

# Sisteme Distribuite - Laborator 11

## Micronaut - *serverless computing*

### Micronaut - descriere generală

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

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

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

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

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

**Atenție!** Cei care lucrează de pe stațiile din laborator, săriți direct la Secțiunea “Exemple de tipuri de aplicații folosind Micronaut CLI”.

### Instalarea Micronaut

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

```
wget https://github.com/micronaut-projects/micronaut-starter/releases/download/v3.1.4/micronaut-cli-3.1.4.zip
```

(ultima versiune stabilă disponibilă la momentul scrierii laboratorului este **1.3.4**. Deci verificați dacă nu a apărut una mai nouă. Fiți atenți însă că această tehnologie este în plină dezvoltare (nu a atins maturitatea) și atunci este posibil ca exemplele să aibă probleme pe versiunile mai noi din cauza modificărilor care sunt uneori semnificative în unele zone)

Dezarhivați arhiva descărcată utilizând arhivatorul grafic sau comanda următoare:

```
unzip micronaut-cli-3.1.4.zip
```

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

```
sudo mv micronaut-cli-3.1.4 /opt
```

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

```
export PATH="$PATH:/opt/micronaut-cli-3.1.4/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**.

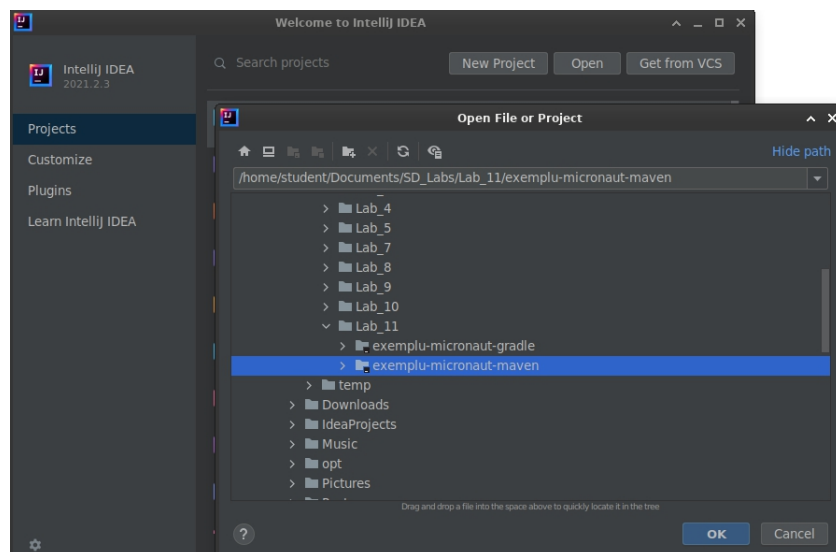


Figura 1 - Importare proiect Micronaut

**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**.

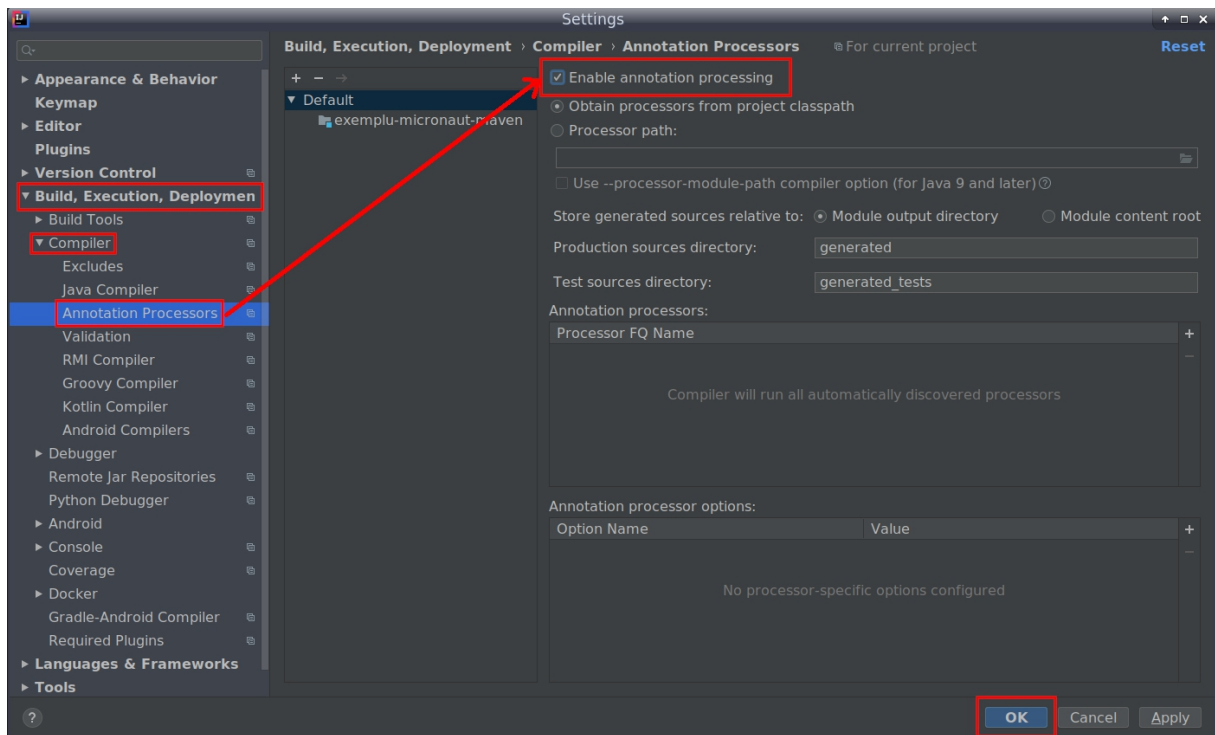


Figura 2 - Activarea procesării adnotărilor în IntelliJ

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

```
package com.sd.laborator

import io.micronaut.runtime.Micronaut.*
fun main(args: Array<String>) {
    build()
        .args(*args)
        .packages("com.sd.laborator")
        .start()
}
```

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 Hello
```

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.HttpStatus

@Controller("/hello")
class HelloController {
    @Get(uri="/", produces=["text/plain"])
    fun index(): 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 (parametrul **uri** de la adnotarea **@Get**) către calea „/” relativ la „/hello”.

Parametrul **produces** de la adnotarea **@Get** indică tipul de răspuns returnat clientului. Deoarece JSON este tipul implicit de răspuns iar șirul de caractere returnat este nestructurat, se marchează explicit tipul „text plain”.

### Execuția aplicației Micronaut din IntelliJ

#### Execuția cu Maven

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

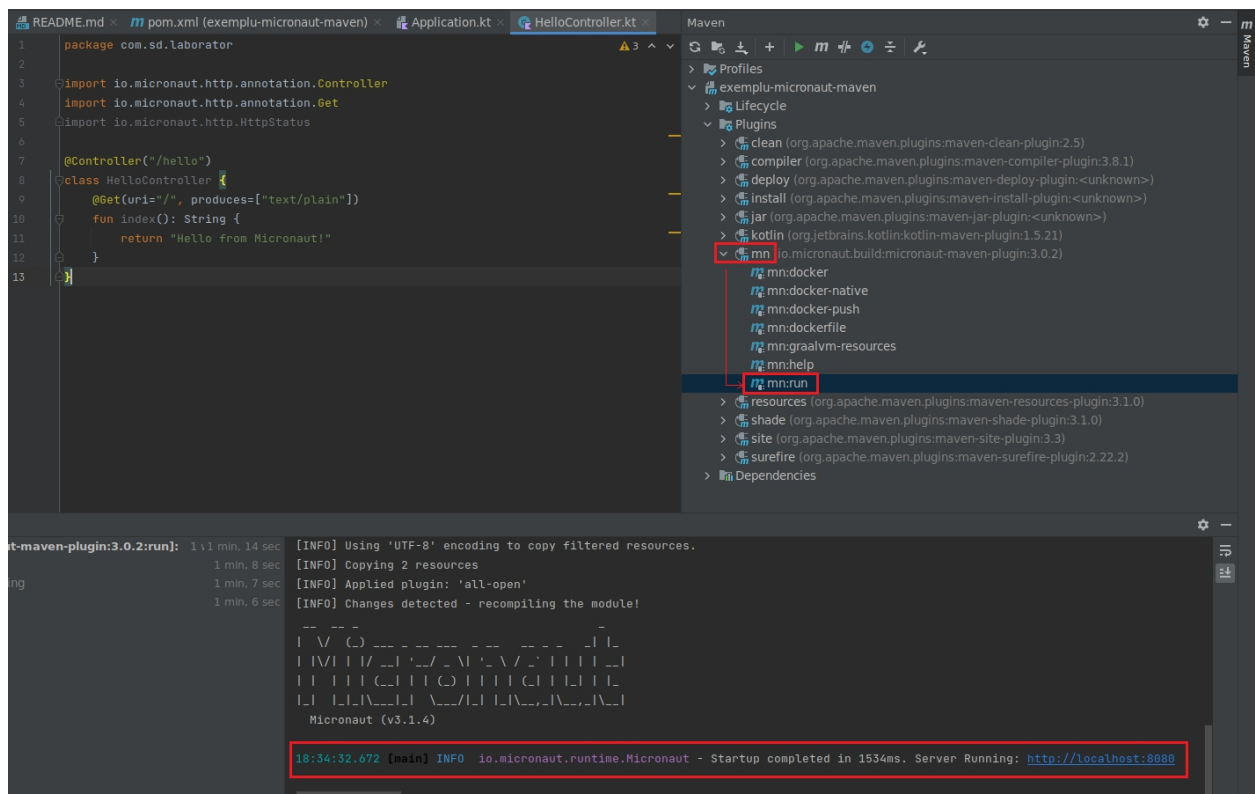


Figura 3 - Execuția unui proiect Micronaut cu schelet Maven

#### Execuția cu Gradle

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

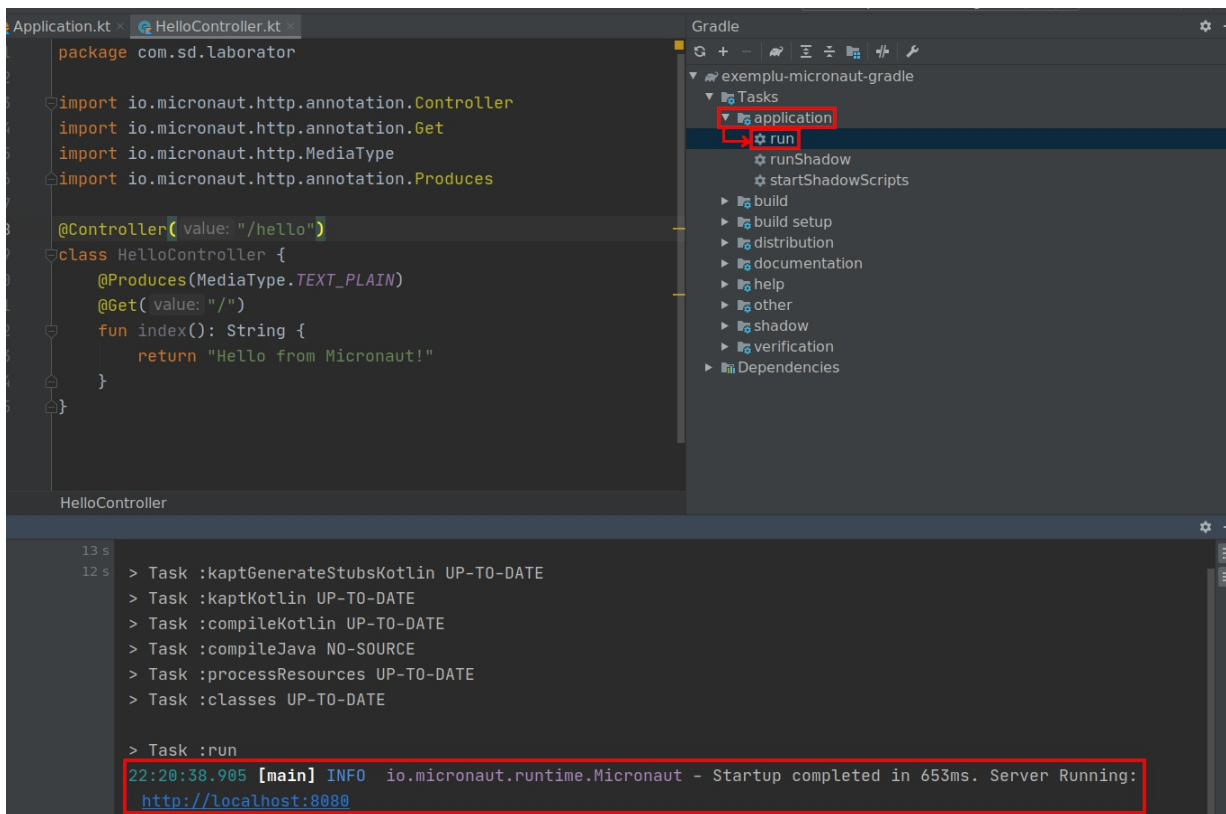


Figura 4 - Execuția unui proiect Micronaut cu schelet Gradle

### *Testarea aplicației Micronaut*

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

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

## Funcții *serverless*

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

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

### Crearea unei funcții *serverless* cu Micronaut CLI

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

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

```
mn create-function-app com.sd.laborator.book-example-maven --lang  
kotlin --build maven
```

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

```
mn create-function-app com.sd.laborator.book-example-gradle --lang  
kotlin --build gradle
```

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

Dacă în comanda `create-function-app` nu se folosește opțiunea `--features`, atunci implicit aceasta se execută cu parametrul `aws-lambda`. Pentru mai multe detalii despre această comandă:

```
mn create-function-app --help
```

Scheletul de cod generat conține următoarele clase:

- **Book**

```
package com.sd.laborator  
import io.micronaut.core.annotation.Introspected  
  
@Introspected  
class Book {  
    var name: String? = null  
}
```

- **BookSaved**

```
package com.sd.laborator  
import io.micronaut.core.annotation.Introspected  
  
@Introspected  
class BookSaved {  
    var name: String? = null  
    var isbn: String? = null  
}
```

Adnotarea `@Introspected` indică faptul că acea clasă peste care este aplicată va produce un tip de dată **BeanIntrospection** în momentul compilării. Acesta este rezultatul unei

procesări făcute la compilare, procesare ce include proprietăți și metadata: prin tipul de dată **BeanIntrospection**, se pot instanția *bean*-uri și scrie / citi proprietăți din acestea fără a folosi reflecția.

- **BookRequestHandler**

```
package com.sd.laborator
import io.micronaut.core.annotation.Introspected
import io.micronaut.function.aws.MicronautRequestHandler
import java.util.UUID

@Introspected
class BookRequestHandler : MicronautRequestHandler<Book?, BookSaved?>()
{
    override fun execute(input: Book?): BookSaved? {
        return if (input != null) {
            val bookSaved = BookSaved()
            bookSaved.name = input.name
            bookSaved.isbn = UUID.randomUUID().toString()
            return bookSaved
        } else {
            null
        }
    }
}
```

Se observă că *BookRequestHandler* extinde clasa abstractă *MicronautRequestHandler*, care se ocupă strict de partea de serverless pe AWS - Amazon Web Services (pentru celelalte opțiuni de *cloud* vezi despre opțiunea *features* discutată anterior). Acest *handler* al funcției lambda, implementează metoda care se ocupă de procesarea evenimentelor care au declanșat-o. Când funcția este invocată, serviciul AWS Lambda execută metoda definită în *handler*.

În continuare, pentru testarea exemplului vom folosi 2 opțiuni:

- folosind clasa de test generată
- folosind un controller care va invoca funcția la primirea unei cereri HTTP

## *Testarea funcției serverless*

### *Testarea folosind clasa de test*

Testarea este o componentă importantă în ciclul de dezvoltare a unui produs software. La firme veți observa practica de a folosi clase de test pentru verificarea/testarea codului scris cu scopul de a mări coeficientul de acoperire al codului (eng. *code coverage*). Mai multe detalii despre testarea produselor software veți afla la disciplina Ingineria Programării.

În cadrul creării aplicației, a fost generată clasa de test *BookRequestHandlerTest*:

- **BookRequestHandlerTest**

```
package com.sd.laborator
import org.junit.jupiter.api.Test
import org.junit.jupiter.api.Assertions

class BookRequestHandlerTest {

    @Test
    fun testHandler() {
```

```

    val bookRequestHandler = BookRequestHandler()
    val book = Book()
    book.name = "Building Microservices"
    val bookSaved = bookRequestHandler.execute(book)
    Assertions.assertNotNull(bookSaved)
    Assertions.assertEquals(book.name, bookSaved!!.name)
    Assertions.assertNotNull(bookSaved.isbn)
    bookRequestHandler.applicationContext.close()
  }
}

```

În cadrul acestei metode se fac 3 teste, date de clasa *Assertions*. Pe scurt, la apelarea unei metode al acestei clase se compară rezultatul execuției cu rezultat-ul așteptat. Dacă acestea nu coincid, atunci testul eșuează.

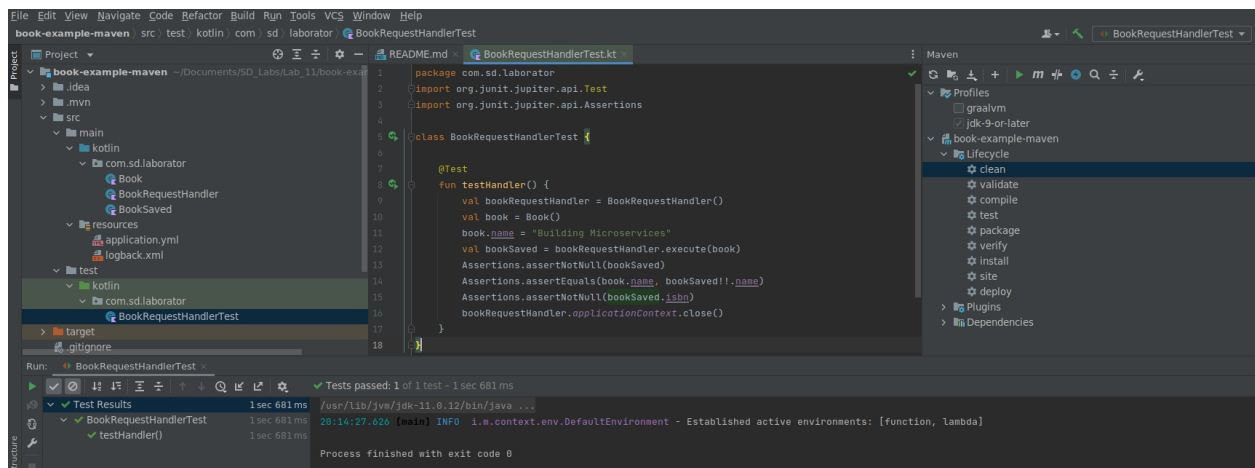


Figura 5 - Exemplu rulare a clasei de test

În acest exemplu testul se execută cu succes. Încercați să modificați metoda *assertNotNull* în *assertNull* și vedeți ce se întâmplă.

### Testarea locală folosind ca declanșator cereri HTTP

Creați un fișier *Application.kt* în pachetul *com.sd.laborator*.

- **Application.kt**

```

package com.sd.laborator

import io.micronaut.http.annotation.Body
import io.micronaut.http.annotation.Controller
import io.micronaut.http.annotation.Post
import io.micronaut.runtime.Micronaut

object Application {
    @JvmStatic
    fun main(args: Array<String>) {
        Micronaut.run(Application::class.java, *args)
    }
}

@Controller
class LambdaController {
    @Post
    fun execute(@Body book: Book?): BookSaved? {
        return handler.execute(book)
    }
}

```



```

    }

    companion object {
        private val handler = BookRequestHandler()
    }
}

```

Adnotarea `@JvmStatic` aplicată asupra funcției `main()` determină generarea unei metode statice adiționale pe baza funcției existente aflate sub influența adnotării. Controller-ul *LambdaController* va fi folosit pentru recepția cererilor HTTP de tipul POST, și va declanșa funcția lambda creată apelând metoda *execute* al *handler*-ului.

### Pentru Maven

În continuare vom crea un profil maven în care vom specifica container-ul în care va fi executat obiectul respectiv. În fișierul *pom.xml*, ca subordonat al tag-ului *project*, adăugați:

```

<profiles>
  <profile>
    <id>local</id>
    <properties>

<exec.mainClass>com.sd.laborator.Application</exec.mainClass>
      <micronaut.runtime>netty</micronaut.runtime>
    </properties>
    <dependencies>
      <dependency>
        <groupId>io.micronaut</groupId>
        <artifactId>micronaut-http-server-
netty</artifactId>
        <scope>compile</scope>
      </dependency>
    </dependencies>
  </profile>
</profiles>

```

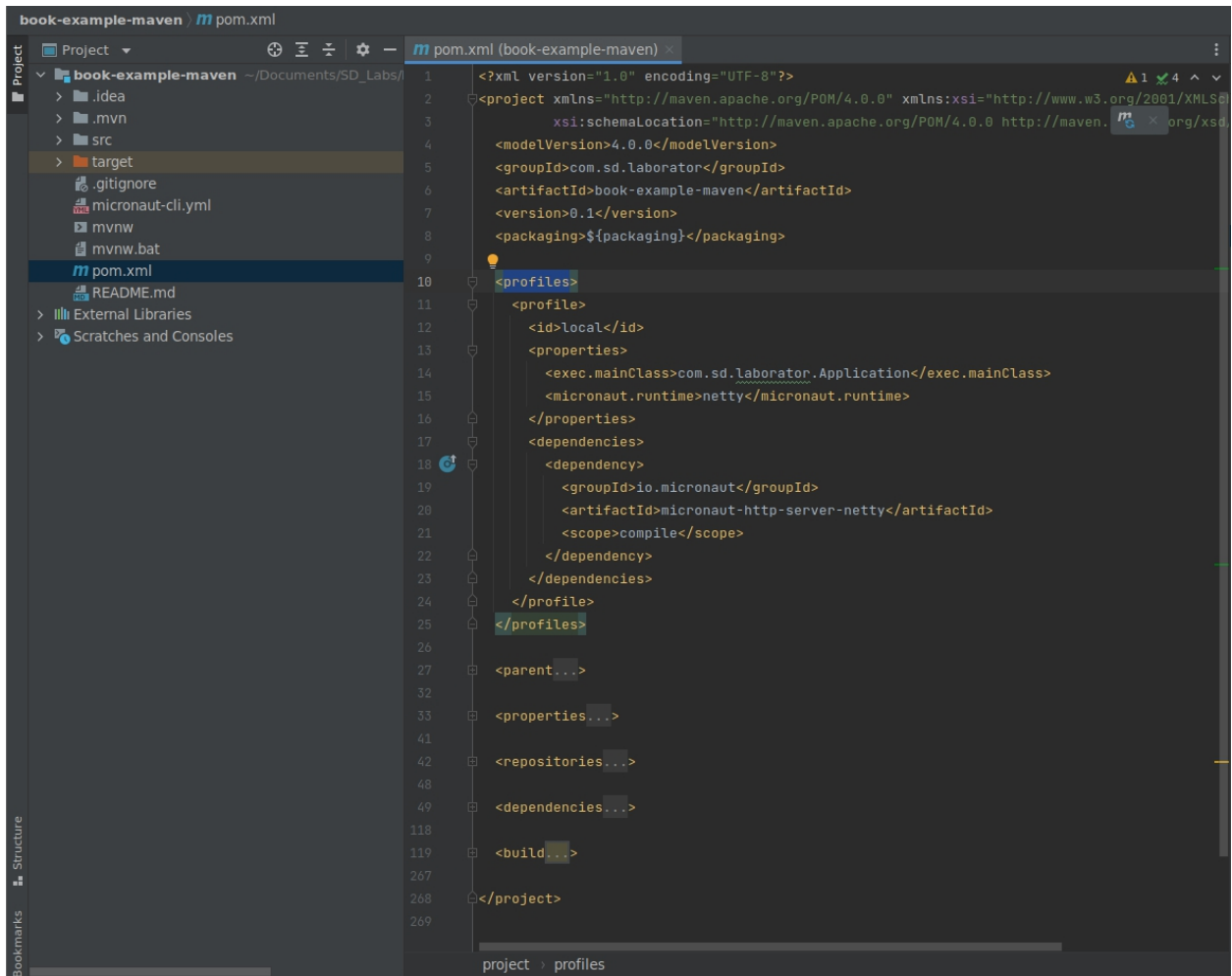


Figura 6 - Adăugarea unui profil în maven

Rulați comanda **mn:run -P local** folosind utilitarul maven.

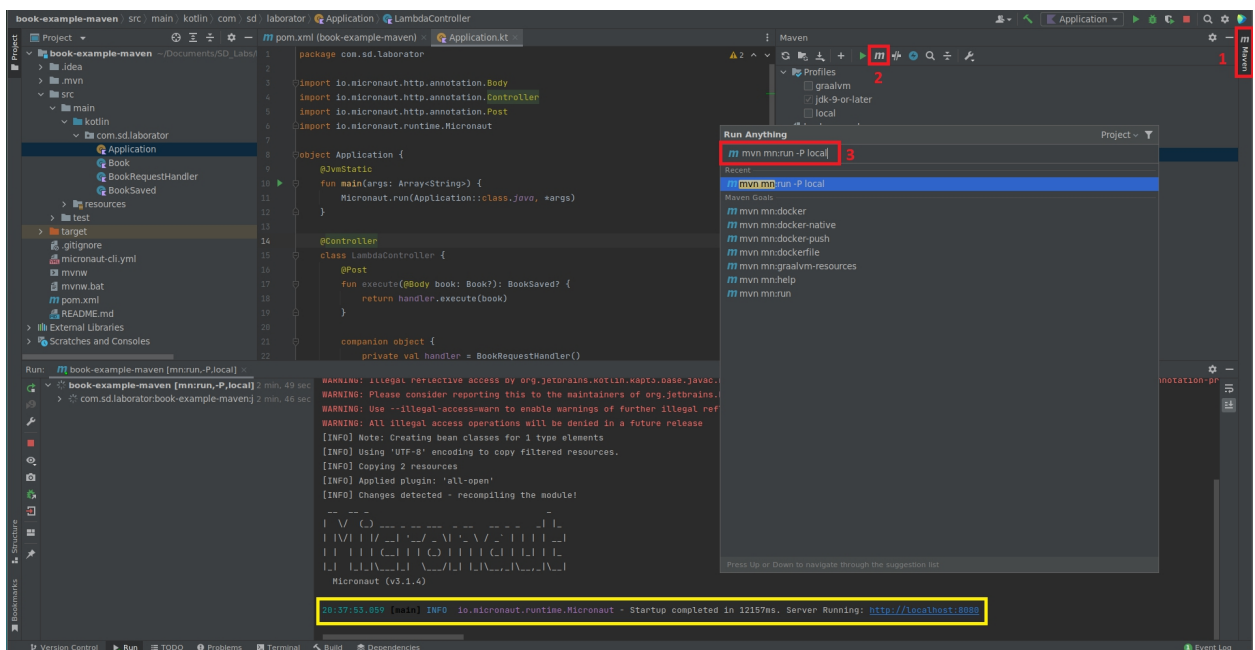


Figura 7 - Execuția aplicației folosind gestionarul maven

Comanda pentru un exemplu de testare este, așadar:

```
curl --location --request POST 'http://localhost:8080/' \
--header 'Content-Type: application/json' \
--data-raw '{"name": "Test"}'
```

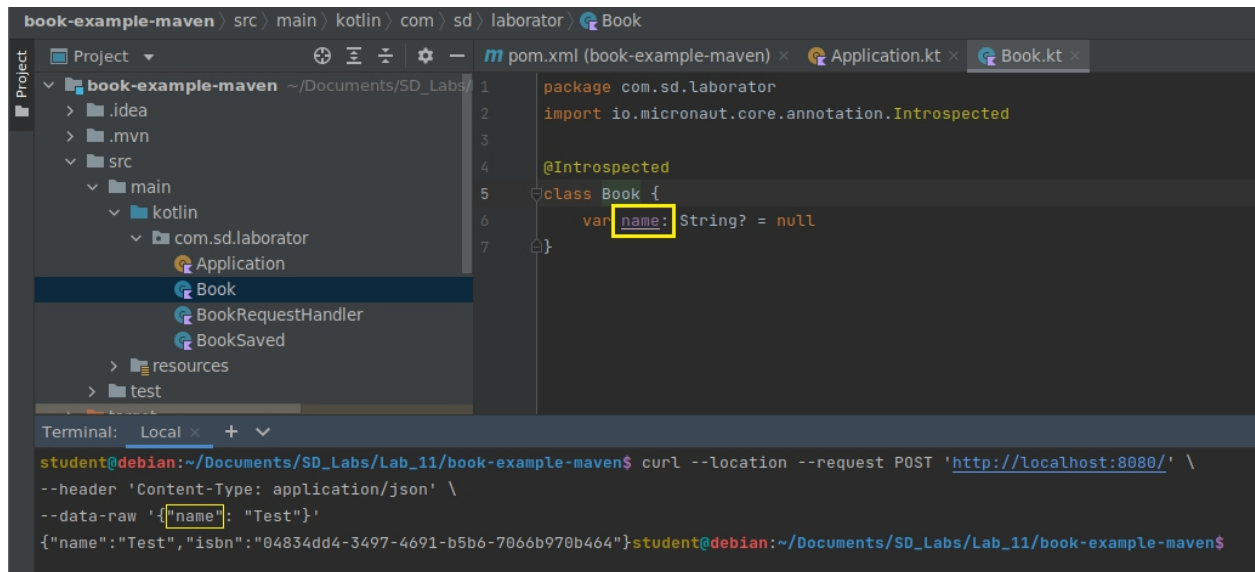


Figura 8 - Execuția funcției *serverless*

## Pentru Gradle

Copiați fișierul *gradle.build* într-un fișier separat, denumit *gradle-local.build*, de exemplu. Vom crea o configurare nouă pentru testarea locală.

Efectuați următoarele modificări în cadrul fișierului nou creat *gradle-local.build*:

- înlocuiți `id("io.micronaut.library")` din cadrul tag-ului *plugins* cu `id("io.micronaut.application")`
- înlocuiți `runtime("lambda")` din cadrul tag-ului *micronaut* cu `runtime("netty")`
- adăugați următoarea secvență în fișierul nou creat *gradle-local.build*:

```
application {
    mainClass.set("com.sd.laborator.Application")
}
```

- adăugați următoarea secvență în fișierul *settings.gradle*:

```
rootProject.buildFileName="build-local.gradle"
```

Implicit, parametrul pentru *rootProject.buildFileName* este *build.gradle*.

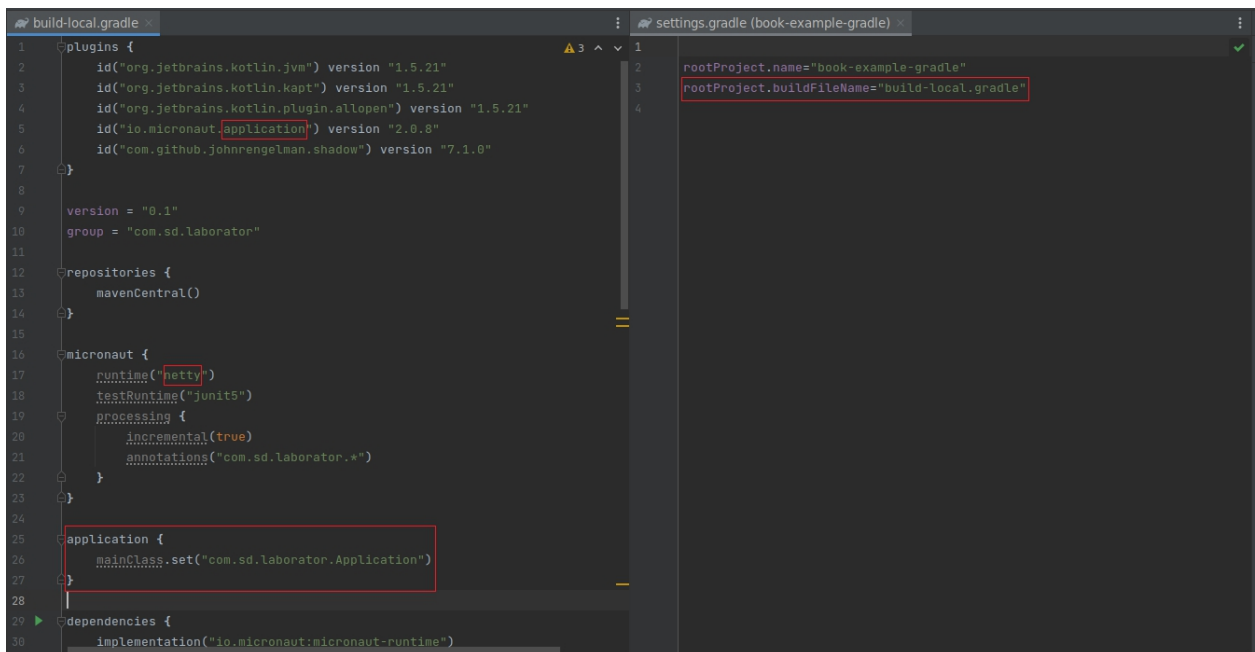


Figura 9 - Adăugarea unui nou fișier de configurare în gestionarul gradle

Pentru testare folosiți *task*-ul *run* din cadrul *application*.

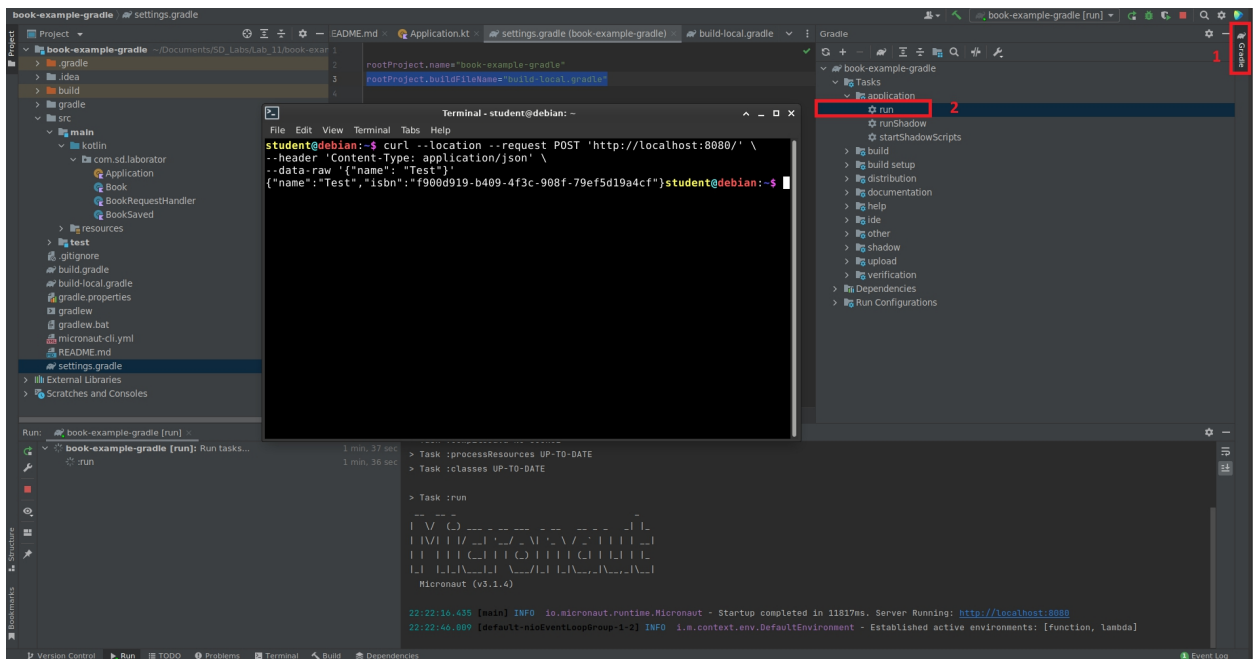


Figura 10 - Execuția aplicației folosind gestionarul gradle

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

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

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

Ștergeți clasele din codul schelet rezultat, **inclusiv clasa de test**.

- Adăugați clasa **EratosteneRequest**:

```
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 clasa **@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 jakarta.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.

- Adăugați clasa **EratosteneFunction** 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 jakarta.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))})
}

```

Se observă adnotarea **@FunctionBean** aplicată clasei **EratosteneFunction**. 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, adică **EratosteneRequest**, iar al doilea reprezintă tipul de date returnat la ieșire: **EratosteneResponse**.

Constructorul **FunctionInitializer** este utilizat pentru inițializarea unei funcții Micronaut. Comportamentul principal al funcției *serverless* este încapsulat în metoda **apply**, specificată în interfața **Function** (față de metoda *execute* al exemplului anterior cu *handler*). Această metodă **aplică** funcția peste argumentul primit la intrare și returnează datele de ieșire.

Se folosește 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.

- Adăugați fișierul *Application* în care va fi implementat un declanșator HTTP, asemănător cu exemplul anterior:

```

package com.sd.laborator

import io.micronaut.http.annotation.Body
import io.micronaut.http.annotation.Controller
import io.micronaut.http.annotation.Post
import io.micronaut.runtime.Micronaut

object Application {
    @JvmStatic

```



```

fun main(args: Array<String>) {
    Micronaut.run(Application::class.java, *args)
}

@Controller
class LambdaController {
    @Post
    fun execute(@Body request: EratosteneRequest):
EratosteneResponse {
        return handler.apply(request)
    }

    companion object {
        private val handler = EratosteneFunction()
    }
}
}

```

În funcție de gestionarul care îl folosiți pentru proiect, creați fie un profil (în cazul maven), fie un alt fișier de configurare (în cazul gradle) pentru testarea locală a aplicației *serverless*.

Testați aplicația cu comanda:

```

curl --location --request POST 'http://localhost:8080/' \
--header 'Content-Type: application/json' \
--data-raw '{"number": 50}'

```

În exemplul dat a fost folosită interfața **Function** pentru crearea funcției lambda. Există mai multe tipuri de funcții Micronaut:

Interfață	Descriere
<b>Supplier</b>	Nu acceptă niciun argument și returnează un singur rezultat
<b>Consumer</b>	Acceptă un singur argument și nu returnează niciun rezultat
<b>Biconsumer</b>	Acceptă 2 argumente și nu returnează niciun rezultat
<b>Function</b>	Acceptă un singur argument și returnează un singur rezultat
<b>BiFunction</b>	Acceptă 2 argumente și returnează un singur rezultat

## Aplicații și teme

### Temă de laborator

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

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

$$a_0 = 1$$

### Teme pentru acasă

1. Transformați funcția *serverless* din ciurul lui Eratostene în așa fel încât aplicația să primească, pe lângă numărul maxim până la care algoritmul face calculul necesar, date de la o coada implementată în alta funcție *serverless* care primește de la un serviciu normal care pune în coada o listă de numere întregi citite dintr-un fișier. Aplicația trebuie să utilizeze ciurul lui Eratostene deja implementat ca să decidă care din numerele respective sunt prime sau nu. Se vor returna **DOAR** numerele prime din cele trimise în cerere, și nu toată lista calculată de algoritm.
2. Implementați un contor de apăsări ale unui buton dintr-o pagină html (vezi abordarea din curs) ca o funcție *serverless*. Butonul va trimite evenimentele într-o coadă RabbitMQ de unde funcția le va prelua apoi procesa (incrementează numărul de procesări și apoi va modifica un parametru dintr-o tabelă menținută într-un server MySQL).
3. Pe baza punctului doirez realizați o pagină web mai complexă unde se vor pune în coadă și apoi salva în tabelă atât numele butonului cât și numărul de apăsări
4. Implementați o aplicație de tip **producător-consumator** folosind 2 funcții *serverless* puse la dispoziție de *framework*-ul Micronaut. Producătorul va prelua fluxul RSS de pe site-ul **xkcd.com** (URL-ul este: <https://xkcd.com/atom.xml>) și va trimite XML-ul către consumator. Consumatorul va prelucra XML-ul astfel:

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">
    
  </entry>

```

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

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

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

## Bibliografie

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