FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

# Zadání bakalářské práce

| | |
|---|---|
| **Název:** | Reaktivní technologie v jazyce Java |
| **Student:** | Patrik Hanes |
| **Vedoucí:** | Ing. Filip Glazar |
| **Studijní program:** | Informatika |
| **Obor / specializace:** | Webové a softwarové inženýrství, zaměření Softwarové inženýrství |
| **Katedra:** | Katedra softwarového inženýrství |
| **Platnost zadání:** | do konce letního semestru 2023/2024 |

## Pokyny pro vypracování

Cílem práce je popsat problematiku reaktivního programování v oblasti webových aplikací. Konkrétně se zaměřte na serverové aplikace implementované v jazyce Java a s ním souvisejících technologií. V rámci práce proveďte analýzu dostupných technologií a řádně je dle běžných softwarově inženýrských postupů otestujte. Postupujte dle následujících kroků.

1) Proveďte rešerši existujících implementací pro reaktivní přístup.
2) Zvolte vhodné metriky pro posouzení vámi zvolených implementací např. Quarkus Reactive, Spring Webflux, VertX apod.
3) Porovnejte zvolené řešení.
4) V případě potřeby implementujte vzorové aplikace v daných řešeních.
5) Zhodnoťte dané řešení a proveďte komplexní srovnání.

*Elektronicky schválil/a Ing. Michal Valenta, Ph.D. dne 15. ledna 2023 v Praze.*

Bakalárska práca

# REAKTIVNÍ TECHNOLOGIE V JAZYCE JAVA

**Patrik Hanes**

Fakulta informačních technologií
Katedra softwarového inženýrství
Vedúci: Ing. Filip Glazar
11. januára 2024

# Obsah

# Zoznam obrázkov

# Zoznam tabuliek



# Zoznam výpisov kódu

*Chcel by som sa v prvom rade poďakovať svojmu vedúcemu, Ing. Filipovi Glazarovi, za pomoc a čas ktorý mi venoval počas tvorby tejto práce. Zároveň by som sa chcel poďakovať mojej rodine a kamarátom za to, že ma počas štúdia neustále podporovali aby som to nevzdal.*

# Vyhlásenie

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací. Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského záakona, ve znění pozdějších předpisů. V souladu s ust. § 2373 odst. 2 zákona č. 89/2012 Sb., občanský zákoník, ve znění pozdějších předpisů, tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou a veškeré jejich dokumentace (dále souhrnně jen " Dílo"), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla, avšak pouze k nevýdělečným účelům. Toto oprávnění je časově, teritoriálňe i množstevně neomezené.

V Praze dňa 11. januára 2024 .....

# Abstrakt

Hlavnou úlohou tejto práce je vysvetliť a popísať reaktívny programovací paradigmat, a jeho uplatnenie v oblasti webových aplikácií. Práca sa zameriava na tvorbu serverových aplikácií v programovacom jazyku Java. Okrem poskytnutia informácií ohľadom toho, čo to reaktívne programovanie je, je cieľom práce predstaviť čitateľovi možné riešenia vo forme implementácií, poskytovaných technológiami Spring Webflux, Quarkus Reactivea Vert.x. V teoretickej časti je vysvetlené reaktívne programovanie, a následne udalosti ktoré viedli ku popularizácií tohto paradigmatu. Tieto technológie sú v teoretickej časti zanalyzované a v praktickej časti podľa zvolených metrík porovnané. Na konci sú zhodnotené výsledky porovnania, a nasleduje diskusia ohľadom najvhodnejšieho riešenia, ktoré si čitateľ môže zvoliť pri písaní reaktívnej webovej aplikácie.

**Klíčová slova** reaktívne programovanie, reactive streams, Java, Spring Webflux, Quarkus Reactive, Vert.x, backend, server, web, Reactive manifesto, observer pattern, CZECH

# Abstract

The main task of this work is to explain and describe the reactive programming paradigm and its application in the field of web applications. Specifically, the focus is on the creation of server applications in the Java programming language. In addition to providing information about what reactive programming is, the aim of the work is to present the reader with possible solutions in the form of implementations provided by technologies Spring Webflux, Quarkus Reactive and Vert.x. These technologies are analyzed in the theoretical part and compared in the practical part according to the chosen metrics. At the end, the results of the comparison are evaluated, followed by a discussion regarding the most appropriate solution that the reader can choose when writing a reactive web application.

**Keywords** reactive programming, reactive streams, Java, Spring Webflux, Quarkus Reactive, Vert.x, backend, server, web, Reactive manifesto, observer pattern, ENGLISH

# Seznam zkratek

| | |
|---|---|
| REST | Representational State Transfer |
| API | Application Programming Interface |
| HTTP | Hypertext Transfer Protocol |
| I/O | Input/Output |
| MVC | Model View Controller |

# Úvod

Posledných pár rokov zohrávajú webové aplikácie v našich životoch čoraz dôležitejšiu úlohu. Či už ide o veľké aplikácie, ako sú sociálne siete, stredne veľké, ako sú stránky elektronického bankovníctva, alebo menšie, ako sú online účtovné systémy alebo nástroje na riadenie projektov pre malé firmy, naša závislosť od týchto služieb jednoznačne rastie.

Bežný užívateľ má určité nároky na tieto aplikácie. Aby boli rýchle a responzívne. Aby nevznikali žiadne nečakané chyby. A ak by náhodou aj nejaká chyba vznikla, tak aby bola adekvátne ošetrená a uživateľ o nej informovaný. Architektúra moderných aplikacií žiaľ z veľkej časti nebýva trivialna. V rámci jednej domény je potrebné aby medzi sebou komunikovalo viacero softwerových systémov. Zároveň sa aplikácia može rozdeliť na menšie časti, aby boľi tieto systémy z dlhodobého hľadiska udržovatelné, a zároveň aby ich bolo možné upraviť podľa potreby.

S vysokými nárokmi na softwér prirodzene vznikajú nové spôsoby ako takéto programy implementovať. Reaktívne programovanie je pojem ktorý vo svete informačných technologií existuje už nejakú dobu, a predstavuje spôsob, akým sa takéto naročné systémy daju písať. Na internete je človek schopný si najsť kopu blogov, knižiek, a článkov na túto tému, no tento paradigmat nie je vyslovene jednoznačný. Existujú mnohé situácie kde sa dá využiť, ale analogicky existujú problemý pri ktorých nám reaktivny spôsob vie zbytočné zkomplikovať riešenie. Čim lepšie pochopenie pre tento paradigmat programátor má, o to lepšie vie kedy ho má použiť a akým spôsobom. Vzhľadom k tomu, že táto téma je veľmi široká, a verím, že bežný programátor by mal vedieť, že niečo takéto existuje, je účeľom tejto práce poskytnúť rešerš v oblasti reaktívneho programovania, a poskytnúť čitateľovi tejto práce súhrn toho najdôležitejšieho, čo sa týka reaktívneho programovania. Nakoľko sa reaktívne programovanie vyskytuje vo viacerých typoch softwéru, v rámci tejto práce som sa zameral konkrétne na serverové aplikácie.

Tému reaktívneho programovania som si vybral, pretože veľmi rád skúmam programovacie jazyky a rôzne programovacie paradigmata. Fascinuje ma sledovať ako vyzerá výsledny kód pri týchto rôznych postupoch. Vzhľadom k tomu, že na univerzite sa reaktívne programovanie moc do detailu nepreberá, som bol motivovaný sa na túto temú pozrieť a niečo o nej napísať.

Tak ako funkcionálne a objektovo-orientované programovanie, ani reaktívne programovanie nie je kompletné riešenie na všetky problémy. Je to len další nástroj, akým programátor môže riešiť špecifický druh problémov.

V teoretickej časti si vysvetlíme všetky dôležité pojmy spojené s reaktívnym programovaním. Čo to je, prečo to vzniklo, a v akých formách sa dá aplikovať na webové

aplikácie. Nasledovať bude rozsiahlejší popis môjho výberu reaktívnych technologií ktoré sú cieľom tejto práce. Ide o technológie Vert.x, Quarkus a jeho reaktívna nadstavba Quarkus Reactive, a nakoniec Spring framework a jeho reaktívna nadstavba Spring Weblux. Po predstavení týchto technologií bude nasledovať praktická časť, v ktorej sa zameriame na metriky ktoré boli zvolené v tejto práci na porovnanie spomínaných technologií. Následne si popíšeme spôsob merania týchto metrík a ku koncu zhodnotíme výsledky porovnania, na základe ktorých vyhodnotíme silné a slabé stránky spomínaných technologií.

# Kapitola 1
# Cieľ práce

Cieľom teoretickej časti je popísať reaktívne programovanie v oblasti webových aplikácií. Následne sa zameriame na 3 konkrétne riešenia v programovaciom jazyku Java, a to Spring Webflux, Quarkus Reactive a Vert.x. Tieto technológie zanalyzujeme a podrobne popíšeme.

Súčasťou praktickej časti je voľba vhodných metrik na porovnanie zvolených technológií. Zároveň k jednotlivým technológiam budú implementované názorné ukážky serverovej aplikácie v jazyku Java. V konečnom dôsledku sa výsledky porovnajú a zhodnotí sa, ktorá technológia je spomedzi ostatných najoptimalnejšia.

Prínosom tejto práce je čitateľovi poskytnúť informácie ohľadom jedinečného programovacieho paradigmatu, a zároveň ponúknuť praktický prehľad o možných riešeniach a názornych ukážkach v programovaciom jazyku Java.

# Kapitola 2

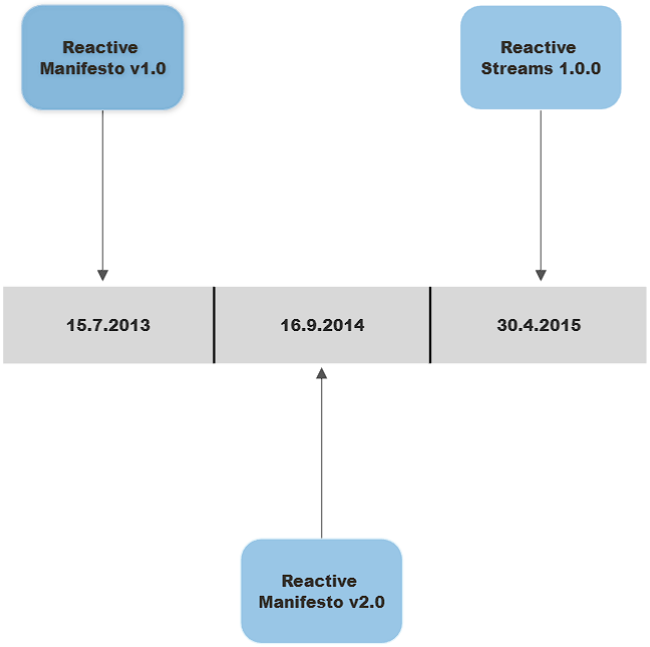
# Reaktívne programovanie

*V tejto kapitole si podrobne vysvetlíme všetky pojmy a termíny spojené s reaktívnym programovacím paradigmatom.*

## 2.1 Motivácia

Vo svete programovania existujú viaceré spôsoby, ako riešiť problémy, konkrétne viaceré spôsoby, ako písať samotný kód. Máme objektovo-orientované programovanie, funkcionálne programovanie, a rôzne iné typy programovania.[1] To ako píšeme kód, z veľkej časti záleží v akom jazyku programujeme, každopádne prostredníctvom ekosystému knižnic, ktoré jazyk má, sme schopný si jazyk upraviť natoľko, aby sme vedeli pracovať aj s inými metodikami, mimo limity samotného jazyka. Napríklad, programovací jazyk Java je primárne určený na objektovo-orientované programovanie, no to neznamená, že sa v ňom nedá funkcionálne programovať. Reaktívne programovanie je tiež jedno z týchto paradigmat, no oproti spomenutým typom programovania je skôr viazané viacej k typu problému aký riešime, ako skôr jazyk ktorý používame. Počiatky reaktívneho programovania začali niekedy v 80-tých rokoch minulého storočia.[2] No popularita tohto paradigmatu narástla v čase vzniku architektúry mikroslužieb. Dôvodom je skutočnosť, že tradičné imperatívne programovanie má určité obmedzenia, pokiaľ ide o zvládanie techických požiadaviek dnešnej doby, kedy aplikácie musia mať vysokú dostupnosť a poskytovať nízke časové odozvy aj pri vysokej záťaži. Pokiaľ si do webového vyhľadávača napíšete reaktívne programovanie, vyskočí na vás veľa rôznych článkov a pojmov, ktoré na prvý pohľad nemusia dávať okamžite zmysel. Slovo reactive sa spája s viacerými inými slovami, a tvoria veľmi rozdielne pojmy. Reactive manifesto, reactive streams, reaktivne programovanie, reactive system a iné. V nasledujúcich kapitolách si najprv predstavíme samotný paradigmat reaktívneho programovania, a následne si vysvetľíme jednotlivé pojmy spojené s týmto paradigmatom, v poradí v akom vznikali, a vybudujeme časovú líniu prostredníctvom ktorej vzniklo reaktívne programovanie, také existuje teraz. Časová os je znázornená na obrázku 2.1.

**Obr. 2.1** Časová os vzniku reaktívneho programovania




## 2.2 Definície spojené s webovými aplikáciami

Ešte pred tým ako prejdeme na samotnú teóriu, je potrebné si objasniť terminológiu v nasledujúcich kapitolách.

### 2.2.1 Server

Server je zariadenie, ktoré ukladá, odosiela a prijíma dáta. Počítač, softvérový program alebo úložné zariadenie môže fungovať ako server a môže poskytovať jednu alebo viacero služieb.[3]

### 2.2.2 Serverová aplikácia

Serverová aplikácia je program, ktorý sa nachádza na strane servera a ktorý poskytuje nejaké služby vo forme poskytovania dát alebo vykonávania nejakej logiky.[4]

### 2.2.3 API

Application programming interface (API) je typ softwérového rozhrania, vďaka ktorému je možné, aby dva alebo viac počítačových programov mohlo medzi sebou navzájom komunikovať.[5]

### 2.2.4 Request/Požiadavka

Sieťové požiadavky sa používajú na získanie alebo úpravu údajov prostredníctvom API zo servera. Ide o komunikáciu primárne za pomoci internetovej siete.[6]

### 2.2.5 HTTP

HTTP je sieťový protokol, ktorý sa používa na prenos údajov cez internet a umožňuje používateľom prístup na webové stránky a iné online zdroje.[7]

### 2.2.6 REST API

REST API je rozhranie, ktoré dodržuje zásady architektonického vzoru REST, a ktoré používa HTTP požiadavky na prístup a manipuláciu údajov, najčastejšie dát ktoré sídlia na nejakom serveri.[8]

### 2.2.7 Concurrency

Concurrency, v preklade funkční paralelizmus, je činnosť vykonávania viacerých úloh naraz. Z hľadiska počítačového systému, pod paralelizmom sa chápe činnosť spravovania viacerých procesov na operačnom systéme. Plánovanie týchto procesov je vykonávané tak, že jadro procesora prepína vykonávanie kódu medzi viacerými vláknami na vytvorenie ilúzie, že tieto procesy sa vykonavajú v rovnaký čas a v rovnaký moment. Paralelizmus sa odkazuje na provoz jadier operačného systému, vďaka ktorým sú inštrukcie na vláknach vykonané v rovnaký moment.[9] Príklad aplikácie ktorá takúto vlastnosť využíva môže byť načúvanie na vstup od uživateľa, popri tom ako aplikácia vykonáva iné výpočty na pozadí. Podobne, paralelizmus vo webových aplikáciach môže byť možnosť obsluhovať viaceré požiadavky naraz v rovnakom čase.

### 2.2.8 Thread per request model

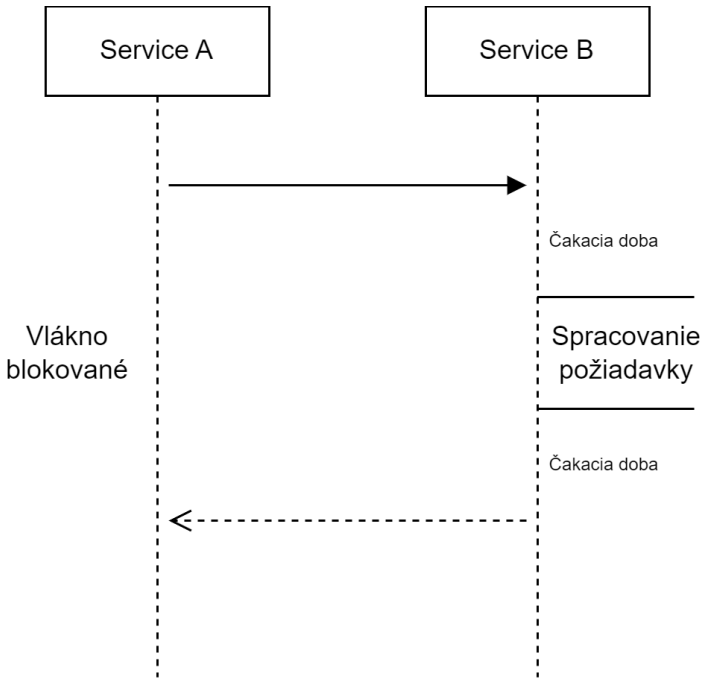
Pre lepšie pochopenie, čo reaktívne programovanie je a aké výhody prináša, zvážme najprv tradičný spôsob vývoja serverovej aplikácie v jazyku Java za pomoci frameworku Spring MVC. Teda ako príklad si zoberme serverovú aplikáciu, ktorá poskytuje REST rozhranie na komunikáciu prostredníctvom HTTP protokolu. Kontajner servletu má vyhradenú oblasť vlákien na spracovanie požiadaviek HTTP, pričom každá prichádzajúca požiadavka dostane priradené vlákno a toto vlákno bude spracovávať celý životný cyklus požiadavky (model *Thread per request*). To znamená, že aplikácia bude schopná spracovať len taký počet súbežných požiadaviek, ktorý sa rovná veľkosti počtu vlákien. Je možné nakonfigurovať veľkosť počtu vlákien, ale keďže každé vlákno si vyhradzuje určitú pamäť, čím vyšší počet vlákien nakonfigurujeme, tým vyššia bude spotreba pamäte.[10]

Výkon aplikacie sme schopný škáľovať v závilosti od počtu požiadavkov, ale za cenu vysokého využitia pamäte. Takže model *Thread per request* by mohol byť dosť nákladný pre aplikácie s vysokým počtom súbežných požiadaviek.

### 2.2.9 Čakanie na I/O operácie

Dôležitou charakteristikou architektúr založených na mikroslužbách (aplikácia je rozdelená na menšie samostatné celky) je, že aplikácia je distribuovaná, beží rozdelená ako viacero samostatných procesov, zvyčajne na viacerých serveroch. Používanie tradičného imperatívneho programovania so synchronným spracovaním požiadávok na komunikáciu medzi službami znamená, že vlákna sa častokrát blokujú pri čakaní na odpoveď inej služby – čo vedie k obrovskému plytvaniu zdrojov.[11] Rovnaký typ plytvania sa vyskytuje aj pri čakaní na dokončenie iných typov I/O operácií, ako je volanie databázy alebo čítanie zo súboru. Vo všetkých týchto situáciách bude vlákno, ktoré si vyžaduje I/O operácie zablokované, a bude nečinné čakať, kým sa I/O operácia nedokončí. Popisovaná situácia je znázornená na 2.2.

**Obr. 2.2** Blokované vlákno čaká na odpoveď




### 2.2.10 Časová odozva

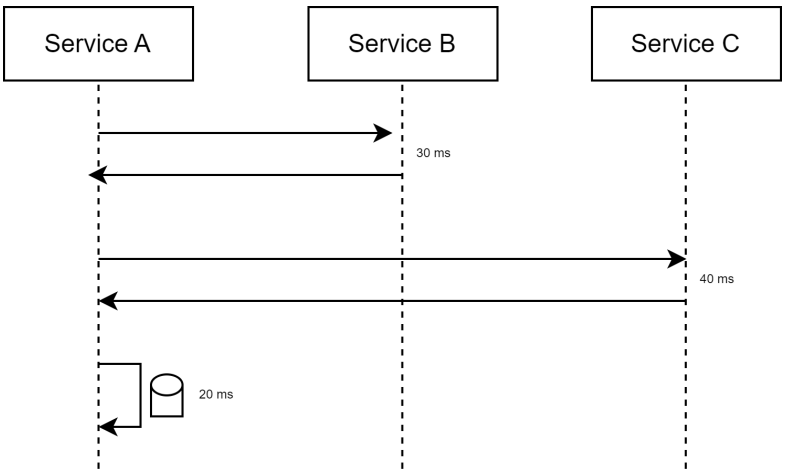
Ďalším problémom tradičného imperatívneho programovania sú výsledné časy odozvy, kedy nejaká služba potrebuje vykonať viac ako jednu I/O požiadavku. Napríklad služba A môže potrebovať zavolať službu B a C a tiež vykonať vyhľadávanie v databáze a následne vrátiť nejaké agregované údaje.

To by znamenalo, že čas odozvy služby A by bol okrem vlastného času súčtom (znázornené na obrázku 2.3):

- čas odozvy služby B
- čas odozvy služby C

- času odozvy databázovej požiadavky

**Obr. 2.3** Blokované vlákno pri práci s viacerými službami




Ak neexistuje žiadny skutočný logický dôvod na uskutočňovanie týchto volaní postupne, určite by to malo veľmi pozitívny vplyv na čas odozvy služby A, ak by sa tieto volania vykonávali paralelne. Aj keď existuje podpora pre vykonávanie asynchrónnych volaní v jazyku Java pomocou *CompletableFutures* a registrácie spätných volaní, rozsiahle používanie takéhoto prístupu v aplikácii by robilo kód zložitejším a ťažšie čitateľným a udržiavaným.[12]

### 2.2.11 Preťaženie klienta

Iný typ problému, ktorý sa môže vyskytnúť v prostredí mikroslužieb, je, keď služba A požaduje nejaké informácie od služby B, povedzme napríklad všetky objednávky zadané počas minulého mesiaca. Ak sa ukáže, že množstvo objednávok je obrovské, pre službu A môže byť problém získať všetky tieto informácie naraz. Služba A môže byť zahltená veľkým množstvom údajov a výsledkom môže byť napríklad chyba nedostatku pamäte.

## 2.3 Definícia reaktívneho programovania

Reaktívne programovanie je programovacia paradigmata, ktorá je postavená na myšlienke stáleho prúdu dát v čase, a propagacie zmeny týchto dát. Uľahčuje to primárne deklaratívnosť vývoju aplikácií riadených udalosťami tak, že umožňuje vývojárom vyjadrovať kód programu z hľadiska toho, čo sa má robiť keď data prídu, a nechať jazyk automaticky riadiť, kedy data spracovať. Inak povedané, programátor napíše čo sa má stať so vstupom, a jazyk porieši na pozadí, kedy sa táto logika spustí. V tejto paradigme sa zmeny stavu automaticky a efektívne šíria po všetkých závislých výpočtov podľa interného modelu vykonávania. [2]

Zoberme si nasledujúcu ukážku kódu, ako triviálny príklad reaktivného programovania.

```
1  var a = 4
2  var b = 3
3  val c = a + b
```

Premenné $a$ a $b$ maju priradené pevne dané hodnoty. Naopak, premenná $c$ je definovaná prostredníctvom $a$ a $b$. To znamená, že pokiaľ by sme zmenili hodnotu v jednej z týchto premenných, táto zmena logicky ovplyvní $c$. Inými slovami povedané, hodnota $c$ sa vždy prepočíta, keď sa zmení hodnota $a$ alebo $b$. Toto je kľúčová myšlienka reaktívneho programovania. Hodnoty sa postupom času menia, a keď sa zmenia, všetky závislé výpočty sa automaticky prepočítajú.

Tento princip sa da elegatne aplikovať na mnohých situáciach. Čo ak uživateľ stlačí nejaké tlačítko na webovej stránke, a je potrebné zmeniť farbu, alebo vypnúť nejaké textové polia ? Čo ak, na serverovú aplikáciu príde požiadavka, a je potrebné túto požiadávku spracovať a poslať odpoveď ? Tieto problémy sa určite dajú riešiť aj procedurálne, no nie je to elegatné riešenie. S vačším počtom zmien, sa zároveň zvyšuje aj veľkosť a komplexita kódu. Udalosti, ktoré vznikajú v rámci programu tvoria prúd dat (stream dat), podľa ktorého budujeme reaktívnu logiku.

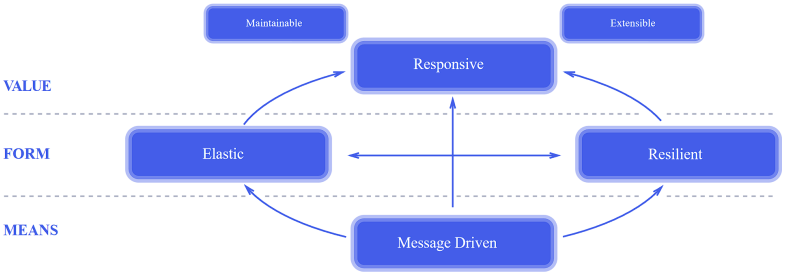
## 2.4 Reaktívne manifesto

Reactive manifesto je dokument, ktorý bol publikovaný v roku 2013 a ktorého autori sú *Jonas Bones*, *Dave Farley*, *Roland Kuhn* a *Martin Thompson*[11]. Vznik tohto manifesta bol jeden z klučovych faktorov ktorý viedol ku popularizácií reaktívneho programovania. Tento dokument vznikol s cieľom zhrnúť poznatky, ktoré odborníci v softwérovom inžinierstve za posledných pár rokov nazhromaždili, a to ako navrhovať vysoko-škáľovateľné a spoľahlivé aplikácie. Pri vzniku manifesta bola hlavná myšlienka zobrať jadro všetkých implementácií systémov, ktoré riešili spoločný problém, a vytvoriť sadu základných principov, ktoré by napomohli zlepšiť súčasný stav v softwérovom inžinierstve. Programátori dovtedy nevedome dospievali k rovnakým riešeniam k problémom, spojených s technologickými pokrokmi v oblasti systémovej architektúry. Architektúra systemú, ktorého manifesto popisuje, konkretné reaktívna architektúra, nie je univerzálne riešenie na všetky problémy, skôr by mala byť chápaná ako východiskový bod podľa ktorého by sa moderný softwér mal inšpirovať. Ďalší spôsob, ako sa môžme pozrieť na reaktívny manifest, je nazvať ho slovníkom osvedčených postupov, z ktorých niektoré už sú dlho známe. Jadro reaktívneho manifesta je opis takzvaného reaktívneho systému. Ide o systém, ktorý z definície spĺňa nasledujúce vlastnosti:

1. Responsive(responzívny)
2. Resilient(odolný)
3. Elastic(prispôsobivý)
4. Message driven(riadený posielanim zpráv)

Systém, ktorý spĺňa tieto vlastnosti nazývame reaktívny. Systémy navrhnuté ako reaktívne, sú viacej flexibilné, voľne viazané a ľakho škáľovatelné. Vďaka týmto principom sa dajú jednoduchšie vyvíjať a je ľahké zapracovať nové zmeny. V momente keď nastane chyba, tak systém vie adekvátne reagovať.[11]

**Obr. 2.4** Reactive Manifesto: [11]




V nasledujúcich podkapitolách si popíšeme každú vlastnosť reaktívneho systému.

### 2.4.1 Responsive

V momente, keď nastane problém, vie systém dostatočne informovať uživateľa, kde nastala chyba. System okamžite odpovedá. Toto chovanie zjednodušuje správu chýb a buduje dôvernosť u uživateľa.[11]

### 2.4.2 Resilient

V situáci, kedy nastane chyba, vie systém advekvátne reagovať. Chyba nastáva v ramci komponenty systému, tým pádom sa nešíri na ďalšie komponenty, a zároveň obnova je možná bez narušenia behu systému. Klient nemá zodpovednosť spravovať chyby.[11]

### 2.4.3 Elastic

V prípade, kedy je veľa práce, tak systém je stále responzívny. Vedia sa zvyšovať alebo znižovať pracovné prostriedky na spracovanie dát podľa potreby, bez straty výkonu.[11]

### 2.4.4 Message driven

Komponenty systému medzi sebou kominikujú asynchronným posielaním správ. To znamená, že nenastáva blokovanie pri komunikacií. Beh aplikácie nie je zavislý od reťazovej reakcie jednotlivých komponent. Tým pádom sú jednotlivé časti systému izolované a nezávislé. [11]

## 2.5 ReactiveX

V roku 2011 spoločnosť Microsoft vydala knižnicu Reactive Extensions (ReactiveX alebo Rx) pre platformu .NET, ktorá poskytuje zjednodušený spôsob vytvárania asynchrónnych programov riadených udalosťami. V priebehu niekoľkých rokov bola Reactive Extensions prenesená do niekoľkých jazykov a platforiem vrátane Java, JavaScript, C++, Python a

Swift. Vývoj implementácie Java - RxJava - bol riadený spoločnosťou Netflix a verzia 1.0 bola vydaná v roku 2014. [13] ReactiveX používa kombináciu návrhového vzoru Iterator a vzoru Observer.[14]

## 2.6 Reactive Streams

Reactive Streams je špecifikácia, určená na implementáciu knižnic na asynchrónne spracovanie streamov dát v jazyku Java. Iniciatíva ku vzniku tejto špecifikácie začala v roku 2013 medzi programátormi v Netflixe, Pyvotal a Typesafe ktorá následne viedla ku vydaniu prvej verzie Reactive Streams API na platforme GitHub.[13] API definuje metódy a pravidlá na implementáciu a interakciu so streamami dát neblokujúcim a asynchrónnym spôsobom.[15] Táto špecifikácia bola vyvinutá na riešenie problémov spojených so vznikom tlaku (backpressure) na vysokovýkonných systémoch. Backpressure v kontexte serverových aplikací, je riziko preťazenia servera požiadavakami natoľko, že ich nezvláda spracovávať.[13] Reactive Streams je podporovaný množstvom populárných technologií v jazyku Java, ako sú napríklad Akka, Quarkus a Spring Reactor. Je tiež používaný v iných programovacích jazykoch, vrátane Kotlin, Scala a Javascript.

Reactive streams sa zvyčajne implementujú pomocou množiny rozhraní a tried, ktoré definujú správanie Publisher, Subscriber a Subscription. Publisher sú zodpovední za vytváranie udalostí a informovanie predplatiteľov, keď sú k dispozícii nové udalosti. Odberatelia využívajú udalosti a podľa potreby požadujú od vydavateľa ďalšie udalosti. Predplatné predstavuje spojenie medzi vydavateľom a predplatiteľom a umožňuje predplatiteľovi riadiť tok udalostí.

Špecifikácia zahŕňa nasledujúce rozhrania:

### 2.6.1 Publisher

Toto rozhranie predstavuje producenta zdroju dát a má práve jednu metódu, ktorá umožňuje objektu Subscriber zaregistrovať sa u objektu Publisher.[16]

```
1     public interface Publisher<T> {
2         public void subscribe(Subscriber<? super T> s);
3 }
```

**Výpis kódu 1** Rozhranie Publisher[17]

### 2.6.2 Subscriber

Rozhranie predstavuje konzumenta a má nasledujúce metódy:

- *onSubscribe* má zavolať objekt Publisher pred začiatkom spracovania a používa sa na odovzdanie objektu Subsription Subscriberovi
- *onNext* sa používa na signalizáciu, že vznikli nové data

```
1     public interface Subscriber<T> {
2         public void onSubscribe(Subscription s);
3         public void onNext(T t);
4         public void onError(Throwable t);
5         public void onComplete();
6 }
```

**Výpis kódu 2** Rozhranie Subscriber[17]

- *onError* sa používa na signalizáciu toho, že Publisher zaznamenal zlyhanie a už nevzniknú žiadne nové data
- *onComplete* sa používa na signalizáciu, že všetky data boli úspešne odoslané [18]

### 2.6.3 Subscription

Subscription obsahuje metódy, ktoré klientovi umožňujú kontrolovať emisiu položiek Publishera (t. j. poskytovanie podpory proti tlaku).[19]

```
1     public interface Subscription {
2         public void request(long n);
3         public void cancel();
4     }
```

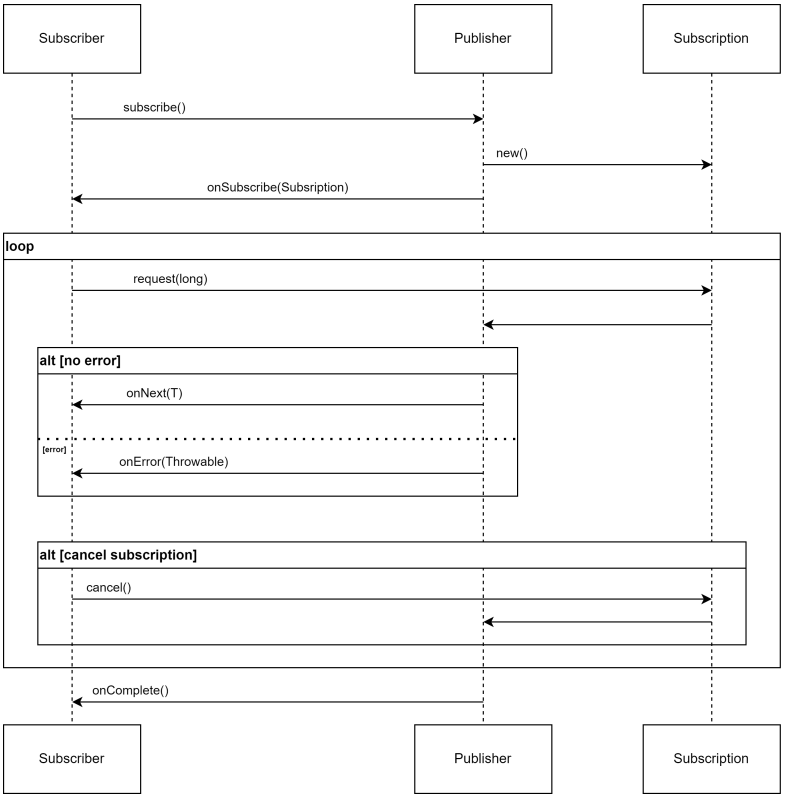
**Výpis kódu 3** Rozhranie Subscription[17]

- *request* umožňuje Subscriberovi informovať Publishera o tom, koľko dodatočných prvkov má byť zverejnených
- *cancel* umožňuje Subsriberovi zrušiť ďalšie vydávanie položiek Publisherom

Na obrázku 2.5) je znázornená komunikácia jednotlivých objektov Reactive Streams. Tu sekvencia udalostí prebieha nasledovne:

1. Subscriber inicializuje požiadavku vyvolaním metódy subscribe() Publishera.
2. Publisher odošle objekt Subscription vyvolaním metódy onSubscribe() Subscribera.
3. Keď Subsriber prijme objekt Subscription, vyvolá funkciu request() Publishera požadujúceho od Publishera zaslanie údajov. Ako parameter k metóde zadávame, koľko dát chceme prijímať.
4. Publisher začne odosielať dáta, len čo sú dostupné, vo forme streamu dát vyvolaním funkcie onNext() Subscriber rozhrania.

**Obr. 2.5** Reactive Streams - zjednodušený diagram komunikácie




5. Po odoslaní všetkých údajov Publisher vyvolá metódu onComplete() Subscribera a celý proces skončí.

V tomto prípade Publisher vyvolá všetky metódy od Subscribera okrem onError(). V opačnom prípade, teda keď nastane chyba sa zavolá funkcia onError().

Hlavna myšlienka tohto toku udalostí je prítomná vo všetkých reaktívnych technológiach s výnimkou až na implementačné detaily. Práca so streamom dát je založená na správnej identifikácií, z čoho stream vytvárame, a keď do streamu prídu nejaké data, čo s nimi následne vykonáme.

# Kapitola 3

# Reaktívne technológie v Jave

*V tejto kapitole sa zameriame na dôvody, ktoré ovplyvnili výber trojice technologií Vert.x, Spring a Quarkus. Zároveň si popíšeme motiváciu, ktorá by programátora mohla osloviť v oblasti tvorby webových aplikacií v jazyku Java. Ku koncu si stručne predstavíme jednotlivé technologie a ich kľučové prvky, ktoré poskytujú reaktivné vlastnosti.*

## 3.1 Výber technológií

Programovací jazyk Java je v čase písania tejto práce najviac využívaný v oblasti serverových aplikacií. Knižnic a frameworkov vďaka ktorým dokážeme komunikovať prostredníctvom HTTP protokolu je veľké množstvo. Určitá časť týchto technologií podporuje model spracovania požiadaviek za pomoci modelu Thread per request. Iné zase od základu majú podporu pre reaktívny spôsob, teda asynchronný neblokujúci. Za určitých podmienok si potencionálny developer vystačí s neblokujúcim spôsobom, za predpokladu, že je malá pravdepodobnosť že v určitom momente bude serverová aplikácia spracovávať veľké množstvo požiadavok v rovnaký moment, a zároveň pokiaľ počítač, na ktorom je aplikácia zprovoznená, ma k dispozícií nadmerné množstvo vykonnostných prostriedkov. Na druhú stranu, ak sa očakáva veľké množstvo požiadavkov, je ideálne zvoliť reaktívny spôsob.

Pre maximálne zúžitkovanie reaktívneho spôsobu by bolo ideálne, aby aplikácia počas celkového vykonávania svojich funkcií bola plne asynchronná. Prakticky to znamená, že od momentu, kedy serverový program dostane požiadavku prostredníctvom HTTP, a následne túto požiadavku spracuje a prípadne komunikuje s inými službami, tak v najlepšom prípade nedôjde ku žiadnemu blokovaniu. To znamená, že okrem spracovania požiadaviek je potrebné asychronne spracovať aj volania na iné služby, ako napríklad iné serverové aplikácie alebo databázy.

Z toho vyplýva, že ideálne spomedzi technológie ktoré su dostupné, budeme vyberať tie, ktoré podporujú reaktívny spôsob programovania, sú ideálne používané v komerčnej sfére, a značne výkonnostne lepšie ako ich blokujúce varianty.

V rámci rešeršu reaktívnych technológií so zameraním na rýchlosť som zvolil webovú stránku Web Framework Benchmarks. Táto stránka hodnotí výkonnosť rôznych frameworkov pre vývoj webu naprieč celým radom platforiem vrátane Java, PHP, Node.js a ďalších. Robí to spustením série porovnaní v rámci rôznych metrik, ktoré simulujú scenáre

webových aplikácií v reálnom svete, ako je poskytovanie statického obsahu, dotazovanie na databázu a spracovanie údajov JSON.

Tieto porovnania merajú čas, ktorý každý framework potrebuje na dokončenie úlohy, ako aj požadované využitie pamäte a CPU. Výsledky sú prezentované v sérii grafov a tabuliek, ktoré umožňujú vývojárom porovnať výkon rôznych rámcov na rôznych platformách.

Z týchto porovnaní som zobral najaktualnejšie porovnanie s dátumom 19.7.2022, a obmedzil som sa na jazyk Java s testom, v ktorom každá požiadavka vedie ku načítaniu viacerých riadkov z jednoduchej databázovej tabuľky a následnou serializáciou týchto riadkov ako odpovede JSON. Spomedzi všetky výsledky som zobral prvých 15. Výsledky testu su na obrázku 3.1:

**Obr. 3.1** Odpovede za sekundu pri 20 dotazoch na požiadavku - Java. Prevzaté z [20].

**Responses per second at 20 queries per request, Dell R440 Xeon Gold + 10 GbE** (76 tests)

| Rnk | Framework | Performance (higher is better) | Errors | Cls | Lng | Plt | FE | Aos | DB | Dos | Orm | IA |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | vertx-postgres | 34,821 100.0% (99.6) | 0 | Plt | Jav | ver | Non | Lin | Pg | Lin | Raw | Rea |
| 2 | inverno | 34,523 99.1% (98.7) | 0 | Mcr | Jav | inv | Non | Lin | Pg | Lin | Mcr | Rea |
| 3 | officefloor-sqlclient | 33,331 95.7% (95.3) | 0 | Plt | Jav | off | woo | Lin | Pg | Lin | Raw | Rea |
| 4 | greenlightning | 33,296 95.6% (95.2) | 0 | Mcr | Jav | Non | Non | Lin | Pg | Lin | Raw | Rea |
| 5 | officefloor-async | 33,207 95.4% (95.0) | 0 | Plt | Jav | off | woo | Lin | Pg | Lin | Raw | Rea |
| 6 | ratpack-pgclient | 32,084 92.1% (91.8) | 0 | Mcr | Jav | Nty | Non | Lin | Pg | Lin | Raw | Rea |
| 7 | wizzardo-http | 31,770 91.2% (90.9) | 0 | Mcr | Jav | Non | Non | Lin | Pg | Lin | Raw | Rea |
| 8 | jooby-pgclient | 31,703 91.0% (90.7) | 0 | Ful | Jav | Utw | Non | Lin | Pg | Lin | Raw | Rea |
| 9 | redkale-graalvm | 31,606 90.8% (90.4) | 0 | Ful | Jav | red | red | Lin | Pg | Lin | Raw | Rea |
| 10 | redkale | 31,318 89.9% (89.6) | 0 | Ful | Jav | red | red | Lin | Pg | Lin | Raw | Rea |
| 11 | redkale-native | 30,607 87.9% (87.5) | 0 | Ful | Jav | red | red | Lin | Pg | Lin | Raw | Rea |
| 12 | officefloor-r2dbc | 30,422 87.4% (87.0) | 0 | Plt | Jav | off | woo | Lin | Pg | Lin | Raw | Rea |
| 13 | helidon | 30,368 87.2% (86.9) | 0 | Mcr | Jav | Nty | Non | Lin | Pg | Lin | Raw | Rea |
| 14 | spring-webflux-pgclient | 29,001 83.3% (82.9) | 0 | Ful | Jav | Nty | Non | Lin | Pg | Lin | Mcr | Rea |
| 15 | quarkus resteasy reactive + hibernate re | 27,972 80.3% (80.0) | 0 | Ful | Jav | res | ver | Lin | Pg | Lin | Ful | Rea |

Následne ma zaujímala popularita frameworkov, aby som vedel ktoré z týchto riešení sú obľúbené v Java komunite, z hľadiska komerčného použitia. Ako zdroj popularity webových technológií som použil dotazník Java Survey Report 2023 od spoločnosti Vaadin[21]. Výsledky ich dotázniku na popularitu webových technologií v jazyku Java sú na obrázku 3.2:

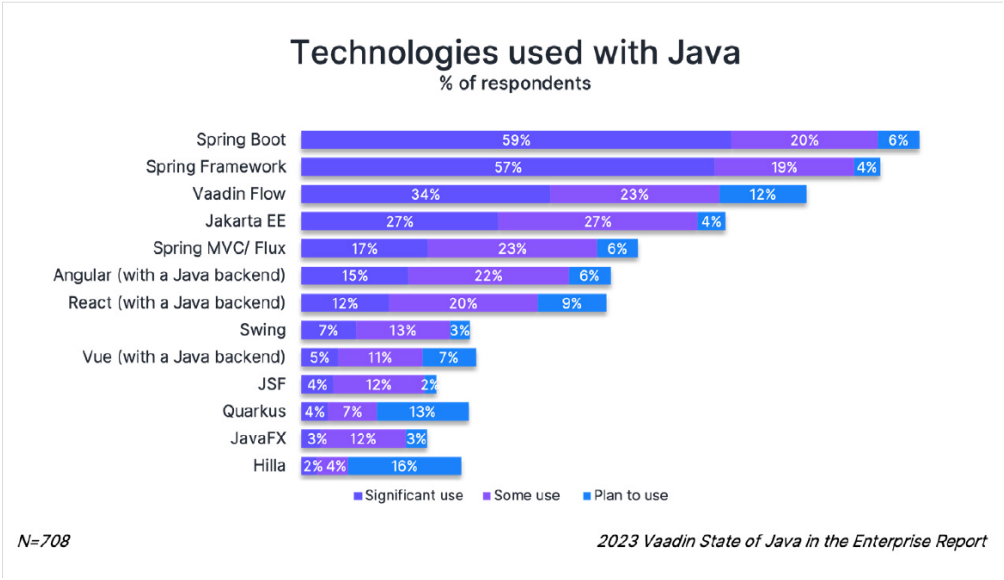
Okrem predošlého dotazníku som sa pozrel aj na dotazník *The State of developer ecosystem - Java programming* od spoločnosti Jetbrains[22], ktorého výsledky sú na obrázku 3.3:

Zo všetkých riešení, ktoré boli najefektívnejšie som si chcel vybrať tie, ktoré boli v najlepšom prípade súčasťou nejakej technológie, ktorá sa používala na tvorbu serverových aplikacií. Teda integrácia s HTTP požiadavkami a pripadným dotazovaním na databázu by ideálne nebola riešená prostredníctvom tretej strany. Zároveň som chcel riešenia, ktoré by spadali pod seriózne produkty, používane v komerčnej sfére, čiže knižnice vytvorené na online repozitároch, ako neoficiálne riešenia som ignoroval.

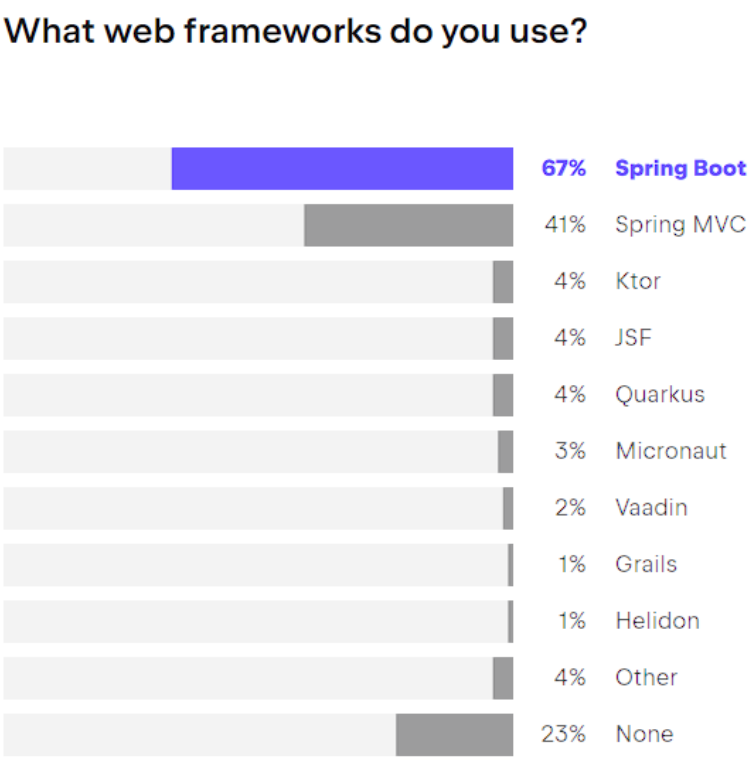
Čo sa týka kandidátov, ktorý spadajú pod najpopulárnejších, tak spomedzi nich som sa zameral na tých, ktorý ponukajú reaktívnu funkčnosť. To znamená, že išlo o technológie na tvorbu serverových aplikácií, ktoré podporovali reaktívne programovanie.

Spomedzi kandidátov, ktorý sa podľa výkonnu nachádzali vo vrchných priečkach a zároveň sú komerčne používane a populárne, som si vybral technologie Spring, Quarkus a Vert.x. Spring a Quarkus sú frameworky, ktoré okrem imperativného riešenia ponúkaju možnosť reaktívne programovať. Spring prostredníctvom Spring Webflux, a Quarkus prostredníctvom Quarkus Reactive. Vert.x na rozdiel od spomenutých dvoch riešení je od základov navrhnutý ako reaktívny. [23]

**Obr. 3.2** Technologies used with Java - Vaadin Report[21]




**Obr. 3.3** Java programming - state of developer ecosystem 2022. Prevzaté z [22]



## 3.2 Spring

Spring je open-source framework na vytváranie webových aplikacií v programovaciom jazyku Java. V nasledujúcích podkapitolách si predstavíme jeho najdôležitejšie časti na tvorbu reaktívnych webových aplikacií.[24]

### 3.2.1 Spring Boot

Spring Boot je framework založený ako nadstavba nad Spring frameworkom, a jeho hlavným prínosom je, že nám ponuká možnosť rýchlejšieho vývoja webových služieb. Spring Boot nám štandardne ponúka ako voľby jeho MVC model, alebo reaktívny model, vo forme Reactive Webflux. V našom prípade sa zameriame na reaktívny model, kedže MVC model je blokujúci.[25]

### 3.2.2 Spring Webflux

Spring Webflux je plne neblokujúca nadstavba nad Spring frameworkom, ktorá podporuje reaktívne streamy. Interne používa knižnicu Project Reactor, ktorá implementuje špecifikáciu Reactive Streams. Kedže ide o asynchronnú funkcionalitu, v kóde nemôžme priamo vraciať samotné objekty. Tieto objekty je potrebné zabaliť do niečoho, čo bude predstavovať asynchronný výsledok. Preto Project Reactor ponúka svoje Subscriber riešenie vo forme objektov Flux a Mono.[10]

### 3.2.3 Flux

Flux je objekt, ktorý v rámci špecifikácie Reactive Streams plní úlohu Publisher, ktorý predstavuje asynchrónnu sekvenciu 0 až N prkvov, voliteľne ukončených buď signálom dokončenia alebo chybou. Tieto tri typy signálovsa prekladajú na volania metód onNext, onComplete a onError následného Subscribera. Pomocou Flux, sme schopný vyjadriť prúd dat, ktorý je buď konečný alebo nekonenčný. Tieto vlastnosti robia Flux reaktívny typ na všeobecné účely.[26]

```
1    Flux<Integer> ints = Flux.range(1, 4)
2        .map(i -> {
3            if (i <= 3) return i;
4            throw new RuntimeException("Got to 4");
5        });
6    ints.subscribe(i -> System.out.println(i),
7        error -> System.err.println("Error: " + error));
```

**Výpis kódu 4** Ukážka práce s objektom Flux [26]

### 3.2.4 Mono

Mono na rozdiel od Fluxu, je špecializovaný Publisher, ktorý vysiela maximálne 1 položku prostredníctvom signálu onNext, potom končí signálom onComplete, alebo vydáva iba jeden signál onError. Mono ponúka iba podmnožinu operátorov, ktoré sú dostupné pre Flux, a niektorí operátori priamo vraciajú ako výsledok Flux.[27]

```
1    Flux<String> ids = ifhrIds();
2    Flux<String> combinations =
3        ids.flatMap(id -> {
4            Mono<String> nameTask = ifhrName(id);
5            Mono<Integer> statTask = ifhrStat(id);
6            return nameTask.zipWith(statTask,
7                        (name, stat) -> "Name " + name + " has stats " + stat);
8                });
9    Mono<List<String>> result = combinations.collectList();
10   List<String> results = result.block();
11   assertThat(results).containsExactly(
12       "Name NameJoe has stats 103",
13       "Name NameBart has stats 104",
14       "Name NameHenry has stats 105",
15       "Name NameNicole has stats 106",
16       "Name NameABSLAJNFOAJNFOANFANSF has stats 121"
17   );
```

**Výpis kódu 5** Ukážka využitia objektu Mono [27]

Mono predstavuje špecializovaný prípad Flux s 0 až 1 elementom. Ide teda o 1 samostatnú hodnotu, alebo prázdny stream.

Mono je analogicky oproti objektu Flux používaný na reprezentáciu 0 až 1 výsledku.[27]

## 3.3 Quarkus

Quarkus je open-source Java framework ktorý sa primárne zameriava na tvorbu cloud-native služieb a webových aplikacií. Kľúčovou vlastnosťou tohto frameworku je, že je optimalizovaný na efektívne využívanie pamäte a značné urýchlenie spustenia aplikácie ktorá je v ňom vyvýjaná. Quarkus je navrhnutý tak, aby sa pri vývoji dal bezproblémovo kombinovať nereaktívny štýl programovania, na ktorý je drvivá väčšina programátorov zvyknutý, a neblokujúci reaktivný štýl.[28]

Quarkus pre svoje reaktívne riešenie používa programovaciu knižnicu Mutiny, ktorej kľúčové prvky si popíšeme podrobnejšie.[29]

### 3.3.1 Mutiny

Mutiny je open source knižnica v programovaciom jazyku Java na reaktívne programovanie, založená na prací s udalosťami v programe. Mutiny je integrovaná do Quarkusu,

ale funguje aj ako nezávislá knižnica, ktorú možno použiť v akejkoľvek Java aplikácií. Je založená na špecifikácií Reactive Streams a je vyvinutá tak, aby bola neblokujúca. Hlavnou vlastnosťou, ktorou sa táto knižnica snaží spríjemniť prácu s reaktívným paradigmatom, je API ktoré je založené na dvoch objektoch, Uni a Multi.[30]

### 3.3.2 Uni

Uni objekt predstavuje stream, ktorý dokáže vyprodukovať len 2 typy udalostí. Je to buď udalosť výsledku, alebo udalosť poruchy. Výsledok ktorý dostaneme síce môže byť null, ale Uni API ponuká veľa možností ako takýto prípad ošetriť. Zároveň máme vďaka tomúto API k dispozicí veľkú ponuku nástrojov na tvorbu, úpravu a organizaciú Uni prúdu. Kedže z principu je Uni lazy, čiže svôj výsledok vyprodukuje až v momente, kedy sa to od neho vyžaduje, je potrebné aby sa na tento objekt zavolala metóda Subscribe.[31] Názorná ukažka práce s Uni je na obrázku 6).

```
1    Uni.createFrom().item(1)
2    .onItem()
3    .transform(i ->"hello " + i)
4    .onItem().delayIt().by(Duration.ofMillis(100))
5    .subscribe().with(System.out::printLn);
```

**Výpis kódu 6** Ukážka práce s objektom Uni [31]

### 3.3.3 Multi

Multi je veľmi podobný objektu Uni, až na jeden rozdiel, a to že namiesto jedného výsledku, ich vie vyprodukovať viacej, pripadne nekonečný počet. Rovnako ako Uni, aj Multi obsahuje operátory na tvorbu a úpravu elementov ktoré produkuje, a rovnako je aj z principu lazy. V porovnaní s Uni, Multi nemože vyprodukovať null hodnoty. Multi sa primárne používa na pracú s kolekciami dát, a tým pádom je schopný produkovať výsledky v podobe prvkov tejto kolekcie, alebo nevraciať nič. V momente keď Multi vyprodukuje udalosť dokončenia, naznačuje nám tým, že už nie sú žiadne položky na vyprodukovanie.[32] Názorná ukažka práce s Multi je na obrázku 7).

```
1    Multi.createFrom().items(1, 2, 3, 4, 5)
2       .onItem().transform(i -> i * 2)
3       .select().first(3)
4       .onFailure().recoverWithItem(0)
5       .subscribe().with(System.out::println);
```

**Výpis kódu 7** Ukážka práce s objektom Multi [32]

## 3.4 Vert.X

Vert.x je vysokovýkonná, ľahká a reaktívna súprava nástrojov na vytváranie distribuovaných aplikácií riadených udalosťami na Java Virtual Machine (JVM). Poskytuje súbor knižníc a nástrojov, ktoré umožňujú vývojárom vytvárať asynchrónne, neblokujúce a škálovateľné aplikácie, ktoré dokážu zvládnuť veľký objem prevádzky a poskytovať výkon v reálnom čase. Vert.x je vysoko modulárny a rozšíriteľný s veľkým množstvom vstavaných modulov a rozšírení, ktoré možno použiť na pridanie funkčnosti a prispôsobenie správania aplikácie. Najdôležitejšími prvkami vo Vert.x ekosystéme su Vertx objekt, Event bus a Verticle.[23]

### 3.4.1 Vertx objekt

Vertx objekt je hlavné jadro celej aplikácie budovanej na technologií Vert.x. Tento objekt má na starosti vytváranie klientov a serverov, a sprostredkovanie Event busu.[33]

### 3.4.2 Event Bus

Event bus je spečialny objekt, vďaka ktorému sú iné časti Vert.x aplikácie schopné medzi sebou komunikovať. Tento objekt podporuje model publish/subscribe vďaka ktorému je kompatibilný so špecifikáciou Reactive Streams API.[34]

### 3.4.3 Verticle

Verticle je časť kódu ktorá je spravovaná a spúštaná Vertx objektom. Typická Vert.x aplikácia pozostáva z viacerých instancií Verticle objektu, ktoré spadajú pod rovnaký Vertx objekt, a komunikujú medzi sebou prostredníctvom Event bus.[35]

```
1  public class MyVerticle extends AbstractVerticle {
2      private HttpServer server;
3
4      public void start(Promise<Void> startPromise) {
5          server = vertx.createHttpServer().requestHandler(req -> {
6              req.response()
7                  .putHeader("content-type", "text/plain")
8                  .end("Hello from Vert.x!");
9              });
10         // Now bind the server:
11         server.listen(8080, res -> {
12             if (res.succeeded()) {
13                 startPromise.complete();
14             } else {
15                 startPromise.fail(res.cause());
16             }
17         });
18     }
19 }
```

**Výpis kódu 8** Ukážka použitia Verticle [35]

Kapitola 4

# Metriky na porovnanie technologií


## Abstract

V tejto kapitole si predstavíme metriky, podľa ktorých technológie budeme porovnávať.

Metriky ktoré sú v tejto práci použité sú z veľkej časti inšpirované publikovanou prácou *An Empirical Study of Metric-based Comparisons of Software Libraries*[36]. Každopádne, metriky sú v niektorých možnostiach poupravené, vhľadom k tomu že existujúce riešenie mi prišlo v niektorých ohľadoch nedostatočne.


## 4.1 Popularita

Programátor, ktorý sa potrebuje rozhodnúť medzi technológiami sa môže rozhodnúť podľa toho, koľko ľudí používa danú technológiu, prípadne aka veľká je komunita okolo tejto technológie. V takom prípade by sme si mohli definovať popularitu ako počet projektov, ktoré tieto technológie používajú.

## 4.2 Podpora

Pod podporou softwéru si predstavíme aktívny vývoj na danej technologií a spoľahlivý systém reklamovania a opravovania chýb. Pod túto kategóriu spadajú nasledovné metriky.

### 4.2.1 Frekvencia vydaní

Dôležitým faktorom pre výber správnej technológie pre vývojara je fakt, že na danej technologií sa aktívne pracuje z hľadiská pridania nových funkcií a opravy chýb. Dôsledkom

toho si definujeme metriku frekvencie vydaní ako priemerný čas medzi dvoma po sebe idúcimi vydaniami knižnice/frameworku.

### 4.2.2 Čas otvorenia záznamu chyby a čas ukončenia opravy chyby

Nevyhnutnou vlastnosťou pre potencionalného klienta môže byť poznatok, či vývojári danej knižnice sú nápomocní, z hľadiska spätnej odozvy na nahlásenie chyby. Jedním kvantifikovateľným spôsobom, ako to tento jav overiť, je čas, ktorý udáva, ako rýchlo sa na nahlásené problémy odpovedá a ako rýchlo sa riešia. V rámci tejto metriky si definujeme čas otvorenia záznamu chyby, ako priemerný čas meraný od počiatku vytvorenia záznamu chyby až po jej prvú reakciu vo forme komentára. Podobne, čas ukončenia opravy chyby definujeme ako priemerný čas, ktorý trval od prvého otvorenia chyby až po jej ukončenie.

### 4.2.3 Posledný dátum modifikácie

Posledný dátum modifikácie knižnice alebo frameworku nám udáva časový údaj, kedy naposledy bola daná technológia upravovaná. Zatiaľ čo frekvencia vydaní nám poskytuje odhad, ako často sa softwér aktualizuje, neposkytuje informácie o aktuálnosti softwéru. Tým pádom nám dátum modifikácie uvádza, či je vývoj na technologií stále aktívny a aktuálny.

## 4.3 Výkonnosť

To ako rýchlo sa aplikácia správa, a ako efektívne pracuje s prostriedkami počítača je kľúčový dôvod prečo by si klient zvolil danú aplikáciu oproti iným. Hlavné metriky na ktoré sa zameriame sú:

- Priemerný čas odozvy požiadavky
- Využitie procesoru
- Využitie pamäte

Z akademických prác z ktorých som čerpal pri hľadaní vhodných testov na meranie výkonnosti som sa zamerial na prácu Reactive vs Non-Reactive Java framework od André Nordlund a Niklas Nordström[37]. V tejto práci si autori definovali 4 testy, vďaka ktorým za pomoci programu JMeter a VisualVM boli schopný zozbierať vyžadované metriky na porovnanie efektívnosti programov. Rovnako aj v tejto práci budú navrhnuté 4 testy prostredníctvom ktorých nameriame efektívnosť aplikacií.

# Kapitola 5
# Meranie metrík


## Abstract

V tejto kapitole si detailne popíšeme, akými nástrojmi a spôsobmi budeme merať metriky, ktoré boli definované v predchádzajúcej kapitole.


## 5.1 Popularita

V predchádzajúcej kapitole sme si definovali populariu technológie ako počet projektov ktorá danú technológiu využívajú. Vzhľadom k tomu, že spomínané technológie su z veľkej častí používané na veľkých komerčných projektoch, ktoré svoje zdrojové kódy maju privátne chránené, by jednoduché vyhľadávanie na internete podľa môjho názoru veľmi nepresne skreslilo tieto čísla. Práve preto, aby výsledky v tejto práci čo najpresnejšie odrážali realitu, sa budeme odkazovať na výsledky prieskumov popularity webových frameworkov a popularite na platforme GitHub, na ktorej sú zdrojové kódy technologií hosťované. Ako hrubý odhad vo výsledkoch uvedieme aspoň 2 prieskumy, vykonané legitímnimy korporáciami a nakoniec, počet hviezd na jednotlivých repozitároch technologií na Githube.

Spomínané údaje síce majú menšiu vernostnú hodnotu ako reálny počet projektov, ktorý tieto technológie využívajú, každopádne prišlo mi dôležité ich doplniť, vzhľadom k tomu, že jemne naznačujú aká veľka je komunita okolo týchto technologií.

## 5.2 Podpora

Na nameranie metrík v kategórií podpory použijeme skripty LibraryMetricScripts [38] ktoré vznikli v rámci práce An Empirical Study of Metric-based Comparisons of Software Libraries [36]. Spôsob spustenia skriptov a načítania výsledkov je definovaný na repozitári,

kde su skripty uložené, preto sa v nasledujúcich kapitolách budeme zameriavať primárne na výkonnosť a z popularity zhodnotíme len výsledky.

## 5.3 Výkonnosť

Po vzoru práce Reactive vs Non-Reactive Java framework [37] sa na nameranie priemerného času odozvy požiadavky použije program JMeter, a na sledovanie využívania pamäte a procesoru sa použije nástroj VisualVM.

## 5.4 JMeter

JMeter je open-source Java aplikácia určená na testovanie funkčného správania a meranie výkonu. JMeter možno použiť na testovanie statických aj dynamických zdrojov a dokáže testovať rôzne protokoly ako HTTP, SOAP a REST. JMeter vykonáva testy na základe testovacích plánov. Tieto testovacie plány môžu zahŕňať rôzne položky, ako sú skupiny vlákien na simuláciu používateľov alebo poslucháčov na uloženie výsledkov z príkladov HTTP požiadaviek.[39]

### 5.4.1 Thread group

Thread group je najdôležitejším prvkom testovacích plánov programu JMeter, pretože riadí všetky ostatné funkcie pre záťažové testovanie. V rámci thread group je možné si definovať nastavenia pre počet vlákien, ktoré budú vykonávať nejaký záťažový test. V thread group, každé vlákno spustí testovací plán a vykoná ho bez ohľadu na iné vlákna.[40]

### 5.4.2 Sampler

Sampler slúži na to, aby posielal požiadavky na server a čakal na odpoveď. V prípade že máme definovaných viac typov sampler, tak sa spracúvajú v poradí, v akom sa zobrazujú v strome konfigurácie. [40]

### 5.4.3 Summary report

Aby bolo možné uložiť údaje zhromaždené samplermi, je v pláne testov potrebné nastaviť zapís výsledkov. Na to slúži summary report. Summary report je jedným z najjednoduchších poslucháčov v programe JMeter a obsahuje parametre na meranie HTTP požiadaviek a porovnávanie výkonov medzi API.[41]

## 5.5 VisualVM

VisualVM je nástroj, ktorý poskytuje vizuálne rozhranie na podrobné prezeranie informácií o aplikáciách založených na technológii Java, ktoré su spustené v JVM. Zobrazené informácie zahŕňajú, ale nie sú obmedzené na využitie haldy a CPU, spustené vlákna a načítané triedy.[42]

### 5.5.1 Snapshot

Hneď ako sa vytvorí spojenie s JVM aplikáciou, VisualVM začne merať a zobrazovať rôzne informácie. Na zaznamenia týchto hodnôt môže VisualVM vytvoriť snímku informácií, zvanú snapshot. Tento snapshot obsahuje informácie prevzaté od založenia spojenia s aplikáciou, až po vytvorenie samotného snapshotu. Snapshot zobrazuje rovnaké informácie, aké možno vidieť pri monitorovaní aplikácie v reálnom čase.[43]

Kapitola 6

# Testovací scénar merania výkonnosti


## Abstract

V predchádzajúcej kapitole sme opísali nástroje ktoré budu použité na meranie metrík. Vzhľadom k tomu, že meranie výkonnosti a s ním zvolené nástroje si vyžadujú komplexnejšiu formu testovania, si v tejto kapitole definujeme testy, na základe ktorých vzniknú vzorové aplikácie implementované v jednotlivých technológiach na testovanie týchto metrík.


## 6.1 Motivácia

Cieľom testovania výkonnosti serverovej aplikácie by v najlepšom prípade mala byť simulácia zaťaženia serveru požiadavkami. Kedže reaktívne programovanie v oblasti serverových aplikacií sa z principu snaží riešiť problematiku zefektívnenia réžie obsluhy servera, pre meranie výkonu by bolo adkvevátne vytvoriť aplikácie, ktoré by podstúpili testy zaťaženia. Výsledkom týchto testov by boli dáta, ktoré by vedeli bližšie priblížiť, ktorá z našich zvolených technologií je v tejto oblasti lepšia. Pre jednoduchosť si predstavme, že máme primitívnu serverovú aplikáciu s REST rozhraním. Pre túto aplikáciu si navrhneme nasledujúce 4 testy.

- Test malej skupiny uživateľov: 2 endpointy s malou skupinou uživateľov
- Test väčších dát: 2 endpointy s väčšiou skupinou uživateľov, a zároveň možnosťou požiadať o vačšie množstvo dát
- Test úpravy a kombinácie: 2 endpointy s vačšíou skupinou uživateľov a možnosťou úpravy dát,

- Test veľkej skupiny uživateľov: 2 endpointy s nadmerným množstvom uživateľov

```
--Test malej skupiny uživateľov
GET /person/{id}, GET /animal/{id}
--Test väčších dát
GET /person/{id}, GET /person/{id}
GET /person, GET /animal
--Test úpravy a kombinácie
PUT /person, GET /group{id}
GET person/{id}, GET animal/{id}
--Test veľkej skupiny uživateľov
GET person/{id}, GET person/{id}
GET animal/{id}, GET animal/{id}
```

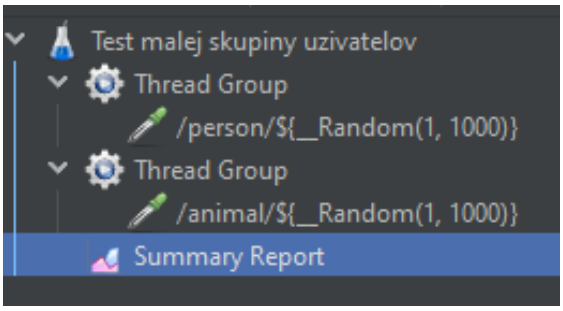
**Výpis kódu 9** Ukážka zjednodušenej konfigurácie JMeter testov

Vzhľadom k tomu, že na meranie výkonnosti použijeme JMeter, si tieto testy bližšie špecifikujeme v konfiguráciach, ktoré budu použité pri spustení testov.

## 6.2 Test malej skupiny uživateľov

V teste na malú skupinku uživateľov budú vytvorené 2 thread group. Každá thread group bude posielať GET dotaz na 1 endpoint za účeľom získania práve 1 objektu. V rámci jednej group bude zadaných 250 uživateľov, ktorý sa budú dotazovať pomocou GET na práve 1 náhodne vybraný objekt. Náhodna premenná, v tomto prípade id, je definovaná v rámci JMeter konfigurácie ako náhodná premenná. Tento test bude trvať 1 minútu.

**Obr. 6.1** Konfigurácia testu malej skupiny uživateľov

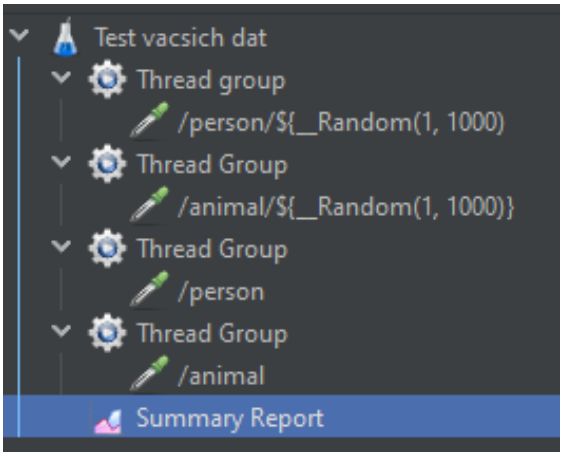



## 6.3 Test väčších dat

Test s väčším množstvom dat bude okrem bežného dotazovania merať aj výkon pri dotaze na väčší objem dát. V tomto teste si definujeme 4 thread group. 2 thread group sa budú dotazovať na 1 endpoint a 1 objekt, rovnako ako v predošlom teste, a zvýšné sa budú

dotazovať na všetky objekty v rámci príslušného endpointu. Tento test bude trvať 1 minútu. Veľkosť dát ktorá sa bude vraciať z aplikácie bude bližšie popísana v nasledujúcej kapitole, v sekcií databáza. Všetky groupy maju zadaný počet uživateľov na 250.

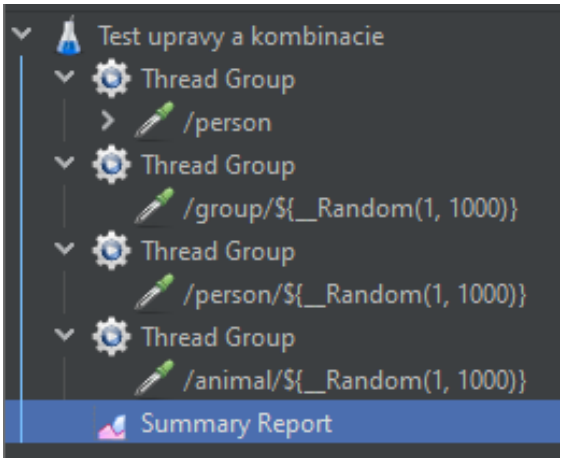
**Obr. 6.2** Konfigurácia testu väčších dat




## 6.4 Test úpravy a kombinácie

Pri teste na úpravu a kombináciu využijeme 4 thread group. 2 thread group po 500 uživateľov sa budú dotazovať na 1 endpoint s cieľom získať 1 objekt. 1 thread group s 250 uživateľmi bude mať za úlohu volať PUT požiadavku, ktorá bude na zvolenom endpointe vyžadovať úpravu objektu. Posledná thread group sa bude dotazovať na endpoint, na ktorom sa na pozadí bude volať dotaz na databázu, ktorý skombinuje 2 objekty, a vráti ich ako odpoveď. Tento test bude trvať 2 minúty.

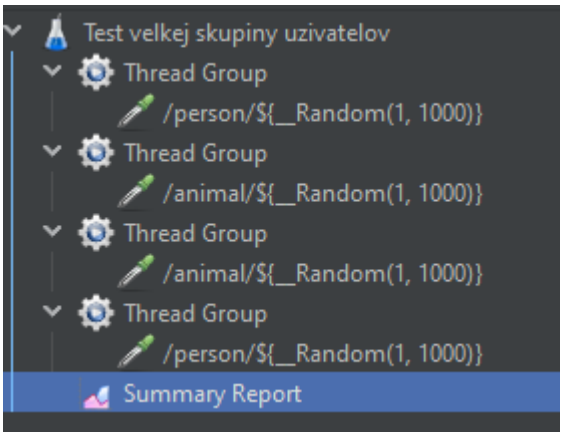
**Obr. 6.3** Konfigurácia testu úpravy a kombinácie



## 6.5 Test veľkej skupiny uživateľov

Na test s veľkým množstvom uživateľov si definujeme 4 thread group po 500 uživateľov, kde každá polovica sa bude dotazovať na samostaný endpoint. Tento test bude trvať 2 minúty.

**Obr. 6.4** Konfigurácia testu veľkej skupiny uživateľov



Kapitola 7

# Prototypy na meranie výkonnosti


## Abstract

V tejto kapitole si popíšeme proces tvorby prototypov, ktoré boli použité na meranie výkonnosti. Zároveň si uviedieme spôsob, akým boli testy na výkon spúšťané voči jednotlivým aplikáciam.


## 7.1 Prototyp

Prototypy su navrhnuté ako serverové aplikácie, ktoré poskytujú REST API, na ktoré sa dá za pomoci HTTP protokolu dotazovať. REST API pozostáva z entít Person a Animal. Ide o jednoduché entity ktoré su uložené v databáze, pričom aplikácia pri dotazovaní tieto objekty vracia, a žiadne iné spracovanie nenastáva. Dôvod prečo sú aplikácie navrhnuté takto jednoducho je, aby pri meraní výkonnosti boli rozdieli čo najmenšie. Každá aplikácia ponúka rovnaké rozhranie a rovnakú funkcionalitu, tým pádom sa zameriame len na ich efektívnosť. Výnimočne si definujeme ešte entitu Group, ktorá predstavuje kombináciu entít Person a Animal. Táto entita sa neeviduje v databáze. Jej úloha v aplikacií je simulovať činnosť, pri ktorej server na pozadí musí vykonať 2 dotazy na vrátenie objektu.

| Absolútna cesta | HTTP metóda |
|---|---|
| /person | GET |
| /person | POST |
| /person/{id} | PUT |
| /person/{id} | GET |
| /person/{id} | DELETE |
| /animal | GET |
| /animal | POST |
| /animal/{id} | PUT |
| /animal/{id} | GET |
| /animal/{id} | DELETE |
| /group/{id} | GET |

**Tabuľka 7.1** Endpointy dostupné v rámci každého prototypu

## 7.2 Testovacie prostredie

Hardware, na ktorom testi prebiehali je notebook, a to konkrétne HP EliteBook 745 G5 Notebook [44], s nasledujúcimi špecifikáciami.

- 16 GB RAM
- AMD Ryzen 7 PRO 2700U w/ Radeon Vega Mobile Gfx 2.20 GHz
- 237 GB Storage

## 7.3 Databáza

Na perzistenciu dát bola použitá databáza PostgreSQL. Každá serverová aplikácia, respektíve, každý prototyp mal vlastnú priradenú databázu, v ktorej boli definované 2 entity, Person a Animal, znázornené na obrázku 7.1.Na tvorbu tabuľiek a vytvorenie testovacích dát bol použitý skript na ukážke 10. Každá vygenerovaná tabuľka mala 1000 záznamov.

**Obr. 7.1** Entity objekty Person a Animal

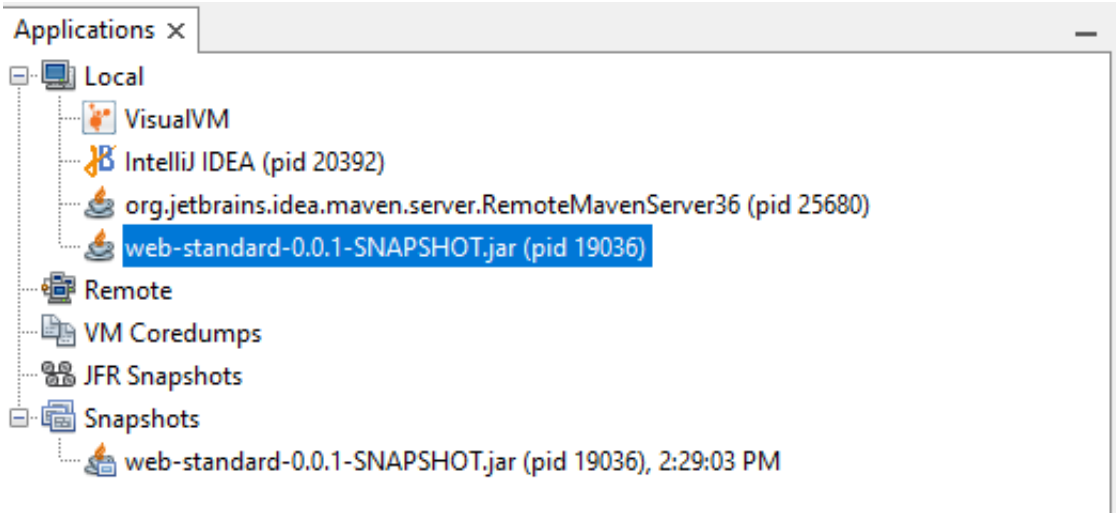
```
drop table if exists person cascade;
drop table if exists animal cascade;

create table person(
                    id bigserial Primary Key,
                    age integer,
                    name varchar(255)
);

create table animal(
                    id bigserial Primary Key,
                    name varchar(255)
);

INSERT INTO person(age, name)
SELECT random()::numeric, md5(random()::text)
FROM generate_series(1, 1000) id;

INSERT INTO animal(name)
SELECT md5(random()::text)
FROM generate_series(1, 1000) id;
```

**Výpis kódu 10** Vytvorenie entít Person a Animal

## 7.4 VisualVM

Keď aplikácie boli pripravené a spustené, tak ešte pred samotným spustením JMeter testu bolo potrebné nastaviť VisualVM. Toto nastavenie prebiehalo manuálne skrz grafické rozhranie. V momente ked aplikácia bola spustená, tak v rozhraní applications bolo možné sa nastaviť na sledovanie danej aplikácie. Následne, v zálozke sampler stačilo kliknúť tlačítko CPU, vďaka ktorému nastalo sledovanie metrík. Po dokončení testu sa vytvoril snapshot, ktorý zaznamenal využitie procesoru a pamäte počas obdobia vykonávania testu. Ukážka VisualVM je na obrázku 7.2.

**Obr. 7.2** VisualVM - ukážka snapshotu




## 7.5 JMeter

Testy popísane v predošlej kapitole su nastavené prostredníctvom JMeter GUI programu a uložené do príslušných súborov Test1.jmx až Test4.jmx. Testy su spustené cez prikazový riadok, a výsledok testu je zapísany do súboru output.csv. Následne, výsledky z output.csv sú exportované do priečinku report_output, v ktorom sa vygeneruje summary report obsahujúci grafy s výsledkami. Na ukážke 11 je znázorené spusteniu testu z príkazového riadku.

```
1    -- Spustenie testu
2    ./jmeter -n -t tests/Test1.jmx -f -l output.csv
3
4    -- Po dokončení testu
5    ./jmeter -g output.csv -f -o report_output
```

**Výpis kódu 11** Spustenie testu malej skupiny

# Kapitola 8
# Výsledky


## Abstract

V tejto kapitole si zhrnieme výsledky meraní a testov a v rámci diskusie usúdime, ktorá technológia z toho vychádza najlepšie.


## 8.1 Popularita

V tabuľke 8.1 je uvedený počet hviezd ktoré majú jednotlivé repozitáre na Githube. Na meranie prieskumov sa odkazujem spätne na 3.2 a 3.3 ktoré boli uvedené v kapitole Reaktívne technológie v Jave.

| Quarkus | Spring | Vert.x |
|---|---|---|
| 12702 | 54083 | 13903 |

**Tabuľka 8.1** Počet hviezd na repozitorári Github

## 8.2 Podpora

V nasledujúcich podkapitolách su vypisané výsledky meraní jednotliv ých metrík.

### 8.2.1 Frekvencia vydaní

Nasledujúca tabuľka obsahuje výsledky spočítania priemerného času medzi vydaniami pre jednotlivé technológie, vyjadrené v dňoch. Na získanie výsledka bol použitý skript releasefrequency.py[38].

| Quarkus | Spring | Vert.x |
|---|---|---|
| 6 | 12 | 27 |

**Tabuľka 8.2** Výsledok merania frekvencie vydaní

### 8.2.2 Priemerný čas otvorenia záznamu chyby a priemerný čas ukončenia opravy chyby

V tabuľke 8.3 je vypísaný priemerný čas otvorenia záznamu chyby, a na tabuľke 8.4 je priemerný čas ukončenia opravy chyby. Obydve metriky boli merané v počte dní so zaokruhlením na 2 desatinné miesta.Na získanie výsledka bol použitý skript issues.py[38].

| Quarkus | Spring | Vert.x |
|---|---|---|
| 0,51 | 0,41 | 0,53 |

**Tabuľka 8.3** Priemerný čas otvorenia záznamu chyby na Github repozitári

| Quarkus | Spring | Vert.x |
|---|---|---|
| 0,95 | 0,78 | 0,97 |

**Tabuľka 8.4** Priemerný čas ukončenia opravy chyby na Github repozitári

### 8.2.3 Posledný dátum modifikácie

Posledný dátum modifikácie bol meraný 8.1.2024. V ten deň boli vo všetkých repozitároch meraných technologií vykonané zmeny v hlavnej vetve.Na získanie výsledka bol použitý skript lastmodificationdate.py[38].

## 8.3 Výkonnosť

### 8.3.1 Priemerný čas odozvy

Tabuľka 8.1 obsahuje číselné údaje priemerného času odozvy požiadaviek na serverovú aplikáciu, ktoré boli nameráne pri jednotlivých vykonnostných testoch. Jednotky času su vyjadrené v milisekundách. Na grafoch v prilohe C, od C.1 až po C.9 sú zobrazené časi požiadaviek vykonaných v jednotlivých testoch.

### 8.3.2 Využitie procesoru

Na obrázkoch v prílohe dotatku A, od A.1 až po A.12, sú zobrazené výsledky merania využitia procesoru pri jednotlivých testoch.

**Obr. 8.1** Priemerný čas odozvy požiadavky na server

| | Quarkus | Spring | Vert.x |
|---|---|---|---|
| **Test malej skupiny užívateľov** | 1818 ms | 461 ms | 566 ms |
| **Test väčších dát** | 32985 ms | 18994 ms | 11410 ms |
| **Test úpravy a kombinácie** | 2665 ms | 1616 ms | 820 ms |
| **Test veľkej skupiny užívateľov** | 6514 ms | 1817 ms | 1860 ms |

### 8.3.3 Využitie pamäte

Na obrázkoch v prílohe dotatku B, od B.1 až po B.12, sú zobrazené výsledky merania využitia pamäte pri jednotlivých testoch.

# Kapitola 9
# Porovnanie a diskusia


## Abstract

V tejto kapitole si prebereme výsledky meraní a zanalyzujeme, v akých oblastiach vynikajú jednotlivé technológie.


## 9.1 Popularita

Z hľadiska popularity neni moc prekvapujúce, že spomedzi trojice kandidátov vychádza najlepšie framework Spring, konkrétne jeho riešenie Webflux. Vo svete budovania serverových aplikacií v Jave je Spring jednoznačne najviac zaužívaný, a to sa jasne odráža, či už na prieskumoch od Java developerov z komerčného prostredia, ale aj na internetovej komunite ktorá ku vývoju Springu neustále prispieva.

## 9.2 Podpora

Na všetkych troch technológiach sa voči dnešnému dátumu stále aktívne pracuje a vyvíja. V jednotlivých metrikach definovaných v rámci tejto kategórie su odhalené len drobné rozdiely.

### 9.2.1 Frekvencia vydaní

Z merania priemerného času medzi vydaniami to vychádza veľmi tesne, každopádne Quarkus z týchto čísel vychádza ako víťaz. Je to do značnej mieri kvôli tomu, že spätne podporuje viaceré verzie, tým pádom aktivita vydaní na Github repozitáry je oveľa častejšia. V konečnom dôsledku by som uviedol, že vzhľadom k tomu že ide o tak malý rozdiel medzi priemerným časom vydaní jednotlivých technologií, sú si v rámci tejto metriky všetky 3 technológie rovné.

### 9.2.2 Priemerny čas otvorenia záznamu chyby a priemerny čas ukončenia opravy chyby

Bez zbytočného opakovania, aj v pripade tejto metriky sme v rovnakej situacií, ako pri frekvencií vydaní. Všetky 3 technológie maju veľmi aktívne komunity a vývojové týmy. Z výsledkov vyplýva, že pri reklamovaní chyby je veľmi pravdepodobné, že opravy si zobere na starosti nejaký developer ešte v ten deň. Rovnako, aj v tejto metrike sú si skúmane technológie skoro rovné.

### 9.2.3 Posledný dátum modifikácie

Ku dátumu vypracovaniu tejto prace sa na všetkých 3 technológiach stále aktívne pracuje.

## 9.3 Výkonnosť

Dôležitý fakt, ktorý treba spomenúť je, že VisualVM neposkytuje možnosť získať priemernú hodnotu využívania procesoru alebo pamäte. Tým pádom závery v nasledujúcich sekciach sú úsudkom z obrázkov v dodatkoch A, B a C.

### 9.3.1 Priemerný čas odozvy

Ešte pred tým ako prejdeme na záver tohto zhodnotenia, je dôležité spomenúť komplikácie, ktoré vznikali pri testovaní prototypov jednotlivých technologií. Pri teste úpravy a kombinácie, a teste veľkej skupiny uživateľov, sa častokrát nespracovali požiadavky. Server hlásil jednotlivé chyby a niektore požiadavky boli odmietnuté a dôsledkom toho sa polovica z nich označila ako neúspešná. Tieto situácie je názorne vidieť na obrázkoch C.2 a C.6. Tieto chyby nastali primárne pri prototypoch Spring Webflux a Quarkus. Bohužial, pri vývoji sa mi nepodarilo zistiť príčinu tejto chyby, ale čiatočne odhadujem, že chybu robila požiadavka, ktorá si volala všetky objekty na pridelenom endpointe príslušnej thread-group. V takom prípade by bol do budúcnosti lepší test, ktorý by implementoval stránkovanie, za účeľom realistického provozu serverovej aplikácie. Každopádne, je to ľen odhad, a prišlo mi dôležité to spomenúť vzhladom k tomu, že to nemusí vyslovene znamenať, že tieto technológie su v tejto oblasti neefektívne.

Každopádne, z výsledkov zvyšných testov nám z toho vychádza najlepšie technológia Vert.x. Zo začiatku sa zdalo, že Spring si vedel s paralelizmom efektívne poradiť, no pri narastajúcej komplexite testov sa zvyšovala aj réžia požiadáviek, a z výsledkov v tabuľke 8.1 sa najlepšie prejavil Vert.x. Do istej miery by sa dalo povedať, že Quarkus zabere druhé miesto v tejto disciplíne, no na druhú stranu, ako bolo už spomínane v predchádzajúcom odstavci, Quarkus a Spring sa v určitých sitáuciach chovali nepredvídateľne, a preto tieto testy nemuseli plne odrážať potenciál týchto technologií.

### 9.3.2 Využitie procesoru

Čo sa týka využívania procesoru, Vertx a Quarkus svojimi hodnotami priemerne dosahovali 30% až 40%. Spring mal tieto hodnoty oveľa vačšie. Viackrát prekročil hranicu 50%,

ako je možné vidieť na prilohach merania Springu, konkrétne na A.6 a A.7. Z toho jasne vyplýva záver, že Vertx a aj Quarkus sú na procesor menej náročné, ako Spring.

### 9.3.3 Priemerné využitie pamäte

Rovnako, ako aj pri využívaní procesoru, sa Vert.x prejavil ako efektívnejšie riešenie, spomedzi všetkých troch technologií. Pri teste s väčšími datami sa hodnota haldy vedela dostať na hranicu 400 MB, no v ostatných prípadoch sa mu darilo značne lepšie ako Springu alebo Quarkusu. Ku rozhodnutiu bol kľučový výsledok testu veľkej skupiny uživateľov. B.10

## 9.4 Zhodnotenie na koniec

Není triviálne zhodnotiť definitivný záver, ktorá technológia je najlepšia. Z hľadiska popularity, všetky 3 medzi sebou majú aktívne vývojové týmy a sú aktuálne podporované. Čo sa týka výkonnosti, tak z výsledkov, ktoré tu boli priložené, sa dá usúdiť , že v kategorií efektivity, z toho vychádza najlepšie Vert.x. Vert.x ekosystém ma veľmi explicitný prístup ku návrhu serverových aplikacií narozdiel od niečoho ako Spring, ktorý dominuje v abstrakcií. Na druhú stranu, Spring je oveľa viacej zaužívaný v komerčnej sfére, či už kvoli pestrej ponuke dokumentácie a ukážok kódu. Quarkus sa ukázal ako stred medzi týmito dvoma technológiami, z hľadiska efektívnosti a svojej popularity. Na záver tohto porovnania by som vyslovil verdikt, že pokiaľ ide developerovi v prvom rade o vysoký výkonnosť, tak Vert.x je ideálny kandidát spomedzi skúmanej trojice. Na druhú stranu, pokiaľ by developer dôveroval viacej osvečenému riešeniu, tak jasnú voľbu by predstavoval Spring.

# Záver

Cieľom tejto práce bol popis reaktívneho programovania, popis technologií Spring Webflux, Vert.x a Quarkus Reactive a ich porovnanie v rámci vybraných metrík.

V teoretickej časti sme okrem samotného paradigmatu opisovali aj špecifikáciu Reactive Streams, ktorá nám umožňuje programovať reaktívne v jazyku Java, a zároveň sme spomenuli aj dokument Reactive Manifesto, ktorý bol kľučový faktorom ku vzniku moderného reaktívneho programovania.

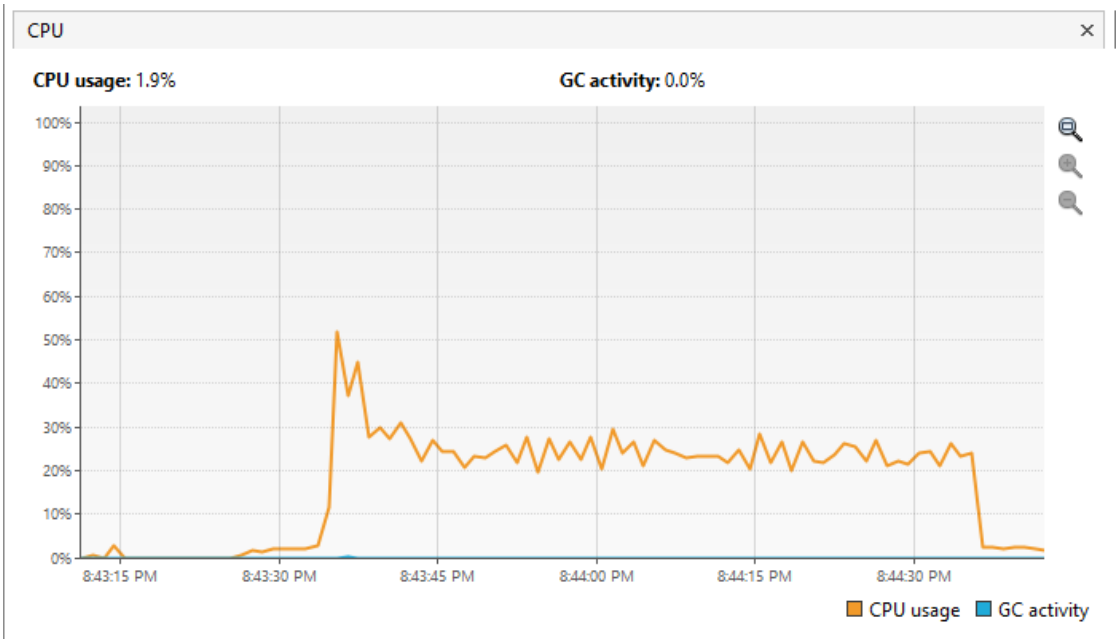
V praktickej časti sme si predstavili technologie Spring Webflux, Vert.x a Quarkus Reactive. Pri jednotlivých sme si vysvetlili princip ich reaktívneho správania. Po predstavení nasledoval výber vhodných metrík na porovnanie týchto technologií. Z týchto metrík sme sa zamerali na metriky výkonu, pričom sme navrhli testy na základe ktorých sme boli schopný adekvátne porovnať efektivitu jednotlivých technologií. Na záver sme všetky výsledky zhrnuli a navzájom ich porovnali.

V rámci tejto práce by bolo možné pridať ďalšie technológie na porovnanie. Zároveň je priestor pre úpravu a zlepšenie existujúcich metrík na porovnanie technologií. Táto praca sa môže rozšíriť o vzorovú aplikáciu ktorá by bola rozsahom väčšia a zároveň by bola navrhnutá podľa architektúry mikroslužieb, za účeľom praktickejšej ukážky reaktivných technologií. Zároveň by bolo možné využiť inú sadu nástrojov na nameranie metrík.

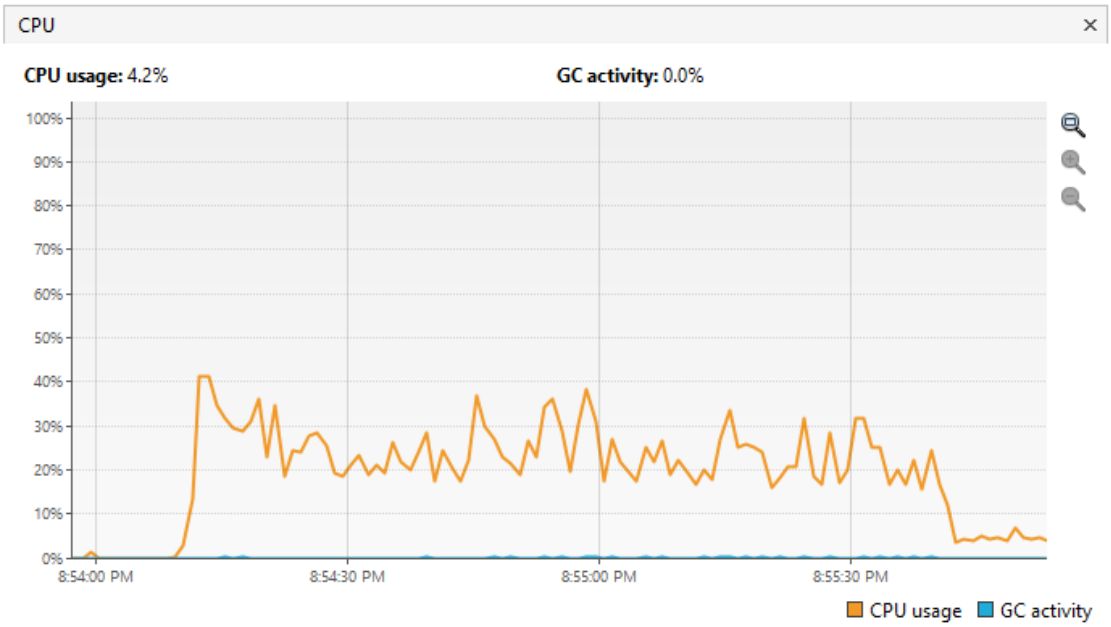
Dodatok A

# Priložené výsledky merania využitia procesoru

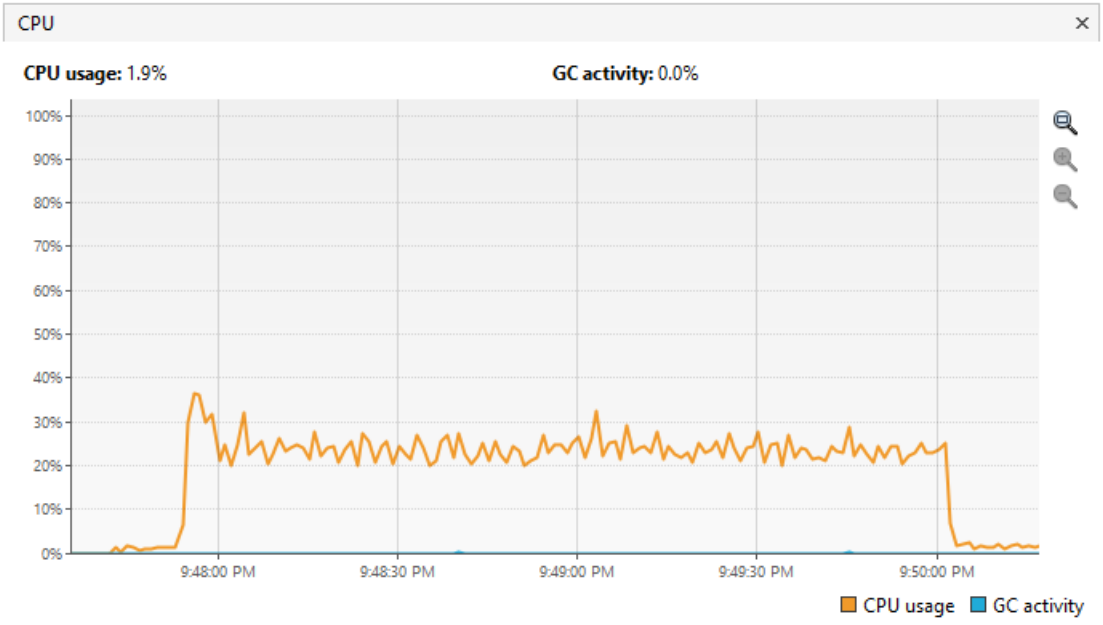
**Obr. A.1** Využitie procesoru Quarkus Aplikácie - Test malej skupiny užívateľov



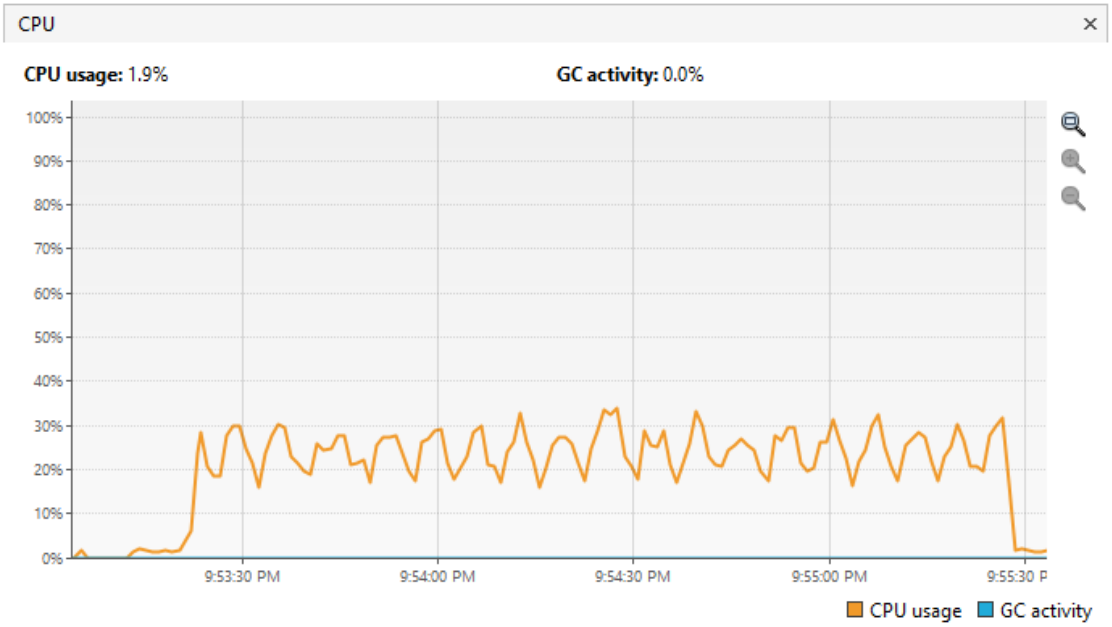
**Obr. A.2** Využitie procesoru Quarkus Aplikácie - Test väčších dát



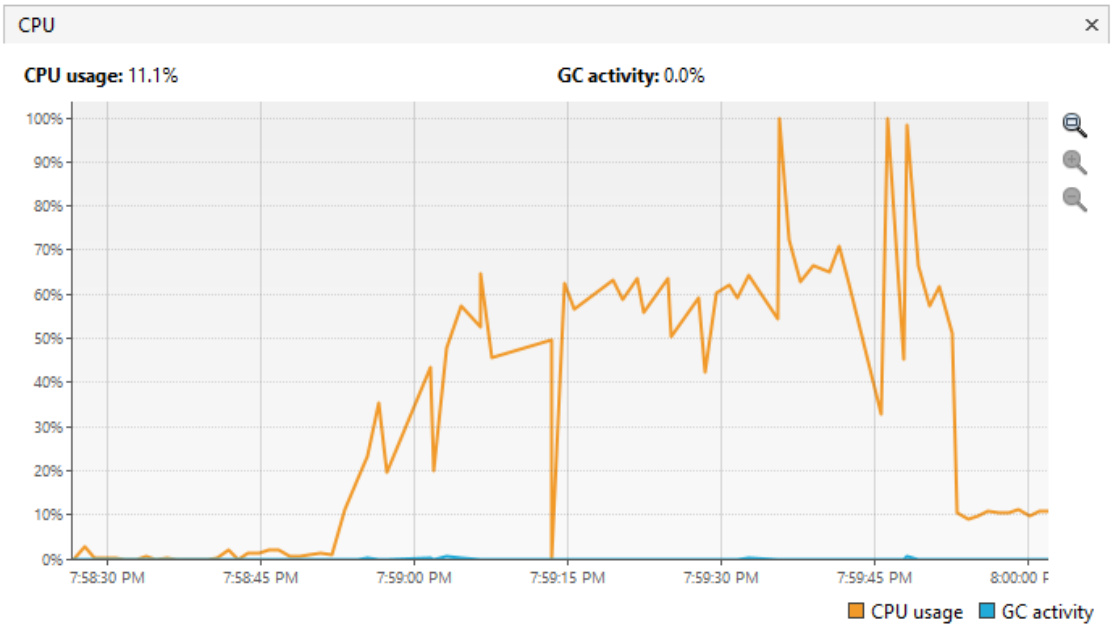

**Obr. A.3** Využitie procesoru Quarkus Aplikácie - Test úpravy a kombinácie



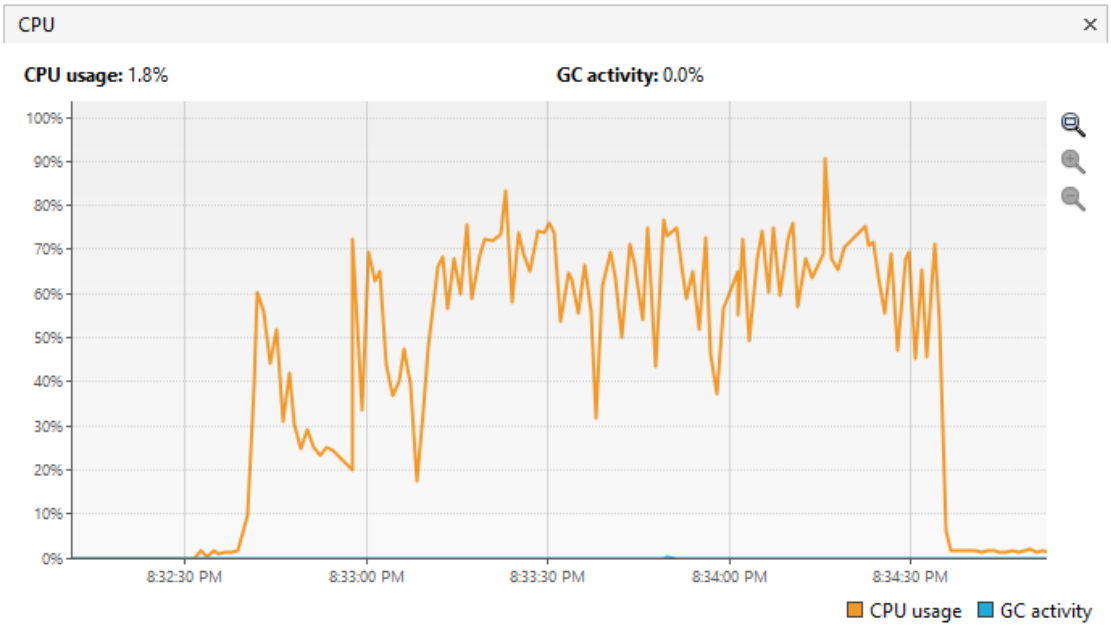
**Obr. A.4** Využitie procesoru Quarkus Aplikácie - Test veľkej skupiny užívateľov



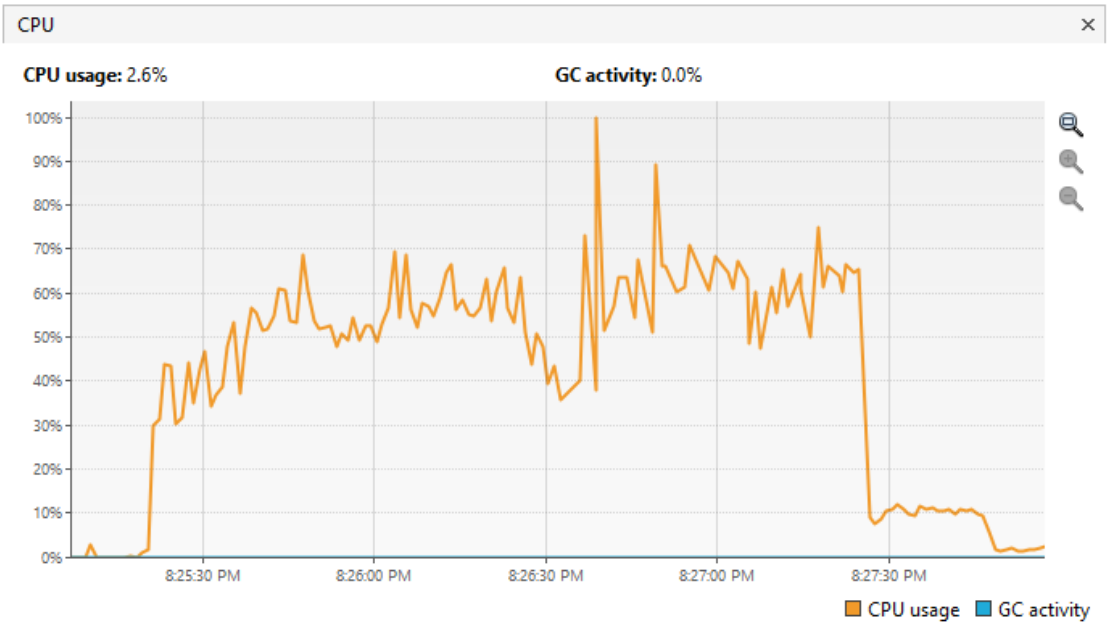

**Obr. A.5** Využitie procesoru Spring Aplikácie - Test malej skupiny užívateľov



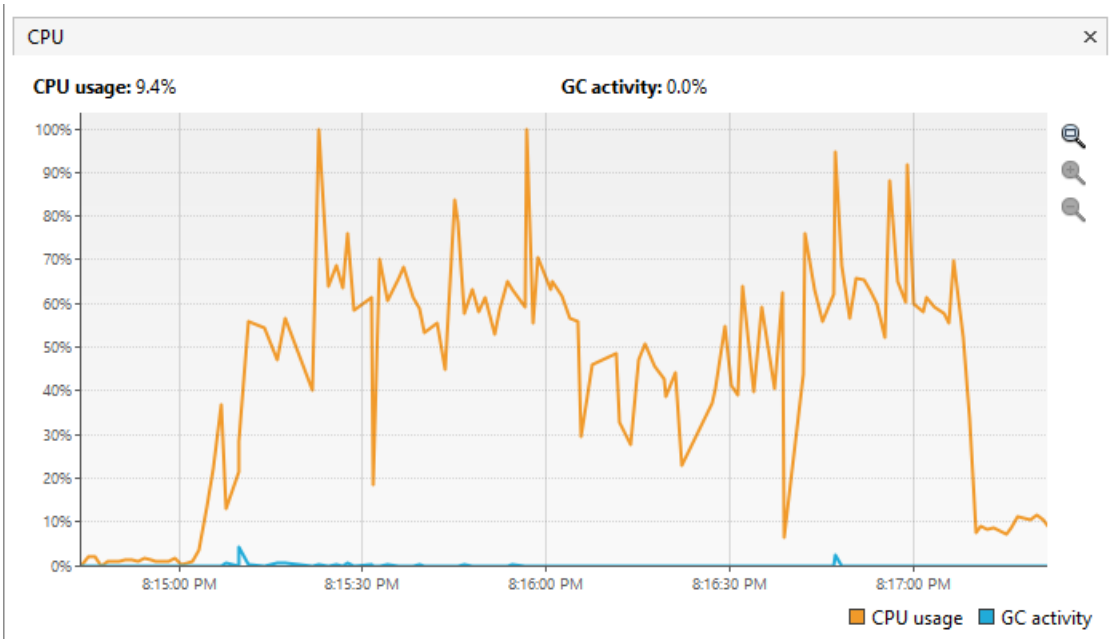
**Obr. A.6** Využitie procesoru Spring Aplikácie - Test väčších dát



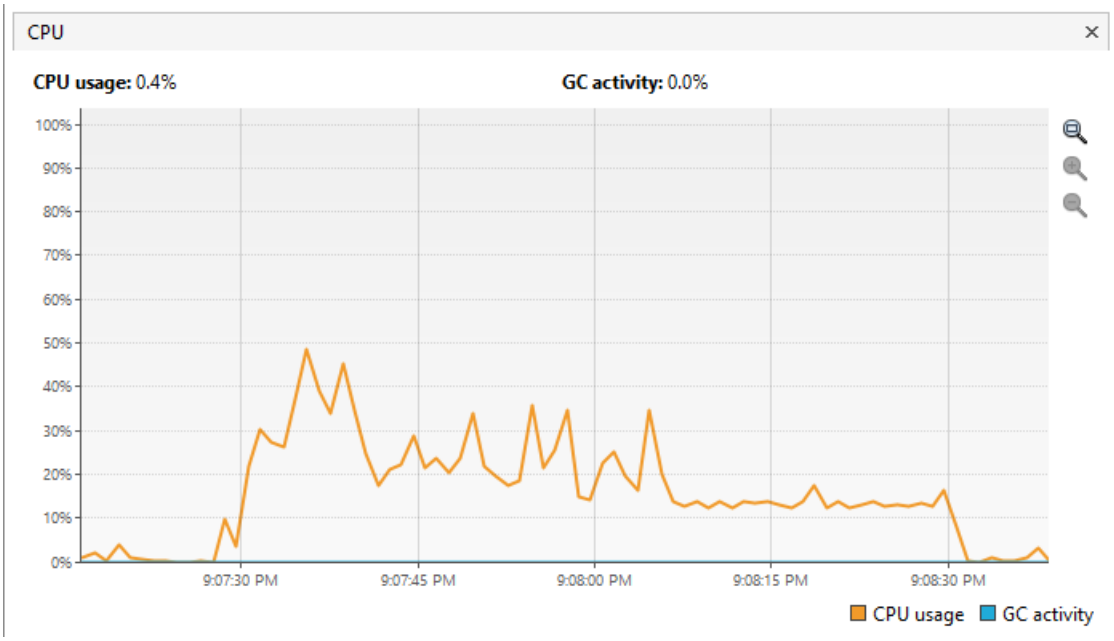

**Obr. A.7** Využitie procesoru Spring Aplikácie - Test úpravy a kombinácie



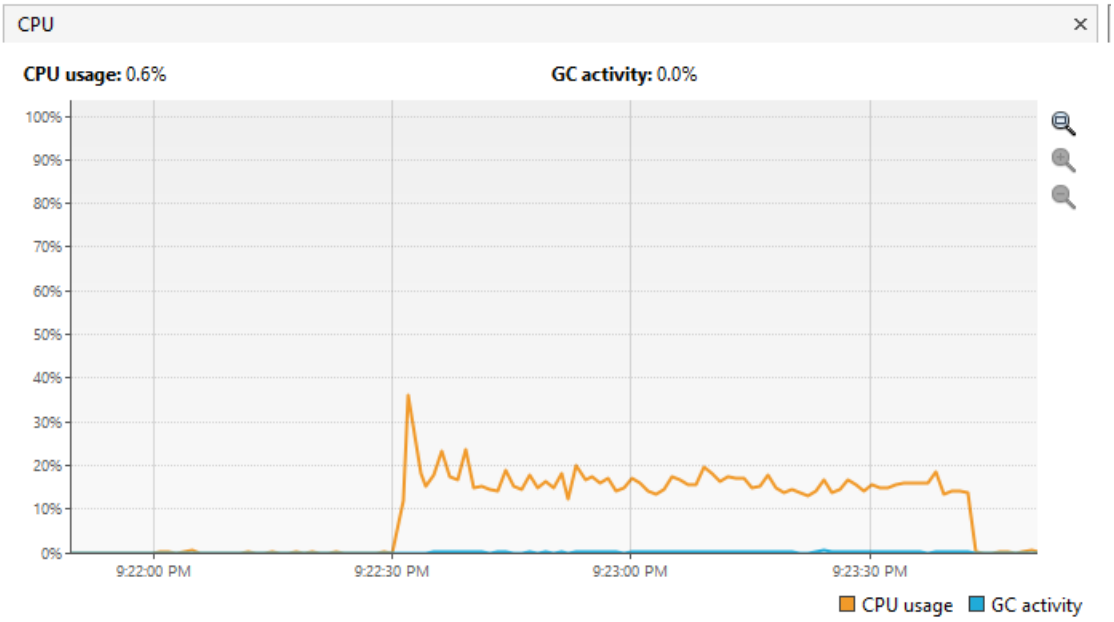
**Obr. A.8** Využitie procesoru Spring Aplikácie - Test veľkej skupiny užívateľov



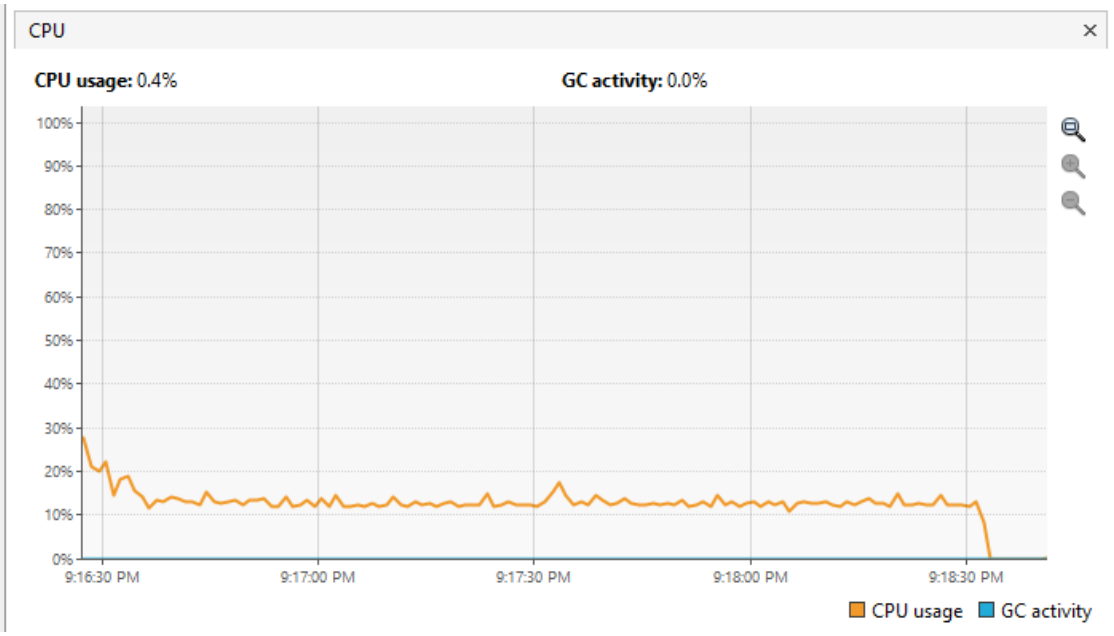

**Obr. A.9** Využitie procesoru Vert.x Aplikácie - Test malej skupiny užívateľov



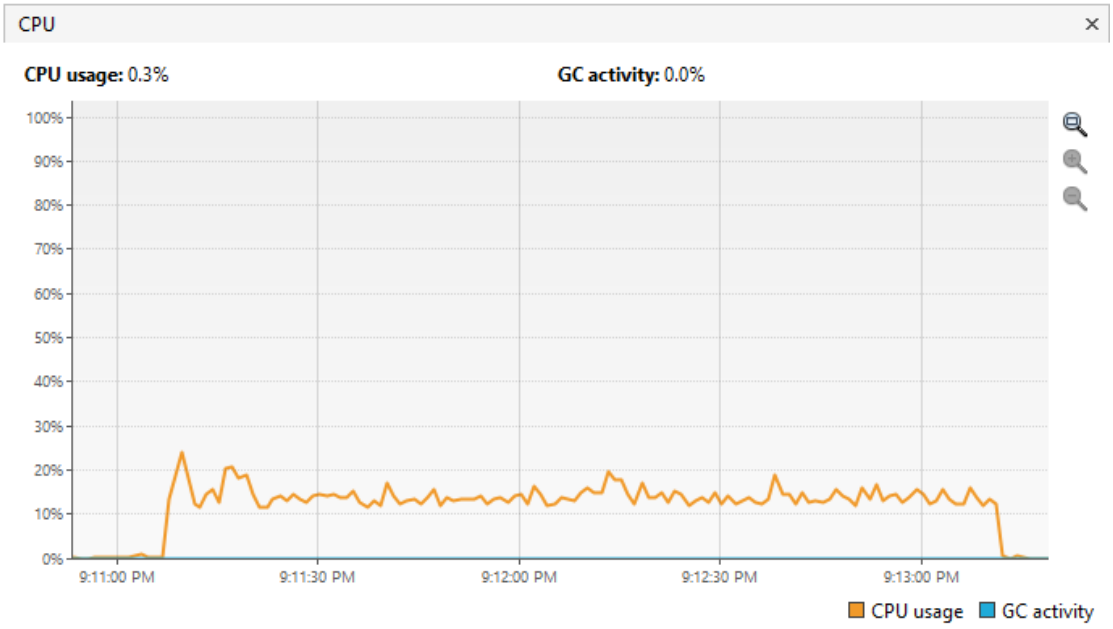
**Obr. A.10** Využitie procesoru Vert.x Aplikácie - Test väčších dát




**Obr. A.11** Využitie procesoru Vert.x Aplikácie - Test úpravy a kombinácie



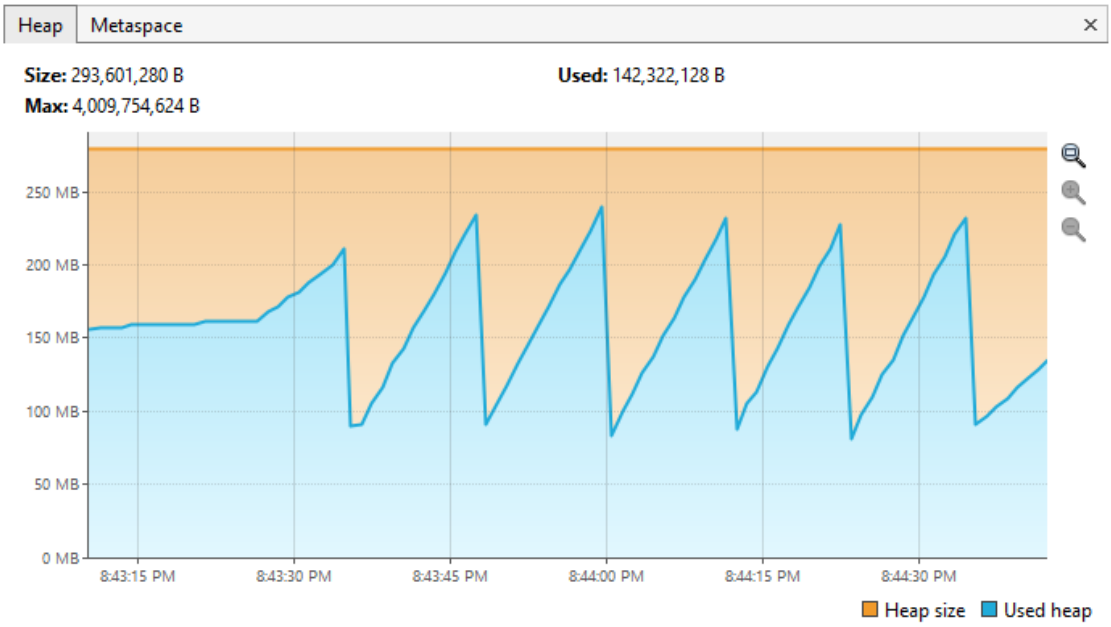
**Obr. A.12** Využitie procesoru Vert.x Aplikácie - Test veľkej skupiny užívateľov



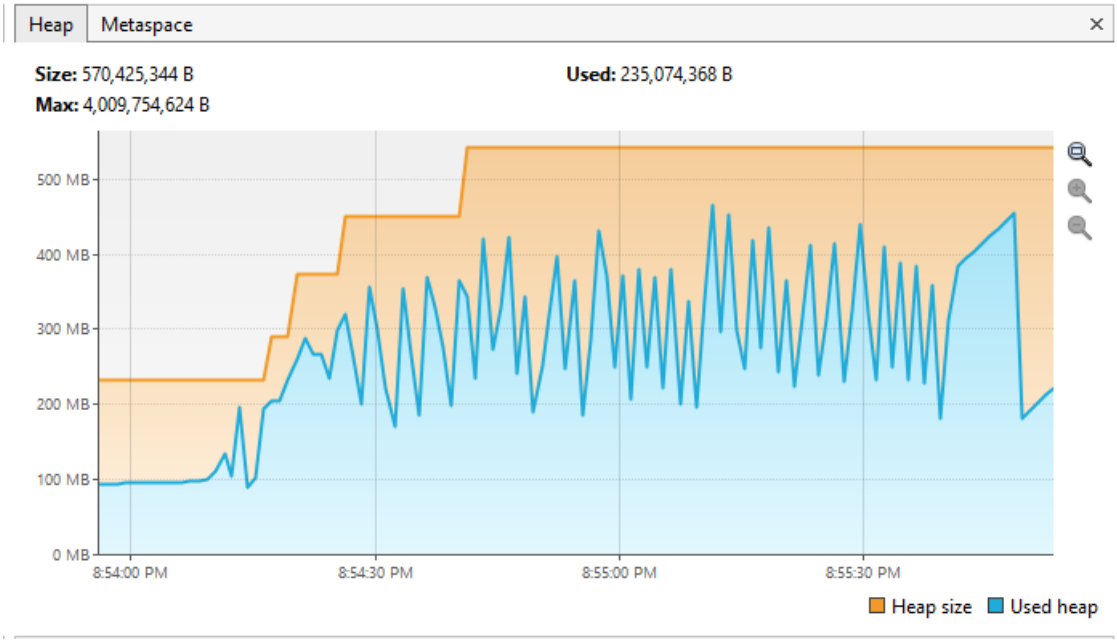
Dodatok B

# Priložené výsledky merania využitia pamäte

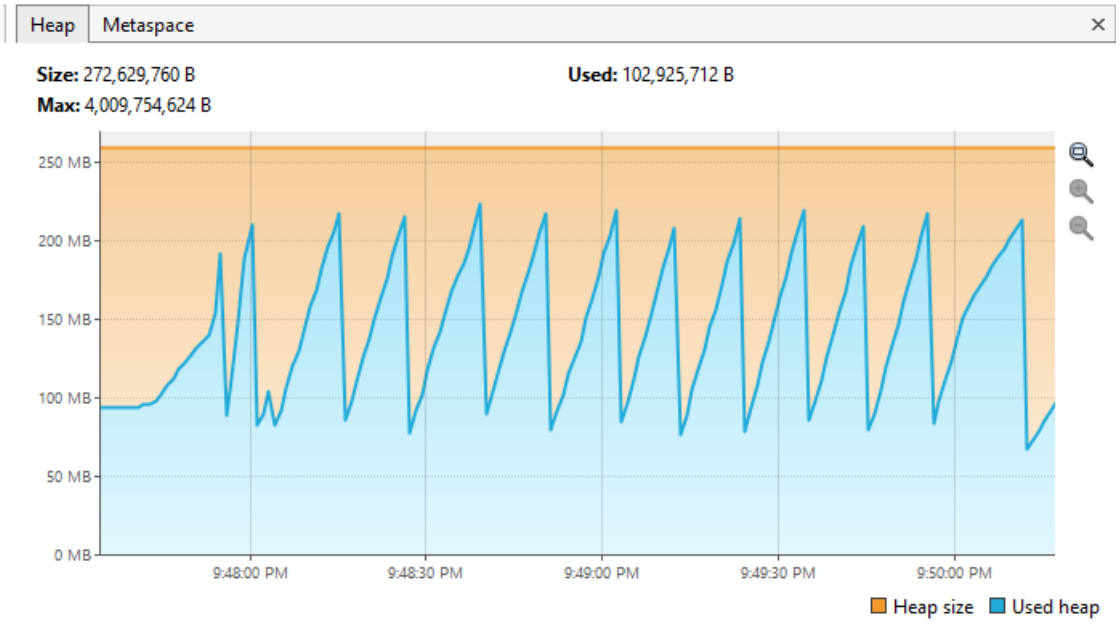
**Obr. B.1** Využitie pamäte Quarkus Aplikácie - Test malej skupiny užívateľov



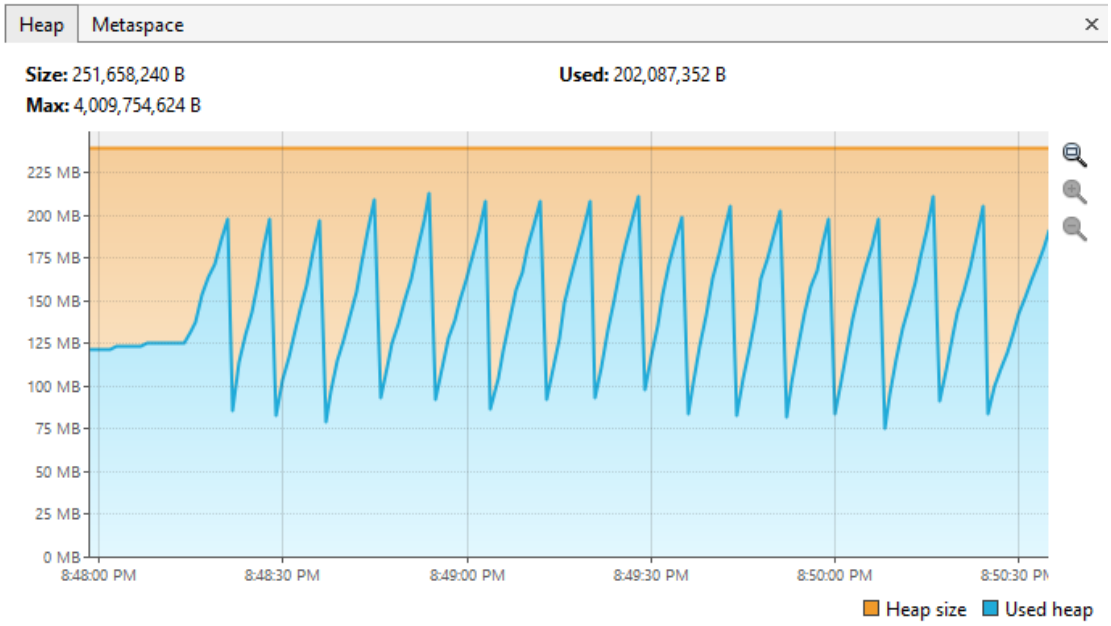
**Obr. B.2** Využitie pamäte Quarkus Aplikácie - Test väčších dát



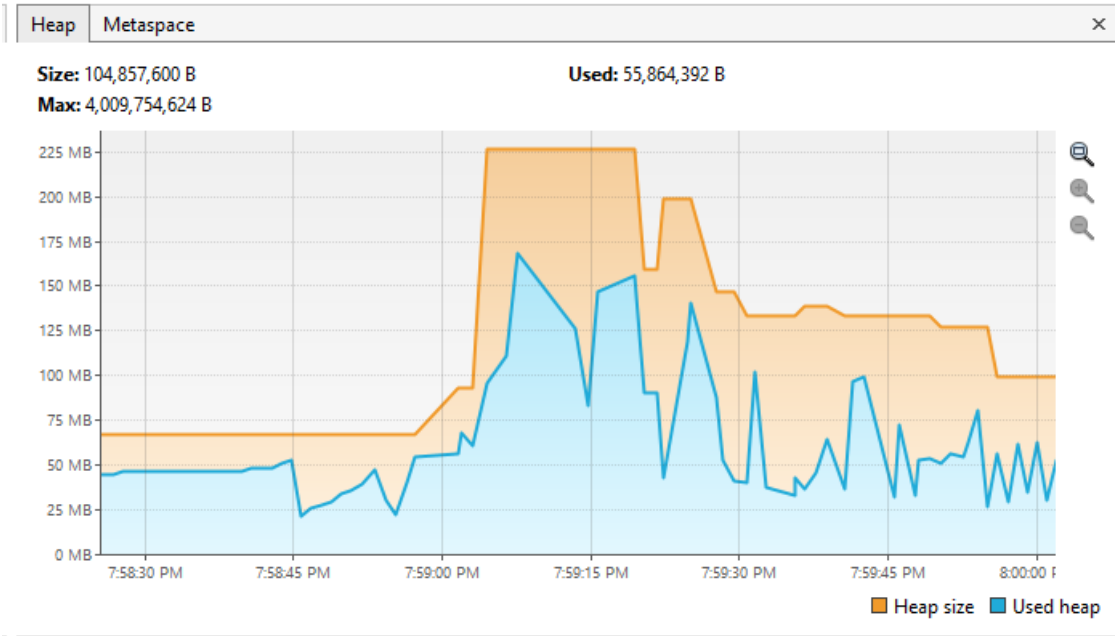

**Obr. B.3** Využitie pamäte Quarkus Aplikácie - Test úpravy a kombinácie



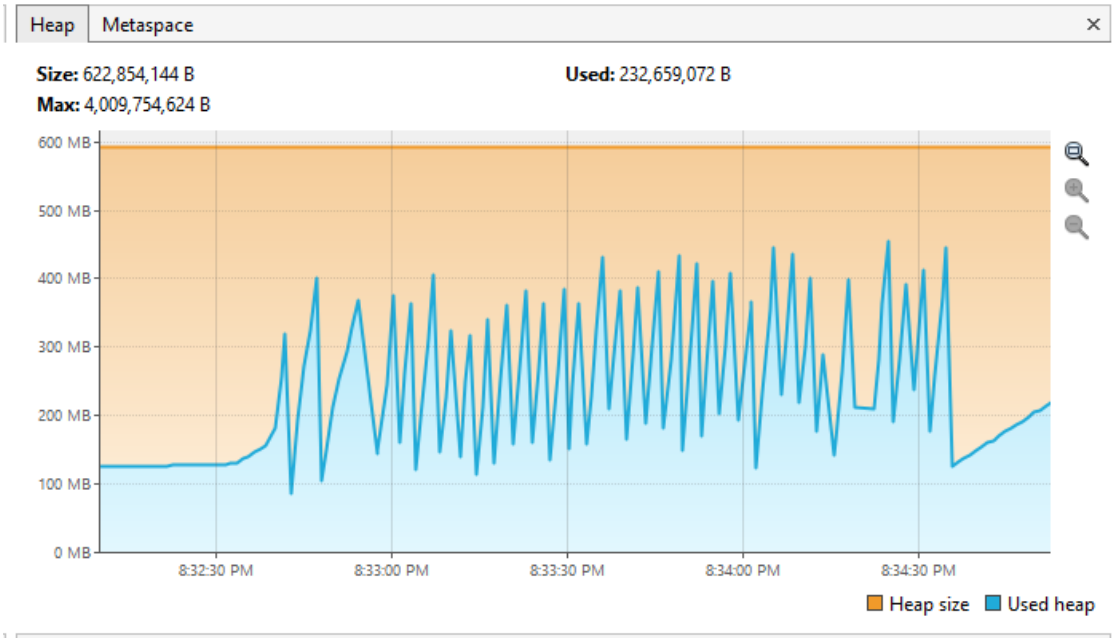
**Obr. B.4** Využitie pamäte Quarkus Aplikácie - Test veľkej skupiny užívateľov



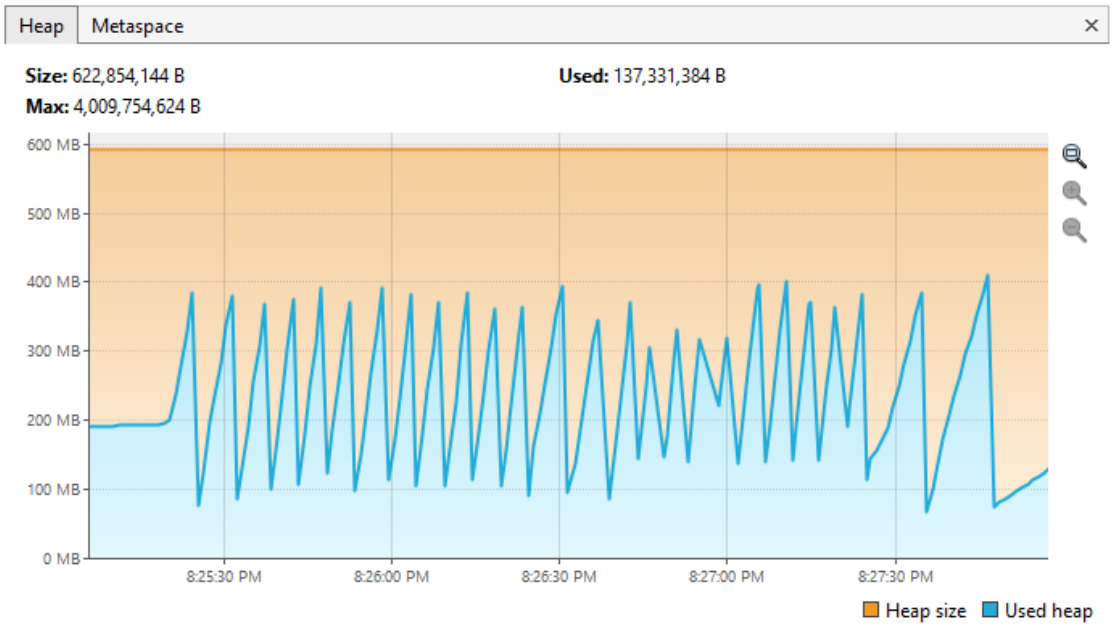

**Obr. B.5** Využitie pamäte Spring Aplikácie - Test malej skupiny užívateľov



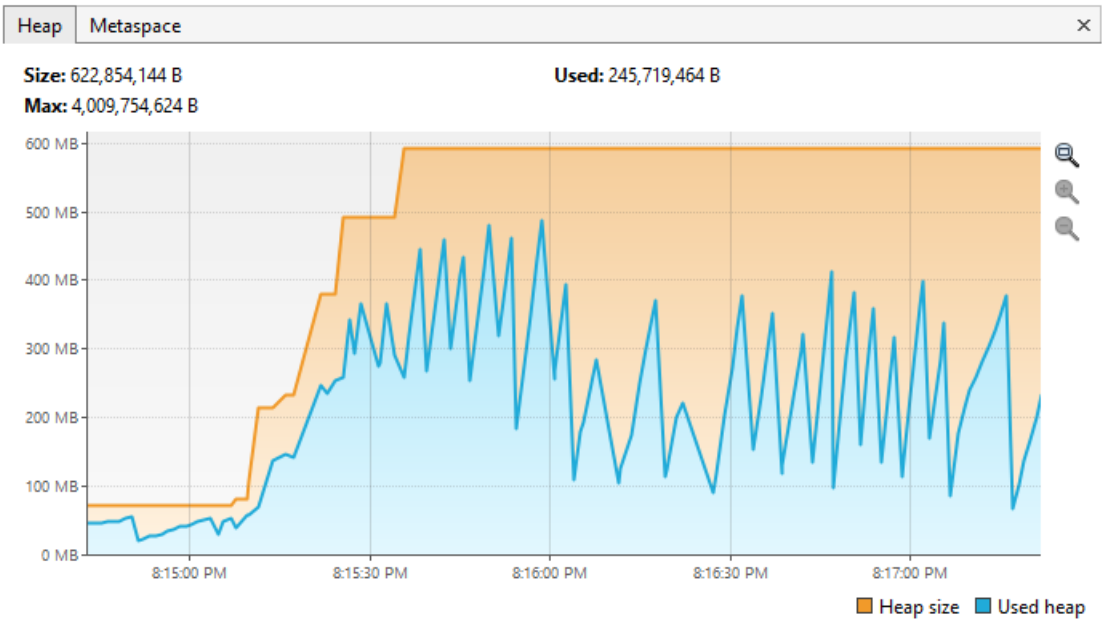
**Obr. B.6** Využitie pamäte Spring Aplikácie - Test väčších dát



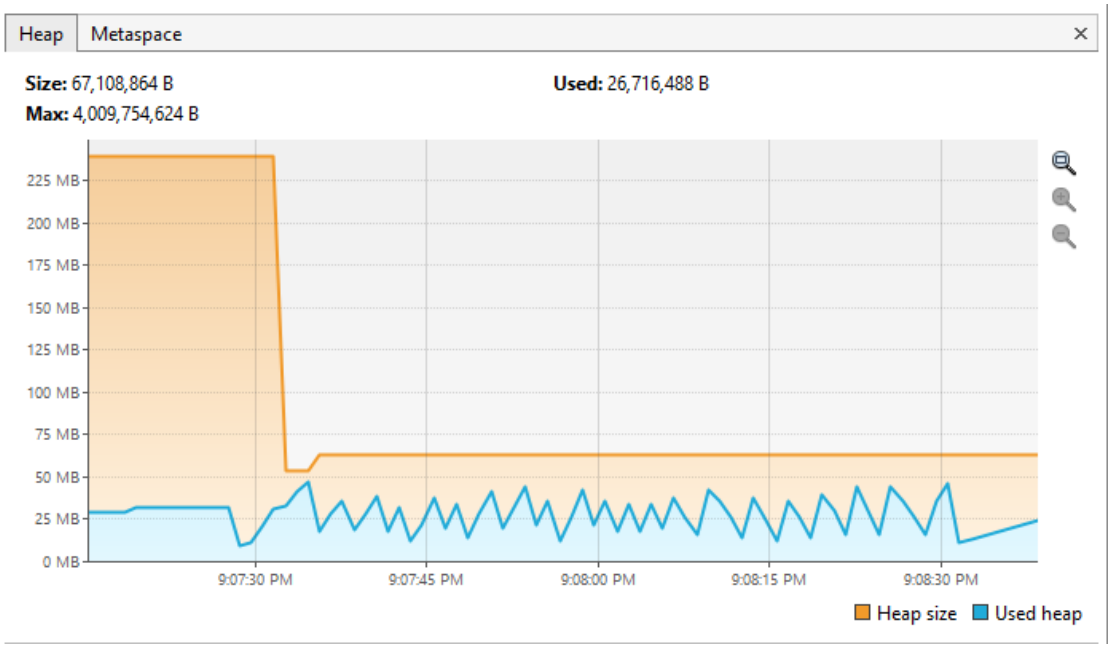

**Obr. B.7** Využitie pamäte Spring Aplikácie - Test úpravy a kombinácie



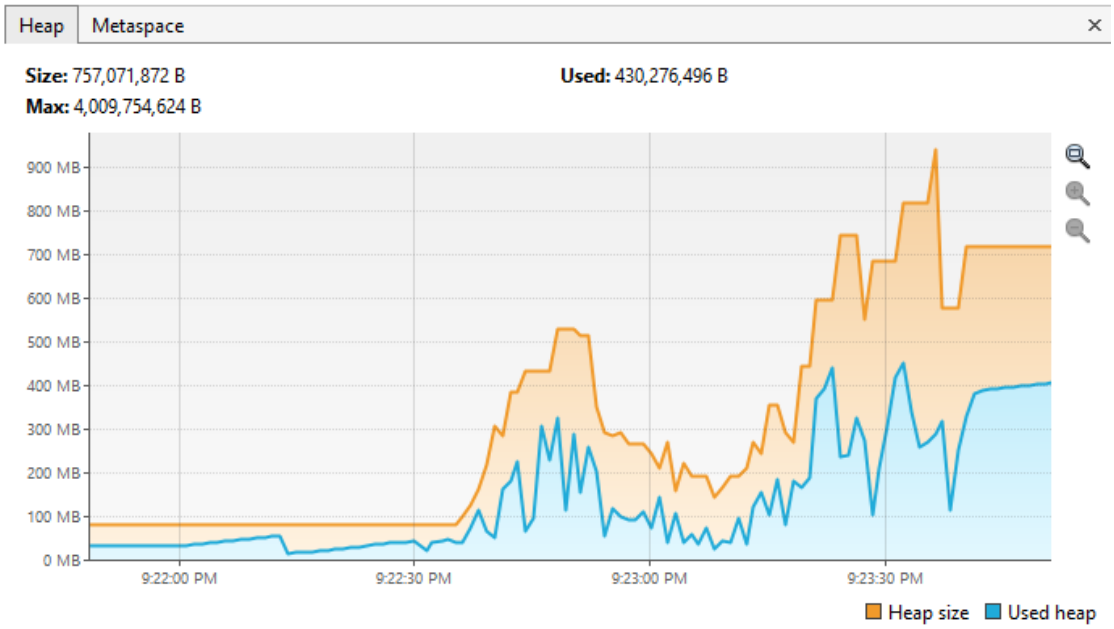
**Obr. B.8** Využitie pamäte Spring Aplikácie - Test veľkej skupiny užívateľov



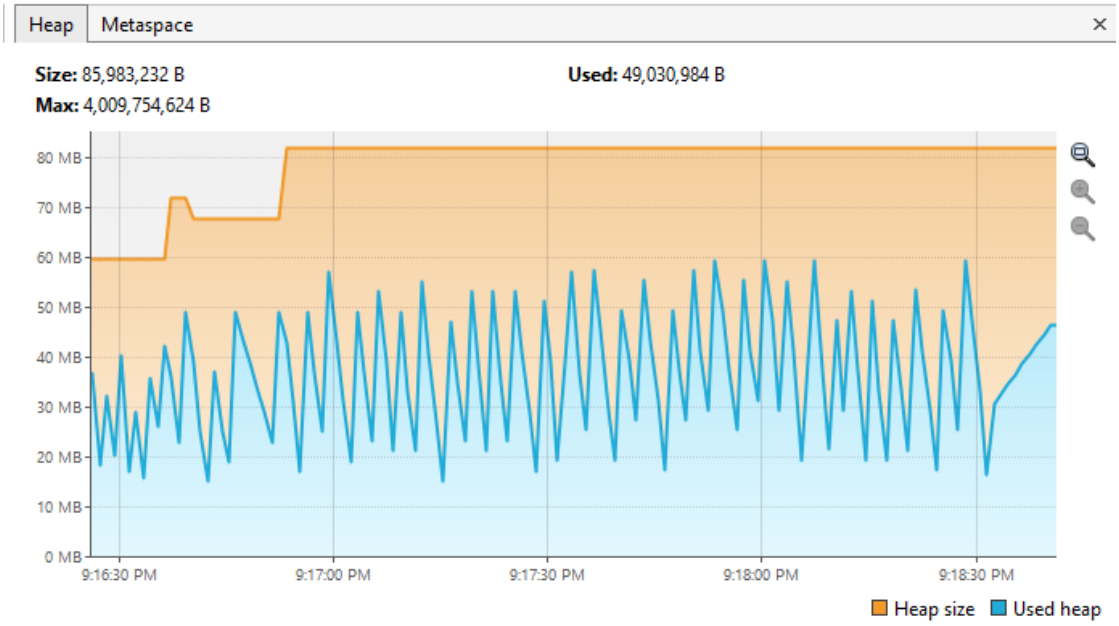

**Obr. B.9** Využitie pamäte Vert.x Aplikácie - Test malej skupiny užívateľov



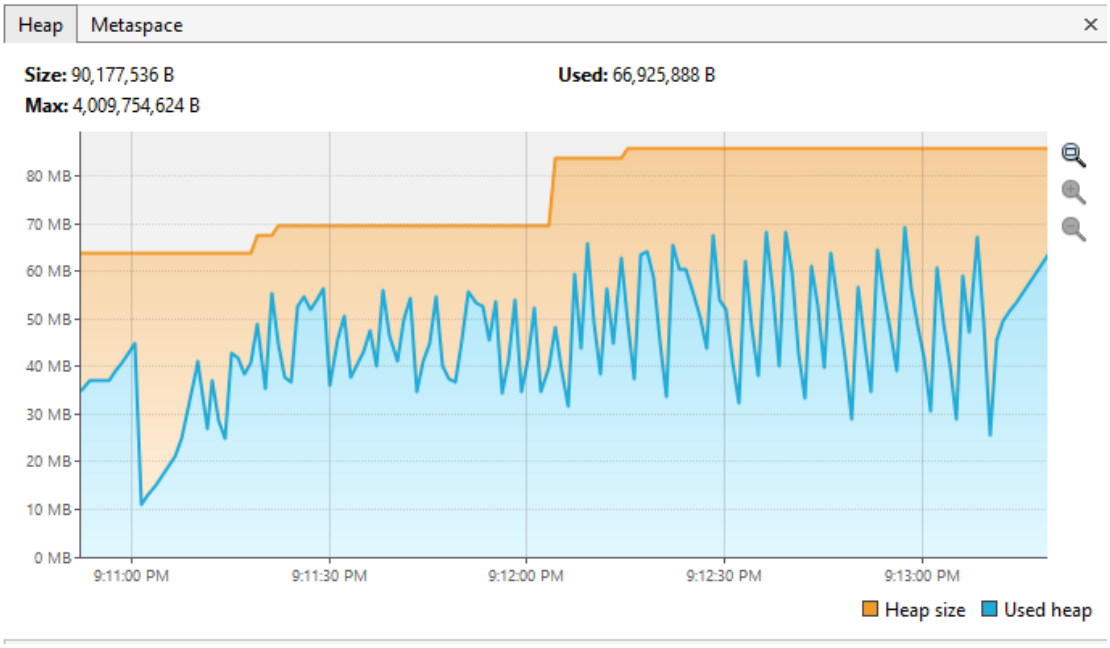
**Obr. B.10** Využitie pamäte Vert.x Aplikácie - Test väčších dát




**Obr. B.11** Využitie pamäte Vert.x Aplikácie - Test úpravy a kombinácie



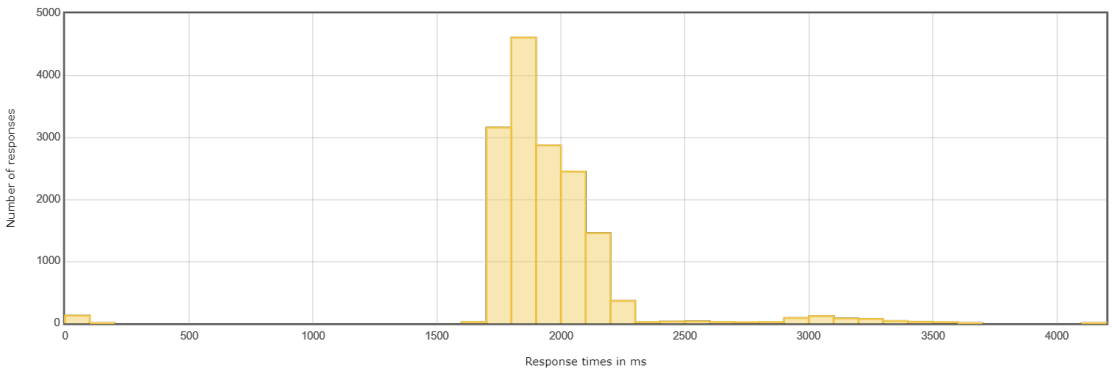
**Obr. B.12** Využitie pamäte Vert.x Aplikácie - Test veľkej skupiny užívateľov



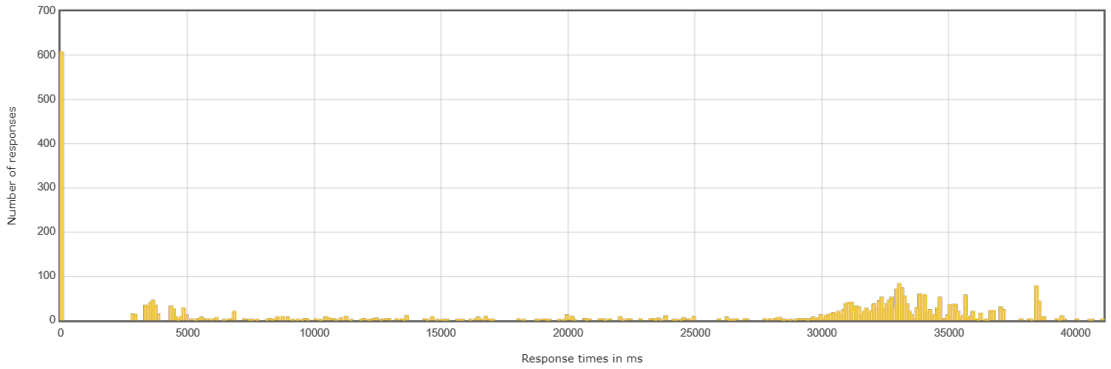
Dodatok C

# Priložené výsledky času odozvy jednotlivých požiadaviek meraných v rámci vykonanných testov

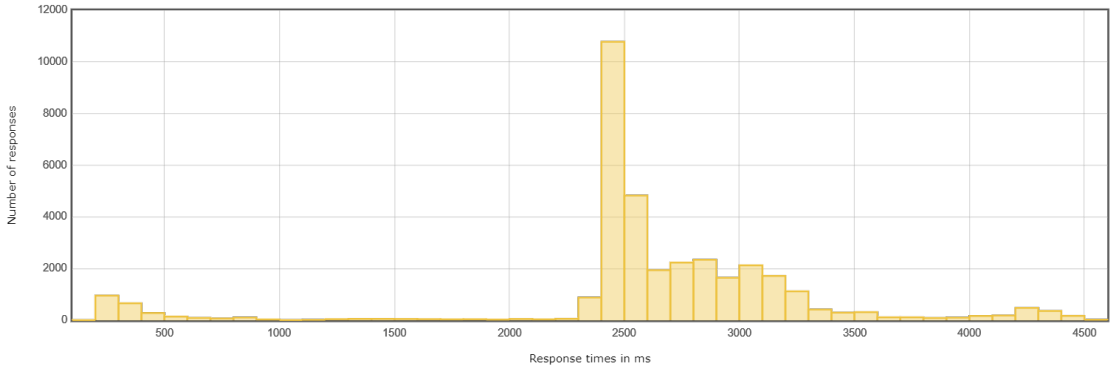
**Obr. C.1** Časy odozvy požiadaviek na Quarkus aplikáciu - Test malej skupiny užívateľov



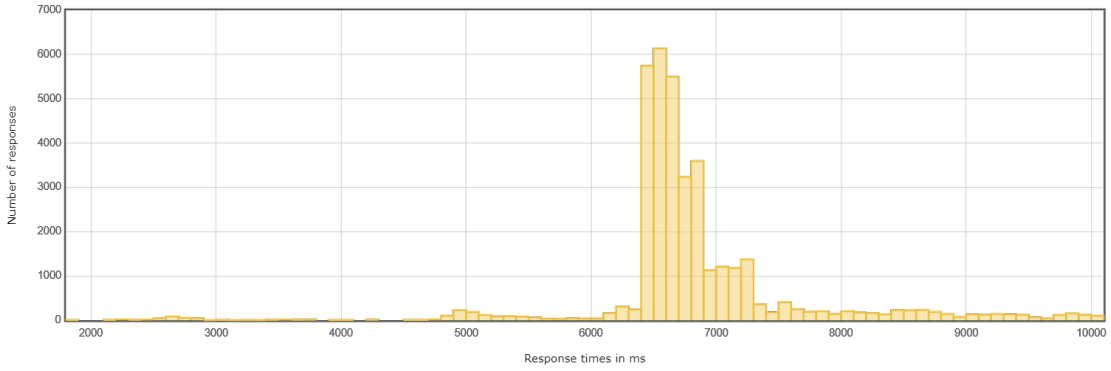
**Obr. C.2** Časy odozvy požiadaviek na Quarkus aplikáciu - Test väčších dát



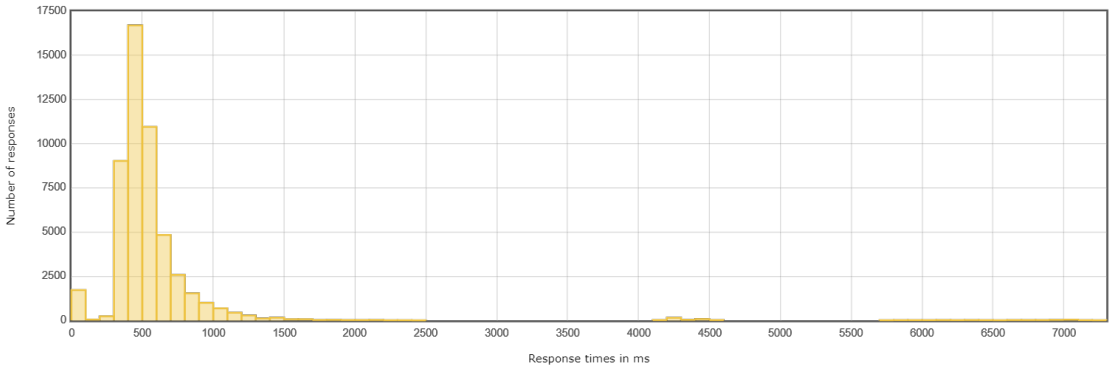

**Obr. C.3** Časy odozvy požiadaviek na Quarkus aplikáciu - Test úpravy a kombinácie



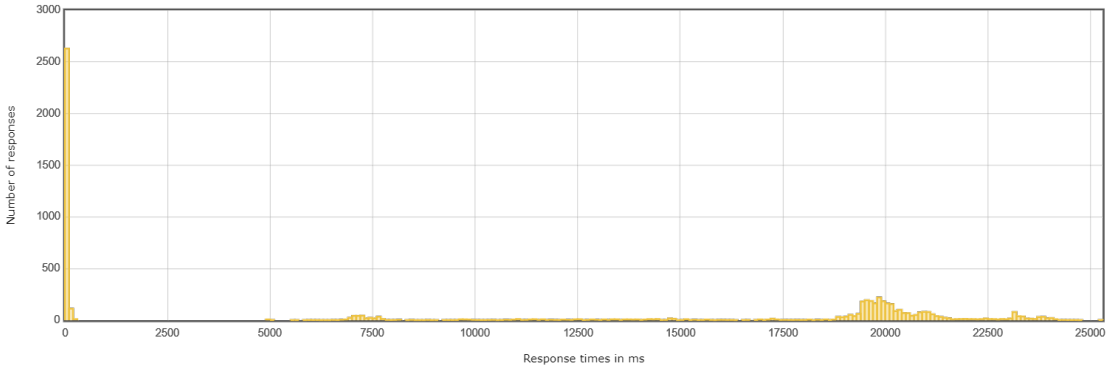

**Obr. C.4** Časy odozvy požiadaviek na Quarkus aplikáciu - Test veľkej skupiny užívateľov



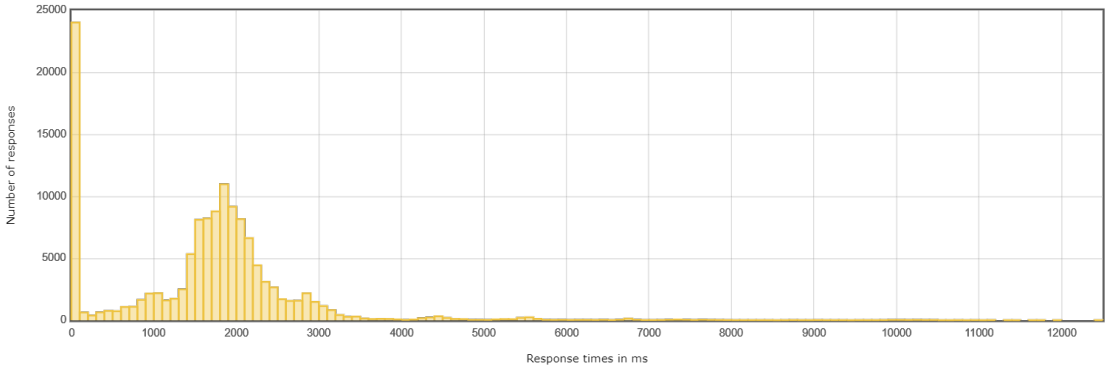
**Obr. C.5** Časy odozvy požiadaviek na Spring aplikáciu - Test malej skupiny užívateľov



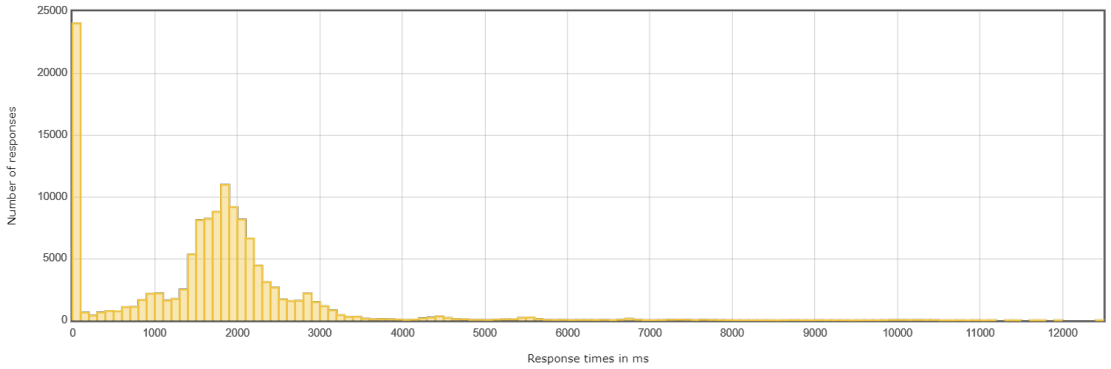

**Obr. C.6** Časy odozvy požiadaviek na Spring aplikáciu - Test väčších dát



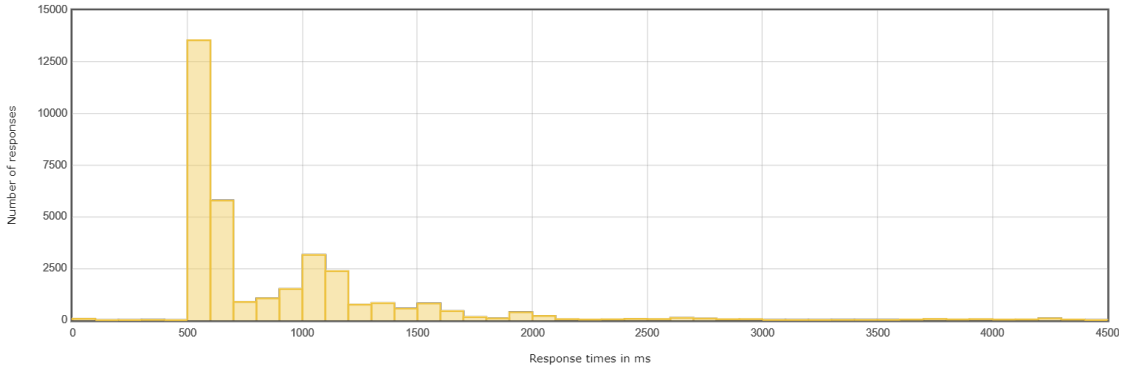

**Obr. C.7** Časy odozvy požiadaviek na Spring aplikáciu - Test úpravy a kombinácie



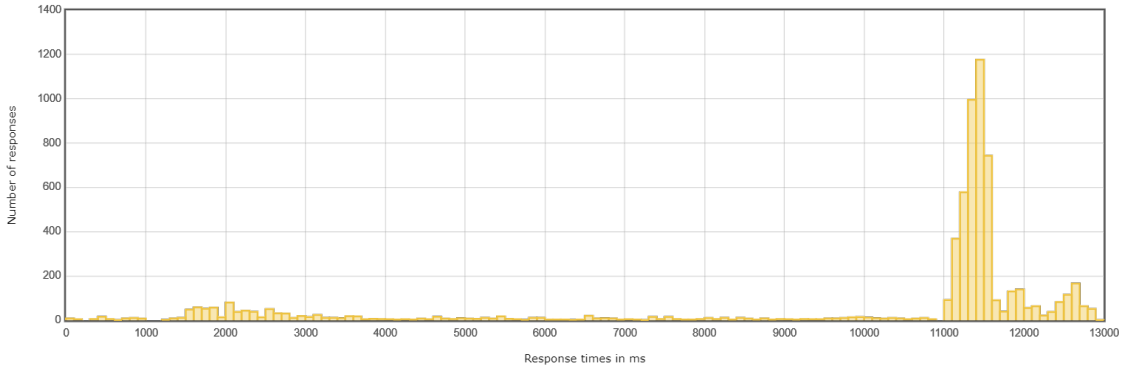
**Obr. C.8** Časy odozvy požiadaviek na Spring aplikáciu - Test veľkej skupiny užívateľov



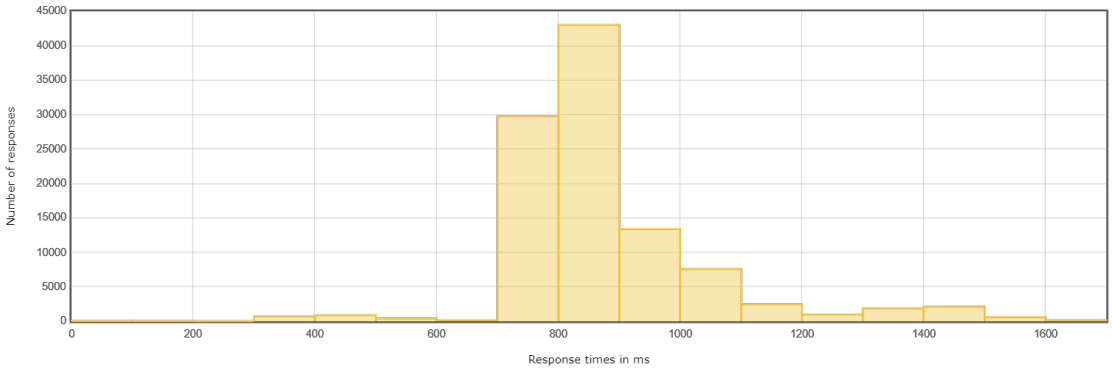

**Obr. C.9** Časy odozvy požiadaviek na Vert.x aplikáciu - Test malej skupiny užívateľov



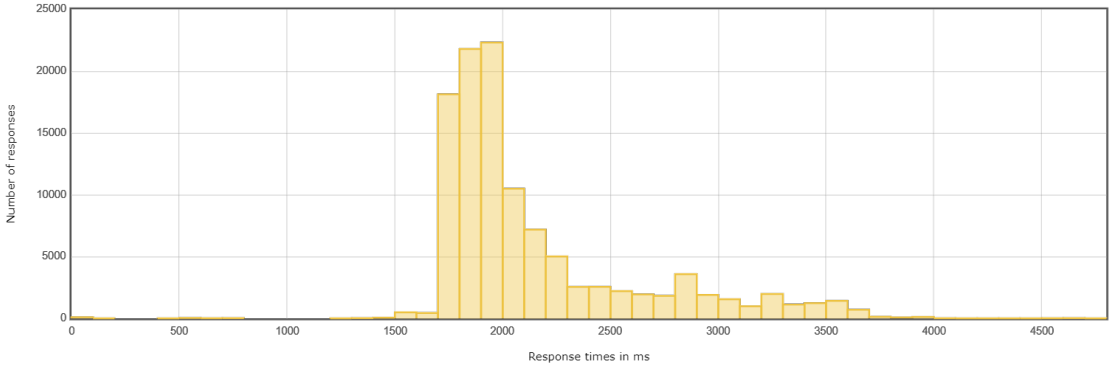

**Obr. C.10** Časy odozvy požiadaviek na Vert.x aplikáciu - Test väčších dát



**Obr. C.11** Časy odozvy požiadaviek na Vert.x aplikáciu - Test úpravy a kombinácie




**Obr. C.12** Časy odozvy požiadaviek na Vert.x aplikáciu - Test veľkej skupiny užívateľov



# Bibliografia


1. VAN ROY, Peter. Programming Paradigms for Dummies: What Every Programmer Should Know. 2012. [cit. 2023-04-26].
2. ENGINEER, BAINOMUGISHA; LOMBIDE CARRETON, Andoni; VAN CUTSEM, Tom; STIJN, MOSTINCKX; DE MEUTER, Wolfgang. A Survey on Reactive Programming. *ACM Computing Surveys*. 2012, roč. 45. Dostupné z DOI: 10.1145/2501654.2501666. [cit. 2023-04-26].
3. *Server* [online]. [cit. 2023-04-21]. Dostupné z : https://www.techtarget.com/whatis/definition/server.
4. *ServerApplication* [online]. [cit. 2023-04-21]. Dostupné z : https://www.techopedia.com/definition/432/application-server.
5. *API* [online]. [cit. 2023-04-21]. Dostupné z : https://www.ibm.com/topics/api.
6. *Request* [online]. [cit. 2023-04-21]. Dostupné z : https://www.computerhope.com/jargon/r/request.htm.
7. *HTTP* [online]. [cit. 2023-04-21]. Dostupné z : https://www.techopedia.com/definition/2336/hypertext-transfer-protocol-http.
8. GAPTA, Lokesh. *What is REST* [online]. 2022. Dostupné tiež z: https://restfulapi.net. [cit. 2023-05-10].
9. ARPACI-DUSSEAU, Remzi H.; ARPACI-DUSSEAU, Andrea C. *Operating Systems: Three Easy Pieces*. 1.10. vyd. Arpaci-Dusseau Books, 2023. [cit. 2024-01-10].
10. *Spring Webflux* [online]. 2022. Dostupné tiež z: https://docs.spring.io/spring-framework/docs/current/reference/html/web-reactive.html#webflux-concurrency-model. [cit. 2023-04-26].
11. *"Reactive Manifesto"* [online]. 2014. Dostupné tiež z: https://www.reactivemanifesto.org/. [cit. 2023-04-26].
12. *CompletableFuture* [online]. 2023. Dostupné tiež z: https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/CompletableFuture.html. [cit. 2023-04-26].
13. KLANG, Viktor. *Reactive streams 1.0.0 interview*. Medium, 2015. Dostupné tiež z: https://medium.com/@viktorklang/reactive-streams-1-0-0-interview-faaca2c00bec.

14. GAMMA, Erich; HELM, Richard; JOHNSON, Ralph; VLISSIDES, John. *Design Patterns: Elements of Reusable Object-Oriented Software*. USA: Addison-Wesley Longman Publishing Co., Inc., 1995. ISBN 0201633612. [cit. 2023-04-26].
15. *Reactive streams* [online]. [B.r.]. Dostupné tiež z: http://www.reactive-streams.org/. [cit. 2023-04-26].
16. *Interface Publisher* [online]. [B.r.]. Dostupné tiež z: http://www.reactive-streams.org/reactive-streams-1.0.4-javadoc/org/reactivestreams/Publisher.html. [cit. 2023-04-26].
17. *Package org.reactivestreams* [online]. [B.r.]. Dostupné tiež z: http://www.reactive-streams.org/reactive-streams-1.0.4-javadoc/org/reactivestreams/package-summary.html. [cit. 2023-04-26].
18. *Interface Subscriber* [online]. [B.r.]. Dostupné tiež z: http://www.reactive-streams.org/reactive-streams-1.0.4-javadoc/org/reactivestreams/Subscriber.html. [cit. 2023-04-26].
19. *Interface Subscription* [online]. [B.r.]. Dostupné tiež z: http://www.reactive-streams.org/reactive-streams-1.0.4-javadoc/org/reactivestreams/Subscription.html. [cit. 2023-04-26].
20. *Web Framework Benchmarks* [online]. [B.r.]. Dostupné tiež z: https://www.techempower.com/benchmarks/#section=data-r21. [cit. 2023-04-26].
21. *2023 State of Java in the Enterprise Report* [online]. [B.r.]. Dostupné tiež z: https://vaadin.com/java-report-2023. [cit. 2023-05-10].
22. *JetBrains Developer Ecosystem* [online]. [B.r.]. Dostupné tiež z: https://www.jetbrains.com/lp/devecosystem-2022/java/. [cit. 2023-05-10].
23. *Introduction to Vert.X* [online]. [cit. 2023-04-21]. Dostupné z : https://vertx.io/introduction-to-vertx-and-reactive/.
24. *Spring* [online]. [cit. 2023-04-21]. Dostupné z : https://docs.spring.io/spring-framework/reference/overview.html.
25. *SpringBoot* [online]. [cit. 2023-04-21]. Dostupné z : https://www.ibm.com/topics/java-spring-boot.
26. *Flux* [online]. [cit. 2023-04-21]. Dostupné z : https://projectreactor.io/docs/core/release/reference/#flux.
27. *Mono* [online]. [cit. 2023-04-21]. Dostupné z : https://projectreactor.io/docs/core/release/reference/#mono.
28. *What is Quarkus ?* [online]. [cit. 2023-04-21]. Dostupné z : https://quarkus.io/about/.
29. *Continuum* [online]. [cit. 2023-04-21]. Dostupné z : https://quarkus.io/continuum/.
30. *What makes Mutiny different ?* [online]. [cit. 2023-04-21]. Dostupné z : https://smallrye.io/smallrye-mutiny/latest/reference/what-makes-mutiny-different/.
31. *Creating Uni* [online]. [cit. 2023-04-21]. Dostupné z : https://smallrye.io/smallrye-mutiny/2.0.0/tutorials/creating-uni-pipelines.

32. *Creating Multi* [online]. [cit. 2023-04-21]. Dostupné z : https://smallrye.io/smallrye-mutiny/2.0.0/tutorials/creating-multi-pipelines/.

33. *VertxObjekt* [online]. [cit. 2023-04-21]. Dostupné z : https://vertx.io/docs/vertx-core/java/#_in_the_beginning_there_was_vert_x.

34. *EventBus* [online]. [cit. 2023-04-21]. Dostupné z : https://vertx.io/docs/vertx-core/java/#event_bus.

35. *Verticle* [online]. [cit. 2023-04-21]. Dostupné z : https://vertx.io/docs/vertx-core/java/#_verticles.

36. MORA, Fernando López de la; NADI, Sarah. An Empirical Study of Metric-Based Comparisons of Software Libraries. In: *Proceedings of the 14th International Conference on Predictive Models and Data Analytics in Software Engineering*. Oulu, Finland: Association for Computing Machinery, 2018, s. 22–31. PROMISE'18. ISBN 9781450365932. Dostupné z DOI: 10.1145/3273934.3273937. [cit. 2024-01-10].

37. A. NORDLUND AND N. NORDSTRÖM. *Reactive vs Non-Reactive Java framework: A comparison between reactive and non-reactive APIs*. 2012. Dostupné tiež z: https://www.diva-portal.org/smash/record.jsf?pid=diva2%3A1672751&dswid=9209. [cit. 2024-01-10].

38. MAINTENANCE, UAlberta Software; REUSE. *Library Metric Scripts* [https://https://github.com/ualberta-smr/LibraryMetricScripts]. GitHub, 2019. [cit. 2024-01-10].

39. *JMeter* [online]. [B.r.]. Dostupné tiež z: https://jmeter.apache.org/. [cit. 2024-01-10].

40. *JMeter: Elements of a test plan* [online]. [B.r.]. Dostupné tiež z: https://jmeter.apache.org/usermanual/test_plan.html. [cit. 2024-01-10].

41. *JMeter: Generating Report Dashboard* [online]. [B.r.]. Dostupné tiež z: https://jmeter.apache.org/usermanual/generating-dashboard.html. [cit. 2024-01-10].

42. *VisualVM Documentation*. 2017. Dostupné tiež z: https://visualvm.github.io/documentation.html. [cit. 2024-01-10].

43. *Snapshot* [online]. 2023. Dostupné tiež z: https://docs.oracle.com/javase/8/docs/technotes/guides/visualvm/snapshots.html. [cit. 2024-01-10].

44. *HP* [online]. [cit. 2024-01-10]. Dostupné z : https://support.hp.com/rs-en/document/c05997454/.

# Obsah přiloženého média

- `src`
  - `impl` ... zdrojové kódy implementácie
    - `web-webflux` ... zdrojový kód aplikácie Spring Webflux
    - `web-quarkus` ... zdrojový kód aplikácie Quarkus
    - `web-vertx` ... zdrojový kód aplikácie Vert.x
    - `test-utils` ... pomocné súbory pri testovaní aplikacií
  - `thesis` ... zdrojová forma práce ve formátu LaTeX
- `text` ... text práce
  - `thesis.pdf` ... text práce ve formátu PDF