

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
```

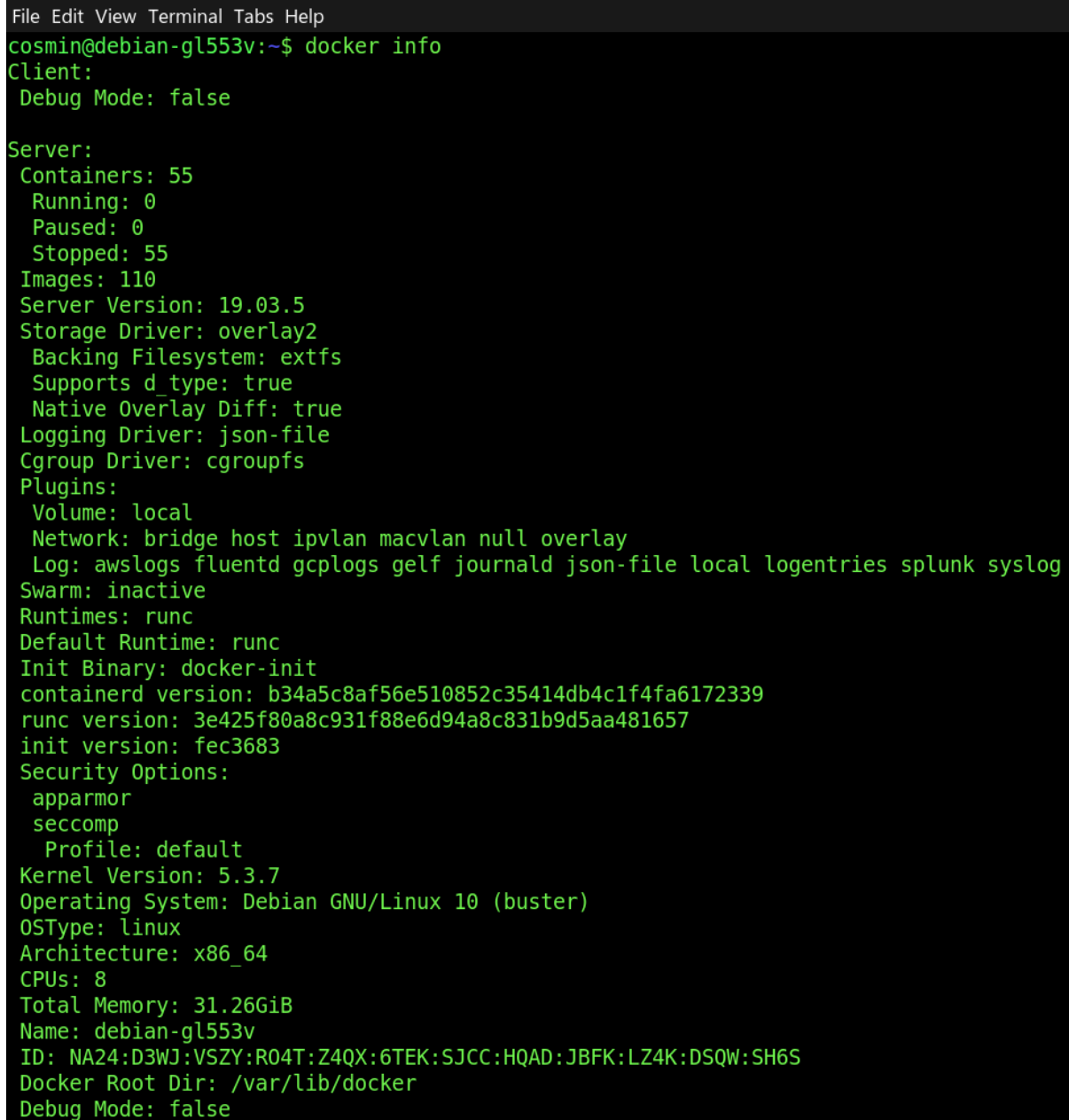
Adăugați utilizatorul neprivilegiat pe care sunteți conectat (pe care îl folosiți pentru lucrul cu Docker) în grupa creată mai sus:

```
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:

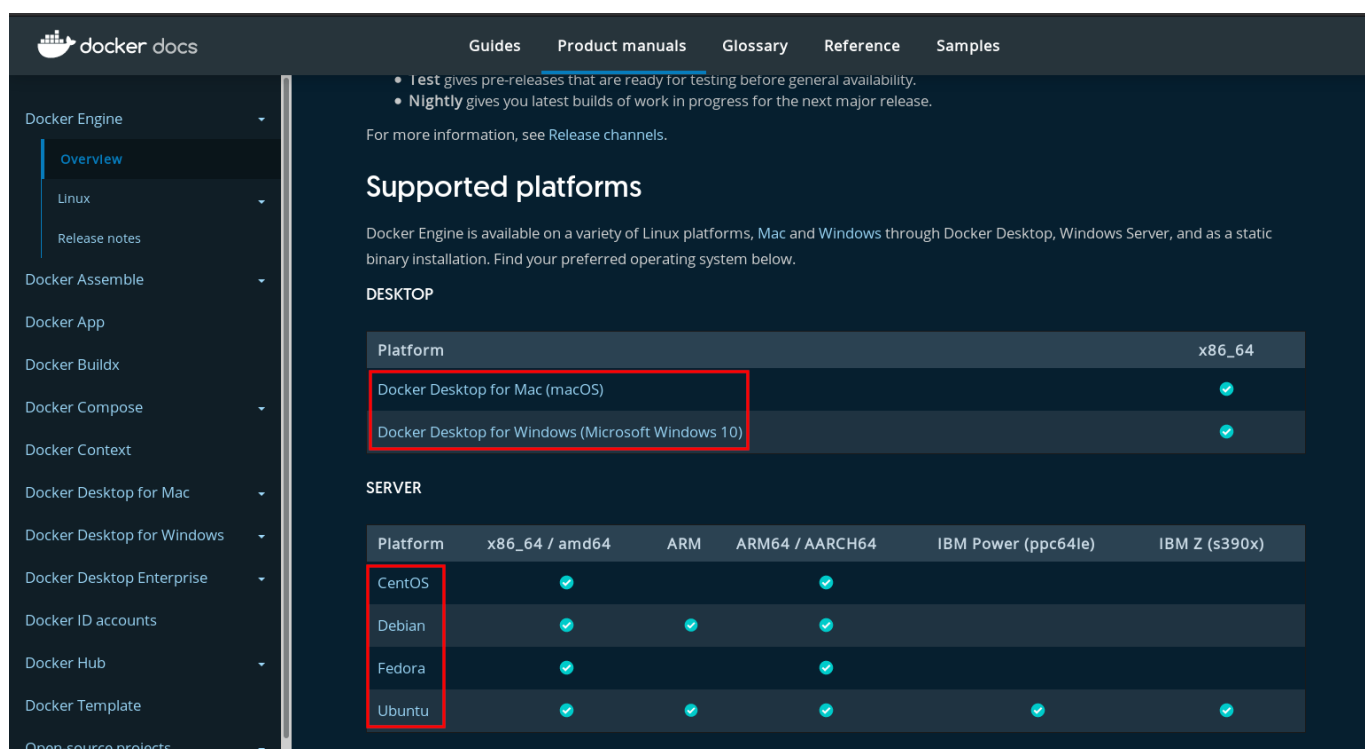
```
sudo nano /etc/docker/daemon.json
```

Copiați conținutul de mai sus, apoi apăsați, în ordine: CTRL+O, ENTER, CTRL+X.

Reporniți 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/>.



The screenshot shows the Docker Docs website with a sidebar on the left containing a navigation menu. The main content area is titled 'Supported platforms' and lists various operating systems and architectures. A red box highlights the 'Docker Desktop for Mac (macOS)' and 'Docker Desktop for Windows (Microsoft Windows 10)' entries under the 'DESKTOP' section. Another red box highlights the 'CentOS', 'Debian', 'Fedora', and 'Ubuntu' entries under the 'SERVER' section.

Platform	x86_64
Docker Desktop for Mac (macOS)	✓
Docker Desktop for Windows (Microsoft Windows 10)	✓

Platform	x86_64 / amd64	ARM	ARM64 / AARCH64	IBM Power (ppc64le)	IBM Z (s390x)
CentOS	✓		✓		
Debian	✓	✓	✓		
Fedora	✓		✓		
Ubuntu	✓	✓	✓	✓	✓

Figura 2 - Instrucțiuni de instalare pentru diverse platforme

Lucrul cu comenzi Docker - crearea unui registru local de imagini

Registrul (engl. **registry**) implicit utilizat de Docker Engine este cel oficial, găzduit de **Docker Hub**: docker.io/library. În registru, utilizatorii își pot crea așa 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 registrul 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
astronos2007/nginx	latest	243727b5c5af	3 months ago	152MB
astronos2007/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. Executați 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:~$
```

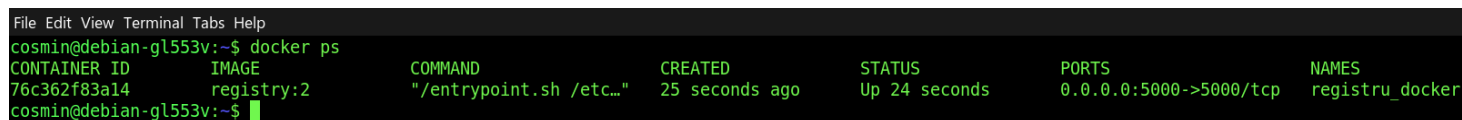
Figura 6 - Pornirea unui container

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:~$
```

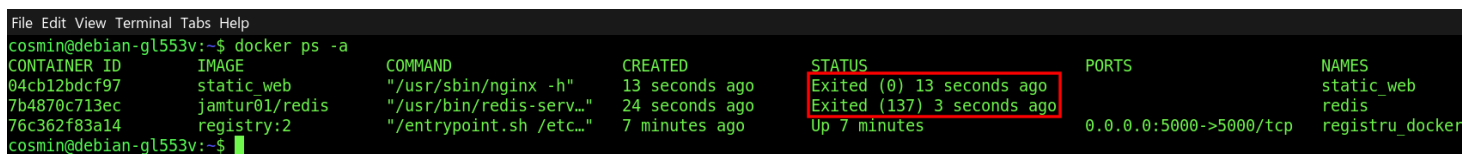
Figura 7 - Verificarea listei de containere pornite

Abrevierea „**ps**” provine de la „**process status**”.

Observați că apare listat containerul pornit în pașii anteriori (**registru_docker**). Comanda returnează **doar lista de containere în execuție**.

Pentru a afișa toate containerele disponibile, indiferent de starea lor (adică inclusiv cele oprite de utilizator, oprite forțat, sau oprite 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  
04cb12bdcf97        static web         "/usr/sbin/nginx -h" 13 seconds ago     Exited (0) 13 seconds ago                static_web  
7b4870c713ec        jamtur01/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 - Afișarea listei tuturor containerelor, indiferent de stare

Acum apar și containerele care au fost pornite la un moment dat, iar apoi au fost oprite de

utilizator sau din altă cauză.

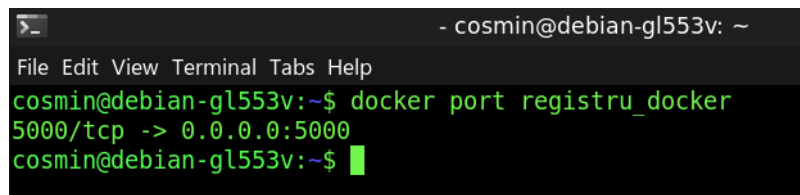
Verificarea porturilor expuse de un container Docker

Containerul **registru_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_IDENTIFICATOR_CONTAINER>
```

În acest caz:

```
docker port registru_docker
```



```

cosmin@debian-gl553v: ~$ docker port registru_docker
5000/tcp -> 0.0.0.0:5000
cosmin@debian-gl553v: ~$
  
```

Figura 9 - Verificarea porturilor expuse de un container Docker

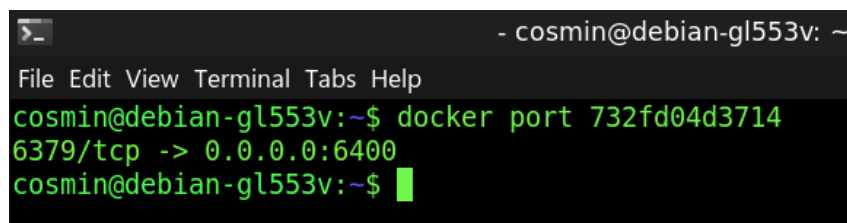
Comanda returnează: **5000/tcp** → **0.0.0.0:5000**

- 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 acceseze, 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: ~$ 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 acceseze 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 trimite 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, așadar trebuie parcurși 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_FUNCȚIA_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 împachetate, 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>-<VERSIONE_PROIECT>-jar-with-dependencies.jar

Consultați următoarea captură de ecran pentru un exemplu concret:

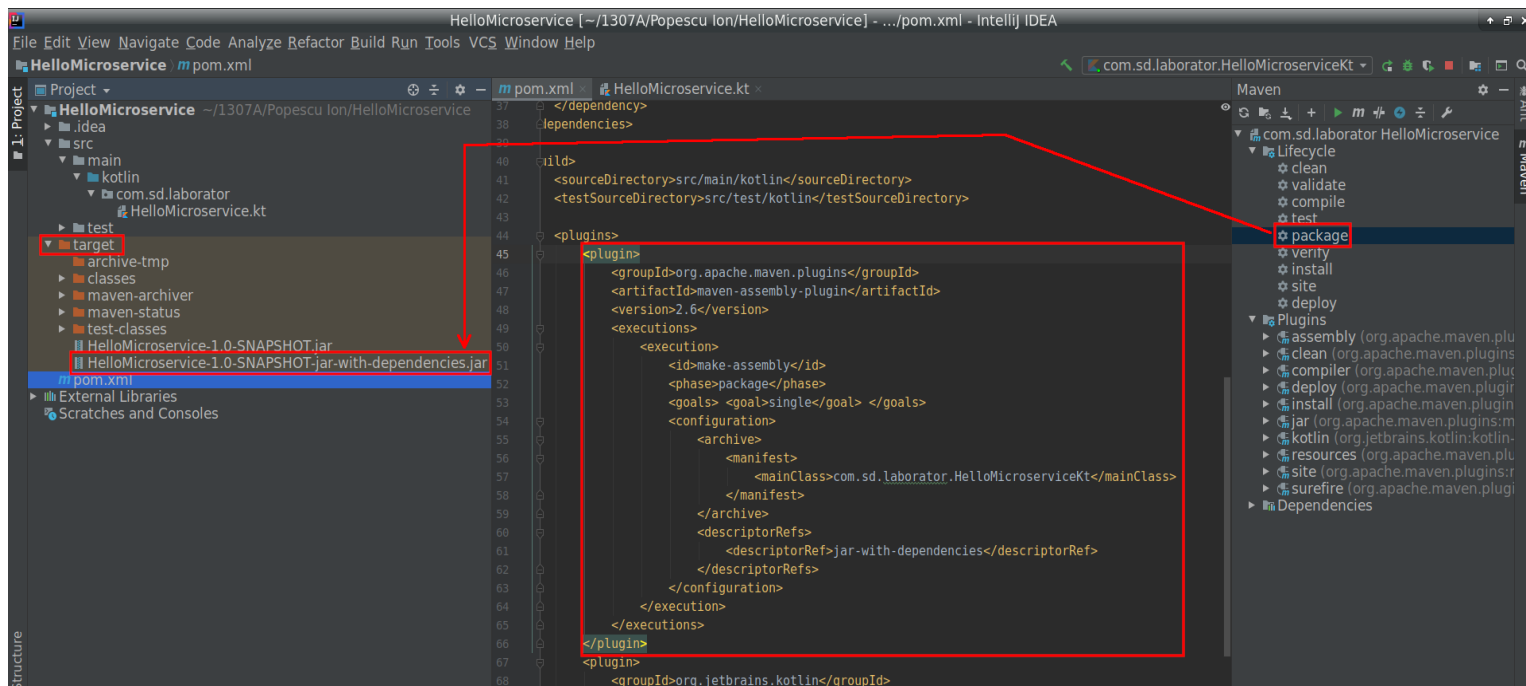


Figura 11 - Împachetarea unui proiect Kotlin/JVM creat cu Maven într-un fat jar

Ca 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_FUNCȚIA_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_PROIECT>-<VERSIONE_PROIECT>-all.jar

Consultați următoarea captură de ecran pentru un exemplu concret:

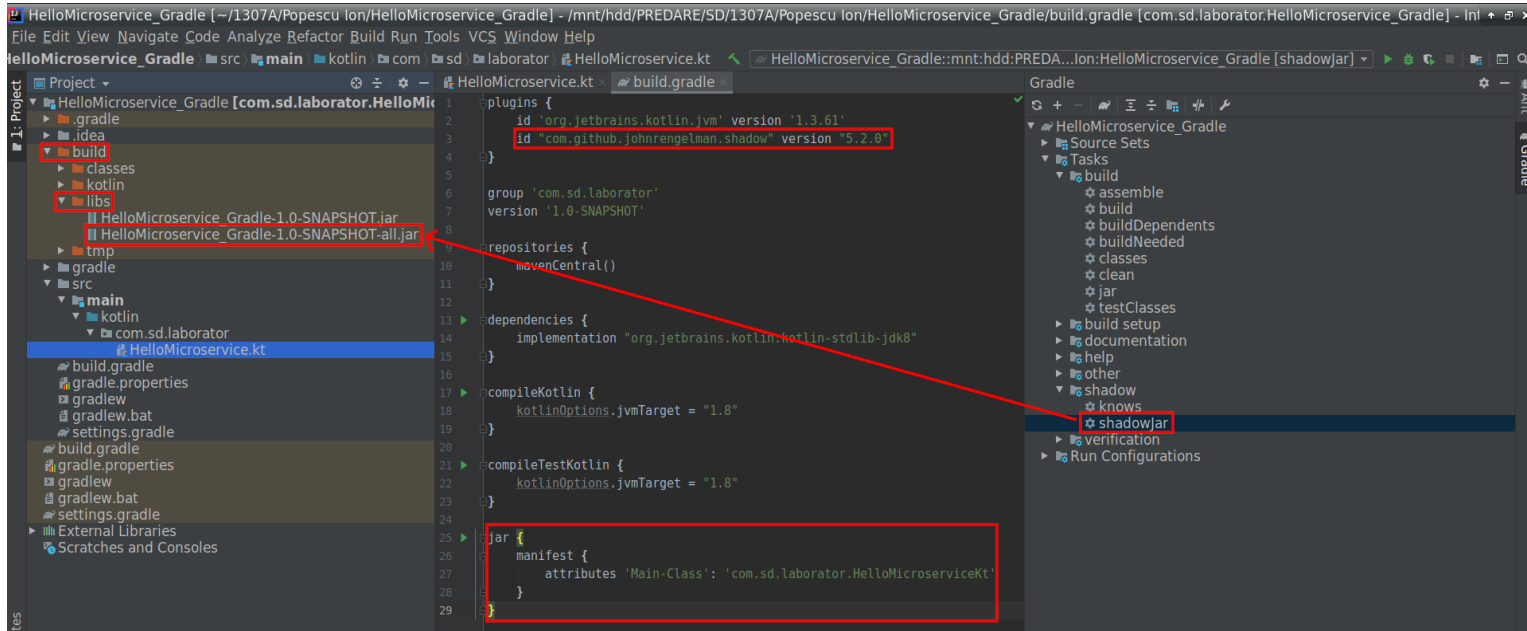


Figura 12 - Împachetarea unui proiect Kotlin/JVM creat cu Gradle într-un fat jar

Ca și consecință: puteți executa aplicația ca un întreg, folosind comanda următoare:

```
java -jar <NUME_ARTEFACT_JAR>.jar
```

Compilare aplicație

Compilați 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_PROIECT>-<VERSIONE>-jar-with-dependencies.jar**

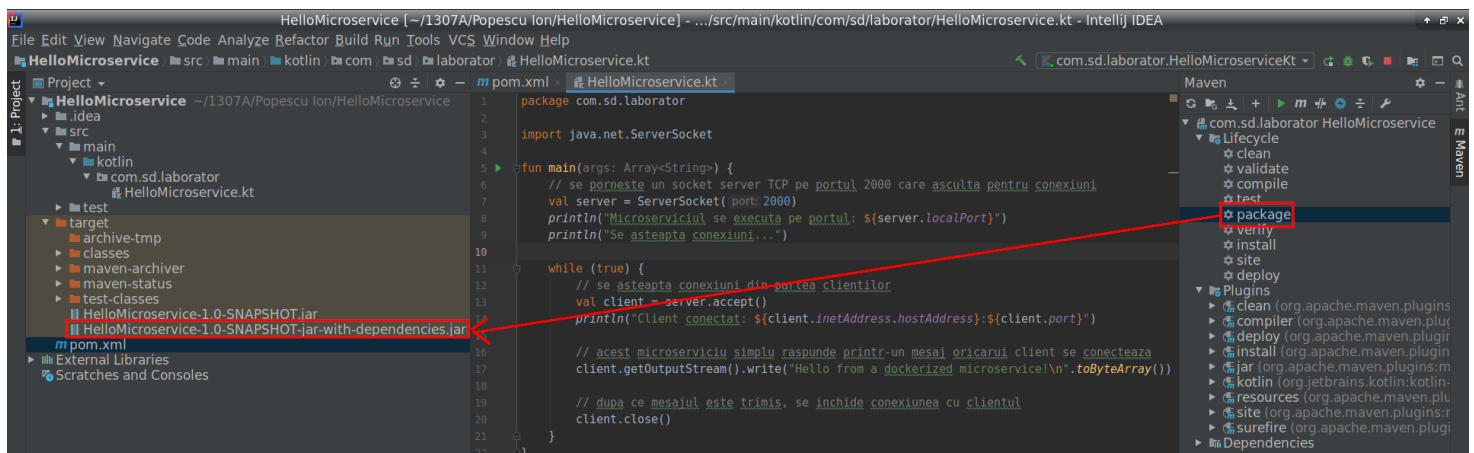


Figura 13 - Împachetarea microserviciului cu Maven

- **Gradle** - folosiți *task*-ul **shadowJar**, iar rezultatul împachetării îl găsiți în folder-ul **build/libs**, denumit sub forma: **<NUME_PROIECT>-<VERSIONE>-all.jar**

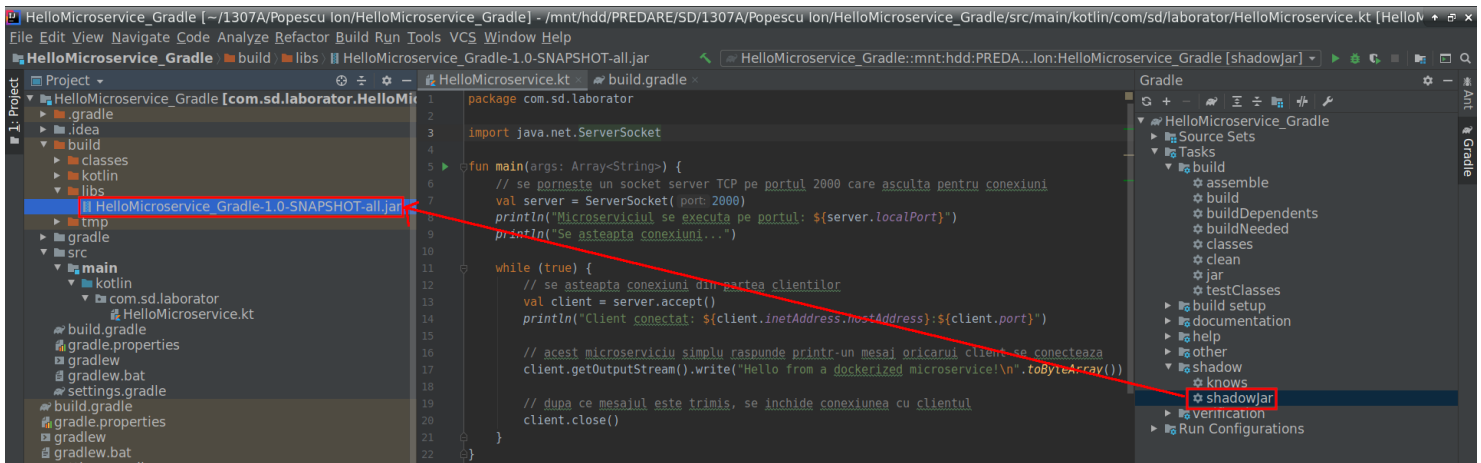


Figura 14 - Împachetarea microserviciului cu Gradle

Crearea unui fișier Dockerfile

În cazul în care nu aveți instalat *plugin*-ul Docker pentru IntelliJ, îl puteți instala din meniul **File** → **Settings** → **Plugins**, iar aici căutați „**docker**”.

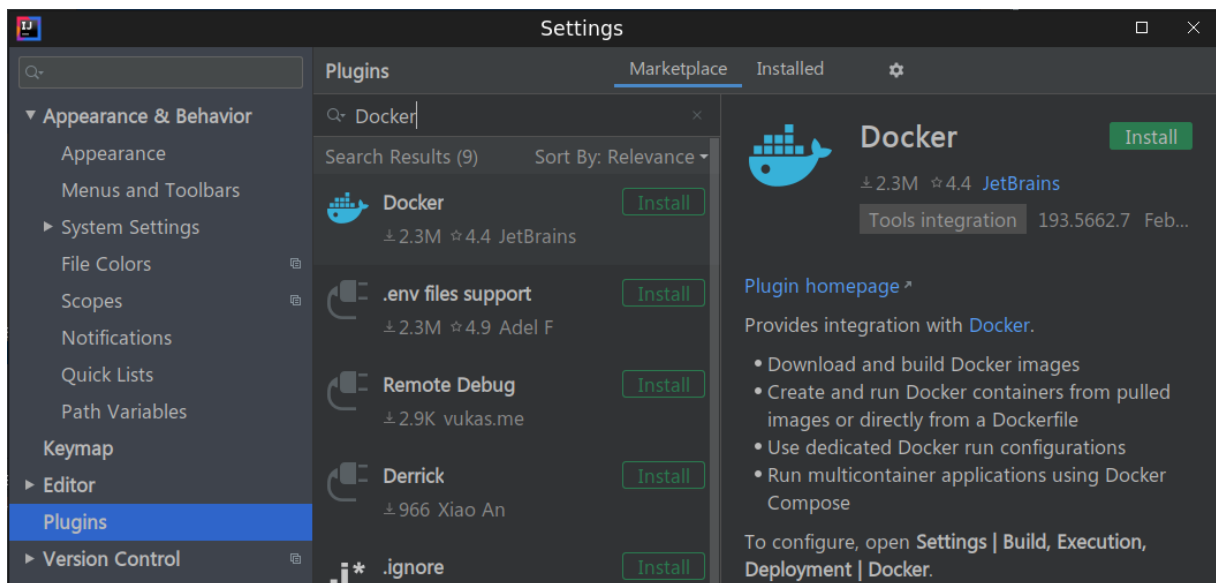


Figura 15 - Instalare plugin Docker în IntelliJ

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 (incluzând 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.

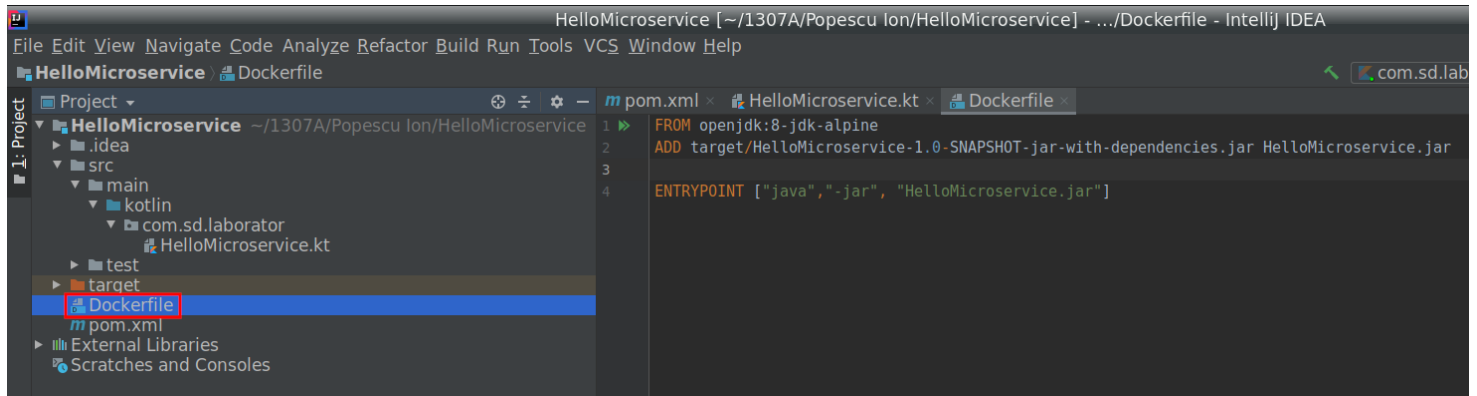


Figura 16 - Dockerfile pentru microserviciul creat cu Maven

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_IMAGE>
```

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**.

Executați 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 identificatorul 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 imaginii 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:

```

cosmin@debian-gl553v: ~
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. Imaginile care sunt încărcate în orice alt registru diferit de cel public (**registry-1.docker.io**) trebuie etichetate sub forma:

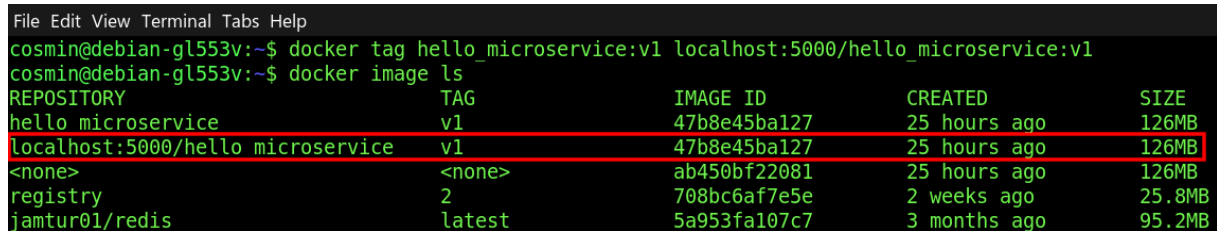
```
<ADRESĂ_REGISTRU>:<PORT_REGISTRU>/<NUME_DEPOZIT>:<ETICHETĂ>
```

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
```



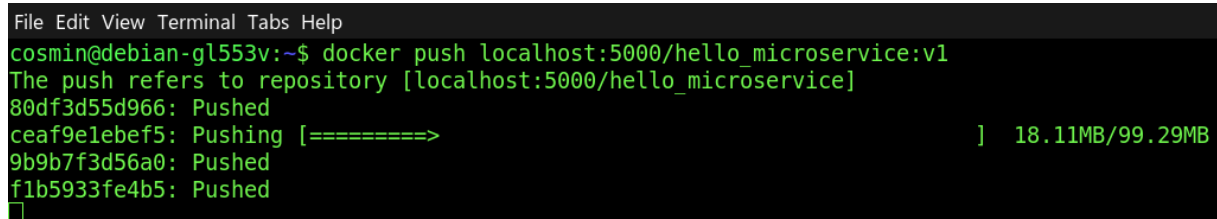
REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
hello_microservice	v1	47b8e45ba127	25 hours ago	126MB
localhost:5000/hello_microservice	v1	47b8e45ba127	25 hours ago	126MB
<none>	<none>	ab450bf22081	25 hours ago	126MB
registry	2	708bc6af7e5e	2 weeks ago	25.8MB
jamtur01/redis	latest	5a953fa107c7	3 months ago	95.2MB

Figura 20 - Imaginea creată, după reetichetare

Apoi încercați din nou să o încărcăți în registru:

```
docker push localhost:5000/hello_microservice:v1
```

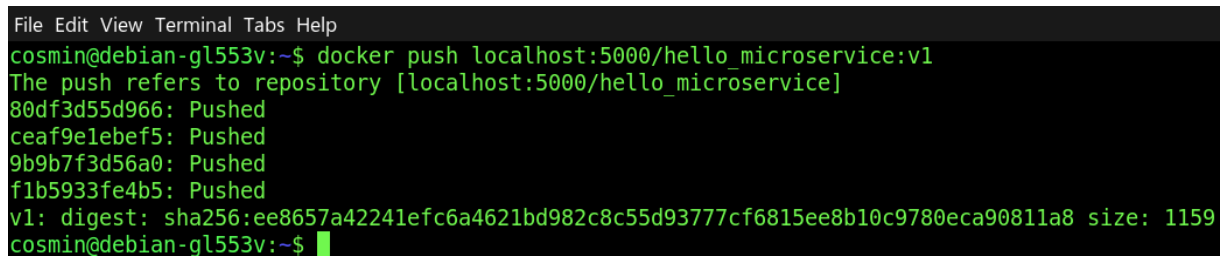
Așteptați până când se încarcă fiecare strat al sistemului de fișiere *Union File System*:



```
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:



```
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, așa 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. Executați comanda următoare pentru a trimite o astfel de cerere:

```
curl -X GET http://localhost:5000/v2/_catalog
```



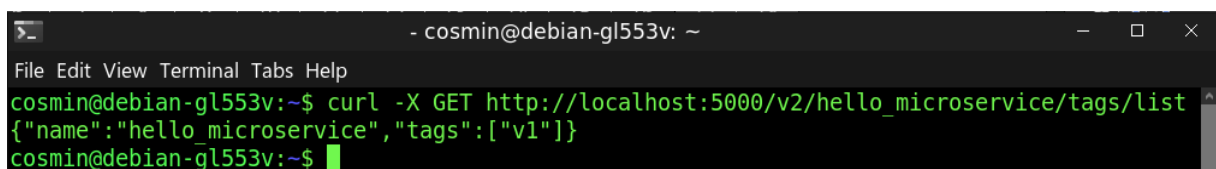
```
- cosmin@debian-gl553v: ~  
File Edit View Terminal Tabs Help  
cosmin@debian-gl553v:~$ curl -X GET http://localhost:5000/v2/_catalog  
{\"repositories\":[\"hello_microservice\"]}  
cosmin@debian-gl553v:~$
```

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 trimite 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:~$
```

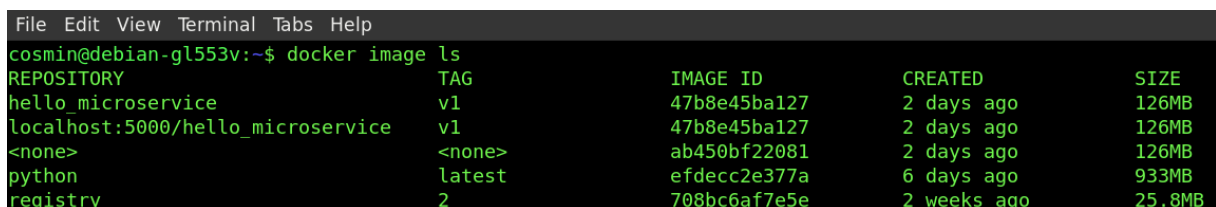
Figura 24 - Listarea etichetelor unui depozit Docker dintr-un registru local

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.**

Ștergerea imaginilor Docker

Ștergeți imaginea locală care încapsulează microserviciul, pentru a putea folosi imaginea încărcată în registrul Docker. Listați mai întâi imaginile locale:

```
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      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 ștergeți imaginea `localhost:5000/hello_microservice:v1`, utilizând una din comenzile 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 - Ștergerea unei imagini Docker

Acest pas îl efectuați strict pentru scop demonstrativ, pentru ca mai departe să utilizați registrul în care această imagine Docker rezidă.

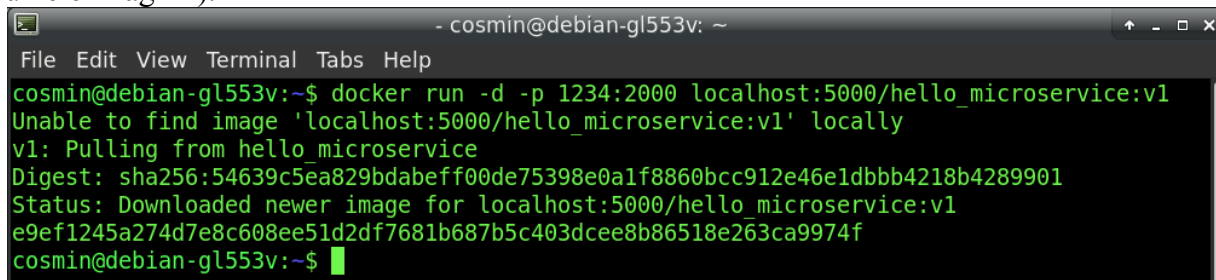
Pornirea microserviciului încărcat în registrul local

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 executați 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 comenzii (înainte de numele imaginii).

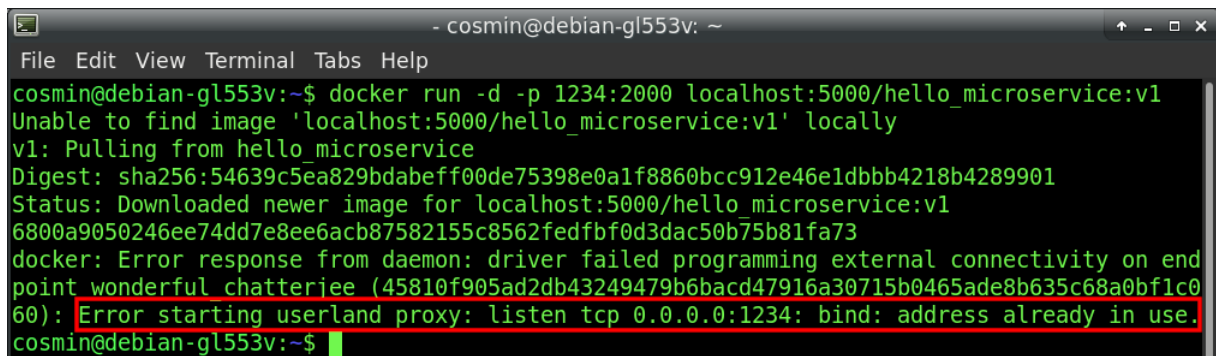


```
- cosmin@debian-gl553v: ~
File Edit View Terminal Tabs Help
cosmin@debian-gl553v:~$ docker run -d -p 1234:2000 localhost:5000/hello_microservice:v1
Unable to find image 'localhost:5000/hello_microservice:v1' locally
v1: Pulling from hello_microservice
Digest: sha256:54639c5ea829bdabeff00de75398e0a1f8860bcc912e46e1dbbb4218b4289901
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 end
point wonderful chatterjee (45810f905ad2db43249479b6bacd47916a30715b0465ade8b635c68a0b1c0
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 end  
point 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
```

```
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.  
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                               suspicious_past  
9d2e57dfab9b   localhost:5000/hello_microservice:v1 "java -jar HelloMicr..." About a minute ago Up About a minute 0.0.0.0:1234->2000/tcp           gifted_vaughan  
6800a9050246   localhost:5000/hello_microservice:v1 "java -jar HelloMicr..." 3 minutes ago Created                               wonderful_chatt  
erjee  
354d163a4a55   registry:2                          "/entrypoint.sh /etc..." 7 minutes ago Up 7 minutes    0.0.0.0:5000->5000/tcp           registru_docker
```

Figura 30 - Container care ocupă portul 1234

Opriți (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
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
e9ef1245a274	localhost:5000/hello_microservice:v1	"java -jar HelloMicr..."	2 minutes ago	Up 2 minutes	0.0.0.0:1234->2000/tcp	sharp_newton
354d163a4a55	registry:2	"/entrypoint.sh /etc..."	16 minutes ago	Up 16 minutes	0.0.0.0:5000->5000/tcp	registru_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.

Log-urile microserviciului se pot vizualiza astfel:

```
docker logs <NUME_SAU_ID_IMAGE>
```

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 si 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 catre microserviciu...")
    sock.send(bytes("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 țintă 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:

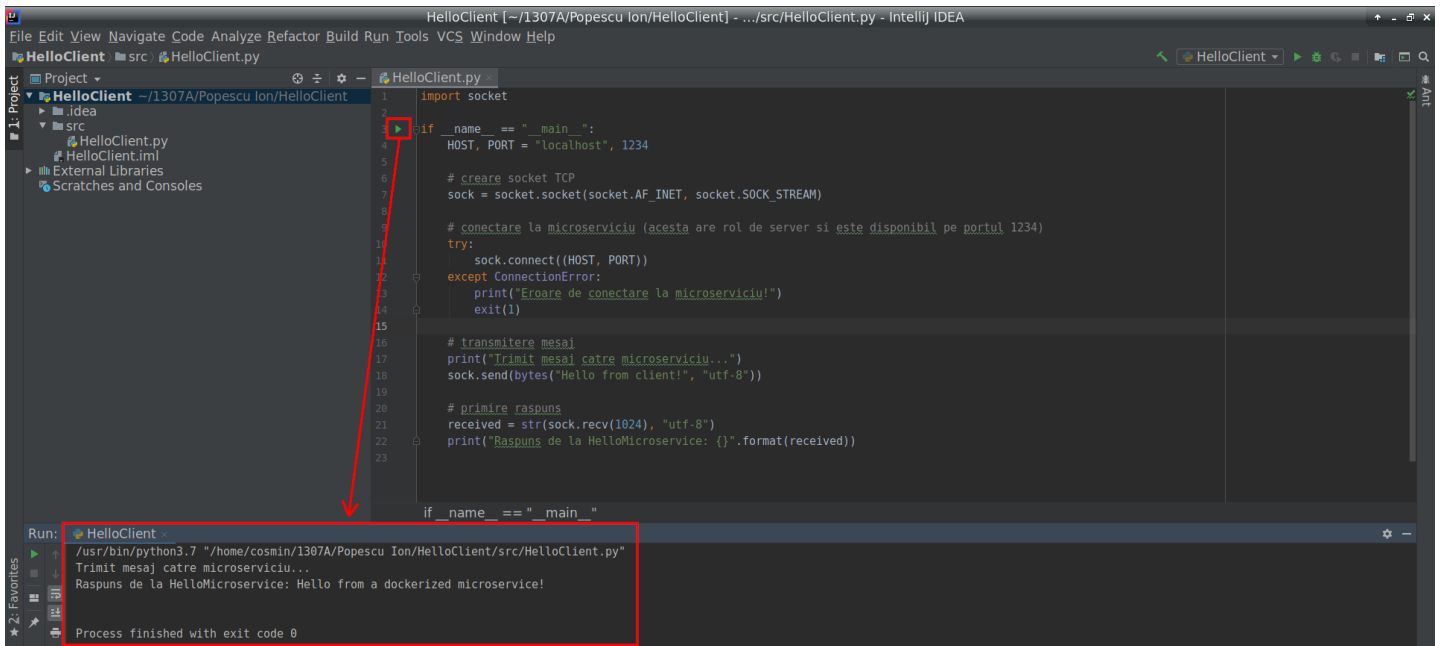


Figura 33 - Testarea microserviciului

Mesajele de pe server se pot verifica accesând log-urile containerului Docker:

```
docker logs e9ef1245a274
```

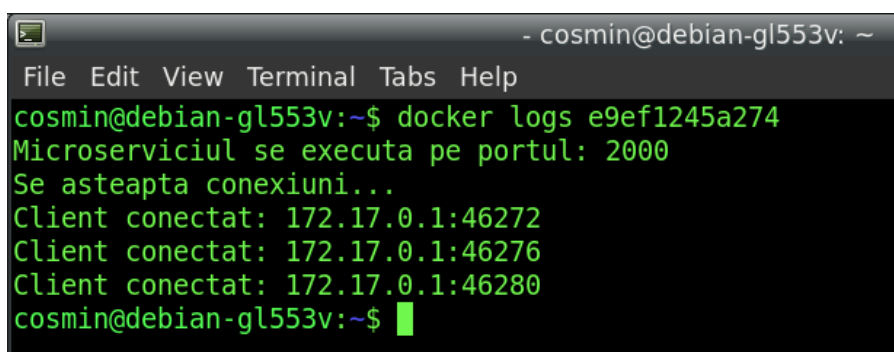


Figura 34 - Log-urile microserviciului după 3 conectări client

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 șterge de tot.

Oprirea containerului se face cu următoarea comandă:

```
docker stop <NUME_SAU_ID_CONTAINER_ÎN_EXECUȚIE>
```

În acest caz:

```
docker stop e9ef1245a274
```

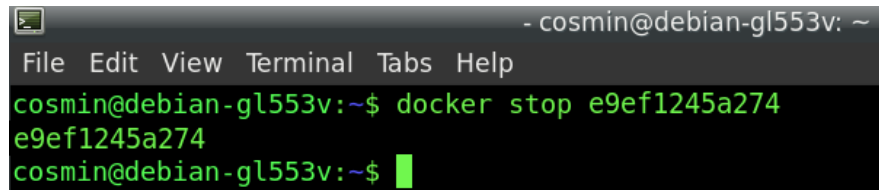


Figura 35 - Oprirea unui container Docker

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)

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
v1ra						
33e091dbd135	531d2369e729	"java -jar HelloMicr..."	23 minutes ago	Created		thirsty_nightin
gale						
a03d01d80118	531d2369e729	"java -jar HelloMicr..."	30 minutes ago	Created		suspicious_past
eur						
6800a9050246	531d2369e729	"java -jar HelloMicr..."	33 minutes ago	Created		wonderful_chatt
eriee						

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 exersa 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

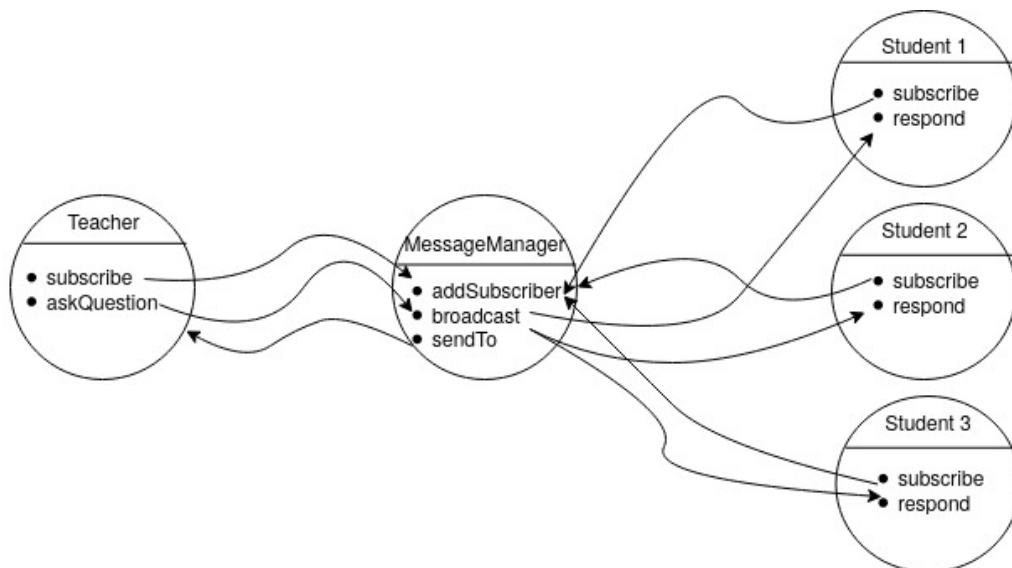


Figura 37 - Diagrama de microservicii

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 înscriu 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:

```
intrebare <DESTINATAR_RĂSPUNS> <CONȚINUT_ÎNTREBARE>
```

MessageManager redirecționează mesajul către toți cei înscriși (*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:

```
raspuns <DESTINATAR_RĂSPUNS> <CONȚINUT_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.

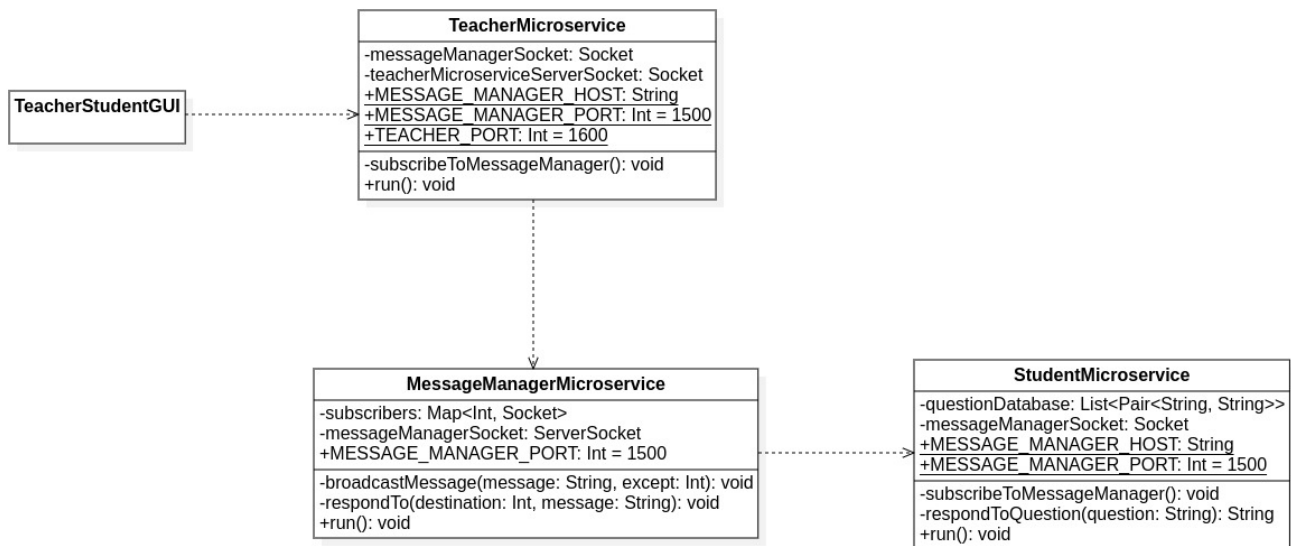


Figura 38 - Diagrama de clase

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 si raspunsurile sunt mentinute intr-o lista 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 intrebari si raspunsuri 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 bufferReader =
BufferedReader(InputStreamReader(messageManagerSocket.inputStream))

        while (true) {
            // se asteapta intrebari trimise prin intermediarul
"MessageManager"
            val response = bufferReader.readLine()

```

```

        if (response == null) {
            // daca se primește un mesaj gol (NULL), atunci
            // înseamnă că cealaltă parte a socket-ului a fost închisă
            println("Microserviciul MessageService
            (${messageManagerSocket.port}) a fost oprit.")
            bufferedReader.close()
            messageManagerSocket.close()
            break
        }

        // se folosește un thread separat pentru tratarea
        // întrebării primite
        thread {
            val (messageType, messageDestination, messageBody) =
            response.split(" ", limit = 3)

            when(messageType) {
                // tipul mesajului cunoscut de acest microserviciu
                // este de forma:
                // întrebare <DESTINATIE_RASPUNS>
                <CONTINUT_INTREBARE>
                "intrebare" -> {
                    println("Am primit o întrebare 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 rețeaua 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>
            responseBody = MessageManagerMicroservice().
            responseBody
        }
    }
}

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 trimite răspunsul unei întrebări înapoi la destinatar, ori se trimite 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 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
    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 (intrebări)...")

    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 fiecărei
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("Trimit catre MessageManager: ${"intrebare
${messageManagerSocket.localPort} $receivedQuestion\n"}")

            messageManagerSocket.getOutputStream().write(("intrebare
${messageManagerSocket.localPort} $receivedQuestion\n").toByteArray())

```

```

        // se asteapta raspuns de la MessageManager
        val messageManagerBufferedReader =
BufferedReader(InputStreamReader(messageManagerSocket.inputStream))
        try {
            val receivedResponse =
messageManagerBufferedReader.readLine()

            // se trimite 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> <CONȚINUT_MESAJ_DE_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).

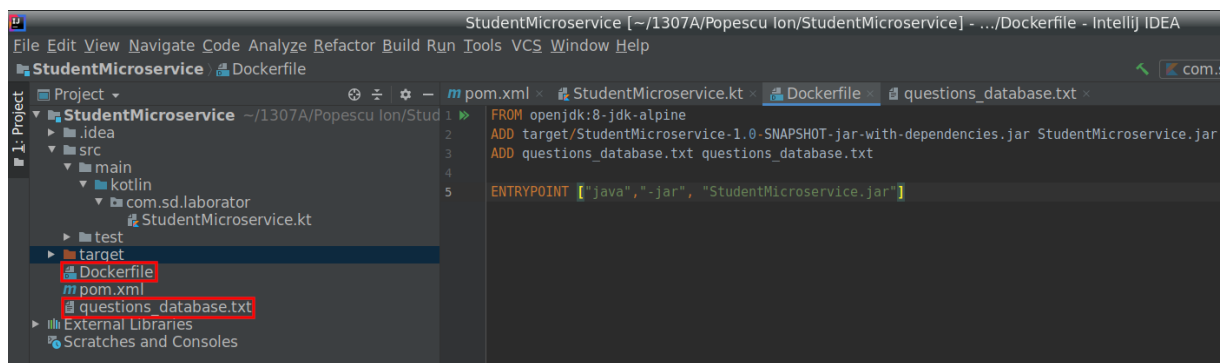


Figura 39 - Dockerfile și fișier cu date pentru StudentMicroservice

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 registului 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

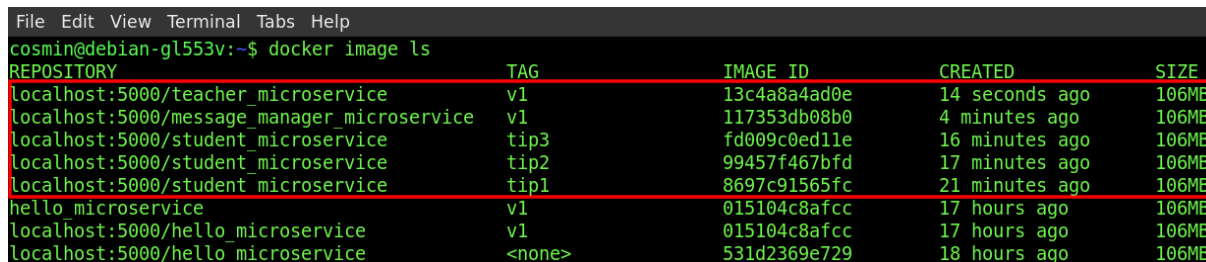
ENTRYPOINT ["java","-jar", "TeacherMicroservice.jar"]
```

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
```

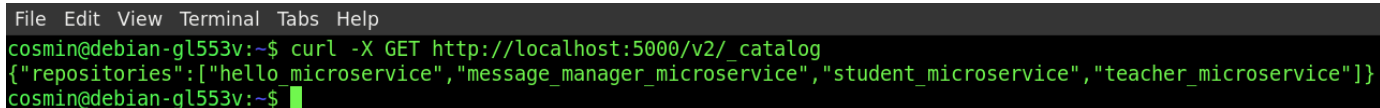


REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
localhost:5000/teacher_microservice	v1	13c4a8a4ad0e	14 seconds ago	106MB
localhost:5000/message_manager_microservice	v1	117353db08b0	4 minutes ago	106MB
localhost:5000/student_microservice	tip3	fd009c0ed11e	16 minutes ago	106MB
localhost:5000/student_microservice	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 construiți 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
```



```
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

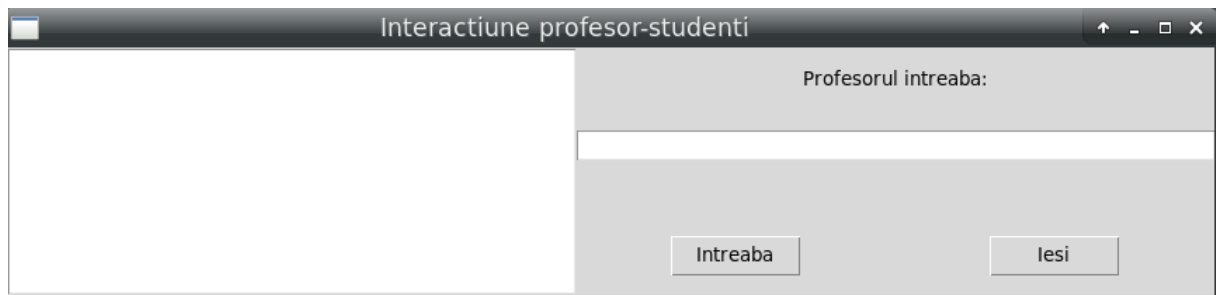
Implementarea interfeței grafice

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 blochează 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 intrebari
    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, se iese 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 principala 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 trimite 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 încep 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 creai 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:~$ 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 trimite 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:~$ docker run -d -p 1600:1600 -e MESSAGE_MANAGER_HOST='message_manage
r' --name teacher_microservice --network=ms-net localhost:5000/teacher_microservice:v1
35bb0fc7c59e1227df9574a2db0349b2f4125a473d2574f758d9afd84d66839d
cosmin@debian-gl553v:~$ docker logs 35bb0fc7c59e1227df9574a2db0349b2f4125a473d2574f758d9af
d84d66839d
Interc sa ma conectez la: message_manager:1500
M-am conectat la MessageManager!
TeacherMicroservice se executa pe portul: 1600
Se asteapta cereri (intrebări)...
cosmin@debian-gl553v:~$
```

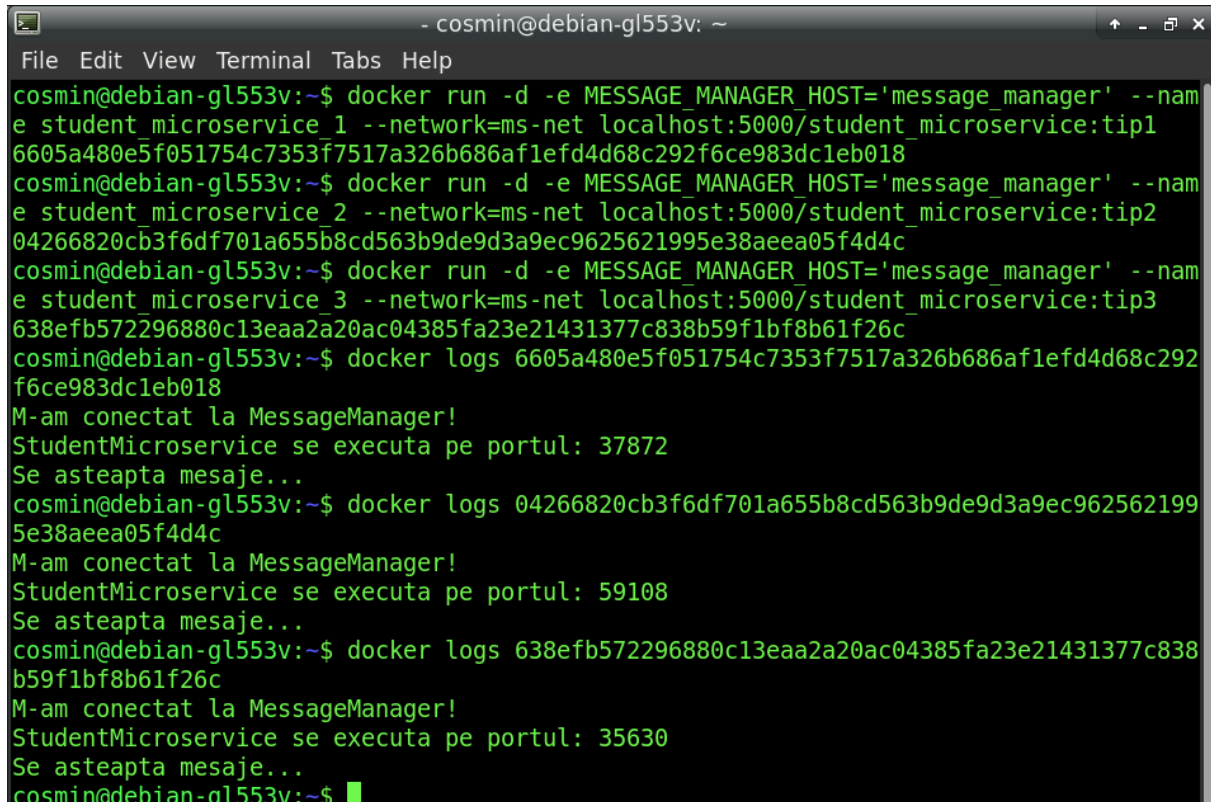
Figura 45 - Pornirea și verificarea microserviciului Teacher

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' --nam
e student_microservice_1 --network=ms-net localhost:5000/student_microservice:tip1
6605a480e5f051754c7353f7517a326b686af1efd4d68c292f6ce983dc1eb018
cosmin@debian-gl553v:~$ docker run -d -e MESSAGE_MANAGER_HOST='message_manager' --nam
e student_microservice_2 --network=ms-net localhost:5000/student_microservice:tip2
04266820cb3f6df701a655b8cd563b9de9d3a9ec9625621995e38aeea05f4d4c
cosmin@debian-gl553v:~$ docker run -d -e MESSAGE_MANAGER_HOST='message_manager' --nam
e student_microservice_3 --network=ms-net localhost:5000/student_microservice:tip3
638efb572296880c13eaa2a20ac04385fa23e21431377c838b59f1bf8b61f26c
cosmin@debian-gl553v:~$ docker logs 6605a480e5f051754c7353f7517a326b686af1efd4d68c292
f6ce983dc1eb018
M-am conectat la MessageManager!
StudentMicroservice se executa pe portul: 37872
Se asteapta mesaje...
cosmin@debian-gl553v:~$ docker logs 04266820cb3f6df701a655b8cd563b9de9d3a9ec962562199
5e38aeea05f4d4c
M-am conectat la MessageManager!
StudentMicroservice se executa pe portul: 59108
Se asteapta mesaje...
cosmin@debian-gl553v:~$ docker logs 638efb572296880c13eaa2a20ac04385fa23e21431377c838
b59f1bf8b61f26c
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
```

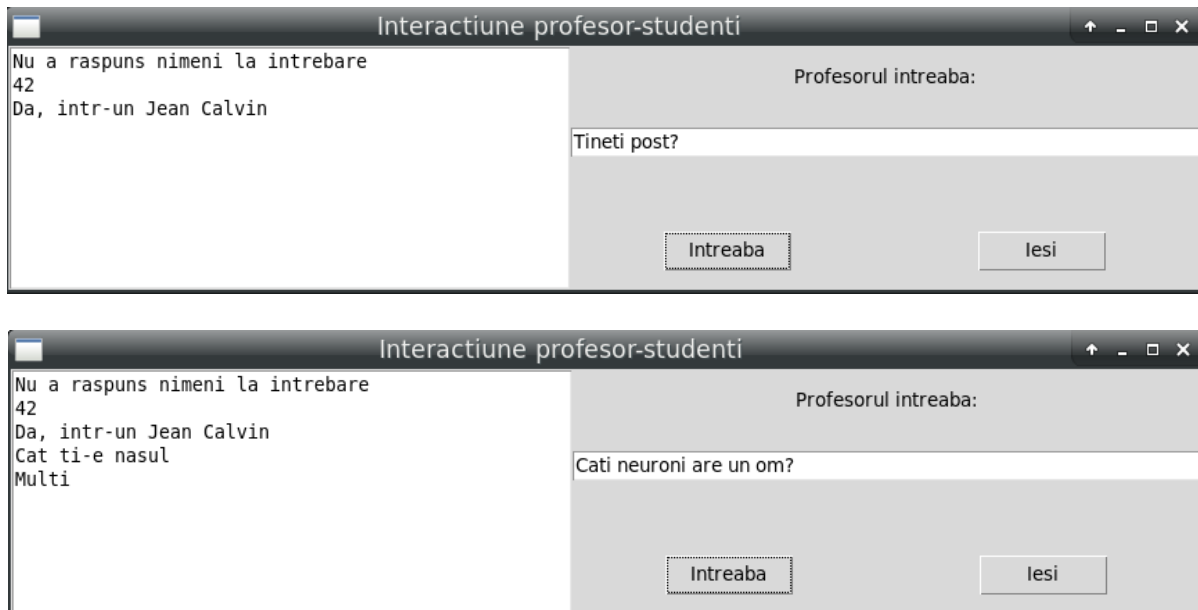
```

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
04266820cb3f      localhost:5000/student_microservice:tip2  "java -jar StudentMi..." About a minute ago Up About a minute   student_micro
service_2
6605a480e5f0      localhost:5000/student_microservice:tip1  "java -jar StudentMi..." About a minute ago Up About a minute   student_micro
service_1
35bb0fc7c59e      localhost:5000/teacher_microservice:v1    "java -jar TeacherMi..." 4 minutes ago      Up 4 minutes        0.0.0.0:1600->1600/tcp teacher_micro
service
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 rețeaua 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?"  
Trimit 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 trimite 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 dezînscriere 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>