

Téma: Využití jazyka Kotlin pro vývoj serverových aplikací

Cíl: Zhodnocení současných možností jazyka Kotlin pro tvorbu serverových aplikací

Otázky:

Proč použít Kotlin na serverový vývoj?

Jaké jsou jeho přínosy?

Lze Kotlinem a jeho proprietárními nástroji plně nahradit současná řešení při vývoji na serverech?

Osnova:

Porovnání jazyka Java a Kotlin

Výběr a hodnocení frameworků pro serverový vývoj

<https://github.com/KotlinBy/awesome-kotlin#libraries-frameworks-tests>

- Web-Microframeworky (Vert.X, Ktor, Micronaut)
  - Dependency Injection (Kodein, Koin)
  - Testování (KotlinTest + MockK)
  - Json/xml processing (Klaxon..)
  - ORM/DB
  - AOP

## Metoda hodnocení web frameworků

- Vybrány mikro-frameworky, které jsou nejvíce využívány + mají nativní podporu Kotlinu:
  - Vert.X
  - Ktor
  - Spark
  - Micronaut

**Výkonost** – měření výkonosti frameworku (využití ext. Zdroje), velikost projekt/warko, využití paměti

### **Popis frameworku:**

**Škálovatelnost** – zhodnocení možností frameworku při velkém rozvoji aplikace

**Ekosystém** – míra využití Kotlin features (corutiny), použitá architektura

**Modifikovatelnost** – míra závislosti frameworku na konkrétní implementaci/míra volnosti pro přizpůsobení

Moduly – mandatorní (vždy přítomné) vs. Volitelné

Komponenty – Možnost rozšíření (míra zapouzdření, rozšiřitelnosti), případně jejich záměna

**Funkčnost** – funkce, které framework poskytuje

Zhodnocení funkčnosti (chybí některá, je nadbytečná, jsou logicky uspořádané (rodič – potomek) a správně seskupeny)

Podpora jazykových mutací

Podpora AJAX

Podpora ORM

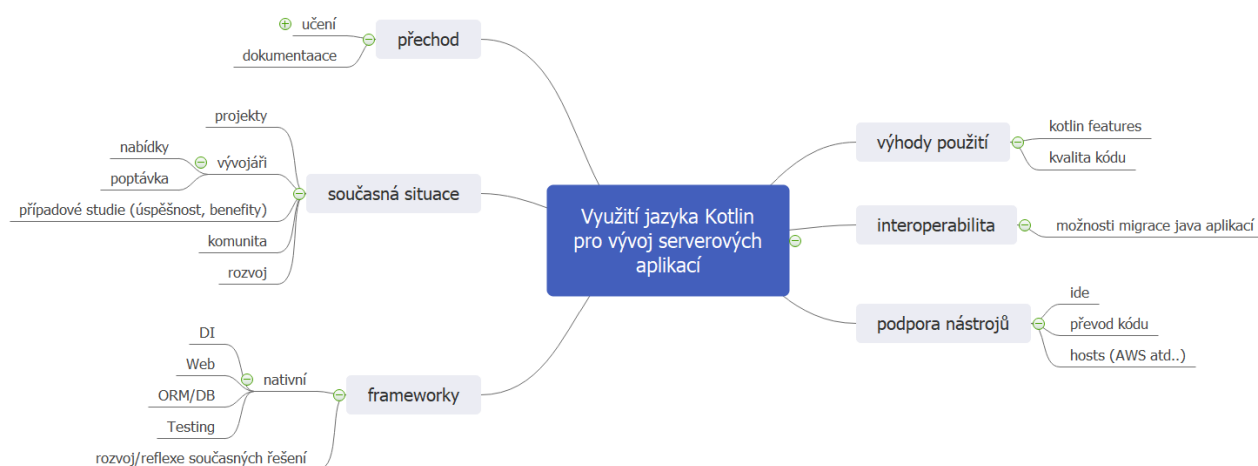
Moduly Security (+ ošetření současných hrozeb), **Templating**, Caching, Form validation, Page Navigation (+ POST, Redirect...)

**Komunita** – oblíbenost v komunitě, míra podpory

**Podpora** – frekvence verzí, dokumentace

**Praxe** – učící křivka, tech. Komplexnost, složitost orientace v projektu, složitost založení projektu, nástroje

**Testovatelnost** – jak je navrženo testování/lze kompletně pokrýt testy



## Historie vývoje serverových aplikací

Server je označení pro zařízení, nebo počítačový program, který poskytuje funkcionalitu, případně služby klientům, kteří jsou většinou reprezentováni programy či zařízeními. První zmínka o serverech se datuje k roku 1969 v RFC-5 (<https://tools.ietf.org/html/rfc5>), což je jeden z dokumentů popisující první globální síť ARPANET. Experimentální síť propojující univerzity, jež byla provozovaná až do roku 1990 a která se mimo jiné označuje jako předchůdce internetu, tak jak ho známe v dnešní době. V prvopočátcích byl vývoj serverových aplikací prováděn velmi ad-hoc, zejména díky tomu že se tvořila pro servery nestandardizovaná API a také se využívali programovací jazyky, které byly zrovna programátorům k dispozici, jednalo se nejčastěji o jazyk C. Programovací jazyk C byl vyvinut v roce 1972, určený pro všeobecné použití, jedná se o imperativní a procedurální jazyk. V této době totiž neexistoval žádný jazyk, nebo jeho nadstavba která by se specializovala na tvorbu serverových aplikací. Každý web server měl vlastní API např. NSAPI, Microsoft ISAPI, proti kterému se programovali serverové aplikace, avšak se jednalo o proprietární a nestandardizovaná řešení, tudíž se vždy vyvíjelo pro konkrétní implementaci webového serveru daného výrobce. Tuto nejednotnost v roce 1993 vyřešil CGI (Common Gateway Interface) standart, který umožnil webovým serverům poskytovat jednotné rozhraní pro zpracování požadavků tzv. http requestů. Velkou výhodou tohoto přístupu byla technologická volnost pro vývojáře, kteří si mohli vybrat způsob implementace a případně jazyk, který jim vyhovoval pro vývoj a vyvíjet aplikace bez závislosti na rozdílné implementaci rozhraní webových serverů různými výrobci.

Kromě výše zmíněného jazyka C se v té době pro serverový vývoj využíval jazyk Perl. Perl je skriptovací jazyk, který byl vyvinut v roce 1987, jedná se o dynamicky slabě typový jazyk. Jméno jazyka Perl je zkratka pro název Practical Extraction and Reporting Language. Jazyk, jak je patrné z názvu byl primárně určený pro vývoj skriptů pro systémy UNIX. Byť se jazyk Perl orientoval na jinou oblast vývoje i přesto první polovinu 90. let ve vývoji pro servery více či méně ovládl a stal se tak mezi serverovými vývojáři nepsaným standardem. Avšak stále neexistoval žádný jazyk, který by plně uspokojil požadavky vývojářů a byl primárně určen pro serverový vývoj. I přes velký rozmach trpěl jazyk Perl celou řadou nedostatků, jelikož jazyk nebyl původně zamýšlený pro vývoj serverových aplikací. Jedním z problémů jazyka byla jeho velká sémantická a syntaktická volnost, dalším problémem bylo jeho primární zaměření pro vývoj v rozsahu velikosti skriptů. Perl byl tím pádem hůře použitelný pro vývoj enterprise

aplikací, které jsou obvykle velmi rozsáhlé a díky výše uvedeným nedostatkům se stávali poměrně obtížně čitelné a spravovatelné pro vývojáře.

V této době se začali poprvé objevovat jazyky, které se používají i v současnosti pro vývoj serverových aplikací. V roce 1991 se objevil Python, interpretovaný, silně dynamicky typovaný objektový jazyk, který byl určen jak pro malé ale i enterprise řešení, jež se vyznačovali zejména vysokou čistotou vyvíjeného kódu aplikace. Dá se říct, že se jednalo o volného následníka jazyka Perl, který se dříve velmi výrazně do popředí v žebříčcích zájmu uživatelů v průběhu tohoto desetiletí, kdy je přiřazován k nejprogresivnějším jazykům dnešní doby.

O pár let později, konkrétně v roce 1995 se objevil velmi populární skriptovací jazyk na straně serveru, jednalo se o jazyk PHP. PHP je interpretovaný jazyk, který je multi-paradigmatický. V počátcích vzniku se v PHP programovalo procedurálně, kdy díky tomuto přístupu často vznikala tzv. špagetový kód, avšak v současné době se programuje zejména objektově, s využitím frameworků, které jsou dnes při jeho využití již v podstatě nutností a vývoj na čistém PHP je velmi ojedinelý.

Ve stejném roce se poprvé představil jazyk Java, který způsobil ve světě serverového vývoje poměrně velký boom. Java přišla s v té době revolučním konceptem WORA – write once, run everywhere, což byla v té době poměrně zásadní vlastnost, díky které se odstínila platformní závislost, se kterou se kompilované jazyky v té době potýkali. Java je interpretovaný, silně staticky typovaný, objektově orientovaný jazyk, který se kompiluje do tzv. bytecode, který je následně spuštěn, přesněji řečeno interpretován v JVM – Java Virtual Machine, který zajišťuje jeho platformní nezávislost. Další vlastností díky, které se jazyk stal populárním je jeho blízká syntaktická podobnost s jazyky z rodiny C, které se v té době stále hojně využívali, avšak oproti nim nabízel jazyk Java přístup na vyšší úrovni a vývojáře odstiňoval od problémů, kterým byly nuceni čelit např. od práce s pamětí v jazycích rodiny C, který byl častým zdrojem chyb. Java oproti tomu disponuje velmi kvalitní automatickou správou paměti a tzv. Garbage collectorem, který se stará o dealokaci paměti automaticky a také širokou škálu kvalitních knihoven, které velmi zpohodlňovali práci vývojáře. Jednou z výrazných předností jazyka, je plná zpětná kompatibilita, což je na druhou stranu v současné době poměrně velkou brzdou v rozvoji jazyka.

Po miléniu na rozmach Javy zareagoval i Microsoft a vydal jazyk C#. Jazyk C# je kompilovaný, silně staticky typovaný, objektově orientovaný jazyk, který vychází z rodiny C. S jazykem C++ dokáže na programové úrovni kooperovat. Jeho primární zaměření není jako v případě Javy

jen na servery, ale jeho využití je mnohostranné, navíc se v C# vyvíjejí i hry zejména pomocí populárního frameworku Unity. Jazyk umožňuje multiplatformnost díky kompilaci do MSIL code (Microsoft Intermediate Language), dnes nazýván CIL (Common Intermediate Language), který je následně just-in-time kompilován na hostiteli, což je podobný koncept jako používá jazyk Java.

Až v současném desetiletí se začali objevovat poměrně inovativní přístupy, které poměrně výrazně změnili pohled na serverový vývoj. V roce 2010 společnost Twitter představila jazyk JavaScript, který byl do té doby používán výhradně pro implementaci klientské strany, na straně serveru. První řešení vykazovalo výkonnostní problémy, avšak po mnoha optimalizacích architektury se podařilo získat velmi uspokojivé výsledky. V roce 2013 byl implementován jeden z prvních větších komerčních projektů, realizovaných právě pomocí JavaScriptu, konkrétně pomocí knihovny Node.js, který umožňuje jednoduché použití JavaScriptu pro serverové aplikace. Jazyk JavaScript byl vydán již v roce 1995 a původně sloužil jako jazyk pro tvorbu skriptů na webové stránky. Jedná se dynamicky typovaný jazyk, který je multi-paradigmatický. Nejčastěji se využívá v kombinaci s nějakým frameworkem na klientu i na serveru, v čisté podobě se vyskytuje ojediněle, mimo jiné také díky velmi produktivní komunitě, která produkuje mnoho rozšíření postavených nad jazykem.

Po půli prvního desetiletí milénia se objevila tendence hledat alternativy pro Javu, které byly podpořeny poměrně dlouhou pauzou mezi vydávanými verzemi, nejdelší trvala 5 let od roku 2006 kdy byla zveřejněna verze 6 do představení verze 7 v roce 2011, tedy poměrně dlouhá doba v porovnání s vývojem konkurenčních jazyků. Java díky zpětné kompatibilitě a malé aktivitě tehdejšího vlastníka jazyka společnosti Sun, nepřidávala zásadní vylepšení prostředí a jazyka jako takového. Vývojáři hledali alternativy, avšak málokdo chtěl úplně opouštět svět, který se točil kolem poměrně kvalitně vybudovaného prostředí JVM. Začali se objevovat jazyky jako Clojure, Scala, Groovy, které byly postavené nad JVM, využívali plně její potenciál a v některých případech i mnohem více než Java. Toto se projevovalo v měření výkonu<sup>1</sup> a alokace paměti, kdy některé jazyky dokázali předčít samotnou Javu, zejména v kódu s funkcionálními prvky. Jazyky také implementovali moderní koncepty a netradiční přístupy, které Java postrádala.

---

<sup>1</sup> <https://github.com/kostya/benchmarks>  
<https://www.slideshare.net/CorneilduPlessis/performance-comparison-jvm-languages>

Groovy je silně dynamicky typovaný skriptovací jazyk pro JVM. Poskytuje pokročilou správu více vláknového zpracování. Scala je multi-paradigmatický jazyk postavený na kombinaci funkcionálních a objektových přístupů. Stejně jako Groovy poskytuje pokročilou správu více vláknového zpracování, avšak na rozdíl od něj je silně staticky typovaný jazyk. Clojure je v podstatě obdoba jazyka Lisp na JVM, která klade důraz na více vláknové zpracování a jedná se o čistý funkcionální jazyk.

Výše zmíněné jazyky byly spíše pokusy, které využívali netradiční přístupy. Díky tomu faktu, se kterým souvisela poměrně náročná adaptace vývojáři a také na vrub ne zcela příznivé kompatibilitě s Javou stály tyto jazyky více méně v ústraní<sup>2</sup>. Aktivně je v komerčních projektech využívalo poměrně malé množství vývojářů v porovnání s původním jazykem postaveným nad JVM a to Javou.

Jeden z mála jazyků, který zaznamenal výraznější úspěch než výše uvedené a přitáhl pozornost mnoha vývojářů je Kotlin. Jazyk, který sází na plnou kompatibilitu s Javou a velmi rychlou adaptaci Java vývojáři. Kotlin je silně staticky typovaný jazyk, umožňující vývoj podle nejpoužívanější paradigmat (procedurální, objektové, funkcionální). Kotlin je plně objektový jazyk, oproti Javě v něm nenajdeme primitivní datové typy. První zmínky o Kotlinu se datují do července roku 2011.

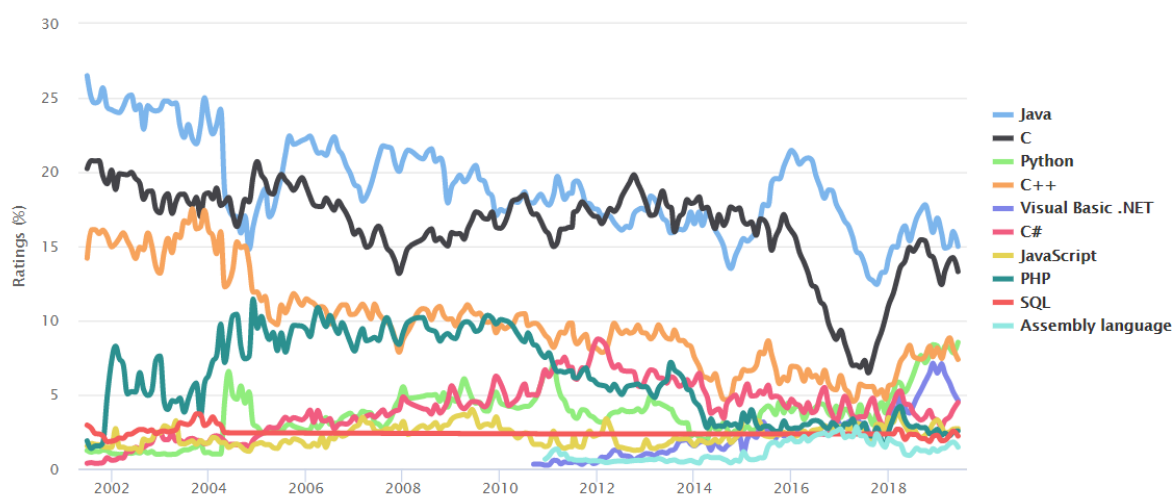
---

<sup>2</sup> <https://dzone.com/articles/the-rise-and-fall-of-jvm-languages>

## Současnost ve vývoji serverových aplikací

Ke zmapování technologií, které se v současné době nejčastěji používají pro vývoj serverových aplikací byla využita veřejně dostupná měření a výzkumy. Vyhodnocení bylo provedeno kombinací vybraných zdrojů, jelikož každý zdroj pokrývá jen část domény a jen ve vzájemné korelaci poskytují směřodátný výsledek.

Pro zhodnocení zájmu o osvojení programovacího jazyka jsem vyhledal metriku sledující trendy v četnosti vyhledávání klíčových slov, která se vztahují k vyhledávání tutoriálů programovacích jazyků, což se dá pokládat za množství zájmu, které se technologii skrze uživatele dostává. Toto hodnocení se nazývá PYPL (Popularity of Programming Language). K červnu 2019 mluví statistiky jasně o vítězi, kterým se stává jazyk Python s 28 % podílem z celkového počtu a relativním přírůstkem o 4,7 %, na druhém místě je Java s 20 % a s úbytkem o 1,8 %, na třetím místě se umístil JavaScript s již pouze 8 % a drobným úbytkem v řádu desetin procent, na čtvrtém místě se 7 % se umísťuje jazyk C#, která má úbytek zhruba půl procenta, páté místo obsadilo PHP s necelými 7 % a poměrně vysokým úbytkem v podobně 1 %. Jazyk Kotlin obsadil třinácté místo s 1,5 % a relativním přírůstkem 0,5 %, což je v procentuálním vyčíslení největší posun v celém žebříčku. Je zřejmé že vývojáři mají tendenci se učit nové a netradiční jazyky.



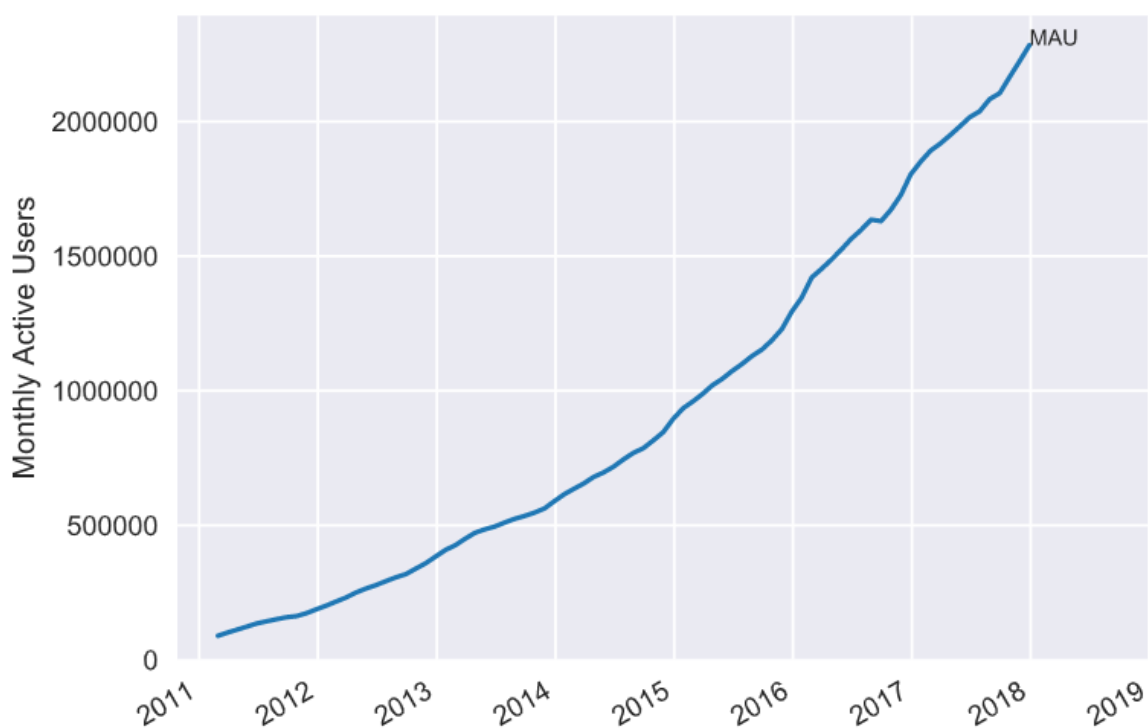
Obrázek 1 TIOBE Index

Další metrikou je TIOBE Index, který měří popularitu jazyků napříč vyhledávači, kde sleduje zájem o jazyky podle vyhledávání klíčových slov a neomezuje se pouze na tutoriály například jako výše zmíněný PYPL. Z grafu, je patrný prudký růst dotazů v posledních letech na jazyk



Python, u ostatních jazyků je trend spíše klesající. Růst Pythonu podpořil také zvýšený zájem o umělou inteligenci a strojové učení, kde se jazyk velmi často využívá. Zajímavý je výrazný propad, který Java zažila po roce 2016, který utnul její poměrně strmý růst.

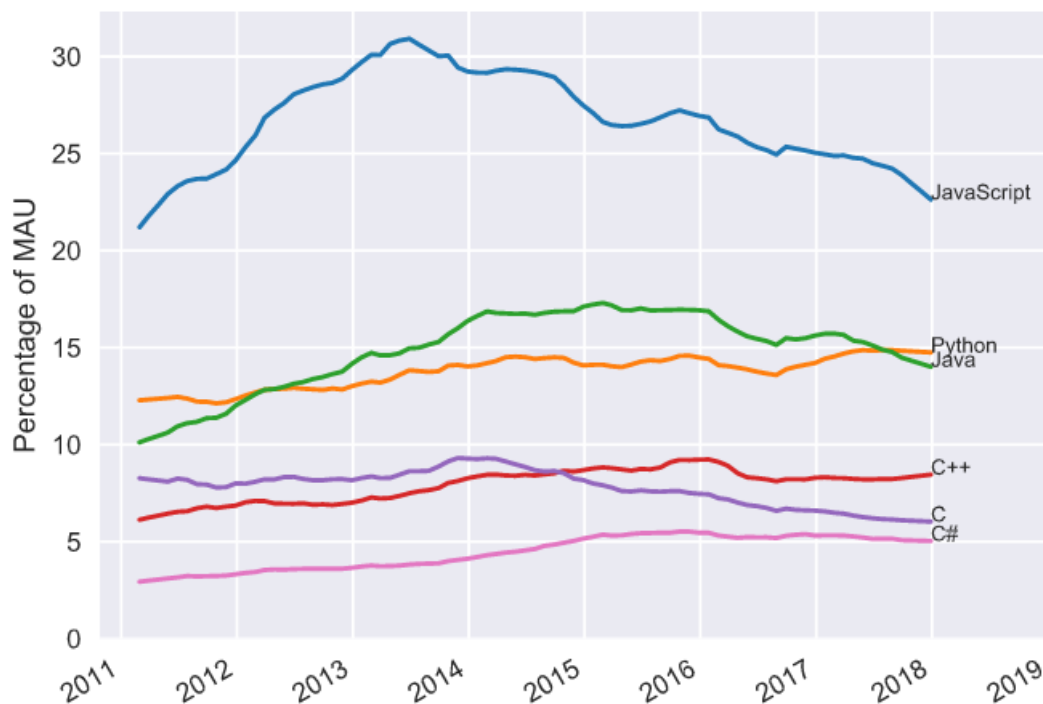
Ben Fredrickson vytvořil hodnocení jazyků podle dat o struktuře repositářů na GitHub, což je veřejné uložisko systému pro verzování Git. Do úložiště přispívá celkem 37 miliónů uživatelů, kteří spravují dohromady 75 miliónů repositářů a uskutečnili již celkem 1,25 miliard transakcí. Každá transakce je spárována s konkrétním repositářem a uživatelem. Fredrickson sjednotil transakce každého uživatele a repositáře za měsíc a díky údajům, které poskytuje repositář, bylo možné přiřadit kolekci sjednocených transakcí, ke konkrétnímu jazyku jako jeho použití. Z toho vznikl ukazatel Monthly Active User, tedy uživatelé, kteří jsou v rozmezí jednoho měsíce aktivní. Hodnoty dosahují v roce 2019 přes 2 miliony aktivních uživatelů za měsíc. Díky tomu rozsahu mají statistiky poměrně dobrou vypovídající hodnotu.



Obrázek 2 Počet aktivních uživatelů za měsíc

První metrikou bylo hodnocení programovacího jazyka podle počtu aktivních uživatelů za měsíc. U této metriky vyšel jako vítěz JavaScript, který měl 23 %, byť má klesající trend a pomalu se vrací až na počáteční úroveň z roku 2011, tedy předtím, než začal expandovat díky jeho uvedení jako full-stack řešení. Druhý Python s 15 % a mírným růstem, Java třetí se 14 % a klesajícím trendem, PHP se umístilo na šestém místě s 6 % a velmi výrazným klesajícím

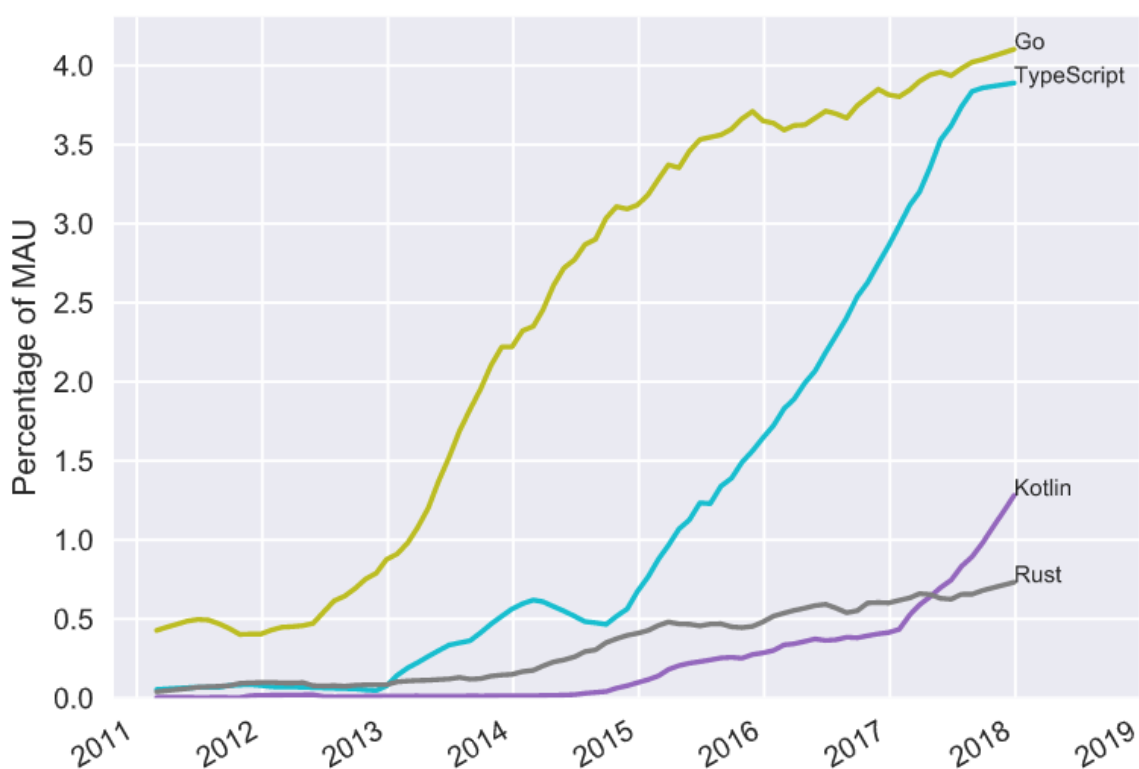
trendem, na sedmém C# s 5 %, které se drží již delší dobu na stejné hodnotě. Kotlin se objevil na patnáctém místě s 1,3 % a velmi prudkým růstem.



Obrázek 3 Počet aktivních uživatelů za měsíc

Druhá statistika zachycuje programovací jazyky, které mají největší relativní přírůstek aktivních uživatelů. Jedná se tedy o výběr nejvíce progresivních jazyků, kde rozhodujícím kritériem je vysoká míra zájmu, kterou jazyku programátoři projevují. Na vrcholu statistiky se již delší dobu drží jazyk Go, jenž je vyvíjený společností Google od roku 2007, avšak první stabilní verze byla uvedena až v roce 2012. Jedná se o kompilovaný, silně staticky typovaný, multi-paradigmatický jazyk, se syntaxí odvozenou z rodiny jazyků C. Hlavní motivací autorů byl záměr poskytnout inovovaný jazyk v mnoha aspektech podobný svému vzoru, avšak při zachování vysokého výkonu. Jazyk měl být obohacen o vlastnosti moderních programovacích jazyků, zvláštní důraz byl kladen na paralelizaci. Mezi klíčové benefity jazyka patří zvýšené zabezpečení paměti, kdy její správu převzal garbage collector a tzv. gorutiny, které umožňují paralelizaci na úrovni jazyka, díky tomu jsou o dost méně náročnější na využívání zdrojů než v případě tradičního řešení v podobě vláken. Go však nedisponuje všemi vlastnostmi, na které jsme zvyklí z jiných jazyků např. dědičnost a generika. Další nové programové konstrukty jsou do jazyka Go přidávány poměrně konzervativně, což mu někteří kritici vyčítají. Do jazyka Go jsou v současnosti nejvíce přepisovány síťové a webové aplikace na kterých je po převodu běžně zaznamenáván nárůst rychlosti v násobcích o řádu desítek a v některých případech

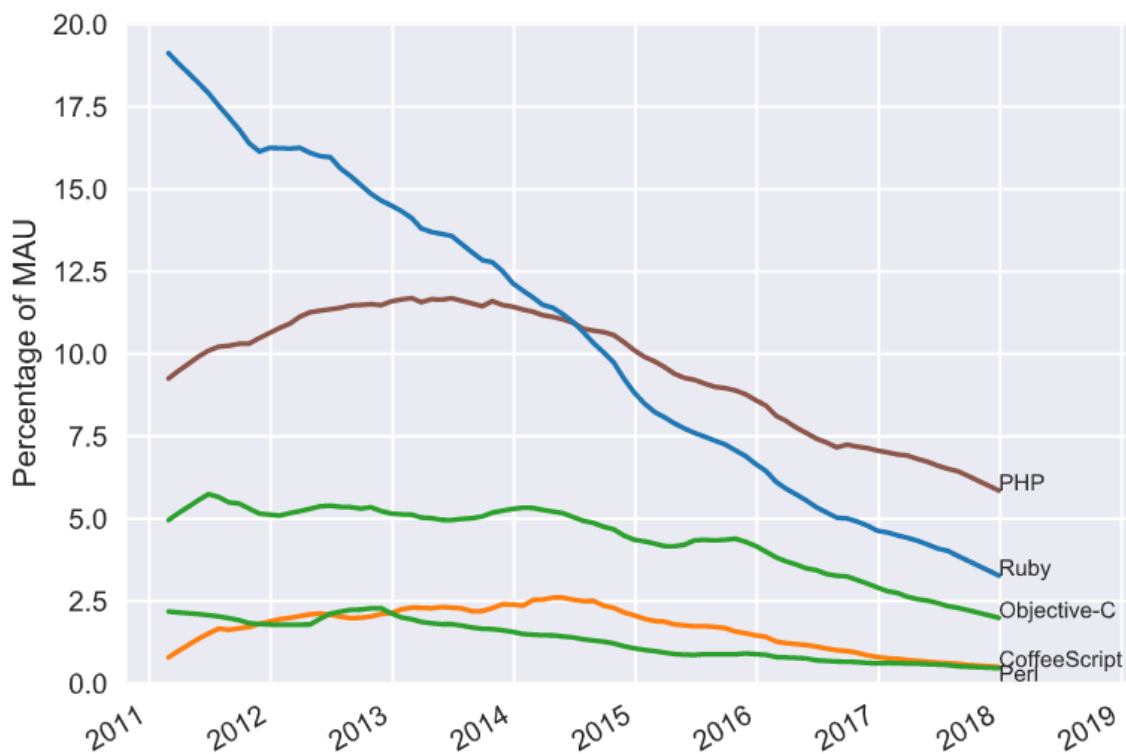
dokonce až stovek. Druhým jazykem je TypeScript, což je nadstavba pro JavaScript, která poskytuje typovou kontrolu a další vlastnosti, které jsou běžně používány v objektových jazycích. Velkou výhodou TypeScriptu je jeho plná kompatibilita s JavaScriptem díky přímému překladu, čím si získal značnou oblibu v komunitě vývojářů. Třetí je jazyk Kotlin, který zažívá poměrně strmý nárůst popularity od roku 2017, což se dá spojit s jeho oficiálním představením Googlem v květnu 2017 jako výchozí jazyk pro vývoj android aplikací. Avšak podle oficiálních statistik GitHub za rok 2018 je Kotlin oceněn jako nejrychleji rostoucí jazyk, který téměř ztrojnásobil počet kontributorů, v porovnání s Go, které dosáhlo pouze jeden a půl násobku.



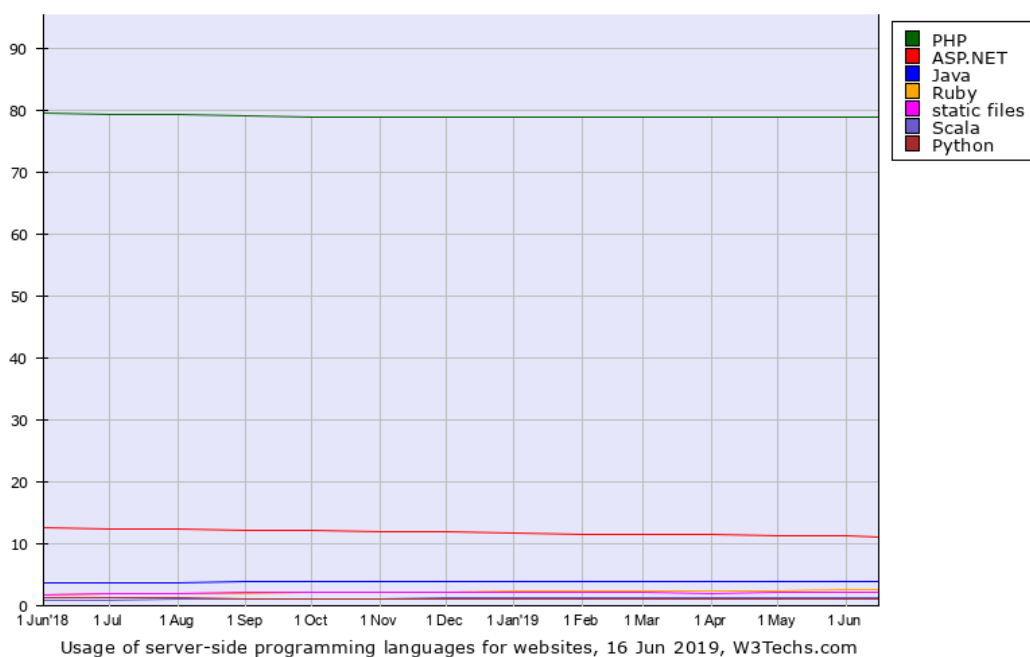
Obrázek 4 Relativní přírůstek aktivních uživatelů

Třetí statistika naopak zobrazuje největší procentuální odlivy aktivních uživatelů od programovacího jazyka. Na tomto grafu můžeme sledovat, které jazyky upadají v zájmu vývojářů, jenž je méně často využívají, respektive nepřispívají do repositářů, kde převládá daný jazyk. Ze serverových jazyků upadá zvláště jazyk Ruby, nad kterým se používá oblíbený webový framework Rails tzv. Ruby on Rails. Dalším velmi populárním jazykem, který upadá zájmu je jazyk PHP, pokles je nejspíše způsoben růstem inovativních jazyků a rozšíření využívání JavaScriptu i na serverové straně, avšak PHP všeobecně zůstává ve středu

pozornosti, co se týče využívání pro tvorbu webových aplikací. Toto potvrzuje průzkum W3Techs, který zkoumá technologie použité na webových stránkách, které se umístili v prvních 10 milionech dle návštěvnosti. Z toho PHP využívá téměř 8 milionů nejnavštěvovanějších stránek světa. Za zmínku stojí i pokles jazyka Perl, který byl zmíněn v kapitole o historii vývoje.

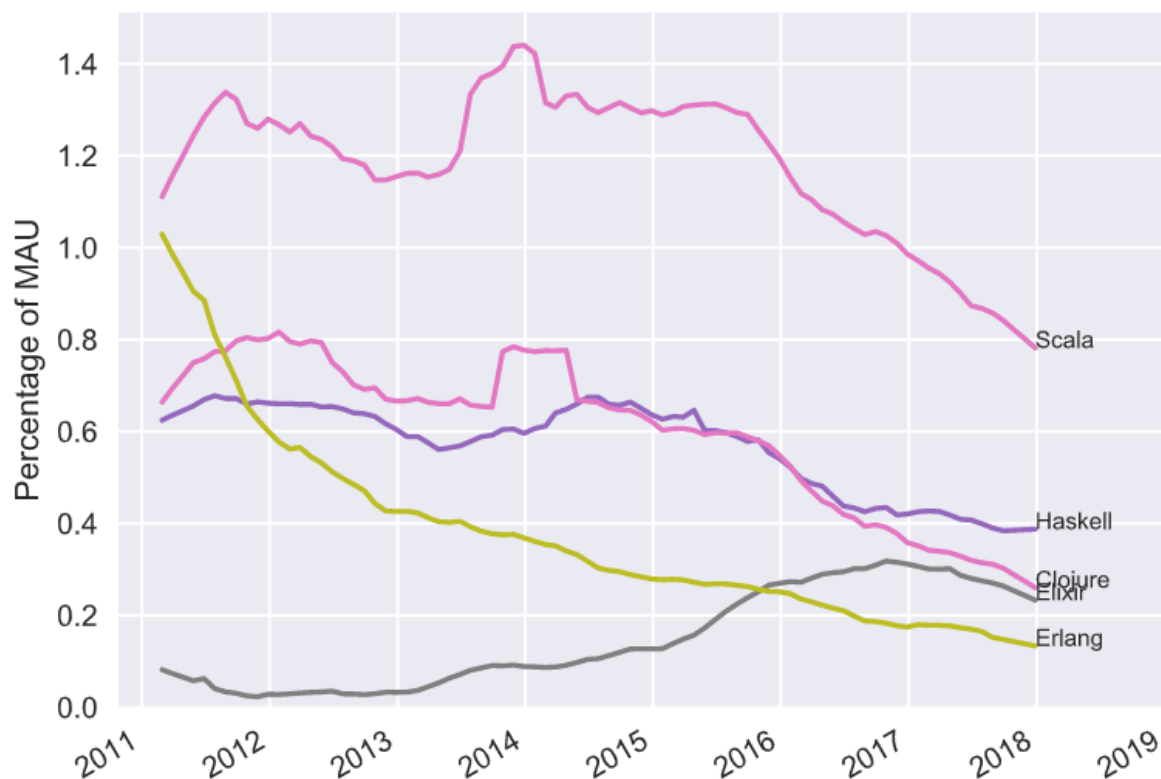


Obrázek 5 Úbytek aktivních uživatelů



Obrázek 6 Průzkum společnosti W3Techs

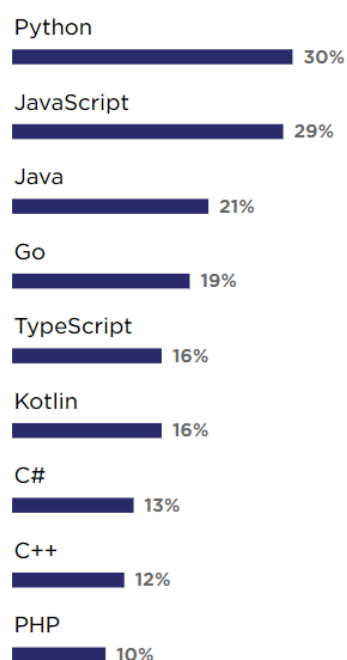
Poslední zajímavou statistikou je také přehled oblíbenosti funkcionálních jazyků, které byly v popředí zájmů i pro vývoj serverových aplikací. Z grafu je patrné že největší boom pro funkcionální jazyky byl mezi roky 2013–2015. Avšak současný trend pro všechny funkcionální jazyky je dlouhodobě klesající.



Obrázek 7 Počet aktivních uživatelů funkcionálních jazyků

Výše zmíněné statistiky byly spíše zaměřeny na technologicky orientována, tzv. tvrdá data. Pro srovnání jsem zvolil několik průzkumů mezi samotnými vývojáři, které by měli korelovat s výše uvedenými statistikami. První statistikou je každoroční průzkum mezi vývojáři, který provádí společnost JetBrains. Ankety byly k roku 2018 a částečné výsledky za rok 2019. Celkem v ní bylo dotazováno přes třináct tisíc vývojářů a ve statistice bylo zastoupeno 58 % backend vývojářů. Jednou z otázek byl dotaz, který jazyk se začnou učit, nebo budou pokračovat v jeho osvojování v roce 2019. Výsledky korelují s výše uvedenými statistikami, oproti nim zde ale více dominuje jazyk Kotlin, který předběhl mnoho známých a populárních jazyků. Tento jev mohl být způsoben lehkým zkreslením díky zacílení sběru dotazníků v okruzích uživatelů či jinak spřízněných osoby s firmou JetBrains. Firma JetBrains vyvíjí jazyk Kotlin, tím pádem je vyšší pravděpodobnost že dotazník zasáhl právě velkou část Kotlin komunity a také Java

vývojáře pro které firma vytváří IDE Idea IntelliJ, jelikož dle dat se většina příznivců Kotlinu rekrutuje právě z Java komunity. Mezi roky 2018 a 2019 v dotazníku významně vzrostl počet uživatelů Kotlinu, v roce 2018 ho aktivně používalo pouze 9 % dotazovaných a 13 % se ho chystalo využívat. Následující rok bylo již 16 % aktivních uživatelů a dalších 10 % se ho chystá v budoucnu využívat. Průzkum také ukazuje, jaké další technologie vývojáři využívající Kotlin ovládají, dominují dva jazyky, Java s 86 % a JavaScript, který ovládá 61 % uživatelů Kotlinu. Téměř 62 % uživatelů využívá Kotlin pro vývoj mobilních aplikací, tedy míří vývoj na platformu Android. Na JVM míří vývoj 57 % uživatelů, z toho celkového počtu je pro serverový vývoj využíván ve 41 % a pro ostatní použití pouze v 16 %. Kotlin je v 96 % případů nasazení využíván pro nové projekty, ve zbylém počtu se jedná o již existující projekty. Co se týká úrovně zkušeností vývojářů, tak Kotlin využívá aktivně méně jak 2 roky téměř 84 % vývojářů, pouze 1 % využívá Kotlin déle jak 4 roky, tyto data jsou k roku 2019.

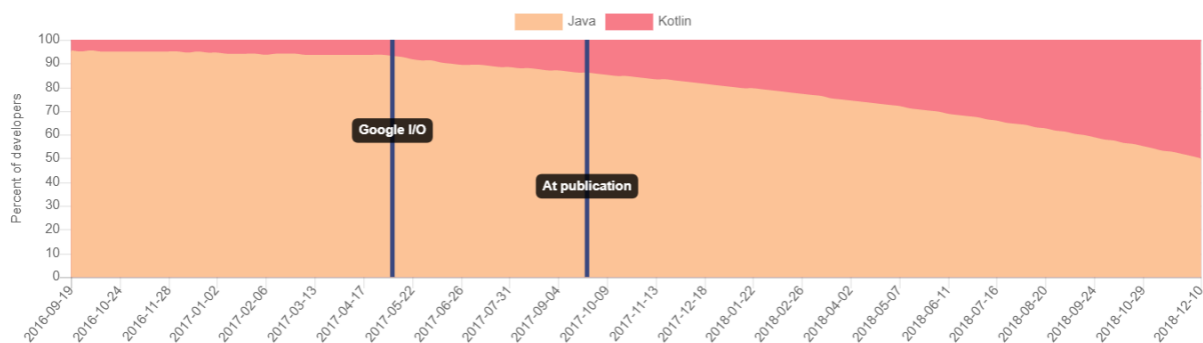


Obrázek 8 výsledky průzkumu State of Developer 2018 - zájem o osvojení nového programovacího jazyka

Dalším uživatelským průzkumem, který probíhá již dlouhodobě a s každoroční pravidelností je dotazník internetového vývojářského fóra StackOverflow, oproti JetBrains statistice není okruh dotazovaných uživatelů tolik vymezen, jelikož fórum navštěvují různí vývojáři bez ohledu na využívané technologie. Pro porovnání jsem vybral statistiky za roky 2018 a 2019 aby se dal pozorovat meziroční rozdíl v hodnotách. Dotazník vyplnilo v roce 2018 100 tisíc vývojářů

a v roce 2019 necelých 90 tisíc vývojářů po celém světě. Více než polovina vývojářů byla zaměřena na backend, stejně jako u dotazníku JetBrains. Nejvíce zastoupenou kategorií vývojářů, byli ti, kteří mají praxi 4-8 let, což odpovídá seniornímu profilu vývojáře. Co se týká technologií 67 % vývojářů ovládá JavaScript, který poklesl o 3,5 %, 41 % Javu, která poklesla o 4 %. Python 41,7 %, který si polepšil o 4 %. C# 31 % a poklesl o 4 %, PHP 26 % a pokleslo o 5 %. Naopak zaznamenal výrazný procentuální růst jazyk Kotlin 6,4 % s nárůstem o 2 %. Další sekce mapovala oblíbenost webových frameworků. Meziročně se u téměř každého produktu zvýšila obliba, avšak pořadí zůstalo stejné. Jako nejpoužívanější framework byl Node.js s 49 %, druhý byl .NET s 37 %, třetí Spring s 16 % u něj došlo k procentuálnímu úbytku, čtvrtý skončil Django s 13 %, což je framework pro jazyk Python. Ve statistice, kde se dotazovali vývojářů, v jakém jazyce by si přáli vyvíjet, zvítězil Python s 25 %, druhý JavaScript s 17 %, třetí Go s 15 % a čtvrtý Kotlin s 11 %. Pořadí zůstalo během roku poměrně beze změn a dá se říct, že žádný výrazný výkyv v přání vývojářů nenastal.

Oba průzkumy potvrdily trendy, které byly patrné z analýzy provedené na tvrdých datech, a to že vývojáři preferují nové, progresivní jazyky, které jim poskytují nové možnosti pro tvorbu serverových aplikací a nezdráhají se využít i poměrně mladé technologie pro produkční nasazení. V současné době jsou využívány hlavně jazyky PHP, Java, C#, JavaScript pro tvorbu serverových aplikací. Avšak mimo tyto zažité technologie se derou, do středu zájmu také méně rozšířené, nebo nové technologie jako je Python, Kotlin, avšak statistiky ukazují že mají potenciál konkurovat již zavedeným jazykům. V případě Pythonu se očekává že doroste, či dokonce přeroste tradiční technologie v řádu několika let, pokud bude pokračovat v současném tempu růstu. U Kotlinu je situace o něco komplikovanější, protože se jedná o jazyk úzce spojený s Javou, avšak v rámci prostředí JVM se jedná o podobný děj jako v případě Pythonu, kdy počet nových projektů zakládaných v Kotlinu se značně zvyšuje oproti počtu těch, které jsou v Javě, na platformě Android již Kotlin převyšuje Javu a má k říjnu 2018 podíl téměř 51 %.



Obrázek 9 Využití Kotlinu a Javy v nových projektech

## Trendy ve vývoji serverových aplikací

V době, kdy se již dostáváme na hranice možností křemíků a škálovat výkon aplikaci, je nutné jinými způsoby než jen pouhým navyšováním výkonu hardwaru. Ke slovu přichází paralelizace a distribuované zpracování, které se čím dál častěji implementuje právě v serverových aplikacích. Trend současnosti je stavět distribuované systémy, které umožňují paralelizaci zpracování a snadné škálování výkonu. V dnešní době toto reflektují i programovací jazyky, které se snaží podporovat a usnadňovat vývojářům paralelizaci např. Go, Kotlin mající řešení, která jsou oproti standardním jazykům a jejich vláknům poměrně nenáročná na zdroje a dají se poměrně masivně škálovat.

Zvládání komplexity velkých aplikací je hlavní výzva pro vývojáře softwaru a architekty, kteří systém navrhují. Podrobněji se touto problematikou zabývali Moseley a Marks v publikaci *Out of the Tar Pit*, kde navrhují nové způsoby zvládání složitosti rozsáhlých systémů. Navrhují rozdělení na základní logiku a také základní a vedlejší stav. V tomto modelu se bussines logika nezabývá stavy, ale pouze definuje relace, integritní omezení a provádí čisté funkce bez side-efektů. Naopak změny stavů neovlivňují logiku aplikace, ale pouze spouštějí akce (funkce bussines logiky) na ostatní elementy v systému. Tento přístup je v podstatě podporován funkcionálním programováním, které je ovšem v ryzí podobě velmi ortodoxní přístup, jak izolovat logiku od stavu a není v současnosti příliš populárním paradigmatickým v programování. Tvůrci moderních programovacích jazyků, ale berou v potaz výhody a usnadnění, které nabízejí deklarativní paradigmata, potažmo funkcionální přístupy k řešení algoritmických problémů. V současnosti se stává populárním aplikovat je v odlehčenější formě do objektově orientovaných jazyků. Na toto reagovala i Java a ve verzi 8 z roku 2014 přidala poměrně

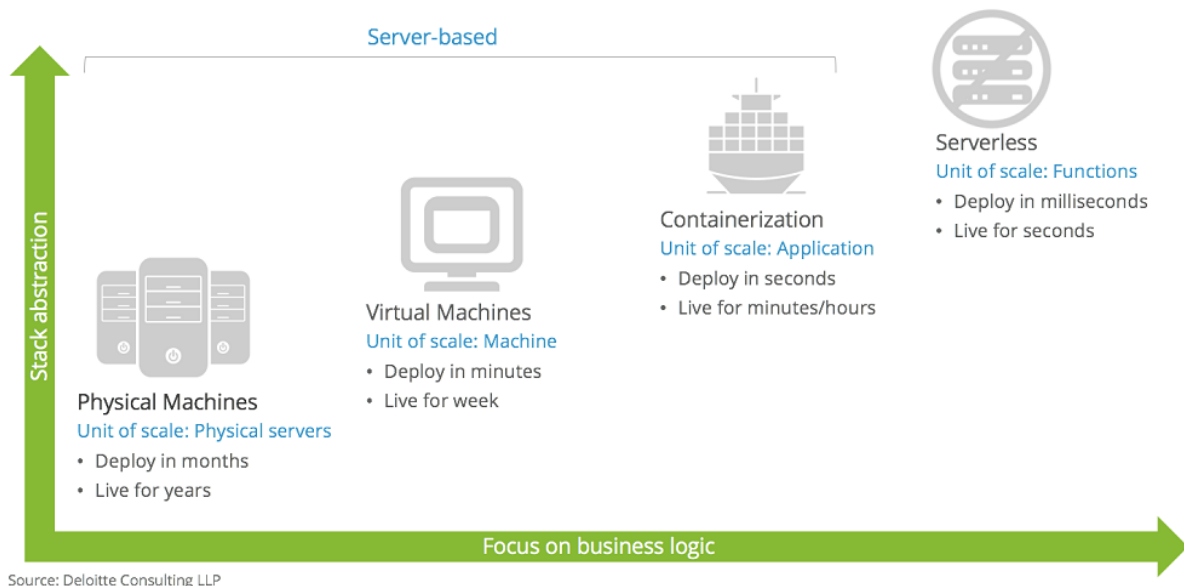


inovativní možnosti do jazyka v podobě lambda výrazů, referencí na metody, streamové zpracování. Některé jazyky jsou více otevřené a podporují funkcionální přístupy v rámci multi-paradigmatického prostředí, což je kompromis oproti čistě funkcionálním jazykům. Současným trendem je zvyšování expresivity jazyků. Mezi roky 2013 a 2015 nastala renesance funkcionálních jazyků díky jejich imutabilitě a přístupu k paralelizaci, na JVM platformě zažil vzestup jazyk Scala, avšak neujal se tak široce. Některé frameworky pro vývoj webových aplikací zvyšují expresivitu do té míry, že se z nich postupně stávají spíše DSL jazyky, které řeší problémy na vyšší úrovni abstrakce a odstiňují vývojáře co nejvíce od technologických detailů tak aby se mohli plně soustředit na řešení vlastní aplikační logiky.

Nejen samotné vývojové technologie ovlivňují trendy, inovace zaznamenalo také prostředí do, kterého jsou serverové aplikace nasazovány a v níž následně běží. V prvopočátcích, kdy se aplikace nasazovali přímo na fyzické stroje manuálně a s postupem času také automaticky, trvalo nasazení aplikace poměrně dlouho dobu a bylo značně komplikované. Postupně se prostředí začala více odstiňovat od fyzického stroje virtualizací, kdy na jednom či více fyzických serverech běželi virtuální instance serverů. V tomto desetiletí se začal využíval běh přímo v kontejnerech, které umožnili odstínění až na úroveň jednotlivé aplikace. Kontejnerizace řeší problém s konfigurací napříč virtuálními instancemi. Díky tomu je možné aplikace nasazovat automaticky v jednotkách minut, nezávisle na prostředí a jeho nastavení. Jelikož aplikace využívá jednotně nakonfigurovaný kontejner, ve kterém je odstíněna od složitého nastavování. Nejčastěji využívanou aplikací pro kontejnerizaci je Docker.

Ve světě serverových aplikací se poměrně často začíná uplatňovat trend serverless přístupu, kdy jsou odstíněné hardwarové servery a kdy aplikace běží v cloudu. Tedy infrastrukturu (IaaS), platformu (PaaS), nebo funkce (FaaS) zajišťuje poskytovatel služby a uživatel platí přímo za čas běhu aplikace, případné množství využitých zdrojů. V dnešní době je na trhu několik leaderů, Amazon se svým AWS, Google s Cloud Platform a Microsoft Azure. Serverless aplikace jsou většinou stavěny jako kompozice mikroslužeb či funkcí, avšak některé aplikace jsou i monolitické.

**Figure 1: Road to Serverless Solutions**  
Paradigm shift in computing evolution



<https://deloitte.wsj.com/cio/2017/11/09/serverless-computings-many-potential-benefits/>

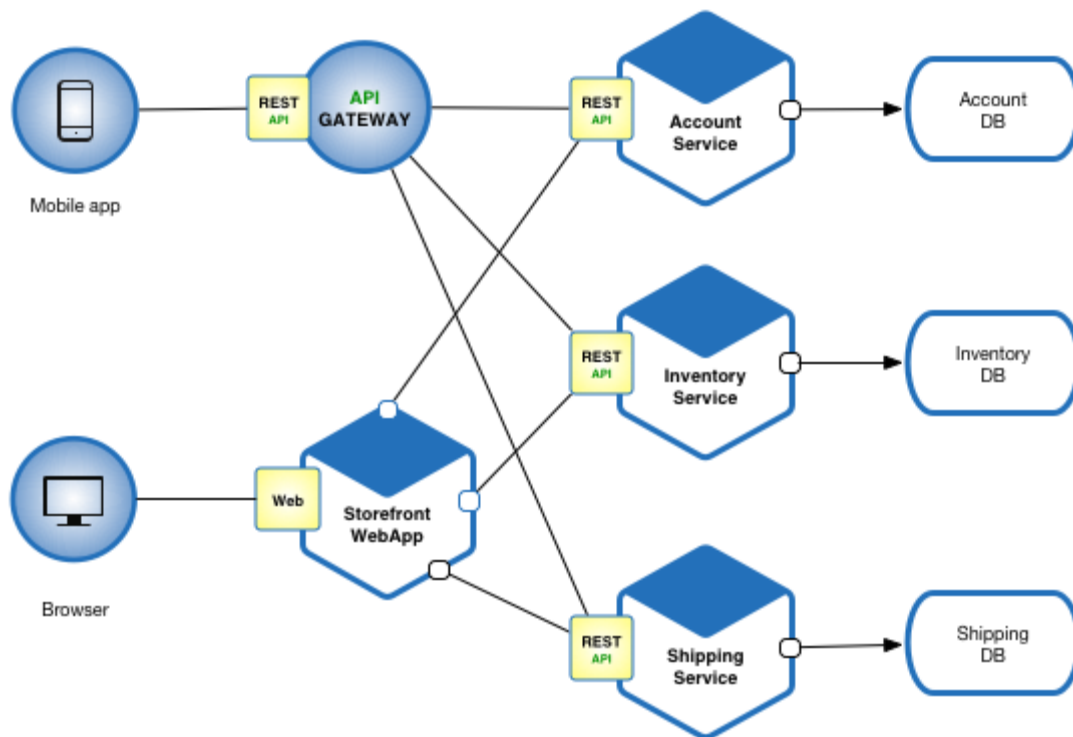
Microservices architektura je jedním ze současných trendů při vývoji serverových aplikací, kdy místo velké monolitické aplikace, vytvoříme kompozici více menších autonomních služeb, které umístíme do kontejnerů a poté mezi sebou propojíme. Vychází z konceptu SOA, tedy servisně orientované architektury. Jednotlivé mikro služby se skládají do kompozic a vzájemně mezi sebou komunikují. Pro orchestraci se dnes využívá populární nástroj Kubernetes, který dokáže zajistit kompletní management kontejnerizovaných aplikací včetně jejich monitoringu. Tuto architekturu podporují mnohé frameworky pro tvorbu serverových aplikací a poskytují také svoje proprietární nástroje pro orchestraci služeb například Spring Cloud a Data Flow. Nasazení mikroslužeb snižuje složitost systému, umožňuje snadněji upravovat a testovat aplikace.<sup>3</sup> Dále usnadňuje škálovatelnost vytížených služeb a také podporují principy continuous delivery a principy DevOps<sup>4</sup>, které se v současnosti stávají velmi populární. Na druhou stranu sebou přináší i všechny nevýhody distribuovaných systémů, pro vývojáře se jedná o složitější opravy chyb a jejich detekci, například při debugování. Na zvyšující se oblību

<sup>3</sup> Richardson, Chris. "Microservice architecture pattern". *microservices.io*. Retrieved 2017-03-19.

<sup>4</sup> Balalaie, Armin; Heydarnoori, Abbas; Jamshidi, Pooyan (May 2016). "Microservices Architecture Enables DevOps: Migration to a Cloud-Native Architecture". *IEEE Software*. **33**(3): 42–52. doi:10.1109/ms.2016.64. hdl:10044/1/40557. ISSN 0740-7459.

Chen, Lianping (2018). *Microservices: Architecting for Continuous Delivery and DevOps. The IEEE International Conference on Software Architecture (ICSA 2018)*. IEEE.

JVM micro-frameworků zareagovalo i nejpoužívanější<sup>5</sup> IDE mezi Java vývojáři Idea IntelliJ a ve verzi 2019.1.3, přidává podporu<sup>6</sup> pro mnoho z nich, mimo jiné i pro Micronaut.



<https://microservices.io/patterns/microservices.html>

<sup>5</sup> <https://www.baeldung.com/java-in-2018>

<sup>6</sup> <https://blog.jetbrains.com/idea/2019/08/whats-next-intellij-idea-2019-3-roadmap/#comment-499765>

## Web frameworky

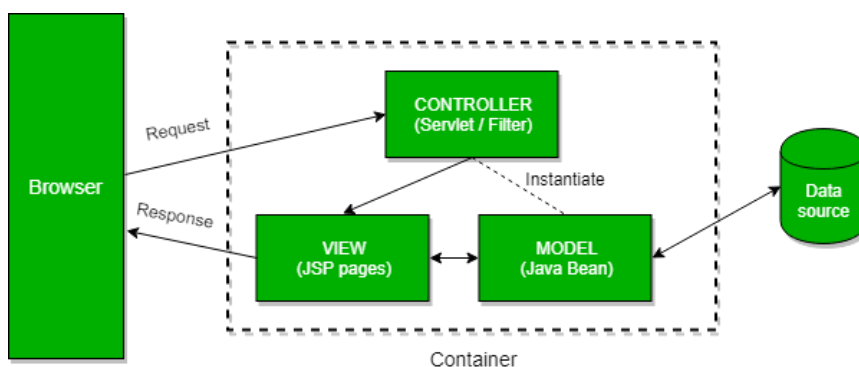
Framework je terminus technicus pro sadu softwarových nástrojů distribuovaných formou komplexního aplikačního rámce, který poskytuje vývojáři podporu při vývoji aplikací. Obecný význam existence frameworků je poskytnutí řešení pro typické problémy v oblastech, pro které je určen. Vývojář se díky nim může soustředit pouze na řešení problému na business úrovni a je odstíněn od implementace rutinních a opakujících se technických součástí aplikace, které jsou řešeny podle zavedných best-practices v dané oblasti. Díky tomu je při použití frameworku možné vývojáře snadno usměrňovat, vést jejich vývoj do stanovené architektury, odpadá velká část kontroly k dodržování best-practices, díky defaultní implementaci, a to vše vede k celkovému zlepšení kvality aplikace jako celku. Avšak framework vývojáře zcela nezprošťuje od řešení technických úskalí, spíše poskytuje vyšší úroveň abstrakce, či DSL<sup>7</sup> nad danou technickou doménou. V současné době jsou frameworky poměrně pokročilé a vyvíjeny profesionálními týmy, tudíž je zaručena efektivita a kvalita kódu ve kterém je framework napsán. Poměrně mnoho článků a diskuzí se zabývá použitím frameworků při vývoji aplikací. Nejčastější argumenty pro použití frameworku, kromě těch výše zmíněných je zvláště bezpečnost, která je zaručena velkým množstvím vývojářů, kteří se podílejí na vývoji a testování. Do testování je v podstatě zahrnuta celá komunita uživatelů, tudíž u vývojově pokročilého frameworku se již zásadní chyby téměř neobjevují a pokud ano, jedná se o drobné defekty, které jsou velmi rychle odhaleny a opravovány. Oproti proprietárním řešení poskytují frameworky většinou kvalitní dokumentaci a pro nově příchozí vývojáře je výrazně lepší učící křivka, navíc případně mohou využít svoje přechozí znalosti, jelikož frameworky jsou stavěny poměrně obecně aby v dané oblasti mohli pokrýt co největší část potřeb. V neposlední řadě je výhodou časová úspora téměř při všech fázích vývoje aplikace, která je při dnešních nákladech poměrně zásadní. Nevýhody frameworků je jejich obecné zaměření, kdy některé specifické potřeby při vývoji nelze jednoduše provést, například díky jeho omezením. Pro některé druhy řešení může framework vykazovat nedostatečný výkon, který nesplňuje požadavky. V neposlední řadě se jedná o určitou formu softwarového vendor-locku<sup>8</sup>, protože změna frameworku většinou znamená zásadní úpravu kódu aplikace.

---

<sup>7</sup> <https://whatis.techtarget.com/definition/domain-specific-language-DSL>

<sup>8</sup> <https://www.techopedia.com/definition/26802/vendor-lock-in>

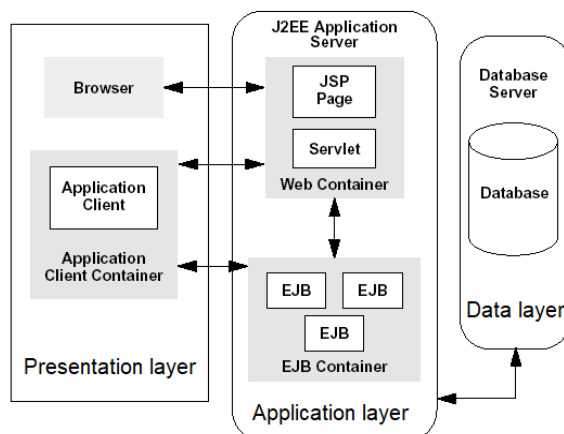
Webový framework jak je z názvu patrné je určen pro ulehčení tvorby webových aplikací na straně serveru. První webové frameworky se začali objevovat kolem roku 1995. Pro Javu se populární webové frameworky, které jsou využívány i v současné době začali objevovat až po novém miléniu. Webové frameworky primárně poskytují rozhraní mezi serverovou stranou aplikace a klienty, kterými jsou v současnosti nejčastěji různí front-end konzumenti (web, mobilní zařízení) reprezentující získaná data uživateli. Většina frameworků je vystavěna na jednotné architektuře, do které komponuje vývojář svoje řešení a která dotváří tzv. development guidelines a podporuje architektonické best-practices. V současnosti se nejvíce využívá model MVC (Model-View-Controller) a jeho variace. Jedná se o léty ověřený architektonický vzor, jehož hlavním významem je oddělit business data a logiku (Model) od jejich prezentace (View) a zpracování vstupů (Controller). Od tohoto vzoru jsou odvozeny další, které modifikují vzor do jiných kontextů např. MVP, MVVM. Tento vzor patří mezi tzv. push-based architektury, nebo také nazývané jako action-based. To znamená, že framework reaguje na akce, které zpracuje a následně zasílá (push) do view aby byly výsledky prezentovány.



Obrázek 10 Ukázka architektury MVC v Java EE  
zdroj: <https://www.baeldung.com/spring-mvc-interview-questions>

Většina webových frameworků dobře zapadá do tzv. třívrstvé architektury, který separuje celou aplikaci do nezávislých vrstev, které mají slabou vazbu tzv. loose-coupled. První vrstva je prezentační (front-end), která se stará o zobrazování dat. Druhá je aplikační vrstva (back-end), ve které se nachází business logika. A poslední vrstvou je datová (RDBMS), která uchovává data a tvoří abstrakci pro aplikační vrstvu. Každá vrstva komunikuje pouze se svými sousedy. Tento model je poměrně podobný výše uvedenému vzoru MVC, avšak tento vrstevnatý model řeší uspořádání celé aplikace. Webový framework se nachází v aplikační

vrstvě a vzor MVC, řeší její vnitřní strukturu. Lépe celou situaci dokresluje ilustrace na obrázku níže.



Obrázek 11 Třívrstvá architektura

Jak bylo výše zmíněno, webové frameworky dnes neposkytují pouze nástroje pro tvorbu rozhraní, ale zahrnují velkou škálu dalších modulů, které pokrývají téměř celou doménu vývoje serverových aplikací. Mezi nejčastější podporovaná rozšíření patří šablonovací knihovny, které usnadňují tvorbu statických HTML stránek na straně serveru, avšak v dnešní době většina knihoven umožňuje doplnění o AJAX, který podporuje tvorbu dynamických HTML stránek, díky využití asynchronního JavaScriptu, který běží na pozadí v prohlížeči a dodává webovým stránkám dynamiku bez nutnosti jejich znovu načítání ze serveru. První šablonovací jazyk pro Javu se objevil v roce 1999 jednalo se o JSP (Java Server Page), šlo o abstrakci nad servlety, do kterých jsou stránky překládány za běhu. Všeobecně se moc velké obliby nedočkali. V zápětí po vydání JSP se objevila další technologie JSF (Java Server Faces), která poskytovala celý MVC framework pro tvorbu webových aplikací v Javě. Pro úplnost je nutné uvést, že jazyk Kotlin nelze využít pro skripty uvnitř JSP stránek, avšak lze ho použít pro JSF. Daleko větší oblibu získaly šablonovací enginy třetích stran. Mezi nejznámější patří FreeMarker, Thymeleaf, které lze použít pro Kotlin i Javu. Thymeleaf šablony jsou oproti FreeMarkeru plně validní HTML stránky. Pro jazyk Kotlin v současné době neexistuje mnoho nativních variant, kvalitní knihovna, která stojí za pozornost je Kotlinx.html. Je vyvíjena přímo vývojáři Kotlinu. Oproti přechozím variantám má velkou výhodu a to tím, že umožňuje psát šablony ve stylu a syntaxi Kotlinu, což žádná jiná alternativa nenabízí. Pro srovnání jsem vyhledal výkonnostní test, který srovnává šablonovací enginy. Test byl proveden Luisem Durate a výsledek publikoval na

webu Dzone<sup>9</sup>. Autor článku je současně vývojářem šablonovacího enginu HtmlFlow, tudíž se dá předpokládat vysoká relevantnost testu. Autor se zaměřil na slabé stránky současných řešení, kde zmiňuje chybějící systém validací při generování HTML stránek, zvláště absenci statické validace již při kompilaci, namísto upozornění na chybu až při běhu aplikace, které jsou provázeny neočekávanými pády. Další nevýhodu spatřuje v nedostatečném výkonu, složité syntaxi, která vývojáře zbytečně rozptyluje a omezenou flexibilitu, kdy je u většiny frameworku poskytována pouze velmi omezená paleta možností pro kontrolu toku dat v šabloně a provádění komplexních akcí v šabloně.

```
fun studentTemplate(student: Student): String {
    return createHTMLDocument()
        .html {
            body {
                ul {
                    li { student.name }
                    li { student.number }
                }
            }
        }.serialize(false)
}

class KotlinTemplates {
    companion object {
        fun studentTemplate(student: Student): String { ... }
    }
}
```

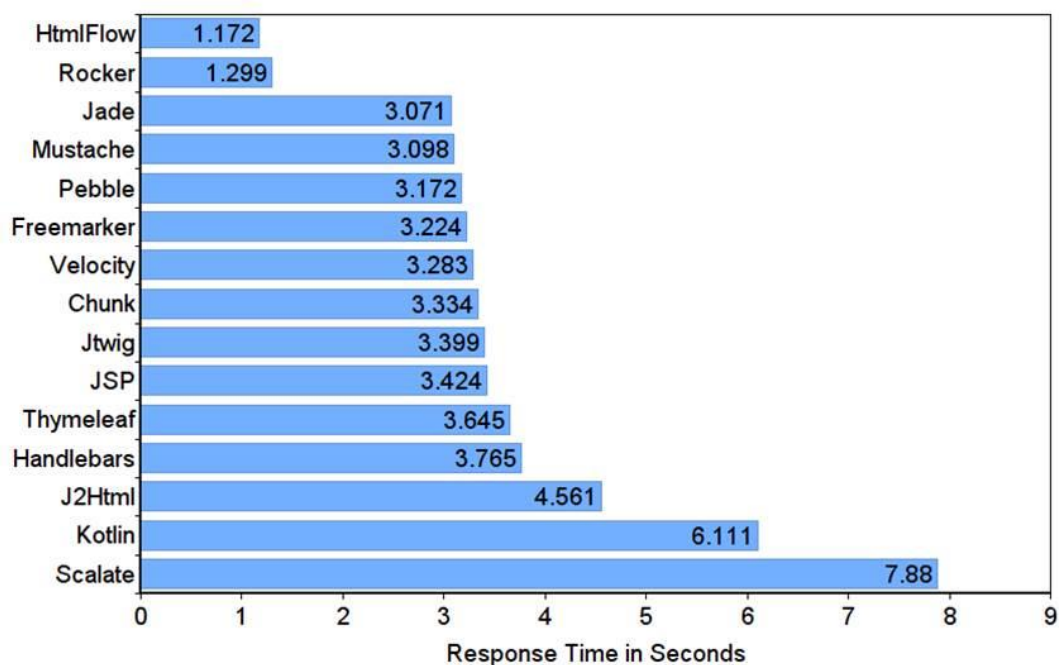
*Kód 1 Ukázka použití knihovny Kotlinx.HTML*

Knihovna Kotlinx.HTML značně snižuje nutnost použití textu, pro vytvoření HTML stránky, kdy je celá šablona psána přímo v Kotlinu. Knihovna je plně validní s HTML 5 a validuje správnou syntaxi, která je prováděna při kompilaci. Drobnou nevýhodou je absence validace atributů, které mohou nabývat různých hodnot a ošetření je ponecháno na vrub vývojáři. Pro testování výkonu byly použity dva nejoblíbenější testy pro šablonovací enginy. První je Spring test<sup>10</sup>, který testuje engin pod nápoem 25 paralelních klientů, kteří dohromady odešlou 25 tisíc requestů. Měří se celkový čas, který zabere obsluhu tohoto množství.

---

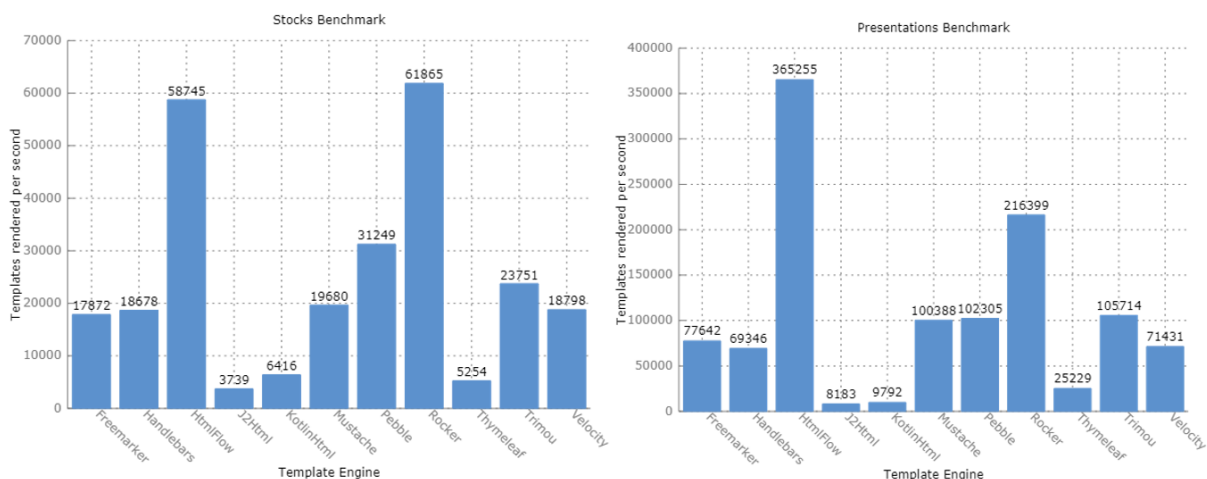
<sup>9</sup> <https://dzone.com/articles/modern-type-safe-template-engines>

<sup>10</sup> <https://github.com/xmlet/spring-comparing-template-engines#how-were-the-results-measured>



Obrázek 12 Výsledek Spring testu

Druhý test spočívá v rychlosti vykreslování šablon, využity jsou dvě. První šablona Stocks je více zaměřena na tzv. binding (napojení datových položek do šablony a naopak) a má také více dat (20 objektů). Druhá šablona Presentation má pouze 10 objektů. První šablona tedy více ověří práci se řetězci a více prověří výkon díky mnoha voláním metod.



Obrázek 13 Test rychlosti vykreslování

V těchto testech nevyšla, co se týká výkonu knihovna Kotlinx.HTML nejlépe, avšak je třeba brát v potaz, že oproti vítězi ve funkčnosti zaostávala pouze v absenci validace atributů a na rozdíl od všech testovaných enginů je plně v duchu jazyka Kotlin, což velmi usnadní vývoj a vývojář se nemusí rozptylovat syntaxí a sémantikou šablonovacího enginu. Avšak pokud přechází



argumenty nedosahují kýžených přínosů a výkon je klíčovou prioritou je nejspíše vhodné poohlédnout se po jiném řešení a použít například, obecné šablonovací enginy i za cenu kompromisu v podobě menší palety funkčnosti a zvýšené složitosti vývoje komplexnějších šablon.

Při tvorbě webových aplikací se často potýkáme s problémy v rychlosti vracení odpovědi (HTTP reponses) na příchozí zprávy klienta (HTTP request). Zvláště pokud pracujeme s rozsáhlou databází, provádíme mnoho volání vzdálených služeb, či složité výpočty je efektivní znovupoužití vynaloženého úsilí na získání dat. V principu lze říci, že frekventované služby využívá mnoho klientů a mnoho z nich tvoří velké množství duplicitních požadavků. Je tedy nasnadě uchovávat výsledky v paměti, která umožňuje rychlé čtení a zápis. Díky tomu můžeme odbavit zprávy v klienta už v servisní vrstvě, aniž bychom využívaly vrstvu pro získávání dat (repository, service-client). Tato funkčnost se všeobecně nazývá cachování a paměť pro uložení cache<sup>11</sup>. Cache je oproti standardnímu uložišti určena na dočasné uchování dat. Data jsou v ní často uložena v upravené struktuře, nejčastěji ve struktuře klíč-hodnota, kdy uchovávaná hodnota může být určitá projekce modelu, či pouze primitivní datový typ. Oproti standardní paměti zvládně obsloužit větší množství požadavků ve velmi krátkém čase. Avšak oproti standardní paměti bývá značně drahá při velkých objemech dat, a navíc ke standardní paměti často přistupujeme přes mnoho kódu a infrastruktury, čímž vždy ztrácí oproti operační paměti. I v případě že je disk napojen přímo ve fyzickém stroji, díky rozdílným rychlostem sběrnice, kdy DRAM je až desetkrát rychlejší než připojení disku a dokáže přenést až 20 GB/s<sup>12</sup>. Pro vývojáře vystávají zásadní otázky a problémy, jak navrhnout algoritmus který bude řídit cachování, jež musí řešit integritu uložených dat v čase a uplaňovat pravidla pro uchovávání záznamů v paměti. Nejčastější pravidla pro uchovávání záznamů, které se v praxi používají jsou uchování dle četnosti použití (LFU), kdy jsou nejméně používané záznamy odstraňovány. Dále se používá mazání záznamů dle jejich posledního použití (LRU/MRU), kdy se používá buď mazání nejdéle použitých, či naopak mazání nejnověji použitých, tohoto se nejčastěji využívá v systémech, kde jsou nejstarší položky nejvíce přistupovány. Tyto pravidla se často aplikují ve vztahu k době života záznamu (TTL) či s počítadlem konkrétních akcí (např. přístupy, změny), v případě, že je nechceme limitovat časem. Velkou výzvou pro vývojáře je navrhnout robustní

---

<sup>11</sup> <https://searchstorage.techtarget.com/definition/cache>

<sup>12</sup> <https://superuser.com/questions/1173675/how-much-faster-is-memory-ram-compared-to-ssd-for-random-access>

systém, který hlídá integritu dat. Všeobecně se používají tři politiky pro řízení integrity dat. První je přístup write-through, kdy při každém zápisu dat do paměti je rovnou uložíme do cache, write-around obchází vkládní do cache při zápisu dat a záznam do cache vloží až při jeho prvním načtením z paměti, poslední způsob write-back při kterém jsou data zapsána do cache a až později se z cache uloží do trvalé paměti (asynchronní přístup), díky tomuto je zajištěn rychlejší zápis než v případě write-through. Dále je nutné zajistit správu již neaktuálních záznamů, které byly upraveny a jsou nekonzistentní (např. aktualizovány, nebo smazány). Všechny tyto způsoby však neřeší modifikaci záznamů z jiného místa v systému, ale v tomto případě se jedná o řešení na architektonické úrovni celého systému a přesahuje rozsah jedné webové aplikace. Pro rutinní použití ve webové aplikaci poskytují frameworky podporu pro cachování. Většina frameworků implementuje jednoduchá pravidla a hlídá integritu pouze v rámci záznamů v cache. Některé umožňují zaintegrovat sofistikovanější systémy třetích stran, které umožňují využití složitějších pravidel, avšak s nativním přístupem přes webový framework.

Moderní frameworky nabízejí možnost generování projektu, případně i celé vnitřní struktury a jednotlivých komponent automaticky, čímž dokáží šetřit vývojový čas, který je nutný při započetí projektu. Ušetří nás zvláště složitého komponování a konfigurování modulů frameworku a dalších knihoven, což bývá problematické, zvláště u rozsáhlých monolitických frameworků např. Spring. V případě tvorby microservices, kdy pro každou službu zakládáme projekt samostatně, může tato funkcionality ušetřit nezanedbatelné množství času.

Většina z frameworků také usnadňuje přístup k datům, které jsou uloženy v jiném systému. Nejčastěji jsou to databáze, či jiné webové služby. Pro databáze webové frameworky nabízí API, které odstiňuje vývojáře od přímé komunikace s ní, tyto prostředky se liší v úrovni abstrakce přístupu k datům. Nejvíce je vývojář odstíněn při použití technologií založených na ORM<sup>13</sup>, které mapují položky tabulek z databáze na entitní třídy a naopak. Při jejich použití je vývojář plně odstíněn od práce s databází, avšak nevýhodou je nižší možnost kontroly v komunikaci s databází, byť jsou dnešní ORM nástroje velmi dobře přizpůsobitelné. Mezi nejčastější podporované implementace pro ORM patří Hibernate, některé frameworky dokonce nabízejí implementaci pomocí vlastních řešení. Nižší úrovní je programová podpora práce s SQL, která umožňuje manuální tvorbu operací, avšak přináší jazykové konstrukty a

---

<sup>13</sup> <https://www.techopedia.com/definition/24200/object-relational-mapping--orm>

často i validaci. Mimo přímé podpory pro práci s databází poskytují frameworky podporu transakčního zpracování a databázových migrací.

Pro jazyk Kotlin momentálně neexistuje proprietární náhrada za Hibernate. Existuje několik pokusů o implementaci ORM přímo pro Kotlin. Jedním z pokusů, který stojí za zmínku je knihovna Ktorm, avšak stále je oproti Hibernate v poměrně ranném vývojovém stádiu, navíc je více zaměřena jako SQL DSL knihovna, kdy ORM je spíše okrajová záležitost. Všeobecně má Kotlin více knihoven, které se zabývají právě poskytnutím DSL pro relační databáze využívající jazyk SQL. Nejzralejší knihovnou a přímo od tvůrců Kotlinu je knihovna Exposed. Na základě vlastního subjektivního hodnocení mi přijde, že tato knihovna oproti Ktormu pracuje na nižší úrovni abstrakce a vyžaduje vyšší expresivitu ve zdrojovém kódu od vývojáře. Ktorm má navíc větší škálu funkčnosti a příjemnější syntaxi. Ktorm má potenciál značně přesáhnout Exposed, jelikož udělal obrovský pokrok, od první beta verze, která vyšla v prosinci roku 2018 a nyní o rok později je ve verzi 2.5, což svědčí o velké aktivitě autorů. Oproti tomu Exposed v současnosti ani není ve stabilní verzi 1.0, byť jeho vývoj probíhá od roku 2016.

Framework mají často podporu pro automatické mapování JSON z tříd a naopak. Čímž je vývojář odstíněn od zpracování a často jen deklaruje chování mapperu pomocí anotací nad položkami, či v konfiguračním souboru pro obecné nastavení chování. Toto je velmi podstatná vlastnost, kterou by měl framework umožňovat, jelikož při nejrozšířenější komunikaci pomocí REST se používají téměř výhradně data formátovaná dle JSON specifikace, jenž vytlačila XML formáty pro popis přenášených dat.

Aspektově orientované programování a scheduling jsou funkce, které frameworky vývojářům nabízí a jež usnadňují implementaci enterprise aplikací, kdy často nestačí objektové paradigma, nebo není úplně vhodné pro řešení pro business pravidel a technických řešení, která se prolínají napříč aplikací. Scheduling je funkce pro usnadnění automatických akcí, které jsou vykonávány v definovaném čase, často bývají implementovány ve stylu Linux nástroje Cron. Kdy vývojář pouze označí metodu a definuje u ní intervaly kdy se má spouštět, některé frameworky často umožňují další předdefinovaná pravidla. Výhodou je že vývojář se nemusí starat o implementaci samotného triggerování funkce, kterou přebírá framework.

Velmi významným dílem se webové frameworky podílejí také na zabezpečení aplikace. Většina z nich podporuje integraci využívaných autentizačních standardů jako je Basic, OAuth2. Autorizaci poskytují pomocí proprietárního řešení v kombinaci s autentizačním standardem, nebo přímo integrují autorizační standard jako je například OpenId. Mimo jiné frameworky

v rámci zabezpečení poskytují správu veřejné publikace a přístupy ke statickým zdrojovým souborům.

## Výběr webových frameworků

Pro výběr frameworku byla klíčovým kritériem podpora Kotlinu. Frameworky lze v tomto ohledu dělit na dvě skupiny, a to ty které jsou celé napsány v Kotlinu a ty, který ho podporují tím, že je jeho část napsána právě v Kotlinu. V současné době je nejvíce zastoupen druhý typ, což je způsobeno nedostatečným množstvím zralých frameworků, které by byly vytvořeny pouze pro Kotlin. Do srovnání jsem vybral frameworky, které se vzájemně liší buď svojí filozofií přístupu, způsobem implementace, zralostí, nebo jiným signifikantním znakem. Jeden z nejznámějších webových frameworků pro JVM, kterému je v současnosti věnována velká pozornost je Eclipse Vert.x, který za sebou v testech nechává ostatní frameworky díky svému výbornému výkonu se řadí mezi ty nejrychlejší, které JVM hostí. Dalším frameworkem je zástupce těch, které jsou napsány celé v Kotlinu a jsou určeny pouze na něj. Jedná se o Ktor, což je rozsáhlý framework přímo od tvůrců Kotlinu, avšak oproti jiným jeho vývoj ještě nedosáhl vysokého stupně zralosti. Mezi další nativní webové frameworky se řadí http4k, který je naprosto v minimalistickém provedení, jedná se spíše o komplexní sada nástrojů pro HTTP, a proto je zmíněn okrajově jako zajímavá alternativa k ostatním zmíněným řešením a může být adekvátní náhradou Servletu v Kotlinu. Poslední dva frameworky jsou napsány v Javě a jsou specifické jejich přístupem a rozsahem. První z nich jsou Spark a Javalin jedná se o minimalistické framework, který je oblíbený mezi vývojáři. Další je Micronaut, který se vyznačuje svým zajímavým přístupem, který spočívá v odstranění proxy a běhové reflexe, navíc se jedná o poměrně robustní řešení s primárním zaměřením na budování microservices, avšak stejně jako v případě Ktor se jedná o ranný vývojový stupeň. "

## Kritéria pro hodnocení frameworků

Pro hodnocení frameworků byla stanovena kritéria pro usnadnění jejich vzájemného porovnání a pro případné ulehčení volby při výběru. Kritéria a váhy hodnocení byly zvoleny na základě rešerší odborných zdrojů, vlastní praxe a v neposlední řadě po konzultacích se specialisty, kteří se v praxi zabývají programováním webových aplikací. Tyto kritéria jsou tedy více praktičtějšího charakteru. Avšak díky tomu mohou být výsledky hodnocení lépe aplikovatelné přímo pro praktické využití, tedy usnadnit rozhodování při výběru webového frameworku, či zdali má vůbec smysl Kotlin použít pro vývoj na straně serveru. Hodnocení je ve čtyřech stupních kdy 0 znamená že framework toto kritérium vůbec nesplňuje, hodnocení 1 znamená že splňuje ale s velkými výhradami, hodnocení 2 je uděleno, pokud framework splňuje, avšak se nejedná o nejlepší možné řešení, spíše o průměrné naplnění. Nejvyšší stupeň 3 je udělen, pokud framework výrazně převyšuje standard v hodnocené oblasti. Pro hodnocení frameworku byla využita metoda Fullerova trojúhelníku. Tabulka s váhami v podobě Fullerova trojúhelníku je zobrazena v příloze.

### Podpora Kotlinu

Toto kritérium zkoumá, jak moc framework podporuje Kotlin a do jaké míry. Zda jde pouze o tenkou fasádu, která umožňuje použít Kotlin tak, že hlavní komponenty, se kterými vývojář komunikuje jsou s Kotlinem použitelné bez složitého ošetřování, jako je například potřeba v případě integrace s Javou, což způsobuje zbytečné množství kódu navíc. Zvláště pozitivně jsou hodnoceny frameworky, kde je vývojář kompletně odstíněn od Javy, nebo dokonce kdy je framework kompletně napsaný v Kotlinu. Dalším významným pozitivním vlivem na hodnocení je podpora nejnovějších funkcionalit Kotlinu přímo ve frameworku, anebo podpora vývoje aplikace dle zvyklostí a stylu zápisu pro Kotlin, který je jiný než u jazyka Java.

### Moduly

Kritérium nazvané moduly je souhrnné, jež obsahuje několik pod-kritérií, která hodnotí framework napříč jeho poskytovanými funkcionalitami. Vybrány jsou ty, které se nejčastěji využívají pro tvorbu serverových aplikací. Seznam modulů rozhodně není vyčerpávající, avšak

pokrývá dostatečně širokou paletu funkcionality, která se využívá pro běžně vytvářené enterprise aplikace na serverové straně.

### *Web*

V tomto kritériu se hodnotí, jaké technologie framework podporuje pro vývoj webových aplikací. Jakou nabízí podporu práce v rámci HTTP např. hlavičky, těla zpráv, parametry, podpora HTTPS a HTTP/2. Dále práci se sockety, podpora nahrávání souborů, správa cookies a sessions, poskytování statického obsahu, obsluhu routování a správa status stránek. Mimo jiné také podporované typy rozhraní jako je REST, SOA, GraphQL. Hodnocena je i podpora jazykových mutací webové aplikace a systémů pro tvorbu automatické dokumentace.

### *Security*

Úroveň bezpečnosti je hodnocena podle rozsahu pokrytí oblastí, které framework chrání a také množství technologií se kterými framework v rámci integrace umí spolupracovat. Dále je hodnocena míra flexibility, do jaké si vývojář může přizpůsobit zabezpečení dle vlastních potřeb a zintegrovat frameworkem nepodporovaný systémem, či případně si vyvinout čistě vlastní systém zabezpečení.

### *Templating*

U podpory šablonovacích systému je hodnoceno množství podporovaných enginů. Pokud framework disponuje proprietárním řešením je hodnocena jeho vlastní implementace. Mimo jiné i to, zda podporuje výše zmíněnou knihovnu Kotlinx.html.

### *Caching*

U cachingu je hodnocena vlastní implementace. V níž se zkoumají možnosti a funkce proprietárního řešení. Hodnocena je také podpora externích systému pro cachování a možné kombinace s proprietárním řešením.

### *Dependency injection*

Toto kritérium zhodnocuje přístup frameworku pro podporu volných vazeb mezi komponenty a zlepšení jejich testovatelnosti. Kdy se bere v hodnocení potaz, jaké technologie framework podporuje pro DI, či zda dokonce nabízí vlastní proprietární řešení. To je hodnoceno z hlediska snadnosti použití vývojářem a také z pohledu jeho vnitřní implementace.

### *JSON*

Kritérium, které hodnotí podporu automatické serializace a deserializace instancí tříd do formátu JSON a naopak. Hodnotí se podpora automatického zpracování, množství knihoven, které pro toto zpracování framework podporuje. Mimo jiné je zohledněna složitost konfigurace.

### *Integrace*

Úroveň integrace je hodnocena podle množství technologií, které framework podporuje a umožňuje snadnou integraci. Nadstandardně jsou hodnoceny proprietární programová rozhraní a implementace poskytované frameworkem k integračním technologiím, jež usnadňují jejich využití v rámci kódu aplikace. V hodnocení je také zohledněno, pokud framework umožňuje snadnou integraci nepodporovaných nástrojů přímo vývojářem, případně podpora alternativních implementací pro konkrétní druh integrace.

### *Přístup k datům*

U podpory přístupu k datům se hodnotí, zda framework poskytuje vlastní implementaci pro přístup k datům, hodnotí se také jeho implementace a snadnost použití vývojářem. Hodnocena je také škála podporovaných externích knihoven pro přístup k datům. Dalším kritériem hodnocení je počet podporovaných technologií zprostředkujících uložení dat.

### *AOP*

U tohoto kritéria se hodnotí, zda framework poskytuje podporu pro aspektové programování. Hodnotí se, jestli jde o proprietární implementaci, či se jedná o podporu externích knihoven. U vlastní implementace se hodnotí její provedení a uživatelská přívětivost a robustnost řešení.

### *Scheduling*

Toto kritérium hodnotí, zda framework podporuje plánování periodických činností. Hodnotí se, jestli framework poskytuje vlastní implementaci, či podporu jiných externích knihoven. Dále se hodnotí robustnost a flexibilita podporovaného řešení a jeho uživatelská přívětivost.

### *Testovatelnost*

Testovatelnost je kritérium, které hodnotí podporu frameworku v oblasti testování. Hodnotí se množství podporovaných testovacích frameworků, proprietární nástroje a knihovny, které framework poskytuje pro jeho testování. Hodnocena bude i celková struktura a architektura, z hlediska její testovatelnosti napříč všemi vrstvami aplikace. Do této oblasti zasahuje také podpora mockování a snadnost zajištění izolace pro jednotkové testování. Mimo jednotkové testování bude hodnoceno provádění e2e testů, které na rozdíl od jednotkových testují celé workflow, tudíž bude hodnocena podpora a snadnost napojení na testovací úložiště přímo v testech.

### *Podpora a komunita*

Pro každého vývojáře a firmu je toto často klíčová vlastnost. Hodnocena je četnost vydávaných verzí, zralost frameworku, aktivita uživatelské základny, možnosti eskalace problémů a jejich řešení. Dále bude hodnocena aktivita komunity, a to na diskuzních kanálech a v četnosti kontribuce do frameworku.

### *Praxe*

Toto kritérium hodnotí přívětivost frameworku, k jeho uživatelům, tedy vývojářům. Hodnotí se kvalita dokumentace k frameworku. Množství informačních zdrojů a literatury a množství



kurzů. Dále je zahrnuta do hodnocení podpora frameworku ze strany vývojových nástrojů (IDE).

## Výkon

Toto kritérium hodnotí frameworky podle měření na základě několika výkonostních testů. Tyto testy jsou vyhodnoceny a dle průměrně dosaženého pořadí jsou rozděleny body.

## Vert.x

### Historie

První stabilní verze frameworku 1.0 se objevila již v květnu 2011. Hlavním vývojářem je Tim Fox, který framework vyvíjel během svého působení jako zaměstnanec ve firmě VMware. Jméno je odvozeno od počátečního jména projektu, které bylo Node.x. X v názvu má vyjadřovat jeho nativní podporu mnoha programovacích jazyků. Po čase se projekt kvůli možným právním konfliktům přejmenoval právě na Vert.x. V roce 2013 byl framework přesunut pod správu Eclipse Foundation. V roce 2014 získal framework cenu za nejinnovativnější technologii pro Javu v rámci JAX Innovation awards<sup>14</sup>. V roce 2016 opouští Tim Fox projekt a místo přebírá dlouholetý přispěvatel Julien Viet. V současné době je framework vývojově zralý a poslední verze je 3.8.1 z října letošního roku.

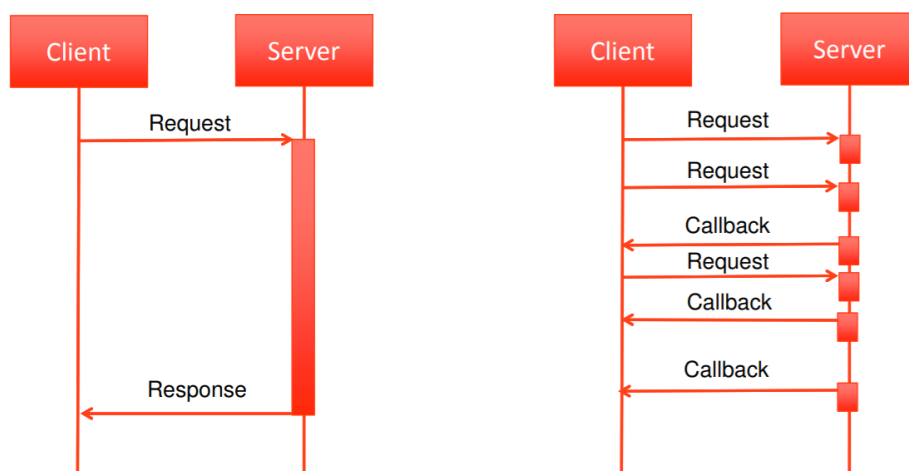
### Technologie

Mezi hlavní znaky frameworku patří výše zmíněná podpora mnoha jazyků. Framework je postaven na jedno vláknovém modelu. V tomto modelu je použito pouze jedno hlavní vlákno (Event loop) zachytávající události (HTTP requesty), které jsou zpracovávány asynchronně a

---

<sup>14</sup> <https://jaxenter.com/jax-innovation-awards-2014-champions-declared-107796.html>

vývojář je odstíněn od úskalí spojených s více vláknovým programováním, které by bylo nutné pro dosažení stejného reaktivního chování.



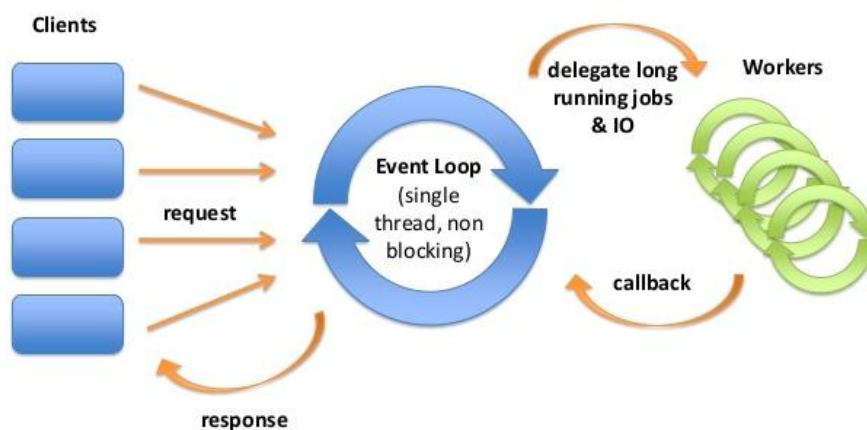
Obrázek 14 Synchronní a asynchronní (reaktivní) zpracování HTTP requestů

Vert.x využívá pro nízko úrovněvé IO operace knihovnu Netty, která je vyvíjena již od roku 2004 a v současné době vývoj stále probíhá. Knihovna je význačná tím, že neblokuje vstupní a výstupní kanály, díky asynchronní komunikaci. Netty je postaven na architektuře Reactor<sup>15</sup>. Reaktor byl popsán již roku J.Coplienem a D. Schmidtem v jejich společné knize Pattern Languages of Program Design. Tento vzor měl sloužit jako návrh obsluhy pro souběžné zpracování požadavků ze služeb, které jsou ale interně zpracovávány synchronně pomocí handlerů. Vert.x tento vzor přebírá a dále ho rozšiřuje na architekturu Multi-Reactor, ve kterém je využíváno více vláken zachytávající události, díky tomu že současné procesory obsahují více jader a může běžet více paralelních operací v jeden okamžik. Vert.x hlídá nadměrné blokování vláken a vývojáře na tuto skutečnost upozorňuje. Vert.x řeší tzv. C10k problem<sup>16</sup>, jež byl zmíněn Danem Kegelem v roce 1999. Problém spočíval v obsluze 10 tisíc otevřených spojení souběžně. S tímto mají webové frameworky, které zpracovávají HTTP requesty synchronně problém, jelikož při takovém náporu nezvládnou obsloužit v požadovaném čase a skončí tzv. timeoutem. Výhodu asynchronního zpracování dobře zachycuje Obrázek 14, kdy na první ilustraci je zaslán HTTP request, který se provede v čase T při synchronním zpracování. Na vedlejší ilustraci jsou tři totožné HTTP requesty, které zaberou stejnou dobu T, avšak díky asynchronnímu zpracování jsou odpovědi zasílány konstantně v čase T od zaslání z klienta na

<sup>15</sup> <http://www.dre.vanderbilt.edu/~schmidt/PDF/reactor-siemens.pdf>

<sup>16</sup> <http://www.kegel.com/c10k.html>

server díky tomu, že zde není žádné blokování přechozím requestem, takže všechny tyto requesty jsou souběžně zpracovány v celkovém čase velmi blízkému T pokud by byly odeslány všechny současně, oproti celkovému času o velikosti přibližně 3T v případě synchronního zpracování. V dnešní době se problém C10k již podařilo mnohonásobně překonat, a v roce 2010 byl stanoven nový problém C10M<sup>17</sup>, který je výzvou pro následující dekádu.

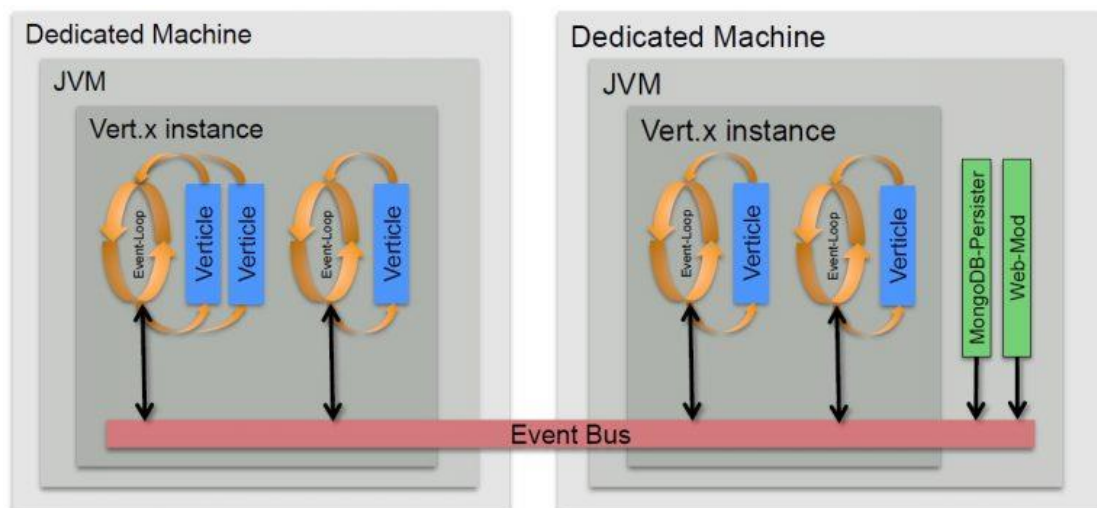


Obrázek 15 Architektura Reactor

<https://www.celum.com/de/blog/technologie/the-reactor-pattern-and-non-blocking-io>

V prostředí frameworku se často používá termín Verticle, jedná se o součást propracovaného distribuovaného systému, který framework nabízí. Verticle je základní stavební jednotka jedná se o modul, který sdružuje určitou část skriptů a jsou distribuovány pomocí class souborů, nebo jako balíček jar. Aplikace může obsahovat jeden nebo více těchto modulů. Komunikace mezi moduly probíhá přes proprietární Event Bus. Tyto moduly Verticle jsou spuštěny uvnitř Vert.x instance, která běží uvnitř JVM, tato instance dokáže paralelně obsloužit velké množství Verticles. Event Bus poskytuje abstrakci nad topologií, ve které jsou instance nasazeny proto je možné komunikovat pomocí sběrnice s moduly Verticles, které běží v jiné Vert.x instanci, ale i ty které jsou spuštěny v jiné JVM, avšak od tohoto je vývojář odstíněn a framework zajišťuje tuto komunikační režii. Kromě zpráv je také možné sdílet data mezi moduly Verticles, tohoto je ve Vert.x docíleno pomocí globalní cache s názvem Shared Map pro kterou Vert.x poskytuje metody a díky tomu ji lze využít kdekoliv v kódu aplikace.

<sup>17</sup> <http://highscalability.com/blog/2013/5/13/the-secret-to-10-million-concurrent-connections-the-kernel-i.html>



Obrázek 16 Možná topologie frameworku Vert.x  
<https://blog.oio.de/2016/07/29/springmvc-vs-vert-x/>

## Podpora Kotlinu

Kotlin je frameworkem plně podporován, což vychází z jednoho z pilířů Vert.x a to je cíl být polyglotický webový framework. Pro Kotlin je separátně implementována knihovna, která je ve formě modulu. Vert.x v současnosti podporuje využití novinky Kotlinu tzv. korutiny<sup>18</sup>, které umožňují psát asynchronní kód, který vypadá jako kód synchronní, jenž má analogické chování. Jedná se o poměrně čerstvou novinku jazyka, která byla do Kotlinu ve stabilní verzi přidána až v jeho verzi 1.3 pomocí doplňkové knihovny. Výhodou korutin je že využívají implementaci, která je nenáročně škálovatelná a díky tomu je možné v operacích tvořit velké množství paralelních operací, aniž by to výrazně zatížilo systém, což v případě vláken není možné, a navíc přinesou další problémy díky své asynchronnosti. Framework je tedy naroubován na využití korutin tím, že obsahuje třídy upraveny přímo pro Kotlin např. místo třídy Verticles, která se používá pro jazyk Java, je speciálně pro Kotlin upravená třída CoroutinesVerticle napsána tak že využívá korutiny místo normálních metod. Díky tomuto lze psát kód, který je plně ve stylu jazyka Kotlin.

## Hodnocení (0-3): 2

<sup>18</sup> <https://kotlinlang.org/docs/reference/coroutines-overview.html>

## Moduly

### Web

Webový modul u Vert.x je poměrně rozsáhlý. Oproti jiným frameworkům musíme lehce změnit paradigma zápisu kódu. V rámci Vert.x se vše nastavuje přímo ve zdrojovém kódu, nejsou zde anotace a stavba aplikace probíhá pomocí skládání kódu a případně jeho zanořování do handlerů. Tím je značně liší od klasických frameworků jako je například Spring. Práce s HTTP requesty je poskytována na přes programové rozhraní, kdy máme k dispozici celý request, který se framework zašle do odpovídajícího handleru, takže následné zpracování je plně v režii vývojáře a poskytuje mu značnou volnost. Vert.x podporuje HTTP/2 a HTTPS.

Routování je řešeno pomocí kompozice metod, které jsou řetězeny registrací voláním metody na globálním objektu Router. Jeho zpracování je velmi podobné přístupu, který se používá v Node.js. Vyhodnocování je prováděno v pořadí kompozice, lze dokonce použít regulární výrazy pro zachycení cesty. Díky tomu že vyhodnocování cesty je prováděno v jasně daném pořadí je možné vytvářet i orchestrace, kdy uvnitř handleru, vyvoláme pokračování vyhodnocování od současného bodu v kompozici. Tímto je možné dosáhnout velmi jednoduchého rozšíření funkčnosti např. pro ověření přihlášení uživatele, které se provede, před každým voláním, to lze provést z jednoho místa bez nutnosti zásahu do kódu na jiných místech. Velmi zajímavou možností je routování na základě typu obsahu HTTP requestu podle tzv. MIME<sup>19</sup>. Vert.x poskytuje podporu pro globální error handling, pokud není vývojářem upravena poskytuje framework proprietární obsluhu, včetně vlastních error pages. Podporuje nahrávání souborů, kdy vývojáře odstiňuje od řešení zpracování HTTP multipart<sup>20</sup> těla zprávy a poskytuje rovnou list se soubory. Statické soubory framework dokáže zpřístupnit automaticky pomocí předdefinovaného handleru, který se vloží do požadovaného mappingu pro statické soubory. Tento handler lze samozřejmě dále upravovat pomocí konfigurace při jeho vytváření (využívá návrhový vzor Builder<sup>21</sup>). Vert.x podporuje tvorbu multitenant<sup>22</sup> aplikací, rozlišování je založeno na bázi toho, že podle definované položky v hlavičce HTTP requestu, se předá do odpovídající větve routeru k danému tenantu, který je v aplikaci implementován

---

<sup>19</sup> <https://whatis.techtarget.com/definition/MIME-Multi-Purpose-Internet-Mail-Extensions>

<sup>20</sup> [https://www.w3.org/Protocols/rfc1341/7\\_2\\_Multipart.html](https://www.w3.org/Protocols/rfc1341/7_2_Multipart.html)

<sup>21</sup> <https://www.geeksforgeeks.org/builder-design-pattern/>

<sup>22</sup> <https://whatis.techtarget.com/definition/multi-tenancy>

pomocí třídy `MultiTenantHandler` a pro každý tenant obsahuje vlastní implementaci routeru. Vert.x nabízí i podporu pro WebSockets, která umožňuje plně duplexní komunikaci mezi klientem a serverem.

Vert.x podporuje práci s cookies a sessions. Pro oboje nabízí jednoduché programové rozhraní, které je dostupné z routovacího kontextu aplikace, který je předáván do handlerů. Globální `SessionHandler` je registrován před všemi dalšími route handlery, funguje v podstatě jako HTTP filtr.

Lokalizace je řešena pomocí automatického načtení z položky `Accept-Language` v hlavičce HTTP requestu, Vert.x poskytuje programové rozhraní pro usnadnění vyhodnocování podporovaného jazyka na klientu, pomocí metod, které jsou umístěny do requestu jenž je předáván do handleru. Samotná logika zpracování překladu už je plně v režii vývojáře.

Vert.x podporuje moderní rozhraní GraphQL pomocí interních modulů. Pro dokumentaci REST API poskytuje modul, který generuje dokumentaci ve standardu OpenApi3.

### **Hodnocení (0-3): 3**

#### *Security*

Pro zajištění bezpečnosti webové aplikace poskytuje framework Vert.x knihovnu. Knihovna poskytuje podporu pro autentifikaci uživatele a jeho autorizaci u autority. Framework má nastavenou velmi obecnou definici zabezpečení, tak aby bylo možné flexibilně přizpůsobovat požadovanému řešení a nebyla podřízena využití pouze vybraných způsobů autentifikací a autorit pro ověření uživatele. Autentifikace probíhá pomocí JSON objektu s informacemi o uživateli, který může být založen např. na bázi JWT Token, nebo OAuth bearer token, jenž je ověřen a namapován asynchronně a vývojář obdrží výsledek do handleru, kde může provádět akce po autorizaci. Obdobným způsobem probíhá i autorizace. Podporované standardy jsou BasicAuth, JWT, OAuth2, Shiro, .htdigest. Framework nabízí mimo podporované standardy i možnost vlastní implementaci ověřování, kdy vystavuje rozhraní `AuthProvider` pro vlastní implementaci autentifikace a abstraktní třídu `AbstractUser` pro implementaci vlastní autorizace. Framework přímo v core modulu poskytuje automatizovanou ochranu před CSRF

útoky<sup>23</sup>. Mimo výše zmíněné způsoby ochrany také poskytuje možnost nastavení šifrované komunikace s klientem pomocí protokolů SSL/TLS. Nabízí programové rozhraní, které umožní pohodlné nastavení přímo v kódu aplikace, které se nastaví před vytvořením serveru.

## **Hodnocení (0-3): 2**

### *Templating*

Vert.x podporuje velkou škálu templatovacích enginů. Jejich nastavení probíhá programově. Framework podporuje enginy MVEL, Jade, Handlebars, Thymeleaf, Apache FreeMarker, Pebble, Rocker. Výhodou je že framework zprostředkuje programové rozhraní pro předávání modelu do šablony, sám vyhledá šablony ve složce a vrátí ji jako tělo HTTP response.

## **Hodnocení (0-3): 2**

### *Caching*

Vert.X nepodporuje cache tak jak jsou vývojáři zvyklí ze Springu, kde je možné přidat anotaci nad metodu. Zde je nutné využít vlastní implementaci. Jedinou cache, kterou framework poskytuje je pro statické soubory, které lze cachovat automaticky. U této cache je možné provádět konfiguraci přímo v kódu aplikace. Avšak podporuje klienta pro enterprise in-memory storage jako je Hazelcast, Apache Ignite, Apache Zookeeper, Infinispan.

## **Hodnocení (0-3): 1**

### *Dependency injection*

Framework nemá vlastní implementaci pro dependency injection, ale podporuje integraci již existujících knihoven, které ji poskytují. Podporovány jsou Guice, HK2, Eclipse SISU. Dále je umožněna integrace se Spring DI, tak aby bylo možné využívat Spring Beans i ve frameworku.

---

<sup>23</sup> <https://www.zdrojak.cz/clanky/co-je-cross-site-request-forgery-a-jak-se-branit/>

## **Hodnocení (0-3): 1**

### *JSON*

Zpracování JSON je prováděno programově, v plné režii vývojáře. Framework mu poskytuje dvě třídy, které poskytují možnost automatické serializace a deserializace. Pro Kotlin je poskytnut builder, který využívá jeho funkce a je možné JSON zkonstruovat pomocí DSL, kdy zanořujeme jednotlivé části do sebe. Automatické parsery je možné programově nastavit. Celkově framework poskytuje dostatečnou paletu funkcí pro práci s JSON.

## **Hodnocení (0-3): 2**

### *Integrace*

Vert.x poskytuje mnoho integračních nástrojů. Základní je HTTP klient, dále pak MQTT klient. SOAP integraci framework neposkytuje. Framework poskytuje integraci s messaging systémy AMQP, STOMP, RabbitMQ, Apache Kafka. Vert.x lze integrovat s Java EE serverem pomocí knihovny JCA (Java Connector Architecture). Podpora integrace se SMTP servery je zajištěna Mail klientem. Integrovat tedy lze s většinou standardních a běžně využívaných systémů.

## **Hodnocení (0-3): 3**

### *Přístup k datům*

Framework obsahuje klienty pro integraci databázovými systémy. Podporovány jsou tyto PostgreSQL, MySQL, MongoDB, Redis, Cassandra. Pro PostgreSQL a MySQL jsou k dispozici i reaktivní verze klientů, byť v současné době pouze v preview verzi. Dále je obsažen JDBC klient. Vert.x podporuje také jOOQ což je knihovna DSL pro SQL. Framework nenabízí integraci s ORM systémem, avšak poskytuje jednoduchou knihovnu, která mapuje entitní třídy pro MySQL a MongoDB. Avšak oproti Hibernate se jedná o velmi odlehčenou verzi. Framework vývojáře spíše směřuje do využití DSL knihovny pro práci s databází.



## Hodnocení (0-3): 2

### AOP

Vert.x díky své architektuře a koncepci nepodporuje využití aspektově orientovaného programování.

## Hodnocení (0-3): 0

### Scheduling

Framework umožňuje vykonávat periodické a oddálené akce. Poskytuje pro toto programové rozhraní. V základu se jedná o velmi jednoduché řešení, které neposkytne podporu pro složitější plánování akcí. Větší možnosti se naskýtají při využití knihovny Vertx-Cron, která se nastavuje a zasílá notifikace v JSON po Event Bus, pro nastavení se využívá standardní Cron notace. Další knihovnou je Chime. Obě tyto knihovny jsou od externích tvůrců. Což není zcela optimální, a navíc nejsou již delší dobu aktualizovány.

## Hodnocení (0-3): 1

### Testovatelnost

Vert.x poskytuje několik možností, jak pokrýt projekt testy. Nabízí vlastní knihovnu vertx-unit, která usnadňuje testování asynchronního kódu aplikace. Avšak testy jsou poměrně nepřehledné a nelze zcela jednoduše dosáhnout izolovanosti tak jak se standardně provádějí unit testy v jiných frameworkcích. Mockování není jednoduše proveditelné. Je možnost integrovat interní testovací knihovnu s JUnit, avšak vždy se jedná o testování e2e, výhodou je aspoň značné zlepšení čitelnosti testů. Při testování se standardně využívá in-memory databáze.

## Hodnocení (0-3): 1

## Podpora a komunita

Framework je vývojově zralý, je kontinuálně vyvíjen od roku 2011. Nyní má silnou podporu Eclipse Foundation. Za dobu jeho existence bylo vydáno 94 releasů. Do projektu přispělo již 153 kontributorů, v současnosti je v repozitáři 58 pull requestů. V issue trackeru je v současné době evidováno 168 otevřených ticketů, kdy u téměř u každého je nějaká reakce, ze strany komunity, nebo přímo od vývojářů. Za dobu existence frameworku bylo vyřešeno 1482 ticketů. Verze jsou vydávány pravidelně, v dohledné době se chystá nová major verze 4, která přinese mnoho velkých změn. Vert.x má velmi silnou komunitu, která se vyskytuje na google user groups, která je velmi aktivní, dále se vyskytuje na serveru StackOverflow, kde má téměř 1700 otázek a 700 odběratelů tématu. Mimo tyto hlavní sítě je možnost kontaktu na Gitteru, což je real-time chat. Zajímavé informace o frameworku a novinkách autoři publikují na svém blogu a také na Twitteru, který sledují 3 tisíce odběratelů.

## Hodnocení (0-3): 3

## Praxe

Kolem frameworku Vert.x je zajištěna výborná dokumentace. Stránky jsou doplněny o návody, přehled modulů a jejich ukázky, včetně kompletních řešení různé problematiky s pomocí frameworku. Lze říct, že v tomto ohledu kompletně postačuje web frameworku, který obsahuje vyčerpávající informace a jsou velmi snadno k dohledání. Autoři mají ke každému podporovanému jazyku dokumentace zvlášť což značně zvyšuje přehlednost a kvalitu. Web odkazuje i na interní wiki, kde jsou detailně popsány všechny verze frameworku včetně změn, které obsahují. O frameworku už bylo i publikováno několik knih, dostala se i do série knih in Action<sup>24</sup>, které popisují technologie do nejmenších detailů a díky této knize je možné plně porozumět frameworku v ucelené formě. Na výukovém serveru Udemy lze také najít několik video kurzů zaměřených čistě na tento framework. Na webu je k dispozici rozhraní generátoru projektů, který je velmi přehledný a umožňuje vytvořit projekt přesně na míru s vybranými

---

<sup>24</sup> <https://www.manning.com/books/vertx-in-action>

moduly. V současné době v IDE IntelliJ není poskytována zvláštní podpora ani pomocí žádných pluginů, avšak v následující verzi je přislíbeno rozšíření v oblasti micro-frameworků, takže existuje šance, že bude framework podporován interními nástroji.

## Hodnocení (0-3): 3

### Ktor

#### Historie

První zmínky se o frameworku objevily kolem počátku roku 2016. První stabilní verze 1.0 byla vydána až v listopadu 2018. Tento framework je vyvíjen přímo týmem JetBrains jež vyvíjí jazyk Kotlin. V současné době je framework ve verzi 1.2.5, která byla vydána na konci srpna tohoto roku. V dohledné době je již vydána beta verze 1.3, tudíž lze v dohledné době očekávat novou verzi frameworku. Celkově jde o poměrně nový framework, jež za sebou ještě nemá bohatou historii.

#### Technologie

Ktor je plně implementován v jazyce Kotlin a je to jediný jazyk pro, který je framework určen. Od počátku je koncipován jako plně asynchronní, je postaven na korutinách. Díky tomu lze programovat lineárně, místo používání callbacků, kdy často vzniká tzv. callback hell<sup>25</sup>. Ktor v sobě implicitně obsahuje embedded server (Tomcat, Jetty, Netty, Corrutine I/O) pak je aplikace distribuována jako tzv. fat-jar, avšak toto lze přes nastavení potlačit a nasazovat WAR archiv do externího servlet kontejneru, který podporuje Servlet 3.0+ API. Zajímavostí je že Kotlin poskytuje vlastní knihovnu kotlinx-io pro asynchronní I/O operace, díky tomu není nutné využívat řešení třetích stran a lze tedy postavit celý projekt čistě v Kotlinu. Asynchronní zpracování je řešeno pomocí pipelines, které jsou složeny z fází (phases) a příjemců (interceptors). Standardní aplikační pipeline se skládá z fází: setup (příprava volání, zpracování

---

<sup>25</sup> <http://callbackhell.com/>

atributů), monitoring (logování, metriky, zpracování chyb), features (infrastrukturní zpracování), call (zpracování a odeslání odpovědi), fallback (obsluha nezpracovaných volání). Na každou fázi jsou napojeni příjemci, kteří ji zpracovávají v dané fázi. Fáze features je přizpůsobena na rozšíření o uživatelem definované příjemce. Aplikaci je možné jednoduše dělit na moduly přímo v kódu, kdy jednotlivé moduly se tvoří jako extension funkce nad třídou Application, tyto extensions se pouze zaregistrují při definici serveru aplikace. Celkově Ktor staví na myšlence volnosti pro vývojáře, tak aby si mohl zvolit, které technologie na projektu použije a nezavazuje ho využívat pouze proprietární řešení. Zajímavým rozšířením, které vývojáři nabízejí je knihovna pro klientskou stranu aplikace, jedná se o asynchronního HTTP klienta, jenž využívá multiplatformní podporu, která je v Kotlinu od verze 1.2 v experimentální verzi. Díky tomu je možné ji použít na JVM, Android, iOS (Kotlin/Native) a také v JavaScriptu (Kotlin/Js).

## Podpora Kotlinu

Framework je kompletně napsán v Kotlinu. Dále využívá všechny novinky, které jsou do Kotlinu přidávány a dají se využít pro vývoj serverových aplikací, a to i ty které jsou teprve v experimentálních verzích, tj. může u nich nastat zásadní změna v implementaci a rozhraní tzv. breaking-changes. Framework tedy poskytuje nejvyšší možnou úroveň podpory pro programovací jazyk Kotlin. Veškerou konfiguraci frameworku je možné dělat programově.

## Hodnocení (0-3): 3

### Moduly

### Web

Webový modul obsahuje širokou paletu základní funkčnosti. Obsluha HTTP requestů je prováděna přes router, který je reprezentovaný jako strom, který je složen ze zanořených selektorů a uzlů. Selektory můžeme třídit requesty podle cesty, kdy ji můžeme určovat také pomocí regulárních výrazů a dalších prostředků. Také je možné využít třídění podle typu obsahu, který request obsahuje. I do stromu je možné vkládat vlastní pipelines, které mohou

sloužit jako HTTP filtr. Routování je velmi flexibilní, jelikož je možné všechny funkce mezi sebou kombinovat a je tedy možné pokrýt i složité struktury. Při zpracování requestu je z koncového uzlu možné přistupovat ke parametru call, přes který je možno získávat všechna data spojená s obsluhou requestu, včetně jeho samého. Pro odesílání HTTP responses je možné využít generické metody z objektu call, které automaticky složí odpověď. Pro obsluhu multipart requestů framework poskytuje programové rozhraní a vývojář pracuje přímo s instancemi ParData, jež reprezentují objekty z těla requestu. Ktor poskytuje automatizovaný systém logování requestů, který je dále možné customizovat. Pomocí externí knihovny je podporováno rozhraní GraphQL, nejlepší implementaci poskytuje v současné době knihovna graphql-kotlin.

Pro statické soubory Ktor nabízí speciální routovací funkci static, pomocí které je možné flexibilně publikovat statické soubory na mnoha různých url cestách s odlišným nastavením. Výhodou je že se dá metoda poměrně dobře škálovat zanořováním, které dokáže vytvořit komplexní strukturu.

Error handling a status pages jsou spravovatelné pomocí rozšíření, které se programově nastavuje a lze pomocí něj globálně obsluhovat jakékoliv třídy výjimek, HTTP response status kódy a status pages pro jednotlivé error kódy.

Ktor umožňuje správu cookies a sessions. Cookies lze upravovat okamžitě bez konfigurace, programové rozhraní je přímo dostupné přes request. V případě sessions je nutné nastavit programově, k dispozici jsou předdefinovaná úložiště v soubory a v paměti, je však možné po implementaci jednoduchého rozhraní napojit vlastní úložiště.

Ktor podporuje HTTP/2 a HTTPS, díky nativní podpoře SSL. Také poskytuje podporu pro práci s web sockety, avšak neposkytuje další nástroje, vývojář tak musí pracovat na nízké úrovni, stejně jako s nativními sockety v jazyku Java.

Při spojení s Kotlin/Js, lze tvořit fullstack webové aplikace, kompletně napsané v Kotlinu, přičemž je možné sdílet kód mezi frontend a backend částí aplikace. V každém případě jde o více flexibilnější a robustnější řešení než při použití pouze HTML šablon doplněných o JavaScript/AJAX. Podporu pro jazykové mutace framework zatím nepodporuje.

**Hodnocení (0-3): 3**

## Security

Framework poskytuje podporu pro autentifikaci pomocí doplňkových knihoven. Ta se programově zintegruje v kódu a také poskytuje nové metody do routeru, kdy je možné obalit skupinu funkcí a vynutit tak na nich autentizaci. Knihovny poskytuje integraci pro základní autentifikační standardy jako je Basic, Digest, JWT/JWK, LDAP, OAuth. Integrace vlastního systému přes vlastní implementace rozhraní framework nenabízí, avšak je možné přizpůsobit knihovnou poskytnuté řešení.

### Hodnocení (0-3): 2

## Templating

Ktor poskytuje podporu pro známé šablonovací enginy. Podporuje Freemarker, Mustache, Thymeleaf, Velocity. Všechny jsou poskytovány skrze knihovny. A instalují se programově. Ktor podporuje vlastní HTML DSL knihovnu, která umožňuje zapisovat HTML přímo v Kotlinu pomocí funkcí, které reprezentují jednotlivé tagy. Jde o podobné řešení jako nabízela knihovna kotlinx.html, avšak zde je možnost vytvářet třídy které jsou poděděny z generické třídy Template a je tak možné dělat kompletní šablony přímo pomocí kompozice tříd.

### Hodnocení (0-3): 3

## Caching

Framework neposkytuje proprietární implementaci pro cachování. Lze jí vytvořit pomocí features a manuálně přidat k vybraným selektorům v routing tree. Jedinou podporu, kterou Ktor nabízí je caching dle hlaviček, kdy můžeme dle vlastního vyhodnocení odchozího obsahu uložit do cache. Tato funkce se nejvíce hodí pro odesílání objemnějších souborů. V současné době nejsou poskytnuty nástroje pro integraci externích cachingových řešení.

### Hodnocení (0-3): 1

## *Dependency injection*

Ktor nemá vlastní implementaci pro dependency injection, avšak poskytuje podporu pro knihovny, které ji pro Kotlin poskytují. Mezi podporované patří Guice od Google, jež umožňuje použít anotace, avšak za cenu reflexe, závislosti se organizují do modulů. Další možností je Koin což je velmi light-weight řešení, které nevyužívá proxy, reflexe a negeneruje kód, nevýhodou je, že vývojář musí manuálně tvořit moduly a provazovat závislosti, což u větších a komplexnějších projektů není moc pohodlné. Lepší alternativou, kterou Ktor podporuje je Kodein je robustnější, má více funkcionality, je lépe dokumentovaný a dokáže sám provázat závislosti mezi sebou, takže vývojář pouze deklaruje třídy v modulu. Kodein je nejlepší volbou pro DI, navíc při jeho použití lze lehce mockovat vybrané závislosti.

**Hodnocení (0-3): 3**

## *JSON*

V rámci frameworku je možné využít automatickou konverzi, kterou je nutné programově nastavit. Pro formát JSON je podporována integrace dvou knihoven, které umožňují jeho automatickou serializaci při odesílání a deserializaci pro příjem. Podporována je knihovna Gson, což je nativní knihovna přímo pro Kotlin a také Jackson. Obě jsou distribuovány formou externí knihovny. Jejich konfigurace probíhá při její instalaci jež probíhá programově v kódu, včetně konfigurace samotné.

**Hodnocení (0-3): 3**

## *Integrace*

Ktor nenabízí v současnosti moduly pro integraci externích systémů a služeb. Framework nabízí pouze modul pro asynchronního HTTP klienta. Avšak je možné chybějící funkcionalitu pomocí, externích knihoven, které jsou pro Kotlin.

**Hodnocení (0-3): 1**

## *Přístup k datům*

Stejně jako u integrace, Ktor nenabízí proprietární moduly pro přístup k datům. Je možné použít již existující řešení, která jsou pro Kotlin. Samotný JetBrains vyvíjí knihovnu Exposed, jež usnadňuje přístup k SQL databázím.

### **Hodnocení (0-3): 1**

## *AOP*

Ktor nepodporuje AOP, avšak je možné částečně nahradit pomocí implementace features. Podpora se v blízké době ani nedá očekávat, jelikož hlavní architekt jazyka Kotlin Roman Elizarov se ve svém článku<sup>26</sup> staví celkově k AOP skepticky a nabízí alternativní náhradu v podobě funkcionálně orientované implementace jež nahrazuje klasické AOP. Implementace dle Elizarova je realizována pomocí globálních inline funkcí, které implementují rozhraní obsahující metodu přetěžující operátor plus a díky tomu lze řetěžit funkce.

### **Hodnocení (0-3): 1**

## *Scheduling*

Ktor momentálně nepodporuje periodické akce. V nedávné době byla do frameworku přidána feature branch s implementací od kontributora, která umožňuje poměrně flexibilní způsob jak nastavit periodické akce přes programové rozhraní. V současnosti lze využít nativní řešení, které poskytuje Kotlin ve standardních knihovnách, konkrétně sadou extension metod schedule na třídu Timer, která se nachází ve standardních knihovnách jazyka Java. Pomocí těchto metod lze nastavit jednorázové akce dle data, podle časového odstavu, anebo periodické akce. Samotné metody však nedokáží konkurovat knihovnám, které mají plánování založené ve stylu Cron.

---

<sup>26</sup> <https://medium.com/@elizarov/aop-vs-functions-2dc66ae3c260>



## Hodnocení (0-3): 1

### Testovatelnost

Ktor nabízí vlastní nástroje pro testování. Testování je prováděno ve stylu e2e testů, kdy není možné provádět izolované unit testy. Pro e2e testy nabízí poměrně kvalitní programové rozhraní, díky kterému lze psát přehledné testy. Ktor poskytuje přímo speciální prostředí pro testy, díky nimž je možné testovat přímo router, bez síťové komunikace, kdy by se musel spustit server s aplikací a proti němu spouštět testy, což ušetří poměrně hodně času a testy jsou rychlé. Ktor nemá popsány další možnosti testování a podporu integrace jiných knihoven jako je např. JUnit, ale pro většinu testů postačí proprietární řešení.

## Hodnocení (0-3): 2

### Podpora a komunita

Framework je zatím stále v prvopočátcích, byť už je vydána jeho několikátá stabilní verze. Je vyvíjen teprve od roku 2016 a za tu dobu autoři vydali 96 releasů. Do projektu přispělo již 94 kontributorů a v současnosti je v projektu 36 pull requestů. V blízké době je očekávána nová verze 1.3, jelikož v současnosti je již k dispozici její první beta verze. V issue trackeru je v současnosti 387 otevřených ticketů, za celou dobu vývoje bylo již 651 ticketů vyřešeno. Autoři všechny tickety třídí, tudíž je možné okamžitě vidět, jak se bude problém ubírat. Avšak je zde poměrně velké množství ticketů se štítkem bug, což značí že framework stále trpí velkým počtem chyb. Některé bug tickety jsou i poměrně starého data, tudíž byt je reakce vývojářů pohotová, samotná oprava nemusí nastat v rozumném čase. Pro přímou komunikaci je založen topic v chatovacím nástroji Slack ve skupině, která je společná pro všechny projekty v rámci jazyka Kotlin, avšak pro vstup do skupiny je nutné pozvání, které zájemce obdrží po vyplnění dotazníku v řádu dnů, což není úplně praktické. Jiný komunikační kanál nikde není publikován.

## Hodnocení (0-3): 1

## Praxe

V současné době pro framework neexistuje mnoho relevantních zdrojů. Nejvíce obsáhle informace poskytuje oficiální webová stránka. Avšak informace jsou poměrně stručné, a ne příliš dobře strukturované, díky tomu je orientace poměrně ztížená. Web obsahuje mnoho ukázek, které jsou stručně popsány, avšak celkově pro učení frameworku není web nejlépe koncipován. Web obsahuje online generátor projektu, který je poměrně přehledný a lehce použitelný. Celkově framework dostává dobrou podporu ze strany nástrojů firmy JetBrains, v IntelliJ měl již delší dobu podporu pomocí pluginu, v současnosti je plně podporován a zintegrován v IDE. Tudiž lze řešit generování projektu přímo v IntelliJ, kromě toho jsou poskytovány další funkce, jež usnadňují vývoj a orientaci v kódu projektu. Ktor má také kvalitní podporu pro automatický reloading, který urychluje práci vývojáře při vývoji, kdy automatizovaně nasazuje aplikaci, pokud nastanou změny v projektu.

**Hodnocení (0-3): 2**

## Spark/Javalin

### Historie

První verze frameworku se objevila již roku 2013, a v létě téhož roku vyšla první stabilní verze. Framework je vyvíjen Perem Wendelem, jenž působí jako softwarový architekt ve firmě Sony. Spark je od roku 2013 pravidelně aktualizován, kdy nové verze vycházejí zhruba po každém půl roce. Tým se za dobu vývoje rozrostl na tři členy a v současnosti je jeden vývojář vyčleněn na rozvoj frameworku pro Kotlin. V současné době je framework ve verzi 2.9.1 z května tohoto roku, momentálně probíhá sběr požadavků na novou major verzi 3. Framework je navržen kvalitně, avšak některé detaily nejsou zapracovány, tak jak by si většina vývojářů představovala, proto vznikla odštěpená větev Javalin, která řeší tyto problémy a celkově rozšiřuje funkcionalitu. Javalin plně podporuje jazyk Kotlin a na rozdíl od Sparku je v něm napsán. První verze Javalinu se objevila v květnu roku 2017 a již v listopadu téhož roku byla vydána první stabilní verze. Javalin je vyvíjen Davidem Asem.

## Technologie

Spark je koncipován jak lightweight webový framework pro jazyky Java a Kotlin. Hlavní myšlenkou frameworku je poskytnout DSL pro rychlé prototypování aplikací, bez zbytečného kódu navíc, to vše díky jeho důrazu na expresivitu. Spark je navržen s inspirací v JavaScript frameworkcích, zejména v Node.js. Spark tvoří tenkou obálku nad Servletem, který využívá pro komunikaci a díky tomu musí být provozován na aplikačních serverech. Primárním embedded serverem je Jetty, případně je možné aplikaci distribuovat pomocí war a nasadit na libovolný aplikační server. Hlavní důraz je kladen na poskytnutí kvalitního routování než na široké spektrum funkcionalit. Framework nevynucuje žádnou dědičnost a pouze výjimečně vyžaduje implementaci rozhraní. Také vynechává anotace a nevyužívá běhovou reflexi. Spark má kombinované API, kdy v některých případech je datům přistupováno přes statické položky a v jiných případech zase přes instance, což působí značně nekonzistentně. Oproti tomu Javalin poskytuje přístup pro vše skrze instance. Javalin může běžet i mimo aplikační server.

## Podpora Kotlinu

Pro jazyk Kotlin, poskytuje Spark samostatnou knihovnu, která kompletně pouze tvoří wrapper nad originální knihovnou a umožňuje psát idiomatický kód v Kotlinu. Knihovna je v první stabilní verzi, avšak stále jen v preview verzi z roku 2017. Pro lepší podporu Kotlinu je vhodnější využít framework Javalin. Další hodnocení se budou vztahovat pouze k Javalinu, jelikož všechnu funkcionalitu, kterou poskytuje Spark, poskytuje i Javalin. Navíc Javalin je kompletně napsaný v Kotlinu.

## Hodnocení (0-3): 3

## Moduly

## Web

Javalin stejně jako předchozí frameworky provádí obsluhu requestů pomocí stromu, který je složen ze sekvence metod. Webový modul je nejrozsáhlejší část frameworku, avšak oproti předchozím je pojat zcela minimalisticky. Mapovat endpointy lze plnými názvy případně doplněným o wildcard symbol, díky kterému může být zbytek cesty libovolný. Celý strom lze obohatit o filtry, které se spouští před, či po volání všech endpointů, či vybraného u kterého platí stejná pravidla jako při mapování endpointů. Handlers pro obsluhu jsou pojaty také minimalisticky a do metody předávají jako parametr pouze jediný objekt Context, přes který se obsluhuje. Všechny metody tohoto objektu jsou přístupné v první úrovni, což lehce naznačuje antipattern God object<sup>27</sup>. Obsluha HTTP requestu probíhá tedy výlučně přes tento objekt, handlersy nevadí žádnou hodnotu. Avšak výhodou tohoto přístupu je možnost rozšiřovat funkcionalitu, kdy registrujeme třídu do objektu Context při spuštění, poté se vytvoří její instance před prvním použitím a k té je pak možné přistupovat přes objekt Context a využívat její metody. Pomocí metody path se dají cesty zanořovat a díky tomu lze vytvořit složité struktury u kterých je přesto zachována dobrá přehlednost. Pro zvýšení čitelnosti poskytuje Javalin metodu crud, která mapuje všechny obvyklé datové operace na danou cestu. Javalin poskytuje vlastní systém správy chybových stránek a stavů. Pro vyvolání stačí pouze vyhodit odpovídající výjimku, do které je možné přidat další popisné atributy. Všechny výjimky a chybové stavy lze odchytávat a řídit pomocí registrace odpovídajících handlerů. Javalin nativně podporuje HTTPS, HTTP/2 a poskytuje kvalitní rozhraní pro komunikaci přes WebSocket. Práci se sessions, cookies a nahráváním souborů usnadňuje framework dostatečně pomocí programového rozhraní. Pro sessions poskytuje předdefinovaný systém pro integraci úložiště či cache. Javalin podporuje dokumentaci rozhraní podle standardu OpenApi a nabízí možnost programového přístupu anebo pomocí anotování. Javalin podporuje více jazyčné aplikace, které fungují na základě automatického výběru z resource souboru, ve kterém jsou různé typy jazykových mutací.

### Hodnocení (0-3): 3

#### *Security*

---

<sup>27</sup> [https://en.wikipedia.org/wiki/God\\_object](https://en.wikipedia.org/wiki/God_object)

Javalin zvolil poměrně volný přístup pro implementaci zabezpečení. Kdy poskytuje rozhraní pro implementaci vlastního, ve kterém má vývojář naprostou svobodu ve způsobu implementace. Celý systém zabezpečení podporuje autentizaci i autorizaci, funguje na principu globálního vstupního filtru, přes který prochází každý HTTP request, avšak oproti klasickému filtru se volá až po vyhodnocení routování, tudíž jsou do metody předávány povolené role pro daný endpoint. Pro usnadnění framework poskytuje integraci pro Basic Auth, které automaticky mapuje Auth header HTTP requestu. Dále je podporován standard JWT v kombinaci s externí knihovnou.

### **Hodnocení (0-3): 3**

#### *Templating*

Javalin umožňuje použít celou řadu šablonovacích enginů. Framework vybírá šablonovací engin podle přípony šablony a teoreticky lze využívat více šablonovacích enginů v rámci jedné aplikace. Mezi podporované systémy patří Thymeleaf, Velocity, Freemarker, Mustache, Jtwig, Pebble, Commonmark. Kromě toho framework poskytuje nativní podporu Vue.js a je možné tvořit fullstack projekt. S touto knihovnou nabízí velmi úzkou integraci a lze velmi snadno kromě skládání komponent také sdílet data přímo z kódu aplikace.

### **Hodnocení (0-3): 3**

#### *Caching*

Framework nenabízí implementaci pro cache a ani neposkytuje na integraci externích služeb pro cache. Cache poskytuje pouze pro statické obsah.

### **Hodnocení (0-3): 1**

#### *Dependency injection*

Javalin umožňuje modularizaci pomocí externí knihovny Guice jež je od společnosti Google. Avšak používání této knihovny stojí psaní mnoha kódu navíc, zesložituje celou strukturu projektu. Vlastní implementaci dependency injection framework neposkytuje.

**Hodnocení (0-3): 1**

### *JSON*

Framework poskytuje automatické mapování z formátu JSON a zpět do něj. Defaultně podporuje knihovny Gson a Jackson. Knihovna musí zaregistrovat metody, které se používají pro serializaci a deserializaci. Díky tomu je možné zintegrovat i jinou knihovnu než jsou výše dvě zmíněné.

**Hodnocení (0-3): 3**

### *Integrace*

Javalin nenabízí žádnou přímou podporu nástrojů se kterými by umožňoval zjednodušenou integraci integraci. Framework nenabízí ani HTTP klienta. Javalin podporuje využívat Java EE Servlety a Filtry, které běží na stejné instanci serveru Jetty.

**Hodnocení (0-3): 0**

### *Přístup k datům*

Framework neposkytuje vlastní implementaci ani zjednodušenou podporu externí knihovny pro přístup k datům. Je nutné využít externí knihovny a provést vlastní intergaci do projektu.

**Hodnocení (0-3): 0**

## AOP

Přímou podporu framework nenabízí, avšak je možné využít filtry, které se volají před vybranými, nebo všemi routovacími cestami. Což je částečná náhrada pro aspektový způsob programování.

**Hodnocení (0-3): 1**

## Scheduling

Ani pro plánování úloh nenabízí framework vlastní implementaci, či podporu externích knihoven. Je tedy nutné využít externí knihovnu, či defaultní implementaci v Kotlinu.

**Hodnocení (0-3): 0**

## Testovatelnost

Javalin podporuje testování, které je koncipováno jako e2e testy. Tím že je do handleru vložen mock a po provolání handleru se zvaliduje výsledky volání. Pro mockování musí být použita externí knihovna, Javalin nenabízí vlastní implementaci. Pro testování je nutné upravit způsob implementace a vyjmout reference handleru. Implementace v projektu pak vypadá více modulárně. Nevýhodou je že pro testování je nutné upravit kód již existující kód.

**Hodnocení (0-3): 2**

## Podpora a komunita

Javalin je zralý framework. V současné době je ve verzi 3.6 autoři přispívají novými updaty velmi pravidelně, v průměru dvakrát měsíčně po celou dobu existence frameworku. Autoři pečlivě spravují issue board na Githubu, kde mají otevřeno pouze 19 ticketů, jež pouze jeden byl minulého roku a u každé issue je tag a komentář autorů. Za celou dobu existence vyřežili již 480 ticketů. Na Github je dohromady 76 kontributorů do kódu, kteří vytvořili již 295 pull

requestů, v současnosti jich je otevřeno 5 a ke každému autoři poskytují feedback. Nevýhodou frameworku je že neexistuje žádný komunikační kanál, kde by probíhala mezi uživateli a autory neformální debata. Tag Javalin se vyskytuje i na Stackoverflow, kde je v současné době 16 otázek s tímto tagem a celkově ho odebírá pouze 8 uživatelů.

## **Hodnocení (0-3): 2**

### Praxe

K frameworku Javalin existuje dokumentace na oficiálním webu, dále je zde k dispozici několik konkrétních návodů pokrývajících různou problematiku. Web není informačně vyčerpávající, pro základní osvojení frameworku však postačuje. Celkově jde o framework dohledat pouze minimum informací. Žádná literatura ani kurzy nejsou v současné době k dispozici. Taktéž není nativně podporován ze strany vývojových nástrojů.

## **Hodnocení (0-3): 1**

### Micronaut

#### Histore

Micronaut je vývojově poměrně mladý framework, práce na něm započaly teprve v roce 2017. Framework spravuje společnost Object Computing jako open-source projekt. Tato společnost má již na svém kontě velmi úspěšný webový framework pro Groovy, mimochodem část vývojářů jsou také bývalí zaměstnanci společnosti Pivotal, která se zabývá vývojem legendárního webového frameworku Spring. Primární inspirace tedy pochází z frameworků Spring a Grails z nichž Micronaut obsahuje ty nejlepší a osvědčené návrhy, avšak s vlastní inovativní implementací. Naopak se vyhýbá slabým stránkám těchto frameworků. Micronaut vznikl primárně pro snadnou tvorbu microservices, celkově je na toto zaměření kladen velký důraz.



## Technologie

Micronaut je navržen pro klasickou MVC a třívrstvou architekturu. Kód aplikace je velmi podobný kódu psanému ve Springu. Celý framework je postaven komponentově a nechává vývojáři volnost pro náhradu proprietárních řešení. Avšak stejně jako Spring Boot je framework defaultně auto-konfigurován a zbytečně nezatěžuje vývojáře, pokud vyloženě nepožaduje specifickou konfiguraci. Hlavním benefitem frameworku je koncepce, která nevyužívá téměř žádné run-time proxy a kompletně vyřadil využívání reflexe za běhu. Tím umožňuje velmi rychlé starty aplikace, menší alokaci paměti. Většina operací jako je dependency injection, AOP proxy, konfigurace, která u monolitických frameworků probíhá při startu, provádí Micronaut při kompilaci. Všechny anotace při kompilaci překládá do ASM<sup>28</sup> byte kódu, který propojuje komponenty frameworku. Reflexe je nahrazena díky využití nativního Kotlin anotačního procesoru Kapt<sup>29</sup>. Výhodou tohoto přístupu je nezávislost množství potřebné paměti a rychlosti startu aplikace vůči velikosti jejího kódu. Celý tento koncept je pojmenován jako Micronaut AoT (Ahead-of-Time) kompliance. Aplikace napsaná v Micronautu se vejde do 10 MB JAR souboru, zabere 10 MB operační paměti a nashoduje během jedné sekundy. Framework se snaží také minimalizovat externí závislosti na jiných knihovnách. Micronaut podporuje reaktivní zpracování, implementuje Reactive Streams standard, včetně RxJava, Reactor a ve verzi 1.3 přibude podpora pro Kotlin korutiny. Celý framework je postaven na I/O neblokuující knihovně Netty. Micronaut podporuje kompilaci do GraalVM image a díky tomu může v tomto alternativním virtuálním stroji, která ještě navýší jeho výkonost.

## Podpora Kotlinu

Celý framework je napsan v jazyku Java. Neposkytuje wrapper, avšak i tak se dá psát kód v Kotlinu dostatečně idiomaticky. V současné době je vyvíjen modul Micronaut-Kotlin, který doplňuje framework o podporu knihovny config4k, která umožňuje psát konfigurační soubor přímo v Kotlinu a rozšiřující knihovnu pro Jackson, která umožňuje automatickou konverzi z Kotlin tříd do formátu JSON a naopak. Další zajímavou funkcionalita je podpora frameworku

---

<sup>28</sup> <https://asm.ow2.io/>

<sup>29</sup> <https://kotlinlang.org/docs/reference/kapt.html>

Ktor, kdy je možné využít Ktor místo nativního HTTP serveru a díky tomu využívat funkce (DI, AOP, ORM atd.), které nabízí Micronaut přímo v aplikaci Ktor, jenž nemá takový rozsah poskytované funkcionality.

## **Hodnocení (0-3): 2**

### Moduly

#### Web

Framework poskytuje nástroje pro kompletní obsluhu HTTP requestů. Handlers vypadají totožně jako ve Springu. Většina konfigurace je skrze anotace, avšak některé parametry předává do anotovaných controllerů framework automaticky po jejich deklaraci jako parametr metody. Framework podporuje reaktivní controllery, díky kterým není blokován tok a samotná obsluha HTTP requestů. Programově je podporováno nahrávání a zpracování multipart requestů, kdy vývojář pouze definuje názvy položek a framework zajistí automatické nahrání. Statický obsah a přístup k němu je možné spravovat přes konfigurační soubory. Micronaut poskytuje možnost globálního error handlingu, kdy je možné odchyťovat specifické typy HTTP response kódu, případně vyhozené výjimky. Užitečným nástrojem je podpora verzování API, kdy je možné přes konfigurační soubor spravovat skupinu endpointů, které jsou anotovány stejnou verzí. Tímto lze snadno přepínat verze API bez nutnosti rekompilovat a nasazovat celou aplikaci. Framework umožňuje využití HTTP filtrů. Pro práci se sessions a cookies je poskytováno kvalitní programové rozhraní, které značně usnadňuje rutinní operace. Konfigurace aplikace se provádí skrze konfigurační soubory, avšak vývojář má poměrně širokou možnost volby, podporovány jsou soubory ve formátech yml, properties, json, groovy. Micronaut nativně podporuje HTTPS a podpora HTTP/2 se připravuje do verze 2.0. Framework poskytuje velmi kvalitní aplikační rozhraní pro WebSockets, které umožňuje i neblokující obsluhu zpráv. Framework podporuje velmi dobře SOA architekturu na programové úrovni. Micronaut poskytuje vlastní modul pro implementaci GraphQL v projektu. Automatická dokumentace ve standardu OpenApi rozhraní je podporována nativním modulem.

## **Hodnocení (0-3): 3**

## Security

Micronaut pro zabezpečení poskytuje samostatný modul, který podporuje velkou škálu standardních systému pro zabezpečení. Pro autentizaci a autorizaci podporuje BasicAuth, Session Authentication, JWT, LDAP, OAuth 2.0, OpenID. Micronaut nabízí možnost úpravy, či vytvoření svého systému pro ověřování skrze implementaci rozhraní. Tvorba vlastní implementace je velmi triviální a pro základní autentizace je nutné implementovat pouze rozhraní obsahující jednu metodu. Framework umožňuje pomocí konfigurace nastavit předávání tokenů, tato funkce značně usnadňuje předávání tokenů v rámci sítě microservices, dále je možné nastavit multi-tenantnost aplikace.

**Hodnocení (0-3): 3**

## Templating

Framework poskytuje modul views, který umožňuje integraci šablonovacích enginů. Podporovány jsou Thymeleaf, Handlebars, Velocity, Freemarker. Řešení pro Kotlin nejsou v současnosti nativně podporována.

**Hodnocení (0-3): 2**

## Caching

Ve frameworku je nativní implementace cache. Implementace je založena na AOP, pomocí několika anotací je možné nastavovat operace k metodám. Anotace jsou pro označení cachované metody, dále metody, která vkládá do cache (cache-supplier) a nakonec metody po jejímž zavolání se provede vyčištění cache (cache-evict). Pro větší kontrolu vývojářem je možné implementovat CacheManager, který umožňuje převzít správu na cachování. Vlastnosti jednotlivé cache lze také nastavovat skrze konfigurační soubor. Nativně je také podporováno úložiště v Redis, který lze aktivovat skrze konfigurační soubor, při nastavení jeho adresy.

### **Hodnocení (0-3): 3**

#### *Dependency injection*

Micronaut poskytuje vlastní implementaci pro DI. Jedná se o velmi robustní řešení, které lze srovnávat se Spring DI, se kterým je mimo jiné kompatibilní a dokáže využívat Spring Beans. Oproti Springu však probíhá již při kompilaci a díky tomu je více výkonný. Celý systém DI je velmi nezávislý díky modularitě frameworku a dá se samostatně využít i v jiných projektech např. Android. Podporovány jsou také tzv. configuration injections, kdy je možné vkládat hodnoty z konfigurace do atributů, či dokonce automaticky do celé struktury tříd a atributů.

### **Hodnocení (0-3): 3**

#### *JSON*

Micronaut poskytuje nativně serializaci a deserializaci obsahu těch HTTP zpráv, které obsahují JSON, nebo objekt automaticky. Konverze probíhá pomocí knihovny Jackson, tu lze nastavovat skrze konfigurační soubor. Vývojář může samozřejmě využít vlastní řešení pomocí implementace abstraktních tříd. Je možné kompletně nahradit knihovnu Jackson, či případně pouze implementovat třídy, které modifikují pouze vybrané aspekty a mít v symbióze vlastní a nativní řešení, bez nutnosti kompletní implementace.

### **Hodnocení (0-3): 3**

#### *Integrace*

Framework podporuje celou řadu nativních modulů pro integraci s externími systémy. Poskytuje implementaci klientů pro cloudové providery jako je Netflix (Ribbon, Eureka), Amazon (AWS). Podporuje Kubernetes, Zipkin, Jaeger pro tvorbu sítě microservices. Poskytuje klienta GRPC pro vzdálené volání metod a vlastní implementaci HTTP klienta, ve dvou variantách abstrakce. Micronaut má nativní klienty pro messaging, konkrétně pro Kafku

a RabbitMQ. Pro měření a analitiku poskytuje framework integraci s externím nástrojem Micrometer.

### **Hodnocení (0-3): 3**

#### *Přístup k datům*

Framework má modul pro přístup k datům, dříve známý jako Predator (Precomputed data repositories), dnes pod názvem Data. Stejně jako ostatní moduly i tento je v souladu s AoT a tudíž nevyžaduje další režii za běhu aplikace. Modul poskytuje širokou funkcionalitu, na výběr je možnost přístupu k datům na nižší úrovni pomocí Data JDBC, anebo využít zdrojově náročnější přístup pomocí Data JPA, který však zpohodlní práci vývojáře. Práce s daty při použití knihovny Data je totožná jako u frameworku Spring a umožňuje plně využít ORM včetně validací. Micronaut poskytuje reaktivní, neblokující konektory k SQL databázím (PostgreSQL) a také k NoSQL (Neo4j, Redis, MongoDB, Cassandra). Podporuje pomocí modulu také využití Flyway pro validaci a verzování schémat DB.

### **Hodnocení (0-3): 3**

#### *AOP*

Framework poskytuje podporu pro aspektové programování. Micronaut nabízí proprietární implementaci, jež je v souladu s přístupem AoT. Oproti jiným implementacím, které využívají běhové proxy a reflexi, jenž zpomalují celou aplikaci, způsobují obtížnější debugování a jsou paměťově náročné. Micronaut AOP řeší již při kompilaci. Micronaut podporuje around advice (obaluje metodu), introduction advice (poskytují funkčnost) a adapter advice (vkládá implementaci). I přesto, že Micronaut AOP implementace není vše vyčerpávající, poskytuje velmi robustní řešení a je možné pokrýt téměř všechny případy užití. Micronaut podporuje také zpracování Spring AOP anotací, kompatibilitu mezi anotacemi je dosaženo pomocí speciální anotace, která zajišťuje aliasing pro parametry ve Spring anotacích.

### **Hodnocení (0-3): 3**

## Scheduling

Plánování periodických operací je Micronautem poskytováno vlastní implementací. Implementace je založena na AOP. Micronaut poskytuje anotace, kterými je možné anotovat metody, jejichž spuštění má být naplánováno, řešení je tedy stejné jako nabízí Spring. Nastavovat spuštění je možné pomocí CRON zápisu, případně lze využít defaultní schémata jako je spuštění v pravidelném intervalu, spuštění se zpožděním, spuštění po startu se zpožděním. Pokud není možné upokojivě nastavit spuštění pouze pomocí anotace anebo interval je definován až za běhu je možné využít pro TaskScheduler, který poskytuje programové rozhraní pro plánování spuštění úloh přímo v kódu aplikace. Konfigurací je možné řídit i thread pool, který je vyčleněn pro vykonávání naplánovaných úloh. Celý systém plánování je velmi kvalitně zpracovaný a lze skrze něj pokrýt všechny případy, které mohou nastat.

**Hodnocení (0-3): 3**

## Testovatelnost

Micronaut díky své MVC architektuře umožňuje testovat všechny vrstvy aplikace a to i separátně. Pro testování Micronaut podporuje testovací frameworky Spock (Groovy), JUnit5 (Java), KotlinTest (Kotlin). Tyto frameworky lze volit nezávisle na jazyku ve kterém je napsán kód aplikace. Framework poskytuje kvalitní prostředí, které umožní odstínit vnější závislosti a poskytuje programové rozhraní pro jejich mockování v rámci testovací třídy. Poskytuje také programové rozhraní pro testovací server a http klienta díky tomu je možné jednoduše provádět e2e testy.

**Hodnocení (0-3): 3**

## Podpora a komunita

Micronaut je stářím poměrně nový, avšak co se týká vývoje tak se jedná o zralý framework. V současné době je vydána verze 1.2.6 a v dohledné době bude vydána verze 1.3. Autoři vydávají verze core modulu poměrně pravidelně, souběžně s vývojem na core také vylepšují existující moduly a vydávají i nové. Do hlavního modulu přispívá 170 kontributorů, v současné době je otevřeno 11 pull requestů a celkem za dobu vývoje jich bylo otevřeno přes 850. Společnost Object Computing nabízí pro uživatele mnoho cest k podpoře při využívání frameworku. Nabízí placenou komerční podporu. Pro komunikaci od uživatelů využívají vývojáři dashboard na GitHubu, kde je možné reportovat issues, pro real-time řešení problému poslouží chat na Gitteru, kde je v současnosti k odběru přihlášeno téměř tisíc uživatelů, problémy tu zde komunikují přímo vývojáři frameworku s velmi rychlou odezvou. Poměrně silnou základnu má Micronaut na StackOverflow, kde má 420 otázek, z toho více než polovinu s odpověďmi. Celkově téma odebírá 150 uživatelů.

### **Hodnocení (0-3): 3**

#### Praxe

Webové stránky poskytují velké množství informací, které jsou přehledně uspořádány do referenčního manuálu, který pokrývá kompletní funkčnost dané verze. Další zdroje informací jsou guides, což jsou řešené malé tématické návody, které pokrývají mnoho oblastí. Na internetu zatím neexistuje mnoho materiálů a doposud nebyla vydána literatura ani obsáhlejší videokurzy. Avšak zcela dostačující jsou webové stránky frameworku. Micronaut má klienta pro generování projektu, na webu v současné době generátor nemá. V IDE v současnosti není žádná podpora pro framework, avšak je oficiálně potvrzeno, že má být Micronaut nativně podporován v IDE IntelliJ 2019.3.

### **Hodnocení (0-3): 3**

#### Měření výkonu web frameworků

Pro hodnocení výkonů frameworku jsem vyhledal profesionální benchmarking, který provádí společnost TechEmpower. Tato společnost provádí pravidelné testování webových frameworků pro různé jazyky a prostředí. Jedná se o komplexní testy z mnoha úhlů pohledu a pro větší relevantnost jsem vybral právě je, jelikož nastavit správné vlastní testování by značně přesahoval rozsah této práce a ve výsledku by nepřineslo relevantnější výsledky. V současné době (léto 2019) je prováděno již 18. kolo testování, první měření bylo provedeno v roce 2013 a v průměru vyjdou výsledky 2-3 kol za rok. Společnost stále rozšiřuje množství testovaných frameworků a typy testů, dle stránek mají v budoucnosti přibýt testy, při kterých se využívají features jednotlivých frameworků např. cachování, složitější algoritmické testy. Parametry testování jsou nastaveny velmi transparentně a údaje se dají snadno dohledat. Každý test probíhá izolovaně. Po restartu databázového serveru se spustí platforma a framework pomocí jeho nativního startovacího mechanismu. Prvních 5 vteřin probíhá test s 8 paralelními klienty, pro ověření, zda vše běží korektně, výsledky nejsou zaznamenávány. Poté se spustí 15-ti vteřinové rozehrátí s 256 paralelními klienty. Tato zátěž již vyvolá využívání lazy-initializaton, což je technika umožňující zrychlení aplikace, díky odložení tvorby objektů do poslední možné chvíle a just-in-time kompilaci, která zajišťuje zrychlení na úrovni interpretru ve virtuálním stroji, kdy tato technologie zajišťuje okamžitou interpretaci a predikci bajtkódu, který je nezbytně nutné přeložit do strojového kódu, tímto je docíleno rychlého startu a poměrně velké efektivity virtuálního stroje. I tento test ještě není zaznamenáván. Následuje několik 15-ti vteřinových bloků, které jsou již zaznamenávány. Velikost bloků narůstá ve dvojkovém exponenciálu. Pro testy se standardní souběžností začínají od 16 do 512 klientů, pro testy s vysokou souběžností např. test plaintext začínají od 256 do 16384 klientů. Časy testovacích bloků byly zvoleny dle dlouhodobých zkušeností a měření. Původně testy probíhali v 60 vteřinových blocích, ale díky narůstajícím permutacím a testům bylo nutné velikost bloků redukovat, avšak autoři do budoucna plánují prodloužení testovacích i zahřívacích bloků. Měření je prováděno pomocí frameworku Wrk<sup>30</sup>. Testy probíhají na fyzickém stroji i v cloudu. Fyzický stroj je sestaven z procesoru Intel Xeon Gold 5120, který je v současnosti (léto 2019) v čele top ten na prvním místě dle cpubenchmark.net<sup>31</sup>, dále je stroj osazen blíže nespecifikovanou operační pamětí o velikosti 32 GB a SSD disky. Cloud je hostován na Microsoft Azure D3v2 instancích. Ve FAQ jsou vysvětleny rozdílné výsledky testů mezi těmito

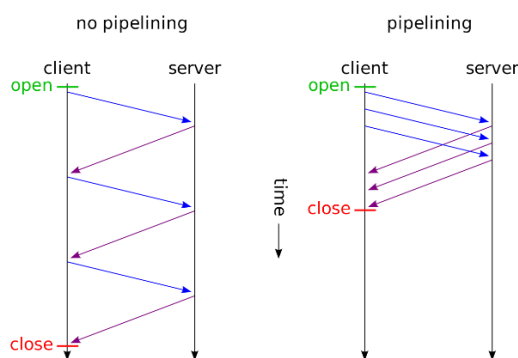
---

<sup>30</sup> <https://github.com/wg/wrk>

<sup>31</sup> <https://www.cpubenchmark.net/cpu.php?cpu=Intel+Xeon+Gold+5120+%40+2.20GHz&id=3154>

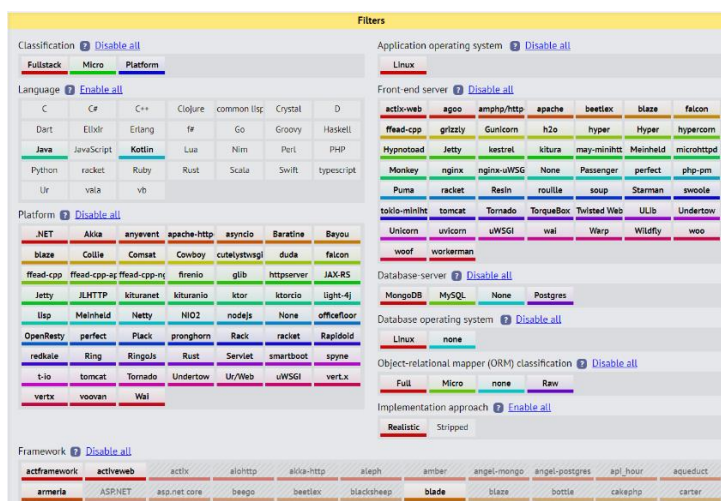


prostředími. U fyzického stroje je limitující šířka gigabit Ethernetu, který se zahltí při HTTP pipeliningu, což znamená že HTTP requesty jsou zasílány jedním TCP spojením bez čekání na potvrzení o doručení. Síť fyzického testovacího stroje se zahltí při 200 tisících requestů bez HTTP pipelingu a s ním se zahltí až při 550 tisících requestech. Toto omezení u cloudu není a jediným omezením je rychlost procesoru, tudíž pro interpretaci výsledků, budu využívat měření provedená v cloudu.



Obrázek 17 HTTP pipelining<sup>32</sup>

Na stránce <https://www.techempower.com/benchmarks> je možné vidět výsledky testů, mimo jiné je umožněno vizualizovat testy na vlastním hw, kdy naimportujeme result.json na stránky a ty jej pak vykreslí ve standardizovaných grafech a tabulkách. Testy byly filtrovány podle jazyků Kotlin a Java. Ostatní filtry zůstaly defaultní. V testech nebyl zahrnut framework Javalin, jelikož na stránkách nebyl mezi měřeními frameworky, místo něj je testován Spark.



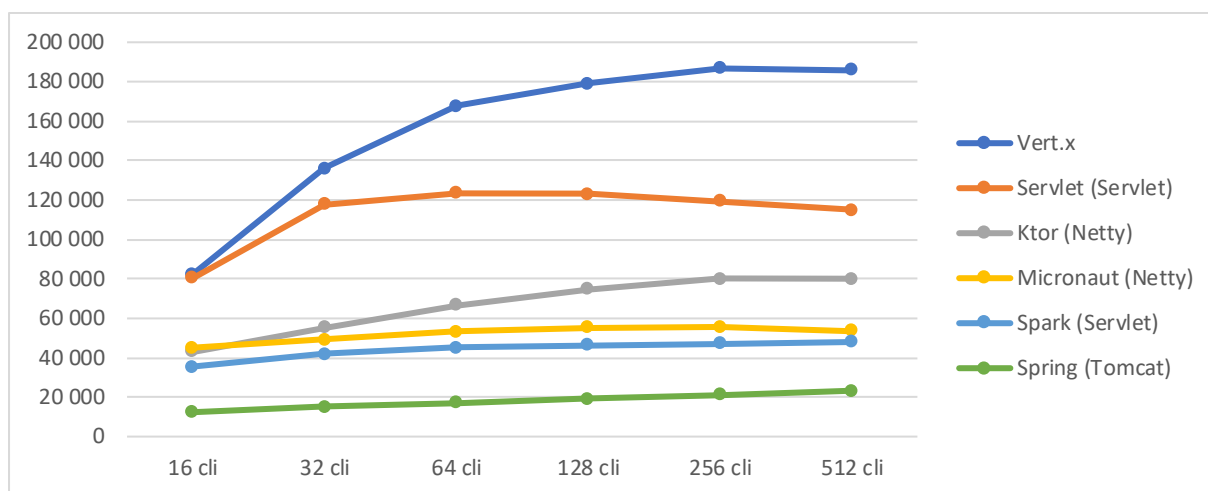
Obrázek 18 Ukázka filtrování

<sup>32</sup> [https://en.wikipedia.org/wiki/HTTP\\_pipelining#/media/File:HTTP\\_pipelining2.svg](https://en.wikipedia.org/wiki/HTTP_pipelining#/media/File:HTTP_pipelining2.svg)

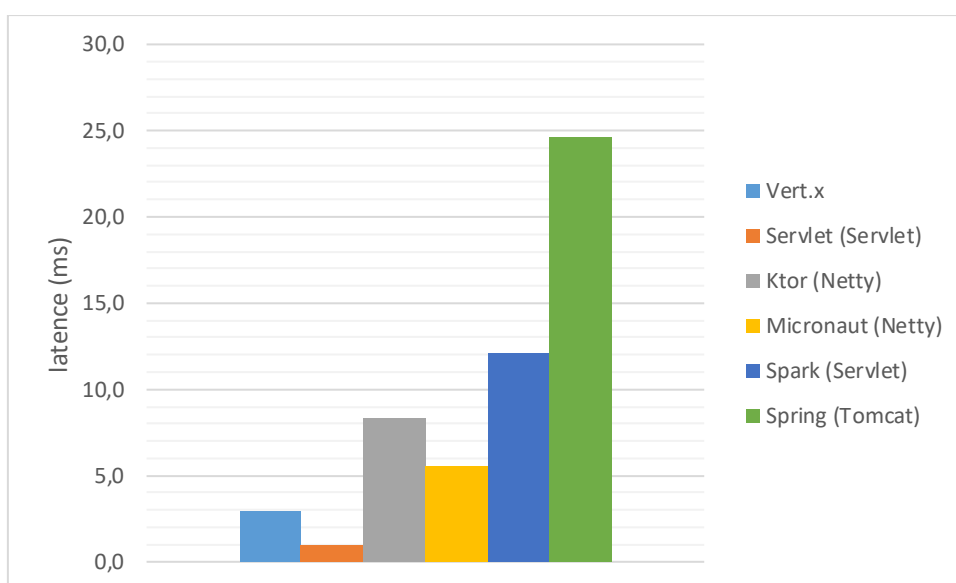
První test je JSON serializace, kdy je zaslán HTTP GET request na server a ten následně zašle response s tělem v podobě plain JSON ve tvaru {"message":"Hello, World!"}, každé tělo response je nově vytvořená instance. Z výsledků nejlépe dopadl Vert.x, který dokázal udržet stoupající tendenci paralelních klientů a počet opovědí se zvedal i při jejich nárůstu, dokázal dokonce předčít standardní Java EE implementaci, která využívá Servlet, který při maximálním vytížení klienty v počtu odpovědí nepatrně klesl. Avšak Servlet dokázal poskytovat nejrychlejší odpovědi průměrně za 1 ms a patří tak mezi nejrychlejší implementace vůbec. O řád níže se pohybovali zbylé frameworky, které dokázali odpovědět na značně menší počet requestů. Avšak samotný čistý framework Netty nad kterým jsou stavěné frameworky dokázal zpracovat kolem 195 tisíc requestů a stal by se tak pomyslným vítězem tohoto testu. Nejhuře dopadl Spring, který je v tabulce uveden jako zástupce nejpoužívanějšího frameworku pro tvorbu webových aplikací v jazyku Java. Pro srovnání Node.js dokázal odeslat 55 tisíc response, při průměrné latenci 1,1 ms.

Framework	Avg. Latence (ms)	16 cli	32 cli	64 cli	128 cli	256 cli	512 cli
Vert.x	2,9	82 401	136 072	167 530	179 040	186 725	186 105
Servlet	1,0	80 350	117 747	123 367	123 136	119 316	114 894
Ktor (Netty)	8,3	42 903	55 373	66 540	74 926	79 970	80 087
Micronaut (Netty)	5,5	45 103	49 306	53 383	55 336	55 614	53 787
Spark (Servlet)	12,1	35 518	41 938	45 117	46 517	46 976	47 970
Spring (Tomcat)	24,6	12 364	15 014	17 169	19 176	21 000	23 268

Tabulka 1 Test JSON serializace



Obrázek 19 Graf HTTP responses



Obrázek 20 Průměrná doba odpovědi

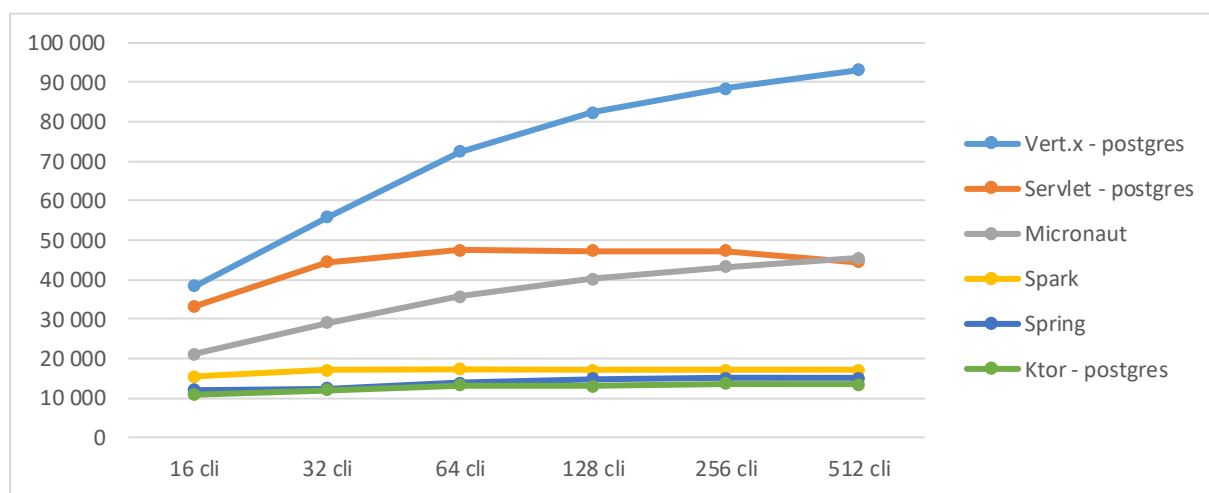
Druhý test spočívá v dotazování databáze obsahující 10 tisíc řádků, která je složena ze sloupce primárního klíče a druhého sloupce obsahující náhodné číslo. Oba dva sloupce jsou typu integer, ze kterých se vybírá pomocí generování náhodných klíčů, databáze není cachovaná. Odpověď ze serveru je plain JSON obsahující dvě položky. Vítězem testu se stal Vert.x s 93 tisíci odeslaných response, čistá konfigurace měla vyšší průměrnou latenci než při použití s proprietárním modulem pro web, který dosahoval pouze 3,1ms místo původních 6,8 ms, avšak v množství odeslaných response nepatrně zaostával. U Ktor je hodnota doby odezvy poměrně vysoká, avšak při použití knihovny reactivepg<sup>33</sup>, lze snížit latenci až na 4ms a celkově se

<sup>33</sup> <http://www.julienviet.com/reactive-pg-client/>

množství odeslaných response více než zdvojnásobí, lze říct že správná volba způsobu napojení na DB je u tohoto frameworku pro tento test klíčová. Spring dopadl nejhůře, ale i u něj existuje možnost, kterou lze zlepšit výkon a to je použití reaktivní nadstavby WebFlux, která zvýší o třetinu množství odeslaných response. Navíc při použití standardní knihovny pro připojení k databázi sníží latenci na 24,5ms což je téměř o polovinu, další zlepšení lze dosáhnout při využití knihovny rxjava2-jdbc<sup>34</sup>, se kterou se průměrná latence sníží na 5,1ms, ale také se sníží množství odeslaných response oproti standardní knihovně, avšak framework by se tak dostal do čela žebříčku co se týká latence. Pro zajímavost Node.js dosáhl maximálně 12 tis. odeslaných response, při průměrné latenci 19 ms.

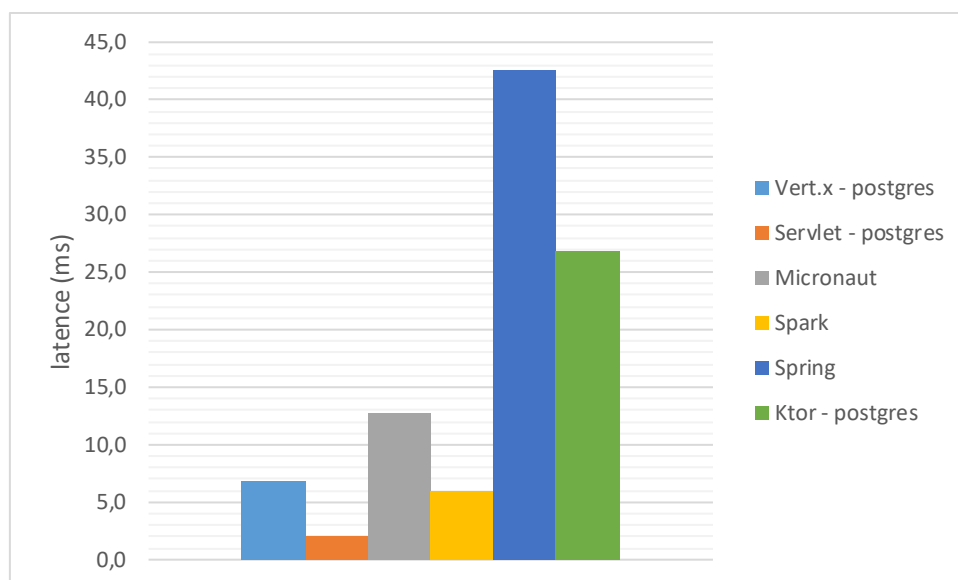
Framework	Avg. Latence (ms)	16 cli	32 cli	64 cli	128 cli	256 cli	512 cli
Vert.x - postgres	6,8	38 250	55 779	72 341	82 356	88 372	93 110
Micronaut	12,8	21 154	28 974	35 673	40 246	43 121	45 490
Servlet - postgres	2,1	33 200	44 390	47 535	47 272	47 287	44 445
Spark	5,9	15 436	17 070	17 217	17 010	17 062	17 074
Spring	42,5	11 933	12 381	13 820	14 700	14 999	15 077
Ktor - postgres	26,8	10 796	11 914	13 087	13 038	13 490	13 350

Tabulka 2 Test Signle query



Obrázek 21 Graf HTTP responses

<sup>34</sup> <https://github.com/davidmoten/rxjava2-jdbc>



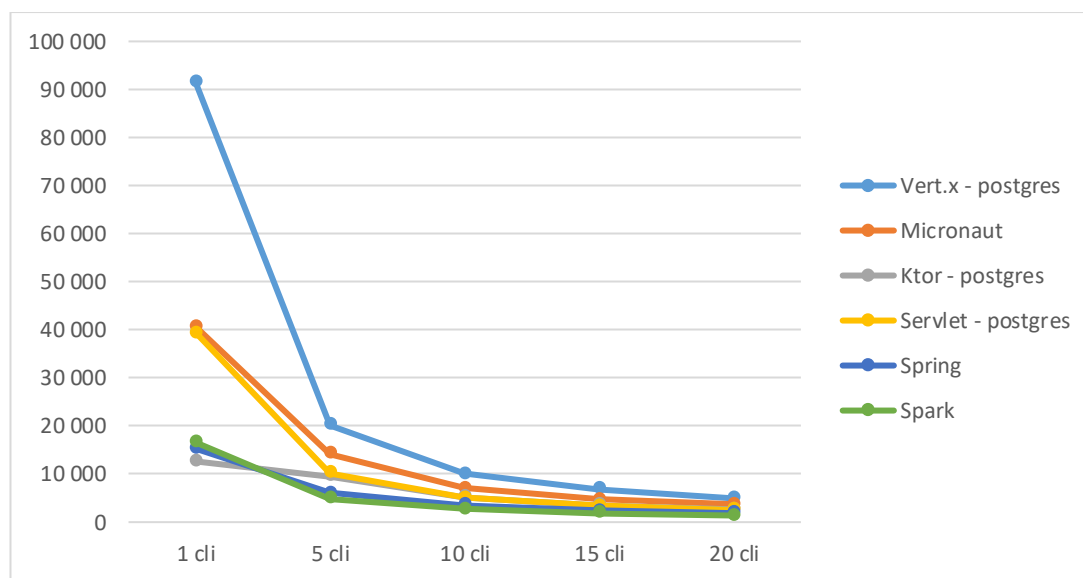
Obrázek 22 Průměrná doba odpovědi

Třetí test je variací druhého testu, který také využívá databázi slov z testu dva. Test spočívá v zasílání http requestů s path variable obsahující náhodné číslo v rozsahu 1-500, které odpovídá velikosti počtu řádků, které budou vráceny v response. Každá řádka bude vyhledána pomocí náhodného výběru, stejně jako u druhého testu. HTTP response obsahuje tělo ve formátu plain JSON obsahující list objektů obsahující položky id a random number, které jsou obě získány z jednotlivých dotazů do databáze. Oproti předchozím testům se připojovalo pouze 1, 5, 10, 15, 20 klientů paralelně. Jakékoliv features frameworků vč. cachování nebyly v implementaci využity. Vítězem se stal s výrazným nárůstem Vert.x, při použití modulu web se počty response nepatrně snížily, ale lehce se zlepšila odezva. Druhý se stal Micronaut, který dokázal o značný počet response překonat zbylé frameworky. Třetí se umístil Ktor, který při nejvyšším vytížení paralelně připojenými klienty dokázal nejlépe odesílat response bez jakéhokoliv rozšíření, naopak s knihovnou reactivepg byl výkon dvojnásobný pouze při jednom připojeném klientu, ale již při pěti paralelních klientech se výkon snížil na polovinu proti nativní implementaci, průměrná latence byla s knihovnou více než dvojnásobná. Implementace pomocí Servletu se zde nekvalifikovala na přední příčky jako u předchozích testů. Při použití rozšíření WebFlux pro framework Spring a knihovny pgclient lze dosáhnout nárůstu až na 3 858 response při dvaceti připojených klientech, což je zlepšení o více než dvojnásobek, odezva klesne z 274,4 ms na 132,2 ms což je také dvojnásobné zlepšení. Nejhuře dopadl ve srovnání Spark, avšak napříč všemi platformami poskytl dobrý výkon, pro srovnání Node.js při dvaceti

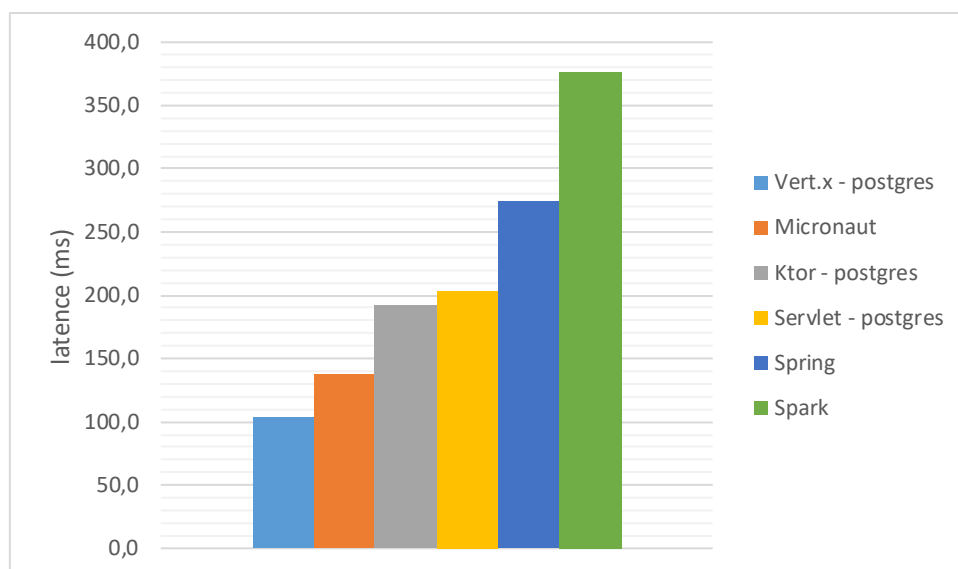
paralelních spojení zvládl odeslat pouze 865 response, což je pouze polovina, kterou zvládl Spark, navíc při průměrné latenci 581 ms.

Framework	Avg. Latence (ms)	1 cli	5 cli	10 cli	15 cli	20 cli
Vert.x - postgres	103,5	91 410	20 211	9 979	6 841	4 938
Micronaut	137,1	40 637	14 152	6 991	4 736	3 723
Ktor - postgres	191,4	12 626	9 462	5 247	3 528	2 658
Servlet - postgres	203,3	39 403	10 160	5 111	3 347	2 501
Spring	274,4	15 271	6 069	3 469	2 453	1 864
Spark	376,3	16 566	4 831	2 702	1 832	1 376

Tabulka 3 Test Multiple query



Obrázek 4 Graf HTTP responses



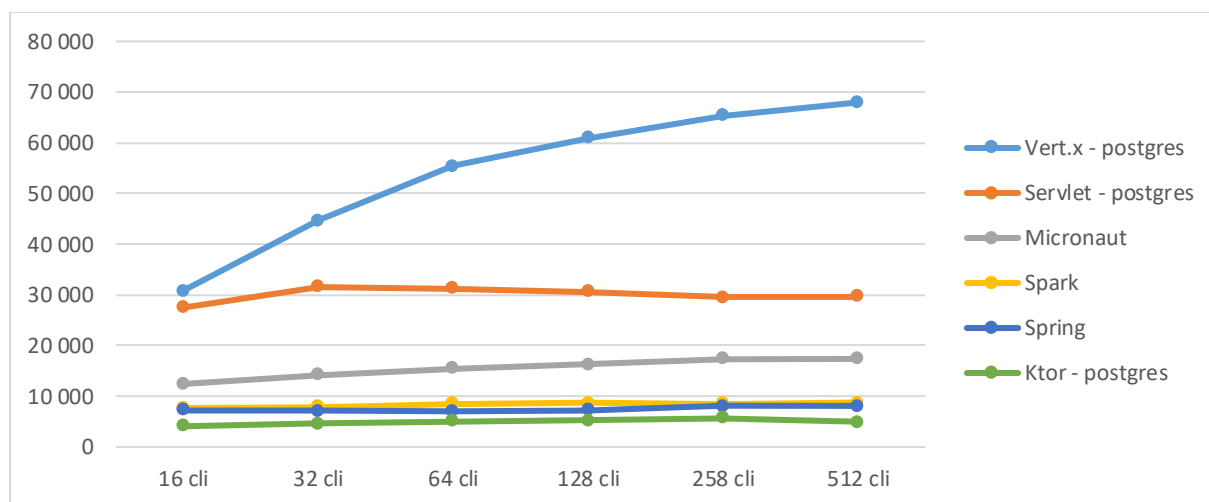
Obrázek 5 Průměrná doba odpovědi

Čtvrtý test pokrývá testování ORM, připojení k databázi, práci s kolekcemi, které mají dynamickou velikost, třídění a šablonování na serveru. Během testu se server napojí do databáze o neznámé velikosti (databáze obsahuje 12 řádků, avšak aplikace tuto informaci nemá), která obsahuje sloupec s indexem v číselné podobě a druhý sloupec obsahující textovou zprávu. Aplikace přečte postupně všechny řádky tabulky, které vkládá jednotlivě do dynamické struktury a po každém vložení je setříděna dle textu zprávy. Úplnou kolekci dat následně aplikace převede do HTML šablony, kterou odešle jako response. V tomto testu opět vyhrál framework Vert.x, který dvojnásobně předčil ostatní frameworky. Čistý framework dokázal odeslat dvojnásobek response oproti jeho variantě s modulem web, avšak toto sestavení mělo daleko lepší latence které která bylo průměrně pouze 2,7 ms oproti čistému sestavení, které dosahovalo trojnásobku. Druhý se umístila implementace pomocí Servletu, která dokázala zaslat pouze poloviční počet response, avšak dosáhla nejlepší latence ze všech testovaných frameworku a to 1,4 ms. Dále se umístil Micronaut, který dokázal zaslat poloviční počet response, i tak dosáhl při necelých 18 tisících odpovědích nadprůměrný výsledek. Zbylé frameworky zaslali méně než 10 tisíc odpovědí, což je šestinový výkon proti implementaci pomocí Vert.x. Framework Ktor, který dopadl nejhůře se 4 tisíci odpovědi, lze dostat na hranici přes 11 tisíc response při použití knihovny reactivepg, mimo jiné se sníží i latence z 48 ms na 22,7 ms a po výběru vhodné knihovny již může framework konkurovat výše zmíněnému Micronautu. Spring v čisté formě dopadl poměrně špatně, ale stejně jako v případě Ktor, lze pomocí doplňkových modulů, značně zlepšit výkony. Při využití modulu podporující reaktivní

obsahu requestů WebFlux, lze značně snížit latenci z 66 ms na pouhé 4 ms, čímž by Spring dokázal předčit i vítěze tohoto testu, avšak reaktivní verze nezvýšila počet response. Pro srovnání Node.js zvládl odeslat necelých 10 tisíc response s průměrnou latencí 24,8 ms, takže by se výsledky zařadil mezi průměrně JVM frameworky.

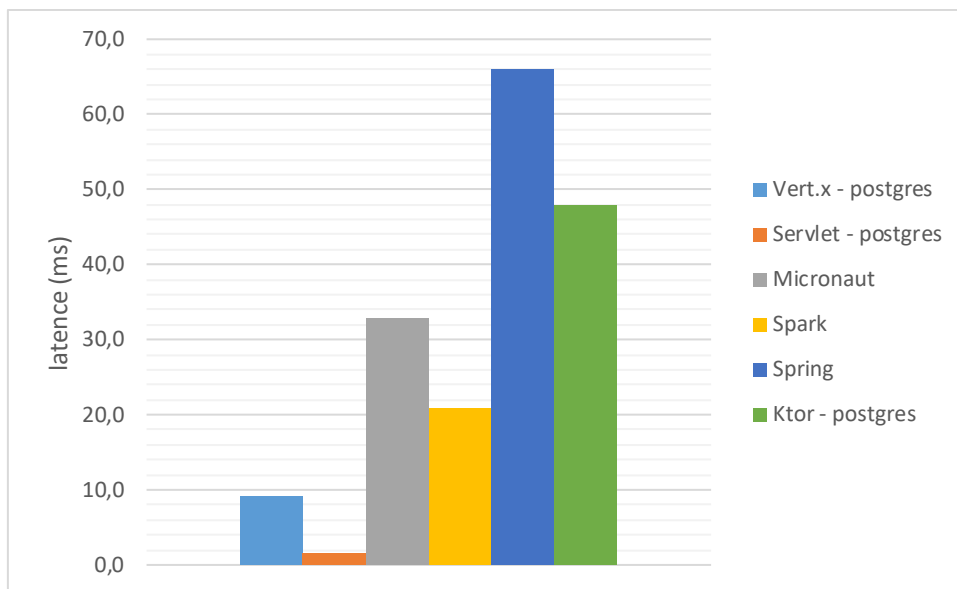
Framework	Avg. Latence (ms)	16 cli	32 cli	64 cli	128 cli	258 cli	512 cli
Vert.x - postgres	9,1	30 859	44 645	55 338	60 990	65 426	67 961
Servlet - postgres	1,4	27 500	31 548	31 293	30 574	29 498	29 649
Micronaut	32,9	12 439	14 239	15 504	16 322	17 327	17 384
Spark	20,8	7 710	7 935	8 590	8 750	8 504	8 643
Spring	66,1	7 352	7 056	7 045	7 241	8 077	8 118
Ktor - postgres	48,0	4 096	4 668	5 100	5 234	5 644	4 971

Tabulka 6 Test Fortune



Obrázek 7 Graf HTTP responses





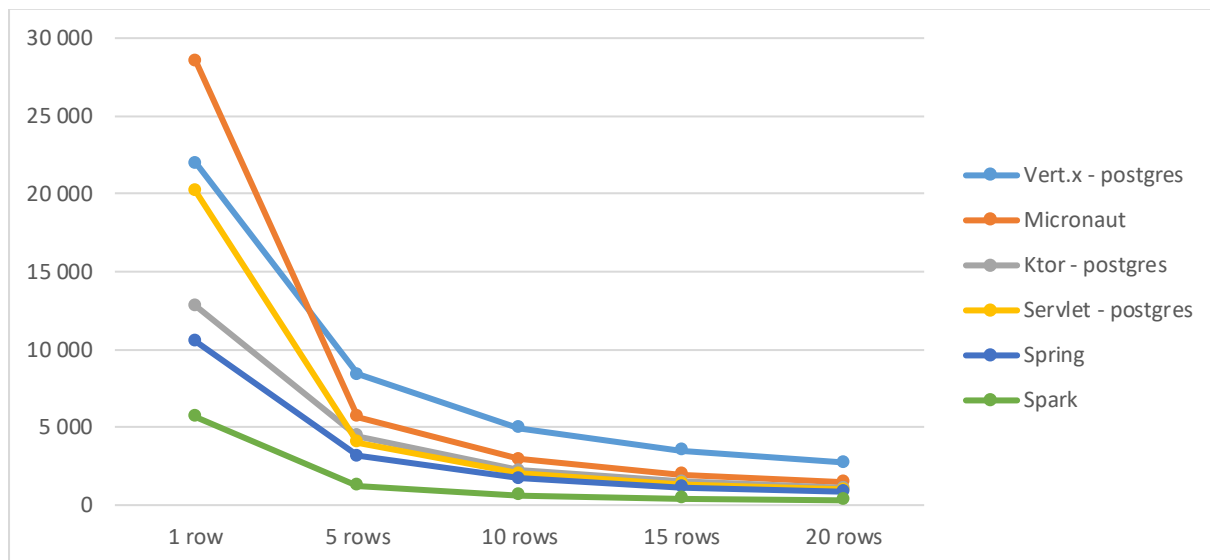
Obrázek 8 Průměrná doba odpovědi

Pátý test ověřuje načítání a aktualizaci řádků databáze. Databáze je použita z druhého testu. Po přijetí requestu aplikace načte dle zasláního parametru odpovídající počet řádků z databáze. Ty načítá jednotlivě podle náhodně generovaného indexu, pak je pomocí ORM namapuje do objektů, které jsou uloženy v operační paměti. Každému objektu uloženému v paměti upraví jeden atribut a aktualizuje odpovídající řádek v tabulce databáze, toto provede pro každý objekt jednotlivě. Poté serializuje všechny objekty z paměti jako plain JSON a ten zašle v response. Test probíhá postupně aktualizací 1, 5, 10, 15 a 20 řádek. Avšak počet dotazů do databáze je dvojnásobný, protože musíme přičíst ke každé aktualizaci řádku i dotaz na jeho získání. Testováno je vždy 512 paralelních spojení. Vert.x vyhrál tento test, jeho výsledky jsou srovnatelné i při použití modulu web. Druhý se překvapivě umístil Micronaut, který dokázal při požadavku na změnu jednoho řádku překonat i Vert.x. Ktor se umístil na třetím místě a v čisté formě dopadl o něco lépe než při použití knihovny reactivepg, která v tomto testu spíše uškodila. Výsledky Springu lze zlepšit při použití modulu WebFlux a to a zvýšením počtu response na 1 443 při nejvyšším zatížení a zlepšení průměrné latence na 351 ms. Pro srovnání Node.js zvládl odeslat při nejvyšším zatížení pouze necelých 300 response s průměrnou latencí 850 ms.

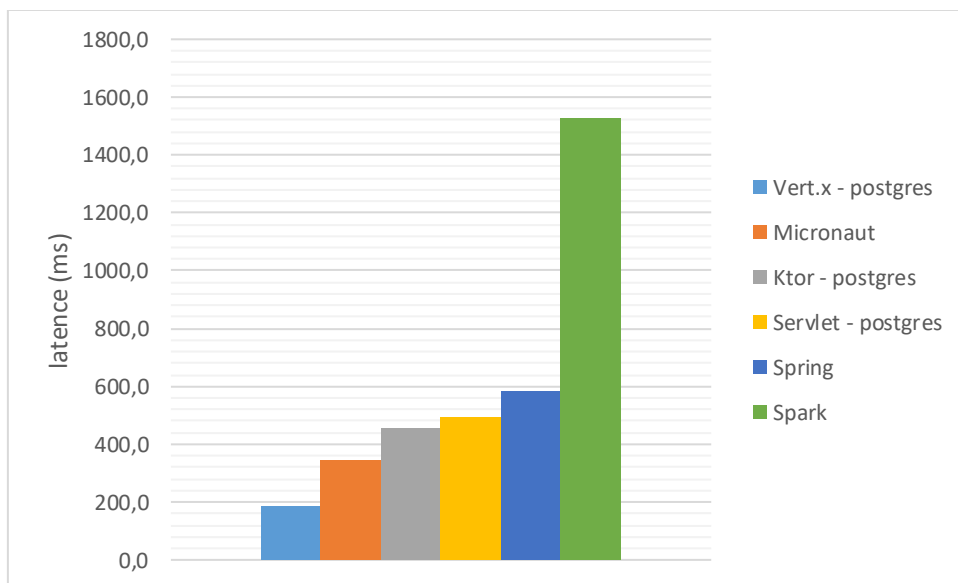
Framework	Avg. Latence (ms)	1 row	5 rows	10 rows	15 rows	20 rows
Vert.x - postgres	185,5	21 987	8 413	4 961	3 536	2 743

Micronaut	342,7	28 517	5 669	2 920	1 955	1 475
Ktor - postgres	453,0	12 789	4 453	2 221	1 492	1 113
Servlet - postgres	495,1	20 145	4 018	2 032	1 357	1 016
Spring	583,2	10 511	3 159	1 694	1 154	861
Spark	1530,0	5 653	1 230	613	412	306

Tabulka 9 Test Update



Obrázek 23 Graf HTTP responses

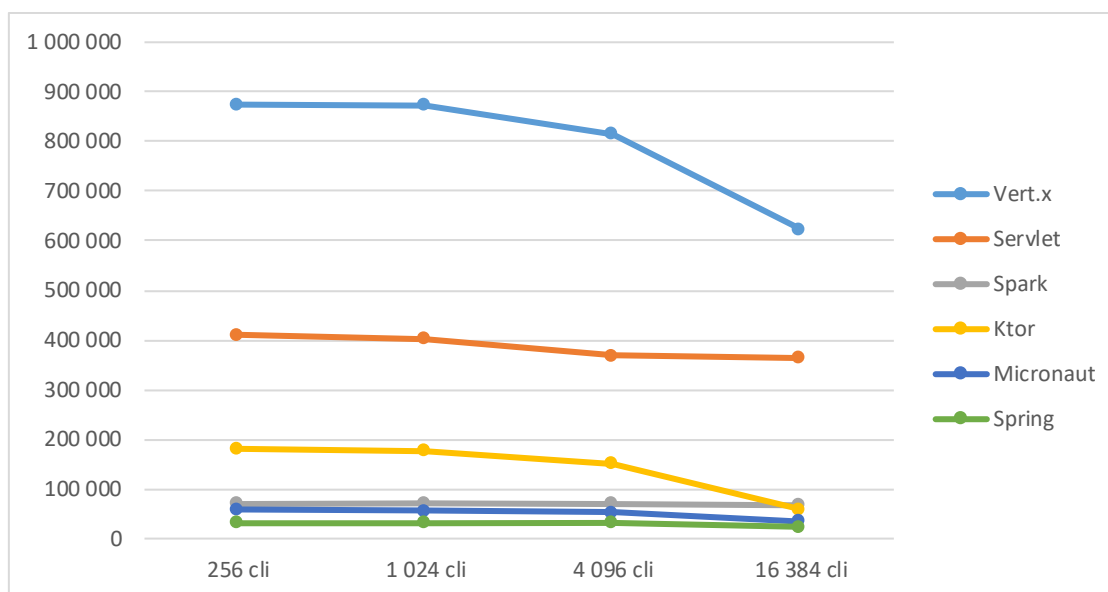


Obrázek 24 Průměrná doba odpovědi

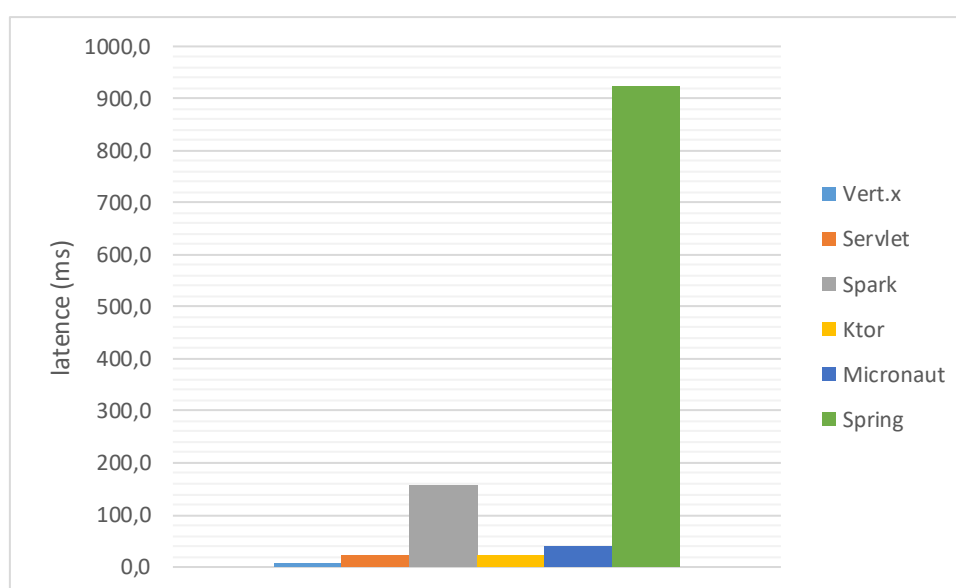
Poslední test je oproti předchozím poměrně triviální, aplikace po přijetí HTTP requestu odpoví requestem obsahující pouze plain text „Hello, World!“. Díky malé velikosti response, není Ethernet limitující faktor a nedojde k saturaci linky, proto může být značně zvětšen počet paralelních spojení oproti předchozím testům. U tohoto testu je povolen HTTP pipelining. Maximální počet testovaných paralelních klientů se zvýšil až na 16 384. Nejvíce response dokázal odeslat Vert.x, který předčil Servlet implementaci téměř o dvojnásobek. Nejlepší je využít variantu bez modulu web, který sníží výkon na polovinu a průměrnou latenci zvýší z 5,7 ms na 21,3 ms, což je téměř čtyřnásobné zpomalení. Druhý skončil Servlet, který odeslal 363 tisíc response, průměrná latence je 23,2 ms, která je srovnatelná s Vert.x při použití modulu web a napříč frameworky je průměrná. Na třetím místě se nečekaně umístil Spark avšak s latencí, která průměrně dosahuje 150 ms, je lepší využít například Ktor, který má o něco méně odeslaných response, avšak latenci má průměrně kolem 22 ms a lze jí o pár milisekund zlepšit při využití knihovny reactivepg. Micronaut a Spring se umístili na konci tabulky, markantní je u Springu jeho průměrná latence, která dosahovala 923 ms, což je v porovnání s průměrem velmi vysoká hodnota, Spring skončil na posledním místě v této metrice. Bohužel nebylo testováno sestavení Springu s reaktivním modulem WebFlux, takže není možné porovnat rozdíl jako u přechozích testů. Pro srovnání Node.js zvládl odeslat při nejvyšším zatížení 126 tisíc response s průměrnou latencí 22,2 ms, čímž se umístil v tomto testu nad průměrné JVM frameworky.

Framework	Avg. Latence (ms)	256 cli	1 024 cli	4 096 cli	16 384 cli
Vert.x	5,7	874 433	872 674	815 194	623 904
Servlet	23,2	410 661	401 967	367 873	363 714
Spark	156,1	70 972	71 032	70 276	67 051
Ktor	22,9	180 880	176 730	150 648	59 219
Micronaut	41,3	58 686	55 737	51 913	34 292
Spring	923,7	31 110	31 249	31 620	23 725

Tabulka 10 Plain text



Obrázek 25 Graf HTTP responses



Obrázek 26 Průměrná doba odpovědi

Ve fórech<sup>35</sup> se objevuje názor, že testy by bylo vhodné provádět na reálnějším prostředí ve kterém bývají nasazeny microservices v tzv. tiny containers, které mají daleko restriktivnější limity na prostředky např. systém poskytuje pouze jedno jádro procesoru a 500 MB operační paměti. Testy by bylo zajímavé v budoucnu aktualizovat, jelikož tvůrci plánují rozšíření test casů např. cachování. Samotné výkonosti testy poskytují pouze zlomek z mozaiky informací o použitelnosti frameworku. Spíše dotvářejí představu o jeho chování na testovacích případech,

<sup>35</sup> <https://github.com/TechEmpower/FrameworkBenchmarks/issues/133>

kteřé mají napodobovat situace, se kterým jsou frameworky při produkčním nasazení konfrontovány. Mementem této kapitoly může být fakt, že není pouze podstatná volba frameworku jako takového, ale je třeba brát v potaz použití modulů a knihoven jež ovlivňují výkon, a to včetně těch, které jsou proprietární. Moduly a knihovny mohou markantně zvýšit, anebo naopak snížit výkon. Toto platí i v rámci jednoho modulu či knihovny, která při jednom testu exceluje a při dalším naopak ubírá na výkonu. Tudíž je nutné nebrat jejich případy použití jak dogma, ale spíše pro jejich výběr tvořit specifické testovací případy a na nich zkoušet jejich chování a výkon.

Hodnocení výkonosti bylo započteno za každý výkonnostní test, kdy vítězný framework dostal 3 body a za každý stupeň níže byl stržen jeden bod, poslední framework nedostal tedy žádný bod. Celkem mohl framework získat až 12 bodů, dosažené výsledky byly přepočteny na bodovou stupnici 0-3, dle dosažených procent z celkových bodů. Bodové výsledky byly matematicky zaokrouhleny nahoru na celé číslo.

Framework	test 1	test 2	test 3	test 4	body	procento	hodnocení
Vert.x	3	3	3	3	12	100 %	3
Micronaut	1	2	2	2	7	58 %	2
Ktor	2	0	1	0	3	25 %	1
Spark	0	1	0	1	2	17 %	1

## Vyhodnocení

Na základě výše uvedeného hodnocení byly každému frameworku body a v této části je provedeno vyhodnocení výsledků. V tabulce je souhrn přiřazení bodů a jejich váhy. Konečný výsledek je uveden na posledním řádku tabulky. Celkový výsledek je uveden jako procento dosaženého hodnocení z maximálního možného hodnocení. Podrobná tabulka s hodnocením vah dle Fullerovy metody je uvedena v příloze.

	Vert.x	Ktor	Spark/Javalin	Micronaut	Váha kritéria
Podpora Kotlinu	2	3	3	2	13/105
Web	3	3	3	3	10/105
Security	2	2	3	3	9/105
Templating	2	3	3	2	1/105
Caching	1	1	1	3	1/105
Dependency injection	1	3	1	3	6/105
JSON	2	3	3	3	7/105
Integrace	3	1	0	3	4/105
Přístup k datům	2	1	0	3	5/105
AOP	0	1	1	3	3/105
Scheduling	1	1	0	3	1/105
Testovatelnost	1	2	2	3	11/105
Podpora a komunita	3	1	2	3	10/105
Praxe	3	2	1	3	11/105
Výkon	3	1	1	2	9/105
Hodnocení	73%	68%	63%	92%	

Výsledky jsou poměrně těsné, jediný framework, který výrazně vybočuje je Micronaut. Micronaut se tak stal vítězem, u tohoto frameworku je poměrně malé množství oblastí, ve kterých by v současnosti zaostával. Místa pro rozvoj shledávám v lepší podpoře Kotlinu, kde je poměrně velká oblast pro zlepšení a dále pak ve výkonu, jenž v některých výkonnostních testech lehce zaostával. Micronaut lze jednoznačně doporučit pro produkční nasazení, výhodou je jeho inovativní přístup, při kterém ale zachovává syntaktickou a sémantickou blízkost k tradičním webovým frameworkům. Druhý se umístil Vert.x jenž dokázal předčít všechny frameworky po výkonnostní stránce, funkčně je dostatečný pro produkční nasazení. Avšak jedná se o poměrně náročný framework, jenž vyžaduje delší dobu osvojení. Třetí se umístil Ktor, jenž po funkční stránce dostačuje pro nasazení, navíc výborně podporuje Kotlin, avšak celkově se jedná o nezralý framework, ke kterému v současnosti není dostatek referencí.

Nelze ho bez výhrad doporučit pro produkční nasazení v současné době. Jako poslední se umístil minimalistické frameworky Javalin a Spark. Tyto frameworky nepokrývají všechny oblasti, díky tomu získali znatelně nižší hodnocení, avšak v pokrytých oblastech svoji funkci plní výborně. Lze je doporučit pro prototypování a menší aplikace, nelze je však doporučit pro enterprise aplikace.

Všechny frameworky výborně podporují jejich primární účel, a to tvorbu webových aplikací. Největší rozdíly jsou pak v rozsahu další přidané funkčnosti, která slouží jako podpora hlavního modulu. Frameworky poměrně rozdílně podporují Kotlin, nejlépe jsou na tom ty, které jsou v Kotlinu přímo napsány, u ostatních je nejvyšší stupeň integrace poskytnutí wrapperu pro hlavní části frameworku. V bezpečnosti jsou frameworky poměrně vyrovnané, liší se pouze poskytovanými integracemi s bezpečnostními standardy, v tomto kritériu nelze vybrat špatně mezi testovanými frameworky. Šablonování podporují všechny frameworky většinu standardních enginů. Caching je k dispozici pouze u frameworků, které umožňují svojí strukturou automatizovaně aplikovat cache, většinou je k této funkčnosti nutná podpora AOP v rámci frameworku. Dependency injection je poměrně kámen úrazu u micro-frameworků, které nativně většinou nepočítají s využitím DI díky svojí odlišné struktuře oproti klasickým webovým frameworkům, většina nabízí využití externích nástrojů, avšak většinou za cenu narušení syntaxe a sémantiky oproti čistému kódu bez DI. Automatizovanou konverzi zpráv podporují všechny frameworky, avšak na různé úrovni autonomie. Integrace služeb třetích stran bylo kritérium, ve kterém byly mezi frameworky diametrální rozdíly. Některé nabízely mnoho modulů a některé vůbec žádné, zde je nutné zvážit případný čas, který by bylo nutné investovat pro manuální integrace. Stejně tak na tom byl přístup k datům, ve kterém měli nedostatky všechny frameworky, kromě vítězného Micronautu, na rozdíl od kritéria integrací se jedná o poměrně zásadní funkčnost, jejíž absence značně ovlivní dobu vývoje aplikace. Zbylé dvě kritéria v rámci modulů jsou poměrně málo významné, jedná se o podpory aspektového programování a plánování úloh, obě funkce lze substituovat pomocí knihoven, či interních funkcí, avšak u enterprise aplikací může jít o majoritně využívané funkce, jejichž nativní podpora značně ulehčí vývoj, oproti menším aplikacím, kde se tato funkcionalita využívá pouze jako podpůrná. Testovatelnost je u frameworku poměrně rozdílná, všechny frameworky podporují testy, avšak nepodporují nativně vysoký stupeň izolace zvláště díky jejich struktuře, která se skládá z vnořených funkcí. Separace business logiky je tak zcela v rukou vývojáře a

vyžaduje vyšší kázeň a zkušenosti. Oproti tomu klasicky strukturované frameworky poskytují větší izolaci, která je i nativně podporovaná, včetně jednoduchého mockování závislostí, a to včetně bean. Podpora a komunita je velmi důležité kritérium pro volbu micro-frameworku, jelikož všechny jsou poměrně krátkou dobu od počátečního vydání a vývojář musí často „prošlapávat cestu“, při řešení specifických úloh, a ne všechny frameworky mají aktivní komunitu a autory jež svižně reagují na její potřeby. V praxi se se frameworky značně lišili, zvláště v obtížnosti jejich osvojení a podpoře nástroji u tohoto, kritéria je nutné delší zkoušení, nejlépe přepsání menší vlastní aplikace. Jelikož ne všechna úskalí se projeví při zjednodušených testovacích aplikacích, které zejména demonstrují pouze silné stránky. Po výkonnostní stránce jsou značné rozdíly mezi frameworky, je však nutné brát v úvahu, že šlo o testovací případy a pro aplikace v nichž je kladen důraz na rychlost, je nutné vytvořit testovací případy na míru, či vytvořit jednoduchý prototyp aplikace jenž může být rychle implementován a testován.

## Závěr

V teoretické části práce byla představena v první kapitole historie vývoje serverových aplikací. Ve druhé byly popsány současné trendy, kterými se vývoj těchto aplikací ubírá. Třetí kapitola poskytuje informace o využití frameworků v současné době, informace pocházejí ze syntézy více zdrojů. Tato část sloužila jako podkladová informace pro výběr technologie, kterou se má v současnosti smysl více zabývat. Následující kapitola rozebírá celou problematiku webových frameworků a v této kapitole jsou vyhrazeny všechny aspekty, které se dále v praktické části práce ověřovali.

Praktická část začíná kapitolou, ve které se řeší výběr frameworků k testování. V následující části jsou vybrána a zdůvodněna kritéria podle kterých budou frameworky hodnoceny a potažmo co se v rámci daného kritéria hodnotí. Dále pak následují kapitoly, které se věnují podrobným popisem a hodnocením jednotlivých frameworků. Po této části následuje kapitola, ve které jsou popsány způsoby výkonnostního testování a uvedeny výsledky měření



frameworků. Tato kapitola končí vyhodnocením celé sekce měření. Poslední kapitola praktické části se zabývá vyhodnocením finálních výsledků a shrnutí nabytých poznatků.

Na základě dosažených výsledků, lze konstatovat, že práce naplnila stanovené cíle práce. Podařilo se zmapovat a vyhodnotit současné možnosti, které jsou při volbě jazyka Kotlin pro serverový vývoj k dispozici. Tato práce může být dále rozšířena, či upravena jednak s ohledem na rozvoj testovaných frameworků, tak i pro rozšíření o další nové frameworky. Upraveny mohou být váhy kritérií a tím pádem je možné přizpůsobit hodnocení frameworků pro individuální potřeby čtenáře.

## Použitá literatura

### Internet:

DANILOV, Anton. Analýza programovacího jazyka Kotlin, jeho srovnání s jazykem Java 8 a praktické využití při vývoji na platformě Android [online]. Praha, 2017 [cit. 2019-06-11]. Dostupné z: <<https://theses.cz/id/u5c85i/>>. Diplomová práce. Vysoká škola ekonomická v Praze. Vedoucí práce Jarmila Pavlíčková.

JAROŠ, Jan. Použitelnost jazyka Kotlin při vývoji mobilních aplikací [online]. Praha, 2018 [cit. 2019-06-11]. Dostupné z: <<https://theses.cz/id/fps0ec/>>. Bakalářská práce. Vysoká škola ekonomická v Praze. Vedoucí práce Rudolf Pecinovský.

MAI, Duc Anh. Transformation of Android Mobile App from Java to Kotlin [online]. Praha, 2018 [cit. 2019-06-11]. Dostupné z: <<https://theses.cz/id/fukc4l/>>. Diplomová práce. Vysoká škola ekonomická v Praze. Vedoucí práce Alena Buchalceová.

MOTTL, Jan. Moderní vývoj aplikací pro platformu Android [online]. Praha, 2018 [cit. 2019-06-11]. Dostupné z: <<https://theses.cz/id/tk66dq/>>. Diplomová práce. Vysoká škola ekonomická v Praze. Vedoucí práce Jarmila Pavlíčková.

KOZÁKOVÁ, Eva. Specifikace server-side technologií, jejich implementace a porovnání [online]. Hradec Králové, 2019 [cit. 2019-06-11]. Dostupné z: <<https://theses.cz/id/u461g4/>>. Diplomová práce. Univerzita Hradec Králové, Fakulta informatiky a managementu. Vedoucí práce doc. Ing. Filip Malý, Ph.D..

NEZDARA, Vojtěch. Vývoj moderních webových aplikací [online]. Praha, 2016 [cit. 2019-06-11]. Dostupné z: <<https://theses.cz/id/uqff1w/>>. Diplomová práce. Vysoká škola ekonomická v Praze. Vedoucí práce Alena Buchalceová.

### Zahraniční:

Perception and effects of implementing Kotlin in existing projects:

<https://pdfs.semanticscholar.org/85c4/824a4ae6a2978d60cb56c58f40b8b1b439da.pdf>

Performance Evaluation of Kotlin and Java on Android Runtime:

[http://www.nada.kth.se/~ann/exjobb/patrik\\_schwermer.pdf](http://www.nada.kth.se/~ann/exjobb/patrik_schwermer.pdf)

Developer survey 2019:

<https://insights.stackoverflow.com/survey/2019#technology--most-loved-dreaded-and-wanted-web-frameworks>

Developer survey 2018:

<https://insights.stackoverflow.com/survey/2018#technology>

Devecosystem 2018:

<https://www.jetbrains.com/research/devecosystem-2018/>

Devecosystem – Kotlin 2018:

<https://www.jetbrains.com/research/devecosystem-2018/kotlin/>

Devecosystem 2019:

<https://www.jetbrains.com/research/devecosystem-2019/>

Devecosystem – Kotlin 2019:

<https://www.jetbrains.com/research/devecosystem-2019/kotlin/>

Ranking Programming Languages by GitHub Users:

<https://www.benfrederickson.com/ranking-programming-languages-by-github-users/>

PYPL Popularity of Programming Language:

<http://pypl.github.io/PYPL.html>

TIOBE Index for June 2019:

<https://www.tiobe.com/tiobe-index/>

Knihy:

Samuel, S. and Bocutiu, S. (n.d.). *Programming kotlin*.

Denis Kalinin, Modern Web Development with Kotlin

Alexey Soshin, Hands-on Design Patterns with Kotlin: Build scalable applications using traditional, reactive, and concurrent design patterns in Kotlin

Igor Kucherenko, Mastering High Performance with Kotlin: Overcome performance difficulties in Kotlin with a range of exciting techniques and solutions

Raghavendra Rao K - Raghavendra Rao K , Kotlin for Enterprise Applications using Java EE  
Raghavendra Rao K Packt Publishing Ltd

Java magazine: march-april [online]. Oracle, 2019 [cit. 2019-09-28]. Dostupné z:

<https://blogs.oracle.com/javamagazine/>

Vuksanovic, Irena & Sudarevic, Bojan. (2011). Use of web application frameworks in the development of small applications. 458-462.

Ostatní:

<https://softwareengineering.stackexchange.com/questions/158745/what-was-the-first-server-side-language>

<https://www.zive.cz/clanky/java-vs-c---ktery-jazyk-zvolit/sc-3-a-104694/default.aspx>

<https://dri.es/a-history-of-javascript-across-the-stack>

<https://www.quora.com/Are-there-languages-that-make-better-use-of-a-JVM-than-Java>

<https://dzone.com/articles/benchmarking-scala-against>

[https://w3techs.com/technologies/overview/programming\\_language/all](https://w3techs.com/technologies/overview/programming_language/all)

<https://octoverse.github.com/projects#languages>

<https://realm.io/realm-report/>

<https://hackernoon.com/major-programming-trends-to-prepare-for-in-2019-169987cc75f4>

<https://techbeacon.com/app-dev-testing/9-code-framework-trends-watch-2018>

<https://www.tomaskubica.cz/post/2017/serverless-existuji-aplikace-bez-serveru/>

<https://martinfowler.com/articles/serverless.html>

Erb, Benjamin. (2012). Concurrent Programming for Scalable Web Architectures. 10.18725/OPARU-2423.

MOSELEY, Ben MARKS, Peter: Out of the Tar Pit, (2006)

<https://berb.github.io/diploma-thesis>

<https://www.forbes.com/sites/forbestechcouncil/2018/10/05/15-predictions-for-the-next-big-thing-in-software-development/#30a935e0522f>

<https://dzone.com/articles/trends-in-enterprise-software-development-2019-and>

[https://dzone.com/articles/serverless-architectures-against-containerized-arc?edition=491306&utm\\_source=Daily%20Digest&utm\\_medium=email&utm\\_campaign=Daily%20Digest%202019-06-25](https://dzone.com/articles/serverless-architectures-against-containerized-arc?edition=491306&utm_source=Daily%20Digest&utm_medium=email&utm_campaign=Daily%20Digest%202019-06-25)

<https://www.ludekvesely.cz/proc-pouzivat-kubernetes/>

<https://medium.com/sw-d/microservice-4609d656ccf8>

<https://blogs.oracle.com/arungupta/non-blocking-io-using-servlet-31:-scalable-applications-using-java-ee-7-totd-188>

<https://projectreactor.io/>

<https://medium.com/priceline-labs/choosing-a-reactive-framework-for-the-jvm-ec66f6cde552>

<https://djangostars.com/blog/what-is-a-web-framework/>

<https://pdfs.semanticscholar.org/8353/e9b61783ef7d3a06bcca207d91b624c45c6e.pdf>

<https://www.sciencedirect.com/science/article/pii/S187770581400352X?via%3Dihub>

<https://www.techopedia.com/definition/24649/three-tier-architecture>

<https://dzone.com/articles/modern-type-safe-template-engines>

<https://dzone.com/articles/modern-type-safe-template-engines-part-2>

<https://searchstorage.techtarget.com/definition/cache>

[https://dzone.com/articles/where-is-my-cache-architectural-patterns-for-cachi?edition=521353&utm\\_source=Weekly%20Digest&utm\\_medium=email&utm\\_campaign=Weekly%20Digest%202019-10-02](https://dzone.com/articles/where-is-my-cache-architectural-patterns-for-cachi?edition=521353&utm_source=Weekly%20Digest&utm_medium=email&utm_campaign=Weekly%20Digest%202019-10-02)

<https://www.cubrid.org/blog/inside-vertx-comparison-with-nodejs/>

<https://www.celum.com/de/blog/technologie/the-reactor-pattern-and-non-blocking-io>

<https://www.kotlinddevelopment.com/why-kotlin-for-backend-is-a-real-thing/>

<https://vertx.io/>

<https://ktor.io>

<https://blog.kotlin-academy.com/ktor-for-fast-server-prototyping-6e7c6d2ec296>

<https://19yw4b240vb03ws8qm25h366-wpengine.netdna-ssl.com/wp->

<content/uploads/using-spark-java-to-program-apis.pdf>

<https://javalin.io/>

<http://sparkjava.com/>

<https://micronaut.io/>

<https://objectcomputing.com/news/tag/micronaut>

## Přílohy

### Příloha A: Fullerův trojúhelník pro stanovení vah hodnotících kritérií

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1		1	1	1	1	1	1	1	1	1	1	1	1	1	1
2			1	1	1	1	1	1	1	1	1	0	0	0	1
3				1	1	1	1	1	1	1	1	1	0	0	0
4					0	0	0	0	0	0	1	0	0	0	0
5						0	0	0	0	0	0	0	0	0	0
6							0	1	1	1	1	0	0	0	0
7								1	1	1	1	0	0	0	0
8									0	1	1	0	0	0	0
9										1	1	0	0	0	0
10											1	0	0	0	0
11												0	0	0	0
12													1	1	1
13														0	1
14															1
15															

Zdroj: autor

1. Podpora Kotlinu
2. Web
3. Security
4. Templating
5. Caching
6. Dependency injection
7. JSON
8. Integrace
9. Přístup k datům
10. AOP
11. Scheduling
12. Testovatelnost
13. Podpora a komunita
14. Praxe
15. Výkon