ESTUDO COMPARATIVO DOS FRAMEWORKS SPRING BOOT, MICRONAUT E QUARKUS PARA O DESENVOLVIMENTO DE MICROSSERVIÇOS EM JAVA *

Roniê Julian de Assis[†] Jorge Luis Boira Bavaresco[‡]

2022

Resumo

A arquitetura de microsserviços vem ganhando cada vez mais espaço no mercado e uma das linguagens mais utilizadas para seu desenvolvimento é a Java, o que acabou gerando uma grande quantidade de frameworks. Dentre os mais populares para a utilização em microsserviços estão o Spring Boot, o Micronaut e o Quarkus. O propósito deste trabalho foi comparar estes frameworks para saber se é possível, por meio de comparações usando determinadas métricas, escolher qual é a melhor opção para o desenvolvimento de microsserviços em Java, levando em conta fatores como desempenho, tempo de inicialização ou facilidade no desenvolvimento. Para realizar esta comparação foram feitos três microsserviços com cada framework e após o desenvolvimento de todos os microsserviços, foram feitos testes de inicialização, do código gerado e de desempenho, mostrando que cada um dos frameworks se destacou em um destes pontos, com isso cabe ao desenvolvedor escolher a melhor opção dependendo da sua principal necessidade para a aplicação em microsserviços há ser desenvolvida.

Palavras-chaves: Java. Microsserviços. Spring Boot. Quarkus. Micronaut.

1 Introdução

A arquitetura de microsserviços consiste na divisão da aplicação em pequenos serviços independentes e focados em um componente da aplicação que se comunicam entre si. Com a evolução do desenvolvimento de softwares, essa arquitetura vem ganhando cada vez mais espaço no mercado segundo Fowler (2019).

*Trabalho de Conclusão de Curso (TCC) apresentado ao Curso de Bacharelado em Ciência da Com-putação do Instituto Federal de Educação, Ciência e Tecnologia Sul-rio-grandense, Campus Passo Fundo, como requisito parcial para a obtenção do título de Bacharel em Ciência da Computação, na cidade de Passo Fundo, em 2022.

[†]Acadêmico do curso de Ciência da Computação no instituto Federal Sul-rio-grandense - Câmpus Passo Fundo

[‡]Orientador do trabalho (Professor do curso de Ciência da Computação no instituto Federal Sul-rio-grandense - Câmpus Passo Fundo).

Uma das linguagens mais utilizadas para esta arquitetura é a Java, que acabou se tornando muito popular por ser orientada a objetos, pela sua independência de plataforma, pela estabilidade previsível, e por ser considerada bastante modular, o que acabou gerando uma grande quantidade de frameworks. Dentre os mais populares para a utilização em microsserviços estão o Spring Boot sendo este com 311 mil repositórios no github[1], o Quarkus com 11 mil repositórios e o Micronaut com cerca de 5 mil repositórios na mesma plataforma. Por conta disso, foi feito uma análise nessa pesquisa para saber se é possível, por meio de comparações usando determinadas métricas, escolher qual é a melhor opção para o desenvolvimento de microsserviços em Java.

Como resultado, espera-se obter subsídio para escolher o framework ideal de acordo com as necessidades do desenvolvimento, dentre elas a fácil implementação, a rápida resposta ou desempenho na inicialização.

O presente artigo está organizado da seguinte forma: : A seção 2 apresenta as tecnologias utilizadas no desenvolvimento do trabalho. Na seção 3 será apresentado como o trabalho foi desenvolvido e os testes realizados. A seção 4 apresenta os resultados obtidos a partir da avaliação dos frameworks. Por fim a última seção conterá as conclusões e considerações finais.

2 Referencial Teórico

2.1 Microsserviços

Trata-se de uma arquitetura de software que nasceu da necessidade de resolver problemas que a arquitetura monolítica continha, dentre eles, a falta de escalabilidade, a falta de eficiência, a lenta velocidade de desenvolvimento e a dificuldade de adotar novas tecnologias.

Segundo Fowler (2019), adotar a arquitetura de microsserviços para o desenvolvimento, seja a partir do zero ou dividindo uma aplicação monolítica, já existente em microsserviços independentes, pode resolver os problemas citados anteriormente. Com a arquitetura de microsserviços, uma aplicação pode ser facilmente escalada tanto horizontalmente quanto verticalmente, a produtividade e a velocidade de desenvolvimento aumentam drasticamente e tecnologias podem ser facilmente trocadas de um microsserviço a outro.

O autor também afirma que há desafios, já que para um ecossistema com esta arquitetura seja bem-sucedido, precisa de uma infraestrutura estável e sofisticada, além de necessitar de uma boa organização da empresa que está adotando esta técnica, e também de conhecimento e boa comunicação entre as equipes de desenvolvimento.

Concordando, Benevides e Posta (2019) conceituam microsserviços como "uma abordagem para construir sistemas de software que decompõe modelos de domínio de negócios em contextos menores, consistentes e limitados implementados por serviços".

Benevides e Posta (2019) trazem a ideia de dividir o time de desenvolvimento em equipes pequenas, sendo que cada uma destas equipes será responsável por um microsserviço e terá

[1] Disponível em https://github.com/search?q=Spring+boot, https://github.com/search?q=Quarkus e https://github.com/search?q=Micronaut

autonomia para fazer alterações sem causar grandes impactos ao sistema como um todo.

Por fim, os autores destacam que a utilização de microsserviços pode trazer alguns desafios, como o consumo de mais recursos, pode gerar duplicações de funcionalidades, traz uma complexidade operacional maior e é mais difícil realizar uma depuração de código, o que acaba exigindo conhecimento da equipe para se trabalhar com este tipo de arquitetura.

2.2 Frameworks

Um dos principais objetivos da engenharia de software é a otimização do desenvolvimento, sendo que o principal modo de conseguir isso é a reutilização de códigos aumentando a qualidade e reduzindo o esforço de desenvolvimento como é dito por Junior (2006).

Uma das formas de conseguir reutilizar códigos é com a tecnologia de frameworks que vem ganhando destaque nos últimos anos. Existem várias definições de frameworks na literatura, e entre elas, tem-se que um framework é um conjunto de classes que constitui um projeto abstrato para a solução de uma família de problemas (JOHNSON; FOOTE, 1988 apud JUNIOR, 2006). Já Lundberg e Mattsson (1996 apud JUNIOR, 2006), descrevem um framework como uma arquitetura desenvolvida com o objetivo de atingir a máxima reutilização, representada como um conjunto de classes abstratas e concretas, com grande potencial de especialização. Para Buschmann et al. (1996 apud JUNIOR, 2006) um framework é definido como um software parcialmente completo projetado para ser instanciado. O framework define uma arquitetura para uma família de subsistemas e oferece os construtores básicos para criá-los. Também são explicitados os lugares ou pontos de extensão (hot-spots) nos quais adaptações do código para um funcionamento específico de certos módulos devem ser feitas.

Como é possível de se observar, apesar das diferentes definições, nenhuma delas se contradiz, de forma que é possível afirmar que a funcionalidade de um framework é a reutilização de projeto e código com o objetivo de tornar mais sólido e facilitar o desenvolvimento de sistemas.

2.3 Spring Framework

O Spring Framework, segundo Johnson et al. (2007), é uma das principais estruturas de desenvolvimento de código-fonte Java para um desenvolvimento de aplicativos mais produtivo. Ele tem como objetivo facilitar o desenvolvimento de aplicações Web tendo como principais funcionalidades o Spring MVC, o suporte para Java Database Connectivity (JDBC) e Java Persistence API (JPA), a inversão de controle e a injeção de dependências.

O Spring MVC é um framework para criação de aplicações web e serviços RESTful. Ele é bastante conveniente, pois, muitas das aplicações de hoje em dia precisam atender requisições web (JUNIOR; AFONSO, 2017).

O JDBC e o JPA são formas de interação com o banco de dados e o Spring fornece suporte para sua utilização já que a persistência de dados também é um recurso muito utilizado nas aplicações e é difícil conceber uma aplicação hoje que não grave ou consulte algum banco de dados (JUNIOR; AFONSO, 2017).

A inversão de controle, segundo Junior e Afonso (2017), acontece quando uma classe responsável por métodos necessários têm sua forma de instanciar invertida, ou seja, ela não é determinada diretamente pelo programador. Nesse caso, o controle é delegado a uma infraestrutura de software muitas vezes chamada de container ou a qualquer outro componente que possa tomar controle sobre a execução.

A injeção de dependências é um tipo de inversão de controle que dá nome ao processo de prover instâncias de classes que um objeto precisa para funcionar e com isso é possível programar voltado para interfaces e manter o baixo acoplamento entre as classes de um mesmo projeto.

Dentre essas e outras funcionalidades disponibilizadas por este framework é possível agilizar e simplificar o processo de desenvolvimento de aplicações.

2.3.1 Spring Boot

Por mais que o Spring simplifica os componentes para o desenvolvimento Web, sua configuração é demasiada complexa e por isso o desenvolvedor perdia o principal foco da ferramenta que são as regras de negócio e a rápida entrega do software. Segundo Boaglio (2017), o Spring Boot pode ser considerado uma maneira eficiente e eficaz de criar uma aplicação em Spring e facilmente colocá-la no ar, funcionando sem depender de um servidor de aplicação. Sendo que sua configuração é gerada automaticamente, ele simplesmente analisa o projeto e o configura, sendo possível realizar alterações, porém ele segue o padrão que a maioria das aplicações precisas, então, muitas vezes não é preciso fazer nada.

Por fim, Junior e Afonso (2017) descrevem que o Spring Boot facilita na importação de dependências através dos Starters que são dependências que agrupam outras, assim, basta adicionar uma única entrada no pom.xml, que todas as dependências necessárias serão adicionadas ao classpath.

Ainda é possível utilizar a plataforma Spring Initializr para criação de projetos, onde esse processo é simplificado, e combinado a outras ferramentas para simplificação do framework Spring, é possível encapsular muitas configurações e manter o foco no desenvolvimento.

2.4 Quarkus

O Quarkus é um framework Java nativo em Kubernetes[2] desenvolvido pela RedHat que tem como base a filosofia de priorização de containers, por isso, ele é otimizado para uso reduzido da memória e tempos de inicialização mais rápidos.

Para Clingan e Finnigan (2022) ele foi projetado para ser tão produtivo quanto o Node.js para os desenvolvedores e consumir tão poucos recursos quanto o Golang. O Quarkus parece novo e familiar ao mesmo tempo por incluir muitos recursos novos e impactantes, oferecendo suporte às APIs com as quais os desenvolvedores já estão familiarizados.

Os autores comentam também que embora o Quarkus seja usado em uma ampla variedade de ambientes de implantação, ele tem aprimoramentos e otimizações específicas para contêineres Linux e Kubernetes. Por esse motivo, o Quarkus é conhecido como “Kubernetes Native Java”.

[2] Kubernetes é um orquestrador de contêineres open-source que automatiza a sua utilização.

Koleoso (2020) descreve o Quarkus como um framework construído para o desenvolvimento de softwares modernos, se destacando com implementações em nuvem, seja como containers, um servidor autônomo ou como uma estrutura serverless, oferecendo quase tudo que outros frameworks famosos disponibilizam com outros benefícios, podendo executá-lo no local, na nuvem e em qualquer lugar entre eles.

2.5 Micronaut

O Micronaut é um framework baseado em JVM que é especializado para a criação de microsserviços. Ele foi inspirado pelo Spring e o Grails[3], permitindo criar aplicações em Java, Kotlin ou Groovy. Segundo Jeleń e Dzieńkowski (2021), ele, assim como o Spring Boot, enfatiza a implementação rápida de aplicativos na arquitetura de microsserviços. No entanto, ao contrário do Spring Boot, os criadores do Micronaut prestaram uma atenção especial no baixo consumo de memória.

Jeleń e Dzieńkowski (2021) ainda citam algumas vantagens no framework que são: o suporte nativo a plataforma GraalVM[4], que tem impacto na obtenção de alto desempenho e disponibilidade das aplicações desenvolvidas; possuir um mecanismo de contêiner de aplicativos assim como o Spring Boot, mas seu gerenciamento é implementado de maneira diferente; a utilização de um processador de anotação, que permite uma inicialização mais rápida do aplicativo e detecção de erros na fase de compilação, em vez de quando o aplicativo é iniciado, como é o caso do Spring Boot.

Para Błaszczyk, Pucek e Kopniak (2021), o Micronaut é muito semelhante ao Spring Boot em relação a sua sintaxe como convenções de nomenclatura semelhantes ou mecanismos de anotação praticamente idênticos. Porém, ele se difere no mecanismo de injeção de dependências, já que no Micronaut isso ocorre em tempo de compilação, o que segundo eles gera um tempo de inicialização menor.

3 Desenvolvimento

Neste capítulo será descrito como o trabalho foi desenvolvido, passando pela modelagem da aplicação seguindo os diagramas de componentes e de classe e em um segundo momento, a descrição de como foi realizada a comparação entre as aplicações desenvolvidas.

3.1 Modelagem da aplicação

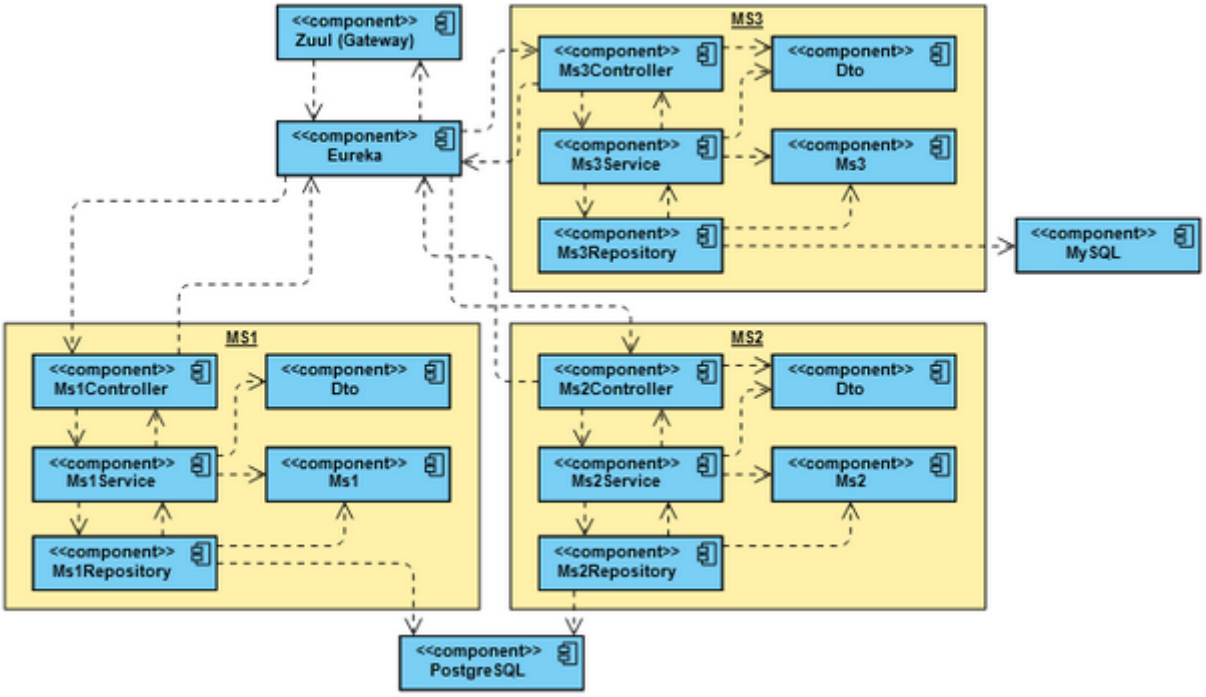
Para essa pesquisa, foi desenvolvida uma aplicação simples com a arquitetura de microsserviços com os três frameworks escolhidos para a comparação.

Essa aplicação consiste em três microsserviços, sendo dois deles conectados entre si e se comunicando com um banco de dados em Postgresql e outro microsserviço isolado conectado a um banco de dados MySQL. Todos os microsserviços estão conectados a um serviço de Discovery

[3] Grails é um framework focado na criação de aplicações Web utilizando a linguagem Groovy.
[4] GraalVM é uma máquina virtual com o objetivo de compilar rapidamente e diminuir o consumo de memória da aplicação.

desenvolvido com a biblioteca Eureka que está por fim conectado a um serviço de gateway feito com a biblioteca Zuul, como é possível ver na Figura 1.

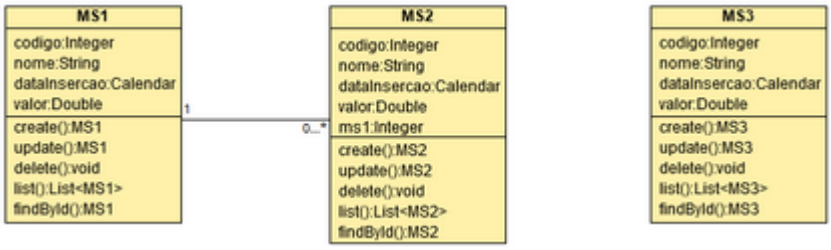
Figura 1 – Diagrama de componentes



Fonte: Do autor, 2022

A estrutura dos microsserviços são semelhantes, sendo composta por nome, data de inserção e valor, sendo que o Microsserviço "MS2" contem uma referência 1 para n com o modelo do "MS1", seguindo o diagrama de classes da Figura 2.

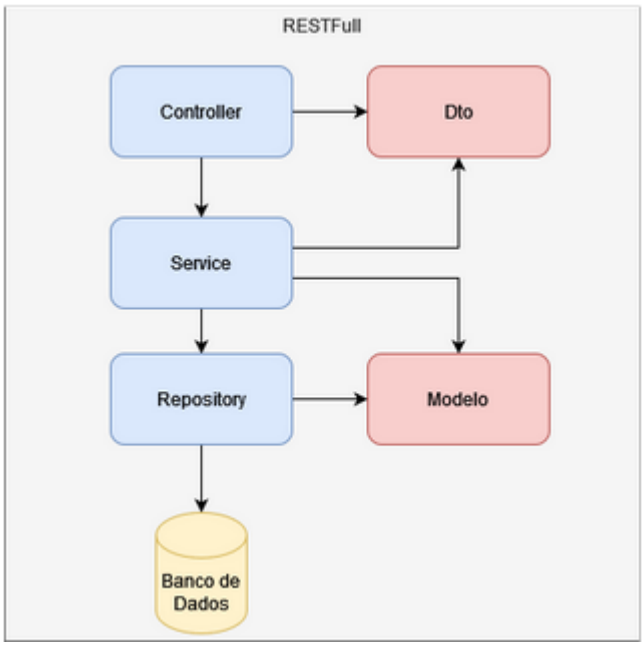
Figura 2 – Diagrama de classes



Fonte: Do autor, 2022

Os serviços seguiram a arquitetura Rest, e são compostos de um *Controller*, *Service* e um *Repository*, como demonstrado na Figura 3, exceto no caso do Quarkus onde foi utilizado a classe Resource, que incorporou o Controller e o Service, Já a classe Repository, não foi necessário graças a extensão nos modelos da classe PanacheEntity.

Figura 3 – Arquitetura dos componentes



Fonte: Do autor, 2022

3.2 Comparação entre aplicações

Com o objetivo de comparar, todos os serviços foram desenvolvidos e analisados utilizando a versão 11 do Java com os três frameworks utilizando a IDE Visual Studio por conta de sua alta capacidade de personalização com plugins, compatibilidade com os frameworks e familiaridade com a utilização, sendo executado em uma máquina com 8 Gigabytes DDR3 de Memória RAM e um processador Core I5-5200U com sistema operacional Windows 10.

Em seguida, serão listados os parâmetros e os tipos de análise que foram realizados:

- **Análise do código necessário para cada aplicação:** nesta comparação, foi necessária a utilização da ferramenta JArchitect, na qual, importando o projeto é possível obter o número total de linhas válidas, métodos, pacotes e dependências, tendo também uma média da complexidade dos métodos, dados que foram analisados e comparados entre todas as aplicações desenvolvidas.
- **Análise de tempo de inicialização:** para esta comparação, as aplicações foram inicializadas e posteriormente reinicializadas separadamente fazendo uma média aritmética, levando em conta o tempo disponibilizado no terminal de resposta da inicialização por todos os frameworks que foram testados.
- **Análise do tempo de resposta de requisições:** para este item, foram avaliados dois cenários, primeiramente requisições de forma única com a ferramenta Postman onde foi coletado o tempo necessário para obter a resposta, e, posteriormente, de forma simultânea, sendo utilizada a ferramenta Jmeter, e em todas as requisições será pego o tempo medio de

resposta, desvio padrão, porcentagem de erros e vazão, sendo feitos 6 testes com diferentes cargas.

Após obter todos os resultados das análises feitas, os resultados foram analisados e comparados, sendo disponibilizados em gráficos para demonstrar as melhores opções levando em consideração cada ponto avaliado.

4 Resultados

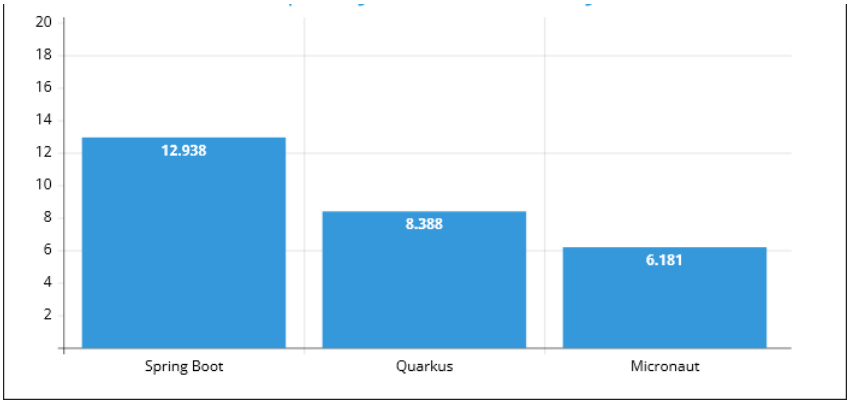
Nesta seção são apresentados os resultados obtidos após o desenvolvimento dos nove microsserviços, sendo três deles com o Spring boot, três com o Micronaut e três com Quarkus. Estes resultados foram obtidos pelos testes de inicialização, do código necessário para a criação dos serviços e do desempenho de todas as aplicações.

Sendo que todos os testes foram feitos utilizando o mesmo hardware que foi uma máquina com 8 Gigabytes DDR3 de Memória RAM e um processador Core I5-5200U.

4.1 Testes de inicialização

Para o ambiente dos testes, foi inicializado o serviço de discovery, o serviço de gateway, dois serviços do mesmo framework e o terceiro sendo inicializado e reinicializado dez vezes, levando em conta até o tempo de registro no serviço de discovery, que corresponde quando finalmente o serviço pode ser utilizado. Por fim, foi feita uma média aritmética com os resultados para cada framework, levando em conta cada um dos serviço testados para eles, sendo os resultados demonstrados no gráfico da Figura 4.

Figura 4 – Comparação de inicialização



Fonte: Do autor, 2022

Dentre os três frameworks, o Spring boot obteve o pior resultado, com um tempo médio de 12.938 milissegundos. Isso ocorreu devido ao Spring boot fazer mais processamentos em tempo de execução, o que lhe torna mais lento nesse quesito.

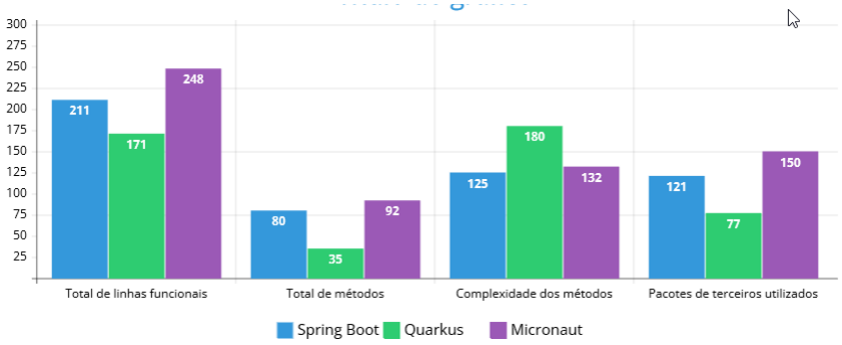
Em segundo lugar, o framework Quarkus apresentou um tempo médio de 8.388 milissegundos. Mesmo com o tempo de inicialização rápida como o Micronaut, seu tempo total foi mais alto, isso aconteceu devido ao tempo alto para o registro no serviço de discovery.

Por fim, o Micronaut obteve o melhor resultado, sendo seu tempo médio de 6.181 milissegundos, demonstrando ser a melhor opção levando em conta o parâmetro de inicialização.

4.2 Testes do código necessário para a criação dos serviços

O segundo teste é o do código necessário para a criação dos serviços, e para isso foi utilizada a ferramenta JArchitect, onde os microsserviços gerados para cada framework foram analisados em conjunto, sendo possível obter alguns dados, como o total de linhas funcionais, quantidade e complexidade média de métodos e pacotes de terceiros utilizados, os resultados obtidos foram organizados o gráfico apresentado na Figura 5.

Figura 5 – Comparação do código gerado



Fonte: Do autor, 2022

Nesta análise foi possível perceber uma grande semelhança no desenvolvimento com o Micronaut e o Spring boot, tendo uma estrutura com um controlador que contém a lógica de comunicação com endpoints, um serviço onde contem as regras de negócio, classes de modelo e dtos[5], a classe de repository para gerenciar os dados do banco de dados, uma classe que irá servir como cliente para comunicar um microsserviço com outro quando há necessidade. Nota-se apenas algumas diferenças em nomes de anotações e configurações. Por este motivo o Micronaut obteve números maiores, onde em linhas de códigos teve um total de 248, sendo que o Spring Boot atingiu um valor de 211 neste mesmo parâmetro. Observando as métricas de métodos e pacotes de terceiros, novamente o Micronaut obteve valores próximos, porém, mais altos que o Spring Boot, sendo um total de 92 métodos, com uma pontuação de complexidade média de 1.32 e 150 pacotes foram utilizados. Já o Spring Boot conseguiu uma complexidade de 1.25 pontos para 80 métodos e 121 pacotes de terceiros utilizados.

O framework Quarkus, em seu desenvolvimento segue uma estrutura inspirada no desenvolvimento com Jakarta EE, utilizando diversos componentes do mesmo, onde tem uma classe

[5] Data Transfer Object (DTO) é um padrão de projetos usado em Java para o transporte de dados entre diferentes componentes de um sistema, diferentes instâncias ou processos de um sistema distribuído ou diferentes sistemas via serialização.

resource que irá gerenciar tanto a comunicação quanto às regras de negócio do serviço, seguindo das classes modelo e Dto, sendo que diferentemente dos outros frameworks, as classes modelo estendem a classe “PanacheEntityBase” que irá gerenciar os dados referentes a classe modelo no banco, excluindo a necessidade de um repository, tendo também por fim uma classe que servirá como cliente para outro microsserviço quando necessário. Nos dados obtidos pelo JArchitect, é possível observar um número menor de linhas de código, sendo o resultado de 171 linhas, e obtendo uma pontuação média de complexidade de 1.8 para 35 métodos, por fim, foi utilizado um total de 77 pacotes externos.

Como resultado é possível ver que o Quarkus gerou um código mais compacto, porém com uma maior complexidade. Um ponto de destaque que não é possível obter pelas métricas analisadas é a quantidade de conteúdo na internet disponível para os frameworks, neste ponto, o Spring Boot, por estar mais tempo consolidado, tem um número bem maior de informações e conteúdo disponíveis.

4.3 Testes de desempenho

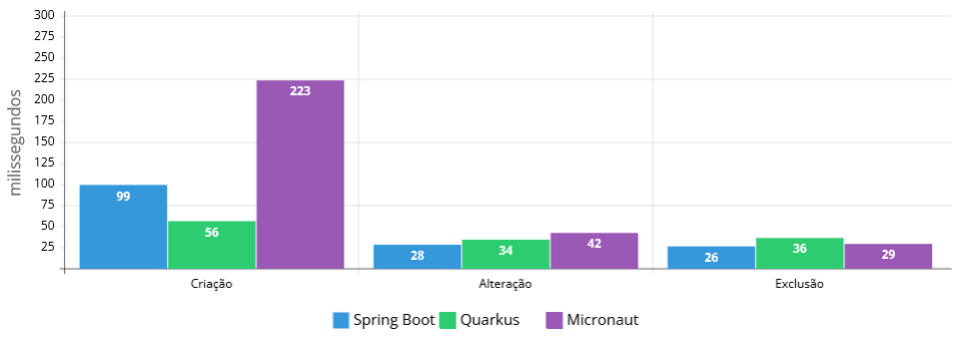
A última análise, na qual foi verificado o desempenho dos serviços, foi dividida em dois ambientes, sendo um de levando em conta requisições de forma isolada e outro avaliando o desempenho com diversas requisições simultaneamente.

4.3.1 Requisições isoladas

No primeiro ambiente, as requisições foram feitas de forma isoladas, com 10 mil registros para cada serviço disponível, essas requisições foram feitas utilizando a ferramenta Postman e foi colhido o tempo até o retorno das requisições 5 vezes, sendo feita uma média aritmética dos valores obtidos.

Dentre as operações onde se manipulava apenas um dado, sendo essas operações de cadastro, alteração e exclusão de dados, todos os frameworks obtiveram valores muito parecidos, sendo eles:

Figura 6 – Comparação do tempo de requisições de criação, alteração e exclusão



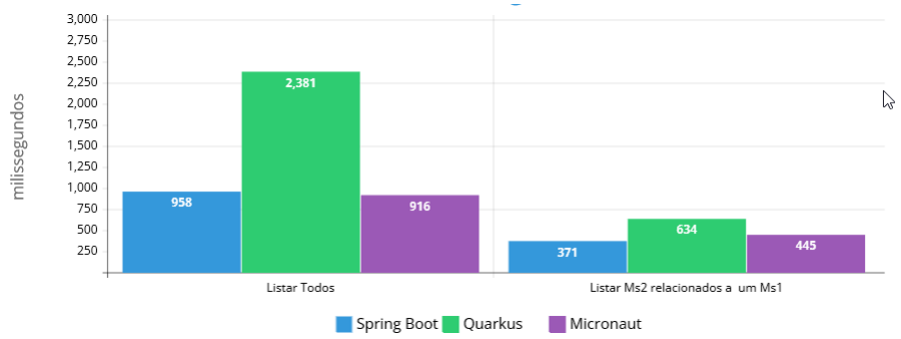
Fonte: Do autor, 2022

Com os resultados é possível considerar um desempenho semelhante entre os três fra-

meworks, levando em consideração apenas que a primeira requisição com o Micronaut, a qual é responsável por abrir a conexão com o banco de dados é mais lenta do que os demais frameworks.

Já em requisições onde havia um grande tráfego de informações como operações de listagem de todos os registros e registros vinculados a um objeto, os resultados foram mais dispersos, como é possível ver na Figura 7:

Figura 7 – Comparação do tempo de requisições do tipo GET com grande tráfego de dados



Fonte: Do autor, 2022

É possível observar que ao aumentar a quantidade de dados que estão sendo buscados, o framework Quarkus obteve uma resposta pior gradativamente, o que impactou nos testes com requisições simultâneas do mesmo.

4.3.2 Requisições simultâneas

Para as requisições simultâneas foram feitos seis testes para cada framework, onde os usuários iriam fazer seis requisições em sequência, sendo elas: listar todos os registros do primeiro microsserviço, listar os registros do segundo microsserviço relacionados a um registro do primeiro microsserviço, listar todos os registros do segundo microsserviço, buscar um registro do segundo microsserviço, listar todos os registros do terceiro microsserviço e, por fim, buscar um registro do terceiro microsserviço.

No primeiro teste foi observado o desempenho utilizando 50 usuários executando a sequência de requisições três vezes com 1.000 registros para cada serviço, gerando um total de 900 requisições realizadas e obtendo os seguintes resultados:

Tabela 1 – 1º Teste de desempenho com requests múltiplos

Framework	Média de tempo por requisição	Desvio padrão	Porcentagem de Erro	Vazão
Spring Boot	1.376 milissegundos	1.623,23	0,00%	33,31606
Micronaut	9.177 milissegundos	20.911,04	27,778%	2,81667
Quarkus	3.574 milissegundos	8.110,52	4,444%	12,15822

Fonte: Fonte: Do autor, 2022

O segundo teste contou com 100 usuários executando a sequência de requisições três

vezes com 1.000 registros para cada serviço, gerando um total de 1.800 requisições realizadas e obtendo os seguintes resultados:

Tabela 2 – 2º Teste de desempenho com requests múltiplos

Framework	Média de tempo por requisição	Desvio padrão	Porcentagem de Erro	Vazão
Spring Boot	1.729 milissegundos	2.743,79	0,00%	49,03029
Micronaut	19.578 milissegundos	31.768,39	17,056%	4,07224
Quarkus	4.586 milissegundos	4.330,08	0,00%	19,10787

Fonte: Fonte: Do autor, 2022

O terceiro teste contou com 300 usuários executando a sequência de requisições três vezes com 1.000 registros para cada serviço, gerando um total de 5.400 requisições realizadas e obtendo os seguintes resultados:

Tabela 3 – 3º Teste de desempenho com requests múltiplos

Framework	Média de tempo por requisição	Desvio padrão	Porcentagem de Erro	Vazão
Spring Boot	3.284 milissegundos	4.499,91	24,296%	72,33661
Micronaut	5.516 milissegundos	16.422,64	46,796%	14,86301
Quarkus	10.768 milissegundos	14.014,85	26,963%	22,0821

Fonte: Fonte: Do autor, 2022

O quarto teste contou com 50 usuários executando a sequência de requisições seis vezes com 10.000 registros para cada serviço, gerando um total de 1.800 requisições realizadas e obtendo os seguintes resultados:

Tabela 4 – 4º Teste de desempenho com requests múltiplos

Framework	Média de tempo por requisição	Desvio padrão	Porcentagem de Erro	Vazão
Spring Boot	7.012 milissegundos	6.168,84	0,111%	6,7647
Micronaut	12.470 milissegundos	18.979,14	18,222%	3,54053
Quarkus	23.599 milissegundos	20.156,58	2,111%	2,09467

Fonte: Fonte: Do autor, 2022

O quinto teste contou com 100 usuários executando a sequência de requisições seis vezes com 10.000 registros para cada serviço, gerando um total de 3.600 requisições realizadas e obtendo os seguintes resultados:

Tabela 5 – 5º Teste de desempenho com requests múltiplos

Framework	Média de tempo por requisição	Desvio padrão	Porcentagem de Erro	Vazão
Spring Boot	13.256 milissegundos	15.683,66	1,639%	7,03394
Micronaut	23.732 milissegundos	35.601,44	17,944%	3,82342
Quarkus	18.681 milissegundos	27.879,82	81,583%	4,09973

Fonte: Fonte: Do autor, 2022

O último teste contou com 300 usuários executando a sequência de requisições seis vezes com 10.000 registros para cada serviço, gerando um total de 10.800 requisições realizadas e obtendo os seguintes resultados:

Tabela 6 – 6º Teste de desempenho com requests múltiplos

Framework	Média de tempo por requisição	Desvio padrão	Porcentagem de Erro	Vazão
Spring Boot	29.686 milissegundos	26.583,02	25,546%	9,16161
Micronaut	32.390 milissegundos	36.754,18	46,185%	8,24161
Quarkus	19.570 milissegundos	36.996,69	93,083%	7,52941

Fonte: Fonte: Do autor, 2022

Após analisar os dados resultantes dos testes, é possível notar uma taxa alta de erros no framework Micronaut e nos casos mais extremos com o Quarkus. Isso ocorreu devido a uma instabilidade na classe que faz a comunicação entre serviços do Micronaut e o tempo muito elevado nas respostas do Quarkus, excedendo o tempo limite de resposta do serviço de gateway. Também é possível notar que o Spring Boot obteve um desempenho bem melhor e mais estável que seus concorrentes, podendo ser considerado a melhor opção em questões de desempenho, principalmente quando há muitos dados envolvidos em requisições.

5 Considerações finais

Este trabalho teve como objetivo avaliar e comparar o desempenho e a facilidade de desenvolvimento de microsserviços em Java usando os frameworks Spring Boot, Quarkus e Micronaut para o desenvolvimento de microsserviços em Java. Para isso foi desenvolvida uma aplicação contendo três serviços para cada framework, e posteriormente, foram feitos testes de inicialização, dos códigos gerados e o desempenho das aplicações ao receber requisições.

Seguindo a metodologia proposta para o desenvolvimento, foi possível notar que não houve diferenças significativas no desenvolvimento com Spring Boot e MIcronaut, onde suas aplicações ficaram semelhantes em diversos aspectos, já com o Quarkus foi unificado o *Controller* e o *Services* em uma classe só, assim como a exclusão de uma interface *Repository* por conta do PanacheEntity, o que gerou um código mais compacto, demonstrado ser de forma técnica a melhor opção neste quesito.

Graças ao fato de tanto o Micronaut quanto o Quarkus terem como um dos seus objetivos reduzir o tempo de inicialização em comparação ao Spring Boot, eles obtiveram um resultado muito superior neste quesito, sendo o Micronaut com o melhor resposta, graças a sua compatibilidade com o serviço de discovery escolhido, desta forma sendo possível registrar os serviços mais rapidamente.

Por fim o último teste, no qual foi avaliado o desempenho das aplicações quando há requisições, o Quarkus já nas requisições de forma única apresentou problemas para tratar grande quantidade de dados. Por este motivo seu desempenho nos testes feitos de forma simultânea tiveram um resultado aquém dos demais, por mais que com um número baixo de falhas, o que não foi o caso do Micronaut, que apresentou problemas, ao tratar de requisições simultâneas

que interagiam com o client responsável pela comunicação do serviço Ms1 com o serviço Ms2, se mostrando ser instável nestes casos. Já o Spring Boot se mostrou bem mais eficiente, obtendo respostas mais rápidas e gerando poucas falhas em relação aos demais frameworks testados, sendo considerado a melhor opção para aplicações que tenham um grande número de interações.

Analisando os três testes feitos foi possível perceber que cada um dos frameworks se destacou em um dos pontos avaliados, sendo o Micronaut no quesito de inicialização rápida, o Quarkus com um código mais compacto e o Spring Boot com melhor desempenho. Com isso a escolha da melhor opção cabe ao desenvolvedor dependendo da sua principal necessidade para a aplicação em microsserviços a ser desenvolvida. Porém, considerando que a arquitetura de microsserviços é muito utilizada e está em constante evolução, é muito provável que estes frameworks apresentem soluções e melhorias em seus pontos de menor destaque em atualizações futuras.

COMPARATIVE STUDY OF SPRING BOOT, MICRONAUT AND QUARKUS FRAMEWORKS FOR THE DEVELOPMENT OF MICROSSERVICES IN JAVA

Roniê Julian de Assis[‖] Jorge Luis Boira Bavaresco[**]

2022

Abstract

The microservices architecture has been gaining more and more space in the market and one of the most used languages for its development is Java, which ended up generating a large number of frameworks. Among the most popular for use in microservices are Spring Boot, Micronaut and Quarkus. The purpose of this work was to compare these frameworks to know if it is possible, through comparisons using certain metrics, to choose the best option for the development of microservices in Java, taking into account factors such as performance, startup time or ease of development. To make this comparison, three microservices were made with each framework and after the development of all microservices, initialization, generated code and performance tests were carried out, showing that each of the frameworks stood out in one of these points, so it is up to the developer choose the best option depending on their main need for the microservices application to be developed.

Key-words: Java. Microservices. Spring Boot. Quarkus. Micronaut.

Referências

BENEVIDES, R.; POSTA, C. *Microservices for Java Developers: A Hands-on Introduction to Frameworks and Containers*. O'Reilly Media, 2019. Disponível em: <https://books.google.com.br/books?id=leYlyAEACAAJ>. Citado na página 2.

BŁASZCZYK, M.; PUCEK, M.; KOPNIAK, P. Comparison of lightweight frameworks for java by analyzing proprietary web applications. *Journal of Computer Sciences Institute*, v. 19, p. 159–164, 2021. Citado na página 5.

[‖]Acadêmico do curso de Ciência da Computação no instituto Federal Sul-rio-grandense - Câmpus Passo Fundo

[**]Orientador do trabalho (Professor do curso de Ciência da Computação no instituto Federal Sul-rio-grandense - Câmpus Passo Fundo).

BOAGLIO, F. *Spring Boot: Acelere o desenvolvimento de microsserviços*. [S.l.]: Casa do Código, 2017. Citado na página 4.

BUSCHMANN, F. et al. *Pattern-Oriented Software Architecture: A System of Patterns*. [S.l.]: John wiley & sons, 1996. v. 1. Citado na página 3.

CLINGAN, J.; FINNIGAN, K. *Kubernetes Native Microservices with Quarkus and MicroProfile*. Manning, 2022. ISBN 9781617298653. Disponível em: <https://books.google.com.br/books?id=eatczgEACAAJ>. Citado na página 4.

FOWLER, S. *Microsserviços prontos para a produção: Construindo sistemas padronizados em uma organização de engenharia de software*. Novatec Editora, 2019. ISBN 9788575227473. Disponível em: <https://books.google.com.br/books?id=pN6RDwAAQBAJ>. Citado 2 vezes nas páginas 1 e 2.

JELEŃ, M.; DZIEŃKOWSKI, M. The comparative analysis of java frameworks: Spring boot, micronaut and quarkus. *Journal of Computer Sciences Institute*, v. 21, p. 287–294, 2021. Citado na página 5.

JOHNSON, R. et al. *Professional Java Development with the Spring Framework*. Wiley, 2007. ISBN 9780471748946. Disponível em: <https://books.google.com.br/books?id=oMVIzzKjJCcC>. Citado na página 3.

JOHNSON, R. E.; FOOTE, B. Designing reusable classes. *Journal of object-oriented programming*, v. 1, n. 2, p. 22–35, 1988. Citado na página 3.

JUNIOR, C. G. B. *Agregando Frameworks de Infra-Estrutura em uma Arquitetura Baseada em Componentes: Um Estudo de Caso no Ambiente AulaNet*. Tese (Doutorado) — PUC-Rio, 2006. Citado na página 3.

JUNIOR, N. J. M.; AFONSO, A. *Produtividade no Desenvolvimento de Aplicações Web Com Spring Boot. 2º Edição*. Algaworks, 2017. Disponível em: <https://cafe.algaworks.com/fn013-download-livro-spring-boot/>. Citado 2 vezes nas páginas 3 e 4.

KOLEOSO, T. *Beginning Quarkus Framework: Build Cloud-Native Enterprise Java Applications and Microservices*. Apress, 2020. ISBN 9781484260319. Disponível em: <https://books.google.com.br/books?id=A2x7zQEACAAJ>. Citado na página 5.

LUNDBERG, C.; MATTSSON, M. Using legacy components with object-oriented frameworks. In: *Systemarkitekturer´ 96 Borås*. [S.l.: s.n.], 1996. Citado na página 3.