Sisteme Distribuite - Laborator 7

Microservicii reactive cu Kotlin

Descriere generală

Microserviciile reactive se bazează pe **paradigma de programare reactivă**, ţinta lor fiind să ofere viteza de reacţie bună şi uniformă (engl. *responsiveness*), rezilienţă (engl. *resiliency*) - vezi modelare şi simulare precum şi orientare pe mesaje asincrone (engl. *message-driven*).

Diferenţa faţă de programarea imperativă

Într-o aplicaţie scrisă utilizând paradigma imperativă, dezvoltatorul scrie codul astfel încât o instrucţiune cere un anumit lucru, apoi aşteaptă rezultatul a ceea ce a cerut. În timpul în care rezultatul este aşteptat, aplicaţia este **blocată**.

Să exemplificăm printr-un bloc de cod Kotlin:

```
var x = computePI(nrZecimale = 10000)
println("Numarul PI cu 10.000 de zecimale este: ${x}")
```

Se observă că pe prima linie, se cere calculul numărului π cu 10.000 de zecimale. Rezultatul apelului funcţiei **`computePI()`** nu este disponibil imediat, ci doar după un anumit timp (în funcţie de parametrul trimis). Aşadar, variabila **`x`** nu va fi populată instant cu valoarea returnată de funcţie, deci programul **se va bloca pe linia 1** până când rezultatul produs de funcţie va fi disponibil pentru atribuire. Abia după aceea se va executa şi linia a 2-a, care are nevoie de variabila **`x`** pentru afişare.

Cea mai importantă parte a acestui bloc de cod este faptul că programul Kotlin se blochează până datele de care are nevoie sunt disponibile complet, deci spunem că metoda **`computePI`** este **blocantă**.

Varianta utilizând programarea reactivă este următoarea:

```
subscribe(::computePI, args=arrayOf(10000)).whenDone(::println)
```

(**exemplul de mai sus este didactic, implementarea efectivă diferă!**)

Semnificaţia liniei de cod de mai sus este următoarea: programul se înscrie (engl. *subscribes*) unei metode, iar când execuţia acelei metode este încheiată, rezultatul este trimis unei alte metode. În acest exemplu, când numărul PI este calculat, se apelează metoda **`println()`** şi se afişează rezultatul.

Semnificativ pentru cel de-al doilea exemplu este faptul că după instrucţiunea de mai sus, programul îşi continuă execuţia, deci va putea rula alte instrucţiuni în continuare. Acest tip de instrucţiune se numeşte **neblocantă**.

Fluxul de date reactiv

De asemenea, programul se poate înscrie (*subscribe*) nu numai unui singur rezultat, ci unui întreg flux de date (engl. *data stream*) rezultate din anumite prelucrări. Când fluxul de date îşi începe emisia, metoda la care programul s-a înscris va fi apelată progresiv, pentru datele din flux. În acest caz, dacă vom considera fluxul de date:

```
10, 123, 500, 593, 1026, 1043, 493, 209, 4000, 412000, ...
```

aplicaţia va primi aceste numere întregi pe rând şi va apela metoda **`computePI()`** progresiv, cu câte un parametru preluat din fluxul de date. Când calculul se încheie pentru un anumit

parametru, se apelează metoda **`println()`** care afişează rezultatul obţinut conform parametrului curent:

1. S-a primit numărul **`10`** în fluxul de date → apelez **`computePI(10)`** → s-a încheiat execuţia **`computePI(10)`**, apelez **`println(rezultat)`**
2. S-a primit numărul **`123`** în fluxul de date → apelez **`computePI(123)`** → s-a încheiat execuţia **`computePI(123)`**, apelez **`println(rezultat)`**
3. S-a primit numărul **`500`** în fluxul de date → apelez **`computePI(500)`** → s-a încheiat execuţia **`computePI(500)`**, apelez **`println(rezultat)`**

...
ş.a.m.d.

Aşadar, un **flux de date reactiv** este o colecţie de date emisă încontinuu, pe măsură ce datele sunt pregătite. De exemplu, în loc să cereţi anumite câmpuri dintr-o bază de date şi să aşteptaţi rezultatul, baza de date începe să trimită rezultate pe măsură ce acestea sunt pregătite.

Acest nou model de programare permite obţinerea de performanţe mai mari în aplicaţiile dezvoltate, deoarece se pot procesa mai multe cereri decât în modelul tradiţional, blocant. Această abordare foloseşte resursele mult mai eficient şi acest lucru ar putea reduce inclusiv necesarul de infrastructură pentru aplicaţii.

Principiile programării reactive

- **percepția pozitivă a utilizatorului cu privire la viteza de reacție a aplicației** (*responsiveness*) - aplicaţiile moderne ar trebui să răspundă cererilor în timp util, dar nu numai utilizatorilor care le utilizează, ci şi rezolvarea problemelor şi recuperarea după apariţia erorilor trebuie să se conformeze constrângerilor de timp;
- **reziliența** (*resilience*) - realizabilă şi prin replicare, care la rândul ei depinde de scalabilitatea sistemului
- **elasticitatea** (*elasticity*) - păstrarea constantă a unei viteze de răspuns în caz d emare încărcare. Astfel sistemele reactive trebuie să fie elastice, astfel încât să se poată adapta sub diverse grade de încărcare (exemplu: număr mare de cereri), replicând resursele disponibile în funcţie de nevoie
- **orientare spre mesaje** asincrone (*message-driven*) - sistemele reactive folosesc mesaje asincrone pentru a transmite informaţia prin diverse componente, având cuplare foarte slabă ce permite interconectarea acestor sisteme în izolare
- **supra-saturarea fluxurilor** (*back-pressure*) - se produce atunci când un sistem reactiv publică mesaje într-un ritm mai alert decât pot fi gestionate de entităţile înscrise pentru a primi mesajele. Acest fenomen apare dacă nu se utilizează abordări moderne în sistemele de cozi.

Pentru mai multe detalii despre programarea reactivă şi principiile sale, consultaţi „The Reactive Manifesto, disponibil la următorul URL:

https://github.com/reactivemanifesto/reactivemanifesto

Biblioteca RxKotlin

În acest laborator veţi utiliza biblioteca **RxKotlin** pentru implementarea microserviciilor reactive.

RxKotlin este o bibliotecă care extinde **RxJava**, adăugând funcţionalităţi noi şi metode de tip extensie care sporesc productivitatea lucrului cu programarea reactivă în Kotlin.

RxJava este o bibliotecă utilizată pentru lucrul cu cod asincron, bazat pe evenimente,

care foloseşte secvenţe observabile şi operatori în stil funcţional, permiţând execuţii parametrizate prin planificatori (engl. *schedulers*).

Observables

Un obiect **`Observable`** este un fel de secvenţă de valori (mesaje), cu câteva proprietăţi speciale. Una din ele, poate chiar cea mai importantă, este că secvenţa este asincronă. Obiectele **`Observable`** produc evenimente, proces denumit ca şi **emitere** (engl. *emitting*), într-o anumită perioadă de timp. Evenimentele pot conţine valori, precum numere, sau instanţe ale unui tip personalizat de date. De asemenea, evenimentele pot fi rezultatul unor gesturi, cum ar fi apăsarea de butoane.

Un obiect **`Observable`** emite câte un eveniment (**`next`**), până când:

- se produce o eroare, şi atunci se emite **`error`**, iar **`Observable`**-ul se încheie
- se încheie fluxul de date, şi atunci se emite un eveniment de tip **`complete`**

Exemplu simplu de obiect **`Observable`**:

```
val observable = Observable.fromIterable(listOf(1, 2, 3))
```

Acest obiect va putea emite date de tip număr întreg, într-un flux de 3 elemente. Dacă nu există niciun client înscris la acest flux (engl. *subscriber*), nu se emite nicio valoare. Aşadar, se creează un *subscriber*, astfel:

```
observable.subscribeBy (onNext = {
        println(it)
    }, onComplete = {
        println("Completed!")
    }, onError = {
        println("Error: $it!"
    }
)
```

Codul de mai sus afişează:

```
1
2
3
Completed!
```

Constructorul de flux reactiv **`fromIterable`** este un exemplu cu care se poate prelua un flux de date dintr-o colecţie existentă. Metoda **`subscribeBy()`** primeşte ca parametrii 3 funcţii lambda corespunzătoare celor 3 evenimente posibile emise din fluxul reactiv: **`onNext`**, **`onComplete`** şi **`onError`**.

Un alt exemplu de flux reactiv predefinit:

```
val observable = Observable.range(1, 10)

observable.subscribe{
    val n = it.toDouble()
    val fibonacci = ((Math.pow(1.61803, n) - Math.pow(0.61803, n)) /
            2.23606).roundToInt()
    println(fibonacci)
}
```

În cazul acestui flux, *subscriber*-ul este interesat doar de emiterea următorului element,

nu şi de cazurile de eroare, respectiv de evenimentul emis când fluxul s-a încheiat. Metoda **`subscribe()`** primeşte o funcţie lambda care tratează evenimentul **`next()`**.

În practică, se folosesc fluxuri definite personalizat, în funcţie de caz.

Exemplu de creare a unui flux reactiv personalizat:

```
val someErrorHappened = True
val observable = Observable.create<String> { emitter ->
    emitter.onNext("1")
    emitter.onNext("2")
    if (someErrorHappened)
        emitter.onError(RuntimeException("Error"))
    else
        emitter.onComplete()
}

val subscription = observable.subscribeBy (
        onNext = { println(it) },
        onComplete = { println("Completed") },
        onError = { println(it) }
    ))

subscription.dispose()
```

În acest exemplu se creează un obiect **`Observable`** personalizat care emite 2 şiruri de caractere în flux, iar apoi, în funcţie de o condiţie, emite eroare sau termină fluxul în mod obişnuit. De asemenea, obiectul **`Subscription`** returnat de metoda **`subscribeBy`** este capturat şi utilizat pentru a elibera memoria utilizată, prin apelul metodei **`dispose()`** asupra acestuia.

Subscriber-ul la acest flux va primi următoarele:

```
1
2
java.lang.RuntimeException: Error
```

Pentru alte tipuri de obiecte corespunzătoare paradigmei reactive, precum şi alte exemple, consultaţi cartea **Reactive Programming with Kotlin**, de Alex Sullivan.

Aplicaţie exemplu - Okazii

Se cere implementarea unei aplicaţii bazate pe microservicii reactive, care să modeleze comportamentul ofertanţilor participanţi la o licitaţie, împreună cu procesarea tuturor ofertelor primite şi calcularea rezultatului.

Diagrama de microservicii

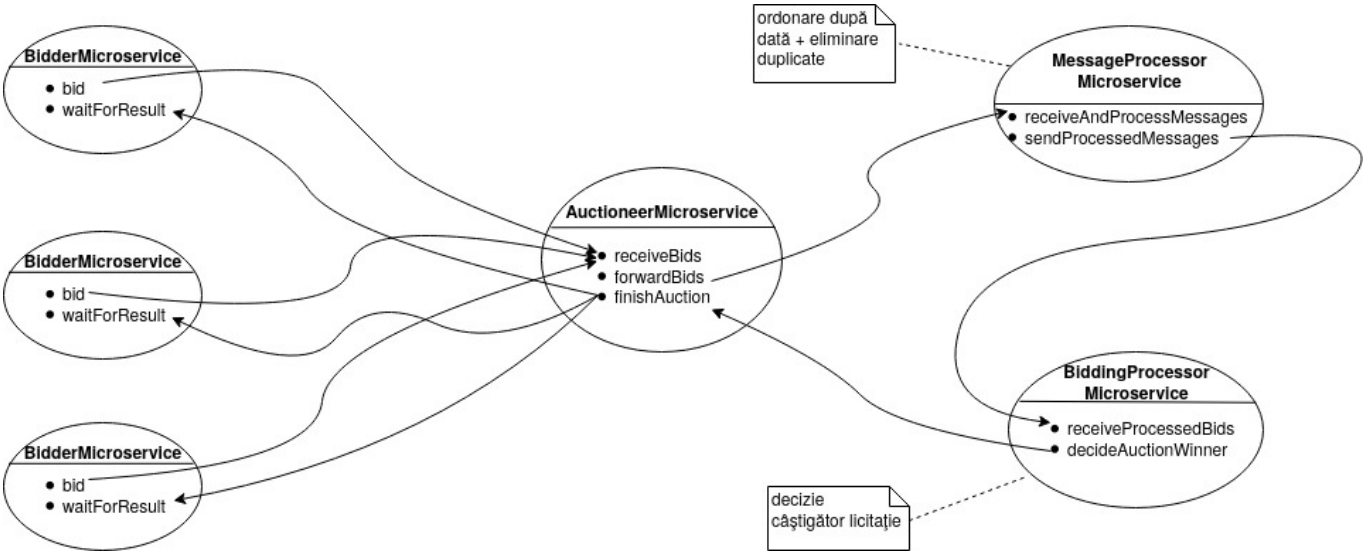


Figura 1 - Diagrama de microservicii

Diagrama de clase

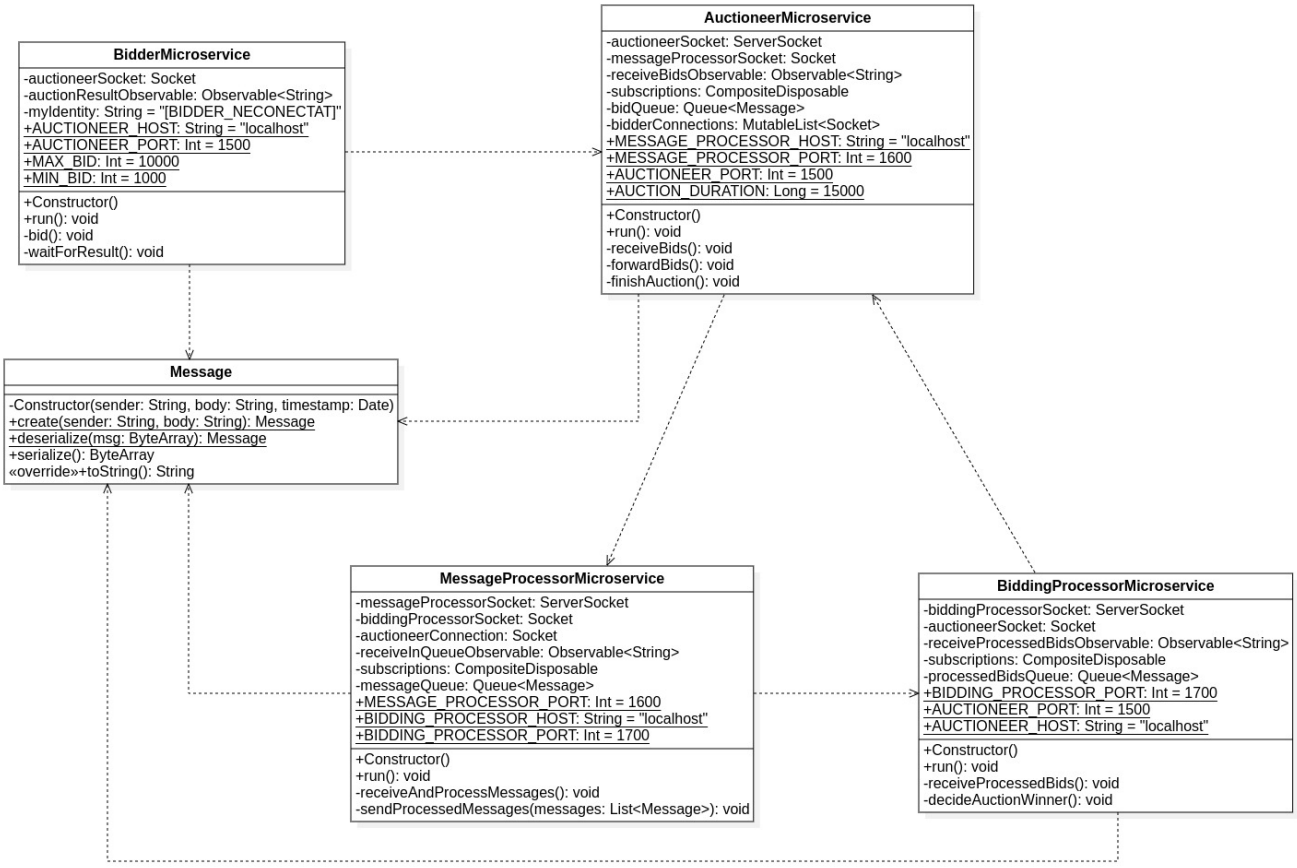


Figura 2 - Diagrama de clase

Semnificaţiile microserviciilor

• **BidderMicroservice** - modelează un ofertant participant la licitaţie. Se conectează la `AuctioneerMicroservice` şi trimite un mesaj de forma:

```
licitez <SUMA_LICITATĂ>
```

unde **SUMA_LICITATĂ** este un număr întreg între 1000 şi 10.000, ales aleator. Aceasta reprezintă oferta participantului respectiv. După ce trimite mesajul iniţial, aşteaptă răspuns de la **AuctioneerMicroservice** pentru a primi rezultatul licitaţiei.

• **AuctioneerMicroservice** - reprezintă punctul central al licitaţiei care primeşte toate ofertele şi **încheie licitaţia după un timp dinainte stabilit** (în exemplul din laborator, **după 15 secunde**). Mesajele pe care le primeşte sunt reţinute într-o coadă, acestea fiind transmise mai departe spre procesare, către **MessageProcessorMicroservice**.

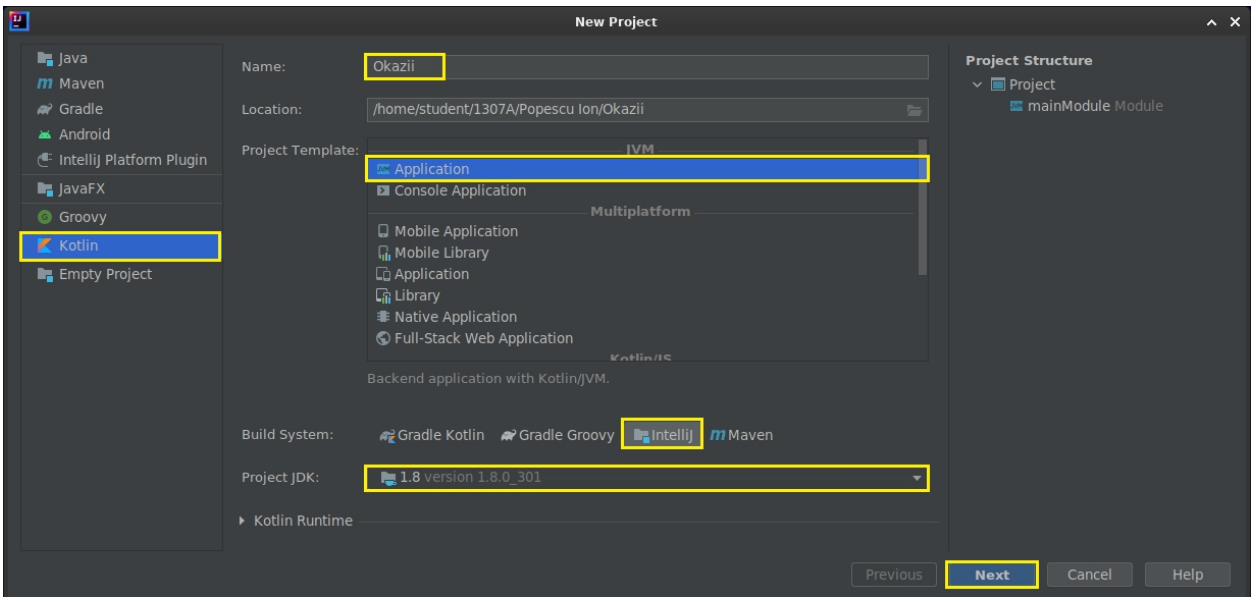
După aceea, aşteaptă rezultatul licitaţiei de la **BiddingProcessorMicroservice** pentru a-l transmite ofertanţilor.

• **MessageProcessorMicroservice** - are rol de procesare a tuturor mesajelor pe care le primeşte: elimină duplicatele şi ordonează mesajele după dată (**de implementat la laborator**). **Se consideră că ceasurile maşinilor de calcul pe care se execută microserviciile componente sunt sincronizate**.

• **BiddingProcessorMicroservice** - decide cine câştigă licitaţia în funcţie de ofertele primite de la **MessageProcessorMicroservice**. Anunţă câştigătorul trimiţând mesajul ofertantului care a câştigat către **AuctioneerMicroservice**.

Implementarea aplicaţiei exemplu

Creaţi un proiect Kotlin/JVM folosind IntelliJ IDEA, denumit, spre exemplu, **Okazii**. **Nu folosiţi gestionare de proiecte (Maven / Gradle) sau arhetipuri**. Nu sunt necesare pentru acest laborator.



Adăugaţi 5 noi module în proiect:

1. **AuctioneerMicroservice**
2. **BidderMicroservice**
3. **BiddingProcessorMicroservice**
4. **MessageLibrary**
5. **MessageProcessorMicroservice**

Pentru a adăuga un modul, se apasă dreapta pe numele proiectului, **Okazii**, din panoul din stânga → **New** → **Module...**. Se selectează **Kotlin** în lista din stânga şi **JVM | IDEA** în panoul care apare în centru → **Next** → Completaţi câmpul „**Module name**" din partea de sus cu numele modulului, conform listei de mai sus şi apăsaţi „**Finish**".

Atenţie: fiecare microserviciu reprezintă un modul separat în proiect.

Structura proiectului ar trebui să arate ca în figură:

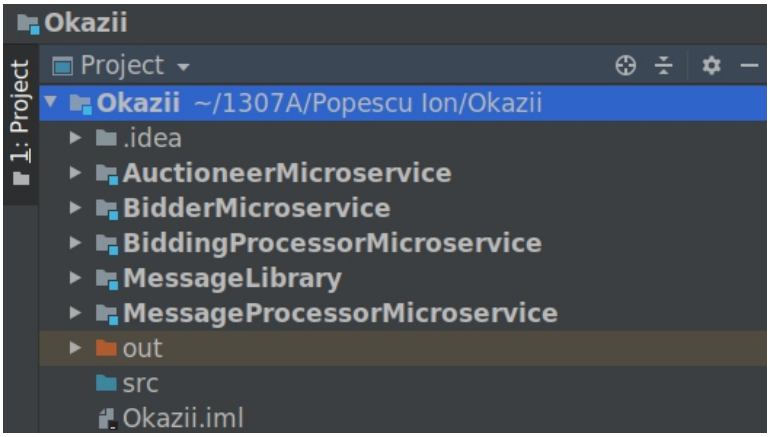
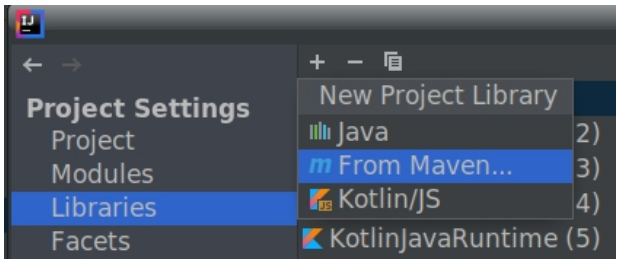


Figura 3 - Structura proiectului

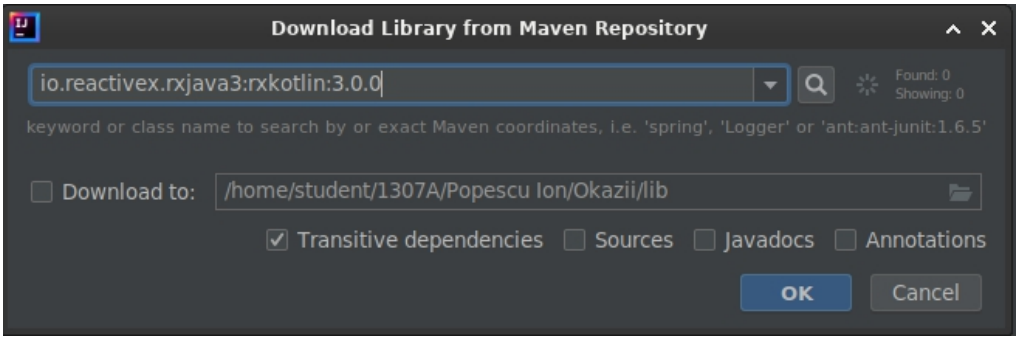
Pentru modulele ce implementează cele 4 microservicii, veţi avea nevoie de biblioteca RxKotlin adăugată ca şi dependenţă:

File → Project Structure... → Libraries → click pe pictograma „+" → **From Maven**.

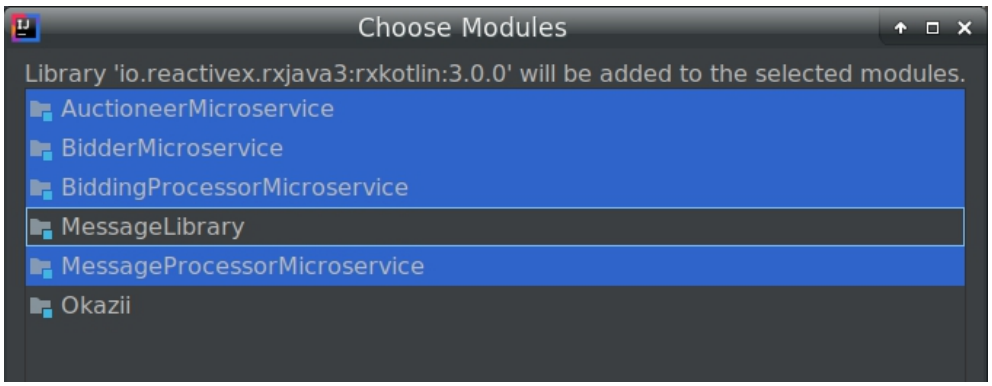


Introduceţi următoarele în căsuţa de căutare şi apăsaţi „**Ok**":

```
io.reactivex.rxjava3:rxkotlin:3.0.0
```



Alegeţi cele 4 module corespunzătoare celor 4 microservicii în următoarea fereastră care apare („**Choose modules**").



În continuare, începeți implementarea cu modulul **`MessageLibrary`**, deoarece toate celelalte depind de acesta. Acest modul conține formatul de mesaj care circulă între microservicii, ce pune la dispoziție metode de serializare / deserializare și de creare a mesajului pe baza emițătorului și a corpului, elemente date ca parametri.

- **Message.kt**

```
import java.text.SimpleDateFormat
import java.util.*

class Message private constructor(val sender: String, val body: String,
val timestamp: Date) {
    companion object {
        fun create(sender: String, body: String): Message {
            return Message(sender, body, Date())
        }

        fun deserialize(msg: ByteArray): Message {
            val msgString = String(msg)
            val (timestamp, sender, body) = msgString.split(' ', limit
= 3)

            return Message(sender, body, Date(timestamp.toLong()))
        }
    }

    fun serialize(): ByteArray {
        return "${timestamp.time} $sender $body\n".toByteArray()
    }

    override fun toString(): String {
        val dateString = SimpleDateFormat("dd-MM-yyyy
HH:mm:ss").format(timestamp)
        return "[$dateString] $sender >>> $body"
    }
}

fun main(args: Array<String>) {
    val msg = Message.create("localhost:4848", "test mesaj")
    println(msg)
    val serialized = msg.serialize()
    val deserialized = Message.deserialize(serialized)
    println(deserialized)
}
```

Pentru fiecare modul în parte, activaţi împachetarea automată sub formă de artefact JAR: **File → Project Structure... → Artifacts**. Apăsaţi pe pictograma „+” → Selectaţi **JAR → From modules with dependencies...** .

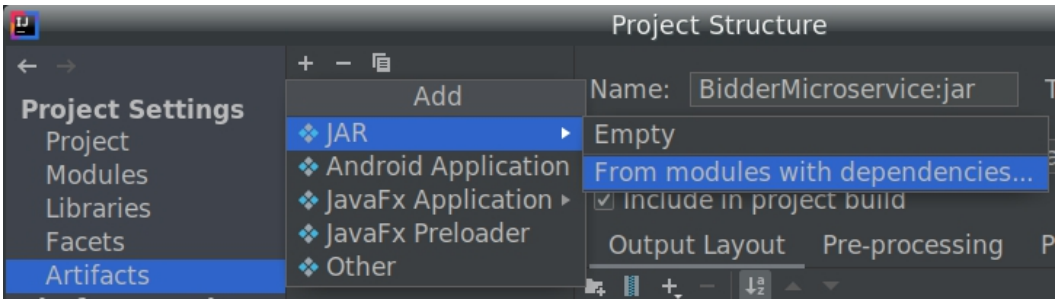


Figura 4 - Adăugare împachetare sub formă de artefact JAR

La **Module** selectaţi numele modulului din listă, apoi apăsaţi pe pictograma folder de la secţiunea „**Main Class:**” şi selectaţi clasa corespunzătoare modulului respectiv.

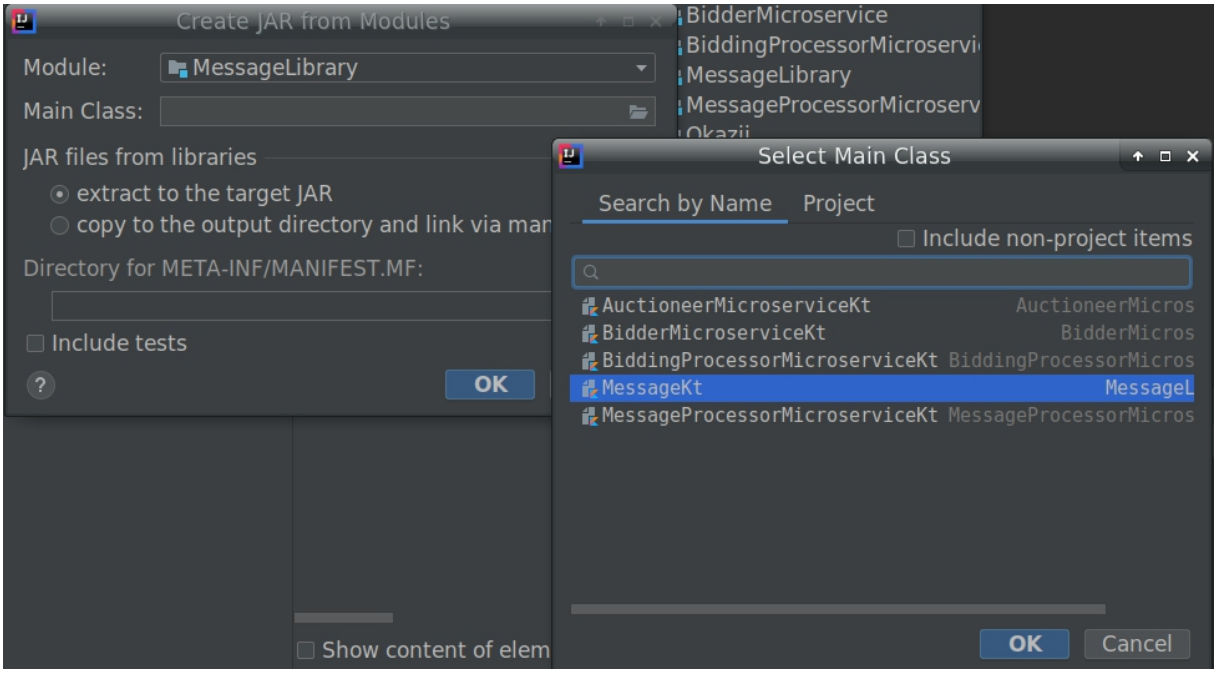
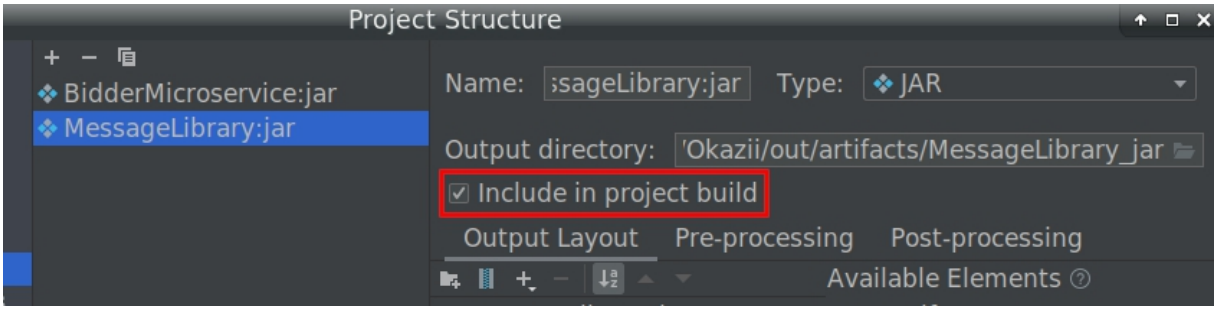


Figura 5 - Configurare împachetare JAR din modul IntelliJ

Apăsaţi „**Ok**”, apoi iar „**Ok**”. **Nu uitaţi să bifaţi „Include in project build” în panoul cu setările artefactului pe care l-aţi configurat**:

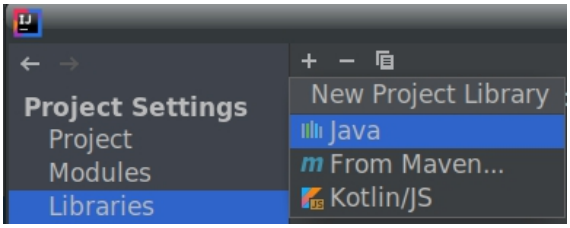


Compilaţi modulul **`MessageLibrary`** (click dreapta pe el → **Build module**

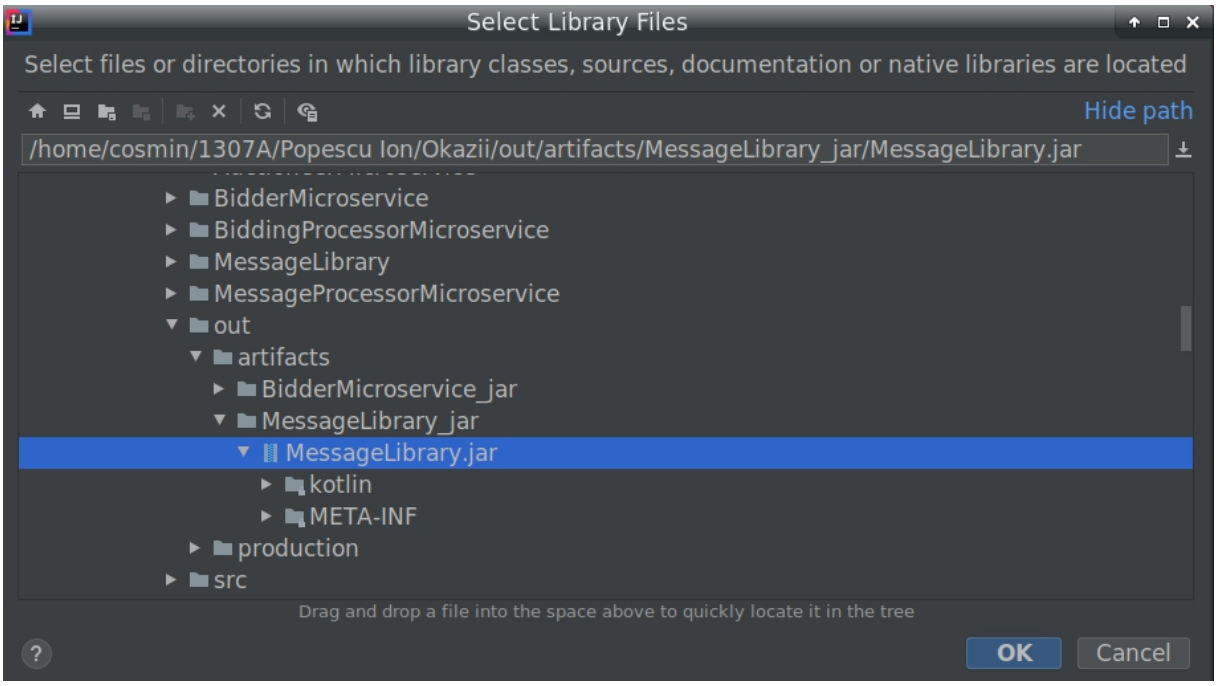
„MessageLibrary”). Va rezulta un fişier JAR în folder-ul **`out/artifacts/MessageLibrary_jar`**, denumit **`MessageLibrary.jar`**.

Adăugaţi `MessageLibrary.jar` ca bibliotecă dependentă pentru celelalte module ale proiectului:

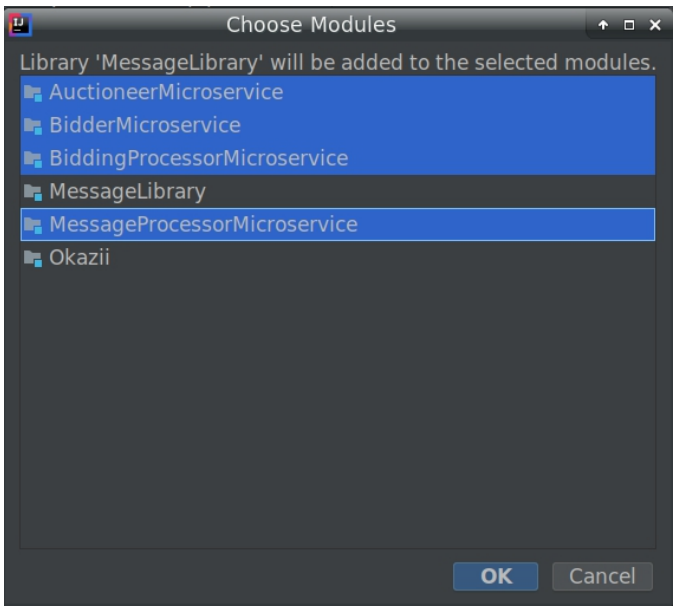
File → Project Structure... → Libraries → click pe pictograma „+” → selectaţi **Java**



În fereastra de navigare, selectaţi artefactul JAR denumit **`MessageLibrary.jar`** rezultat după compilarea modulului **`MessageLibrary`**.



În fereastra „**Choose modules**” care apare, selectaţi celelalte 4 module ale proiectului, conform cu figura următoare, şi apăsaţi „**Ok**”:



În continuare, se implementează fiecare din cele 4 microservicii componente. **Acestea comunică prin socket-uri TCP, iar fiecare mesaj primit / trimis prin aceste socket-uri reprezintă un element într-un flux reactiv.**

Codul conține comentarii explicative pe care le puteți consulta pentru a înțelege structura fișierelor sursă care urmează. S-au evidențiat în cod evenimentele de primire a unui nou mesaj în fluxul reactiv (`next`), de terminare a fluxului (`complete`), respectiv de întâmpinare a unei erori (`error`).

- **BidderMicroservice.kt**

```
import io.reactivex.rxjava3.core.Observable
import io.reactivex.rxjava3.kotlin.subscribeBy
import java.io.BufferedReader
import java.io.InputStreamReader
import java.net.Socket
import kotlin.Exception
import kotlin.random.Random
import kotlin.system.exitProcess

class BidderMicroservice {
    private var auctioneerSocket: Socket
    private var auctionResultObservable: Observable<String>
    private var myIdentity: String = "[BIDDER_NECONECTAT]"

    companion object Constants {
        const val AUCTIONEER_HOST = "localhost"
        const val AUCTIONEER_PORT = 1500
        const val MAX_BID = 10_000
        const val MIN_BID = 1_000
    }

    init {
        try {
            auctioneerSocket = Socket(AUCTIONEER_HOST, AUCTIONEER_PORT)
            println("M-am conectat la Auctioneer!")
```

```
            myIdentity = "[${auctioneerSocket.localPort}]"

            // se creeaza un obiect Observable ce va emite mesaje
primite printr-un TCP
            // fiecare mesaj primit reprezinta un element al fluxului
de date reactiv
            auctionResultObservable = Observable.create<String>
{ emitter ->
                // se citeste raspunsul de pe socketul TCP
                val bufferReader =
BufferedReader(InputStreamReader(auctioneerSocket.inputStream))
                val receivedMessage = bufferReader.readLine()

                // daca se primeste un mesaj gol (NULL), atunci
inseamna ca cealalta parte a socket-ului a fost inchisa
                if (receivedMessage == null) {
                    bufferReader.close()
                    auctioneerSocket.close()

                    emitter.onError(Exception("AuctioneerMicroservice
s-a deconectat."))
                    return@create
                }

                // mesajul primit este emis in flux
                emitter.onNext(receivedMessage)

                // deoarece se asteapta un singur mesaj, in continuare
se emite semnalul de incheiere al fluxului
                emitter.onComplete()

                bufferReader.close()
                auctioneerSocket.close()
            }
        } catch (e: Exception) {
            println("$myIdentity Nu ma pot conecta la Auctioneer!")
            exitProcess(1)
        }
    }

    private fun bid() {
        // se genereaza o oferta aleatorie din partea bidderului
curent
        val pret = Random.nextInt(MIN_BID, MAX_BID)

        // se creeaza mesajul care incapsuleaza oferta
        val biddingMessage =
Message.create("${auctioneerSocket.localAddress}:${auctioneerSocket.lo
calPort}",
            "licitez $pret")

        // bidder-ul trimite pretul pentru care doreste sa liciteze
        val serializedMessage = biddingMessage.serialize()
        auctioneerSocket.getOutputStream().write(serializedMessage)

        // exista o sansa din 2 ca bidder-ul sa-si trimita oferta de 2
```

```
ori, eronat
        if (Random.nextBoolean()) {
            auctioneerSocket.getOutputStream().write(serializedMessage)
        }
    }

    private fun waitForResult() {
        println("$myIdentity Astept rezultatul licitatiei...")
        // bidder-ul se inscrie pentru primirea unui raspuns la oferta
trimisa de acesta
        val auctionResultSubscription =
auctionResultObservable.subscribeBy(
            // cand se primeste un mesaj in flux, inseamna ca a sosit
rezultatul licitatiei
            onNext = {
                val resultMessage: Message =
Message.deserialize(it.toByteArray())
                println("$myIdentity Rezultat licitatie:
${resultMessage.body}")
            },
            onError = {
                println("$myIdentity Eroare: $it")
            }
        )

        // se elibereaza memoria obiectului Subscription
        auctionResultSubscription.dispose()
    }

    fun run() {
        bid()
        waitForResult()
    }
}

fun main(args: Array<String>) {
    val bidderMicroservice = BidderMicroservice()
    bidderMicroservice.run()
}
```

- **AuctioneerMicroservice.kt**

```
import io.reactivex.rxjava3.core.Observable
import io.reactivex.rxjava3.disposables.CompositeDisposable
import io.reactivex.rxjava3.kotlin.subscribeBy
import io.reactivex.rxjava3.plugins.RxJavaPlugins
import java.io.BufferedReader
import java.io.InputStreamReader
import java.net.ServerSocket
import java.net.Socket
import java.net.SocketTimeoutException
import java.util.*
import kotlin.collections.ArrayList
import kotlin.system.exitProcess

class AuctioneerMicroservice {
```

```
    private var auctioneerSocket: ServerSocket
    private lateinit var messageProcessorSocket: Socket
    private var receiveBidsObservable: Observable<String>
    private val subscriptions = CompositeDisposable()
    private val bidQueue: Queue<Message> = LinkedList<Message>()
    private val bidderConnections: MutableList<Socket> =
mutableListOf()

    companion object Constants {
        const val MESSAGE_PROCESSOR_HOST = "localhost"
        const val MESSAGE_PROCESSOR_PORT = 1600
        const val AUCTIONEER_PORT = 1500
        const val AUCTION_DURATION: Long = 15_000 // licitatia dureaza
15 secunde
    }

    init {
        auctioneerSocket = ServerSocket(AUCTIONEER_PORT)
        auctioneerSocket.setSoTimeout(AUCTION_DURATION.toInt())
        println("AuctioneerMicroservice se executa pe portul:
${auctioneerSocket.localPort}")
        println("Se asteapta oferte de la bidderi...")

        // se creeaza obiectul Observable cu care se genereaza
evenimente cand se primesc oferte de la bidderi
        receiveBidsObservable = Observable.create<String> { emitter ->
            // se asteapta conexiuni din partea bidderilor
            while (true) {
                try {
                    val bidderConnection = auctioneerSocket.accept()
                    bidderConnections.add(bidderConnection)

                    // se citeste mesajul de la bidder de pe socketul
TCP
                    val bufferReader =
BufferedReader(InputStreamReader(bidderConnection.inputStream))
                    val receivedMessage = bufferReader.readLine()

                    // daca se primeste un mesaj gol (NULL), atunci
inseamna ca cealalta parte a socket-ului a fost inchisa
                    if (receivedMessage == null) {
                        // deci subscriber-ul respectiv a fost
deconectat
                        bufferReader.close()
                        bidderConnection.close()

                        emitter.onError(Exception("Eroare: Bidder-ul
${bidderConnection.port} a fost deconectat."))
                    }

                    // se emite ce s-a citit ca si element in fluxul
de mesaje
                    emitter.onNext(receivedMessage)
                } catch (e: SocketTimeoutException) {
                    // daca au trecut cele 15 secunde de la pornirea
licitatiei, inseamna ca licitatia s-a incheiat
```

```
                    // se emite semnalul Complete pentru a incheia
fluxul de oferte
                    emitter.onComplete()
                    break
                }
            }
        }
    }

    private fun receiveBids() {
        // se incepe prin a primi ofertele de la bidderi
        val receiveBidsSubscription =
receiveBidsObservable.subscribeBy(
            onNext = {
                val message = Message.deserialize(it.toByteArray())
                println(message)
                bidQueue.add(message)
            },
            onComplete = {
                // licitatia s-a incheiat
                // se trimit raspunsurile mai departe catre procesorul
de mesaje
                println("Licitatia s-a incheiat! Se trimit ofertele
spre procesare...")
                forwardBids()
            },
            onError = { println("Eroare: $it") }
        )
        subscriptions.add(receiveBidsSubscription)
    }

    private fun forwardBids() {
        try {
            messageProcessorSocket = Socket(MESSAGE_PROCESSOR_HOST,
MESSAGE_PROCESSOR_PORT)

subscriptions.add(Observable.fromIterable(bidQueue).subscribeBy(
                onNext = {
                    // trimitere mesaje catre procesorul de mesaje

messageProcessorSocket.getOutputStream().write(it.serialize())
                    println("Am trimis mesajul: $it")
                },
                onComplete = {
                    println("Am trimis toate ofertele catre
MessageProcessor.")
                    val bidEndMessage = Message.create(

"${messageProcessorSocket.localAddress}:${messageProcessorSocket.local
Port}",
                        "final"
                    )

messageProcessorSocket.getOutputStream().write(bidEndMessage.serialize
())
```

```
                    // dupa ce s-a terminat licitatia, se asteapta
raspuns de la MessageProcessorMicroservice
                    // cum ca a primit toate mesajele
                    val bufferReader =
BufferedReader(InputStreamReader(messageProcessorSocket.inputStream))
                    bufferReader.readLine()

                    messageProcessorSocket.close()

                    finishAuction()
                }
            ))
        } catch (e: Exception) {
            println("Nu ma pot conecta la MessageProcessor!")
            auctioneerSocket.close()
            exitProcess(1)
        }
    }

    private fun finishAuction() {
        // se asteapta rezultatul licitatiei
        try {
            val biddingProcessorConnection = auctioneerSocket.accept()
            val bufferReader =
BufferedReader(InputStreamReader(biddingProcessorConnection.inputStrea
m))

            // se citeste rezultatul licitatiei de la
AuctioneerMicroservice de pe socketul TCP
            val receivedMessage = bufferReader.readLine()

            val result: Message =
Message.deserialize(receivedMessage.toByteArray())
            val winningPrice = result.body.split(" ")[1].toInt()
            println("Am primit rezultatul licitatiei de la
BiddingProcessor: ${result.sender} a castigat cu pretul:
$winningPrice")

            // se creeaza mesajele pentru rezultatele licitatiei
            val winningMessage =
Message.create(auctioneerSocket.localSocketAddress.toString(),
                "Licitatie castigata! Pret castigator: $winningPrice")
            val losingMessage =
Message.create(auctioneerSocket.localSocketAddress.toString(),
                "Licitatie pierduta...")

            // se anunta castigatorul
            bidderConnections.forEach {
                if (it.remoteSocketAddress.toString() == result.sender)
{

it.getOutputStream().write(winningMessage.serialize())
                } else {

it.getOutputStream().write(losingMessage.serialize())
                }
```

```
                it.close()
            }
        } catch (e: Exception) {
            println("Nu ma pot conecta la BiddingProcessor!")
            auctioneerSocket.close()
            exitProcess(1)
        }

        // se elibereaza memoria din multimea de Subscriptions
        subscriptions.dispose()
    }

    fun run() {
        receiveBids()
    }
}

fun main(args: Array<String>) {
    val bidderMicroservice = AuctioneerMicroservice()
    bidderMicroservice.run()
}
```

- **MessageProcessorMicroservice.kt**

```
import io.reactivex.rxjava3.core.Observable
import io.reactivex.rxjava3.disposables.CompositeDisposable
import io.reactivex.rxjava3.kotlin.subscribeBy
import java.io.BufferedReader
import java.io.InputStreamReader
import java.net.ServerSocket
import java.net.Socket
import java.util.*
import kotlin.system.exitProcess

class MessageProcessorMicroservice {
    private var messageProcessorSocket: ServerSocket
    private lateinit var biddingProcessorSocket: Socket
    private var auctioneerConnection:Socket
    private var receiveInQueueObservable: Observable<String>
    private val subscriptions = CompositeDisposable()
    private val messageQueue: Queue<Message> = LinkedList<Message>()

    companion object Constants {
        const val MESSAGE_PROCESSOR_PORT = 1600
        const val BIDDING_PROCESSOR_HOST = "localhost"
        const val BIDDING_PROCESSOR_PORT = 1700
    }

    init {
        messageProcessorSocket = ServerSocket(MESSAGE_PROCESSOR_PORT)
        println("MessageProcessorMicroservice se executa pe portul:
${messageProcessorSocket.localPort}")
        println("Se asteapta mesaje pentru procesare...")

        // se asteapta mesaje primite de la AuctioneerMicroservice
```

```
        auctioneerConnection = messageProcessorSocket.accept()
        val bufferReader =
BufferedReader(InputStreamReader(auctioneerConnection.inputStream))

        // se creeaza obiectul Observable cu care se captureaza
mesajele de la AuctioneerMicroservice
        receiveInQueueObservable = Observable.create<String> { emitter
->
            while (true) {
                // se citeste mesajul de la AuctioneerMicroservice de
pe socketul TCP
                val receivedMessage = bufferReader.readLine()

                // daca se primeste un mesaj gol (NULL), atunci
inseamna ca cealalta parte a socket-ului a fost inchisa
                if (receivedMessage == null) {
                    // deci subscriber-ul respectiv a fost deconectat
                    bufferReader.close()
                    auctioneerConnection.close()

                    emitter.onError(Exception("Eroare:
AuctioneerMicroservice ${auctioneerConnection.port} a fost
deconectat."))
                    break
                }

                // daca mesajul este cel de incheiere a licitatiei
(avand corpul "final"), atunci se emite semnalul Complete
                if
(Message.deserialize(receivedMessage.toByteArray()).body == "final") {
                    emitter.onComplete()

                    break
                } else {
                    // se emite ce s-a citit ca si element in fluxul
de mesaje
                    emitter.onNext(receivedMessage)
                }
            }
        }
    }

    private fun receiveAndProcessMessages() {
        // se primesc si se adauga in coada mesajele de la
AuctioneerMicroservice
        val receiveInQueueSubscription = receiveInQueueObservable
            ///TODO --- filtrati duplicatele folosind operatorul de
filtrare
            .subscribeBy(
                onNext = {
                    val message = Message.deserialize(it.toByteArray())
                    println(message)
                    messageQueue.add(message)
                },
                onComplete = {
                    // s-a incheiat primirea tuturor mesajelor
```

```
                    ///TODO --- se ordoneaza in functie de data si ora
cand mesajele au fost primite

                    // s-au primit toate mesajele de la
AuctioneerMicroservice, i se trimite un mesaj pentru a semnala
                    // acest lucru
                    val finishedMessagesMessage = Message.create(

"${auctioneerConnection.localAddress}:${auctioneerConnection.localPort}
",
                        "am primit tot"
                    )

auctioneerConnection.getOutputStream().write(finishedMessagesMessage.s
erialize())
                    auctioneerConnection.close()

                    // se trimit mai departe mesajele procesate catre
BiddingProcessor
                    sendProcessedMessages()
                },
                onError = { println("Eroare: $it") }
            )
        subscriptions.add(receiveInQueueSubscription)
    }

    private fun sendProcessedMessages() {
        try {
            biddingProcessorSocket = Socket(BIDDING_PROCESSOR_HOST,
BIDDING_PROCESSOR_PORT)

            println("Trimit urmatoarele mesaje:")
            Observable.fromIterable(messageQueue).subscribeBy(
                onNext = {
                    println(it.toString())

                    // trimitere mesaje catre procesorul licitatiei,
care decide rezultatul final

biddingProcessorSocket.getOutputStream().write(it.serialize())
                },
                onComplete = {
                    val noMoreMessages = Message.create(

"${biddingProcessorSocket.localAddress}:${biddingProcessorSocket.local
Port}",
                        "final"
                    )

biddingProcessorSocket.getOutputStream().write(noMoreMessages.serializ
e())
                    biddingProcessorSocket.close()

                    // se elibereaza memoria din multimea de
Subscriptions
                    subscriptions.dispose()
```

```
                }
            )
        } catch (e: Exception) {
            println("Nu ma pot conecta la BiddingProcessor!")
            messageProcessorSocket.close()
            exitProcess(1)
        }
    }

    fun run() {
        receiveAndProcessMessages()
    }
}

fun main(args: Array<String>) {
    val messageProcessorMicroservice = MessageProcessorMicroservice()
    messageProcessorMicroservice.run()
}
```

- **BiddingProcessorMicroservice.kt**

```
import io.reactivex.rxjava3.core.Observable
import io.reactivex.rxjava3.disposables.CompositeDisposable
import io.reactivex.rxjava3.kotlin.subscribeBy
import java.io.BufferedReader
import java.io.InputStreamReader
import java.net.ServerSocket
import java.net.Socket
import java.util.*
import kotlin.system.exitProcess

class BiddingProcessorMicroservice {
    private var biddingProcessorSocket: ServerSocket
    private lateinit var auctioneerSocket: Socket
    private var receiveProcessedBidsObservable: Observable<String>
    private val subscriptions = CompositeDisposable()
    private val processedBidsQueue: Queue<Message> =
LinkedList<Message>()

    companion object Constants {
        const val BIDDING_PROCESSOR_PORT = 1700
        const val AUCTIONEER_PORT = 1500
        const val AUCTIONEER_HOST = "localhost"
    }

    init {
        biddingProcessorSocket = ServerSocket(BIDDING_PROCESSOR_PORT)
        println("BiddingProcessorMicroservice se executa pe portul:
${biddingProcessorSocket.localPort}")
        println("Se asteapta ofertele pentru finalizarea
licitatiei...")

        // se asteapta mesaje primite de la
MessageProcessorMicroservice
        val messageProcessorConnection =
```

```
biddingProcessorSocket.accept()
        val bufferReader =
BufferedReader(InputStreamReader(messageProcessorConnection.inputStrea
m))

        // se creeaza obiectul Observable cu care se captureaza
mesajele de la MessageProcessorMicroservice
        receiveProcessedBidsObservable = Observable.create<String>
{ emitter ->
            while (true) {
                // se citeste mesajul de la
MessageProcessorMicroservice de pe socketul TCP
                val receivedMessage = bufferReader.readLine()

                // daca se primeste un mesaj gol (NULL), atunci
inseamna ca cealalta parte a socket-ului a fost inchisa
                if (receivedMessage == null) {
                    // deci MessageProcessorMicroservice a fost
deconectat
                    bufferReader.close()
                    messageProcessorConnection.close()

                    emitter.onError(Exception("Eroare:
MessageProcessorMicroservice ${messageProcessorConnection.port} a fost
deconectat."))
                    break
                }

                // daca mesajul este cel de tip „FINAL DE LISTA DE
MESAJE” (avand corpul "final"), atunci se emite semnalul Complete
                if
(Message.deserialize(receivedMessage.toByteArray()).body == "final") {
                    emitter.onComplete()

                    // s-au primit toate mesajele de la
MessageProcessorMicroservice, i se trimite un mesaj pentru a semnala
                    // acest lucru
                    val finishedBidsMessage = Message.create(

"${messageProcessorConnection.localAddress}:${messageProcessorConnecti
on.localPort}",
                        "am primit tot"
                    )


messageProcessorConnection.getOutputStream().write(finishedBidsMessage.
serialize())
                    messageProcessorConnection.close()

                    break
                } else {
                    // se emite ce s-a citit ca si element in fluxul
de mesaje
                    emitter.onNext(receivedMessage)
                }
            }
        }
```

```
    }

    private fun receiveProcessedBids() {
        // se primesc si se adauga in coada ofertele procesate de la
MessageProcessorMicroservice
        val receiveProcessedBidsSubscription =
receiveProcessedBidsObservable
            .subscribeBy(
                onNext = {
                    val message = Message.deserialize(it.toByteArray())
                    println(message)
                    processedBidsQueue.add(message)
                },
                onComplete = {
                    // s-a incheiat primirea tuturor mesajelor
                    // se decide castigatorul licitatiei
                    decideAuctionWinner()
                },
                onError = { println("Eroare: $it") }
            )
        subscriptions.add(receiveProcessedBidsSubscription)
    }

    private fun decideAuctionWinner() {
        // se calculeaza castigatorul ca fiind cel care a ofertat cel
mai mult
       val winner: Message? = processedBidsQueue.toList().maxByOrNull
{
            // corpul mesajului e de forma "licitez <SUMA_LICITATA>"
            // se preia a doua parte, separata de spatiu
            it.body.split(" ")[1].toInt()
        }

        println("Castigatorul este: ${winner?.sender}")

        try {
            auctioneerSocket = Socket(AUCTIONEER_HOST, AUCTIONEER_PORT)

            // se trimite castigatorul catre AuctioneerMicroservice

auctioneerSocket.getOutputStream().write(winner!!.serialize())
            auctioneerSocket.close()

            println("Am anuntat castigatorul catre
AuctioneerMicroservice.")
        } catch (e: Exception) {
            println("Nu ma pot conecta la Auctioneer!")
            biddingProcessorSocket.close()
            exitProcess(1)
        }
    }

    fun run() {
        receiveProcessedBids()

        // se elibereaza memoria din multimea de Subscriptions
```

```
        subscriptions.dispose()
    }
}

fun main(args: Array<String>) {
    val biddingProcessorMicroservice = BiddingProcessorMicroservice()
    biddingProcessorMicroservice.run()
}
```

Compilarea şi execuţia proiectului

Proiectul se compilează astfel: se foloseşte meniul **Build → Build Project,** pentru a compila toate modulele componente deodată.

Proiectul se execută astfel: se porneşte fiecare microserviciu în parte (eventual de la linia de comandă), cu următoarele menţiuni:

- mai întâi se pornesc **`MessageProcessorMicroservice`** şi **`BiddingProcessorMicroservice`**, în orice ordine se doreşte
- apoi se porneşte **`AuctioneerMicroservice`**, **dar acesta va încheia licitaţia, în mod implicit, după 15 secunde**. Aşadar, **aveţi 15 secunde la dispoziţie să porniţi câteva microservicii `BidderMicroservice`**.
- în final, se pornesc câteva microservicii de tip **`BidderMicroservice`**, care să reprezinte ofertanţii ce participă la licitaţie

Exemplu de execuţie de la linia de comandă:

```
java -jar MessageProcessorMicroservice.jar
```

Exemplu de execuţie

Se deschide câte o sesiune de terminal pentru rularea fiecărui microserviciu. Se poate utilizaemulatorul de terminal "terminator" care facilitează vizualizarea mai multor sesiuni de terminal. Pentru instalare:

```
sudo apt-get install terminator
```

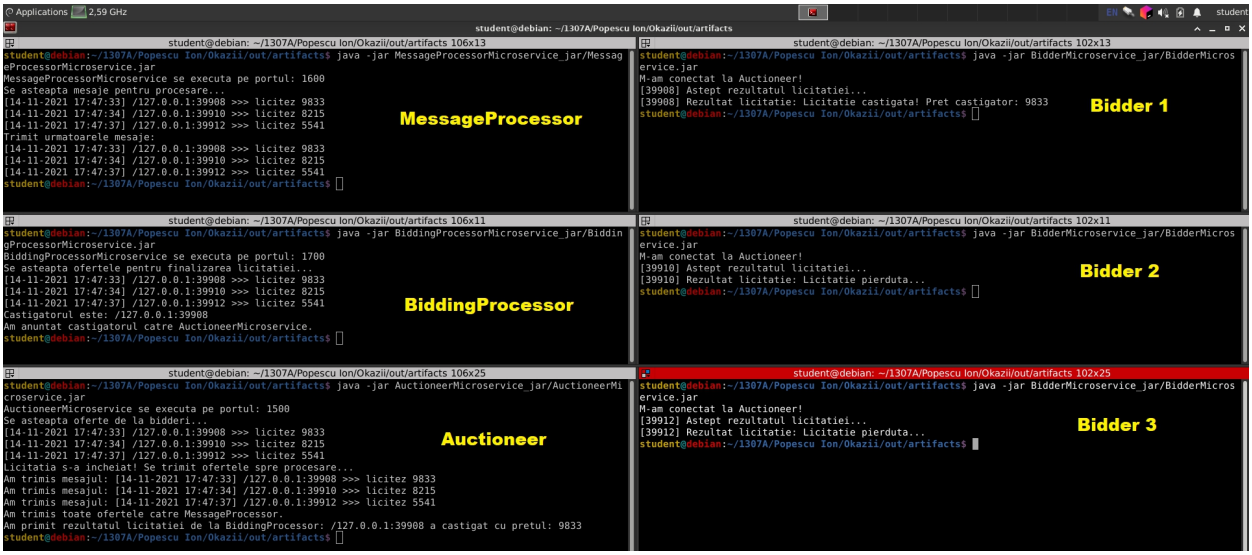


Figura 6 - Exemplu de execuţie

Teme de laborator

1. Implementaţi şi executaţi aplicaţia exemplu din laborator, pornind licitaţia cu un număr mare de microservicii **`Bidder`** (de exemplu, 100). Puteţi crea, spre exemplu, un script Bash care porneşte 100 de procese **`Bidder`**.
2. Modificaţi microserviciul **`MessageProcessor`** astfel încât să filtraţi mesajele duplicate din fluxul reactiv primit de la **`Auctioneer`**.
3. Modificaţi microserviciul **`MessageProcessor`** astfel încât, atunci când s-a primit tot fluxul de mesaje de la **`Auctioneer`**, să sortaţi mesajele primite înainte de a le redirecţiona spre **`BiddingProcessor`**.

Temă pentru acasă

1. Modificaţi aplicaţia astfel încât ofertanţii să fie identificaţi de un nume, un număr de telefon şi un e-mail. Mesajele care circulă între microservicii trebuie să includă şi datele noi introduse. Fiecare microserviciu trebuie sa aiba un jurnal al executiei (face o operatie o scrie aolo, proceseaza niste date o scrie acolo). Cand microserviciul crapă va verifica acest jurnal si dacă există un ciclu curent de procesare intrerupt va incarca datele il va termina si apoi va continua cu procesarea noilor evenimente.
2. Se va adăuga un microserviciu care pe bază de mecanism de heart beat (cum se creează o instanta se înscrie la el şi el verifica în docker starea acestei) în caz de fail va reporni respectiva instanta care are deja mecanism de stare (vezi punctul 1)
3. Să se includă capabilităţi de instrumentare a microserviciilor componente, pentru colectarea datelor legate de încărcare, execuţie (vezi punctul 1) şi funcţionare. Datele colectate vor fi scrise într-un fişier log generalizat cu sincronizare cu jurnalele locale ale microserviciilor (scriu local și trimit și masterului).

Se vor realiza diagrama de uservicii și cea UML și se vor verifica respectarea principiilor SOLID pentru microservicii. IN diagrama de uservicii se va arăta și nivelul/stratul din care face fiecare parte (conform abordării multilayer-multitier)

Bibliografie

[1]: The Reactive Manifesto - https://github.com/reactivemanifesto/reactivemanifesto
[2]: Reactive Programming with Kotlin, Alex Sullivan, First Edition - https://store.raywenderlich.com/products/reactive-programming-with-kotlin
[3]: Reactive Programming in Kotlin, Rivu Chakraborty, Packt Publishing
[4]: RxKotlin - RxJava bindings for Kotlin - https://github.com/ReactiveX/RxKotlin