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 (využity 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 implementace/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

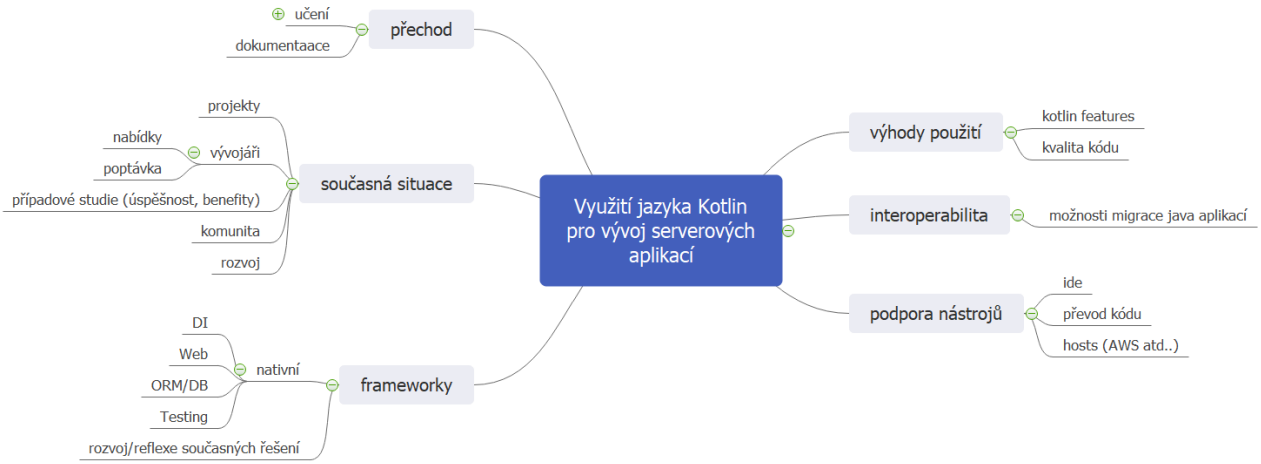
Moduly Security (+ ošetření současných hrozeb), Templating, Caching, Form validation, Page Navigation (+ POST, Redirect…)

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


Využití jazyka Kotlin pro vývoj serverových aplikací
přechod
učení
dokumentaace
současná situace
projekty
vývojáři
nabídky
poptávka
případové studie (úspěšnost, benefity)
komunita
rozvoj
frameworky
nativní
DI
Web
ORM/DB
Testing
rozvoj/reflexe současných řešení
výhody použití
kotlin features
kvalita kódu
interoperabilita
možnosti migrace java aplikací
podpora nástrojů
ide
převod kódu
hosts (AWS atd..)

Historie vývoje serverových aplikací

Server je označení pro zařízení, nebo počítačový program, který poskytuje funkcionalitu, případně služby klientům, kteří jsou většinou reprezentovány 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 tom 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 rozraní 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 problému 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 dere 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 vznikal 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 ojedinělý.

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 Langugae), dnes nazýván CIL (Common Intermediate Langugae), 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 opoustit 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[1] a alokace paměti, kdy některé jazyky dokázali předčit samotnou Javu, zejména v kódu

[1] https://github.com/kostya/benchmarks
https://www.slideshare.net/CorneilduPlessis/performance-comparison-jvm-languages

s funkcionálními prvky. Jazyky také implementovali moderní koncepty a netradiční přístupy, které Java postrádala.

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[2]. Aktivně je v komerčních projektech využívalo poměrně malé množství vývojářů v porovnání s původním jazykem postaveným nad JVM a to Javou.

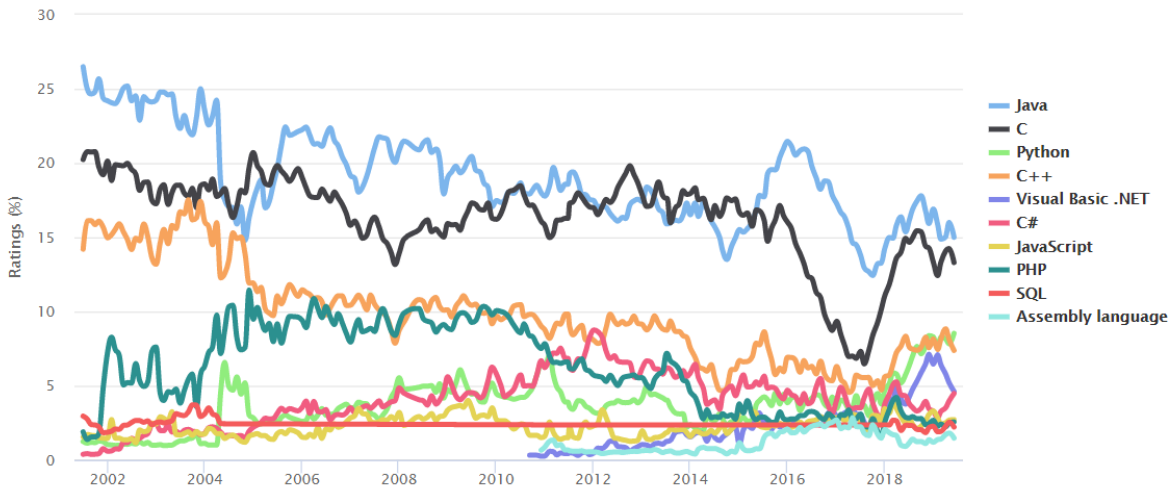
Jeden z mála jazyků, který zaznamenal výraznější úspěch než výše uvedené a přitáhl pozornost mnoha vývojářů je Kotlin. Jazyk, který sází na plnou kompatibilitu s Javou a velmi rychlou adaptaci Java vývojáři. Kotlin je silně staticky typovaný jazyk, umožňující vývoj podle nejpoužívanější paradigmat (procedurální, 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.

[2] https://dzone.com/articles/the-rise-and-fall-of-jvm-languages

Současnost ve vývoji serverových aplikací

Ke zmapovaní 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 směrodatný 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 umisťuje 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.

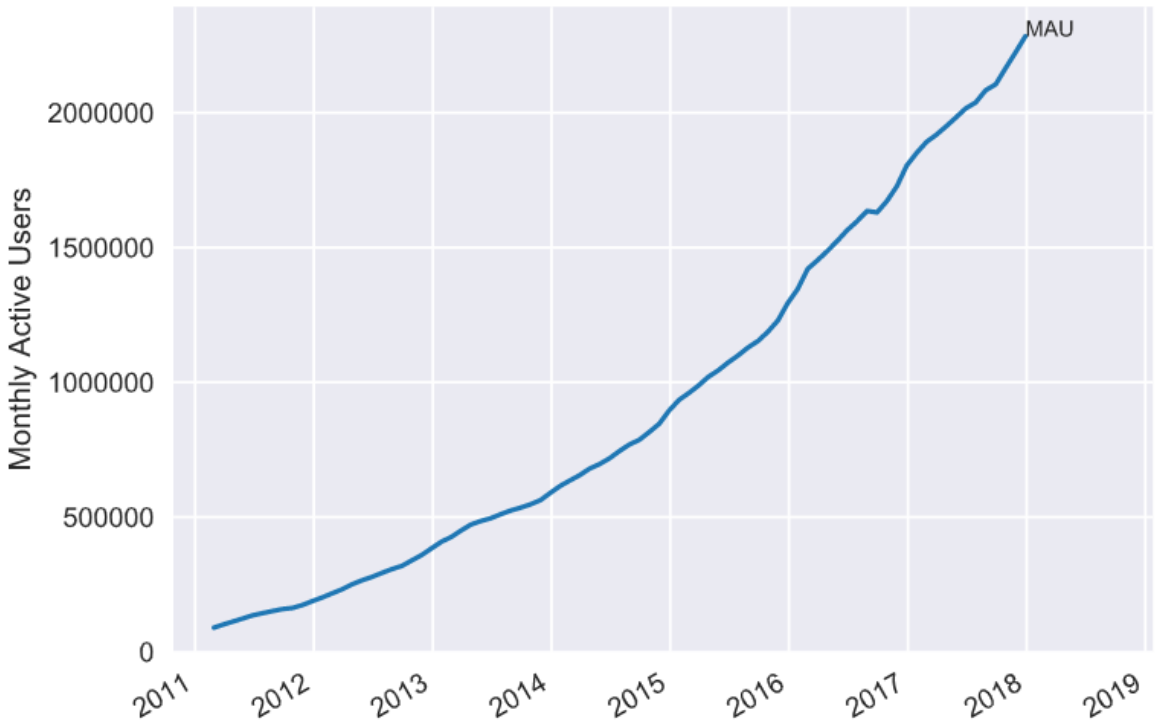


Obrázek 1 TIOBE Index

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 zájmem 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.

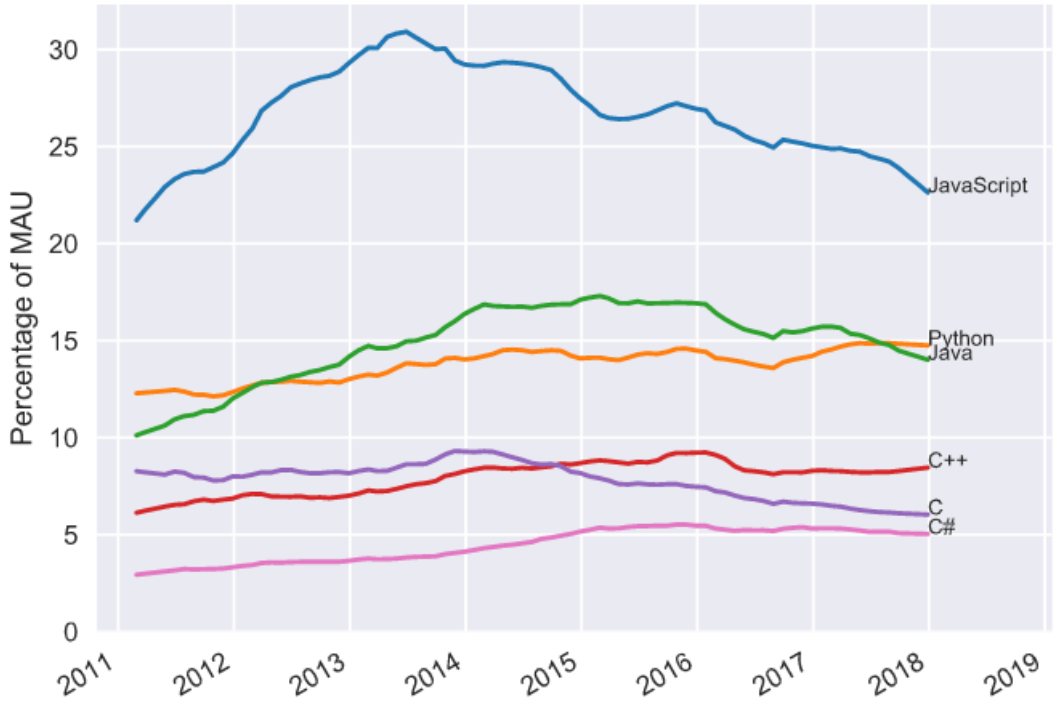
Ben Fredirickson vytvořil hodnocení jazyků podle dat o struktuře repositářů na GitHub, což je veřejné uložiště systému pro verzování Git. Do úložiště přispívá celkem 37 miliónů uživatelů, kteří spravují dohromady 75 miliónů repositářů a uskutečnili již celkem 1,25 miliard transakcí. Každá transakce je spárovaná s konkrétním repositářem a uživatelem. Fredirickson 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

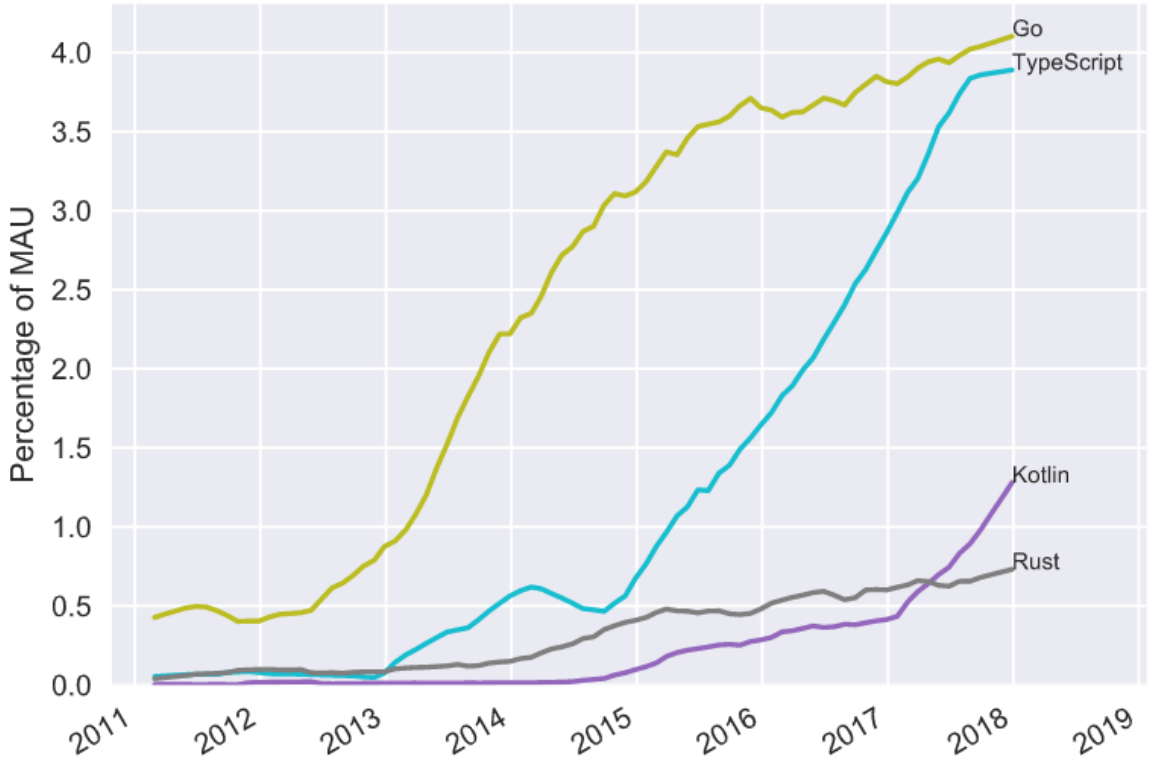
jeho uvedení jako full-stack řešení. Druhý Python s 15 % a mírným růstem, Java třetí se 14 % a klesajícím trendem, PHP se umístilo na šestém místě s 6 % a velmi výrazným klesajícím trendem, na sedmém C# s 5 %, které se drží již delší dobu na stejné hodnotě. Kotlin se objevil na patnáctém místě s 1,3 % 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

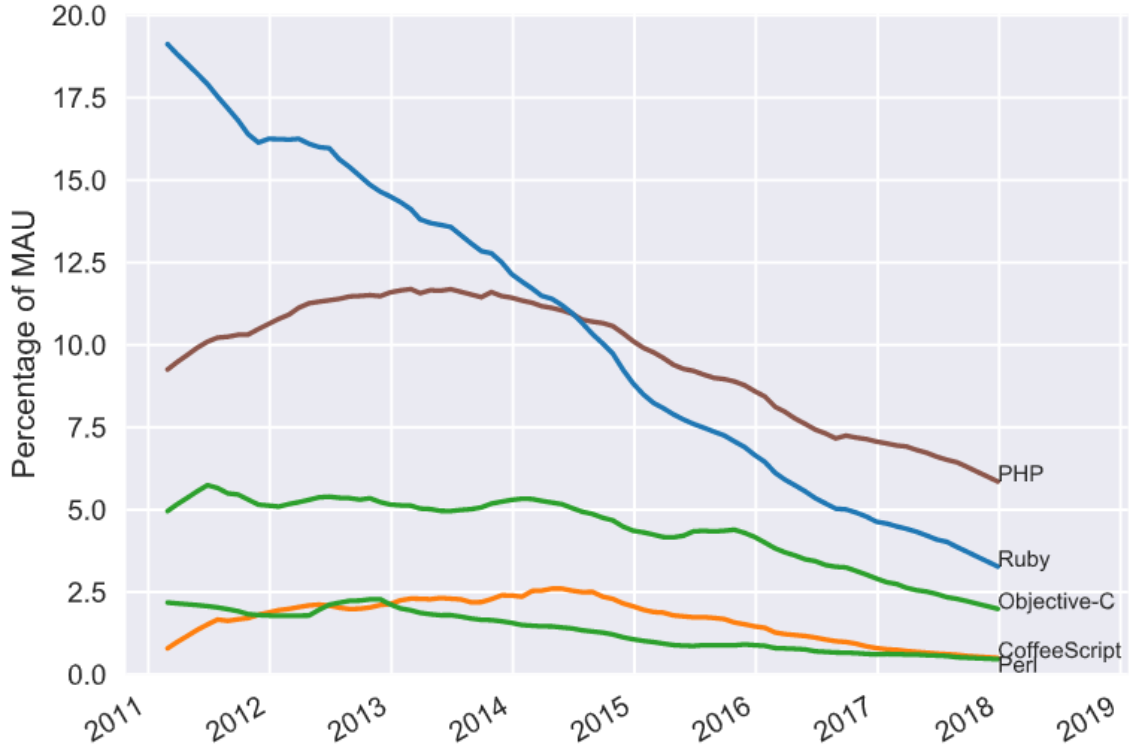
do jazyka Go přidávány poměrně konzervativně, což mu někteří kritici vyčítají. Do jazyka Go jsou v současnosti nejvíce přepisovány síťové a webové aplikace na kterých je po převodu běžně zaznamenáván nárůst rychlosti v násobcích o řádu desítek a v některých případech dokonce až stovek. Druhým jazykem 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 oblíbenost v komunitě vývojářů. Třetí je jazyk Kotlin, který zažívá poměrně strmý nárust 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.



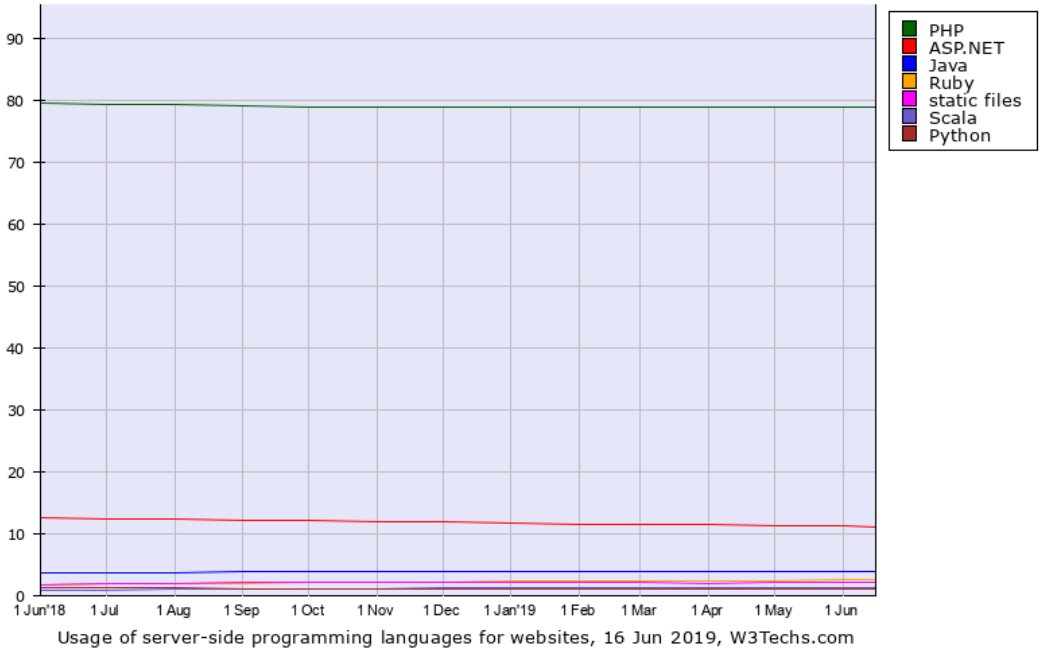
Obrázek 4 Relativní přírůstek aktivních uživatelů

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 častěji 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 miliónech dle návštěvnosti. Z toho PHP využívá téměř 8 miliónů nejnavštěvovanějších stránek světa. Za zmínku stojí i pokles jazyka Perl, který byl zmíněn v kapitole o historii vývoje.

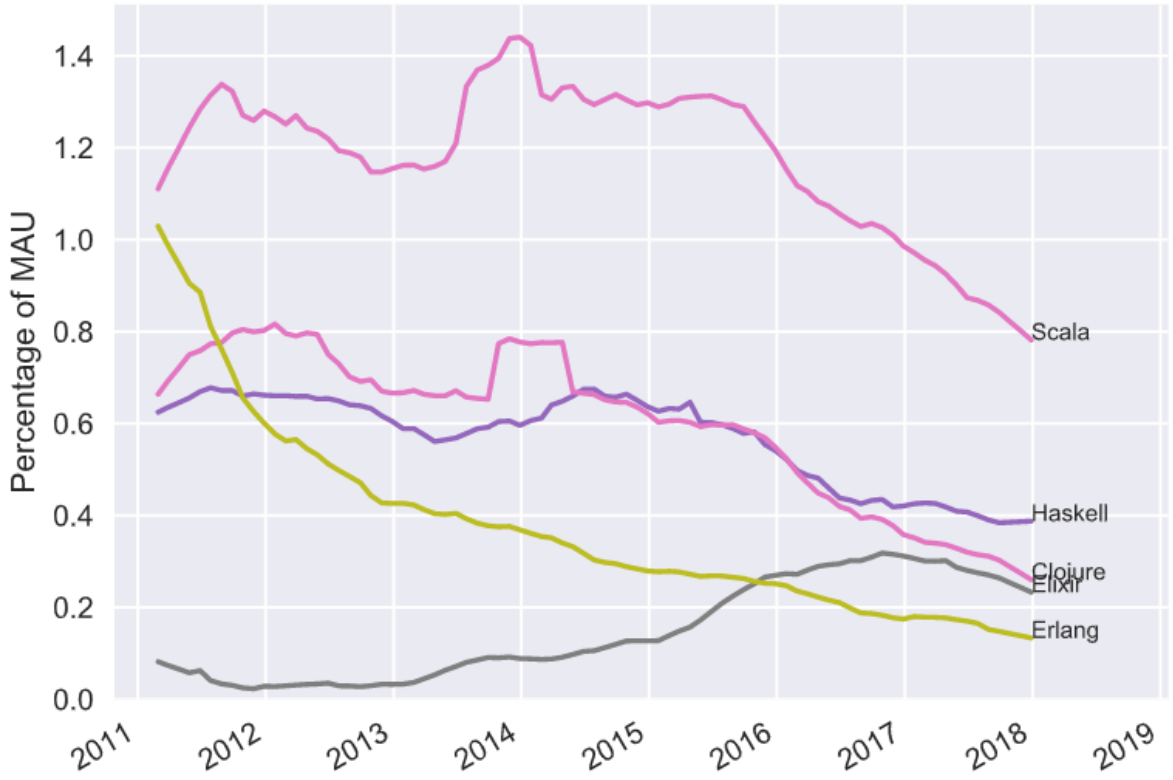


Obrázek 5 Úbytek aktivních uživatelů



Obrázek 6 Průzkum společnosti W3Techs

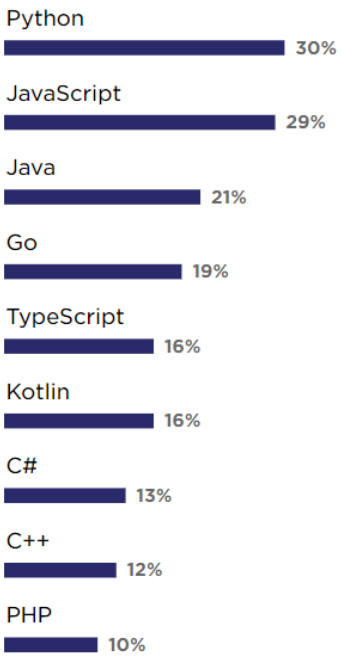
Poslední zajímavou statistikou je také přehled oblíbenosti funkcionálních jazyků, které byly v popředí zájmů i pro vývoj serverových aplikací. Z grafu je patrné že největší boom pro funkcionální jazyky byl mezi roky 2013–2015. Avšak současný trend pro všechny funkcionální jazyky je dlouhodobě klesající.



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.

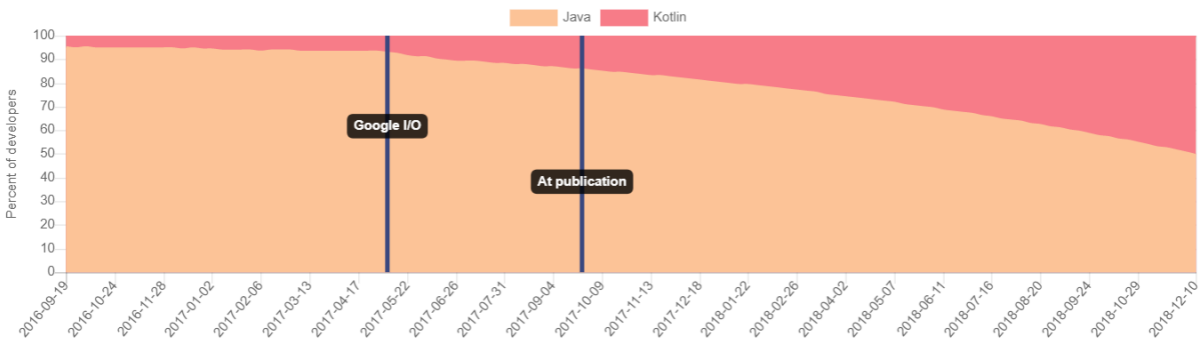


Obrázek 8 výsledky průzkumu State of Developer 2018 - zájem o osvojení nového programovacího jazyka

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

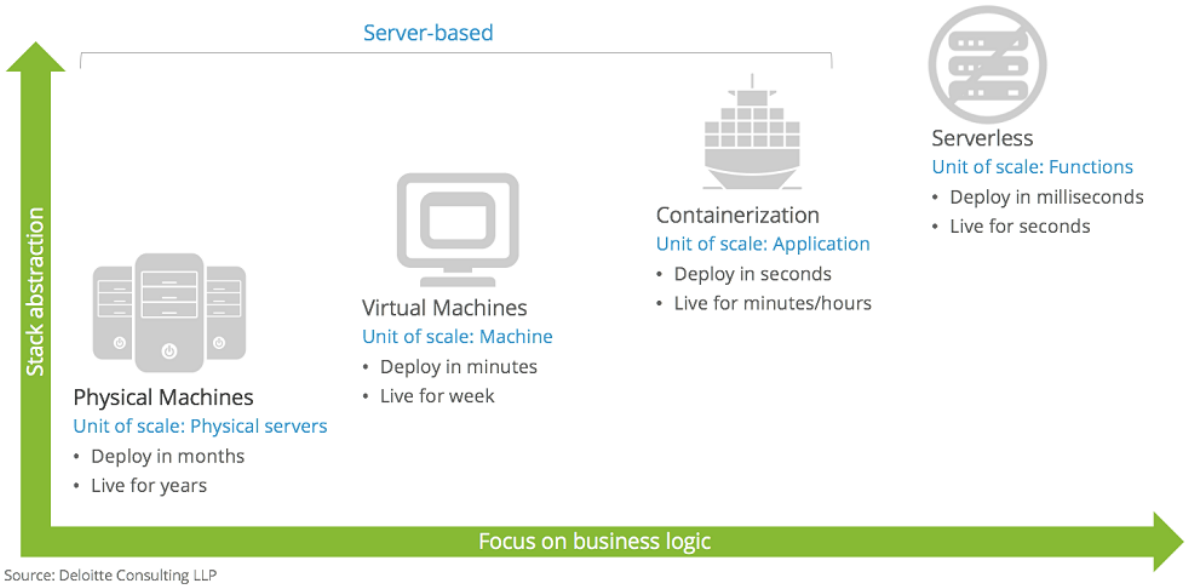
V době, kdy se již dostáváme na hranice možností křemíků a škálovat výkon aplikaci, je nutné jinými způsoby než jen pouhým navyšováním výkonu hardwaru. Ke slovu přichází paralelizace a distribuované zpracování, které se čím dál častěji implementuje právě v serverových aplikacích. Trend současnosti je 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 paradigmatem 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 niž 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 aplikacích 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



Source: Deloitte Consulting LLP

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.[3] Dále usnadňuje škálovatelnost vytížených služeb a také podporují principy continuous delivery a principy DevOps[4], 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 oblibu JVM micro-frameworků zareagovalo i nejpoužívanější[5] IDE mezi Java vývojáři Idea IntelliJ a ve verzi 2019.1.3, přidává podporu[6] pro mnoho z nich, mimo jiné i pro Micronaut.

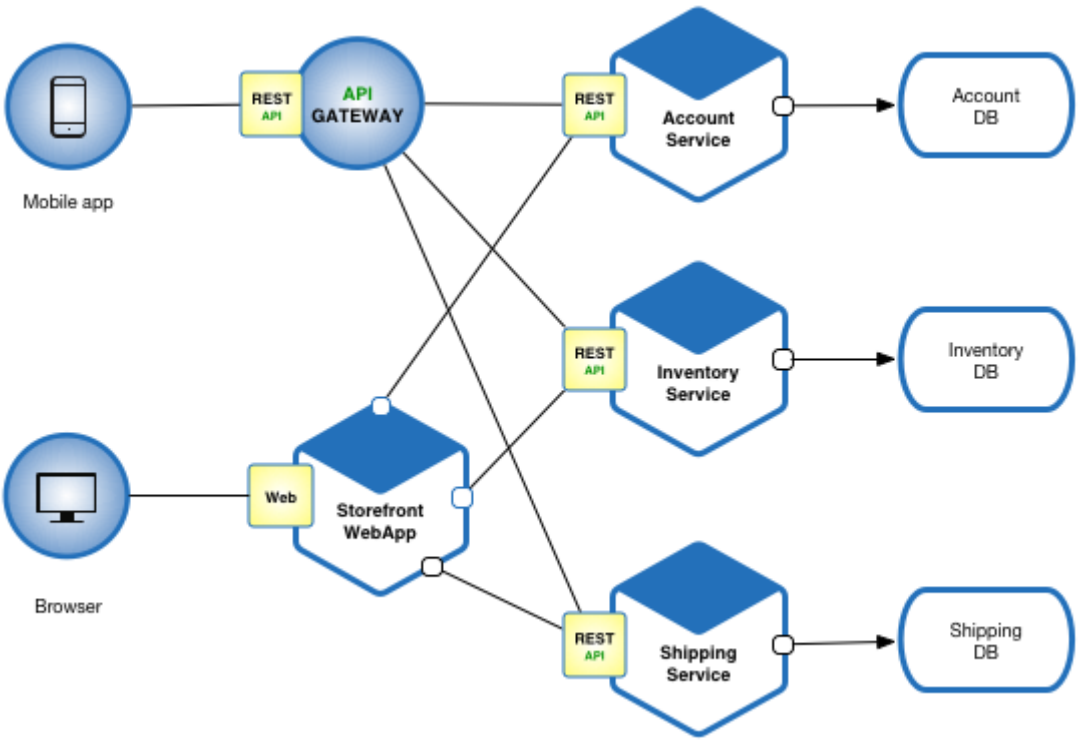
[3] Richardson, Chris. "Microservice architecture pattern". *microservices.io*. Retrieved 2017-03-19.

[4] *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.

[5] https://www.baeldung.com/java-in-2018

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



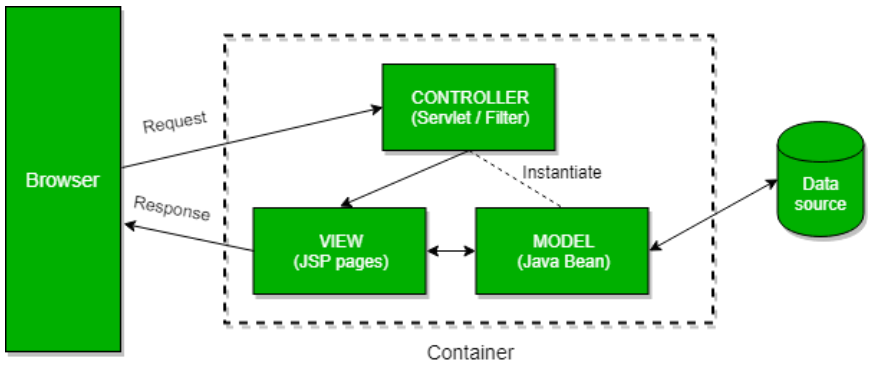
https://microservices.io/patterns/microservices.html

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-practicies, 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[7] 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 dokumentací 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 fází 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[8], protože změna frameworku většinou znamená zásadní úpravu kódu aplikace.

[7] https://whatis.techtarget.com/definition/domain-specific-language-DSL
[8] https://www.techopedia.com/definition/26802/vendor-lock-in

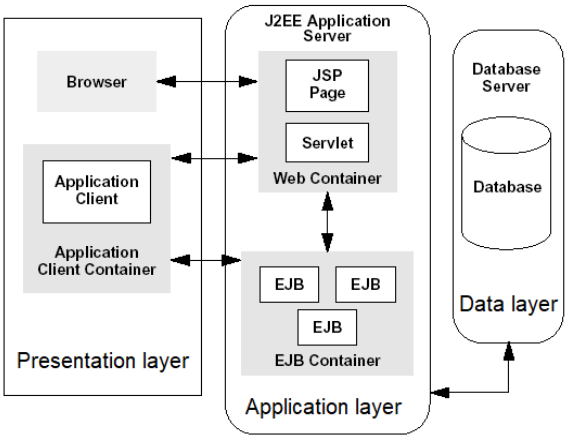
Webový framework jak je z názvu patrné je určen pro ulehčení tvorby webových aplikací na straně serveru. První webové frameworky se začali objevovat kolem roku 1995. Pro Javu se populární webové frameworky, které jsou využívány i v současné době začali objevovat až po novém miléniu. Webové frameworky 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-practicies. 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 busines 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 mez 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 frameworku 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 knihovnu, která stojí za pozornost je Kotlinx.html. Je vyvíjena přímo vývojáři Kotlinu. Oproti přechozím variantám má velkou výhodu a to tím, že umožnuje psát šablony ve stylu a syntaxi Kotlinu, cože 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[9]. 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.

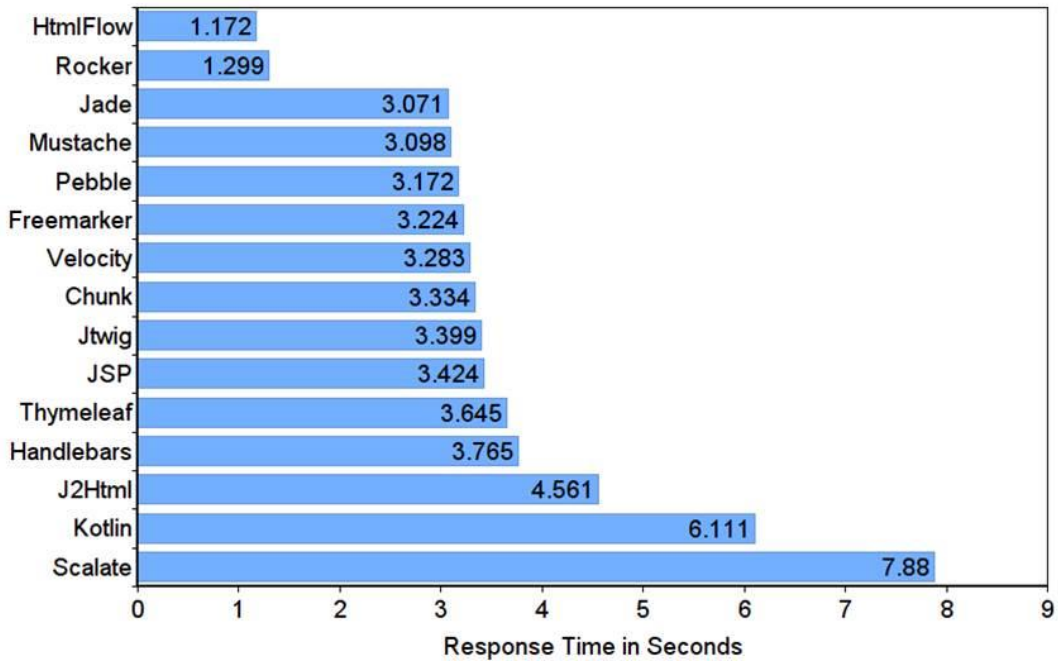
```
fun studentTemplate(student: Student): String {
  return createHTMLDocument()
    .html {
        body {
            ul {
                li { student.name }
                li { student.number }
            }
        }
    }.serialize(false)
}

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[10],

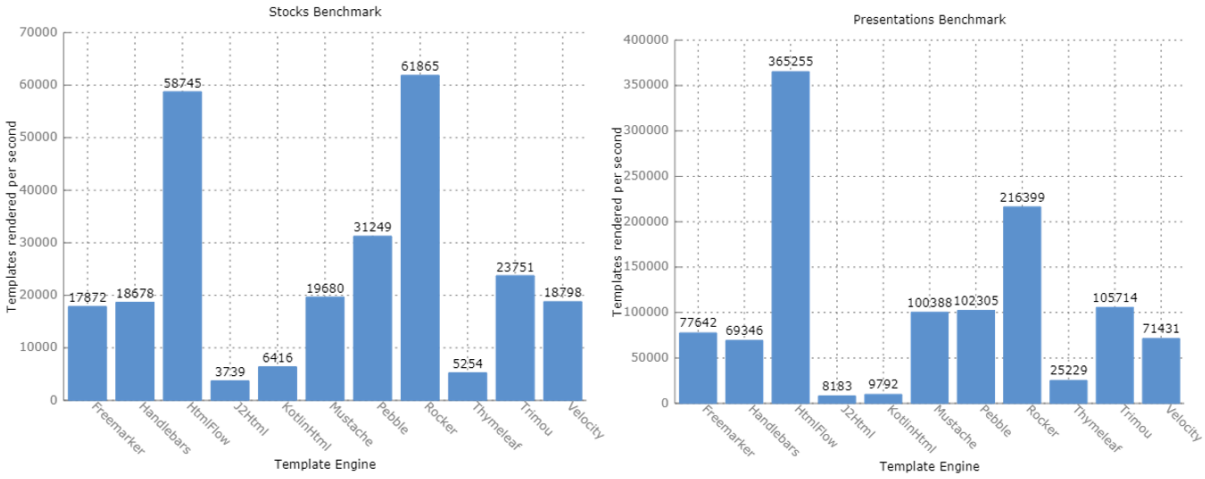
[9] https://dzone.com/articles/modern-type-safe-template-engines
[10] https://github.com/xmlet/spring-comparing-template-engines#how-were-the-results-measured

který testuje engin pod náporem 25 paralelních klientů, kteří dohromady odešlou 25 tisíc requestů. Měří se celkový čas, který zabere obsloužení tohoto množství.



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 přechozí 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 rychlostí vracení odpovědí (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 pamět pro uložení cache[11]. 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[12]. 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

[11] https://searchstorage.techtarget.com/definition/cache
[12] https://superuser.com/questions/1173675/how-much-faster-is-memory-ram-compared-to-ssd-for-random-access

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 systém, který hlídá integritu dat. Všeobecně se používají tři politiky pro řízení integrity dat. První je přístup write-trough, 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-trough. 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 systém, 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 funkcionalita 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[13], 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

[13] https://www.techopedia.com/definition/24200/object-relational-mapping--orm

komunikací 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 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 knohvna 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, OAtuh2. 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 je Spark jedná se o minimalistický framework, který je oblíbený mezi vývojáři. Druhý 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 framewrok splňuje, avšak se nejedná o nejlepší možné řešení, spíš o průměrné naplnění. Nejvyšší stupeň 3 je udělen, pokud framework výrazně převyšuje standard v hodnocené oblasti. Pro hodnocení frameworku byla využita metoda Fullerova trojúhelníku.

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. Zvlášť 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 funkcionkcemi. 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 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 zaintegrovat 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 niž 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. Nadstandardně jsou hodnoceny proprietární rozhraní a implementace poskytované frameworkem k integračním technologiím, jež usnadňují jejich využití v rámci kódu aplikace. Hodnoceno je také 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 uložišť 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 testovaní. 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 výkonnostních testů. Tyto testy jsou vyhodnoceny a dle průměrně dosaženého pořadí jsou rozděleny body.

Vert.x

Historie

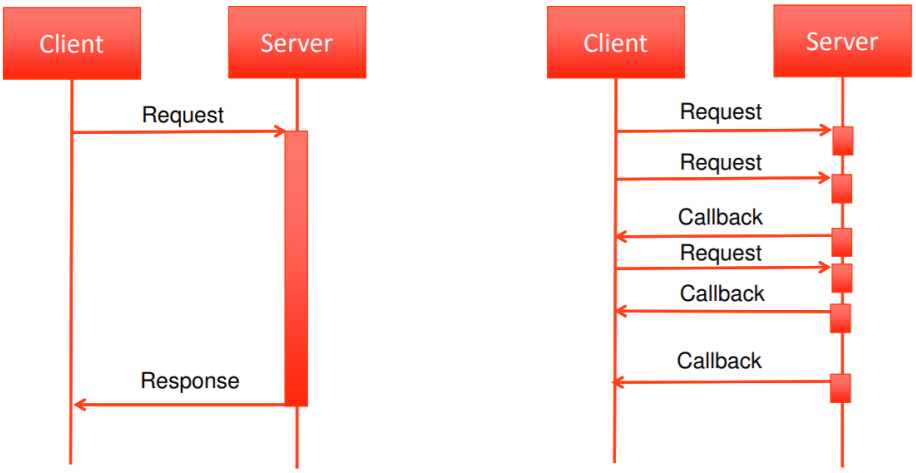
První stabilní verze framewroku 1.0 se objevila již v květnu 2011. Hlavním vývojářem je Tim Fox, který framework vyvíjel během svého působení jako zaměstnanec ve firmě WMware. Jméno je odvozeno od počátečního jména projektu, které bylo Node.x. X v názvu má vyjadřovat jeho nativní podporu mnoha programovacích 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 nejinovativnější technologii pro Javu v rámci JAX Innovation awards[14]. 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

[14] https://jaxenter.com/jax-innovation-awards-2014-champions-declared-107796.html

(Event loop) zachytávající události (HTTP requesty), které jsou zpracovávány asynchronně a vývojář je odstíněn od úskalí spojených s více vláknovým programováním, které by bylo nutné pro dosažení stejného reaktivního chování.

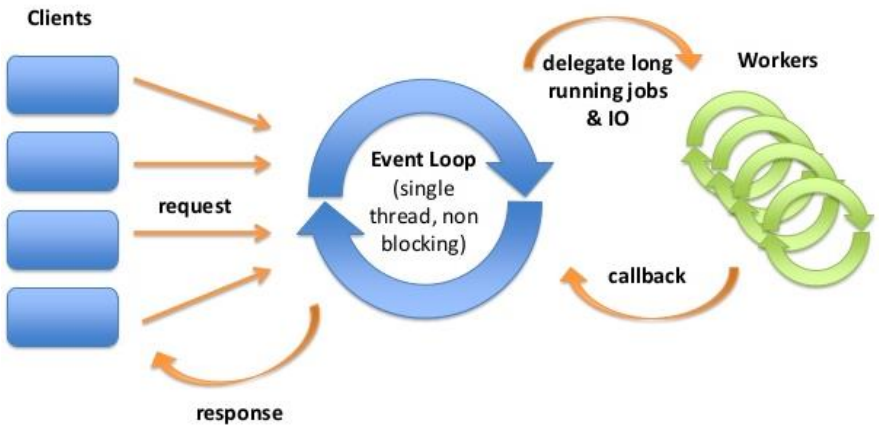


Obrázek 14 Sychronní a aynchronní (reaktivní) zpracování HTTP reqestů

Vert.x využívá pro nízko úrovňové 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[15]. 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 artitekturu 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 vlákna a vývojáře na tuto skutečnost upozorňuje. Vert.x řeší tzv. C10k problem[16], 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

[15] http://www.dre.vanderbilt.edu/~schmidt/PDF/reactor-siemens.pdf
[16] http://www.kegel.com/c10k.html

asynchronnímu zpracování jsou odpovědi zasílány konstantně v čase T od zaslání z klienta na 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[17], který je výzvou pro následující dekádu.

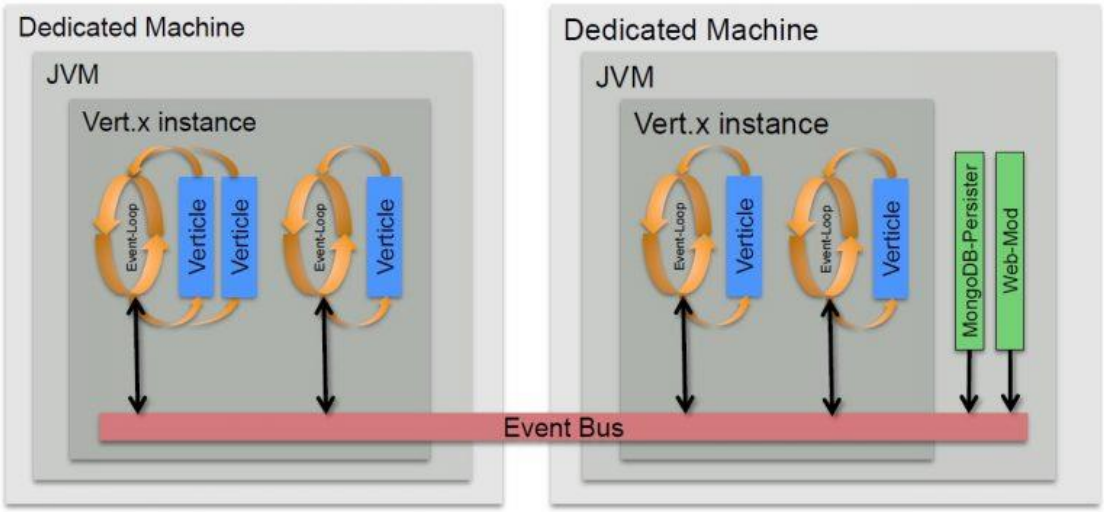


Obrázek 15 Architektura Reactor
https://www.celum.com/de/blog/technologie/the-reactor-pattern-and-non-blocking-io

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,

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

tohoto je ve Vert.x docíleno pomocí globalní cache s názvem Shared Map pro kterou Vert.x poskytuje metody a díku tomu ji lze využít kdekoliv v kódu aplikace.



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[18], 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

[18] https://kotlinlang.org/docs/reference/coroutines-overview.html

CorutinesVerticle napsána tak že využívá korutiny místo normálních metod. Díky tomuto lze psát kód, který je plně ve stylu jazyka Kotlin.

Hodnocení (0-3): 2

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[19]. Vert.x poskytuje podporu pro globální error handling. Podporuje nahrávání souborů, kdy vývojáře odstiňuje od řešení zpracování HTTP multipart[20] těla zprávy a poskytuje rovnou list se soubory.

Vert.x podporuje práci s cookies a sessons. Pro oboje nabízí jednoduché programové rozhraní, které je dostupné z routovacího kontextu aplikace, který je předáván do handlerů. Globální

[19] https://whatis.techtarget.com/definition/MIME-Multi-Purpose-Internet-Mail-Extensions
[20] https://www.w3.org/Protocols/rfc1341/7_2_Multipart.html

SessionHandler je registrován před všemi dalšími route hnadlery, 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 předávanán do handleru. Samotná logika zpracování překladu už je plně v režii vývojáře.

GraphQL

OpenApi 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 implmentaci 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[21]. 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.

Templating

[21] https://www.zdrojak.cz/clanky/co-je-cross-site-request-forgery-a-jak-se-branit/

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.

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 chache, kterou framework poskytuje je pro statické soubory, které lze cachovat automaticky. U této cahce je možné provádět konfiguraci přímo v kódu aplikace. Avšak podporuje klienta pro enterprise in-memory storage jako je Hazelcast, Apache Ignite, Apache Zookeeper, Infinispan.

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.

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.

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.

Přístup k datům

Framework obsahuje klienty pro integraci databázovými systémy. Podporovány jsou tyto PostgreSQL, MySQL, MongoDB, Redis, Cassandra. Pro PostgresSQL 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 systém, 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.

AOP

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

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.

Testovatelnost

Testovatelnost

Podpora a komunita

Pro

Praxe

Toto

Ktor

Spark

Micronaut

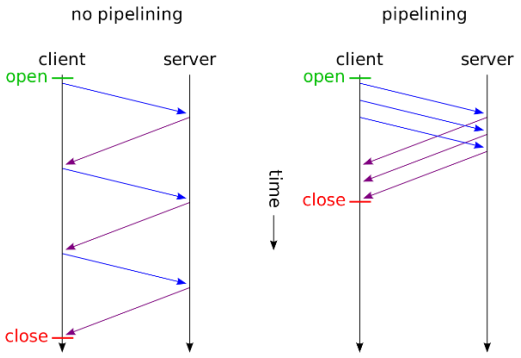
TODO popis a charakteristika frameworků

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 vetší 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

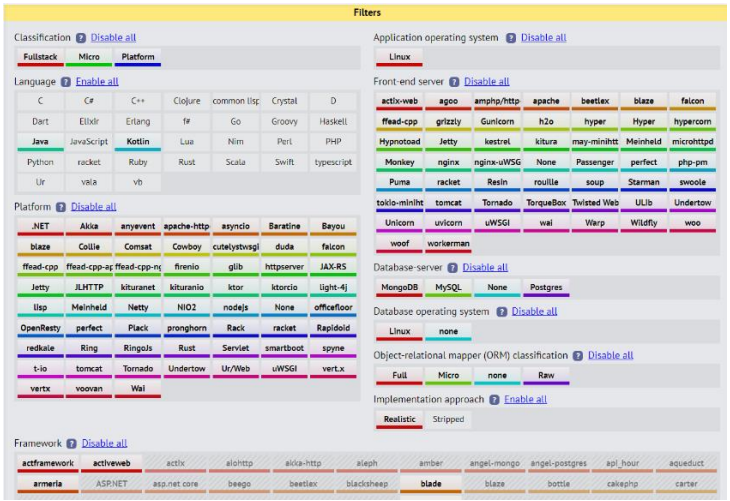
a v průměru vyjdou výsledky 2-3 kol za rok. Společnost stále rozšiřuje množství testovaných frameworků a typy testů, dle stránek mají v budoucnosti přibýt testy, při kterých se využívají features jednotlivých frameworků např. cachování, složitější algoritmické testy. Parametry testování jsou nastaveny 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 mechanizmu. Prvních 5 vteřin probíhá test s 8 paralelními klienty, pro ověření, zda vše běží korektně, výsledky nejsou zaznamenávány. Poté se spustí 15-ti vteřinové rozehřátí 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 zajištuje 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[22]. 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[23], dále je stroj osazen blíže nespecifikovanou operační paměti o velikosti 32 GB a SSD disky. Cloud je hostován na Microsoft Azure D3v2 instancích. Ve FAQ jsou vysvětleny rozdílné výsledky testů mezi těmito 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 pipelinigu a s ním se zahltí až při 550 tiscích requestech. Toto omezení u cloudu není a jediným omezením je rychlost procesoru, tudíž pro interpretaci výsledků, budu využívat měření provedená v cloudu.

[22] https://github.com/wg/wrk
[23] https://www.cpubenchmark.net/cpu.php?cpu=Intel+Xeon+Gold+5120+%40+2.20GHz&id=3154



Obrázek 17 HTTP pipelining[24]

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.



Obrázek 18 Ukázka filtrování

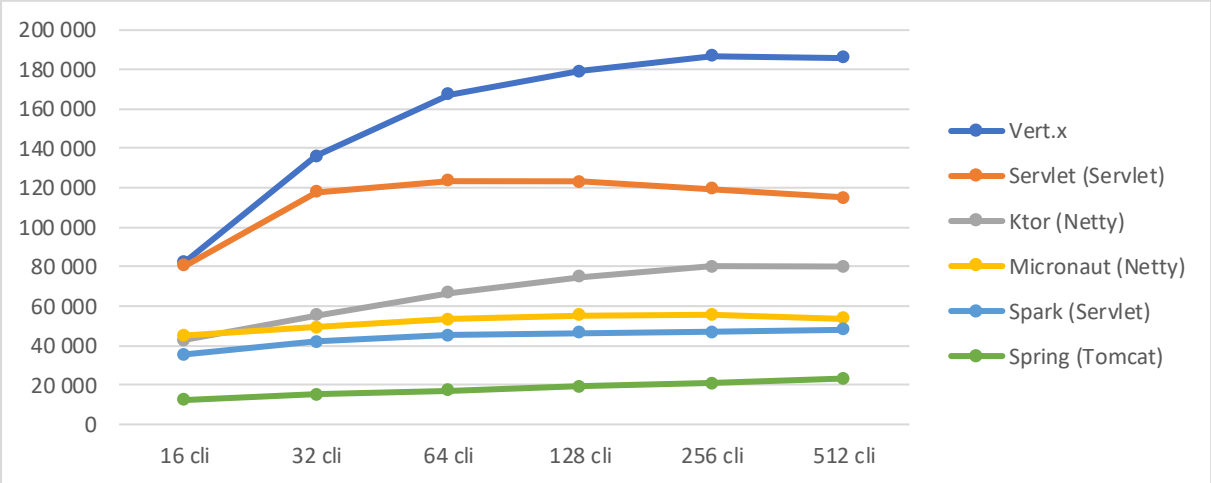
První test je JSON serializace, kdy je zaslán HTTP GET request na server a ten následně zašle reposnse s tělem v podobě plain JSON ve tvaru {"message":"Hello, World!"}, každé tělo response je nově vytvořená instance. Z výsledků nejlépe dopadl Vert.x, který dokázal udržet stoupající tendenci paralelních klientů a počet opovědí se zvedal i při jejich nárustu, 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

[24] https://en.wikipedia.org/wiki/HTTP_pipelining#/media/File:HTTP_pipelining2.svg

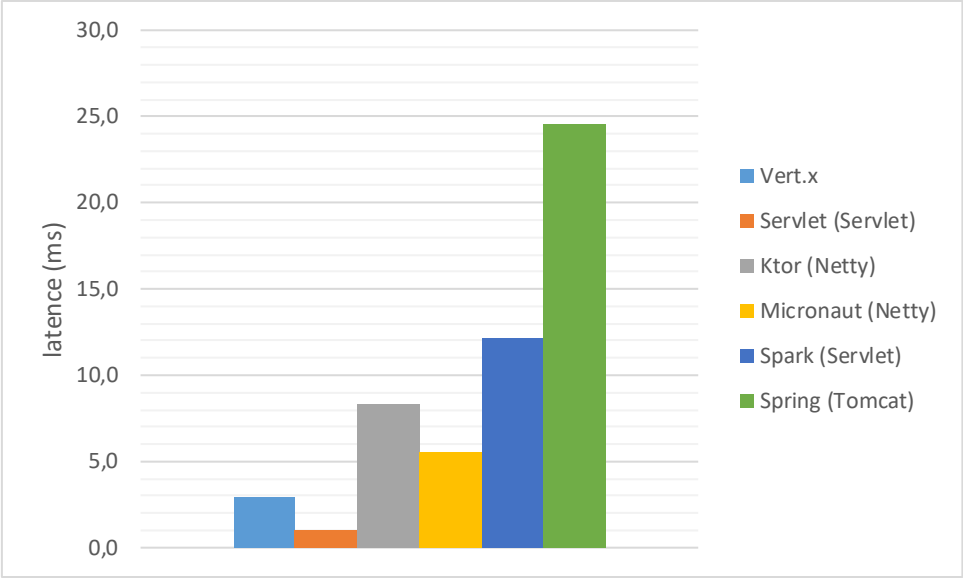
pohybovali zbylé frameworky, které dokázali odpovědět na značně menší počet reqestů. Avšak samotný čistý framework Netty nad kterým jsou stavěné frameworky dokázal zpracovat kolem 195 tisíc requestů a stal by se tak pomyslným vítězem tohoto testu. Nejhůře dopadl Spring, který je v tabulce uveden jako 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	Responses/sec.	Avg. Latence (ms)	16 cli	32 cli	64 cli	128 cli	256 cli	512 cli
Vert.x	186 725	2,9	82 401	136 072	167 530	179 040	186 725	186 105
Servlet	123 367	1,0	80 350	117 747	123 367	123 136	119 316	114 894
Ktor (Netty)	80 087	8,3	42 903	55 373	66 540	74 926	79 970	80 087
Micronaut (Netty)	55 614	5,5	45 103	49 306	53 383	55 336	55 614	53 787
Spark (Servlet)	47 970	12,1	35 518	41 938	45 117	46 517	46 976	47 970
Spring (Tomcat)	23 268	24,6	12 364	15 014	17 169	19 176	21 000	23 268

Tabulka 1 Test JSON serializace



Obrázek 19 Graf HTTP responses



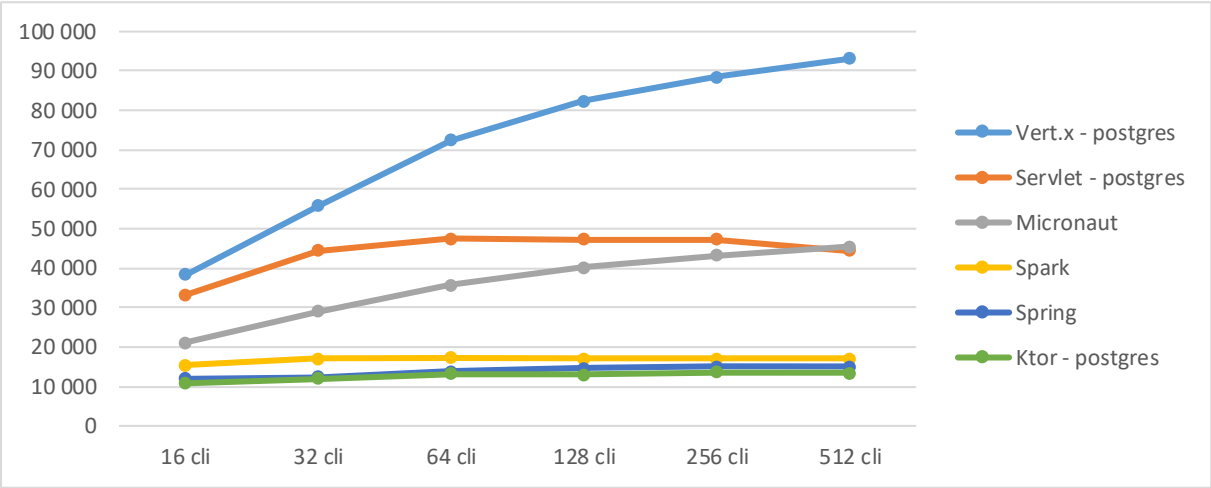
Obrázek 20 Průměrná doba odpovědi

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[25], lze snížit latenci až na 4ms a celkově se 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[26], 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.

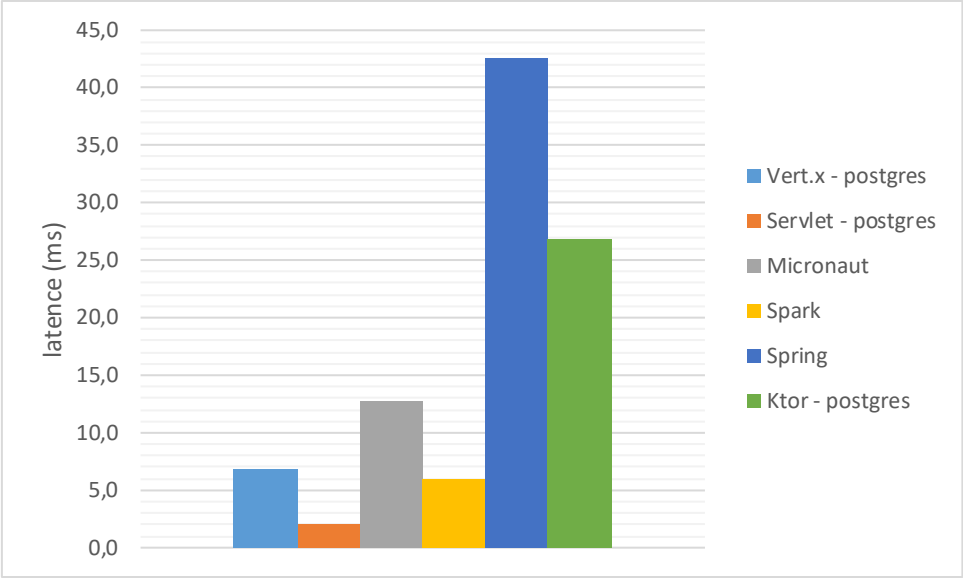
[25] http://www.julienviet.com/reactive-pg-client/
[26] https://github.com/davidmoten/rxjava2-jdbc

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

Tabulka 2 Test Signle query



Obrázek 21 Graf HTTP responses



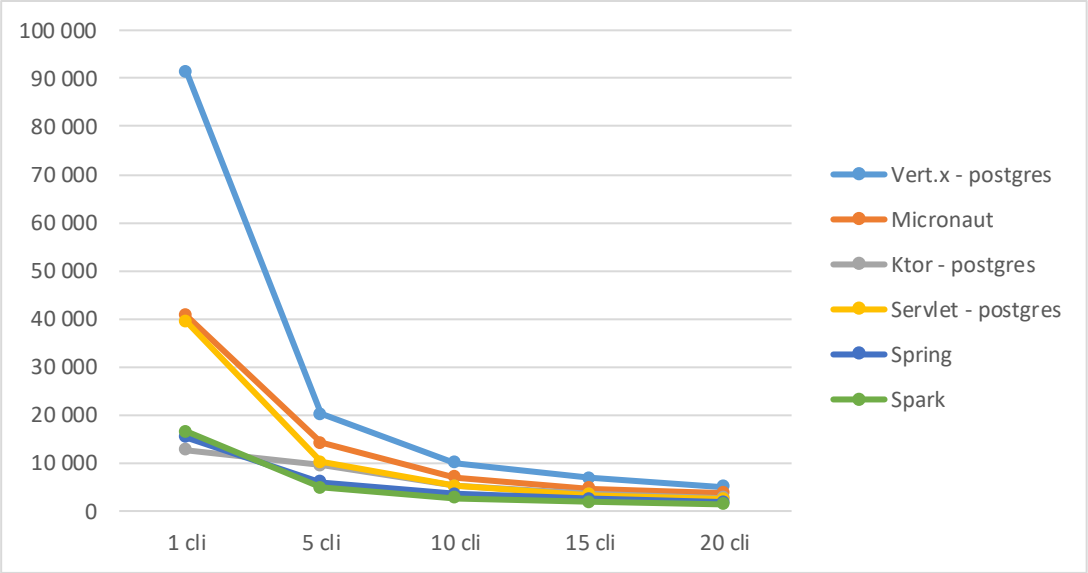
Obrázek 22 Průměrná doba odpovědi

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 vraceny 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 náskokem 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 nárůst až na 3 858 reponse 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. Nejhůře dopadl ve srovnání Spark, avšak napříč všemi platformami poskytl dobrý výkon, pro srovnání Node.js při dvaceti

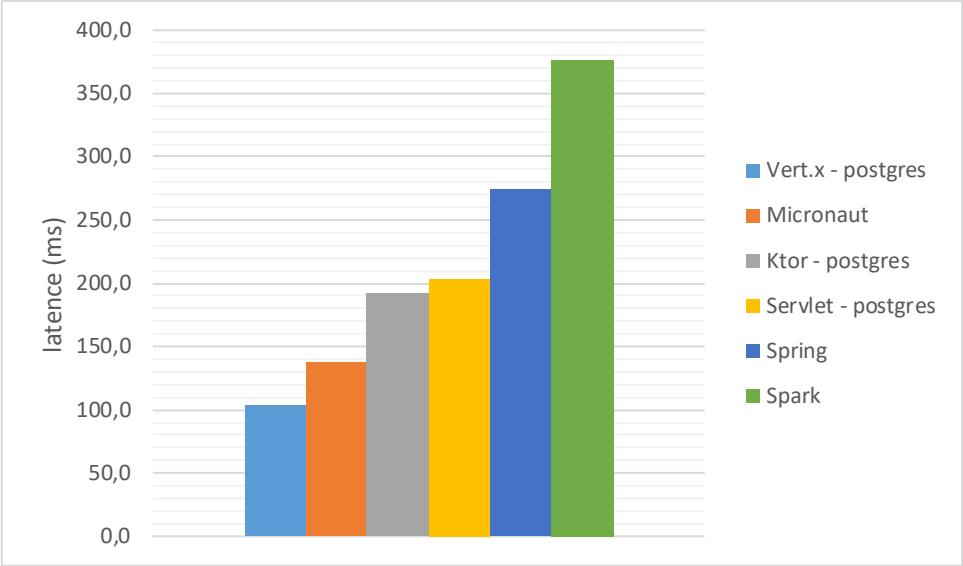
paralelních spojení zvládl odeslat pouze 865 response, což je pouze polovina, kterou zvládl Spark, navíc při průměrné latenci 581 ms.

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



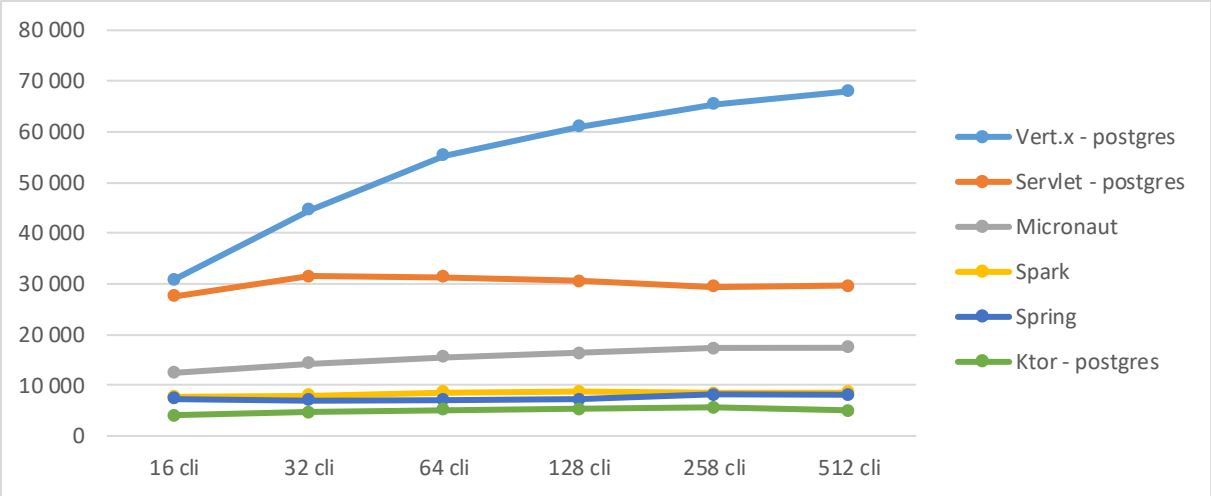
Obrázek 5 Průměrná doba odpovědi

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 odpovědí nadprůměrný výsledek. Zbylé frameworky zaslali méně něž 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

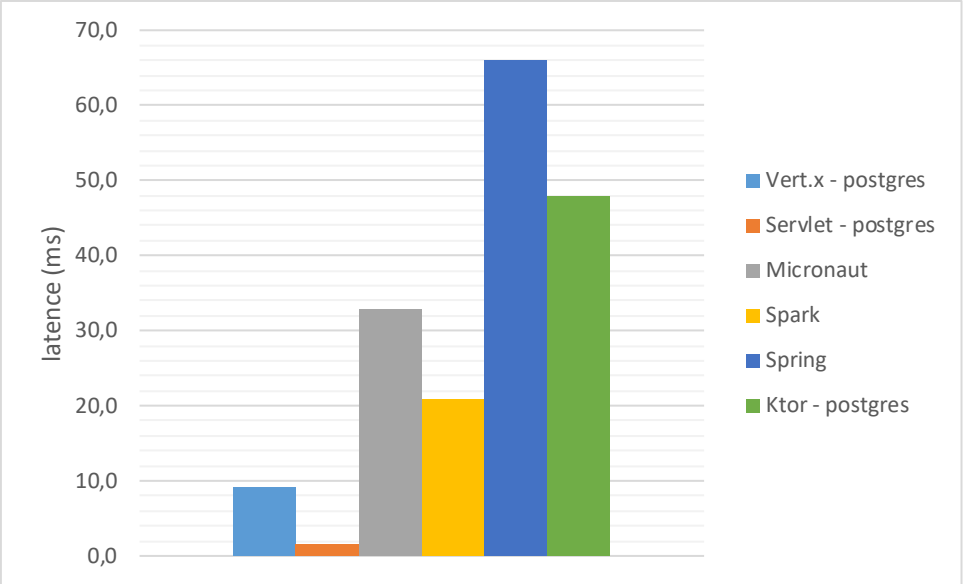
obsluhu requestů WebFlux, lze značně snížit latenci z 66 ms na pouhé 4 ms, čímž by Spring dokázal předčit i vítěze tohoto testu, avšak reaktivní verze nezvýšila počet response. Pro srovnání Node.js zvládl odeslat necelých 10 tisíc response s průměrnou latencí 24,8 ms, takže by se výsledky zařadil mezi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

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

Tabulka 6 Test Fortune



Obrázek 7 Graf HTTP responses



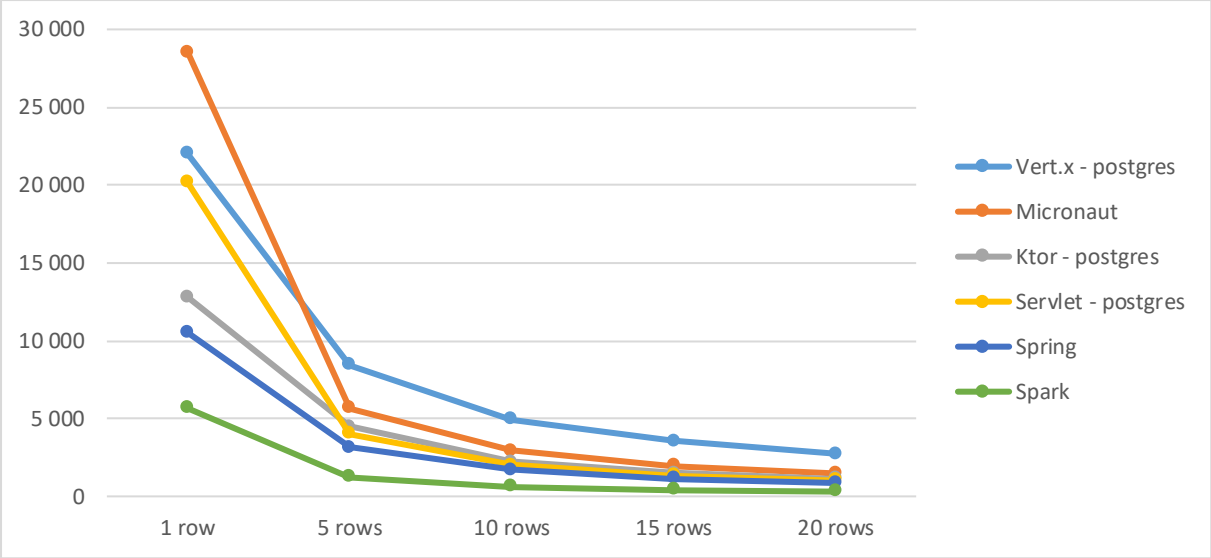
Obrázek 8 Průměrná doba odpovědi

Pátý test ověřuje načítání a aktualizaci řádků databáze. Databáze je použita z druhého testu. Po přijetí requestu aplikace načte dle zaslaného 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.

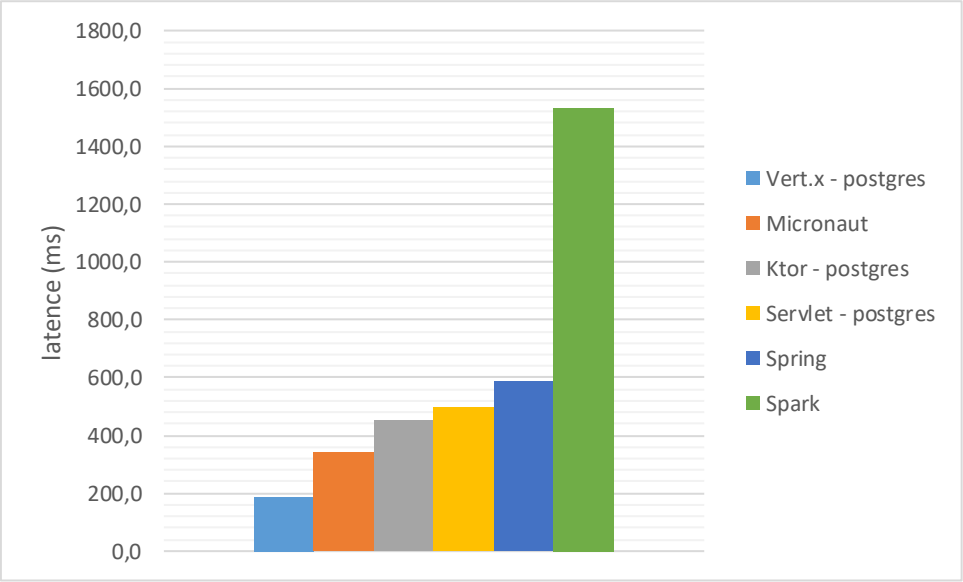
Framework	Responses/sec.	Avg. Latence (ms)	1 row	5 rows	10 rows	15 rows	20 rows

Vert.x - postgres	2 743	185,5	21 987	8 413	4 961	3 536	2 743
Micronaut	1 475	342,7	28 517	5 669	2 920	1 955	1 475
Ktor - postgres	1 113	453,0	12 789	4 453	2 221	1 492	1 113
Servlet - postgres	1 016	495,1	20 145	4 018	2 032	1 357	1 016
Spring	861	583,2	10 511	3 159	1 694	1 154	861
Spark	306	1530,0	5 653	1 230	613	412	306

Tabulka 9 Test Update



Obrázek 23 Graf HTTP responses

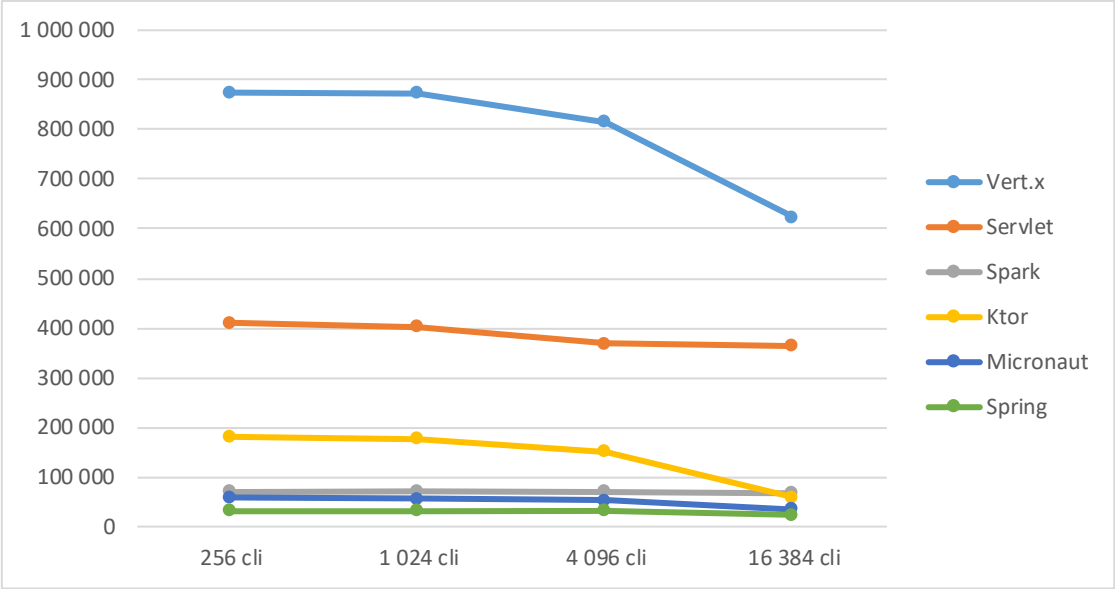


Obrázek 24 Průměrná doba odpovědi

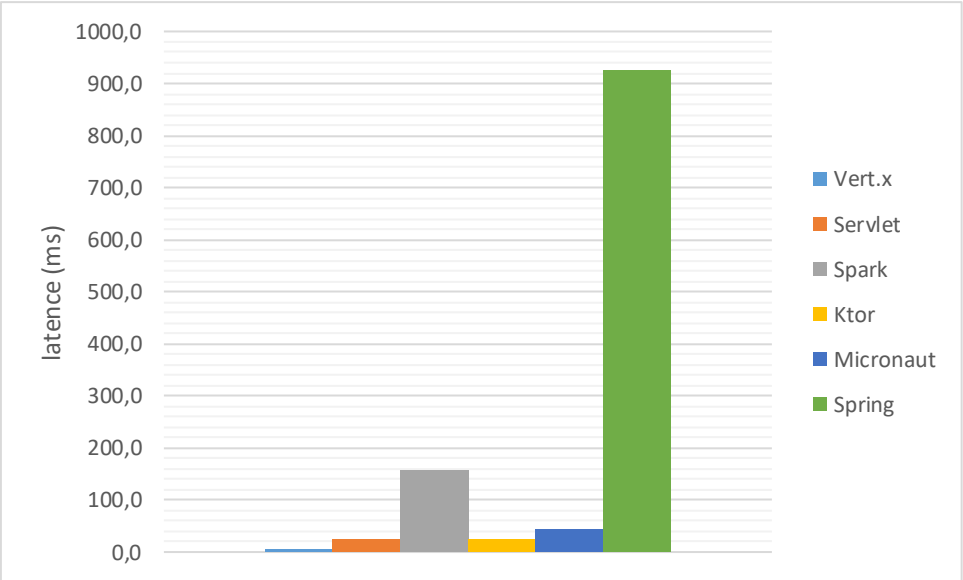
Poslední test je oproti předchozím poměrně triviální, aplikace po přijetí HTTP requestu odpoví requestem obsahující pouze plain text „Hello, World!“. Díky malé velikosti response, není Ethernet limitující faktor a nedojde k saturaci linky, proto může být značně zvětšen počet paralelních spojení oproti 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	Responses/sec.	Avg. Latence (ms)	256 cli	1 024 cli	4 096 cli	16 384 cli
Vert.x	623 904	5,7	874 433	872 674	815 194	623 904
Servlet	363 714	23,2	410 661	401 967	367 873	363 714
Spark	67 051	156,1	70 972	71 032	70 276	67 051
Ktor	59 219	22,9	180 880	176 730	150 648	59 219
Micronaut	34 292	41,3	58 686	55 737	51 913	34 292
Spring	23 725	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[27] 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 výkonostní testy poskytují pouze zlomek z mozaiky informací o použitelnosti frameworku. Spíše dotvářejí představu o jeho chování na testovacích případech,

[27] https://github.com/TechEmpower/FrameworkBenchmarks/issues/133

které 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 chovaní a výkon.

Vyhodnocení

Závěr