenziile descrise în cele ce urmează decât după ce adăugați *bean*-uri.

După ce ați compilat și împachetat aplicația sub formă de artefact EAR, click dreapta pe folder-ul **ear** → **Open in Terminal**. Se va deschide o sesiune de terminal în partea de jos a ferestrei IntelliJ.

Încărcarea proiectului (prima dată) se face cu următoarea comandă:

```
<LOCATIE_SERVER_GLASSFISH>/bin/asadmin deploy
/CALE/CĂTRE/ARTEFACT/<NUME_ARTEFACT>.ear
```

De exemplu:

```
~/opt/glassfish5/bin/asadmin deploy ./target/JEE-App.ear
```

```
Terminal: Local x Local (2) x Local (3) x +
cosmin@debian-gl553v:~/1307A/Popescu_Ion/JEE-App/ear$ ls
pom.xml  src  target
cosmin@debian-gl553v:~/1307A/Popescu_Ion/JEE-App/ear$ /home/cosmin/glassfish5/bin/asadmin deploy ./target/JEE-App.ear
Application deployed with name JEE-App.
Command deploy executed successfully.
cosmin@debian-gl553v:~/1307A/Popescu_Ion/JEE-App/ear$
```

1.7. Stergerea proiectului (undeploy) de pe server-ul GlassFish

Dacă se dorește ca aplicația enterprise să fie stearsă de pe server, se poate folosi comanda:

```
<LOCATIE_SERVER_GLASSFISH>/bin/asadmin undeploy <NUME_APICATIE>
```

Exemplu:

```
~/opt/glassfish5/bin/asadmin undeploy JEE-App
```

1.8. Reîncărcarea proiectului (redeploy) pe server-ul GlassFish

După ce încărcați pentru prima dată o aplicație JEE pe server, dacă actualizați fișierele sursă și doriți să vedeți modificările, trebuie să reîncărcați artefactul nou rezultat în urma împachetării. Deoarece artefactul are același nume după reîmpachetare (dacă nu îl modificați), server-ul nu vă permite să suprascrieți aplicația veche, decât folosind o comandă de **reîncărcare (redeploy)**:

```
<LOCATIE_SERVER_GLASSFISH>/bin/asadmin redeploy <NUME_APLICATIE>
/CALE/CĂTRE/ARTEFACT/<NUME_ARTEFACT>.ear
```

Exemplu:

```
~/opt/glassfish5/bin/asadmin redeploy --name JEE-App ./target/JEE-
App.ear
```

```
Terminal: Local × Local (2) × Local (3) × +
cosmin@debian-gl553v:~/1307A/Popescu_Ion/JEE-App/ear$ /home/cosmin/glassfish5/bin/asadmin redeploy --name JEE-App ./target/JEE-App.ear
Application deployed with name JEE-App.
Command redeploy executed successfully.
cosmin@debian-gl553v:~/1307A/Popescu_Ion/JEE-App/ear$
```

Dacă încercați să folosiți comanda **deploy** pentru un artefact deja existent (cu același nume), veți primi o eroare de tipul:

```
Terminal: Local × Local (2) × Local (3) × +
cosmin@debian-gl553v:~/1307A/Popescu_Ion/JEE-App/ear$ /home/cosmin/glassfish5/bin/asadmin deploy ./target/JEE-App.ear
remote failure: Error occurred during deployment: Application with name JEE-App is already registered. Either specify that redeployment must be forced, or redeploy the application. Or if this is a new deployment, pick a different name. Please see server.log for more details.
Command deploy failed.
cosmin@debian-gl553v:~/1307A/Popescu_Ion/JEE-App/ear$
```

Stateless Session Beans

Stateless Session Bean-ul este un *enterprise bean* care, de obicei, efectuează operații independente de clientul apelant (nu creează nicio legătură bean ↔ client, nu păstrează starea conversației dintre cele 2 părți participante).

Pentru a exemplifica acest tip de *enterprise bean*, veți expune clientului o interfață simplă prin care poate prelua date curentă de la server și prin care poate aduna 2 numere întregi. Această interfață trebuie cunoscută atât de server-ul de aplicații *enterprise* (ca să știe ce metode trebuie să expună în momentul încărcării aplicației), cât și de clientul apelant (pentru ca el să știe ce metode sunt disponibile pentru apel).

Folosiți în continuare proiectul JEE creat în pașii anteriori.

1.9. Codul de business

Adăugați un pachet nou în folder-ul **ejbs/src/main/java**, denumit **interfaces**. În pachetul respectiv, creați o interfață Java numită **StatelessSessionBeanRemote**, cu următorul conținut:

```
package interfaces;

public interface StatelessSessionBeanRemote {
    String getCurrentTime();
```

```
    Integer addNumbers(Integer a, Integer b);  
}
```

Acum, veți crea implementarea conform interfeței, corpul efectiv al *stateless session bean*-ului. În folder-ul **ejb/src/main/java**, creați un pachet denumit **ejb** (Enterprise Java Beans). În pachetul respectiv, creați o clasă Java denumită **StatelessSessionBeanImpl**, cu următorul conținut:

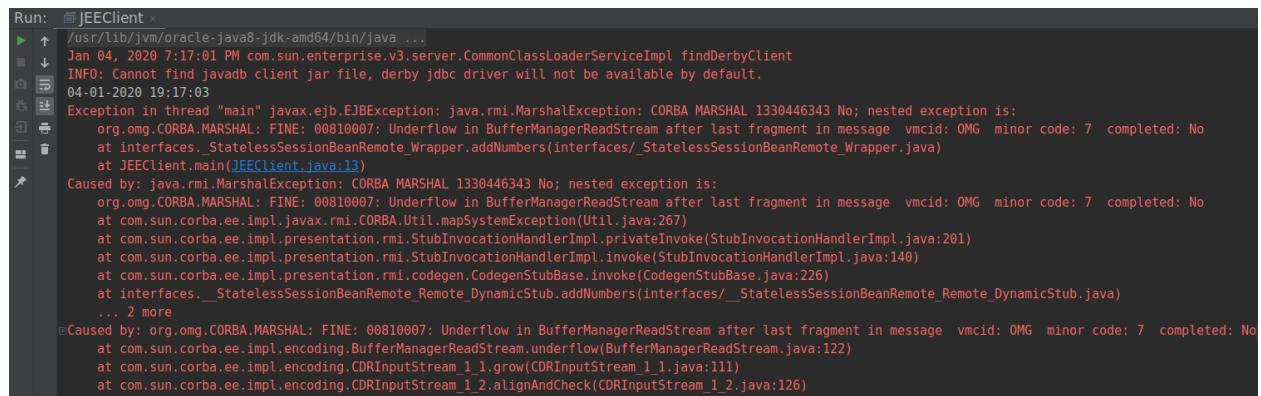
```
package ejb;  
  
import interfaces.StatelessSessionBeanRemote;  
  
import java.io.Serializable;  
import java.text.SimpleDateFormat;  
import java.util.Date;  
  
public class StatelessSessionBeanImpl implements  
StatelessSessionBeanRemote, Serializable {  
    public StatelessSessionBeanImpl() {  
        System.out.println("[Glassfish] S-a instantiat un stateless  
session bean: " +  
                           StatelessSessionBeanImpl.class.getName());  
    }  
  
    public String getCurrentTime() {  
        System.out.println("[Glassfish] S-a apelat metoda  
getCurrentTime()");  
        Date date = new Date();  
        SimpleDateFormat formatter = new SimpleDateFormat("dd-MM-yyyy  
HH:mm:ss");  
        return formatter.format(date);  
    }  
  
    public Integer addNumbers(Integer a, Integer b) {  
        System.out.println("[Glassfish] S-a apelat metoda addNumbers(" +  
+ a + ", " + b + ")");  
        return a + b;  
    }  
}
```

Clasa ce reprezintă acest bean trebuie să implementeze interfața **Serializable**, deoarece codul va fi apelat prin RMI (**Remote Method Execution**) de la o aplicație client complet decuplată de server, și atunci entitățile conținute de acea clasă (inclusiv clasa însăși) trebuie să fie serializabile.

Atenție: nu folosiți tipuri primitive de date în *enterprise beans* care vor fi apelate prin RMI (*bean-uri nelocale!*)! Tipurile primitive de date (int**, **char**, **double** etc.) nu sunt serializabile. Folosiți, în schimb, clasele lor corespondente Java: **Integer**, **Char**, **Double** etc. De aceea, pentru metoda **addNumbers**, s-au folosit parametri de tip **Integer**, și nu **int**, atât în interfață, cât și în implementare.**

Dacă *bean-urile enterprise* conțin tipuri primitive de date, clientul apelant va primi eroare de acest tip:

Sisteme Distribuite Laboratorul 2



```
Run: JEEClient x
/usr/lib/jvm/oracle-java8-jdk-amd64/bin/java ...
Jan 04, 2020 7:17:01 PM com.sun.enterprise.v3.server.CommonClassLoaderServiceImpl findDerbyClient
INFO: Cannot find javadb client jar file, derby jdbc driver will not be available by default.
04-01-2020 19:17:03
Exception in thread "main" javax.ejb.EJBException: java.rmi.MarshalException: CORBA MARSHAL 1330446343 No; nested exception is:
  org.omg.CORBA.MARSHAL: FINE: 00810007: Underflow in BufferManagerReadStream after last fragment in message vmcid: OMG minor code: 7 completed: No
    at interfaces._StatelessSessionBeanRemote_Wrapper.addNumbers(interfaces/_StatelessSessionBeanRemote_Wrapper.java)
    at JEEClient.main(JEEClient.java:13)
Caused by: java.rmi.MarshalException: CORBA MARSHAL 1330446343 No; nested exception is:
  org.omg.CORBA.MARSHAL: FINE: 00810007: Underflow in BufferManagerReadStream after last fragment in message vmcid: OMG minor code: 7 completed: No
    at com.sun.corba.ee.impl.javax.rmi.CORBA.Util.mapSystemException(Util.java:267)
    at com.sun.corba.ee.impl.presentation.rmi.StubInvocationHandlerImpl.privateInvoke(StubInvocationHandlerImpl.java:201)
    at com.sun.corba.ee.impl.presentation.rmi.StubInvocationHandlerImpl.invoke(StubInvocationHandlerImpl.java:140)
    at com.sun.corba.ee.impl.presentation.rmi.CodegenStubBase.invoke(CodegenStubBase.java:226)
    at interfaces._StatelessSessionBeanRemote_Remote_DynamicStub.addNumbers(interfaces/_StatelessSessionBeanRemote_Remote_DynamicStub.java)
... 2 more
Caused by: org.omg.CORBA.MARSHAL: FINE: 00810007: Underflow in BufferManagerReadStream after last fragment in message vmcid: OMG minor code: 7 completed: No
  at com.sun.corba.ee.impl.encoding.BufferManagerReadStream.underflow(BufferManagerReadStream.java:122)
  at com.sun.corba.ee.impl.encoding.CDRInputStream_1_1.grow(CDRInputStream_1_1.java:111)
  at com.sun.corba.ee.impl.encoding.CDRInputStream_1_2.alignAndCheck(CDRInputStream_1_2.java:126)
```

În continuare, folosind descriptori XML, trebuie să îi specificăm server-ului de aplicații *enterprise* ce reprezintă clasa creată mai sus (un *bean stateless*), cum se expune aceasta clientului, unde poate găsi interfața pe care o implementează, respectiv unde găsește clasa cu implementarea. Așadar, în folder-ul `ejbs/src/main/resources/META-INF`, adăugați următorul conținut în fișierul `ejb-jar.xml`:

```
<ejb-jar xmlns="http://java.sun.com/xml/ns/javaee"
           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
           xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
           http://java.sun.com/xml/ns/javaee/ejb-jar_3_0.xsd"
           version="3.0">
  <enterprise-beans>
    <session>
      <ejb-name>StatelessSessionBeanExample</ejb-name>
      <mapped-name>ssb-example</mapped-name>
      <business-
local>interfaces.StatelessSessionBeanRemote</business-local>
      <business-
remote>interfaces.StatelessSessionBeanRemote</business-remote>
      <ejb-class>ejb.StatelessSessionBeanImpl</ejb-class>
      <session-type>Stateless</session-type>
    </session>
  </enterprise-beans>
</ejb-jar>
```

În acest XML, se declară faptul că aplicația conține un *session bean* (tag-ul `<session>`), cu următoarele proprietăți:

- un nume descriptiv (**nu are legătură cu numele clasei!**): tag-ul `<ejb-name>`
- identificatorul sub care *bean*-ul este mapat în JNDI (*Java Naming and Directory Interface*): tag-ul `<mapped-name>`
- interfețele care sunt implementate de clasa ce reprezintă *bean*-ul (locală - `business-local`, respectiv de la distanță - `business-remote`), date sub forma `<pachet>.〈nume_clasă〉`.
- numele clasei care conține implementarea *bean*-ului, dat sub forma `<pachet>.〈nume_clasă〉`: tag-ul `<ejb-class>`
- tipul de *bean*: tag-ul `<session-type>`

Faceți următoarele operațiuni asupra proiectului: **clean → compile → package → deploy de la consolă** (nu uitați să porniți server-ul GlassFish, dacă nu este deja pornit).

Dacă totul a decurs bine, veți regăsi în consola de administrare GlassFish bean-ul creat printre celealte componente deja existente pe server:

Modules and Components (5)					
Module Name	Engines	Component Name	Type	Action	
com.sd.laborator-ejbs-1.0-SNAPSHOT.jar	[ejb]	-----	-----		
com.sd.laborator-ejbs-1.0-SNAPSHOT.jar		StatelessSessionBeanExample	StatelessSessionBean		
com.sd.laborator-servlet-1.0-SNAPSHOT.war	[web]	-----	-----		
com.sd.laborator-servlet-1.0-SNAPSHOT.war		default	Servlet		
com.sd.laborator-servlet-1.0-SNAPSHOT.war		jsp	Servlet		

1.10. Extragerea numelui JNDI al bean-ului

Server-ul de aplicații enterprise expune bean-ul respectiv printr-un nume JNDI mapat conform a ceea ce s-a specificat în descriptorul XML (conținutul tag-ului **<mapped-name>**). Însă, un client care cauță și utilizează bean-ul de pe server are nevoie de numele complet JNDI stabilit de GlassFish, conform specificațiilor proprii. Acest nume poate fi găsit în log-ul server-ului, disponibil la locația următoare:

```
<LOCATIE_SERVER_GLAFFISH>/glassfish/domains/domain1/logs/server.log
```

În acest caz, log-ul se află în locația:

```
/home/student/opt/glassfish5/glassfish/domains/domain1/logs/server.log
```

Deschideți log-ul cu un editor de text și căutați numele mapat bean-ului pe care l-ați creat (în acest caz, căutați „**ssb-example**”).

```
9993 [2020-01-02T18:41:25.807+0200] [glassfish 5.0] [INFO] [AS-EJB-00054] [javax.enterprise.ejb.container] [tid: _ThreadID=44
9994 ThreadName=admin-listener(3)] [timeMillis: 1577983285807] [levelValue: 800] [[
9995 Portable JNDI names for EJB StatelessSessionBeanImpl: [java:global/JEE-Test/
9996 StatelessSessionBeanImpl!interfaces.StatelessSessionBeanRemote, java:global/JEE-Test/StatelessSessionBeanImpl]]
9997 [2020-01-02T18:41:25.807+0200] [glassfish 5.0] [INFO] [AS-EJB-00055] [javax.enterprise.ejb.container] [tid: _ThreadID=44
9998 ThreadName=admin-listener(3)] [timeMillis: 1577983285807] [levelValue: 800] [[
9999 Glassfish-specific (Non-portable) JNDI names for EJB StatelessSessionBeanImpl: [ssb-example,
0000 ssb-example#interfaces.StatelessSessionBeanRemote]]]
0001 [2020-01-02T18:41:25.845+0200] [glassfish 5.0] [INFO] [AS-WEB-GLUE-00172] [javax.enterprise.web] [tid: _ThreadID=44
0002 ThreadName=admin-listener(3)] [timeMillis: 1577983285845] [levelValue: 800] [[
0003 Loading application [JEE-Test] at [/JEE-Test]]]
```

Notați numele complet JNDI al bean-ului atribuit de server (**de tip Glassfish-specific, nu cel portabil!**), veți avea nevoie de el în aplicația client. În acest caz:

```
ssb-example#interfaces.StatelessSessionBeanRemote
```

La modul general, numele JNDI specific GlassFish al unui bean enterprise atribuit de de server este:

```
<NUME_MAPAT_ÎN_DESCRIPTOR>#<NUME_PACHET>.<NUME_INTERFAȚĂ>
```

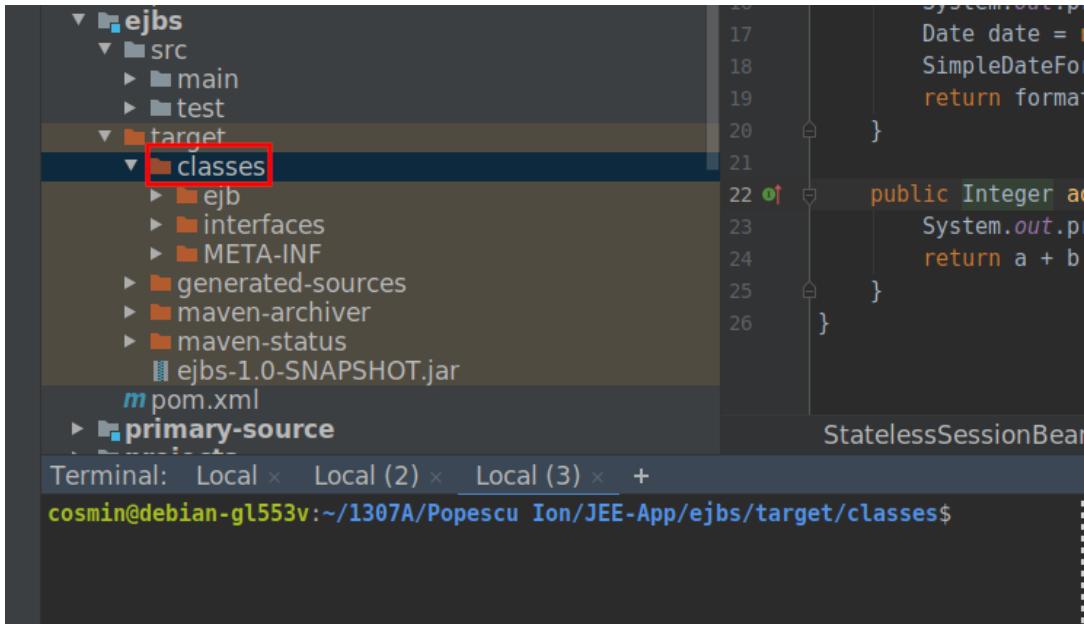
1.11. Aplicație client

Clientul va fi, în acest caz, complet decuplat de server, deci aplicația client care va utiliza bean-ul creat anterior va fi executată sub propriul JVM. Așadar, clientul trebuie să aibă acces (cumva) la interfața bean-ului, pentru a putea apela metodele din acesta, de la distanță.

O soluție în acest sens ar fi să împachetați sub formă de librărie JAR (**Java ARchive**) codul obiect rezultat după compilarea fișierului sursă ce conține interfața bean-ului. Acest fișier nu conține implementarea, deci are sens să fie distribuit și clientului spre utilizare (de aici decuplarea).

Sisteme Distribuite Laboratorul 2

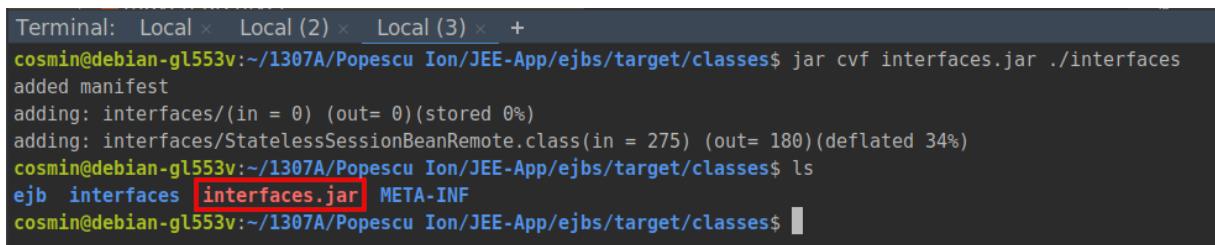
Deschideți un terminal în folder-ul **ejbs/target/classes**. Din IntelliJ, se poate face cu click dreapta pe acest folder în structura de proiect din partea stângă → **Open in Terminal**. Se va deschide un terminal în partea de jos a ferestrei IntelliJ.



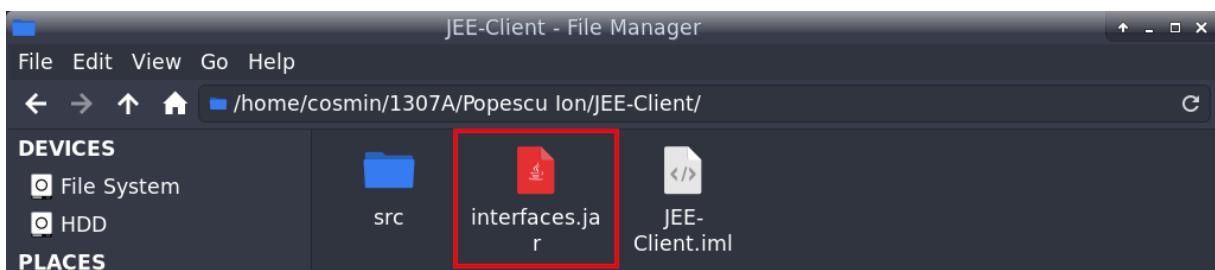
Execuați următoarea comandă:

```
jar cvf interfaces.jar ./interfaces
```

Comanda va împacheta conținutul folder-ului **interfaces** într-un fișier JAR.



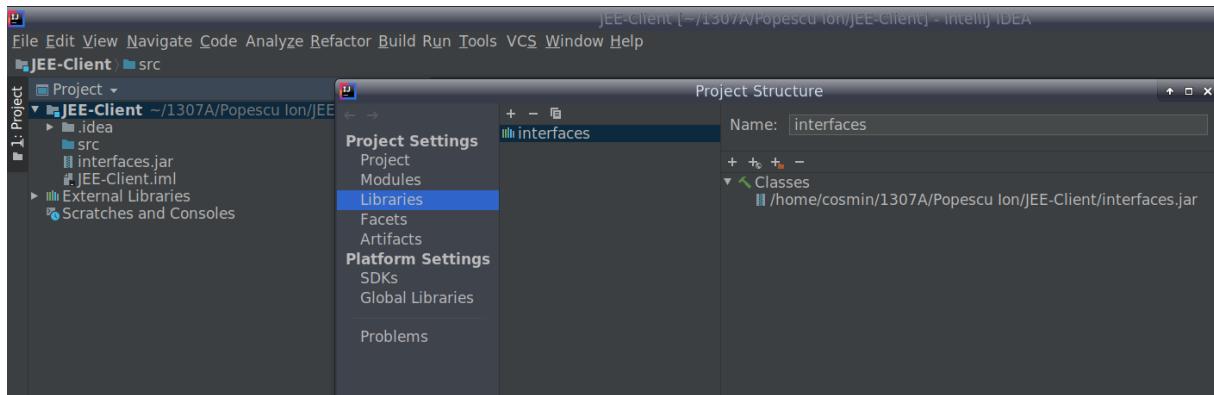
Creați un proiect nou IntelliJ de tip Java (simplu, fără manager de proiect Maven, etc.). Denumiți-l, spre exemplu, **JEE-Client**. Copiați fișierul **interfaces.jar** în folder-ul rădăcină al proiectului.



Acum, trebuie să adăugați acest fișier ca librărie de care proiectul **JEE-Client** depinde (ca aplicația client să poată utiliza interfața **StatelessSessionBeanRemote**). Accesați: **File** → **Project Structure...** → **Libraries** → click pe pictograma în formă de plus (**New Project**

Sisteme Distribuite Laboratorul 2

Library) → Java → căutați și selectați fișierul `interfaces.jar` copiat în pasul anterior → în fereastra „Choose modules” apăsați doar **OK → apăsați **OK** în fereastra cu setările proiectului.**



Creați o clasă Java în folder-ul `src` cu surse, denumită `JEEClient`, având următorul conținut:

```
import interfaces.StatelessSessionBeanRemote;

import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;

public class JEEClient {
    public static void main(String[] args) throws NamingException {
        Context ctx = new InitialContext();
        StatelessSessionBeanRemote ssb = (StatelessSessionBeanRemote)
ctx.lookup("ssb-example#interfaces.StatelessSessionBeanRemote");
        System.out.println(ssb.getCurrentTime());
        System.out.println("1 + 3 = " + ssb.addNumbers(1, 3));
    }
}
```

În codul de mai sus, se importă interfața *bean*-ului de tip *stateless session*. Apoi, se inițializează un context inițial pentru operația de căutare în JNDI ce urmează (este necesară această inițializare, deoarece toate operațiile de căutare în JNDI sunt relative la un context).

Apoi, se caută în JNDI bean-ul expus sub numele pe care l-ați găsit și notat anterior. După ce s-a „pus mâna” pe obiectul stub care apelează metodele remote ale *bean*-ului, se fac apelurile efective de metode: se preia data și ora curentă, respectiv se face calculul complicat care adună numerele 1 și 3.

Totuși, aplicația client nu este pregătită de utilizare, deoarece nu sunt disponibile implementările claselor client specifice GlassFish. Acestea se regăsesc în fișierul `gf-client.jar` din folder-ul `glassfish/lib` al server-ului GlassFish (în acest caz, locația este: `/home/student/opt/glassfish5/glassfish/lib/gf-client.jar`).

Adăugați acest fișier JAR ca dependentă la proiect, exact cum ați adăugat și pachetul `interfaces.jar`.

1.12. Testare stateless session bean

Execuați aplicația client prin apăsarea butonului verde din dreptul funcției `main()`. IntelliJ va crea automat o configurație de execuție Java.

Sisteme Distribuite Laboratorul 2

The screenshot shows the IntelliJ IDEA interface with the project 'JEE-Client' open. The code editor displays the 'JEEClient.java' file, which contains Java code for a Stateless Session Bean client. A red arrow points from the code editor to the 'Run' tab at the bottom, which shows the execution log. The log output includes the command run, the Glassfish server startup message, and the result of the 'addNumbers' method call.

```
10004 [2020-01-02T18:41:44.755+0200] [glassfish 5.0] [INFO] [] [] [tid: _ThreadID=107 _ThreadName=Thread-8] [timeMillis: 1577983304755] [levelValue: 800] [[
10005 [Glassfish] S-a instantiat un stateless session bean: ejb.StatelessSessionBeanImpl]]
10006 [2020-01-02T18:41:44.755+0200] [glassfish 5.0] [INFO] [] [] [tid: _ThreadID=107 _ThreadName=Thread-8] [timeMillis: 1577983304755] [levelValue: 800] [[
10007 [Glassfish] S-a apelat metoda getCurrentTime()])
10008 [2020-01-02T18:41:44.763+0200] [glassfish 5.0] [INFO] [] [] [tid: _ThreadID=107 _ThreadName=Thread-8] [timeMillis: 1577983304763] [levelValue: 800] [[
10009 [Glassfish] S-a apelat metoda addNumbers(1, 3)]]
10010
10011
10012
10013
10014
```

După apel, dacă consultați log-ul server-ului Glassfish, puteți observa mesajele care au fost generate din conținutul *bean*-ului, după apelarea metodelor încapsulate de acesta:

The screenshot shows the Glassfish server log terminal. It displays several log entries corresponding to the execution of the Stateless Session Bean methods. The log entries show the creation of the bean, the execution of the getCurrentTime() method, and the execution of the addNumbers(1, 3) method.

Pentru a observa în timp real apelurile efectuate pe server, deschideți un terminal și executați următoarea comandă:

```
tail -f <LOCATIE_GLAFFISH>/glassfish/domains/domain1/logs/server.log
```

Comanda va afișa în timp real ultimele linii din fișierul log, pe măsură ce acesta este populat. Puteți deschide unul sau mai mulți clienți deodată și observați cum *bean*-ul creat este (re)utilizat pentru clienți diferiți.

Atenție: chiar dacă *bean*-ul creat este de tip *Stateless Session*, nu înseamnă că server-ul va utiliza o singură instanță a acestuia mereu! Presupunând că în *container*-ul de *business* există, pentru moment, o singură instanță a *bean*-ului, iar un client încă „ocupă” acest *bean* (deoarece o metodă apelată încă se execută), atunci server-ul va crea o nouă instanță a *bean*-ului respectiv și o va oferi spre utilizare unui alt client. De aici ideea de „*Stateless Session*”: nu contează care din *bean*-uri este folosit de care client, deoarece nu există ideea de stare menținută cu un anumit client.

Stateful Session Beans

Spre deosebire de *stateless session beans*, aceste tipuri de *beans* mențin o legătură între clientul apelant și *bean*-ul care a fost apelat. Așadar, dacă un client accesează un *stateful session bean*, server-ul este obligat să instantțeze pentru fiecare un *bean* separat, deoarece fiecare *bean* are starea proprie, și nu este permis ca un client să acceseze informații din starea unui *bean* ce

apartine de alt client.

De exemplu, presupunând că pe server există un *bean* ce expune o metodă care afișează lista de e-mail-uri a unui utilizator, cum ar fi ca un alt client să apeleze această metodă din *bean* și să primească lista dvs. de mail-uri, în loc de lista proprie?

Pentru a ilustra funcționarea unui *stateful session bean*, veți crea o aplicație de gestiune a unui cont bancar „deschis” în momentul în care un client accesează prima dată una din operațiunile disponibile:

- depunere numerar
- retragere numerar
- interrogare sold

1.13. Codul de business

Mai întâi, creați interfața de *business* a *bean*-ului, care expune unui client operațiunile ce pot fi efectuate. Adăugați un fișier de tip interfață Java în **ejbs/src/main/java/interfaces** și denumiți-l **BankAccountBeanRemote**. Adăugați următorul conținut:

```
package interfaces;

public interface BankAccountBeanRemote {
    Boolean withdraw(Integer amount);
    void deposit(Integer amount);
    Integer getBalance();
}
```

Apoi, creați implementarea corespunzătoare *bean*-ului în pachetul **ejb** din același modul (**ejbs/src/main/java/**). Denumiți clasa Java **BankAccountBeanImpl**, spre exemplu.

```
package ejb;

import interfaces.BankAccountBeanRemote;
import java.io.Serializable;

public class BankAccountBeanImpl implements BankAccountBeanRemote,
Serializable {
    private Integer availableAmount = 0;

    public Boolean withdraw(Integer amount) {
        if(availableAmount >= amount) {
            availableAmount -= amount;
            return true;
        } else {
            return false;
        }
    }

    public void deposit(Integer amount) {
        availableAmount += amount;
    }

    public Integer getBalance() {
        return availableAmount;
    }
}
```

```
}
```

Înregistrați clasa creată ca și *stateful session bean* în descriptorii XML din fișierul **ejb-jar.xml**, din **ejbs/src/main/resources/META-INF**. Adăugați acest descriptor ca și subordonat al tag-ului **<enterprise-beans>**:

```
<session>
    <ejb-name>BankAccountBean</ejb-name>
    <mapped-name>bankaccount</mapped-name>
    <business-local>interfaces.BankAccountBeanRemote</business-local>
    <business-remote>interfaces.BankAccountBeanRemote</business-
remote>
    <ejb-class>ejb.BankAccountBeanImpl</ejb-class>
    <session-type>Stateful</session-type>
</session>
```

De această dată, veți accesa *bean*-ul din *web tier*, spre deosebire de capitolul anterior, unde ați creat o aplicație client Java separată.

1.14. Codul din web tier

În continuare, creați meniul de acces pentru client, ca și pagină JSP. Puteți folosi pagina **index.jsp** deja existentă ca și meniu.

Modificați **servlets/servlet/src/main/webapp/index.jsp** astfel:

```
<%@ page contentType="text/html; charset=UTF-8" %>
<html>
    <head>
        <title>Meniu principal</title>
        <meta charset="utf-8" />
    </head>
    <body>
        <h1>Meniu principal</h1>
        <h3>Gestiune cont bancar</h3>
        <form action=".//process-bank-operation" method="post">
            <fieldset label="operatiuni">
                <legend>Alegeti operatiunea dorita:</legend>
                <select name="operation">
                    <option value="deposit">Depunere numerar</option>
                    <option value="withdraw">Retragere
numerar</option>
                    <option value="balance">Interrogare sold</option>
                </select>
                <br />
                <br />
                Introduceti suma: <input type="number" name="amount" />
                <br />
                <br />
                <button type="submit">Efectuare</button>
            </fieldset>
        </form>
    </body>
</html>
```

Pagina conține un formular simplu în care utilizatorul poate selecta operațiunea dorită și

Sisteme Distribuite Laboratorul 2

poate introduce o sumă de bani într-un câmp numeric, utilizat la operațiunile de retragere / depunere.

Ținta formularului este un servlet mapat pe ruta de acces `./process-bank-operation`, către care datele din formular sunt trimise prin metoda **POST**. Așadar, acum veți crea servlet-ul respectiv: o clasă Java denumită **ProcessBankOperationServlet**, plasată în modulul **servlet**, în următorul folder: **servlets/servlet/src/main/java**. Adăugați codul următor:

```
import interfaces.BankAccountBeanRemote;

import javax.naming.InitialContext;
import javax.naming.NamingException;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;

public class ProcessBankOperationServlet extends HttpServlet {
    @Override
    protected void doPost(HttpServletRequest request,
    HttpServletResponse response) throws ServletException, IOException {
        // preluare parametri din cererea HTTP
        String operation = request.getParameter("operation");
        String amountString = request.getParameter("amount");

        // nu conteaza suma introdusa in campul numeric daca
        operatiunea este de tip "Sold cont"
        Integer amount = (!amountString.equals("")) ?
        Integer.parseInt(amountString) : 0;

        // se incercă preluarea bean-ului folosind obiectul
        HttpSession, care pastreaza o sesiune HTTP intre client si server
        BankAccountBeanRemote bankAccount;
        bankAccount =
        (BankAccountBeanRemote)request.getSession().getAttribute("bankAccount-
        Bean");

        // daca nu exista nimic pastrat in sesiunea HTTP, inseamna ca
        bean-ul se preia prin JNDI lookup
        if (bankAccount == null) {
            try {
                InitialContext ctx = new InitialContext();
                bankAccount = (BankAccountBeanRemote)
                ctx.lookup("bankaccount#interfaces.BankAccountBeanRemote");

                // dupa preluarea bean-ului prin JNDI, obiectul se
                stocheaza in sesiune pentru a fi refolosit ulterior
                // cererile urmatoare vor utiliza obiectul remote
                stocat in sesiune
                request.getSession().setAttribute("bankAccountBean",
                bankAccount);
            } catch (NamingException e) {
                e.printStackTrace();
                return;
            }
        }
    }
}
```

Sisteme Distribuite Laboratorul 2

```
        }

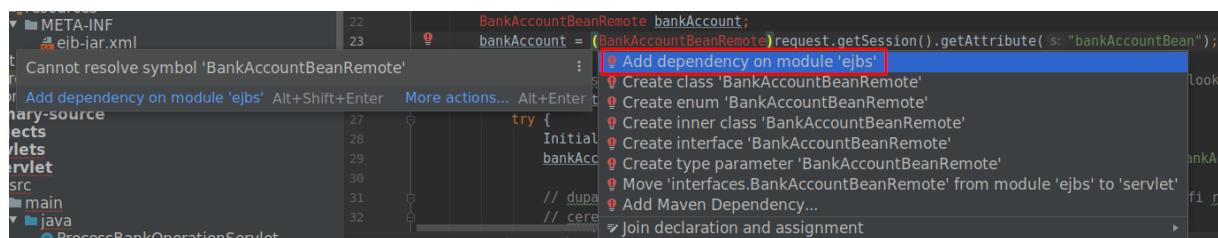
        Integer accountBalance = null;
        String message = "";

        // in functie de operatia selectata de client, se apeleaza
metoda corespunzatoare din obiectul remote
        if (operation.equals("deposit")) {
            bankAccount.deposit(amount);
            message = "In contul dvs. au fost depusa suma: " + amount
+ ".";
        } else if (operation.equals("withdraw")) {
            if (bankAccount.withdraw(amount)) {
                message = "Din contul dvs. s-a retras suma de: " +
amount + ".";
            } else {
                message = "Operatiunea a esuat! Fonduri
insuficiente.";
            }
        } else if (operation.equals("balance")) {
            accountBalance = bankAccount.getBalance();
        }

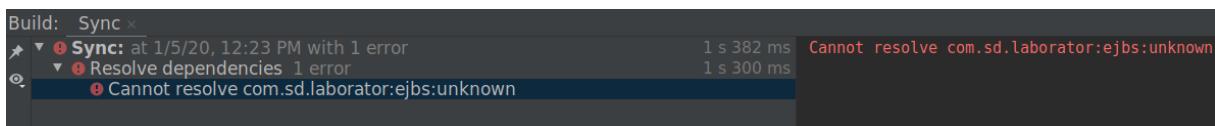
        message += "<br /><br />";
        if (accountBalance != null) {
            message += "Sold cont: " + accountBalance;
        }
        message += "<br /><a href='.'/>Inapoi la meniul
principal</a>";

        // dupa construirea mesajului raspuns, acesta este trimis ca
si continut HTML inapoi la clientul apelant
        response.setContentType("text/html");
        response.getWriter().print(message);
    }
}
```

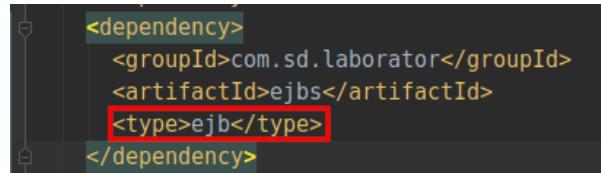
După ce scrieți codul respectiv în fișierul sursă, veți observa o problemă: nu aveți acces din acest modul la pachetul cu *bean-uri enterprise* (**ejbs**), și deci nu se poate importa interfața **BankAccountBeanRemote**. Pentru a rezolva acest lucru, modulul Maven **ejbs** trebuie adăugat ca dependență la modulul Maven **servlet**. IntelliJ ajută utilizatorul în acest sens: dați click pe **BankAccountBeanRemote** marcat cu roșu, apăsați **ALT+ENTER** și alegeți „**Add dependency on module ejbs**”.



Nu este suficient, deoarece IntelliJ nu poate detecta singur ce tip de modul este cel importat.



Așa încât, modificați manual fișierul **servlets/servlet/pom.xml**, iar pentru dependența adăugată de IntelliJ, specificați și atributul **<type>ejb</type>** (deoarece se importă un modul de tip *Enterprise Java Bean*, conform schemei DTD al fișierului POM).



Nu uitați de „**Import changes**” pentru a sincroniza proiectul Maven și modulele sale.

Corpul servlet-ului conține cod de tratare a cererilor de tip POST, deoarece formularul care îl trimite datele prin această metodă HTTP (se putea folosi și GET, spre exemplu).

Bean-ul de tip *stateful* se preia prin *JNDI lookup* prima dată când clientul accesează servlet-ul. În acel moment, server-ul îi creează și asociază o instanță a bean-ului în contextul de execuție, iar această instanță este reținută de servlet în sesiunea HTTP, deoarece este nevoie de ea la fiecare cerere. **Dacă nu ați fi reținut instanța primită, cererile ulterioare vor prelua instanțe noi de bean de la server (chiar dacă este stateful! Starea este păstrată pe aceeași instanță. În momentul în care se face lookup din nou, e ca și cum ați cere o instanță nouă, deci ați pierdut starea).**

Cererile ulterioare primei vor prelua bean-ul din sesiune, iar apelurile metodelor acestuia vor ține cont de starea anterioară, deoarece clientul este același, iar server-ul *enterprise* va folosi aceeași instanță din memorie pentru a deservi clientul respectiv. Dacă un alt client cere o instanță de bean, server-ul va crea o instanță nouă și i-o va asocia lui, păstrându-i starea în cererile ulterioare, și.a.m.d, pentru fiecare client **diferit**.

Nu uitați să creați descriptorul XML pentru servlet, pentru ca server-ul *enterprise* să îl mapeze pe ruta dorită și să îl gestioneze corespunzător.

Deschideți **servlets/servlet/src/main/webapp/WEB-INF/web.xml** și adăugați o nouă mapare pentru servlet-ul **ProcessBankOperationServlet**:

```
<servlet>
    <servlet-name>ProcessBankOperation</servlet-name>
    <servlet-class>ProcessBankOperationServlet</servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name>ProcessBankOperation</servlet-name>
    <url-pattern>/process-bank-operation</url-pattern>
</servlet-mapping>
```

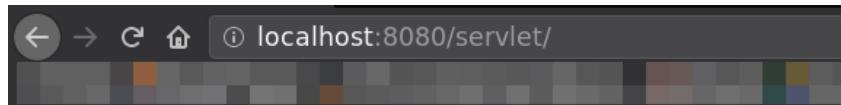
Urmează pașii cu care deja v-ați obișnuit: **compile → package → redeploy de la consolă**.

1.15. Testare stateful session bean

Testarea bean-ului de tip *stateful* presupune existența a 2 clienți diferenți (sesiuni diferențiate HTTP). Aveți 2 variante, întrucât server-ul este local: folosiți 2 browsere diferențate, sau folosiți 1 singur browser, o dată din modul obișnuit de navigare, și a doua oară din modul **incognito**.

Deschideți un browser și navigați la adresa:

<http://localhost:8080/servlet/>



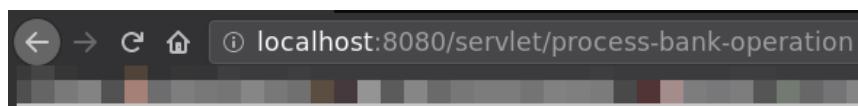
Meniu principal

Gestiune cont bancar

Alegeți operațiunea dorita:

Introduceti suma:

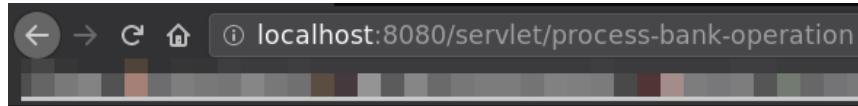
Selectați **Depunere numerar**, introduceți o sumă și apoi apăsați pe „**Efectuare**”. În acest moment, clientului curent (browser-ul web, în acest caz) a primit o instanță de *stateful session bean*, iar aceasta a fost stocată în sesiune.



In contul dvs. au fost depusa suma: 120.

[Inapoi la meniul principal](#)

Înapoi la meniul principal și verificați soldul. Valoarea este preluată din *bean*-ul existent în memoria server-ului *enterprise*.



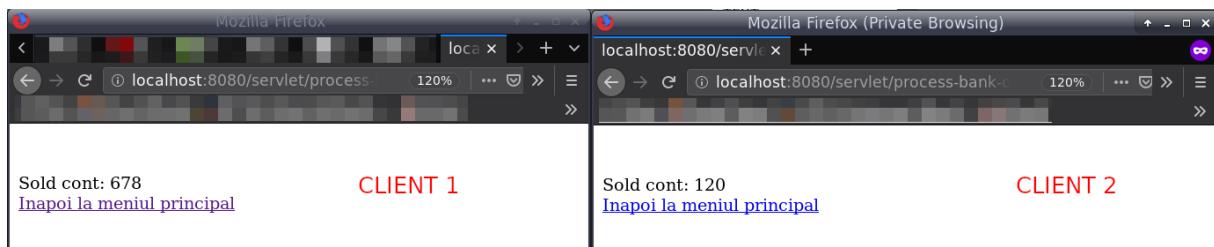
Sold cont: 120

[Inapoi la meniul principal](#)

Puteți testa alte operațiuni de retragere / depunere pentru a confirma că starea *bean*-ului este păstrată de la un apel la altul.

Acum deschideți un al 2-lea browser (sau modul incognito dacă folosiți același browser: **CTRL+SHIFT+N** pentru Google Chrome sau **CTRL+SHIFT+P** pentru Mozilla Firefox).

Faceți o depunere și observați că soldul diferă de cel al primului client, deoarece acesta este un client nou, deci server-ul va utiliza o altă instanță de *bean* atunci când se face căutarea (*lookup*) în JNDI.



JPA Entities

Entitățile JPA (*JPA Entities*) au fost precedate, ca tehnologie, de *Entity Beans*. Începând de la standardul EJB 3.0, *Entity Bean*-urile sunt marcate ca învechite și, de aceea, veți folosi tehnologia nouă în acest capitol al laboratorului.

1.16. Configurare persistență

Pentru a folosi *Java Persistence API* în proiectul JEE, trebuie să adăugați un descriptor de persistență într-un fișier denumit **persistence.xml**, ce rezidă în folder-ul **ejb3/src/main/resources/META-INF**. Creați fișierul și adăugați următorul conținut:

```
<persistence xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
  http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd"
  version="2.0"
  xmlns="http://java.sun.com/xml/ns/persistence">
  <persistence-unit name="bazaDeDateSQLite" transaction-
  type="RESOURCE_LOCAL">

  <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
    <class>ejb.StudentEntity</class>
    <properties>
      <property name="javax.persistence.jdbc.driver"
value="org.sqlite.JDBC" />
      <property name="javax.persistence.jdbc.url"
value="jdbc:sqlite:/home/student/studenti.db" />
      <property name="eclipselink.logging.level" value="ALL" />
      <property name="eclipselink.ddl-generation" value="create-
tables" />
    </properties>
  </persistence-unit>
</persistence>
```

În acest XML s-a definit o așa-numită unitate de persistență (*persistence unit*), adică o grupare logică a tuturor entităților gestionate de instanțele unui **EntityManager** (detalii despre această clasă în cele ce urmează) în aplicația JEE. Unitatea de persistență poate folosi o bază de date SQL / NoSQL, o bază de date *embedded* etc. În acest caz se utilizează o bază de date SQLite, în care tranzacțiile se fac doar cu „resurse locale”, deoarece această bază de date este complet încapsulată într-un fișier cu extensia **db**. S-a marcat acest lucru prin atributul **transaction-type="RESOURCE_LOCAL"**.

JPA trebuie configurat în așa fel încât să știe ce furnizor și *driver* de persistență să folosească atunci când utilizatorul lucrează cu acest API și cu datele ce se transmit. În acest caz,

furnizorul (*provider*-ul) de persistență este interfața **PersistenceProvider** din implementarea JPA **EclipseLink** - disponibilă implicit odată cu server-ul GlassFish.

Pentru unitatea de persistență se specifică, la început (după tag-ul **<provider>**), care sunt clasele ce vor fi gestionate de **EntityManager** - clasele de domeniu, adică cele care sunt mapate în baza de date sub formă de tabele.

Apoi, se dă proprietățile furnizorului de persistență: driver-ul folosit, calea pe disc către fișierul unde este încapsulată baza de date SQLite (în acest caz, **/home/student/studenti.db**), nivelul de *logging* și, la final, este specificat faptul că EclipseLink va crea schema bazei de date folosind o interogare de tip „**CREATE TABLE...**” în momentul în care aplicația este încărcată pe server.

Atenție: fișierul pe care îl specificați ca și bază de date SQLite trebuie să se afle într-o locație în care utilizatorul care a pornit server-ul GlassFish are drepturi de scriere!

Pentru ca aplicația JEE să poată utiliza *driver*-ul JDBC pentru SQLite (clasa **org.sqlite.JDBC**), acest driver trebuie adăugat ca dependentă în modulul Maven în care este folosit, adică în modulul **ejbs**. Deschideți fișierul **ejbs/pom.xml** și adăugați următoarea dependență:

```
<dependency>
    <groupId>org.xerial</groupId>
    <artifactId>sqlite-jdbc</artifactId>
    <version>3.30.1</version>
</dependency>
```

1.17. Adăugare entitate JPA

Creați o entitate JPA denumită **StudentEntity**, ca și clasă Java situată în **/ejbs/src/main/java/ejb**. Codul clasei este următorul:

```
package ejb;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;

@Entity
public class StudentEntity {
    @Id
    @GeneratedValue
    private int id;
    private String nume;
    private String prenume;
    private int varsta;

    public StudentEntity() {
    }

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }
}
```

```

public String getNume() {
    return nume;
}

public void setNume(String nume) {
    this.nume = nume;
}

public String getPrenume() {
    return prenume;
}

public void setPrenume(String prenume) {
    this.prenume = prenume;
}

public int getVarsta() {
    return varsta;
}

public void setVarsta(int varsta) {
    this.varsta = varsta;
}
}

```

Observați că entitatea JPA seamănă cu un **POJO** (*Plain Old Java Object*) ce conține adnotări corespunzătoare, unde este cazul: adnotarea **@Entity** marchează faptul că această clasă este o entitate JPA, adnotarea **@Id** indică după care câmp vor fi identificate înregistrările din tabela rezultată în baza de date (cheia primară), adnotarea **@GeneratedValue** marchează faptul că acel câmp adnotat (în acest caz, câmpul **id**) conține o valoare generată automat la inserarea unei înregistrări - nu trebuie dat explicit de utilizator.

Câmpurile încapsulate trebuie să fie tipuri de date simple (primitive sau nu), și să fie serializabile.

1.18. Codul din web tier

Pentru a ilustra lucrul cu această entitate JPA simplă, veți folosi același formular de introducere a datelor unui student, utilizat în laboratorul 1. Adăugați codul acestuia în fișierul **servlets/servlet/src/main/webapp/formular.jsp**.

```

<html xmlns:jsp="http://java.sun.com/JSP/Page">
    <head>
        <title>Formular student</title>
        <meta charset="UTF-8" />
    </head>
    <body>
        <h3>Formular student</h3>
        Introduceti datele despre student:
        <form action=".//process-student" method="post">
            Nume: <input type="text" name="nume" />
            <br />
            Prenume: <input type="text" name="prenume" />
            <br />
            Varsta: <input type="number" name="varsta" />
            <br />
        </form>
    </body>
</html>

```

Sisteme Distribuite Laboratorul 2

```
<br />
<button type="submit" name="submit">Trimite</button>
</form>
</body>
</html>
```

Formularul va prelua aceste date și le va trimite unui servlet spre procesare. Servlet-ul preia datele din cererea HTTP și le persistă într-o bază de date *embedded* de tip **SQLite** (încapsulată într-un fișier pe disc). Creați acest servlet în **servlets/servlet/src/main/java/ProcessStudentServlet.java**, cu următorul conținut:

```
import ejb.StudentEntity;

import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.EntityTransaction;
import javax.persistence.Persistence;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;

public class ProcessStudentServlet extends HttpServlet {
    @Override
    protected void doPost(HttpServletRequest request,
    HttpServletResponse response) throws ServletException, IOException {
        // se citesc parametrii din cererea de tip POST
        String nume = request.getParameter("nume");
        String prenume = request.getParameter("prenume");
        int varsta = Integer.parseInt(request.getParameter("varsta"));

        // pregatire EntityManager
        EntityManagerFactory factory =
        Persistence.createEntityManagerFactory("bazaDeDateSQLite");
        EntityManager em = factory.createEntityManager();

        // creare entitate JPA si populare cu datele primite din
        formular
        StudentEntity student = new StudentEntity();
        student.setNume(nume);
        student.setPrenume(prenume);
        student.setVarsta(varsta);

        // adaugare entitate in baza de date (operatiune de
        persistenta)
        // se face intr-o tranzactie
        EntityTransaction transaction = em.getTransaction();
        transaction.begin();
        em.persist(student);
        transaction.commit();

        // inchidere EntityManager
        em.close();
        factory.close();
    }
}
```

```

    // trimitere raspuns inapoi la client
    response.setContentType("text/html");
    response.getWriter().println("Datele au fost adaugate in baza
de date." +
        "<br /><br /><a href='.'/>Inapoi la meniul
principal</a>");
    }
}

```

Clasa **EntityManager** este folosită, după cum este și denumită, la gestiunea tuturor entităților persistente (*JPA Entities*) de care este nevoie în aplicația JEE. O instanță **EntityManager** se obține de la **EntityManagerFactory**, pe baza numelui unității de persistență.

Creați încă un servlet folosit pentru afișarea listei de studenți disponibilă în baza de date. Denumiți servlet-ul **FetchStudentListServlet** și adăugați următorul conținut în fișierul sursă:

```

import ejb.StudentEntity;

import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;
import javax.persistence.TypedQuery;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;
import java.util.List;

public class FetchStudentListServlet extends HttpServlet {
    @Override
    protected void doGet(HttpServletRequest request,
    HttpServletResponse response) throws ServletException, IOException {
        // pregatire EntityManager
        EntityManagerFactory factory =
        Persistence.createEntityManagerFactory("bazaDeDateSQLite");
        EntityManager em = factory.createEntityManager();

        StringBuilder responseText = new StringBuilder();
        responseText.append("<h2>Lista studenti</h2>");
        responseText.append("<table
border='1'><thead><tr><th>ID</th><th>Nume</th><th>Prenume</th><th>Vars
ta</th></thead>");
        responseText.append("<tbody>");

        // preluare date studenti din baza de date
        TypedQuery<StudentEntity> query = em.createQuery("select
student from StudentEntity student", StudentEntity.class);
        List<StudentEntity> results = query.getResultList();
        for (StudentEntity student : results) {
            // se creeaza cate un rand de tabel HTML pentru fiecare
student gasit
            responseText.append("<tr><td>" + student.getId() +

```

Sisteme Distribuite Laboratorul 2

```
"</td><td>" +  
            student.getNume() + "</td><td>" +  
student.getPrenume() +  
            "</td><td>" + student.getVarsta() + "</td></tr>");  
    }  
  
    responseText.append("</tbody></table><br /><br /><a  
href='./'>Inapoi la meniul principal</a>");  
  
    // inchidere EntityManager  
    em.close();  
    factory.close();  
  
    // trimitere raspuns la client  
    response.setContentType("text/html");  
    response.getWriter().print(responseText.toString());  
}  
}
```

Observați că preluarea listei de studenți se face printr-o interogare care seamănă cu limbajul SQL. Acest limbaj se numește **JPQL (Java Persistence Query language)**.

Nu uitați de maparea servleților prin descriptori XML. Modificați **web.xml** astfel:

```
...  
  
<servlet>  
    ...  
</servlet>  
<servlet>  
    <servlet-name>ProcessStudent</servlet-name>  
    <servlet-class>ProcessStudentServlet</servlet-class>  
</servlet>  
<servlet>  
    <servlet-name>FetchStudentList</servlet-name>  
    <servlet-class>FetchStudentListServlet</servlet-class>  
</servlet>  
  
<servlet-mapping>  
    ...  
</servlet-mapping>  
<servlet-mapping>  
    <servlet-name>ProcessStudent</servlet-name>  
    <url-pattern>/process-student</url-pattern>  
</servlet-mapping>  
<servlet-mapping>  
    <servlet-name>FetchStudentList</servlet-name>  
    <url-pattern>/fetch-student-list</url-pattern>  
</servlet-mapping>  
  
...
```

Modificați meniul principal din pagina **index.jsp** ca să includeți ancore către formularul de adăugare a unui student, respectiv către servlet-ul care afișează lista studenților:

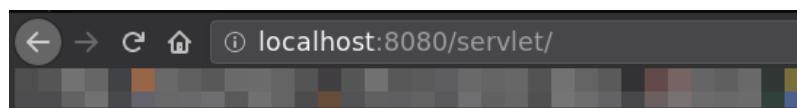
```
...  
...
```

```
</form>
<hr />
<h3>Baza de date cu studenti</h3>
<a href=". /formular.jsp">Adaugare student</a>
<br />
<a href=". /fetch-student-list">Afisare lista studenti</a>
...
...
```

1.19. Testare entitate JPA

Curătați aplicația (**clean**), apoi compilați (**compile**), împachetați aplicația (**package**) în artefact EAR și încărcați-o pe server (**redeploy** dacă ați mai încărcat-o anterior, sau **deploy** dacă e prima dată când încărcați - nu uitați, **de la consolă**).

Navigați la <http://localhost:8080/servlet>.



Meniu principal

Gestiune cont bancar

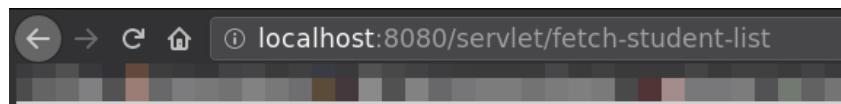
Alegeti operatiunea dorita:

Introduceti suma:

Baza de date cu studenti

[Adaugare student](#)
[Afisare lista studenti](#)

Adăugați studenți în baza de date din meniul corespunzător, apoi testați dacă lista de studenți poate fi preluată din baza de date (din fișierul **studenti.db**).



Lista studenti

ID	Nume	Prenume	Varsta
1	Popescu	Ion	23
51	Ala	Bala	13
101	Porto	Cala	40

[Inapoi la meniul principal](#)

Aplicații și teme

- **Temă de laborator**

Adăugați operațiile de actualizare și de ștergere a unui student din baza de date SQLite:

- modificați meniul principal din pagina **index.jsp** pentru a indica aceste operații
- adăugați încă 2 servleți corespunzători acestor operații. În acești servleți, folosiți clasa **EntityManager** și limbajul **JPQL** (*Java Persistence Query Language*) pentru a implementa operațiile cerute. Sugestie:
 - actualizare: căutați studentul în baza de date după nume și / sau prenume, apoi preluăți identificatorul acestuia (ID-ul). Având acest ID, faceți o operație de UPDATE folosind JPQL cu noile date preluate de la utilizator (eventual dintr-un formular JSP).
 - ștergere: căutați studentul în baza de date după nume și / sau prenume, apoi preluăți identificatorul acestuia (ID-ul). Având acest ID, faceți o operație de DELETE folosind JPQL, după ID-ul preluat.
- **Alternativă**: atunci când se preiau studenții pentru afișare, puteți prelua și ID-ul din baza de date și să îl setați ca parametru URL pentru servlet-ul apelat la click pe un buton de ștergere / actualizare pus în dreptul fiecărui student, asemănător cu:

```
<a href=".//update-student?id=ID&nume=NUME&prenume=...>Actualizeaza</a>
<a href=".//delete-student?id=ID">Sterge</a>
```

- **Temă pe acasă**

Adăugați încă o tabelă în baza de date **studenti.db**, numită **Cursuri** (deci o nouă clasă de tip *JPA Entity*). Pentru această nouă entitate, implementați toate operațiile de adăugare, preluare, actualizare, ștergere, punându-le în evidență printr-un formular JSP asemănător cu cel din laborator.

Bibliografie

Enterprise Java Beans - https://docs.oracle.com/cd/E24329_01/web.1211/e24446/ejbs.htm

Accesarea bean-urilor - <https://docs.oracle.com/javaee/7/tutorial/ejb-intro004.htm>

Introducere în Java Persistence API - <https://docs.oracle.com/javaee/7/tutorial/persistence-intro.htm>

JPA Entities - https://docs.oracle.com/cd/E16439_01/doc.1013/e13981/undejbs003.htm

Persistence Units - <https://docs.oracle.com/cd/E19798-01/821-1841/bnbrj/index.html>

Sisteme Distribuite Laboratorul 2

JPQL Language reference - https://docs.oracle.com/html/E13946_04/ejb3_langref.html

Entity Beans - https://docs.oracle.com/cd/B14099_19/web.1012/b15505/entity.htm

Message-Driven Beans - <https://docs.oracle.com/javaee/6/tutorial/doc/gipko.html>

Sisteme Distribuite - Laborator 3

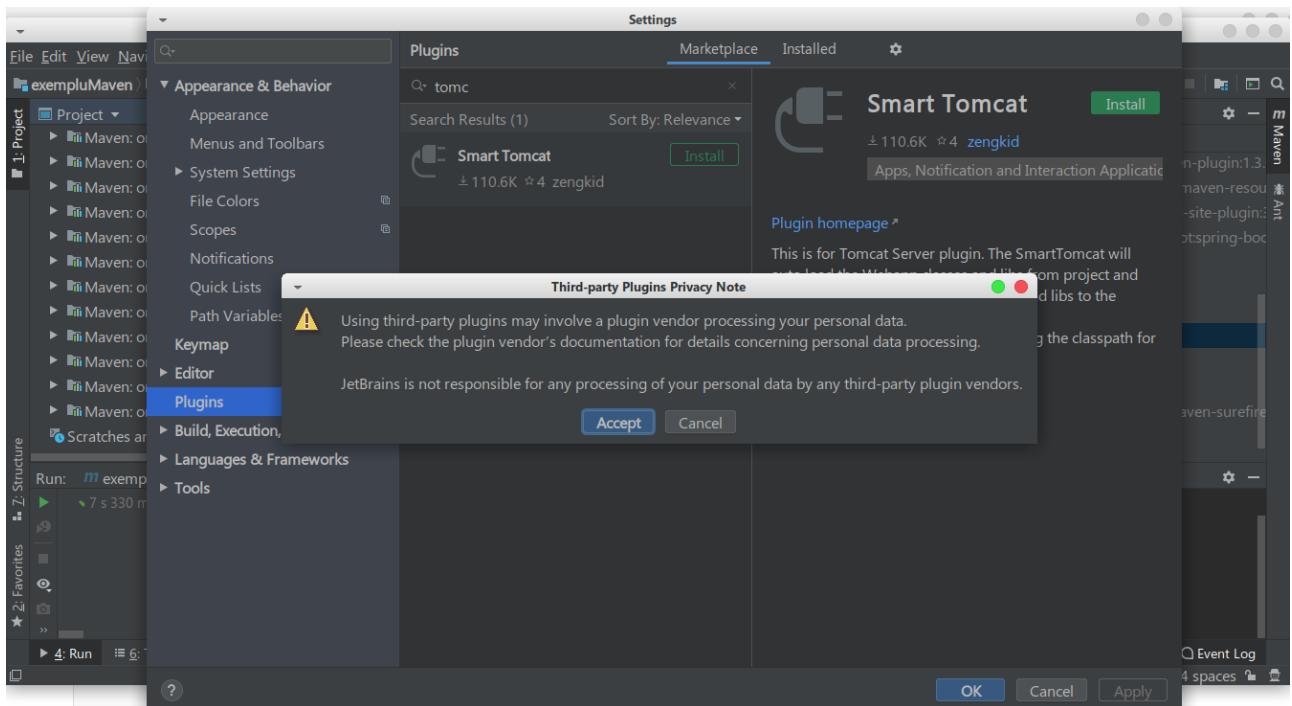
Prima aplicație utilizând Spring Boot

Obiectivele lucrării de laborator sunt următoarele:

- crearea unui proiect **Spring Boot**;
 - folosind **Maven**;
 - folosind **Gradle**;
- primii pași în utilizarea expunerii serviciilor folosind *plugin-ul Spring Boot*;
- crearea unei aplicații **Spring** simple.

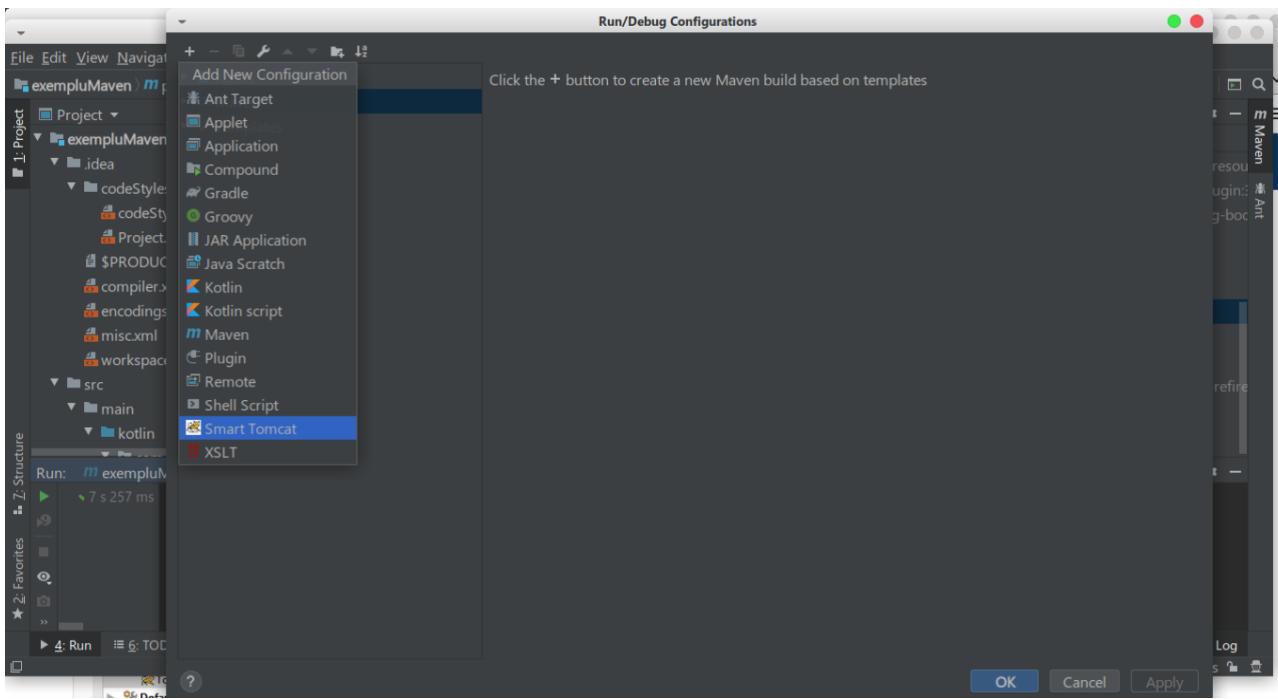
ALT+ENTER este prietenul vostru!

Pentru a putea executa aplicațiile specifice, trebuie instalat un server de **Tomcat** în mediul virtual al IntelliJ. **File → Settings → Plugins** și se caută și instalează *plugin-ul „Smart Tomcat”* (vezi figura următoare).



În continuare, se adaugă o configurație de execuție pentru a porni Smart Tomcat.

Sisteme Distribuite - Laborator 3



Acesta este *plugin*-ul pentru serverul de **Tomcat** care va autoîncărca clasele aplicației Web precum și bibliotecile din proiect și modulele. De asemenea, va autoconfigura și căile către clase necesare serverului Tomcat. Suportă Tomcat începând cu versiunea 6.

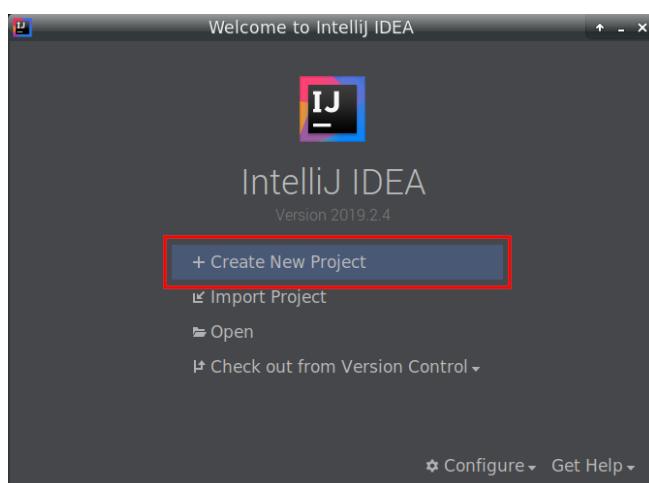
1. Crearea unui proiect Spring Boot

Spring Boot ușurează crearea de aplicații bazate pe *framework*-ul Spring, prin punerea la dispoziția utilizatorului a unui proiect preconfigurat.

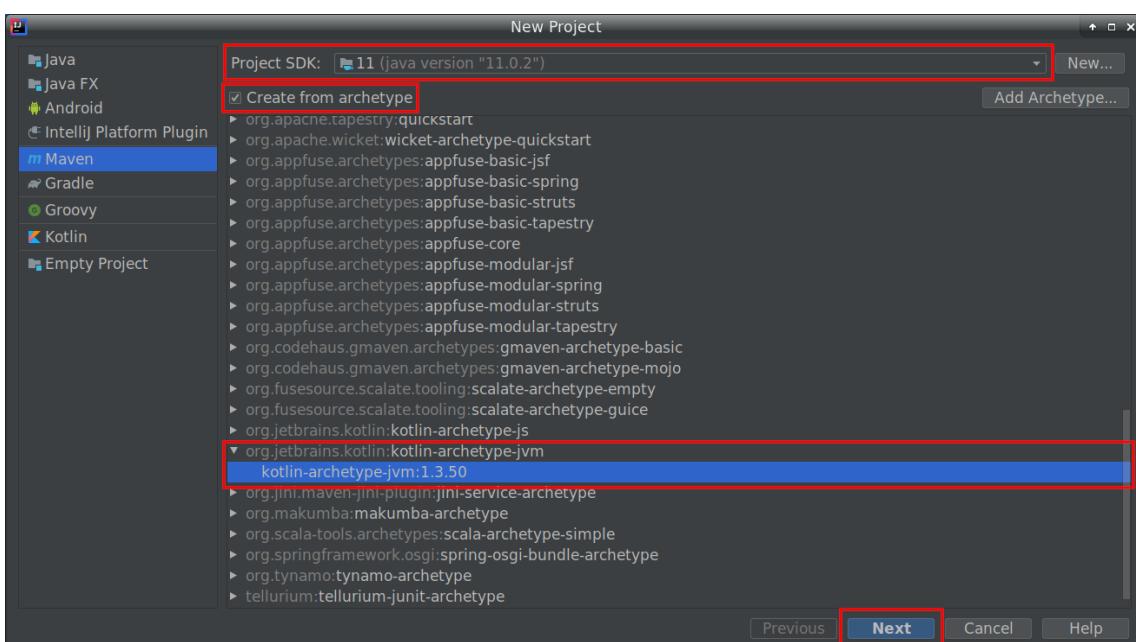
Deoarece mediul de dezvoltare IntelliJ IDEA versiunea „Community Edition” (disponibilă în laborator) nu oferă opțiunea de a crea direct un proiect de tip „Spring Boot Application”, va trebui să se urmeze pașii prezențați în cele ce urmează, în funcție de utilitarul folosit pentru management-ul proiectelor (Maven / Gradle).

1.1. Proiect Spring Boot folosind *Maven*

1. Deschideți aplicația **IntelliJ IDEA Community**
2. Selectați „Create New Project” din meniul principal



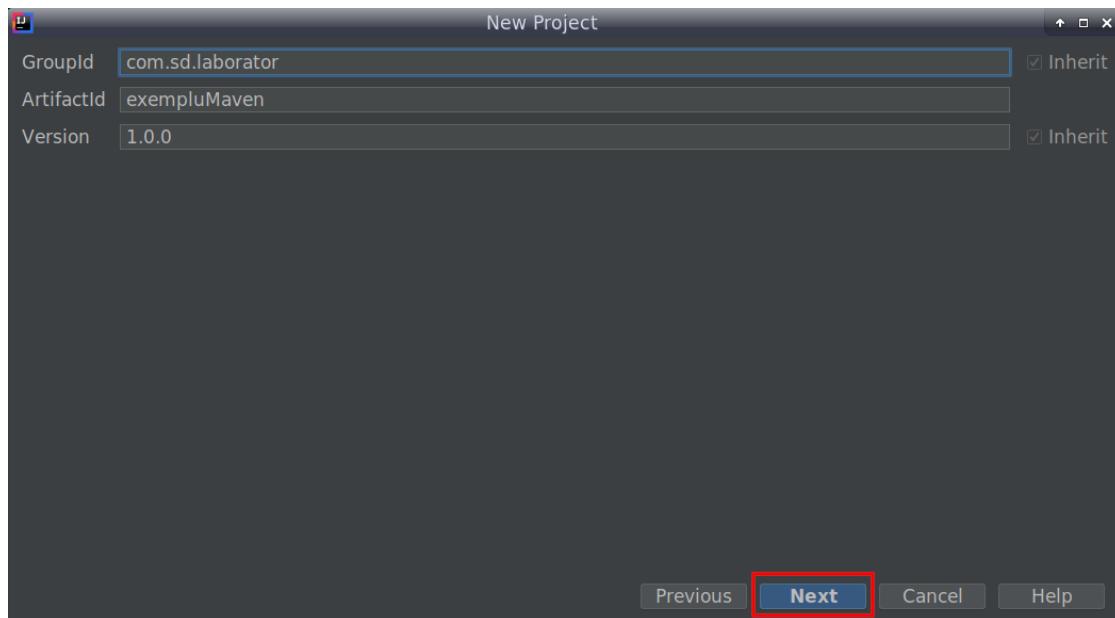
3. Alegeti ca tip de proiect „**Maven**”, selectați **versiunea 11** de **Java SDK**, apoi bifați „Create from archetype”. Din lista de mai jos, selectați „**org.jetbrains.kotlin:kotlin-archetype-jvm**” → „**kotlin-archetype-jvm:<versiune_Kotlin>**”, apoi apăsați pe „**Next**”.



4. Completăți metadatele proiectului, conform cu specificațiile Apache Maven, disponibile la următorul URL:

<https://maven.apache.org/guides/mini/guide-naming-conventions.html>

- a) groupId: **com.sd.laborator**
- b) artifactId: **exempluMaven**
- c) version: **1.0.0**



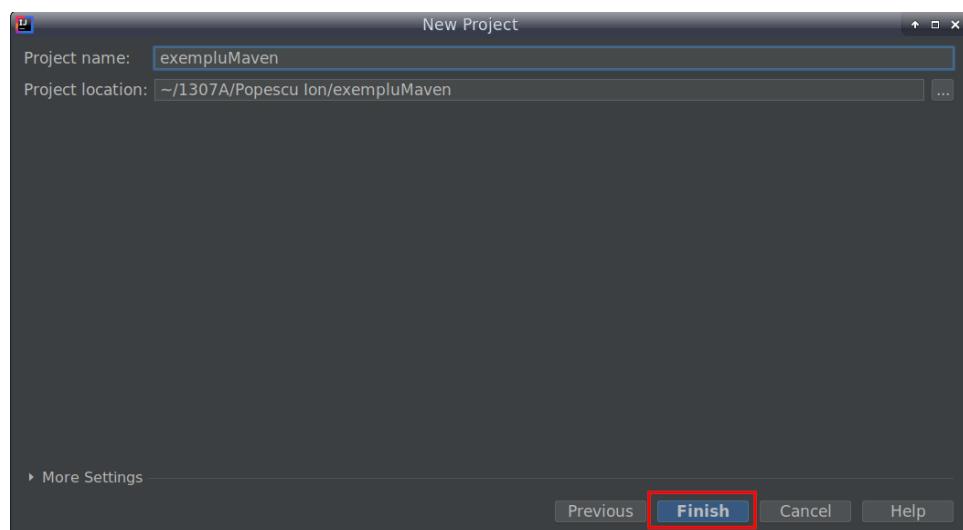
5. Se va apăsa pe „Next”, iar în secțiunea următoare nu modificați nici o setare. Se va apăsa din nou pe „Next”.

6. Completăți numele proiectului și selectați locația sa pe disc, apoi apăsați „Finish”.

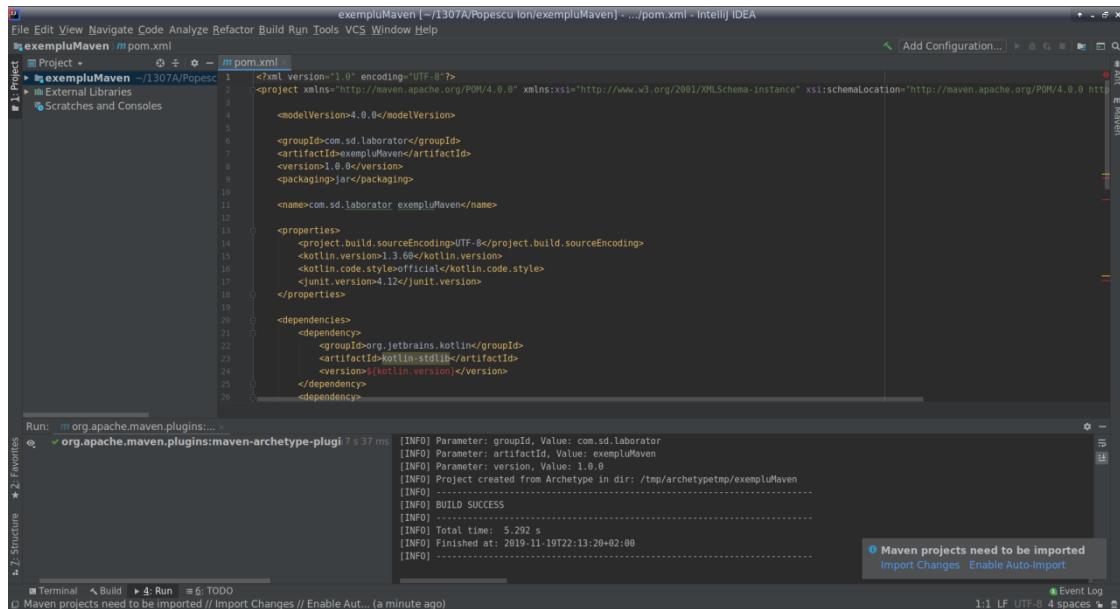
În cazul în care nu există deja, se creează un folder în locația **/home/student** cu numele grupei din care faceți parte. În folder-ul grupei, se creează un alt folder cu numele dvs., acela fiind folder-ul de bază în care se va lucra la laborator.

Exemplu: **/home/student/1307A/Popescu_Ion**

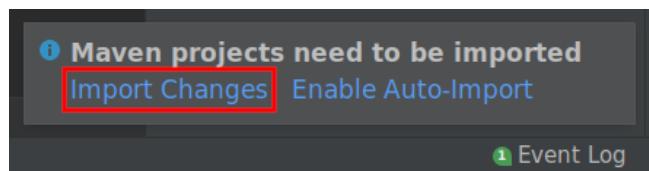
Atenție! Fișierele proiect vor fi plasate **direct** în ultimul folder din calea specificată în „Project location”. IntelliJ nu creează automat niciun subfolder pentru utilizator. Așadar, ar fi indicat ca locația proiectului să indice către un folder gol!



7. Se va deschide o fereastră IntelliJ, cu noul proiect Maven creat, afișând automat conținutul fișierului „**pom.xml**”. Acesta reprezintă un **Project Object Model**, un fișier XML care conține toate informațiile necesare compilării proiectului respectiv. Se poate observa că, pentru moment, acesta conține metadatele introduse la pasul 4, în tag-urile XML corespunzătoare.



8. În momentul în care IntelliJ afișează următorul mesaj, selectați „**Import Changes**”:



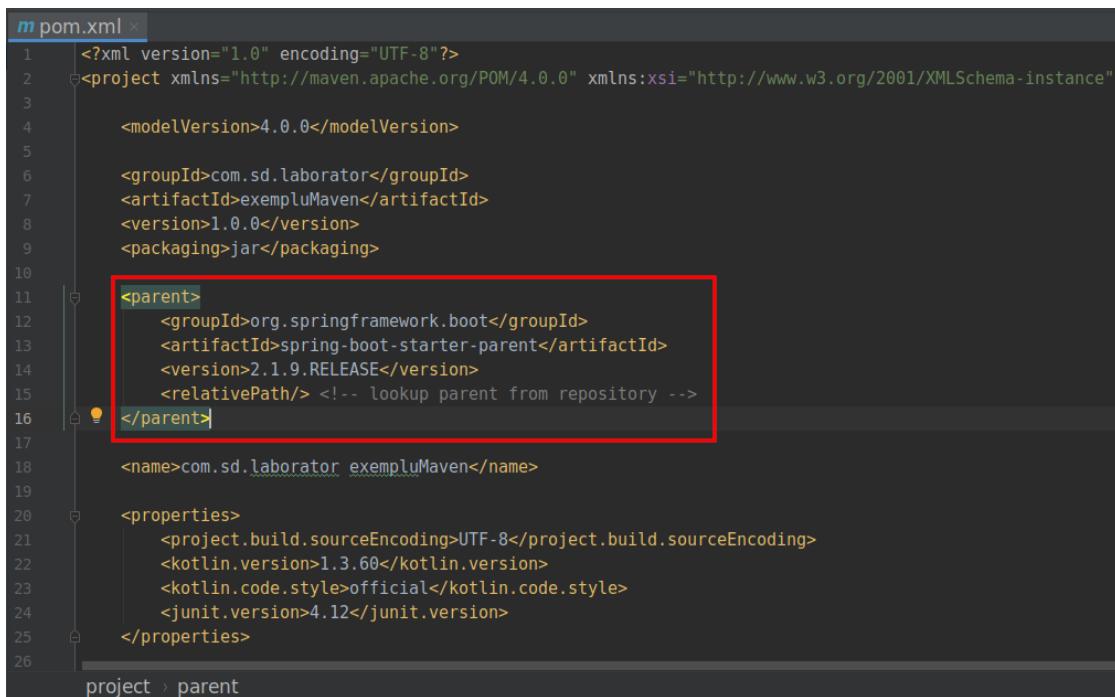
Operația va conduce la sincronizarea dependențelor specificate în fișierul de configurare **pom.xml**, în momentul în care este editat de către utilizator. Ca efect, modificarea făcută în pasul 7 de mai sus va determina ca IntelliJ să descarce automat toate dependențele specificate în interiorul tag-ului **<dependencies>**.

9. În fișierul POM (**pom.xml**), adăugați următoarele elemente, ca și subordonați ai tag-ului **<project>**:

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.1.9.RELEASE</version>
    <relativePath/> <!-- lookup parent from repository -->
</parent>
```

*Tag-urile adăugate mai sus determină ca proiectul să moștenească dependențele și setările unei aplicații **Spring Boot** din *repository*-urile Maven.*

Sisteme Distribuite - Laborator 3



```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.sd.laborator</groupId>
    <artifactId>exempluMaven</artifactId>
    <version>1.0.0</version>
    <packaging>jar</packaging>

    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.1.9.RELEASE</version>
        <relativePath/> <!-- lookup parent from repository -->
    </parent>

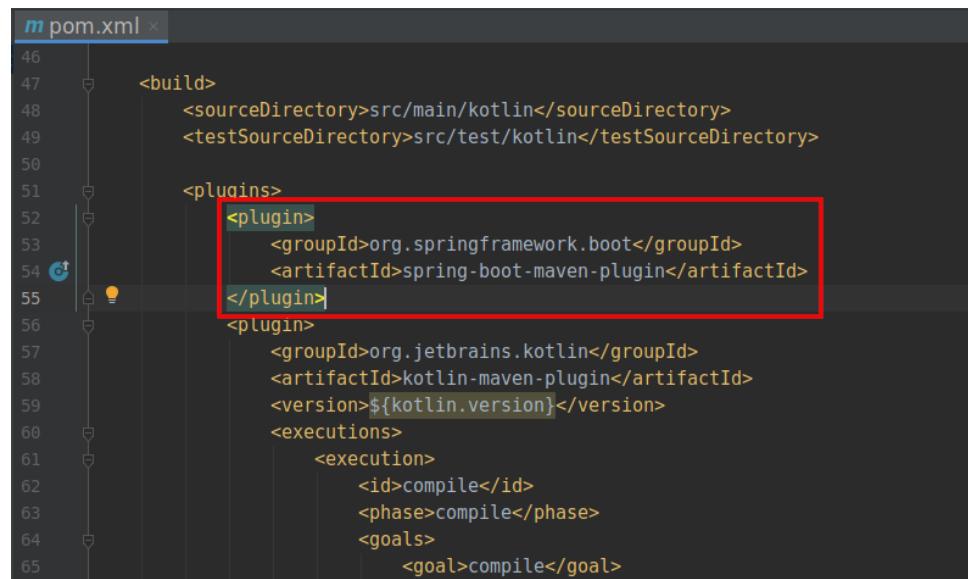
    <name>com.sd.laborator exempluMaven</name>

    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <kotlin.version>1.3.60</kotlin.version>
        <kotlin.code.style>official</kotlin.code.style>
        <junit.version>4.12</junit.version>
    </properties>

```

10. Ca subordonat al *tag-ului <plugins>*, adăugați **Spring Boot Plugin**:

```
<plugin>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-maven-plugin</artifactId>
</plugin>
```



```
<build>
    <sourceDirectory>src/main/kotlin</sourceDirectory>
    <testSourceDirectory>src/test/kotlin</testSourceDirectory>

    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
        </plugin>
    </plugins>

```

11. Adăugați următoarele dependențe suplimentare (elemente subordonate al *tag-ului <dependencies>*):

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
    <groupId>org.jetbrains.kotlin</groupId>
```

```
<artifactId>kotlin-reflect</artifactId>
<version>${kotlin.version}</version>
</dependency>
```

```
m pom.xml x
20     <properties>
21         <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
22         <kotlin.version>1.3.60</kotlin.version>
23         <kotlin.code.style>official</kotlin.code.style>
24         <junit.version>4.12</junit.version>
25     </properties>
26
27     <dependencies>
28         <dependency>
29             <groupId>org.springframework.boot</groupId>
30             <artifactId>spring-boot-starter-web</artifactId>
31         </dependency>
32         <dependency>
33             <groupId>org.jetbrains.kotlin</groupId>
34             <artifactId>kotlin-stdlib</artifactId>
35             <version>${kotlin.version}</version>
36         </dependency>
37     </dependencies>
```

Clasele în Kotlin sunt, în mod implicit, marcate ca și **final**, deci nu se pot moșteni decât dacă dezvoltatorul le marchează explicit ca **open** (de exemplu, **open class MyClass ...**). **Spring** necesită ca acele clase ce vor primi anumite tipuri de adnotări (cum ar fi **@Component** sau **@Service**) să fie moștenibile, adică marcate cu **open**. Acest lucru este făcut automat de *plugin-ul kotlin-maven-allopen*, aşadar îl veți adăuga ca dependentă la compilare, astfel:

Adăugați următorul element de configurare în interiorul *tag-ului <plugin>*, corespunzător *plugin-ului kotlin-maven-plugin* (consultați figura de mai jos pentru locația exactă):

```
<dependencies>
    <dependency>
        <groupId>org.jetbrains.kotlin</groupId>
        <artifactId>kotlin-maven-allopen</artifactId>
        <version>${kotlin.version}</version>
    </dependency>
</dependencies>
```

```
65     <plugins>
66         <plugin>
67             <groupId>org.jetbrains.kotlin</groupId>
68             <artifactId>kotlin-maven-plugin</artifactId>
69             <version>${kotlin.version}</version>
70             <executions>
71                 <execution>
72                     <id>compile</id>
73                     <phase>compile</phase>
74                     <goals>
75                         <goal>compile</goal>
76                     </goals>
77                 </execution>
78                 <execution>
79                     <id>test-compile</id>
80                     <phase>test-compile</phase>
81                     <goals>
82                         <goal>test-compile</goal>
83                     </goals>
84                 </execution>
85             </executions>
86             <dependencies>
87                 <dependency>
88                     <groupId>org.jetbrains.kotlin</groupId>
89                     <artifactId>kotlin-maven-allopen</artifactId>
90                     <version>${kotlin.version}</version>
91                 </dependency>
92             </dependencies>
93         </plugin>
```

Apoi, adăugați *plugin-ul* **spring** ca și dependență la faza de compilare, cu următorul element copil al tag-ului **<execution>**, pus în locația indicată în figura ce urmează.

```
<configuration>
    <compilerPlugins>
        <plugin>spring</plugin>
    </compilerPlugins>
</configuration>
```

```
65  <plugins>
66      <plugin>
67          <groupId>org.jetbrains.kotlin</groupId>
68          <artifactId>kotlin-maven-plugin</artifactId>
69          <version>${kotlin.version}</version>
70          <executions>
71              <execution>
72                  <id>compile</id>
73                  <phase>compile</phase>
74                  <goals>
75                      <goal>compile</goal>
76                  </goals>
77                  <configuration>
78                      <compilerPlugins>
79                          <plugin>spring</plugin>
80                      </compilerPlugins>
81                  </configuration>
82              </execution>
83          </executions>

```

Pentru productivitate sporită, adăugați ca dependență și **Spring Boot Developer Tools**:
<https://mvnrepository.com/artifact/org.springframework.boot/spring-boot-devtools>

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-devtools</artifactId>
    <version>${project.parent.version}</version>
</dependency>
```

Adăugând această dependență, *plugin-ul Spring Boot* va reîncărca automat artefactele rezultate din compilarea surselor, atunci când acestea sunt modificate de utilizator. Deci, **odată pornite aplicația și server-ul Tomcat (folosind goal-ul **spring-boot:run**)**, dacă modificați sursele și vreți să vezi rezultatul modificărilor, doar le compilați cu *lifecycle-ul* Maven **compile** și atât. **Aplicația vi se reîncarcă automat (dacă nu sunt erori!).**

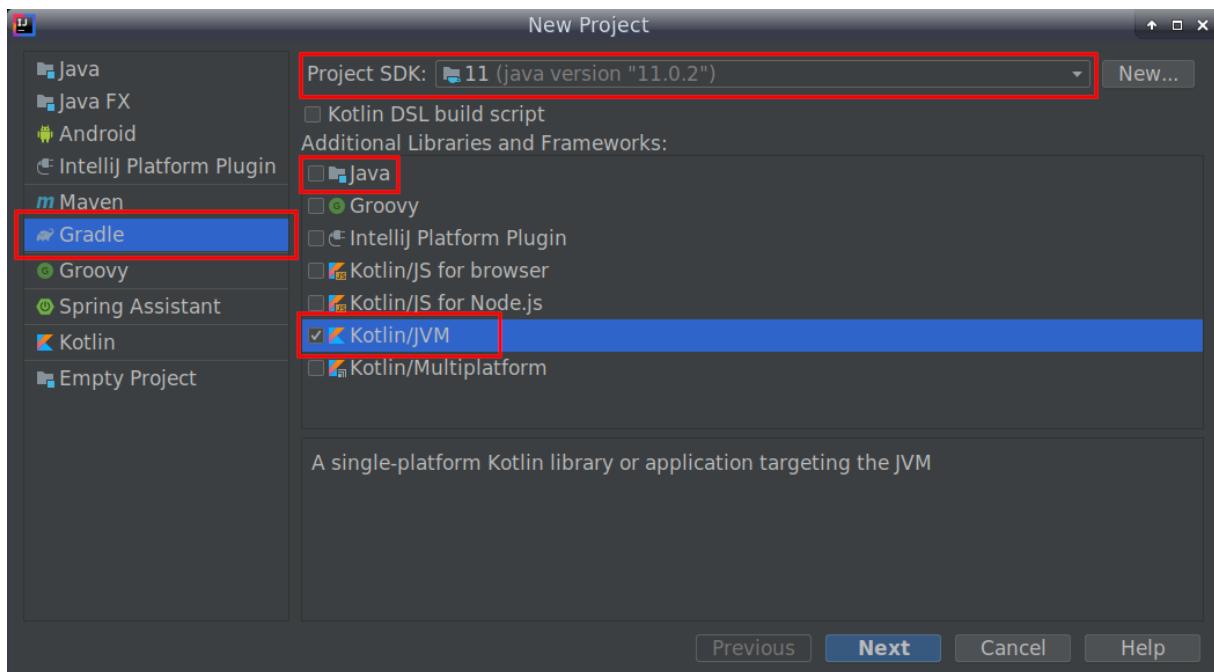
În acest moment, s-a obținut un proiect Maven creat folosind un şablon (*archetype*) pentru Kotlin, în care ați actualizat fișierul POM pentru a include ca și dependențe „**Spring Boot**”, „**Spring Boot Developer Tools**” și „**Spring Boot Starter Web**”. Cea din urmă este necesară pentru construirea de aplicații web, folosind Spring MVC.

1.2. Project Spring Boot folosind **Gradle**

1. Deschideți aplicația **IntelliJ IDEA Community**
2. Selectați „Create New Project” din meniul principal

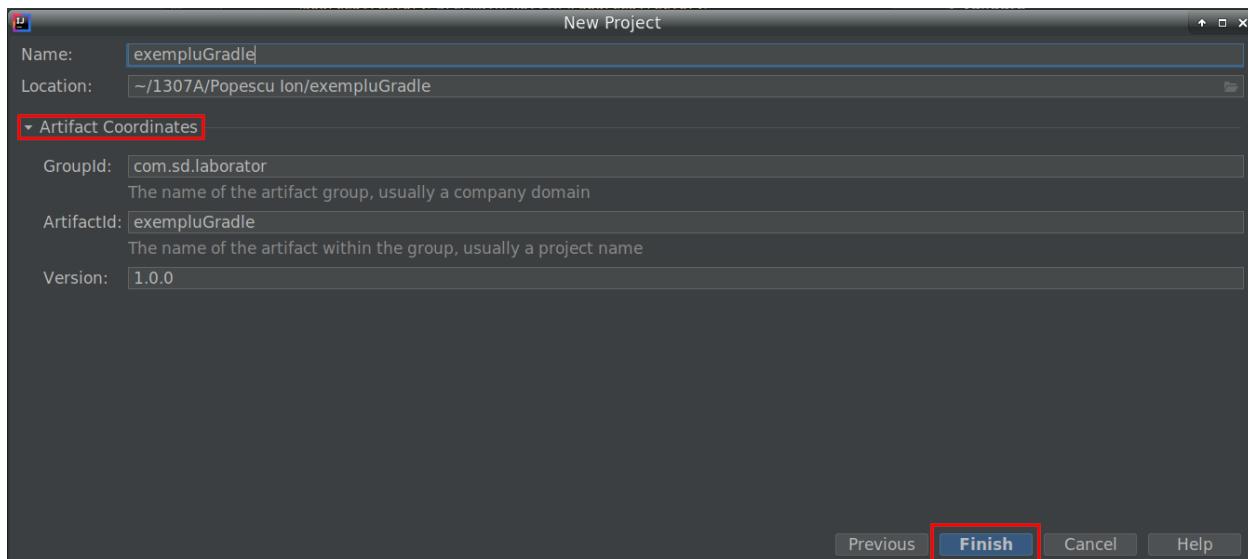


3. Se va alege ca tip de proiect „**Gradle**”, selectați **versiunea 11** de **Java SDK**, apoi, în secțiunea „Additional Libraries and Frameworks”, **debifați „Java”** și bifați „**Kotlin/JVM**”. Se va apăsa pe „**Next**”.



4. Completați numele proiectului și locația sa pe disc, respectiv metadatele proiectului, expandând secțiunea „**Artifact Coordinates**”:
 - a) groupId: **com.sd.laborator**
 - b) artifactId: **exempluGradle**
 - c) version: **1.0.0**

Sisteme Distribuite - Laborator 3



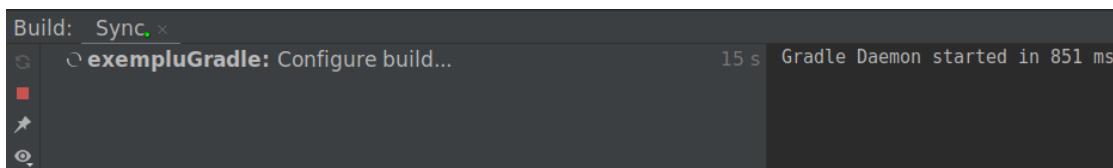
În cazul în care nu există deja, se creează un folder în locația **/home/student** cu numele grupei din care faceți parte. În folder-ul grupei, se creează un alt folder cu numele dvs., acela fiind folder-ul de bază în care se va lucra la laborator.

Exemplu: **/home/student/1307A/Popescu Ion**

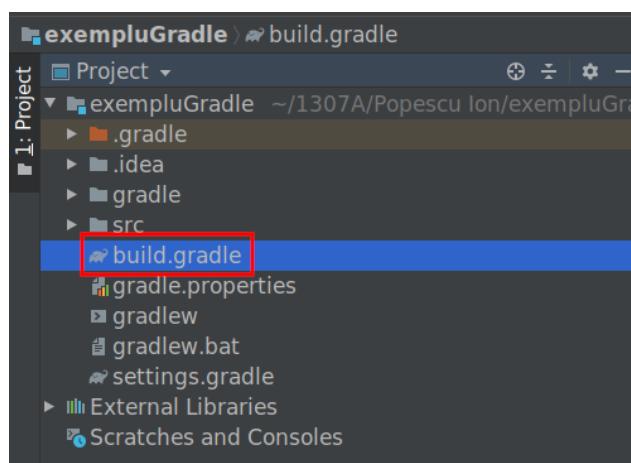
Atenție! Fișierele proiect vor fi plasate **direct** în ultimul folder din calea specificată în „Project location”. IntelliJ nu creează automat niciun subfolder pentru utilizator. Așadar, ar fi indicat ca locația proiectului să indice către un folder gol!

Apăsați „Finish”.

5. Așteptați ca gestionarul de proiect Gradle să configureze automat proiectul nou creat.



6. După ce proiectul a fost configurat și în partea stângă a ferestrei apare ierarhia de fișiere și foldere, deschideți fișierul „**build.gradle**”.



Adăugați următoarea listă de *plugin-uri* (ca subordonați în secțiunea **plugins**):

```
id 'org.springframework.boot' version '2.2.1.RELEASE'
id 'org.jetbrains.kotlin.plugin.spring' version '<VERSIUNE_KOTLIN>'
id 'io.spring.dependency-management' version '1.0.8.RELEASE'
```

Înlocuiți <VERSIUNE_KOTLIN> cu versiunea de Kotlin existentă în repositories la momentul creării proiectului. Aceasta apare pe linia de mai sus, adăugată automat de IntelliJ:

```
id 'org.jetbrains.kotlin.jvm' version '1.3.50'
```

The screenshot shows the build.gradle file in IntelliJ IDEA. The dependencies section has been modified:

```
plugins {
    id 'org.jetbrains.kotlin.jvm' version '1.3.50'
    id 'org.springframework.boot' version '2.2.1.RELEASE'
    id 'org.jetbrains.kotlin.plugin.spring' version '1.3.50'
    id 'io.spring.dependency-management' version '1.0.8.RELEASE'
}

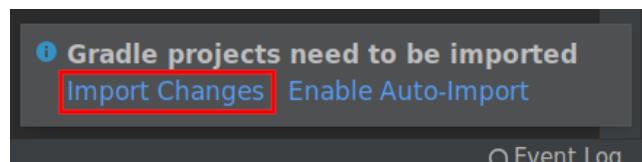
group 'com.sd.laborator'
version '1.0.0'

repositories {
    mavenCentral()
}

dependencies {
    implementation "org.jetbrains.kotlin:kotlin-stdlib-jdk8"
}
```

Red boxes highlight the 'version' attribute for each plugin and dependency, indicating they have been updated.

După modificarea fișierului **build.gradle**, IntelliJ va afișa un mesaj în partea din dreapta-jos a ferestrei. Selectați „Import Changes”.



Prin aceasta se va realiza sincronizarea automată a dependențelor specificate în fișierul de configurare **build.gradle**, în momentul în care este editat de către utilizator. Ca efect, modificarea făcută în pasul de mai sus va determina ca IntelliJ să descarce automat toate pluginurile specificate în secțiunea **plugins**.

7. Adăugați **Kotlin Reflect** și **Spring Boot Web Application** ca și dependențe (elemente subordonate ale secțiunii **dependencies**):

```
implementation "org.jetbrains.kotlin:kotlin-reflect"
implementation "org.springframework.boot:spring-boot-starter-web"
```

Sisteme Distribuite - Laborator 3

```
8     group 'com.sd.laborator'
9     version '1.0.0'
10
11    repositories {
12        mavenCentral()
13    }
14
15    dependencies {
16        implementation "org.jetbrains.kotlin:kotlin-stdlib-jdk8"
17        implementation "org.jetbrains.kotlin:kotlin-reflect"
18        implementation "org.springframework.boot:spring-boot-starter-web"
19    }
20
21    compileKotlin {
22        kotlinOptions.jvmTarget = "1.8"
23    }

```

Pentru productivitate sporită, adăugați ca dependență și **Spring Boot Developer Tools**:
<https://docs.spring.io/spring-boot/docs/1.5.16.RELEASE/reference/html/using-boot-devtools.html>

```
compileOnly "org.springframework.boot:spring-boot-devtools"
```

```
▶   dependencies {
        implementation "org.jetbrains.kotlin:kotlin-stdlib-jdk8"
        implementation "org.jetbrains.kotlin:kotlin-reflect"
        implementation "org.springframework.boot:spring-boot-starter-web"
        compileOnly "org.springframework.boot:spring-boot-devtools"
    }
```

Apoi, adăugați o secțiune nouă în fișierul **build.gradle**:

```
configurations {
    compileOnly
    runtimeClasspath {
        extendsFrom compileOnly
    }
}
```

```
15    configurations {
16        compileOnly
17        runtimeClasspath {
18            extendsFrom compileOnly
19        }
20    }
21
22    dependencies {
23        implementation "org.jetbrains.kotlin:kotlin-stdlib-jdk8"
24        implementation "org.jetbrains.kotlin:kotlin-reflect"
25        implementation "org.springframework.boot:spring-boot-starter-web"
26
27        compileOnly "org.springframework.boot:spring-boot-devtools"
28    }
29
```

Adăugând această dependență, *plugin-ul Spring Boot* va reîncărca automat artefactele rezultate din compilarea surselor, atunci când acestea sunt modificate de utilizator. Deci, **odată** pornite aplicația și server-ul Tomcat (folosind *task-ul application/bootRun*), dacă modificați

sursele și vreți să vedeți rezultatul modificărilor, doar le compilați cu *task-ul Gradle build/build* și atât. Aplicația vi se reîncarcă automat (dacă nu sunt erori!).

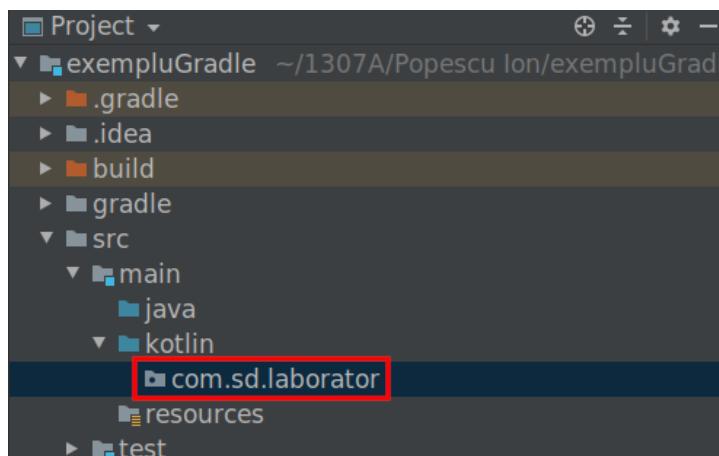
În acest moment, aveți un proiect Gradle, la care s-a adăugat librăria adițională Kotlin/JVM, proiect în care ați actualizat fișierul **build.gradle** pentru a include ca și dependențe „**Spring Boot**” și „**Spring Boot Starter Web**”. Cea din urmă este necesară pentru construirea de aplicații web, folosind Spring MVC.

2. Crearea unei aplicații simple

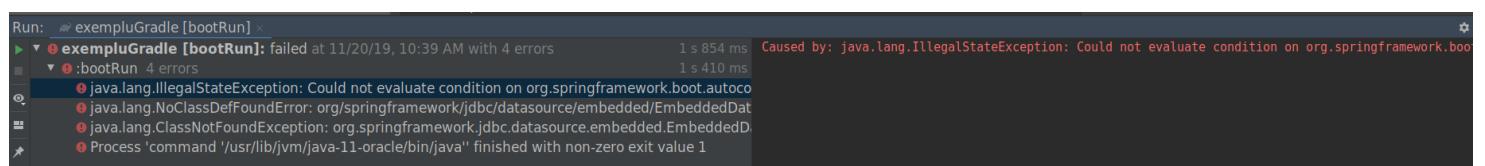
Pentru început se va crea o aplicație Spring MVC de test, care va afișa un simplu mesaj: „Hello World”. Aceasta va cuprinde un serviciu expus în calea /helloworld, care este accesibilă local prin intermediul URL-ului: <http://localhost:8080/helloworld>.

2.1. Adăugarea fișierelor sursă

- **pentru proiecte de tip Gradle:** se creează manual un pachet în care vor fi plasate fișierele sursă. În panoul Project din partea stângă apăsați dreapta pe folder-ul **kotlin** din <Nume_Proiect>/src/main → New → Package → introduceți „**com.sd.laborator**” și apăsați „OK”.



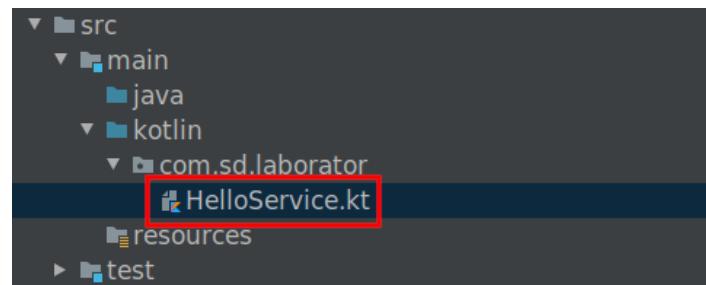
Atenție! Dacă nu se creează cel puțin un pachet pentru fișierele sursă și se plasează direct în folder-ul **kotlin** (adică în pachetul **default**), se va primi o excepție generată de Spring la pornirea aplicației:



- **pentru proiecte de tip Maven:** un pachet implicit va fi creat automat pe baza câmpului **GroupId** introdus ca metadată în pasul 4 de creare a unui proiect Maven. În acest caz, numele pachetului va fi **com.sd.laborator**.

În pachetul **com.sd.laborator**, se creează un fișier sursă Kotlin, denumit **HelloService.kt**.

Se va apăsa dreapta pe numele pachetului → New → **Kotlin File/Class** → Se va introduce „**HelloService**” ca și nume și se va lăsa selectat tipul de fișier „File”.



În continuare, se creează o clasă denumită **HelloService**, ce conține o funcție simplă fără parametri, denumită **getHello()**. Funcția va returna șirul de caractere „Hello World!”.

```
package com.sd.laborator

import org.springframework.stereotype.Service

@Service
class HelloService {
    fun getHello() = "Hello World!"
}
```

Pentru ca *framework*-ul Spring să recunoască această clasă ca și componentă (*bean*) și să o adauge în contextul de execuție pentru utilizarea la cerere, trebuie să adnotați clasa cu „**@Service**”. Pentru aceasta, este nevoie de import-ul interfeței **Service**, aflată în pachetul **org.springframework.stereotype**.

În mod asemănător, se creează o clasă *controller*, denumită **HelloController**, într-un fișier sursă Kotlin separat:

```
package com.sd.laborator

@Controller
class HelloController {
    val service: HelloService = HelloService()

    @RequestMapping(value = ["/helloworld"], method =
    [RequestMethod.GET])
    @ResponseBody
    fun hello() = service.getHello()
}
```

Importurile necesare nu au fost adăugate în mod intenționat, pentru a folosi facilitățile **IntelliSense** ale mediului de dezvoltare. IntelliJ avertizează și oferă potențiale soluții în cazul situațiilor de acest tip, spre exemplu atunci când se utilizează clase inexistente, care nu au fost importate, variabile neinitializate, specificatori de acces lipsă etc.



Se va apăsa pe primul identificator pe care IntelliJ nu îl găsește (în acest caz, nu este cunoscută interfața **Controller**), și apăsați **ALT+ENTER**.



Se va alege „**Import**” din lista de soluții propuse, iar IntelliJ va adăuga automat la începutul fișierului această linie:

```
import org.springframework.stereotype.Controller
```

Se va proceda la fel pentru toate celelalte erori raportate de mediul de dezvoltare, pentru a importa automat toate dependențele Spring de care aplicația are nevoie.

Fișierul **HelloController.kt** conține o clasă controller (adnotată cu „**@Controller**”), care expune o cale accesibilă prin HTTP: **/helloworld**. În controller, este instantiat un obiect de tip **HelloService** prin care este accesat serviciu creat anterior (care conține metoda de interes **getHello()**). Mai concret: în momentul în care utilizatorul face o cerere HTTP de tip GET către calea **/helloworld** pe server-ul pe care se execută aplicația Spring (în acest caz, **localhost**), Spring va instanția un *bean* de tip **HelloService** în contextul de execuție, din care va apela metoda **getHello()**. Aceasta returnează un sir de caractere ce este trimis clientului apelant ca și corp de răspuns HTTP (**@ResponseBody**).

(Maven) Proiectul Maven va conține un fișier sursă Kotlin numit **Hello.kt**. **Ștergeți conținutul său** înainte de următorul pas.

Creați alt fișier sursă **Hello.kt** (dacă nu este deja creat), și adăugați următorul cod:

```
package com.sd.laborator

@SpringBootApplication
class Hello
```

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

Adăugați dependențele lipsă semnalate de IntelliJ în aceeași manieră explicată mai sus.

Observați că funcția `main()` conține un apel de metodă `runApplication`, ce acceptă o clasă template, în acest caz `Hello`. `runApplication` este metoda de inițializare a unei aplicații Spring. Clasa template trimisă ca parametru trebuie adnotată cu `@SpringBootApplication` și reprezintă clasa folosită pentru configurarea aplicației (în cazul acesta, `Hello` nu conține niciun membru și nicio metodă).

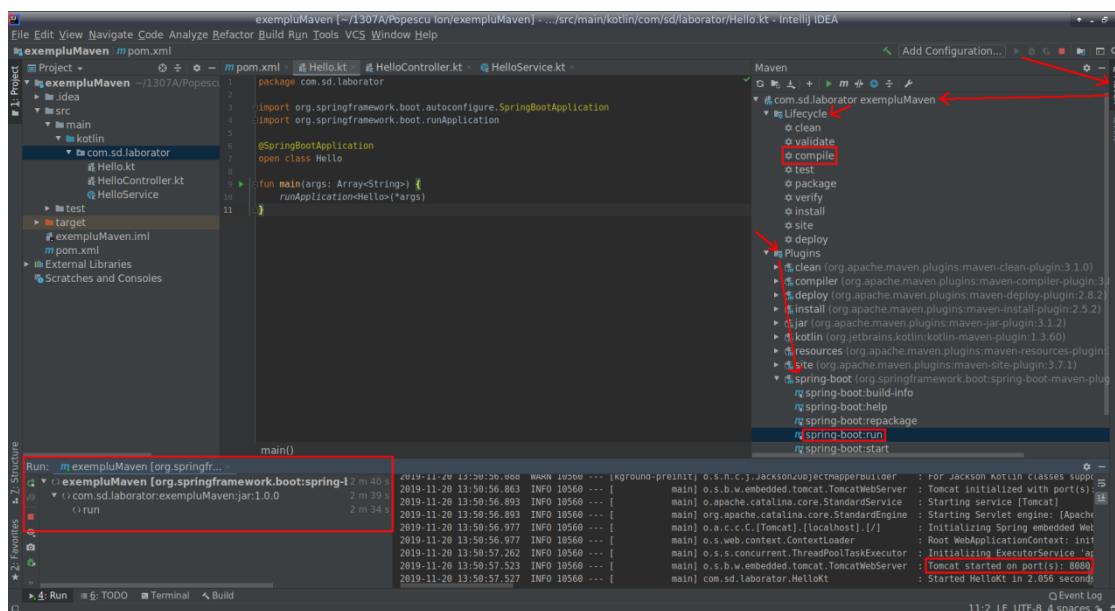
Adnotarea `@SpringBootApplication` declanșează scanarea de componente, configurarea automată și scanarea de proprietăți configurabile adiționale. Pentru detalii se poate consulta documentația, disponibilă aici:

<https://docs.spring.io/springboot/docs/current/api/org/springframework/boot/autoconfigure/SpringBootApplication.html>

2.2. Compilarea și execuția aplicației Spring

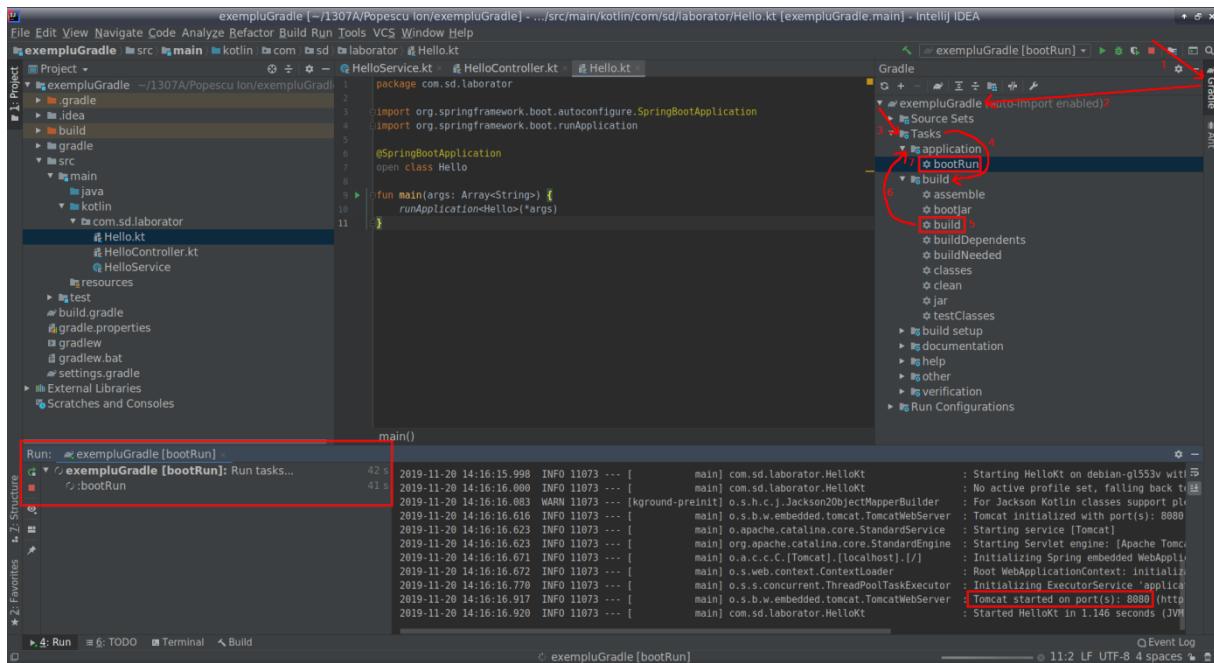
2.2.1. Compilare și execuție proiect Maven

Se va expanda meniul „Maven” din partea dreaptă a ferestrei și execuțiți *lifecycle-ul „compile”*. După compilare, în caz că nu există erori, aplicația Spring se poate executa folosind un *Maven goal* denumit „`spring-boot:run`” din categoria „**Plugins → spring-boot**”. Se va porni automat un server **Apache Tomcat**, expus în mod implicit pe portul **8080**, care așteaptă cereri de la clienți.



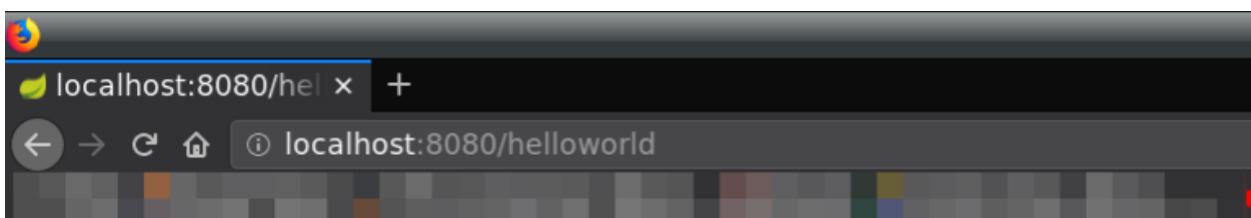
2.2.2. Compilare și execuție proiect Gradle

Se va expanda meniu „Gradle” din partea dreaptă a ferestrei și execuți task-ul „**compile**” din categoria „**build**”. După compilare, în caz că nu există erori, aplicația Spring se poate executa folosind un task *Gradle* denumit „**bootRun**” din categoria „**Tasks → application**”. Se va porni automat un server **Apache Tomcat**, expus în mod implicit pe portul **8080**, care așteaptă cereri de la clienți.



2.3. Testarea funcționării aplicației Spring

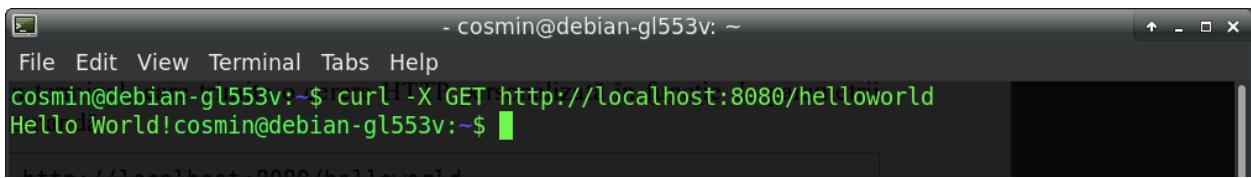
Puteți testa exemplul, navigând la adresa URL <http://localhost:8080/helloworld> cu un browser web. Ar trebui să să se primească ca răspuns mesajul „Hello World!”.



De asemenea, o altă variantă de a testa funcționarea serviciului expus este folosirea comenzi „curl” din terminalul IntelliJ, care trimite o cerere HTTP personalizată în funcție de parametrii dați la linia de comandă:

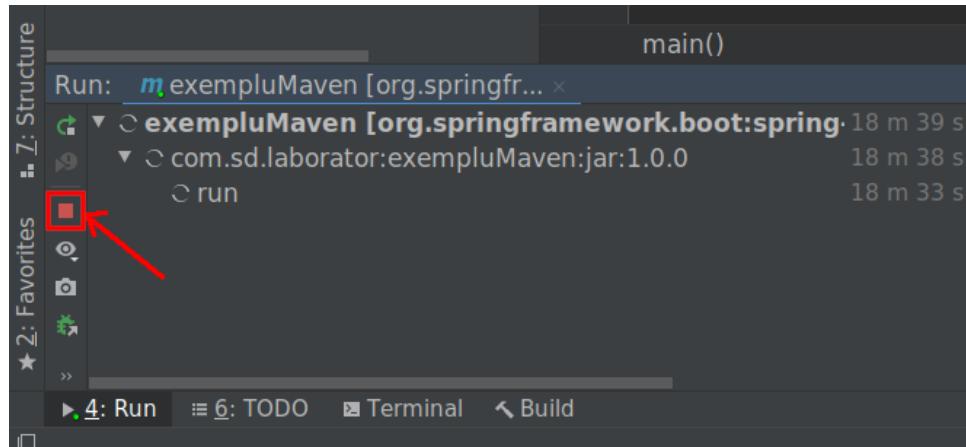
```
curl -X GET http://localhost:8080/helloworld
```

În acest exemplu, **curl** va trimite o cerere HTTP de tip **GET** către server-ul **localhost**, portul **8080**, și calea **/helloworld**.



```
- cosmin@debian-gl553v: ~
File Edit View Terminal Tabs Help
cosmin@debian-gl553v:~$ curl -X GET http://localhost:8080/helloworld
Hello World!cosmin@debian-gl553v:~$
```

Aplicația își continuă execuția la infinit, deoarece server-ul funcționează în continuare, așteptând alte conexiuni din partea clientilor. Pentru a opri aplicația, se va apăsa pe butonul roșu „Stop” în partea din stânga jos a ferestrei.

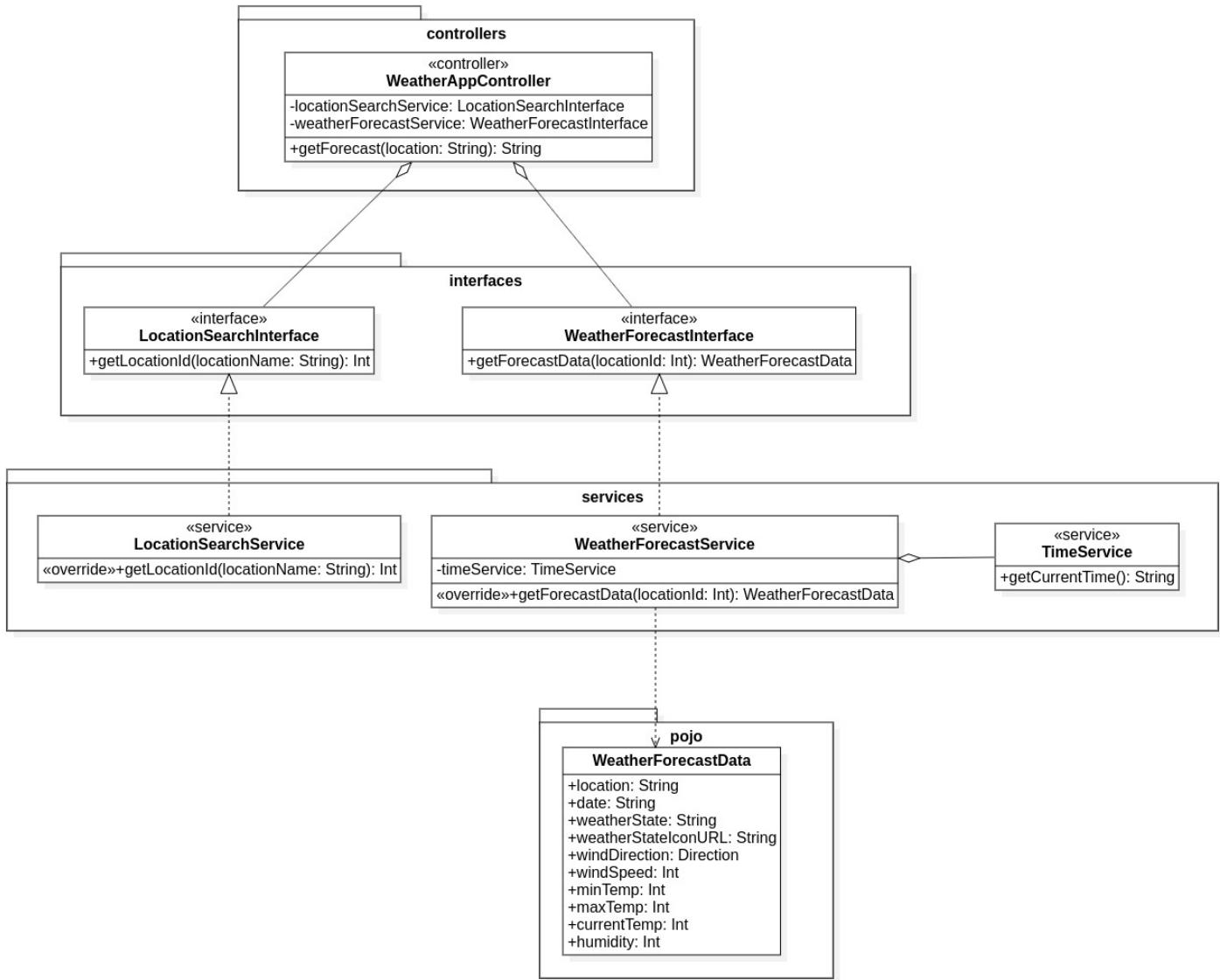


2.4. Modificarea surselor și reîncărcarea aplicației

Dacă modificați fișierele sursă și doriți să testați modificările făcute, datorită dependenței **Spring Boot Developer Tools**, trebuie doar să recompilezi aplicația în IntelliJ (cu metoda corespunzătoare gestionarului de proiect ales), iar apoi Spring Boot va reîncărca automat artefactele modificate prin recomplirare.

3. Aplicație meteo cu Spring Web

În următorul exemplu veți crea o aplicație meteo care preia câteva date despre vreme folosind servicii web dezvoltate cu *framework*-ul Spring. Arhitectura este reprezentată în diagrama următoare:



Clasele au fost împărțite în pachete în funcție de scopul acestora în modelarea aplicației. Spre exemplu, pentru implementare, se va folosi o abordare de tip „*bottom-up*”. Fiecare pachet, clasă, respectiv rolul în aplicație sunt explicate în cele ce urmează.

Creați un proiect Spring Boot nou, la care adăugați ca dependență **Spring Web** (exact cum ați procedat la aplicația demonstrativă din secțiunea anterioară), folosind un utilitar manager de proiect la alegere (Maven / Gradle). Denumiți-l, spre exemplu, **WeatherApp**.

Nu vă speriați de blocurile de cod ce urmează. Sunt explicate pas cu pas!

3.1. Clasele POJO

Se pornește de la pachetul **pojo**, ce conține obiecte de tip *Plain Old Java Object*. Acestea sunt folosite pentru a trimite unui client datele cerute, încapsulate într-un obiect. De asemenea,

POJO-urile mai sunt utilizate în *layer-ul* de prezentare pentru afișarea datelor (de exemplu, dacă se folosește un *framework* de prezentare, cum ar fi *Thymeleaf*).

Atenție: obiectele POJO nu procesează date. Ele doar reprezintă rezultatul unei procesări anterioare (făcute în codul de *business*).

În acest caz, veți reprezenta datele meteo încapsulate într-un obiect POJO denumit **WeatherForecastData**. Pe lângă datele obișnuite, obiectul conține și un membru care reprezintă direcția din care bate vântul, sub formă de enumerație cu punctele cardinale.

Creați pachetul **com.sd.laborator.pojo** și adăugați clasa de mai sus sub formă de **data class**. În Kotlin, o clasă de tip **data class** are ca și scop încapsularea de date, iar datele sunt reprezentate de proprietăți mutabile sau imutabile. **Aceste clase trebuie obligatoriu să aibă un constructor primar.**

```
package com.sd.laborator.pojo

data class WeatherForecastData (
    var location: String,
    var date: String,
    var weatherState: String,
    var weatherStateIconURL: String,
    var windDirection: String,
    var windSpeed: Int, // km/h
    var minTemp: Int, // grade celsius
    var maxTemp: Int,
    var currentTemp: Int,
    var humidity: Int // procent
)
```

3.2. Serviciile web

În continuare, trebuie implementate serviciile web (clasele marcate cu stereotipul **service**). Pentru a ascunde și a nu depinde de implementarea serviciilor atunci când sunt utilizate de *controller*, se folosesc **interfețe** (unde este cazul). Cu excepția serviciului **TimeService**, care este utilizat de un alt serviciu, celelalte 2 derivă din interfețe.

Începeți cu serviciul **TimeService**, deoarece este cel mai simplu. Acesta expune o metodă **getCurrentTime()** ce returnează data și ora curentă sub formă de sir de caractere. Creați clasa **TimeService** într-un pachet **services** subordonat pachetului principal **com.sd.laborator**.

```
package com.sd.laborator.services

import org.springframework.stereotype.Service
import java.text.SimpleDateFormat
import java.util.*

@Service
class TimeService {
    fun getCurrentTime():String {
        val formatter = SimpleDateFormat("dd/MM/yyyy HH:mm:ss")
        return formatter.format(Date())
    }
}
```

Clasa este anotată cu **@Service** pentru ca *framework-ul* Spring să o poată găsi și configura corespunzător sub formă de **serviciu** în timpul scanării de componente.

Pentru celelalte 2 servicii web, începeți cu interfețele. Creați un pachet **interfaces**

subordonat pachetului principal și adăugați cele 2 interfețe:

• LocationSearchInterface

```
package com.sd.laborator.interfaces

interface LocationSearchInterface {
    fun getLocationId(locationName: String): Int
}
```

Aceasta expune o metodă pentru preluarea unui identificator de locație (**WOEID - Where On Earth ID**) ce va fi folosit mai departe la prognoza meteo. De unde a apărut acest identificator? Este utilizat de API-ul pe care îl veți folosi aici pentru a prelua datele despre vreme, și anume MetaWeather API: <https://www.metaweather.com/api/>. S-a ales acest API deoarece este ușor de folosit și **nu necesită cont și cheie API**.

• WeatherForecastInterface

```
package com.sd.laborator.interfaces

import com.sd.laborator.pojo.WeatherForecastData

interface WeatherForecastInterface {
    fun getForecastData(locationId: Int): WeatherForecastData
}
```

Aici se expune o metodă care, pe baza unui identificator de tip **WOEID**, returnează un obiect ce încapsulează toate informațiile meteo.

Având interfețele definite, se continuă cu implementările:

• LocationSearchService

```
package com.sd.laborator.services

import com.sd.laborator.interfaces.LocationSearchInterface
import org.springframework.stereotype.Service
import java.net.URL
import org.json.JSONObject
import java.net.URLEncoder
import java.nio.charset.StandardCharsets

@Service
class LocationSearchService : LocationSearchInterface {
    override fun getLocationId(locationName: String): Int {
        // codificare parametru URL (deoarece poate conține caractere speciale)
        val encodedLocationName = URLEncoder.encode(locationName,
StandardCharsets.UTF_8.toString())

        // construire obiect de tip URL
        val locationSearchURL =
URL("https://www.metaweather.com/api/location/search/?query=$encodedLoca-
tionName")

        // preluare răspuns HTTP (se face cerere GET și se preia
        // conținutul răspunsului sub formă de text)
    }
}
```

```

    val rawResponse: String = locationSearchURL.readText()

    // parsare obiect JSON
    val responseRootObject = JSONObject("{\"data\": ${rawResponse}}")
    val responseContentObject =
responseRootObject.getJSONArray("data").takeUnless { it.isEmpty }
        ?.getJSONObject(0)
    return responseContentObject?.getInt("woeid") ?: -1
}
}
}

```

Atenție: pentru a utiliza clasa `JSONObject`, adăugați artefactual `org.json.json` (<https://mvnrepository.com/artifact/org.json/json>) ca **dependentă globală**:

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

```

<dependency>
    <groupId>org.json</groupId>
    <artifactId>json</artifactId>
    <version>20190722</version>
</dependency>

```

- **pentru proiecte Gradle:** adăugați următoarea dependență în interiorul secțiunii `dependencies`:

```
compile group: 'org.json', name: 'json', version: '20190722'
```

Implementarea acestui serviciu procedează în felul următor: trimit o cerere HTTP către un URL în format standard (descriș în API-ul din documentație), prin care se poate căuta o anumită locație pe baza numelui complet sau parțial. URL-ul este de forma:

[https://www.metaweather.com/api/location/search/?query=\\$locationName](https://www.metaweather.com/api/location/search/?query=$locationName)

Răspunsul returnat de serviciul MetaWeather la acest tip de interogare poate fi verificat direct din browser și este de forma:

```

{
  "0": {
    "title": "San Francisco",
    "location_type": "City",
    "woeid": 2487956,
    "latt_long": "37.777119, -122.41964"
  },
  "1": {
    "title": "San Diego",
    "location_type": "City",
    "woeid": 2487889,
    "latt_long": "32.715691, -117.161720"
  },
  "2": {
    "title": "San Jose",
    "location_type": "City",
    "woeid": 2488042,
    "latt_long": "37.338581, -121.885567"
  },
  "3": {
    "title": "San Antonio",
    "location_type": "City",
    "woeid": 2487796,
    "latt_long": "29.424580, -98.494614"
  }
}

```

Se observă că MetaWeather răspunde cu un obiect JSON ce conține un vector cu rezultatele căutării utilizatorului (în acest caz, s-a căutat după cuvântul cheie „San”. Pentru simplitate, se va lăsa în considerare doar primul rezultat al căutării (dacă există), deci primul element din vectorul returnat.

Pentru parsarea obiectului JSON, se folosește clasa **JSONObject**, care primește în constructor acel obiect sub formă de sir de caractere. Deoarece la această interogare se returnează un vector care nu este încapsulat într-un obiect JSON părinte, se adaugă manual un element rădăcină denumit **data**:

```
val responseRootObject = JSONObject("{\"data\": ${rawResponse}}")
```

Conform standardului JSON, trebuie obligatoriu să existe un element rădăcină ce încapsulează restul elementelor. Dacă nu faceți această încapsulare „la mânuță”, clasa **JSONObject** va genera o excepție la parsare.

În continuare, se preia obiectul de tip vector ca și elementul copil al **data**: **responseRootObject.getJSONArray("data")**. În cazul în care căutarea locației nu a returnat niciun rezultat, obiectul rezultat va fi un vector gol, și deci nu se dorește apelul niciunei metode asupra acestuia. Așadar, se folosește (spre exemplu), blocul **takeUnless** specific Kotlin, care, pe baza predicatului **isEmpty**, apelează înlățuit metoda **getJSONObject()** doar în cazul în care obiectul apelant satisfacă predicatul (adică vectorul nu este gol).

```
val responseContentObject =  
    responseRootObject.getJSONArray("data").takeUnless { it.isEmpty }  
        ?.getJSONObject(0)
```

Blocul **takeUnless**, urmat de **getJSONObject(0)** returnează deci conținutul primului element din vectorul cu rezultate, adică ceea ce este evidențiat în figură:

```
▼ 0:  
  title: "San Francisco"  
  location_type: "City"  
  woeid: 2487956  
  latt_long: "37.777119, -122.41964"
```

Reamintire: s-a folosit operatorul *safe call* (**?.**) din Kotlin, deoarece **takeUnless** returnează **null** în cazul în care predicatul trimis nu este satisfăcut.

Având acces direct la elementele de interes, se poate apela acum **get<TIPOFDATE>(nume_proprietate)** pentru a prelua ceea ce se dorește, de exemplu câmpul **woeid**, de tip număr întreg:

```
responseContentObject?.getInt("woeid") ?: -1
```

Acest câmp va fi folosit mai departe pentru prognoza meteo a locației cu identificatorul respectiv. S-a folosit și aici operatorul *safe call*, în conjuncție cu operatorul *null coalescing* (**?:**) pentru a returna **-1** dacă nu s-a găsit nicio locație pentru cuvintele cheie căutate de utilizator.

• WeatherForecastService

```
package com.sd.laborator.services  
  
import com.sd.laborator.interfaces.WeatherForecastInterface  
import com.sd.laborator.pojo.WeatherForecastData  
import org.json.JSONObject
```

```

import org.springframework.stereotype.Service
import java.net.URL
import kotlin.math.roundToInt

@Service
class WeatherForecastService (private val timeService: TimeService) :
WeatherForecastInterface {
    override fun getForecastData(locationId: Int): WeatherForecastData {
        // ID-ul locației nu trebuie codificat, deoarece este numeric
        val forecastDataURL =
URL("https://www.metaweather.com/api/location/$locationId/")

        // preluare conținut răspuns HTTP la o cerere GET către URL-ul de
mai sus
        val rawResponse: String = forecastDataURL.readText()

        // parsare obiect JSON primit
        val responseRootObject = JSONObject(rawResponse)
        val weatherDataObject =
responseRootObject.getJSONArray("consolidated_weather").getJSONObject(0)

        // construire și returnare obiect POJO care încapsulează datele
meteo
        return WeatherForecastData(
            location = responseRootObject.getString("title"),
            date = timeService.getCurrentTime(),
            weatherState =
weatherDataObject.getString("weather_state_name"),
            weatherStateIconURL =
"https://www.metaweather.com/static/img/weather/png/${weatherDataObject.g
etString("weather_state_abbr")}.png",
            windDirection =
weatherDataObject.getString("wind_direction_compass"),
            windSpeed =
weatherDataObject.getFloat("wind_speed").roundToInt(),
            minTemp =
weatherDataObject.getFloat("min_temp").roundToInt(),
            maxTemp =
weatherDataObject.getFloat("max_temp").roundToInt(),
            currentTemp =
weatherDataObject.getFloat("the_temp").roundToInt(),
            humidity =
weatherDataObject.getFloat("humidity").roundToInt()
        )
    }
}

```

Ca și principiu de funcționare, acest serviciu este asemănător cu **LocationSearchService**. URL-ul expus de API-ul MetaWeather pentru preluarea datelor meteo este de forma:

[https://www.metaweather.com/api/location/**1234567890**](https://www.metaweather.com/api/location/1234567890)

Conținutul răspunsului HTTP este de forma:

```

{
  "consolidated_weather": [
    {
      "id": 5764786186878976,
      "weather_state_name": "Clear",
      "weather_state_abbr": "c",
      "wind_direction_compass": "WSW",
      "created": "2020-01-13T10:20:11.665627Z",
      "applicable_date": "2020-01-13",
      "min_temp": -0.1149999999999999,
      "max_temp": 7.02,
      "the_temp": 4.734999999999999,
      "wind_speed": 5.828544966945041,
      "wind_direction": 245.16662387203695,
      "air_pressure": 1027,
      "humidity": 77,
      "visibility": 9.46037640181341,
      "predictability": 68
    },
    {
      "id": 6132782851948544,
      "weather_state_name": "Light Cloud",
      "weather_state_abbr": "lc",
      "wind_direction_compass": "WSW",
      "created": "2020-01-13T10:20:14.667391Z",
      "applicable_date": "2020-01-14",
      "min_temp": -1.585,
      "max_temp": 7.885,
      "the_temp": 4.98,
      "wind_speed": 2.773149102280018,
      "wind_direction": 244.50026675861483,
      "air_pressure": 1026.5,
      "humidity": 81,
      "visibility": 6.482144277419868
    }
  ]
}

```

Observați că răspunsul este un obiect JSON ce conține deja datele încapsulate într-un element rădăcină, numit **consolidated_weather**. Ca și subordonat, se primește un vector cu date meteo de la mai mulți furnizori. MetaWeather agregă rezultate de la mai mulți furnizori meteo și le încapsulează într-un singur răspuns. Din nou, pentru simplitate, luati în considerare doar răspunsul de la primul furnizor (primul element din vectorul rezultat).

Așadar, se preia din prima obiectul conținut în primul element al vectorului respectiv:

```

val weatherDataObject =
responseRootObject.getJSONArray("consolidated_weather").getJSONObject(0)

```

Deci, variabila weatherDataObject încapsulează următoarele date în acest moment:

```

▼ consolidated_weather:
  ▼ 0:
    id: 5764786186878976
    weather_state_name: "Clear"
    weather_state_abbr: "c"
    wind_direction_compass: "WSW"
    created: "2020-01-13T10:20:11.665627Z"
    applicable_date: "2020-01-13"
    min_temp: -0.1149999999999999
    max_temp: 7.02
    the_temp: 4.734999999999999
    wind_speed: 5.828544966945041
    wind_direction: 245.16662387203695
    air_pressure: 1027
    humidity: 77
    visibility: 9.46037640181341
    predictability: 68
  ▼ 1:
    id: 6132782851948544
    weather_state_name: "Light Cloud"
    weather_state_abbr: "lc"
    wind_direction_compass: "WSW"
    created: "2020-01-13T10:20:14.6673017Z"
  
```

Deoarece acum aveți acces direct (prin această variabilă) la proprietățile de interes din obiectul JSON, informațiile se pot prelua cu metodele de tipul `get<TIPOFDATE>(nume_proprietate)`:

- starea generală a vremii:

```
weatherDataObject.getString("weather_state_name")
```

- viteza vântului, rotunjită la întreg

```
weatherDataObject.getFloat("wind_speed").roundToInt()
```

§.a.m.d.

Excepție face numele locației pentru care se afișează prognoza, care face parte din elementul rădăcină **consolidated_weather**, și deci se preia astfel:

```
responseRootObject.getString("title")
```

3.3. Clasa controller

```

package com.sd.laborator.controllers

import com.sd.laborator.interfaces.LocationSearchInterface
import com.sd.laborator.interfaces.WeatherForecastInterface
import com.sd.laborator.pojo.WeatherForecastData
import com.sd.laborator.services.TimeService
import org.springframework.beans.factory.annotation.Autowired
import org.springframework.stereotype.Controller
import org.springframework.web.bind.annotation.PathVariable
import org.springframework.web.bind.annotation.RequestMapping
import org.springframework.web.bind.annotation.RequestMethod
  
```

```

import org.springframework.web.bind.annotation.ResponseBody

@Controller
class WeatherAppController {
    @Autowired
    private lateinit var locationSearchService: LocationSearchInterface

    @Autowired
    private lateinit var weatherForecastService: WeatherForecastInterface

    @RequestMapping("/getforecast/{location}", method =
    [RequestMethod.GET])
    @ResponseBody
    fun getForecast(@PathVariable location: String): String {
        // se incarca preluarea WOEID-ului locatiei primite in URL
        val locationId = locationSearchService.getLocationId(location)

        // dacă locația nu a fost găsită, răspunsul va fi corespondent
        if (locationId == -1) {
            return "Nu s-au putut gasi date meteo pentru cuvintele cheie
\"$location\"!"
        }

        // pe baza ID-ului de locație, se interoghează al doilea serviciu
        care returnează datele meteo
        // încapsulează într-un obiect POJO
        val rawForecastData: WeatherForecastData =
        weatherForecastService.getForecastData(locationId)

        // fiind obiect POJO, funcția toString() este suprascrisă pentru
        o afișare mai prietenoasă
        return rawForecastData.toString()
    }
}

```

Adnotarea **@Autowired** marchează faptul că proprietățile **locationSearchService**, respectiv **weatherForecastService** sunt dependențe rezolvate prin facilitățile *dependency injection* ale *framework-ului* Spring. După ce Spring construiește *bean-ul* corespondent clasei **WeatherAppController**, rezolvă și injectează automat aceste 2 dependențe (în acest caz, serviciile de care *controller-ul* are nevoie).

Observați că aceste 2 proprietăți nu sunt instanțiate nicăieri în clasă, ci sunt doar declarate. **Inițializarea este făcută automat de Spring**. De aceea a fost nevoie de specificatorul **lateinit**, deoarece Kotlin nu permite declararea unei proprietăți fără inițializarea acesteia în constructor sau imediat după declarație.

Mai mult de atât, tipurile de date al proprietăților respective corespund interfețelor serviciilor create anterior, **și nu implementărilor**. Motivul este, din nou, decuplarea *controller-ului* de implementarea efectivă a serviciilor dependente: dacă dezvoltatorul vrea să schimbe unul din servicii și să folosească o altă clasă **care respectă același contract** (aceeași interfață), atunci ar trebui modificat și recompilat *controller-ul*! În acest caz, dacă dorîți să modificați complet serviciul de căutare a locației, spre exemplu, pur și simplu creați o altă clasă care implementează interfața **LocationSearchInterface** și nu modificați nimic altceva.

În *controller* se definește o metodă de tratare a cererilor HTTP de tip GET către calea **/getforecast/{location}**. **location** este un parametru trimis de utilizator direct în URL, aşadar acest parametru este injectat ca parametru al metodei **getForecast()** sub formă de

variabilă de cale (*path variable*). De aici adnotarea **@PathVariable**.

În continuare, sunt utilizate serviciile disponibile pentru a:

1. Căuta WOEID-ul pe baza cuvintelor cheie primite de la utilizator
2. Returna datele meteo specifice aceluui WOEID sub formă de obiect POJO

3.4. Punctul de intrare în aplicația Spring

Având toate elementele componente definite, se creează clasa principală ce reprezintă punctul de intrare (*entrypoint*) al aplicației Spring. Aici se declară clasa de configurare, în care se pot face configurații suplimentare.

Adăugați un fișier nou **WeatherApp.kt** în pachetul **com.sd.laborator**, cu următorul conținut:

```
package com.sd.laborator

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

@SpringBootApplication
class WeatherApp

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

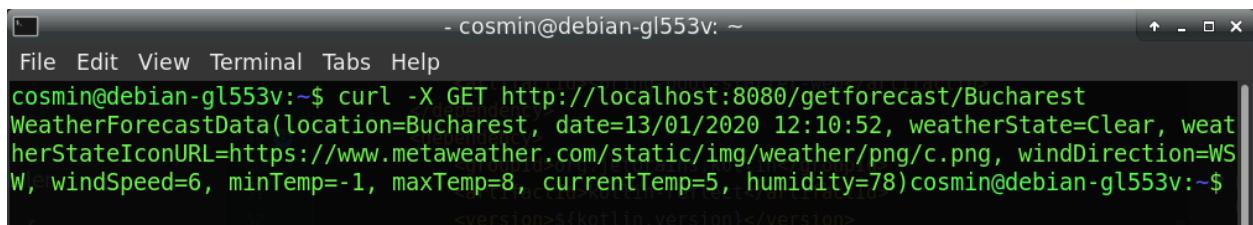
3.5. Testarea aplicației

Accesați următorul URL după compilarea și pornirea aplicației Spring Boot:

<http://localhost:8080/getforecast/Bucharest>

Ar trebui să primiți un răspuns ce conține datele meteo pentru locația **București**. Testați și pentru alte locații.

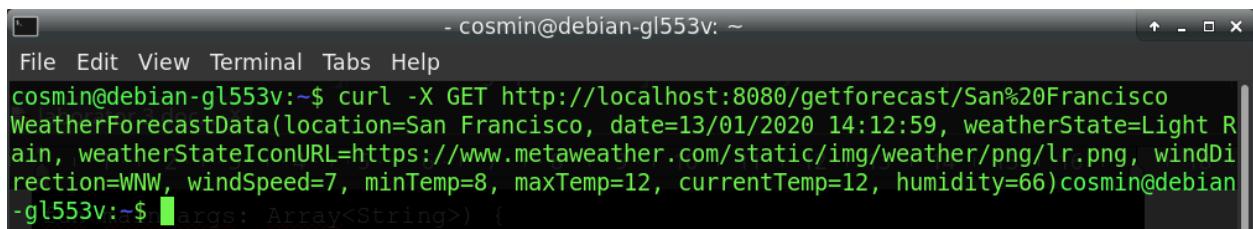
Atenție: nu sunt disponibile toate locațiile, mai ales cele din România.



```
- cosmin@debian-gl553v: ~
File Edit View Terminal Tabs Help
cosmin@debian-gl553v:~$ curl -X GET http://localhost:8080/getforecast/Bucharest
WeatherForecastData(location=Bucharest, date=13/01/2020 12:10:52, weatherState=Clear, weatherStateIconURL=https://www.metaweather.com/static/img/weather/png/c.png, windDirection=WSW, windSpeed=6, minTemp=-1, maxTemp=8, currentTemp=5, humidity=78)cosmin@debian-gl553v:~$ <version>$ (kotlin.version)</version>
```

Pentru a folosi spații în cuvintele cheie, adăugați codificarea corespunzătoare caracterului spațiu în URL: **%20**. Spre exemplu, pentru a se căuta „San Francisco”, URL-ul va fi de forma:

<http://localhost:8080/getforecast/San%20Francisco>



```
- cosmin@debian-gl553v: ~
File Edit View Terminal Tabs Help
cosmin@debian-gl553v:~$ curl -X GET http://localhost:8080/getforecast/San%20Francisco
WeatherForecastData(location=San Francisco, date=13/01/2020 14:12:59, weatherState=Light Rain, weatherStateIconURL=https://www.metaweather.com/static/img/weather/png/lr.png, windDirection=NNW, windSpeed=7, minTemp=8, maxTemp=12, currentTemp=12, humidity=66)cosmin@debian-gl553v:~$ args: Array<String> (
```

Aplicații și teme propuse

- **Temă de laborator**

Reproiectați diagrama de clase a aplicației meteo, cu respectarea principiilor de proiectare și modificați aplicația corespunzător.

- **Temă pe acasă**

1. Reproiectați și reimplementați aplicația meteo astfel încât să utilizați înlănțuirea (engl. *chaining*) de servicii.
2. Reproiectați și reimplementați aplicația meteo astfel încât să utilizați orchestrarea (engl. *orchestration*) de servicii.

Bibliografie

- [1] Apache Maven - <https://maven.apache.org/>
- [2] Spring Boot - <https://spring.io/projects/spring-boot>
- [3] Project Object Model (POM) - <https://maven.apache.org/guides/introduction/introduction-to-the-pom.html>
- [4] Gradle Build Tool - <https://gradle.org/>
- [5] Documentația Spring – <https://docs.spring.io>
- [6] Documentația de referință Spring Boot – <https://docs.spring.io/spring-boot/docs/current/reference/html/>
- [7] Clase de tip Data Class în Kotlin - <https://kotlinlang.org/docs/reference/data-classes.html>
- [8] Metaweather API - <https://www.metaweather.com/api/>

Sisteme Distribuite - Laborator 4

Servicii RESTful

Servicii RESTful - descriere generală

Serviciile web RESTful sunt servicii slab cuplate, potrivite pentru crearea de API-uri destinate clienților din Internet. Acronimul **REST** provine de la „REpresentational State Transfer”, un stil arhitectural de creare a aplicațiilor client-server orientate pe transferul reprezentărilor de resurse prin cereri și răspunsuri.

Aceste servicii se bazează pe comunicațiile prin protocolul fără stare **HTTP (Hypertext Transfer Protocol - RFC 7231)**. Deoarece stilul arhitectural REST este **orientat pe resursă**, atunci comunicațiile reprezintă practic operațiile ce se pot efectua asupra resurselor puse la dispoziție.

O resursă este reprezentată de **date** sau **funcționalitate**, depinzând de context. Un exemplu de resursă ar fi starea meteo pentru un anume oraș. Resursele sunt accesate folosind **URI-uri (Uniform Resource Identifiers)**, iar operațiile care se pot face pe o anumită resursă sunt bine definite de API-ul REST pus la dispoziție în funcție de aplicație.

Structura unui URI

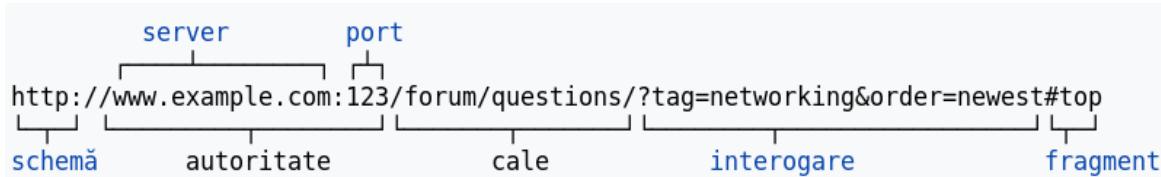


Figura 1 - Structura unui URI

Structura unei cereri HTTP

O cerere HTTP are forma:

```

<VERB_HTTP> <CALE_RESURSA> <VERSIUNE_HTTP>
<ANTET_1>: <VALOARE_ANTET_1>
<ANTET_2>: <VALOARE_ANTET_2>
...
<ANTET_N>: <VALOARE_ANTET_N>
<RÂND_GOL>
<CORP_CERERE>

```

Exemplu de cerere GET care preia resursa web identificată prin calea `/html/rfc5789` de pe server-ul `tools.ietf.org`:

```

GET /html/rfc5789 HTTP/1.1
Host: tools.ietf.org

```

Exemplu de cerere POST trimisă ca rezultat al completării unui formular web cu 2 câmpuri (**nume** și **prenume**) către server-ul `www.ac.tuiasi.ro`:

```

POST /procesare-student HTTP/1.1
Host: www.ac.tuiasi.ro
Content-Type: application/x-www-form-urlencoded
Content-Length: 27

```

```
nume=Luke&prenume=Skywalker
```

Structura unui răspuns HTTP

```
<VERSIUNE_HTTP> <COD_RĂSPUNS> <EXPLICAȚIE_COD_RĂSPUNS>
<ANTET_1>: <VALOARE_ANTET_1>
<ANTET_2>: <VALOARE_ANTET_2>
...
<ANTET_N>: <VALOARE_ANTET_N>
<RÂND_GOL>
<CORP_CERERE>
```

Exemplu de răspuns HTTP:

```
HTTP/1.1 200 OK
Date: Mon, 23 May 2005 22:38:34 GMT
Content-Type: text/html; charset=UTF-8
Content-Length: 138
Last-Modified: Wed, 08 Jan 2003 23:11:55 GMT
Server: Apache/1.3.3.7 (Unix) (Red-Hat/Linux)
ETag: "3f80f-1b6-3e1cb03b"
Accept-Ranges: bytes
Connection: close

<html>
  <head>
    <title>The resource you requested</title>
  </head>
  <body>
    <p>The content of the web resource</p>
  </body>
</html>
```

Coduri de răspuns HTTP

- informative: **1XX**
- de succes: **2XX**
- de redirecționare: **3XX**
- eroare la client: **4XX**
- eroare la server: **5XX**

Pentru detalii despre fiecare cod de eroare în parte, consultați secțiunea 6.1 din RFC 7231: <https://tools.ietf.org/html/rfc7231#section-6.1>

Maparea operațiilor REST peste metodele HTTP

Un API REST bine construit este alcătuit din operații bine definite asupra resurselor țintă, operații care se mapează corespunzător peste metodele HTTP disponibile conform protocolului. Se recomandă respectarea semnificației metodelor HTTP și a codurilor de răspuns transmise de server către client, în funcție de caz.

Operațiile de tip **CRUD** care se pot face asupra unei resurse sunt următoarele:

- (Create) crearea unei resurse - **POST** sau **PUT**

– POST se folosește pentru crearea unei resurse al cărui identificator unic (ID) nu este cunoscut de client și este generat de server
– PUT se folosește atunci când clientul decide cum este identificată resursa respectivă

- (Retrieve) preluarea unei resurse - **GET**
- (Update) actualizarea unei resurse - **PUT** sau **PATCH** ([RFC 5789](#))
 - PUT înlocuiește complet resursa cu reprezentarea acesteia din corpul cererii
 - cererile de tip PATCH conțin doar diferențele dintre reprezentările resurselor referențiate, și deci server-ul va modifica resursa existentă aplicând *patch*-ul trimis de client
- (Delete) ștergerea unei resurse - **DELETE**

Aplicație demonstrativă - agendă telefonică

Pentru a ilustra conceptele REST prezентate anterior, veți crea o aplicație simplă care va gestiona o agendă telefonică ce conține următoarele date:

- identificator unic
- nume
- prenume
- număr de telefon

Schema RESTful

Se începe cu proiectarea schemei RESTful, care expune resursele puse la dispoziție, operațiile care pot fi efectuate asupra lor, precum și tipurile de răspuns în funcție de cererile făcute de client pe resursele respective.

CALE	VERB HTTP	COD RĂSPUNS	SEMNIFICATIE
<code>/person/{id}</code>	GET	200 OK	Clientul a cerut datele unei persoane din agendă și a primit rezultatul
		404 NOT FOUND	Clientul a cerut datele unei persoane care nu există în agendă
	PUT	202 ACCEPTED	Clientul a cerut actualizarea persoanei cu ID-ul id , iar server-ul a actualizat intrarea în agendă
		404 NOT FOUND	Clientul a cerut actualizarea unei persoane care nu există în agendă
	DELETE	200 OK	Clientul a cerut ștergerea persoanei cu ID-ul id din agendă, iar ștergerea s-a efectuat cu succes
		404 NOT FOUND	Clientul a cerut ștergerea unei persoane care nu există în agendă
<code>/person</code>	POST	201 CREATED	Clientul a cerut crearea unei intrări în agendă pe server, ca și subordonat al resursei person . Adăugarea persoanei în agendă a fost făcută cu succes.
		400 BAD REQUEST	Clientul a cerut crearea unei intrări în agendă pe server, dar datele pe care le-a trimis în corpul cererii nu sunt corecte

<p>/agenda</p> <p>cu parametrii URL:</p> <ul style="list-style-type: none"> • firstName • lastName • telephone 	GET	200 OK	Clientul a cerut o listă de persoane din agendă (conform cu eventualele filtre din URL) și a primit o listă nevidă, validă
		204 NO CONTENT	Clientul a cerut o listă de persoane din agendă, dar nu există niciun element în listă care corespunde filtrelor cerute

Diagrama serviciului AgendaService



Implementarea claselor

Pentru implementare, creați o aplicație **Spring Boot**, conform modelului explicitat în laboratorul 3. Utilitarul de gestiune a proiectului este la alegere (Maven / Gradle).

Creați pachetele corespunzătoare diagramei de clase prezentată anterior. Structura proiectului este următoarea:

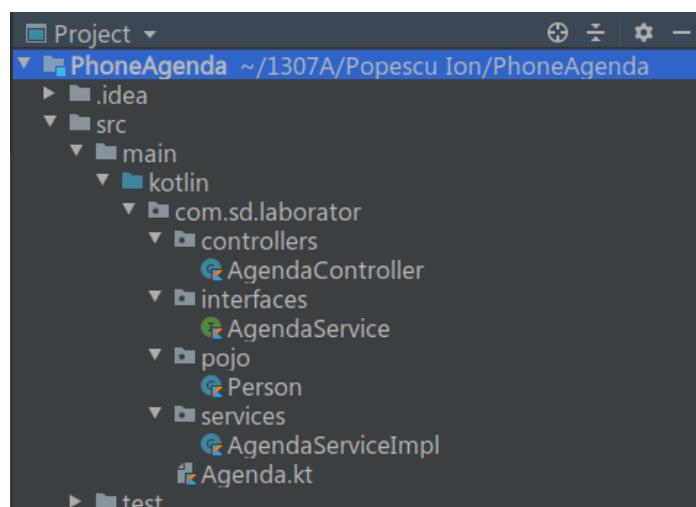


Figura 2 - Structura proiectului

- clasa **Person**

```

package com.sd.laborator.pojo

data class Person(
    var id: Int = 0,
    var lastName: String = "",
    var firstName: String = "",
  
```

```

    var telephoneNumber: String = ""
)

```

Aceasta încapsulează datele care „circulă” prin componentele aplicației, și este serializată în momentul în care datele trebuie trimise clientului, respectiv deserializată când clientul trimit date în cererile către server.

De aceea, pentru implementare s-a folosit o clasă de tip **data class** din Kotlin.

- interfața **AgendaService**

```

package com.sd.laborator.interfaces

import com.sd.laborator.pojo.Person

interface AgendaService {
    fun getPerson(id: Int) : Person?
    fun createPerson(person: Person)
    fun deletePerson(id: Int)
    fun updatePerson(id: Int, person: Person)
    fun searchAgenda(lastNameFilter: String, firstNameFilter: String,
telephoneNumberFilter: String): List<Person>
}

```

Aceasta este interfața ce expune sub formă de serviciu un set de operațiuni care sunt folosite în continuare de *controller*. Implementările efective pentru metodele expuse se află în bean-ul **AgendaServiceImpl**, explicitat în cele ce urmează.

- serviciul **AgendaServiceImpl**

```

package com.sd.laborator.services;

import com.sd.laborator.interfaces.AgendaService
import com.sd.laborator.pojo.Person
import org.springframework.stereotype.Service
import java.util.concurrent.ConcurrentHashMap

@Service
class AgendaServiceImpl : AgendaService {
    companion object {
        val initialAgenda = arrayOf(
            Person(1, "Hello", "Kotlin", "1234"),
            Person(2, "Hello", "Spring", "5678"),
            Person(3, "Hello", "Microservice", "9101112")
        )
    }

    private val agenda = ConcurrentHashMap<Int, Person>(
        initialAgenda.associateBy { person: Person -> person.id }
    )

    override fun getPerson(id: Int): Person? {
        return agenda[id]
    }

    override fun createPerson(person: Person) {
        agenda[person.id] = person
    }
}

```

```

    }

    override fun deletePerson(id: Int) {
        agenda.remove(id)
    }

    override fun updatePerson(id: Int, person: Person) {
        deletePerson(id)
        createPerson(person)
    }

    override fun searchAgenda(lastNameFilter: String, firstNameFilter: String, telephoneNumberFilter: String): List<Person> {
        return agenda.filter {
            it.value.lastName.toLowerCase().contains(lastNameFilter, ignoreCase = true) &&
            it.value.firstName.toLowerCase().contains(firstNameFilter, ignoreCase = true) &&
            it.value.telephoneNumber.contains(telephoneNumberFilter)
        }.map {
            it.value
        }.toList()
    }
}

```

Serviciul **AgendaServiceImpl** este expus sub formă de *bean* în contextul de execuție Spring (de aici adnotarea **@Service**).

Pentru simplitate, nu s-a folosit o bază de date sau o altă modalitate de persistență a datelor, ci agenda telefonică este păstrată în memorie și este validă atât timp cât aplicația Spring se află în execuție. Agenda este păstrată într-o structură de date de tip **ConcurrentHashMap** pentru a asigura proprietatea **thread-safe**.

Înțial, agenda telefonică este populată cu câteva date de test menținute într-un obiect companion Kotlin (**companion object**). De ce? Pentru că acele date inițiale sunt aceleași pentru orice instanță s-ar crea în memorie, și deci vectorul **initialAgenda** trebuie să aparțină clasei, și nu instanței clasei respective.

Implementările metodelor nu sunt complicate, ci de la sine înțelese, cu excepția (poate) a metodei **searchAgenda**: aceasta filtrează elementele din colecția **agenda**, pe baza unui predicat logic trimis funcției **filter**. Rezultatul funcției **filter** este tot o colecție de același tip, deci un **ConcurrentHashMap**, dar care conține doar elementele ce satisfac predicatul respectiv. Colecția rezultată este transformată folosind funcția **map**, ce preia fiecare pereche **<Int, Person>** și o transformă într-un element de tip **Person** (practic, se ignoră identificatorul). De ce se face acest lucru? Pentru că se dorește a se trimite ca răspuns o listă cât mai simplă, formată doar din obiecte de tip **Person**. Lista este construită cu metoda **toList()** aplicată pe **HashMap**-ul rezultat.

- controller-ul **AgendaController**

```

package com.sd.laborator.controllers

import com.sd.laborator.interfaces.AgendaService
import com.sd.laborator.pojo.Person

```

```

import org.springframework.beans.factory.annotation.Autowired
import org.springframework.http.HttpStatus
import org.springframework.http.ResponseEntity
import org.springframework.web.bind.annotation.*

@RestController
class AgendaController {
    @Autowired
    private lateinit var agendaService: AgendaService

    @RequestMapping(value = ["/person"], method =
    [RequestMethod.POST])
    fun createPerson(@RequestBody person: Person): ResponseEntity<Unit> {
        agendaService.createPerson(person)
        return ResponseEntity(Unit, HttpStatus.CREATED)
    }

    @RequestMapping(value = ["/person/{id}"], method =
    [RequestMethod.GET])
    fun getPerson(@PathVariable id: Int): ResponseEntity<Person?> {
        val person: Person? = agendaService.getPerson(id)
        val status = if (person == null) {
            HttpStatus.NOT_FOUND
        } else {
            HttpStatus.OK
        }
        return ResponseEntity(person, status)
    }

    @RequestMapping(value = ["/person/{id}"], method =
    [RequestMethod.PUT])
    fun updatePerson(@PathVariable id: Int, @RequestBody person: Person): ResponseEntity<Unit> {
        agendaService.getPerson(id)?.let {
            agendaService.updatePerson(it.id, person)
            return ResponseEntity(Unit, HttpStatus.ACCEPTED)
        } ?: return ResponseEntity(Unit, HttpStatus.NOT_FOUND)
    }

    @RequestMapping(value = ["/person/{id}"], method =
    [RequestMethod.DELETE])
    fun deletePerson(@PathVariable id: Int): ResponseEntity<Unit> {
        if (agendaService.getPerson(id) != null) {
            agendaService.deletePerson(id)
            return ResponseEntity(Unit, HttpStatus.OK)
        } else {
            return ResponseEntity(Unit, HttpStatus.NOT_FOUND)
        }
    }

    @RequestMapping(value = ["/agenda"], method = [RequestMethod.GET])
    fun search(@RequestParam(required = false, name = "lastName",
    defaultValue = "") lastName: String,
               @RequestParam(required = false, name =
    "firstName", defaultValue = "") firstName: String,

```

```
        @RequestParam(required = false, name =
"telephone", defaultValue = "") telephoneNumber: String):
    ResponseEntity<List<Person>> {
    val personList = agendaService.searchAgenda(lastName,
firstName, telephoneNumber)
    var httpStatus = HttpStatus.OK
    if (personList.isEmpty()) {
        httpStatus = HttpStatus.NO_CONTENT
    }
    return ResponseEntity(personList, httpStatus)
}
```

Clasa *controller* este anotată cu **@RestController** pentru ca Spring să o configureze corespunzător și să simplifice munca dezvoltatorului: serializările / deserializările se fac automat în și din obiecte JSON (transparent pentru dezvoltator), iar răspunsurile trimise în metodele mapate pe căile de acces se consideră în mod automat ca fiind corpul răspunsurilor HTTP la cererile clientului (din nou, fără intervenția dezvoltatorului).

Controller-ul implementează câte o metodă de tratare a fiecărei operații posibile din schema REST prezentată anterior.

Serviciul **AgendaService** este injectat automat ca dependență în faza de scanare de componente a Spring (de aici adnotarea **@Autowired** - va fi ales *bean*-ul ce conține o implementare a interfeței **AgendaService**, adică **AgendaServiceImpl**, în acest caz).

Fiecare metodă tratează cazurile de excepție ce pot apărea din vina clientului. Spre exemplu, considerând operația de actualizare a unei persoane în agenda de telefon:

```
    @RequestMapping(value = ["/person/{id}"], method =
[RequestMethod.PUT])
    fun updatePerson(@PathVariable id: Int, @RequestBody person: Person): ResponseEntity<Unit> {
        agendaService.getPerson(id)?.let {
            agendaService.updatePerson(it.id, person)
            return ResponseEntity(Unit, HttpStatus.ACCEPTED)
        } ?: return ResponseEntity(Unit, HttpStatus.NOT_FOUND)
    }
}
```

Variabila de cale `id` este preluată automat din URL folosind adnotarea `@PathVariable`, iar corpul cererii clientului (obiectul în sine care urmează a fi actualizat) este preluat folosind adnotarea `@RequestBody`. Așadar, în interiorul metodei `updatePerson()`, dezvoltatorul are acces la corpul cererii clientului, respectiv la parametrii trimiși în URL.

Dacă utilizatorul cere să actualizeze o persoană cu un ID inexistent, metoda trimite ca răspuns un cod de eroare de tip **404 NOT FOUND**, semantic echivalent cu situația apărută. Pentru a trimite un răspuns personalizat înapoi la client, pe baza logicii implementate de dezvoltator, se încapsulează corpul răspunsului într-un obiect de tip **ResponseEntity**. Această clasă șablon (*template*-izată) acceptă ca membri conținutul răspunsului efectiv și codul de răspuns HTTP dorit de dezvoltator.

Dacă se dorește ca metoda să returneze un obiect nul, sau un obiect gol, se folosește clasa **ResponseEntity** în conjuncție cu clasa **Unit** din Kotlin, ce ține locul tipului de date **void**. În acest caz, pentru metoda **updatePerson()**, nu se dorește a fi trimis niciun corp de răspuns, ci doar codul în sine (**202 ACCEPTED**, sau **404 NOT FOUND**), și deci răspunsul va fi de tip **ResponseEntity<Unit>** și clientul primește un răspuns fără conținut.

Considerând operația de preluare a unei persoane din agendă:

```

@RequestMapping(value = ["/person/{id}"], method =
[RequestMethod.GET])
fun getPerson(@PathVariable id: Int): ResponseEntity<Person?> {
    val person: Person? = agendaService.getPerson(id)
    val status = if (person == null) {
        HttpStatus.NOT_FOUND
    } else {
        HttpStatus.OK
    }
    return ResponseEntity(person, status)
}

```

Asemănător cu cazul anterior, se dorește ca atunci când poate apărea o problemă din vina clientului, aceasta să fie tratată corespunzător: dacă se cer datele unei persoane cu un ID inexistent, server-ul trebuie să notifice clientul de acest lucru, printr-un răspuns de tip **404 NOT FOUND**.

Dacă persoana respectivă există, atunci server-ul o va prelua din colecția internă și o va servi clientului serializând obiectul ce o încapsulează în corpul răspunsului HTTP (împreună cu un cod **200 OK**).

- punctul de intrare al aplicației Spring - **Agenda.kt**

```

package com.sd.laborator

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

@SpringBootApplication
class PhoneAgenda

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

```

Testarea aplicației cu Postman

Având de a face cu un API de tip REST, testarea aplicației implementate va presupune comunicații HTTP, cereri client de diverse tipuri în funcție de ceea ce se dorește din partea server-ului. De asemenea, corpul cererilor trebuie să conțină date în format JSON, deoarece acesta este modul implicit de lucru cu date în cazul serviciilor REST.

O variantă simplă de a lucra cu cereri HTTP personalizate și de a vizualiza răspunsurile într-un mod facil este folosirea aplicației **Postman**. Descărcați-l de aici:

<https://www.getpostman.com/product/api-client>

Pe Linux nu necesită instalare, ci doar dezarhivare și apoi execuția este simplă, deschizând un terminal în folder-ul **Postman** dezarhivat și folosind comanda:

```
./Postman
```

La pornire, veți fi întrebați dacă doriti să vă creați cont, să vă înregistrați etc. Ignorați toate aceste cereri până ajungeți la interfața principală.

Sisteme Distribuite - Laborator 4

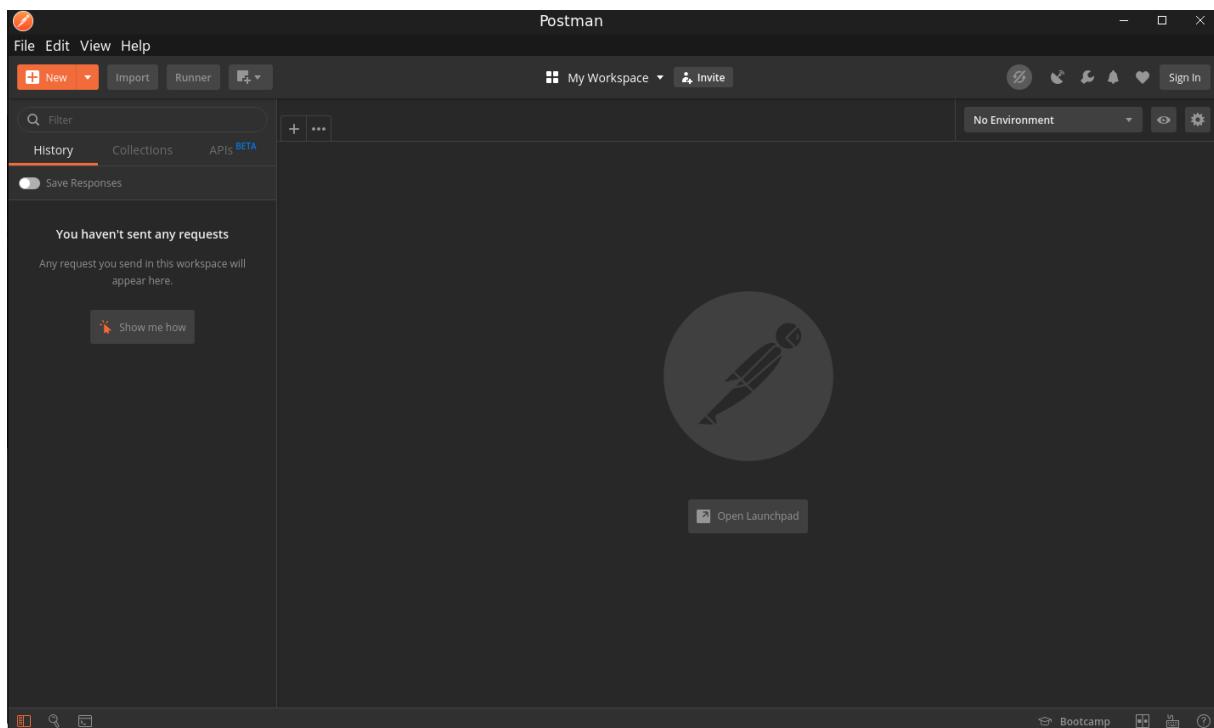


Figura 3 - Interfața Postman

Apăsați pe butonul cu un semn „plus” pentru a crea o cerere HTTP nouă.

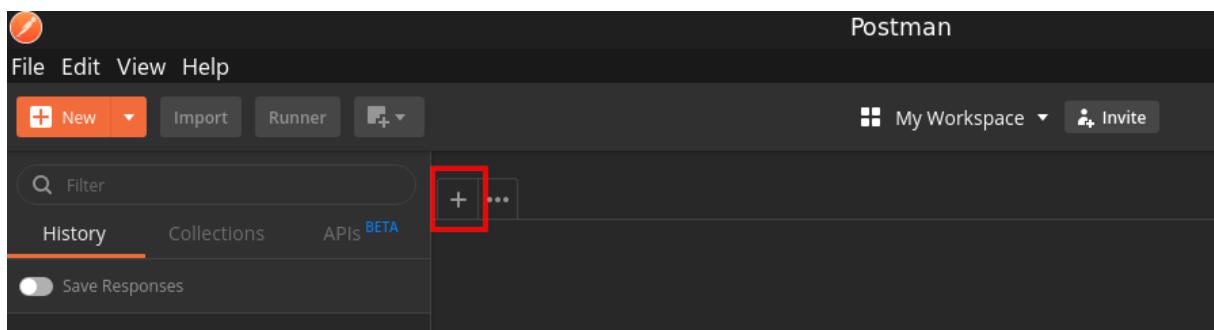


Figura 4 - Introducere cerere nouă în Postman

În secțiunea „**Params**” se pot adăuga parametri de filtrare în URL. Aceștia au rolul, în cazul API-urilor REST, de filtrare a răspunsurilor trimise clientului în funcție de preferințe.

În secțiunea „**Body**” se completează corpul mesajului HTTP trimis către server.

Pentru început, preluati întreaga listă de persoane din agenda telefonică (nu uitați să compilați proiectul și să porniți aplicația Spring Boot din IntelliJ).

Conform schemei REST a aplicației, pentru a prelua o listă de persoane, trebuie să accesați resursa disponibilă la calea **/agenda**, deci, în câmpul „**Enter request URL**”, introduceți URL-ul: <http://localhost:8080/agenda>. În partea stângă a acestui câmp selectați **GET** ca și tip de cerere. Nu introduceți niciun parametru pentru moment, și apăsați butonul „**Send**” din partea dreaptă.

The screenshot shows a POSTMAN interface with the following details:

- Tip cerere:** GET
- URL:** http://localhost:8080/agenda
- Status:** 200 OK
- Body (Pretty):**

```

1 [
2   {
3     "id": 1,
4     "lastName": "Hello",
5     "firstName": "Kotlin",
6     "telephoneNumber": "1234"
7   },
8   {
9     "id": 2,
10    "lastName": "Hello",
11    "firstName": "Spring",
12    "telephoneNumber": "5678"
13  },
14  {
15    "id": 3,
16    "lastName": "Hello",
17    "firstName": "Microservice",
18    "telephoneNumber": "9101112"
19  }
20 ]

```

Figura 5 - Exemplu de cerere HTTP de tip GET

Se observă că server-ul a răspuns prin trimiterea întregii agende telefonice sub formă de obiect JSON (deoarece controller-ul REST utilizează acest format pentru obiectele trimise în comunicațiile client-server).

Încercați acum să aplicați un filtru, spre exemplu să căutați un număr de telefon pe baza unui nume. Adăugați un parametru URL care să filtreze rezultatele pe baza prenumelui „**Kotlin**”:

- key: **firstName**
- value: **Kotlin**

The screenshot shows a POSTMAN interface with the following details:

- Tip cerere:** GET
- URL:** http://localhost:8080/agenda?firstName=Kotlin
- Status:** 200 OK
- Body (Pretty):**

```

1 [
2   {
3     "id": 1,
4     "lastName": "Hello",
5     "firstName": "Kotlin",
6     "telephoneNumber": "1234"
7   }
8 ]

```

Figura 6 - Exemplu de cerere GET cu filtre

În acest caz, a fost returnat un singur rezultat ce corespunde filtrului.

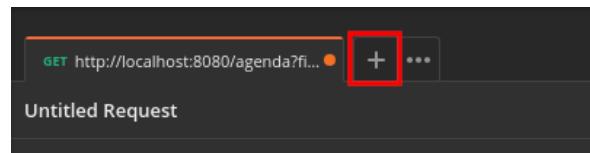
Acum, încercați să aplicați un filtru care nu generează niciun rezultat, spre exemplu: **firstName: 123**.

The screenshot shows the Postman interface with an 'Untitled Request' tab. The method is set to 'GET' and the URL is 'http://localhost:8080/agenda?firstName=123'. In the 'Params' section, there is a table with one row: 'firstName' (Key) and '123' (Value). Below the table, the status bar shows 'Status: 204 No Content'. The 'Body' tab is selected.

Figura 7 - Răspuns vid la cerere de tip GET

Se observă cum server-ul a răspuns cu **204 NO CONTENT**, deoarece nu există niciun element în agenda telefonică ce să corespundă cu filtrul ales.

Acum, încercați să adăugați o nouă intrare în agendă. Conform schemei REST a aplicației, trebuie să se trimită o cerere HTTP de tip **POST** către calea **/person**, iar în corpul cererii se încapsulează obiectul JSON cu datele despre persoană. Apăsați pe butonul „plus” pentru a deschide alt tab cu o cerere nouă.



Selectați tipul de cerere „POST” și completați URL-ul: <http://localhost:8080/person>. Apoi, selectați tab-ul „Body”, bifați „raw” și selectați ca tip de conținut „JSON”. Completați corpul cererii astfel:

```
{
  "id": 10,
  "firstName": "Luke",
  "lastName": "Skywalker",
  "telephoneNumber": "123456789"
}
```

Apăsați pe butonul „Send” și observați cum server-ul răspunde cu **201 CREATED**, deci resursa dorită a fost creată cu succes (vedeți figura de mai jos).

Dacă veți schimba acum la tab-ul anterior cu cererea GET și veți șterge filtrele adăugate (sau dezactiva, folosind bifa din partea stângă), după trimiterea cererii respective, obțineți lista actualizată cu agenda telefonică, ce conține și noul obiect ce tocmai a fost adăugat.

The screenshot shows the Postman interface with an 'Untitled Request' tab. A POST request is selected with the URL `http://localhost:8080/person`. The 'Body' tab is active, showing a JSON payload:

```

1: {
2:   "id": 10,
3:   "firstName": "Luke",
4:   "lastName": "Skywalker",
5:   "telephoneNumber": "123456789"
6: }

```

The 'Send' button is highlighted with a red box. Below the request, the response section shows the status: `201 Created`, time: `34ms`, size: `137 B`.

Figura 8 - Exemplu de cerere POST

Pentru a exemplifica și tratarea cazurilor de excepție, încercați să ștergeți o resursă de pe server. Din nou, conform schemei REST, pentru a șterge o intrare din agendă, trebuie trimisă o cerere HTTP de tip **DELETE** către calea `/person/{id}`, cu variabila de cale `id` având una din valorile existente în colecția de obiecte de tip **Persoana**.

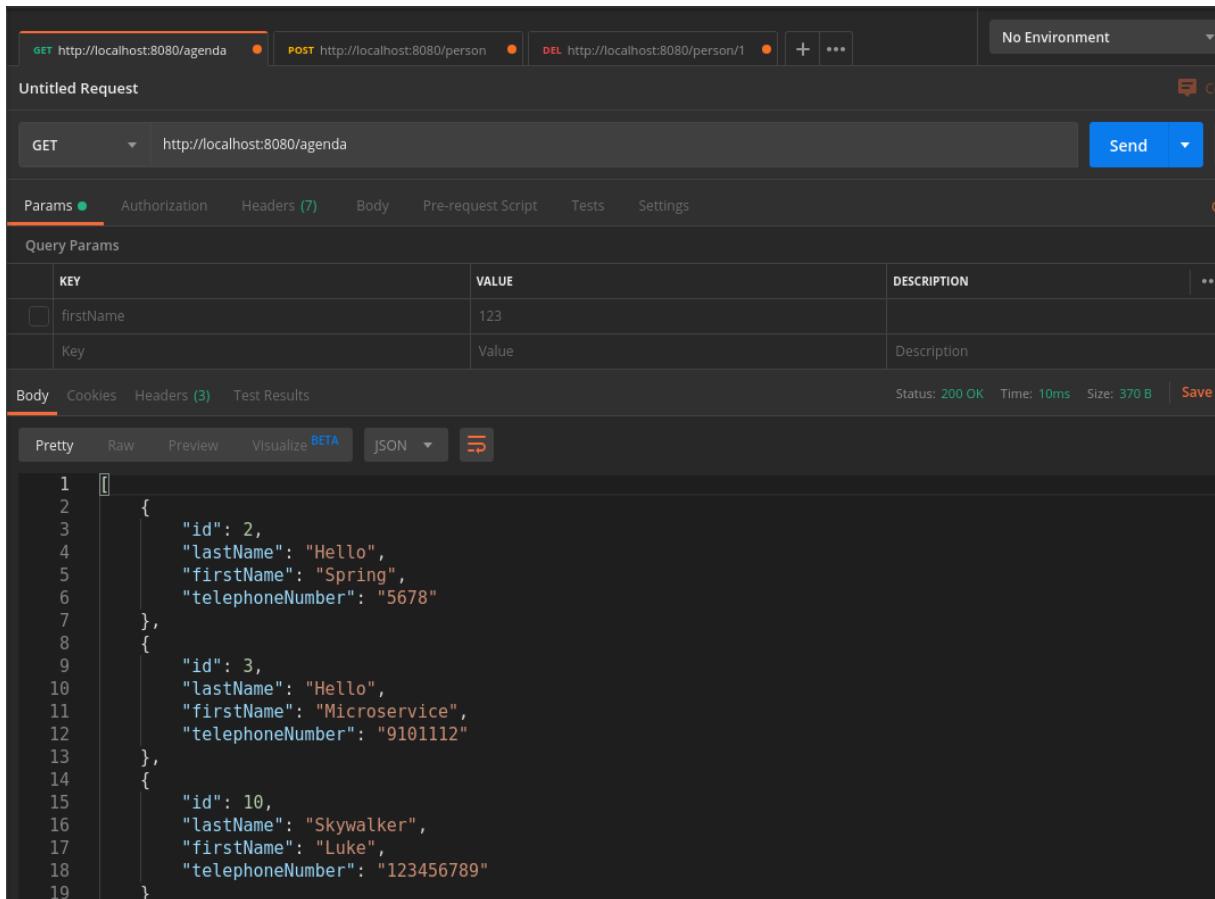
De exemplu, ștergeți persoana cu ID-ul 1: apăsați din nou pe butonul „plus”, selectați **DELETE** ca și tip de cerere, completați URL-ul cu <http://localhost:8080/person/1>, lăsați celelalte câmpuri necomplete și apăsați „Send”.

The screenshot shows the Postman interface with an 'Untitled Request' tab. A DELETE request is selected with the URL `http://localhost:8080/person/1`. The 'Params' tab is active. The response section shows the status: `200 OK`, time: `93ms`, size: `132 B`.

Figura 9 - Exemplu de cerere de tip DELETE

Server-ul a șters resursa cu succes, conform răspunsului primit, de tip 200 OK. Confirmați acest lucru preluând din nou întreaga listă de persoane din agendă, cu cererea HTTP configurată în primul tab:

Sisteme Distribuite - Laborator 4



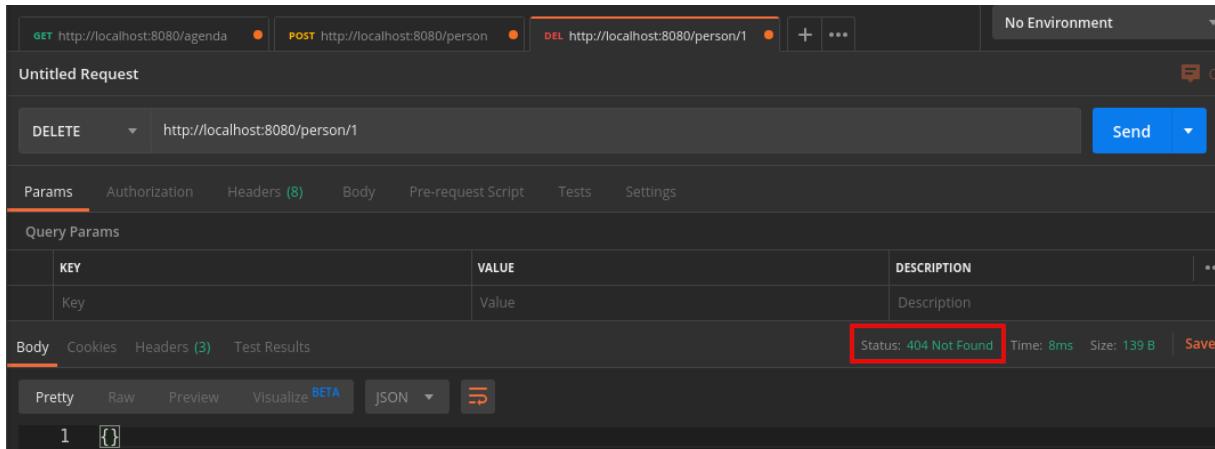
The screenshot shows the Postman interface with an 'Untitled Request' tab. At the top, there are three buttons: 'GET http://localhost:8080/agenda' (highlighted in orange), 'POST http://localhost:8080/person' (yellow), and 'DEL http://localhost:8080/person/1' (red). To the right is a 'No Environment' dropdown. Below the buttons, the URL 'http://localhost:8080/agenda' is entered in the address bar, along with a 'Send' button. Under the 'Params' tab, there is a 'Query Params' table with two rows: 'firstName' with value '123' and 'Key' with value 'Value'. The 'Body' tab is selected, showing a JSON response:

```
1 [ ]  
2 {  
3     "id": 2,  
4     "lastName": "Hello",  
5     "firstName": "Spring",  
6     "telephoneNumber": "5678"  
7 },  
8 {  
9     "id": 3,  
10    "lastName": "Hello",  
11    "firstName": "Microservice",  
12    "telephoneNumber": "9101112"  
13 },  
14 {  
15     "id": 10,  
16     "lastName": "Skywalker",  
17     "firstName": "Luke",  
18     "telephoneNumber": "123456789"  
19 }
```

At the bottom right, status information is displayed: Status: 200 OK, Time: 10ms, Size: 370 B.

Figura 10 - Lista de persoane după ștergere

Dacă încercați să ștergeți din nou persoana cu ID-ul 1, veți primi o eroare de la server, deoarece aceasta nu mai există (conform cu acele cazuri de excepție tratate în codul de business):



The screenshot shows the Postman interface with an 'Untitled Request' tab. At the top, there are three buttons: 'GET http://localhost:8080/agenda' (orange), 'POST http://localhost:8080/person' (yellow), and 'DEL http://localhost:8080/person/1' (red). Below the buttons, the URL 'http://localhost:8080/person/1' is entered in the address bar, along with a 'Send' button. Under the 'Params' tab, there is a 'Query Params' table with one row: 'Key' with value 'Value'. The 'Body' tab is selected, showing a JSON response:

```
1 [ ]
```

At the bottom right, status information is displayed: Status: 404 Not Found, Time: 8ms, Size: 139 B.

Figura 11 - Eroare la operația de DELETE

Celelalte teste pentru operațiile și cazurile de excepție rămase sunt lăsate ca exercițiu în timpul laboratorului.

Aplicații și teme

Teme de laborator

1. Extrageți diagrama de clase din codul exemplu al aplicației **Agendă**, disponibil în

laborator.

2. Adăugați o interfață grafică simplă bazată pe un formular HTML, cu care să puteți implementa operațiile **GET** și **POST**. Modificați apoi HTML-ul astfel încât să se poată trimite către program interogări HTTP de tip **PUT** și **DELETE** (cu iubitul vostru JavaScript - folosiți **fetch** sau **AJAX**).

Temă pentru acasă

Proiectați și implementați o aplicație cu servicii RESTful pentru **gestiunea cheltuielilor curente pentru membrii a unei familii** (nivel de complexitate asemănător cu cel din laborator). Pentru aplicația respectivă, creați și o interfață folosind **framework-ul Flask** din Python.

Documentație utilă:

- documentația Flask: <https://flask.palletsprojects.com/en/1.1.x/>
 - cum se instalează Flask:
<https://flask.palletsprojects.com/en/1.1.x/installation/#install-flask>
 - **Building REST APIs with Flask** - Kunal Relan
 - **Flask Web Development** - Miguel Grinberg
-

Bibliografie

- [1]: Creating a RESTful Web Service with Spring Boot -
<https://kotlinlang.org/docs/tutorials/spring-boot-restful.html>
- [2]: What Are RESTful Web Services? - <https://javaee.github.io/tutorial/jaxrs001.html>
- [3]: Hypertext Transfer Protocol - <https://tools.ietf.org/html/rfc7231>
- [4]: PATCH Method for HTTP - <https://tools.ietf.org/html/rfc5789>
- [5]: HTTP request methods - <https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods>
- [6]: Obiecte companion în Kotlin - <https://kotlinlang.org/docs/reference/object-declarations.html#companion-objects>

Laborator 5 - Servicii cu Spring Boot și Kotlin

Introducere

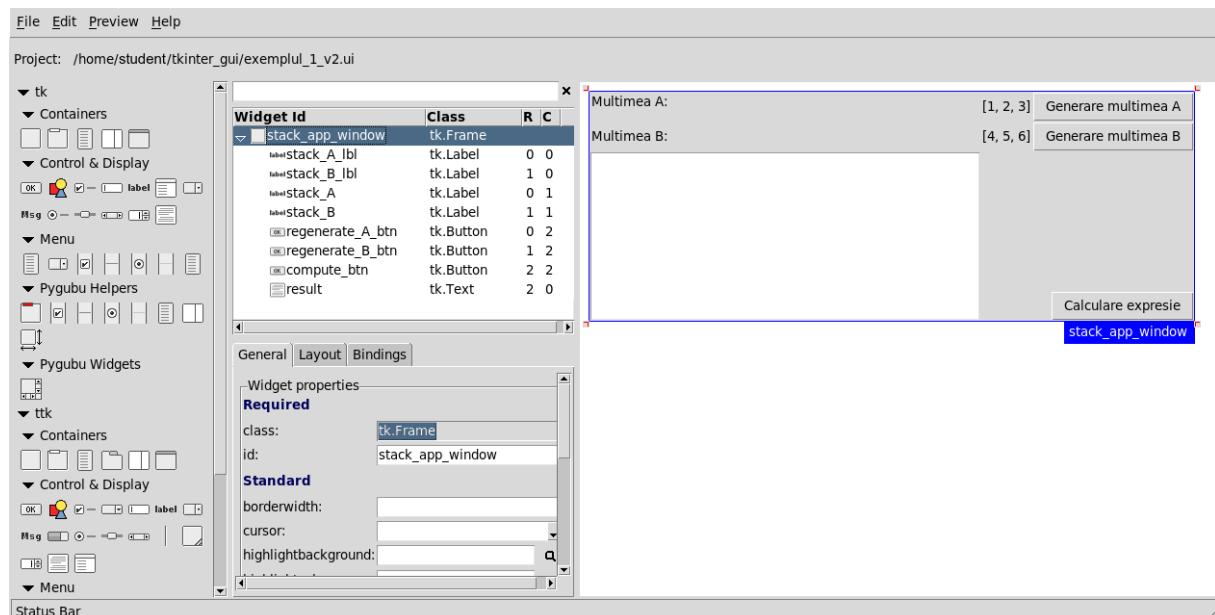
GUI cu Tkinter

Pentru implementarea unei interfețe grafice utilizând Tkinter, vezi cursul 5 de la disciplina Paradigme de Programare. De asemenea, vezi:

- <https://tkdocs.com/tutorial/index.html>
- <https://docs.python.org/3/library/tk.html>
- <https://pythonbasics.org/tkinter/>.

Pentru o soluție de tip Drag-and-Drop (Tkinter), se va deschide un terminal și se vor executa următoarele comenzi:

```
sudo apt install python3-pip # pip package manager for python packages
sudo pip3 install pygubu # GUI designer for Tkinter
pygubu-designer # run the designer
```



Designer-ul grafic PyGubu (Tkinter)

GUI cu PyQt5

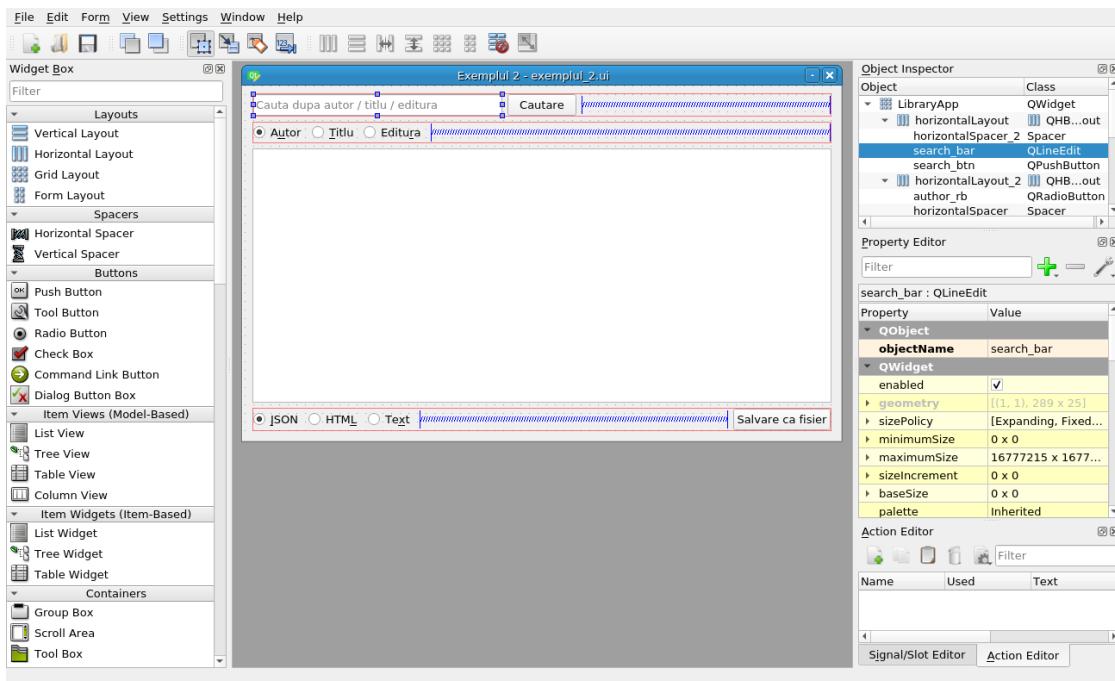
Pentru implementarea unei interfețe grafice utilizând PyQt5, vezi:

- <https://www.riverbankcomputing.com/static/Docs/PyQt5/>
- <https://pythonspot.com/pyqt5/>
- <https://likegeeks.com/pyqt5-tutorial/>

Pentru un designer grafic (PyQt5), se vor executa în terminal comenziile:

```
wget https://download.qt.io/official_releases/online_installers/qt-unified-linux-x64-online.run
chmod a+x qt-unified-linux-x64-online.run
./qt-unified-linux-x64-online.run
```

Laborator 5



Designer-ul grafic Qt5 Designer (PyQt5)

Crearea proiectului

Pentru crearea unui proiect **Spring Boot** folosind Maven / Gradle, vezi laboratorul 3 de la disciplina Sisteme Distribuite (Capitolul 1. Crearea unui proiect Spring Boot, subpunctele 1-10).

După ce s-a creat proiectul și s-au adăugat plugin-urile specificate în laboratorul 3, adăugați următoarea dependență suplimentară (element subordonat al tag-ului `<dependencies>`):

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-amqp</artifactId>
</dependency>
```

Clașele în Kotlin sunt, în mod implicit, marcate ca și **final**, deci nu se pot moșteni decât dacă dezvoltatorul le marchează explicit ca **open** (de exemplu, **open** class MyClass ...). **Spring necesită ca acele clase ce vor primi anumite tipuri de adnotări** (cum ar fi **@Component**) **să fie moștenibile**, adică marcate cu **open**. Acest lucru este făcut automat de plugin-ul **kotlin-maven-allopen**, aşadar îl veți adăuga ca dependență la compilare, astfel:

Adăugați următorul element de configurare în interiorul tag-ului `<plugin>`, corespunzător plugin-ului **kotlin-maven-plugin** (consultați figura de mai jos pentru locația exactă):

```
<dependencies>
    <dependency>
        <groupId>org.jetbrains.kotlin</groupId>
        <artifactId>kotlin-maven-allopen</artifactId>
        <version>${kotlin.version}</version>
    </dependency>
</dependencies>
```

```

49      <plugins>
50        <plugin>
51          <groupId>org.jetbrains.kotlin</groupId>
52          <artifactId>kotlin-maven-plugin</artifactId>
53          <version>${kotlin.version}</version>
54          <executions>
55            <execution>
56              <id>compile</id>
57              <phase>compile</phase>
58              <goals>
59                <goal>compile</goal>
60              </goals>
61              <configuration>
62                <compilerPlugins>
63                  <plugin>spring</plugin>
64                </compilerPlugins>
65              </configuration>
66            </execution>
67            <execution>
68              <id>test-compile</id>
69              <phase>test-compile</phase>
70              <goals>
71                <goal>test-compile</goal>
72              </goals>
73            </execution>
74          </executions>
75          <dependencies>
76            <dependency>
77              <groupId>org.jetbrains.kotlin</groupId>
78              <artifactId>kotlin-maven-allopen</artifactId>
79              <version>${kotlin.version}</version>
80            </dependency>
81          </dependencies>
82      </plugin>

```

Apoi, se adaugă plugin-ul **spring** ca și dependentă la faza de compilare, cu următorul copil al tag-ului **<execution>**, pus în locația indicată în figura anterioară.

```

<configuration>
  <compilerPlugins>
    <plugin>spring</plugin>
  </compilerPlugins>
</configuration>

```

Principii SOLID

S.O.L.I.D. este un acronim pentru cinci principii de proiectare orientate pe obiecte.

- **Single-Responsibility Principle (SRP)** - o clasă ar trebui să aibă un singur motiv de schimbare, ceea ce înseamnă că o clasă ar trebui să aibă o singură sarcină (job)
- **Open-Closed Principle (OCP)** - entitățile software (clase, module, funcții, etc) ar trebui să fie deschise pentru extindere, dar închise pentru modificări
- **Liskov Substitution Principle (LSP)** - Subtipurile (clasele derivate) trebuie să fie substituibile tipului de bază (clasei de bază)
- **Interface Segregation Principle (ISP)** - clientii nu ar trebui obligați să depindă de metode pe care nu le utilizează
- **Dependency Inversion Principle (DIP):**
 - modulele high-level nu ar trebui să depindă de modulele low-level. Ambele ar trebui să depindă de abstractizări;
 - abstractizările nu ar trebui să depindă de detaliu. Detaliile ar trebui să depindă de abstractizări

Pentru mai multe detalii, vezi cartea „**Agile software development: principles, patterns, and practices**” (2003) scrisă de **Robert C. Martin** (cunoscut ca „Uncle Bob”).

Cozi de mesaje

O coadă de mesaje este utilizată pentru comunicarea între procese, sau între firele de execuție (thread-urile) același proces. Acestea oferă un protocol de comunicare asincron în

Laborator 5

care emițătorul și receptorul nu au nevoie să interacționeze în același timp (mesajele sunt reținute în coadă până când destinatarul le citește)

Avantajele utilizării cozilor de mesaje:

- redundanță - procesele trebuie să confirme citirea mesajului și faptul că acesta poate fi eliminat din coadă
- vârfuri de trafic (traffic spikes) - adăugarea în coadă previne aceste spike-uri, asigurând stocarea datelor în coadă și procesarea lor (chiar dacă va dura mai mult)
- mesaje asincrone
- îmbunătățirea scalabilității
- garantarea faptului că tranzacția se execută o dată
- monitorizarea elementelor din coadă

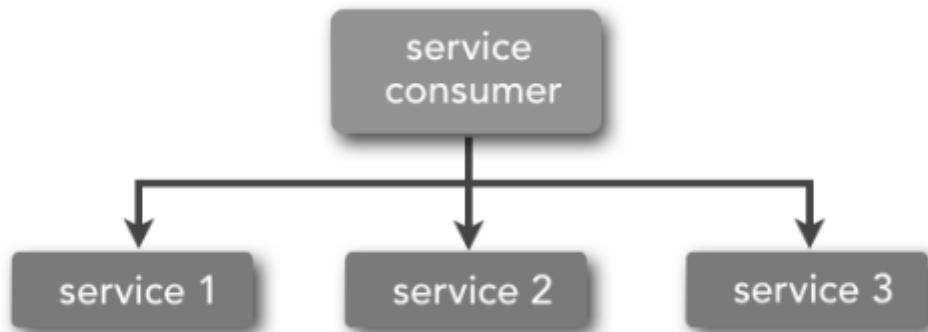
Servicii

Toate arhitecturile bazate pe servicii sunt în general arhitecturi distribuite, componentele fiind accesate *remote* printr-un anumit protocol (REST, SOAP, AMQP, JMS, RMI, etc).

Arhitecturile bazate pe servicii oferă îmbunătățiri față de aplicațiile monolitice, dar introduc de asemenea un nivel mai mare de complexitate (contractele serviciilor, disponibilitatea, securitatea, tranzacțiile).

Orchestrarea serviciilor (*Service orchestration*)

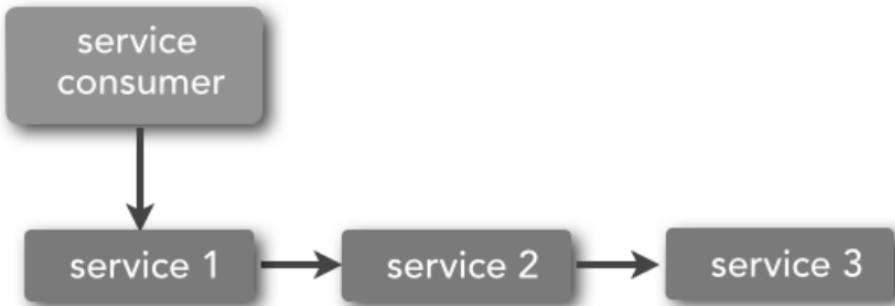
Orchestrarea serviciilor se referă la coordonarea mai multor servicii printr-un mediator centralizat, precum un consumator de servicii.



Pentru a înțelege mai bine, vă puteți gândi la o orchestră. Un număr de muzicieni cântă la diferite instrumente la tempi diferenți, dar sunt cu toții coordonați de o persoană centrală - dirijorul. Similar, consumatorul de servicii coordonează toate serviciile necesare pentru completarea tranzacției de afaceri (business transaction)

Coregrafie serviciilor

Coregrafie serviciilor se referă la coordonarea mai multor servicii fără un mediator central. Un serviciu apelează un alt serviciu care poate apela mai departe un alt serviciu și tot așa, rezultând **înlănțuirea serviciilor** (service chaining).



Coregrafia serviciilor

Pentru o ilustrare descriptivă, vă puteți gândi la o companie de dansuri care interpretează pe scenă. Fiecare dansator se mișcă sincronizat cu ceilalți dansatori, dar nimici nu le coordonează mișările.

Pentru mai multe detalii, vezi cursurile de sisteme distribuite și cartea „*Fundamentals of Software Architecture: A Comprehensive Guide to Patterns, Characteristics, and Best Practices*“ (2020), scrisă de Neal Ford și Mark Richards.

Configurări

Instalare server RabbitMQ:

```
sudo apt install -y rabbitmq-server
```

Verificare status RabbitMQ:

```
sudo systemctl status rabbitmq-server.service
```

Dacă serviciul nu este activ, se execută comanda:

```
sudo systemctl start rabbitmq-server.service
```

Activarea serviciului RabbitMQ la pornirea sistemului:

```
sudo systemctl enable rabbitmq-server
```

Activarea plugin-ului de gestionare:

```
sudo rabbitmq-plugins enable rabbitmq_management
```

Crearea și configurarea utilizatorului:

```

sudo rabbitmqctl add_user student student
sudo rabbitmqctl set_user_tags student administrator
sudo systemctl restart rabbitmq-server.service
sudo rabbitmqctl set_permissions -p / student ".*" ".*" ".*"
  
```

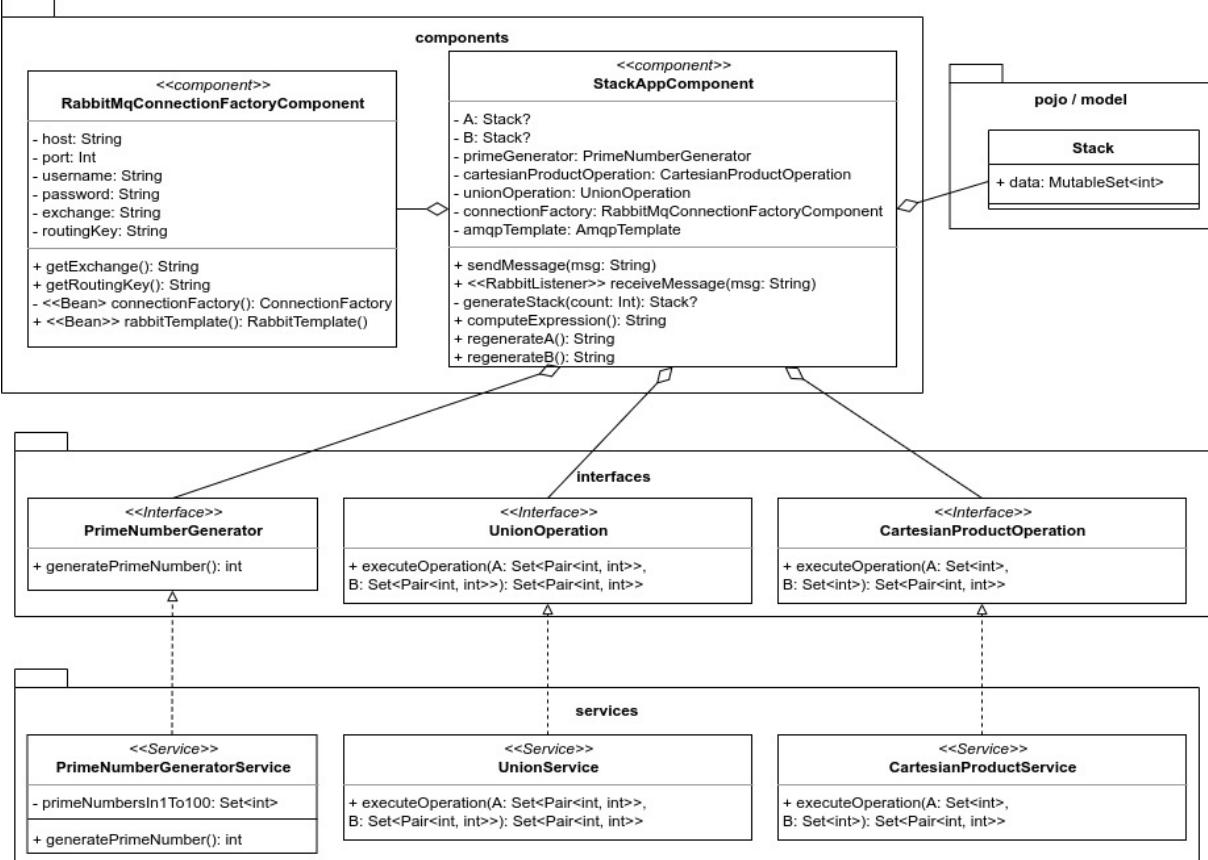
Pentru accesarea consolei de administrare, se deschide un browser web și se accesează <http://localhost:15672/>. Numele de utilizator: **student**, parola: **student**.

Exemple

Exemplul 1: StackApp

Laborator 5

Cerință: Pornind de la două mulțimi A și B care conțin 20 de elemente prime aleator depuse în două colecții separate și ținând cont de $A \times B = \{(a, b) | a \in A \wedge b \in B\}$, să se scrie un program Kotlin care va calcula prin intermediul unor servicii expresia $(A \times B) \cup (B \times B)$ utilizând funcții specifice colecțiilor și principiile SOLID. Rezultatul este depus într-un dictionar, iar acesta va fi afișat.



Arhitectura aplicației StackApp

Comunicarea prin cozi de mesaje

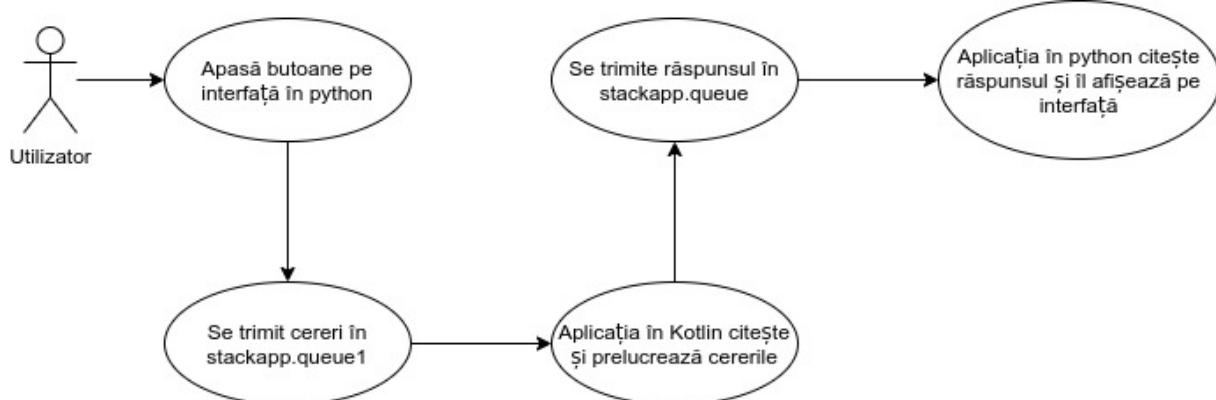


Diagrama de use-case pentru StackApp

În diagrama de mai sus, se observă existența a două cozi de mesaje (stackapp.queue și stackapp.queue1). Aceasta se datorează faptului că atât aplicația kotlin cât și aplicația python citesc/scriu într-o coadă de mesaje. Dacă ar fi fost utilizată doar o coadă, logica aplicației s-ar fi complicat (verificarea că mesajele au ajuns la „cine“ trebuie). Așadar, aplicația python (interfață) va scrie în stackapp.queue1 fiecare apăsare de buton (ex.: regenerare multimea A, calculare expresie). Aplicația kotlin va citi din stackapp.queue1, va efectua acele operații și va scrie rezultatul acestora în stackapp.queue, de unde va fi preluat spre a fi afișat pe interfață.

Model

Pachetul **model** (sau **pojo**) conține o singură clasă, **Stack** ce reprezintă un sir de elemente unice (o mulțime). Aceasta va stoca elementele mulțimilor A și B.

Observație: existența pachetului **model** și a clasei **Stack** nu este necesară în acest caz particular, având în vedere faptul că acea clasă conține o singură variabilă de tip **MutableSet**. Variabilele **A** și **B** din **StackAppComponent** puteau fi de tip **MutableSet<Int>**. Exemplul este pur academic.

Services

Au fost create trei servicii:

- *PrimeNumberGeneratorService* – acesta conține o variabilă cu toate numerele prime din intervalul [1, 100] și o funcție care alegeră aleator un număr din acest set.
- *UnionService* – realizează prin intermediul unei funcții specifice seturilor reuniunea dintre două mulțimi, cu precizarea că elementele mulțimii sunt de fapt tuple de două numere întregi (rezultatul produsului cartezian).
- *CartesianProductService* – realizează produsul cartezian dintre două mulțimi prin intermediul unor funcții lambda imbriicate (câte una pentru fiecare mulțime)

Components

Se remarcă faptul că **StackAppComponent** se folosește de abstractizări (interfețe), nu de implementările propriu-zise (**Dependency inversion principle**). Având în vedere simplitatea exemplului, soluția propusă respectă și celelalte principii SOLID, dar acest aspect este vizibil în acest caz particular doar pentru *Single responsibility principle* și *Open-closed principle*.

StackAppComponent folosește cele trei servicii „injectate” (a se vedea laboratorul 3 – dependency injection - `@Autowired`), expunând funcții de comunicare prin cozi de mesaje și funcții ce generează o mulțime de numere prime și calculează expresia din cerință.

RabbitMqConnectionFactoryComponent citește fișierul de configurări *application.properties*, încărcând valorile respective în proprietățile sale (atributele sale). Această componentă conține toate setările necesare conectării la coada de mesaje, expunând o metoda *rabbitTemplate()* ce returnează un obiect capabil de trimiterea de mesaje.

View

Interfața grafică a fost realizată atât cu PyQt5 cât și cu Tkinter, fiind ușor de folosit.

Laborator 5

Multimea A: [47, 71, 97, 41, 23, 79, 59, 31, 83, 7, 5, 61, 67, 3, 13, 19, 73, 2, 43, 17] Generare multimea A

Multimea B: [5, 79, 73, 71, 67, 19, 53, 41, 89, 2, 23, 61, 83, 29, 31, 7, 13, 59, 97, 3] Generare multimea B

A: [47, 71, 97, 41, 23, 79, 59, 31, 83, 7, 5, 61, 67, 3, 13, 19, 73, 2, 43, 17]
B: [5, 79, 73, 71, 67, 19, 53, 41, 89, 2, 23, 61, 83, 29, 31, 7, 13, 59, 97, 3]
result: [(47, 5), (47, 79), (47, 73), (47, 71), (47, 67), (47, 19), (47, 53), (47, 41), (47, 89), (47, 2), (47, 23), (47, 61), (47, 83), (47, 29), (47, 31), (47, 7), (47, 13), (47, 59), (47, 97), (47, 3), (71, 5), (71, 79), (71, 73), (71, 71), (71, 67), (71, 19), (71, 53), (71, 41), (71, 89), (71, 2), (71, 23), (71, 61), (71, 83), (71, 29), (71, 31), (71, 7), (71, 13), (71, 59), (71, 97), (71, 3), (97, 5), (97, 79), (97, 73), (97, 71), (97, 67), (97, 19), (97, 53), (97, 41), (97, 89), (97, 2), (97, 23), (97, 61), (97, 83), (97, 29), (97, 31), (97, 7), (97, 13), (97, 59), (97, 97), (97, 3), (41, 5), (41, 79), (41, 73), (41, 71), (41, 67), (41, 19), (41, 53), (41, 41), (41, 89), (41, 2), (41, 23), (41, 61), (41, 83), (41, 29), (41, 31), (41, 7), (41, 13), (41, 59), (41, 97), (41, 3), (23, 5), (23, 79), (23, 73), (23, 71), (23, 67), (23, 19), (23, 53), (23, 41), (23, 89), (23, 2), (23, 23), (23, 61), (23, 83), (23, 29), (23, 31), (23, 7), (23, 13), (23, 59), (23, 97), (23, 3), (79, 5), (79, 79), (79, 73), (79, 71), (79, 67), (79, 19), (79, 53), (79, 41), (79, 89), (79, 2), (79, 23), (79, 61), (79, 83), (79, 29), (79, 31), (79, 7), (79, 13), (79, 59), (79, 97), (79, 3), (59, 5), (59, 79), (59, 73), (59, 71), (59, 67), (59, 19), (59, 53), (59, 41), (59, 89), (59, 2), (59, 23), (59, 61), (59, 83), (59, 29), (59, 31), (59, 7), (59, 13), (59, 59), (59, 97), (59, 3), (31, 5), (31, 79), (31, 73), (31, 71), (31, 67), (31, 19), (31, 53), (31, 41), (31, 89), (31, 2)]

Calculare expresie

Interfață grafică realizată cu Qt Creator și PyQt5

Pentru a porni interfața, se compilează întâi proiectul în kotlin: se execută din fereastra Maven --> Plugins --> spring-boot --> spring-boot:run. Apoi, se deschide un terminal în folder-ul interfeței (*qt_gui* sau *tkinter_gui*) și se execută comanda:

```
python3 exemplul_1_v1.py # pentru interfata cu PyQt5  
# sau  
python3 exemplul_1_v2.py # pentru interfata cu Tkinter
```

Observație: dacă nu sunt instalate dependențele, se execută următoarele comenzi:

```
sudo apt install python3-venv # pentru medii de lucru virtuale  
sudo apt install python3-pip # package manager pentru python3  
# cd path/to/StackApp  
python3 -m venv env # creare mediu de lucru virtual  
source env/bin/activate # activare mediu de lucru virtual  
pip3 install -r requirements.txt # install pyqt5 tkinter, pygubu,  
pika, retry
```

Multimea A: [71, 23, 29, 61, 19, 17, 47, 2, 89, 41, 13, 67, 5, 31, 3, 43, 53, 83, 7, 79] Generare multimea A

Multimea B: [5, 79, 73, 71, 67, 19, 53, 41, 89, 2, 23, 61, 83, 29, 31, 7, 13, 59, 97, 3] Generare multimea B

A: [71, 23, 29, 61, 19, 17, 47, 2, 89, 41, 13, 67, 5, 31, 3, 43, 53, 83, 7, 79]
B: [5, 79, 73, 71, 67, 19, 53, 41, 89, 2, 23, 61, 83, 29, 31, 7, 13, 59, 97, 3]
result: [(71, 5), (71, 79), (71, 73), (71, 71), (71, 67), (71, 19), (71, 53), (71, 41), (71, 89), (71, 2), (71, 23), (71, 61), (71, 83), (71, 29), (71, 31), (71, 7), (71, 13), (71, 59), (71, 97), (71, 3), (23, 5), (23, 79), (23, 73), (23, 71), (23, 67), (23, 19), (23, 53), (23, 41), (23, 89), (23, 2)]

Calculare expresie

Interfață grafică realizată cu PyGubu și Tkinter

Exemplul 1: Configurări

Se accesează **localhost:15672**, se introduc credențialele, apoi se navighează pe tab-ul de „Exchanges”. Se va configura următorul exchange:

▼ Add a new exchange

Name:	stackapp.direct	*
Type:	direct	▼
Durability:	Durable	▼
Auto delete:	No	▼
Internal:	No	▼
Arguments:	= String ▼	
Add Alternate exchange ? <input type="button" value="Add exchange"/>		

Se navighează apoi pe tab-ul „Queues” și se creează două cozi de mesaje:

▼ Add a new queue

Name:	stackapp.queue	*
Durability:	Durable	▼
Auto delete:	No	▼
Arguments:	= String ▼	
Add Message TTL ? Auto expire ? Max length ? Max length bytes ? Overflow behaviour ? Dead letter exchange ? Dead letter routing key ? Maximum priority ? Lazy mode ? Master locator ?		
<input type="button" value="Add queue"/>		

Prima coadă de mesaje pentru StackApp

Similar, se creează și a doua coadă de mesaje, denumită **stackapp.queue1**

Tot pe tab-ul de „Queues”, în tabelul de mai jos, se da click întâi pe stackapp.queue:

Overview		Messages				Message rates			+/-
Name	Features	State	Ready	Unacked	Total	incoming	deliver / get	ack	
libraryapp.queue	D	idle	1	0	1				
libraryapp.queue1	D	idle	0	0	0				
stackapp.queue	D	idle	1	0	1				
stackapp.queue1	D	idle	0	0	0				

Tabel cozi de mesaje create

Prima coadă de mesaje se configerează ca mai jos:

Laborator 5

From: stackapp.direct *

Routing key: stackapp.routingkey

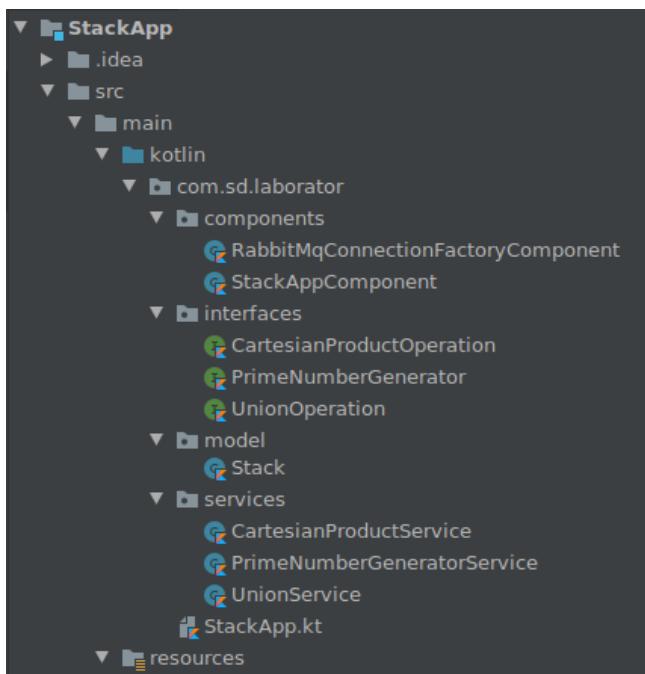
Arguments: = String

Add binding to this queue

Bind

Similar, se dă click pe a doua coadă de mesaje creată, iar la **Routing key**: stackapp.routingkey1

Structurarea proiectului



Ierarhia proiectului StackApp

Configurarea parametrilor pentru conexiunea cu RabbitMQ

Pentru a modifica cu ușurință setările ulterior, acestea vor fi încărcate dintr-un fișier de configurare ce va fi creat în pachetul resources (din folder-ul main). Se creează astăzi fișierul *application.properties* cu următorul conținut:

```
spring.rabbitmq.host=localhost
spring.rabbitmq.port=5672
spring.rabbitmq.username=student
spring.rabbitmq.password=student
stackapp.rabbitmq.queue=stackapp.queue1
```

```
stackapp.rabbitmq.exchange=stackapp.direct
stackapp.rabbitmq.routingkey=stackapp.routingkey
```

Exemplul 1: Codul sursă

Se recomandă ca întâi să accesați în browser consola de administrare a RabbitMQ (localhost:15672) și să stergeți cozile de mesaje create de colegii voștri, apoi să le refaceti (cu binding-uri ca în configurațiile de mai sus).

Aplicația python (interfața grafică)

În folder-ul cu interfața în python, se creează un fișier numit *requirements.txt*, cu următorul conținut:

```
tk==0.1.0
pika==1.1.0
retry==0.9.2
```

Se creează un mediu de lucru virtual și se instalează dependențele de mai sus, executând într-un terminal deschis în folder-ul interfeței comenziile:

```
python3 -m venv env
venvact # alias for: source env/bin/activate
pip3 install -r requirements.txt
```

Terminalul va rămâne deschis pentru a porni din mediul virtual interfața grafică.

```
# exemplul_1_v3.py
import os
import json
import tkinter as tk
from functools import partial
from mq_communication import RabbitMq

class StackApp:
    ROOT_DIR = os.path.dirname(os.path.abspath(__file__))
    A = None
    B = None

    def __init__(self, gui):
        self.gui = gui
        self.gui.title('Exemplul 1 cu Tkinter')

        self.gui.geometry("1050x300")

        self.stack_A_lbl = tk.Label(master=self.gui,
                                    text="Multimea A:")
        self.stack_B_lbl = tk.Label(master=self.gui,
                                    text="Multimea B:")
        self.stack_A = tk.Label(master=self.gui, text="[1, 2, 3]")
        self.stack_B = tk.Label(master=self.gui, text="[4, 5, 6])

        self.regenerate_A_btn = tk.Button(master=self.gui,
                                         text="Generare multimea A",
                                         command=partial(self.send_request,
                                         request='regenerate_A'))
```

Laborator 5

```
self.regenerate_B_btn = tk.Button(master=self.gui,
                                   text="Generare multimea B",
                                   command=partial(self.send_request,
                                                  request='regenerate_B'))
self.compute_btn = tk.Button(master=self.gui,
                             text="Calculare expresie",
                             command=partial(self.send_request,
                                            request='compute'))

self.result = tk.Text(self.gui, width=50, height=10)

# alignment on the grid
self.stack_A_lbl.grid(row=0, column=0)
self.stack_B_lbl.grid(row=1, column=0)
self.stack_A.grid(row=0, column=1)
self.stack_B.grid(row=1, column=1)
self.regenerate_A_btn.grid(row=0, column=2)
self.regenerate_B_btn.grid(row=1, column=2)
self.compute_btn.grid(row=2, column=2)
self.result.grid(row=2, column=0)

self.rabbit_mq = RabbitMq(self)
self.gui.mainloop()

def set_response(self, variable, response):
    if variable == 'A':
        self.regenerate_A(response)
    elif variable == 'B':
        self.regenerate_B(response)
    elif variable == 'compute':
        self.compute(response)

def send_request(self, request):
    self.rabbit_mq.send_message(message=request)
    self.rabbit_mq.receive_message()

def regenerate_A(self, response):
    self.A = response
    current_result = self.result.get("1.0", tk.END).split('\n')
    current_result[0] = 'A: ' + self.A
    self.stack_A['text'] = self.A
    self.result.delete("1.0", tk.END)
    self.result.insert(tk.END, '\n'.join(current_result))

def regenerate_B(self, response):
    self.B = response
    current_result = self.result.get("1.0", tk.END).split('\n')
    if len(current_result) == 1:
        current_result.append('B: ' + self.B)
    else:
        current_result[1] = 'B: ' + self.B
    self.stack_B['text'] = self.B
    self.result.delete("1.0", tk.END)
    self.result.insert(tk.END, '\n'.join(current_result))

def compute(self, response):
    dict_response = json.loads(response)
```

```

result = ''
for key in dict_response:
    result += '{}: {}\n'.format(key, dict_response[key])
self.stack_A['text'] = dict_response['A']
self.stack_B['text'] = dict_response['B']
self.result.delete("1.0", tk.END)
self.result.insert(tk.END, result)

if __name__ == '__main__':
    root = tk.Tk()
    app = StackApp(root)
    root.mainloop()

```

Se remarcă utilizarea funcției **partial** din modulul `functools`. În mod normal, un buton nu poate trimite un parametru la apelul funcției asignate evenimentului `click`. Totuși, utilizând funcția **partial**, poate fi trimis un parametru suplimentar cu o valoare prestabilită. Pentru mai multe detalii, vezi <https://docs.python.org/3/library/functools.html>

Se observă de asemenea partea de comunicare prin cozi de mesaje (funcțiile `send_request` și `set_response` - apelată din modulul `mq_communication`).

Modulul `mq_communication` realizează conexiunea propriu-zisă cu RabbitMQ.

```

# mq_communication.py
import pika
from retry import retry

class RabbitMq:
    config = {
        'host': '0.0.0.0',
        'port': 5678,
        'username': 'student',
        'password': 'student',
        'exchange': 'stackapp.direct',
        'routing_key': 'stackapp.routingkey1',
        'queue': 'stackapp.queue'
    }
    credentials = pika.PlainCredentials(config['username'],
                                         config['password'])
    parameters = (pika.ConnectionParameters(host=config['host']),
                  pika.ConnectionParameters(port=config['port']),
                  pika.ConnectionParameters(credentials=credentials))

    def __init__(self, ui):
        self.ui = ui

    def on_received_message(self, blocking_channel,
                           deliver, properties, message):
        result = message.decode('utf-8')
        blocking_channel.confirm_delivery()
        try:
            variable, response = result.split('~')
            self.ui.set_response(variable, response)
        except Exception as e:
            print(e)
            print("wrong data format")
        finally:

```

```

blocking_channel.stop_consuming()

@retry(pika.exceptions.AMQPConnectionError, delay=5, jitter=(1,
3))
def receive_message(self):
    # automatically close the connection
    with pika.BlockingConnection(self.parameters) as connection:
        # automatically close the channel
        with connection.channel() as channel:
            channel.basic_consume(self.config['queue'],
                                  self.on_received_message)
    try:
        channel.start_consuming()
    # Don't recover connections closed by server
    except pika.exceptions.ConnectionClosedByBroker:
        print("Connection closed by broker.")
    # Don't recover on channel errors
    except pika.exceptions.AMQPChannelError:
        print("AMQP Channel Error")
    # Don't recover from KeyboardInterrupt
    except KeyboardInterrupt:
        print("Application closed.")

def send_message(self, message):
    # automatically close the connection
    with pika.BlockingConnection(self.parameters) as connection:
        # automatically close the channel
        with connection.channel() as channel:
            self.clear_queue(channel)
            channel.basic_publish(
                exchange=self.config['exchange'],
                routing_key=self.config['routing_key'],
                body=message)

def clear_queue(self, channel):
    channel.queue_purge(self.config['queue'])

```

Se remarcă funcțiile **receive_message** și **send_message** ce citesc scriu într-o coadă de mesaje.

De reținut: utilizarea unui bloc **with** automatizează închiderea variabilei deschise (spre exemplu: închiderea fișierului, a conexiunii, a canalului, etc), apelând la ieșirea din blocul **with** identat, funcția **close**.

Se remarcă utilizarea unui design pattern: **decoratorul**. Acesta este folosit prin adnotarea **@retry** ce reîncearcă apelarea funcției **receive_message** la apariția unei erori de tipul **AMQPConnectionError**. Pentru mai multe detalii, vezi <https://pypi.org/project/retry/>

Se observă de asemenea utilizarea unei funcții de callback - **on_received_message**, ce va fi apelată în momentul în care se citește un mesaj din coadă.

Pentru a porni interfața, se revine la terminalul cu mediul virtual pornit și se execută:

```
python3 exemplul_1_v3.py # modificați denumirea corespunzător
```

Aplicația kotlin (procesarea request-urilor)

Se creează întâi fișierul *src/main/kotlin/com.sd.laborator/StackApp.kt*

```

package com.sd.laborator

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

@SpringBootApplication
class StackApp

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

```

Pachetul model

Se creează fișierul Stack.kt:

```

package com.sd.laborator.model

data class Stack(var data: MutableSet<Int>)

```

Pachetul interfaces

- CartesianProductOperation.kt

```

package com.sd.laborator.interfaces

interface CartesianProductOperation {
    fun executeOperation(A: Set<Int>, B: Set<Int>): Set<Pair<Int,
    Int>>
}

```

- PrimeNumberGenerator.kt

```

package com.sd.laborator.interfaces

interface PrimeNumberGenerator {
    fun generatePrimeNumber(): Int
}

```

- UnionOperation.kt

```

package com.sd.laborator.interfaces

interface UnionOperation {
    fun executeOperation(A: Set<Pair<Int, Int>>, B: Set<Pair<Int,
    Int>>): Set<Pair<Int, Int>>
}

```

Pachetul services

- CartesianProductService.kt

```

package com.sd.laborator.services

import com.sd.laborator.interfaces.CartesianProductOperation
import org.springframework.stereotype.Service

@Service

```

Laborator 5

```
class CartesianProductService: CartesianProductOperation {
    override fun executeOperation(A: Set<Int>, B: Set<Int>):
Set<Pair<Int, Int>> {
        var result: MutableSet<Pair<Int, Int>> = mutableSetOf()
        A.forEach { a -> B.forEach { b -> result.add(Pair(a, b)) } }
        return result.toSet()
    }
}
```

- PrimeNumberGeneratorService.kt

```
package com.sd.laborator.services

import com.sd.laborator.interfaces.PrimeNumberGenerator
import org.springframework.stereotype.Service

@Service
class PrimeNumberGeneratorService: PrimeNumberGenerator {
    private val primeNumbersIn1To100: Set<Int> = setOf(2, 3, 5, 7, 11,
13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79,
83, 89, 97)

    override fun generatePrimeNumber(): Int {
        return primeNumbersIn1To100.elementAt((0 until
primeNumbersIn1To100.count()).random())
    }
}
```

- UnionService.kt

```
package com.sd.laborator.services

import com.sd.laborator.interfaces.CartesianProductOperation
import com.sd.laborator.interfaces.UnionOperation
import org.springframework.stereotype.Service

@Service
class UnionService: UnionOperation {
    override fun executeOperation(A: Set<Pair<Int, Int>>, B:
Set<Pair<Int, Int>>): Set<Pair<Int, Int>> {
        return A union B
    }
}
```

Pachetul components

- StackAppComponent.kt

```
package com.sd.laborator.components

import com.sd.laborator.interfaces.CartesianProductOperation
import com.sd.laborator.interfaces.PrimeNumberGenerator
import com.sd.laborator.interfaces.UnionOperation
import com.sd.laborator.model.Stack
import org.springframework.amqp.core.AmqpTemplate
```

```

import org.springframework.amqp.rabbit.annotation.RabbitListener
import org.springframework.beans.factory.annotation.Autowired
import org.springframework.stereotype.Component

@Component
class StackAppComponent {
    private var A: Stack? = null
    private var B: Stack? = null

    @Autowired
    private lateinit var primeGenerator: PrimeNumberGenerator
    @Autowired
    private lateinit var cartesianProductOperation:
CartesianProductOperation
    @Autowired
    private lateinit var unionOperation: UnionOperation
    @Autowired
    private lateinit var connectionFactory:
RabbitMqConnectionFactoryComponent

    private lateinit var amqpTemplate: AmqpTemplate

    @Autowired
    fun initTemplate() {
        this.amqpTemplate = connectionFactory.rabbitTemplate()
    }

    @RabbitListener(queues = ["${stackapp.rabbitmq.queue}"])
    fun recieveMessage(msg: String) {
        // the result: 114,101,103,101,110,101,114,97,116,101,95,65 -> needs processing
        val processed_msg = (msg.split(",")).map { it.toInt().toChar() }.joinToString(separator="")
        var result: String? = when(processed_msg) {
            "compute" -> computeExpression()
            "regenerate_A" -> regenerateA()
            "regenerate_B" -> regenerateB()
            else -> null
        }
        println("result: ")
        println(result)
        if (result != null) sendMessage(result)
    }

    fun sendMessage(msg: String) {
        println("message: ")
        println(msg)
        this.amqpTemplate.convertAndSend(connectionFactory.getExchange(),
                                         connectionFactory.getRoutingKey(),
                                         msg)
    }

    private fun generateStack(count: Int): Stack? {
        if (count < 1)
            return null
    }
}

```

Laborator 5

```
var X: MutableSet<Int> = mutableSetOf()
while (X.count() < count)
    X.add(primeGenerator.generatePrimeNumber())
return Stack(X)
}

private fun computeExpression(): String {
    if (A == null)
        A = generateStack(20)
    if (B == null)
        B = generateStack(20)
    if (A!!.data.count() == B!!.data.count()) {
        // (A x B) U (B x B)
        val partialResult1 =
cartesianProductOperation.executeOperation(A!!.data, B!!.data)
        val partialResult2 =
cartesianProductOperation.executeOperation(B!!.data, B!!.data)
        val result =
unionOperation.executeOperation(partialResult1, partialResult2)
        return "compute~" + "{\"A\": \"\" + A?.data.toString() +
\"\", \"B\": \"\" + B?.data.toString() + "\", \"result\": \"\" +
result.toString() + \"\"}"
    }
    return "compute~" + "Error: A.count() != B.count()"
}

private fun regenerateA(): String {
    A = generateStack(20)
    return "A~" + A?.data.toString()
}

private fun regenerateB(): String {
    B = generateStack(20)
    return "B~" + B?.data.toString()
}
}
```

Se remarcă adnotarea clasei cu **@Component** pentru a putea fi descoperită la pornirea aplicației cu spring. De asemenea, se observă că variabilele private sunt declarate ca **lateinit**, deoarece vor fi injectate de spring (sunt adnotate cu **@Autowired**). Pentru variabila **amqpTemplate**, a fost creată o metodă care să injecteze valoarea (un RabbitTemplate), aceasta fiind adnotată cu **@Autowired** în locul variabilei propriu-zise.

Se poate vedea faptul că **listener-ul** (funcția care citește din coadă) este adnotată cu **@RabbitListener(queues = ["\${stackapp.rabbitmq.queue}"])**, primind ca parametru coada de mesaje din care citește. Totodată, se observă că funcția primește direct parametrul **msg** de tip String, ce reprezintă mesajul citit din coadă.

Pentru metoda de trimitere a unui mesaj, este nevoie de numele **exchange-ului** și de **routing key**. **Atenție la cheia de rutare! aceasta selectează practic coada destinație (exchange-ul fiind același pentru ambele cozi).**

StackAppComponent este mediatorul (vezi orchestrarea serviciilor). Aici se utilizează toate serviciile cu scopul de a realiza funcționalitatea dorită (calcularea expresiei în cazul de față).

- RabbitMqConnectionFactoryComponent.kt

```
package com.sd.laborator.components
```

```

import org.springframework.amqp.rabbit.connection.CachingConnectionFactory
import org.springframework.amqp.rabbit.connection.ConnectionFactory
import org.springframework.amqp.rabbit.core.RabbitTemplate
import org.springframework.beans.factory.annotation.Value
import org.springframework.context.annotation.Bean
import org.springframework.stereotype.Component

@Component
class RabbitMqConnectionFactoryComponent {
    @Value("\${spring.rabbitmq.host}")
    private lateinit var host: String
    @Value("\${spring.rabbitmq.port}")
    private val port: Int = 0
    @Value("\${spring.rabbitmq.username}")
    private lateinit var username: String
    @Value("\${spring.rabbitmq.password}")
    private lateinit var password: String
    @Value("\${stackapp.rabbitmq.exchange}")
    private lateinit var exchange: String
    @Value("\${stackapp.rabbitmq.routingkey}")
    private lateinit var routingKey: String

    fun getExchange(): String = this.exchange

    fun getRoutingKey(): String = this.routingKey

    @Bean
    private fun connectionFactory(): ConnectionFactory {
        val connectionFactory = CachingConnectionFactory()
        connectionFactory.host = this.host
        connectionFactory.username = this.username
        connectionFactory.setPassword(this.password)
        connectionFactory.port = this.port
        return connectionFactory
    }

    @Bean
    fun rabbitTemplate(): RabbitTemplate =
        RabbitTemplate(connectionFactory())
}

```

Se remarcă adnotarea clasei cu **@Component** pentru a putea fi descoperită la pornirea aplicației cu spring. De asemenea, se observă adnotările **@Value** care inițializează variabilele adnotate cu valorile din fișierul *application.properties*. Pentru ca spring-ul să poată gestiona obiectele **ConnectionFactory** și respectiv **RabbitTemplate** create, funcțiile care creează aceste obiecte trebuie adnotate cu **@Bean**. Pentru mai multe detalii despre bean-uri, vezi;

- <https://www.baeldung.com/spring-bean>
- <https://docs.spring.io/spring-javaconfig/docs/1.0.0.m3/reference/html/creating-bean-definitions.html>

Pentru pornirea aplicației, din meniul **Maven**, se apasă **clean --> compile --> spring-boot:run**. Apoi dintr-un terminal cu mediul virtual pornit (și dependențele din requirements.txt instalate), se execută comanda **python3 exemplul_1_v1.py**

Exemplul 2: LibraryApp

Cerință: Să se scrie un program Kotlin care să realizeze gestiunea unei biblioteci prin intermediul unor servicii, utilizând principiile SOLID. Aplicația va conține trei moduri de afișare a datelor (HTML, JSON și Raw) și va expune utilizatorului prin interfață funcționalități de tip CRUD (Create, Retrieve, Update, Delete).

Arhitectura aplicației este reprezentată în diagrama de mai jos. Spre deosebire de exemplul anterior, se observă și *Interface segregation principle* în cadrul LibraryPrinter. Deși LibraryPrinter înglobează toate cele trei tipuri de afișare, această funcționalitate poate fi ușor modificată implementând doar interfețele necesare unui client.

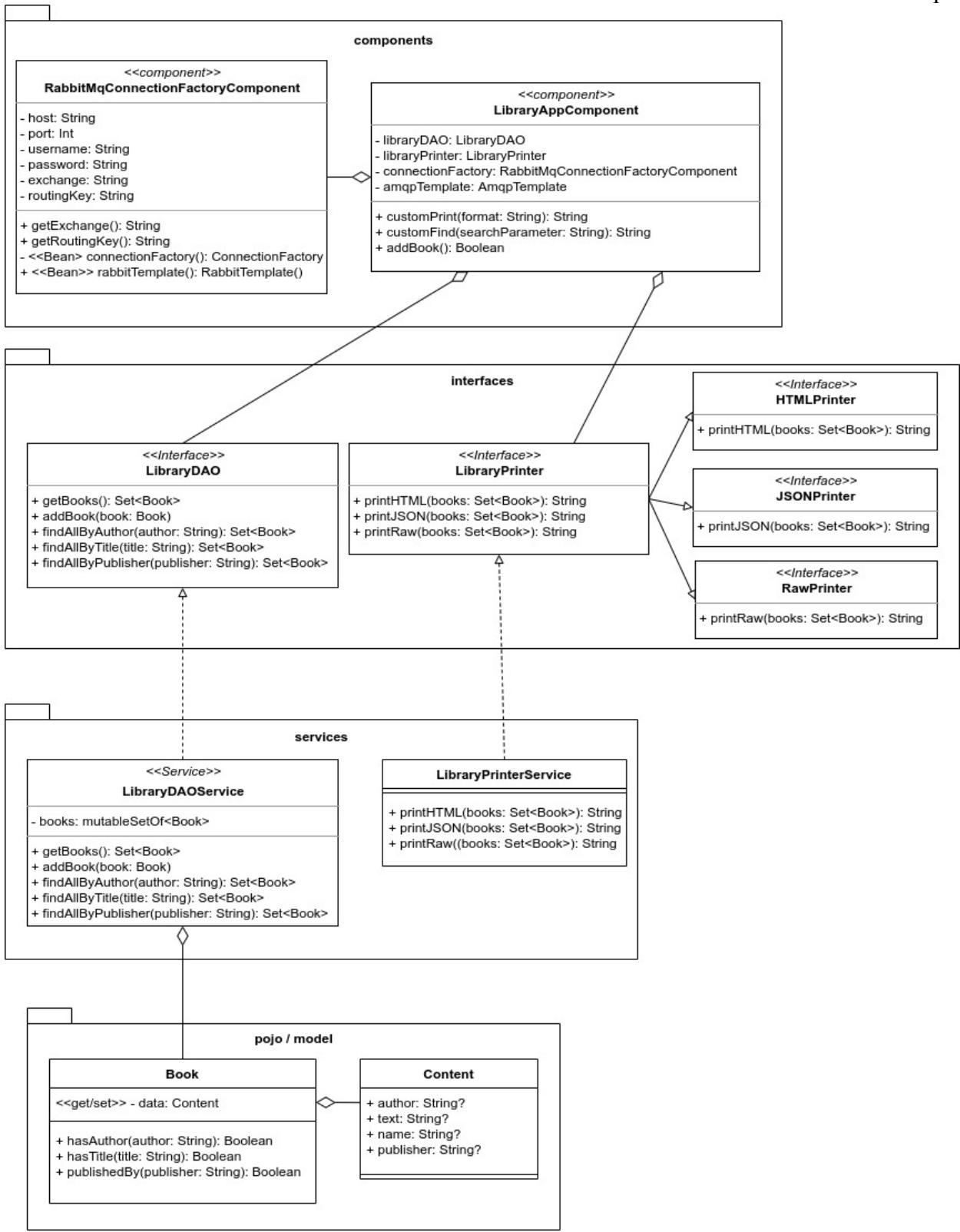
Similar cu exemplu anterior, compilați întâi proiectul în kotlin, executați din fereastra Maven --> Plugins --> spring-boot --> spring-boot:run. Apoi, deschideți un terminal în folder-ul interfeței (*qt_gui*) și executați comanda:

```
python3 exemplul_2.py # interfata cu PyQt5
```

Soluția propusă are următoarele flow-uri:

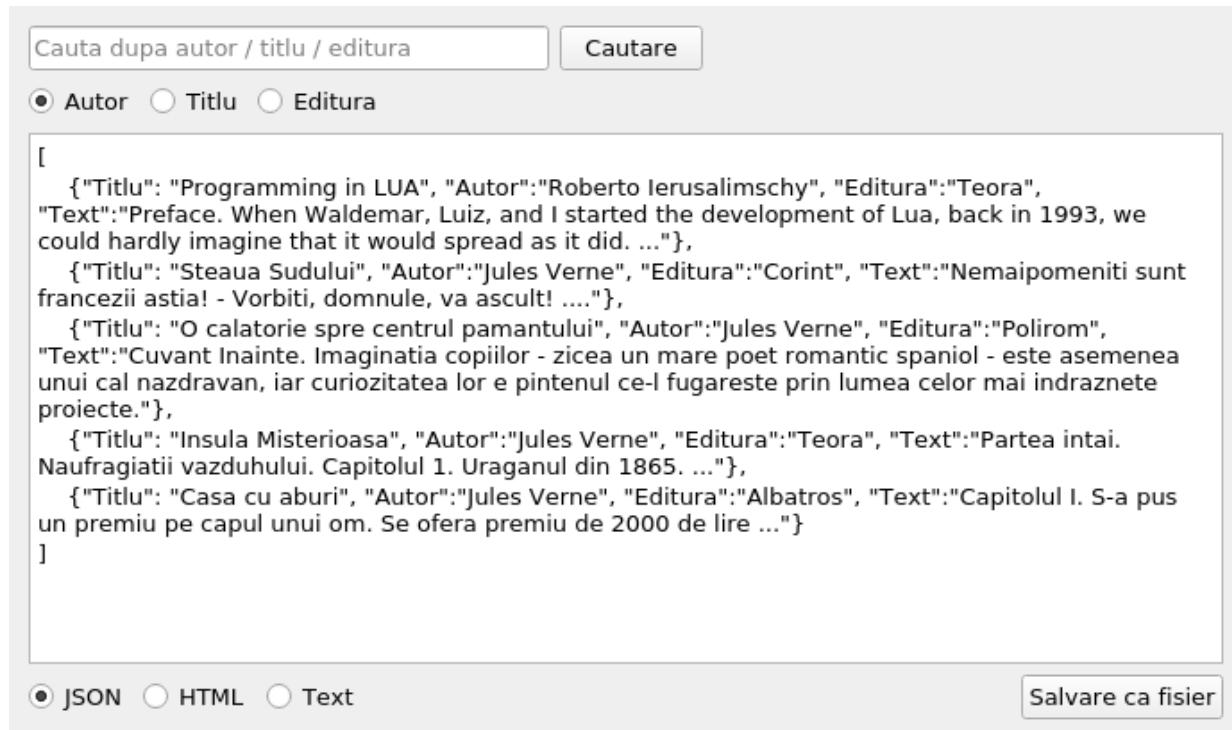
- căutare fără introducere de cuvinte cheie -> indiferent de selecția căutării (autor / titlu / editura), programul va afișa toate cărțile în formatul specificat (JSON / HTML / Text)
- căutare cu introducere de cuvinte cheie -> programul va filtra lista de cărți în funcție de câmpul dorit (autor / titlu / editură), afișând rezultatul în formatul selectat
- căutare urmată de salvare fișier -> va salva conținutul găsit într-un fișier cu extensia .html, .json sau .txt (în funcție de selecție)

Observație: la realizarea unei căutări cu filtrare, rezultatul va fi afișat în format JSON indiferent de selecția curentă. Aceasta se datorează faptului că interfața din python nu transmite (momentan) tipul de fișier. Puteti modifica exemplul astfel încât să trimiteți încă un parametru (modul de printare) la căutarea cu filtrare.



Arhitectura aplicației LibraryApp

Laborator 5



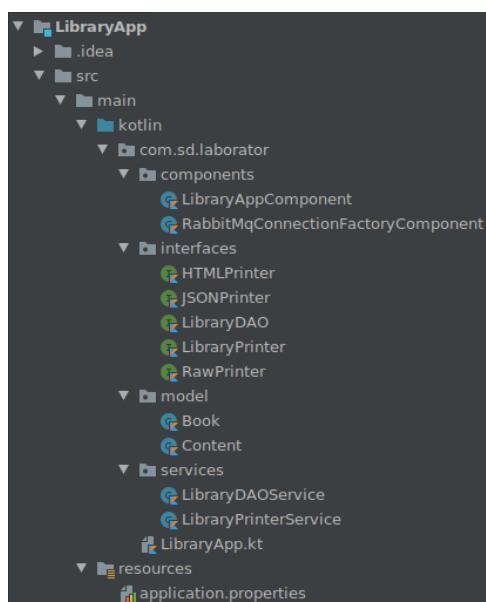
Interfață grafică pentru LibraryApp realizată cu PyQt5

Exemplul 2: Configurări

Analog cu exemplul 1, se creează:

- un exchange: **libraryapp.direct**
- două cozi
 - **libraryapp.queue**
 - **libraryapp.queue1**
- două binding-uri:
 - **libraryapp.queue -> libraryapp.direct, libraryapp.routingkey**
 - **libraryapp.queue1 -> libraryapp.direct, libraryapp.routingkey1**

Structura proiectului



Ierarhia proiectului LibraryApp

Configurarea parametrilor pentru conexiunea cu RabbitMQ

Se creează fișierul *src/main/resources/application.properties* cu următorul conținut:

```
spring.rabbitmq.host=localhost
spring.rabbitmq.port=5672
spring.rabbitmq.username=student
spring.rabbitmq.password=student
libraryapp.rabbitmq.queue=libraryapp.queue1
libraryapp.rabbitmq.exchange=libraryapp.direct
libraryapp.rabbitmq.routingkey=libraryapp.routingkey
```

Configurarea proiectului

Se reiau pașii de la exemplul 1 (sunt aceleași dependențe și plugin-uri).

Exemplul 2: codul sursă

Se creează întâi în pachetul *com.sd.laborator* fișierul *LibraryApp.kt*:

```
package com.sd.laborator

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

@SpringBootApplication
class LibraryApp

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

Pachetul model

- Book.kt

Se remarcă *getters* și *setters* care accesează atribute ale variabilei **data** de tip **Content**.

```
package com.sd.laborator.model

class Book(private var data: Content) {

    var name: String?
        get() {
            return data.name
        }
        set(value) {
            data.name = value
        }

    var author: String?
        get() {
            return data.author
        }
        set(value) {
            data.author = value
        }
}
```

Laborator 5

```
var publisher: String?
    get() {
        return data.publisher
    }
    set(value) {
        data.publisher = value
    }

var content: String?
    get() {
        return data.text
    }
    set(value) {
        data.text = value
    }

fun hasAuthor(author: String): Boolean {
    return data.author.equals(author)
}

fun hasTitle(title: String): Boolean {
    return data.name.equals(title)
}

fun publishedBy(publisher: String): Boolean {
    return data.publisher.equals(publisher)
}

}
```

- Content.kt

```
package com.sd.laborator.model

data class Content(var author: String?, var text: String?, var name: String?, var publisher: String?)
```

Pachetul interfaces

- HTMLPrinter.kt

```
package com.sd.laborator.interfaces

import com.sd.laborator.model.Book

interface HTMLPrinter {
    fun printHTML(books: Set<Book>): String
}
```

- JSONPrinter.kt

```
package com.sd.laborator.interfaces

import com.sd.laborator.model.Book
```

```
interface JSONPrinter {
    fun printJSON(books: Set<Book>): String
}
```

- LibraryDAO.kt

```
package com.sd.laborator.interfaces

import com.sd.laborator.model.Book

interface LibraryDAO {
    fun getBooks(): Set<Book>
    fun addBook(book: Book)
    fun findAllByAuthor(author: String): Set<Book>
    fun findAllByTitle(title: String): Set<Book>
    fun findAllByPublisher(publisher: String): Set<Book>
}
```

- LibraryPrinter.kt

```
package com.sd.laborator.interfaces

interface LibraryPrinter: HTMLPrinter, JSONPrinter, RawPrinter
```

- RawPrinter.kt

```
package com.sd.laborator.interfaces

import com.sd.laborator.model.Book

interface RawPrinter {
    fun printRaw(books: Set<Book>): String
}
```

Pachetul services

- LibraryDAOService.kt

Observație: Abrevierea DAO înseamnă Data Access Object.

```
package com.sd.laborator.services

import com.sd.laborator.interfaces.LibraryDAO
import com.sd.laborator.model.Book
import com.sd.laborator.model.Content
import org.springframework.stereotype.Service

@Service
class LibraryDAOService: LibraryDAO {
    private var books: MutableSet<Book> = mutableSetOf(
        Book(Content("Roberto Ierusalimschy", "Preface. When Waldemar,
        Luiz, and I started the development of Lua, back in 1993, we could
        hardly imagine that it would spread as it did. ...", "Programming in
        LUA", "Teora")),
```

Laborator 5

```
        Book(Content("Jules Verne","Nemaipomeniti sunt francezii  
astia! - Vorbiti, domnule, va ascult! ....","Steaua  
Sudului","Corint")),  
        Book(Content("Jules Verne","Cuvant Inainte. Imaginatia  
copiilor - zicea un mare poet romantic spaniol - este asemenea unui  
cal nazdravan, iar curiozitatea lor e pintenul ce-l fugareste prin  
lumea celor mai indraznate proiecte.","O calatorie spre centrul  
pamantului","Polirom")),  
        Book(Content("Jules Verne","Partea intai. Naufragiati  
vazduhului. Capitolul 1. Uraganul din 1865. ....","Insula  
Misterioasa","Teora")),  
        Book(Content("Jules Verne","Capitolul I. S-a pus un premiu pe  
capul unui om. Se ofera premiu de 2000 de lire ...","Casa cu  
aburi","Albatros"))  
    )  
    override fun getBooks(): Set<Book> {  
        return this.books  
    }  
  
    override fun addBook(book: Book) {  
        this.books.add(book)  
    }  
  
    override fun findAllByAuthor(author: String): Set<Book> {  
        return (this.books.filter { it.hasAuthor(author) }).toSet()  
    }  
  
    override fun findAllByTitle(title: String): Set<Book> {  
        return (this.books.filter { it.hasTitle(title) }).toSet()  
    }  
  
    override fun findAllByPublisher(publisher: String): Set<Book> {  
        return (this.books.filter { it.publishedBy(publisher)  
    }).toSet()  
    }  
}
```

- LibraryPrinterService.kt

```
package com.sd.laborator.services  
  
import com.sd.laborator.interfaces.LibraryPrinter  
import com.sd.laborator.model.Book  
import org.springframework.stereotype.Service  
  
@Service  
class LibraryPrinterService: LibraryPrinter {  
    override fun printHTML(books: Set<Book>): String {  
        var content: String = "<html><head><title>Libraria mea  
HTML</title></head><body>"  
        books.forEach {  
            content +=  
                "<p><h3>${it.name}</h3><h4>${it.author}</h4><h5>${it.publisher}</h5>${  
                    it.content}</p><br/>"  
        }  
        content += "</body></html>"  
        return content  
    }  
}
```

```

    }

    override fun printJSON(books: Set<Book>): String {
        var content: String = "[\n"
        books.forEach {
            if (it != books.last())
                content += "      {\\"Titlu\\": \\"${it.name}\",
\"Autor\\": \\"${it.author}\", \\"Editura\\": \\"${it.publisher}\",
\"Text\\": \\"${it.content}\", \n"
            else
                content += "      {\\"Titlu\\": \\"${it.name}\",
\"Autor\\": \\"${it.author}\", \\"Editura\\": \\"${it.publisher}\",
\"Text\\": \\"${it.content}\"}\n"
        }
        content += "]\n"
        return content
    }

    override fun printRaw(books: Set<Book>): String {
        var content: String = ""
        books.forEach {
            content +=
"${it.name}\n${it.author}\n${it.publisher}\n${it.content}\n\n"
        }
        return content
    }
}

```

Pachetul components

- RabbitMqConnectionFactoryComponent.kt

```

package com.sd.laborator.components

import org.springframework.amqp.rabbit.connection.CachingConnectionFactory
import org.springframework.amqp.rabbit.connection.ConnectionFactory
import org.springframework.amqp.rabbit.core.RabbitTemplate
import org.springframework.beans.factory.annotation.Value
import org.springframework.context.annotation.Bean
import org.springframework.stereotype.Component

@Component
class RabbitMqConnectionFactoryComponent {
    @Value("\${spring.rabbitmq.host}")
    private lateinit var host: String
    @Value("\${spring.rabbitmq.port}")
    private val port: Int = 0
    @Value("\${spring.rabbitmq.username}")
    private lateinit var username: String
    @Value("\${spring.rabbitmq.password}")
    private lateinit var password: String
    @Value("\${libraryapp.rabbitmq.exchange}")
    private lateinit var exchange: String
    @Value("\${libraryapp.rabbitmq.routingkey}")
    private lateinit var routingKey: String

    fun getExchange(): String = this.exchange
}

```

Laborator 5

```
fun getRoutingKey(): String = this.routingKey

@Bean
private fun connectionFactory(): ConnectionFactory {
    val connectionFactory = CachingConnectionFactory()
    connectionFactory.host = host
    connectionFactory.username = username
    connectionFactory.setPassword(password)
    connectionFactory.port = port
    return connectionFactory
}

@Bean
fun rabbitTemplate(): RabbitTemplate =
RabbitTemplate(this.connectionFactory())

}
```

- LibraryAppComponent.kt

```
package com.sd.laborator.components

import com.sd.laborator.interfaces.LibraryDAO
import com.sd.laborator.interfaces.LibraryPrinter
import com.sd.laborator.model.Book
import org.springframework.amqp.core.AmqpTemplate
import org.springframework.amqp.rabbit.annotation.RabbitListener
import org.springframework.beans.factory.annotation.Autowired
import org.springframework.stereotype.Component
import java.lang.Exception

@Component
class LibraryAppComponent {
    @Autowired
    private lateinit var libraryDAO: LibraryDAO

    @Autowired
    private lateinit var libraryPrinter: LibraryPrinter

    @Autowired
    private lateinit var connectionFactory:
RabbitMqConnectionFactoryComponent
    private lateinit var amqpTemplate: AmqpTemplate

    @Autowired
    fun initTemplate() {
        this.amqpTemplate = connectionFactory.rabbitTemplate()
    }

    fun sendMessage(msg: String) {
        this.amqpTemplate.convertAndSend(connectionFactory.getExchange(),
                                         connectionFactory.getRouting-
Key(),
                                         msg)
    }
}
```

```

@RabbitListener(queues = ["${libraryapp.rabbitmq.queue}"])
fun recieveMessage(msg: String) {
    // the result needs processing
    val processedMsg = (msg.split(",")).map { it.toInt().toChar()
}).joinToString(separator="")
    try {
        val (function, parameter) = processedMsg.split(":")
        val result: String? = when(function) {
            "print" -> customPrint(parameter)
            "find" -> customFind(parameter)
            else -> null
        }
        if (result != null) sendMessage(result)
    } catch (e: Exception) {
        println(e)
    }
}

fun customPrint(format: String): String {
    return when(format) {
        "html" -> libraryPrinter.printHTML(libraryDAO.getBooks())
        "json" -> libraryPrinter.printJSON(libraryDAO.getBooks())
        "raw" -> libraryPrinter.printRaw(libraryDAO.getBooks())
        else -> "Not implemented"
    }
}

fun customFind(searchParameter: String): String {
    val (field, value) = searchParameter.split("=")
    return when(field) {
        "author" ->
this.libraryPrinter.printJSON(this.libraryDAO.findAllByAuthor(value))
        "title" ->
this.libraryPrinter.printJSON(this.libraryDAO.findAllByTitle(value))
        "publisher" ->
this.libraryPrinter.printJSON(this.libraryDAO.findAllByPublisher(va-
lue))
        else -> "Not a valid field"
    }
}

fun addBook(book: Book): Boolean {
    return try {
        this.libraryDAO.addBook(book)
        true
    } catch (e: Exception) {
        false
    }
}
}

```

Interfață în python este realizată similar cu cea de la exemplul 1. Aceasta va fi preluată din codul sursă atașat laboratorului.

Aplicații și teme

Laborator 5

Aplicații de laborator:

- Respectând principiul lui **Liskov**, să se implementeze o alternativă a *LibraryPrinterService* din exemplul 2.
- Să se modifice exemplul 2 astfel încât fișierul salvat în urma unei căutări cu filtrare (după autor/titlu/editură) să fie salvat și în format HTML / text.
- La salvarea fișierului, să se modifice formatul în funcție de opțiunea aleasă (json / html / text)
- Să se îmbunătățească opțiunea de căutare prin detectarea unor potriviri parțiale (ex.: „Insula“ -> „Insula Misterioasă“), iar căutarea să fie case-insensitive.
- Respectând principiile **SOLID**, să se adauge opțiunea de a printa și în format XML.
- Să se modifice interfața ultimului exemplu: se va adăuga un buton care să deschidă o nouă fereastră cu un formular (autor, text, denumire, editură) pentru introducerea unei cărți în bibliotecă. După preluarea datelor de pe interfață, se va apela metoda *addBook* din *LibraryAppComponent*.

Teme pe acasă:

- Să se reimplementeze primul exemplu folosind înlățuirea serviciilor (chaining) în loc de orchestrarea lor.
- Să se reimplementeze comunicarea prin cozi de mesaje din aplicația python (exemplul 2) utilizând un adaptor asincron (vezi <https://pypi.org/project/pika/> pentru adaptoarele oferite de modulul *pika* și documentația acestuia: <https://pika.readthedocs.io/en/stable/>)
- Să se reimplementeze unul dintre cele două exemple (la alegere), utilizând un alt framework pentru comunicarea prin cozi de mesaje.

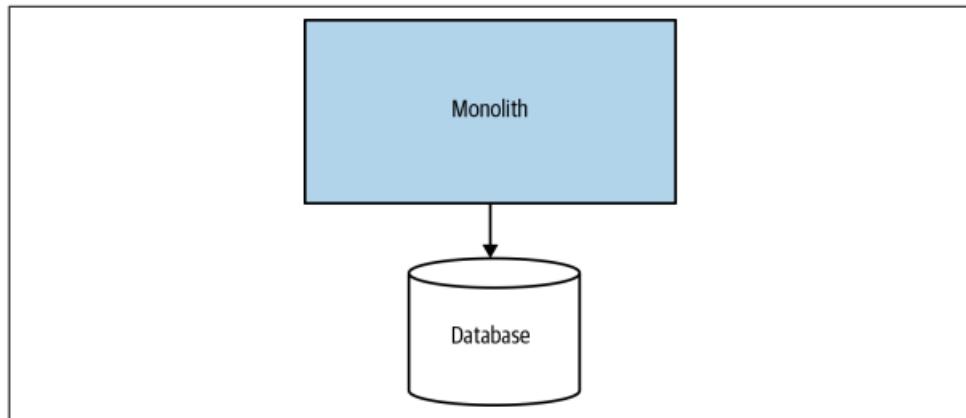
Laborator 6 SD: Microservicii Web cu Spring Boot în Kotlin

Introducere

Aplicații monolitice

Primele aplicații erau de dimensiuni mici, iar logica de prezentare era la un loc cu logica de business. Apoi, modelarea domeniului a devenit complexă, ceea ce a dat naștere mai multor modele de proiectare arhitecturală din categoria separării funcționalităților (Separation of Concerns).

În aplicațiile clasice monolit, toate funcționalitățile aplicației sunt grupate într-un singur pachet și executate ca un singur proces. Interfața cu utilizatorul, nivelul de accesare a datelor și nivelul de stocare a datelor sunt strâns cuplate.



Arhitectura unei aplicații monolit clasice

Avantaje:

- Sunt simplu de dezvoltat (toate instrumentele pentru dezvoltare suportă acest tip de aplicații)
- Sunt simplu de lansat (toate componentele fiind împachetate la un loc)
- Este o scalare ușoară a întregii aplicații

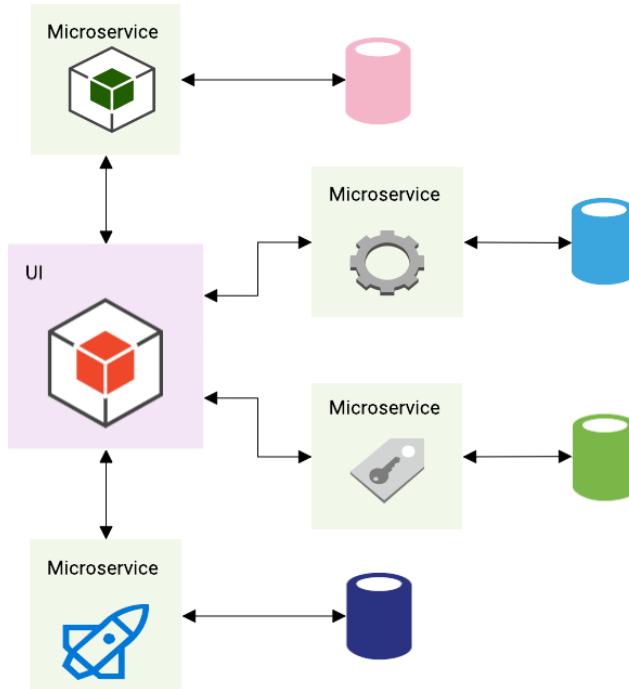
Dezavantaje:

- Sunt complexe și ca urmare este dificilă îmbunătățirea în timpul de viață.
- Adaptarea la noile tehnologii este destul de dificilă
- Sunt greu de integrat într-un proces continuu de dezvoltare CI/CD (Continuous Integration / Continuous Delivery)
- Pornirea aplicației durează destul de mult, din cauza faptului că toate componentele trebuie încărcate înainte de lansarea în execuție.
- Comportamentul eronat al unei componente va conduce la oprirea parțială sau totală a execuției

Aplicații cu microservicii

Termenul de microaplicații fost introdus în 2011 de către James Lewis de la ThoughtWorks care a început să studieze micro aplicații. Acesta era un model de proiectare care începuse deja să fie utilizat de unele companii care utilizau SOC. Ulterior, în 2012, în urma unor discuții la un congres de specialitate, s-a modificat în microservicii pentru a evita confuziile.

În cazul aplicațiilor web bazate pe microservicii acestea sunt formate dintr-o suită de servicii mici, executate independent, care comunică prin mecanisme simple (lightweight) bazate pe HTTP. Acestea au de multe ori un acces centralizat prin intermediul unui API.



Arhitectura decentralizată a microserviciilor

Avantaje:

- Microserviciile pot folosi cele mai noi tehnologii.
- Compozabilitatea este ridicată.
- Microserviciile independente pot fi scalate separat (nu e nevoie de scalarea întregului sistem).
- Defectarea unei componente nu va duce la cădereea sistemului.
- Pentru dezvoltare se utilizează echipe mici care lucrează în paralel la dezvoltarea microserviciilor. Ca rezultat, timpul de dezvoltare se micșorează.
- Procesul de integrare/dezvoltare continuă este nativ.

Dezavantaje:

- Mantenanța codului de bază independent este foarte dificilă.
- Monitorizarea întregului sistem este o adevărată provocare, din cauza decentralizării.
- Are un cost (overhead) de performanță adițional din cauza latenței rețelei (network latency)

Pentru mai multe detalii, vezi:

- [Building Microservices \(2015\) - Sam Newman](#)
- [Hands-on microservices with Kotlin \(2018\) - Juan Antonio Medina Iglesias](#)

Principii generale pentru proiectarea cu microservicii

- **Modeled around business capabilities** - proiectarea software are o componentă de abstractizare, dezvoltatorii fiind obișnuiți să primească sarcini și să le implementeze, dar trebuie luat în considerare cum o să fie înțeleasă soluția, atât acum cât și în viitor. **Se recomandă să se lucreze îndeaproape cu experții din domeniul respectiv.**
- **Cuplare scăzută (Loosely couple)** - Nici un microserviciu nu există pe cont propriu, fiecare sistem având nevoie să interacționeze cu altele (alte microservicii), dar e nevoie ca această interacție să fie slab cuplată. De exemplu, dacă se proiectează un microserviciu care returnează numărul de oferte disponibile pentru un anumit client, este

nevoie de o relație către clientul respectiv (customer ID), iar acesta ar fi nivelul maxim de cuplare acceptat.

- **Responsabilitate unică (single responsibility)** - Fiecare microserviciu are ca responsabilitate o singură parte din funcționalitatea aplicației, iar acea responsabilitate este încapsulată în interiorul lui.
- **Ascunderea implementării (Hiding implementation)** - Microserviciile au în general un contract (o interfață) clar și ușor de înțeles, care ascunde detaliile de implementare. Detaliile interne nu ar trebui expuse, nici implementarea tehnică, nici regulile de business care o conduc.
- **Izolare (Isolation)** - Un microserviciu trebuie izolat fizic și/sau logic de infrastructură care utilizează sistemul de care depinde (baza de date, server, etc). Astfel, se poate garanta că nimic extern nu poate afecta funcționalitatea aplicației, iar aplicația nu poate afecta ceva extern.
- **Instalare independentă (Independently deployable)** - Un microserviciu trebuie să poată fi instalat (deployed) în mod independent. În caz contrar, există un nivel de cuplare în interiorul arhitecturii care trebuie rezolvat. **Abilitatea de a livra în mod constant este un avantaj al arhitecturii microserviciilor; orice constrângere ar trebui înălțaturată, la fel de mult cum dezvoltatorii rezolvă erori în aplicațiile lor.**
- **Creat pentru gestiunea posibilelor erori (Build for failure) - Dacă ceva poate merge prost, va merge prost (Murphy)** - Nu contează câte teste sunt realizate, câte alerte pot fi declanșate; dacă microserviciul „pică”, dezvoltatorii trebuie să ia în calcul acea posibilă eroare, să o trateze pe cât de elegant posibil și să definească cum se poate corecta (recovery). Când se proiecteză un microserviciu, se au în vedere următoarele arii:
 - **Upstream** = înțelegerea felului în care dezvoltatorii o să trimită sau nu notificări de eroare clienților, ținând totodată cont de evitarea cuplării
 - **Downstream** = cum vor gestiona dezvoltatorii defectarea unui microserviciu sau a unui sistem (precum o bază de date) de care depind
 - **Logging** = afișarea tuturor erorilor într-un fișier de log, ținând cont de cât de des se realizează salvarea acestor informații, de cantitatea de date și cum pot fi acestea accesate. De asemenea, trebuie luate în considerare și cazuri speciale, cum ar fi informații sensibile și implicații de performanță.
 - **Monitoring** = Monitorizarea trebuie să fie proiectată cu mare atenție. Este foarte dificil de gestionat o eroare fără informațiile potrivite în sistemele de monitorizare. Dezvoltatorii trebuie să determine ce elemente ale aplicației au informații semnificative.
 - **Alerting** = presupune înțelegerea cărora semnale pot indica faptul că ceva nu este în regulă, legătura semnalelor cu sistemul de monitorizare și logging-ul.
 - **Recovery** = proiectarea modului în care se revine (în urma unor erori) într-o stare normală. Revenirea automată (automatic recovery) este ideală, dar având în vedere că aceasta poate eșua, nu trebuie evitată revenirea manuală (manual recovery).
 - **Fallbacks** = Un mecanism bun de tratare a erorilor permite ca aplicația să funcționeze în continuare după apariția unei erori în sistem, în timp de dezvoltatorii lucrează să rezolve problema respectivă.
- **Scalabilitate (Scalability)** - Microserviciile trebuie să fie scalabile independent. Dacă este nevoie să se mărească numărul de cereri care poate fi gestionat, sau câte înregistrări pot fi stocate, acestea trebuie făcute în izolare. Se evită scalarea aplicației prin scalarea mai multor componente, impusă de o cuplare mare.
- **Automatizarea (Automation)** - Microserviciile trebuie proiectate ținând cont de lanțul specific CI/CD, de la construire și testare până la instalare și monitorizare. Modelul pentru dezvoltare/integrare continuă CI/CD trebuie proiectat de la începutul arhitecturii.

Domain-Driven Design (DDD)

Proiectarea bazată pe analiza domeniului reprezintă o manieră pentru dezvoltarea aplicațiilor complexe prin conectarea continuă a implementării la un model (care evoluează continuu) a conceptelor business de bază.

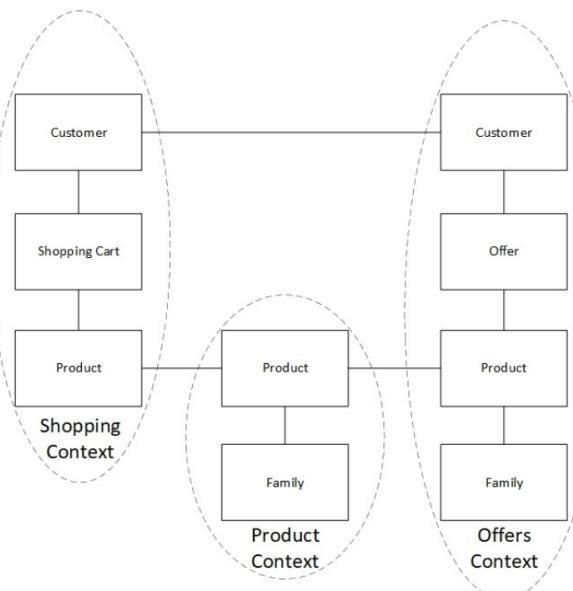
Premisele DDD:

- accentul principal al proiectului cade pe domeniul de bază (core domain) și pe logica domeniului (domain logic)
- Proiectele mai complexe trebuie bazate pe un model
- inițierea unei colaborări creative între experții tehnici și experții din domeniu pentru a ajunge din ce în ce mai aproape de modelul conceptual al problemei

Problema: când complexitatea scapă de sub control, software-ul nu mai poate fi înțeles suficient de bine pentru a putea fi schimbat sau extins cu ușurință. Dacă complexitatea domeniului nu este tratată în proiectare, nu contează că tehnologia infrastructurii suport este bine concepută.

Principiile de proiectare:

- **Context mărginit (Bounded context):** Când se abordează un sistem complex, de obicei se abstractizează într-un model care descrie aspectele diferite ale sistemului și cum poate fi folosit pentru a rezolva probleme. Când există mai multe modele, iar codul de bază al diferitelor modele este combinat, software-ul devine plin de erori (buggy), nesigur și greu de înțeles. În DDD, se definește contextul în care se aplică un model, se stabilesc explicit granițele în ceea ce privește organizarea echipei și utilizarea în anumite părți ale aplicației, păstrând modelul **consecvent** cu aceste limite.
- **Limbaj generic specific (Ubiquitous language):** În DDD trebuie alcătuit un limbaj comun și riguros între dezvoltatori și utilizatori. Acest limbaj trebuie să fie bazat pe modelul de domeniu, ajutând în a avea o conversație generală între toți experții din domeniu, acest lucru fiind esențial la abordarea testării.
- **Capturarea/maparea contextului (Context mapping):** Într-o aplicație de dimensiuni mari, proiectată pentru mai multe contexte mărginite (bounded contexts), se poate pierde vederea de ansamblu. Inevitabil, contextele mărginite vor fi nevoie să comunice date între ele. O mapare de context este o vedere de ansamblu (global view) asupra sistemului ca un întreg, care ilustrează maniera în care contextele mărginite ar trebui să comunice între ele.



Exemplu de mapare de context

Pentru mai multe detalii, vezi cartea „**Domain-Driven Design**” scrisă de **Eric Evans**, precum și comunitatea DDD: <https://dddcommunity.org/>.

Folosirea DDD în microservicii

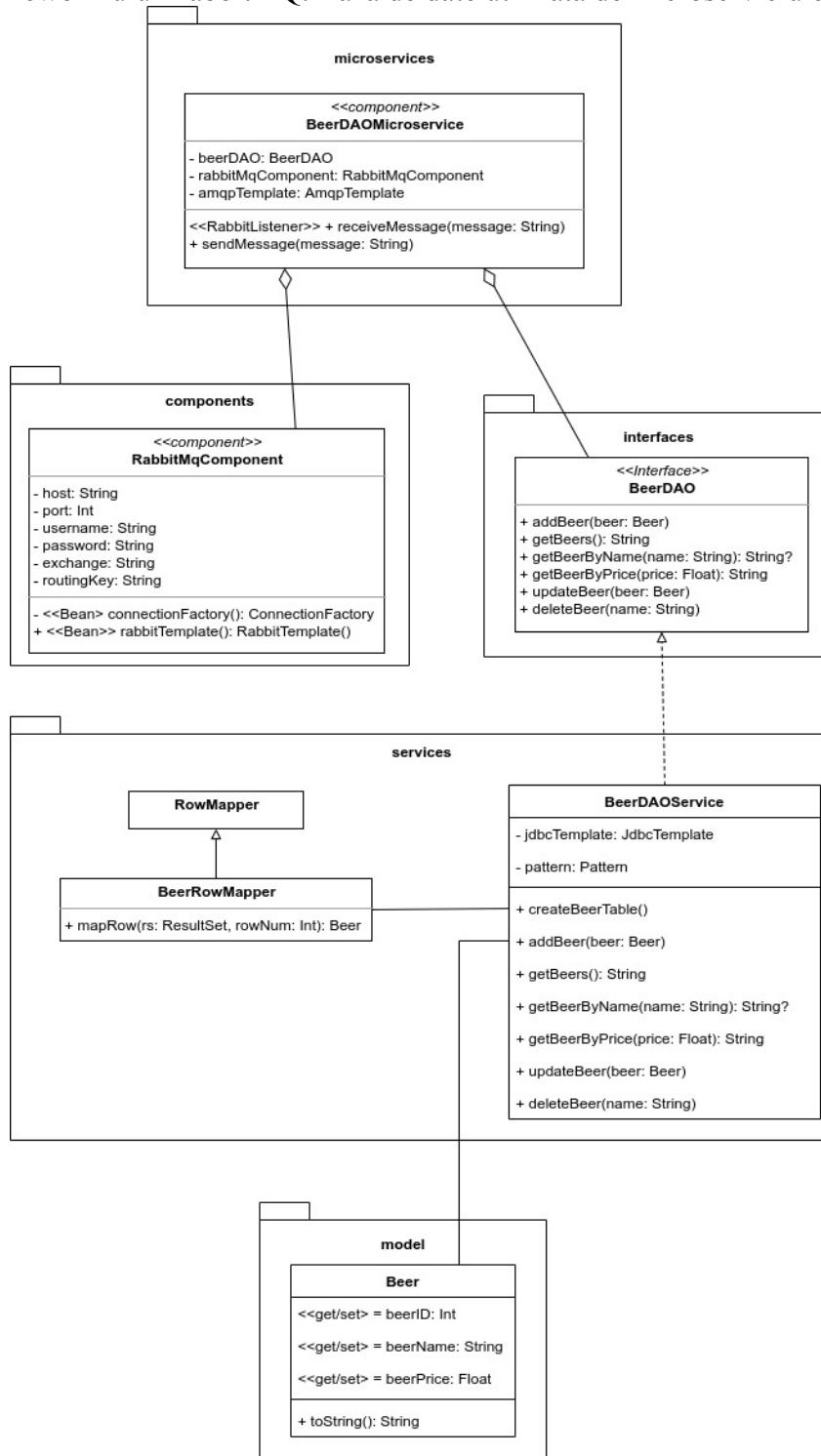
- **Bounded Context** - Nu trebuie creat un microserviciu care include mai mult de un context mărginit
- **Ubiquitous Language** - Dezvoltatorii trebuie să se asigure că maniera de comunicare utilizată este suficient de general valabil, astfel încât operațiile și interfețele care sunt expuse să fie exprimate utilizând limbajul domeniului context
- **Context Model** - Modelul utilizat de microserviciu trebuie definit într-un context mărginit și să folosească un limbaj generic (ubiquitous language), chiar și pentru entități care nu sunt expuse în nici o interfață pe care o oferă microserviciul
- **Context Mapping** - Trebuie examinat contextul mărginit al întregului sistem pentru a înțelege dependențele și cuplarea microserviciilor.

Exemple

Pentru un exemplu de microservicii în Kotlin cu framework-ul **Ktor**, se recomandă parcurgerea modelului de la adresa: <https://dzone.com/articles/kotlin-microservices-with-ktor>

Exemplul 1: *SQLiteExample*

Cerință: Să se creeze un microserviciu în Kotlin care să gestioneze o bază de date cu tipurile de bere. Aplicația Kotlin va comunica cu o interfață de tip CLI din python prin intermediul framework-ului RabbitMQ. Baza de date utilizată de microserviciu este SQLite3.



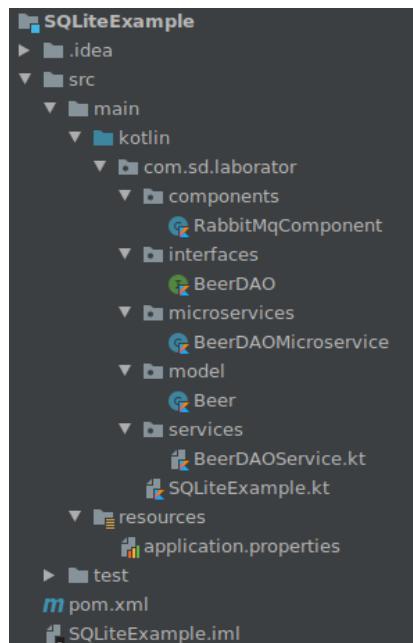
Arhitectura aplicației *SQLiteExample*

Exemplul 1: Configurări

Similar cu laboratorul 5 de la disciplina Sisteme Distribuite, se configurează următoarele cozi de mesaje, exchange-uri și chei de rutare în interfața RabbitMQ (localhost:15672):

- un exchange: **sqliteexample.direct**
- două cozi
 - **sqliteexample.queue**
 - **sqliteexample.queue1**
- două binding-uri:
 - **sqliteexample.queue** -> **sqliteexample.direct**, **sqliteexample.routingkey**
 - **sqliteexample.queue1** -> **sqliteexample.direct**, **sqliteexample.routingkey1**

Structura proiectului



Ierarhia aplicației SQLiteExample

Configurarea parametrilor din fișierul application.properties

```

spring.datasource.url=jdbc:sqlite:beer.db
spring.rabbitmq.host=localhost
spring.rabbitmq.port=5672
spring.rabbitmq.username=student
spring.rabbitmq.password=student
sqliteexample.rabbitmq.queue=sqliteexample.queue1
sqliteexample.rabbitmq.exchange=sqliteexample.direct
sqliteexample.rabbitmq.routingkey=sqliteexample.routingkey
  
```

Se remarcă parametrul **spring.datasource.url** care precizează faptul că jdbc-ul (java database connector) va utiliza o bază de date de tip sqlite denumită beer.db.

Crearea proiectului

Proiectul se creează conform instrucțiunilor din laboratorul 5 de la disciplina Sisteme Distribuite.

Se adaugă dependențele pentru sqlite în fișierul pom.xml:

```

<dependency>
  <groupId>org.springframework.boot</groupId>
  
```

Laborator 6

```
<artifactId>spring-boot-starter-jdbc</artifactId>
</dependency>
<dependency>
    <groupId>org.xerial</groupId>
    <artifactId>sqlite-jdbc</artifactId>
    <version>3.28.0</version>
</dependency>
```

Exemplul 1: Codul sursă

- **SQLiteExample.kt**

```
package com.sd.laborator

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

@SpringBootApplication
class SQLiteExample

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

- **Beer.kt**

```
package com.sd.laborator.model

class Beer(private var id: Int, private var name: String, private var price: Float) {

    var beerID: Int
        get() {
            return id
        }
        set(value) {
            id = value
        }
    var beerName: String
        get() {
            return name
        }
        set(value) {
            name = value
        }
    var beerPrice: Float
        get() {
            return price
        }
        set(value) {
            price = value
        }

    override fun toString(): String {
        return "Beer [id=$beerID, name=$beerName, price=$beerPrice]"
    }
}
```

- **BeerDAO.kt**

```
package com.sd.laborator.interfaces

import com.sd.laborator.model.Beer

interface BeerDAO {
    // Create
    fun createBeerTable()
    fun addBeer(beer: Beer)

    // Retrieve
    fun getBeers(): String
    fun getBeerByName(name: String): String?
    fun getBeerByPrice(price: Float): String?

    // Update
    fun updateBeer(beer: Beer)

    // Delete
    fun deleteBeer(name: String)
}
```

- **BeerDAOService.kt**

```
package com.sd.laborator.services

import com.sd.laborator.interfaces.BeerDAO
import com.sd.laborator.model.Beer
import org.springframework.beans.factory.annotation.Autowired
import org.springframework.jdbc.core.JdbcTemplate
import org.springframework.jdbc.core.RowMapper
import org.springframework.stereotype.Service
import java.sql.ResultSet
import java.sql.SQLException
import java.util.regex.Pattern

class BeerRowMapper : RowMapper<Beer?> {
    @Throws(SQLException::class)
    override fun mapRow(rs: ResultSet, rowNum: Int): Beer {
        return Beer(rs.getInt("id"), rs.getString("name"),
        rs.getFloat("price"))
    }
}

@Service
class BeerDAOService: BeerDAO {
    //Spring Boot will automatically wire this object using
    application.properties:
    @Autowired
    private lateinit var jdbcTemplate: JdbcTemplate
    var pattern: Pattern = Pattern.compile("\\W")

    override fun createBeerTable() {
        jdbcTemplate.execute("""CREATE TABLE IF NOT EXISTS beers(
            id INTEGER PRIMARY KEY
            AUTOINCREMENT,
            name TEXT,
            price REAL)""")
    }

    override fun addBeer(beer: Beer) {
        jdbcTemplate.update("INSERT INTO beers(name, price) VALUES (?, ?)", beer.name, beer.price)
    }

    override fun getBeers(): String {
        val results = jdbcTemplate.query("SELECT * FROM beers", RowMapper { rs: ResultSet, _: Int ->
            Beer(rs.getInt("id"), rs.getString("name"), rs.getFloat("price"))
        })
        return Gson().toJson(results)
    }

    override fun getBeerByName(name: String): Beer? {
        val result = jdbcTemplate.query("SELECT * FROM beers WHERE name = ?", RowMapper { rs: ResultSet, _: Int ->
            Beer(rs.getInt("id"), rs.getString("name"), rs.getFloat("price"))
        }, name)
        return if (result.size == 1) result[0] else null
    }

    override fun getBeerByPrice(price: Float): Beer? {
        val result = jdbcTemplate.query("SELECT * FROM beers WHERE price = ?", RowMapper { rs: ResultSet, _: Int ->
            Beer(rs.getInt("id"), rs.getString("name"), rs.getFloat("price"))
        }, price)
        return if (result.size == 1) result[0] else null
    }

    override fun updateBeer(beer: Beer) {
        jdbcTemplate.update("UPDATE beers SET name = ?, price = ? WHERE id = ?",
            beer.name, beer.price, beer.id)
    }

    override fun deleteBeer(name: String) {
        jdbcTemplate.update("DELETE FROM beers WHERE name = ?",
            name)
    }
}
```

Laborator 6

```
        name VARCHAR(100) UNIQUE,
        price FLOAT) """)
    }

    override fun addBeer(beer: Beer) {
        if(pattern.matcher(beer.beerName).find()) {
            println("SQL Injection for beer name")
            return
        }
        jdbcTemplate.update("INSERT INTO beers(name, price) VALUES (?, ?)", beer.beerName, beer.beerPrice)
    }

    override fun getBeers(): String {
        val result: MutableList<Beer?> = jdbcTemplate.query("SELECT * FROM beers", BeerRowMapper())
        var stringResult: String = ""
        for (item in result) {
            stringResult += item
        }
        return stringResult
    }

    override fun getBeerByName(name: String): String? {
        if(pattern.matcher(name).find()) {
            println("SQL Injection for beer name")
            return null
        }
        val result: Beer? = jdbcTemplate.queryForObject("SELECT * FROM beers WHERE name = '$name'", BeerRowMapper())
        return result.toString()
    }

    override fun getBeerByPrice(price: Float): String {
        val result: MutableList<Beer?> = jdbcTemplate.query("SELECT * FROM beers WHERE price <= $price", BeerRowMapper())
        var stringResult: String = ""
        for (item in result) {
            stringResult += item
        }
        return stringResult
    }

    override fun updateBeer(beer: Beer) {
        if(pattern.matcher(beer.beerName).find()) {
            println("SQL Injection for beer name")
            return
        }
        jdbcTemplate.update("UPDATE beers SET name = ?, price = ? WHERE id = ?", beer.beerName, beer.beerPrice, beer.beerID)
    }

    override fun deleteBeer(name: String) {
        if(pattern.matcher(name).find()) {
            println("SQL Injection for beer name")
            return
        }
    }
}
```

```

        jdbcTemplate.update("DELETE FROM beers WHERE name = ?", name)
    }
}

```

- **BeerDAOMicroservice.kt**

```

package com.sd.laborator.microservices

import com.sd.laborator.components.RabbitMqComponent
import com.sd.laborator.interfaces.BeerDAO
import com.sd.laborator.model.Beer
import org.springframework.amqp.core.AmqpTemplate
import org.springframework.rabbit.annotation.RabbitListener
import org.springframework.beans.factory.annotation.Autowired
import org.springframework.stereotype.Component

@Component
class BeerDAOMicroservice {
    @Autowired
    private lateinit var beerDAO: BeerDAO

    @Autowired
    private lateinit var rabbitMqComponent: RabbitMqComponent

    private lateinit var amqpTemplate: AmqpTemplate

    @Autowired
    fun initTemplate() {
        this.amqpTemplate = rabbitMqComponent.rabbitTemplate()
    }
    //citesc din queue1
    // scriu in queue
    @RabbitListener(queues = ["${sqliteexample.rabbitmq.queue}"])
    fun recieveMessage(msg: String) {
        val processed_msg = (msg.split(",")).map { it.toInt().toChar() }
    }.joinToString(separator="")
        val (operation, parameters) = processed_msg.split('~')
        var beer: Beer? = null
        var price: Float? = null
        var name: String? = null

        // id=1;name=Corona;price=3.6
        if("id=" in parameters) {
            println(parameters)
            val params: List<String> = parameters.split(';')
            try {
                beer = Beer(
                    params[0].split('=')[1].toInt(),
                    params[1].split('=')[1],
                    params[2].split('=')[1].toFloat()
                )
            } catch (e: Exception) {
                print("Error parsing the parameters: ")
                println(params)
                return
            }
        } else if ("price=" in parameters) {

```

Laborator 6

```
        price = parameters.split('=')[1].toFloat()
    } else if ("name=" in parameters) {
        name = parameters.split("=")[1]
    }
    println(parameters)
    println(name)
    println(price)
    println(beer)
    var result: Any? = when(operation) {
        "createBeerTable" -> beerDAO.createBeerTable()
        "addBeer" -> beerDAO.addBeer(beer!!)
        "getBeers" -> beerDAO.getBeers()
        "getBeerByName" -> beerDAO.getBeerByName(name!!)
        "getBeerByPrice" -> beerDAO.getBeerByPrice(price!!)
        "updateBeer" -> beerDAO.updateBeer(beer!!)
        "deleteBeer" -> beerDAO.deleteBeer(name!!)
        else -> null
    }
    println("result: ")
    println(result)
    if (result != null) sendMessage(result.toString())
}

fun sendMessage(msg: String) {
    println("message: ")
    println(msg)

this.amqpTemplate.convertAndSend(rabbitMqComponent.getExchange(),
rabitMqComponent.getRoutingKey(), msg)
}
}
```

• RabbitMqComponent

```
package com.sd.laborator.components

import org.springframework.amqp.rabbit.connection.CachingConnectionFactory
import org.springframework.amqp.rabbit.connection.ConnectionFactory
import org.springframework.amqp.rabbit.core.RabbitTemplate
import org.springframework.beans.factory.annotation.Value
import org.springframework.context.annotation.Bean
import org.springframework.stereotype.Component

@Component
class RabbitMqComponent {
    @Value("\${spring.rabbitmq.host}")
    private lateinit var host: String
    @Value("\${spring.rabbitmq.port}")
    private val port: Int = 0
    @Value("\${spring.rabbitmq.username}")
    private lateinit var username: String
    @Value("\${spring.rabbitmq.password}")
    private lateinit var password: String
    @Value("\${sqliteexample.rabbitmq.exchange}")
    private lateinit var exchange: String
    @Value("\${sqliteexample.rabbitmq.routingkey}")
}
```

```

private lateinit var routingKey: String

fun getExchange(): String = this.exchange

fun getRoutingKey(): String = this.routingKey

@Bean
private fun connectionFactory(): ConnectionFactory {
    val connectionFactory = CachingConnectionFactory()
    connectionFactory.host = this.host
    connectionFactory.username = this.username
    connectionFactory.setPassword(this.password)
    connectionFactory.port = this.port
    return connectionFactory
}

@Bean
fun rabbitTemplate(): RabbitTemplate =
RabbitTemplate(connectionFactory())
}

```

Aplicația python

```

import pika
from retry import retry

class RabbitMq:
    config = {
        'host': '0.0.0.0',
        'port': 5678,
        'username': 'student',
        'password': 'student',
        'exchange': 'sqliteexample.direct',
        'routing_key': 'sqliteexample.routingkey1',
        'queue': 'sqliteexample.queue'
    }
    credentials = pika.PlainCredentials(config['username'],
                                         config['password'])
    parameters = (pika.ConnectionParameters(host=config['host']),
                  pika.ConnectionParameters(port=config['port']),
                  pika.ConnectionParameters(credentials=credentials))

    def on_received_message(self, blocking_channel,
                           deliver, properties, message):
        result = message.decode('utf-8')
        blocking_channel.confirm_delivery()
        try:
            print(result)
        except Exception:
            print("wrong data format")
        finally:
            blocking_channel.stop_consuming()

    @retry(pika.exceptions.AMQPConnectionError, delay=5, jitter=(1,
3))

```

Laborator 6

```
def receive_message(self):
    # automatically close the connection
    with pika.BlockingConnection(self.parameters) as connection:
        # automatically close the channel
        with connection.channel() as channel:
            channel.basic_consume(self.config['queue'],
                                  self.on_received_message)
    try:
        channel.start_consuming()
    # Don't recover connections closed by server
    except pika.exceptions.ConnectionClosedByBroker:
        print("Connection closed by broker.")
    # Don't recover on channel errors
    except pika.exceptions.AMQPChannelError:
        print("AMQP Channel Error")
    # Don't recover from KeyboardInterrupt
    except KeyboardInterrupt:
        print("Application closed.")

def send_message(self, message):
    # automatically close the connection
    with pika.BlockingConnection(self.parameters) as connection:
        # automatically close the channel
        with connection.channel() as channel:
            self.clear_queue(channel)
            channel.basic_publish(
                exchange=self.config['exchange'],
                routing_key=self.config['routing_key'],
                body=message)

def clear_queue(self, channel):
    channel.queue_purge(self.config['queue'])

def print_menu():
    print('0 --> Exit program')
    print('1 --> addBeer')
    print('2 --> getBeers')
    print('3 --> getBeerByName')
    print('4 --> getBeerByPrice')
    print('5 --> updateBeer')
    print('6 --> deleteBeer')
    return input("Option=")

if __name__ == '__main__':
    rabbit_mq = RabbitMq()
    rabbit_mq.send_message("createBeerTable~")
    while True:
        option = print_menu()
        if option == '0':
            break
        elif option == '1':
            name = input("Beer name: ")
            price = float(input("Beer price: "))
            rabbit_mq.send_message(
                "addBeer~id=-1;name={};price={}".format(name, price))
```

```

        elif option == '2':
            rabbit_mq.send_message("getBeers~")
            rabbit_mq.receive_message()
        elif option == '3':
            name = input("Beer name: ")
            rabbit_mq.send_message(
                "getBeerByName~name={ }".format(name))
            rabbit_mq.receive_message()
        elif option == '4':
            price = float(input("Beer price: "))
            rabbit_mq.send_message(
                "getBeerByPrice~price={ }".format(price))
            rabbit_mq.receive_message()
        elif option == '5':
            id = int(input("Beer ID: "))
            name = input("Beer name: ")
            price = float(input("Beer price: "))
            rabbit_mq.send_message(
                "updateBeer~id={};name={};price={ }".format(id, name,
                                                price))
            rabbit_mq.receive_message()
        elif option == '6':
            name = input("Beer name: ")
            rabbit_mq.send_message("deleteBeer~name={ }".format(name))
        else:
            print("Invalid option")
    
```

Pentru testarea exemplului, din IntelliJ se execută **Maven -> Lifecycle -> clean + compile + package**. La final, se deschide directorul **target** creat și se execută în terminal comanda:

```
java -jar SQLiteExample-1.0.0.jar
```

Pentru a porni interfața CLI, se execută într-un terminal deschis în directorul cu aplicația python comenziile:

```

python3 -m venv env
source env/bin/activate
pip3 install pika==1.1.0 retry==0.9.2
python3 sqlite_example.py
    
```

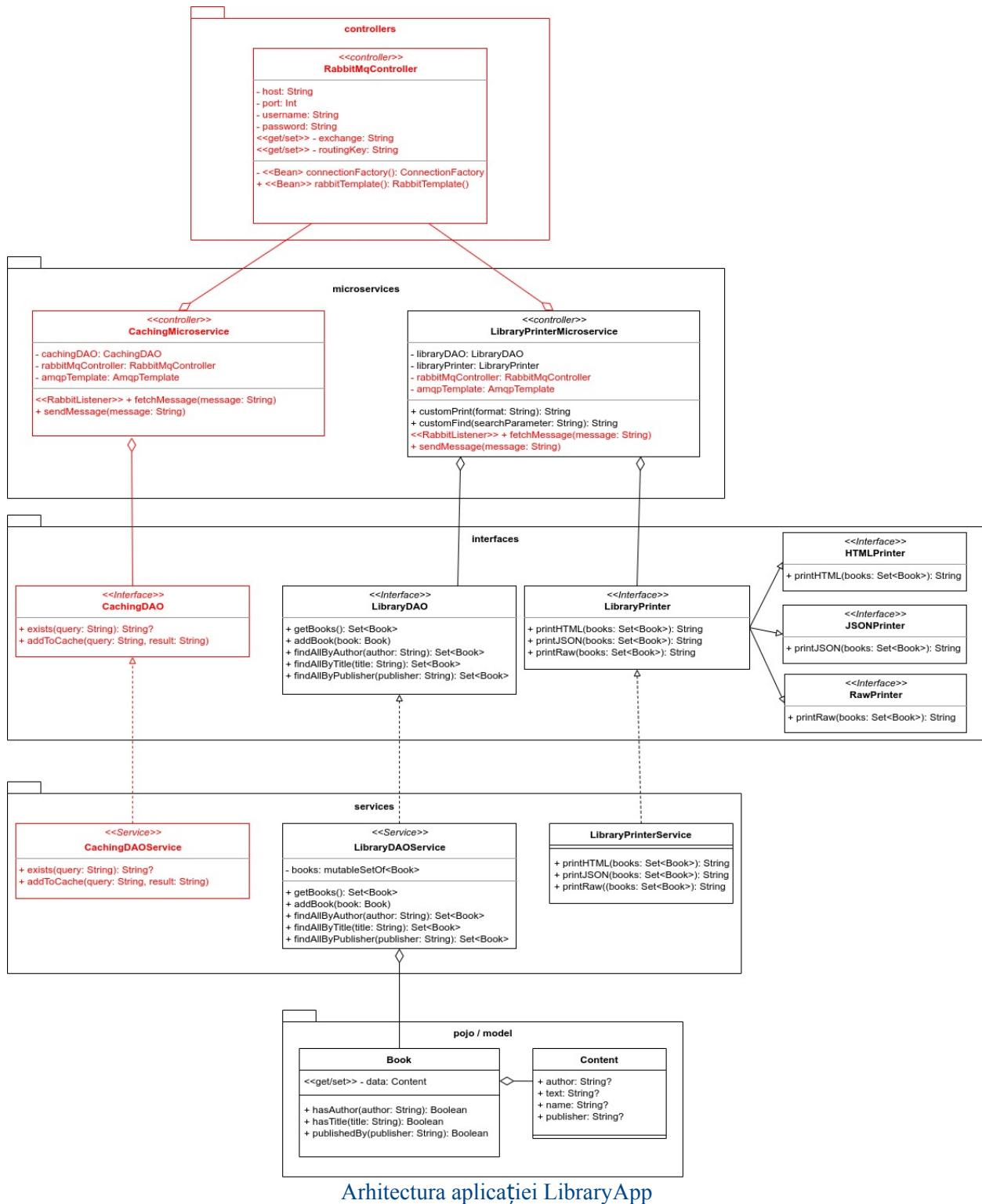
Exemplul 2: LibraryApp

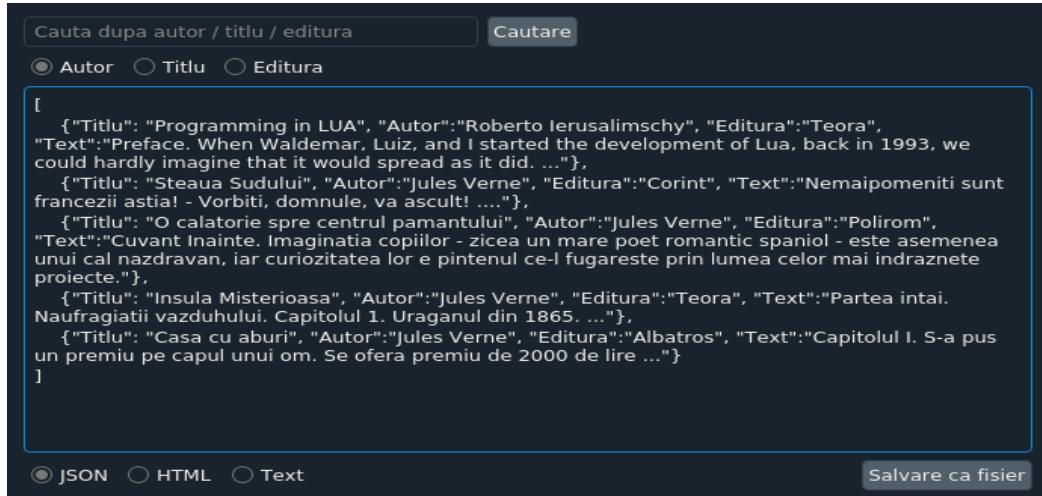
Cerință: Să se scrie un program Kotlin care să realizeze prin intermediul unui microserviciu WEB gestiunea unei biblioteci utilizând principiile SOLID. Aplicația va conține trei moduri de afișare a datelor (HTML, JSON și Raw) și va expune utilizatorului prin interfață funcționalități de tip CRUD (Create, Retrieve, Update, Delete).

Arhitectura este reprezentată în diagrama de clase de mai jos. Spre deosebire de exemplul anterior, microserviciul din LibraryApp va fi de tip Web, epunând către interfața din python metodele de afișare ale bibliotecii.

Observație: componentele cu roșu din diagrama de mai jos nu se regăsesc în exemplul curent. Acestea trebuie implementate în tema pe acasă.

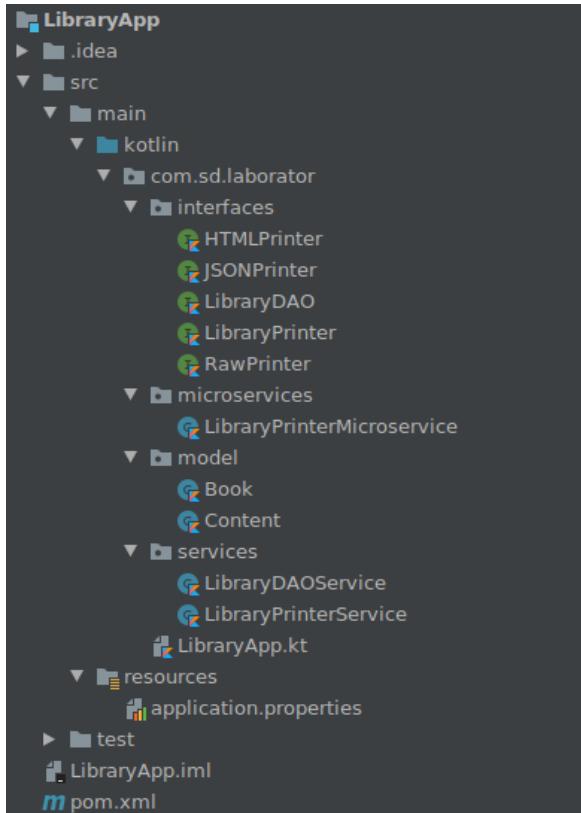
Laborator 6





Interfață grafică pentru LibraryApp realizată cu PyQt5

Structura proiectului



Ierarhia aplicației LibraryApp

Crearea proiectului

Proiectul se creează similar cu exemplul anterior. La final, se mai adaugă dependența de **spring-boot-starter-web** în fișierul pom.xml:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

Exemplul 2: Codul sursă

- **LibraryApp.kt**

```
package com.sd.laborator

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

@SpringBootApplication
class LibraryApp

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

- **Content.kt**

```
package com.sd.laborator.model

data class Content(var author: String?, var text: String?, var name: String?, var publisher: String?)
```

- **Book.kt**

```
package com.sd.laborator.model

class Book(private var data: Content) {

    var name: String?
        get() {
            return data.name
        }
        set(value) {
            data.name = value
        }

    var author: String?
        get() {
            return data.author
        }
        set(value) {
            data.author = value
        }

    var publisher: String?
        get() {
            return data.publisher
        }
        set(value) {
            data.publisher = value
        }

    var content: String?
        get() {
            return data.text
        }
}
```

```

    set(value) {
        data.text = value
    }

    fun hasAuthor(author: String): Boolean {
        return data.author.equals(author)
    }

    fun hasTitle(title: String): Boolean {
        return data.name.equals(title)
    }

    fun publishedBy(publisher: String): Boolean {
        return data.publisher.equals(publisher)
    }
}

```

- **LibraryPrinterService.kt**

```

package com.sd.laborator.services

import com.sd.laborator.interfaces.LibraryPrinter
import com.sd.laborator.model.Book
import org.springframework.stereotype.Service

@Service
class LibraryPrinterService: LibraryPrinter {
    override fun printHTML(books: Set<Book>): String {
        var content: String = "<html><head><title>Libraria mea
HTML</title></head><body>"
        books.forEach {
            content +=
                "<p><h3>${it.name}</h3><h4>${it.author}</h4><h5>${it.publisher}</h5>${it.content}<br/>"
        }
        content += "</body></html>"
        return content
    }

    override fun printJSON(books: Set<Book>): String {
        var content: String = "[\n"
        books.forEach {
            if (it != books.last())
                content += "    {\"Titlu\": \"${it.name}\",
\"Autor\": \"${it.author}\", \"Editura\": \"${it.publisher}\",
\"Text\": \"${it.content}\",\n"
            else
                content += "    {\"Titlu\": \"${it.name}\",
\"Autor\": \"${it.author}\", \"Editura\": \"${it.publisher}\",
\"Text\": \"${it.content}\",\n"
        }
        content += "] \n"
        return content
    }

    override fun printRaw(books: Set<Book>): String {
        var content: String = ""

```

Laborator 6

```
        books.forEach {
            content +=
        "${it.name}\n${it.author}\n${it.publisher}\n${it.content}\n\n"
    }
    return content
}
}
```

- **LibraryDAOService.kt**

```
package com.sd.laborator.services

import com.sd.laborator.interfaces.LibraryDAO
import com.sd.laborator.model.Book
import com.sd.laborator.model.Content
import org.springframework.stereotype.Service

@Service
class LibraryDAOService: LibraryDAO {
    private var books: MutableSet<Book> = mutableSetOf(
        Book(Content("Roberto Ierusalimschy", "Preface. When Waldemar,
Luiz, and I started the development of Lua, back in 1993, we could
hardly imagine that it would spread as it did. ...", "Programming in
LUA", "Teora")),
        Book(Content("Jules Verne", "Nemaipomeniti sunt francezii
astia! - Vorbiti, domnule, va ascult! ....", "Steaua
Sudului", "Corint")),
        Book(Content("Jules Verne", "Cuvant Inainte. Imaginatia
copiilor - zicea un mare poet romantic spaniol - este asemenea unui
cal nazdravan, iar curiozitatea lor e pintenul ce-l fugareste prin
lumea celor mai indraznute proiecte.", "O calatorie spre centrul
pamantului", "Polirom")),
        Book(Content("Jules Verne", "Partea intai. Naufragiatii
vazduhului. Capitolul 1. Uraganul din 1865. ...", "Insula
Misterioasa", "Teora")),
        Book(Content("Jules Verne", "Capitolul I. S-a pus un premiu pe
capul unui om. Se ofera premiu de 2000 de lire ...", "Casa cu
aburi", "Albatros"))
    )
    override fun getBooks(): Set<Book> {
        return this.books
    }

    override fun addBook(book: Book) {
        this.books.add(book)
    }

    override fun findAllByAuthor(author: String): Set<Book> {
        return (this.books.filter { it.hasAuthor(author) }).toSet()
    }

    override fun findAllByTitle(title: String): Set<Book> {
        return (this.books.filter { it.hasTitle(title) }).toSet()
    }

    override fun findAllByPublisher(publisher: String): Set<Book> {
        return (this.books.filter { it.publishedBy(publisher)
    }
}
```

```
}).toSet()
    }
}
```

- **HTMLPrinter**

```
package com.sd.laborator.interfaces

import com.sd.laborator.model.Book

interface HTMLPrinter {
    fun printHTML(books: Set<Book>): String
}
```

- **JSONPrinter**

```
package com.sd.laborator.interfaces

import com.sd.laborator.model.Book

interface JSONPrinter {
    fun printJSON(books: Set<Book>): String
}
```

- **LibraryDAO**

```
package com.sd.laborator.interfaces

import com.sd.laborator.model.Book

interface LibraryDAO {
    fun getBooks(): Set<Book>
    fun addBook(book: Book)
    fun findAllByAuthor(author: String): Set<Book>
    fun findAllByTitle(title: String): Set<Book>
    fun findAllByPublisher(publisher: String): Set<Book>
}
```

- **LibraryPrinter**

```
package com.sd.laborator.interfaces

interface LibraryPrinter: HTMLPrinter, JSONPrinter, RawPrinter
```

- **RawPrinter**

```
package com.sd.laborator.interfaces

import com.sd.laborator.model.Book

interface RawPrinter {
    fun printRaw(books: Set<Book>): String
}
```

- **LibraryPrinterMicroservice**

Se observă faptul că microserviciul are acces prin intermediul unui LibraryDAO la baza de date (pe moment, la datele hardcodate) și totodată la funcționalitățile de printare ale LibraryPrinter. Funcționalitățile microserviciului WEB sunt expuse la următoarele URL-uri:

Laborator 6

```
» http://localhost:8080/print?format=html
» http://localhost:8080/find?author=Jules%20Verne
» http://localhost:8080/find?title=Steaua%20Sudului
» http://localhost:8080/find?publisher=Corint
```

```
package com.sd.laborator.microservices

import com.sd.laborator.interfaces.LibraryDAO
import com.sd.laborator.interfaces.LibraryPrinter
import org.springframework.beans.factory.annotation.Autowired
import org.springframework.stereotype.Controller
import org.springframework.web.bind.annotation.RequestMapping
import org.springframework.web.bind.annotation.RequestMethod
import org.springframework.web.bind.annotation.RequestParam
import org.springframework.web.bind.annotation.ResponseBody

@Controller
class LibraryPrinterMicroservice {
    @Autowired
    private lateinit var libraryDAO: LibraryDAO
    @Autowired
    private lateinit var libraryPrinter: LibraryPrinter

    @RequestMapping("/print", method = [RequestMethod.GET])
    @ResponseBody
    fun customPrint(@RequestParam(required = true, name = "format",
defaultValue = "") format: String): String {
        return when(format) {
            "html" -> libraryPrinter.printHTML(libraryDAO.getBooks())
            "json" -> libraryPrinter.printJSON(libraryDAO.getBooks())
            "raw" -> libraryPrinter.printRaw(libraryDAO.getBooks())
            else -> "Not implemented"
        }
    }

    @RequestMapping("/find", method = [RequestMethod.GET])
    @ResponseBody
    fun customFind(@RequestParam(required = false, name = "author",
defaultValue = "") author: String,
                @RequestParam(required = false, name = "title",
defaultValue = "") title: String,
                @RequestParam(required = false, name = "publisher",
defaultValue = "") publisher: String): String {
        if (author != "") {
            return
        }
        this.libraryPrinter.printJSON(this.libraryDAO.findAllByAuthor(author))
        if (title != "") {
            return
        }
        this.libraryPrinter.printJSON(this.libraryDAO.findAllByTitle(title))
        if (publisher != "") {
            return
        }
        this.libraryPrinter.printJSON(this.libraryDAO.findAllByPublisher(pu-
blisher))
        return "Not a valid field"
    }
}
```

Aplicația python

În codul de mai jos, se remarcă că, spre deosebire de aplicația din laboratorul precedent în care comunicația era realizată prin cozi de mesaje, exemplul curent utilizează modulul *requests* trimițând cereri HTTP de tip GET către microserviciul Web din Kotlin.

```

import os
import sys
import requests
import qdarkstyle
from requests.exceptions import HTTPError
from PyQt5.QtWidgets import (QWidget, QApplication, QFileDialog,
                             QMessageBox)
from PyQt5 import QtCore
from PyQt5.uic import loadUi

def debug_trace(ui=None):
    from pdb import set_trace
    QtCore.pyqtRemoveInputHook()
    set_trace()
    # QtCore.pyqtRestoreInputHook()

class LibraryApp(QWidget):
    ROOT_DIR = os.path.dirname(os.path.abspath(__file__))

    def __init__(self):
        super(LibraryApp, self).__init__()
        ui_path = os.path.join(self.ROOT_DIR, 'library_manager.ui')
        loadUi(ui_path, self)
        self.search_btn.clicked.connect(self.search)
        self.save_as_file.clicked.connect(self.save_as_file)

    def search(self):
        search_string = self.search_bar.text()
        url = None
        if not search_string:
            if self.json_rb.isChecked():
                url = '/print?format=json'
            elif self.html_rb.isChecked():
                url = '/print?format=html'
            else:
                url = '/print?format=raw'
        else:
            if self.author_rb.isChecked():
                url = '/find?author={}'.format(
                    search_string.replace(' ', '%20'))
            elif self.title_rb.isChecked():
                url = '/find?title={}'.format(
                    search_string.replace(' ', '%20'))
            else:
                url = '/find?publisher={}'.format(
                    search_string.replace(' ', '%20'))
        full_url = "http://localhost:8080" + url
        try:
            response = requests.get(full_url)

```

Laborator 6

```
        self.result.setText(response.content.decode('utf-8'))
    except HTTPError as http_err:
        print('HTTP error occurred: {}'.format(http_err))
    except Exception as err:
        print('Other error occurred: {}'.format(err))

    def save_as_file(self):
        options = QFileDialog.Options()
        options |= QFileDialog.DontUseNativeDialog
        file_path = str(
            QFileDialog.getSaveFileName(self,
                                        'Salvare fisier',
                                        options=options))

        if file_path:
            file_path = file_path.split("//") [1]
            if not file_path.endswith('.json') and not
file_path.endswith(
                '.html') and not file_path.endswith('.txt'):
                if self.json_rb.isChecked():
                    file_path += '.json'
                elif self.html_rb.isChecked():
                    file_path += '.html'
                else:
                    file_path += '.txt'
        try:
            with open(file_path, 'w') as fp:
                if file_path.endswith(".html"):
                    fp.write(self.result.toHtml())
                else:
                    fp.write(self.result.toPlainText())
        except Exception as e:
            print(e)
            QMessageBox.warning(self, 'Library Manager',
                                'Nu s-a putut salva fisierul')

    if __name__ == '__main__':
        app = QApplication(sys.argv)

        stylesheet = qdarkstyle.load_stylesheet_pyqt5()
        app.setStyleSheet(stylesheet)

        window = LibraryApp()
        window.show()
        sys.exit(app.exec_())
```

Pentru testarea exemplului, din IntelliJ se execută **Maven -> Lifecycle -> clean + compile**, apoi **Maven -> Plugins -> spring-boot -> spring-boot:run**.

Pentru a porni interfața grafică, se execută într-un terminal deschis în directorul cu aplicația python comenziile:

```
python3 -m venv env
source env/bin/activate
pip3 install -r requirements.txt
python3 library_manager.py
```

Aplicații și teme

Aplicații de laborator:

- Să se reimplementeze stocarea datelor în exemplul LibraryApp utilizând o bază de date SQLite (ca în exemplul 1), având tabela **Book** în diagrama E-R de mai jos.

Cache		Book	
PK	id: INTEGER	PK	id: INTEGER
UK	timestamp: INTEGER query: VARCHAR result: VARCHAR	UK	author: VARCHAR title: VARCHAR publisher: VARCHAR text: TEXT

Diagrama Entitate-Relație

- Să se combine funcționalitatea de căutare cu cea de afișare într-un anumit format (HTML, JSON, raw) sub forma unui nou end-point accesibil printr-un request HTTP de tip GET la un URL: <http://localhost:8080/find?author=<author-name>/print?format=json>

Teme pe acasă:

- Să se implementeze un mecanism de caching care stochează în baza de date interogarea utilizatorului (dacă nu există deja în baza de date), rezultatul căutării pe baza interogării și timestamp-ul căutării. Vezi diagrama UML de clase, precum și diagrama E-R de mai sus pentru mai multe detalii despre implementare. Microserviciul **CachingMicroservice** trebuie să comunice prin intermediul a două cozi de mesaje cu **LibraryPrinterMicroservice**. Astfel, la fiecare interogare introdusă de utilizator, se verifică întâi cache-ul. În cazul unui HIT (interrogarea a mai fost introdusă anterior), dacă timestamp-ul nu este mai vechi de o oră, se ia rezultatul din cache. Dacă timestamp-ul depășește intervalul de o oră sau în cazul unui MISS, se realizează căutarea propriu-zisă, iar rezultatul este actualizat/scris în cache.

Observație: Pentru simplitate, CachingMicroservice a fost introdus în cadrul aceluiași proiect. Totuși, pentru a respecta principiile de proiectare ale microserviciilor, trebuie creat un proiect nou pentru acesta, la final rezultând un fișier jar pentru fiecare microserviciu.

Sisteme Distribuite - Laborator 7

Microservicii reactive cu Kotlin

Descriere generală

Microserviciile reactive se bazează pe **paradigma de programare reactivă**, ținând lor fiind să ofere receptivitate (engl. *responsiveness*), rigiditate (engl. *resiliency*) și orientare pe mesaje (engl. *message-driven*).

Diferența față de programarea imperativă

Într-o aplicație scrisă utilizând paradigma imperativă, dezvoltatorul scrie codul astfel încât o instrucțiune cere un anumit lucru, apoi așteaptă rezultatul a ceea ce a cerut. În timpul în care rezultatul este așteptat, aplicația este **blocată**.

Să exemplificăm printr-un bloc de cod Kotlin:

```
var x = computePI(nrZecimale = 10000)
println("Numarul PI cu 10.000 de zecimale este: $x")
```

Se observă că pe prima linie, se cere calculul numărului π cu 10.000 de zecimale. Rezultatul apelului funcției **computePI()** nu este disponibil imediat, ci doar după un anumit timp (în funcție de parametrul trimis). Așadar, variabila **x** nu va fi populată instant cu valoarea returnată de funcție, deci programul se va **bloca pe linia 1** până când rezultatul produs de funcție va fi disponibil pentru atribuire. Abia după aceea se va executa și linia a 2-a, care are nevoie de variabila **x** pentru afișare.

Cea mai importantă parte a acestui bloc de cod este faptul că programul Kotlin se blochează până datele de care are nevoie sunt disponibile complet, deci spunem că metoda **computePI** este **blocantă**.

Varianta utilizând programarea reactivă este următoarea:

```
subscribe(::computePI, args= arrayOf(10000)).whenDone(::println)
```

(exemplul de mai sus este didactic, implementarea efectivă diferă!)

Semnificația liniei de cod de mai sus este următoarea: programul se înscrive (engl. *subscribes*) unei metode, iar când execuția acelei metode este încheiată, rezultatul este trimis unei alte metode. În acest exemplu, când numărul PI este calculat, se apelează metoda **println()** și se afișează rezultatul.

Semnificativ pentru cel de-al doilea exemplu este faptul că după instrucțiunea de mai sus, programul își continuă execuția, deci va putea rula alte instrucțiuni în continuare. Acest tip de instrucțiune se numește **neblocantă**.

Fluxul de date reactiv

De asemenea, programul se poate înscrive (*subscribe*) nu numai unui singur rezultat, ci unui întreg flux de date (engl. *data stream*) rezultate din anumite prelucrări. Când fluxul de date își începe emisia, metoda la care programul s-a înscris va fi apelată progresiv, pentru datele din flux. În acest caz, dacă vom considera fluxul de date:

```
10, 123, 500, 593, 1026, 1043, 493, 209, 4000, 412000, ...
```

aplicația va primi aceste numere întregi pe rând și va apela metoda **computePI()** progresiv, cu câte un parametru preluat din fluxul de date. Când calculul se încheie pentru un anumit parametru, se apelează metoda **println()** care afișează rezultatul obținut conform parametrului

current:

1. S-a primit numărul **10** în fluxul de date → apelez `computePI(10)` → s-a încheiat execuția `computePI(10)`, apelez `println(resultat)`
 2. S-a primit numărul **123** în fluxul de date → apelez `computePI(123)` → s-a încheiat execuția `computePI(123)`, apelez `println(resultat)`
 3. S-a primit numărul **500** în fluxul de date → apelez `computePI(500)` → s-a încheiat execuția `computePI(500)`, apelez `println(resultat)`
- ...
- §.a.m.d.

Așadar, un **flux de date reactiv** este o colecție de date emisă încontinuu, pe măsură ce datele sunt pregătite. De exemplu, în loc să cereți anumite câmpuri dintr-o bază de date și să așteptați rezultatul, baza de date începe să trimită rezultate pe măsură ce acestea sunt pregătite.

Acest nou model de programare permite obținerea de performanțe mai mari în aplicațiile dezvoltate, deoarece se pot procesa mai multe cereri decât în modelul tradițional, blocant. Această abordare folosește resursele mult mai eficiente și acest lucru ar putea reduce inclusiv necesarul de infrastructură pentru aplicații.

Principiile programării reactive

- **receptivitatea** (*responsiveness*) - aplicațiile moderne ar trebui să răspundă cererilor în timp util, dar nu numai utilizatorilor care le utilizează, ci și rezolvarea problemelor și recuperarea după apariția erorilor trebuie să se conformeze constrângерilor de timp;
- **rigiditatea** (*resilience*) - realizabilă prin replicare, care la rândul ei depinde de scalabilitatea sistemului
- **elasticitatea** (*elasticity*) - sistemele reactive trebuie să fie elastice, astfel încât să se poată adapta sub diverse grade de încărcare (exemplu: număr mare de cereri), scalând resursele disponibile în funcție de nevoie
- **orientare spre mesaje** (*message-driven*) - sistemele reactive folosesc mesaje asincrone pentru a transmite informația prin diverse componente, având cuplare foarte slabă ce permite interconectarea acestor sisteme în izolare
- **supra-saturarea fluxurilor** (*back-pressure*) - se produce atunci când un sistem reactiv publică mesaje într-un ritm mai alert decât pot fi gestionate de entitățile înscrise pentru a primi mesajele.

Pentru mai multe detalii despre programarea reactivă și principiile sale, consultați „The Reactive Manifesto”, disponibil la următorul URL:

<https://github.com/reactivemanifesto/reactivemanifesto>

Biblioteca RxKotlin

În acest laborator veți utiliza biblioteca **RxKotlin** pentru implementarea microserviciilor reactive.

RxKotlin este o bibliotecă care extinde **RxJava**, adăugând funcționalități noi și metode de tip extensie care sporesc productivitatea lucrului cu programarea reactivă în Kotlin.

RxJava este o bibliotecă utilizată pentru lucrul cu cod asincron, bazat pe evenimente, care folosește secvențe observabile și operatori în stil funcțional, permitând execuții parametrizate prin planificatori (engl. *schedulers*).

Observables

Un obiect **Observable** este un fel de secvență de valori (mesaje), cu câteva proprietăți

speciale. Una din ele, poate chiar cea mai importantă, este că secvența este asincronă. Obiectele **Observable** produc evenimente, proces denumit ca și **emitere** (engl. *emitting*), într-o anumită perioadă de timp. Evenimentele pot conține valori, precum numere, sau instanțe ale unui tip personalizat de date. De asemenea, evenimentele pot fi rezultatul unor gesturi, cum ar fi apăsarea de butoane.

Un obiect **Observable** emite câte un eveniment (**next**), până când:

- se produce o eroare, și atunci se emite **error**, iar **Observable**-ul se încheie
- se încheie fluxul de date, și atunci se emite un eveniment de tip **complete**

Exemplu simplu de obiect **Observable**:

```
val observable = Observable.fromIterable(listOf(1, 2, 3))
```

Acest obiect va putea emite date de tip număr întreg, într-un flux de 3 elemente. Dacă nu există niciun client înscris la acest flux (engl. *subscriber*), nu se emite nicio valoare. Așadar, se creează un *subscriber*, astfel:

```
observable.subscribeBy (onNext = {
    println(it)
}, onComplete = {
    println("Completed!")
}, onError = {
    println("Error: $it!")
})
```

Codul de mai sus afișează:

```
1
2
3
Completed!
```

Constructorul de flux reactiv **fromIterable** este un exemplu cu care se poate prelua un flux de date dintr-o colecție existentă. Metoda **subscribeBy()** primește ca parametrii 3 funcții lambda corespunzătoare celor 3 evenimente posibile emise din fluxul reactiv: **onNext**, **onComplete** și **onError**.

Un alt exemplu de flux reactiv predefinit:

```
val observable = Observable.range(1, 10)

observable.subscribe{
    val n = it.toDouble()
    val fibonacci = ((Math.pow(1.61803, n) - Math.pow(0.61803, n)) /
        2.23606).roundToInt()
    println(fibonacci)
}
```

În cazul acestui flux, *subscriber*-ul este interesat doar de emiterea următorului element, nu și de cazurile de eroare, respectiv de evenimentul emis când fluxul s-a încheiat. Metoda **subscribe()** primește o funcție lambda care tratează evenimentul **next()**.

În practică, se folosesc fluxuri definite personalizat, în funcție de caz.

Exemplu de creare a unui flux reactiv personalizat:

```
val someErrorHappened = True
val observable = Observable.create<String> { emitter ->
```

```

        emitter.onNext("1")
        emitter.onNext("2")
        if (someErrorHappened)
            emitter.onError(RuntimeException("Error"))
        else
            emitter.onComplete()
    }

val subscription = observable.subscribeBy (
    onNext = { println(it) },
    onComplete = { println("Completed") },
    onError = { println(it) }
)

subscription.dispose()

```

În acest exemplu se creează un obiect **Observable** personalizat care emite 2 siruri de caractere în flux, iar apoi, în funcție de o condiție, emite eroare sau termină fluxul în mod obișnuit. De asemenea, obiectul **Subscription** returnat de metoda **subscribeBy** este capturat și utilizat pentru a elibera memoria utilizată, prin apelul metodei **dispose()** asupra acestuia.

Subscriber-ul la acest flux va primi următoarele:

```

1
2
java.lang.RuntimeException: Error

```

Pentru alte tipuri de obiecte corespunzătoare paradigmiei reactive, precum și alte exemple, consultați cartea **Reactive Programming with Kotlin**, de Alex Sullivan.

Aplicație exemplu - Okazii

Se cere implementarea unei aplicații bazate pe microservicii reactive, care să modeleze comportamentul ofertanților participanți la o licitație, împreună cu procesarea tuturor ofertelor primite și calcularea rezultatului.

Diagrama de microservicii

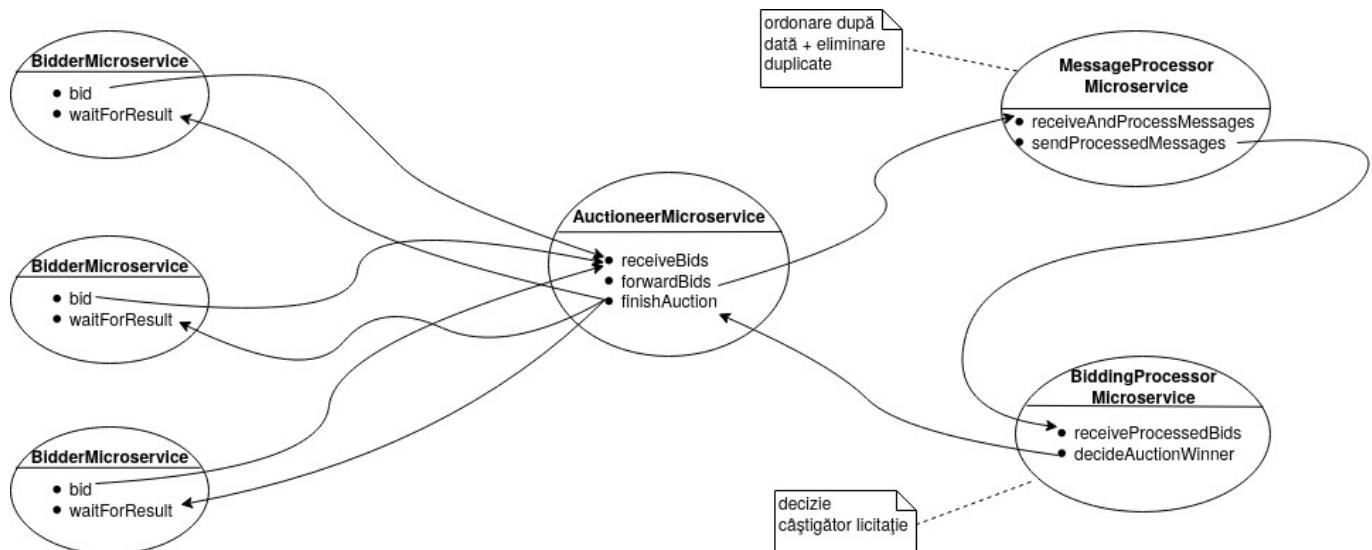


Figura 1 - Diagrama de microservicii

Diagrama de clase

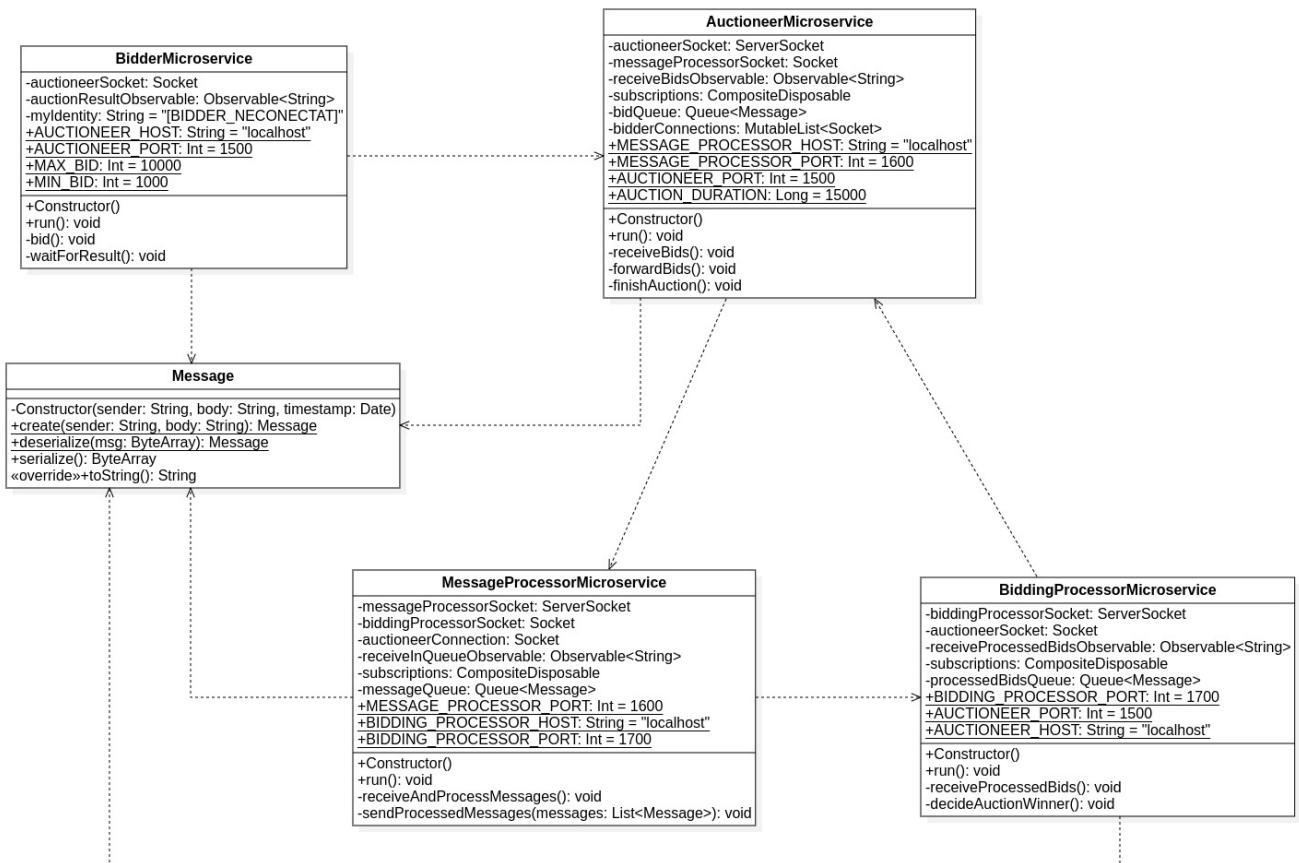


Figura 2 - Diagrama de clase

Semnificațiile microserviciilor

- **BidderMicroservice** - modelează un oferent participant la licitație. Se conectează la **AuctioneerMicroservice** și trimite un mesaj de forma:

```
licitez <SUMA_LICITATA>
```

unde **SUMA_LICITATA** este un număr întreg între 1000 și 10.000, ales aleator. Aceasta reprezintă oferta participantului respectiv. După ce trimite mesajul inițial, așteaptă răspuns de la **AuctioneerMicroservice** pentru a primi rezultatul licitației.

- **AuctioneerMicroservice** - reprezintă punctul central al licitației care primește toate ofertele și **încheie licitația după un timp dinainte stabilit** (în exemplul din laborator, **după 15 secunde**). Mesajele pe care le primește sunt reținute într-o coadă, acestea fiind transmise mai departe spre procesare, către **MessageProcessorMicroservice**.

După aceea, așteaptă rezultatul licitației de la **BiddingProcessorMicroservice** pentru a-l transmite oferanților.

- **MessageProcessorMicroservice** - are rol de procesare a tuturor mesajelor pe care le primește: elimină duplicatele și ordonează mesajele după dată (**de implementat la laborator**). Se consideră că **ceasurile mașinilor de calcul pe care se execută microserviciile componente sunt sincronizate**.

- **BiddingProcessorMicroservice** - decide cine câștigă licitația în funcție de ofertele

primite de la **MessageProcessorMicroservice**. Anunță câștigătorul trimițând mesajul ofertantului care a câștigat către **AuctioneerMicroservice**.

Implementarea aplicației exemplu

Creați un proiect Kotlin/JVM folosind IntelliJ IDEA, denumit, spre exemplu, **Okazii**. **Nu folosiți gestionare de proiecte (Maven / Gradle) sau arhetipuri.** Nu sunt necesare pentru acest laborator.

Adăugați 5 noi module în proiect:

1. **AuctioneerMicroservice**
2. **BidderMicroservice**
3. **BiddingProcessorMicroservice**
4. **MessageLibrary**
5. **MessageProcessorMicroservice**

Pentru a adăuga un modul, se apasă dreapta pe numele proiectului, **Okazii**, din panoul din stânga → New → Module.... Se selectează **Kotlin** în lista din stânga și **JVM | IDEA** în panoul care apare în centru → Next → Completați câmpul „**Module name**” din partea de sus cu numele modulului, conform listei de mai sus și apăsați „**Finish**”.

Atenție: fiecare microserviciu reprezintă un modul separat în proiect.

Structura proiectului ar trebui să arate ca în figură:

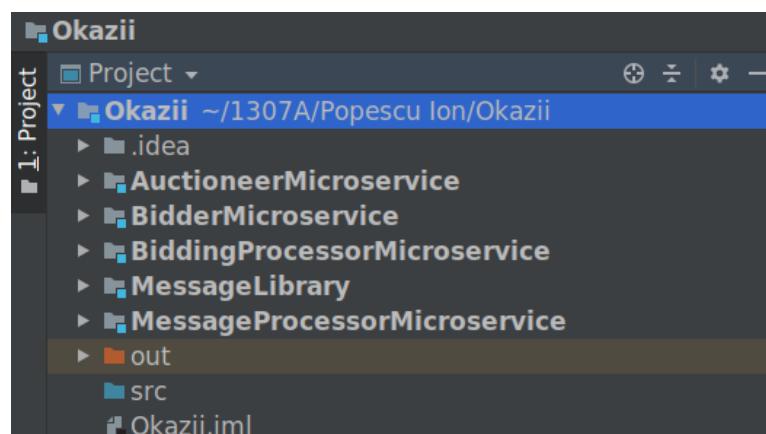
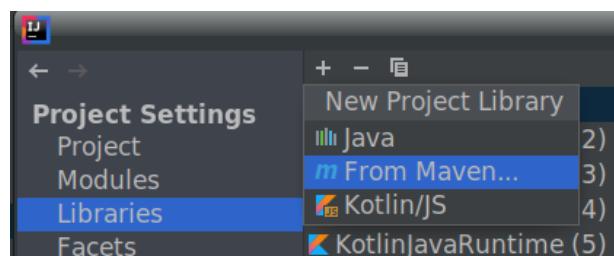


Figura 3 - Structura proiectului

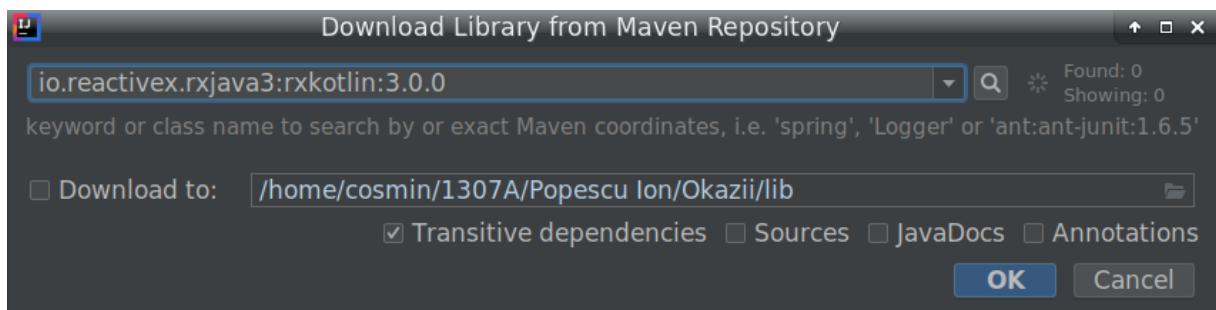
Pentru modulele ce implementează cele 4 microservicii, veți avea nevoie de biblioteca RxKotlin adăugată ca și dependență:

File → Project Structure... → Libraries → click pe pictograma „+” → From Maven.

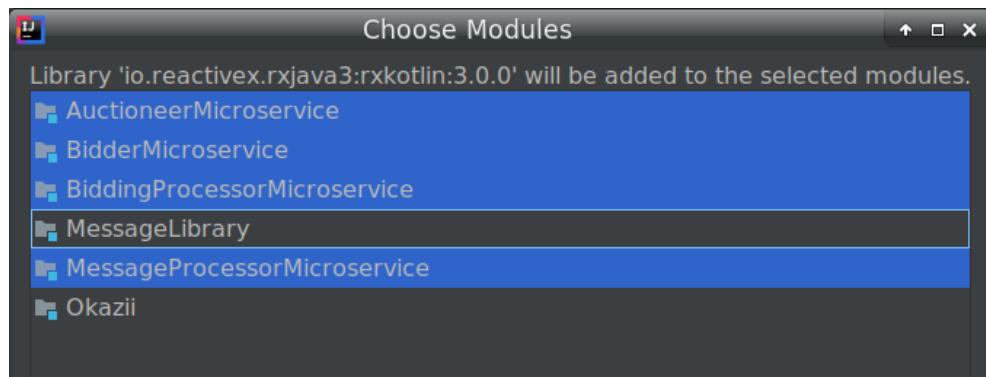


Introduceți următoarele în căsuța de căutare și apăsați „Ok”:

```
io.reactivex.rxjava3:rxkotlin:3.0.0
```



Alegeți cele 4 module corespunzătoare celor 4 microservicii în următoarea fereastră care apare („Choose modules”).



În continuare, începeți implementarea cu modulul **MessageLibrary**, deoarece toate celelalte depind de acesta. Acest modul conține formatul de mesaj care circulă între microservicii, ce pune la dispoziție metode de serializare / deserializare și de creare a mesajului pe baza emițătorului și a corpului, elemente date ca parametri.

• Message.kt

```
import java.text.SimpleDateFormat
import java.util.*

class Message private constructor(val sender: String, val body: String, val timestamp: Date) {
    companion object {
        fun create(sender: String, body: String): Message {
            return Message(sender, body, Date())
        }

        fun deserialize(msg: ByteArray): Message {
            val msgString = String(msg)
            val (timestamp, sender, body) = msgString.split(' ', limit = 3)

            return Message(sender, body, Date(timestamp.toLong()))
        }
    }

    fun serialize(): ByteArray {
        return "${timestamp.time} $sender $body\n".toByteArray()
    }

    override fun toString(): String {

```

```

        val dateString = SimpleDateFormat("dd-MM-yyyy
HH:mm:ss").format(timestamp)
        return "[\$dateString] \$sender >>> \$body"
    }
}

fun main(args: Array<String>) {
    val msg = Message.create("localhost:4848", "test mesaj")
    println(msg)
    val serialized = msg.serialize()
    val deserialized = Message.deserialize(serialized)
    println(deserialized)
}

```

Pentru fiecare modul în parte, activați împachetarea automată sub formă de artefact JAR: File → Project Structure... → Artifacts. Apăsați pe pictograma „+” → Selectați JAR → From modules with dependencies... .

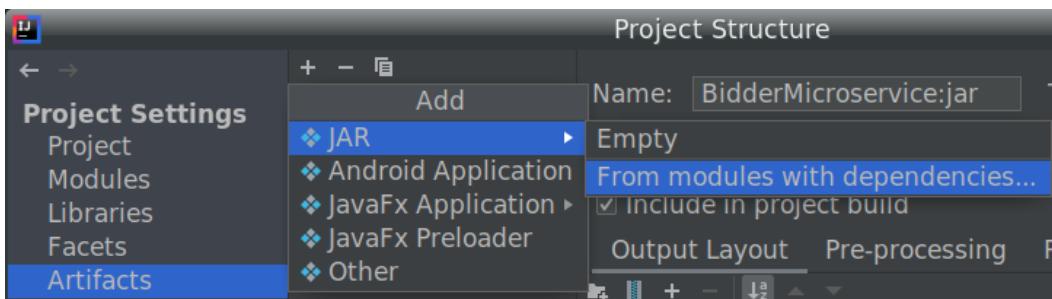


Figura 4 - Adăugare împachetare sub formă de artefact JAR

La **Module** selectați numele modulului din listă, apoi apăsați pe pictograma folder de la secțiunea „**Main Class:**” și selectați clasa corespunzătoare modulului respectiv.

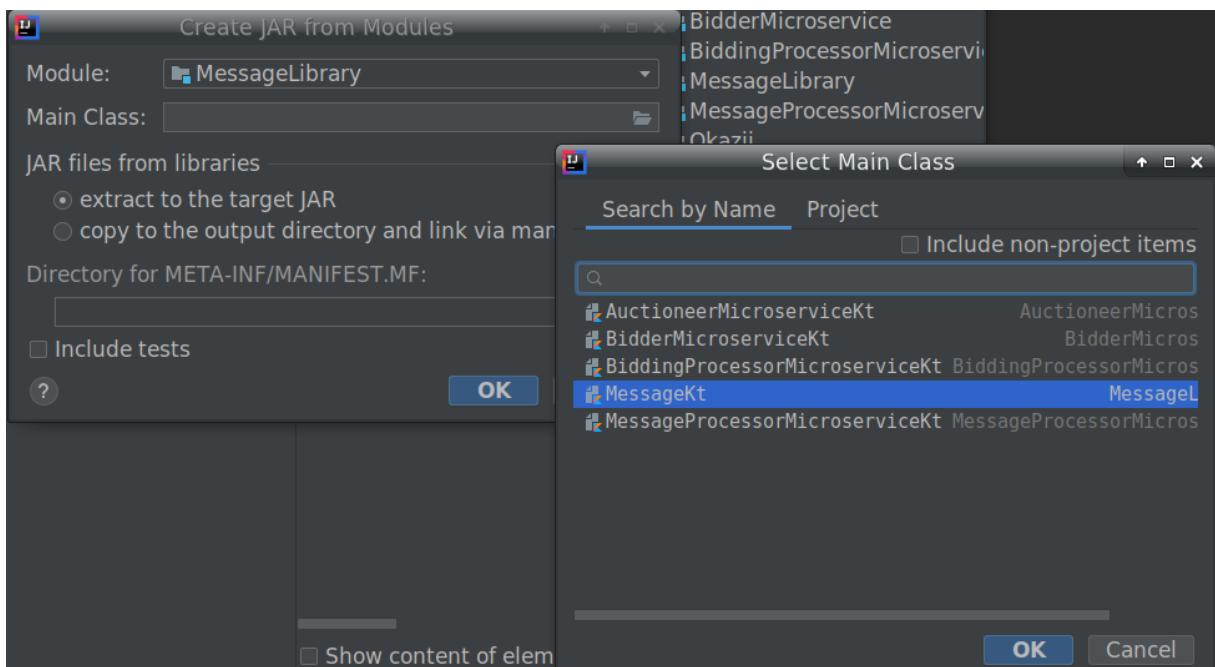
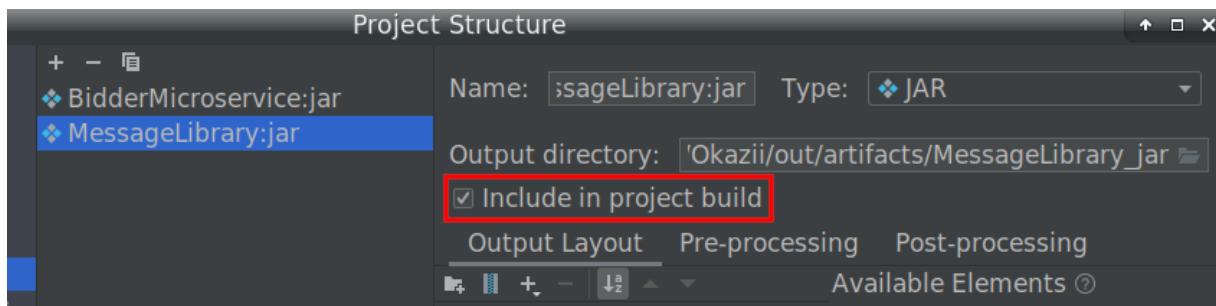


Figura 5 - Configurare împachetare JAR din modul IntelliJ

Apăsați „Ok”, apoi iar „Ok”. **Nu uitați să bifăți „Include in project build”** în panoul

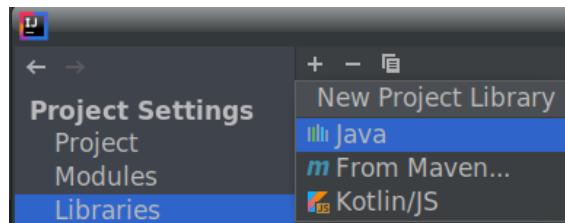
cu setările artefactului pe care l-ați configurat:



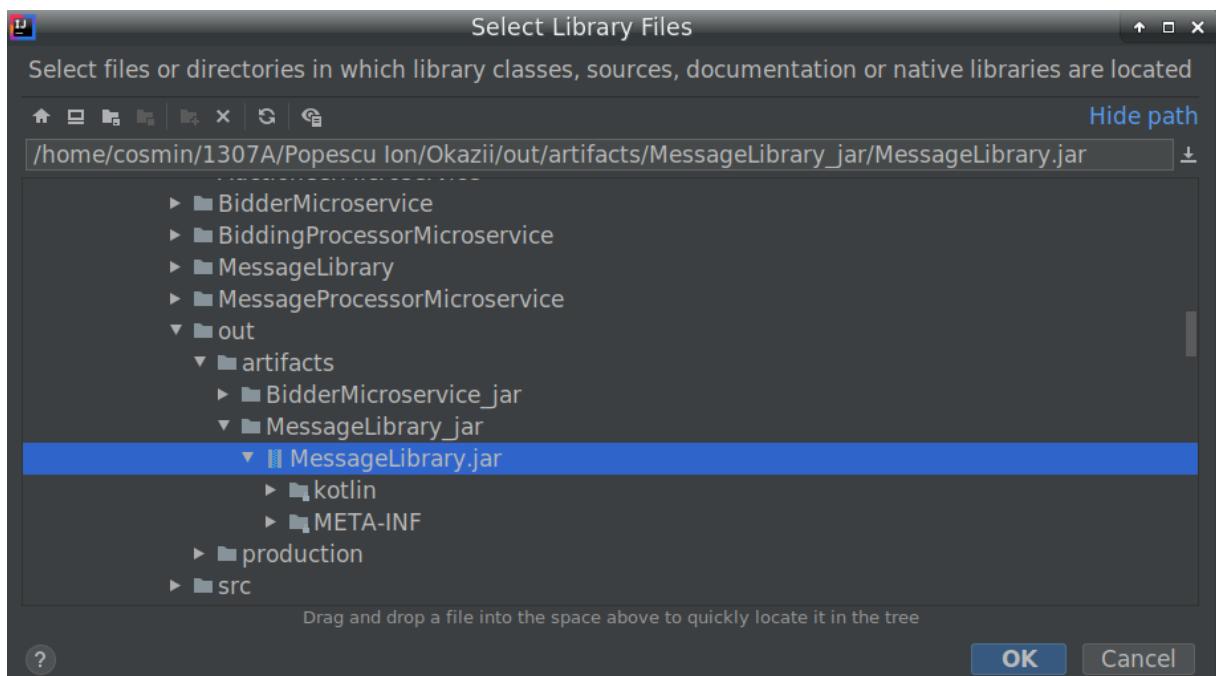
Comparați modulul **MessageLibrary** (click dreapta pe el → **Build module „MessageLibrary”**). Va rezulta un fișier JAR în folder-ul **out/artifacts/MessageLibrary_jar**, denumit **MessageLibrary.jar**.

Adăugați **MessageLibrary.jar ca bibliotecă dependentă pentru celelalte module ale proiectului:**

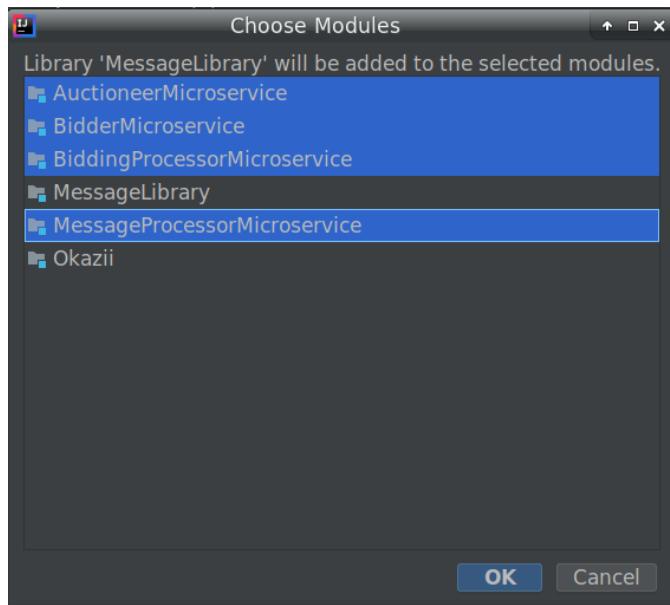
File → Project Structure... → Libraries → click pe pictograma „+” → selectați **Java**



În fereastra de navigare, selectați artefactul JAR denumit **MessageLibrary.jar** rezultat după compilarea modulului **MessageLibrary**.



În fereastra „**Choose modules**” care apare, selectați celelalte 4 module ale proiectului, conform cu figura următoare, și apăsați „**Ok**”:



În continuare, se implementează fiecare din cele 4 microservicii componente. **Acestea comunică prin socket-uri TCP, iar fiecare mesaj primit / trimis prin aceste socket-uri reprezintă un element într-un flux reactiv.**

Codul conține comentarii explicative pe care le puteți consulta pentru a înțelege structura fișierelor sursă care urmează. S-au evidențiat în cod evenimentele de primire a unui nou mesaj în fluxul reactiv (**next**), de terminare a fluxului (**complete**), respectiv de întâmpinare a unei erori (**error**).

- **BidderMicroservice.kt**

```
import io.reactivex.rxjava3.core.Observable
import io.reactivex.rxjava3.kotlin.subscribeBy
import java.io.BufferedReader
import java.io.InputStreamReader
import java.net.Socket
import kotlin.Exception
import kotlin.random.Random
import kotlin.system.exitProcess

class BidderMicroservice {
    private var auctioneerSocket: Socket
    private var auctionResultObservable: Observable<String>
    private var myIdentity: String = "[BIDDER_NECONECTAT]"

    companion object Constants {
        const val AUCTIONEER_HOST = "localhost"
        const val AUCTIONEER_PORT = 1500
        const val MAX_BID = 10_000
        const val MIN_BID = 1_000
    }

    init {
        try {
            auctioneerSocket = Socket(AUCTIONEER_HOST,
AUCTIONEER_PORT)
            println("M-am conectat la Auctioneer!")
        }
    }
}
```

```

myIdentity = "[${auctioneerSocket.localPort}]"

        // se creeaza un obiect Observable ce va emite mesaje
primite printr-un TCP
        // fiecare mesaj primit reprezinta un element al fluxului
de date reactiv
        auctionResultObservable = Observable.create<String> {
emitter ->
        // se citeste raspunsul de pe socketul TCP
        val bufferReader =
BufferedReader(InputStreamReader(auctioneerSocket.inputStream))
        val receivedMessage = bufferReader.readLine()

        // daca se primeste un mesaj gol (NULL), atunci
inseamna ca cealalta parte a socket-ului a fost inchisa
        if (receivedMessage == null) {
            bufferReader.close()
            auctioneerSocket.close()

            emitter.onError(Exception("AuctioneerMicroservice
s-a deconectat."))
            return@create
        }

        // mesajul primit este emis in flux
        emitter.onNext(receivedMessage)

        // deoarece se asteapta un singur mesaj, in continuare
se emite semnalul de incheiere al fluxului
        emitter.onComplete()

        bufferReader.close()
        auctioneerSocket.close()
    }
} catch (e: Exception) {
    println("$myIdentity Nu ma pot conecta la Auctioneer!")
    exitProcess(1)
}
}

private fun bid() {
    // se genereaza o oferta aleatorie din partea bidderului
current
    val pret = Random.nextInt(MIN_BID, MAX_BID)

    // se creeaza mesajul care incapsuleaza oferta
    val biddingMessage =
Message.create("${auctioneerSocket.localAddress}:${auctioneerSocket.lo
calPort}",
        "licitez $pret")

    // bidder-ul trimite pretul pentru care doreste sa liciteze
    val serializedMessage = biddingMessage.serialize()
    auctioneerSocket.getOutputStream().write(serializedMessage)
}

```

```

        // exista o sansa din 2 ca bidder-ul sa-si trimita oferta de 2
ori, eronat
        if (Random.nextBoolean()) {

auctioneerSocket.getOutputStream().write(serializedMessage)
    }
}

private fun waitForResult() {
    println("$myIdentity Astept rezultatul licitatiei...")
    // bidder-ul se inscrie pentru primirea unui raspuns la oferta
trimisa de acesta
    val auctionResultSubscription =
auctionResultObservable.subscribeBy(
        // cand se primeste un mesaj in flux, inseamna ca a sosit
rezultatul licitatiei
        onNext = {
            val resultMessage: Message =
Message.deserialize(it.toByteArray())
            println("$myIdentity Rezultat licitatie:
${resultMessage.body}")
        },
        onError = {
            println("$myIdentity Eroare: $it")
        }
    )

    // se elibereaza memoria obiectului Subscription
    auctionResultSubscription.dispose()
}

fun run() {
    bid()
    waitForResult()
}
}

fun main(args: Array<String>) {
    val bidderMicroservice = BidderMicroservice()
    bidderMicroservice.run()
}

```

• AuctioneerMicroservice.kt

```

import io.reactivex.rxjava3.core.Observable
import io.reactivex.rxjava3.disposables.CompositeDisposable
import io.reactivex.rxjava3.kotlin.subscribeBy
import io.reactivex.plugins.RxJavaPlugins
import java.io.BufferedReader
import java.io.InputStreamReader
import java.net.ServerSocket
import java.net.Socket
import java.net.SocketTimeoutException
import java.util.*
import kotlin.collections.ArrayList
import kotlin.system.exitProcess

```

```

class AuctioneerMicroservice {
    private var auctioneerSocket: ServerSocket
    private lateinit var messageProcessorSocket: Socket
    private var receiveBidsObservable: Observable<String>
    private val subscriptions = CompositeDisposable()
    private val bidQueue: Queue<Message> = LinkedList<Message>()
    private val bidderConnections: MutableList<Socket> =
        mutableListOf()

    companion object Constants {
        const val MESSAGE_PROCESSOR_HOST = "localhost"
        const val MESSAGE_PROCESSOR_PORT = 1600
        const val AUCTIONEER_PORT = 1500
        const val AUCTION_DURATION: Long = 15_000 // licitatia dureaza
15 secunde
    }

    init {
        auctioneerSocket = ServerSocket(AUCTIONEER_PORT)
        auctioneerSocket.setSoTimeout(AUCTION_DURATION.toInt())
        println("AuctioneerMicroservice se executa pe portul:
${auctioneerSocket.localPort}")
        println("Se asteapta oferte de la bidderi...")

        // se creeaza obiectul Observable cu care se genereaza
evenimente cand se primesc oferte de la bidderi
        receiveBidsObservable = Observable.create<String> { emitter ->
            // se asteapta conexiuni din partea bidderilor
            while (true) {
                try {
                    val bidderConnection = auctioneerSocket.accept()
                    bidderConnections.add(bidderConnection)

                    // se citeste mesajul de la bidder de pe socketul
TCP
                    val bufferedReader =
BufferedReader(InputStreamReader(bidderConnection.inputStream))
                    val receivedMessage = bufferedReader.readLine()

                    // daca se primeste un mesaj gol (NULL), atunci
inseamna ca cealalta parte a socket-ului a fost inchisa
                    if (receivedMessage == null) {
                        // deci subscriber-ul respectiv a fost
deconectat
                        bufferedReader.close()
                        bidderConnection.close()

emitter.onError(Exception("Eroare: Bidder-ul
${bidderConnection.port} a fost deconectat."))
                }
            }

            // se emite ce s-a citit ca si element in fluxul
de mesaje
            emitter.onNext(receivedMessage)
        } catch (e: SocketTimeoutException) {
    }
}

```

```

        // daca au trecut cele 15 secunde de la pornirea
licitatiei, inseamna ca licitatia s-a inchis
        // se emite semnalul Complete pentru a inchisa
fluxul de oferte
        emitter.onComplete()
        break
    }
}
}

private fun receiveBids() {
    // incepe prin a primi ofertele de la bidderi
    val receiveBidsSubscription =
receiveBidsObservable.subscribeBy(
    onNext = {
        val message = Message.deserialize(it.toByteArray())
        println(message)
        bidQueue.add(message)
    },
    onComplete = {
        // licitatia s-a inchis
        // se trimit raspunsurile mai departe catre procesorul
de mesaje
        println("Licitatia s-a inchis! Se trimit ofertele
spre procesare...")
        forwardBids()
    },
    onError = { println("Eroare: $it") }
)
    subscriptions.add(receiveBidsSubscription)
}

private fun forwardBids() {
    try {
        messageProcessorSocket = Socket(MESSAGE_PROCESSOR_HOST,
MESSAGE_PROCESSOR_PORT)

subscriptions.add(Observable.fromIterable(bidQueue).subscribeBy(
    onNext = {
        // trimitere mesaje catre procesorul de mesaje
messageProcessorSocket.getOutputStream().write(it.serialize())
        println("Am trimis mesajul: $it")
    },
    onComplete = {
        println("Am trimis toate ofertele catre
MessageProcessor.")
        val bidEndMessage = Message.create(
"${messageProcessorSocket.localAddress}:${messageProcessorSocket.lo-
calPort}",
        "final"
    )

messageProcessorSocket.getOutputStream().write(bidEndMessage.seriali-

```

```

ze()))

        // dupa ce s-a terminat licitatia, se asteapta
raspuns de la MessageProcessorMicroservice
        // cum ca a primit toate mesajele
        val bufferedReader =
BufferedReader(InputStreamReader(messageProcessorSocket.inputStream))
        bufferedReader.readLine()

        messageProcessorSocket.close()

        finishAuction()
    }
))

} catch (e: Exception) {
    println("Nu ma pot conecta la MessageProcessor!")
    auctioneerSocket.close()
    exitProcess(1)
}
}

private fun finishAuction() {
    // se asteapta rezultatul licitatiei
    try {
        val biddingProcessorConnection = auctioneerSocket.accept()
        val bufferedReader =
BufferedReader(InputStreamReader(biddingProcessorConnection.inputs-
tream))

        // se citeste rezultatul licitatiei de la
AuctioneerMicroservice de pe socketul TCP
        val receivedMessage = bufferedReader.readLine()

        val result: Message =
Message.deserialize(receivedMessage.toByteArray())
        val winningPrice = result.body.split(" ")[1].toInt()
        println("Am primit rezultatul licitatiei de la
BiddingProcessor: ${result.sender} a castigat cu pretul:
$winningPrice")

        // se creeaza mesajele pentru rezultatele licitatiei
        val winningMessage =
Message.create(auctioneerSocket.localSocketAddress.toString(),
            "Licitatie castigata! Pret castigator: $winningPrice")
        val losingMessage =
Message.create(auctioneerSocket.localSocketAddress.toString(),
            "Licitatie pierduta...")

        // se anunta castigatorul
        bidderConnections.forEach {
            if (it.remoteSocketAddress.toString() ==
result.sender) {

it.getOutputStream().write(winningMessage.serialize())
        } else {

```

```

        it.getOutputStream().write(losingMessage.serialize())
    }
    it.close()
}
} catch (e: Exception) {
    println("Nu ma pot conecta la BiddingProcessor!")
    auctioneerSocket.close()
    exitProcess(1)
}

// se elibereaza memoria din multimea de Subscriptions
subscriptions.dispose()
}

fun run() {
    receiveBids()
}
}

fun main(args: Array<String>) {
    val bidderMicroservice = AuctioneerMicroservice()
    bidderMicroservice.run()
}

```

• [MessageProcessorMicroservice.kt](#)

```

import io.reactivex.rxjava3.core.Observable
import io.reactivex.rxjava3.disposables.CompositeDisposable
import io.reactivex.rxjava3.kotlin.subscribeBy
import java.io.BufferedReader
import java.io.InputStreamReader
import java.net.ServerSocket
import java.net.Socket
import java.util.*
import kotlin.system.exitProcess

class MessageProcessorMicroservice {
    private var messageProcessorSocket: ServerSocket
    private lateinit var biddingProcessorSocket: Socket
    private var auctioneerConnection: Socket
    private var receiveInQueueObservable: Observable<String>
    private val subscriptions = CompositeDisposable()
    private val messageQueue: Queue<Message> = LinkedList<Message>()

    companion object Constants {
        const val MESSAGE_PROCESSOR_PORT = 1600
        const val BIDDING_PROCESSOR_HOST = "localhost"
        const val BIDDING_PROCESSOR_PORT = 1700
    }

    init {
        messageProcessorSocket = ServerSocket(MESSAGE_PROCESSOR_PORT)
        println("MessageProcessorMicroservice se executa pe portul: ${messageProcessorSocket.localPort}")
        println("Se asteapta mesaje pentru procesare...")
    }
}

```

```

    // se asteapta mesaje primite de la AuctioneerMicroservice
    auctioneerConnection = messageProcessorSocket.accept()
    val bufferReader =
    BufferedReader(InputStreamReader(auctioneerConnection.inputStream))

    // se creeaza obiectul Observable cu care se captureaza
    mesajele de la AuctioneerMicroservice
    receiveInQueueObservable = Observable.create<String> { emitter
->
        while (true) {
            // se citeste mesajul de la AuctioneerMicroservice de
            pe socketul TCP
            val receivedMessage = bufferReader.readLine()

            // daca se primeste un mesaj gol (NULL), atunci
            inseamna ca cealalta parte a socket-ului a fost inchisa
            if (receivedMessage == null) {
                // deci subscriber-ul respectiv a fost deconectat
                bufferReader.close()
                auctioneerConnection.close()

                emitter.onError(Exception("Eroare:
AuctioneerMicroservice ${auctioneerConnection.port} a fost
deconectat."))
                break
            }

            // daca mesajul este cel de incheiere a licitatiei
            (avand corpul "final"), atunci se emite semnalul Complete
            if
            (Message.deserialize(receivedMessage.toByteArray()).body == "final") {
                emitter.onComplete()

                break
            } else {
                // se emite ce s-a citit ca si element in fluxul
                de mesaje
                emitter.onNext(receivedMessage)
            }
        }
    }

    private fun receiveAndProcessMessages() {
        // se primesc si se adauga in coada mesajele de la
        AuctioneerMicroservice
        val receiveInQueueSubscription = receiveInQueueObservable
            //TODO --- filtrati duplicatele folosind operatorul de
            filtrare
            .subscribeBy(
                onNext = {
                    val message =
                    Message.deserialize(it.toByteArray())
                    println(message)
                    messageQueue.add(message)
                },
            )
    }
}

```

```

onComplete = {
    // s-a incheiat primirea tuturor mesajelor
    //TODO --- se ordoneaza in functie de data si ora
cand mesajele au fost primite

        // s-au primit toate mesajele de la
AuctioneerMicroservice, i se trimite un mesaj pentru a semnala
        // acest lucru
        val finishedMessagesMessage = Message.create(
"$${auctioneerConnection.localAddress}:$${auctioneerConnection.local-
Port}",
            "am primit tot"
        )

auctioneerConnection.getOutputStream().write(finishedMessagesMessage.s-
erialize())
        auctioneerConnection.close()

        // se trimit mai departe mesajele procesate catre
BiddingProcessor
        sendProcessedMessages()
},
onError = { println("Eroare: $it") }
)
subscriptions.add(receiveInQueueSubscription)
}

private fun sendProcessedMessages() {
    try {
        biddingProcessorSocket = Socket(BIDDING_PROCESSOR_HOST,
BIDDING_PROCESSOR_PORT)

        println("Trimit urmatoarele mesaje:")
        Observable.fromIterable(messageQueue).subscribeBy(
            onNext = {
                println(it.toString())

                // trimitere mesaje catre procesorul licitatiei,
care decide rezultatul final
            },
            onComplete = {
                val noMoreMessages = Message.create(
"$${biddingProcessorSocket.localAddress}:$${biddingProcessorSocket.lo-
calPort}",
                    "final"
            )

biddingProcessorSocket.getOutputStream().write(noMoreMessages.seri-
alize())
                biddingProcessorSocket.close()

                // se elibereaza memoria din multimea de

```

```

Subscriptions
    subscriptions.dispose()
}
)
} catch (e: Exception) {
    println("Nu ma pot conecta la BiddingProcessor!")
    messageProcessorSocket.close()
    exitProcess(1)
}
}

fun run() {
    receiveAndProcessMessages()
}
}

fun main(args: Array<String>) {
    val messageProcessorMicroservice = MessageProcessorMicroservice()
    messageProcessorMicroservice.run()
}

```

• BiddingProcessorMicroservice.kt

```

import io.reactivex.rxjava3.core.Observable
import io.reactivex.rxjava3.disposables.CompositeDisposable
import io.reactivex.rxjava3.kotlin.subscribeBy
import java.io.BufferedReader
import java.io.InputStreamReader
import java.net.ServerSocket
import java.net.Socket
import java.util.*
import kotlin.system.exitProcess

class BiddingProcessorMicroservice {
    private var biddingProcessorSocket: ServerSocket
    private lateinit var auctioneerSocket: Socket
    private var receiveProcessedBidsObservable: Observable<String>
    private val subscriptions = CompositeDisposable()
    private val processedBidsQueue: Queue<Message> =
        LinkedList<Message>()

    companion object Constants {
        const val BIDDING_PROCESSOR_PORT = 1700
        const val AUCTIONEER_PORT = 1500
        const val AUCTIONEER_HOST = "localhost"
    }

    init {
        biddingProcessorSocket = ServerSocket(BIDDING_PROCESSOR_PORT)
        println("BiddingProcessorMicroservice se executa pe portul: ${biddingProcessorSocket.localPort}")
        println("Se asteapta ofertele pentru finalizarea licitatiei...")

        // se asteapta mesaje primite de la
        MessageProcessorMicroservice
    }
}

```

```

        val messageProcessorConnection =
bidningProcessorSocket.accept()
        val bufferReader =
BufferedReader(InputStreamReader(messageProcessorConnection.inputs-
tream))

        // se creeaza obiectul Observable cu care se captureaza
mesajele de la MessageProcessorMicroservice
        receiveProcessedBidsObservable = Observable.create<String> {
emitter ->
            while (true) {
                // se citeste mesajul de la
MessageProcessorMicroservice de pe socketul TCP
                val receivedMessage = bufferReader.readLine()

                // daca se primeste un mesaj gol (NULL), atunci
inseamna ca cealalta parte a socket-ului a fost inchisa
                if (receivedMessage == null) {
                    // deci MessageProcessorMicroservice a fost
deconectat
                    bufferReader.close()
                    messageProcessorConnection.close()

                    emitter.onError(Exception("Eroare:
MessageProcessorMicroservice ${messageProcessorConnection.port} a fost
deconectat."))
                    break
                }

                // daca mesajul este cel de tip „FINAL DE LISTA DE
MESAJE” (avand corpul "final"), atunci se emite semnalul Complete
                if
(Message.deserialize(receivedMessage.toByteArray()).body == "final") {
                    emitter.onComplete()

                    // s-au primit toate mesajele de la
MessageProcessorMicroservice, i se trimit un mesaj pentru a semnala
                    // acest lucru
                    val finishedBidsMessage = Message.create(
"${messageProcessorConnection.localAddress}:${messageProcessorConnec-
tion.localPort}",
                        "am primit tot"
                    )
                }

messageProcessorConnection.getOutputStream().write(finishedBidsMessa-
ge.serialize())
                    messageProcessorConnection.close()

                    break
                } else {
                    // se emite ce s-a citit ca si element in fluxul
de mesaje
                    emitter.onNext(receivedMessage)
                }
}

```

```

        }
    }

    private fun receiveProcessedBids() {
        // se primesc si se adauga in coada ofertele procesate de la
MessageProcessorMicroservice
        val receiveProcessedBidsSubscription =
receiveProcessedBidsObservable
            .subscribeBy(
                onNext = {
                    val message =
Message.deserialize(it.toByteArray())
                    println(message)
                    processedBidsQueue.add(message)
                },
                onComplete = {
                    // s-a incheiat primirea tuturor mesajelor
                    // se decide castigatorul licitatiei
                    decideAuctionWinner()
                },
                onError = { println("Eroare: $it") }
            )
        subscriptions.add(receiveProcessedBidsSubscription)
    }

    private fun decideAuctionWinner() {
        // se calculeaza castigatorul ca fiind cel care a ofertat cel
mai mult
        val winner: Message? = processedBidsQueue.toList().maxBy {
            // corpul mesajului e de forma "licitez <SUMA_LICITATA>"
            // se preia a doua parte, separata de spatiu
            it.body.split(" ")[1].toInt()
        }

        println("Castigatorul este: ${winner?.sender}")

        try {
            auctioneerSocket = Socket(AUCTIONEER_HOST,
AUCTIONEER_PORT)

            // se trimit castigatorul catre AuctioneerMicroservice
auctioneerSocket.getOutputStream().write(winner!!.serialize())
            auctioneerSocket.close()

            println("Am anuntat castigatorul catre
AuctioneerMicroservice.")
        } catch (e: Exception) {
            println("Nu ma pot conecta la Auctioneer!")
            biddingProcessorSocket.close()
            exitProcess(1)
        }
    }

    fun run() {

```

```
receiveProcessedBids()

    // se elibereaza memoria din multimea de Subscriptions
    subscriptions.dispose()
}

}

fun main(args: Array<String>) {
    val biddingProcessorMicroservice = BiddingProcessorMicroservice()
    biddingProcessorMicroservice.run()
}
```

Compilarea și execuția proiectului

Proiectul se compilează astfel: se folosește meniul **Build → Build Project**, pentru a compila toate modulele componente deodată.

Proiectul se execută astfel: se pornește fiecare microserviciu în parte (eventual de la linia de comandă), cu următoarele mențiuni:

- mai întâi se pornesc **MessageProcessorMicroservice** și **BiddingProcessorMicroservice**, în orice ordine se dorește
- apoi se pornește **AuctioneerMicroservice**, dar acesta va încheia licitația, în mod implicit, după 15 secunde. Așadar, **aveți 15 secunde la dispoziție să porniți câteva microservicii BidderMicroservice**.
- în final, se pornesc câteva microservicii de tip **BidderMicroservice**, care să reprezinte ofertanții ce participă la licitație

Exemplu de execuție de la linia de comandă:

```
java -jar MessageProcessorMicroservice.jar
```

Exemplu de execuție

The figure consists of six terminal windows arranged in two columns and three rows. The left column contains three windows: 'MessageProcessor' (red text), 'BiddingProcessor' (blue text), and 'Auctioneer' (green text). The right column contains three windows: 'Bidder 1' (red text), 'Bidder 2' (blue text), and 'Bidder 3' (green text). Each window shows the command-line interface for running a specific microservice (e.g., java -jar ./MessageProcessor.jar) and its corresponding log output.

Figura 6 - Exemplu de execuție

Teme de laborator

- Implementați și executați aplicația exemplu din laborator, pornind licitația cu un număr mare de microservicii **Bidder** (de exemplu, 100). Puteți crea, spre exemplu, un script Bash care pornește 100 de procese **Bidder**.
- Modificați microserviciul **MessageProcessor** astfel încât să filtreze mesajele duplicate din fluxul reactiv primit de la **Auctioneer**.
- Modificați microserviciul **MessageProcessor** astfel încât, atunci când s-a primit tot fluxul de mesaje de la **Auctioneer**, să sortați mesajele primite înainte de a le redirecționa spre **BiddingProcessor**.

Temă pentru acasă

- Modificați aplicația astfel încât ofertanții să fie identificați de un nume, un număr de telefon și un e-mail. Mesajele care circulă între microservicii trebuie să includă și datele noi introduse.
- Să se includă capabilități de instrumentare a microserviciilor componente, pentru colectarea datelor legate de încărcare și funcționare. Datele colectate vor fi scrise într-un fișier log.

Bibliografie

- [1]: The Reactive Manifesto - <https://github.com/reactivemanifesto/reactivemanifesto>
[2]: Reactive Programming with Kotlin, Alex Sullivan, First Edition -

<https://store.raywenderlich.com/products/reactive-programming-with-kotlin>

[3]: Reactive Programming in Kotlin, Rivu Chakraborty, Packt Publishing

[4]: RxKotlin - RxJava bindings for Kotlin - <https://github.com/ReactiveX/RxKotlin>

Sisteme Distribuite - Laborator 8

Instalarea microserviciilor folosind Docker

Instalarea Docker Engine

Se oferă instrucțiuni de instalare pentru sistemul de operare **Debian**, disponibil pe stațiile din laborator. Comenzile ce urmează vor fi executate într-o sesiune de **terminal**.

1. Actualizați indecșii gestionarului de pachete **apt**:

```
sudo apt update
```

2. Permiteți folosirea depozitelor de pachete (engl. *repositories*) prin HTTPS + instalați câteva pachete adiționale necesare:

```
sudo apt-get install \
    apt-transport-https \
    ca-certificates \
    curl \
    gnupg2 \
    software-properties-common
```

3. Adăugați cheia GPG a Docker în inelul de chei al gestionarului de pachete **apt**:

```
curl -fsSL https://download.docker.com/linux/debian/gpg | sudo apt-key
add -
```

4. Adăugați depozitul de pachete oficial Docker în lista sistemului:

```
sudo add-apt-repository \
    "deb [arch=amd64] https://download.docker.com/linux/debian \
    $(lsb_release -cs) \
    stable"
```

5. Actualizați din nou indecșii **apt** (pas necesar pentru că ați modificat mai sus baza de date cu depozite de pachete):

```
sudo apt update
```

6. Instalați Docker Engine (*Community Edition*, de aici provine sufixul “**-ce**”), împreună cu utilitarele pentru linia de comandă ale Docker (*Command Line Interface*, de aici provine sufixul “**-cli**”):

```
sudo apt install -y docker-ce docker-ce-cli
```

În mod implicit, nu puteți trimite comenzi procesului *daemon dockerd* (componenta server a Docker), deoarece acesta se execută sub utilizatorul **root**. Așa încât, sunt necesari următorii pași suplimentari, pentru a nu fi nevoie să prefixați fiecare comandă Docker cu „**sudo**”:

7. Creați grupa **docker** pe sistemul gazdă:

```
sudo groupadd docker
```

Adăugați utilizatorul neprivilegiat pe care sunteți conectat (pe care îl folosiți pentru lucrul cu Docker) în grupa creată mai sus:

```
sudo usermod -aG docker $USER
```

Pentru ca modificările aduse mai sus să aibă efect, deconectați-vă de pe utilizatorul cu care v-ați conectat (*log off*) și conectați-vă din nou (*log in*). Ca alternativă, puteți reporni sistemul.

Apoi, verificați că Docker Engine funcționează, cu următoarea comandă:

```
docker info
```

```
File Edit View Terminal Tabs Help
cosmin@debian-gl553v:~$ docker info
Client:
  Debug Mode: false

Server:
  Containers: 55
    Running: 0
    Paused: 0
    Stopped: 55
  Images: 110
  Server Version: 19.03.5
  Storage Driver: overlay2
    Backing Filesystem: extfs
    Supports d_type: true
    Native Overlay Diff: true
  Logging Driver: json-file
  Cgroup Driver: cgroupfs
  Plugins:
    Volume: local
    Network: bridge host ipvlan macvlan null overlay
    Log: awslogs fluentd gcplogs gelf journald json-file local logentries splunk syslog
  Swarm: inactive
  Runtimes: runc
    Default Runtime: runc
  Init Binary: docker-init
  containerd version: b34a5c8af56e510852c35414db4c1f4fa6172339
  runc version: 3e425f80a8c931f88e6d94a8c831b9d5aa481657
  init version: fec3683
  Security Options:
    apparmor
    seccomp
      Profile: default
  Kernel Version: 5.3.7
  Operating System: Debian GNU/Linux 10 (buster)
  OSType: linux
  Architecture: x86_64
  CPUs: 8
  Total Memory: 31.26GiB
  Name: debian-gl553v
  ID: NA24:D3WJ:VSZY:R04T:Z4QX:6TEK:SJCC:HQAD:JBFK:LZ4K:DSQW:SH6S
  Docker Root Dir: /var/lib/docker
  Debug Mode: false
```

Figura 1 - Informații Docker Engine

Permiterea lucrului cu registre Docker nesecurizate

Pentru a putea folosi registrul Docker local creat mai departe în laborator, trebuie să configurați Docker Engine astfel încât să accepte registrele nesecurizate, acest lucru fiind implicit nepermis. Folosind un utilizator privilegiat, editați (sau creați, dacă nu există) fișierul **/etc/docker/daemon.json** și adăugați următorul conținut:

```
{
  "insecure-registries" : ["localhost:5000"]
}
```

Exemplu:

```
sudo nano /etc/docker/daemon.json
```

Copiați conținutul de mai sus, apoi apăsați, în ordine: CTRL+O, ENTER, CTRL+X.

Reportați Docker Engine:

```
sudo service docker restart
```

Pentru alte sisteme de operare, consultați documentația care conține instrucțiuni de instalare pentru fiecare în parte, disponibilă la următorul URL: <https://docs.docker.com/install/>.

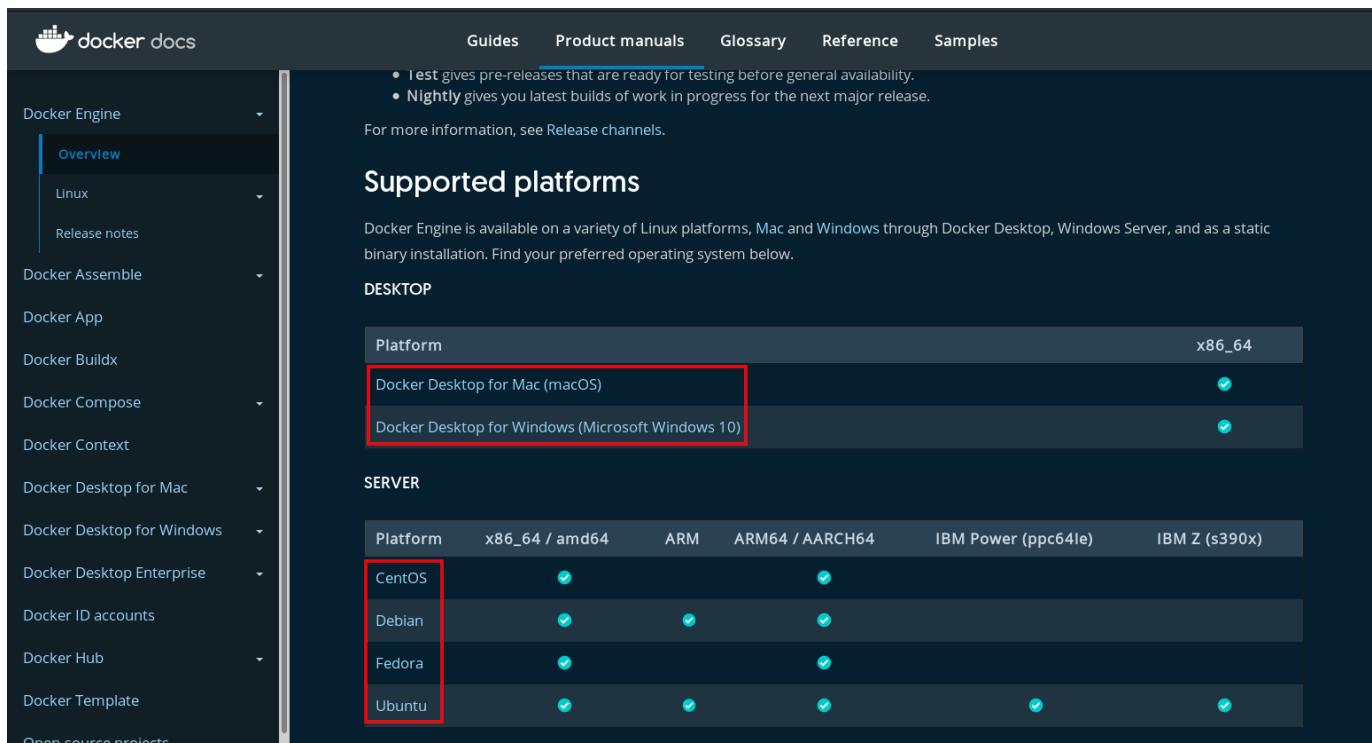


Figura 2 - Instrucțiuni de instalare pentru diverse platforme

Lucrul cu comenzi Docker - crearea unui registru local de imagini

Registru (engl. **registry**) implicit utilizat de Docker Engine este cel oficial, găzduit de **Docker Hub**: docker.io/library. În registru, utilizatorii își pot crea aşa numitele depozite (engl. **repository**), în care se pot publica imagini Docker. Însă, utilizatorul nu are control asupra imaginilor publicate, acestea fiind public accesibile și sub autoritatea Docker Inc, iar pentru a avea acces pentru publicare, este necesar un cont Docker Hub. Așadar, imaginile Docker pe care le veți crea pe parcursul laboratoarelor le veți publica într-un registru privat, local.

În continuare, veți folosi, gradual, comenzi Docker primare pentru lucrul cu imagini și containere. La finalul laboratorului, veți avea disponibil un registru local Docker, precum și câteva imagini încărcate în acesta, gata de utilizare.

Descărcarea unei imagini Docker

Pentru început, descărcați local imaginea Docker **registry** din registru oficial Docker Hub, folosind următoarea comandă:

```
docker pull registry:2
```

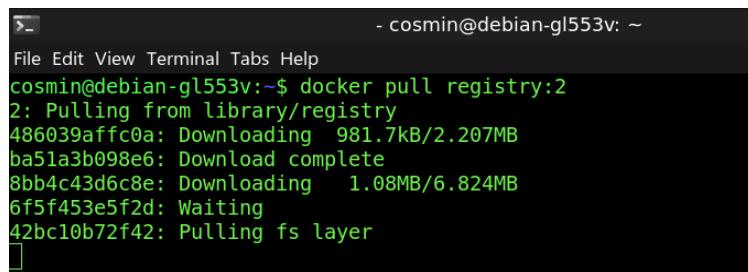
Această comandă va prelua din registrul specificat (în acest caz, cel implicit, adică Docker Hub) imaginea trimisă ca ultim parametru. O imagine este identificată prin **nume:etichetă**. Eticheta are rol de versionare, pot exista mai multe imagini cu același nume, dar cu etichete diferite în funcție de situație. În acest caz, am cerut versiunea **2** a imaginii **registry**.

Dacă nu specificați nicio etichetă atunci când folosiți / descărcați o imagine Docker, atunci se va folosi eticheta implicită latest. De exemplu:

```
docker pull debian
```

va descărca ultima versiune de imagine cu Debian.

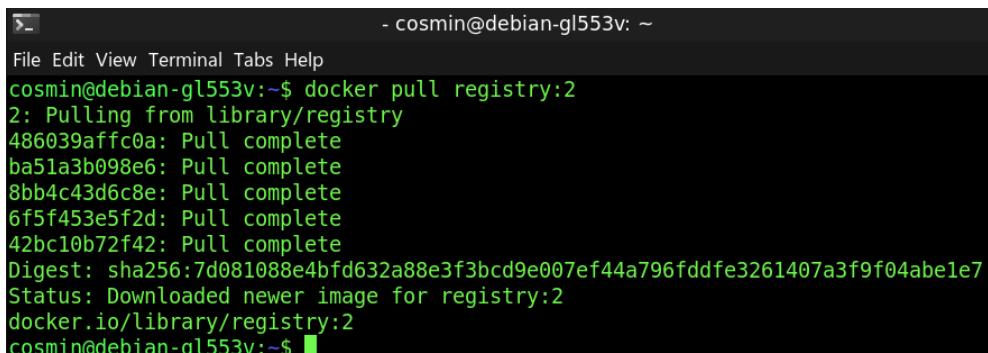
Docker va începe să preia fiecare strat în parte din sistemul de fișiere stratificat (**Union File System**):



```
- cosmin@debian-gl553v: ~
File Edit View Terminal Tabs Help
cosmin@debian-gl553v:~$ docker pull registry:2
2: Pulling from library/registry
486039affc0a: Downloading 981.7kB/2.207MB
ba51a3b098e6: Download complete
8bb4c43d6c8e: Downloading 1.08MB/6.824MB
6f5f453e5f2d: Waiting
42bc10b72f42: Pulling fs layer
```

Figura 3 - Preluarea unei imagini Docker

După ce se termină de descărcat, imaginea este disponibilă pentru utilizare:



```
- cosmin@debian-gl553v: ~
File Edit View Terminal Tabs Help
cosmin@debian-gl553v:~$ docker pull registry:2
2: Pulling from library/registry
486039affc0a: Pull complete
ba51a3b098e6: Pull complete
8bb4c43d6c8e: Pull complete
6f5f453e5f2d: Pull complete
42bc10b72f42: Pull complete
Digest: sha256:7d081088e4bfd632a88e3f3bcd9e007ef44a796fddfe3261407a3f9f04abe1e7
Status: Downloaded newer image for registry:2
docker.io/library/registry:2
cosmin@debian-gl553v:~$
```

Figura 4 - Descărcare completă a unei imagini Docker

Verificați că imaginea **registry:2** există în registrul local, cu următoarea comandă:

```
docker image ls
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
registry	2	708bc6af7e5e	2 weeks ago	25.8MB
jamtur01/redis	latest	5a953fa107c7	3 months ago	95.2MB
jamtur01/sinatra	latest	14c3193c863f	3 months ago	347MB
astronomos2007/nginx	latest	243727b5c5af	3 months ago	152MB
astronomos2007/static_web	latest	2576993ae8c4	3 months ago	161MB
ccrihan/static_web	latest	2576993ae8c4	3 months ago	161MB

Figura 5 - Listare imagini existente local

SAU: o alternativă la comanda de mai sus:

```
docker images
```

Pornirea unui container Docker (pornirea registrului local)

Acum, folosiți imaginea **registry** descărcată pentru a porni un container Docker ce expune local, pe portul **5000**, un depozit (**registry**) privat. Execuția următoarea comandă în terminal:

```
docker run --restart=always -d -p 5000:5000 --name registru_docker  
registry:2
```

Comanda de mai sus pornește un container Docker (**docker run**), pe care îl repornește automat în caz de eroare / închiderea sistemului (**--restart=always**), containerul fiind pornit în mod detașat (**-d**). Se expune portul **5000** din container și se mapează pe portul **5000** al sistemului, iar containerul este identificat prin numele **registru_docker**.



```
- cosmin@debian-gl553v: ~  
File Edit View Terminal Tabs Help  
cosmin@debian-gl553v:~$ docker run --restart=always -d -p 5000:5000 --name registru_docker  
registry:2  
76c362f83a14b391db8c85527746264302cf51921f21f749d81eef1d75c74acb  
cosmin@debian-gl553v:~$
```

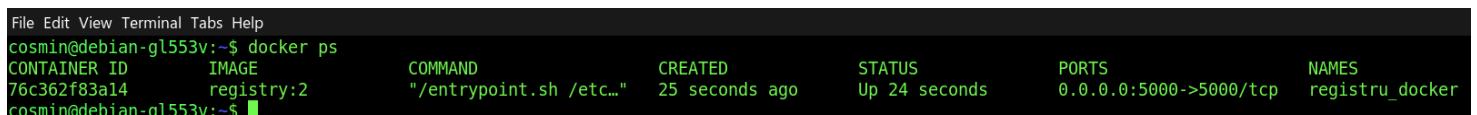
Figura 6 - Pornirea unui container

Observați că Docker a returnat un identificator unic alfanumeric, atașat containerului care tocmai a fost pornit. Nu s-a returnat nicio eroare, niciun alt mesaj, deci containerul a fost pornit cu succes și se execută în fundal.

Verificarea stării containerelor Docker

Se pot verifica stările containerelor aflate **în execuție** astfel:

```
docker ps
```



```
File Edit View Terminal Tabs Help  
cosmin@debian-gl553v:~$ docker ps  
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES  
76c362f83a14 registry:2 "/entrypoint.sh /etc..." 25 seconds ago Up 24 seconds 0.0.0.0:5000->5000/tcp registru_docker  
cosmin@debian-gl553v:~$
```

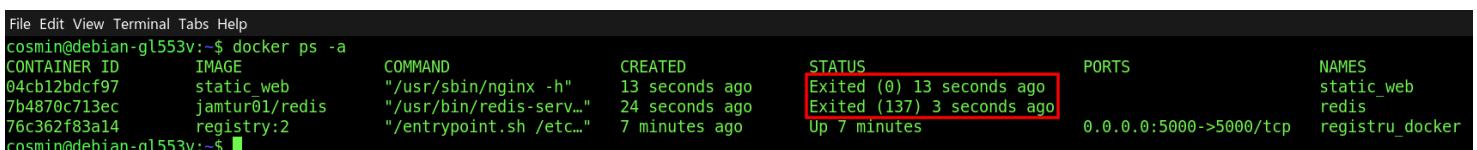
Figura 7 - Verificarea listei de containere pornite

Abrevierea „**ps**” provine de la „**process status**”.

Observați că apare listat containerul pornit în pași anteriori (**registru_docker**). Comanda returnează **doar lista de containere în execuție**.

Pentru a afișa toate containerele disponibile, indiferent de starea lor (adică inclusiv cele opuse de utilizator, opuse forțat, sau opuse din cauza unei erori), adăugați parametrul **-a** (**--all**) la comanda de mai sus:

```
docker ps -a
```



```
File Edit View Terminal Tabs Help  
cosmin@debian-gl553v:~$ docker ps -a  
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES  
04cb12bd9f97 static web "/usr/sbin/nginx -h" 13 seconds ago Exited (0) 13 seconds ago 0.0.0.0:5000->5000/tcp static_web  
7b4870c713ec janitur01/redis "/usr/bin/redis-serv..." 24 seconds ago Exited (137) 3 seconds ago redis  
76c362f83a14 registry:2 "/entrypoint.sh /etc..." 7 minutes ago Up 7 minutes 0.0.0.0:5000->5000/tcp registru_docker  
cosmin@debian-gl553v:~$
```

Figura 8 - Afisarea listurilor tuturor containerelor, indiferent de stare

Acum apar și containerele care au fost pornite la un moment dat, iar apoi au fost opuse de

utilizator sau din altă cauză.

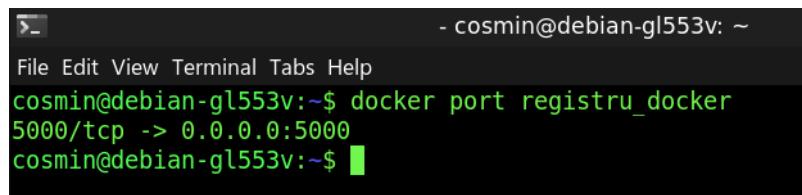
Verificarea porturilor expuse de un container Docker

Containerul **registrar_docker** pe care l-ați pornit mai sus expune portul 5000. Pentru a verifica și confirma acest lucru, se poate folosi comanda:

```
docker port <NUME_SAU_IDENTIFIER_CONTAINER>
```

În acest caz:

```
docker port registrar_docker
```



```
- cosmin@debian-gl553v: ~
File Edit View Terminal Tabs Help
cosmin@debian-gl553v:~$ docker port registrar_docker
5000/tcp -> 0.0.0.0:5000
cosmin@debian-gl553v:~$
```

Figura 9 - Verificarea porturilor expuse de un container Docker

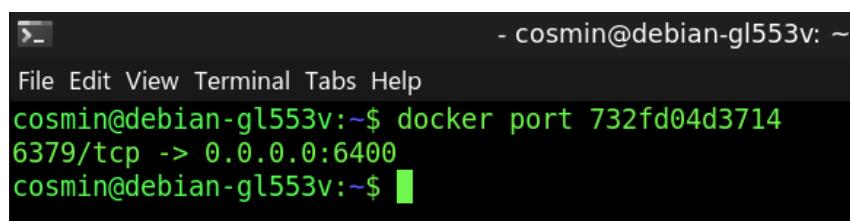
Comanda returnează: **5000/tcp → 0.0.0.0:5000**

- în partea stângă - portul utilizat în interiorul containerului și expus în afara acestuia (**5000**), împreună cu protocolul de transport utilizat (**TCP**, în acest caz)
- în partea dreaptă - adresa **0.0.0.0** semnifică orice adresă IP a sistemului, împreună cu portul mapat în exterior (în acest caz, tot **5000**).

Așadar, pentru ca utilizatorul să poată accesa serviciul expus de aplicația containerizată pe portul 5000 (în interiorul containerului), trebuie să accesze, în acest caz, tot portul 5000 de pe sistemul gazdă. Docker face maparea port gazdă ↔ port container.

Atenție! Portul 5000 al mașinii gazdă trebuie să fie neocupat de o altă aplicație!
Dacă este ocupat, puteți alege alt port sau închide aplicația care îl folosește.

Un alt exemplu de mapare de porturi, în care porturile implicate sunt diferite:



```
- cosmin@debian-gl553v: ~
File Edit View Terminal Tabs Help
cosmin@debian-gl553v:~$ docker port 732fd04d3714
6379/tcp -> 0.0.0.0:6400
cosmin@debian-gl553v:~$
```

Figura 10 - Verificarea porturilor mapate

Pentru cazul din figură, utilizatorul trebuie să accesze portul **6400** de pe mașina gazdă pentru a face trimitere la serviciul expus pe portul **6379** în interiorul containerului Docker cu identificatorul **732fd04d3714**.

Crearea unui microserviciu

Implementarea unui microserviciu de tip „Echo server”

Creați un proiect Kotlin/JVM, folosind gestionarul de proiect dorit (Maven sau Gradle). Folosiți pașii explicați în laboratorul 3 pentru scheletul de proiect Spring Boot, **dar nu adăugați Spring ca dependență** și nicio altă dependență legată de Spring. Folosiți doar arhetipul **kotlin-archetype-jvm** (în cazul **Maven**), respectiv selectați librăria **Kotlin-JVM** la crearea

proiectului (în cazul **Gradle**). Denumiți proiectul, spre exemplu, **HelloMicroservice**.

Creați un pachet (de exemplu: **com.sd.laborator**).

Codul sursă pentru acest microserviciu simplu conține un singur fișier. Creați următorul fișier sursă și adăugați codul ce urmează:

- **HelloMicroservice.kt**

```
package com.sd.laborator

import java.net.ServerSocket

fun main(args: Array<String>) {
    // se porneste un socket server TCP pe portul 1234 care asculta
    pentru conexiuni
    val server = ServerSocket(2000)
    println("Microserviciul se executa pe portul:
    ${server.localPort}")
    println("Se asteapta conexiuni...")

    while (true) {
        // se asteapta conexiuni din partea clientilor
        val client = server.accept()
        println("Client conectat:
        ${client.inetAddress.hostAddress}:${client.port}")

        // acest microserviciu simplu raspunde printr-un mesaj
        oricarui client se conecteaza
        client.getOutputStream().write("Hello from a dockerized
        microservice!\n".toByteArray())

        // dupa ce mesajul este trimis, se inchide conexiunea cu
        clientul
        client.close()
    }
}
```

Fișierul sursă conține o aplicație Kotlin minimală ce implementează un server TCP care ascultă pe portul **2000** pentru conexiuni. Aplicația așteaptă mesaje trimise din partea clienților care se conectează, iar pentru fiecare client conectat trimită înapoi un răspuns sub formă de mesaj „Hello...”. În final, conexiunea cu acel client este închisă.

Împachetarea și încapsularea microserviciului

Microserviciile sunt decuplate, de sine stătătoare, aşadar trebuie parcursi câțiva pași pentru a **împacheta** aplicația rezultată și a o **izola** într-un container.

În primul rând, configurați proiectul Maven / Gradle creat pentru a putea împacheta aplicația într-un artefact JAR de sine stătător, împreună cu toate dependențele:

Configurarea pentru împachetare a unui proiect Kotlin/JVM cu Maven

Pentru proiectele Kotlin/JVM construite cu **Maven**, împachetarea întregii aplicații, împreună cu toate dependențele necesare sub formă de artefact JAR se face astfel:

- adăugați următorul *plugin* în fișierul **pom.xml**:

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-assembly-plugin</artifactId>
```

```
<version>2.6</version>
<executions>
    <execution>
        <id>make-assembly</id>
        <phase>package</phase>
        <goals> <goal>single</goal> </goals>
        <configuration>
            <archive>
                <manifest>
                    <mainClass><CLASA_MAIN_AICI></mainClass>
                </manifest>
            </archive>
            <descriptorRefs>
                <descriptorRef>jar-with-dependencies</descriptorRef>
            </descriptorRefs>
        </configuration>
    </execution>
</executions>
</plugin>
```

- înlocuiți **<CLASA_MAIN_AICI>** cu numele clasei Kotlin în care se află funcția **main()**.

Atenție: cel mai probabil funcția **main()** rezidă într-un fișier Kotlin în afara unei clase definite de utilizator. Așadar, numele clasei principale (*main*) este format astfel:

<NUME_PACHET><NUME_FIȘIER_CU_FUNCTIA_MAIN>Kt

De exemplu, dacă aveți funcția **main()** scrisă în fișierul sursă **HelloMicroservice.kt**, care este pus în pachetul **com.sd.laborator**, atunci numele clasei principale (*main*) este următorul:

com.sd.laborator.HelloMicroserviceKt

Pentru împachetare, se folosește *lifecycle*-ul Maven **package**, care generează (cu *plugin*-ul de mai sus adăugat) 2 artefacte JAR în folder-ul **target**: unul care nu conține librăriile necesare împachetare, respectiv unul de tip „*fat jar*”, care încapsulează tot ce este nevoie pentru execuția separată, izolată a aplicației create. Veți avea nevoie de acesta din urmă, denumit:

<NUME_PROIECT>-<VERSIUNE_PROIECT>-jar-with-dependencies.jar

Consultați următoarea captură de ecran pentru un exemplu concret:

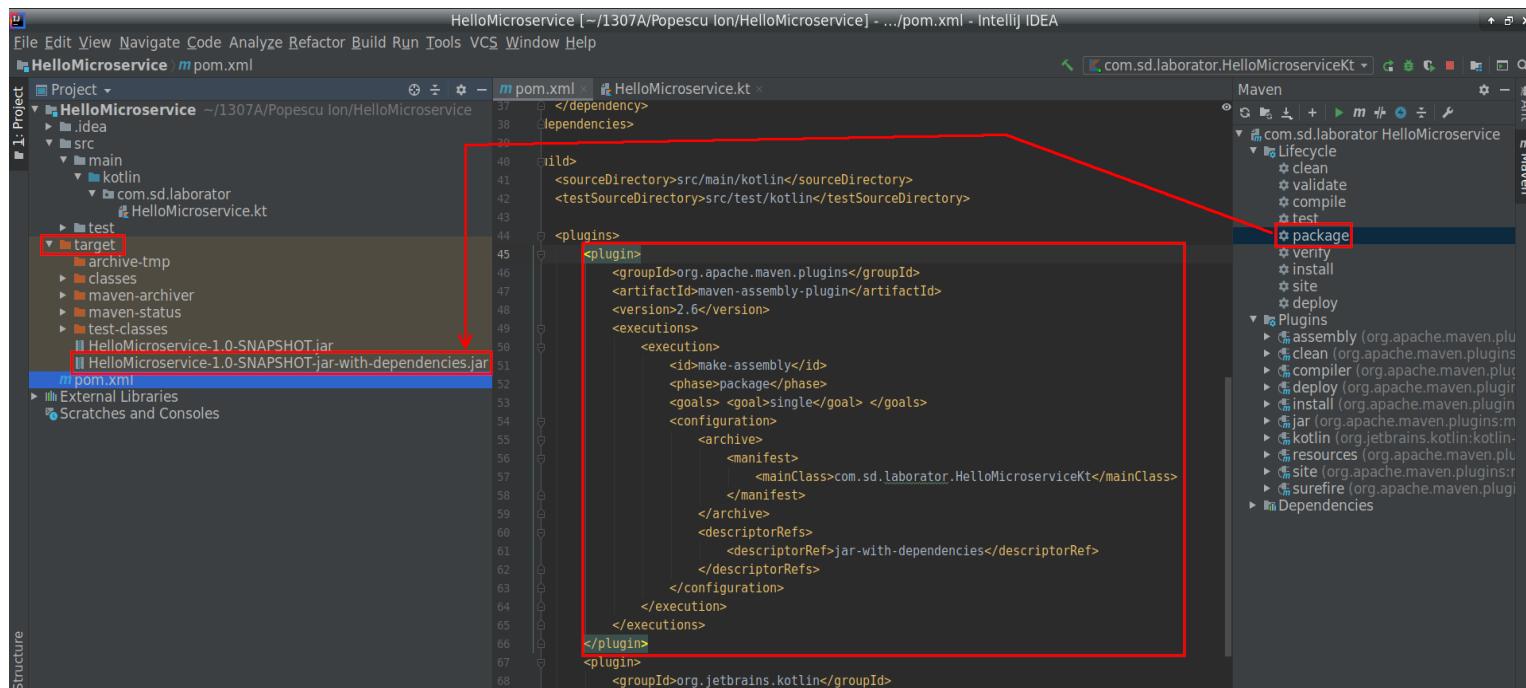


Figura 11 - Împachetarea unui proiect Kotlin/JVM creat cu Maven într-un fat jar

Că și consecință: puteți executa aplicația ca un întreg, folosind comanda următoare:

```
java -jar <NUME_ARTEFACT_JAR>.jar
```

(instrucțiunile sunt disponibile și la următorul URL: <https://kotlinlang.org/docs/reference/using-maven.html#self-contained-jar-file>)

Configurarea pentru împachetare a unui proiect Kotlin/JVM cu Gradle

Pentru proiectele Kotlin/JVM construite cu **Gradle**, împachetarea întregii aplicații, împreună cu toate dependențele necesare sub formă de artefact JAR se face astfel:

- adăugați următorul *plugin* în secțiunea **plugins** din fișierul **build.gradle**:

```
id "com.github.johnrengelman.shadow" version "5.2.0"
```

- adăugați următoarea secțiune nouă în fișierul **build.gradle**:

```
jar {
    manifest {
        attributes 'Main-Class': '<CLASA_MAIN_AICI>'
    }
}
```

- înlocuiți **<CLASA_MAIN_AICI>** cu numele clasei Kotlin în care se află funcția **main()**.

Atenție: cel mai probabil funcția **main()** rezidă într-un fișier Kotlin în afara unei clase definite de utilizator. Așadar, numele clasei principale (*main*) este format astfel:

<NUME_PACHET><NUME_FIȘIER_CU_FUNCTIA_MAIN>Kt

De exemplu, dacă aveți funcția **main()** scrisă în fișierul sursă **HelloMicroservice.kt**, care este pus în pachetul **com.sd.laborator**, atunci numele clasei principale (*main*) este următorul:

com.sd.laborator.HelloMicroserviceKt

Pentru împachetare, folosiți *task-ul Gradle shadow/shadowJar* (adăugat de *plugin-ul shadow*) pentru a genera un artefact JAR, care încapsulează întreaga aplicație, împreună cu dependențele acesteia. Artefactul este generat în folder-ul **build/libs** și este denumit astfel:

<NUME_PROJECT>-<VERSIUNE_PROJECT>-all.jar

Consultați următoarea captură de ecran pentru un exemplu concret:

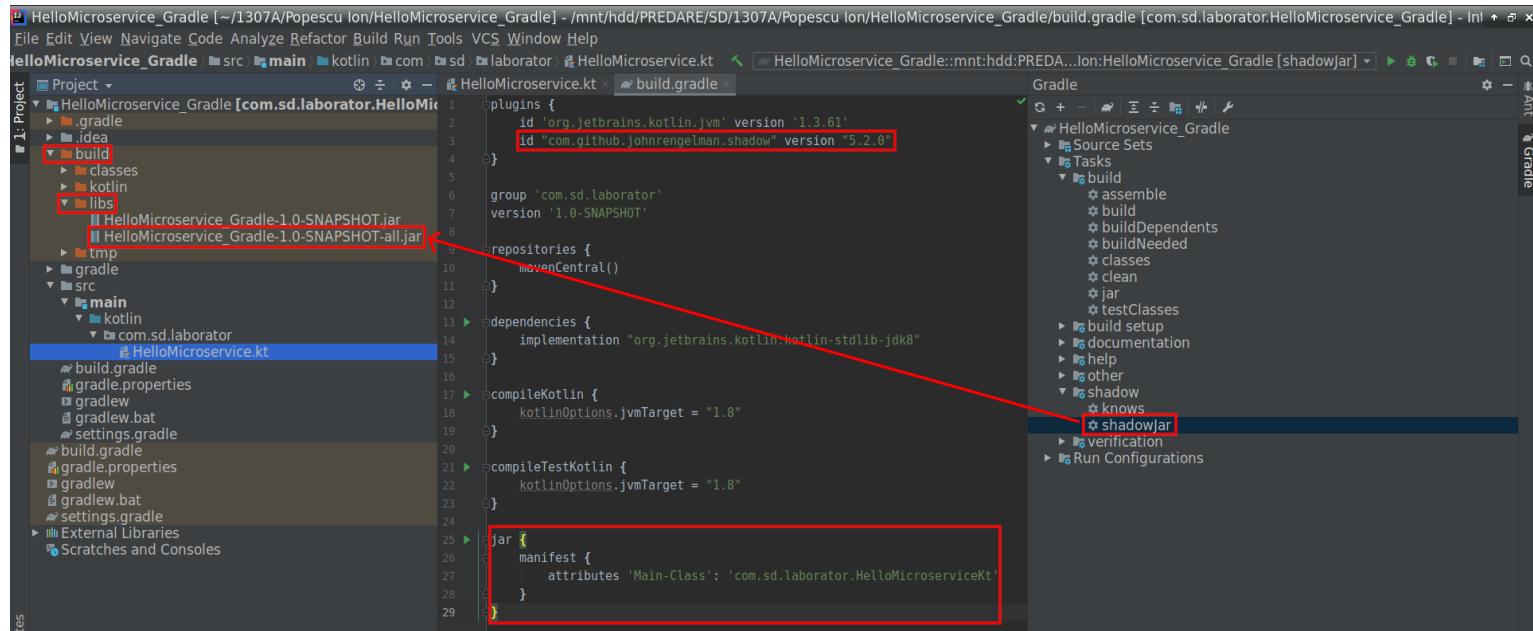


Figura 12 - Împachetarea unui proiect Kotlin/JVM creat cu Gradle într-un fat jar

Ca și consecință: puteți executa aplicația ca un întreg, folosind comanda următoare:

```
java -jar <NUME_ARTEFACT_JAR>.jar
```

Compilare aplicație

Comparați aplicația folosind *lifecycle-ul Maven compile* sau, în cazul Gradle, folosind *task-ul build*. Apoi, împachetați aplicația într-un fișier JAR, astfel:

- **Maven** - folosiți *lifecycle-ul package*, iar rezultatul împachetării îl găsiți în folder-ul **target**, denumit sub forma: **<NUME_PROJECT>-<VERSIUNE>-jar-with-dependencies.jar**

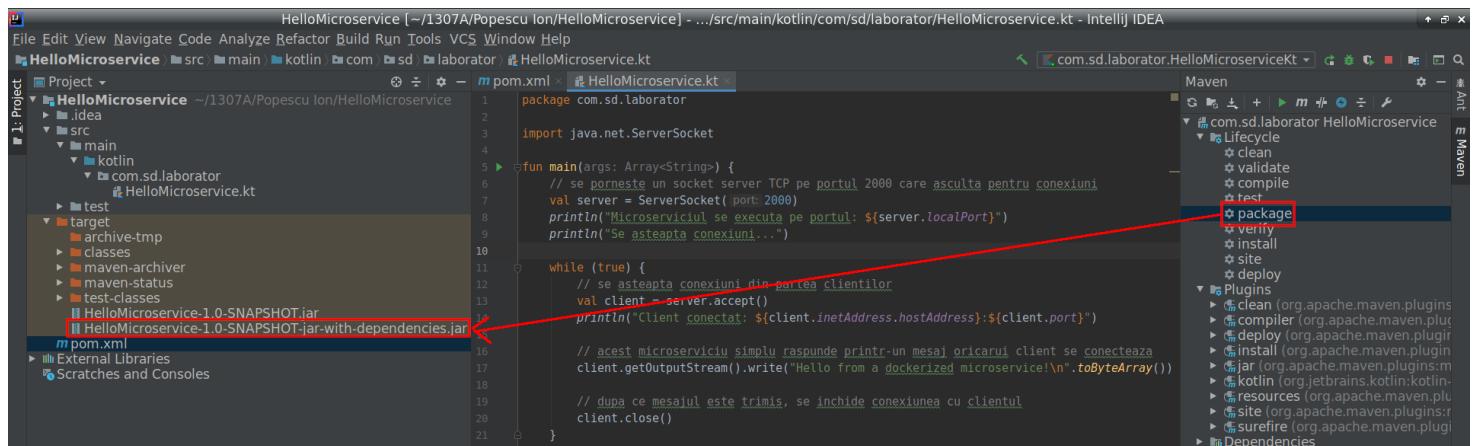


Figura 13 - Împachetarea microserviciului cu Maven

- **Gradle** - folosiți *task-ul shadowJar*, iar rezultatul împachetării îl găsiți în folder-ul **build/libs**, denumit sub forma: <NUME_PROJECT>-<VERSIUNE>-all.jar

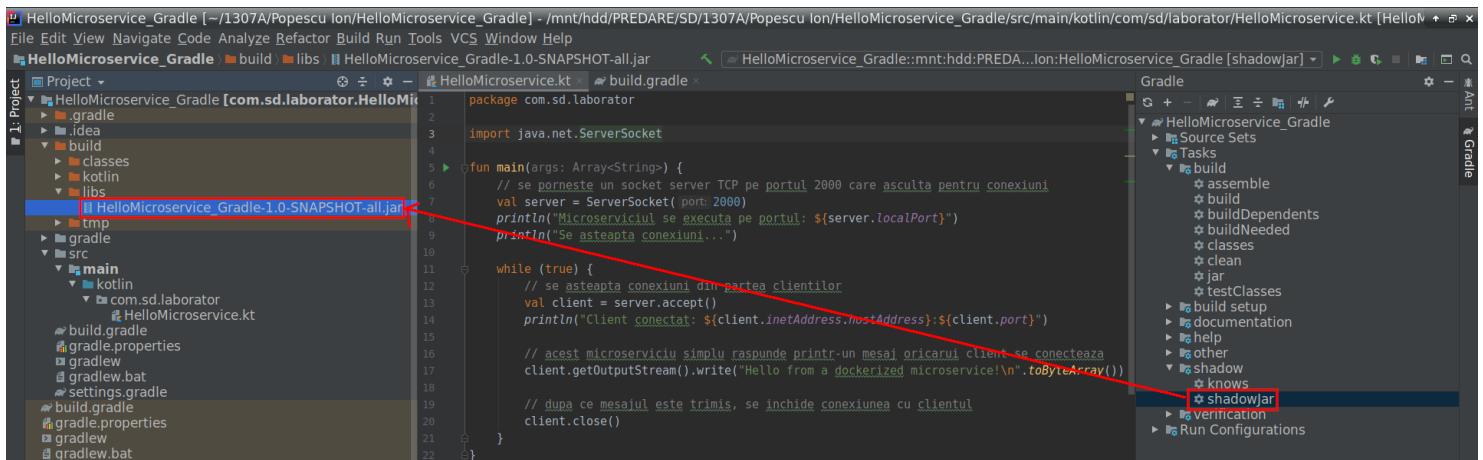


Figura 14 - Împachetarea microserviciului cu Gradle

Crearea unui fișier Dockerfile

În cazul în care nu aveți instalat *plugin-ul Docker* pentru IntelliJ, îl puteți instala din meniu **File → Settings → Plugins**, iar aici căutați „**docker**”.

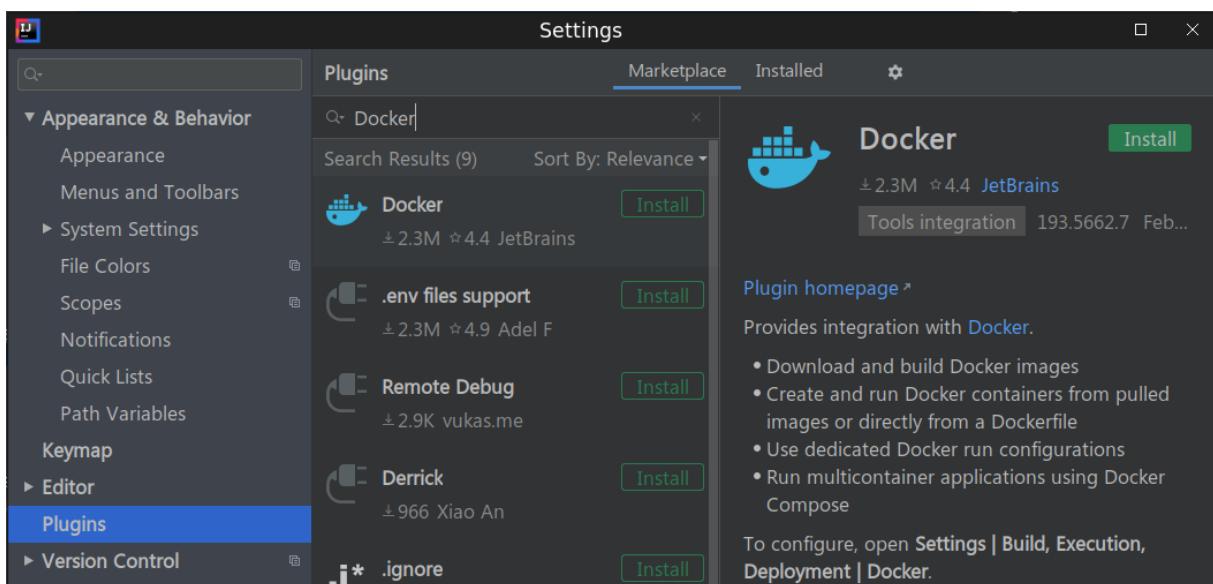


Figura 15 - Instalare plugin Docker în IntelliJ

Reporniți mediul de dezvoltare după instalarea *plugin-ului*.

Click dreapta pe numele proiectului → **New** → **File** → Introduceți ca nume „**Dockerfile**”. **Atenție: acest fișier trebuie să se numească obligatoriu astfel (inclusiv prima literă mare), deoarece este recunoscut de Docker, sub acest nume standard.**

Introduceți următorul conținut:

```
FROM openjdk:8-jdk-alpine
ADD target/<NUME_ARTEFACT_JAR> HelloMicroservice.jar

ENTRYPOINT ["java", "-jar", "HelloMicroservice.jar"]
```

Înlocuiți <NUME_ARTEFACT_JAR> corespunzător cu numele artefactului generat de

gestionarul de proiect ales.

The screenshot shows the IntelliJ IDEA interface with the following details:

- File Bar:** File Edit View Navigate Code Analyze Refactor Build Run Tools VCS Window Help
- Project Tree:** Shows the project structure under "HelloMicroservice" including ".idea", "src" (with "main" and "kotlin" subfolders), "test", "target", and "Dockerfile".
- Editor:** The "Dockerfile" tab is selected, displaying the following code:


```

      FROM openjdk:8-jdk-alpine
      ADD target/HelloMicroservice-1.0-SNAPSHOT-jar-with-dependencies.jar HelloMicroservice.jar
      ENTRYPOINT ["java", "-jar", "HelloMicroservice.jar"]
      
```
- Status Bar:** com.sd.lab

Figura 16 - Dockerfile pentru microserviciul creat cu Maven

Fișierul **Dockerfile** are rol de „fișier sursă” pentru imaginea Docker ce va încapsula microserviciul.

- Instrucțiunea **FROM** (care trebuie să fie neapărat prima din **Dockerfile**) este folosită pentru a specifica imaginea de bază peste care vor fi adăugate următoarele straturi. În acest caz, se folosește o imagine de bază ce conține **Alpine Linux** (o distribuție minimală de Linux bazată pe BusyBox), în care este instalat **OpenJDK 8**.
- Instrucțiunea **ADD** specifică fișiere externe care să fie adăugate în imaginea rezultată. Sintaxa este:

ADD <CALE_FIȘIER_PE_MAȘINA_GAZDĂ> <CALE_FIȘIER_ÎN_IMAGINE>

În fișierul **Dockerfile** exemplificat, se adaugă artefactul JAR rezultat în urma împachetării microserviciului în imaginea Docker, în folder-ul rădăcină al acesteia (/), implicit.

Atenție: fișierele trimise ca parametru de pe mașina gazdă sunt căutate relativ la același folder în care se află fișierul Dockerfile!

- Instrucțiunea **ENTRYPOINT** se configurează executabilul care va fi rulat de containerul respectiv. Practic, este o comandă executată imediat ce containerul respectiv pornește și reprezintă un vector care conține, ca prim element, un fișier executabil (sau un program existent în calea implicită - **\$PATH**), iar următoarele elemente sunt parametrii în linie de comandă, dați sub formă de șiruri de caractere **fără spații**.

În cazul de față, containerul va executa următoarea comandă:

```
java -jar HelloMicroservice.jar
```

Construirea imaginii Docker pe baza Dockerfile

Deschideți un terminal (folosind IntelliJ sau utilitarul furnizat de sistemul de operare) având calea curentă în folder-ul în care se află fișierul **Dockerfile**.

Execuați următoarea comandă:

```
docker build -t hello_microservice:v1 .
```

```

Terminal: Local x Local (2) x +
cosmin@debian-gl553v:/mnt/hdd/PREDARE/SD/1307A/Popescu Ion/HelloMicroservice$ docker build -t hello_microservice:v1 .
Sending build context to Docker daemon 21.29MB
Step 1/3 : FROM openjdk:8-jdk-alpine
--> a3562aa0b991
Step 2/3 : ADD target/HelloMicroservice-1.0-SNAPSHOT.jar HelloMicroservice.jar
--> Using cache
--> 52f69bf72149
Step 3/3 : ENTRYPOINT ["java","-jar", "HelloMicroservice.jar"]
--> Using cache
--> 47b8e45ba127
Successfully built 47b8e45ba127
Successfully tagged hello_microservice:v1
cosmin@debian-gl553v:/mnt/hdd/PREDARE/SD/1307A/Popescu Ion/HelloMicroservice$
```

Figura 17 - Construirea imaginii Docker care încapsulează microserviciul exemplificat

Comanda de mai sus construiește imaginea Docker specifică fișierului **Dockerfile** din folder-ul curent, strat cu strat (**Step x / 3** = stratul **x**). Parametrul **-t** a fost folosit pentru a eticheta imaginea rezultată sub formă de **nume:etichetă** (**hello_microservice:v1**). Ultimul parametru („.” = folder-ul curent) reprezintă calea de unde Docker preia fișierul **Dockerfile**, respectiv celelalte fișiere externe folosite în construirea imaginii.

O imagine Docker este identificată astfel: **<NUME_DEPOZIT>:<ETICHETĂ>**, sau, direct, prin identifierul unic asociat (**IMAGE_ID**).

Verificați lista de imagini disponibile local, folosind comanda:

```
docker image ls
```

File	Edit	View	Terminal	Tabs	Help
cosmin@debian-gl553v:~\$ docker image ls					
REPOSITORY	TAG	IMAGE ID	CREATED	SIZE	
hello microservice	v1	47b8e45ba127	25 hours ago	126MB	
<none>	<none>	ab450bf22081	25 hours ago	126MB	
registry	2	708bc6af7e5e	2 weeks ago	25.8MB	
jamtur01/redis	latest	5a953fa107c7	3 months ago	95.2MB	

Figura 18 - Verificarea existenței imaginii construite

Observați că imaginea construită apare în listă, însă este disponibilă doar local.

Încărcarea unei imagini într-un registru Docker

Registrul ar putea fi plasat pe o altă mașină din rețea / din afara rețelei! Pentru laborator, se folosește o instanță locală, doar pentru exemplificare.

Încercați să încărcați imaginea **hello_microservice:v1** în registrul Docker pe care l-ați creat și pornit anterior, disponibil la adresa: **localhost:5000**. Acest lucru se poate face cu comanda următoare:

```
docker push localhost:5000/hello_microservice:v1
```

Se specifică adresa registrului, împreună cu portul pe care acesta se execută în fața numelui imaginii Docker, elemente separate de un slash (/).

Veți primi o eroare:

```

File Edit View Terminal Tabs Help
cosmin@debian-gl553v:~$ docker push localhost:5000/hello_microservice:v1
The push refers to repository [localhost:5000/hello_microservice]
An image does not exist locally with the tag: localhost:5000/hello_microservice
cosmin@debian-gl553v:~$
```

Figura 19 - Eroare la încărcarea imaginii Docker

Acest lucru se întâmplă deoarece imaginea nu este etichetată corespunzător pentru a se conforma convenției de nume corespunzătoare cu registrul Docker personalizat, diferit de cel oficial. Imaginele care sunt încărcate în orice alt registru diferit de cel public (**registry-1.docker.io**) trebuie etichetate sub forma:

```
<ADRESĂ_REGISTRU>:<PORT_REGISTRU>/<NUME_DEPOZIT>:<ETICHETĂ>
```

Așadar, folosiți comanda următoare pentru a reeticheta imaginea creată:

```
docker tag hello_microservice:v1 localhost:5000/hello_microservice:v1
```

Verificați modificarea făcută:

```
docker image ls
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
hello_microservice	v1	47b8e45ba127	25 hours ago	126MB
localhost:5000/hello_microservice	v1	47b8e45ba127	25 hours ago	126MB
<none>	<none>	ab450bf22081	25 hours ago	126MB
registry	2	708bc6af7e5e	2 weeks ago	25.8MB
jamtur01/redis	latest	5a953fa107c7	3 months ago	95.2MB

Figura 20 - Imaginea creată, după reetichetare

Apoi încercați din nou să o încărcați în registru:

```
docker push localhost:5000/hello_microservice:v1
```

Așteptați până când se încarcă fiecare strat al sistemului de fișiere *Union File System*:

```
File Edit View Terminal Tabs Help  
cosmin@debian-gl553v:~$ docker push localhost:5000/hello_microservice:v1  
The push refers to repository [localhost:5000/hello_microservice]  
80df3d55d966: Pushed  
ceaf9e1ebef5: Pushing [=====] 18.11MB/99.29MB  
9b9b7f3d56a0: Pushed  
f1b5933fe4b5: Pushed
```

Figura 21 - Încărcarea unei imagini Docker în registrul local

Încărcarea completă, cu succes, arată astfel:

```
File Edit View Terminal Tabs Help  
cosmin@debian-gl553v:~$ docker push localhost:5000/hello_microservice:v1  
The push refers to repository [localhost:5000/hello_microservice]  
80df3d55d966: Pushed  
ceaf9e1ebef5: Pushed  
9b9b7f3d56a0: Pushed  
f1b5933fe4b5: Pushed  
v1: digest: sha256:ee8657a42241efc6a4621bd982c8c55d93777cf6815ee8b10c9780eca90811a8 size: 1159  
cosmin@debian-gl553v:~$
```

Figura 22 - Imagine Docker încărcată cu succes în registru

În acest moment, aveți microserviciul **hello_microservice** încapsulat și încărcat într-o imagine Docker disponibilă în registrul local.

Listarea imaginilor disponibile într-un registru Docker personalizat

Nu există nicio comandă specifică Docker pentru a lista ce imagini / depozite sunt disponibile (încărcate) într-un registru, aşa încât se folosește API-ul REST pus la dispoziție de

specificațiile *Docker Registry*: <https://docs.docker.com/registry/spec/api/>.

API-ul specifică faptul că o cerere HTTP de tip GET trimisă către server-ul ce găzduiește registrul Docker, având calea **/v2/_catalog** va returna un obiect ce conține lista de depozite disponibile în registrul respectiv. Execuați comanda următoare pentru a trimite o astfel de cerere:

```
curl -X GET http://localhost:5000/v2/_catalog
```

```
- cosmin@debian-gl553v: ~
File Edit View Terminal Tabs Help
cosmin@debian-gl553v:~$ curl -X GET http://localhost:5000/v2/_catalog
{"repositories":["hello_microservice"]}
cosmin@debian-gl553v:~$
```

Figura 23 - Listarea depozitelor disponibile într-un registru Docker

Depozitul „**hello_microservice**” poate conține mai multe imagini Docker - nu uitați că o imagine este identificată prin **<NUME_DEPOZIT>:<ETICHETĂ>**.

Pentru listarea tuturor etichetelor corespunzătoare unui depozit, conform API-ului REST, se trimit o cerere GET către calea **/v2/<NUME_DEPOZIT>/tags/list**, astfel:

```
curl -X GET http://localhost:5000/v2/hello_microservice/tags/list
```

```
- cosmin@debian-gl553v: ~
File Edit View Terminal Tabs Help
cosmin@debian-gl553v:~$ curl -X GET http://localhost:5000/v2/hello_microservice/tags/list
[{"name": "hello_microservice", "tags": ["v1"]}
cosmin@debian-gl553v:~$
```

Figura 24 - Listarea etichetelor unui depozit Docker dintr-un registru local

Se observă că se returnează un obiect care conține numele depozitului, alături de un vector de etichete disponibile. **Aceste comenzi sunt utile în cazul în care ați uitat numele imaginilor pe care le-ați încărcat, sau numele etichetelor asociate.**

Stergerea imaginilor Docker

Stergeți imaginea locală care încapsulează microserviciul, pentru a putea folosi imaginea încărcată în registrul Docker. Listați mai întâi imaginile locale:

```
docker image ls
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
hello_microservice	v1	47b8e45ba127	2 days ago	126MB
localhost:5000/hello_microservice	v1	47b8e45ba127	2 days ago	126MB
<none>	<none>	ab450bf22081	2 days ago	126MB
python	latest	efdecc2e377a	6 days ago	933MB
registry	2	708bc6af7e5e	2 weeks ago	25.8MB

Figura 25 - Listarea imaginilor Docker locale

Acum stergeti imaginea **localhost:5000/hello_microservice:v1**, utilizând una din comenziile următoare:

```
docker image rm localhost:5000/hello_microservice:v1
```

SAU:

```
docker rmi localhost:5000/hello_microservice:v1
```

```
cosmin@debian-gl553v:~$ docker image rm localhost:5000/hello_microservice:v1
Untagged: localhost:5000/hello_microservice:v1
Untagged: localhost:5000/hello_microservice@sha256:ee8657a42241efc6a4621bd982c8c55d93777cf6815ee8b10c9780eca90811a8
cosmin@debian-gl553v:~$ █
```

Figura 26 - Stergerea unei imagini Docker

Acest pas îl efectuați strict pentru scop demonstrativ, pentru ca mai departe să utilizați registrul în care această imagine Docker rezidă.

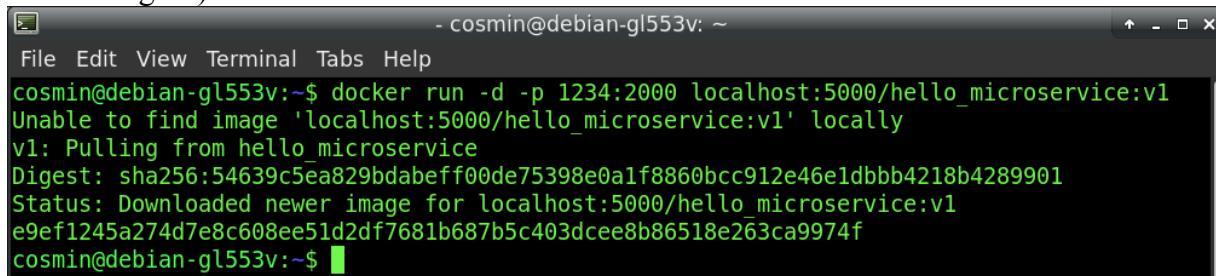
Pornirea microserviciului încărcat în registrul local

În momentul în care cereți pornirea unui container bazat pe o imagine Docker care nu există local, Docker va încerca să descarce acea imagine și să pornească apoi containerul. Nu este neapărat necesar să execuți o comandă **docker pull**, urmată de comanda **docker run** . . .

Porniți microserviciul disponibil în registrul Docker de pe mașina locală, expunând portul 2000 din container pe portul 1234 al mașinii locale:

```
docker run -d -p 1234:2000 localhost:5000/hello_microservice:v1
```

Dacă doriți să repornească automat în caz de eroare sau la repornirea sistemului, adăugați parametrul: **--restart=always** oriunde înainte de ultimul parametru al comenzi (înainte de numele imaginii).

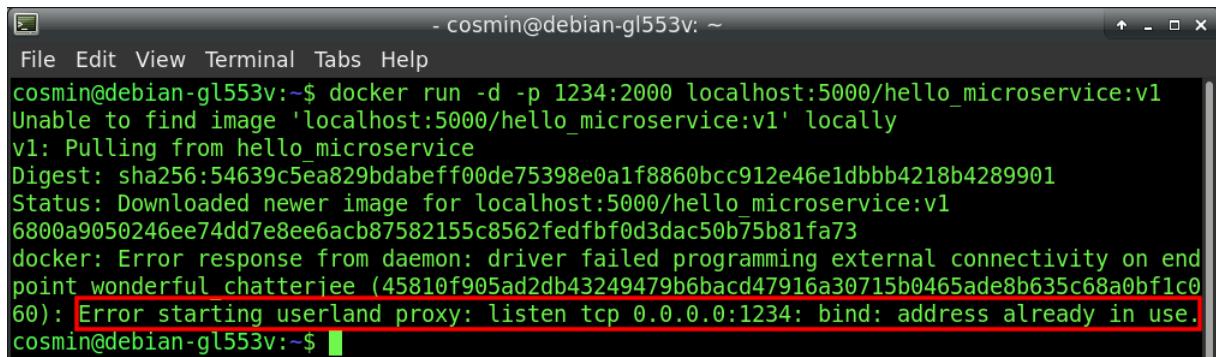


```
- cosmin@debian-gl553v: ~
File Edit View Terminal Tabs Help
cosmin@debian-gl553v:~$ docker run -d -p 1234:2000 localhost:5000/hello_microservice:v1
Unable to find image 'localhost:5000/hello_microservice:v1' locally
v1: Pulling from hello_microservice
Digest: sha256:54639c5ea829bdabeff00de75398e0a1f8860bcc912e46e1dbbb4218b4289901
Status: Downloaded newer image for localhost:5000/hello_microservice:v1
e9ef1245a274d7e8c608ee51d2df7681b687b5c403dcee8b86518e263ca9974f
cosmin@debian-gl553v:~$ █
```

Figura 27 - Pornirea microserviciului

Observați că Docker nu a găsit imaginea respectivă local, așa încât a descărcat-o automat din registrul cu care a fost prefixată în etichetare (**localhost:5000**). Apoi, a pornit un container din această imagine, în fundal, returnând un identificator pentru containerul respectiv.

Dacă primiți o eroare de acest tip:



```
- cosmin@debian-gl553v: ~
File Edit View Terminal Tabs Help
cosmin@debian-gl553v:~$ docker run -d -p 1234:2000 localhost:5000/hello_microservice:v1
Unable to find image 'localhost:5000/hello_microservice:v1' locally
v1: Pulling from hello_microservice
Digest: sha256:54639c5ea829bdabeff00de75398e0a1f8860bcc912e46e1dbbb4218b4289901
Status: Downloaded newer image for localhost:5000/hello_microservice:v1
6800a9050246ee74dd7e8ee6acb87582155c8562fedfbf0d3dac50b75b81fa73
docker: Error response from daemon: driver failed programming external connectivity on endpoint wonderful_chatterjee (45810f905ad2db43249479b6bacd47916a30715b0465ade8b635c68a0bf1c0
60): Error starting userland proxy: listen tcp 0.0.0.0:1234: bind: address already in use.
cosmin@debian-gl553v:~$ █
```

Figura 28 - Eroare la maparea porturilor

atunci portul pe care doriți să-l utilizați pe mașina gazdă este deja ocupat (de un alt container Docker sau de o aplicație pornită de utilizator). Aveți 2 variante: eliberați portul închizând aplicația sau containerul care îl folosește, sau porniți acest container pe alt port liber de pe mașină).

Dacă primiți o eroare de acest tip:

```
- cosmin@debian-gl553v: ~
File Edit View Terminal Tabs Help
cosmin@debian-gl553v:~$ docker run -d -p 1234:2000 localhost:5000/hello_microservice:v1
Unable to find image 'localhost:5000/hello_microservice:v1' locally
v1: Pulling from hello_microservice
Digest: sha256:54639c5ea829bdabeff00de75398e0a1f8860bcc912e46e1dbbb4218b4289901
Status: Downloaded newer image for localhost:5000/hello_microservice:v1
a03d01d801180054458a382904e84d7c173ed1574bab4ab38755564fbab82476
docker: Error response from daemon: driver failed programming external connectivity on endpoint suspicious_pasteur (e85150c0b0e8578e0bfed979eb1675aed7065b75b4fb916944fd992e83c053a4
()): Bind for 0.0.0.0:1234 failed: port is already allocated.
```

Figura 29 - Eroare la alocarea porturilor

atunci există un container care nu a pornit cu succes sau a fost oprit din diverse motive, care ocupă acel port (Docker a alocat acel port pentru containerul respectiv). Aflați care container ocupă portul respectiv, cu comanda:

```
docker ps -a
```

```
File Edit View Terminal Tabs Help
cosmin@debian-gl553v:~$ docker run -d -p 1234:2000 localhost:5000/hello_microservice:v1
Unable to find image 'localhost:5000/hello_microservice:v1' locally
v1: Pulling from hello_microservice
Digest: sha256:54639c5ea829bdabeff00de75398e0a1f8860bcc912e46e1dbbb4218b4289901
Status: Downloaded newer image for localhost:5000/hello_microservice:v1
a03d01d801180054458a382904e84d7c173ed1574bab4ab38755564fbab82476
docker: Error response from daemon: driver failed programming external connectivity on endpoint suspicious_pasteur (e85150c0b0e8578e0bfed979eb1675aed7065b75b4fb916944fd992e8
3c053a4): Bind for 0.0.0.0:1234 failed: port is already allocated.
cosmin@debian-gl553v:~$ docker ps -a
CONTAINER ID        IMAGE               COMMAND             CREATED            STATUS              PORTS                         NAMES
a03d01d80118        localhost:5000/hello_microservice:v1   "java -jar HelloMicr..."   58 seconds ago     Created
eur                localhost:5000/hello_microservice:v1   "java -jar HelloMicr..."   About a minute ago   Up About a minute   0.0.0.0:1234->2000/tcp   gifted_vaugan
6800a9050246        localhost:5000/hello_microservice:v1   "java -jar HelloMicr..."   3 minutes ago      Created
erjee               registry:2                  "/entrypoint.sh /etc..."   7 minutes ago      Up 7 minutes       0.0.0.0:5000->5000/tcp   wonderful_chatt
354d163a4a55        registry:2                  "/entrypoint.sh /etc..."   7 minutes ago      Up 7 minutes       registru_docker
```

Figura 30 - Container care ocupă portul 1234

Oriți (dacă este pornit) și ștergeți containerul care ocupă portul pe care doriți să îl folosiți:

```
docker stop <IDENTIFICATOR>
docker rm <IDENTIFICATOR>
```

În acest caz:

```
docker stop 9d2e57dfab9b
docker rm 9d2e57dfab9b
```

Verificați starea containerului **hello_microservice**:

```
docker ps
```

File	Edit	View	Terminal	Tabs	Help
cosmin@debian-gl553v:~\$ docker ps					
CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS
e9ef1245a274	localhost:5000/hello_microservice:v1	"java -jar HelloMicroservice.jar"	2 minutes ago	Up 2 minutes	0.0.0.0:1234->2000/tcp
354d163a4a55	registry:2	"/entrypoint.sh /etc..."	16 minutes ago	Up 16 minutes	0.0.0.0:5000->5000/tcp
cosmin@debian-gl553v:~\$					NAMES
					sharp_newton
					registrar_docker

Figura 31 - Verificarea stării containerelor în execuție

Containerul se află în execuție și se poate observa comanda specificată ca și punct de intrare (ENTRYPOINT): **java -jar HelloMicroservice.jar**. De asemenea, se observă în coloana **PORTS** faptul că orice serviciu se execută pe portul 2000 din container este accesibil din exteriorul acestuia pe portul 1234.

Logurile microserviciului se pot vizualiza astfel:

```
docker logs <NUME_SAU_ID_IMAGINE>
```

În acest caz:

```
docker logs e9ef1245a274
```

```
- cosmin@debian-gl553v: ~
File Edit View Terminal Tabs Help
cosmin@debian-gl553v:~$ docker logs e9ef1245a274
Microserviciul se executa pe portul: 2000
Se asteapta conexiuni...
cosmin@debian-gl553v:~$
```

Figura 32 - Vizualizarea log-urilor microserviciului

Testarea microserviciului - crearea unei aplicații client

Creați un proiect nou de tip Python din IntelliJ și denumiți-l, spre exemplu, **HelloClient**. Adăugați un nou folder **src**, în care, într-un fișier sursă **HelloClient.py** adăugați următorul cod:

```
import socket

if __name__ == "__main__":
    HOST, PORT = "localhost", 1234

    # creare socket TCP
    sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

    # conectare la microserviciu (acesta are rol de server și este
    # disponibil pe portul 1234)
    try:
        sock.connect((HOST, PORT))
    except ConnectionError:
        print("Eroare de conectare la microserviciu!")
        exit(1)

    # transmitere mesaj
    print("Trimit mesaj către microserviciu...")
    sock.sendall(b"Hello from client!", "utf-8")

    # primire raspuns
```

```

received = str(sock.recv(1024), "utf-8")
print("Raspuns de la HelloMicroservice: {}".format(received))

```

Codul conține o implementare clasică de tip client pentru un socket server. Aplicația client realizează o conexiune pe portul 1234 cu un server găsit la adresa **localhost** (în acest caz, pentru că microserviciul sănătății se execută pe mașina locală). După conectare, clientul trimite un mesaj (poate conține orice) și așteaptă un răspuns de la server, pe care îl afișează la consolă.

Rulați codul și veți primi răspunsul în consola Run a IntelliJ:

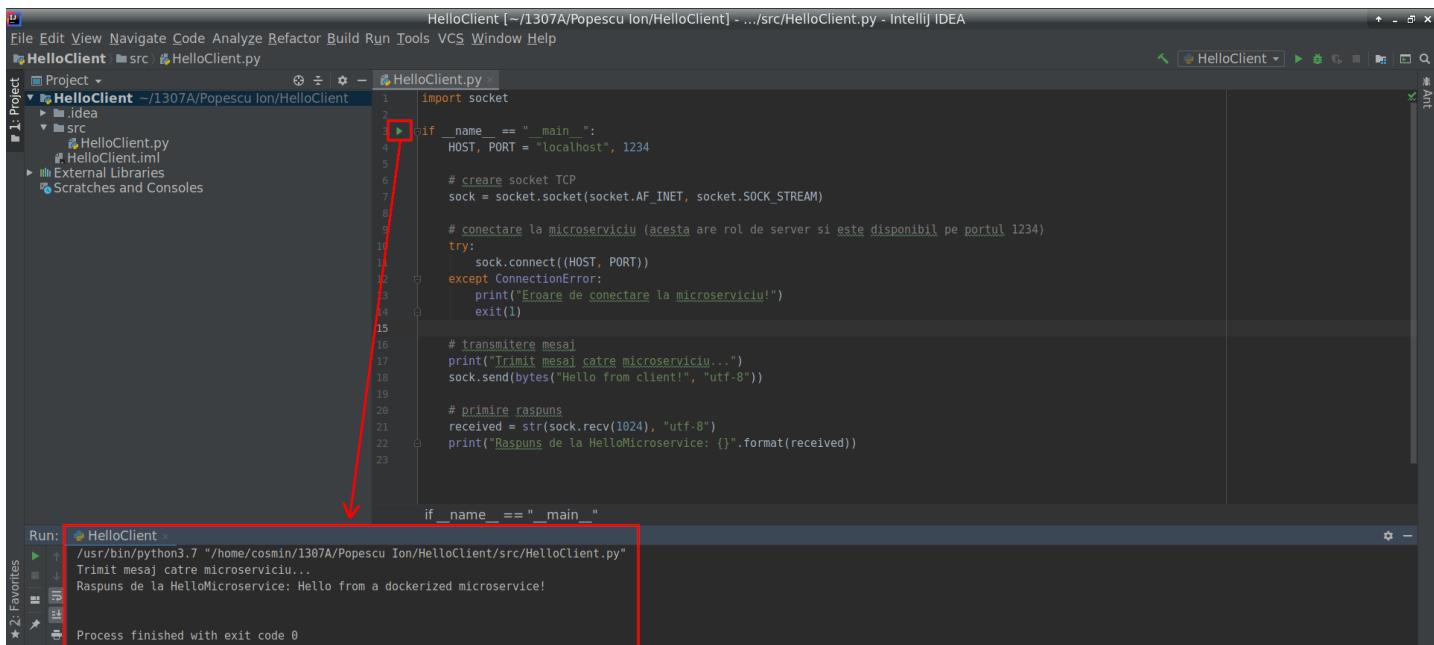


Figura 33 - Testarea microserviciului

Mesajele de pe server se pot verifica accesând log-urile containerului Docker:

```
docker logs e9ef1245a274
```

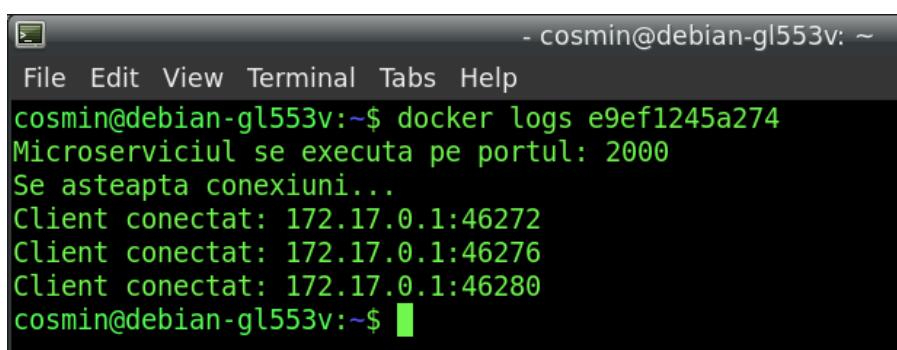


Figura 34 - Log-urile microserviciului după 3 conectări client

Oprirea unui container Docker (oprirea microserviciului)

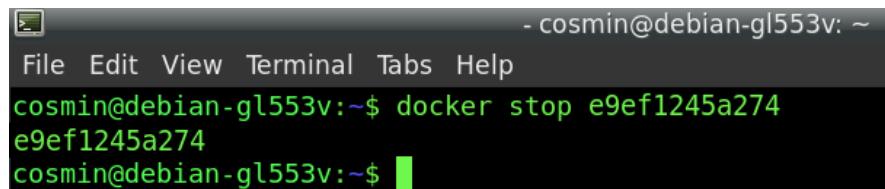
Microserviciul disponibil în containerul pe care l-ați pornit rămâne în execuție până când utilizatorul îl oprește sau îl sterge de tot.

Oprirea containerului se face cu următoarea comandă:

```
docker stop <NUME_SAU_ID_CONTAINER_ÎN_EXECUȚIE>
```

În acest caz:

```
docker stop e9ef1245a274
```



```
- cosmin@debian-gl553v: ~
File Edit View Terminal Tabs Help
cosmin@debian-gl553v:~$ docker stop e9ef1245a274
e9ef1245a274
cosmin@debian-gl553v:~$
```

Figura 35 - Oprirea unui container Docker

Se poate verifica starea containerului folosind comanda:

```
docker ps -a
```

(nu uitați că, dacă nu folosiți parametrul **-a**, se listează doar containerele **în execuție**, deci containerul de mai sus nu ar apărea în lista returnată de comandă)

File	Edit	View	Terminal	Tabs	Help	
cosmin@debian-gl553v:~\$ docker ps -a						
CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
e9ef1245a274	localhost:5000/hello_microservice:v1	"java -jar HelloMicr..."	4 minutes ago	Exited (143) 2 seconds ago		optimistic_maha
vira	531d2369e729	"java -jar HelloMicr..."	23 minutes ago	Created		thirsty_nightin
33e091dbd135	531d2369e729	"java -jar HelloMicr..."	30 minutes ago	Created		suspicious_past
gale	531d2369e729	"java -jar HelloMicr..."	33 minutes ago	Created		wonderful_chatt
a03d01d80118	531d2369e729	"java -jar HelloMicr..."	33 minutes ago	Created		
eur	531d2369e729	"java -jar HelloMicr..."	33 minutes ago	Created		
6800a9050246	531d2369e729	"java -jar HelloMicr..."	33 minutes ago	Created		
arie						

Figura 36 - Verificarea stării containerelor oprite

Pentru a reporni containerul, pur și simplu folosiți comanda:

```
docker start <NUME_SAU_ID_CONTAINER_OPRIT>
```

În acest caz:

```
docker start e9ef1245a274
```

Atenție: nu mai este nevoie de eventualii parametri trimiși prima dată când acest container a fost creat (parametrii de la comanda `docker run ...`), deoarece sunt reținuți de container și persistă la repornire!

Interoperabilitate microservicii - modelarea comunicației student-profesor

Pentru a exresa în continuare lucrul cu microservicii independent instalabile și a ilustra interoperabilitatea, se exemplifică proiectarea și implementarea unei aplicații care modelează comunicarea dintre studenți și profesori. Profesorul pune întrebări studenților, iar studenții răspund în cazul în care știu răspunsul la întrebare.

Proiectarea microserviciilor

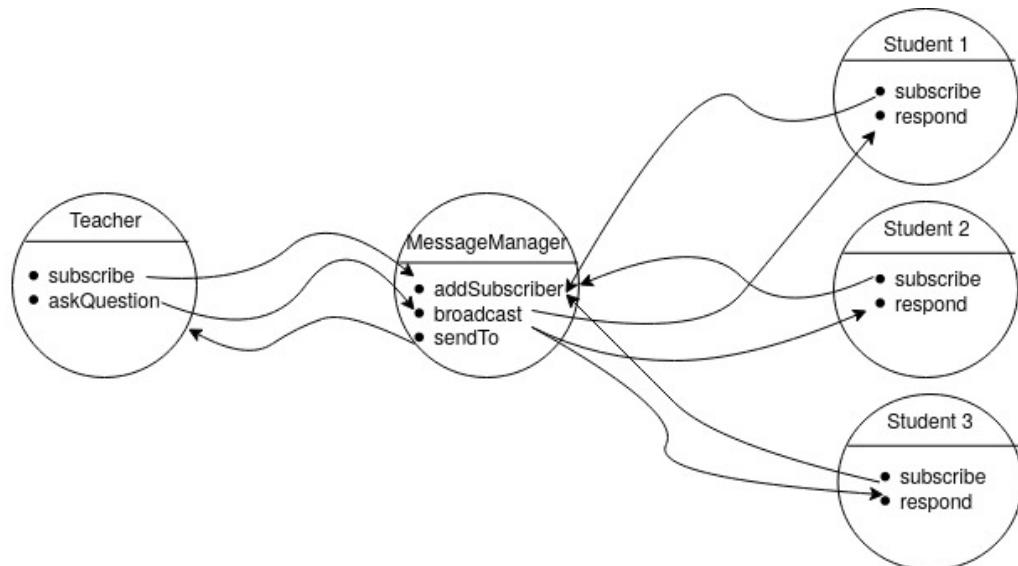


Figura 37 - Diagrama de microservicii

Se observă că acest model folosește coregrafia de microservicii: **MessageManager** este un microserviciu intermediar care mediază mesajele trimise între celelalte entități. Fluxul de *business* începe de la microserviciul **Teacher** (apelat de aplicația client). Acesta, împreună cu toate microserviciile de tip **Student**, se înscriv ca *subscribers* la **MessageManager**, pentru a putea participa la comunicațiile implicate (de aici mesajele de tip **subscribe**).

Profesorul (microserviciul **Teacher**) pune o întrebare sub formă de mesaj trimis microserviciului **MessageManager**. Mesajul are forma:

întrebare <DESTINATAR_RĂSPUNS> <CONTINUT_ÎNTREBARE>

MessageManager redirecționează mesajul către toți cei înscrisi (*subscribers*) la comunicație, mai puțin emițătorul respectivului mesaj, adică face **broadcast**. În cazul în care cel ce primește mesajul redirecționat știe să răspundă la întrebare, formează un mesaj răspuns cu următorul format:

răspuns <DESTINATAR_RĂSPUNS> <CONTINUT_RĂSPUNS>

Fiecare entitate **Student** știe să răspundă doar la anumite întrebări.

Atunci când **MessageManager** primește un răspuns de la un microserviciu **Student**, îl trimite înapoi destinatarului (adică microserviciului **Teacher**, în acest caz). Dacă niciun student nu știe răspunsul la întrebare, nu se trimite niciun răspuns, iar **Teacher** își va da seama că nu primește răspuns atunci când expiră un timp de dinainte stabilit.

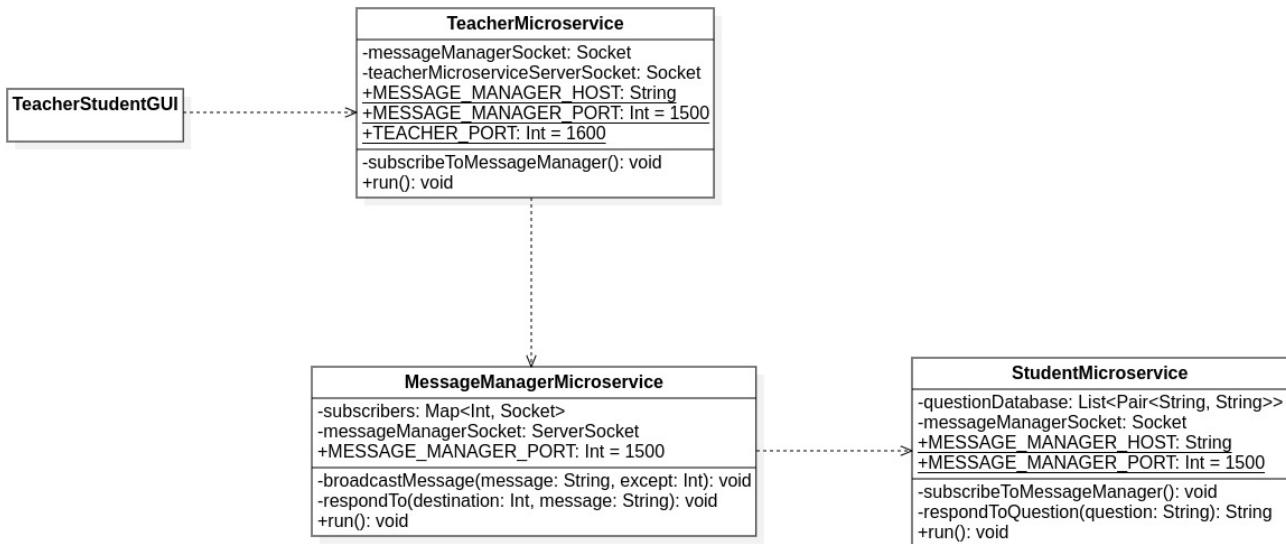


Figura 38 - Diagrama de clase

Implementarea microserviciilor

Pentru fiecare microserviciu în parte, creați un proiect Maven / Gradle de tip Kotlin/JVM. Nu uitați de configurarea proiectului pentru împachetare în fișier JAR cu tot cu dependențe.

StudentMicroservice

```

package com.sd.laborator

import java.io.BufferedReader
import java.io.File
import java.io.InputStreamReader
import java.lang.Exception
import java.net.Socket
import kotlin.concurrent.thread
import kotlin.system.exitProcess

class StudentMicroservice {
    // intrebarile și răspunsurile sunt menținute într-o listă de
    perechi de forma:
    // [<INTREBARE 1, RASPUNS 1>, <INTREBARE 2, RASPUNS 2>, ... ]
    private lateinit var questionDatabase: MutableList<Pair<String,
    String>>
    private lateinit var messageManagerSocket: Socket

    init {
        val databaseLines: List<String> =
File("questions_database.txt").readLines()
        questionDatabase = mutableListOf()

        /*
        "baza de date" cu intrebări și răspunsuri este de forma:

        <INTREBARE_1>\n
        <RASPUNS_INTREBARE_1>\n
        <INTREBARE_2>\n
        ...
        */
    }
}
  
```

```

<RASPUNS_INTREBARE_2>\n
...
 */
for (i in 0..(databaseLines.size - 1) step 2) {
    questionDatabase.add(Pair(databaseLines[i],
databaseLines[i + 1]))
}
}

companion object Constants {
    // pentru testare, se foloseste localhost. pentru deploy,
server-ul socket (microserviciul MessageManager) se identifica dupa un
"hostname"
    // acest hostname poate fi trimis (optional) ca variabila de
mediu
    val MESSAGE_MANAGER_HOST =
System.getenv("MESSAGE_MANAGER_HOST") ?: "localhost"
    const val MESSAGE_MANAGER_PORT = 1500
}

private fun subscribeToMessageManager() {
    try {
        messageManagerSocket = Socket(MESSAGE_MANAGER_HOST,
MESSAGE_MANAGER_PORT)
        println("M-am conectat la MessageManager!")
    } catch (e: Exception) {
        println("Nu ma pot conecta la MessageManager!")
        exitProcess(1)
    }
}

private fun respondToQuestion(question: String): String? {
    questionDatabase.forEach {
        // daca se gaseste raspunsul la intrebare, acesta este
returnat apelantului
        if (it.first == question) {
            return it.second
        }
    }
    return null
}

public fun run() {
    // microserviciul se inscrie in lista de "subscribers" de la
MessageManager prin conectarea la acesta
    subscribeToMessageManager()

    println("StudentMicroservice se executa pe portul:
${messageManagerSocket.localPort}")
    println("Se asteapta mesaje...")

    val bufferedReader =
BufferedReader(InputStreamReader(messageManagerSocket.inputStream))

    while (true) {
        // se asteapta intrebari trimise prin intermediarul
"MessageManager"
        val response = bufferedReader.readLine()
    }
}

```

```

        if (response == null) {
            // daca se primeste un mesaj gol (NULL), atunci
            inseamna ca cealalta parte a socket-ului a fost inchisa
            println("Microserviciul MessageService
(${messageManagerSocket.port}) a fost oprit.")
            bufferReader.close()
            messageManagerSocket.close()
            break
        }

        // se foloseste un thread separat pentru tratarea
        intrebarii primite
        thread {
            val (messageType, messageDestination, messageBody) =
            response.split(" ", limit = 3)

            when(messageType) {
                // tipul mesajului cunoscut de acest microserviciu
                este de forma:
                // intrebare <DESTINATIE_RASPUNS>
<CONTINUT_INTREBARE>
                "intrebare" -> {
                    println("Am primit o intrebare de la
$messageDestination: \\"${messageBody}\\"")
                    var responseToQuestion =
                    respondToQuestion(messageBody)
                    responseToQuestion?.let {
                        responseToQuestion = "raspuns
$messageDestination $it"
                        println("Trimit raspunsul:
\"${response}\\"")

                        messageManagerSocket.getOutputStream().write((responseToQuestion +
"\n").toByteArray())
                    }
                }
            }
        }
    }

    fun main(args: Array<String>) {
        val studentMicroservice = StudentMicroservice()
        studentMicroservice.run()
    }
}

```

Acest microserviciu citește dintr-un fișier denumit **questions_database.txt** lista cu întrebări și răspunsuri pe care el o deține. Comunicația se face prin socket-uri TCP, entitățile Student conectându-se la **MessageManager** pe un port stabilit (în acest caz, 1500).

StudentMicroservice se conectează la un socket server TCP pornit de microserviciul **MessageManager**. Așadar, pentru stabilirea conexiunii, server-ul trebuie identificat în rețea în care microserviciile funcționează, prin perechea **<IP / hostname, port>**. Când testați microserviciile în IntelliJ, puteți folosi **localhost** pe post de nume al gazdei server-ului,

deoarece acestea sunt pornite în rețeaua locală (loopback), iar microserviciile se pot identifica unele pe altele prin: **<localhost, port>**. **Când veți încapsula microserviciile în containere Docker și le veți porni independent, această abordare nu mai funcționează**: Docker identifică un container în rețeaua virtuală pe baza numelui acestuia (care devine **hostname**). De aceea, pentru a rezolva această problemă, se citește o variabilă de mediu numită **MESSAGE_MANAGER_HOST**, ce conține numele gazdei (*hostname*) pentru server-ul socket deschis în **MessageManager**. Dacă această variabilă nu e setată, atunci se revine la „localhost”. **Atunci când Teacher sau Student încearcă să se conecteze la microserviciul MessageManager, vor folosi hostname-ul pe post de identificator al server-ului socket!**

```
val MESSAGE_MANAGER_HOST = System.getenv("MESSAGE_MANAGER_HOST") ?:  
"localhost"
```

Se folosesc fire de execuție (*thread-uri*) separate pentru tratarea fiecărui mesaj primit pe socket cu scopul de a nu bloca *thread-ul* principal și a putea primi mai multe mesaje asincron.

Urmăriți comentariile din cod pentru explicații suplimentare legate de implementare.

MessageManagerMicroservice

```
package com.sd.laborator

import java.io.BufferedReader
import java.io.InputStreamReader
import java.net.ServerSocket
import java.net.Socket
import kotlin.concurrent.thread

class MessageManagerMicroservice {
    private val subscribers: HashMap<Int, Socket>
    private lateinit var messageManagerSocket: ServerSocket

    companion object Constants {
        const val MESSAGE_MANAGER_PORT = 1500
    }

    init {
        subscribers = hashMapOf()
    }

    private fun broadcastMessage(message: String, except: Int) {
        subscribers.forEach {
            it.takeIf { it.key != except }?.value?.getOutputStream()?.write((message +
"\\n").toByteArray())
        }
    }

    private fun respondTo(destination: Int, message: String) {
        subscribers[destination]?.getOutputStream()?.write((message +
"\\n").toByteArray())
    }

    public fun run() {
        // se porneste un socket server TCP pe portul 1500 care
        asculta pentru conexiuni
        messageManagerSocket = ServerSocket(MESSAGE_MANAGER_PORT)
```

```

        println("MessageManagerMicroservice se executa pe portul:
${messageManagerSocket.localPort}")
        println("Se asteapta conexiuni si mesaje...")

        while (true) {
            // se asteapta conexiuni din partea clientilor subscriberi
            val clientConnection = messageManagerSocket.accept()

            // se porneste un thread separat pentru tratarea
            conexiunii cu clientul
            thread {
                println("Subscriber conectat:
${clientConnection.inetAddress.hostAddress}:${clientConnection.port}")

                // adaugarea in lista de subscriberi trebuie sa fie
                atomica!
                synchronized(subscribers) {
                    subscribers[clientConnection.port] =
clientConnection
                }

                val bufferedReader =
BufferedReader(InputStreamReader(clientConnection.inputStream))

                while (true) {
                    // se citeste raspunsul de pe socketul TCP
                    val receivedMessage = bufferedReader.readLine()

                    // daca se primeste un mesaj gol (NULL), atunci
                    inseamna ca cealalta parte a socket-ului a fost inchisa
                    if (receivedMessage == null) {
                        // deci subscriber-ul respectiv a fost
                        deconectat
                        println("Subscriber-ul
${clientConnection.port} a fost deconectat.")
                        synchronized(subscribers) {
                            subscribers.remove(clientConnection.port)
                        }
                        bufferedReader.close()
                        clientConnection.close()
                        break
                    }

                    println("Primit mesaj: $receivedMessage")
                    val (messageType, messageDestination, messageBody) =
receivedMessage.split(" ", limit = 3)

                    when (messageType) {
                        "intrebare" -> {
                            // tipul mesajului de tip intrebare este
                            de forma:
                            // intrebare <DESTINATIE_RASPUNS>
<CONTINUT_INTREBARE>
                            broadcastMessage("intrebare
${clientConnection.port} $messageBody", except =
clientConnection.port)
                    }
                }
            }
        }
    }
}

```

```

        }
        "raspuns" -> {
            // tipul mesajului de tip raspuns este de
forma:
            // raspuns <CONTINUT_RASPUNS>
            respondTo(messageDestination.toInt(),
messageBody)
        }
    }
}
}

fun main(args: Array<String>) {
    val messageManagerMicroservice = MessageManagerMicroservice()
    messageManagerMicroservice.run()
}

```

MessageManager pornește un socket server TCP și așteaptă conexiuni din partea celorlalte entități (**Teacher** sau **Student**). În momentul în care primește o conexiune, o înscrie în lista internă de *subscribers*. Fiecare *subscriber* este identificat în mod unic prin portul pe care conexiunea socket a fost stabilită cu acesta. **Acest lucru funcționează pentru că toate microserviciile sunt executate local. Dacă se află pe mașini diferite, identificatorul trebuie să fie perechea <IP, port>**.

Conexiunile cu entitățile înscrise în comunicație sunt tratate în fire de execuție separate, pentru a putea trata mai multe conexiuni simultan. Dezavantajul este că, deoarece se lucrează din *thread*-uri separate, accesul la lista de înscriși trebuie să fie atomic.

Thread-ul de tratare a conexiunii nu face altceva decât să aștepte încontinuu mesaje din partea partenerului de conexiune (celălalt capăt al socket-ului). Atunci când se primește un mesaj, se decodifică și se verifică tipul acestuia. În funcție de tipul mesajului, se ia o decizie: ori se trimit răspunsul unei întrebări înapoi la destinatar, ori se trimit întrebarea primită tuturor entităților înscrise.

TeacherMicroservice

```

package com.sd.laborator

import java.io.BufferedReader
import java.io.InputStreamReader
import java.net.*
import kotlin.concurrent.thread
import kotlin.system.exitProcess

class TeacherMicroservice {
    private lateinit var messageManagerSocket: Socket
    private lateinit var teacherMicroserviceServerSocket: ServerSocket

    companion object Constants {
        // pentru testare, se foloseste localhost. pentru deploy,
server-ul socket (microserviciul MessageManager) se identifica după un
"hostname"
        // acest hostname poate fi trimis (optional) ca variabilă de
mediu
        val MESSAGE_MANAGER_HOST =

```

Sisteme Distribuite - Laborator 8

```
System.getenv("MESSAGE_MANAGER_HOST") ?: "localhost"
    const val MESSAGE_MANAGER_PORT = 1500
    const val TEACHER_PORT = 1600
}

private fun subscribeToMessageManager() {
    try {
        messageManagerSocket = Socket(MESSAGE_MANAGER_HOST,
MESSAGE_MANAGER_PORT)
        messageManagerSocket.soTimeout = 3000
        println("M-am conectat la MessageManager!")
    } catch (e: Exception) {
        println("Nu ma pot conecta la MessageManager!")
        exitProcess(1)
    }
}

public fun run() {
    // microserviciul se inscrie in lista de "subscribers" de la
MessageManager prin conectarea la acesta
    subscribeToMessageManager()

    // se porneste un socket server TCP pe portul 1600 care
asculta pentru conexiuni
    teacherMicroserviceServerSocket = ServerSocket(TEACHER_PORT)

    println("TeacherMicroservice se executa pe portul:
${teacherMicroserviceServerSocket.localPort}")
    println("Se asteapta cereri (intrebari)...")

    while (true) {
        // se asteapta conexiuni din partea clientilor ce doresc
sa puna o intrebare
        // (in acest caz, din partea aplicatiei client GUI)
        val clientConnection =
teacherMicroserviceServerSocket.accept()

        // se foloseste un thread separat pentru tratarea fiecarei
conexiuni client
        thread {
            println("S-a primit o cerere de la:
${clientConnection.inetAddress.hostAddress}:${clientConnection.port}")

            // se citeste intrebarea dorita
            val clientBufferedReader =
BufferedReader(InputStreamReader(clientConnection.inputStream))
            val receivedQuestion = clientBufferedReader.readLine()

            // intrebarea este redirectionata catre microserviciul
MessageManager
            println("Trimite catre MessageManager: ${"intrebare
${messageManagerSocket.localPort} $receivedQuestion\n"}")

messageManagerSocket.getOutputStream().write(("intrebare
${messageManagerSocket.localPort} $receivedQuestion\n").toByteArray())
    
```

```

        // se asteapta raspuns de la MessageManager
        val messageManagerBufferReader =
BufferedReader(InputStreamReader(messageManagerSocket.getInputStream()))
        try {
            val receivedResponse =
messageManagerBufferReader.readLine()

            // se trimit raspunsul inapoi clientului apelant
            println("Am primit raspunsul:
\"$receivedResponse\"")

clientConnection.getOutputStream().write((receivedResponse +
"\n").toByteArray())
        } catch (e: SocketTimeoutException) {
            println("Nu a venit niciun raspuns in timp util.")
            clientConnection.getOutputStream().write("Nu a
raspuns nimeni la intrebare\n".toByteArray())
        } finally {
            // se inchide conexiunea cu clientul
            clientConnection.close()
        }
    }
}

fun main(args: Array<String>) {
    val teacherMicroservice = TeacherMicroservice()
    teacherMicroservice.run()
}

```

TeacherMicroservice se conectează inițial la entitatea **MessageManager** pentru a asigura funcționarea: delegarea întrebărilor primite de la aplicația client și primirea răspunsurilor. De asemenea, își deschide un socket server TCP propriu pe un port stabilit (în acest caz, 1600) pentru a asigura comunicarea cu aplicația client. Pot exista mai mulți clienți care să pună întrebări în același timp, de aceea, și în acest caz, fiecare conexiune din exterior este tratată într-un fir de execuție separat.

Atunci când un client se conectează (aplicația trimite un mesaj de tipul „Vreau un răspuns la întrebarea X”), **TeacherMicroservice** preia întrebarea și formează corect mesajul sub formatul pe care îl așteaptă **MessageManager**:

intrebare <DESTINATAR_RĂSPUNS> <CONTINUT_MESAJ_LA_CLIENT>

Acest lucru demonstrează că microserviciile pot comunica sub diverse protocoale, iar prin coregrafie se poate adapta un mesaj în funcție de caz.

TeacherMicroservice așteaptă maxim 3 secunde pentru primirea unui răspuns pentru o anumită întrebare (**messageManagerSocket.soTimeout = 3000**), după care înștiințează clientul că „Nu știe nimeni să răspundă”.

Împachetarea și instalarea microserviciilor

Fiecare din aceste microservicii le împachetați într-un artefact JAR cu metoda prezentată anterior în laborator. Apoi, creați câte un fișier **Dockerfile** pentru fiecare microserviciu, astfel:

StudentMicroservice

```
FROM openjdk:8-jdk-alpine
```

```
ADD target/StudentMicroservice-1.0-SNAPSHOT-jar-with-dependencies.jar  
StudentMicroservice.jar  
ADD questions_database.txt questions_database.txt  
  
ENTRYPOINT ["java", "-jar", "StudentMicroservice.jar"]
```

Înlocuiți numele artefactului corespunzător cu fișierul JAR generat de proiect, în acest caz: **StudentMicroservice-1.0-SNAPSHOT-jar-with-dependencies.jar**

Nu uitați că acest microserviciu citește din fișierul **questions_database.txt** lista cu întrebări sub forma:

```
<ÎNTREBARE_1>  
<RĂSPUNS_1>  
<ÎNTREBARE_2>  
<RĂSPUNS_2>  
...
```

De aceea, creați un astfel de fișier în folder-ul rădăcină al proiectului, alături de **Dockerfile** (în folder-ul rădăcină se pot pune fișiere citite de programe prin cale relativă, și deci este util pentru testare).

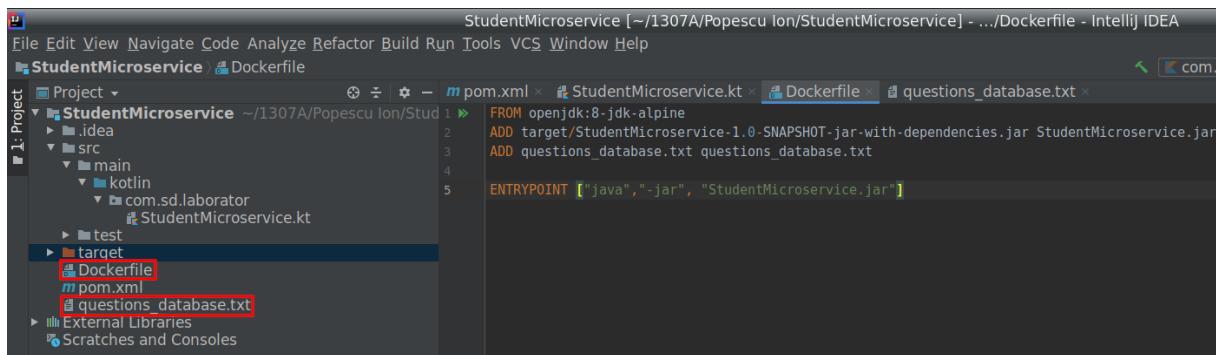


Figura 39 - Dockerfile și fișier cu date pentru StudentMicroservice

Simularea mai multor tipuri de microservicii **Student** se poate face prin variația întrebărilor din fișierul cu baza de date. Adică: primul microserviciu încapsulat într-o imagine Docker va conține prima versiune a **questions_database.txt** (puteți eticheta direct cu prefixul registruului local, pentru a trece peste pasul de reetichetare):

```
docker build -t localhost:5000/student_microservice:tip1 .
docker push localhost:5000/student_microservice:tip1
```

Apoi, modificați fișierul **questions_database.txt** pentru a include alte întrebări, de exemplu:

```
Care e sensul vietii?  
42  
Cat e ceasul?  
Cat ti-e nasul  
De ce a trecut gaina strada?  
Ca sa faca un ou
```

Construiți o altă imagine Docker cu noua „bază de date”. **Observați că nu modificați codul microserviciului:**

```
docker build -t localhost:5000/student_microservice:tip2 .
docker push localhost:5000/student_microservice:tip2
```

Modificați încă o dată fișierul cu alte întrebări și construiți și un al treilea tip de microserviciu **Student**:

```
docker build -t localhost:5000/student_microservice:tip3 .
docker push localhost:5000/student_microservice:tip3
```

MessageManagerMicroservice

```
FROM openjdk:8-jdk-alpine
ADD target/MessageManagerMicroservice-1.0-SNAPSHOT-jar-with-
dependencies.jar MessageManagerMicroservice.jar

ENTRYPOINT ["java", "-jar", "MessageManagerMicroservice.jar"]
```

De asemenea, nu uitați să înlocuiți numele artefactului în mod corespunzător. Instalați microserviciul într-un container Docker, în mod asemănător:

```
docker build -t localhost:5000/message_manager_microservice:v1 .
docker push localhost:5000/message_manager_microservice:v1
```

TeacherMicroservice

```
FROM openjdk:8-jdk-alpine
ADD target/TeacherMicroservice-1.0-SNAPSHOT-jar-with-dependencies.jar
TeacherMicroservice.jar

ENTRYPOINT ["java", "-jar", "TeacherMicroservice.jar"]
```

La fel și pentru acesta, instalați microserviciul astfel:

```
docker build -t localhost:5000/teacher_microservice:v1 .
docker push localhost:5000/teacher_microservice:v1
```

Confirmați că toate microserviciile sunt încapsulate cu succes în imagini Docker:

```
docker image ls
```

File	Edit	View	Terminal	Tabs	Help
REPOSITORY		TAG	IMAGE ID	CREATED	SIZE
localhost:5000/teacher_microservice		v1	13c4a8a4ad0e	14 seconds ago	106MB
localhost:5000/message_manager_microservice		v1	117353db08b0	4 minutes ago	106MB
localhost:5000/student_microservice		tip3	fd009c0ed11e	16 minutes ago	106MB
localhost:5000/student_microservice		tip2	99457f467bfd	17 minutes ago	106MB
localhost:5000/student_microservice		tip1	8697c91565fc	21 minutes ago	106MB
hello_microservice		v1	015104c8afcc	17 hours ago	106MB
localhost:5000/hello_microservice		v1	015104c8afcc	17 hours ago	106MB
localhost:5000/hello_microservice		<none>	531d2369e729	18 hours ago	106MB

Figura 40 - Microserviciile încapsulate în imagini Docker

Cu observația următoare: ați folosit comenzi `docker push` pentru a încărca aceste imagini și în registrul Docker privat creat anterior în laborator. **Acest lucru este necesar doar în cazul în care se dorește folosirea microserviciilor de pe alte mașini diferite de mașina locală.** După ce construiești imaginile Docker, acestea vor fi disponibile local, iar acum sunt disponibile și în registru. Confirmați acest lucru cu următoarea comandă:

```
curl -X GET http://localhost:5000/v2/_catalog
```

```
File Edit View Terminal Tabs Help
cosmin@debian-gl553v:~$ curl -X GET http://localhost:5000/v2/_catalog
{"repositories":["hello_microservice","message_manager_microservice","student_microservice","teacher_microservice"]}
cosmin@debian-gl553v:~$
```

Figura 41 - Verificarea disponibilității imaginilor Docker în registrul privat

Implementarea interfeței grafice

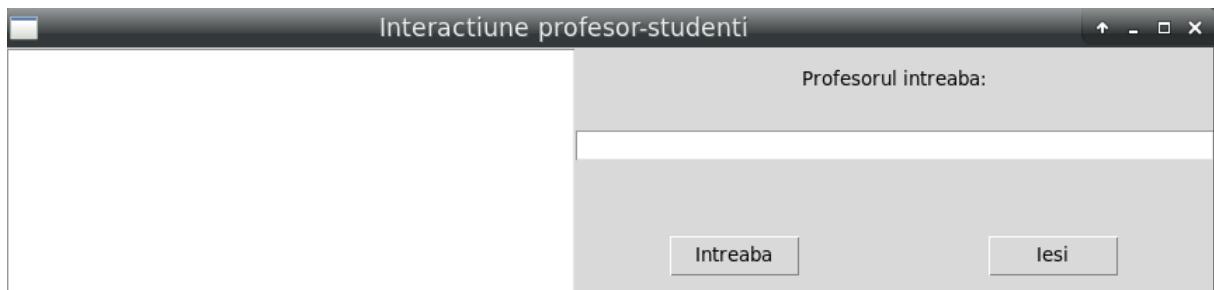


Figura 42 - Interfața grafică

Pentru interfața grafică și interacțiunea cu utilizatorul, puteți folosi, de exemplu, librăria **tkinter** pentru elementele grafice și librăria **socket** pentru comunicarea prin socket-uri TCP.

Folosind IntelliJ, creați un proiect Python cu un fișier sursă **TeacherStudentGUI.py**, având următorul conținut:

```
from tkinter import *
from tkinter import ttk
import threading
import socket

HOST = "localhost"
TEACHER_PORT = 1600

def resolve_question(question_text):
    # creare socket TCP
    sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

    # incercare de conectare catre microserviciul Teacher
    try:
        sock.connect((HOST, TEACHER_PORT))

        # transmitere intrebare - se deleaga intrebarea catre
        # microserviciu
        sock.send(bytes(question_text + "\n", "utf-8"))

        # primire raspuns -> microserviciul Teacher foloseste
        # coregrafia de microservicii pentru a trimite raspunsul inapoi
        response_text = str(sock.recv(1024), "utf-8")

    except ConnectionError:
        # in cazul unei erori de conexiune, se afiseaza un mesaj
        response_text = "Eroare de conectare la microserviciul
Teacher!"

    # se adauga raspunsul primit in caseta text din interfata grafica
    response_widget.insert(END, response_text)

def ask_question():
    # preluare text intrebare de pe interfata grafica
    question_text = question.get()
```

```

# pornire thread separat pentru tratarea intrebării respective
# astfel, nu se blocheaza interfata grafica!
threading.Thread(target=resolve_question,
args=(question_text,)).start()

if __name__ == '__main__':
    # elementul radacina al interfetei grafice
    root = Tk()

    # la redimensionarea ferestrei, cadrele se extind pentru a prelua
    spatiul ramas
    root.columnconfigure(0, weight=1)
    root.rowconfigure(0, weight=1)

    # cadrul care incapsuleaza intregul continut
    content = ttk.Frame(root)

    # caseta text care afiseaza raspunsurile la intrebări
    response_widget = Text(content, height=10, width=50)

    # eticheta text din partea dreapta
    question_label = ttk.Label(content, text="Profesorul intreaba:")

    # caseta de introducere text cu care se preia intrebarea de la
    utilizator
    question = ttk.Entry(content, width=50)

    # butoanele din dreapta-jos
    ask = ttk.Button(content, text="Intreaba", command=ask_question)
    # la apasare, se apeleaza functia ask_question
    exitbtn = ttk.Button(content, text="Iesi", command=root.destroy)
    # la apasare, seiese din aplicatie

    # plasarea elementelor in layout-ul de tip grid
    content.grid(column=0, row=0)
    response_widget.grid(column=0, row=0, columnspan=3, rowspan=4)
    question_label.grid(column=3, row=0, columnspan=2)
    question.grid(column=3, row=1, columnspan=2)
    ask.grid(column=3, row=3)
    exitbtn.grid(column=4, row=3)

    # bucla principală a interfetei grafice care asteapta evenimente
    de la utilizator
    root.mainloop()

```

Interfața conține o metodă de tratare a apăsării butonului „Întreabă”, în care programul se conectează într-un *thread* separat (**pentru a nu bloca interfața grafică!**) la microserviciul **Teacher**. După conectarea cu succes, se trimit textul întrebării preluat din caseta text de pe interfață și se așteaptă răspunsul de la microserviciul respectiv. Răspunsurile se adaugă progresiv la sfârșitul casetei text multilinie din partea stângă.

Urmăriți comentariile din cod pentru explicații suplimentare legate de implementare.

Testarea microserviciilor

Porniți containerele Docker ce încapsulează microserviciile pe care le-ați creat, cu mențiunea că **MessageManager** trebuie pornit primul, întrucât celelalte entități se conectează la

acesta în momentul în care își începe execuția.

Fiecare microserviciu va fi identificat de un nume, deoarece acest nume este folosit de Docker pentru adresarea într-o rețea virtuală. Conexiunile la serverele socket TCP pe care microserviciile le folosesc se bazează pe aceste nume.

Așadar, mai întâi trebuie să creați o rețea virtuală Docker, folosind următoarea comandă:

```
docker network create ms-net
```

```
cosmin@debian-gl553v:~$ docker network create ms-net
83de2c7f4a01f9a1c803b2785fa9be953614cd9500af3ea094ddd126ab9c27c3
cosmin@debian-gl553v:~$
```

Figura 43 - Creare unei rețele virtuale Docker

În continuare, fiecare container pe care îl veți porni trebuie conectat la această rețea. Astfel, se simulează acel „localhost” disponibil pe mașina gazdă.

Pornirea microserviciului MessageManager

```
docker run -d -p 1500:1500 --name message_manager --network=ms-net
localhost:5000/message_manager_microservice:v1
```

Se expune portul 1500 pe același port gazdă, pentru a nu crea confuzie.

```
- cosmin@debian-gl553v: ~
File Edit View Terminal Tabs Help
cosmin@debian-gl553v:~$ docker run -d -p 1500:1500 --name message_manager --network=ms-net
localhost:5000/message_manager_microservice:v1
dc0b5a5a900e9d3d03e7d7fd2b7b13d3cc7eabc2432ad7e95ab3bf6a1b2501f2
cosmin@debian-gl553v:~$ docker logs dc0b5a5a900e9d3d03e7d7fd2b7b13d3cc7eabc2432ad7e95ab3bf
6a1b2501f2
MessageManagerMicroservice se executa pe portul: 1500
Se asteapta conexiuni si mesaje...
cosmin@debian-gl553v:~$
```

Figura 44 - Pornirea și verificarea microserviciului MessageManager

Pornirea microserviciului Teacher

```
docker run -d -p 1600:1600 -e MESSAGE_MANAGER_HOST='message_manager' -
--name teacher_microservice --network=ms-net
localhost:5000/teacher_microservice:v1
```

Se trimită o variabilă de mediu (parametrul **-e**) acestui container, denumită **MESSAGE_MANAGER_HOST**, care este folosită în codul microserviciului **Teacher** pentru a identifica serverul socket care se execută în microserviciul **MessageManager**. **message_manager** este numele containerului pe care l-ați pornit mai sus, și are rol de hostname: identifică acel container în rețeaua **my-net**.

La fel și aici, se expune portul 1600 din container mapat pe același port 1600 pe mașina gazdă.

```
- cosmin@debian-gl553v: ~
File Edit View Terminal Tabs Help
cosmin@debian-gl553v:~$ docker run -d -p 1600:1600 -e MESSAGE_MANAGER_HOST='message_manager'
--name teacher_microservice --network=ms-net localhost:5000/teacher_microservice:v1
35bb0fc7c59e1227df9574a2db0349b2f4125a473d2574f758d9af84d66839d
cosmin@debian-gl553v:~$ docker logs 35bb0fc7c59e1227df9574a2db0349b2f4125a473d2574f758d9af
d84d66839d
Incerca sa ma conectez la: message_manager:1500
M-am conectat la MessageManager!
TeacherMicroservice se executa pe portul: 1600
Se asteapta cereri (intrebari)...
cosmin@debian-gl553v:~$
```

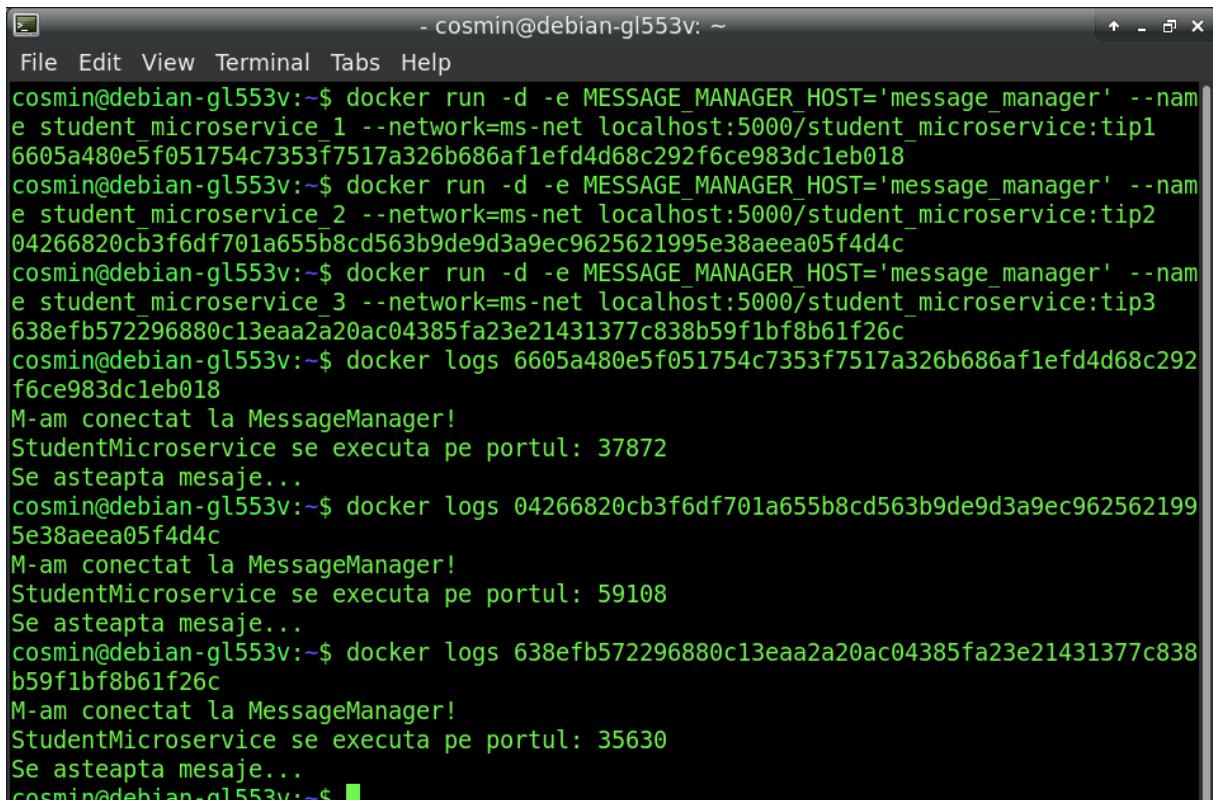
Figura 45 - Pornirea și verificarea microserviciului Teacher

Pornirea microserviciilor Student, de mai multe tipuri

```
docker run -d -e MESSAGE_MANAGER_HOST='message_manager' --name student_microservice_1 --network=ms-net
localhost:5000/student_microservice:tip1

docker run -d -e MESSAGE_MANAGER_HOST='message_manager' --name student_microservice_2 --network=ms-net
localhost:5000/student_microservice:tip2

docker run -d -e MESSAGE_MANAGER_HOST='message_manager' --name student_microservice_3 --network=ms-net
localhost:5000/student_microservice:tip3
```



```
- cosmin@debian-gl553v: ~
File Edit View Terminal Tabs Help
cosmin@debian-gl553v:~$ docker run -d -e MESSAGE_MANAGER_HOST='message_manager' --name student_microservice_1 --network=ms-net localhost:5000/student_microservice:tip1
6605a480e5f051754c7353f7517a326b686af1efd4d68c292f6ce983dc1eb018
cosmin@debian-gl553v:~$ docker run -d -e MESSAGE_MANAGER_HOST='message_manager' --name student_microservice_2 --network=ms-net localhost:5000/student_microservice:tip2
04266820cb3f6df701a655b8cd563b9de9d3a9ec9625621995e38aeea05f4d4c
cosmin@debian-gl553v:~$ docker run -d -e MESSAGE_MANAGER_HOST='message_manager' --name student_microservice_3 --network=ms-net localhost:5000/student_microservice:tip3
638efb572296880c13eaa2a20ac04385fa23e21431377c838b59f1bf8b61f26c
cosmin@debian-gl553v:~$ docker logs 6605a480e5f051754c7353f7517a326b686af1efd4d68c292f6ce983dc1eb018
M-am conectat la MessageManager!
StudentMicroservice se executa pe portul: 37872
Se asteapta mesaje...
cosmin@debian-gl553v:~$ docker logs 04266820cb3f6df701a655b8cd563b9de9d3a9ec9625621995e38aeea05f4d4c
M-am conectat la MessageManager!
StudentMicroservice se executa pe portul: 59108
Se asteapta mesaje...
cosmin@debian-gl553v:~$ docker logs 638efb572296880c13eaa2a20ac04385fa23e21431377c838b59f1bf8b61f26c
M-am conectat la MessageManager!
StudentMicroservice se executa pe portul: 35630
Se asteapta mesaje...
cosmin@debian-gl553v:~$
```

Figura 46 - Pornirea și verificarea microserviciilor Student

Verificați și pornirea cu succes a microserviciilor Student prin analiza log-urilor, ca în figura de mai sus.

```
docker logs <IDENTIFICATOR_CONTAINER>
```

În acest moment, ar trebui să aveți 5 containere Docker pornite, fiecare încapsulând câte un microserviciu. Verificați acest lucru folosind comanda:

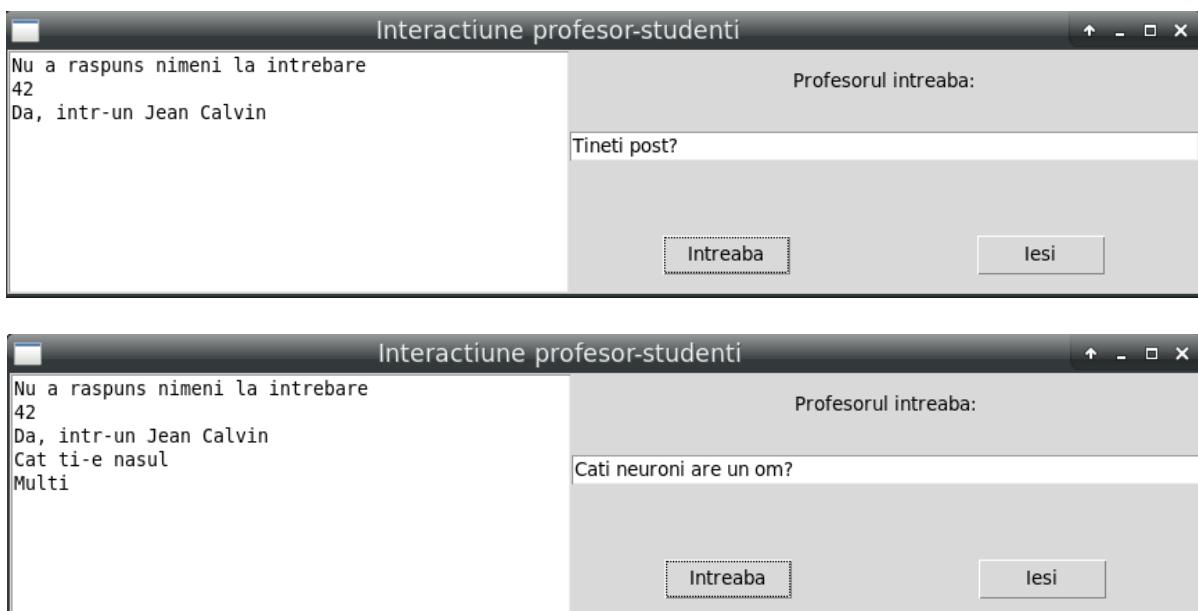
```
docker ps
```

Sisteme Distribuite - Laborator 8

```
File Edit View Terminal Tabs Help
cosmin@debian-gl553v:~$ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED            STATUS              PORTS               NAMES
638efb572296        localhost:5000/student_microservice:tip3   "java -jar StudentMi..."   About a minute ago   Up About a minute   student_micro
service_3           localhost:5000/student_microservice:tip2   "java -jar StudentMi..."   About a minute ago   Up About a minute   student_micro
service_2           localhost:5000/student_microservice:tip1   "java -jar StudentMi..."   About a minute ago   Up About a minute   student_micro
service_1           localhost:5000/teacher_microservice:v1     "java -jar TeacherMi..."  4 minutes ago       Up 4 minutes      0.0.0.0:1600->1600/tcp  teacher_micro
dc0b5a5a900e        localhost:5000/message_manager_microservice:v1 "java -jar MessageMa..."  9 minutes ago       Up 9 minutes      0.0.0.0:1500->1500/tcp  message_manag
er
354d163a4a55        registry:2                           "/entrypoint.sh /etc..."  21 hours ago       Up 6 hours        0.0.0.0:5000->5000/tcp  registru_dock
er
cosmin@debian-gl553v:~$
```

Figura 47 - Execuția tuturor celor 5 microservicii încapsulate în containere Docker

Deschideți aplicația client scrisă în Python și încercați să introduceți diverse întrebări (care există sau nu în bazele de date pe care le-ați încărcat) în caseta text din partea dreaptă.



Aplicația client se conectează la microserviciul **Teacher**, pe portul expus de acesta (1600) și cere răspunsurile la întrebări prin contactarea celorlalte microservicii din rețea internă Docker. Clientul este total decuplat de logica de business: nu îl interesează decât că poate trimite mesaje cu întrebări și trebuie să primească răspuns.

Puteți verifica și mesajele care circulă prin microservicii, inspectându-le log-urile din containere:

```

- cosmin@debian-gl553v: ~
File Edit View Terminal Tabs Help
cosmin@debian-gl553v:~$ docker logs dc0b5a5a900e
MessageManagerMicroservice se executa pe portul: 1500
Se asteapta conexiuni si mesaje...
Subscriber conectat: 172.19.0.3:34252
Subscriber conectat: 172.19.0.4:37872
Subscriber conectat: 172.19.0.5:59108
Subscriber conectat: 172.19.0.6:35630
Primit mesaj: intrebare 34252 Ce?
Primit mesaj: intrebare 34252 Care e sensul vietii?
Primit mesaj: raspuns 34252 42
Primit mesaj: intrebare 34252 Tineti post?
Primit mesaj: raspuns 34252 Da, intr-un Jean Calvin
Primit mesaj: intrebare 34252 Cat e ceasul?
Primit mesaj: raspuns 34252 Cat ti-e nasul
Primit mesaj: intrebare 34252 Cati neuroni are un om?
Primit mesaj: raspuns 34252 Multi
cosmin@debian-gl553v:~$ docker logs 638efb572296
M-am conectat la MessageManager!
StudentMicroservice se executa pe portul: 35630
Se asteapta mesaje...
Am primit o intrebare de la 34252: "Ce?"
Am primit o intrebare de la 34252: "Care e sensul vietii?"
Am primit o intrebare de la 34252: "Tineti post?"
Trimite raspunsul: "intrebare 34252 Tineti post?"
Am primit o intrebare de la 34252: "Cat e ceasul?"

```

Aplicații și teme

Temă de laborator

1. Modificați aplicația exemplu, astfel încât și studentul să poată să pună întrebări atât la profesor, cât și la alți studenți, sau la un anumit student (comunicare de tip **one to one** sau **one to many**). Răspunsul poate să fie public sau privat (**one to one** = mesaj privat, **one to many** = mesaj public).

Teme pe acasă

1. Reimplementați aplicația de laborator utilizând corutine Kotlin.
2. Implementați un mecanism de **heartbeat** sub formă de microserviciu separat care trimit mesaje *dummy*, în scopul verificării funcționării corecte ale celorlalte microservicii.
3. Implementați un mecanism primar care să imite un **Docker registry**, bazat pe metoda *publish - subscribe*. Mecanismul trebuie să conțină operațiile de înscriere și dezinscriere și să fie încapsulat într-un microserviciu separat, instalat separat (engl. *deployed*).

La instalarea microserviciilor de la problemele 2 și 3 (eng. *deployment*), **heartbeat**-ul să fie plasat în aceeași mașină cu **registry**-ul, deoarece au funcționalități comune și trebuie să fie instalate în același loc.

Bibliografie

- [1] Docker Registry - <https://docs.docker.com/registry/>
- [2] Pornirea automată a containerelor Docker - <https://docs.docker.com/config/containers/start-containers-automatically/>
- [3] Informații despre starea Docker Engine (comanda **docker info**) - <https://docs.docker.com/engine/reference/commandline/info/>
- [4] Preluarea imaginilor sau a depozitelor din registre Docker (comanda **docker pull**) - <https://docs.docker.com/engine/reference/commandline/pull/>
- [5] Lucrul cu imagini Docker (comanda **docker image**) - <https://docs.docker.com/engine/reference/commandline/image/>

- [6] Inspectarea unui container Docker (comanda **docker inspect**) -
<https://docs.docker.com/engine/reference/commandline/inspect/>
- [7] Listarea stării containerelor Docker (comanda **docker ps**) -
<https://docs.docker.com/engine/reference/commandline/ps/>
- [8] Pornirea / oprirea containerelor Docker (comenzile **docker start** și **docker stop**) -
<https://docs.docker.com/engine/reference/commandline/start/>
<https://docs.docker.com/engine/reference/commandline/stop/>
- [9] Crearea de fișiere Dockerfile - <https://docs.docker.com/engine/reference/builder/>
- [10] Construirea de imagini Docker folosind Dockerfile (comanda **docker build**) -
<https://docs.docker.com/engine/reference/commandline/build/>
- [11] Încărcarea unei imagini Docker într-un registru (comanda **docker push**) -
<https://docs.docker.com/engine/reference/commandline/push/>
- [12] Vizualizarea log-urilor dintr-un container Docker (comanda **docker logs**) -
<https://docs.docker.com/engine/reference/commandline/logs/>
- [13] Alpine Linux - <https://alpinelinux.org/>
- [14] Docker Registry API - <https://docs.docker.com/registry/spec/api/>
- [15] Socket Server / Client în Python - <https://docs.python.org/3/library/socketserver.html>

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**.

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 comenziile „**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/>.

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

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

The image shows three separate terminal windows side-by-side. The leftmost window displays logs for a 'DataFlow' application, including initialization of 'ReleaseStateUpdateService', 'ExecutorService', and 'Tomcat' on port 7577. The middle window shows logs for 'JpaWebConfiguration\$JpaWebConfiguration', which includes a warning about 'spring.jpa.open-in-view' being enabled by default. The rightmost window shows logs for 'dataflow 2.4.2.RELEASE', including startup of 'TaskExecutor', 'ProtocolHandler', and 'TomcatWebServer' on port 9393.

```

cosmin@debian-gl553v:~/mnt/hdd/PREDARE/SD/laboratoare/laborator 9/DataFlow$ ./dataflow 2.4.2.RELEASE
[...]
2020-04-09 18:19:21.191  WARN 25889 --- [           main] JpaBaseConfiguration$JpaWebConfiguration : spring.jpa.open-in-view is enabled by default. Therefore, database queries may be performed during view rendering. Explicitly configure spring.jpa.open-in-view to disable this warning
2020-04-09 18:19:21.409  INFO 25889 --- [           main] o.s.concurrent.ThreadPoolTaskExecutor : Initializing ExecutorService 'applicationTaskExecutor'
2020-04-09 18:19:22.474  INFO 25889 --- [           main] .s.c.DataFlowControllerAutoConfiguration : Skipper URI [http://localhost:7577/api]
2020-04-09 18:19:22.697  INFO 25889 --- [           main] o.a.coyote.http11.Http11NioProtocol : Starting ProtocolHandler ["http-nio-9393"]
2020-04-09 18:19:22.745  INFO 25889 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port(s): 9393 (http) with context path ''
2020-04-09 18:19:22.751  INFO 25889 --- [           main] o.s.c.d.s.s.DataFlowServerApplication : Started DataFlowServerApplication in 11.887 seconds (JVM running for 12.377)
cosmin@debian-gl553v:~/mnt/hdd/PREDARE/SD/laboratoare/laborator 9/DataFlow$ java -jar dataflow-shell-2.4.2.RELEASE.jar
[...]
2020-04-09 18:19:26.572  INFO 25914 --- [           main] o.s.s.concurrent.ThreadPoolTaskExecutor : Setting up ReleaseStateUpdateService
2020-04-09 18:19:26.705  WARN 25914 --- [           main] JpaBaseConfiguration$JpaWebConfiguration : spring.jpa.open-in-view is enabled by default. Therefore, database queries may be performed during view rendering. Explicitly configure spring.jpa.open-in-view to disable this warning
2020-04-09 18:19:26.934  INFO 25914 --- [           main] o.s.concurrent.ThreadPoolTaskExecutor : Initializing ExecutorService 'applicationTaskExecutor'
2020-04-09 18:19:28.133  INFO 25914 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer : 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.SkipperServerApplication : 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.RepositoryInitializationService : Initializing repository database for name local
2020-04-09 18:19:28.499  INFO 25914 --- [           main] .s.c.d.s.s.DeployerInitializationService : Added 'Local' platform account 'default' into Deployer repository.

```

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>

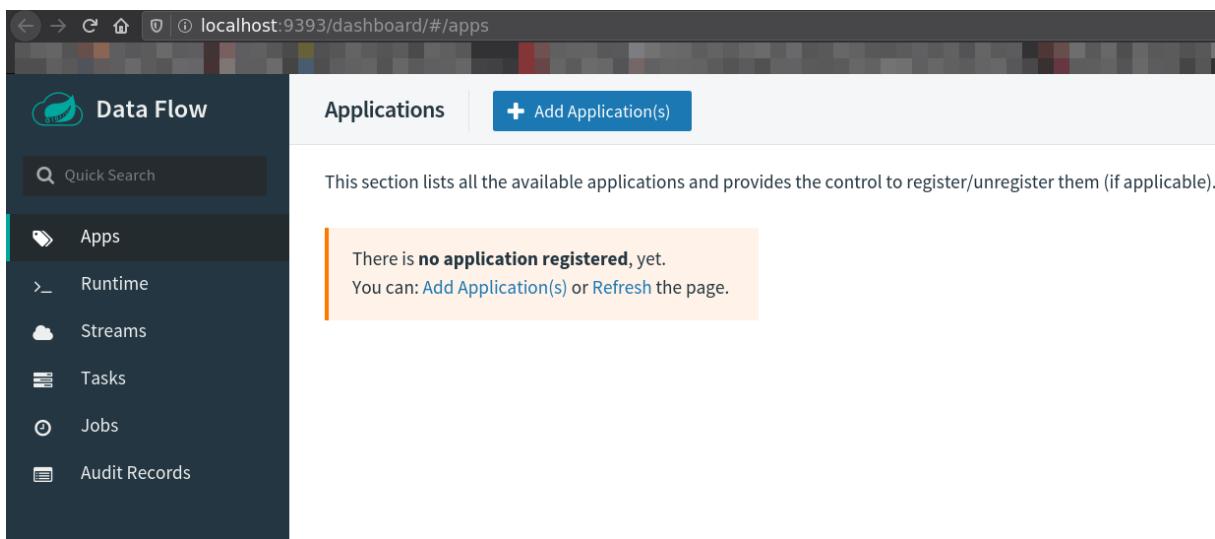


Figura 1 - Panoul de control Spring Data Flow

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.

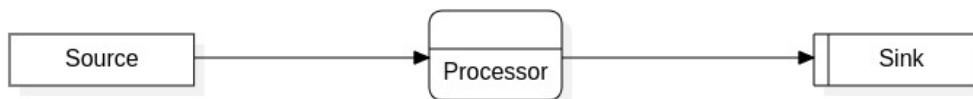


Figura 2 - Diagrama unui flux de date

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.

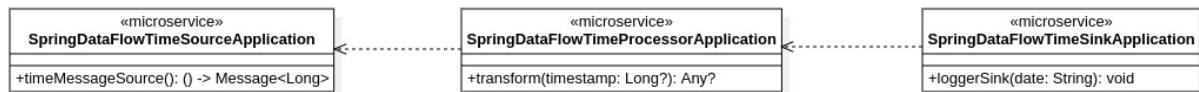


Figura 3 - Diagrama de clase

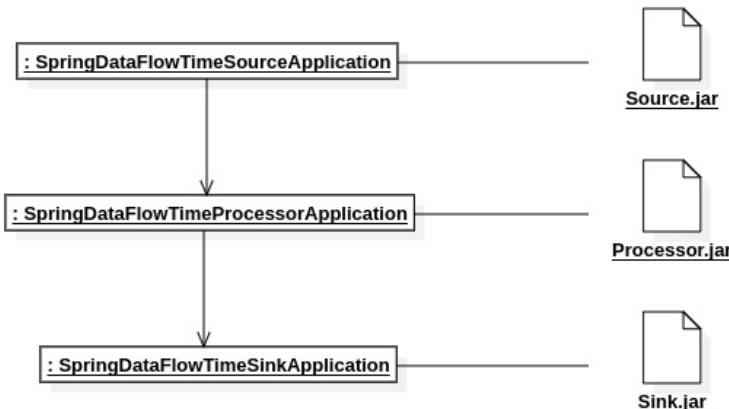


Figura 4 - Diagrama de obiecte

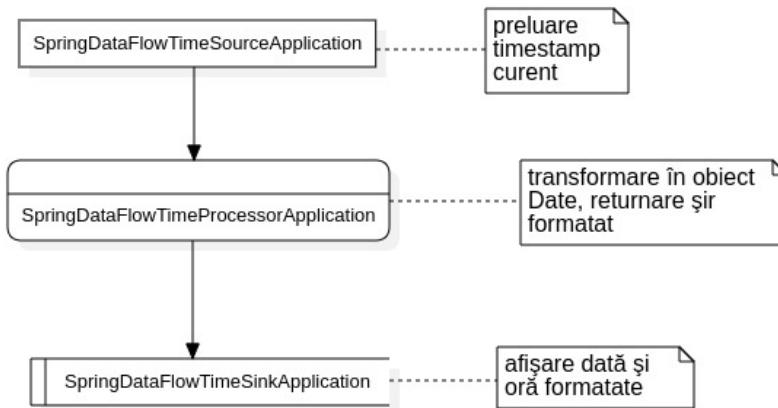


Figura 5 - Diagrama fluxului de date

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().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.SpringApplication
import org.springframework.cloud.stream.annotation.EnableBinding
import org.springframework.cloud.stream.messaging.Processor
import org.springframework.integration.annotation.Transformer
import java.text.SimpleDateFormat
import java.text.DateFormat

@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>) {
    SpringApplication.run(SpringDataFlowTimeSinkApplication::class.java, *args)
}
```

Conform adnotării **@EnableBinding(Sink::class)**, aplice 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 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_SURSA
```

Înlocuiți **CALE_CĂTRE_ARTEFACTUL_JAR_AL_APlicațIEI_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 ce conține acel artefact, apoi folosiți comanda **realpath**:

```
Terminal: Local × Local (2) × +
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 comenziilor 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

[REDACTED LOGS]

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_APPLICATIE> --type <TIPO_APPLICATIE>
```

Crearea fluxului de date din entitățile înregisterate 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 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 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:

The screenshot shows the Data Flow dashboard at the URL <http://localhost:9393/dashboard/#/apps>. On the left, there is a sidebar with navigation links: Data Flow, Apps (which is selected), Runtime, Streams, Tasks, Jobs, and Audit Records. The main content area is titled "Applications" and contains a table with three rows. The columns are "Name", "Type", "Url", and "Versions". The rows are:

Name	Type	Url	Versions
logging-sink	SINK	file:///mnt/hdd/LUCRU/Spring%20Cloud/DataFlow%20Sink/target/DataFlowSink-1.0-SNAPSHOT.jar	1.0-SNAPSHOT
time-processor	PROCESSOR	file:///mnt/hdd/LUCRU/Spring%20Cloud/DataFlow%20Processor/target/DataFlowProcessor-1.0-SNAPSHOT.jar	1.0-SNAPSHOT
time-source	SOURCE	file:///mnt/hdd/LUCRU/Spring%20Cloud/DataFlow%20Source/target/DataFlowSource-1.0-SNAPSHOT.jar	1.0-SNAPSHOT

At the bottom of the table, it says "items per page: 30 | 1-3 applications of 3 applications".

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).

The screenshot shows the Data Flow interface with the sidebar menu open, highlighting the 'Runtime' option. The main area is titled 'Runtime applications' and contains the following information:

- Stream time-to-log**
- time-to-log.time-processor-v1**: DEPLOYED, 1 instance
- time-to-log.time-source-v1**: DEPLOYED, 1 instance
- time-to-log.logging-sink-v1**: DEPLOYED, 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 sidebar menu open, highlighting the 'Streams' option. The main area is titled 'Streams' and contains the following information:

- time-to-log**: DEPLOYED

Below the table, it says 'items per page: 30 | 1-1 stream definition of 1 stream definition'.

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).

A detailed view of the 'time-to-log.logging-sink-v1' application card, showing its status and deployment details:

- time-to-log.logging-sink-v1**
- DEPLOYED**
- 1 instance

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

Instances for app **time-to-log.logging-sink-v1**

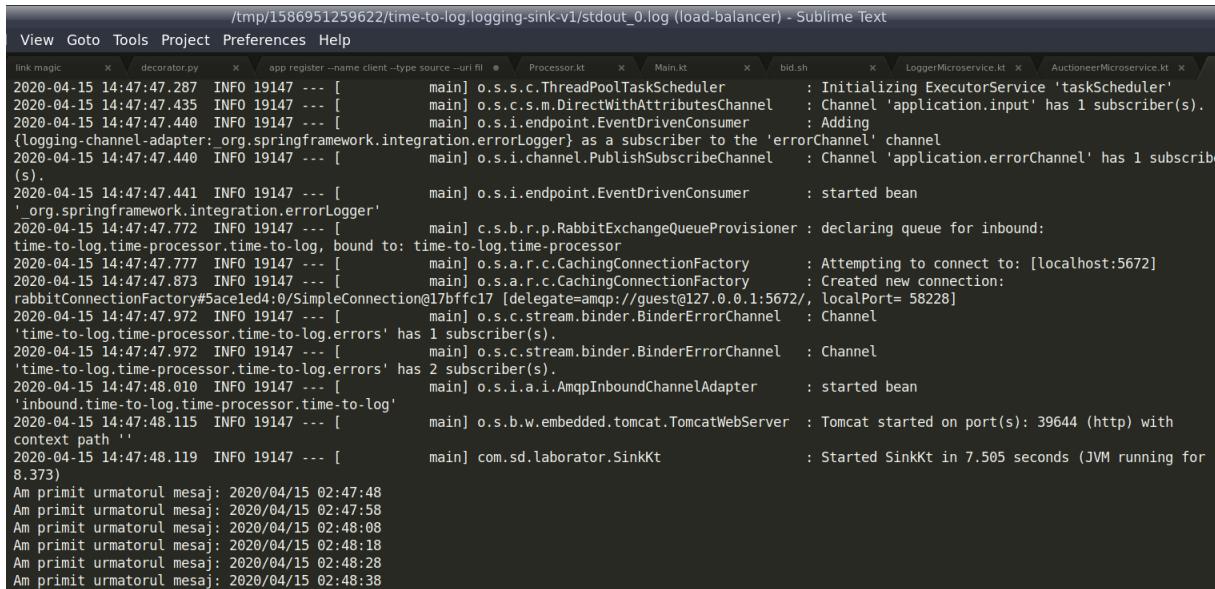
Instance **time-to-log.logging-sink-v1-0** DEPLOYED

guid	45681
pid	20959
port	45681
skipper.application.name	logging-sink
skipper.release.name	time-to-log
skipper.release.version	1
stderr	/tmp/1586620062499/time-to-log.logging-sink-v1/stderr_0.log
stdout	/tmp/1586620062499/time-to-log.logging-sink-v1/stdout_0.log
url	http://127.0.1.1:45681
working.dir	/tmp/1586620062499/time-to-log.logging-sink-v1

Cancel

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/1586620062499/time-to-log.logging-sink-v1/stdout_0.log (load-balancer) - Sublime Text
View Goto Tools Project Preferences Help
File magic x decorator.py x app register --name client --type source --uri file Processor.kt x Main.kt x bid.sh x LoggerMicroservice.kt x AuctioneerMicroservice.kt x
2020-04-15 14:47:47.287 INFO 19147 --- [main] o.s.s.c.ThreadPoolTaskScheduler : Initializing ExecutorService 'taskScheduler'
2020-04-15 14:47:47.435 INFO 19147 --- [main] o.s.c.s.m.DirectWithAttributesChannel : Channel 'application.input' has 1 subscriber(s).
2020-04-15 14:47:47.440 INFO 19147 --- [main] o.s.i.endpoint.EventDrivenConsumer : Adding {logging-channel-adapter: org.springframework.integration.errorLogger} as a subscriber to the 'errorChannel' channel
2020-04-15 14:47:47.440 INFO 19147 --- [main] o.s.i.channel.PublishSubscribeChannel : Channel 'application.errorChannel' has 1 subscriber(s).
2020-04-15 14:47:47.441 INFO 19147 --- [main] o.s.i.endpoint.EventDrivenConsumer : started bean 'org.springframework.integration.errorLogger'
2020-04-15 14:47:47.772 INFO 19147 --- [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
2020-04-15 14:47:47.777 INFO 19147 --- [main] o.s.a.r.c.CachingConnectionFactory : Attempting to connect to: [localhost:5672]
2020-04-15 14:47:47.873 INFO 19147 --- [main] o.s.a.r.c.CachingConnectionFactory : Created new connection: rabbitConnectionFactory#5ace1ed4:0/SimpleConnection@17bfffc17 [delegate=amqp://guest@127.0.0.1:5672/, localPort= 58228]
2020-04-15 14:47:47.972 INFO 19147 --- [main] o.s.c.stream.binder.BinderErrorChannel : Channel 'time-to-log.time-processor.time-to-log.errors' has 1 subscriber(s).
2020-04-15 14:47:47.972 INFO 19147 --- [main] o.s.c.stream.binder.BinderErrorChannel : Channel 'time-to-log.time-processor.time-to-log.errors' has 2 subscriber(s).
2020-04-15 14:47:48.010 INFO 19147 --- [main] o.s.i.a.i.AmqpInboundChannelAdapter : started bean 'inbound.time-to-log.time-processor.time-to-log'
2020-04-15 14:47:48.115 INFO 19147 --- [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port(s): 39644 (http) with context path ''
2020-04-15 14:47:48.119 INFO 19147 --- [main] com.sd.laborator.SinkKt : Started SinkKt in 7.505 seconds (JVM running for 8.373)
Am primit urmatorul mesaj: 2020/04/15 02:47:48
Am primit urmatorul mesaj: 2020/04/15 02:47:58
Am primit urmatorul mesaj: 2020/04/15 02:48:08
Am primit urmatorul mesaj: 2020/04/15 02:48:18
Am primit urmatorul mesaj: 2020/04/15 02:48:28
Am primit urmatorul mesaj: 2020/04/15 02:48:38
```

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 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

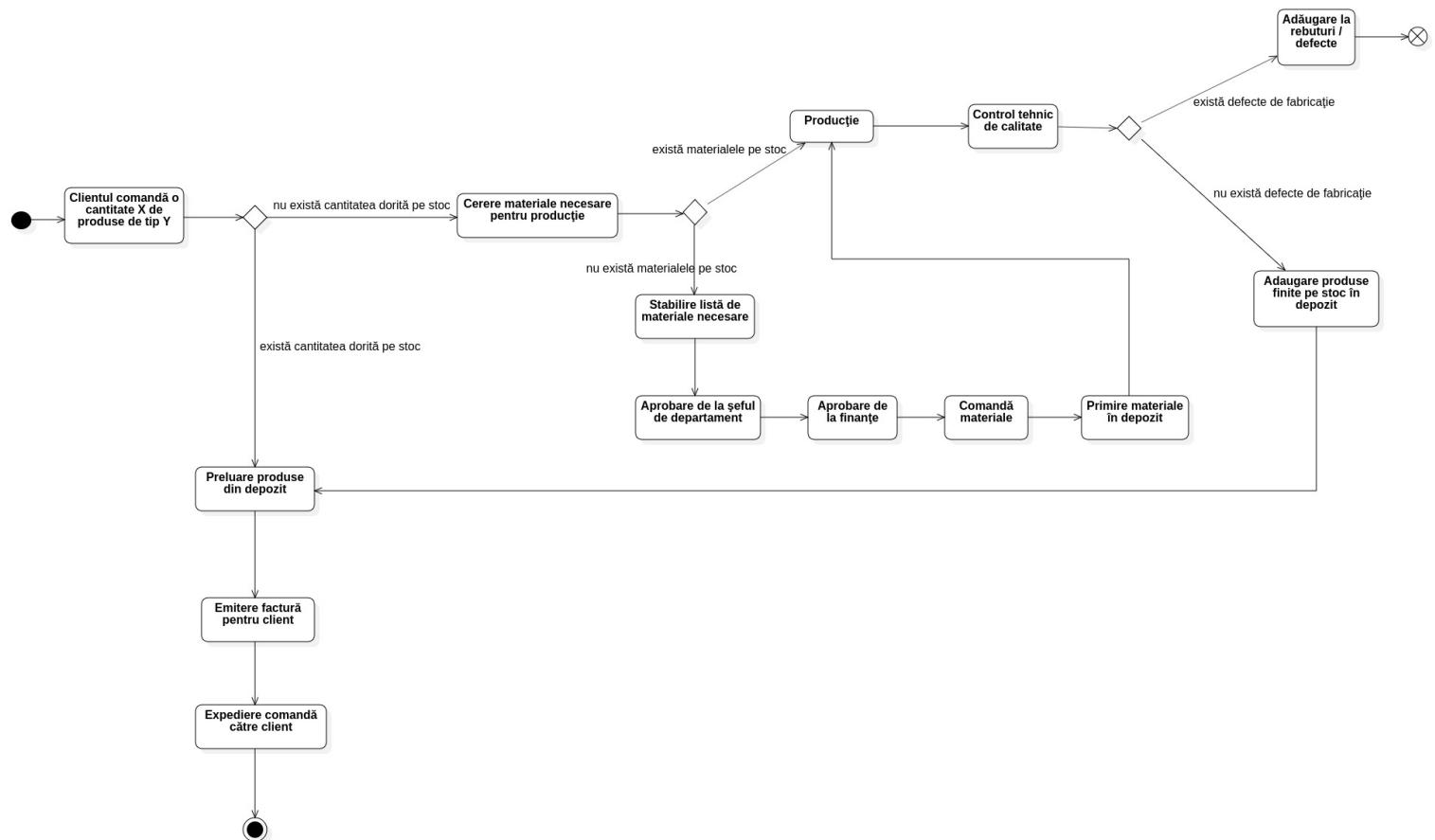


Figura 13 - Fluxul de afaceri

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.

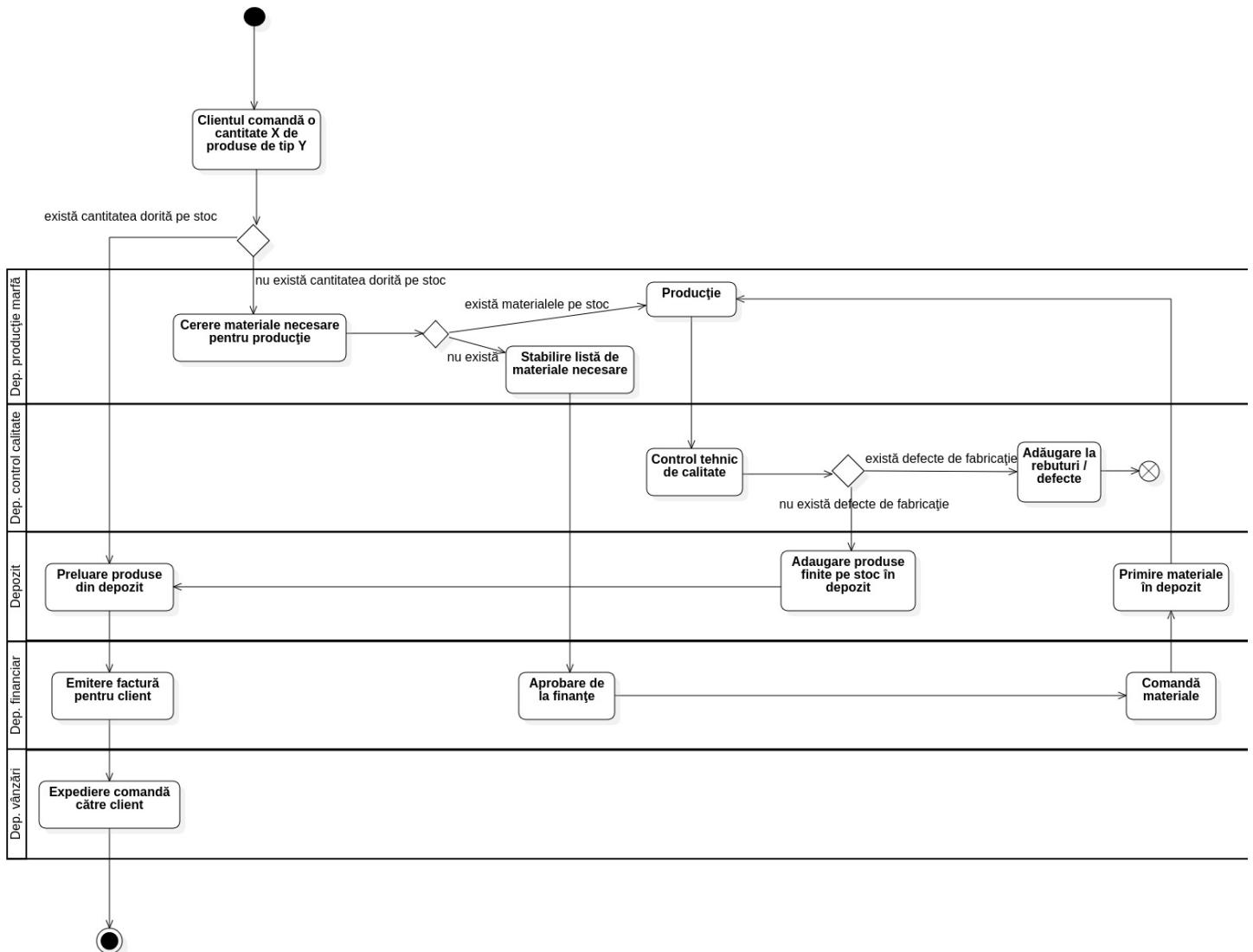


Figura 14 - Împărțirea pe departamente

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.

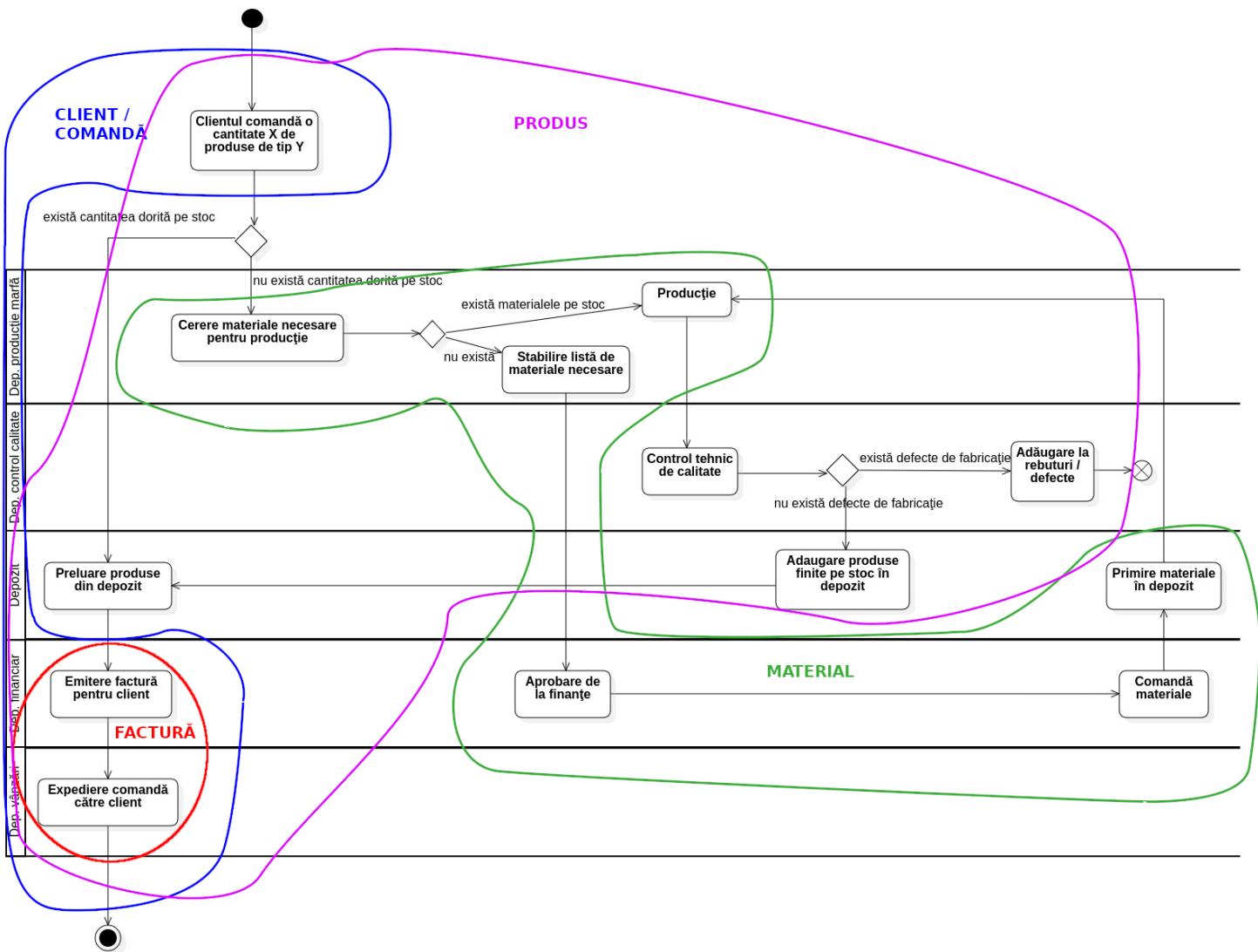


Figura 15 - Contextele mărginite

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 atrbute 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*).

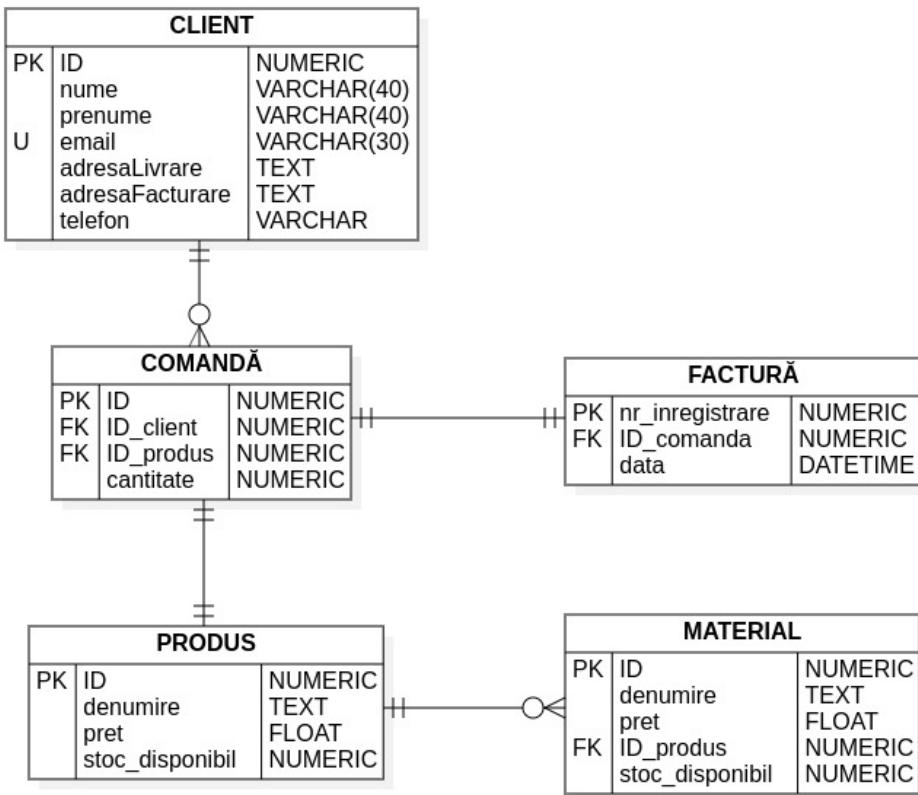


Figura 16 - Diagrama ERD

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 aceluui 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 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 să 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*).

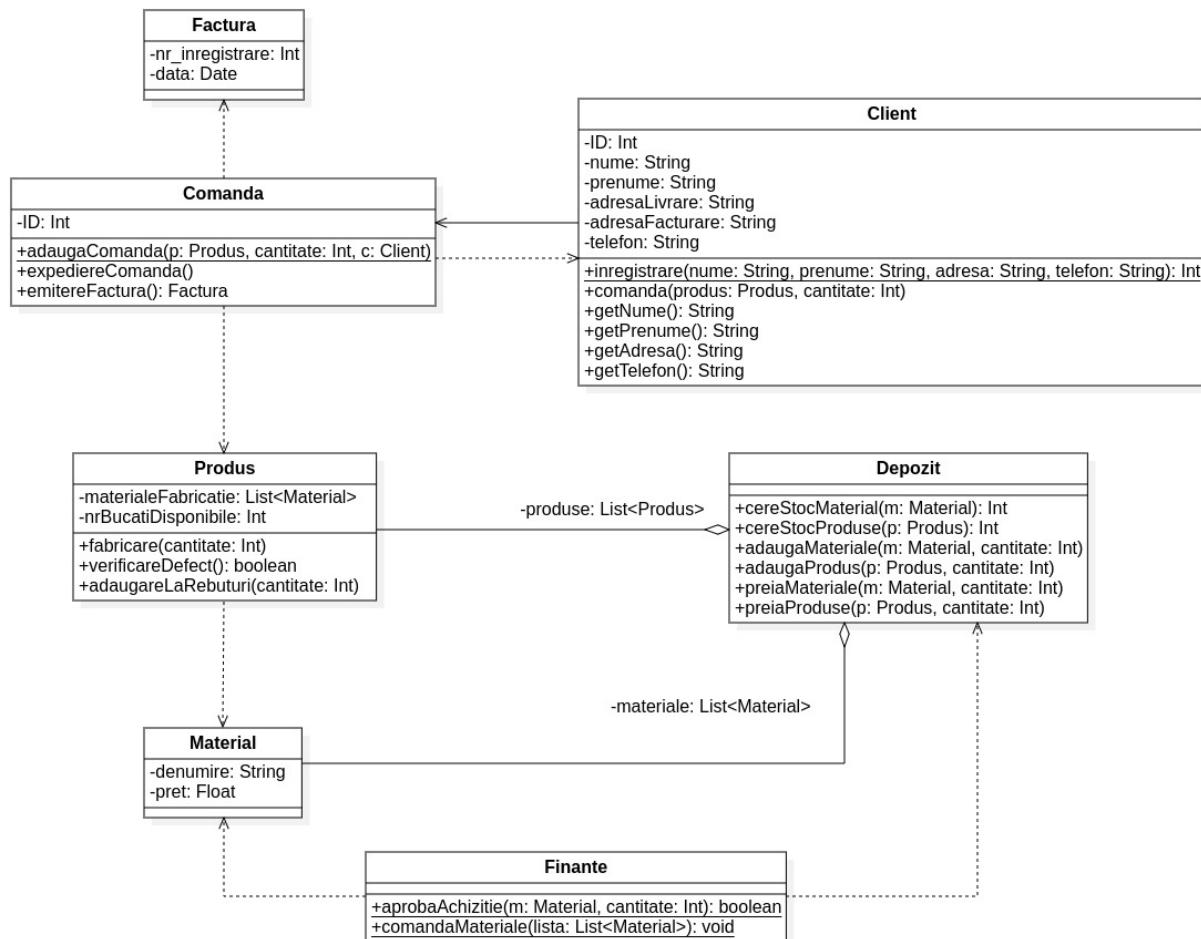


Figura 17 - Diagrama de clase

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:

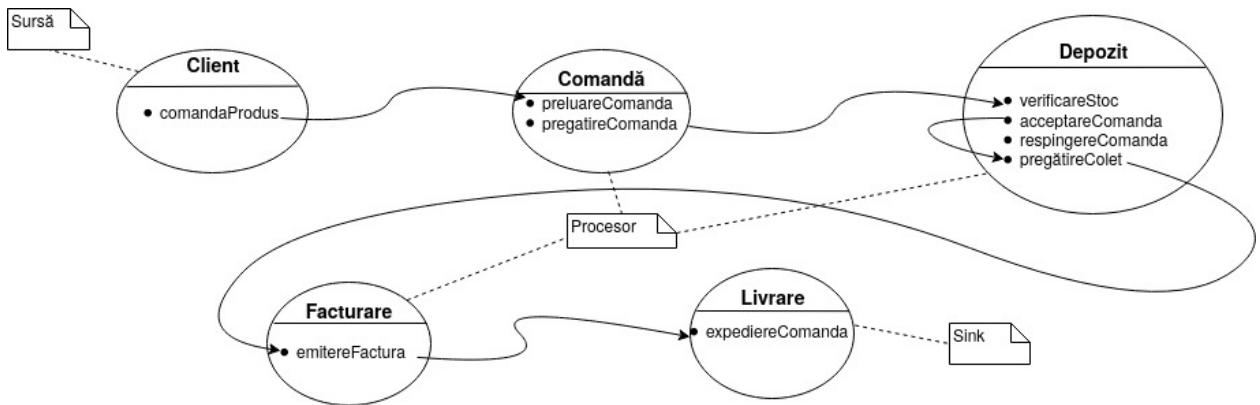


Figura 18 - Diagrama de microservicii

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 Vlăsiei 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 pregeteste 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
    //specifica
    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 = 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
--

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 „|“), stabilite 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 comenzi care a mai rămas de executat către următorul procesor, și.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, aşadar, 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.pmmi.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.tcp.metadata, sink.pgc.copy.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.pmmi, 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.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.pgc.copy, processor.python-http.metadata, sink.jdbc.metadata, sink.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, processor.groovy-transform.metadata, processor.aggregator, sink.sftp.metadata, processor.tensorflow.metadata, sink.throughput.metadata, sink.tcp, source.mail, 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]
dataflow:>
```

2. Să se modifice exemplul 2 din laborator astfel:

- realizarea unei interfețe GUI
- utilizarea unei baze de date relaționale (de ex. SQLite)
- 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.
- 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**).

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

Sisteme Distribuite - Laborator 10

Aplicații flux și monitorizare utilizând Apache Kafka

Apache Kafka - descriere generală

Kafka este un sistem de mesagerie de tip *publish / subscribe*, descris ca un „*commit log distribuit*”, sau o „*platformă de streaming distribuită*”. Față de alte sisteme de acest tip, Kafka a fost proiectat ca datele să fie stocate pe termen lung, iar acestea pot fi citite în mod determinist și în ordinea în care au fost scrise.

Unitatea de bază cu care funcționează sistemul Kafka este **mesajul** (eng. *message*). Ca și concept, este similar cu o înregistrare într-un tabel dintr-o bază de date. Mesajele pot avea metadate folosite pentru identificare, numite **chei** (eng. *keys*). Cheile sunt utilizate pentru scrierea mesajelor în **partiții**, fapt care asigură un control mai granular.

Mesajele în Kafka sunt organizate pe **subiecte** (eng. *topics*). Acestea pot fi asemănăte cu un folder în sistemul de fișiere sau un tabel într-o bază de date. La rândul lor, subiectele sunt împărțite în **partiții** (eng. *partitions*). O partiție este echivalentă cu un *commit log*. Partițiile reprezintă o modalitate prin care Kafka oferă redundanță și scalabilitate.

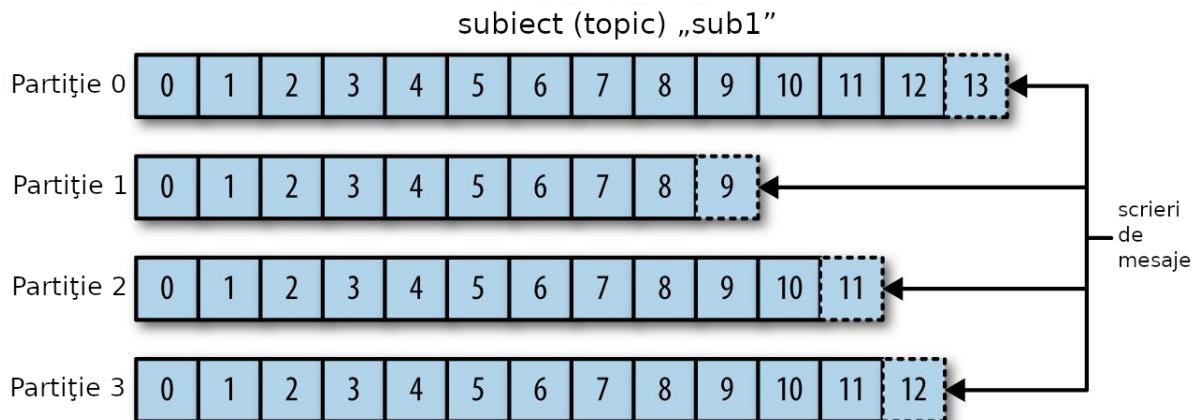


Figura 1 - Subiect Kafka cu 4 partitii

Entitățile componente utilizate în laborator, precum și relațiile dintre acestea sunt scoase în evidență în următoarea diagramă:

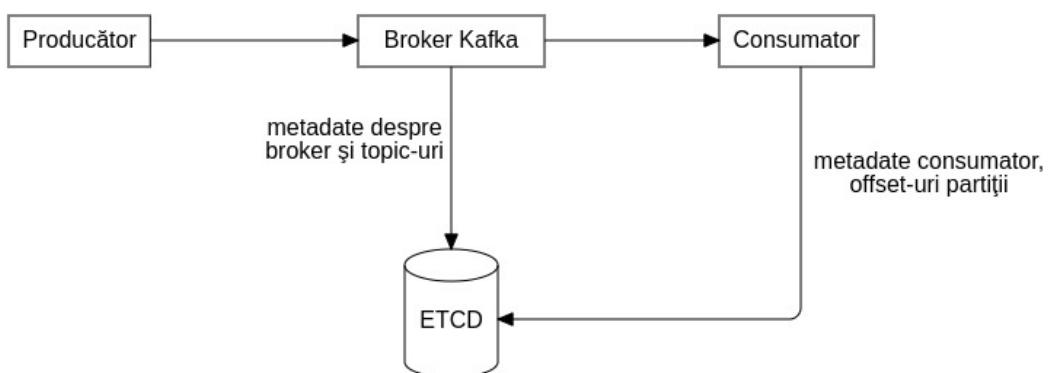


Figura 2 - Funcționarea Kafka și ETCD

ETCD

ETCD este un depozit distribuit de tip cheie-valoare, care asigură o consistență sporită și furnizează o modalitate sigură de a stoca date ce trebuie accesate de un sistem distribuit sau de un cluster de mașini de calcul. De asemenea, gestionează automat alegerile de lider din timpul partaționărilor rețelei și poate tolera defecte ale mașinilor de calcul, chiar și la nivelul nodului lider.

Datele sunt stocate în directoare organizate ierarhic, asemenea unui sistem de fișiere clasic. Sistemul de stocare persistă mai multe versiuni ale datelor simultan: atunci când o valoare este înlocuită de una nouă, se păstrează versiunea anterioară a acesteia. Depozitul cheie-valoare este imutabil, operațiile nu actualizează structura de date, ci mereu generează o altă structură actualizată, având posibilitatea de a accesa și versiunile anterioare ale valorilor corespunzătoare cheilor.

Valorile scrise într-un depozit ETCD pot fi urmărite pentru modificări, astfel încât aplicațiile care le folosesc să fie notificate pentru eventuale reconfigurări.

Instalare și configurare Apache Kafka

Pe lângă server-ul Kafka propriu-zis, trebuie instalat și un serviciu de configurare distribuit, pe care Kafka îl utilizează pentru a stoca metadatele *broker*-ilor: nodurile componente din cluster, *topic*-urile, partațiiile etc.

Aveți nevoie de **JRE versiunea 8 și de Docker Engine** instalate pe sistemul dvs. pentru ca pașii următori să funcționeze corect.

Pornire serviciu de configurare ETCD

Pentru laborator, s-a ales sistemul de stocare distribuit de tip cheie-valoare **ETCD** (<https://github.com/etcd-io/etcd>).

Mai întâi, creați folder-ele în care ETCD își va stoca fișierele temporare, respectiv datele și asignați-i permisiunile corecte:

```
sudo mkdir /var/etcd-data /var/etcd-data.tmp  
sudo chgrp docker /var/etcd-data /var/etcd-data.tmp  
sudo chmod g=rwx /var/etcd-data /var/etcd-data.tmp
```

Apoi, utilizând Docker, porniți o instanță locală de ETCD, cu următoarea comandă (**dacă nu doriți repornirea automată a containerului, eliminați parametrul marcat cu roșu**):

```
docker run -d --restart=always -p 2379:2379 -p 2380:2380 --mount  
type=bind,source=/var/etcd-data.tmp,destination=/var/etcd-data --name  
etcd-gcr-v3.4.7 gcr.io/etcd-development/etcd:v3.4.7  
/usr/local/bin/etcd --name s1 --data-dir /var/etcd-data --listen-  
client-urls http://0.0.0.0:2379 --advertise-client-urls  
http://0.0.0.0:2379 --listen-peer-urls http://0.0.0.0:2380 --initial-  
advertise-peer-urls http://0.0.0.0:2380 --initial-cluster  
s1=http://0.0.0.0:2380 --initial-cluster-token tkn --initial-cluster-  
state new --log-level info --logger zap --log-outputs stderr
```

Folosiți comanda **docker logs** ca să verificați funcționarea serviciului ETCD. Preluați ID-ul container-ului Docker returnat de comanda anterioară (sau folosiți comanda **docker ps**) și execuția:

```
docker logs <ID_CONTAINER_ETCD>
```

Un exemplu de output este cel din figura următoare:

```
cosmin@debian-gl553v:~$ docker logs a68338afe01c27e71d701ffc9a083ab8436113fa2212e1a58668386b899fc67f
{"level": "info", "ts": "2020-04-21T15:52:15.553Z", "caller": "etcdmain/etcd.go:134", "msg": "server has been already initialized", "data-dir": "/var/etcd-data", "dir-type": "member"}
{"level": "info", "ts": "2020-04-21T15:52:15.553Z", "caller": "embed/etcd.go:117", "msg": "configuring peer listeners", "listen-peer-urls": ["http://0.0.0.0:2380"]}
{"level": "info", "ts": "2020-04-21T15:52:15.553Z", "caller": "embed/etcd.go:127", "msg": "configuring client listeners", "listen-client-urls": ["http://0.0.0.0:2379"]}
{"level": "info", "ts": "2020-04-21T15:52:15.553Z", "caller": "embed/etcd.go:299", "msg": "starting an etcd server", "etcd-version": "3.4.7", "git-sha": "e694b7bb0", "go-version": "go1.17", "go-os": "linux", "go-arch": "amd64", "max-cpu-set": 8, "max-cpu-available": 8, "member-initialized": true, "name": "s1", "data-dir": "/var/etcd-data", "wal-dir": "", "wal-dir-dedicated": "", "member-dir": "/var/etcd-data/member", "force-new-cluster": false, "heartbeat-interval": "100ms", "election-timeout": "1s", "initial-election-tick-advance": true, "snapshot-count": 100000, "snapshot-catchup-entries": 5000, "initial-advertise-peer-urls": ["http://0.0.0.0:2380"], "listen-peer-urls": ["http://0.0.0.0:2380"], "advertise-client-urls": ["http://0.0.0.0:2379"], "listen-client-urls": ["http://0.0.0.0:2379"], "listen-metrics-urls": [{"cors": "*"}], "host-whitelist": ["*"], "initial-cluster": "", "initial-cluster-state": "new", "initial-cluster-token": "", "quota-size-bytes": 2147483648, "pre-vote": false, "initial-corrupt-check": false, "corrupt-check-time-interval": "0s", "auto-compaction-mode": "periodic", "auto-compaction-retention": "0s", "auto-compaction-interval": "0s", "discovery-url": "", "discovery-proxy": ""}
{"level": "info", "ts": "2020-04-21T15:52:15.554Z", "caller": "etcdserver/backend.go:79", "msg": "opened backend db", "path": "/var/etcd-data/member/snap/db", "took": "522.232µs"}
{"level": "info", "ts": "2020-04-21T15:52:15.557Z", "caller": "etcdserver/raft.go:506", "msg": "restarting local member", "cluster-id": "f9f44c4ba0e96dd8", "local-member-id": "59a9c584ea2c3f35", "commit-index": 382}
{"level": "info", "ts": "2020-04-21T15:52:15.557Z", "caller": "raft/raft.go:1530", "msg": "59a9c584ea2c3f35 switched to configuration voters={()}"}
{"level": "info", "ts": "2020-04-21T15:52:15.557Z", "caller": "raft/raft.go:700", "msg": "59a9c584ea2c3f35 became follower at term 4"}
{"level": "info", "ts": "2020-04-21T15:52:15.557Z", "caller": "raft/raft.go:383", "msg": "newRaft 59a9c584ea2c3f35 [peers: [], term: 4, commit: 382, applied: 0, lastindex: 382, lastterm: 4]"}
{"level": "warn", "ts": "2020-04-21T15:52:15.560Z", "caller": "auth/store.go:1318", "msg": "simple token is not cryptographically signed"}
{"level": "info", "ts": "2020-04-21T15:52:15.561Z", "caller": "etcdserver/quota.go:98", "msg": "enabled backend quota with default value", "quota-name": "v3-applier", "quota-size-bytes": 2147483648, "quota-size": "2.1 GB"}
{"level": "info", "ts": "2020-04-21T15:52:15.562Z", "caller": "etcdserver/server.go:792", "msg": "starting etcd server", "local-member-id": "59a9c584ea2c3f35", "local-server-version": "3.4.7", "cluster-version": "to be decided"}
{"level": "info", "ts": "2020-04-21T15:52:15.562Z", "caller": "etcdserver/server.go:680", "msg": "starting initial election tick advance", "election-ticks": 10}
{"level": "info", "ts": "2020-04-21T15:52:15.562Z", "caller": "raft/raft.go:1530", "msg": "59a9c584ea2c3f35 switched to configuration voters=(6460912315094810421)"}
{"level": "info", "ts": "2020-04-21T15:52:15.562Z", "caller": "membership/cluster.go:392", "msg": "added member", "cluster-id": "f9f44c4ba0e96dd8", "local-member-id": "59a9c584ea2c3f35", "added-peer-id": "59a9c584ea2c3f35", "added-peer-urls": ["http://0.0.0.0:2380"]}
{"level": "info", "ts": "2020-04-21T15:52:15.562Z", "caller": "membership/cluster.go:558", "msg": "set initial cluster version", "cluster-id": "f9f44c4ba0e96dd8", "local-member-id": "59a9c584ea2c3f35", "cluster-version": "3.4"}
{"level": "info", "ts": "2020-04-21T15:52:15.562Z", "caller": "api/capability.go:76", "msg": "enabled capabilities for version", "cluster-version": "3.4"}
{"level": "info", "ts": "2020-04-21T15:52:15.563Z", "caller": "embed/etcd.go:241", "msg": "now serving peer/client/metrics", "local-member-id": "59a9c584ea2c3f35", "initial-advertise-peer-urls": ["http://0.0.0.0:2380"], "listen-peer-urls": ["http://0.0.0.0:2380"], "advertise-client-urls": ["http://0.0.0.0:2379"], "listen-client-urls": ["http://0.0.0.0:2379"], "listen-metrics-urls": []}
{"level": "info", "ts": "2020-04-21T15:52:15.563Z", "caller": "embed/etcd.go:576", "msg": "serving peer traffic", "address": "[::]:2380"}
{"level": "info", "ts": "2020-04-21T15:52:17.458Z", "caller": "raft/raft.go:923", "msg": "59a9c584ea2c3f35 is starting a new election at term 4"}
{"level": "info", "ts": "2020-04-21T15:52:17.458Z", "caller": "raft/raft.go:713", "msg": "59a9c584ea2c3f35 became candidate at term 5"}
{"level": "info", "ts": "2020-04-21T15:52:17.458Z", "caller": "raft/raft.go:824", "msg": "59a9c584ea2c3f35 received MsgVoteResp from 59a9c584ea2c3f35 at term 5"}
{"level": "info", "ts": "2020-04-21T15:52:17.458Z", "caller": "raft/raft.go:765", "msg": "59a9c584ea2c3f35 became leader at term 5"}
{"level": "info", "ts": "2020-04-21T15:52:17.458Z", "caller": "raft/node.go:325", "msg": "raft.node: 59a9c584ea2c3f35 elected leader 59a9c584ea2c3f35 at term 5"}
{"level": "info", "ts": "2020-04-21T15:52:17.459Z", "caller": "etcdserver/server.go:2026", "msg": "published local member to cluster through raft", "local-member-id": "59a9c584ea2c3f35", "local-member-attributes": "Name:s1 ClientURLs:[http://0.0.0.0:2379]", "request-path": "/members/59a9c584ea2c3f35/attributes", "cluster-id": "f9f44c4ba0e96dd8", "publish-timeout": "7s"}
{"level": "info", "ts": "2020-04-21T15:52:17.460Z", "caller": "embed/server.go:139", "msg": "serving client traffic insecurely: this is strongly discouraged!", "address": "[::]:2379"}
```

Figura 3 - Verificarea funcționării serviciului ETCD containerizat

Compilare Kafka cu suport ETCD

Aveți nevoie de utilitarul gradle instalat pentru a putea compila Kafka.

(se poate instala pe Debian 10 folosind comanda: **sudo apt install gradle**)

Se clonează repository-ul GitHub:

```
git clone https://github.com/banzaicloud/apache-kafka-on-k8s.git
```

Se modifică fișierul **build.gradle**, adăugând un parametru nou în vectorul **scalaCompileOptions.additionalParameters** de pe linia 325:

```
"-target:jvm-1.8"
```

În folderul **apache-kafka-on-k8s** se execută comanda:

```
gradle
```

Apoi se compilează proiectul cu comanda:

```
./gradlew releaseTarGz -x signArchives
```

Rezultatul compilării este o arhivă numită **kafka_2.11-2.0.0-SNAPSHOT.tgz**, care se regăsește în folder-ul **core/build/distributions**, relativ la folder-ul proiectului.

Instalare Apache Kafka

Folosiți versiunea modificată de Apache Kafka, compilată cu pașii de mai sus, care poate utiliza ETCD cu rol de serviciu de configurare. Se consideră că aveți arhiva **kafka_2.11-2.0.0-SNAPSHOT.tgz** obținută în urma compilării Kafka.

Dezarhivați arhiva:

```
tar -xzvf kafka_2.11-2.0.0-SNAPSHOT.tgz
```

Copiați conținutul în folder-ul **/opt**:

```
sudo mv kafka_2.11-2.0.0-SNAPSHOT /opt/kafka
```

Asignați-vă utilizatorul și grupa ca și proprietari ai folder-ului **kafka**:

```
sudo chown -R $USER:$groups $(groups | awk '{ print $1 }') /opt/kafka
```

Configurare Kafka pentru utilizarea ETCD ca serviciu de configurare

Deschideți fișierul **/opt/kafka/config/server.properties** cu un editor de text și modificați proprietatea **metastore.connect** astfel încât să aibă valoarea **etcd://localhost:2379**.

```
112 # The interval at which log segments are checked to see if they can be deleted according
113 # to the retention policies
114 log.retention.check.interval.ms=300000
115
116 ##### Zookeeper #####
117
118 # MetaStore connection string without any prefix it will use zk as a metastore,
119 # but if you prefixes it with etcd:// it will use etcd as a metastore.
120 # This is a comma separated host:port pairs, each corresponding to a metaStore
121 # server. e.g. "127.0.0.1:3000,127.0.0.1:3001,127.0.0.1:3002".
122 # You can also append an optional chroot string to the urls to specify the
123 # root directory for all kafka znodes.
124 metastore.connect=etcd://localhost:2379
125
126 # Timeout in ms for connecting to zookeeper
127 zookeeper.connection.timeout.ms=6000
128
129
130 ##### Group Coordinator Settings #####
131
```

Figura 4 - Setarea ETCD ca serviciu de configurare Kafka

Pornire server Kafka

Nu uitați să aveți pornit containerul Docker cu ETCD înainte de a porni Kafka!

Server-ul Apache Kafka poate fi pornit astfel:

```
sudo /opt/kafka/bin/kafka-server-start.sh -daemon
/opt/kafka/config/server.properties
```

(comanda nu afișează nimic după execuție)

Se verifică apoi pornirea corectă a server-ului:

```
ps -ef | grep kafka
```

Dacă server-ul a pornit fără erori, comanda anterioară va returna un proces Java cu mulți parametri în linie de comandă:

Aplicații flux și monitorizare utilizând Apache Kafka

```
File Edit View Terminal Tabs Help
cosmin@debian-g1553v:~$ ps -ef | grep kafka
root      2572   1 54 20:26 pts/3    00:00:05 java -Xmx1G -Xms1G -server -XX:+UseG1GC -XX:MaxGCPauseMillis=20 -XX:InitiatingHeapOccupancyPercent=35 -XX:+ExplicitGCInvokesConcurrent -Djava.awt.headless=true -Xloggc:/opt/kafka/bin/..../logs/kafkaServer-gc.log -verbose:gc -XX:+PrintGCDetails -XX:+PrintGCDetails -XX:+PrintGCDateStamps -XX:+PrintGCTimeStamps -XX:+UseGCLogFileRotation -XX:NumberOfGCLogFiles=10 -XX:GCLogFileSize=100M -Dcom.sun.management.jmxremote -Dcom.sun.management.jmxremote.authenticate=false -Dcom.sun.management.jmxremote.ssl=false -Dkafka.logs.dir=/opt/kafka/bin/..../logs -Dlog4j.configuration=file:/opt/kafka/bin/..../config/log4j.properties -cp /opt/kafka/bin/..../libs/activation-1.1.1.jar:/opt/kafka/bin/..../libs/apollo-alliance-repackaged-2.5.0-b32.jar:/opt/kafka/bin/..../libs/argparse4j-0.7.0.jar:/opt/kafka/bin/..../libs/audience-annotations-0.5.0.jar:/opt/kafka/bin/..../libs/commons-lang3-3.5.jar:/opt/kafka/bin/..../libs/connect-api-2.0.0-SNAPSHOT.jar:/opt/kafka/bin/..../libs/connect-file-2.0.0-SNAPSHOT.jar:/opt/kafka/bin/..../libs/connect-json-2.0.0-SNAPSHOT.jar:/opt/kafka/bin/..../libs/connect-runtime-2.0.0-SNAPSHOT.jar:/opt/kafka/bin/..../libs/connect-transforms-2.0.0-SNAPSHOT.jar:/opt/kafka/bin/..../libs/error-prone-annotations-2.1.2.jar:/opt/kafka/bin/..../libs/grpc-context-1.9.1.jar:/opt/kafka/bin/..../libs/grpc-core-1.9.1.jar:/opt/kafka/bin/..../libs/grpc-netty-1.9.1.jar:/opt/kafka/bin/..../libs/grpc-protobuf-lite-1.9.1.jar:/opt/kafka/bin/..../libs/grpc-stub-1.9.1.jar:/opt/kafka/bin/..../libs/grpc-stub-2.7.jar:/opt/kafka/bin/..../libs/guava-19.0.jar:/opt/kafka/bin/..../libs/guava-20.0.jar:/opt/kafka/bin/..../libs/hk2-api-2.5.0-b32.jar:/opt/kafka/bin/..../libs/hk2-locator-2.5.0-b32.jar:/opt/kafka/bin/..../libs/hk2-utils-2.5.0-b32.jar:/opt/kafka/bin/..../libs/instrumentation-api-0.4.3.jar:/opt/kafka/bin/..../libs/jackson-annotations-2.9.5.jar:/opt/kafka/bin/..../libs/jackson-core-2.9.5.jar:/opt/kafka/bin/..../libs/jackson-data-bind-2.9.5.jar:/opt/kafka/bin/..../libs/jackson-jaxrs-base-2.9.5.jar:/opt/kafka/bin/..../libs/jackson-jaxrs-json-provider-2.9.5.jar:/opt/kafka/bin/..../libs/jackson-module-jaxb-annotations-2.9.5.jar:/opt/kafka/bin/..../libs/javassist-3.20.0-GA.jar:/opt/kafka/bin/..../libs/javassist-3.21.0-GA.jar:/opt/kafka/bin/..../libs/javax.annotation-api-1.2.jar:/opt/kafka/bin/..../libs/javax.inject-1.jar:/opt/kafka/bin/..../libs/javax.inject-2.5.0-b32.jar:/opt/kafka/bin/..../libs/javax.servlet-api-3.1.0.jar:/opt/kafka/bin/..../libs/javax.ws.rs-api-2.0.1.jar:/opt/kafka/bin/..../libs/jaxb-api-2.3.0.jar:/opt/kafka/bin/..../libs/jersey-client-2.25.1.jar:/opt/kafka/bin/..../libs/jersey-container-servlet-2.25.1.jar:/opt/kafka/bin/..../libs/jersey-media-jaxb-2.25.1.jar:/opt/kafka/bin/..../libs/jersey-server-2.25.1.jar:/opt/kafka/bin/..../libs/jetcd-common-0.0.2.jar:/opt/kafka/bin/..../libs/jetcd-core-0.0.2.jar:/opt/kafka/bin/..../libs/jetcd-resolver-0.0.2.jar:/opt/kafka/bin/..../libs/jetty-client-9.2.24.v20180105.jar:/opt/kafka/bin/..../libs/jetty-continuation-9.2.24.v20180105.jar:/opt/kafka/bin/..../libs/jetty-http-9.2.24.v20180105.jar:/opt/kafka/bin/..../libs/jetty-io-9.2.24.v20180105.jar:/opt/kafka/bin/..../libs/jetty-continuation-9.2.24.v20180105.jar:/opt/kafka/bin/..../libs/jetty-server-9.2.24.v20180105.jar:/opt/kafka/bin/..../libs/jetty-servlet-9.2.24.v20180105.jar:/opt/kafka/bin/..../libs/jetty-util-9.2.24.v20180105.jar:/opt/kafka/bin/..../libs/jopt-simple-5.0.4.jar:/opt/kafka/bin/..../libs/jsr305-3.0.0.jar:/opt/kafka/bin/..../libs/kafka-2.11-2.0.0-SNAPSHOT-jar:/opt/kafka/bin/..../libs/kafka-2.11-2.0.0-SNAPSHOT-sources.jar:/opt/kafka/bin/..../libs/kafka-clients-2.0.0-SNAPSHOT.jar:/opt/kafka/bin/..../libs/kafka-log4j-appender-2.0.0-SNAPSHOT.jar:/opt/kafka/bin/..../libs/kafka-streams-2.0.0-SNAPSHOT.jar:/opt/kafka/bin/..../libs/kafka-streams-examples-2.0.0-SNAPSHOT.jar:/opt/kafka/bin/..../libs/kafka-streams-scala-2.11-2.0.0-SNAPSHOT.jar:/opt/kafka/bin/..../libs/kafka-tools-2.0.0-SNAPSHOT.jar:/opt/kafka/bin/..../libs/Log4j-2.17.jar:/opt/kafka/bin/..../libs/lz4-java-1.4.1.jar:/opt/kafka/bin/..../libs/maven-artifact-3.5.3.jar:/opt/kafka/bin/..../libs/metrics-core-2.2.0.jar:/opt/kafka/bin/..../libs/netty-buffer-4.1.17.Final.jar:/opt/kafka/bin/..../libs/netty-codec-socks-4.1.17.Final.jar:/opt/kafka/bin/..../libs/netty-codec-http2-4.1.17.Final.jar:/opt/kafka/bin/..../libs/netty-common-4.1.17.Final.jar:/opt/kafka/bin/..../libs/netty-handler-4.1.17.Final.jar:/opt/kafka/bin/..../libs/netty-handler-proxy-4.1.17.Final.jar:/opt/kafka/bin/..../libs/netty-resolver-4.1.17.Final.jar:/opt/kafka/bin/..../libs/netty-transport-4.1.17.Final.jar:/opt/kafka/bin/..../libs/opencensus-api-0.10.0.jar:/opt/kafka/bin/..../libs/opencensus-contrib-grpc-metrics-0.10.0.jar:/opt/kafka/bin/..../libs/osgi-resource-locator-1.0.1.jar:/opt/kafka/bin/..../libs/plexus-utils-3.1.0.jar:/opt/kafka/bin/..../libs/protobuf-java-3.5.1.jar:/opt/kafka/bin/..../libs/protobuf-java-util-3.5.1.jar:/opt/kafka/bin/..../libs/proto-google-common-protos-1.0.0.jar:/opt/kafka/bin/..../libs/reflection-0.9.11.jar:/opt/kafka/bin/..../libs/rocksdbjni-5.7.3.jar:/opt/kafka/bin/..../libs/scala-library-2.11.12.jar:/opt/kafka/bin/..../libs/snappy-java-1.1.7.1.jar:/opt/kafka/bin/..../libs/validation-api-1.1.0.Final.jar:/opt/kafka/bin/..../libs/zkclient-0.10.jar:/opt/kafka/bin/..../libs/zookeeper-3.4.12.jar kafka.Kafka /opt/kafka/config/server.properties
cosmin 25879 30836 0 20:26 pts/3    00:00:00 grep kafka
```

Figura 5 - Verificarea funcționării server-ului Kafka

Testare broker Kafka

În continuare, testați **crearea unui subiect (topic)**:

```
sudo /opt/kafka/bin/kafka-topics.sh --create --zookeeper etcd://localhost:2379 --replication-factor 1 --partitions 1 --topic test
```

Verificați *topic*-ul creat astfel:

```
sudo /opt/kafka/bin/kafka-topics.sh --zookeeper etcd://localhost:2379 --describe --topic test
```

Testați producerea unor mesaje *dummy* pentru *topic*-ul creat mai sus:

```
sudo /opt/kafka/bin/kafka-console-producer.sh --broker-list localhost:9092 --topic test
```

Comanda de mai sus va aștepta să introduceți, linie cu linie, câte un mesaj. Introduceți câte unul, după care apăsați ENTER. Când ați terminat, apăsați CTRL+D.

Testați acum consumarea mesajelor de test produse mai sus:

```
sudo /opt/kafka/bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 --topic test --from-beginning
```

Comanda va prelua mesajele de test anterioare și le va afișa pe rând la consolă. Pentru a elibera consola și a ieși din procesul consumator de mesaje, apăsați CTRL+C.

Pentru a sterge un subiect (topic), puteți folosi comanda:

```
sudo /opt/kafka/bin/kafka-topics.sh --delete --zookeeper etcd://localhost:2379 --topic NUME_TOPIC
```

Atenție: ștergerea unui topic este de tip *lazy*. Nu are efect imediat!

Nu uitați că, atât pe parcursul laboratorului, cât și atunci când se lucrează cu aplicațiile din laborator, aveți nevoie de instanța containerizată de ETCD și de server-ul Apache Kafka pornite!

Exemplu de aplicație Kafka utilizând Python

Creați un proiect Python, iar în mediul virtual instalat modulul **kafka-python**, deoarece interfațarea cu server-ul Kafka se face utilizând API-ul expus de acest modul (<https://github.com/dpkp/kafka-python>).

```
python3 -m venv exemplu_kafka_env
source ./exemplu_kafka_env/bin/activate
pip3 install kafka-python
```

Dacă doriti să executați codul Python direct din PyCharm, se poate instala modulul **kafka-python astfel:** File → Settings → Project: NUME_PROJECT → Python Interpreter → apăsați pe semnul plus din dreapta (**Install**) și căutați „**kafka-python**”. Apăsați pe „**Install Package**” după ce ați selectat modulul din lista de rezultate.

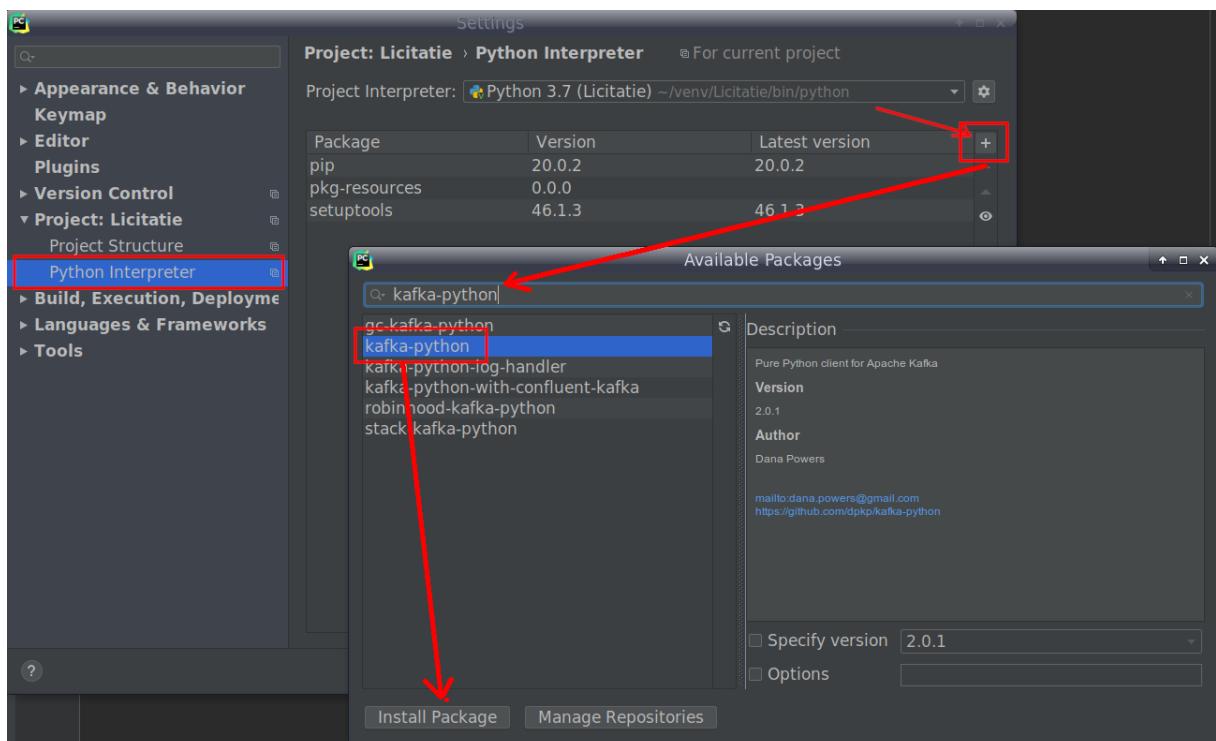


Figura 6 - Instalarea modulului **kafka-python** din PyCharm

Următorul cod exemplu folosește Kafka pentru a trimite mesaje între 2 entități: un *thread* producător de mesaje și un *thread* consumator de mesaje. Cele 2 *thread*-uri comunică prin *topicul topic_exemplu_python*. Funcționarea aplicației este redată în următoarea diagramă de activitate:

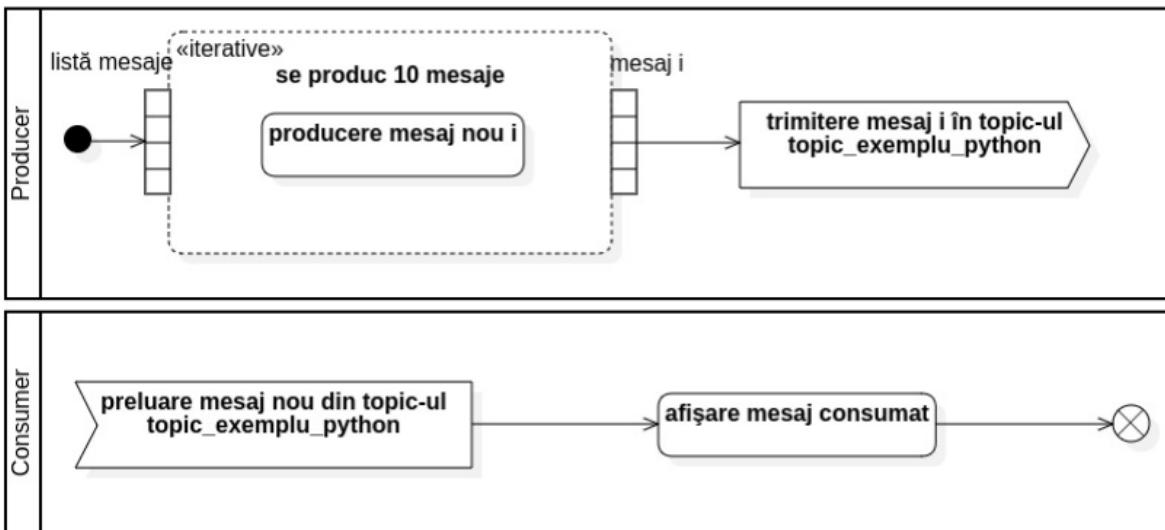


Figura 7 - Funcționarea aplicației exemplu de tip producător-consumator

• exemplu.py

```
from kafka import KafkaConsumer
from kafka import KafkaProducer
import threading

class Consumer(threading.Thread):
    def __init__(self, topic):
        super().__init__()
        self.topic = topic

    def run(self) -> None:
        consumer = KafkaConsumer(self.topic,
auto_offset_reset='earliest')
        # topicul va fi creat automat, daca nu exista deja

        # thread-ul consumator primește mesajele din topic
        for msg in consumer:
            print("Am consumat mesajul: " + str(msg.value,
encoding="utf-8"))

class Producer(threading.Thread):
    def __init__(self, topic):
        super().__init__()
        self.topic = topic

    def run(self) -> None:
        producer = KafkaProducer()
        for i in range(10):
            message = 'mesaj {}'.format(i)

            # thread-ul producator trimite mesaje catre un topic
            producer.send(topic=self.topic, value=bytearray(message,
encoding="utf-8"))
            print("Am produs mesajul: {}".format(message))
```

```
# metoda flush() asigura trimiterea batch-ului de mesaje
produse
    producer.flush()

if __name__ == '__main__':
    # se creeaza 2 thread-uri: unul producator de mesaje si celalalt
    # consumator
    producer_thread = Producer("topic_exemplu_python")
    consumer_thread = Consumer("topic_exemplu_python")

    producer_thread.start()
    consumer_thread.start()

    producer_thread.join()
    consumer_thread.join()
```

Încercați să executați aplicația de 2-3 ori. Ce observați că afișează *thread*-ul consumator de mesaje? De ce apar și mesajele anterioare?

Exemplu de aplicație Kafka utilizând Kotlin

Aplicația Kotlin va expune un *endpoint* web care răspunde la o cerere HTTP de tip PUT pe calea `/publish`. Pentru fiecare cerere HTTP primită, se preia corpul cererii, iar acesta va fi produs sub formă de mesaj Kafka în *topic*-ul `topic_exemplu_kotlin`.

Consumatorul este o componentă Spring separată, sub formă de *listener* Kafka. Atunci când în *topic*-ul menționat apare un mesaj nou, *listener*-ul consumă respectivul mesaj și așteaptă în continuare alte mesaje.

Creați un proiect **Spring Boot** (ca în laboratorul 3), **fără să adăugați dependența Spring Web**.

Adăugați dependența **Spring Kafka** astfel:

• pentru **Maven**:

```
<dependency>
    <groupId>org.springframework.kafka</groupId>
    <artifactId>spring-kafka</artifactId>
    <version>2.2.9.RELEASE</version>
</dependency>
```

• pentru **Gradle**:

```
compile group: 'org.springframework.kafka', name: 'spring-kafka',
version: '2.2.9.RELEASE'
```

<https://mvnrepository.com/artifact/org.springframework.kafka/spring-kafka>

Atenție la versiune: 2.2.9 funcționează corect, având în vedere combinațiile de versiuni Kafka și Spring utilizate în laborator.

• **Producer.kt**

```
package com.sd.laborator
```

```
import org.springframework.kafka.core.KafkaTemplate
import org.springframework.stereotype.Component

@Component
class KotlinProducer(private val kafkaTemplate: KafkaTemplate<String,
String>) {
    fun send(message: String) {
        kafkaTemplate.send("topic_exemplu_kotlin", message)
    }
}
```

- **Consumer.kt**

```
package com.sd.laborator

import org.springframework.kafka.annotation.KafkaListener
import org.springframework.stereotype.Component

@Component
class KotlinConsumer {
    @KafkaListener(topics = ["topic_exemplu_kotlin"], groupId =
"exemplu-consumer-kotlin")
    fun processMessage(message: String) {
        println("Am consumat urmatorul mesaj: $message")
    }
}
```

- **KafkaController.kt**

```
package com.sd.laborator

import org.springframework.http.HttpStatus
import org.springframework.http.ResponseEntity
import org.springframework.stereotype.Controller
import org.springframework.web.bind.annotation.RequestBody
import org.springframework.web.bind.annotation.RequestMapping
import org.springframework.web.bind.annotation.RequestMethod
import org.springframework.web.bind.annotation.ResponseBody

@Controller
class KafkaController(private val kotlinProducer: KotlinProducer) {
    @RequestMapping(path = ["/publish"], method = [RequestMethod.PUT])
    @ResponseBody
    fun publishMessage(@RequestBody message: String):
    ResponseEntity<HttpStatus> {
        kotlinProducer.send(message)
        return ResponseEntity(HttpStatus.CREATED);
    }
}
```

- **KafkaApp.kt**

```
package com.sd.laborator

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

@SpringBootApplication
```

```
class SpringKafkaApplication

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

Comparați, împachetați și executați aplicația Spring Boot.
Pentru testare, puteți trimite cereri HTTP de tip PUT astfel:

```
curl -X PUT http://localhost:8080/publish --header "Content-Type: text/plain" --data-raw "mesaj test 1"
```

Alternativ, puteți folosi aplicația **Postman**.

Ar trebui să găsiți în log-ul aplicației Spring Boot un mesaj de la componenta **Consumer**, de tipul celui din figură:

```
2020-04-25 11:37:08.955 INFO 9508 --- [nio-8080-exec-4] o.a.kafka.common.utils.AppInfoParser      : Kafka
version : 2.0.1
2020-04-25 11:37:08.955 INFO 9508 --- [nio-8080-exec-4] o.a.kafka.common.utils.AppInfoParser      : Kafka
commitId : fa14705e51bd2ce5
2020-04-25 11:37:08.963 INFO 9508 --- [ad | producer-1] org.apache.kafka.clients.Metadata          : Cluster
ID: TzA2yZeDR0iNMg7Pr39bSg
Am consumat urmatorul mesaj: mesaj test 1
```

Figura 8 - Testarea aplicației Kafka cu Kotlin

Kafka sub formă de broker de mesaje pentru arhitecturi bazate pe flux - procesarea fluxurilor

Se reia aplicația „Okazii” din **laboratorul 7**, care modelează funcționarea unei sesiuni de licitație. De această dată, **comunicarea dintre microserviciile participante la licitație se face printr-un broker de mesaje Kafka**.

Proiectarea inițială a aplicației Okazii folosind Kafka pentru comunicare

Așadar, diagrama de microservicii a aplicației **Okazii** din laboratorul 7 se modifică astfel, la prima vedere:

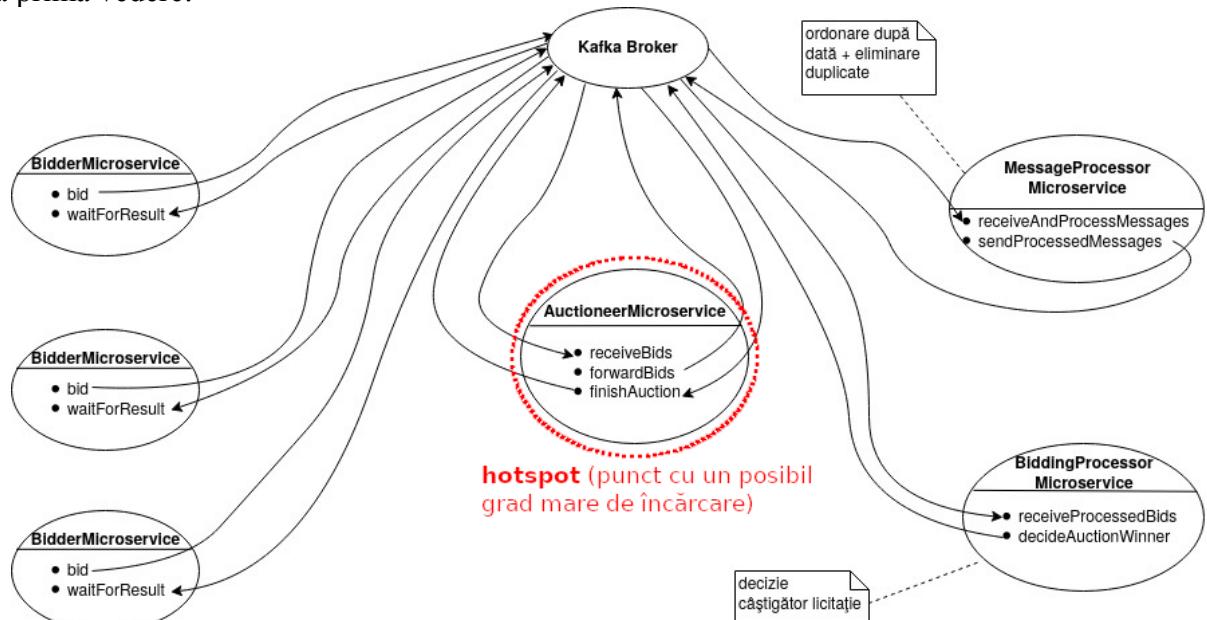


Figura 9 - Diagrama modificată de microservicii a aplicației Okazii (cu *broker* Kafka)

Se observă că toată comunicația se face prin intermediul Kafka. Microserviciile se folosesc de *topic*-uri pentru a transmite mesaje în mod asincron și decuplat de destinatarii acestora. S-a marcat pe diagramă **microserviciul care ar putea avea un nivel mare de încărcare**; atunci când numărul de entități **Bidder** crește, numărul de mesaje primite de **Auctioneer** va fi prea mare pentru a putea fi procesat de o singură instanță.

Se pot face însă câteva observații pentru îmbunătățirea și reproiectarea aplicatiei:

- Odată ce un mesaj este consumat dintr-un *topic*, el rămâne stocat până în momentul în care expiră perioada definită de **politica de retenție** (implicit, mesajele sunt șterse după 168 de ore de la publicare). Așadar, faptul că **AuctioneerMicroservice** consumă un mesaj primit de la o entitate **Bidder** nu determină distrugerea imediată a mesajului din modelul de persistență Kafka.

```

95 ##### Log Retention Policy #####
96
97 # The following configurations control the disposal of log segments. The policy can
98 # be set to delete segments after a period of time, or after a given size has accumulated.
99 # A segment will be deleted whenever *either* of these criteria are met. Deletion always happens
100 # from the end of the log.
101
102 # The minimum age of a log file to be eligible for deletion due to age
103 log.retention.hours=168

```

Figura 10 - Politica de retenție implicită a Kafka (</opt/kafka/config/server.properties>)

- Fiecare consumator are câte un *offset* în cadrul unei partii dintr-un *topic*. Utilizând acel *offset*, consumatorul poate citi mesajele de câte ori dorește, în ce direcție dorește. Așadar, **microservicii diferite pot citi din același topic în mod independent, fiecare având propriul offset**.
 - Kafka permite ca mesajele să fie preluate simultan de mai mulți consumatori. Se pot defini astfel aşa-numitele **grupuri de consumatori** (eng. *consumer groups*). Acest lucru ajută la eliminarea redundanței, în sensul că: spre deosebire de modalitatea de procesare a mesajelor din laboratorul 7, microserviciul **MessageProcessor** își poate prelua singur setul de mesaje pe care trebuie să le proceseze **direct din același topic în care le primește microserviciul Auctioneer** (consecință a observației anterioare).
 - Deci, nu mai este nevoie ca microserviciul **Auctioneer** să redirecționeze mesajele primite către microserviciul **MessageProcessor**.
 - Totodată, microserviciul **BiddingProcessor** nu trebuie să primească efectiv setul procesat de mesaje, ci trebuie doar să citească ce a publicat **MessageProcessor** într-un alt *topic* specific mesajelor „curate” (fără duplicate, ordonate după data și oră)
 - Kafka asigură o **echilibrare minimală a încărcării** prin scalarea pe orizontală a consumatorilor. Așineând fiecărui consumator un identificator în cadrul grupei (*group ID*), Kafka folosește un algoritm de tip **Round Robin** pentru a citi mesajele în mod echilibrat, în funcție de numărul de consumatori din grupă. Acest lucru asigură și **toleranță la defecte**, din punct de vedere al consumatorilor: dacă un microserviciu consumator se defectează, **mesajele vor fi redistribuite în funcție de celelalte entități active din grupul de consumatori** și de numărul de partii în care a fost împărțit *topic*-ul care este folosit. **Mesajele nu se pierd în caz de defecte! Se redistribuie.**
 - de exemplu: fie un *topic* separat în 4 partii, la care se înscrive un grup de 2 consumatori. Primul și al doilea mesaj vor fi primite de **consumatorul 1**, al treilea și al patrulea vor fi primite de **consumatorul 2**, următoarele 2 mesaje vor fi primite din nou de **consumatorul 1** și.m.d.

Reprojecțarea aplicației Okazii

În continuare, se reproiectează aplicația, aplicându-se proprietățile și avantajele oferite de Kafka, conform cu observațiile anterioare.

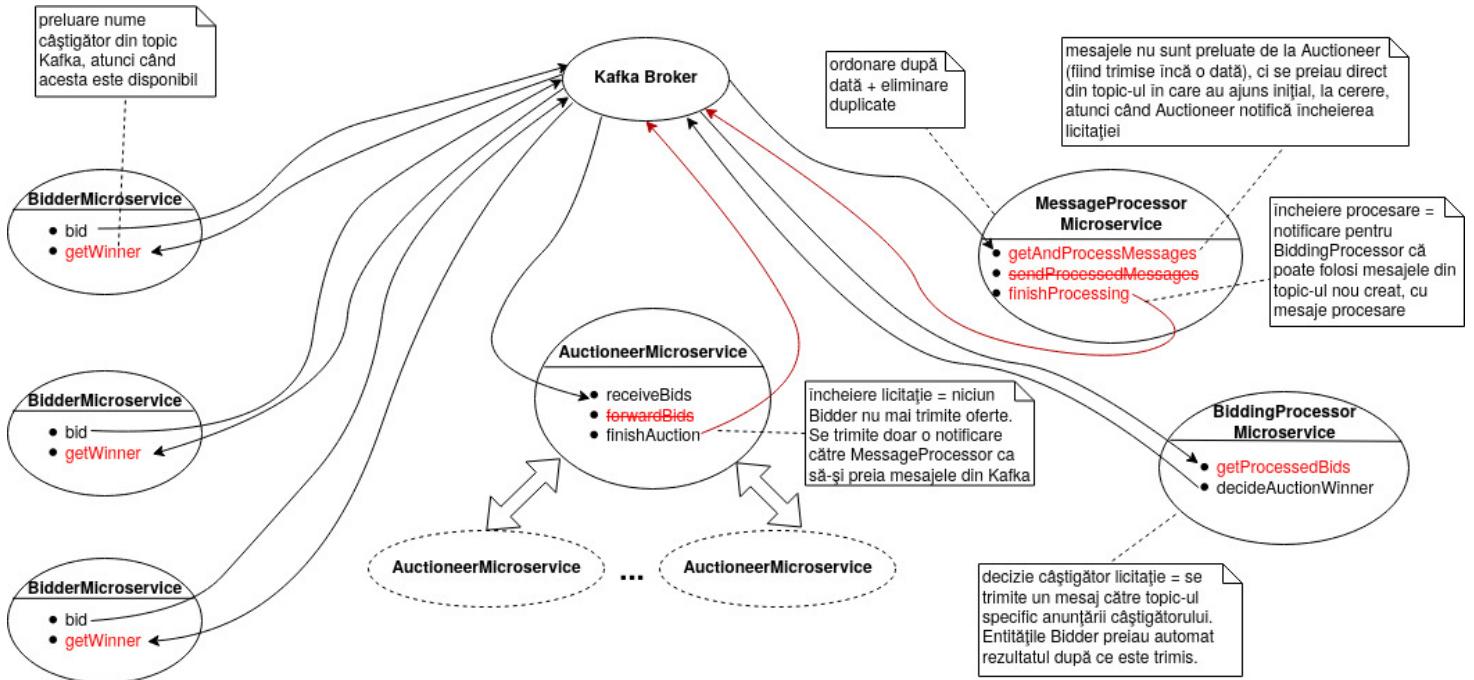


Figura 11 - Diagrama de microservicii reproiectată

S-au marcat pe diagramă modificările aduse microserviciilor. De asemenea, s-a scos în evidență posibilitatea de scalare a microserviciului **Auctioneer**.

Implementarea microserviciilor

Aplicația va fi implementată în Python, folosind modulul **kafka-python** pentru comunicarea cu *broker-ul Kafka*. Diagrama de clase este prezentată în figura următoare:

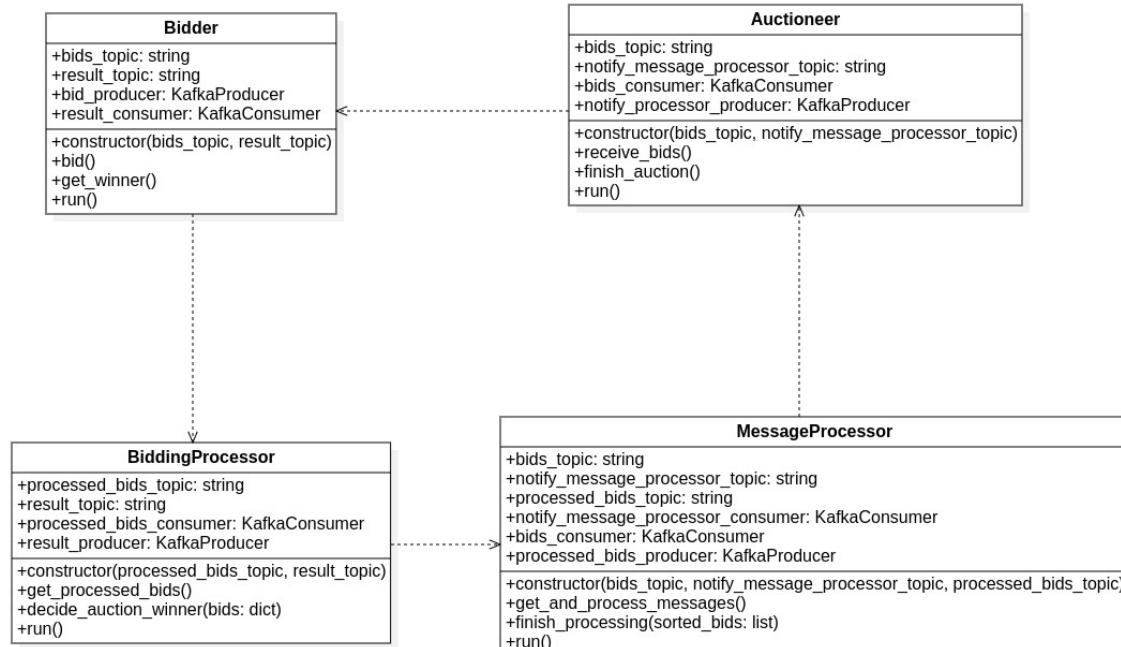


Figura 12 - Diagrama de clase aplicație Okazii

Relațiile de dependență provin de la necesitatea de comunicare prin broker-ul de mesaje. Așadar, aceste relații s-ar putea traduce prin „**are nevoie să comunice cu ...**”, sau „**așteaptă un mesaj de la...**”.

Creați un proiect Python în PyCharm și adăugați fișierele sursă menționate în continuare.

- **Bidder.py**

```
from kafka import KafkaProducer, KafkaConsumer
from random import randint
from uuid import uuid4

class Bidder:
    def __init__(self, bids_topic, result_topic):
        super().__init__()
        self.bids_topic = bids_topic
        self.result_topic = result_topic

        # producatorul pentru oferte de licitatie
        self.bid_producer = KafkaProducer()

        # consumatorul pentru rezultatul licitatiei
        self.result_consumer = KafkaConsumer(
            self.result_topic,
            auto_offset_reset="earliest" # mesajele se preiau de la
cel mai vechi la cel mai recent
        )

        self.my_bid = randint(1000, 10_000) # se genereaza oferta ca
numar aleator intre 1000 si 10.000
        self.my_id = uuid4() # se genereaza un identificator unic
pentru ofertant

    def bid(self):
        # se construieste mesajul pentru licitare
        print("Trimite licitatia mea: {}...".format(self.my_bid))
        bid_message = bytearray("licitez", encoding="utf-8") # corpul
contine doar mesajul "licitez"
        bid_headers = [ # antetul mesajului contine identitatea
ofertantului si, respectiv, oferta sa
            ("amount", self.my_bid.to_bytes(2, byteorder='big')),
            ("identity", bytes("Bidder {}".format(self.my_id),
encoding="utf-8"))
        ]

        # se trimite licitatia sub forma de mesaj catre Kafka
        self.bid_producer.send(topic=self.bids_topic,
value=bid_message, headers=bid_headers)

        # exista o sansa din 2 ca oferta sa fie trimisa de 2 ori
pentru a simula duplicatele
        if randint(0, 1) == 1:
            self.bid_producer.send(topic=self.bids_topic,
```

```

value=bid_message, headers=bid_headers)

        self.bid_producer.flush()
        self.bid_producer.close()

    def get_winner(self):
        # se asteapta raspunsul licitatiei
        print("Astept rezultatul licitatiei...")
        result = next(self.result_consumer)

        # se verifica identitatea castigatorului
        for header in result.headers:
            if header[0] == "identity":
                identity = str(header[1], encoding="utf-8")

        if identity == self.my_id:
            print("Am castigat!!!")
        else:
            print("Am pierdut...")

        self.result_consumer.close()

    def run(self):
        self.bid()
        self.get_winner()

if __name__ == '__main__':
    bidder = Bidder(
        bids_topic="topic_oferte",
        result_topic="topic_rezultat"
    )
    bidder.run()

```

Microserviciul **Bidder** se folosește de 2 *topic*-uri Kafka:

- **topic_oferte** → aici își va publica oferta (generată aleator) sub formă de mesaj Kafka. Mesajul este de forma:
 - corp: sirul de caractere „**licitez**”
 - metadate:
 - **amount: PRETUL_LICITAT**
 - **identity: IDENTIFICATOR_OFERTANT**
- **topic_rezultat** → din acest *topic* va consuma un singur mesaj: rezultatul licitației care va apărea în momentul în care **BiddingProcessor** va stabili câștigătorul. **Bidder**-ul va aștepta până se primește răspunsul în acest topic.

• Auctioneer.py

```

from kafka import KafkaConsumer, KafkaProducer

class Auctioneer:
    def __init__(self, bids_topic, notify_message_processor_topic):
        super().__init__()
        self.bids_topic = bids_topic
        self.notify_processor_topic = notify_message_processor_topic

```

```

# consumatorul pentru ofertele de la licitatie
self.bids_consumer = KafkaConsumer(
    self.bids_topic,
    auto_offset_reset="earliest", # mesajele se preiau de la
cel mai vechi la cel mai recent
    group_id="auctioneers",
    consumer_timeout_ms=15_000 # timeout de 15 secunde
)

# producatorul pentru notificarea procesorului de mesaje
self.notify_processor_producer = KafkaProducer()

def receive_bids(self):
    # se preiau toate ofertele din topicul bids_topic
    print("Astept oferte pentru licitatie...")
    for msg in self.bids_consumer:
        for header in msg.headers:
            if header[0] == "identity":
                identity = str(header[1], encoding="utf-8")
            elif header[0] == "amount":
                bid_amount = int.from_bytes(header[1], 'big')

        print("{} a licitat {}".format(identity, bid_amount))

    # bids_consumer genereaza exceptia StopIteration atunci cand
    # se atinge timeout-ul de 10 secunde
    # => licitatia se incheie dupa ce timp de 15 secunde nu s-a
    # primit nicio oferta
    self.finish_auction()

def finish_auction(self):
    print("Licitatia s-a incheiat!")
    self.bids_consumer.close()

    # se notifica MessageProcessor ca poate incepe procesarea
    # mesajelor
    auction_finished_message = bytarray("incheiat",
encoding="utf-8")

self.notify_processor_producer.send(topic=self.notify_processor_topic,
value=auction_finished_message)
    self.notify_processor_producer.flush()
    self.notify_processor_producer.close()

def run(self):
    self.receive_bids()

if __name__ == '__main__':
    auctioneer = Auctioneer(
        bids_topic="topic_oferte",

notify_message_processor_topic="topic_notificare_procesor_mesaje"
    )
    auctioneer.run()

```

Microserviciul **Auctioneer** folosește 2 *topic*-uri Kafka, astfel:

- **topic_oferte** → de aici consumă mesajele cu oferte primite de la microserviciile **Bidder**. Se observă că fiecare instanță de **Auctioneer** este asignată ca și consumator în grupul de consumatori „**auctioneers**” - a se vedea parametrul **group_id="auctioneers"** adăugat la construirea obiectului **KafkaConsumer**. *Topic*-ul este împărțit în 4 partii, și deci Kafka va distribui mesajele în mod echilibrat între consumatorii aceluiași grup. Dacă se pornesc mai multe instanțe de **Auctioneer**, fiecare va primi o parte din fluxul de mesaje de la entitățile **Bidder**.
- **topic_notificare_procesor_mesaje** → deoarece licitația se încheie după 15 secunde de inactivitate la așteptarea de oferte (conform parametrului **consumer_timeout_ms=15_000**), **MessageProcessor** este notificat prin acest *topic* atunci când nu se mai primesc oferte, ca el să poată prelua mesajele spre procesare.

• MessageProcessor.py

```
from datetime import datetime
from kafka import KafkaConsumer, KafkaProducer

class MessageProcessor:
    def __init__(self, bids_topic, notify_message_processor_topic,
                 processed_bids_topic):
        super().__init__()
        self.bids_topic = bids_topic
        self.notify_message_processor_topic =
notify_message_processor_topic
        self.processed_bids_topic = processed_bids_topic

        # consumatorul notificarii de la Auctioneer cum ca s-a
terminat licitatia
        self.notify_message_processor_consumer = KafkaConsumer(
            self.notify_message_processor_topic,
            auto_offset_reset="earliest" # mesajele se preiau de la
cel mai vechi la cel mai recent
        )

        # consumatorul pentru ofertele de la licitatie
        self.bids_consumer = KafkaConsumer(
            self.bids_topic,
            auto_offset_reset="earliest",
            consumer_timeout_ms=1000
        )

        # producatorul pentru mesajele procesate
        self.processed_bids_producer = KafkaProducer()

        # ofertele se pun in dictionar, sub forma de perechi
<IDENTITATE_OFERTANT, MESAJ_OFERTA>
        self.bids = dict()

    def get_and_process_messages(self):
        # se asteapta notificarea de la Auctioneer pentru incheierea
licitatiei
        print("Astept notificare de la toate entitatile Auctioneer
pentru incheierea licitatiei...")
```

```

        auction_end_message =
next(self.notify_message_processor_consumer)

        # a ajuns prima notificare, se asteapta si celelalte
notificari timp de maxim 15 secunde

self.notify_message_processor_consumer.config["consumer_timeout_ms"] =
15_000
        for auction_end_message in
self.notify_message_processor_consumer:
            pass
        self.notify_message_processor_consumer.close()

        if str(auction_end_message.value, encoding="utf-8") ==
"incheiat":
            # se preiau toate ofertele din topicul bids_topic si se
proceseaza
            print("Licitatie incheiata. Procesez mesajele cu
oferte...")

            for msg in self.bids_consumer:
                for header in msg.headers:
                    if header[0] == "identity":
                        identity = str(header[1], encoding="utf-8")

                    # eliminare duplicate
                    if identity not in self.bids.keys():
                        self.bids[identity] = msg

            self.bids_consumer.close()

            # sortare dupa timestamp
            sorted_bids = sorted(self.bids.values(), key=lambda bid:
bid.timestamp)

            self.finish_processing(sorted_bids)

def finish_processing(self, sorted_bids):
    print("Procesarea s-a incheiat! Trimit urmatoarele oferte:")
    for bid in sorted_bids:
        for header in bid.headers:
            if header[0] == "identity":
                identity = str(header[1], encoding="utf-8")
            elif header[0] == "amount":
                bid_amount = int.from_bytes(header[1], 'big')
        print("[{}]\t{} a licitat
{}.".format(datetime.fromtimestamp(bid.timestamp / 1000), identity,
bid_amount))

        # se stocheaza mesajele ordonate dupa timestamp si fara
duplicate intr-un topic separat

self.processed_bids_producer.send(topic=self.processed_bids_topic,
value=bid.value, headers=bid.headers)

        self.processed_bids_producer.flush()

```

```

        self.processed_bids_producer.close()

    def run(self):
        self.get_and_process_messages()

if __name__ == '__main__':
    message_processor = MessageProcessor(
        bids_topic="topic_oferte",

notify_message_processor_topic="topic_notificare_procesor_mesaje",
        processed_bids_topic="topic_oferte_procesate"
    )
    message_processor.run()

```

MessageProcessor folosește 3 *topic*-uri Kafka:

- **topic_oferte** → de aici își preia mesajele pentru procesare, după ce a fost notificat că licitația a fost încheiată
- **topic_notificare_procesor_mesaje** → așteaptă un mesaj trimis de **Auctioneer**, cu textul „încheiat” pentru a sătă că nu mai sunt entități **Bidder** care vor mai trimite oferte, și poate începe deci procesarea mesajelor din *topic*-ul **topic_oferte**.
- **topic_oferte_procesate** → în acest *topic* sunt produse mesajele ce conțin ofertele finale, ordonate după data și oră, și fără duplicate. **Kafka asigură primirea în ordine a mesajelor din aceeași partitură a unui topic**. Deoarece **topic_oferte_procesate** este creat cu o singură partitură, este asigurată astfel ordonarea mesajelor obținute după procesare.

• BiddingProcessor.py

```

from kafka import KafkaConsumer, KafkaProducer

class BiddingProcessor:
    def __init__(self, processed_bids_topic, result_topic):
        super().__init__()
        self.processed_bids_topic = processed_bids_topic
        self.result_topic = result_topic

        # consumatorul pentru ofertele procesate
        self.processed_bids_consumer = KafkaConsumer(
            self.processed_bids_topic,
            auto_offset_reset="earliest", # mesajele se preiau de la
cel mai vechi la cel mai recent
            consumer_timeout_ms=3000
        )

        # producatorul pentru trimiterea rezultatului licitatiei
        self.result_producer = KafkaProducer()

    def get_processed_bids(self):
        # se preiau toate ofertele procesate din topicul
processed_bids_topic
        print("Aștept ofertele procesate de MessageProcessor...")
```

```

# ofertele se stocheaza sub forma de perechi <PRET_LICITAT,
MESAJ_OFERTA>
    bids = dict()
    no_bids_available = True

    while no_bids_available:
        for msg in self.processed_bids_consumer:
            for header in msg.headers:
                if header[0] == "amount":
                    bid_amount = int.from_bytes(header[1], 'big')
                    bids[bid_amount] = msg

    # daca inca nu exista oferte, se asteapta in continuare
    if len(bids) != 0:
        no_bids_available = False

    self.processed_bids_consumer.close()
    self.decide_auction_winner(bids)

def decide_auction_winner(self, bids):
    print("Procesez ofertele...")

    if len(bids) == 0:
        print("Nu exista nicio oferta de procesat.")
        return

    # sortare dupa oferte, descrescator
    sorted_bids = sorted(bids.keys(), reverse=True)

    # castigatorul este ofertantul care a oferit pretul cel mai
mare
    winner = bids[sorted_bids[0]]

    for header in winner.headers:
        if header[0] == "identity":
            winner_identity = str(header[1], encoding="utf-8")

    print("Castigatorul este:")
    print("\t{} - pret licitat: {}".format(winner_identity,
sorted_bids[0]))

    # se trimit rezultatul licitatiei pentru ca entitatile Bidder
sa il preia din topicul corespunzator
    self.result_producer.send(topic=self.result_topic,
value=winner.value, headers=winner.headers)
    self.result_producer.flush()
    self.result_producer.close()

def run(self):
    self.get_processed_bids()

if __name__ == '__main__':
    bidding_processor = BiddingProcessor(
        processed_bids_topic="topic_oferte_procesate",

```

```
        result_topic="topic_rezultat"
    )
bidding_processor.run()
```

BiddingProcessor folosește 2 *topic*-uri Kafka, astfel:

- **topic_oferte_procesate** → se așteaptă primirea ofertelor procesate de la **MessageProcessor**, iar pe măsură ce acestea sunt disponibile în topic, sunt preluate și adăugate într-o colecție internă.
- **topic_rezultat** → după decizia câștigătorului, mesajul efectiv trimis de acesta este publicat în **topic_rezultat**, pentru ca fiecare **Bidder** să îl preia de acolo și să afle cine a câștigat licitația.

Încapsularea microserviciilor în imagini Docker

Se oferă un exemplu de încapsulare a microserviciului **Auctioneer** într-o imagine Docker, pe baza căreia se poate porni un număr variabil de containere.

Creați un folder denumit, spre exemplu, **Auctioneer_Docker**, în care adăugați fișierul sursă al microserviciului **Auctioneer** (**Auctioneer.py**).

Alături de acel fișier sursă, creați un fișier **Dockerfile**, în care adăugați următorul conținut:

```
FROM python:alpine

# instalare modul kafka-python
RUN pip install kafka-python

# adaugare fisier sursa pentru microserviciul Auctioneer
WORKDIR /auctioneer
ADD Auctioneer.py $WORKDIR

# comanda de executie este: python <nume_fisier>.py
CMD ["python", "Auctioneer.py"]
```

Fișierul Dockerfile va fi folosit pentru construirea unei imagini Docker care încapsulează microserviciul **Auctioneer**. Nu este nevoie să o construiți imediat, deoarece acest lucru va fi făcut automat de Docker Compose (urmează mai jos).

Execuția aplicației

Se recomandă execuția de la terminal, așa încât deschideți unul în folder-ul proiectului și creați un mediu virtual Python în care instalați modulul **kafka-python**:

```
python3 -m venv okazii_env
source ./okazii_env/bin/activate
pip3 install kafka-python
```

Nu uitați să verificați că server-ul Kafka și containerul Docker cu ETCD sunt pornite!

Pregătirea topic-urilor

Înainte de fiecare pornire a aplicației, folosiți următorul script Python pentru a pregăti *topic*-urile Kafka pentru pornirea licitației:

- [PrepareAuction.py](#)

```
from kafka import KafkaAdminClient
from kafka.admin import NewTopic
import time

if __name__ == '__main__':
    admin = KafkaAdminClient()

    used_topics = (
        "topic_oferte",
        "topic_rezultat",
        "topic_oferte_procesate",
        "topic_notificare_procesor_mesaje",
    )

    # se sterg topic-urile, daca exista deja
    print("Se sterg topic-urile existente...")

    kafka_topics = admin.list_topics()
    for topic in kafka_topics:
        if topic in used_topics:
            print("\tSe sterge {}...".format(topic))
            admin.delete_topics(topics=[topic], timeout_ms=2000)

            # se asteapta putin ca stergerea sa aiba loc
            time.sleep(2)

    # se creeaza topic-urile necesare aplicatiei
    print("Se creeaza topic-urile necesare:")
    lista_topicuri = [
        NewTopic(name=used_topics[0], num_partitions=4,
replication_factor=1),
        NewTopic(name=used_topics[1], num_partitions=1,
replication_factor=1),
        NewTopic(name=used_topics[2], num_partitions=1,
replication_factor=1),
        NewTopic(name=used_topics[3], num_partitions=1,
replication_factor=1)
    ]
    for topic in lista_topicuri:
        print("\t{}".format(topic.name))
    admin.create_topics(lista_topicuri, timeout_ms=3000)

    print("Gata! Microserviciile participante la licitatie pot fi
pornite.")
```

Atenție: înainte de a-l executa, asigurați-vă că nu există niciun consumator Kafka pornit (niciun microserviciu în aşteptare de mesaje, niciun script ce foloseşte API-ul Kafka pentru consumarea mesajelor din *topic*-urile folosite de aplicaţie).

Script-ul va şterge *topic*-urile existente pentru a forţa golirea mesajelor existente de la execuţii anterioare ale aplicaţiei. Apoi, *topic*-urile vor fi recreate cu parametrii necesari.

```
/home/cosmin/venv/Licitatie/bin/python "/mnt/hdd/PREDARE/SD/laboratoare/laborator 10/Okazii_v2/PrepareAuction.py"
Se sterg topic-urile existente...
  Se sterge topic_oferte_procesate...
  Se sterge topic_oferte...
  Se sterge topic_notificare_procesor_mesaje...
  Se sterge topic_rezultat...
Se creeaza topicurile necesare:
  topic_oferte
  topic_rezultat
  topic_oferte_procesate
  topic_notificare_procesor_mesaje
Gata! Microserviciile participante la licitatie pot fi pornite.
```

Figura 13 - Pregătire *topic*-uri Kafka

Dacă primiți o eroare cum că există deja un anumit *topic*: măriți timpul de așteptare pentru ștergere (parametrul `time.sleep(...)`). Verificați să nu fie pornit niciun consumator de mesaje din acele *topic*-uri conținute în tupla `used_topics`.

Dacă nu rulați script-ul înainte de aplicatie, microserviciile vor recita mesajele anterioare, deoarece sunt persistente până la expirare. Puteti profita de această proprietate a Kafka pentru testarea independentă, izolată a microserviciilor: de exemplu, dacă porniți doar **MessageProcessor** și există mesaje din execuții anterioare, acesta va recita ofertele existente în Kafka și le va procesa din nou, trimițând un alt set de mesaje procesate pentru **BiddingProcessor**.

Pornirea microserviciilor

Deoarece prin varianta de implementare cu Kafka s-a obținut un grad mare de decuplare, iar fiecare microserviciu așteaptă mesajele necesare lui pentru a funcționa, **microserviciile se pot porni în orice ordine se dorește**, cu mențiunea că **fiecare instanță a microserviciului Auctioneer așteaptă maxim 15 secunde de inactivitate din partea Bidder-ilor**.

Deci, se recomandă să porniți **Auctioneer** ca ultimul microserviciu.

Pornirea microserviciilor care nu sunt încapsulate în imagini Docker

Deschideți câte un terminal în folder-ul cu fișierele sursă ale proiectului, și porniți fiecare microserviciu (**în afara de Auctioneer**) astfel:

```
source CALE_CĂTRE_VENV/bin/activate
python3 NUME_MICROSERVICIU.py
```

Pornirea Auctioneer cu factor de scalare parametrizat

Microserviciul **Auctioneer** se pornește folosind **Docker Compose**, pentru a putea fi scalat cu ușurință: astfel, se pot porni oricâte instanțe **Auctioneer** dorește utilizatorul.

Creați un fișier numit **docker-compose.yml** în folder-ul **Auctioneer_Docker** creat anterior, și adăugați următorul conținut:

```
version: "3.4"
services:
  auctioneer:
    build:
      context: .
    network_mode: host
```

Deci, pentru a porni **2 instanțe** ale microserviciului **Auctioneer**, executați următoarea comandă în folder-ul **Auctioneer_Docker**, în care se regăsește fișierul **docker-**

compose.yml:

```
docker-compose up --scale auctioneer=2
```

Un exemplu de execuție se regăsește în figura următoare:

```
(okazii_env) cosmin@debian-gl553v:/mnt/hdd/PREDARE/SD/laborator/10/Okazii_v2$ python3 MessageProcessor.py
Astept notificare de la toate entitatile Auctioneer pentru incheierea licitatiei...
Licitatia incheiat. Procesez mesajele cu oferte...
Procesarea s-a incheiat! Trimite urmatoarele oferte:
[2020-04-26 13:06:35.896000] Bidder db5310d1-af7e-4ade-b157-ab0732e687ed - pret licitat: 8937
[2020-04-26 13:06:37.462000] Bidder 6dd0f056-8e16-476b-87e6-b6fff9427990 a licitat 3487.
[2020-04-26 13:06:38.719000] Bidder 749be582-e556-499f-9458-00dd6261f003 a licitat 1920.
(okazii_env) cosmin@debian-gl553v:/mnt/hdd/PREDARE/SD/laboratoare/laborator 10/Okazii_v2$ 
```

MESSAGE PROCESSOR

```
(okazii_env) cosmin@debian-gl553v:/mnt/hdd/PREDARE/SD/laboratoare/laborator 10/Okazii_v2$ python3 Bidder.py
Trimite licitatia mea: 8937...
Astept rezultatul licitatiei...
[db5310d1-af7e-4ade-b157-ab0732e687ed] Am castigat!!!
(okazii_env) cosmin@debian-gl553v:/mnt/hdd/PREDARE/SD/laboratoare/laborator 10/Okazii_v2$ 
```

BIDDER 1

```
(okazii_env) cosmin@debian-gl553v:/mnt/hdd/PREDARE/SD/laboratoare/laborator 10/Okazii_v2$ python3 Bidder.py
Trimite licitatia mea: 3487...
Astept rezultatul licitatiei...
[6dd0f056-8e16-476b-87e6-b6fff9427990] Am pierdut...
(okazii_env) cosmin@debian-gl553v:/mnt/hdd/PREDARE/SD/laboratoare/laborator 10/Okazii_v2$ 
```

BIDDER 2

```
(okazii_env) cosmin@debian-gl553v:/mnt/hdd/PREDARE/SD/laboratoare/laborator 10/Okazii_v2$ python3 Bidder.py
Trimite licitatia mea: 1920...
Astept rezultatul licitatiei...
[749be582-e556-499f-9458-00dd6261f003] Am pierdut...
(okazii_env) cosmin@debian-gl553v:/mnt/hdd/PREDARE/SD/laboratoare/laborator 10/Okazii_v2$ 
```

AUCTIONEER 1 ȘI 2

BIDDER 3

Figura 14 - Exemplu de execuție

Puteți observa împărțirea mesajelor pe partiții în cazul entităților **Auctioneer** și, de asemenea, existența unui mesaj duplicat care a fost filtrat de **MessageProcessor**.

Utilizarea Kafka pentru monitorizarea aplicațiilor în dev-ops

Kafka poate fi utilizat și pentru statistici și monitorizare, de exemplu în sensul de colectare a informațiilor din *topic*-urile utilizate de aplicația bazată pe microservicii: numărul de mesaje primite, numărul de mesaje procesate etc. - toate acestea pot oferi informații legate de **gradul de încărcare al fluxului de mesaje**.

De asemenea, se pot prelua și metrii oferite de Docker Engine în scopul monitorizării gradului de încărcare general al aplicației în timp real: **utilizare CPU**, **utilizare memorie** etc.

În scopul monitorizării aplicației **Okazii** din laborator, creați o aplicație separată Spring Boot (numită, de exemplu, **MonitoringApplication**), la care adăugați dependența **Spring-Kafka** (ca în exemplul 2 din laborator). Această aplicație reprezintă un microserviciu utilizat pentru instrumentație și monitorizare.

• KafkaMonitor.kt

```
package com.sd.laborator

import org.apache.kafka.clients.consumer.ConsumerRecord
import org.springframework.kafka.annotation.KafkaListener
```

```

import org.springframework.kafka.annotation.PartitionOffset
import org.springframework.kafka.annotation.TopicPartition
import org.springframework.scheduling.annotation.Scheduled
import org.springframework.stereotype.Component
import java.io.BufferedReader
import java.io.InputStreamReader
import java.net.URLEncoder
import java.time.LocalDateTime

@Component
class KafkaMonitor {
    companion object {
        private var number_of_bids: Int = 0
        private var number_of_processed_bids: Int = 0

        private var BASE_DOCKER_API_COMMAND = "curl --unix-socket  

/var/run/docker.sock http:/v1.40"
    }

    init {
        println("KafkaMonitor instantiated!")
    }

    @KafkaListener(
        topicPartitions = [
            TopicPartition(
                topic = "topic_oferte",
                partitionOffsets = [
                    PartitionOffset(partition = "0", initialOffset =
"0"),
                    PartitionOffset(partition = "1", initialOffset =
"0"),
                    PartitionOffset(partition = "2", initialOffset =
"0"),
                    PartitionOffset(partition = "3", initialOffset =
"0")
                ]
            ),
            TopicPartition(
                topic = "topic_oferte_procesate",
                partitionOffsets = [
                    PartitionOffset(partition = "0", initialOffset =
"0")
                ]
            )
        ]
    )
    fun monitorKafkaMessages(message: ConsumerRecord<String, String>)
    {
        when(message.topic()) {
            "topic_oferte" -> ++number_of_bids
            "topic_oferte_procesate" -> ++number_of_processed_bids
        }
    }
}

@Scheduled(fixedDelay=1000)

```

```

    fun showKafkaStats() {
        println("[${LocalDateTime.now()}] Grad incarcare Auctioneer:
$numberOfBids oferte primite")
        println("[${LocalDateTime.now()}] Grad incarcare
MessageProcessor: $numberOfProcessedBids oferte procesate")
    }

@Scheduled(fixedDelay=2000)
fun monitorAuctioneerContainer() {
    // preluare lista de instante ale Auctioneer
    val auctioneerContainersFilter =
URLEncoder.encode("{\"ancestor\":
[\"auctioneer_docker_auctioneer:latest\"]}"), "utf-8")
    val auctioneerListProcess: Process =
Runtime.getRuntime().exec("$BASE_DOCKER_API_COMMAND/containers/json?fi-
lters=$auctioneerContainersFilter")
    val auctioneerListProcessInput =
BufferedReader(InputStreamReader(auctioneerListProcess.inputStream))

    // se citeste raspunsul de la Docker Engine API sub forma de
JSON
    val auctioneerListOutput =
auctioneerListProcessInput.readLine()
    auctioneerListProcessInput.close()

    // se preia lista de ID-uri ale container-elor de tip
Auctioneer
    val containerIdRegex = Regex("\"Id\":\"([a-f0-9]*)\"")
    containerIdRegex.findAll(auctioneerListOutput).forEach {
        // pentru fiecare ID in parte se preiau statisticile la
runtime
        val auctioneerContainerID = it.groupValues[1].take(12)

        // se foloseste ruta /containers/ID/stats de la API-ul
Docker Engine
        val auctioneerContainerStatsProcess: Process =
Runtime.getRuntime().exec("$BASE_DOCKER_API_COMMAND/containers/$auc-
tioneerContainerID/stats?stream=0")
        val auctioneerContainerStatsProcessInput =
BufferedReader(InputStreamReader(auctioneerContainerStatsProcess.in-
putStream))

        // din output-ul cu statistici, se colecteaza utilizarea
CPU si a memoriei
        val auctioneerContainerStatsOutput =
auctioneerContainerStatsProcessInput.readLine()

        val cpuUsageRegex =
Regex("\"cpu_stats\":\\{\"cpu_usage\":\\{\"total_usage\":[0-9]*,\"")
cpuUsageRegex.find(auctioneerContainerStatsOutput)?.groupValues?.get(1)
?.let {
            println("[\$auctioneerContainerID] Utilizare procesor:
\$it")
}
    }
}

```

```
        }
    }

    val memoryUsageRegex =
Regex("\"memory_stats\":\\{\"usage\":([0-9]*),\"")

memoryUsageRegex.find(auctioneerContainerStatsOutput)?.groupValues?.get(1).let {
    println("[${auctioneerContainerID}] Utilizare memorie: $it")
}
}
```

Componenta de monitorizare conține:

- un *listener* pentru Kafka, care urmărește mesajele primite în *topic*-urile **topic_oferte**, respectiv **topic_oferte_procesate**, deoarece acestea ar fi punctele de interes cu posibilitate mare de încărcare. Pentru fiecare *topic* în parte, trebuie specificate partiiile care vor fi urmărite, respectiv *offset*-ul de la care se începe citirea mesajelor (**0** = de la început). Fiind aplicație de monitorizare, toate cele 4 partii de la *topic*-ul **topic_oferte** sunt de interes. *Listener*-ul contorizează mesajele primite, în timp real.
 - o funcție recurrentă care afișează numărul de mesaje preluate de *listener*-ul Kafka: **showKafkaStats()**
 - o funcție recurrentă care monitorizează instanțele microserviciului **Auctioneer** sub formă de containere Docker (indiferent de numărul lor, în mod dinamic): **showAuctioneerContainersStats()**. Funcția utilizează API-ul Docker Engine (<https://docs.docker.com/engine/api/v1.40>) pentru a trimite cereri către procesul *daemon* al Docker prin socket-ul dedicat:
 - **/containers/json** → afișează informații despre containerele în execuție. Se pot filtra rezultatele cu parametrul **filters**.
 - **/containers/<ID_CONTAINER>/stats** → afișează metrii despre un anumit container. Pentru a primi doar ultimele valori (și nu un flux continuu), se folosește parametrul URL **stream=0**.

Metodele annoteate cu `@Scheduled` se execută o dată la perioada de timp specificată ca parametru (în milisecunde).

- **MonitoringApplication.kt**

```
package com.sd.laborator

import org.springframework.boot.autoconfigure.SpringBootApplication
import org.springframework.boot.runApplication
import org.springframework.scheduling.annotation.EnableScheduling

@SpringBootApplication
@EnableScheduling
class MonitoringApplication

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

S-a folosit adnotarea `@EnableScheduling` pentru a activa posibilitatea de a folosi

funcțiile recurente din Spring (adnotate cu `@Scheduler`).

Utilizați aplicația de monitorizare pentru a prelua metrii în timp real din aplicația Okazii. Porniți **MonitoringApplication** după ce a început licitația, și veți putea găsi metricile afișate în *log-urile* Spring:

```
[2020-04-26T23:14:55.157] Grad incarcare Auctioneer: 7 oferte primite
[2020-04-26T23:14:55.158] Grad incarcare MessageProcessor: 5 oferte procesate
[2020-04-26T23:14:56.158] Grad incarcare Auctioneer: 7 oferte primite
[2020-04-26T23:14:56.158] Grad incarcare MessageProcessor: 5 oferte procesate
[2020-04-26T23:14:57.159] Grad incarcare Auctioneer: 7 oferte primite
[2020-04-26T23:14:57.159] Grad incarcare MessageProcessor: 5 oferte procesate
[0e6372067795] Utilizare procesor: 181941570
[0e6372067795] Utilizare memorie: 11595776
[87d323d07ae1] Utilizare procesor: 262941315
[87d323d07ae1] Utilizare memorie: 11649024
[78f06d1293e5] Utilizare procesor: 242057484
[78f06d1293e5] Utilizare memorie: 11657216
[2020-04-26T23:15:02.366] Grad incarcare Auctioneer: 7 oferte primite
[2020-04-26T23:15:02.366] Grad incarcare MessageProcessor: 5 oferte procesate
[2020-04-26T23:15:03.366] Grad incarcare Auctioneer: 7 oferte primite
[2020-04-26T23:15:03.367] Grad incarcare MessageProcessor: 5 oferte procesate
[0e6372067795] Utilizare procesor: 272503533
[0e6372067795] Utilizare memorie: 11907072
[87d323d07ae1] Utilizare procesor: 351367441
[87d323d07ae1] Utilizare memorie: 11677696
[78f06d1293e5] Utilizare procesor: 330037046
[78f06d1293e5] Utilizare memorie: 11698176
[2020-04-26T23:15:09.439] Grad incarcare Auctioneer: 7 oferte primite
[2020-04-26T23:15:09.439] Grad incarcare MessageProcessor: 11 oferte procesate
[2020-04-26T23:15:10.440] Grad incarcare Auctioneer: 7 oferte primite
[2020-04-26T23:15:10.440] Grad incarcare MessageProcessor: 11 oferte procesate
```

Figura 15 - Monitorizarea aplicației Okazii

Valorile pentru utilizarea procesorului și a memoriei sunt de fapt valori brute raportate de sistemul de fișiere virtual **/proc/stats**. Pentru a calcula efectiv procentele de utilizare, se poate face un calcul suplimentar în maniera explicată aici:

https://rosettacode.org/wiki/Linux_CPU_utilization

Aplicații și teme

Teme de laborator

- Încapsulați și microserviciul **Bidder** într-o imagine Docker. Creați Dockerfile pentru acesta și folosiți comanda **docker build** (exemplu de folosire în **laboratorul 8**) pentru construirea imaginii.
- Utilizând **Docker Compose**, porniți un număr mare de astfel de microservicii (de exemplu, 100) la o execuție a aplicației. Scalați microserviciul **Auctioneer** la un număr de 4 instanțe și verificați că aplicația face față fluxului mare de mesaje.

Teme pentru acasă

- Completați aplicația de monitorizare a resurselor astfel încât să afișeze un grafic în timp real pentru resursele monitorizate.

Puteți folosi, eventual, biblioteca **Plotly**:

- <https://github.com/mipt-npm/plotly.kt>
- <https://medium.com/@altavir/plotly-kt-a-dynamic-approach-to-visualization-in-kotlin-38e4feaf61f7>

2. Implementați licitația engleză (eng. *English auction*), pornind de la exemplul de aplicație din laborator (**Okazii v2**). Adăugați un nou procesor de flux care să implementeze logica necesară pentru regulile licitației engleze și integrați acest procesor în fluxul existent încât utilizatorul să poată să aleagă ce tip de licitație dorește: tipul inițial de licitație, cu plic închis, sau tipul nou implementat, de tip *English auction*.

Găsiți regulile licitației engleze aici:

<https://corporatefinanceinstitute.com/resources/knowledge/finance/english-auction/>

Bibliografie

- [1]: Neha Narkhede, Kafka: The Definitive Guide - Real-Time Data and Stream Processing at Scale
- [2]: Introducere în Apache Kafka - <https://kafka.apache.org/intro>
- [3]: Ghid rapid pentru instalare și configurare Kafka - <https://kafka.apache.org/quickstart>
- [4]: ETCD - <https://etcd.io/>
- [5]: Biblioteca Kafka-Python - <https://github.com/dpkp/kafka-python>
- [6]: API-ul Kafka-Python - <https://kafka-python.readthedocs.io/en/master/apidoc/modules.html>
- [7]: Proiectul Spring Kafka - <https://spring.io/projects/spring-kafka>
- [8]: Documentație Spring Kafka (**codul din documentație este scris în Java - se poate transforma / adapta pentru limbajul Kotlin**) - <https://docs.spring.io/spring-kafka/docs/2.4.5.RELEASE/reference/html/>
- [9] Introducere în Docker Compose - <https://docs.docker.com/compose/gettingstarted/>
- [10] Docker Compose - formatul de fișier YAML pentru configurare - <https://docs.docker.com/compose/compose-file/>
- [11] Scalarea containerelor folosind Docker Compose - <https://docs.docker.com/compose/reference/scale/>
- [12] Monitorizarea containerelor Docker - comanda **docker stats** - <https://docs.docker.com/engine/reference/commandline/stats/>
- [13] API-ul Docker Engine - <https://docs.docker.com/engine/api/v1.40/>
- [14] Exemple de utilizare ale API-ului Docker Engine - <https://docs.docker.com/engine/api/sdk/examples/>

Sisteme Distribuite - Laborator 11

Micronaut - *serverless computing*

Micronaut - descriere generală

Micronaut este un *framework* modern folosit pentru crearea de microservicii, proiectat pentru construirea de aplicații modulare și testabile cu ușurință. Întâia acestui *framework* este furnizarea tuturor uneltelelor necesare pentru aplicațiile bazate pe microservicii, precum:

- injectarea dependențelor la compilare (eng. *compile-time dependency injection*) și inversarea controlului (eng. *Inversion of Control*)
- configurare automată
- partajarea configurațiilor
- descoperirea de servicii (eng. *service discovery*)
- rutare HTTP
- client HTTP cu echilibrarea încărcării la client

Față de alte *framework*-uri precum Spring / Spring Boot sau Grails, Micronaut încearcă să rezolve unele probleme apărute în tehnologiile menționate, oferind următoarele avantaje:

- pornirea rapidă a aplicațiilor
- grad mai mic de utilizare a memoriei
- grad mic de utilizare a reflecției (eng. *reflection*)
- grad mic de utilizare a proxy-urilor
- dimensiune mică a artefactelor JAR rezultate
- testare unitară facilă

Avantajele menționate ale Micronaut s-au obținut prin utilizarea **procesoarelor de adnotări Java** (eng. *annotation processors*), care se pot folosi în orice limbaj de programare ce țintește mașina virtuală Java, și care suportă acest concept. Aceste procesoare de adnotare precompilează metadatele necesare pentru a asigura injectarea dependențelor, definirea proxy-urilor AOP și configuraerea aplicației în scopul execuției acesteia într-un mediu al microserviciilor.

Instalarea Micronaut

Se descarcă arhiva de pe site-ul oficial (<https://micronaut.io/download.html>), astfel:

```
wget https://github.com/micronaut-projects/micronaut-core/releases/download/v1.3.4/micronaut-1.3.4.zip
```

(ultima versiune stabilă disponibilă la momentul scrierii laboratorului este **1.3.4**)

Dezarhivați arhiva descărcată utilizând arhivatorul grafic sau comanda următoare:

```
unzip micronaut-1.3.4.zip
```

Mutați server-ul Micronaut în folder-ul standard cu software optional:

```
sudo mv micronaut-1.3.4 /opt
```

Adăugați binarul **mn** la **PATH**-ul sistemului:

```
export PATH="$PATH:/opt/micronaut-1.3.4/bin"
```

Atenție: comanda de mai sus este valabilă pentru sesiunea de terminal curentă!

Dacă deschideți un terminal nou și vreți să lucrați cu Micronaut, trebuie să rulați comanda anterioară din nou. Alternativ, adăugați comanda de mai sus la sfârșitul fișierului `$HOME/.bashrc`.

Testați server-ul Micronaut utilizând comanda următoare:

```
mn --version
```

Exemple de tipuri de aplicații folosind Micronaut CLI

Deși modul de creare a aplicațiilor Micronaut din CLI nu este neapărat necesar (proiectul poate fi creat manual și scheletul scris apoi), se poate folosi linia de comandă a server-ului în acest scop, pentru a simplifica inițializarea unui proiect de tip Micronaut.

Crearea unui proiect Micronaut - limbaj Kotlin, gestionar de proiect Maven

```
mn create-app com.sd.laborator.exemplu-micronaut-maven --lang kotlin --build maven
```

Crearea unui proiect Micronaut - limbaj Kotlin, gestionar de proiect Gradle

```
mn create-app com.sd.laborator.exemplu-micronaut-gradle --lang kotlin --build gradle
```

După ce ați creat un proiect Micronaut folosind **unul din cele 2 tipuri de gestionare de proiect**, puteți importa proiectul în IntelliJ: **Open** (sau „**Open or Import**”) → selectați folder-ul generat de comanda de creare a proiectului și apăsați **OK**.

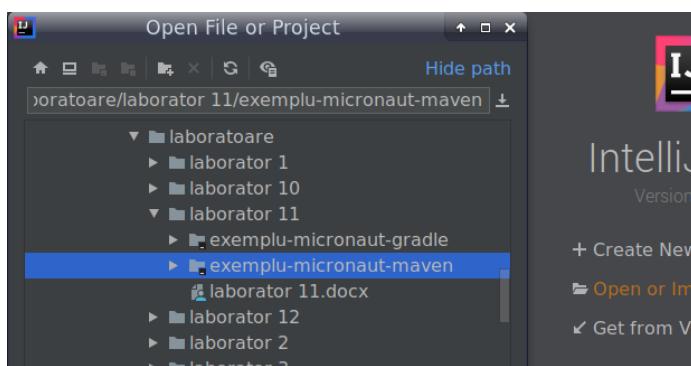


Figura 1 - Importare proiect Micronaut

Așteptați până când IntelliJ rezolvă toate dependențele necesare și până generează toată structura proiectului.

Activăți procesarea adnotărilor din IntelliJ, astfel: **File** → **Settings** → **Build, Execution, Deployment** → **Compiler** → **Annotation Processors** → bifăți **Enable annotation processing**.

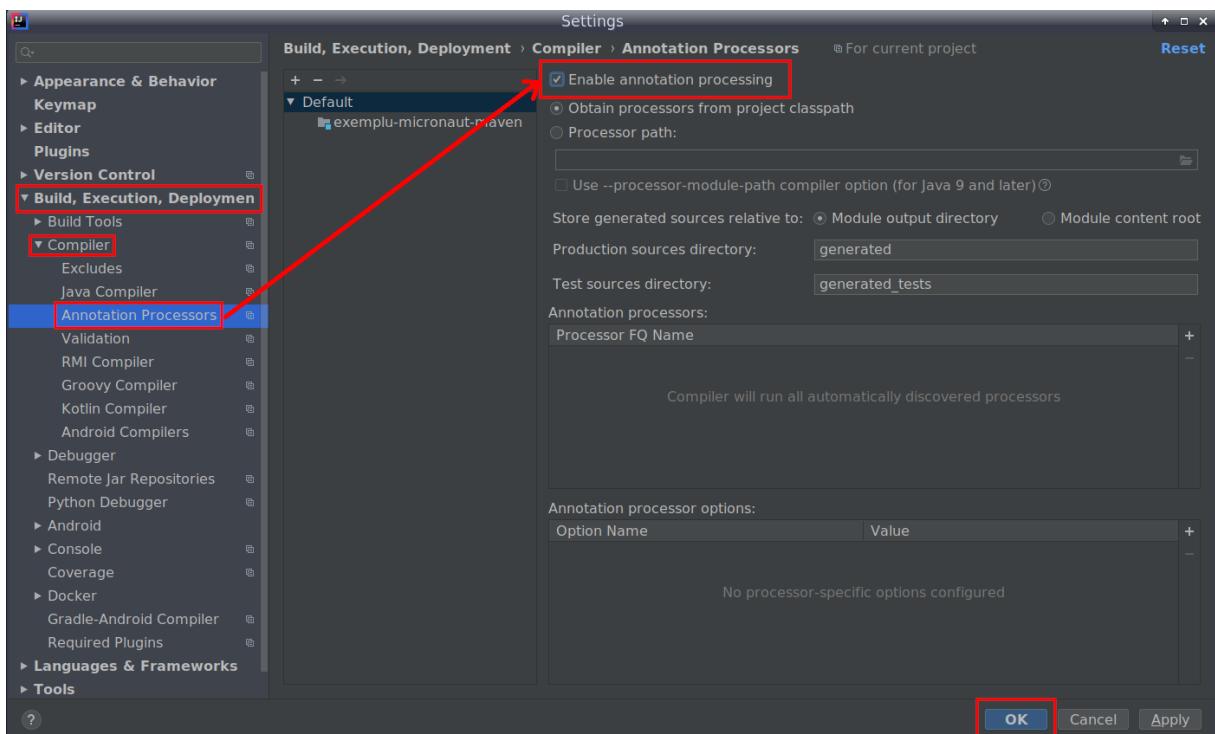


Figura 2 - Activarea procesării adnotărilor în IntelliJ

Codul din fișierul sursă **Application.kt** generat automat arată astfel:

```
package com.sd.laborator

import io.micronaut.runtime.Micronaut

object Application {

    @JvmStatic
    fun main(args: Array<String>) {
        Micronaut.build()
            .packages("com.sd.laborator")
            .mainClass(Application.javaClass)
            .start()
    }
}
```

Adnotarea **@JvmStatic** aplicată asupra funcției **main()** determină generarea unei metode statice adiționale pe baza funcției existente aflate sub influența adnotării. Apelul **start()** asupra *builder*-ului de *runtime* Micronaut reprezintă punctul de intrare al aplicației.

Crearea unui controller folosind Micronaut CLI

O componentă *controller* poate fi creată în mod facil tot folosind linia de comandă Micronaut. Executați comanda următoare **în folder-ul proiectului**:

```
mn create-controller HelloController
```

Comanda va modifica proiectul Micronaut existent adăugând o componentă de tip *controller*. Modificați codul *controller*-ului (clasa generată **HelloController**) astfel:

```
package com.sd.laborator

import io.micronaut.http.annotation.Controller
```

```

import io.micronaut.http.annotation.Get
import io.micronaut.http.MediaType
import io.micronaut.http.annotation.Produces

@Controller("/hello")
class HelloController {
    @Produces(MediaType.TEXT_PLAIN)
    @Get("/")
    fun hello(): String {
        return "Hello from Micronaut!"
    }
}

```

Controller-ul aplicației Micronaut este asemănător cu un *controller* Spring: clasa *controller* se adnotează cu `@Controller`, iar parametrul adnotării reprezintă calea de bază pentru toate metodele mapate. În acest exemplu simplu, metoda `hello()` este executată la o cerere HTTP de tip GET (adnotarea `@Get`) către calea „/” relativ la „/hello”.

Adnotarea `@Produces(MediaType.TEXT_PLAIN)` indică tipul de răspuns returnat clientului. Deoarece JSON este tipul implicit de răspuns iar sirul de caractere returnat este nestructurat, se marchează explicit tipul „text plain”.

Execuția aplicației Micronaut din IntelliJ

Execuția cu Maven

Folosiți *goal*-ul `exec` din *plugin*-ul `exec` pentru a porni aplicația Micronaut cu schelet Maven:

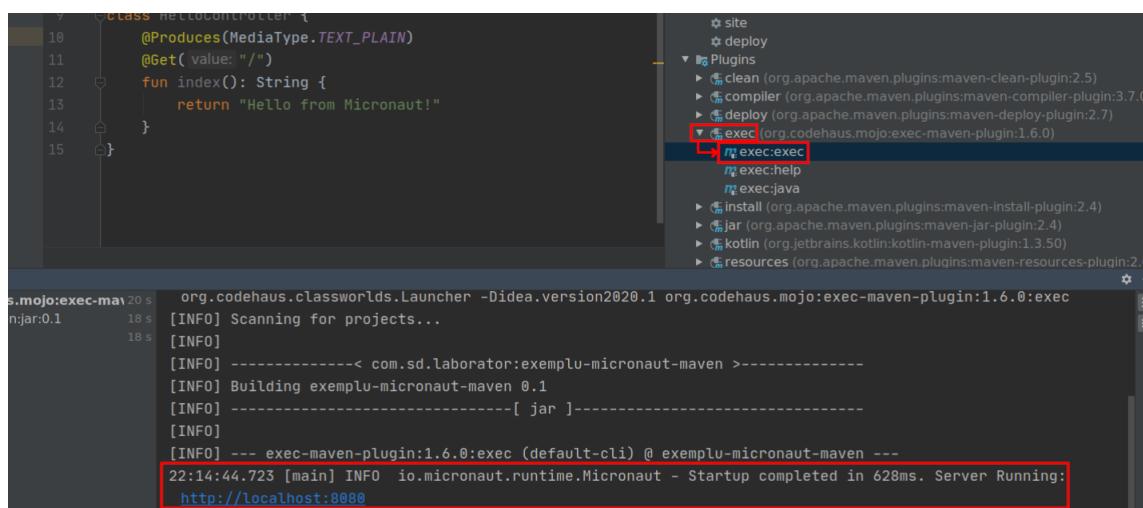


Figura 3 - Execuția unui proiect Micronaut cu schelet Maven

Execuția cu Gradle

Folosiți *task*-ul `run` din secțiunea `application` pentru a porni aplicația Micronaut cu schelet Gradle:

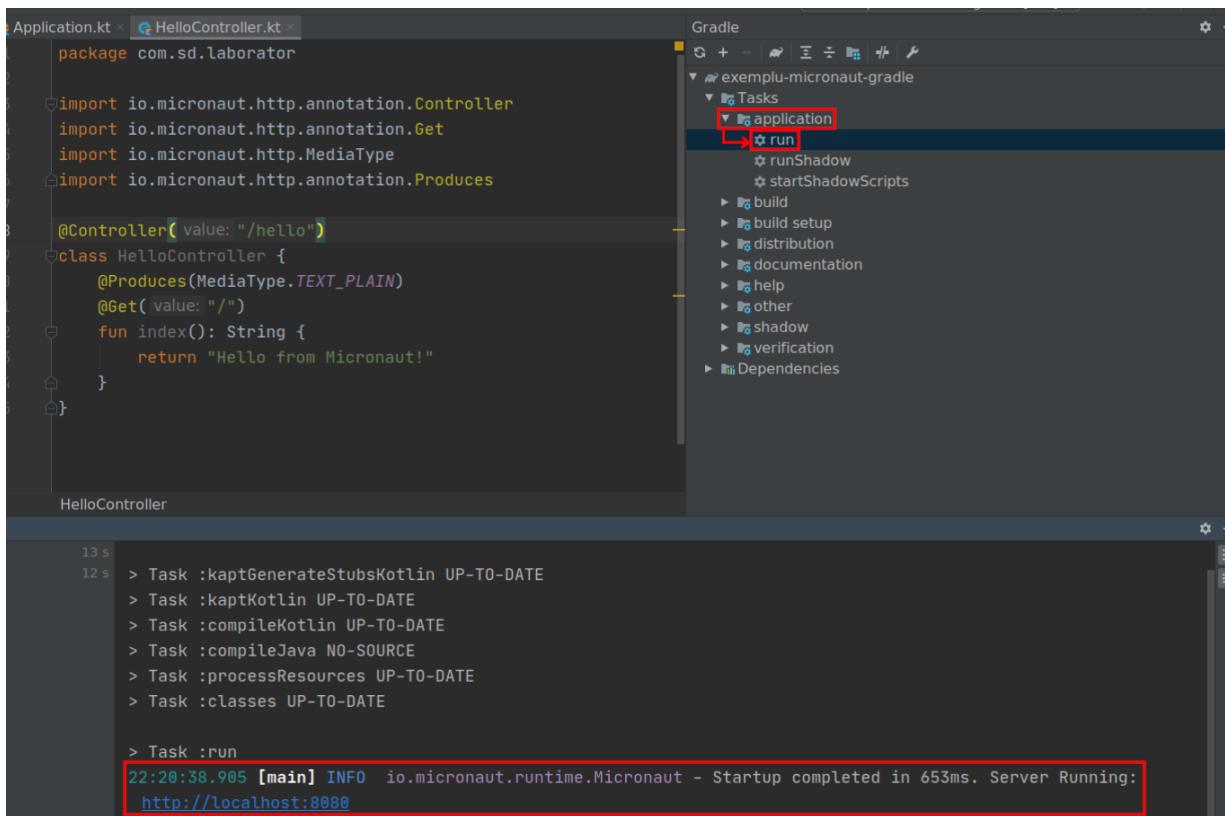


Figura 4 - Execuția unui proiect Micronaut cu schelet Gradle

Testarea aplicației Micronaut

Controller-ul răspunde pe calea /hello/ al server-ului HTTP incorporat, bazat pe Netty. Așadar, trimiteți o cerere de tip GET astfel:

```
curl -X GET http://localhost:8080/hello/
```

Funcții serverless

Funcțiile *serverless* sunt gestionate de o infrastructură *cloud* și sunt executate în procese efemere - codul este rulat de obicei în containere fără stare și poate fi declanșat de evenimente precum: cereri HTTP, alerte, evenimente recurente, încărcări de fișiere etc.

Aceste funcții sunt invocate prin Internet și sunt găzduite și menținute de companii de *cloud computing*. Furnizorul de *cloud* este responsabil pentru execuția codului încapsulat în funcții, alocând în mod dinamic resursele necesare pentru acestea. Acest model se mai numește și „*Function as a Service*”.

Crearea unei funcții serverless cu Micronaut CLI

Funcția *serverless* Micronaut este de fapt un nou tip de aplicație și se creează ca și **proiect separat**. Așadar, în afara oricărui proiect Micronaut, executați următoarea comandă, în funcție de gestionarul de proiect dorit:

- creare funcție serverless cu schelet Maven:

```
mn create-function com.sd.laborator.hello-world-maven --lang kotlin --build maven
```

- creare funcție serverless cu schelet Gradle:

```
mn create-function com.sd.laborator.hello-world-gradle --lang kotlin --
```

```
-build gradle
```

Proiectul generat se deschide în IntelliJ în aceeași manieră explicată anterior.

Dacă ati ales gestionarul de proiect Maven: adăugați următoarea dependență suplimentară în **pom.xml**:

```
<dependency>
    <groupId>org.apache.logging.log4j</groupId>
    <artifactId>log4j-api</artifactId>
    <version>2.13.2</version>
    <scope>runtime</scope>
</dependency>
```

Scheletul de cod generat conține următoarele clase:

- **Application** - identic ca cel de la exemplul anterior
- **HelloWorld Maven / HelloWorld Gradle**

```
package com.sd.laborator

import io.micronaut.core.annotation.*

@Introspected
class HelloWorldGradle { // sau HelloWorld Maven
    lateinit var name: String
}
```

Adnotarea **@Introspected** indică faptul că acea clasă peste care este aplicată va produce un tip de dată **BeanIntrospection** în momentul compilării. Acesta este rezultatul unei procesări făcute la compilare, procesare ce include proprietăți și metadate: prin tipul de dată **BeanIntrospection**, se pot instanția *bean*-uri și scrie / citi proprietăți din acestea fără a folosi reflecția.

- **HelloWorldGradleFunction / HelloWorld Maven Function**

```
package com.sd.laborator;

import io.micronaut.function.executor.FunctionInitializer
import io.micronaut.function.FunctionBean;
import javax.inject.*;
import java.util.function.Function;

@FunctionBean("hello-world-gradle")
class HelloWorldGradleFunction : FunctionInitializer(),
    Function<HelloWorldGradle, HelloWorldGradle> { // sau HelloWorld Maven

    override fun apply(msg : HelloWorldGradle) : HelloWorldGradle {
        return msg
    }
}

/**
 * This main method allows running the function as a CLI application
 using: echo '{}' | java -jar function.jar

```

```
* where the argument to echo is the JSON to be parsed.  
*/  
fun main(args : Array<String>) {  
    val function = HelloWorldGradleFunction()  
    function.run(args, { context ->  
function.apply(context.get(HelloWorldGradle::class.java)) })  
}
```

Se observă adnotarea **@FunctionBean** aplicată clasei **HelloWorldGradleFunction**. Efectul este expunerea clasei respective sub formă de funcție în aplicația Micronaut. Clasa respectivă trebuie să implementeze una din interfețele de tip **Function**. În acest caz, se implementează interfața **Function** (a se vedea tabelul de mai jos). Primul parametru *template* reprezintă tipul de date primit la intrare, iar al doilea reprezintă tipul de date returnat la ieșire: **HelloWorldGradle**.

Există următoarele tipuri de funcții Micronaut:

Interfață	Descriere
Supplier	Nu acceptă niciun argument și returnează un singur rezultat
Consumer	Acceptă un singur argument și nu returnează niciun rezultat
Biconsumer	Acceptă 2 argumente și nu returnează niciun rezultat
Function	Acceptă un singur argument și returnează un singur rezultat
BiFunction	Acceptă 2 argumente și returnează un singur rezultat

Constructorul **FunctionInitializer** este utilizat pentru inițializarea unei funcții Micronaut.

Comportamentul principal al funcției *serverless* este încapsulat în metoda **apply** (specificată în interfața **Function**). Această metodă **aplică** funcția peste argumentul primit la intrare și returnează datele de ieșire. Pentru acest exemplu simplu, funcția preia parametrul de intrare și îl returnează așa cum este.

Împachetarea funcției *serverless*

Funcția *serverless* Micronaut poate fi împachetată într-un artefact JAR în mod asemănător cu proiectul de tip Spring Boot:

- **împachetare cu Maven** → folosiți *lifecycle-ul package* (la fel ca la **Spring Boot**); artefactual rezultat se află în folder-ul **target** și are denumirea proiectului și sufixul versiunii (implicit „0.1”).
- **împachetare cu Gradle** → folosiți *task-ul assemble* din secțiunea **build**; artefactual rezultat se află în folder-ul **build/libs** și are denumirea proiectului, apoi versiunea, **apoi sufixul “-all”**.

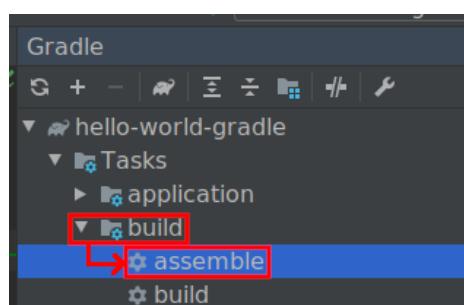


Figura 5 - Împachetarea funcției *serverless* folosind Gradle

Testarea funcției *serverless*

În mod implicit, funcția *serverless* generată de Micronaut va citi de la dispozitivul standard de intrare (*stdin*) și va scrie la dispozitivul standard de ieșire (*stdout*).

După ce ați împachetat proiectul sub formă de artefact JAR, testați funcția *serverless* trimițând valorile câmpurilor din clasa ce reprezintă parametrul de intrare sub formă de obiect JSON. Clasa **HelloWorldGradle** (sau **HelloWorldMaven**) conține un singur membru de tip string, numit „**name**”. Deci, dacă se dorește trimiterea valorii „**covid**” pentru membrul „**name**”, se codifică sub formă de obiect JSON astfel: **{"name": "covid"}**. Comanda de execuție este, aşadar:

```
echo '{"name": "covid"}' | java -jar hello-world-gradle-0.1-all.jar
```

Pipe-ul (simbolul „|”) care conectează cele 2 comenzi Linux face legătura între ieșirea standard (**stdout**) a comenzi **echo** ... și intrarea standard (**stdin**) a comenzi **java** Deci, aplicația Micronaut care încapsulează funcția *serverless* va primi la intrare textul afișat de comanda **echo** și acesta va fi decodificat din JSON astfel încât să populeze câmpurile clasei **HelloWorldGradle** (sau **HelloWorldMaven**).

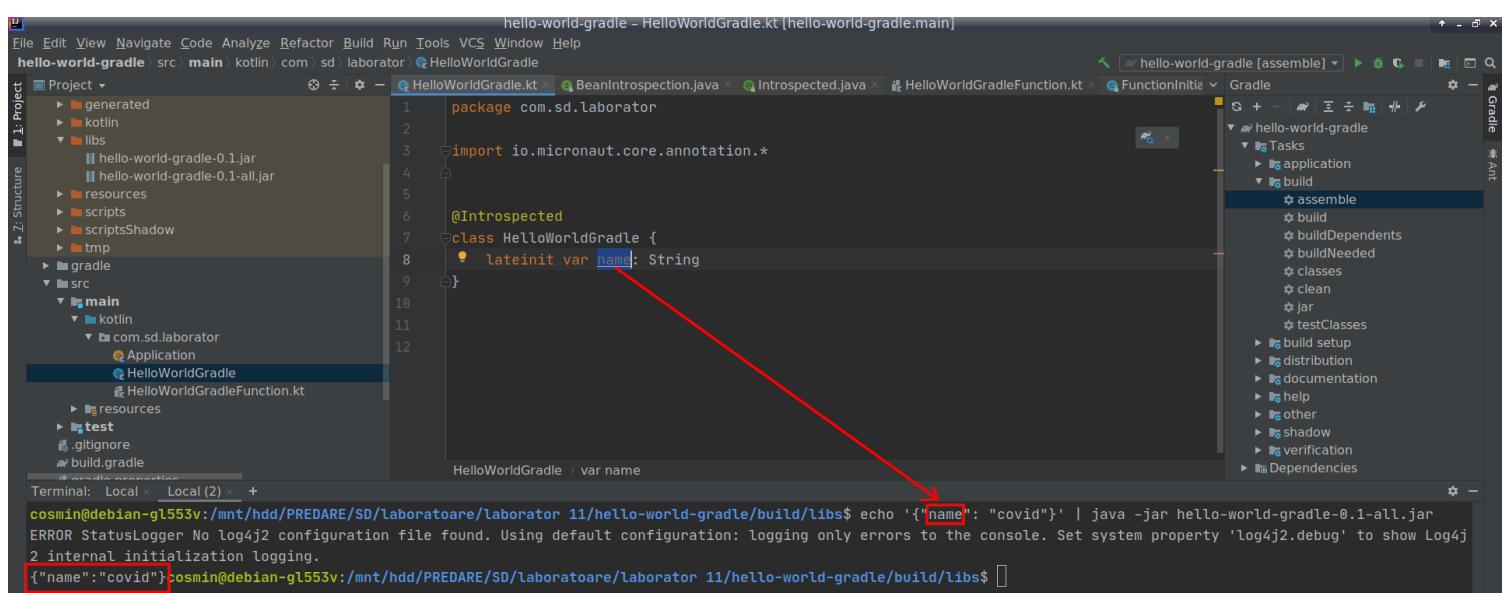


Figura 6 - Execuția funcției *serverless* împachetată cu Gradle

Dacă ați folosit Maven ca gestionar de proiect, s-ar putea să primiți câteva erori la execuția funcției *serverless* (a se vedea figura de mai jos). Sunt legate de plugin-ul **log4j2, iar cauza este pierderea câtorva definiții din fișierul **Log4j2Plugins.dat** utilizat de Micronaut, atunci când se îmbină mai multe artefacte JAR într-unul singur. Se pot îngora fără a afecta funcționalitatea aplicației din laborator.**

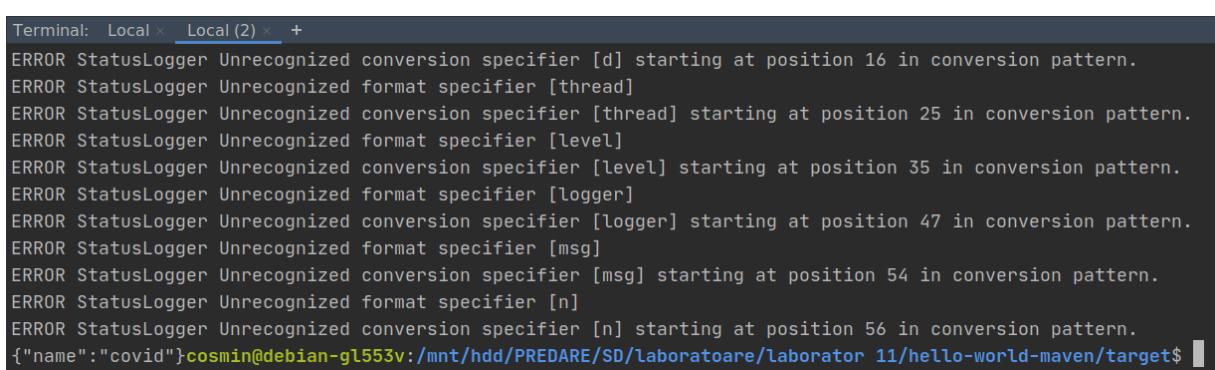


Figura 7 - Execuția funcției *serverless* împachetată cu Maven

Ciurul lui Eratostene - sub formă de funcție *serverless* Micronaut

Ciurul lui Eratostene este un algoritm simplu de descoperire a tuturor numerelor prime până la un întreg specificat ca parametru. Algoritmul are complexitatea $O(n \cdot \log(n))$, în varianta clasică. În laborator se va folosi o variantă optimizată, având complexitatea $O(n)$.

Creați o funcție *serverless* Micronaut folosind linia de comandă, cu denumirea „**com.sd.laborator.eratostene**”.

Modificați codul schelet rezultat astfel:

- Redenumiți clasa **Eratostene** sub forma **EratosteneRequest** și modificați codul astfel:

```
package com.sd.laborator

import io.micronaut.core.annotation.Introspected

@Introspected
class EratosteneRequest {
    private lateinit var number: Integer

    fun getNumber(): Int {
        return number.toInt()
    }
}
```

Această clasă va încapsula cererea primită ca și parametru de intrare. Cererea conține numărul maxim până la care se va calcula lista de numere prime din ciurul lui Eratostene (membrul **number**).

- Adăugați o nouă clasă **@Introspected** care va reprezenta răspunsul dat de funcția *serverless*. Denumiți clasa **EratosteneResponse**.

```
package com.sd.laborator

import io.micronaut.core.annotation.Introspected

@Introspected
class EratosteneResponse {
    private var message: String? = null
    private var primes: List<Int>? = null

    fun getPrimes(): List<Int>? {
        return primes
    }

    fun setPrimes(primes: List<Int>?) {
        this.primes = primes
    }

    fun getMessage(): String? {
        return message
    }

    fun setMessage(message: String?) {
        this.message = message
    }
}
```

Răspunsul funcției *serverless* conține un mesaj cu care se notifică starea de succes a execuției algoritmului, respectiv o listă de numere întregi prime rezultate în urma algoritmului. Lista este populată doar dacă nu există vreo eroare raportată prin variabila **message**.

- Adăugați o componentă **Singleton** numită **EratosteneSieveService**:

```
package com.sd.laborator

import java.util.*
import javax.inject.Singleton

@Singleton
class EratosteneSieveService {
    // implementare preluata de la
    https://www.geeksforgeeks.org/sieve-eratosthenes-on-time-complexity/
    val MAX_SIZE = 1000001

    /*
        isPrime[] : isPrime[i] este adevarat daca numarul i este prim
        prime[] : stocheaza toate numerele prime mai mici ca N
        SPF[] (Smallest Prime Factor) - stocheaza cel mai mic factor prim
        al numarului
        [de exemplu : cel mai mic factor prim al numerelor '8' si '16'
        este '2', si deci SPF[8] = 2 , SPF[16] = 2 ]
    */
    private val isPrime = Vector<Boolean>(MAX_SIZE)
    private val SPF = Vector<Int>(MAX_SIZE)
    fun findPrimesLessThan(n: Int): List<Int> {
        val prime: MutableList<Int> = ArrayList()
        for (i in 2 until n) {
            if (isPrime[i]) {
                prime.add(i)

                // un numar prim este propriul sau cel mai mic factor
                prim
                SPF[i] = i
            }

            /*
                Se sterg toti multiplii lui i * prime[j], care nu sunt
                primi
                setand isPrime[i * prime[j]] = false
                si punand cel mai mic factor prim al lui i * prime[j] ca
                si prime[j]
                [de exemplu: fie i = 5, j = 0, prime[j] = 2 [i * prime[j]
                = 10],
                si deci cel mai mic factor prim al lui '10' este '2' care
                este prime[j] ]

                Aceasta bucla se executa doar o singura data pentru
                numerele care nu sunt prime
            */
            var j = 0
            while (j < prime.size && i * prime[j] < n && prime[j] <=
SPF[i]) {
                isPrime[i * prime[j]] = false

                // se pune cel mai mic factor prim al lui i * prime[j]
            }
        }
    }
}
```

```

        SPF[i * prime[j]] = prime[j]
        j++
    }
}
return prime
}

init {
    // initializare vectori isPrime si SPF
    for (i in 0 until MAX_SIZE) {
        isPrime.add(true)
        SPF.add(2)
    }
    // 0 and 1 are not prime
    isPrime[0] = false
    isPrime[1] = false
}
}
}

```

Clasa **EratosteneSieveService** încapsulează algoritmul de calcul al ciurului lui Eratostene până la un număr întreg **N**. Fiind serviciu unic, este adnotat cu **@Singleton** pentru a fi instantiat o singură dată de componenta injector a Micronaut.

- Modificați funcția **EratosteneFunction** rezultată astfel:

```

package com.sd.laborator;

import io.micronaut.function.FunctionBean
import io.micronaut.function.executor.FunctionInitializer
import org.slf4j.Logger
import org.slf4j.LoggerFactory
import java.util.function.Function
import javax.inject.Inject

@FunctionBean("eratostene")
class EratosteneFunction : FunctionInitializer(),
Function<EratosteneRequest, EratosteneResponse> {
    @Inject
    private lateinit var eratosteneSieveService:
EratosteneSieveService

    private val LOG: Logger =
LoggerFactory.getLogger(EratosteneFunction::class.java)

    override fun apply(msg : EratosteneRequest) : EratosteneResponse {
        // preluare numar din parametrul de intrare al functiei
        val number = msg.getNumber()

        val response = EratosteneResponse()

        // se verifica daca numarul nu depasesete maximul
        if (number >= eratosteneSieveService.MAX_SIZE) {
            LOG.error("Parametru prea mare! $number > maximul de
${eratosteneSieveService.MAX_SIZE}")
            response.setMessage("Se accepta doar parametri mai mici ca
" + eratosteneSieveService.MAX_SIZE)
            return response
        }
    }
}

```

```

        LOG.info("Se calculeaza primele $number numere prime ...")

        // se face calculul si se seteaza proprietatile pe obiectul cu
rezultatul

response.setPrimes(eratosteneSieveService.findPrimesLessThan(number))
    response.setMessage("Calcul efectuat cu succes!")

    LOG.info("Calcul incheiat!")

    return response
}
}

/**
 * This main method allows running the function as a CLI application
using: echo '{}' | java -jar function.jar
 * where the argument to echo is the JSON to be parsed.
 */
fun main(args : Array<String>) {
    val function = EratosteneFunction()
    function.run(args, { context ->
function.apply(context.get(EratosteneRequest::class.java)) })
}

```

S-a creat o clasă **EratosteneFunction** expusă sub formă de funcție Micronaut. Interfața folosită este **Function**, deci se va citi un argument de la intrarea standard și se va scrie un singur răspuns la ieșirea standard.

Se folosește anotarea **@Inject** pentru a injecta automat dependența **EratosteneSieveService**, necesară pentru returnarea listei de numere prime. De asemenea, este utilizată o instanță de tip **Logger**, folosită pentru mesaje informative sau de eroare, în funcție de nivelul de *logging* dorit.

Atenție: NU folosiți metode de tipul `println()` pentru afișarea mesajelor la consolă într-o aplicație Micronaut cu funcții serverless. După cum s-a menționat anterior, funcția serverless Micronaut trimite rezultatul, în mod implicit, către dispozitivul de ieșire standard (**stdout**). Însă, același lucru îl face și funcția `println()` din Kotlin, și alte funcții asemănătoare. **Așadar, pentru a afișa mesaje informative sau erori, folositi exclusiv `Logger-ul` pus la dispoziție de Micronaut, care nu intervine peste canalul standard de ieșire!**

De asemenea, **Logger-ul** este implicit configurat să afișeze **doar mesaje de eroare** la consolă.

Împachetați aplicația și testați-o prin trimiterea datelor necesare în format JSON, către intrarea standard (**stdin**) a funcției *serverless*:

```
echo '{"number": 50}' | java -jar eratostene-0.1-all.jar
```

Aplicații și teme

Temă de laborator

- Modificați aplicația din laborator care implementează ciurul lui Eratostene astfel încât să calculeze, recursiv, termenii din sirul cu numere întregi definit astfel:

$$a_n = a_{n-1} + 2 \cdot \frac{a_{n-1}}{n}, \forall n \geq 1$$

$$a_0 = 1$$

Teme pentru acasă

- Transformați funcția *serverless* din ciurul lui Eratostene în aşa fel încât aplicația să primească, pe lângă numărul maxim până la care algoritmul face calculul necesar, o listă de numere întregi. Aplicația trebuie să utilizeze ciurul lui Eratostene deja implementat ca să decidă care din numerele respective sunt prime sau nu. Se vor returna **DOAR** numerele prime din cele trimise în cerere, și nu toată lista calculată de algoritm.

Sugestie: se poate modifica funcția *serverless* astfel încât să fie de tip **BiFunction**.

Alternativ, se pot încapsula numerele în aceeași cerere cu un singur argument.

- Implementați o aplicație de tip **producător-consumator** folosind 2 funcții *serverless* puse la dispoziție de *framework*-ul Micronaut. Producătorul va prelua fluxul RSS de pe site-ul **xkcd.com** (URL-ul este: <https://xkcd.com/atom.xml>) și va trimite XML-ul către consumator. Consumatorul va prelucra XML-ul astfel:

- se va prelua conținutul tag-ului **<title>**
- se va prelua conținutul atributului **href** din tag-urile de tip **<link href=...></link>**

```

-<feed xml:lang="en">
  <title>xkcd.com</title>
  <link href="https://xkcd.com/" rel="alternate"/>
  <id>https://xkcd.com/</id>
  <updated>2020-05-01T00:00:00Z</updated>
-<entry>
  <title>Turtle Sandwich Standard Model</title>
  <link href='https://xkcd.com/2301/' rel="alternate"/>
  <updated>2020-05-01T00:00:00Z</updated>
  <id>https://xkcd.com/2301/</id>
  -<summary type="html">
    
-</entry>

```

Răspunsul returnat de consumator la ieșirea standard este format dintr-o listă de perechi **<TITLE, URL>** formate utilizând datele extrase din fluxul RSS.

Sugestie: folosiți funcții de tipul **Supplier** și **Consumer / BiConsumer**. Pentru preluarea de pe web a fluxului RSS, se poate folosi biblioteca Kotlin KHTTP (<https://khttp.readthedocs.io/en/latest/>).

URL-ul către Maven Central pentru dependența KHTTP este:

<https://mvnrepository.com/artifact/khttp/khttp/1.0.0>

Bibliografie

- [1]: Documentație Micronaut - <https://micronaut.io/documentation.html>
- [2]: Cum se creează o aplicație minimală Micronaut - <https://guides.micronaut.io/creating-your-first-micronaut-app/guide/index.html>
- [3]: Funcții *serverless* - <https://docs.micronaut.io/latest/guide/serverlessFunctions.html>
- [4]: Biblioteca KHTTP - <https://khttp.readthedocs.io/en/latest/>

Sisteme Distribuite - Laborator 12

Operații Big Data cu Map-Reduce și Hadoop

Introducere

Big Data

Big Data¹ sunt date care conțin o **varietate** mai mare (tipurile de date care sunt disponibile, structurate, semi-structurate și nestructurate), cu un **volum** în continuă creștere (poate ajunge la terabytes de date sau chiar petabytes pentru unele companii) și cu o **viteză** tot mai mare (rata cu care datele sunt primite și poate prelucrate). Aceasta mai este cunoscută și ca „cei trei V”. Mai târziu, au mai fost inclusi doi V: **valoare** și **veridicitate**. Valoarea intrinsecă a datelor e irelevantă dacă nu este descoperită. De asemenea, veridicitatea datelor este crucială pentru a putea lua decizii în baza lor.

Cazuri de utilizare:

- Dezvoltarea produsului: anticiparea cererilor clienților
- Mantenanță predictivă
- Experiența clientului
- Învățare automată (machine learning)
- Fraudă și conformitate

Ce presupune Big Data?

- Integrare: obținerea și prelucrarea datelor într-un format pe care analistul îl recunoaște;
- Gestionarea spațiului de stocare: datele pot fi stocate în cloud, local (în sedii), sau în ambele;
- Analiza datelor cu scopul de a lua decizii.

MapReduce

MapReduce² este o paradigmă de programare pentru calculul distribuit. Acest model implică, în general, existența unui nod de procesare cu rol de coordonator (sau master sau inițiator) și mai multe noduri de procesare cu rol de worker.

Modelul **MapReduce** cuprinde două etape: Map (task-ul de mapare) și Reduce (task-ul de reducere):

1. Etapa de **mapare** – primește un set de date de intrare și îl convertește în alt set de date, unde elementele individuale sunt „sparte” în perechi cheie-valoare
 - nodul cu rol de *coordonator* împarte problema „originală” în subprobleme și le distribuie către *workeri* pentru procesare;
 - divizarea problemei de lucru (a datelor de procesat) se realizează într-o manieră similară divide-et-impera – în unele cazuri nodurile worker pot diviza la rândul lor subproblema primită și pot trimite aceste subdiviziuni către alți workeri, rezultând o arhitectură arborescentă;
 - divizarea caracteristică acestei etape nu trebuie să coreleză efectiv dimensiunea datelor de intrare cu numărul de workeri din sistem; un worker poate primi mai multe subprobleme de rezolvat;

¹<https://www.oracle.com/big-data/guide/what-is-big-data.html>

²<https://www.ibm.com/analytics/hadoop/mapreduce>

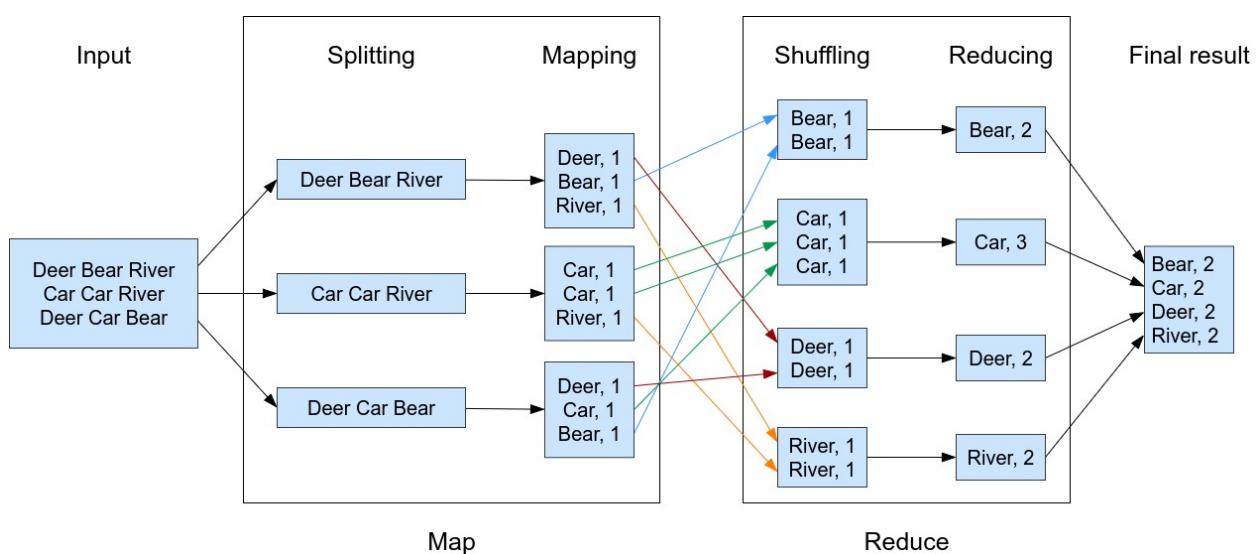
2. Etapa de **reducere** – primește ca date de intrare ieșirea de la etapa de mapare și combină perechile cheie-valoare astfel încât câmpul cheie va fi cuvântul, iar câmpul valoare va fi frecvența de apariție a cuvântului în text. Așadar, rezultă un set de date mai mic.
 - nodul cu rol de *coordonator* (sau un set de noduri cu rol de worker „desemnat” de *coordonator*) colectează soluțiile subproblemelor și le combină pentru a obține rezultatul final al procesării dorite.

Michael Kleber³ (Google Inc.) definește următoarele etape implicate în paradigma MapReduce:

1. pre-procesare – datele sunt pregătite pentru funcția de mapare
2. mapare – stabilirea datelor de interes
3. amestecare și sortare – datele pot fi organizate astfel încât să fie optimizată etapa de reducere
4. reducere – determinarea rezultatului
5. stocare rezultat

Avantaje:

- Scalabilitate masivă pe sute sau mii de servere într-un cluster;
- Flexibilitate (accesarea mai ușoară a mai multor surse și tipuri de date)
- Parallelizare automată;
- Echilibrarea încărcării (load balancing)
- Gestionarea defectiunilor (machine failures) prin reasignarea task-ului unui alt woker;



Exemplu de aplicare a Map-Reduce pentru Word Counter

Hadoop

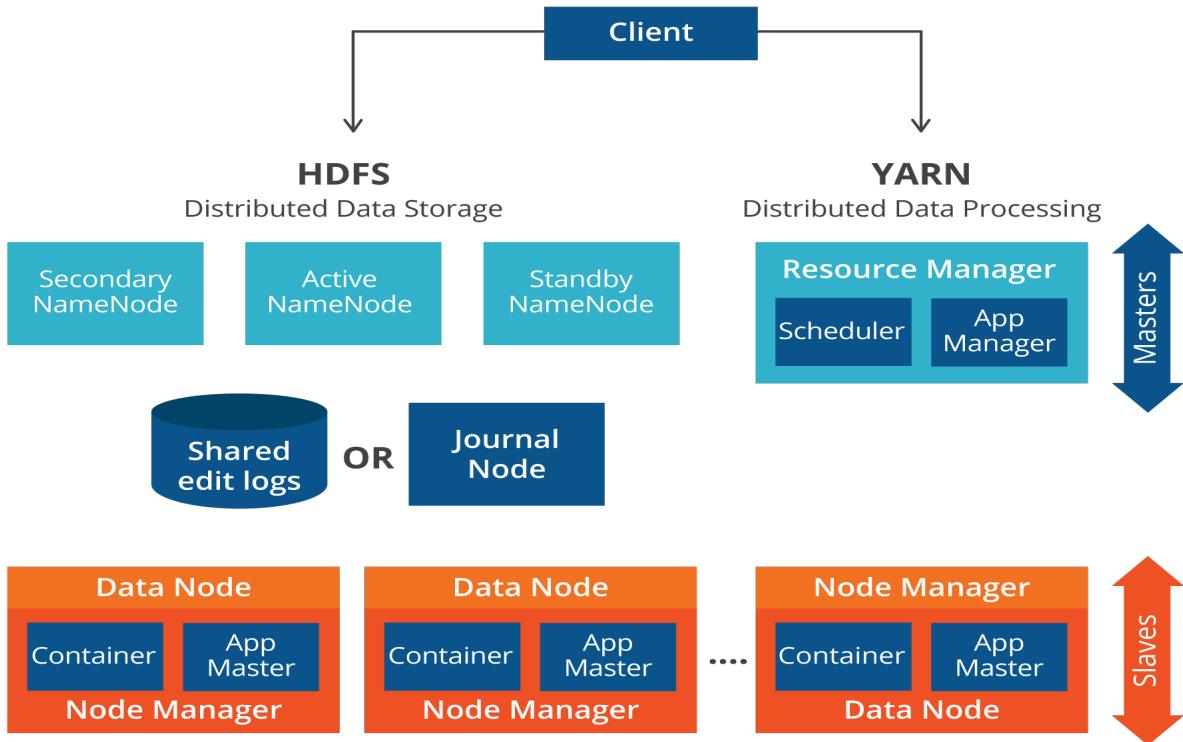
Hadoop este un sistem de fișiere distribuit și totodată un sistem de procesare distribuită a unor seturi immense de date pe cluster-e folosind paradigme de programare precum MapReduce. Acesta a fost introdus de Apache, fiind un framework open-source scris în java. Hadoop este proiectat să mărească spațiul de stocare și puterea de calcul de la un singur server până la mii de calculatoare. De asemenea, acesta a fost proiectat să fie „tolerant la defecte” (fault tolerant), adică în cazul în care un nod din cluster se defectează, datele nu se vor pierde.

Deși framework-ul este scris în java, el conține un utilitar numit Hadoop Streaming (utilizat în cadrul exemplului din laborator) ce permite utilizatorilor să creeze și să execute task-uri cu orice tip de executabil (ex.: python, java) ca Mapper și/sau Reducer.

Hadoop are o arhitectură de tip master-slave, reprezentată schematic în figura de mai jos..

³http://www.csce.uark.edu/~mqhuang/courses/5013/s2018/lecture/5_intro_to_mapreduce.pdf

Apache Hadoop 2.0 and YARN



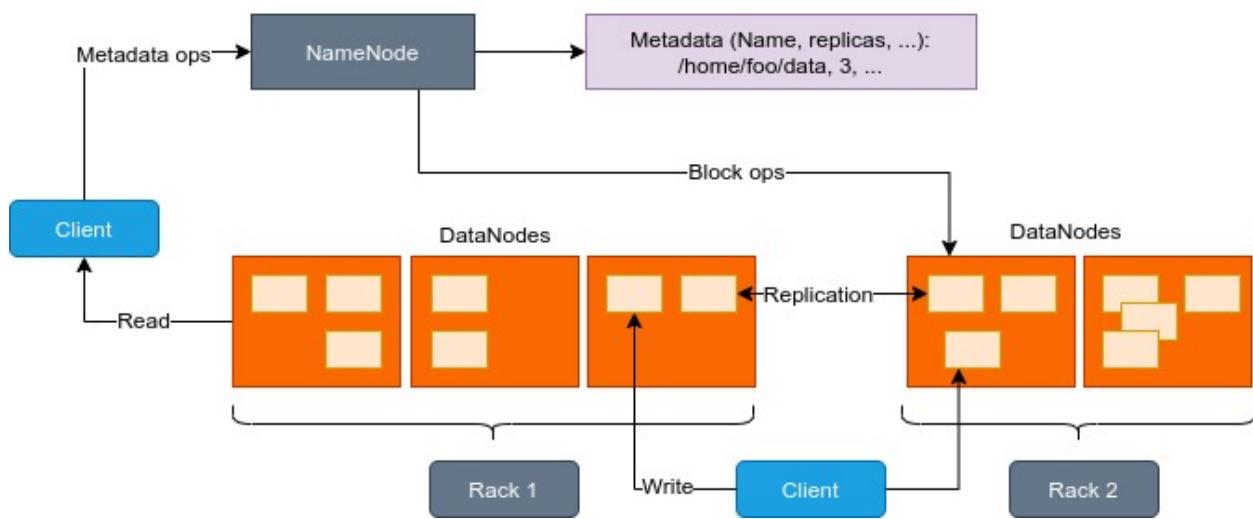
Principalele componente ale Apache Hadoop sunt:

- **NameNode-ul** – controlează funcționarea job-urilor de date
- **DataNode** – scrie datele în blocuri în sistemul local de stocare și replică blocuri de date către alte DataNode-uri
- **SecondaryNameNode** – rezervă pentru DataNode în cazul în care se defectează
- **JobTracker** – trimite job-uri de tip MapReduce către nodurile din cluster
- **TaskTracker** – acceptă task-uri de la JobTracker
- **Yarn (Yet Another Resource Manager)** – pornește componentele ResourceManager și NodeManager; Yarn-ul este un framework pentru programarea job-urilor (job scheduling) și gestionarea resurselor din cluster
- **Client Application** – programul pe care îl scrii (MapReduce)
- **Application Master** - execută comenzi shell într-un container după cum îi dictează yarn-ul

Apache HDFS (Hadoop Distributed File System)⁴ este un sistem de fișiere structurat pe blocuri în care fiecare fișier este împărțit în blocuri de dimensiune pre-determinată, acestea fiind stocate într-un cluster cu una sau mai multe mașini de calcul. HDFS are o arhitectură master/slave ce poate fi observată în figura de mai jos, unde cluster-ul este format dintr-un singur NameNode (nodul master) și toate celelalte noduri sunt DataNode-uri (noduri slave).

⁴<https://www.edureka.co/blog/apache-hadoop-hdfs-architecture/>

HDFS Architecture



Configurarea mediului de lucru

Instalare Oracle Java versiunea 1.8.0_221

Atenție! Hadoop NU funcționează cu versiunile de Java > 8, aşadar trebuie să folosiți o versiune de **Java SDK cel mult egală cu 8**. Verificați versiunea de Java cu următoarea comandă:

```
java -version
```

În cazul în care sunt disponibile mai multe versiuni de Java pe stația de lucru (cu sistemul de operare Debian), iar acestea sunt înregistrate ca alternative, se poate schimba versiunea de Java la cerere, folosind comanda:

```
sudo update-alternatives --config java
```

La executarea ultimei comenzi, este necesară introducerea indexului versiunii java ca în figura de mai jos. Se alege **java 8**.

```
There are 2 choices for the alternative java (providing /usr/bin/java).
```

Selection	Path	Priority	Status
*	/usr/lib/jvm/java-11-openjdk-amd64/bin/java	1111	auto mode
1	/usr/lib/jvm/java-11-openjdk-amd64/bin/java	1111	manual mode
2	/usr/lib/jvm/oracle-java8-jdk-amd64/bin/java	1081	manual mode

```
Press <enter> to keep the current choice[*], or type selection number: 2
```

Selectarea versiunii de java

Similar, se va configura și javac (java compiler):

```
sudo update-alternatives --config javac
```

Configurarea mediului de lucru

```
There are 2 choices for the alternative javac (providing /usr/bin/javac).
```

Selection	Path	Priority	Status
*	/usr/lib/jvm/java-11-openjdk-amd64/bin/javac	1111	auto mode
1	/usr/lib/jvm/java-11-openjdk-amd64/bin/javac	1111	manual mode
2	/usr/lib/jvm/oracle-java8-jdk-amd64/bin/javac	1071	manual mode

```
Press <enter> to keep the current choice[*], or type selection number: 2
```

Selectarea versiunii de javac (Java Compiler)

Dacă versiunea Java 8 nu este instalată, urmați pașii de mai jos:

1. Se va descărca de pe <https://www.oracle.com/java/technologies/javase/javase-jdk8-downloads.html>
jdk-8u251-linux-x64.tar.gz.
2. Se va descărca de pe <https://www.oracle.com/java/technologies/javase-jre8-downloads.html>
jre-8u251-linux-x64.tar.gz

Se vor executa următoarele comenzi:

```
tar -xvf jdk-8u251-linux-x64.tar.gz
sudo mv jdk1.8.0_251 /usr/lib/jvm
tar -xvf jre-8u251-linux-x64.tar.gz
sudo mv jre1.8.0_251 /usr/lib/jvm

sudo update-alternatives --install /usr/bin/java java
/usr/lib/jvm/jdk1.8.0_251 1103
sudo update-alternatives --config java
```

La ultima comandă executată de mai sus, se introduce numărul opțiunii
/usr/lib/jvm/jdk1.8.0_251.

Dacă este setată variabila *JAVA_HOME* și în /etc/environment, va trebui actualizată calea:

```
sudo vim /etc/environment
```

```
JAVA_HOME=/usr/lib/jvm/oracle-java8-jdk-amd64
```

```
. /etc/environment # reload environment file
java -version # check java version
```

Instalare Apache Hadoop versiunea 3.2.1

Se adaugă un utilizator de sistem pentru Hadoop

```
sudo addgroup hadoop # create group hadoop
sudo adduser --ingroup hadoop hduser # create user hduser inside the
hadoop group
```

```
Adding user 'hduser' ...
Adding new user 'hduser' (1001) with group `hadoop' ...
Creating home directory `/home/hduser' ...
Copying files from `/etc/skel' ...
New password: parola: hduser
Retype new password:
passwd: password updated successfully
Changing the user information for hduser
Enter the new value, or press ENTER for the default
    Full Name []:
    Room Number []:
    Work Phone []:
    Home Phone []:
    Other []:
Is the information correct? [Y/n] Y
```

Crearea utilizatorului *hduser*

Se vor executa într-un terminal următoarele comenzi:

```
cd ~/Downloads # change directory
wget https://www-eu.apache.org/dist/hadoop/common/stable/hadoop-
3.2.1.tar.gz # download hadoop
tar -xvf hadoop-3.2.1.tar.gz # extract from archive
sudo chown -R hduser:hadoop hadoop-3.2.1 # change owner and group to
hduser and hadoop
sudo mv hadoop-3.2.1 hadoop # rename to hadoop
sudo mv hadoop /opt # move hadoop to /opt directory
```

Instalare și configurare SSH

Pentru instalare, se execută comenziile de mai jos:

```
sudo apt update
sudo apt install openssh-server
```

Pentru a verifica instalarea corectă, se poate utiliza comanda:

```
sudo systemctl status sshd
```

Este necesară generarea unei chei SSH. La întrebarea cu locația cheii, se apasă ENTER pentru locația default.

```
su - hduser # switch user to hduser
# <password for hduser> # enter the password for hduser

ssh-keygen -t rsa -P "" # generate SSH key
```

```
hduser@debian:~$ ssh-keygen -t rsa -P ""
Generating public/private rsa key pair.
Enter file in which to save the key (/home/hduser/.ssh/id_rsa):
Created directory '/home/hduser/.ssh'.
Your identification has been saved in /home/hduser/.ssh/id_rsa.
Your public key has been saved in /home/hduser/.ssh/id_rsa.pub.
The key fingerprint is:
SHA256:ZdTm0xJxCKwNf7Ae8SoYF600XPIWc6gPjPsdncc+2g hduser@debian
The key's randomart image is:
+---[RSA 2048]---+
|       .+=o.   |
|     + + =o==  |
|   = & * B= .  |
| . X B =.oo   |
|o +   OS      |
| *       +     |
| . +   . o    |
| o .   E.    |
| . . . o...   |
+---[SHA256]-----+
```

Generarea unei chei SSH

Este necesară autorizarea cheii SSH pentru a putea realiza și testa o conexiune pe localhost.

```
cat ~/.ssh/id_rsa.pub >> ~/.ssh/authorized_keys # authorize SSH key
ssh localhost # check connection
```

Conexiunea SSH este stabilită cu setările default.

[OPTIONAL]: Dacă se dorește securizarea accesului la server, se recomandă modificarea configurațiilor SSH (a se vedea fișierul de configurare *sshd_config* anexat și comentariile aferente, introduse prin caracterul #). Pentru modificarea setărilor SSH, se deschide fișierul */etc/ssh/sshd_config* cu editorul preferat (sunt necesare drepturi de admin, deci se execută cu *sudo*).

```
sudo featherpad /etc/ssh/sshd_config
```

Dacă s-au realizat modificări în fișierul de configurare al server-ului SSH, trebuie forțată reîncărcarea configurațiilor cu comanda:

```
sudo service ssh reload
```

Configurarea variabilelor *HADOOP_HOME* și *JAVA_HOME*

Se actualizează fișierul */home/hduser/.bashrc*.

```
su - hduser
# <password for hduser>
vim ~/.bashrc # deschidere fisier pentru editare
# sau
nano ~/.bashrc
# sau
mousepad ~/.bashrc # pentru un editor grafic
```

Se adaugă la finalul fișierului următoarele linii:

```
# Set HADOOP HOME
```

Laborator 12

```
export HADOOP_HOME=/opt/hadoop
export HADOOP_CONF_DIR=$HADOOP_HOME/etc/hadoop
export HADOOP_MAPRED_HOME=$HADOOP_HOME
export HADOOP_COMMON_HOME=$HADOOP_HOME
export HADOOP_HDFS_HOME=$HADOOP_HOME
export YARN_HOME=$HADOOP_HOME
export HADOOP_COMMON_LIB_NATIVE_DIR=$HADOOP_HOME/lib/native
export PATH=$PATH:$HADOOP_HOME/sbin:$HADOOP_HOME/bin

# Set JAVA HOME
export JAVA_HOME=/usr/lib/jvm/jdk1.8.0_251 # check path
export PATH=$JAVA_HOME/bin:$PATH

# Mofify pdsh's default rcmd to ssh
export PDSH_RCMD_TYPE=ssh
# export HADOOP_SSH_OPTS="-p 4567" # if you modified the port
```

Se reîncarcă configurările anterioare executând comanda:

```
source ~/.bashrc
```

Configurare Apache Hadoop

Pentru configurarea Apache Hadoop, trebuie editate mai multe fișiere. Într-un **terminal nou**, se execută următoarele comenzi:

```
sudo vim <nume_fisier>
# exemplu:
sudo mousepad /opt/hadoop/etc/hadoop/hadoop-env.sh
```

1. /opt/hadoop/etc/hadoop/hadoop-env.sh

Se adaugă următoarele linii:

```
export JAVA_HOME=/usr/lib/jvm/jdk1.8.0_251 # check path
export HADOOP_OPTS=-Djava.net.preferIPv4Stack=true
```

2. /opt/hadoop/etc/hadoop/core-site.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>
<configuration>
    <property>
        <name>fs.default.name</name>
        <value>hdfs://localhost:9000</value>
    </property>
</configuration>
```

3. /opt/hadoop/etc/hadoop/mapred-site.xml

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>
<configuration>
    <property>
        <name>mapreduce.framework.name</name>
        <value>yarn</value>
    </property>
    <property>
        <name>yarn.app.mapreduce.am.resource.mb</name>
```

```

<value>1024</value>
</property>
<property>
    <name>yarn.app.mapreduce.am.command-opts</name>
    <value>-Xmx983m</value>
</property>
<property>
    <name>mapreduce.map.memory.mb</name>
    <value>1024</value>
</property>
<property>
    <name>mapreduce.reduce.memory.mb</name>
    <value>1024</value>
</property>
<property>
    <name>mapreduce.map.java.opts</name>
    <value>-Xmx983m</value>
</property>
<property>
    <name>mapreduce.reduce.java.opts</name>
    <value>-Xmx983m</value>
</property>
<property>
    <name>yarn.scheduler.capacity.maximum-am-resource-percent</name>
    <value>100</value>
</property>
<property>
    <name>yarn.app.mapreduce.am.env</name>
    <value>HADOOP_MAPRED_HOME=$HADOOP_HOME</value>
</property>
<property>
    <name>mapreduce.map.env</name>
    <value>HADOOP_MAPRED_HOME=$HADOOP_HOME</value>
</property>
<property>
    <name>mapreduce.reduce.env</name>
    <value>HADOOP_MAPRED_HOME=$HADOOP_HOME</value>
</property>
</configuration>

```

4. /opt/hadoop/etc/hadoop/hdfs-site.xml

```

<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>
<configuration>
    <property>
        <name>dfs.replication</name>
        <value>1</value>
    </property>
    <property>
        <name>dfs.permission</name>
        <value>false</value>
    </property>
</configuration>

```

5. /opt/hadoop/etc/hadoop/yarn-site.xml

Laborator 12

```
<?xml version="1.0"?>
<configuration>
  <property>
    <name>yarn.scheduler.minimum-allocation-mb</name>
    <value>1228</value>
  </property>
  <property>
    <name>yarn.scheduler.maximum-allocation-mb</name>
    <value>9830</value>
  </property>
  <property>
    <name>yarn.nodemanager.resource.memory-mb</name>
    <value>9830</value>
  </property>
  <property>
    <name>yarn.nodemanager.aux-services</name>
    <value>mapreduce_shuffle</value>
  </property>
  <property>
    <name>yarn.nodemanager.auxservices.mapreduce.shuffle.class</name>
    <value>org.apache.hadoop.mapred.ShuffleHandler</value>
  </property>
  <property>
    <name>yarn.nodemanager.disk-health-checker.max-disk-utilization-per-disk-percentage</name>
    <value>100</value>
  </property>
</configuration>
```

6. /etc/hosts

Pentru a obține adresa IP privată, se execută în terminal comanda:

```
ip a | grep "inet 192" | awk '{print $2}' | cut -d/ -f1
```

Pentru a obține hostname-ul, rulați în terminal comanda:

```
hostname
```

Se deschide cu drepturi de administrator fișierul /etc/hosts:

```
sudo featherpad /etc/hosts
```

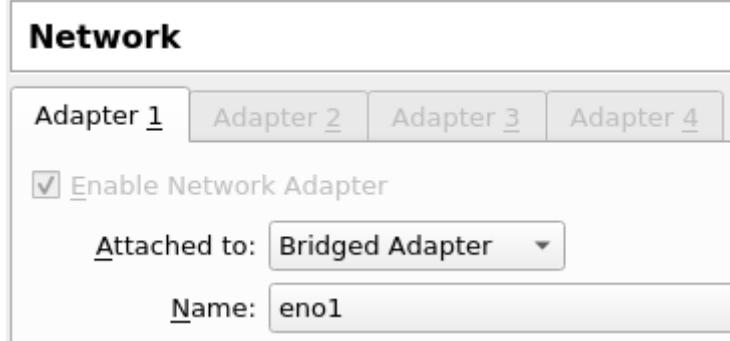
Se modifică cu conținutul:

```
#127.0.0.1 localhost
#127.0.1.1 debian

# În loc de 192.168.0.100 puneți adresa IP privată obținută anterior
# În loc de debian puneți hostname-ul obținut anterior
192.168.0.100 debian localhost

# The following lines are desirable for IPv6 capable hosts
#:1      localhost ip6-localhost ip6-loopback
#ff02::1 ip6-allnodes
#ff02::2 ip6-allrouters
```

Dacă se utilizează o mașină virtuală, se vor modifica setările pentru Network ca în imaginea de mai jos:



Configurarea setărilor de rețea pentru VirtualBox

Observație: În loc de *eno1* este posibil să fie afișat *eth0* sau alte denumiri de rețea. Nu se selectează rețelele wireless (ex.: *wlp2s0*).

Initializare HDFS

Într-un **terminal deschis pe user-ul *hduser***, pentru formatarea componentei NameNode, se execută comanda de mai jos.

```
hdfs namenode -format
```

Atenție: aceasta va șterge toate fișierele pe care stocate anterior în hdfs. Dacă se reformatează, se vor relua pașii prin care se copiază datele în hdfs.

Dacă apare întrebarea „Re-format filesystem in Storage Directory root= /tmp/hadoop-hduser/dfs/name; location= null ? (Y or N)” se răspunde Y și se apasă tasta ENTER.

Pentru a porni/opri cluster-ul cu un singur nod, există două posibilități

1. Se pornesc toate componente:

```
start-all.sh  
# respectiv  
stop-all.sh
```

2. Se pornesc componente individual:

```
start-dfs.sh  
start-yarn.sh  
jps # afiseaza componente active  
# respectiv  
stop-yarn.sh  
stop-dfs.sh
```

Pentru crearea directorului */user/hduser/input* în hdfs, se execută comanda:

```
hdfs dfs -mkdir -p /user/hduser/input
```

Exemplu MapReduce

Exemplul propus este un Word Counter care folosește două scripturi python: mapper.py și reducer.py. Mapper-ul sparge efectiv fișierele text în perechi de forma:

```
<cuvânt_0, 1>
<cuvânt_1, 1>
<cuvânt_0, 1>
<cuvânt_0, 1>
```

Reducer-ul reunește acele perechi, obținând astfel numărul de apariții ale cuvintelor. Rezultatul final este:

```
<cuvânt_0, 3>
<cuvânt_1, 1>
```

Codul sursă

- *~/exemplu_mapreduce/mapper.py*

```
#!/usr/bin/env python
"""mapper.py"""

import sys

for line in sys.stdin:
    line = line.strip()
    words = line.split()
    for word in words:
        # write the results to STDOUT
        # this output will be the input for the Reduce step
        print('%s\t%s' % (word, 1))
```

- *~/exemplu_mapreduce/reducer.py*

```
#!/usr/bin/env python
"""reducer.py"""

import sys

current_word = None
current_count = 0
word = None

for line in sys.stdin:
    line = line.strip()
    word, count = line.split()

    try:
        count = int(count)
    except ValueError:
        continue

    # this IF-switch only works because Hadoop sorts map output
    # by key (here: word) before it is passed to the reducer
    if current_word != word:
        if current_word:
            print(current_word, current_count)

        current_word = word
        current_count = 0

    current_count += int(count)

print(word, current_count)
```

```

        print('%.s\t%.s' % (current_word, current_count))
    current_count = count
    current_word = word
    current_count += count

if current_count > 0:
    print('%.s\t%.s' % (current_word, current_count))

```

Se deschide **un nou terminal** (pe user-ul curent, nu *hduser*) în folder-ul \$HOME și se execută comenziile:

```

sudo chown -R hduser:hadoop exemplu_mapreduce
sudo mv exemplu_mapreduce/ /home/hduser

```

Se poate închide terminalul curent și **se revine la cel anterior (în care este logat *hduser*)**.

Acum se copiază fișierele text în hdfs (Hadoop Distributed File System), executând comanda:

```

hdfs dfs -put /home/hduser/exemplu_mapreduce/input/*
/user/hduser/input

```

Dacă apare eroarea „put: File /user/hduser/input/pg20417.txt._COPYING_ could only be written to 0 of the 1 minReplication nodes. There are 0 datanode(s) running and 0 node(s) are excluded in this operation.”, executați următoarele comenzi:

```

stop-all.sh
rm -rf /tmp/hadoop-
hdfs namenode -format
start-all.sh
hdfs dfs -mkdir -p /user/hduser/input
hdfs dfs -put /home/hduser/exemplu_mapreduce/input/*
/user/hduser/input

```

Dacă se observă un mesaj de eroare „*ipc.Client: Retrying connect to server: 0.0.0.0/0.0.0.0:8032. Already tried 0 time(s)*”, Resource Manager-ul nu este activ. Executați comanda:

```
start-yarn.sh
```

Pentru testarea exemplului, executați comanda de mai jos:

```

hadoop jar $HADOOP_HOME/share/hadoop/tools/lib/hadoop-streaming*.jar
-input /user/hduser/input -output /user/hduser/output -mapper
/home/hduser/exemplu_mapreduce/mapper.py -reducer
/home/hduser/exemplu_mapreduce/reducer.py -file
/home/hduser/exemplu_mapreduce/mapper.py -file
/home/hduser/exemplu_mapreduce/reducer.py

```

Dacă au fost urmați toți pașii, output-ul obținut este următorul:

```

2020-02-07 03:13:47,564 INFO mapreduce.Job: The url to track the job: http://localhost:8088/proxy/application_1581063088186_0001/
2020-02-07 03:13:47,568 INFO mapreduce.Job: Running job: job_1581063088186_0001
2020-02-07 03:13:59,894 INFO mapreduce.Job: Job job_1581063088186_0001 running in uber mode : false
2020-02-07 03:13:59,899 INFO mapreduce.Job: map 0% reduce 0%
2020-02-07 03:14:17,190 INFO mapreduce.Job: map 100% reduce 0%
2020-02-07 03:14:25,293 INFO mapreduce.Job: map 100% reduce 100%
2020-02-07 03:14:26,321 INFO mapreduce.Job: Job job_1581063088186_0001 completed successfully
2020-02-07 03:14:26,466 INFO mapreduce.Job: Counters: 55

```

Pentru verificarea rezultatului, se listează conținutul directorului *output*, apoi se copiază fișierul de ieșire în exteriorul hdfs.

Laborator 12

```
hdfs dfs -ls /user/hduser/output
```

```
Found 2 items
-rw-r--r-- 1 hduser supergroup      0 2020-02-07 03:14 /user/hduser/output/_SUCCESS
-rw-r--r-- 1 hduser supergroup  888040 2020-02-07 03:14 /user/hduser/output/part-00000
```

```
hdfs dfs -get /user/hduser/output/part-00000 /home/hduser/
xdg-open /home/hduser/part-00000
```

Rezultatul va fi un fișier text în care pe fiecare linie va fi un cuvânt și numărul de apariții ale cuvântului respectiv în fișierele de intrare.

```
"Italians,"      2
"Je,      2
"La      2
"Le      4
"Leonardo     2
"Les      2
"Libricciuolo  2
"Libro     2
"Lionardo     2
"Lionardo"    2
"Lipsia,      2
"Locho      2
"Magnifico",   2
"Majestati    2
"Man       2
"Many      2
```

Rezultatul obținut

Aplicații și teme

Teme de laborator

1. Să se implementeze un algoritm de sortare disitribuit (se va utiliza sistemul de fișiere distribuit Hadoop), bazat pe algoritmul Map-Reduce. Funcția de mapare va extrage cuvintele dintr-un set de fișiere text și va returna perechi de forma: <prima_literă_din_cuvânt, cuvântul>, iar funcția de reducere va reuni perechile generate de funcția de mapare în perechi de forma: <litteră, [cuvânt0, cuvânt1, ...]>, valoarea fiind practic o listă de cuvinte care încep cu acea literă.
2. Să se implementeze o funcție GREP distribuită (se va utiliza Hadoop) pe baza algoritmului Map-Reduce. Funcția de mapare va returna liniile din output-ul unei comenzi linux (obținut cu ajutorul modulului *subprocess*) care corespund unui regex dat ca parametru, iar funcția de reducere va reuni acele linii într-un singur string.

Teme pentru acasă

1. Să se implementeze o aplicație care generează utilizând algoritmul Map-Reduce (și Hadoop) un Site-Map pentru fiecare pagină WEB dintr-un set de URL-uri. Funcția de mapare va face un request HTTP de tip GET pentru conținutul unei pagini, va parsa HTML-ul pentru a extrage toate ancorele (link-urile) și va genera perechi de forma <URL_paginiă, URL_intern>. Funcția de reducere va reuni acele perechi, returnând elemente de forma <URL_paginiă, [URL_intern0, URL_intern1, ...]> în care valoarea este o listă ce conține toate URL-urile interne din acea pagină (practic, Site-Map-ul).
2. Să se implementeze o aplicație care extrage istoricul URL-urilor accesate dintr-un browser (spre exemplu Firefox) și calculează pe baza algoritmului Map-Reduce frecvența de accesare a acestora. Funcția de mapare va genera perechi de forma <URL, 1>, iar funcția de reducere va aduna toate valorile comune pentru aceeași pagină, generând o pereche de forma: <URL, frecvența_de_accesare>. Observație: se pot elimina fragmentele din URL (#fragment).

HINT: Pentru browser-ul Firefox, baza de date SQLite3 cu istoricul URL-urilor accesate se află în locația: `~/.mozilla/firefox/abcd12ef.default/places.sqlite` (partea cu roșu variază), tabela de interes fiind *moz_places*. Întrucât *places.sqlite* oferă deja frecvența de accesare, funcția map va genera perechi de forma <host, frecvență_URL> (exemplu: <google.com, 12345>, <google.com, 3120>)

Bibliografie

- [1] Jeffrey Dean, Sanjay Ghemawat, „*MapReduce: Simplified data processing on large clusters*”, OSDI, 2004
- [2] Christopher D. Manning, Prabhakar Raghavan, Hinrich Schütze, „*An Introduction to Information Retrieval*”, Cambridge University Press , 2009.
- [3] Garry Turkington, Tanmay Deshpande, Sandeep Karanth, „*Hadoop: Data processing and modelling*”
- [4] Srinath Perera, Thilina Gunarathne, „*Hadoop MapReduce Cookbook*”
- [5] Documentația Apache Hadoop: <https://hadoop.apache.org/docs/current/>
- [6] Documentația SQLite3: <https://docs.python.org/3/library/sqlite3.html>

Sisteme Distribuite - Laborator 13

Apache Kafka și Apache Spark

Introducere

Apache Kafka

Apache Kafka este o platformă de fluxuri de evenimente distribuite, capabilă să gestioneze miliarde de evenimente pe zi. Această platformă poate:

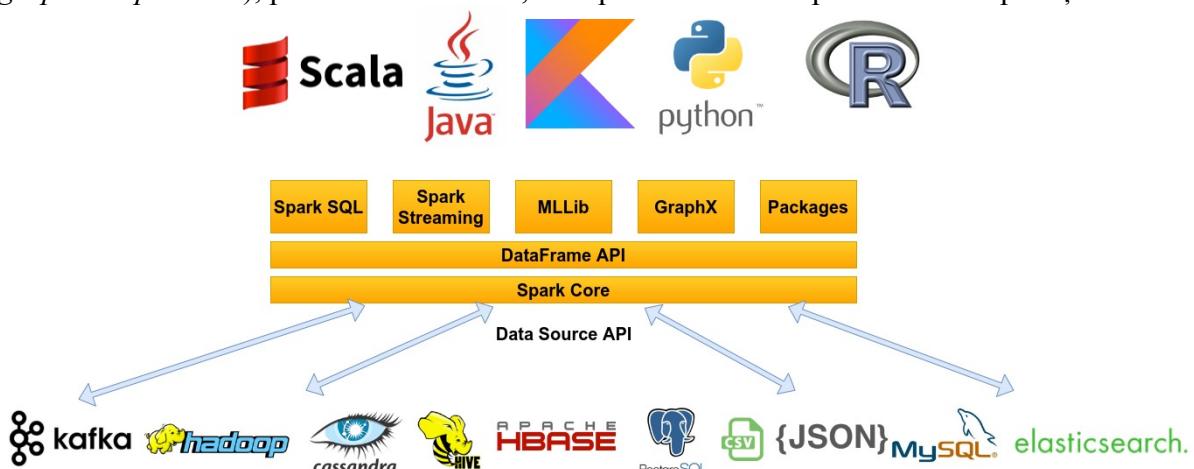
- să publice/să se aboneze la fluxuri de înregistrări (eng. *records*), în mod similar cu o coadă de mesaje;
- să stocheze fluxuri de înregistrări într-un mod durabil și tolerant la erori (*fault-tolerant*)
- să proceseze fluxuri de înregistrări pe măsură ce apar

Pentru mai multe detalii, vezi:

- Cursurile de la disciplina *Sisteme distribuite*
- Laboratorul 10 de la disciplina *Sisteme distribuite*
- <https://www.confluent.io/what-is-apache-kafka/>
- <https://kafka.apache.org/intro>
- <https://kafka.apache.org/documentation/>

Apache Spark

Spark este un motor de uz general de procesare distribuită a datelor. Peste *framework*-ul Spark, există biblioteci pentru SQL, învățare automată (*machine learning*), calcule pe grafuri (*graph computation*), procesări de fluxuri, care pot fi folosite împreună într-o aplicație.



Cazuri tipice de utilizare:

- **Procesarea fluxurilor (*stream processing*)**: deși este fezabilă stocarea datelor pe disk și procesarea lor ulterior, uneori este necesar să se acționeze asupra datelor pe măsură ce acestea sosesc (spre exemplu tranzacțiile financiare trebuie procesate în timp real pentru a identifica și refuza tranzacții potențial frauduloase)
- **Învățare automată (*machine learning*)**: Soluțiile software pot fi antrenate să identifice și să acționeze în funcție de trigger-ele din seturile de date bine înțelese, înainte de a aplica aceleași soluții la date noi și la date necunoscute. Abilitatea *framework*-ului Spark de a stoca date în memorie și de a executa rapid interogări

repetate îl face o alegere potrivită pentru antrenarea algoritmilor de învățare automată.

- **Analiză interactivă (*interactive analytics*):** procesul de interogare interactivă necesită un sistem precum Spark care să fie capabil să răspundă și să se adapteze rapid.
- **Integrarea datelor (*data integration*):** Spark și Hadoop sunt folosite din ce în ce mai mult pentru a reduce costul și timpul necesar pentru procesul **ETL** (*Extract, Transform, Load*)

Avantaje:

- Simplitate
- Viteză
- Suport
- *Pipeline*-uri de date

Pentru mai multe detalii despre framework-ul Spark, precum și istoria acestuia (de ce s-a trecut de la Hadoop la Spark), vezi:

- Cursurile de la disciplina *Sisteme distribuite*
- <https://mapr.com/blog/spark-101-what-it-is-and-why-it-matters/>
- <https://spark.apache.org/docs/2.4.5/>

Spark RDD (Resilient Distributed Dataset)

Abstractizarea principală oferită de Spark este un **RDD**, mai precis, o colecție de elemente partaționate pe nodurile dintr-un cluster pe care se poate acționa în paralel.

O altă abstractizare în Spark reprezintă **variabilele partajate** care pot fi utilizate în operații paralele. În mod implicit, când Spark execută o funcție în paralel ca un set de task-uri pe diferite noduri, furnizează o copie a fiecărei variabile utilizate în funcție către fiecare task. Uneori, este nevoie totuși de partajarea unei variabile între task-uri. Framework-ul Spark suportă două tipuri de variabile partajate: *variabile de broadcast* care pot fi utilizate pentru păstrarea în cache a unei variabile în memorie pe toate nodurile, și *acumulatori* care sunt variabile în care doar se adaugă (*counter*, sume).

Pentru mai multe detalii, vezi:

- <https://spark.apache.org/docs/2.4.5/rdd-programming-guide.html>

Crearea unui proiect Spark (RDD)

Se creează un nou **proiect Maven**, se bifează opțiunea „**Create from archetype**” și se selectează: **org.jetbrains.kotlin:kotlin-archetype-jvm** → **kotlin-archetype-jvm:1.3.72**. Se adaugă apoi următoarea dependență în **pom.xml**:

```
<dependency>
    <groupId>org.apache.spark</groupId>
    <artifactId>spark-core_2.12</artifactId>
    <version>2.4.5</version>
</dependency>
```

Este nevoie de crearea unui obiect **JavaSparkContext** care îi spune **framework-ului** Spark cum să acceseze un cluster.

Observație: Pentru simplitate, se va configura Spark să fie executat local. Pentru o imagine de ansamblu asupra configurării Spark, vezi:

- <https://spark.apache.org/docs/2.4.5/submitting-applications.html>

Spark SQL

Spark SQL este un modul al *framework-ului* Spark pentru procesarea datelor structurate. O utilizare a Spark SQL este pentru a executa interogări SQL. Rezultatele returnate în urma unei interogări vor fi *Dataset-uri* sau *DataFrame-uri*.

Dataset-ul este o colecție distribuită de date, ce are atât avantajele unui RDD (funcții lambda) cât și pe cele ale motorului de execuție optimizat al Spark SQL.

Un *DataFrame* este un *Dataset* organizat în coloane cu denumiri. Conceptual, este echivalent cu o tabelă dintr-o bază de date relațională, dar cu mai multe optimizări.

Pentru mai multe detalii, vezi:

- <https://spark.apache.org/docs/2.4.5/sql-getting-started.html>

De asemenea, vezi funcțiile de agregare pentru DataFrame-uri:

- [https://spark.apache.org/docs/2.4.5/api/scala/index.html#org.apache.spark.sql.functions\\$](https://spark.apache.org/docs/2.4.5/api/scala/index.html#org.apache.spark.sql.functions$)

Adăugarea componentei Spark SQL

Pentru a include componența Spark SQL în proiect, este necesară adăugarea următoarei dependențe în fișierul *pom.xml*:

```
<dependency>
    <groupId>org.apache.spark</groupId>
    <artifactId>spark-sql_2.12</artifactId>
    <version>2.4.5</version>
</dependency>
```

Securitate

Securitatea este dezactivată în mod implicit în Spark. Acest lucru înseamnă că, în mod implicit, aplicația este vulnerabilă la atacuri.

Framework-ul Spark poate fi securizat prin următoarele aspecte:

- Spark RPC (protocolul de comunicare între procese Spark)
- Criptarea spațiului de stocare local (*local storage encryption*)
- Web UI
- Configurarea porturilor pentru securitatea rețelei
- Kerberos
- Jurnal de evenimente (*event logging*)

```
val sparkConf = SparkConf().setMaster("local[4]")
    .setAppName("Spark Example")
    .set("spark.io.encryption.enabled", "true")
    .set("spark.io.encryption.keySizeBits", "256")
val sparkContext = JavaSparkContext(sparkConf)
```

Pentru mai multe detalii privind opțiunile de securizare ale unei aplicații Spark, precum și modul în care pot fi adăugate aceste configurații, vezi:

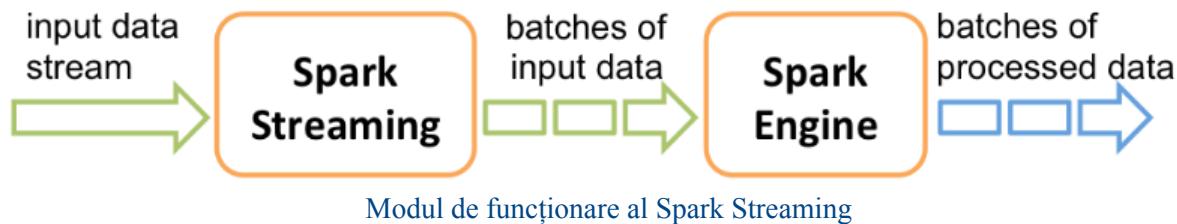
- <https://spark.apache.org/docs/latest/configuration.html>
- <https://spark.apache.org/docs/latest/security.html>

Pentru securizarea Apache Kafka, vezi documentația oficială, capitolul 7:

- <https://kafka.apache.org/documentation/#security>

Spark Streaming

Spark Streaming primește fluxuri de date în timp real, le împarte în loturi (eng. *batches*) care sunt procesate apoi de motorul Spark (*Spark Engine*) pentru a genera fluxul final de rezultate, tot sub formă de loturi (*batches*).



Spark Streaming oferă o abstractizare de nivel înalt numită *flux discret (discretized stream)* sau *DStream*, ce reprezintă un flux continuu de date. *DStream*-urile pot fi create fie din fluxuri de date de intrare din surse precum *Kafka*, *Flume* și *Kinesis*, fie aplicând operațiuni de nivel înalt pe alte *DStream*-uri. Un *DStream* este reprezentat intern ca o secvență de RDD (abstractizarea Spark-ului pentru un set de date imutabil distribuit).

ATENȚIE: Un program *Spark Streaming* executat local trebuie să utilizeze în URL-ul master “`localhost[n]`”, unde $n \geq 2$, deoarece un *thread* va executa *receiver*-ul, iar restul vor procesa datele primite.

Există mai multe surse de date pentru *Spark Streaming*:

- TCP:

```
val lines: JavaReceiverInputDStream<String> =
streamingContext.socketTextStream("localhost", 9999)
```

- fișiere text:

```
streamingContext.textFileStream(dataDirectory)
```

- flux de fișiere:

```
streamingContext.fileStream<KeyClass, ValueClass,
InputFormatClass>(dataDirectory)
```

- surse avansate: Kafka, Flume, Kinesis

Pentru mai multe detalii, vezi:

- <https://spark.apache.org/docs/2.4.5/streaming-programming-guide.html> (**în special transformările pe DStream-uri și operațiile de output pe DStream-uri**)
- <https://spark.apache.org/docs/2.4.5/streaming-custom-receivers.html>

Addăugarea componentei Spark Streaming

Pentru a include componenta *Spark Streaming* în proiect, este necesară adăugarea următoarei dependențe în fișierul *pom.xml*:

```
<dependency>
    <groupId>org.apache.spark</groupId>
    <artifactId>spark-streaming_2.12</artifactId>
    <version>2.4.5</version>
</dependency>
```

Integrarea Spark Streaming cu Kafka

Se va adăuga următoarea dependență în fișierul *pom.xml*:

```
<dependency>
    <groupId>org.apache.spark</groupId>
    <artifactId>spark-streaming-kafka-0-10_2.12</artifactId>
    <version>2.4.5</version>
</dependency>
```

Pentru mai multe detalii, vezi:

- <https://spark.apache.org/docs/2.4.5/streaming-kafka-0-10-integration.html>

Exemple

Exemplul 1: Spark RDD

Vezi comentariile cu explicații din cod.

```
package com.sd.laborator

import org.apache.spark.SparkConf
import org.apache.spark.api.java.JavaSparkContext
import org.apache.spark.api.java.function.Function
import org.apache.spark.api.java.function.Function2
import org.apache.spark.broadcast.Broadcast
import org.apache.spark.storage.StorageLevel

internal class GetLength : Function<String?, Int?> {
    override fun call(p0: String?): Int? {
        return p0?.length ?: 0
    }
}

internal class Sum : Function2<Int?, Int?, Int?> {
    override fun call(p0: Int?, p1: Int?): Int? {
        return (p0?.toInt() ?: 0) + (p1?.toInt() ?: 0)
    }
}

fun main(args: Array<String>) {
    // configurarea Spark
    val sparkConf = SparkConf().setMaster("local").setAppName("Spark
Example")
    // initializarea contextului Spark
    val sparkContext = JavaSparkContext(sparkConf)

    val items = listOf("123/643/7563/2134/ALPHA",
"2343/6356/BETA/2342/12", "23423/656/343")
    // paralelizarea colectiilor
    val distributedDataset = sparkContext.parallelize(items)

    // 1) spargerea fiecarui string din lista intr-o lista de
substring-uri si reunirea intr-o singura lista
    // 2) filtrarea cu regex pentru a pastra doar numerele
```

Sisteme Distribuite - Laborator 13

```
// 3) conversia string-urilor filtrate la int prin functia de
mapare
// 4) sumarea tuturor numerelor prin functia de reducere
val sumOfNumbers = distributedDataset.flatMap {
it.split("/").iterator() }
    .filter { it.matches(Regex("[0-9]+")) }
    .map { it.toInt() }
    .reduce {total, next -> total + next }
println(sumOfNumbers)

// seturi de date externe
// setul de date nu este inca incarcat in memorie (si nu se
actioneaza inca asupra lui)
val lines = sparkContext.textFile("src/main/resources/data.txt")

// pentru utilizarea unui RDD de mai multe ori, trebuie apelata
metoda persist:
lines.persist(StorageLevel.MEMORY_ONLY())

// functia de mapare reprezinta o transformare a setului de date
initial (nu este calculat imediat)
// abia cand se ajunge la functia de reducere (care este o
actiune) Spark imparte operatiile in task-uri
// pentru a fi rulate pe masini separate (fiecare masina executand
o parte din map si reduce)
// exemplu cu functii lambda:
val totalLength0 = lines.map { s->s.length }.reduce { a: Int, b:
Int -> a + b }

// trimiterea unor functii catre Spark
val totalLength1= lines.map(object : Function<String?, Int?> {
    override fun call(p0: String?): Int? {
        return p0?.length ?: 0
    }
}).reduce(object : Function2<Int?, Int?, Int?> {
    override fun call(p0: Int?, p1: Int?): Int? {
        return (p0?.toInt() ?: 0) + (p1?.toInt() ?: 0)
    }
})

// sau daca scrierea functiilor inline sau a celor lambda este
greoarie, se pot utiliza clase
val totalLength2 = lines.map(GetLength()).reduce(Sum())
println(totalLength0)
println(totalLength1)
println(totalLength2)

// variabila partajata de tip broadcast
// trimiterea unui set de date ca input catre fiecare nod intr-o
maniera eficienta:
val broadcastVar: Broadcast<List<Int>> =
sparkContext.broadcast(listOf(1, 2, 3))
val totalLength3 = lines.map { s->s.length +
broadcastVar.value()[0] }.reduce { a: Int, b: Int -> a + b }
println(totalLength3)

//variabila partajata de tip acumulator
```

```

val accumulator = sparkContext.sc().longAccumulator()
sparkContext.parallelize(listOf(1, 2, 3, 4)).foreach { x ->
accumulator.add(x.toLong()) }
println(accumulator)

// oprirea contextului Spark
sparkContext.stop()
}

```

Atenție: înainte de execuția aplicației Spark RDD, se va crea fișierul **src/main/resources/data.txt** cu câteva linii de text oarecare

Exemplul 2.1: Spark SQL - conectare la bază de date MySQL

În cazul în care nu aveți un server MySQL instalat și configurat local, puteți urma pașii de instalare disponibili la următorul URL: <https://linuxize.com/post/how-to-install-mysql-on-debian-10/>.

Pentru conectarea la o bază de date MySQL este necesară adăugarea următoarei dependențe în fișierul *pom.xml*:

```

<dependency>
  <groupId>mysql</groupId>
  <artifactId>mysql-connector-java</artifactId>
  <version>8.0.19</version>
</dependency>

```

Accesați consola MySQL astfel:

```
mysql -u root -p
```

Se va introduce parola pentru utilizatorul **root** al server-ului MySQL, configurată în pașii de instalare.

Creați un utilizator nou pentru Spark, din consola MySQL deschisă anterior:

```

CREATE USER 'spark'@'localhost' IDENTIFIED BY 'sparksq1';
GRANT ALL PRIVILEGES ON *.* TO 'spark'@'localhost' WITH GRANT OPTION;
CREATE USER 'spark'@'%' IDENTIFIED BY 'sparksq1';
GRANT ALL PRIVILEGES ON *.* TO 'spark'@'%' WITH GRANT OPTION;

```

Închideți consola MySQL **root** (folosind comanda **exit**) și conectați-vă cu noul utilizator „**spark**”:

```
mysql -u spark -p
```

Se va introduce parola „**sparksq1**”.

Pentru crearea unei baze de date și a unei tabele **Person**, precum și pentru popularea acesteia, se va executa următorul script în consola MySQL:

```

CREATE DATABASE IF NOT EXISTS sd_database;
USE sd_database;
CREATE TABLE IF NOT EXISTS Person (ID int PRIMARY KEY, LastName
varchar(255), FirstName varchar(255), Age int);
INSERT INTO Person(ID, LastName, FirstName, Age) VALUES(1, "Popescu",
"Ion", 20);
INSERT INTO Person(ID, LastName, FirstName, Age) VALUES(2, "Ionescu",
"Mariana", 20);
INSERT INTO Person(ID, LastName, FirstName, Age) VALUES(3, "Codreanu",

```

```

"Cristian", 20);
INSERT INTO Person(ID, LastName, FirstName, Age) VALUES(4, "Cantemir",
"Roxana", 21);
INSERT INTO Person(ID, LastName, FirstName, Age) VALUES(5, "Anghel",
"Vlad", 21);
INSERT INTO Person(ID, LastName, FirstName, Age) VALUES(6, "Bujor",
"Florina", 22);
COMMIT;

```

Observație: se poate pune scriptul într-un fișier **mysql_init.sql**, apoi dintr-un terminal deschis în directorul scriptului se pornește consola MySQL și se execută comanda:

```
source mysql_init.sql
```

Codul sursă

Vezi comentariile cu explicații din cod.

```

package com.sd.laborator

import org.apache.spark.sql.SparkSession

fun main(args: Array<String>) {
    val spark = SparkSession.builder()
        .appName("Java Spark SQL example")
        .config("spark.master", "local[4]")
        .orCreate
    // Crearea unui DataFrame pe baza unei tabele "Person" stocate
    // într-o bază de date MySQL
    //val url =
    "jdbc:mysql://yourIP:yourPort/databaseName?user=yourUsername&password=
    yourPassword&serverTimezone=UTC"
    val url =
    "jdbc:mysql://localhost:3306/sd_database?user=spark&password=sparksql&
    serverTimezone=UTC"
    val df = spark.sqlContext()
        .read()
        .format("jdbc")
        .option("url", url)
        .option("dbtable", "Person")
        .load()

    // Afisarea schemei DataFrame-ului
    df.printSchema()

    // Numararea persoanelor după varsta
    val countsByAge = df.groupBy("age").count()
    countsByAge.show()

    // Salvarea countsByAge în src/main/resources/sql_output în format
    // JSON

    countsByAge.write().format("json").save("src/main/resources/sql_out-
    put")
}

```

Înlocuiți „**admin**” cu „**root**” și „**pass**” cu parola setată în pașii de configurare a server-ului MySQL.

Exemplul 2.2: Spark SQL

```

package com.sd.laborator

import org.apache.spark.api.java.JavaRDD
import org.apache.spark.api.java.function.MapFunction
import org.apache.spark.sql._
import org.apache.spark.sql.functions.col
import java.io.Serializable

// clasa bean
class Person: Serializable {
    var name: String? = null
    var age = 0
}

fun main(args: Array<String>) {
    // configurarea si crearea sesiunii Spark SQL
    val sparkSession = SparkSession.builder()
        .appName("Java Spark SQL example")
        .config("spark.master", "local")
        .orCreate

    // initializarea unui DataFrame prin citirea unui json
    val df: Dataset<Row> =
        sparkSession.sqlContext().read().json("src/main/resources/people.json")

    // afisarea continutului din DataFrame la consola
    df.show()

    // Afisarea schemei DataFrame-ului intr-o forma arborescenta
    df.printSchema();

    // Selectarea coloanei nume si afisarea acestia
    df.select("name").show();

    // Selectarea tuturor datelor si incrementarea varstei cu 1
    df.select(col("name"), col("age").plus(1)).show();

    // Selectarea persoanelor cu varsta > 21 ani
    df.filter(col("age").gt(21)).show();

    // Numararea persoanelor dupa varsta
    df.groupBy("age").count().show();

    // Inregistarea unui DataFrame ca un SQL View temporar
    df.createOrReplaceTempView("people")

    // Utilizarea unei interogari SQL pentru a selecta datele
    val sqlDF: Dataset<Row> = sparkSession.sql("SELECT * FROM people")
    sqlDF.show()

    // Inregistrarea unui DataFrame ca un SQL View global temporar
    df.createGlobalTempView("people");
}

```

Sisteme Distribuite - Laborator 13

```
// Un SQL View global temporar este legat de o baza de date a sistemului: `global_temp`
sparkSession.sql("SELECT * FROM global_temp.people").show();

// Un view global temporar este vizibil intre sesiuni
sparkSession.newSession().sql("SELECT * FROM global_temp.people").show();

// Crearea seturilor de date (Dataset)
val person0 = Person() // instanta de clasa Bean
person0.name = "Mihai"
person0.age = 20
val person1 = Person() // instanta de clasa Bean
person1.name = "Ana"
person1.age = 19

// Este creat un codificator (Encoder) pentru bean-ul Java
val personEncoder: Encoder[Person] =
Encoders.bean(Person::class.java)
// Se creeaza Dataset-ul de persoane
val javaBeanDS: Dataset<Person> =
sparkSession.createDataset(listOf(person0, person1), personEncoder)
javaBeanDS.show()

// Codificatoarele (Encoders) pentru tipurile primitive sunt furnizate de clasa Encoders
val integerEncoder = Encoders.INT()
val primitiveDS: Dataset<Int> =
sparkSession.createDataset(listOf(1, 2, 3), integerEncoder)
val transformedDS = primitiveDS.map(
    MapFunction { value: Int -> value + 1 } as MapFunction<Int,
Int>,
    integerEncoder
)
transformedDS.collect() // [2, 3, 4]
transformedDS.show() // afisarea listei transformate

// DataFrame-urile pot fi convertite intr-un Dataset
val path = "src/main/resources/people.json"
val peopleDS: Dataset<Person> =
sparkSession.read().json(path).`as`(personEncoder)
peopleDS.show()

// Interoperabilitatea cu RDD-uri
// Crearea unui RDD de obiecte Person dintr-un fisier text
val peopleRDD: JavaRDD<Person> = sparkSession.read()
    .textFile("src/main/resources/people.txt")
    .javaRDD()
    .map { line ->
        val parts: List<String> = line.split(",")
        val person = Person()
        person.name = parts[0]
        person.age = parts[1].trim { it <= ' ' }.toInt()
        person // return
    }
    // Aplicarea unei scheme pe un RDD de Java Bean-uri pentru a obtine DataFrame
```

```

    val peopleDF: Dataset<Row> =
sparkSession.createDataFrame(peopleRDD, Person::class.java)
    // Înregistrarea DataFrame-ului ca un view temporar
    peopleDF.createOrReplaceTempView("people")

    // Selectarea persoanelor intre 13 si 19 ani cu o interogare SQL
    val teenagersDF: Dataset<Row> = sparkSession.sql("SELECT name FROM
people WHERE age BETWEEN 13 AND 19")

    // Coloanele dintr-un Row din rezultat pot fi accesate dupa
indexul coloanei
    val stringEncoder = Encoders.STRING()
    val teenagerNamesByIndexDF = teenagersDF.map(
        MapFunction { row: Row -> "Name: " + row.getString(0) } as
MapFunction<Row, String>,
        stringEncoder
    )
    teenagerNamesByIndexDF.show()

    // sau dupa numele coloanei
    val teenagerNamesByFieldDF = teenagersDF.map(
        MapFunction { row: Row -> "Name: " + row.getAs("name") } ,
        stringEncoder
    )
    teenagerNamesByFieldDF.show()
}

```

Înainte de a executa aplicația, adăugați următoarele fișiere în proiect:

- **src/main/resources/people.json**

```
{
  "name": "Ionel"
}
{
  "name": "Maria", "age": 23
}
{
  "name": "Vasile", "age": 18
}
{
  "name": "Ana", "age": 23
}
```

- **src/main/resources/people.txt**

```

Ionel, 17
Maria, 23
Vasile, 18
Ana, 23

```

Exemplul 3: Integrarea Spark Streaming cu Kafka

```

package com.sd.laborator

import org.apache.kafka.clients.consumer.ConsumerRecord
import org.apache.kafka.common.serialization.StringDeserializer
import org.apache.spark.SparkConf
import org.apache.spark.TaskContext
import org.apache.spark.api.java.JavaRDD
import org.apache.spark.api.java.JavaSparkContext
import org.apache.spark.streaming.Durations
import org.apache.spark.streaming.api.java.JavaInputDStream
import org.apache.spark.streaming.api.java.JavaStreamingContext
import org.apache.spark.streaming.kafka010.*
import scala.Tuple2

```

```

fun main() {
    // configurarea Kafka
    val kafkaParams = mutableMapOf<String, Any>(
        "bootstrap.servers" to "localhost:9092",
        "key.deserializer" to StringDeserializer::class.java,
        "value.deserializer" to StringDeserializer::class.java,
        "group.id" to "use_a_separate_group_id_for_each_stream",
        "auto.offset.reset" to "latest",
        "enable.auto.commit" to false
    )

    // configurarea Spark
    val sparkConf =
    SparkConf().setMaster("local[4]").setAppName("KafkaIntegration")

    // initializarea contextului de streaming
    val streamingContext = JavaStreamingContext(sparkConf,
Durations.seconds(1))

    val topics = listOf("topicA", "topicB")

    // crearea unui flux de date direct (DirectStream)
    val stream: JavaInputDStream<ConsumerRecord<String, String>> =
KafkaUtils.createDirectStream(
    streamingContext,
    LocationStrategies.PreferConsistent(),
    ConsumerStrategies.Subscribe(topics, kafkaParams)
)
    stream.mapToPair{record: ConsumerRecord<String, String> ->
Tuple2(record.key(), record.value()) }

    // crearea unui RDD (set de date imutabil distribuit)
    val offsetRanges =
        arrayOf( /* topicul, partitia, offset-ul de inceput, offset-ul
final */
            OffsetRange.create("test", 0, 0, 100),
            OffsetRange.create("test", 1, 0, 100)
        )
    val rdd: JavaRDD<ConsumerRecord<String, String>> =
KafkaUtils.createRDD(
    streamingContext.sparkContext(),
    kafkaParams,
    offsetRanges,
    LocationStrategies.PreferConsistent()
)

    // obtinerea offset-urilor
    stream.foreachRDD { rdd ->
        val offsetRanges = (rdd.rdd() as
HasOffsetRanges).offsetRanges()
        rdd.foreachPartition { consumerRecords ->
            val o = offsetRanges.get(TaskContext.get().partitionId())
            println(
                o.topic() + " " + o.partition() + " " + o.fromOffset()
+ " " + o.untilOffset()
            )
        }
    }
}

```

```

    }

    // stocarea offset-urilor
    stream.foreachRDD { rdd =>
        val offsetRanges = (rdd.rdd() as HasOffsetRanges).offsetRanges()
        (stream.inputDStream() as
        CanCommitOffsets).commitAsync(offsetRanges)
    }

    streamingContext.start()
    streamingContext.awaitTerminationOrTimeout(1000)
}

```

Aplicații și teme

Aplicații de laborator

1. Consultând documentația oficială, reluați exemplul 1 și configurați Spark să fie executat local pe câte *core*-uri sunt disponibile și cu un număr de maxim 6 încercări de execuție a unei task (RDD).
2. Să se calculeze cu ajutorul Spark RDD, Spark SQL și Spark Streaming (deci 3 aplicații diferite) histograma caracterelor din alfabetul latin ([a-zA-Z]) dintr-un fișier text de tip *ebook*.
Observație: când se utilizează Spark Streaming, fișierele text trebuie create în timp real (în timpul execuției aplicației).
3. Să se implementeze un **Kafka Producer** în Python care să citească un fișier text, să extragă cuvintele și să le publice într-un topic. Să se creeze un direct stream folosind Spark Streaming și din histograma cuvintelor, să se returneze cele mai frecvente 15 cuvinte.

Observație: nu uitați să porniți server-ul Kafka (a se consulta laboratorul 10 pentru detalii).

Teme pe acasă

- Să se implementeze un Kafka Producer în Python care să preia coordonatele mouse-ului timp de 10 secunde și să le publice într-un topic (mouse-ul va trebui „plimbăt” pe diagonala ecranului). Pornind de la exemplul 3, să se preia fluxul de puncte utilizând Spark Streaming și să se realizeze o regresie liniară, calculând covarianța.

Pentru preluarea coordonatelor mouse-ului în Python, se poate utiliza scriptul de mai jos:

```

from pynput.mouse import Controller
import time

mouse = Controller()

if __name__ == "__main__":
    for _ in range(100):
        print(mouse.position)
        time.sleep(0.1)

```

Trebuie instalată dependența **pynput**:

```

python3 -m venv env
source env/bin/activate
pip3 install pynput==1.6.8

```

Laborator 14: Django, Apache Spark & Kafka

Introducere

Pentru o imagine de ansamblu asupra framework-ului Django, vezi:

- <https://docs.djangoproject.com/en/3.0/>
- <https://buildmedia.readthedocs.org/media/pdf/django/latest/django.pdf>
- „Django 3 by example” - Antonio Melé
- „Django 3 web development cookbook” (4th edition) - Aidas Bendoraitis și Jake Kronika

Pentru documentația framework-ului Apache Spark în Python, vezi:

- <https://spark.apache.org/docs/2.4.5/api/python/>
- <https://spark.apache.org/docs/2.4.5/rdd-programming-guide.html>
- <https://spark.apache.org/docs/2.4.5/submitting-applications.html>
- <https://spark.apache.org/docs/2.4.5/sql-getting-started.html>
- <https://spark.apache.org/docs/2.4.5/streaming-programming-guide.html> (în special transformările pe DStream-uri și operațiile de output pe DStream-uri)

Exemple

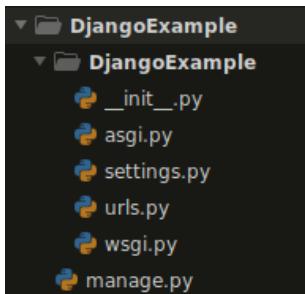
Instalare Django și PySpark:

```
python3 -m venv env
source env/bin/activate
pip3 install Django==3.0.6 django-taggit==1.3.0 pyspark
```

Crearea proiectului Django:

```
django-admin startproject DjangoExample
```

ATENȚIE: Pentru a evita conflictele, proiectele nu se denumesc după module Python built-in sau module Django.



Structura proiectului creat

- **manage.py** (wrapper peste *django-admin.py*) - utilitar de tip linie de comandă folosit pentru a interacționa cu proiectul
- DjangoExample/ - directorul proiectului, care conține următoarele fișiere:
 - › **__init__.py** - fișier gol care marchează faptul că Python va trata acest director ca un modul Python
 - › **asgi.py** - configurarea pentru execuția proiectului ca ASGI¹
 - › **settings.py** - setări și configurații pentru proiect (conține și câteva setări implicite)

¹ ASGI = Asynchronous Server Gateway Interface

- › **urls.py** - maparea URL-urilor cu view-uri
- › **wsgi.py** - configurarea pentru execuția proiectului ca WSGI²

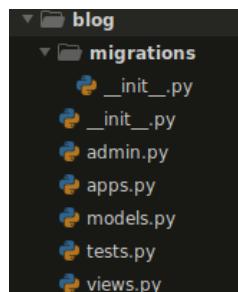
Setările proiectului - settings.py

- DEBUG - variabilă booleană care activează/dezactivează modul de debug. Dacă este setată pe True, Django va afișa pagini de erori detaliate când apare o excepție și nu este tratată. În producție, trebuie dezactivat modul de debug, altfel sunt expuse date sensibile legate de proiect.
- ALLOWED_HOSTS - nu se aplică când modul de debug este activ, sau când se execută teste. În producție, după ce se dezactivează modul de debug, trebuie adăugat domeniul în această listă.
- INSTALLED_APPS - setare care trebuie modificată pentru toate proiectele. Această setare precizează ce aplicații sunt active pentru acest site. Implicit, Django include următoarele aplicații:
 - › django.contrib.admin - un site de administrare
 - › django.contrib.auth - un framework de autentificare
 - › django.contrib.contenttypes - un framework pentru gestionarea tipurilor de conținut
 - › django.contrib.sessions - un framework de sesiune
 - › django.contrib.messages - un framework de mesaje
 - › django.contrib.staticfiles - un framework pentru gestionarea fișierelor statice
- MIDDLEWARE - o listă care conține middleware-urile care vor fi executate
- ROOT_URLCONF - indică locația fișierului urls.py (în cazul proiectului curent: DjangoExample.urls)
- DATABASES - dicționar care conține setările pentru toate bazele de date utilizate în proiect. Întotdeauna există o setare default în acest dicționar. Configurarea implicită utilizează o bază de date SQLite3.
- TIME_ZONE = 'Europe/Bucharest'
- USE_TZ - activează/dezactivează suportul pentru timezone

Crearea unei aplicații de tip blog

Într-un terminal deschis în directorul DjangoExample (în care se află fișierul *manage.py*) se va executa comanda:

```
python3 manage.py startapp blog
```



Structura aplicației blog

- admin.py - pentru înregistrarea modelelor pentru a fi incluse în site-ul de administrarea Django.

² WSGI = Web Server Gateway Interface

- apps.py - configurația principală pentru aplicația blog
- migrations - folder care conține migrările bazei de date a aplicației. Migrările permit ca Django să urmărească schimbările în model și să sincronizeze baza de date corespunzător.
- models.py - modelele de date ale aplicației; toate aplicațiile Django au acest fișier *models.py*, dar poate fi lăsat și gol.
- tests.py - teste pentru aplicație
- views.py - logica aplicației; fiecare view primește o cerere HTTP, o procesează și returnează răspunsul.

blog/models.py

Observație: *slug* este un câmp ce se intenționează a fi utilizat în URL-uri. Acesta este o etichetă scurtă care conține doar litere, numere, sau cratime.

```
from django.db import models
from django.utils import timezone
from django.urls import reverse
from django.contrib.auth.models import User

class PublishedManager(models.Manager):
    def get_queryset(self):
        return super().get_queryset().filter(status='published')


class Post(models.Model):
    STATUS_CHOICES = (
        ('draft', 'Draft'),
        ('published', 'Published'),
    )
    title = models.CharField(max_length=250)
    slug = models.SlugField(max_length=250, unique_for_date='publish')
    author = models.ForeignKey(User,
                               on_delete=models.CASCADE,
                               related_name='blog_posts')
    body = models.TextField()
    publish = models.DateTimeField(default=timezone.now)
    created = models.DateTimeField(auto_now_add=True)
    updated = models.DateTimeField(auto_now=True)
    status = models.CharField(max_length=10,
                             choices=STATUS_CHOICES,
                             default='draft')

    def set_body(self, body):
        self._body = body
        self._body_changed = True

    def get_body(self):
        return self._body

    body = property(get_body, set_body)

    objects = models.Manager() # The default manager.
    published = PublishedManager() # Our custom manager.
```

```

class Meta:
    ordering = ('-publish', )

def __str__(self):
    return self.title

def get_absolute_url(self):
    return reverse('blog:post_detail',
                  args=[self.publish.year, self.publish.month,
                         self.publish.day, self.slug
                     ])

def save(self, *args, **kwargs):
    if getattr(self, '_body_changed', True):
        # TODO - Kafka Producer
        pass
    super(Post, self).save(*args, **kwargs)

```

Activarea aplicației (DjangoExample/settings.py):

```

# ...
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'blog.apps.BlogConfig',
]
# ...

```

blog/templates/blog/post/detail.html

```

{% extends "blog/base.html" %}

{% block title %}{{ post.title }}{% endblock %}

{% block content %}
<h1>{{ post.title }}</h1>
<p class="date">
    Published {{ post.publish }} by {{ post.author }}
</p>
{{ post.body|linebreaks }}
{% endblock %}

```

blog/templates/blog/post/list.html

```

{% extends "blog/base.html" %}

{% block title %}My Blog{% endblock %}

{% block content %}
<h1>My Blog</h1>

```

```

{% for post in posts %}
    <h2>
        <a href="{{ post.get_absolute_url }}">
            {{ post.title }}
        </a>
    </h2>
    <p class="date">
        Published {{ post.publish }} by {{ post.author }}
    </p>
    {{ post.body|truncatewords:30|linebreaks }}
    {% endfor %}
    {% include "pagination.html" with page=page_obj %}
{% endblock %}

```

blog/templates/blog/base.html

```

{% load static %}
<!DOCTYPE html>
<html>
<head>
    <title>{% block title %}{% endblock %}</title>
    <link href="{% static "css/blog.css" %}" rel="stylesheet">
</head>
<body>
    <div id="content">
        {% block content %}
        {% endblock %}
    </div>
    <div id="sidebar">
        <h2>My blog</h2>
        <p>This is my blog.</p>
    </div>
</body>
</html>

```

blog/templates/pagination.html

```

<div class="pagination">
    <span class="step-links">
        {% if page.has_previous %}
            <a href="?page={{ page.previous_page_number }}">Previous</a>
        {% endif %}
        <span class="current">
            Page {{ page.number }} of {{ page.paginator.num_pages }}.
        </span>
        {% if page.has_next %}
            <a href="?page={{ page.next_page_number }}">Next</a>
        {% endif %}
    </span>
</div>

```

blog/static/css/blog.css

```

body {
    margin:0;
}

```

```
padding:0;
font-family:helvetica, sans-serif;
}

a {
    color:#00abff;
    text-decoration:none;
}

h1 {
    font-weight:normal;
    border-bottom:1px solid #bbb;
    padding:0 0 10px 0;
}

h2 {
    font-weight:normal;
    margin:30px 0 0;
}

#content {
    float:left;
    width:60%;
    padding:0 0 0 30px;
}

#sidebar {
    float:right;
    width:30%;
    padding:10px;
    background:#efefef;
    height:100%;
}

p.date {
    color:#ccc;
    font-family: georgia, serif;
    font-size: 12px;
    font-style: italic;
}

/* pagination */
.pagination {
    margin:40px 0;
    font-weight:bold;
}

/* forms */
label {
    float:left;
    clear:both;
    color:#333;
    margin-bottom:4px;
}
input, textarea {
    clear:both;
```

```

float:left;
margin:0 0 10px;
background:#ededed;
border:0;
padding:6px 10px;
font-size:12px;
}
input[type=submit] {
    font-weight:bold;
background:#00abff;
color:#fff;
padding:10px 20px;
font-size:14px;
text-transform:uppercase;
}
.errorlist {
    color:#cc0033;
    float:left;
    clear:both;
    padding-left:10px;
}

/* comments */
.comment {
    padding:10px;
}
.comment:nth-child(even) {
    background:#efefef;
}
.comment .info {
    font-weight:bold;
    font-size:12px;
    color:#666;
}

```

[blog/views.py](#)

```

from django.shortcuts import render, get_object_or_404
from django.core.paginator import Paginator, EmptyPage,
PageNotAnInteger
from django.views.generic import ListView
from .models import Post


def post_list(request):
    object_list = Post.published.all()
    paginator = Paginator(object_list, 3) # 3 posts in each page
    page = request.GET.get('page')
    try:
        posts = paginator.page(page)
    except PageNotAnInteger:
        # If page is not an integer deliver the first page
        posts = paginator.page(1)
    except EmptyPage:
        # If page is out of range deliver last page of results
        posts = paginator.page(paginator.num_pages)

```

```

    return render(request, 'blog/post/list.html', {
        'page': page,
        'posts': posts
    })

def post_detail(request, year, month, day, post):
    post = get_object_or_404(Post,
                            slug=post,
                            status='published',
                            publish__year=year,
                            publish__month=month,
                            publish__day=day)
    return render(request, 'blog/post/detail.html', {'post': post})

class PostListView(ListView):
    queryset = Post.published.all()
    context_object_name = 'posts'
    paginate_by = 3
    template_name = 'blog/post/list.html'

```

blog/admin.py

```

from django.contrib import admin
from .models import Post

@admin.register(Post)
class PostAdmin(admin.ModelAdmin):
    list_display = ('title', 'slug', 'author', 'publish', 'status')
    list_filter = ('status', 'created', 'publish', 'author')
    search_fields = ('title', 'body')
    prepopulated_fields = {'slug': ('title', )}
    raw_id_fields = ('author', )
    date_hierarchy = 'publish'
    ordering = ('status', 'publish')

```

blog/urls.py

```

from django.urls import path
from . import views

app_name = 'blog'

urlpatterns = [
    # post views
    # path('', views.post_list, name='post_list'),
    path('', views.PostListView.as_view(), name='post_list'),
    path('<int:year>/<int:month>/<int:day>/<slug:post>/',
         views.post_detail,
         name='post_detail'),
]

```

DjangoExample/urls.py

```
from django.contrib import admin
from django.urls import path, include

urlpatterns = [
    path('admin/', admin.site.urls),
    path('blog/', include('blog.urls', namespace='blog')),
]
```

Crearea și aplicarea migrărilor

```
python3 manage.py makemigrations blog
python3 manage.py sqlmigrate blog 0001
python3 manage.py migrate
```

Crearea unui superuser pentru site-ul de administrare (deja inclus în proiect)

```
python3 manage.py createsuperuser

# Username (leave blank to use 'student'):
# Email address: student@ac.tuiasi.ro
# Password: studentpw
# Password (again): studentpw
# This password is too short. It must contain at least 8 characters.
# Bypass password validation and create user anyway? [y/N]: y
# Superuser created successfully.
```

Pornirea server-ului

Se execută în terminal comanda `python3 manage.py runserver` apoi se deschide în browser următorul URL: <http://127.0.0.1:8000/admin/>

După ce se adaugă câteva postări pe blog, se navighează la URL-ul <http://127.0.0.1:8000/blog/>

Exemplu de creare a unui flux de date direct cu Kafka în python

Se va instala pyspark cu comanda: `pip3 install pyspark`

Se va descărca `spark-streaming-kafka-0-8-assembly_2.11-2.4.5.jar` de la adresa: https://repo1.maven.org/maven2/org/apache/spark/spark-streaming-kafka-0-8-assembly_2.11/2.4.5/spark-streaming-kafka-0-8-assembly_2.11-2.4.5.jar în folder-ul proiectului.

```
from pyspark.streaming.context import StreamingContext
from pyspark.streaming.kafka import KafkaUtils
from pyspark.context import SparkContext
import os

if __name__ == '__main__':
    os.environ['PYSPARK_SUBMIT_ARGS'] = '--jars ./spark-streaming-kafka-0-8-assembly_2.11-2.4.5.jar pyspark-shell'
    sparkContext = SparkContext(master="local", batchSize=0)
    ssc = StreamingContext(sparkContext, batchDuration=1)

    kafkaParams = {"bootstrap.servers": "localhost:9092"}
```

```

kafkaStream = KafkaUtils.createDirectStream(ssc=ssc,
topics=["blog"], kafkaParams=kafkaParams)
kafkaStream.foreachRDD(lambda rdd: print(rdd.collect()))

ssc.start()
ssc.awaitTermination()

```

ATENȚIE: Calea specificată în variabila de mediu este una relativă la folder-ul în care se află scriptul de mai sus.

Exemplu de streaming pe fișiere text

```

from pyspark import SparkContext
from pyspark.streaming import StreamingContext
import os

if __name__ == '__main__':
    os.environ['PYSPARK_PYTHON'] = '/usr/bin/python3'
    sc = SparkContext("local", "Text File Streaming")
    ssc = StreamingContext(sc, 1)

    ROOT_DIR = os.path.abspath(os.path.dirname(__file__))
    path = "file:///{}" + os.path.join(ROOT_DIR, 'resources/text')
    stream = ssc.textFileStream(path)

    stream.foreachRDD(lambda rdd: print(rdd.collect()))

    ssc.start()
    ssc.awaitTermination()

```

Exemplu de Spark RDD

```

from pyspark import SparkContext, SparkConf, StorageLevel
import os
import re

if __name__ == '__main__':
    ''' configurare variabila de mediu cu versiunea python utilizata
    pentru spark '''
    os.environ['PYSPARK_PYTHON'] = '/usr/bin/python3'
    # configurarea Spark
    spark_conf = SparkConf().setMaster("local").setAppName("Spark
Example")
    # initializarea contextului Spark
    spark_context = SparkContext(conf=spark_conf)

    items = ["123/643/7563/2134/ALPHA", "2343/6356/BETA/2342/12",
    "23423/656/343"]
    # paralelizarea colectiilor
    distributed_dataset = spark_context.parallelize(items) # RDD

    ''' 1) spargerea fiecarui string din lista intr-o lista de
    substring-uri si reunirea intr-o singura lista
    2) filtrarea cu regex pentru a pastra doar numerele

```

```

3) conversia string-urilor filtrate la int prin functia de mapare
4) sumarea tuturor numerelor prin functia de reducere ''
    sum_of_numbers = distributed_dataset.flatMap(lambda item:
item.split("/"))\
        .filter(lambda item: re.match("[0-9]+", item))\
        .map(lambda item: int(item))\
        .reduce(lambda total, next_item: total + next_item)
    print("Sum of numbers =", sum_of_numbers)

''' seturi de date externe
    setul de date nu este inca incarcat in memorie (si nu se
actioneaza inca asupra lui) '''
    ROOT_DIR = os.path.abspath(os.path.dirname(__file__))
    path = "file:///+" + os.path.join(ROOT_DIR, 'resources/data.txt')
    lines = spark_context.textFile(path)

''' pentru utilizarea unui RDD de mai multe ori, trebuie apelata
metoda persist: '''
    lines.persist(StorageLevel.MEMORY_ONLY)

''' functia de mapare reprezinta o transformare a setului de date
initial (nu este calculat imediat)
    abia cand se ajunge la functia de reducere (care este o actiune)
Spark imparte operatiile in task-uri
    pentru a fi rulate pe masini separate (fiecare masina executand o
partea din map si reduce)
    exemplu cu functii lambda: '''
    total_length0 = lines.map(lambda s: len(s)).reduce(lambda acc, i:
acc + i)

    print("Total length =", total_length0)

''' variabila partajata de tip broadcast
    trimitera unui set de date ca input catre fiecare nod intr-o
maniera eficienta: '''
    broadcast_var = spark_context.broadcast([1, 2, 3])
    total_length1 = lines.map(lambda s: len(s) +
broadcast_var.value[0]).reduce(lambda acc, i: acc + i)
    print("Sum(line_length + broadcast_val[0])=", total_length1)
    # variabila partajata de tip accumulator
    accumulator = spark_context.accumulator(0)
    spark_context.parallelize([1, 2, 3, 4]).foreach(lambda x:
accumulator.add(x))
    print("Accumulator =", accumulator)

    # oprirea contextului Spark
    spark_context.stop()

```

Exemplu de Spark SQL

```

from pyspark.sql import SparkSession, Row
import os

if __name__ == '__main__':

```

```

''' configurare variabila de mediu cu versiunea python utilizata
pentru spark '''
os.environ['PYSPARK_PYTHON'] = '/usr/bin/python3'
# configurarea si crearea sesiunii Spark SQL
spark_session = SparkSession\
    .builder\
    .appName("Python Spark SQL example")\
    .config("spark.master", "local")\
    .getOrCreate()

# initializarea unui DataFrame prin citirea unui json
ROOT_DIR = os.path.abspath(os.path.dirname(__file__))
people_json_path = "file:///{}" + os.path.join(ROOT_DIR,
'resources/people.json')
df = spark_session.read.json(people_json_path)

# afisarea continutului din DataFrame la consola
df.show()

# Afisarea schemei DataFrame-ului intr-o forma arborescenta
df.printSchema()

# Selectarea coloanei nume si afisarea acesteia
df.select("name").show()

# Selectarea tuturor datelor si incrementarea varstei cu 1
df.select(df["name"], df["age"] + 1).show()

# Selectarea persoanelor cu varsta > 21 ani
df.filter(df["age"] > 21).show()

# Numararea persoanelor dupa varsta
df.groupBy("age").count().show()

# Inregistarea unui DataFrame ca un SQL View temporar
df.createOrReplaceTempView("people")

# Utilizarea unei interogari SQL pentru a selecta datele
sqlDF = spark_session.sql("SELECT * FROM people")
sqlDF.show()

# Inregistrarea unui DataFrame ca un SQL View global temporar
df.createGlobalTempView("people")

''' Un SQL View global temporar este legat de o baza de date a
sistemului: `global_temp` '''
spark_session.sql("SELECT * FROM global_temp.people").show()

# Un view global temporar este vizibil intre sesiuni
spark_session.newSession().sql("SELECT * FROM
global_temp.people").show()

# Interoperabilitatea cu RDD-uri
# Crearea unui RDD de obiecte Person dintr-un fisier text
people_txt_path = "file:///{}" + os.path.join(ROOT_DIR,
'resources/people.txt')

```

```
lines = spark_session.sparkContext.textFile(people_txt_path)
people = lines.map(lambda l: l.split(",")).map(lambda p:
Row(name=p[0], age=int(p[1])))

''' Aplicarea unei scheme pe un RDD de bean-uri pentru a obtine
DataFrame '''
peopleDF = spark_session.createDataFrame(people)
# Inregistrarea DataFrame-ului ca un view temporar
peopleDF.createOrReplaceTempView("people")

# Selectarea persoanelor intre 13 si 19 ani cu o interogare SQL
teenagersDF = spark_session.sql("SELECT name FROM people WHERE age
BETWEEN 13 AND 19")

teen_names = teenagersDF.rdd.map(lambda p: "Name: " +
p.name).collect()
for name in teen_names:
    print(name)
```

Aplicații și teme

Aplicații de laborator:

1. Pornind de la exemplul de mai sus, să se creeze un Kafka Producer și să se modifice aplicația astfel încât, pe lângă salvarea în baza de date SQLite a postărilor pe blog, să se trimită într-un topic postările respective serializate cu ajutorul modulului *json*.
2. Utilizând modulul django-taggit (trebuie instalat cu pip3) să se adauge un câmp *tags* în clasa Post (care extinde clasa models.Model) de tipul *TaggableManager*

Observație: aplicația *taggit* va fi inclusă în aplicațiile instalate, apoi se vor reface migrările.

Teme pe acasă:

1. Utilizând Spark Streaming și Spark RDD, să se creeze un stream direct prin intermediul KafkaUtils³ și să se realizeze o analiză (statistică) de sentimente pe baza a două fișiere text ce conțin cuvinte pozitive / negative (încărcate în aplicație ca RDD-uri). Practic, vor fi calculate și comparate două procentaje: $cuvinte_pozitive/nr_total_cuvinte * 100$, $cuvinte_negative/nr_total_cuvinte * 100$. Dacă procentajele sunt relativ apropriate (diferență maximă de 5%), postarea va fi considerată neutră. În caz contrar, va fi considerată pozitivă, sau negativă, în funcție de cel mai mare procentaj.
Observație: A fost atașat la laboratorul curent un set de cuvinte pozitive/negative⁴. Se pot utiliza și alte seturi, dacă se dorește. De asemenea, vezi și exemplul de creare a unui flux direct de date cu Kafka în python.
2. Să se creeze un Kafka Producer care să trimită într-un topic tag-urile calculate (pozitiv, neutru, negativ) pentru fiecare postare în parte. Un Kafka Consumer va prelua tag-urile din topic și va modifica baza de date SQLite astfel încât fiecare postare să aibă tag-ul asociat. La final, se va forța o reîncărcare a paginii (nu din cache) pentru a observa rezultatul.

[BONUS]: Utilizând algoritmul K-means din Spark MLlib⁵, să se grupeze postările pe blog astfel încât, în funcție de o postare selectată, să se determine toate postările similare. Pentru aceasta, se vor determina cele mai frecvente 5 cuvinte din fiecare postare (acestea fiind considerate cuvinte cheie), apoi se vor clusteriza postările de pe blog, rezultând 3 centroizi poziționați în funcție de distanța cosinus dintre vectorii de cuvinte cheie ai fiecărei postări.

Observație: Se va utiliza algoritmul TF-IDF disponibil în modulul scikit-learn⁶ pentru a calcula coeficienții necesari pentru calculul distanței cosinus.

³ <https://spark.apache.org/docs/2.4.5/api/python/pyspark.streaming.html#module-pyspark.streaming.kafka>

⁴ <http://www.cs.uic.edu/~liub/FBS/opinion-lexicon-English.rar>

⁵ <https://spark.apache.org/docs/latest/mllib-clustering.html>

⁶ https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.TfidfTransformer.html