APLICAREA FRAMEWORKULUI QUARKUS ÎN CADRUL ARHITECTURII BAZATE PE MICROSERVICII

Gabriel TUREȚCHI[1], Olga CERBU[2*]

[1]Informatica, IA-201, FCIM, UTM, Chișinău, Moldova
[2]Matematică și Informatică, USM, Chișinău, Moldova

*Autorul corespondent: Olga Cerbu, olga.cerbu@gmail.com

Rezumat. *Acest articol abordează o metodă de soluționare în optimizare a limbajului de programare Java în cadrul tehnologiilor cloud, container și a arhitecturilor bazate pe microservicii. Stratul de abstractizare JVM a permis aplicațiilor Java să ruleze pe orice dispozitiv astfel aducându-i o popularitate printre programatori, dar acum în cadrul noilor tehnologii care permit încapsularea și rularea aplicațiilor în mediul lor nativ, această virtualizare doar încetinesc aplicațiile. Pentru a soluționa această problemă a fost introdus frameworkul Quarkus care permite limbajului Java să concureze cu alte limbaje de programare cu o performanță sporită în cadrul diferitor arhitecturi.*

Cuvinte cheie*: microservicii, quarkus, container, abstractizare, nativ.*

Introducere

În prezent, inginerii și programatorii au multe opțiuni tehnologice pentru a rezolva o problemă de afaceri sau tehnică. Java rămâne unul dintre cele mai utilizate limbaje de programare pentru a construi aplicații. Acest limbaj a fost creat când cloud-ul, containerele și sistemele de gestionare a containerelor, cum ar fi Kubernetes, Docker, încă nu existau[2]. Java a fost întotdeauna (in)faimos în ceea ce privește performanța sa, mai ales din cauza straturilor de abstractizare dintre cod și mașina reală cu scopul de a fi multi-platform (Scrieți o dată, rulați oriunde). În zilele noastre, cu arhitectura bazată pe microservicii, poate că nu mai are sens construirea aplicațiilor universale care vor rula întotdeauna pe același loc și platformă (containerul Docker — mediu Linux), când portabilitatea devine mai puțin relevantă, iar acel nivel suplimentar de abstractizare nu este atât de important.

Frameworkul Quarkus

În martie 2019, după mai bine de un an de dezvoltare internă, Quarkus a fost introdus în comunitatea open source. Quarkus este un framework Java adaptat pentru OpenJDK HotSpot și GraalVM, oferind un timp de lansare rapid și utilizare redusă a memoriei. Multe organizații de la lansarea inițială au văzut valoarea imediată a Quarkus-ului și s-au alăturat efortului de dezvoltare[1].

Quarkus oferă, de asemenea, extindere aproape instantanee și utilizare de înaltă densitate în platforme de gestionare a containerelor, cum ar fi Kubernetes. Mai multe instanțe ale aplicației pot fi rulate în aceleași resurse hardware. Încă de la început, Quarkus a fost proiectat pe baza filozofiilor de tip container-first și Kubernetes-native, optimizând consumul memoriei și timpul de lansare. În timpul asamblării codului aplicației se realizează cât mai multă procesare posibilă, inclusiv luarea unei abordări a ipotezei lumii închise pentru construirea și rularea aplicațiilor. Această optimizare înseamnă că, în majoritatea cazurilor, JVM-ul rezultat conține doar cod care are o cale de execuție în timpul lansării.

În Quarkus, clasele utilizate numai la pornirea aplicației sunt invocate în momentul construirii și nu sunt încărcate în JVM. De asemenea, Quarkus evită reflectarea cât mai mult posibil, favorizând în schimb legarea de clasă statică. Aceste principii de proiectare reduc dimensiunea și în cele din urmă amprenta de memorie a aplicației care rulează pe JVM, permițând frameworkului Quarkus să fie „nativ”.

La baza creării frameworkului Quarkus a stat compilarea nativă. A fost optimizat pentru utilizarea capacităților imaginii native GraalVM pentru a compila bytecode JVM într-un format binar nativ de cod-mașină. GraalVM elimină în mod riguros orice cod inaccesibil găsit în interiorul codului sursă al aplicației, precum și oricare dintre dependențele acesteia. Combinată cu containerele Linux și Kubernetes, o aplicație Quarkus rulează asemenea unui executabil Linux nativ, eliminând JVM-ul. Un executabil nativ Quarkus pornește mult mai rapid și utilizează mult mai puțină memorie decât un JVM tradițional.

Capacitățile de imagine nativă similare celor din frameworkul Spring sunt încă considerate experimentale sau beta. Oportunitățile frameworkului Spring de a susține compilarea nativă nu oferă aceleași optimizări al timpului de compilare și a opțiunilor de proiectare, cum are loc în cadrul frameworkului Quarkus, extrem de rapid și eficient din punct de vedere al memoriei atunci când rulează pe JVM sau într-o imagine nativă.

Quarkus nu se axează doar pe furnizarea tehnologiilor noi, pentru că fiecare inovație ar trebui să fie simplă, să aibă o configurație redusă sau deloc și să fie cât mai intuitivă de utilizat. Ar trebui să fie simplu să creezi lucruri esențiale și în același timp relativ ușor să faci lucruri complexe, permițând dezvoltatorilor să se concentreze pe expertiza domeniului și logica lor de afaceri.

O problemă majoră de productivitate cu care se confruntă majoritatea dezvoltatorilor Java astăzi este fluxul de lucru tradițional de dezvoltare Java. Pentru majoritatea dezvoltatorilor, acest proces arată ca în Fig. 1.

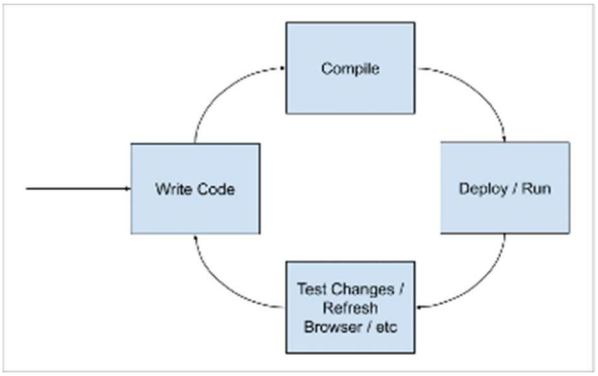

Figura 1. Ciclul tradițional de lucru

Menționăm că ciclul de Compilare și Deploy/Run durează un minut sau mai mult aceasta fiind un dezavantaj în procesul de execuție. Această întârziere este timp pierdut în care un dezvoltator ar putea soluționa alte sarcini. Tehnologiile de codare live a frameworkului Quarkus rezolvă problema timpului lent de execuție. Quarkus va detecta automat modificările aduse fișierelor Java, inclusiv refactorizările clasei sau a metodei, configurația aplicației, resursele statice sau chiar modificările dependenței de clasă. Când este detectată o astfel de modificare, Quarkus recompilează și redistribuie în mod transparent modificările. Redistribuirea Quarkus are loc de obicei în mai puțin de o secundă. Folosind Quarkus, fluxul de lucru de dezvoltare arată acum ca în Fig. 2.

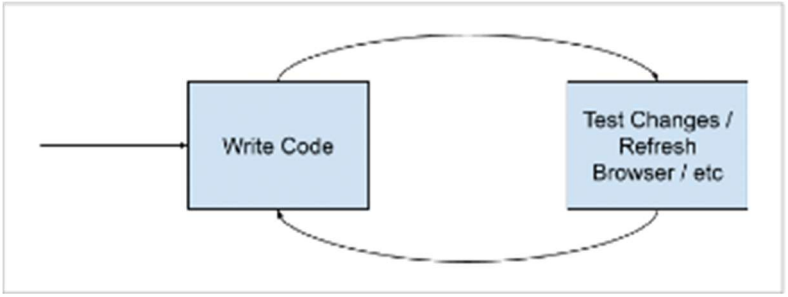

Figura 2. Ciclul Quarkus de lucru

Compararea Frameworkurilor Quarkus și Spring Boot

Spring este o platformă open source bine cunoscută printre rândurile programatorilor Java ce oferă diferite module care ușurează și accelerează dezvoltarea aplicațiilor. Pentru a compara aceste două frameworkuri a fost creată o simplă aplicație care simulează un API ce afișează, adaugă și șterge un element dintr-un obiect JSON. Pentru crearea containerului a fost folosită aplicația Docker [2].

```java
@Path("/api")
public class SimpleResource {

    private Set<JsonExample> jsonObject = Collections.newSetFromMap(Collections.synchronizedMap(new LinkedHashMap<>()));

    public SimpleResource(){
        this.jsonObject.add(new JsonExample( word: "first", UUID.randomUUID().toString(), extractIP()));
        this.jsonObject.add(new JsonExample( word: "second", UUID.randomUUID().toString(), extractIP()));
        this.jsonObject.add(new JsonExample( word: "third", UUID.randomUUID().toString(), extractIP()));
    }

    @GET
    @Path("/list")
    @Produces(MediaType.APPLICATION_JSON)
    public Set<JsonExample> list() {
        return this.jsonObject;
    }

    @GET
    @Path("/add/{word}")
    @Produces(MediaType.APPLICATION_JSON)
    public Set<JsonExample> add(@PathParam String word){
        this.jsonObject.add(new JsonExample(word, UUID.randomUUID().toString(), extractIP()));
        return jsonObject;
    }

    @GET
    @Path("/delete/{word}")
    @Produces(MediaType.APPLICATION_JSON)
    public Set<JsonExample> delete(@PathParam String word){
        this.jsonObject.removeIf(existingJsonElement -> existingJsonElement.word.equals(word));
        return this.jsonObject;
    }

    private String extractIP(){
        try {
            return InetAddress.getLocalHost().getHostAddress();
        } catch (UnknownHostException e) {
            return e.getMessage();
        }
    }
}
```

Figura 3. Aplicația Quarkus

După cum observăm în Fig. 3 la nivel de sintaxă, Quarkus este destul de intuitiv ceea ce ne permite să ne axăm mai mult pe logica aplicației.

În cadrul experimentului, aplicațiile Quarkus și Spring Boot au fost încapsulate în diferite containere folosind Docker, fiecare container a fost recreat și testat de 10 ori, iar pe baza timpului de pornire și a memoriei consumate a fost creată o statistică aproximativă de performanță [3].

În Fig. 4 se observă că Quarkus a fost lansat în două moduri diferite, primul este cel nativ care oferă o performanță multă mai bună din cauza lipsei stratului de abstractizare dintre aplicație și mașină. Cel de-al doilea mod unde este prezent stratul de abstractizare JVM este mai lent, dar similar primului caz este mult mai rapid decât Spring Boot.

În Fig. 5 de-asemenea Quarkus a fost lansat în două moduri și la fel putem observa diferențele considerabile referitor la consumul de memorie în raport cu aplicația Spring Boot.

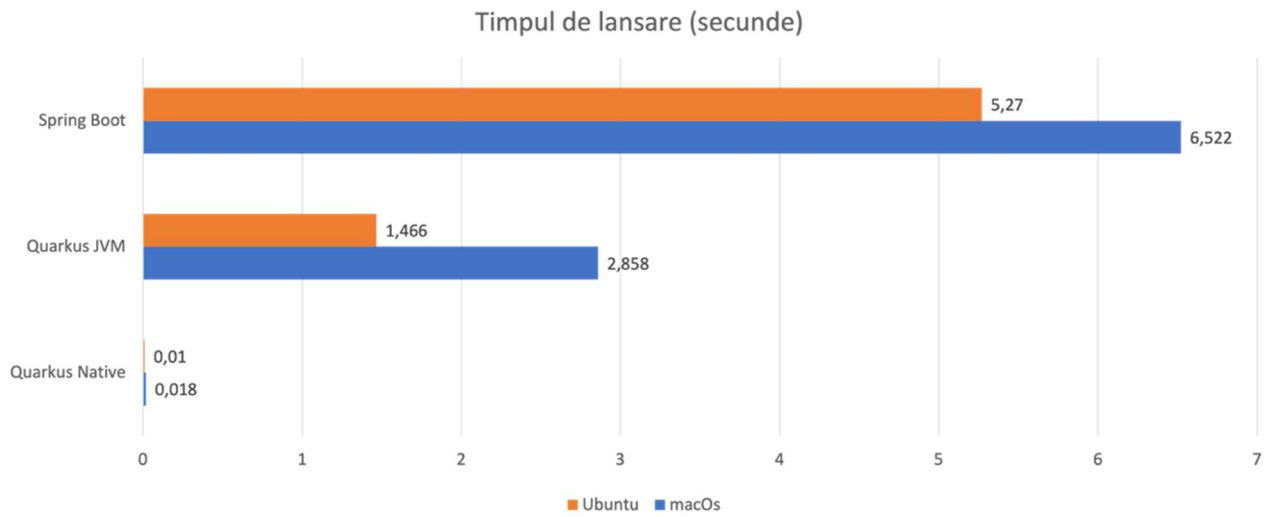

Figura 4. Timpul de lansare a aplicației

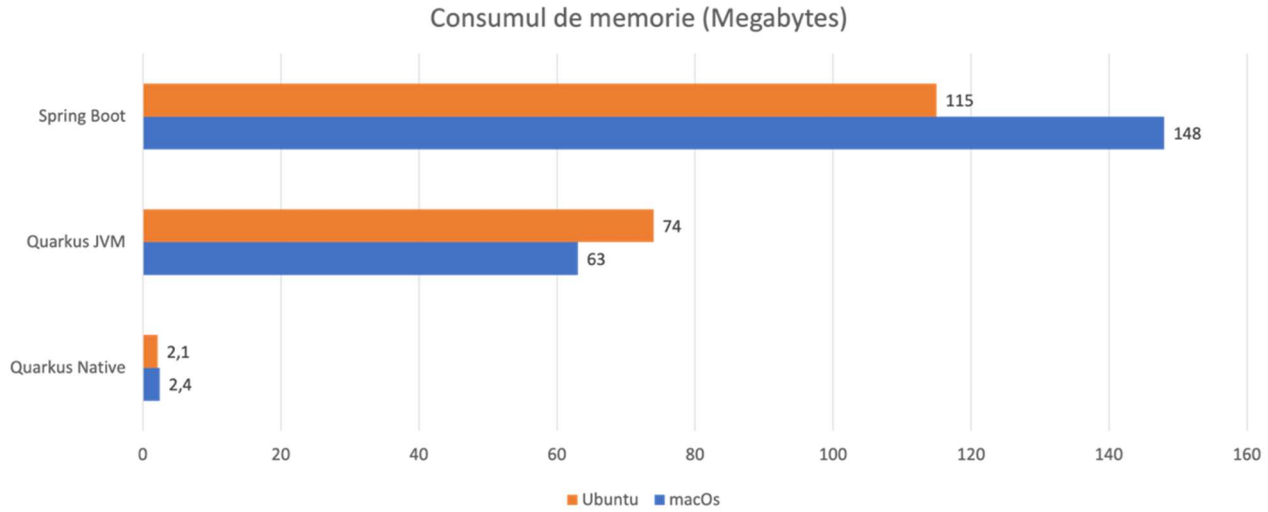

Figura 5. Consumul de memorie la lansarea aplicației

Concluzii

După efectuarea acestui experiment putem concluziona că Quarkus este un framework cu un potențial considerabil care va permite limbajului de programare Java să corespundă noilor tendințe din domeniul tehnologiilor informaționale dar și să concureze cu alte limbaje moderne de programare. Deși Quarkus are o performanță extraordinară în raport cu Spring, la moment este un framework încă tânăr cu o bază de utilizatori și documentație restrânsă ceea ce nu putem spune despre Spring, iar numărul de module și servicii oferite de Spring este imens. Frameworkul Quarkus se îmbină perfect cu diferite module Spring ceea ce conduce la o alegere perfectă pentru proiectele care au la bază arhitectura de microservicii.

Referinţe

1. RED, HAT, *Quarkus For Spring Developers,* E-Book, 2021.
2. NEBRASS, L., *Pro Java Microservices with Quarkus and Kubernetes,* Berkly: Apress, 2021.
3. Microservices Quarkus vs Spring Boot. [online].
   Disponibil: https://dzone.com/articles/microservices-quarkus-vs-spring-boot