FIAP ON

SPRING

SPRING NATIVE

FABIO TADASHI MIYASATO

4

LISTA DE FIGURAS

LISTA DE CÓDIGOS-FONTE

SUMÁRIO

4 SPRING NATIVE

O Spring Native oferece suporte para compilar aplicações Spring de forma nativa utilizando a GraalVM. Ele consome muito menos memória com um tempo de inicialização bastante inferior ao de uma aplicação Spring sendo executada na JVM, mas apresenta como grande desvantagem um maior tempo de compilação.Essas características tornam o Spring Native uma boa opção para boa parte das aplicações Spring disponíveis em Cloud, como microsserviços, REST APIs e funções Serverless entre outras aplicações que são executadas em containers e Kubernetes.

4.1 Versão Beta

Atualmente, o Spring Native encontra-se em versão Beta, ou seja, ainda está em testes e pode sofrer mudanças até sua versão final e estável. A primeira versão beta foi disponibilizada à comunidade em março de 2021 através de um *post* de Sébastian Deleuze, o principal responsável técnico pelo desenvolvimento, no blog do Spring em (https://spring.io/blog/2021/03/11/announcing-spring-native-beta). No *post*, Sébastian relata que o Spring Native estava em desenvolvimento desde novembro de 2020. Dessa forma, mesmo que seja beta, esta já pode ser considerada uma versão de maturidade razoável. Desde a data do *post*, a cada beta *release* é possível observar uma melhora nos tempos de compilação e nos indicativos de uso de memória e tempo de inicialização. No entanto, esta diferença é menos acentuada nas versões mais recentes, o que também é outro indicativo que o módulo está bastante maduro. Vale ressaltar que, mesmo com esses indicativos, qualquer *framework* ou biblioteca em versão beta não deve ser utilizado em aplicações em produção.

4.1.1 Compilação Nativa

O Spring Native não é o primeiro projeto a fornecer compilação nativa para aplicações Java, as quais tradicionalmente rodam na JVM. *Frameworks* Java como Quarkus e Micronaut já oferecem suporte estável a aplicações Java rodando de forma

nativa com a GraalVM; inclusive, o Quarkus Native ainda possui vantagem nos tempos quando comparado com o Spring Native, conforme evidencia a Figura “Tabela comparativa entre frameworks JVM e Nativo”.

Metrics	Spring Boot JVM	Quarkus JVM	Spring Boot Native	Quarkus Native
Startup time (sec)	4.75	2.69	0.20	0.12
Build artifact time (sec)	22.44	7.9	176	121
Artifact size (MB)	28.5	31.5	30.5	31.5
Loaded classes	9842	8863	23445	15658
CPU usage max(%)	60	50	30	20
CPU usage average(%)	18	18	20	20
Heap size startup (MB)	317	264	-	-
Used heap startup (MB)	68	14	44	27
Used heap max (MB)	220	170	480	515
Used heap average (MB)	200	150	300	315
Max threads	23	37	18	28
Average response time (ms)	662	689	591	609
Response time p90 (ms)	942	1027	988	1028

Figura 4.1 – Tabela comparativa entre frameworks JVM e Nativo
Fonte: Baeldung (2022)

Apesar de ter um ótimo *startup time* (tempo de inicialização), o Spring Boot Native consome 0.2 segundos, o que é quase o dobro do tempo de inicialização do Quarkus Native: aproximadamente 0.12 segundos. Um dado interessante é que o uso de memória Heap é menor no início com o Quarkus, com 27 MB, do que com os 44 MB utilizados pelo Spring Native. Entretanto, o uso máximo de memória é maior no Quarkus: 515 MB *versus* 480 MB do Spring. Ademais, na média o Spring também leva vantagem com 300 MB *versus* 315 MB do Quarkus. Em geral, pode-se dizer que os números apresentados pelo Spring Native são excelentes se comparados a um *framework* nativo já estabelecido.

A grande vantagem do Spring Native é a compatibilidade com o Spring Framework, que é o *framework* mais utilizado em Java. Segundo Sébastian Deleuze, fazer parte do Spring Framework pode ser visto como a maior força do Spring Native devido à alta compatibilidade com os outros módulos do *framework*; todavia, é o que mais traz dificuldades no desenvolvimento do Spring Native.

Em comparação com as aplicações tradicionais que são executadas na JVM, as aplicações nativas possuem diversas vantagens, como tempo de inicialização quase instantâneo e menor consumo de memória. A principal desvantagem é que o processo de *build* (construção) do projeto se torna muito mais custoso e lento em comparação às aplicações tradicionais. Além disso, as imagens nativas têm menos opções de otimização em tempo de execução (*runtime)*.

As principais diferenças das aplicações nativas para JVM são:

- A análise estática da aplicação é realizada em *build time*.
- As classes não utilizadas são removidas em *build time* (incluindo dependências e JDK).
- É necessário configuração adicional para funcionalidades como *reflections*, *resources* e *proxies* dinâmicos.
- O classpath é fixado em build time.
- Sem o mecanismo de *lazy load* de classes, tudo que for compilado é carregado na inicialização da aplicação.
- Parte do código é executado em tempo de construção (*build time)*.
- Algumas funcionalidades do Java não são suportadas. No caso do Spring Native, a lista de compatibilidade com o restante do *framework* vem crescendo de modo acelerado, mas ainda não é totalmente compatível.

4.1.2 GraalVM

Uma das principais tecnologias envolvidas nos *frameworks* de compilação nativa é a GraalVM, uma máquina virtual Java (JVM - Java Virtual Machine) de alta performance, desenhada para rodar aplicações escritas em Java, Kotlin, Scala e outras linguagens JVM, além de prover a execução (*runtime*) para aplicações JavaScript, Python e muitas outras linguagens.

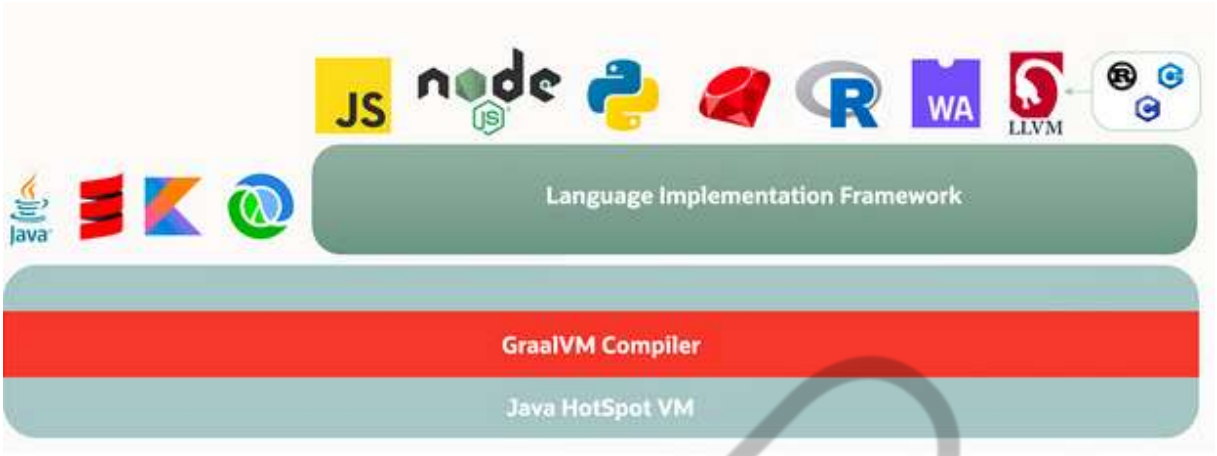

Figura 4.2 – Representação da GraalVM na JVM
Fonte: GraalVM (2022)

A GraalVM oferece um poderoso compilador JiT (Just in Time) que é utilizado pelos principais *frameworks* de compilação nativa Java. Ela disponibiliza três modos de *runtime*:

- **JVM Runtime** é o *runtime* tradicional do Java. Ele utiliza a Java Virtual Machine como ambiente, mas com os benefícios da GraalVM JiT Compiler. As aplicações são carregadas e executadas normalmente na JVM, transformando o código Java ou qualquer outra linguagem compatível para *bytecode*, que a JVM compila para o código de máquina (*machine code*).
- **Imagem nativa** é a inovação que permite compilar um código Java/Kotlin em um código binário executável nativamente pelo sistema operacional. O *bytecode* é processado em tempo de compilação para gerar a imagem nativa, incluindo classes da aplicação, dependências, dependências de terceiros e qualquer outra classe da JDK que for necessária. Este é o *runtime* que *frameworks* como Spring Native utilizam.
- Existe um terceiro modo: o **Java on Truffle**, que é uma implementação da JVM escrita utilizando o *framework* da linguagem de implementação Truffle. Trata-se de uma implementação completa da especificação da JVM que também configura-se como uma forma de implementar outras linguagens de programação na GraalVM.

4.1.3 Ahead of Time Compilation - AOT

O Java Ahead of Time Compilation está relacionado ao Java Enhancement Process, ou processo de melhoria Java, por meio da JEP 295 (https://openjdk.java.net/jeps/295), disponibilizada como experimental no Java versão 9.

O AOT Compilation é uma forma de melhorar a performance das aplicações Java, principalmente o tempo de inicialização na JVM. A JVM executa o *bytecode* Java e compila os códigos mais executados para nativo, um processo denominado de JiT e que é definido em tempo de execução pela JVM. Já o AOT tem como propósito melhorar o processo conhecido como *warming-up* (ou aquecimento) otimizando desta forma as classes JIT na inicialização da JVM.

No caso do Spring, será utilizado o Ahead of Time Compilation com o *plugin* Maven:

```
<build>
    <plugins>
      <!-- ... -->
      <plugin>
        <groupId>org.springframework.experimental</groupId>
        <artifactId>spring-aot-maven-plugin</artifactId>
        <version>0.11.5</version>
        <executions>
          <execution>
            <id>generate</id>
            <goals>
              <goal>generate</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build>
```

Código-Fonte 4.1 – Configuração do plugin AoT no Maven
Fonte: Elaborado pelo autor (2022)

ou Gradle:

```
plugins {;
    // ...
    id("org.springframework.experimental.aot") version "0.11.5"
};
```

Código-Fonte 4.2 – Configuração do plugin AoT no Gradle
Fonte: Elaborado pelo autor (2022)

4.1.4 Compilação

Existem duas formas de compilar uma aplicação com Spring Native: **BuildPack** e **Native**. A principal diferença é que a compilação *buildpack* gera uma imagem Docker leve para ser executada, ao passo que a compilação *native* gera um executável nativo para o sistema operacional.

É importante notar que, para a compilação *buildpack*, é necessário ter o Docker configurado no ambiente, enquanto que na compilação *native* é necessário configurar o Native Build Tools (https://github.com/graalvm/native-build-tools).

4.2 Prática

As aplicações Spring Native são aplicações Spring Boot com algumas restrições de *runtime* da JDK/JVM; no entanto, a grande diferença se dá durante a compilação. Assim sendo, neste exemplo prático serão explorados detalhes de configuração e compilação.

O primeiro passo é criar o projeto e, assim como em qualquer projeto Spring, a melhor forma é através do Spring Initializr (https://start.spring.io/):

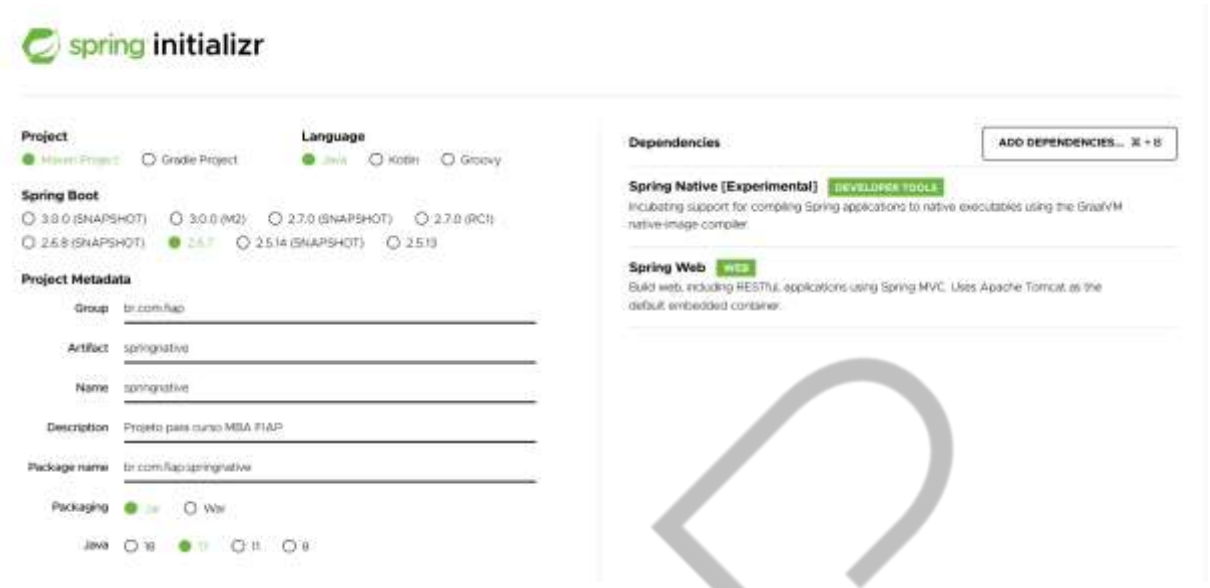

Figura 4.3 – Configuração de projeto no Spring Initializr
Fonte: Spring Initializr (2022)

Ao descompactar o projeto e abrir uma IDE, como IntelliJ, não é possível notar nenhuma diferença com relação a um projeto tradicional que utiliza JVM. Porém, ao abrir o arquivo responsável pelas configurações de *build* e dependências build.gradle, temos duas diferenças chave.

```
plugins {;
    id 'org.springframework.boot' version '2.6.7'
    id 'io.spring.dependency-management' version '1.0.11.RELEASE'
    id 'java'
    id 'org.springframework.experimental.aot' version '0.11.5'
};
```

Código-fonte 4.3 – Configuração do plugin no Gradle
Fonte: Elaborado pelo autor (2022)

Temos a configuração do plugin Ahead of Time Compilation, ainda dentro de um *package* experimental, que é o Spring Native.

```
tasks.named('bootBuildImage') {;
  builder = 'paketobuildpacks/builder:tiny'
  environment = ['BP_NATIVE_IMAGE': 'true']
}
```

Código-fonte 4.4 – Configuração da task buildImage no Gradle
Fonte: Elaborado pelo autor (2022)

Há também a configuração de uma tarefa de *build* chamada “bootBuildImage”, definindo o parâmetro BP_NATIVE_IMAGE = true.

Para aplicações que utilizam Maven, é necessário também adicionar a dependência:

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.6.7</version>
    <relativePath/>
</parent>
```

Código-fonte 4.5 – Configuração do parent no Maven
Fonte: Elaborado pelo autor (2022)

Além de configurar o plugin AOT:

```
<build>
    <plugins>
      <!-- ... -->
      <plugin>
        <groupId>org.springframework.experimental</groupId>
        <artifactId>spring-aot-maven-plugin</artifactId>
        <version>0.11.4</version>
        <executions>
          <execution>
            <id>generate</id>
            <goals>
              <goal>generate</goal>
            </goals>
          </execution>
        </executions>
      </plugin>
    </plugins>
</build>
```

Código-fonte 4.6 – Configuração do plugin no Maven
Fonte: Elaborado pelo autor (2022)

E a configuração da tarefa de buildpack:

```
<plugin>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-maven-plugin</artifactId>
  <configuration>
    <image>
      <builder>paketobuildpacks/builder:tiny</builder>
      <env>
        <BP_NATIVE_IMAGE>true</BP_NATIVE_IMAGE>
      </env>
    </image>
  </configuration>
</plugin>
```

Código-fonte 4.7 – Configuração da task buildImage no Maven
Fonte: Elaborado pelo autor (2022)

É importante ressaltar que estas configurações são definidas para compilação Build Pack, que tem como artefato final uma imagem Docker nativa. Para artefatos nativos é necessário adotar o GraalVM Native Build Tools, conforme visto anteriormente. Para testar a compilação, será criado um Rest Controller, que recebe uma String como parâmetro e retorna esta mesma String de trás para frente, ou “revertida”:

```
1  package br.com.fiap.springnative;
2
3    import org.springframework.web.bind.annotation.GetMapping;
4    import org.springframework.web.bind.annotation.RequestMapping;
5    import org.springframework.web.bind.annotation.RequestParam;
6    import org.springframework.web.bind.annotation.RestController;
7
8    @RestController
9    @RequestMapping("reverse")
10   public class ReverseController {;
11
12   @GetMapping
13   public String reverse(@RequestParam String word){;
14   return new StringBuilder(word).reverse().toString();
15   };
16
17 };
```

Código-fonte 4.8 – Criação de classe Controller
Fonte: Elaborado pelo autor (2022)

Pode-se compilar o projeto e executá-lo da forma tradicional, evitando assim longos tempos de build durante a etapa de desenvolvimento. Então é só “buildar” e, em seguida, executar a aplicação pela própria IDE através do botão “run” ou “debug”, como qualquer outra aplicação Spring.

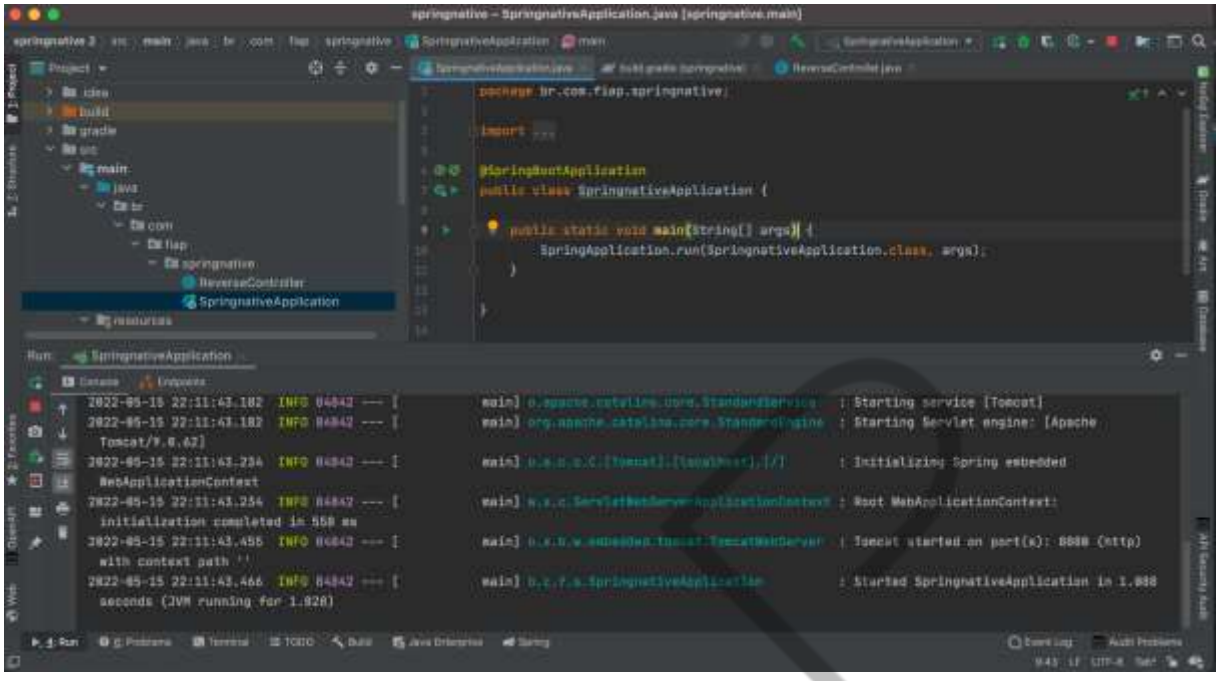

Figura 4.4 – Aplicação rodando na IDE IntelliJ
Fonte: Elaborado pelo autor (2022)

Vale observar que a aplicação simples iniciou em 1,088 segundos. Para obter os benefícios do Spring Native, é necessário realizar a compilação nativa através da nova tarefa configurada utilizando o comando “./gradlew bootBuildImage” (gradle) ou “mvn spring-boot:build-image” (Maven), conforme evidencia a Figura “Log no terminal da aplicação rodando”.

Figura 4.5 – Log no terminal da aplicação rodando
Fonte: Elaborado pelo autor (2022)

Este processo de compilação pode variar bastante conforme a máquina utilizada e a versão do Spring Native. No exemplo da Figura “Tempo de build da aplicação” foi utilizado um equipamento MacBook 2,2GHz 6-core i7 com 16 GBde RAM.

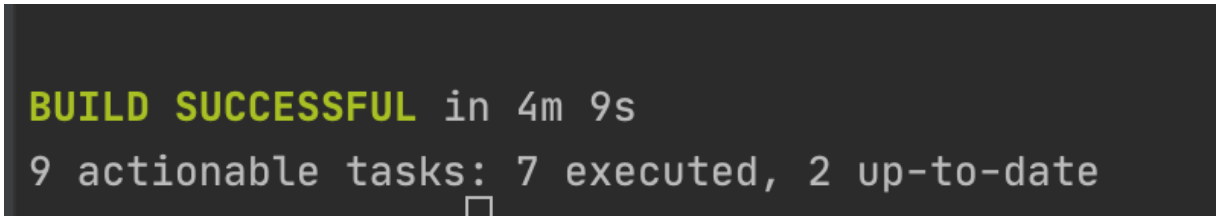

Figura 4.6 – Tempo de build da aplicação
Fonte: Elaborado pelo autor (2022)

Após a compilação, a imagem Docker estará disponível e pode ser conferida através do comando “d*ocker images*”.

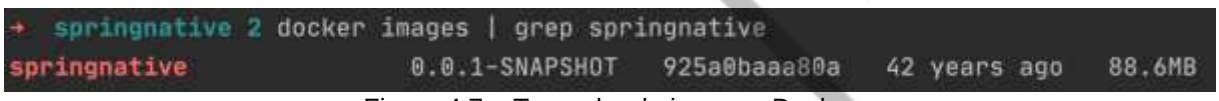

Figura 4.7 – Tamanho da imagem Docker
Fonte: Elaborado pelo autor (2022)

Esta imagem Docker pode ser executada em container como qualquer outra imagem ao se utilizar o comando “docker run -p 8080:8080 springnative:0.0.1-SNAPSHOT”.

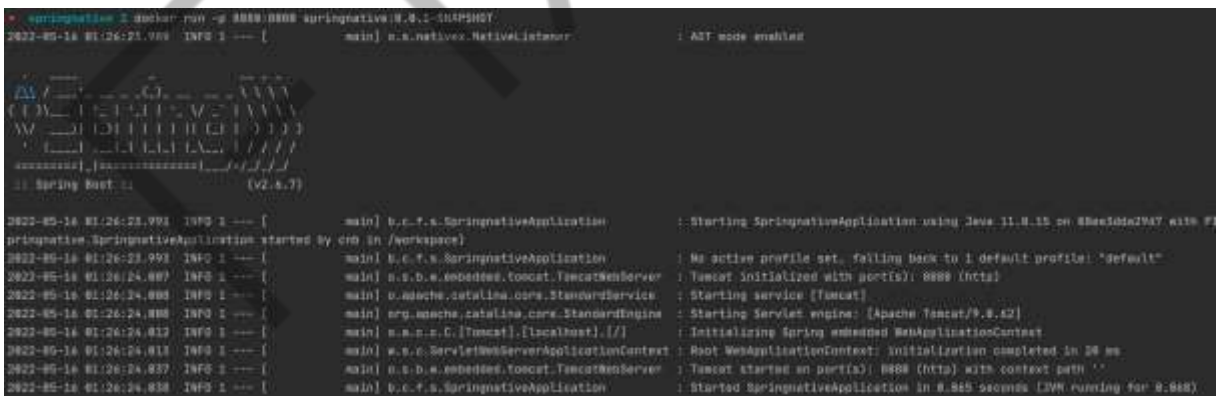

Figura 4.8 – Aplicação rodando via terminal
Fonte: Elaborado pelo autor (2022)

A aplicação inicia em apenas 0,065 segundo, ou seja, mais de 16x mais rápida que a aplicação tradicional JVM. Para testar a aplicação, também não há nenhuma diferença:

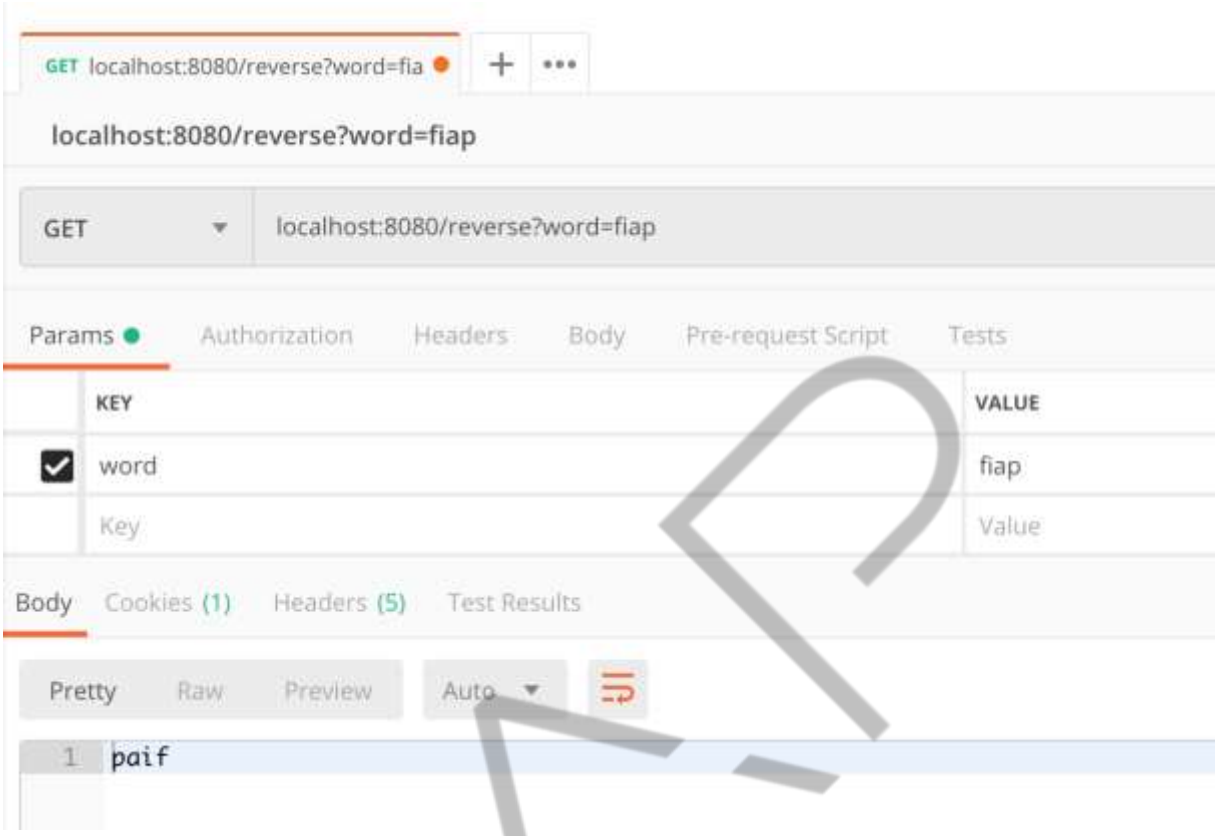

Figura 4.9 – Teste do endpoint utilizando Postman
Fonte: Elaborado pelo autor (2022)

4.3 Comparativos

Os resultados obtidos nos laboratórios anteriores são válidos e podem ser utilizados como base para a avaliação do Spring Native. No entanto, a seguir, serão apresentados mais alguns comparativos desenvolvidos pelo time do Spring e apresentados no blog oficial do Spring (https://spring.io/blog/2021/12/09/new-aot-engine-brings-spring-native-to-the-next-level). É importante observar nos testes que os computadores utilizados estão descritos e detalhados, tendo em vista que o processamento e a memória são recursos bastante exigidos durante a compilação das aplicações.

Além disso, as análises a seguir trazem a comparação entre três diferentes aplicações: uma mais simples (*command line application*), uma utilizando MVC (*webmc-tomcat*) e a famosa *petclinic-jdbc*, que é bastante utilizada em diversos

exemplos do Spring. Essas três aplicações são comparadas sendo compiladas e executadas na tradicional JVM e no Spring Native 0.10 e 0.11.

4.3.1 Tempo de compilação

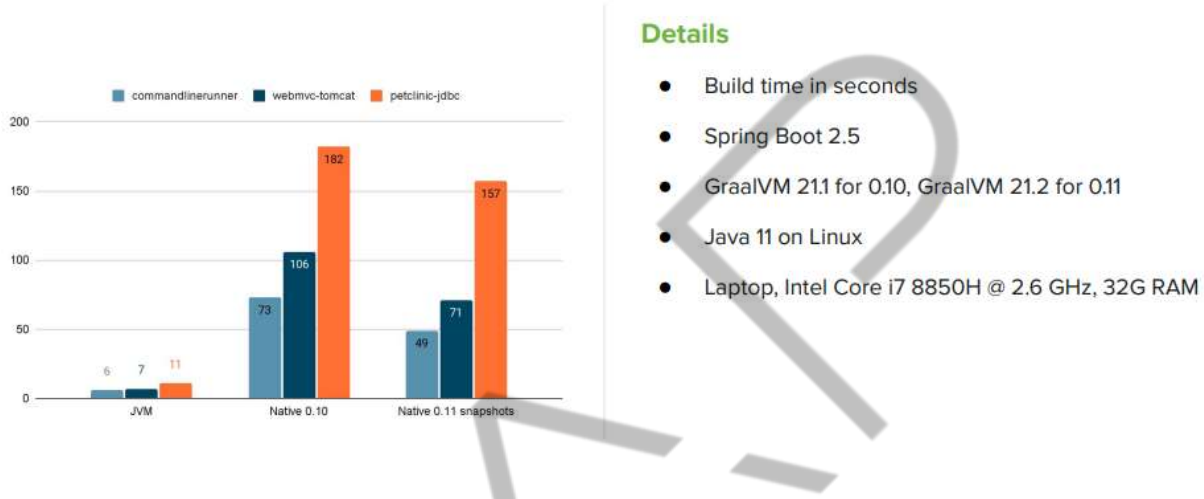

Figura 4.10– Comparativo de tempo de build
Fonte: Deleuze (2021b)

O primeiro ponto de comparação também é o grande ponto negativo do Spring Native: o tempo de compilação é muito superior se comparado à aplicação compilada para rodar na JVM. Tomando como base a aplicação mais complexa (petclinic-jdbc), que consequentemente tem mais classes para serem compiladas, o tempo de compilação chega a ser mais de 16x maior para a aplicação compilada utilizando Spring Native (0.10).

Um ponto interessante deste comparativo é que a máquina utilizada é bastante potente, com bom processador e memória. Em computadores com menos recursos pode-se esperar tempos relativamente maiores.

4.3.2 Tamanho do container

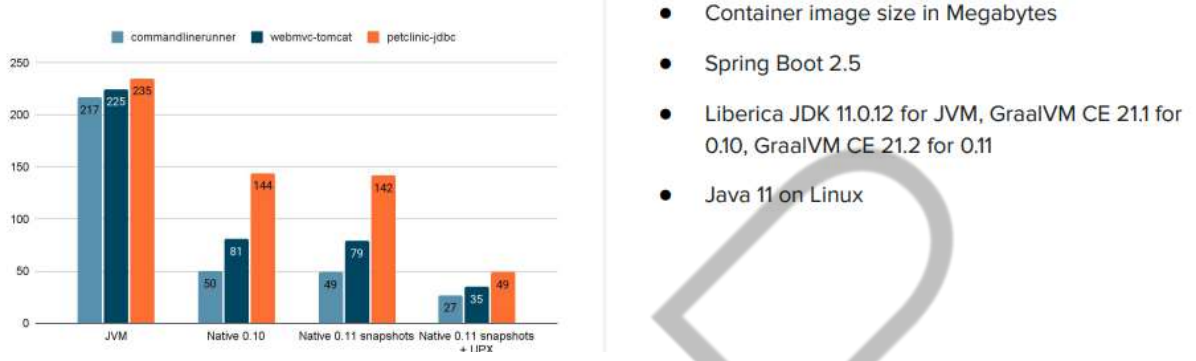

Figura 4.11 – Comparativo de tamanho da imagem Docker
Fonte: Deleuze (2021b)

A compilação Buildpack gera uma imagem Docker, conforme visto no exemplo prático. A diferença de tamanho da imagem Docker utilizando Spring Native é relevante para aperfeiçoar o tempo de *deploy* em clusters k8s, melhorando os tempos de *download*/*upload*.

4.3.3 Uso de memória

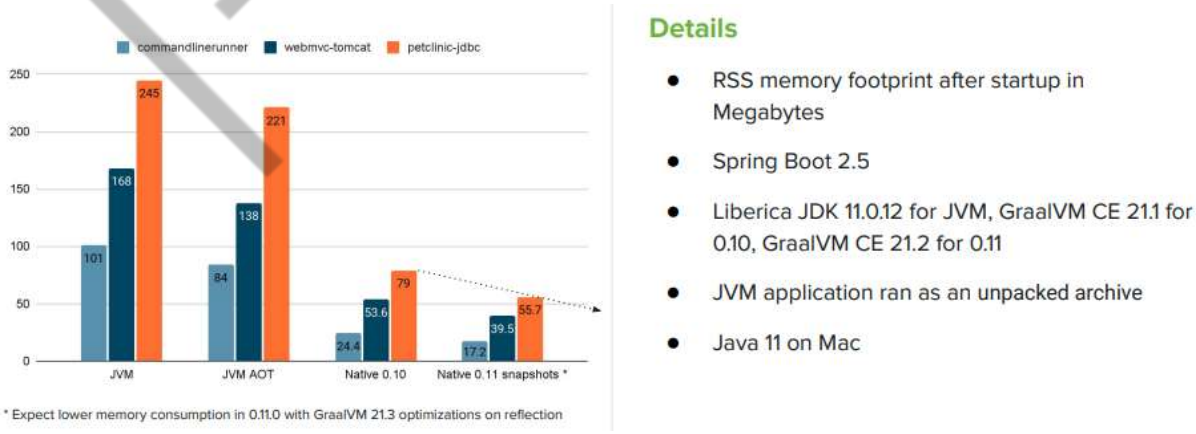

Figura 4.12 – Comparativo de uso de memória pela aplicação
Fonte: Deleuze (2021)

As aplicações Java têm fama de serem “pesadas” e um dos grandes motivos é o alto consumo de memória da JVM. Apesar de existirem algumas iniciativas, como a própria GraalVM ou a Open J9, além de diversos estudos sobre *tuning (*otimização*)* de JVM, a própria JVM é um limitador na otimização do uso de memória.

Como as aplicações Spring Native, seja em BuildPack ou NativeImage, não utilizam a JVM de forma tradicional, neste comparativo pode-se observar uma das maiores vantagens do Spring Native*:* aplicações que usam até 70% menos memória se comparadas a uma aplicação tradicional utilizando JVM. Esta característica do Spring Native é muito importante no mundo das aplicações Cloud que lidamos atualmente, uma vez que a redução financeira do valor da conta é facilmente perceptível.

4.3.4 Tempo de inicialização

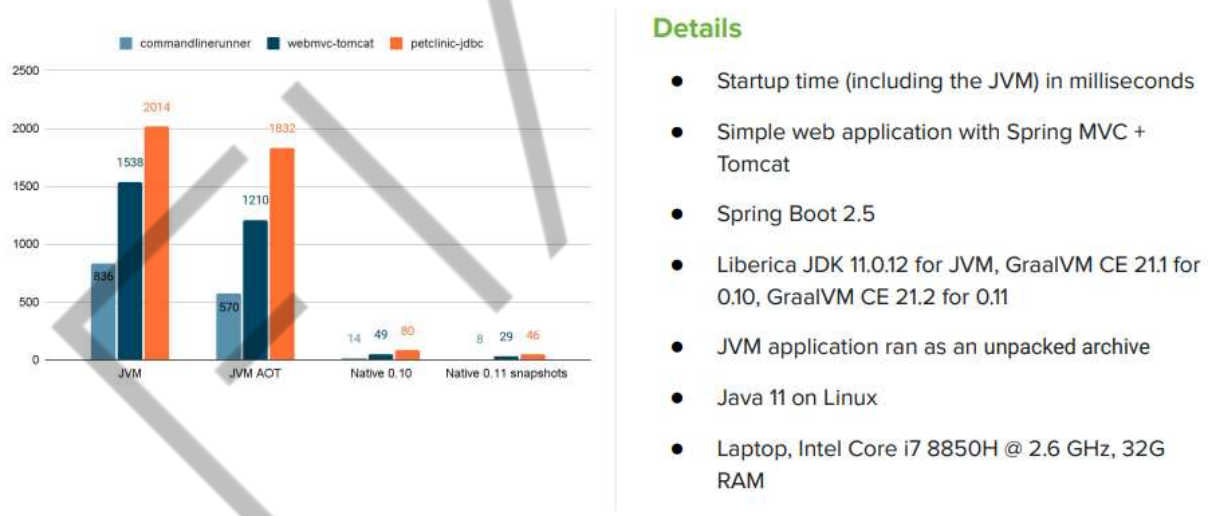

Figura 4.13 – Comparativo de tempo de inicialização
Fonte: Deleuze (2021b)

O tempo de inicialização é considerado a outra grande vantagem do Spring Native. Apesar de possuir um tempo de inicialização razoável, se comparado a outras linguagens e *frameworks*, ele ainda era considerado muito lento para funções *serverless*, como AWS Lambda ou GCP Functions devido ao chamado “*cold start*” ou tempo para executar a função pela primeira vez.

Se for para considerar o tempo de inicialização da aplicação mais complexa (petclinic-jdbc) utilizando a JVM (2014 milissegundos) com Native 0.11 (46 milissegundos), o tempo da JVM é mais de 46 vezes maior, tornando viável a utilização em aplicações *serverless*, além de melhorar o gerenciamento de alta disponibilidade e *rollbacks*, entre outras funcionalidades do Kubernetes.

4.4 Considerações finais

O Spring Native é uma tecnologia que atualiza o Spring Framework frente a concorrentes como Quarkus ou Micronaut. Em tempos de aplicações Cloud, é muito importante que as aplicações possuam otimização de utilização de recursos como uso de memória, posto que isso é traduzido diretamente em economia financeira. Além disso, o Spring Native resolve o problema do tempo de inicialização da JVM que quase inviabiliza a utilização do Spring Framework em aplicações *serverless*; tudo isso graças a compilação Ahead of Time, que, apesar do tempo elevado, permite essas características em aplicações Spring.

Na apresentação Spring One 2021 (https://springone.io/2021/sessions/spring-native), Sebástian Deleuze, responsável técnico do desenvolvimento do Spring Native, disse:

> “Estamos construindo o Framework para a próxima década”.

Esta frase tem um impacto expressivo, uma vez que o Spring é o maior *framework* Java, além de indicar que o Spring Native é encarado como o futuro das aplicações Spring. Esta importância, somada à quantidade de módulos do Spring, pode ser um dos motivos da demora para o lançamento da primeira versão estável. Ainda não se deve utilizar o Spring Native em aplicações em produção, mas é oportuno conhecer e estudá-lo, dado que ele é considerado por muitos especialistas o futuro das aplicações Java.

REFERÊNCIAS

BAELDUNG. **Spring Boot vs Quarkus**. 2022. Disponível em: <https://www.baeldung.com/spring-boot-vs-quarkus>. Acesso em: 04 ago. 2022.

DELEUZE, S. **Announcing Spring Native Beta!** 2021a. Disponível em: <https://spring.io/blog/2021/03/11/announcing-spring-native-beta>. Acesso em: 04 ago. 2022.

DELEUZE, S. **New AOT Engine Brings Spring Native to the Next Level**. 2021b. Disponível em: <https://spring.io/blog/2021/12/09/new-aot-engine-brings-spring-native-to-the-next-level>. Acesso em: 04 ago. 2022.

DELEUZE, S. **Spring Native**. 2021c. Disponível em: <https://springone.io/2021/sessions/spring-native>. Acesso em: 04 ago. 2022.

GITHUB. **graalvm /native-build-tools**. 2022. Disponível em: <https://github.com/graalvm/native-build-tools>. Acesso em> 04 ago. 2022.

KOZLOV, V. **JEP 295**: Ahead-of-Time Compilation. 2018. Disponível em: <https://openjdk.org/jeps/295>. Acesso em: 04 ago. 2022.

GRAALVM. **Introduction to GraalVM**. 2022. Disponível em: <https://www.graalvm.org/22.1/docs/introduction/>. Acesso em: 04 ago. 2022.

SPRING INITIALIZR. **Spring Initializr**. 2022. Disponível em: <https://start.spring.io>. Acesso em: 04 ago. 2022.