

Sisteme Distribuite - Laborator 8

Instalarea microserviciilor folosind Docker

Instalarea Docker Engine

Atenție! Cei care lucrează de pe stațiile din laborator, săriți direct la Secțiunea “Lucrul cu comenzi Docker - crearea unui registru local de imagini”.

Se oferă instrucțiuni de instalare pentru sistemul de operare **Debian**, disponibil pe stațiile din laborator. Comenzile care urmează vor fi executate într-o sesiune de **terminal**.

1. Actualizați indexii gestionarului de pachete **apt**:

```
sudo apt-get update
```

2. Permiteți folosirea depozitelor de pachete (engl. *repositories*) prin HTTPS + instalați câteva pachete adiționale necesare:

```
sudo apt-get install \
ca-certificates \
curl \
gnupg \
lsb-release
```

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 gpg --dearmor -o /usr/share/keyrings/docker-archive-keyring.gpg
```

4. Adăugați depozitul de pachete oficial Docker în lista sistemului:

```
sudo echo \
"deb [arch=$(dpkg --print-architecture) signed-
by=/usr/share/keyrings/docker-archive-keyring.gpg]
https://download.docker.com/linux/debian \
$(lsb_release -cs) stable" | sudo tee
/etc/apt/sources.list.d/docker.list > /dev/null
```

5. Actualizați din nou indeșii **apt** (pas necesar pentru că ați modificat mai sus baza de date cu depozite de pachete):

```
sudo apt-get 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-get install docker-ce docker-ce-cli containerd.io
```

Pentru pornirea serviciului de docker:

```
sudo systemctl start docker
```

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 prefișați 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
```



```

Terminal - student@debian: ~
File Edit View Terminal Tabs Help
student@debian:~$ docker info
Client:
  Context:    default
  Debug Mode: false
  Plugins:
    app: Docker App (Docker Inc., v0.9.1-beta3)
    buildx: Build with BuildKit (Docker Inc., v0.6.3-docker)
    scan: Docker Scan (Docker Inc., v0.9.0)

Server:
  Containers: 3
    Running: 2
    Paused: 0
    Stopped: 1
  Images: 2
  Server Version: 20.10.10
  Storage Driver: overlay2
    Backing Filesystem: extfs
    Supports d_type: true
    Native Overlay Diff: true
    userxattr: false
  Logging Driver: json-file
  Cgroup Driver: systemd
  Cgroup Version: 2
  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: io.containerd.runc.v2 io.containerd.runtime.v1.linux runc
  Default Runtime: runc
  Init Binary: docker-init
  containerd version: 5b46e404f6b9f661a205e28d59c982d3634148f8
  runc version: v1.0.2-0-g52b36a2
  init version: de40ad0
  Security Options:
    apparmor
    seccomp
      Profile: default
    cgroupns
  Kernel Version: 5.10.0-9-amd64
  Operating System: Debian GNU/Linux 11 (bullseye)
  OSType: linux
  Architecture: x86_64
  CPUs: 4
  Total Memory: 7.74GiB
  Name: debian
  ID: K53K:3LBX:FK3Q:7X73:RK7W:WN4L:C7HQ:RWOM:MYCI:0FGF:VQMA:5RAR
  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/>.

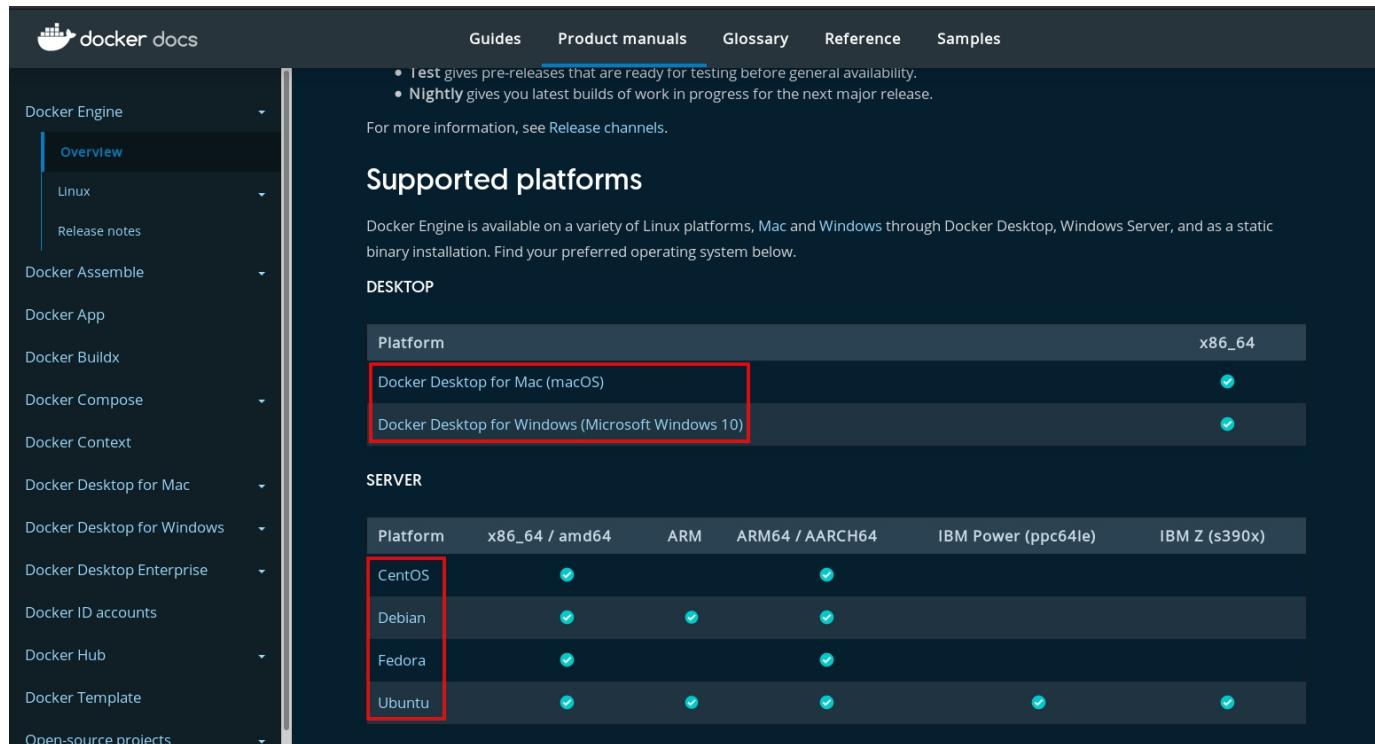


Figura 2 - Instrucțiuni de instalare pentru diverse platforme

Lucrul cu comenzi Docker - crearea unui registru local de imagini

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.

Această abordare este singura admisibilă în contextul DevSecOps unde (mai vedem la cripto) trebuie luate măsuri de securizare a mașinilor care sunt suport pentru microservicii deci nu o neglijăți)

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**):

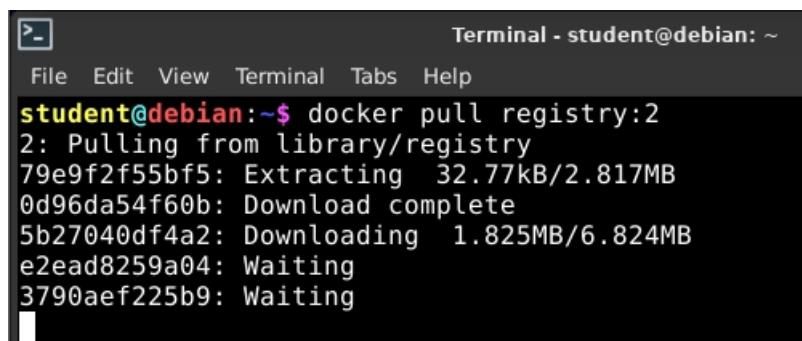


Figura 3 - Preluarea unei imagini Docker

După ce se termină de descărcat, imaginea este disponibilă pentru utilizare:

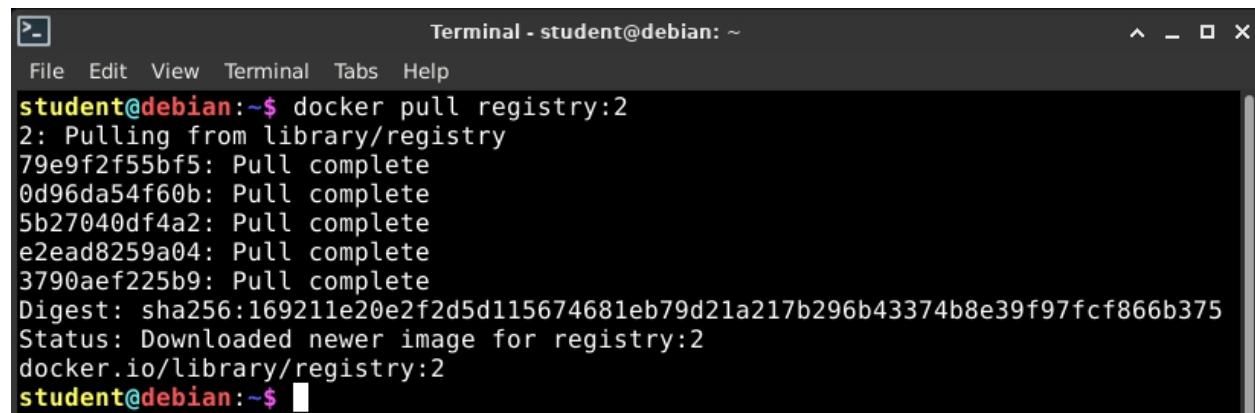


Figura 4 - Descărcare completă a unei imagini Docker

Verificați că imaginea **registry:2** există în registrul local, cu următoarea comandă:

```
docker image ls
```

```
Terminal - student@debian: ~
File Edit View Terminal Tabs Help
student@debian:~$ docker image ls
REPOSITORY TAG IMAGE ID CREATED SIZE
registry 2 b8604a3fe854 4 days ago 26.2MB
student@debian:~$
```

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 detasat (**-d**). Se expune portul **5000** din container și se mapează pe portul **5000** al sistemului, iar containerul este identificat prin numele **registru_docker**.

```
Terminal - student@debian: ~
File Edit View Terminal Tabs Help
student@debian:~$ docker run --restart=always -d -p 5000:5000 --name registru_docker registry:2
419a7d3175d907385f1cfe16258277a59b94c67e51f472ec2a477175a16f8df9
student@debian:~$
```

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
```

```
Terminal - student@debian: ~
File Edit View Terminal Tabs Help
student@debian:~$ docker ps
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
419a7d3175d9 registry:2 "/entrypoint.sh /etc..." 54 seconds ago Up 53 seconds 0.0.0.0:5000->5000/tcp, :::5000->5000/tcp registru_docker
student@debian:~$
```

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 opuse de utilizator, opuse forțat, sau opuse din cauza unei erori), adăugați parametrul **-a** (**--all**)

la comanda de mai sus:

```
docker ps -a
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
37beb8a3bcd7	hello-world	"/hello"	35 seconds ago	Exited (0) 34 seconds ago	0.0.0.0:5000->5000/tcp	dazzling_margulis
419a7d3175d9	registry:2	"/entrypoint.sh /etc..."	3 minutes ago	Up 3 minutes	0.0.0.0:5000->5000/tcp	registrum_docker

Figura 8 - Afisarea listei tuturor containerelor, indiferent de stare

Acum apar și containerele care au fost pornite la un moment dat, iar apoi au fost opriate de utilizator sau din altă cauză.

Verificarea porturilor expuse de un container Docker

Containerul **registrum_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 registrum_docker
```

File	Edit	View	Terminal	Tabs	Help
student@debian:~\$	docker	port	registrum_docker		
			5000/tcp	->	0.0.0.0:5000
			5000/tcp	->	:::5000

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:

File	Edit	View	Terminal	Tabs	Help
student@debian:~\$	docker	port	dc83465a21ba		
			3456/tcp	->	0.0.0.0:2020
			3456/tcp	->	:::2020

Figura 10 - Verificarea porturilor mapate

Pentru cazul din figură, utilizatorul trebuie să acceseze portul **2020** de pe mașina gazdă pentru a face trimitere la serviciul expus pe portul **3456** în interiorul containerului Docker cu identificatorul **dc83465a21ba**.

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 clientilor 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 arțeface 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:

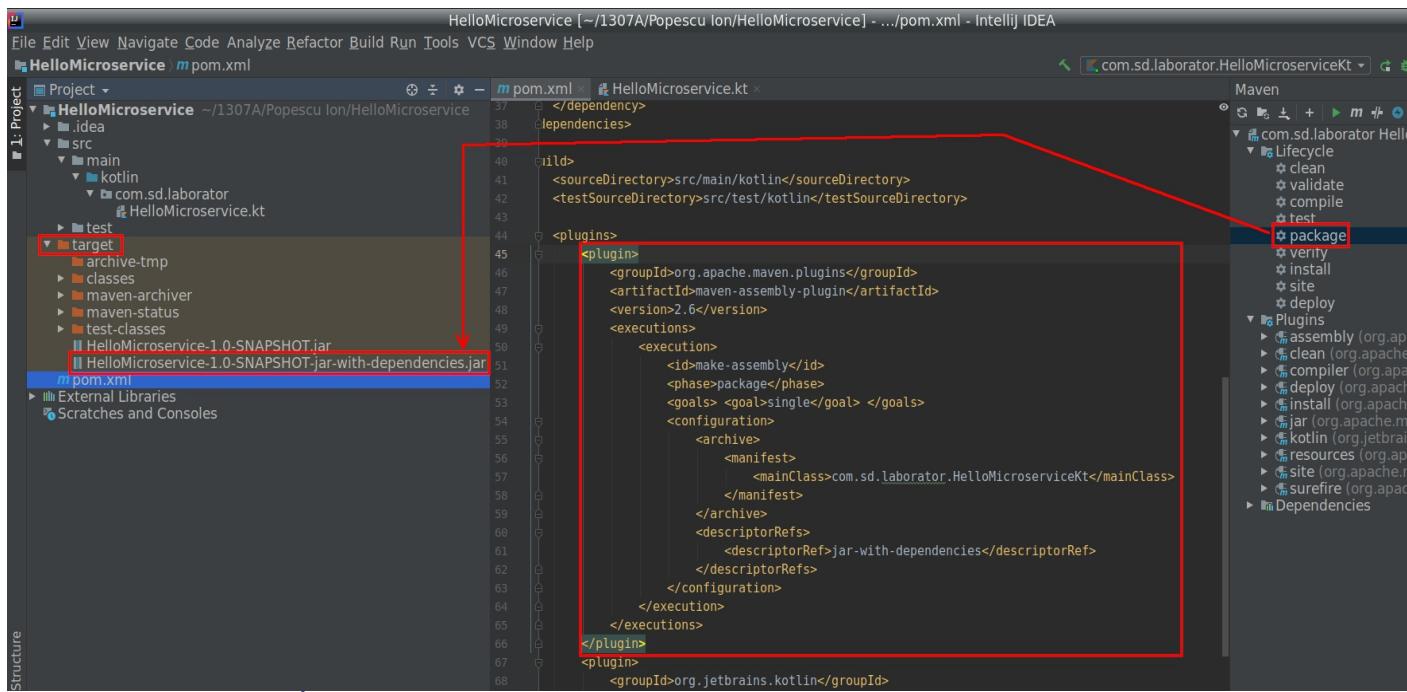


Figura 11 - Împachetarea unui proiect Kotlin/JVM creat cu Maven într-un fat jar

Că și consecință: puteți execuția 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:

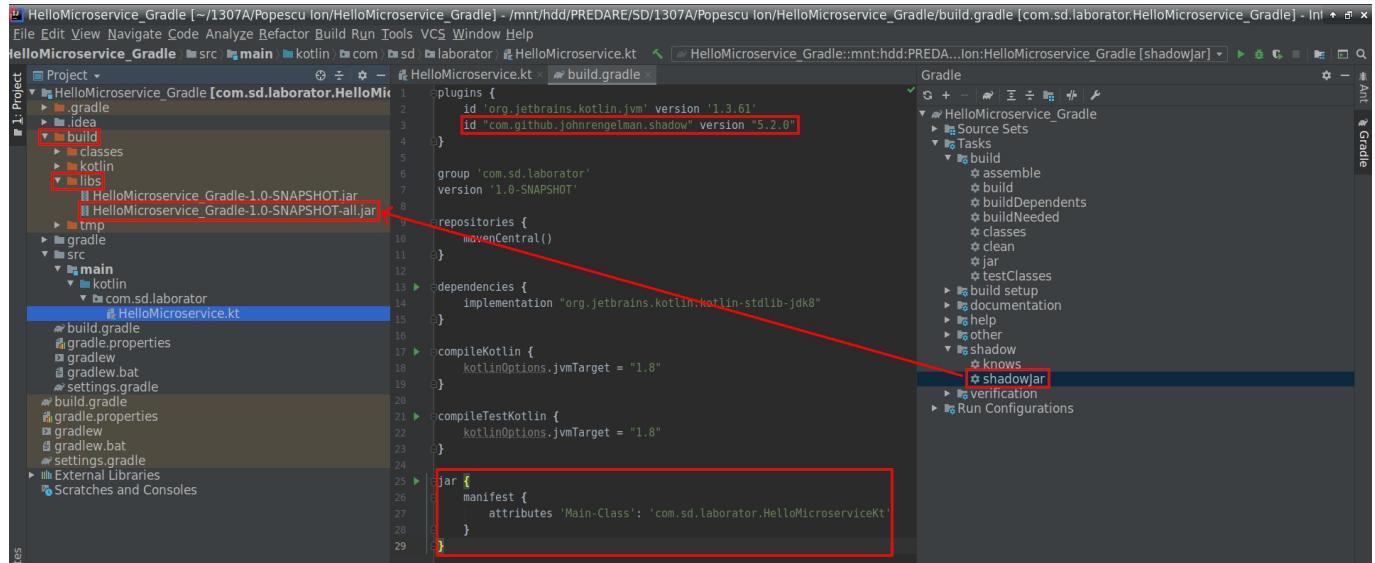


Figura 12 - Împachetarea unui proiect Kotlin/JVM creat cu Gradle într-un fat jar

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**

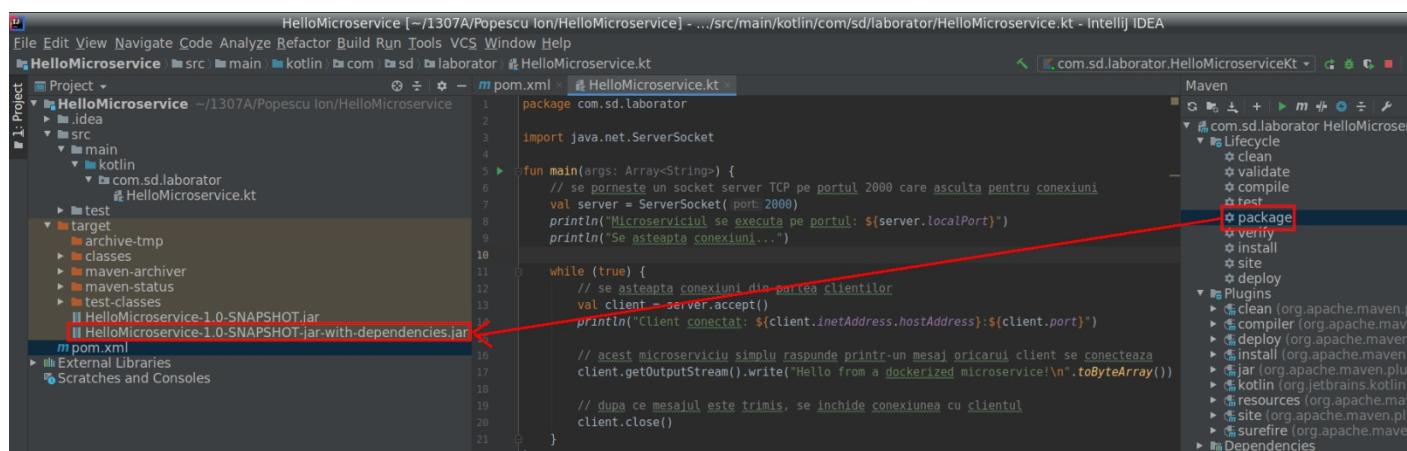


Figura 13 - Împachetarea microserviciului cu Maven

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

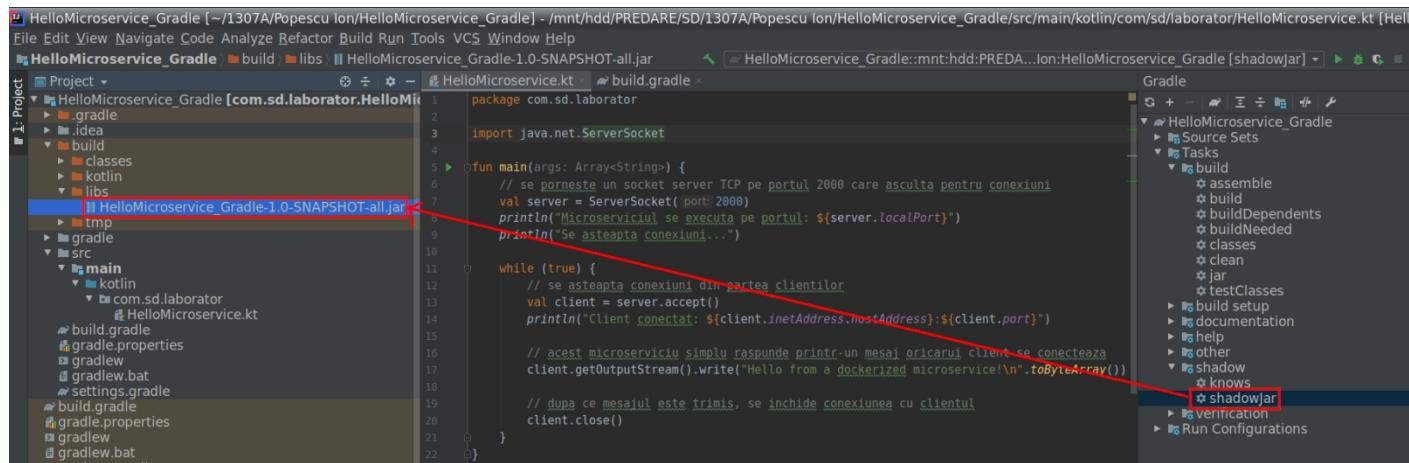


Figura 14 - Împachetarea microserviciului cu Gradle

Crearea unui fișier Dockerfile

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

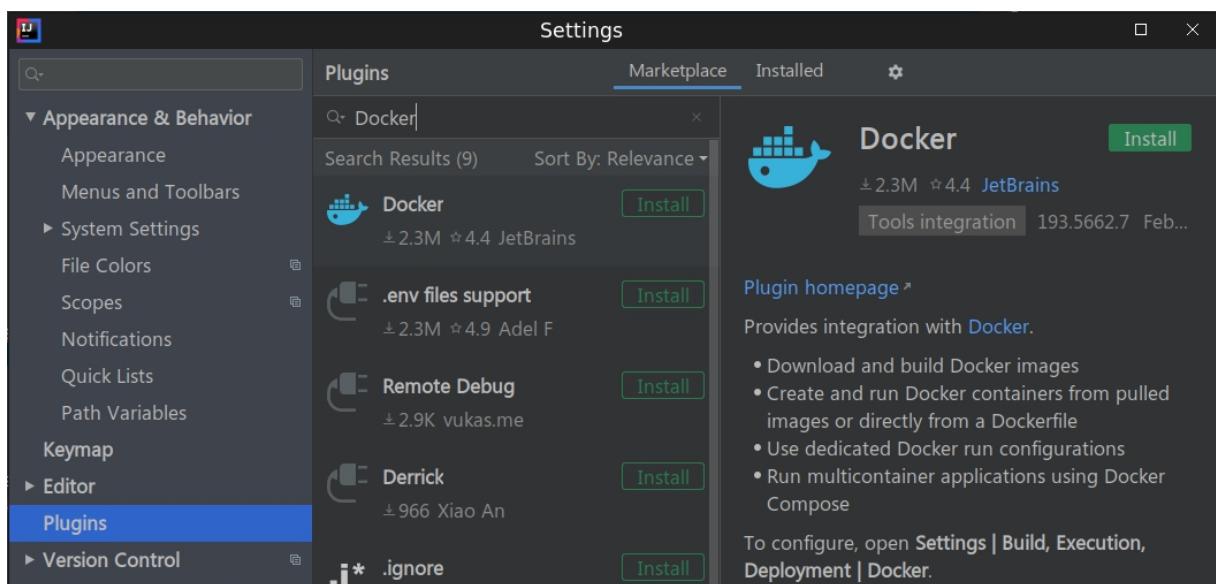


Figura 15 - Instalare plugin Docker în IntelliJ

Reporțați 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.

```

HelloMicroservice [~/1307A/Popescu Ion/HelloMicroservice] - .../Dockerfile - IntelliJ IDEA
File Edit View Navigate Code Analyze Refactor Build Run Tools VCS Window Help
HelloMicroservice > Dockerfile
Project pom.xml HelloMicroservice.kt Dockerfile
HelloMicroservice ~/1307A/Popescu Ion/HelloMicroservice
  .idea
  src
    main
      kotlin
        com.sd.laborator
          HelloMicroservice.kt
    test
  target
  Dockerfile
  pom.xml
External Libraries Scratches and Consoles

```

```

1 FROM openjdk:8-jdk-alpine
2 ADD target>HelloMicroservice-1.0-SNAPSHOT-jar-with-dependencies.jar HelloMicroservice
3
4 ENTRYPOINT ["java", "-jar", "HelloMicroservice.jar"]

```

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 siruri 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 + 
student@debian:~/Documents/SD_Labs/Lab_8/SD_Laborator_08/problema 1>HelloMicroservice_Maven$ docker build -t hello_microservice:v1 .
Sending build context to Docker daemon 1.811MB
Step 1/3 : FROM openjdk:8-jdk-alpine
8-jdk-alpine: Pulling from library/openjdk
e7c96db7181b: Pull complete
f910a506b6cb: Pull complete
c2274a1a0e27: Pull complete
Digest: sha256:94792824df2df33402f201713f932b58cb9de94a0cd524164a0f2283343547b3
Status: Downloaded newer image for openjdk:8-jdk-alpine
--> a3562aa0b991
Step 2/3 : ADD target/HelloMicroservice-1.0-SNAPSHOT-jar-with-dependencies.jar HelloMicroservice.jar
--> 5375ca1f07ca
Step 3/3 : ENTRYPOINT ["java","-jar", "HelloMicroservice.jar"]
--> Running in 029a8cdd382b
Removing intermediate container 029a8cdd382b
--> e327bd910a13
Successfully built e327bd910a13
Successfully tagged hello_microservice:v1
student@debian:~/Documents/SD_Labs/Lab_8/SD_Laborator_08/problema 1>HelloMicroservice_Maven$
```

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
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
hello microservice	v1	e327bd910a13	16 minutes ago	107MB
registry	2	b8604a3fe854	7 days ago	26.2MB
hello-world	latest	feb5d9fea6a5	8 weeks ago	13.3kB
openjdk	8-jdk-alpine	a3562aa0b991	2 years ago	105MB

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:

```
Terminal - student@debian: ~
File Edit View Terminal Tabs Help
student@debian:~$ 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
student@debian:~$
```

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
```

```
Terminal - student@debian: ~
File Edit View Terminal Tabs Help
student@debian:~$ docker tag hello_microservice:v1 localhost:5000/hello_microservice:v1
student@debian:~$ docker image ls
REPOSITORY          TAG      IMAGE ID   CREATED    SIZE
localhost:5000/hello microservice  v1        e327bd910a13  18 minutes ago  107MB
hello_microservice   v1        e327bd910a13  18 minutes ago  107MB
registry             2         b8604a3fe854  7 days ago   26.2MB
hello-world          latest    feb5d9fea6a5  8 weeks ago  13.3kB
openjdk              8-jdk-alpine  a3562aa0b991  2 years ago  105MB
student@debian:~$
```

Figura 20 - Imaginea creată, după reetichetare

Apoi încercați din nou să o încărcați în registrul:

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

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

```
Terminal - student@debian: ~
File Edit View Terminal Tabs Help
student@debian:~$ docker push localhost:5000/hello_microservice:v1
The push refers to repository [localhost:5000/hello_microservice]
455c2a0cddf2: Pushed
ceaf9e1ebef5: Pushing  40.23MB/99.29MB
9b9b7f3d56a0: Pushed
f1b5933fe4b5: Pushed

```

Figura 21 - Încărcarea unei imagini Docker în registrul local

Încărcarea completă, cu succes, arată astfel:

```
Terminal - student@debian: ~
File Edit View Terminal Tabs Help
student@debian:~$ docker push localhost:5000/hello_microservice:v1
The push refers to repository [localhost:5000/hello_microservice]
455c2a0cdddf2: Pushed
ceaf9e1ebef5: Pushed
9b9b7f3d56a0: Pushed
f1b5933fe4b5: Pushed
v1: digest: sha256:ddc6f0a8c7c0272b319e861b23084ca10d624a0889240b23f225c6d5d3baf052 size: 1158
student@debian:~$
```

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
```

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

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
```

```
Terminal - student@debian: ~
File Edit View Terminal Tabs Help
student@debian:~$ curl -X GET http://localhost:5000/v2/hello_microservice/tags/list
[{"name": "hello_microservice", "tags": ["v1"]}
student@debian:~$
```

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	e327bd910a13	5 hours ago	107MB
localhost:5000/hello_microservice	v1	e327bd910a13	5 hours ago	107MB
registry	2	b8604a3fe854	7 days ago	26.2MB
hello-world	latest	feb5d9fea6a5	8 weeks ago	13.3kB
openjdk	8-jdk-alpine	a3562aa0b991	2 years ago	105MB

Figura 25 - Listarea imaginilor Docker locale

Acum ștergeți 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
```

```
student@debian:~$ docker image rm localhost:5000/hello_microservice:v1
Untagged: localhost:5000/hello_microservice:v1
Untagged: localhost:5000/hello_microservice@sha256:ddc6f0a8c7c0272b319e861b23084ca10d624a0889240b23f225c6d5d3baf052
student@debian:~$
```

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 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).

```
student@debian:~$ 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:ddc6f0a8c7c0272b319e861b23084ca10d624a0889240b23f225c6d5d3baf052
Status: Downloaded newer image for localhost:5000/hello_microservice:v1
30098f536f36b7e060638b22a76a46fb5d9de764f8bb89030740389ab2340ffd
student@debian:~$
```

Figura 27 - Pornirea microserviciului

Observați că Docker nu a găsit imaginea respectivă local, aşa 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:

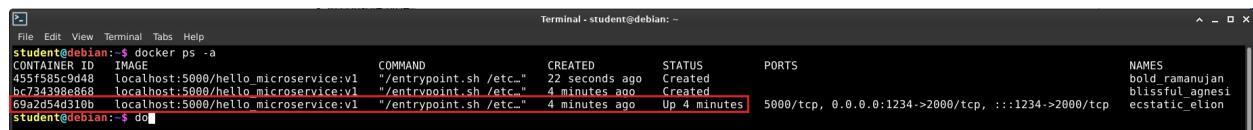


```
student@debian:~$ docker run -d -p 1234:2000 localhost:5000/hello_microservice:v1
bc734398e8681ab8c188775602e233e6fd8be919d75c687c5c6a9c9deb0938ea
docker: Error response from daemon: driver failed programming external connectivity on endpoint blissful_agnesi (49026f7df46b22b0a44317bc1f25d8900c539f451783f6b81cd3a867013801bd): Bind for 0.0.0.0:1234 failed: port is already allocated.
student@debian:~$
```

Figura 28 - Eroare la alocarea 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). Aflați care container ocupă portul respectiv, cu comanda:

```
docker ps -a
```



CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
455f585c9d48	localhost:5000/hello_microservice:v1	"entrypoint.sh /etc..."	22 seconds ago	Created		bold_ramanujan
bc734398e868	localhost:5000/hello_microservice:v1	"entrypoint.sh /etc..."	4 minutes ago	Created		blissful_agnesi
69a2d54d310b	localhost:5000/hello_microservice:v1	"entrypoint.sh /etc..."	4 minutes ago	Up 4 minutes	5000/tcp, 0.0.0.0:1234->2000/tcp, :::1234->2000/tcp	ecstatic_elion

Figura 29 - Container care ocupă portul 1234

Opriți (dacă este pornit) și ștergeți containerul care ocupă portul pe care doriți să îl folosiți:

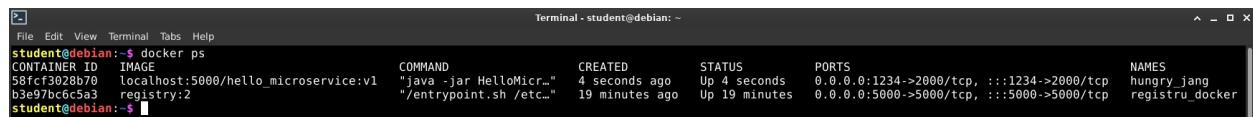
```
docker stop <IDENTIFICATOR>
docker rm <IDENTIFICATOR>
```

În acest caz:

```
docker stop 69a2d54d310b
docker rm 69a2d54d310b
```

Verificați starea containerului **hello_microservice**:

```
docker ps
```



CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
58fcf3028b70	localhost:5000/hello_microservice:v1	"java -jar HelloMicr..."	4 seconds ago	Up 4 seconds	0.0.0.0:1234->2000/tcp, :::1234->2000/tcp	hungry_jang
b3e97b6c5a3	registry:2	"/entrypoint.sh /etc..."	19 minutes ago	Up 19 minutes	0.0.0.0:5000->5000/tcp, :::5000->5000/tcp	registrar_docker

Figura 30 - 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_IMAGINE>
```

În acest caz:

```
docker logs 58fcf3028b70
```

Figura 31 - Vizualizarea log-urilor microserviciului

Testarea microserviciului - crearea unei aplicații client

Creați un proiect nou de tip Python din PyCharm 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("Trimite 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:

Sisteme Distribuite - Laborator 8

```
File Edit View Navigate Code Refactor Run Tools VCS Window Help
HelloClient src / HelloClient.py
Project HelloClient -> Documents/SD_Labs/HelloClient.py
src HelloClient.py
HelloClient.py
HelloClient.java
External Libraries
Scratches and Consoles
HelloClient - HelloClient.py
if __name__ == "__main__":
    import socket
    if __name__ == "__main__":
        HOST, PORT = "localhost", 1234
        # create 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()
        # trimite mesaj
        print("Trimit mesaj catre microserviciu...")
        sock.sendall(b"Hello from client!")
        # primire raspuns
        received = str(sock.recv(1024), "utf-8")
        print("Raspuns de la HelloMicroservice: {}".format(received))
    if __name__ == "__main__":
        if __name__ == "__main__":
            if __name__ == "__main__":
                if __name__ == "__main__":
                    if __name__ == "__main__":
                        if __name__ == "__main__":
                            if __name__ == "__main__":
                                if __name__ == "__main__":
                                    if __name__ == "__main__":
                                        if __name__ == "__main__":
                                            if __name__ == "__main__":
                                                if __name__ == "__main__":
                                                    if __name__ == "__main__":
                                                        if __name__ == "__main__":
                                                            if __name__ == "__main__":
                                                                if __name__ == "__main__":
                                                                    if __name__ == "__main__":
                                                                        if __name__ == "__main__":
                                                                            if __name__ == "__main__":
                                                                                if __name__ == "__main__":
                                                                                    if __name__ == "__main__":
                                                                                        if __name__ == "__main__":
                                                                                            if __name__ == "__main__":
                                                                                                if __name__ == "__main__":
                                                                                                 if __name__ == "__main__":
                                                                                                     if __name__ == "__main__":
                                                                                                         if __name__ == "__main__":
                                                                                                             if __name__ == "__main__":
                                                                                                                 if __name__ == "__main__":
                                                                                                                   if __name__ == "__main__":
                                                                                                                     if __name__ == "__main__":
                                                                                                                       if __name__ == "__main__":
                                                                                                                         if __name__ == "__main__":
                                                                                                                           if __name__ == "__main__":
                                                                                                                             if __name__ == "__main__":
                                                                                                                               if __name__ == "__main__":
                                                                                                                                 if __name__ == "__main__":
                                                                                                                               if __name__ == "__main__":
                                                                                                                                 if __name__ == "__main__":
                                                                                                                               if __name__ == "__main__":
                                                                                                                                 if __name__ == "__main__":
                                                                                                                               if __name__ == "__main__":
                                                                                                                                 if __name__ == "__main__":
                                                                                                                               if __name__ == "__main__":
                                                                                                                               if __name__ == "__main__":
                                                                                                                               if __name__ == "__main__":
                                                                                                                               if __name__ == "__main__":
                                                                                                                               if __name__ == "__main__":
                                                                                                                               if __name__ == "__main__":
                                                                                                                               if __name__ == "__main__":
                                                                                                                               if __name__ == "__main__":
                                                                                                                               if __name__ == "__main__":
                                                                                                                               if __name__ == "__main__":
................................................................
Run HelloClient
Process finished with exit code 0
```

Figura 32 - Testarea microserviciului

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

```
docker logs 58fcf3028b70
```

```
student@debian:~$ docker logs 58fcf3028b70
Microserviciul se executa pe portul: 2000
Se asteapta conexiuni...
Client conectat: 172.17.0.1:60180
Client conectat: 172.17.0.1:60200
Client conectat: 172.17.0.1:60204
student@debian:~$
```

Figura 33 - 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 58fcf3028b70
```

```
student@debian:~$ docker stop 58fcf3028b70
58fcf3028b70
student@debian:~$
```

Figura 34 - 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
58fcf3028b70	localhost:5000/hello microservice:v1	"java -jar HelloMicr..."	6 minutes ago	Exited (143) 34 seconds ago	0.0.0.0:5000->5000/tcp, :::5000->5000/tcp	hungry_jang
b29fbcc0c0e	registry:2	"/entrypoint.sh /etc..."	25 minutes ago	Up 25 minutes		registrar_docker
45f195dd8	0804a3fe854	"/entrypoint.sh /etc..."	29 minutes ago	Created		bald_ramanujan
bc734398a868	0b60a3fe054	"/entrypoint.sh /etc..."	33 minutes ago	Created		blissful_agnesi

Figura 35 - Verificarea stării containerelor opriate

Pentru a reporni containerul, pur și simplu folosiți comanda:

```
docker start <NUME_SAU_ID_CONTAINER_OPRIT>
```

În acest caz:

```
docker start 58fcf3028b70
```

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

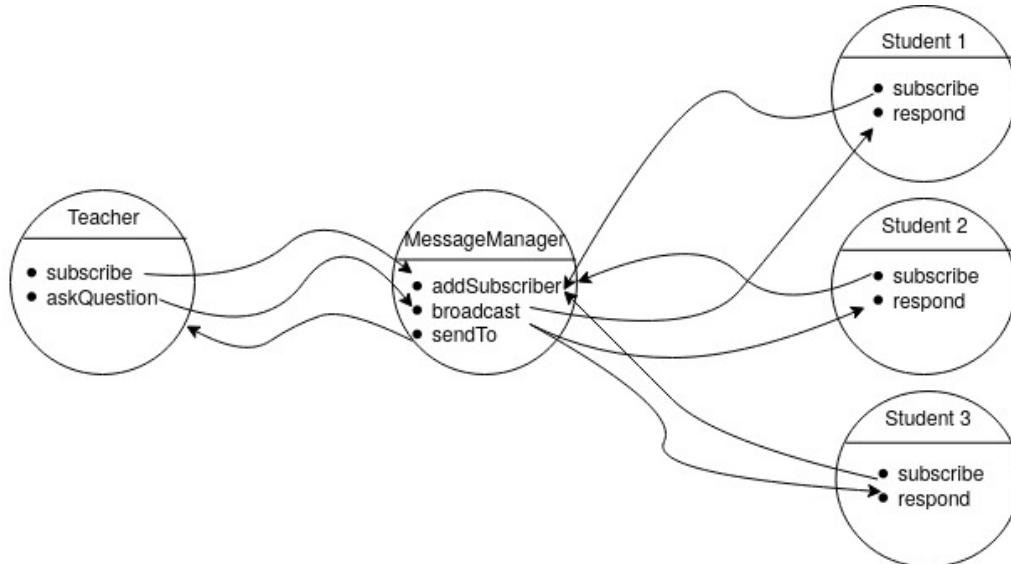


Figura 36 - Diagrama de microservicii

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:

întrebare <DESTINATAR_RĂSPUNS> <CONTINUT_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:

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 trimită înapoi destinatarului (adică microserviciului **Teacher**, în acest caz). Dacă niciun student nu știe răspunsul la întrebare, nu se trimită niciun răspuns, iar **Teacher** își va da seama că nu primește răspuns atunci când expiră un timp de dinainte stabilit.

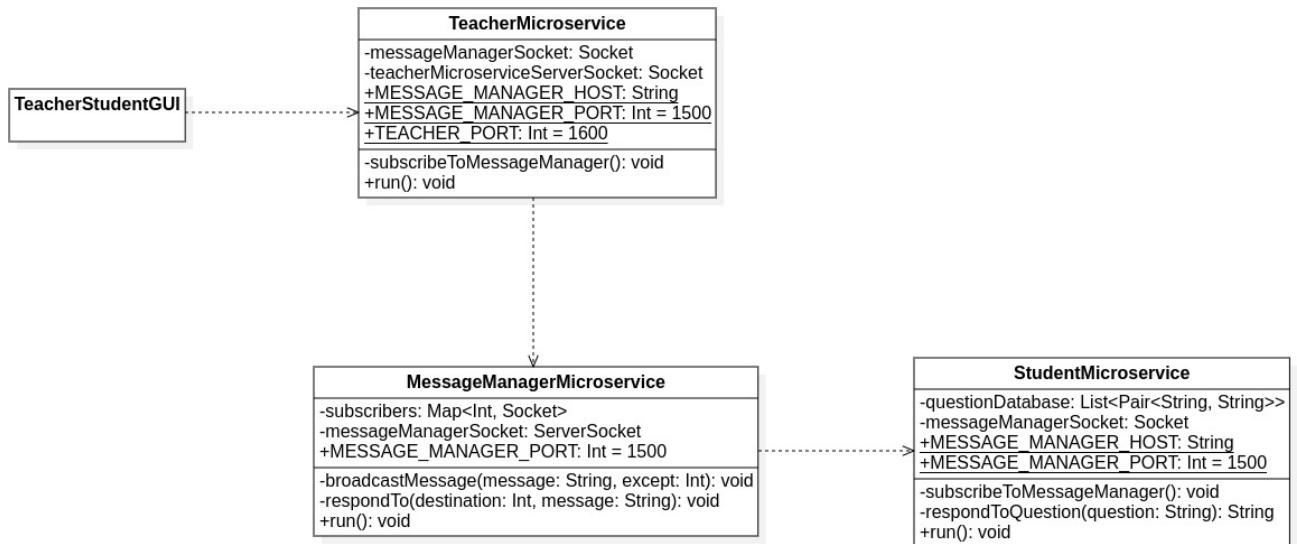


Figura 37 - Diagrama de clase

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()

        /*
         "bază 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 = bufferReader.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 rețea 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 rețea 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
            // 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 înschise 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 înschise.

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 =
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 messageManagerBufferedReader =
BufferedReader(InputStreamReader(messageManagerSocket.getInputStream()))
        try {
            val receivedResponse =
messageManagerBufferedReader.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_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 răspunda”.

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).

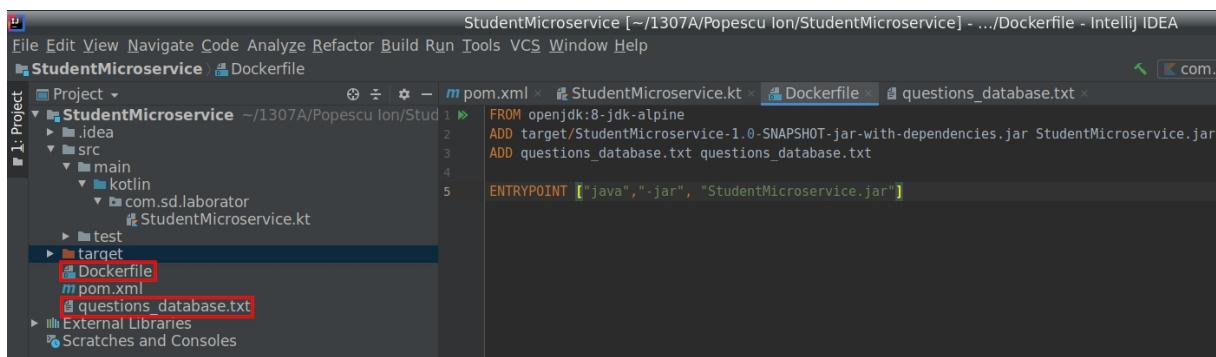


Figura 38 - Dockerfile și fișier cu date pentru StudentMicroservice

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:

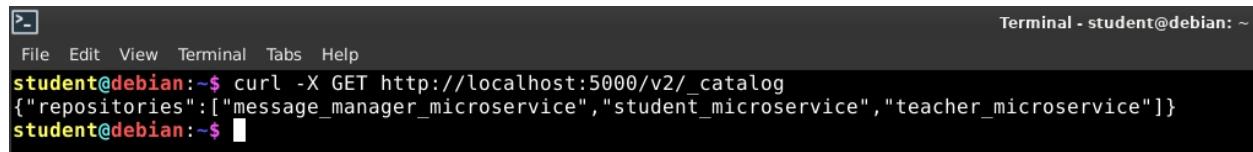
```
docker image ls
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
localhost:5000/teacher_microservice	v1	55fea4e89558	45 seconds ago	107MB
localhost:5000/message_manager_microservice	v1	cf668cf836a	About a minute ago	107MB
localhost:5000/student_microservice	tip3	76c4247f7667	3 minutes ago	107MB
localhost:5000/student_microservice	tip2	5a6cd1e6b5ae	4 minutes ago	107MB
localhost:5000/student_microservice	tip1	bad202e3ef66	5 minutes ago	107MB
localhost:5000/hello_microservice	v1	0615cae89f8c	About an hour ago	107MB
hello_microservice	v1	0615cae89f8c	About an hour ago	107MB
registry	2	b8604a3fe854	8 days ago	26.2MB
hello-world	latest	feb5d9fea6a5	8 weeks ago	13.3kB
openjdk	8-jdk-alpine	a3562aa0b991	2 years ago	105MB

Figura 39 - 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
student@debian:~$ curl -X GET http://localhost:5000/v2/_catalog
{"repositories":["message_manager_microservice","student_microservice","teacher_microservice"]}
student@debian:~$ █
```

Figura 40 - Verificarea disponibilității imaginilor Docker în registrul privat

Implementarea interfeței grafice

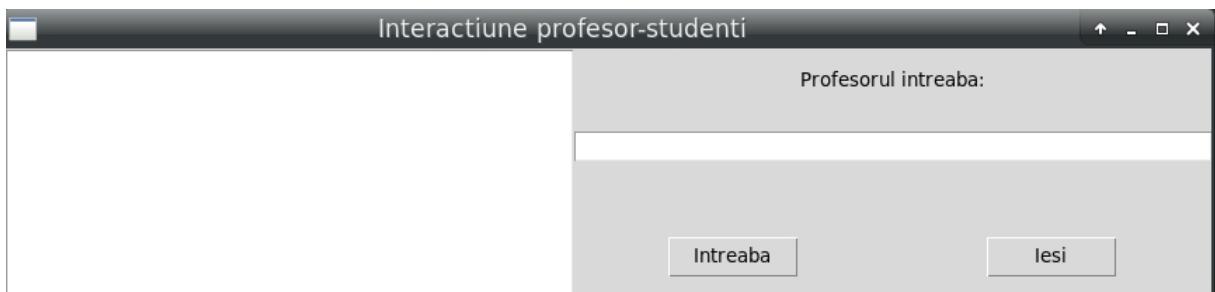


Figura 41 - Interfață grafică

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 intrebarii 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 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, 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 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 creați o rețea virtuală Docker, folosind următoarea comandă:

```
docker network create ms-net
```

```
student@debian:~$ docker network create ms-net
3c7eed0b43ed4d44f50268620a4b2d1bb6c6024b0ea396fdaac1f75463e149ff
student@debian:~$
```

Figura 42 - 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.

```
student@debian:~$ docker run -d -p 1500:1500 --name message_manager --network=ms-net localhost:5000/message_manager_microservice:v1
01a41956964ce71983dda77a123996f1c9e9f05969abaa87e56c6d7154ec2dd2
student@debian:~$ docker logs 01a41956964ce71983dda77a123996f1c9e9f05969abaa87e56c6d7154ec2dd2
MessageManagerMicroservice se executa pe portul: 1500
Se asteapta conexiuni si mesaje...
student@debian:~$
```

Figura 43 - 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 trimit 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.

```
Terminal - student@debian: ~
student@debian:~$ docker run -d -p 1600:1600 -e MESSAGE_MANAGER_HOST='message_manager' --name teacher_microservice --network=ms-net
localhost:5000/teacher_microservice:v1
bad0e86d75bf11fc4ad0ea6211985fe781c1c355f11b072bea0869953d989e2a
student@debian:~$ docker logs bad0e86d75bf11fc4ad0ea6211985fe781c1c355f11b072bea0869953d989e2a
M-am conectat la MessageManager!
TeacherMicroservice se executa pe portul: 1600
Se asteapta cereri (intrebari)...
student@debian:~$
```

Figura 44 - 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
```

```
Terminal - student@debian: ~
File Edit View Terminal Tabs Help
student@debian:~$ docker run -d -e MESSAGE_MANAGER_HOST='message_manager' --name student_microservice_1 --network=ms-net localhost:5000/student_microservice:tip1
3fe0c9ef6d59891cba0b64fa84171f0cde5b88fd5981aa7c03ed9be23b13684
student@debian:~$ docker run -d -e MESSAGE_MANAGER_HOST='message_manager' --name student_microservice_2 --network=ms-net localhost:5000/student_microservice:tip2
063aaad34f53712c233283b9e83553e48c947f196031113dfad03d111965eda
student@debian:~$ docker run -d -e MESSAGE_MANAGER_HOST='message_manager' --name student_microservice_3 --network=ms-net localhost:5000/student_microservice:tip3
d28513dd857a073df2267f927a9947aeae8a6bc4a91b248953d5f141dbd2eab74
student@debian:~$ docker logs 063aaad34f53712c233283b9e83553e48c947f196031113dfad03d111965eda
M-am conectat la MessageManager!
StudentMicroservice se executa pe portul: 53234
Se asteapta mesaje...
student@debian:~$ docker logs 063aaad34f53712c233283b9e83553e48c947f196031113dfad03d111965eda
M-am conectat la MessageManager!
StudentMicroservice se executa pe portul: 38004
Se asteapta mesaje...
student@debian:~$ docker logs d28513dd857a073df2267f927a9947aeae8a6bc4a91b248953d5f141dbd2eab74
M-am conectat la MessageManager!
StudentMicroservice se executa pe portul: 41560
Se asteapta mesaje...
student@debian:~$
```

Figura 45 - 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>
```

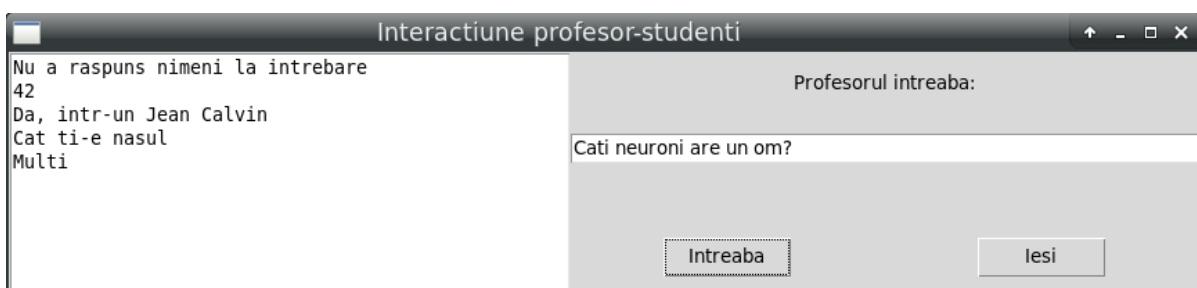
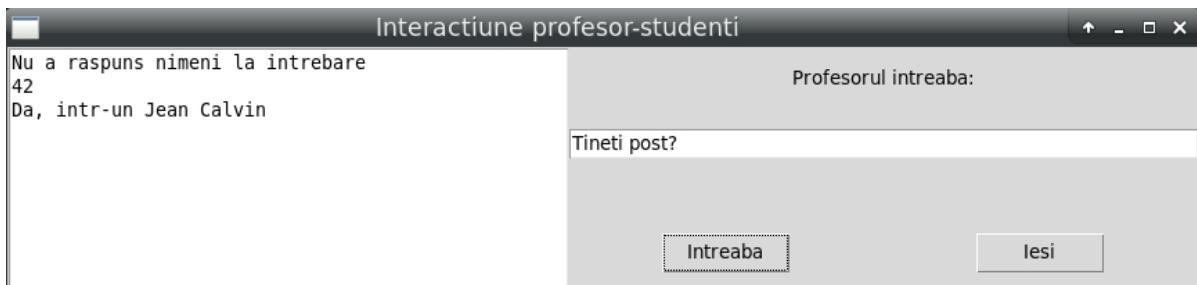
În acest moment, ar trebui să aveți 5 containere Docker pornite, fiecare încapsulând câte un microserviciu. Verificați acest lucru folosind comanda:

```
docker ps
```

```
Terminal - student@debian: ~
File Edit View Terminal Tabs Help
student@debian:~$ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED            STATUS              PORTS                               NAMES
d28513dd857a        localhost:5000/student_microservice:tip3   "java -jar StudentMi..."   About a minute ago   Up About a minute          student_microser
vice_3              localhost:5000/student_microservice:tip2   "java -jar StudentMi..."   About a minute ago   Up About a minute          student_microser
vice_2              localhost:5000/student_microservice:tip1   "java -jar StudentMi..."   About a minute ago   Up About a minute          student_microser
063aaad34f53       localhost:5000/teacher_microservice:v1    "java -jar TeacherMi..."  About an hour ago   Up About an hour          teacher_microser
vice_1              localhost:5000/message_manager_microservice:v1 "java -jar MessageMa..."  About an hour ago   Up About an hour          message_manager
58fcf3028b70        localhost:5000/hello_microservice:v1     "java -jar HelloMicr..."  2 hours ago         Up 2 hours              0.0.0.0:1234->2000/tcp, 0.0.0.0:5000->5000/tcp   hungry_jang
b3e070c6ca3         registry:2                           "/entrypoint.sh /etc..."  2 hours ago         Up 2 hours              0.0.0.0:5000->5000/tcp   registru_docker
student@debian:~$
```

Figura 46 - Execuția tuturor celor 5 microservicii încapsulate în container 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:

```

File Edit View Terminal Tabs Help
student@debian:~$ docker logs 01a41956964c
MessageManagerMicroservice se executa pe portul: 1500
Se asteapta conexiuni si mesaje...
Subsriber conectat: 172.18.0.3:35390
Subsriber conectat: 172.18.0.4:53188
Subsriber conectat: 172.18.0.5:37964
Subsriber conectat: 172.18.0.6:41522
Subsriber-ul 37964 a fost deconectat.
Subsriber-ul 53188 a fost deconectat.
Subsriber-ul 41522 a fost deconectat.
Subsriber conectat: 172.18.0.4:53234
Subsriber conectat: 172.18.0.5:38004
Subsriber conectat: 172.18.0.6:41560
Primit mesaj: intrebare 35390 Tineti post?
Primit mesaj: raspuns 35390 Da, intr-un Jean Calvin
Primit mesaj: intrebare 35390 Tineti post?
Primit mesaj: raspuns 35390 Da, intr-un Jean Calvin
Primit mesaj: intrebare 35390 test
Primit mesaj: intrebare 35390 Care e sensul vietii?
Primit mesaj: intrebare 35390 Tineti post?
Primit mesaj: raspuns 35390 Da, intr-un Jean Calvin
Primit mesaj: intrebare 35390 Cati neuroni are un om?
Primit mesaj: intrebare 35390 Unde se da al 3-lea razboi mondial?
Primit mesaj: raspuns 35390 Pe Facebook
student@debian:~$ 

```

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. [optională] Încercați să folosiți mecanismele docker pentru urmărirea funcționării unei instante userv și realizați o analiză comparativă SWOT care din cele două abordări vi s-au parut mai bune.

4. 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*).

5. modificați microserviciu profesor astfel încât să pună și note care vor fi depuse într-o bază de date separată centralizată (deci propriul ei microserviciu) pentru fiecare student. La incheierea lucrului cu un student se va genera un eveniment care va conduce la calculul notei ca medie a tuturor notelor la răspunsurile puse de asistent sau profesor. Deci suportă număr variabil de întrebări

6. Introduceți un userviciu asistent care pe lângă atribuții de evaluare și notare verifică dacă studentul este chiar cine spune (se presupune că a primit un cod unic care se găseste în baza de date și atunci prima întrebare este de la asistent care verifica acest cod. Dacă codul este greșit microserviciul este scos din lista userv acceptate să comunice cu profesorul și cu asistentul

7. se introduce un microserviciu care monitorizează comunicarea și nu permite ca aceasta să se desfășoare decât dacă id-ul student se află în lista lui (introdusă de asistent)

Se vor desena diagramele de servicii, UML, se pot crea noi uservicii dacă vi se pare că sunt SOLID pentru userv încălcate. Se verifică dacă SOLID pentru userv sunt respectate.

BONUS - Puteți utiliza inclusiv strat de instrumentație dacă îi justificați necesitatea (evidenț mai adăugați noi funcționalități - adică faceți o analiză mai largă și reproiectați de la zero)

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>