Sisteme Distribuite - Laborator 11

Micronaut - *serverless computing*

Micronaut - descriere generală

Micronaut este un *framework* modern folosit pentru crearea de microservicii, proiectat pentru construirea de aplicații modulare şi testabile cu uşurință. Ținta acestui *framework* este furnizarea tuturor uneltelor necesare pentru aplicațiile bazate pe microservicii, precum:

- injectarea dependențelor la compilare (eng. *compile-time dependency injection*) şi inversarea controlului (eng. *Inversion of Control*)
- configurare automată
- partajarea configurărilor
- descoperirea de servicii (eng. *service discovery*)
- rutare HTTP
- client HTTP cu echilibrarea încărcării la client

Față de alte *framework*-uri precum Spring / Spring Boot sau Grails, Micronaut încearcă să rezolve unele probleme apărute în tehnologiile menționate, oferind următoarele avantaje:

- pornirea rapidă a aplicațiilor
- grad mai mic de utilizare a memoriei
- grad mic de utilizare a reflecției (eng. *reflection*)
- grad mic de utilizare a *proxy*-urilor
- dimensiune mică a artefactelor JAR rezultate
- testare unitară facilă

Avantajele menționate ale Micronaut s-au obținut prin utilizarea **procesoarelor de adnotări Java** (eng. *annotation processors*), care se pot folosi în orice limbaj de programare ce ține**ş**te maşina virtuală Java, şi care suportă acest concept. Aceste procesoare de adnotare precompilează metadatele necesare pentru a asigura injectarea dependențelor, definirea *proxy*-urilor AOP şi configurarea aplicației în scopul execuției acesteia într-un mediu al microserviciilor.

Instalarea Micronaut

Se descarcă arhiva de pe site-ul oficial (https://micronaut.io/download.html), astfel:

```
wget https://github.com/micronaut-projects/micronaut-
core/releases/download/v1.3.4/micronaut-1.3.4.zip
```

(ultima versiune stabilă disponibilă la momentul scrierii laboratorului este **`1.3.4`**)

Dezarhivați arhiva descărcată utilizând arhivatorul grafic sau comanda următoare:

```
unzip micronaut-1.3.4.zip
```

Mutați server-ul Micronaut în folder-ul standard cu software opțional:

```
sudo mv micronaut-1.3.4 /opt
```

Adăugați binarul **`mn`** la **`PATH`**-ul sistemului:

```
export PATH="$PATH:/opt/micronaut-1.3.4/bin"
```

Atenție: comanda de mai sus este valabilă pentru sesiunea de terminal curentă!

Dacă deschideți un terminal nou și vreți să lucrați cu Micronaut, trebuie să rulați comanda anterioară din nou. Alternativ, adăugați comanda de mai sus la sfârșitul fișierului `$HOME/.bashrc`.

Testați server-ul Micronaut utilizând comanda următoare:

```
mn --version
```

Exemple de tipuri de aplicații folosind Micronaut CLI

Deși modul de creare a aplicațiilor Micronaut din CLI nu este neapărat necesar (proiectul poate fi creat manual și scheletul scris apoi), se poate folosi linia de comandă a server-ului în acest scop, pentru a simplifica inițializarea unui proiect de tip Micronaut.

Crearea unui proiect Micronaut - limbaj Kotlin, gestionar de proiect Maven

```
mn create-app com.sd.laborator.exemplu-micronaut-maven --lang kotlin --build maven
```

Crearea unui proiect Micronaut - limbaj Kotlin, gestionar de proiect Gradle

```
mn create-app com.sd.laborator.exemplu-micronaut-gradle --lang kotlin --build gradle
```

După ce ați creat un proiect Micronaut folosind **unul din cele 2 tipuri de gestionare de proiect**, puteți importa proiectul în IntelliJ: **Open** (sau „**Open or Import**”) → selectați folder-ul generat de comanda de creare a proiectului și apăsați **OK**.

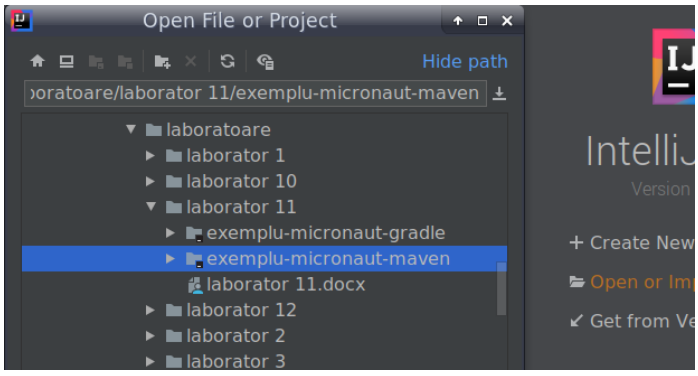

Figura 1 - Importare proiect Micronaut

Așteptați până când IntelliJ rezolvă toate dependențele necesare și până generează toată structura proiectului.

Activați procesarea adnotărilor din IntelliJ, astfel: **File → Settings → Build, Execution, Deployment → Compiler → Annotation Processors →** bifați **Enable annotation processing**.

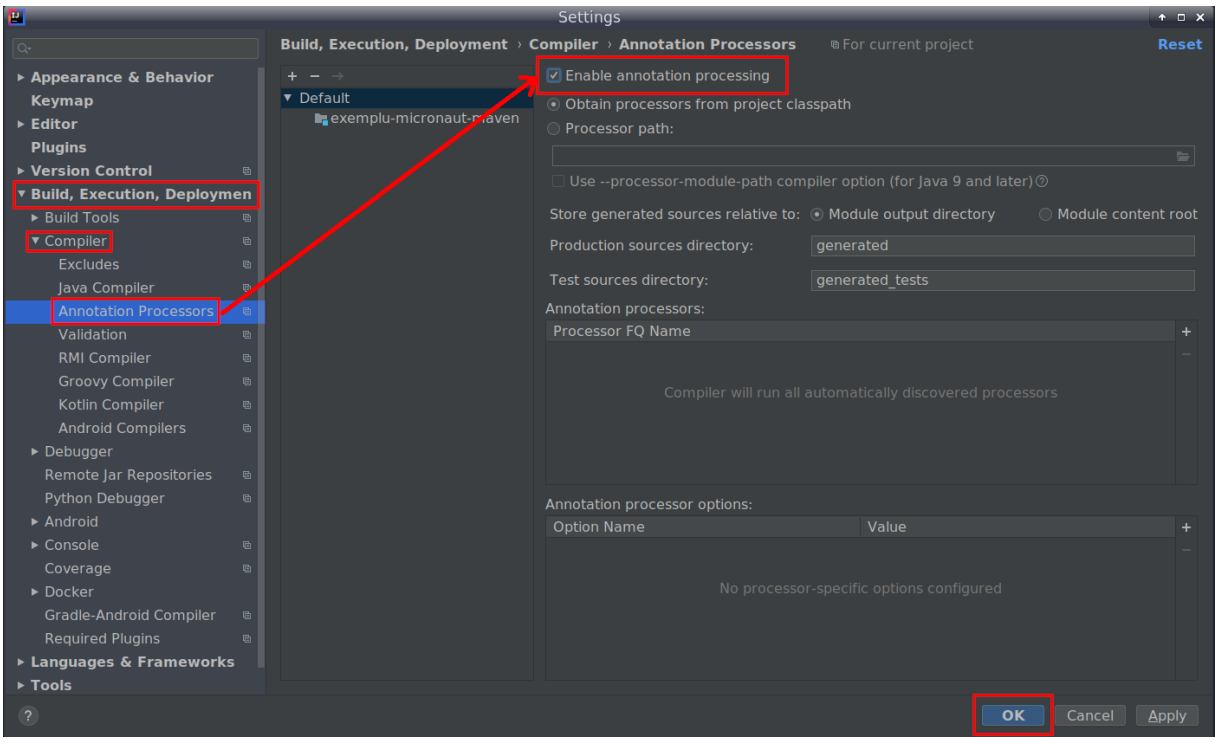

Figura 2 - Activarea procesării adnotărilor în IntelliJ

Codul din fișierul sursă **`Application.kt`** generat automat arată astfel:

```
package com.sd.laborator

import io.micronaut.runtime.Micronaut

object Application {

    @JvmStatic
    fun main(args: Array<String>) {
        Micronaut.build()
                .packages("com.sd.laborator")
                .mainClass(Application.javaClass)
                .start()
    }
}
```

Adnotarea **`@JvmStatic`** aplicată asupra funcției **`main()`** determină generarea unei metode statice adiționale pe baza funcției existente aflate sub influența adnotării. Apelul **`start()`** asupra *builder*-ului de *runtime* Micronaut reprezintă punctul de intrare al aplicației.

Crearea unui controller folosind Micronaut CLI

O componentă *controller* poate fi creată în mod facil tot folosind linia de comandă Micronaut. Executați comanda următoare **în folder-ul proiectului**:

```
mn create-controller HelloController
```

Comanda va modifica proiectul Micronaut existent adăugând o componentă de tip *controller*. Modificați codul *controller*-ului (clasa generată **`HelloController`**) astfel:

```
package com.sd.laborator

import io.micronaut.http.annotation.Controller
```

```
import io.micronaut.http.annotation.Get
import io.micronaut.http.MediaType
import io.micronaut.http.annotation.Produces

@Controller("/hello")
class HelloController {
    @Produces(MediaType.TEXT_PLAIN)
    @Get("/")
    fun hello(): String {
        return "Hello from Micronaut!"
    }
}
```

Controller-ul aplicației Micronaut este asemănător cu un *controller* Spring: clasa *controller* se adnotează cu **`@Controller`**, iar parametrul adnotării reprezintă calea de bază pentru toate metodele mapate. În acest exemplu simplu, metoda **`hello()`** este executată la o cerere HTTP de tip GET (adnotarea **`@Get`**) către calea „**`/`**” relativ la „**`/hello`**”.

Adnotarea **`@Produces(MediaType.TEXT_PLAIN)`** indică tipul de răspuns returnat clientului. Deoarece JSON este tipul implicit de răspuns iar şirul de caractere returnat este nestructurat, se marchează explicit tipul „*text plain*”.

Execuția aplicației Micronaut din IntelliJ

Execuția cu Maven

Folosiți *goal*-ul **`exec`** din *plugin*-ul **`exec`** pentru a porni aplicația Micronaut cu schelet Maven:

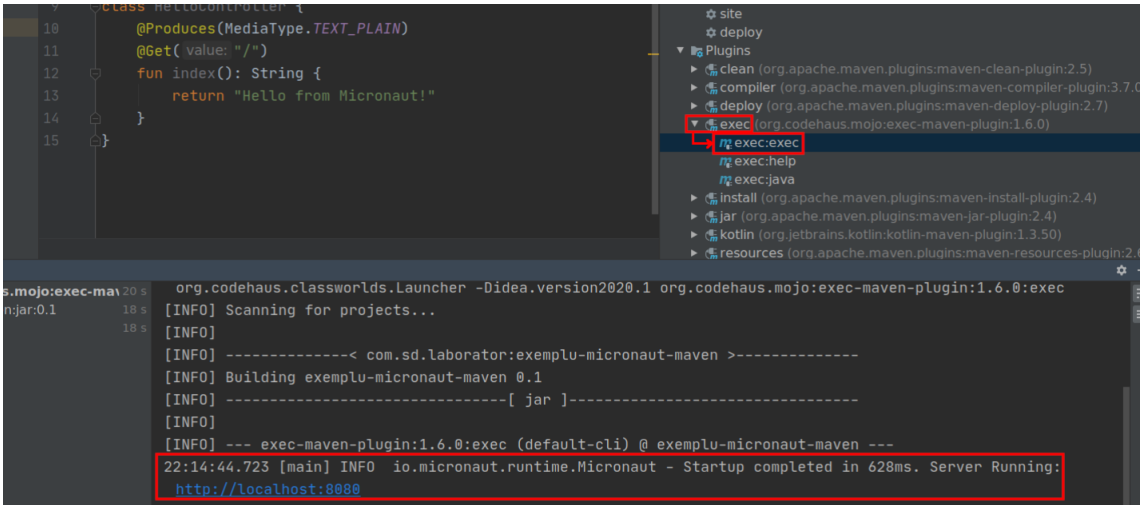

Figura 3 - Execuția unui proiect Micronaut cu schelet Maven

Execuția cu Gradle

Folosiți *task*-ul **`run`** din secțiunea **`application`** pentru a porni aplicația Micronaut cu schelet Gradle:

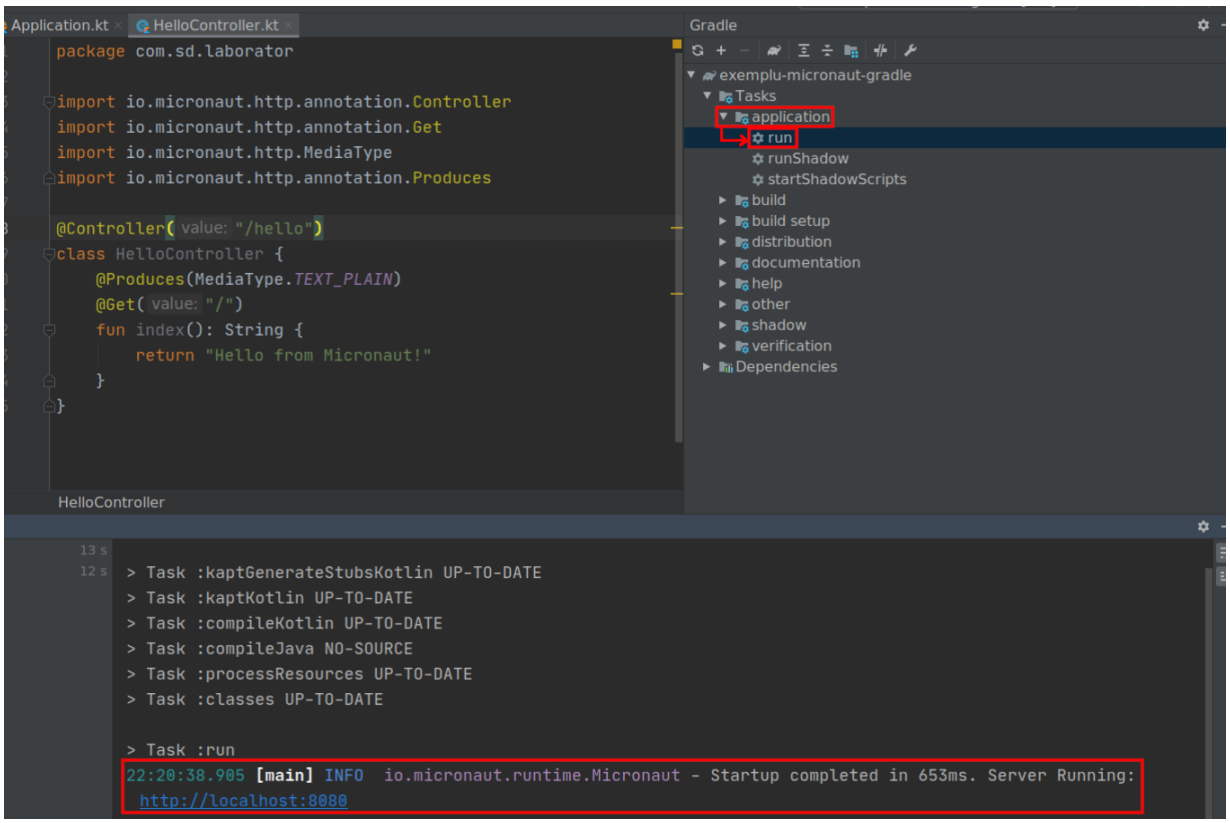

Figura 4 - Execuția unui proiect Micronaut cu schelet Gradle

Testarea aplicației Micronaut

Controller-ul răspunde pe calea **`/hello/`** al server-ului HTTP incorporat, bazat pe **Netty**. Aşadar, trimiteți o cerere de tip GET astfel:

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

Funcții *serverless*

Funcțiile *serverless* sunt gestionate de o infrastructură *cloud* şi sunt executate în procese efemere - codul este rulat de obicei în containere fără stare şi poate fi declanşat de evenimente precum: cereri HTTP, alerte, evenimente recurente, încărcări de fişiere etc.

Aceste funcții sunt invocate prin Internet şi sunt găzduite şi menținute de companii de *cloud computing*. Furnizorul de *cloud* este responsabil pentru execuția codului încapsulat în funcții, alocând în mod dinamic resursele necesare pentru acestea. Acest model se mai numeşte şi „*Function as a Service*”.

Crearea unei funcții serverless cu Micronaut CLI

Funcția *serverless* Micronaut este de fapt un nou tip de aplicație şi **se creează ca şi proiect separat**. Aşadar, în afara oricărui proiect Micronaut, executați următoarea comandă, în funcție de gestionarul de proiect dorit:

- creare funcție *serverless* cu schelet **Maven**:

```
mn create-function com.sd.laborator.hello-world-maven --lang kotlin --build maven
```

- creare funcție *serverless* cu schelet **Gradle**:

```
mn create-function com.sd.laborator.hello-world-gradle --lang kotlin -
```

```
-build gradle
```

Proiectul generat se deschide în IntelliJ în aceeaşi manieră explicată anterior.

Dacă ați ales gestionarul de proiect Maven: adăugați următoarea dependență suplimentară în **pom.xml**:

```
<dependency>
      <groupId>org.apache.logging.log4j</groupId>
      <artifactId>log4j-api</artifactId>
      <version>2.13.2</version>
      <scope>runtime</scope>
</dependency>
```

Scheletul de cod generat conține următoarele clase:

- **Application** - identic ca cel de la exemplul anterior

- **HelloWorldMaven** / **HelloWorldGradle**

```
package com.sd.laborator

import io.micronaut.core.annotation.*

@Introspected
class HelloWorldGradle { // sau HelloWorldMaven
   lateinit var name: String
}
```

Adnotarea **@Introspected** indică faptul că acea clasă peste care este aplicată va produce un tip de dată **BeanIntrospection** în momentul compilării. Acesta este rezultatul unei procesări făcute la compilare, procesare ce include proprietăți şi metadate: prin tipul de dată **BeanIntrospection**, se pot instanția *bean*-uri şi scrie / citi proprietăți din acestea fără a folosi reflecția.

- **HelloWorldGradleFunction** / **HelloWorldMavenFunction**

```
package com.sd.laborator;

import io.micronaut.function.executor.FunctionInitializer
import io.micronaut.function.FunctionBean;
import javax.inject.*;
import java.util.function.Function;

@FunctionBean("hello-world-gradle")
class HelloWorldGradleFunction : FunctionInitializer(),
Function<HelloWorldGradle, HelloWorldGradle> { // sau HelloWorldMaven

    override fun apply(msg : HelloWorldGradle) : HelloWorldGradle {
         return msg
    }
}

/**
 * This main method allows running the function as a CLI application
using: echo '{}' | java -jar function.jar
```

```
 * where the argument to echo is the JSON to be parsed.
 */
fun main(args : Array<String>) {
    val function = HelloWorldGradleFunction()
    function.run(args, { context ->
function.apply(context.get(HelloWorldGradle::class.java))})
}
```

Se observă adnotarea **`@FunctionBean`** aplicată clasei **`HelloWorldGradleFunction`**. Efectul este expunerea clasei respective sub formă de funcție în aplicația Micronaut. Clasa respectivă trebuie să implementeze una din interfețele de tip **`Function`**. În acest caz, se implementează interfața **`Function`** (a se vedea tabelul de mai jos). Primul parametru *template* reprezintă tipul de date primit la intrare, iar al doilea reprezintă tipul de date returnat la ieşire: **`HelloWorldGradle`**.

Există următoarele tipuri de funcții Micronaut:

Interfață	Descriere
`Supplier`	Nu acceptă niciun argument şi returnează un singur rezultat
`Consumer`	Acceptă un singur argument şi nu returnează niciun rezultat
`Biconsumer`	Acceptă 2 argumente şi nu returnează niciun rezultat
`Function`	Acceptă un singur argument şi returnează un singur rezultat
`BiFunction`	Acceptă 2 argumente şi returnează un singur rezultat

Constructorul **`FunctionInitializer`** este utilizat pentru inițializarea unei funcții Micronaut.

Comportamentul principal al funcției *serverless* este încapsulat în metoda **`apply`** (specificată în interfața **`Function`**). Această metodă **aplică** funcția peste argumentul primit la intrare şi returnează datele de ieşire. Pentru acest exemplu simplu, funcția preia parametrul de intrare şi îl returnează aşa cum este.

Împachetarea funcției serverless

Funcția *serverless* Micronaut poate fi împachetată într-un artefact JAR în mod asemănător cu proiectul de tip Spring Boot:

- împachetare cu **Maven** → folosiți *lifecycle*-ul **`package`** (la fel ca la **Spring Boot**); artefactul rezultat se află în folder-ul **`target`** şi are denumirea proiectului şi sufixul versiunii (implicit „**`0.1`**”).
- împachetare cu **Gradle** → folosiți *task*-ul **`assemble`** din secțiunea **`build`**; artefactul rezultat se află în folder-ul **`build/libs`** şi are denumirea proiectului, apoi versiunea, **apoi sufixul “`-all`”**.

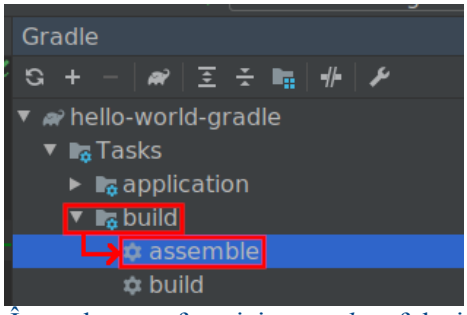

Figura 5 - Împachetarea funcției *serverless* folosind Gradle

Testarea funcției serverless

În mod implicit, funcția *serverless* generată de Micronaut va citi de la dispozitivul standard de intrare (***stdin***) și va scrie la dispozitivul standard de ieșire (***stdout***).

După ce ați împachetat proiectul sub formă de artefact JAR, testați funcția *serverless* trimițând valorile câmpurilor din clasa ce reprezintă parametrul de intrare sub formă de obiect JSON. Clasa **`HelloWorldGradle`** (sau **`HelloWorldMaven`**) conține un singur membru de tip string, numit „**`name`**”. Deci, dacă se dorește trimiterea valorii „**`covid`**” pentru membrul „**`name`**”, se codifică sub formă de obiect JSON astfel: **`{"name": "covid"}`**. Comanda de execuție este, așadar:

```
echo '{"name": "covid"}' | java -jar hello-world-gradle-0.1-all.jar
```

Pipe-ul (simbolul „|”) care conectează cele 2 comenzi Linux face legătura între ieșirea standard (***stdout***) a comenzii **`echo ...`** și intrarea standard (***stdin***) a comenzii **`java ...`**. Deci, aplicația Micronaut care încapsulează funcția *serverless* va primi la intrare textul afișat de comanda echo și acesta va fi decodificat din JSON astfel încât să populeze câmpurile clasei **`HelloWorldGradle`** (sau **`HelloWorldMaven`**).

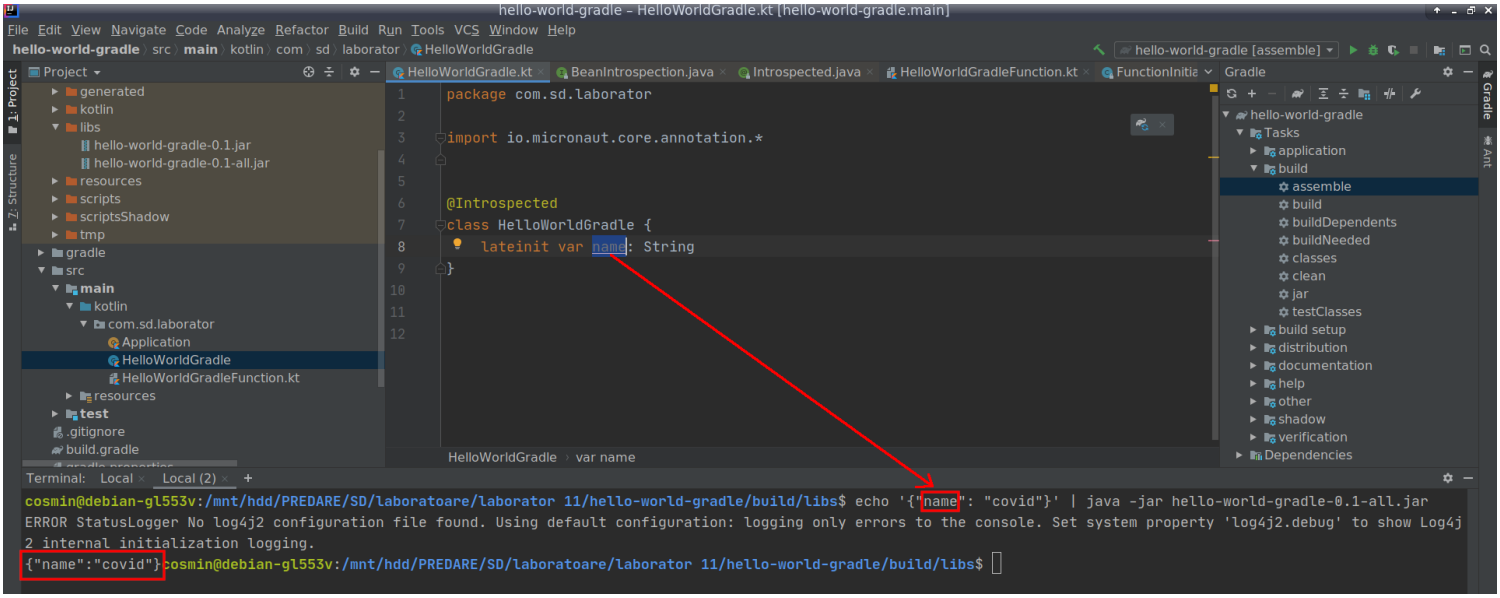

Figura 6 - Execuția funcției *serverless* împachetată cu Gradle

Dacă ați folosit Maven ca gestionar de proiect, s-ar putea să primiți câteva erori la execuția funcției *serverless* (a se vedea figura de mai jos). Sunt legate de *plugin*-ul **`log4j2`**, iar cauza este pierderea câtorva definiții din fișierul **`Log4j2Plugins.dat`** utilizat de Micronaut, atunci când se îmbină mai multe artefacte JAR într-unul singur. Se pot ingora fără a afecta funcționalitatea aplicației din laborator.

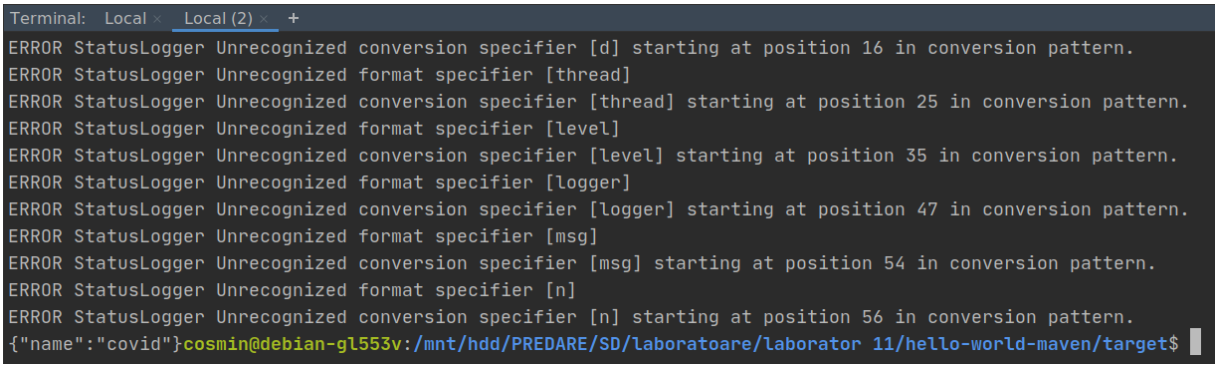

Figura 7 - Execuția funcției *serverless* împachetată cu Maven

Ciurul lui Eratostene - sub formă de funcție *serverless* Micronaut

Ciurul lui Eratostene este un algoritm simplu de descoperire a tuturor numerelor prime până la un întreg specificat ca parametru. Algoritmul are complexitatea **`O(n·log(n))`**, în varianta clasică. În laborator se va folosi o variantă optimizată, având complexitatea **`O(n)`**.

Creați o funcție *serverless* Micronaut folosind linia de comandă, cu denumirea „**`com.sd.laborator.eratostene`**”.

Modificați codul schelet rezultat astfel:

- Redenumiți clasa **`Eratostene`** sub forma **`EratosteneRequest`** și modificați codul astfel:

```
package com.sd.laborator

import io.micronaut.core.annotation.Introspected

@Introspected
class EratosteneRequest {
    private lateinit var number: Integer

    fun getNumber(): Int {
        return number.toInt()
    }
}
```

Această clasă va încapsula cererea primită ca și parametru de intrare. Cererea conține numărul maxim până la care se va calcula lista de numere prime din ciurul lui Eratostene (membrul **`number`**).

- Adăugați o nouă clasă **`@Introspected`** care va reprezenta răspunsul dat de funcția *serverless*. Denumiți clasa **`EratosteneResponse`**.

```
package com.sd.laborator

import io.micronaut.core.annotation.Introspected

@Introspected
class EratosteneResponse {
   private var message: String? = null
   private var primes: List<Int>? = null

   fun getPrimes(): List<Int>? {
      return primes
   }

   fun setPrimes(primes: List<Int>?) {
      this.primes = primes
   }

   fun getMessage(): String? {
      return message
   }

   fun setMessage(message: String?) {
      this.message = message
   }
}
```

Răspunsul funcției *serverless* conține un mesaj cu care se notifică starea de succes a execuției algoritmului, respectiv o listă de numere întregi prime rezultate în urma algoritmului. Lista este populată doar dacă nu există vreo eroare raportată prin variabila **`message`**.

• Adăugați o componentă **`Singleton`** numită **`EratosteneSieveService`**:

```
package com.sd.laborator

import java.util.*
import javax.inject.Singleton

@Singleton
class EratosteneSieveService {
    // implementare preluata de la
https://www.geeksforgeeks.org/sieve-eratosthenes-0n-time-complexity/
    val MAX_SIZE = 1000001

     /*
     isPrime[] : isPrime[i] este adevarat daca numarul i este prim
     prime[] : stocheaza toate numerele prime mai mici ca N
     SPF[] (Smallest Prime Factor) - stocheaza cel mai mic factor prim
al numarului
     [de exemplu : cel mai mic factor prim al numerelor '8' si '16'
     este '2', si deci SPF[8] = 2 , SPF[16] = 2 ]
     */
    private val isPrime = Vector<Boolean>(MAX_SIZE)
    private val SPF = Vector<Int>(MAX_SIZE)
    fun findPrimesLessThan(n: Int): List<Int> {
        val prime: MutableList<Int> = ArrayList()
        for (i in 2 until n) {
            if (isPrime[i]) {
                prime.add(i)

                // un numar prim este propriul sau cel mai mic factor
prim
                SPF[i] = i
            }

            /*
            Se sterg toti multiplii lui i * prime[j], care nu sunt
primi
            setand isPrime[i * prime[j]] = false
            si punand cel mai mic factor prim al lui i * prime[j] ca
si prime[j]
            [de exemplu: fie i = 5, j = 0, prime[j] = 2 [i * prime[j]
= 10],
            si deci cel mai mic factor prim al lui '10' este '2' care
este prime[j] ]

            Aceasta bucla se executa doar o singura data pentru
numerele care nu sunt prime
             */
            var j = 0
            while (j < prime.size && i * prime[j] < n && prime[j] <=
SPF[i]) {
                isPrime[i * prime[j]] = false

                // se pune cel mai mic factor prim al lui i * prime[j]
```

```
                SPF[i * prime[j]] = prime[j]
                j++
            }
        }
        return prime
    }

    init {
        // initializare vectori isPrime si SPF
        for (i in 0 until MAX_SIZE) {
            isPrime.add(true)
            SPF.add(2)
        }
        // 0 and 1 are not prime
        isPrime[0] = false
        isPrime[1] = false
    }
}
```

Clasa **`EratosteneSieveService`** încapsulează algoritmul de calcul al ciurului lui Eratostene până la un număr întreg **`N`**. Fiind serviciu unic, este adnotat cu **`@Singleton`** pentru a fi instanțiat o singură dată de componenta injector a Micronaut.

• Modificați funcția **`EratosteneFunction`** rezultată astfel:

```
package com.sd.laborator;

import io.micronaut.function.FunctionBean
import io.micronaut.function.executor.FunctionInitializer
import org.slf4j.Logger
import org.slf4j.LoggerFactory
import java.util.function.Function
import javax.inject.Inject

@FunctionBean("eratostene")
class EratosteneFunction : FunctionInitializer(),
Function<EratosteneRequest, EratosteneResponse> {
    @Inject
    private lateinit var eratosteneSieveService:
EratosteneSieveService

    private val LOG: Logger =
LoggerFactory.getLogger(EratosteneFunction::class.java)

    override fun apply(msg : EratosteneRequest) : EratosteneResponse {
        // preluare numar din parametrul de intrare al functiei
        val number = msg.getNumber()

        val response = EratosteneResponse()

        // se verifica daca numarul nu depaseste maximul
        if (number >= eratosteneSieveService.MAX_SIZE) {
            LOG.error("Parametru prea mare! $number > maximul de
${eratosteneSieveService.MAX_SIZE}")
            response.setMessage("Se accepta doar parametri mai mici ca
" + eratosteneSieveService.MAX_SIZE)
            return response
        }
```

```
        LOG.info("Se calculeaza primele $number numere prime ...")

        // se face calculul si se seteaza proprietatile pe obiectul cu
rezultatul

response.setPrimes(eratosteneSieveService.findPrimesLessThan(number))
        response.setMessage("Calcul efectuat cu succes!")

        LOG.info("Calcul incheiat!")

        return response
    }
}

/**
 * This main method allows running the function as a CLI application
using: echo '{}' | java -jar function.jar
 * where the argument to echo is the JSON to be parsed.
 */
fun main(args : Array<String>) {
    val function = EratosteneFunction()
    function.run(args, { context ->
function.apply(context.get(EratosteneRequest::class.java))})
}
```

S-a creat o clasă **EratosteneFunction** expusă sub formă de funcție Micronaut. Interfața folosită este **Function**, deci se va citi un argument de la intrarea standard şi se va scrie un singur răspuns la ieşirea standard.

Se foloseşte adnotarea **@Inject** pentru a injecta automat dependența **EratosteneSieveService**, necesară pentru returnarea listei de numere prime. De asemenea, este utilizată o instanță de tip **Logger**, folosită pentru mesaje informative sau de eroare, în funcție de nivelul de *logging* dorit.

Atenție: NU folosiți metode de tipul `println()` pentru afişarea mesajelor la consolă într-o aplicație Micronaut cu funcții *serverless*. După cum s-a menționat anterior, funcția serverless Micronaut trimite rezultatul, în mod implicit, către dispozitivul de ieşire standard (***stdout***). Însă, acelaşi lucru îl face şi funcția **println()** din Kotlin, şi alte funcții asemănătoare. **Aşadar, pentru a afişa mesaje informative sau erori, folosiți exclusiv Logger-ul pus la dispoziție de Micronaut, care nu intervine peste canalul standard de ieşire!**

De asemenea, **Logger**-ul este implicit configurat să afişeze **doar mesaje de eroare** la consolă.

Împachetați aplicația şi testați-o prin trimiterea datelor necesare în format JSON, către intrarea standard (***stdin***) a funcției *serverless*:

```
echo '{"number": 50}' | java -jar eratostene-0.1-all.jar
```

Aplicații şi teme

Temă de laborator

• Modificați aplicația din laborator care implementează ciurul lui Eratostene astfel încât să calculeze, recursiv, termenii din şirul cu numere întregi definit astfel:

$$a_n = a_{n-1} + 2 \cdot \frac{a_{n-1}}{n},\ \forall\ n \geq 1$$
$$a_0 = 1$$

Teme pentru acasă

1. Transformați funcția *serverless* din ciurul lui Eratostene în aşa fel încât aplicația să primească, pe lângă numărul maxim până la care algoritmul face calculul necesar, o listă de numere întregi. Aplicația trebuie să utilizeze ciurul lui Eratostene deja implementat ca să decidă care din numerele respective sunt prime sau nu. Se vor returna **DOAR** numerele prime din cele trimise în cerere, şi nu toată lista calculată de algoritm.

Sugestie: se poate modifica funcția *serverless* astfel încât să fie de tip **`BiFunction`**. **Alternativ**, se pot încapsula numerele în aceeaşi cerere cu un singur argument.

2. Implementați o aplicație de tip **producător-consumator** folosind 2 funcții *serverless* puse la dispoziție de *framework*-ul Micronaut. Producătorul va prelua fluxul RSS de pe site-ul **xkcd.com** (URL-ul este: https://xkcd.com/atom.xml) şi va trimite XML-ul către consumator. Consumatorul va prelucra XML-ul astfel:

a) se va prelua conținutul tag-ului **`<title>`**

b) se va prelua conținutul atributului **`href`** din tag-urile de tip **`<link href=...></link>`**

```
−<feed xml:lang="en">
  <title>xkcd.com</title>
  <link href="https://xkcd.com/" rel="alternate"/>
  <id>https://xkcd.com/</id>
  <updated>2020-05-01T00:00:00Z</updated>
 −<entry>
    <title>Turtle Sandwich Standard Model</title>
    <link href="https://xkcd.com/2301/" rel="alternate"/>
    <updated>2020-05-01T00:00:00Z</updated>
    <id>https://xkcd.com/2301/</id>
   −<summary type="html">
      <img src="https://imgs.xkcd.com/comics/turtle_sandwich
      Shells." alt="It's possible the bread and shell can be split
    </summary>
  </entry>
```

Răspunsul returnat de consumator la ieşirea standard este format dintr-o listă de perechi **`<TITLE, URL>`** formate utilizând datele extrase din fluxul RSS.

Sugestie: folosiți funcții de tipul **`Supplier`** şi **`Consumer`** / **`BiConsumer`**. Pentru preluarea de pe web a fluxului RSS, se poate folosi biblioteca Kotlin KHTTP (https://khttp.readthedocs.io/en/latest/).

URL-ul către Maven Central pentru dependența KHTTP este:
https://mvnrepository.com/artifact/khttp/khttp/1.0.0

Bibliografie

[1]: Documentație Micronaut - https://micronaut.io/documentation.html
[2]: Cum se creează o aplicație minimală Micronaut - https://guides.micronaut.io/creating-your-first-micronaut-app/guide/index.html
[3]: Funcții *serverless* - https://docs.micronaut.io/latest/guide/serverlessFunctions.html
[4]: Biblioteca KHTTP - https://khttp.readthedocs.io/en/latest/