Univerzita Hradec Králové
Fakulta informatiky a managementu
Katedra informatiky a kvantitativních metod

Reaktivní programování

Bakalářská práce

Autor: Kryštof Čížek
Studijní obor: Aplikovaná informatika

Vedoucí práce: doc. Mgr. Tomáš Kozel, Ph.D.

Hradec Králové Duben 2019

Prohlášení

Prohlašuji, že jsem bakalářskou práci zpracoval samostatně a s použitím uvedené literatury.

V Hradci Králové dne 30.4.2019 Kryštof Čížek

Poděkování

Děkuji svému vedoucímu doc. Mgr. Tomáši Kozlovi, Ph.D. za veškerou pomoc, rady a odborné vedení při psaní bakalářské práce. Dále bych rád poděkoval firmě Česká pojišťovna a.s., obzvláště pak Ing. Pavlu Krbálkovi, Ph.D. a Ing. Jakubu Příkazskému za vstřícný přístup a poskytnutí odborných konzultací.

Anotace

Cílem práce je představit koncept reaktivního programování, převážně pak se zaměřením na vývoj webových aplikací. Nejprve jsou popsány teoretické principy reaktivního přístupu s pohledem na reaktivní programování převážně jako na nástroj pro tvorbu reaktivních systémů. Následně jsou představeny některé reaktivní knihovny a jejich obecná specifikace, a to se zaměřením na programovací jazyk Java. V této technologii je v další části navržena a implementována ukázková aplikace, vytvořená s cílem ověřit a otestovat teoreticky představené principy reaktivního programování. V poslední části je pak tato aplikace použita pro přímé kvantitativní porovnání klasického a reaktivního přístupu při tvorbě webových aplikací.

Annotation

Title: Reactive Programming

The goal of this bachelor thesis is to introduce core concepts of reactive programming with an emphasis on development of web applications. First, theoretical principles of reactive approach are described, especially as a way of constructing reactive systems. Then, some of the reactive frameworks are presented, including their common specification, while focusing on Java programming language. Using this language, a testing demo application is designed and implemented in the following thesis module. The purpose of this application is to test and verify reactive principles theoretically introduced in the previous chapters. In the last part of the thesis, this application is also used for direct comparison of standard and reactive approach in the development of web applications.

Obsah

1 Úvod

Termín *reaktivní* není ve světě informačních technologií žádnou novinkou, ale je tomu teprve nedávno, co zvyšující se nároky na moderní webové aplikace poukázaly na vhodnost reaktivního přístupu při jejich návrhu.

V roce 2014 na Gartner summitu (1) v Sydney byla klasická třívrstvá architektura aplikací označena za zastaralou a nedostačující a začalo se tak hromadně hovořit o vývojové standardizaci modernější a flexibilnější architektury. Ve stejném roce vyšel vývojáři dlouho očekávaný Reaktivní manifest od Jonase Bonéra a jeho týmu. Tento manifest mimo jiné přesně definuje principy reaktivní architektury pro návrh tzv. *reaktivních systémů*, které se zaměřují právě na splnění stále se zvyšujících požadavků na moderní webové aplikace.

Tak započala jakási hromadná reaktivizace a v současné době by bylo velmi těžké a snad i nemožné najít moderní programovací jazyk, pro který neexistuje framework podporující reaktivní programování. Některé jazyky dokonce zakomponovali podporu pro reaktivní programování přímo do svého jádra – např. Java přidává implementaci reaktivních streamů v rámci JDK9 dle návrhu JEP-266 (2).

V současné době spolu reaktivní a klasické přístupy bez problému koexistují. Pouze čas ukáže, jestli reaktivní architektura absolutně nahradí klasické architektonické přístupy. Jak ale tato práce poukazuje, minimálně v některých oblastech vývoje to bude nutné a nevyhnutelné.

2 Cíl práce

Cílem této práce je identifikovat klíčové vlastnosti a výhody reaktivního přístupu, převážně pak reaktivního programování, a porovnat jej s přístupem klasickým, imperativním.

V první části práce budou definovány teoretické reaktivní principy. Budou detailně objasněny pojmy jako *reaktivní programování*, *reaktivní systémy* a *reaktivní architektura*. Budou vymezeny výhody, příčiny vzniku a vůbec důvody zavádění a používání těchto principů na úkor principů dosavadních.

Dále v druhé části budou popsány některé hlavní a nejpoužívanější implementace jednotlivých reaktivních principů teoreticky popsaných v části první. Kladen bude důraz na obecnou implementaci používanou napříč všemi nejmodernějšími frameworky a dále na implementaci použitou v následujícím praktickém srovnání.

Ve třetí části bude představena praktická ukázka aplikace napsané za použití principů reaktivního programování. Na této aplikaci budou pak v poslední, čtvrté části otestovány prezentované výhody reaktivního přístupu, a to převážně kvantitativně v přímém srovnání s přístupem klasickým.

3 Teorie reaktivních principů

Tato kapitola se dopodrobna věnuje vymezení teoretických principů stojících za reaktivním přístupem s důrazem na definici pojmů s ním spojených.
V souvislosti s nimi totiž často dochází k nesrovnalostem a mnohokrát se můžeme setkat s články, ale i oficiálními dokumentacemi, které mluví např. o funkcionálně-reaktivním programování, i když se ve skutečnosti jedná o reaktivním programování, nebo naopak mluví o reaktivním programování, ale mají na mysli reaktivní systémy.

3.1 Reaktivní programování

Reaktivní programování je kombinací asynchronního programování a neblokujícího přístupu, ve kterém je zpracování kódu řízeno dostupností nových koherentních dat (Dataflow programming) a událostmi (Event-driven programming) (3).

Základním přístupem v reaktivním programování je pak rozdělení problému na více samostatných článků (komponent) řetězce zpracování, přičemž každý mezičlánek lze spustit asynchronně bez blokování. V tomto řetězci se pak některé komponenty chovají jako zdroje událostí sloužící jako emitory dat. Jiné komponenty tyto události konzumují a „reagují" na ně (transformují data, uloží, přepošlou dále, atp.). Komponenty se mohou chovat současně jako emitory i jako konzumenti, tímto způsobem se různě propojují a vytváří tak již zmíněný řetězec zpracování. Tok dat v řetězci zpracování se pak nazývá *stream*.

Důležitou vlastností reaktivního programování je právě jeho asynchronní povaha. Asynchronní znamená „neexistující nebo neodehrávající se ve stejný čas" (4), což v tomto kontextu znamená, že zpracování události probíhá v blíže neurčeném čase po jejím přijetí.

Díky asynchronnímu přístupu dále přichází v možnost neblokující zpracování, což znamená, že jednotlivá spuštěná vlákna bojující o sdílený zdroj nemusí čekat blokováním (tzv. „uspáním") vlákna, dokud je zdroj zaneprázdněn, ale mohou provádět v mezičase jinou práci.

API pro reaktivní programování jsou typicky buď (3):

1. Callbacková – ke zdroji události jsou přichyceny anonymní callbacky, které jsou volány poté, co událost projde tokem dat
2. Deklarativní – funkcionální přístup založený na skládání standardizovaných funkcí jako např. map, flatMap, filter, apod.

Většina knihoven ale podporuje obě tyto varianty a i jejich interní metody je různým způsobem kombinují.

Reaktivní programování řadíme mezi Dataflow programování (5) a jako takové využívá řadu programovacích abstrakcí založených na tomto paradigmatu (3) (5):

- Future a Promise – představuje asynchronní obal umožňující práci nad objektem, který kvůli asynchronnímu přístupu ještě nemusí být k dispozici
- Stream – teoreticky neomezený tok zpracovávaných dat využívající asynchronní neblokující řetězec zpracování
- Dataflow proměnné – proměnné, které mohou záviset na určitém vstupu, metodě nebo jiné proměnné. Tyto proměnné automaticky aktualizují svojí hodnotu podle změn v závislých entitách.

3.1.1 Výhody reaktivního programování

Vzhledem k asynchronní a neblokující povaze s možností deklarativního přístupu reaktivního programování lze identifikovat následující výhody (3):

- Efektivní využití prostředků vícejádrových procesorů díky neblokujícímu přístupu
- Zvýšená rychlost při zpracování více požadavků zároveň díky snížení počtu synchronizačních bodů (dle Universal Law of Computational Scalability (6))
- Jednoduchý a udržitelný způsob zajištění asynchronních a neblokujících výpočtů a I/O operací bez nutnosti explicitně řízené koordinace mezi aktivními komponentami
- Jednoduchá vrstva abstrakce nad vícevláknovým programováním díky deklarativnímu přístupu

3.1.2 Reaktivní vs. funkcionálně reaktivní programování

Funkcionálně reaktivní programování (FRP) je ale kvůli zavádějícímu názvu často špatně definováno jako pouhé spojení termínů „funkcionální“ + „reaktivní programování“, tedy reaktivní programování definované deklarativně za pomoci skládání funkcí, bez nutnosti použití callbacků. Taková definice je ale nesprávná, jedná se o dvě samostatná a ve své podstatě naprosto odlišná programovací paradigmata.

FRP bylo v roce 1997 přesně definováno Conalem Elliottem v jeho publikaci *Functional Reactive Animation* (7).

V této práci C. Elliot zavádí spolu s principem řízení událostmi také *temporální model*, v rámci kterého definuje speciální hodnoty zvané *behaviors*. Zatímco klasická programovací paradigmata stojí na jednotkových diskrétních operacích jdoucích za sebou, ať už jsou definovány imperativně či deklarativně jako funkce, tak FRP se snaží popsat systém využitím těchto behaviorálních hodnot, kontinuálně se vyvíjejících ve spojitém čase. V klasických přístupech lze kontinuální vývoj zachytit pouze nepřímo skrz stav a mutace tohoto stavu – takový přístup ale sám o sobě implikuje nutnou diskretizaci spojitého kontinua, čemuž se FRP vyhýbá.

Technologie jako např. Reactive Extensions (RxJava, RxJS, Rx.NET…), Project Reactor, Bacon.js, Elm atd. rozhodně nepatří mezi implementace FRP, ale jedná se čistě o implementace principů reaktivního programování.

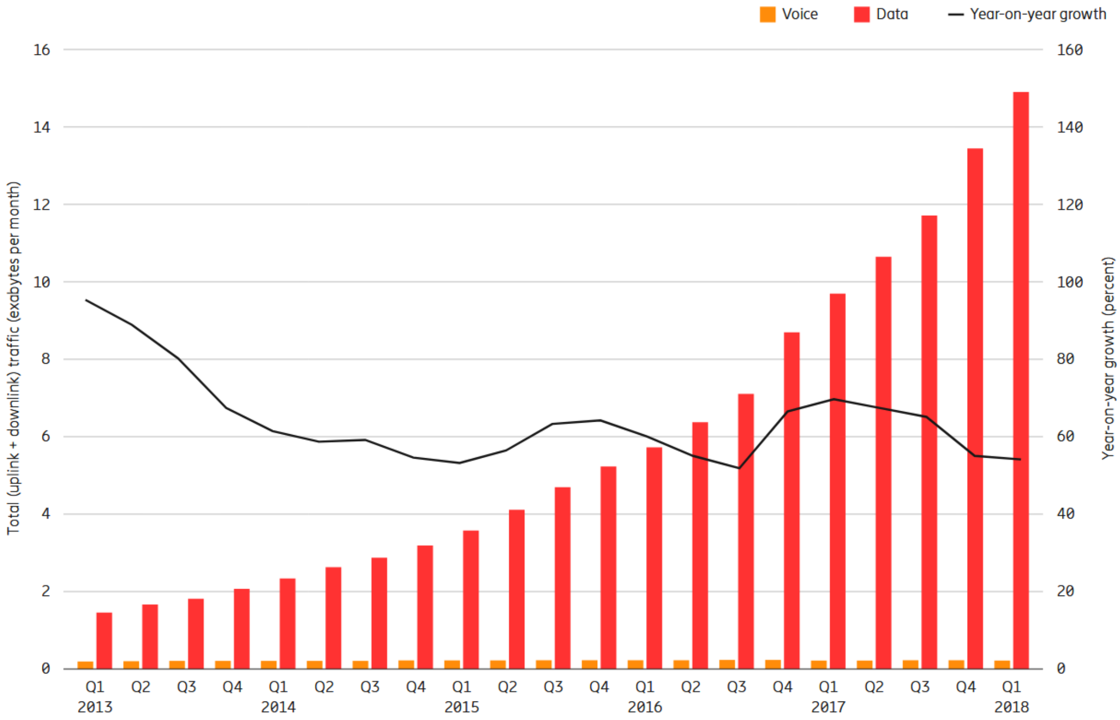
Sám Conal Elliot se ve své přednášce *Essence of FRP* (8) zmiňuje o tomto nesprávném používání definice a oficiálně se tak od výše uvedených a podobných implementací distancuje.

3.2 Programy vs. systémy

Moderní software se stává čím dál tím víc propojenější – kde dříve vznikaly pouze uzavřené lokální programy, dnes vytváříme rozsáhlé vzájemně propojené systémy.

Systémy jsou ze své podstaty velmi komplexní – skládají se z mnoha komponent, přičemž každá komponenta může být dalším systémem. Tedy tím, jak jsou dnešní aplikace stále více závislé na jiných aplikacích, svým propojením vytváří jednu funkční jednotku – systém.

Dále rostou požadavky ze strany uživatelů na webové aplikace. Ti jsou čím dál tím více závislí na dostupnosti a rychlosti služeb, které tyto aplikace poskytují. Například mobilní sektor zaznamenal 54% nárůst přenesených dat v období mezi Q1 2017 a Q1 2018 a nevykazuje známky zpomalení dalšího růstu v dohledné budoucnosti (9).



Obr. 1 Vývoj množství přenášených mobilních dat (9)

V důsledku toho by měly být servery schopny obsloužit miliony současně připojených zařízení bez známek výkyvů stability systému či rychlosti odezvy.

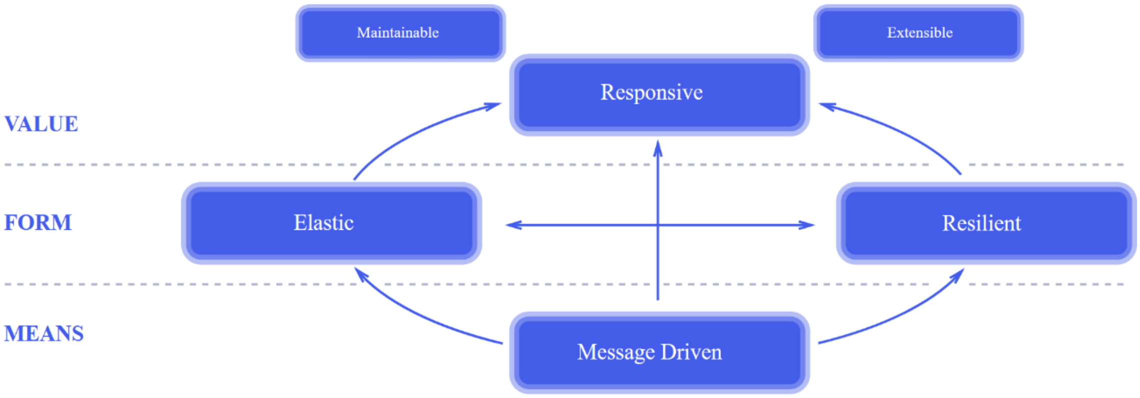
Dalším příkladem může být zvyšující se trend kontejnerizace a virtualizace. Webové aplikace tím pádem čím dál tím více přejímají principy mikroservisní architektury a tvoří tak mnohdy rozsáhlé distribuované systémy.

Právě z těchto důvodu vznikají tzv. *reaktivní systémy*, což jsou architektonické principy návrhu moderních systémů, které jsou schopné naplnit stále se zvyšující požadavky na dnešních moderní aplikace.

3.3 Reaktivní systémy

Principy reaktivních systémů se objevují již od 70-80 let, například v publikacích Fault Tolerance in Tandem Computer Systems (10) nebo Making reliable distributed systems in the presence of software errors (11).
Tito autoři však víceméně předběhli svou dobu a až o několik let později s nástupem vícejádrových procesorů, cloudových řešení a IoT bylo nutné přehodnotit dosavadní postupy při vývoji webových aplikací, potažmo celých systémů.

Moderní definice reaktivních systémů se začíná formovat v roce 2013, kdy Jonas Bonér sestavil tým (jmenovitě Dave Farley, Roland Kuhn, Martin Thompson), který měl za úkol definovat principy vývoje reaktivních systémů a jasně vymezit, co vlastně znamená *reaktivní*. V roce 2014 pak tento tým vydává Reaktivní manifest (12), ve kterém definuje a popisuje klíčové charakteristiky systémů, které pak můžeme nazvat reaktivními.



Obr. 2 Klíčové charakteristiky reaktivních systémů (12)

Základem reaktivních systémů je *responzivnost* – tedy schopnost obsloužit požadavky stejně dobře, nezávisle na místě a čase. To znamená, že nejen že systém musí být schopen dodat správnou odpověď, ale musí ji zároveň dodat kdykoliv je o to požádán, a navíc vždy v rozumném čase. Abychom byli schopni takového systému dosáhnout, musíme zajistit udržení responzivnosti při poruchách a běhových chybách (*odolnost*) a pod zátěží (*pružnost*).

Tyto tři koncepty spolu se systémem *řízení za pomoci zpráv* tvoří jádro reaktivních systémů. Vedlejšími, ale neméně důležitými koncepty jsou *udržitelnost* a *škálovatelnost*.

Vysvětlení těchto principů lze nejsnadněji dosáhnout skrze porovnání s koncepty reaktivního programování.

3.3.1 Reaktivní programování vs. reaktivní systémy

Reaktivní programování je velmi užitečný nástroj používaný při tvorbě samostatných izolovaných aplikací. Pro návrh, následnou konstrukci a začlenění takové aplikace jako komponenty v rámci distribuovaného systému je ale navíc potřeba držet se principů *reaktivní architektury*.

Reaktivní architektura je soubor procesů a zásad pro konstrukci reaktivních systémů a zaměřuje se na dosažení jejich odolnosti a pružnosti.

3.3.2 Řízení pomocí zpráv

Jedním ze základních rozdílů mezi reaktivním programováním a reaktivními systémy je způsob šíření informací mezi komponentami. V komponentách na bázi reaktivního programování dochází ke zpracování požadavků za použití datových toků, které jsou řízeny za pomoci šíření událostí (tzv. Event-driven přístup). Naproti tomu reaktivní systémy se zaměřují na zajištění odolnosti a pružnosti množiny distribuovaných systémů, ve kterých je pak informace šířena za pomoci zasílání zpráv (tzv. Message-driven přístup).

Klíčovým rozdílem mezi systémem řízeným zprávami a modelem řízeným událostmi je adresovatelnost komponent. Zprávy jsou vždy ze své podstaty směrované určitému příjemci, jehož „adresu“ musíme nutně znát. Samotný odesílatel ale nutně adresovatelný být nemusí a může jím tak být i např. anonymní funkce. Zato události žádného cílového adresáta nemají, jsou to pouze pozorovatelná fakta, která tak můžou být pozorována a zachytávána libovolnými anonymními komponentami. Na druhé straně zdroj takovýchto událostí adresovatelný být musí, aby konzumentské komponenty byly schopny se na tento zdroj napojit a vzniklé události zachytávat. Z toho vyplývá, že událostmi řízený systém se zaměřuje na adresovatelné zdroje a zprávami řízený systém se zaměřuje na adresovatelné příjemce (12).

Distribuované reaktivní systémy tvoří množina samostatných komponent komunikujících vzájemně po síti. Kvůli principu předávání informací napříč počítačovou sítí je přirozeně vhodné použít zasílání zpráv. V jednotlivých reaktivních komponentách systému se pak už informace šíří standardně událostmi. Tedy pro zachování principů reaktivního programování a jeho integraci v reaktivních systémech je běžné použít zasílání zpráv jako prostředek propagace událostí po síti mezi mnoha oddělenými aplikacemi. V nejjednodušším případě lze posílat informaci o události přímo jako tělo zprávy. Tím lze docílit zachování jednoduchého programovacího modelu na bázi šíření událostí a zároveň zajistit jeho funkčnost v distribuovaných systémech.

Reaktivní systémy jsou tedy jako celek řízeny jak zprávami, tak událostmi – zprávy slouží ke komunikaci mezi jednotlivými komponentami systému, události pak k řízení chování dílčích komponent.

Dalším ze základních kamenů reaktivních systémů je rozhraní zasílání zpráv (Message Passing Interface – MPI) (13). Toto rozhraní vytváří hranici mezi komponentami a zajišťuje jejich plné oddělení v prostoru (*Transparentnost polohy,* viz Kapitola 3.3.4) a čase (asynchronní zasílání). Toto rozhraní je důležité pro plnou *izolaci* jednotlivých komponent a tvoří tak základ odolného a pružného systému.

3.3.3 Odolnost

Odolnost systému znamená, že je systém responzivní i v případě výskytu softwarové chyby. Systém musí být jako odolný navržen a není to vlastnost, které by šlo jednoduše dosáhnout zpětně. K návrhu odolných systémů lze použít kromě obecných principů a doporučení i příslušné návrhové vzory (*Resiliency patterns*).
Do češtiny se velmi často nesprávně překládá *Systém odolný vůči chybám/poruchám* z anglického *Fault-tolerant systém,* a může tedy docházet k nedorozumění. Princip odolnosti reaktivních systémů sice staví na základech fault-tolerant designu, ale jde až za jeho hranice a říká, že systém by měl být schopen sám se plně zotavit z chyby – tzv. *self-healing* (3). Takovéto zotavení vyžaduje absolutní izolaci vzniklé chyby od

okolí, aby nedocházelo k její propagaci do sousedních komponent, což by vyústilo v sérii kaskádových selhání v systému.

Základním principem návrhu odolných aplikací se schopností samostatného vzpamatování se z chyb je schopnost komponenty vzniklou chybu zadržet, vytvořit novou zprávu dle charakteru chyby a tuto zprávu odeslat komponentám k tomu určeným (tzv. *dozorcům*). Dozorčí komponenty následně mohou výjimku zpracovat a adekvátně na ni reagovat. Tato delegace zpracování chyb na kontext mimo jejich vzniku zajišťuje jejich absolutní izolaci od všech hlavních business komponent, včetně té, ve které vznikly.

Řízení pomocí zpráv je tedy základním předpokladem pro konstrukci odolných systémů, protože umožňuje vyjmutí správy chyb z hlavního řetězce volání a zbavuje tak klienty zodpovědnosti řešit selhání na straně serveru (14).

3.3.4 Pružnost

Pro zajištění neustálé responzivnosti je nutné, aby systém dokázal automaticky správně škálovat propustnost dat a reagoval tak na měnící se požadavky v čase. Škálovat znamená nejen zvyšovat a snižovat obecně propustnost dat dle potřeby, ale také upravovat propustnost toku dat pouze v určité části systému až na úroveň jediné komponenty. Pro zajištění pružnosti se používá princip *back-pressure*.

Back-pressure (také *flow-control*, *backpressure*, princip řízení toku) je mechanizmus komponenty pro vyrovnání se s nadměrnou zátěží, která by jinak mohla způsobit výpadek dané komponenty a potažmo kolaps celého systému (12). Vychází z předpokladu, že v rámci datového toku se mohou nacházet více či méně úzká hrdla či pomalejší úseky způsobené např. složitějším výpočtem v určité části řetězce zpracování. Princip řízení toku pak dovoluje těmto pomalejším komponentám poskytovat zpětnou vazbu komponentám nadřazeným a notifikovat je o svém zahlcení. Tyto nadřazené komponenty nacházející se proti proudu toku dat následně mohou regulovat svoji propustnost, tím vyrovnat zátěž a předejít tak kolapsu. Tento princip je schopen kaskádově se propagovat proti proudu datového toku až na počátek řetězce volání – ke klientovi.

Pružnost je zároveň klíčovým elementem k efektivnímu využití dnes preferovanému cloud computing a SaaS (Software as a Service, software jako služba) modelu aplikací - pomáhá systému k dosažení efektivnějšímu využití hardwarových komponent a tím snižuje cenu provozu aplikačních serverů.

Pro dosažení pružnosti systému je nadále potřeba, aby systém byl adaptivní, tedy aby se dokázal sám bez vnějšího zásahu automaticky škálovat, dynamicky měnit a vystavovat svoji topologii, vyvažovat zátěž, aktualizovat a zálohovat data – a to vše bez statické rekonfigurace systému nebo jeho komponent.

Prostředkem k dosažení takového chování je princip *transparentnosti polohy*. Tento princip spočívá převážně v možnosti škálovat systém vždy stejným způsobem a za použití stejných programovacích abstrakcí. V praxi to znamená, že pokud provádíme distribuovaný výpočet, měli bychom být odstíněni od toho, jestli je výpočet distribuován lokálně mezi více procesory, nebo po síti mezi více servery, případně dalšími komponentami (12). Přínosem tohoto odstínění je možnost škálovat systém přidáváním lokálních či vzdálených komponent bez rozdílu v jejich komunikaci se zbytkem systému. Jediný způsob, jak tohoto odstínění dosáhnout, je znovu skrze zasílání zpráv. Takováto mobilita komponent navíc umožňuje dynamické vystavění topologie systému za běhu a není třeba statické konfigurace.

3.3.5 Produktivita reaktivních systémů

Jelikož většina systémů má vysokou *nezbytnou složitost* (Essential Complexity), je důležité z hlediska návrhu systému zajistit co nejmenší úbytek produktivity při vývoji a údržbě komponent, a zároveň udržovat co nejnižší *náhodnou složitost* (Accidental Complexity).

Nezbytná složitost je v zásadě nevyhnutelná a určená požadavky na chování a fungování aplikace. Zato náhodná složitost je dána až konkrétní implementací požadovaného abstraktního chování a jako takovou ji můžeme ovlivnit (15).

A právě reaktivní systémy jsou nejproduktivnější systémovou architekturou pro vývoj moderních aplikací s důrazem na vícejádrové, cloudové a mobilní řešení, kterou aktuálně známe (3).

Finální shrnutí klíčových vlastností reaktivních systémů dle (3):

- Princip izolace zajišťuje oddělení komponent (tzv *Bulkhead pattern*) a znemožňuje chybám v propagování se do volajících komponent (16)
- Dozorčí komponenty zajišťují oddělené zpracování chyb a umožňují samostatné zotavení komponent (self-healing)
- Transparentnost polohy zajišťující dynamické vystavění topologie systému dovoluje za běhu vyměňovat komponenty, aniž by byla ovlivněna celková dostupnost (responzivita) systému
- Pružnost dovoluje šetřit zdroji sytému při malé zátěži a zároveň jejich efektivní využití při jejím náhlém nárůstu

Díky reaktivní architektuře můžeme tedy tvořit systémy, které se automaticky a snadno vyrovnávají se případnými chybami, fluktuující zátěží a změnami v topologii, a to vše za efektivního využití dostupných prostředků a tím pádem nízkých provozních nákladů.

3.4 Nedostatky reaktivního programování

Díky vysvětlení principů reaktivních systémů lze nyní identifikovat nedostatky reaktivního programování dle (3).

Neadresovatelnost anonymních callbacků nebo funkcí znesnadňuje dosažení odolnosti. Veškeré úspěšné či neúspěšné operace v rámci datového toku jsou zpracovávány přímo a lokálně v anonymních blocích. Tím jsou veškeré chyby v řetězci zpracování vázány na jednotkové požadavky klienta místo toho, aby reprezentovaly celkový stav a zdraví komponenty. Pokud nastane chyba v byť jen jedné části řetězce datového toku, musí být restartován celý řetězec a daná chyba propagována klientovi. Podle principu odolnosti by však měl systém být schopen samostatného vzpamatování se (self-healing) a to bez potřeby vyrozumění klienta.

V čistě reaktivním programování jsou datové toky řízeny událostmi, které sami o sobě nepodporují oddělení v prostoru principem transparentnosti polohy – takového oddělení lze dosáhnout pouze řízením za pomoci zasílání zpráv (Kapitola 3.3.4). Bez dodržování tohoto principu nemůže aplikace dosáhnout plné pružnosti. V důsledku toho je v případě propojení s dalšími službami nutno celkovou topologii definovat staticky, čímž klesá škálovatelnost a zvyšuje se náhodná složitost (Accidental Complexity) výsledného systému.

3.5 Reaktivní programování v kontextu reaktivních systémů

Reaktivní programování je skvělý nástroj pro definici vnitřní logiky a transformačního datového toku v rámci jedné komponenty systému. Zlepšuje čistotu a přehlednost kódu, zvyšuje výkonnost a efektivnost využití prostředků komponenty.

Reaktivní systémy jsou souborem architektonických principů pro distribuovaný výpočet a komunikaci mezi více komponentami s důrazem na odolnost a pružnost systému jako celku.

Z tohoto shrnutí vyplývá, že **reaktivní programování je pouze jednou z implementačních technik v reaktivní architektuře pro stavbu reaktivních systémů zajišťující převážně lokální interní logiku jednotlivých izolovaných komponent**.

V kapitole 3.4. byly zmíněny nedostatky reaktivního programování v kontextu tvorby moderních webových aplikací. Důležité je zdůraznit, že se opravdu jedná pouze o *nedostatky* – ne nevýhody či dokonce chyby. Zatímco pokud dojde v některé části životního cyklu systému k omezení v důsledku nevýhody určitého přístupu, musíme aktivně vynaložit prostředky na odstranění či obejití daného omezení. Oproti tomu při omezení v důsledku určitého nedostatku stačí pouze rozšířit aktuální přístupy o nová příslušná řešení.

V kontextu samostatných izolovaných aplikací může být reaktivní programování plnohodnotným paradigmatem zajišťující pružnost aplikace v čase skrze asynchronní a neblokující přístup. Jakmile ale přejdeme do kontextu dnešních webových aplikací v podobě distribuovaných systémů a mikroservisové architektury, teprve tehdy se začínají objevovat výše zmíněné nedostatky. Proto vzniká koncept reaktivních systémů jako *neinvazivní* rozšíření reaktivního programování a odstraňuje tak jeho nedostatky. Neinvazivní v našem kontextu znamená, že nedochází k porušení či překroucení původních principů reaktivního programování při jejich použití v rámci reaktivních systémů.

Kritickým bodem zakomponování reaktivního programování v rámci reaktivních systémů byl přechod od logiky řízené událostmi (Event-driven programming) k logice řízené zprávami (Message-driven programming) – tedy muselo dojít ke změně celého řídícího paradigmatu. Neinvazivita řešení je zajištěna zapouzdřením vzniklých událostí do zpráv pro jejich distribuci skrz MPI (Message Passing Interface) a v jednotlivých komponentách je pak díky jejich izolovanosti zachována možnost šíření již samostatných událostí.

Při vývoji moderních webových aplikací je tedy vhodné již od počátku stavět na návrhových principech reaktivních systémů pro budoucí škálovatelnost aplikace jako celku. Reaktivní programování je pouze jednou, i když velmi stěžejní, technikou k dosažení vysoce produktivního, odolného a pružného systému.

3.6 Zdroje kapitoly

V celé kapitole 3 bylo kromě explicitně uvedených zdrojů v celém spektru čerpáno z (3), (12) a (17).

4 Implementace reaktivních principů

Tato kapitola popisuje konkrétní implementace reaktivních principů popsaných v kapitole 3. Kladen je důraz především na reaktivní programování, implementační problematika reaktivních systémů je mimo rozsah této práce a je nastíněna pouze okrajově.

V současné době existuje reaktivní framework pro každý z moderních a nejvíce používaných programovacích jazyků, a to převážně díky iniciativě ze strany komunity kolem Reactive Extensions (ReactiveX) (18). V této práci budou implementační specifika popsána pro jazyk Java, který se stabilně umísťuje na první příčce TIOBE Indexu (19), který měří popularitu programovacích jazyků. Kladen bude důraz na framework Spring, který ve své nejnovější 5. verzi přináší podporu reaktivního stacku (20). Za pomoci těchto technologií bude v následující kapitole navržena praktická ukázka testovací aplikace, na které budou empiricky ověřeny výhody reaktivního přístupu popsané v kapitole 3.

4.1 Historie

Zatímco prvky principů reaktivního programování se začaly objevovat již v 70-80 letech, první skutečné implementace vznikly teprve nedávno jako odpověď na zvyšující se požadavky na moderní systémy. Avšak přesné datování vzniku prvních implementací je poněkud obtížné. Přesná definice termínu *reaktivní* vzniká až v roce 2014 soupisem Reaktivního manifestu a ani tehdejší verze reaktivních frameworků zcela nesplňovaly požadavky sepsané v manifestu (jmenovitě především podporu backpressure). Této problematice se podrobněji věnuje kapitola 4.2.

V polovině 90. let začínají vznikat návrhové vzory pro snazší vývoj a lepší udržitelnost stále složitějších objektově orientovaných systémů vykazující prvky reaktivních systémů. Příkladem může být dílo *Decoupling of Object-Oriented Systems: A Collection of Patterns* (21) s prvním vydáním v roce 1996. Zde autor představuje nové návrhové vzory kombinující standardní vzory GOF, snaže se zajistit mimo jiné izolaci subkomponent systému a jejich komunikaci za pomoci zpráv. Spolu s těmito prvky dnešních reaktivních systémů je zde popsáno využití návrhového vzoru *Observer*, což odpovídá Generaci 0 reaktivních API (viz kapitola 4.2.1).

Prvním zlomovým okamžikem byl rok 2009, kdy Erik Meijer z týmu společnosti Microsoft vydává první verzi reaktivního frameworku Reactive Extensions (ReactiveX) (22) pro .NET. Ve stejném roce pak vzniká i projekt Akka (23) pod vedením Jonase Bonéra ze společnosti Lightbend, pozdějšího hlavního tvůrce Reaktivního manifestu.

V roce 2012 pak Microsoft Open Tech uvolňuje Reactive Extensions jako open source (24). Následně na to začíná Ben Christensen ze společnosti Netflix práci na Reactive Extensions verzi pro jazyk Java (RxJava) (25). Do roku 2016 se pak RxJava stává ústřední technologií společnosti Netflix, a tato technologie nyní zajišťuje veškerý datový provoz skrz Netflix API (26).

V roce 2013 vzniká Project Reactor od open-source týmu společnosti Pivotal (26), který je jedním z hlavních tvůrců frameworku Spring (27). Zároveň pokračuje vývoj nástrojů Reactive Extensions pro všechny moderní programovací jazyky (18). Kromě Java a .NET může být dalším příkladem Python (RxPY), JavaScript (RxJS), C++ (RxCpp), PHP (RxPHP), Swift (RxSwift) a další.

Druhým zlomovým okamžikem byl rok 2014. Po vydání Reaktivního manifestu, který sjednocuje teoretický pohled na reaktivní přístupy, vzniká specifikace Reactive Streams (28), která definuje sadu pravidel a nízkoúrovňových rozhraní jako ucelený standard pro implementace reaktivních frameworků. Reaktivním streamům je podrobně věnována kapitola 4.3.

V roce 2017 je pak podpora reaktivního programování zavedena přímo do základních knihoven jazyka Java v rámci vydání Java 9 (2). Ve stejném roce vychází i Spring 5 (29) s ucelenou sadou nástrojů pro tvorbu reaktivních webových aplikací.

Všem výše zmíněným implementacím reaktivního programování v rámci JVM se podrobněji věnuje kapitola 4.4.

4.2 Generace reaktivních knihoven

Implementace principů reaktivního programování a jejich knihovny se jako každá nová technologie velmi rychle vyvíjejí v čase. Prvotní implementace reaktivního programování, jako např. Reactive Extensions pro .NET z roku 2009 se diametrálně liší od dnešní verze knihovny Rx.NET. Kvůli velkým změnám ve struktuře těchto implementací, převážně s příchodem Reaktivního manifestu a Reaktivních streamů, v roce 2016 popsal Dávid Karnarok na svém blogu (30) rozdělení reaktivních knihoven do generací.

Dávid Karnarok je jedním z hlavních vývojářů projektu RxJava, zároveň se podílí i na vývoji na Project Reactor a jeho klasifikace reaktivních knihoven do generací je uznávána a citována na oficiálních stránkách a v dokumentacích jednotlivých implementací (např. (31), (32)).

Generace 0

Do tzv. nulté generace řadíme knihovny využívající pouze klasický návrhový vzor Observer. V Javě by se jednalo např. o knihovny využívající třídu *java.util.Observable* a dále o klasické callback API třídy na bázi listeneru, např. v knihovnách Swing/AWT třídy implementující rozhraní *java.awt.event.ActionListener*.

Obecně sem můžeme zařadit veškeré knihovny a převážně návrhové vzory s prvky reaktivního přístupu používaných před rokem 2009 s nástupem Rx.NET (30).

Největším problémem této generace knihoven je nemožnost skládání událostí, tedy komponent zodpovědných za jejich propagaci. Jednotlivým prvkům chybí společný základ a bylo by nutné vytvořit knihovnu podporující skládání pro každý typ komponenty publisher-subscriber zvlášť. Další možností by bylo pomocí abstrakce vytvořit společný základ, ale následně by bylo znovu nutné vytvořit adaptér samostatně pro každý typ komponenty.

Generace 1

Nedostatky nulté generace byly vyřešeny vydáním Reactive Extensions, tedy první verze Rx.NET, Erikem Meierem v roce 2009. Kromě již zmíněných prvních verzí Rx.NET sem můžeme zařadit např. i první verze RxJava z roku 2012, ale i např. RxJS až do verze 5.

Porty Reactive Extension do dalších jazyků se prvně řídily architekturou Rx.NET, ale v průběhu jejich vývoje se narazilo na nedostatek jejího přístupu. Při použití komponent Observer/Observable v synchronním režimu nelze v této generaci knihoven přerušit či pozastavit probíhající sekvenci proudu dat (32). První verze Rx.NET tento problém obešlo vynucením asynchronicity při použití některých datových zdrojů, muselo ale nutně dojít k pozdější úpravě architektury pro podporu synchronního přerušení.

Druhým problémem byla nepřítomnost podpory backpressure. Velmi snadno tedy docházelo k přehlcení konzumenta daty z vyšší vrstvy, tvoření úzkých hrdel a celkovému zpomalení systému.

Generace 2

Do této generace řadíme knihovnu RxJava 1.x, které se podařilo vyřešit problémy první generace reaktivních knihoven, a to hlavně díky značnému zájmu ze strany společnosti Netflix o tuto technologii.

Kromě vyřešení problému synchronního přerušení byla navíc přidána možnost přidání vlastních operátorů do sekvence (33). Byl představen i nový systém backpressure, který před emitováním dat producentem nejdříve zkontroluje připravenost konzumenta na jejich příjem. Podobný systém byl implementován i pro propagaci událostí.

Generace 3

Kromě určitých optimalizačních problémů neměla RxJava žádné velké adresovatelné nedostatky. Přesto je třetí generace reaktivních knihoven jedna z nejrevolučnějších, protože znamená integraci specifikace Reaktivních Streamů, kterou tvoří 4 rozhraní s 30 pravidly a zajišťuje kompatibilitu mezi reaktivními knihovnami. Díky společným základním rozhraním je vývojář schopen jednoduše měnit implementace reaktivních knihoven 3+ generace a nebo dovoluje odlišným knihovnám komunikovat mezi sebou.

Mezi knihovny třetí generace patří např. Reactor 2, Akka Streams a první verze RxJava 2.

Generace 4

Knihovny Reactor a Akka Streams vcelku bez problémů implementovaly do svého jádra specifikaci Reaktivních streamů, protože se jednalo o celkem nové technologie a jejich tvůrci se na specifikaci Reaktivních streamů již nějakou dobu aktivně podíleli. Naproti tomu RxJava vznikala na základech první verze Rx.NET z roku 2009, jako taková měla zcela odlišnou architekturu a princip implementace a proto musela být RxJava 2 komplementující specifikaci Reaktivních streamů od základů přepsána (30). Během reimplementace si Dávid Karnarok všiml, že by některé operátory mohly být interně či externě sloučeny a mohla by se tak dramaticky snížit režijní náklady.

Čtvrtá generace reaktivních knihoven tedy představuje tzv. fúzi operátorů (operator-fusion). Ta se dále může dělit na makro fúzi, jejíž principem je nahrazení dvou a více navazujících operátorů jedním a mikro fúzi, která spočívá ve sdílení prostředků nebo interní struktury několika operátorů.

Knihovny této generace tedy vypadají navenek svými rozhraními a použitím naprosto stejně, jako generace předchozí, ale v jádru došlo k velkým změnám implementací mnoha operátorů a tím i k velkému snížení režijních nákladů a zvýšení celkové rychlosti knihoven.

Do této generace knihoven spadají novější verze JavaRx 2, a dále Project Reactor 3. Dokumentace Akka Streams sice zmiňuje fúzi operátorů (34), ale jejich přístup je dle dokumentace odlišný a Akka Streams není nativní implementací Reaktivních streamů a jako taková patří do generace 3 (32).

Generace 5+

V současné době řadíme veškeré knihovny maximálně do generace čtvrté a zatím neexistuje žádná pevně daná sada cílů pro generace další. Dávid Karnarok na svém blogu (30) ale např. zmiňuje možnost rozšíření specifikace Reaktivních streamů o podporu I/O operací ve formě oboustranných sekvencí.

Jako další se nabízí zlepšování fúzního principu, kde podle Dávida Karnaroka jsme teprve přibližně v polovině toho, co tento přístup může nabídnout. Podle Karnakových měření (30) má Project Reactor přibližně o 50% vyšší režijní náklady než klasické Java 8 streamy, RxJava má pak režijní náklady vyšší přibližně o 200%. To jsou poměrně příznivé výsledky, s přihlédnutím k tomu, že tyto streamy jsou bržděny asynchronními hranicemi a nutností „konzultovat“ možnost streamování dalších dat s konzumenty (backpressure). Další snížení režijních nákladů a přiblížení se až klasickým streamům a to se zachováním všech výhod reaktivního přístupu by pak představovalo další velký milník této technologie.

4.3 Reaktivní streamy

V roce 2013 se týmy pracující na jednotlivých reaktivních knihovnách snažily vyřešit problém s implementací principu backpressure (35). Expertů na reaktivní problematiku nebylo v té době mnoho, vzájemně se v mnoha případech znali a nemálo z nich přispívalo do více knihoven najednou (např. již zmíněný Dávid Karnarok pracující na projektech RxJava a Reactor). Při společných sezení nad složitou problematikou implementace backpressure a dalších principů se postupně začalo formovat uskupení Reactive Streams. Toto uskupení si kromě řešení implementačních výzev dávalo za úkol vytvořit společné základní rozhraní definující standard pro všechny implementace reaktivních knihoven na bázi streamů a zároveň umožnit komunikaci mezi všemi těmito knihovnami (35).

V roce 2014 Viktor Klang (společnost Lightbend, knihovna Akka) formálně vytváří seskupení Reactive Streams, přičemž mezi ostatní hlavní členy patří např. Ben Christensen (společnost Netflix, knihovna RxJava), Stephane Maldini (společnost Pivotal, knihovna Project Reactor), Roland Kuhn (společnost Lightbend, knihovna Akka) a později se přidávají další vývojáři ze společností Red Hat a Oracle (35).

4.3.1 Cíle

Účelem Reaktivních streamů je poskytnout standard pro zpracování dat za pomoci asynchronních streamů s neblokujícím principem backpressure (28).

Za hlavní cíl Reaktivních streamů můžeme tedy považovat snahu řídit výměnu proudu dat skrz asynchronní hranice. Asynchronní hranice jednoduše představují jiná vlákna – jde tedy o režii předávání elementů jiným vláknům.

Další důležitou vlastností Reaktivních streamů je, že zajišťují zpracování neomezeného proudu dat, ale bez použití neomezeného zásobníku (bufferu) elementů. Vedlejším efektem principu backpressure je právě nutnost implementace omezeného zásobníku. Více o implementaci principu backpressure v kapitole 4.3.6.

Celkem se celá specifikace skládá z 4 rozhraní, 30 pravidel pro implementaci rozhraní a 7 metod (36). Ve zbývajících podkapitolách kapitoly 4.3. budou dle dokumentace (36) tyto rozhraní popsány a vysvětleny.

4.3.2 Publisher

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

Publisher je poskytovatel potenciálně neomezeného množství elementů svým registrovaným konzumentům (*Subscriber*). Tyto elementy poté emituje jako asynchronní proud dat. Publisher je nucen emitovat data pouze na základě připravenosti konzumenta a jeho poptávky po datech – tedy je nucen konformovat principu backpressure.

K tomuto rozhraní se váže 11 doplňujících implementačních pravidel (Publisher Rule ID#1-11).

Subscribe

Jedná se o tovární metodu (factory method), která přijímá *Subscriber* (viz kapitola 4.3.3) a vytváří tak nové *Subscription* (viz kapitola 4.3.4).

Tato jediná metoda tedy registruje konzumenta tomuto emitoru, aby byl schopen daný konzument přijímat emitovaná data, pokud o ně projeví zájem (backpressure). Jednomu emitoru můžeme registrovat neomezený počet konzumentů.

4.3.3 Subscriber

```
public interface Subscriber<T> {
    public void onSubscribe(Subscription s);
    public void onNext(T t);
    public void onError(Throwable t);
    public void onComplete();
}
```

Subscriber je konzumentem dat emitovaných Publishery. Každý Subscriber může být registrován pouze k jednomu Publisheru (pravidlo Subscriber Rule ID#5),

přičemž tento vztah je realizovaný pomocí instance třídy implementující rozhraní *Subscription* (viz kapitola 4.3.4).
K tomuto rozhraní se váže 13 implementačních pravidel (Subscriber Rule ID#1-13).

onSubscribe

Tato metoda je volána vzápětí po registraci konzumenta k určitému emitoru a musí být vždy prvním signálem, který konzument od emitoru obdrží (pravidlo Publisher Rule ID#9).

K registraci konzumenta je potřeba instanci třídy implementující rozhraní *Subscriber* nejdříve zaslat jako parametr metody *subscribe* danému emitoru, který tuto metodu implementuje dle rozhraní *Publisher*. Emitor touto tovární metodou vytvoří instanci třídy implementující rozhraní *Subscription* a obratem tuto instanci zašle zpět konzumentu právě pomocí metody *onSubscribe*.

Ve výsledku je tedy díky této metodě instance *Subscription* k dispozici nejen emitoru, ale také konzumentu, který je tak skrze tuto instanci schopen signalizovat emitoru svou připravenost přijímat data (backpressure).

onNext

Pomocí této metody emituje *Publisher* data všem svým registrovaným konzumentům (*Subscriber*), přičemž jednotlivé datové elementy jsou zasílány v parametru metody.

Tato metoda je volána emitorem pouze pokud konzument o data předem zažádá. Žádost o data je zasílána pomocí metody *request* volané na aktivní registraci konzumenta k emitoru, kterou představuje instance třídy implementující rozhraní *Subscription*. Pokud konzument takto předem nezašle signál o své připravenosti přijímat data, nesmí být tato metoda emitorem volána (pravidlo Subscriber Rule ID#1).

onError

Pokud v emitoru z nějakého důvodu dojde k chybě, je povinen o tom pomocí této metody zpravit všechny své registrované konzumenty (pravidlo Publisher Rule

ID#4). V parametru metody je zasílána informace o vzniklé chybě, aby byli konzumenti schopni adekvátním způsobem reagovat.

Důležitým implementačním pravidlem je, že jakmile emitor signalizuje *onError*, musí být registrace konzumenta k emitoru považována za zrušenou. Toto neplánované zrušení registrace musí být respektováno a ošetřeno jak na straně emitoru, tak na straně konzumenta, neboť oba mají přístup k dané instanci *Subscription* – a z tohoto důvodu je i toto neplánované zrušení popsáno dvěma pravidly, na obou stranách (Publisher Rule ID#6 a Subscriber Rule ID#4).

Chyba na emitoru je tedy z hlediska konzumenta považována za tzv. *neplánovaný terminální stav*. Po dosažení tohoto stavu a signalizace *onError* nesmí dojít ze strany emitoru k žádnému dalšímu zasílání signálů konzumentům (pravidlo Publisher Rule ID#7). Tím se liší od tzv. *plánovaného terminálního stavu*, u kterého může dojít k zasílání signálů i po přechozí žádosti konzumenta o zrušení registrace (viz kapitola 4.3.4).

Jak již bylo uvedeno, emitor může zasílat konzumentu data pouze poté, co konzument prvně signalizuje svoji připravenost a o data zažádá. To se ovšem netýká signálů oznamující neplánovaný terminální stav – konzument tedy musí být připraven přijmout signál *onError*, i když o žádná data nezažádal (pravidlo Subscriber Rule ID#10), neboť selhání emitoru může být plně nezávislé na požadavcích konzumentů. Ovšem nadále platí, že signál *onSubscription* je vždy prvním signálem zaslaným emitorem konzumentu po jeho registraci. Signál *onError* může být ale zaslán ihned vzápětí.

onComplete

Pokud emitor úspěšné dokončí emitaci veškerých svých dat a nehodlá v budoucnu v emitaci pokračovat, je tuto skutečnost povinen pomocí této metody signalizovat všem svým konzumentům (pravidlo Publisher Rule ID#5).

Správná implementace této metody je obzvláště důležitá u konzumentů takových emitorů, o kterých se dá očekávat, že emitují *konečný stream dat*.

Emitor může, ale nemusí, o své konečnosti předem vědět. Obecně tak můžeme konečné emitory rozdělit na *plánovaně konečné*, které jsou si o své konečnosti vědomi po celou dobu svého životního cyklu a mají již předem nějakým způsobem definovanou konečnou množinu dat určenou k emitaci. Druhým typem jsou pak emitory *neplánovaně konečné*, kterým je informace o konci dat, případně pokyn k ukončení emitace veškerých dat, signalizována třetí stranou v předem neznámem čase. Protipólem pak jsou *nekonečné emitory*, u kterých se očekává neustálá připravenost propagovat nová data konzumentům, přičemž vznik nových dat lze považovat za teoreticky nekonečný děj. Na straně takovýchto nekonečných emitorů není pak v určitých případech nutné, aby byla signalizace konzumentům *onComplete* implementována, protože jejich jediným možným způsobem dosažení terminálního stavu je vznik běhové výjimky a signalizace *onError*.

Z toho tedy vyplývá, že z pohledu emitoru může být dokončení emitace konečného streamu považováno za *plánovaný terminální stav* (plánovaně konečné emitory) či *neplánovaný terminální stav* (neplánovaně konečné emitory). Naproti tomu z pohledu konzumenta se vždy jedná o terminální stav neplánovaný, protože konzument je zcela nezávislý na vnitřní implementaci emitoru a jako takový předem nemůže vědět o jeho případné konečnosti.

A právě v důsledku toho, že v případě skončení konečného streamu nastává pro konzumenta neplánovaný terminální stav, platí zde stejně jako u metody *onError* pravidlo Publisher Rule ID#7 – tedy po dosažení neplánovaného terminálního stavu a signalizaci *onComplete* nesmí dojít ze strany emitoru k žádnému dalšímu zasílání signálů konzumentům. Zde se jedná hlavně o zamezení propagace případné běhové chyby po skončení konečného streamu signálem *onError*.

Dále stejně jako v případě metody *onError* zde platí pravidla Publisher Rule ID#6 a Subscriber Rule ID#4 – tedy po signalizaci *onComplete* musí být registrace konzumenta k emitoru považována za zrušenou. Díky tomuto principu lze předcházet zbytečné alokaci zdrojů na registrace konzumentů k emitorům s vyčerpanými daty.

Poslední paralela s metodou *onError* je požadavek na konzumenta být připraven přijmout signál *onComplete* i bez toho, aniž by svému registrovanému emitoru prvně signalizoval svoji připravenost přijímat data (pravidlo Subscriber Rule ID#9). Jak již bylo zmíněno v popisu metody *onError*, signály oznamující neplánovaný terminální stav mají z principu backpressure zjednodušeně řečeno

„výjimku“, neboť vyčerpání konečného streamu může být zcela nezávislé na požadavcích určitého konzumenta.

4.3.4 Subscription

```
public interface Subscription {
    public void request(long n);
    public void cancel();
}
```

Instance třídy implementující toto rozhraní představuje registraci konzumenta (Subscriber) k emitoru (Publisher). Tato instance je vytvořena emitorem v tovární metodě *subscribe* a po jejím vytvoření je obratem zaslána konzumentu pomocí metody *onSubscribe*. Tímto způsobem je Subscription vždy k dispozici jak emitoru, tak i příslušnému konzumentu, který má tak možnost pomocí této instance s registrovaným emitorem komunikovat. Tato komunikace je výhradně jednostranná ze strany konzumenta a sestává ze dvou metod sloužících k signalizaci připravenosti přijímat data (implementace backpressure) a k ukončení registrace ze strany konzumenta (způsob dosažení plánovaného terminálního stavu).
K rozhraní se váže 17 implementačních pravidel (Subscription Rule ID#1-17).

request

Emitor můžeme v kontextu reaktivních streamů obecně označit jako zdroj teoreticky nekonečného proudu dat, přičemž data mohou proudit libovolnou rychlostí (tj. elementy proudí po sobě v libovolných intervalech), která se navíc může v čase měnit. Díky tomu se může snadno stát, že emitor bude vysílat data v takovém množství a s takovou rychlostí, že si s tím určití konzumenti nemusí dostatečně rychle poradit a mohou být vysílanými daty zahlceni. Z tohoto důvodu musela být do specifikace Reactive streams přidána regulace datového toku – tedy již dříve teoreticky popsaný princip backpressure. Přístupů k implementaci tohoto principu může existovat více, v rámci specifikace Reactive streams je jím ale právě metoda *request*.

Pokud chce konzument po registraci k emitoru začít přijímat data, musí nejprve signalizovat touto metodou svoji připravenost, jinak mu nesmí být žádná data zaslána (pravidlo Subscriber Rule ID#1). V parametru metody konzument zasílá číslo *n*, které značí počet elementů, které následně musí být připraven od emitoru v libovolném čase přijmout a zpracovat. Parametr *n* musí být kladné celé číslo (pravidlo Subscription Rule ID#9).

Požadavky touto metodou můžeme označit jako tzv. *n-aditivní* (pravidlo Subscription Rule ID#8). To znamená, že emitor si pro každého svého konzumenta pamatuje celé číslo *N*, přičemž každý signál *request* zvětší toto číslo *N* o parametr metody *n* (tedy metoda *request* provede změnu $N = N + n$). Celkový počet emitovaných elementů danému konzumentu (tedy celkový počet signalizací *onNext*) musí pak být vždy menší nebo roven právě číslu *N* (pravidlo Publisher Rule ID#1).

Z toho vyplývá, že čas signalizace *request* a počet těchto signalizací je libovolný. Konzument nemusí čekat, až od emitoru příjme počet elementů požadovaných předchozím signálem, ale může pružně zvyšovat maximální velikost fronty požadovaných elementů podle svých možností. Obecně lze doporučit při jednom volání metody *request* volit parametr *n* takovým způsobem, ale reflektoval aktuální maximální počet možných elementů, který je konzument schopný zpracovat. V praxi se takové *n* určuje hůře – při zvolení příliš vysokého čísla může v určitých chvílích dojít k vysokému náporu dat, přetížení řetězce datového proudu a vzniku chyb v důsledku nedostatku systémové paměti. Při zvolení příliš nízkého čísla může dojít ke zbytečnému zpomalení systému kvůli zvýšení režijních nákladů na příliš časté signalizování požadavku na další data či kvůli vyšším prodlevám v příjmu dat v důsledku asynchronní povahy signálu *request* mezi konzumentem a emitorem. Tento problém je otázkou optimalizace konzumenta a musí být řešen na míru dané komponentě.

Více o implementaci principu backpressure v kapitole 4.3.6.

cancel

Tímto signálem žádá konzument svůj registrovaný emitor o zrušení této registrace. Jedná se o jediný způsob ze strany konzumenta, jak registraci zrušit a tím pádem se zároveň jedná o jediný způsob dosažení plánovaného terminálního stavu registrace.

Registrováním konzumenta k emitoru vzniká konzumentu povinnost zrušit tuto registraci, pokud daná registrace již není potřeba (pravidlo Subscriber Rule

ID#6). Příkladem může být konzument, kterého zajímá pouze získání určitého elementu, nebo pouze předem daný počet elementů. Tímto pravidlem lze předejít zbytečné alokaci zdrojů a snížit režijní náklady.

Definici této metody určuje vícero pravidel specifikovaných ve všech třech v tuto chvíli představených rozhraních. Základní funkčnost metody specifikuje pravidlo Publisher Rule ID#8, které říká, že pokud je touto metodou registrace konzumenta zrušena, musí emitor časem přestat zasílat konzumentu signály. Stejnou funkcionalitu požaduje obměněné pravidlo Subscription Rule ID#12, které specifikuje, že metoda *cancel* musí signalizovat emitoru, aby časem přestal zasílat data danému registrovanému konzumentu. Toto pravidlo však navíc obsahuje dovětek obsahující důležitou informaci, která byla doposud pouze naznačena slovem „časem“: Tato operace nevyžaduje, aby byla daná registrace ovlivněna okamžitě.

Toto chování již bylo částečně nastíněno v kapitole 4.3.3, kde byl představen koncept plánovaného a neplánovaného terminálního stavu. Tyto stavy se liší tím, že neplánovaný terminální stav vzniká ze strany emitoru, signalizuje *onError* nebo *onComplete* a pokud nastane, nesmí být emitorem zaslán žádný signál jeho registrovaným konzumentům. Naproti tomu plánovaný terminální stav vzniká ze strany konzumenta signalizací *cancel*, ale poté stále může docházet k zasílání dat emitorem metodou *onNext*. Takové chování je dáno asynchronní podstatou reaktivních streamů. Protože *cancel* je asynchronním signálem, může dojít k prodlevě mezi signalizací požadavku o zrušení registrace a přijmutím tohoto signálu emitorem. Pokud tedy konzument zažádá o zrušení registrace, ale stále mu nebyl emitorem zaslán požadovaný počet elementů zažádaných metodou *request*, může stále ze strany emitoru docházet k zasílání dat, dokud k němu požadavek na zrušení nedorazí. Konzument tedy musí být připraven od emitoru přijmout data i po signalizaci *cancel*, pokud o tato data předtím zažádal (pravidlo Subscriber Rule ID#8).

4.3.5 Processor

```
public interface Processor<T, R> extends Subscriber<T>,Publisher<R> {
}
```

Jedná se o rozhraní definující komponentu, která se chová jako emitor a zároveň jako konzument. Toto rozhraní pouze dědí metody z příslušných kontraktů pro konzumenta (Subscriber) a emitor (Publisher), ale žádné nové nepřidává.

Objekty tříd implementující rozhraní Processor slouží většinou pouze jako výpočetní a transformační uzly mezi prvotním emitorem a koncovým konzumentem. I když implementace takovýchto průtokových mezičlánků v řetězci zpracování nemusí být tolik sofistikovaná, jako je tomu u implementace koncových komponent, přesto pro ně platí stejná pravidla a omezení popsaná v předchozích kapitolách pro rozhraní Subscriber a Publisher. Dále se k nim navíc vážou 2 doplňující implementační pravidla (Processor Rule ID#1-2).

4.3.6 Více o back-pressure

Implementační řešení principu backpressure ve specifikaci Reactive streams lze shrnout jako konzumentem kontrolovaná fronta emitorem zasílaných elementů.

Každý emitor si ke svým jednotlivým konzumentům pamatuje celá čísla N a P, kde N je celkový počet konzumentem zažádaných elementů pomocí metody *request* a P je celkový počet emitorem zaslaných elementů pomocí metody *onNext*. Emitor musí za všech situacích dodržet, že $P <= N$ a konzument si může být v každý moment jist, že mu emitor v budoucnu nezašle více elementů než p, kdy $p = N - P$. Regulace toku tedy probíhá ze strany konzumenta mírou zvyšování hodnoty N za pomoci metody *request* v závislosti na jeho aktuální vytíženosti.

Tento přístup, kdy je konzument schopen ovlivňovat tok dat pouze v rámci svoji fronty, skýtá mnoho výhod. Jako příklad lze uvést emitor, který kontinuálně zasílá vysoké množství dat. K tomuto emitoru jsou registrováni dva konzumenti, přičemž první konzument bez problémů stíhá zpracovávat vysoké množství emitovaných dat, ale druhý, se složitějším zpracováním dat, takové množství přijímat nestíhá. Při nejjednodušší přímočaré implementaci principu backpressure by pomalejší konzument signalizoval žádost emitoru o snížení propustnosti dat, emitor by následně tento svůj objem propuštěných dat snížil a v případě potřeby signalizoval stejný požadavek svým nadřazeným komponentám. Tím pádem by došlo ke zbytečnému zpomalení druhého konzumenta, který objem propuštěných dat bez potíží zvládal a navíc by se zpomalení propagovalo do všech emitoru nadřazených komponent nacházejících se v řetězci zpracování proti proudu toku dat. Oproti tomu v implementaci backpressure specifikace Reaktivních streamů, když pomalejší konzument signalizuje emitoru nižší požadavky na objem dat, tak ovlivní pouze velikost své vlastní fronty, ale nedojde ke zpomalení emitoru nadřazených komponent ani druhého konzumenta, který je i tak nadále schopen přijímat nezmenšený objem dat stejnou rychlostí.

Tento přístup však nevylučuje možnost využití naivního přímého kaskádového backpressure, kdy emitor podle potřeby pouze propaguje signál k regulaci toku do nadřazených komponent a sníží tak propustnost v celém řetězci. Použití takového řešení je vhodné zejména pro jednodušší objekty typu *Processor*, které se chovají jako emitory a zároveň jako konzumenti a slouží hlavně jako transportní či transformační mezičlánky uprostřed řetězce zpracování. U těchto jednodušších objektů může být takovýmto přímým řešením zodpovědnost za řízení objemu toku pouze delegována komplexnějším emitorům nacházejících se v nadřazené části řetězce. Využitím tohoto přímého řešení v jednoduchých mezičláncích lze dosáhnout značného snížení režijních nákladů a celkovému zrychlení systému. Ovšem jeho využitím v komplexních článcích či uzlech řetězce lze naopak rychlost drasticky snížit a velmi snadno se pak může stát, že rychlost celého systému degraduje na rychlost nejpomalejšího článku řetězce.

Při úvahách o výše uvedeném příkladu může nastat otázka, co se stane s elementy, které první konzument bez problémů příjme, ale druhý konzument z důvodu zahlcení nemůže. Ten samý problém může nastat i v kaskádovém řešení, kdy nelze backpressure dále propagovat výše, tedy když nelze potlačit přísun nových elementů (např. při přísunu elementů na základě uživatelova pohybu myši, plynutí času apod.). Přístupů existuje více, záleží vždy na individuálních potřebách daného emitoru - nejjednodušším přístupem je samozřejmě neemitované elementy zahazovat. Velmi často se ale nabízí využití jedinečného zásobníku elementů pro každého registrovaného konzumenta. Pokud pak emitor má k dispozici element k zaslání, ale nemůže ho propagovat konzumentu v důsledku jeho nezájmu, může daný element umístit do pro něj určeného zásobníku. V případě žádosti konzumenta o nová data pak emitor zašle nejprve data ze zásobníku a poté až pokračuje v zasílání klasickým způsobem. Tímto stylem lze zajistit, že rychlejší i pomalejší konzument dostanou vždy nakonec stejná data, ve stejném pořadí a v případě konečného streamu pak i stejný počet elementů (za předpokladu shodného času registrace k emitoru). Při konečném streamu pak může vzniknout i situace, kdy emitor dosáhl terminálního stavu v důsledku vyčerpání konečné množiny elementů k emitaci, nelze na něj registrovat nové konzumenty, ale má aktivní registrace a stále emituje data – pouze ale zpětně ze zásobníku, pomalejším konzumentům.

Obecně můžeme implementaci principu backpressure v Reaktivních streamech označit jako *preemptivní*. Jako taková se snaží předcházet zahlcení konzumenta tím, že předem sám musí o data žádat, pokud ví, že je dokáže zpracovat.

4.4 Reaktivní knihovny

Zatímco specifikaci Reaktivních streamů lze se svými rozhraními a sadou implementačních pravidel považovat za obecný předpis implementace reaktivních principů popsaných v Reaktivním manifestu, reaktivní knihovny pak představují konečnou implementaci v rámci jedné specifické technologie.

Prakticky všechny moderní reaktivní frameworky na bázi JVM dnes implementují specifikaci Reactive streams (38) a řadí se tak mezi 3+ generaci reaktivních knihoven. Kromě Reactive streams pro JVM v současnosti existuje i stejná specifikace pro .NET (39) a vzniká i pro další jazyky, mezi kterými je např. JavaScript (40).

Mezi reaktivní knihovny můžeme nepřímo řadit i např. reaktivní ovladače pro databáze, které v současnosti existují např. pro MongoDB, Cassandra, Redis, Couchbase (41). Endpointy těchto databázi se pak umějí chovat jako konzumenti/emitory, komplementují principu backpressure a jsou brány jako implementace specifikace Reactive streams (38).

Z tohoto důvodu byl v této práci v rámci popisu implementace reaktivních principů brán maximální důraz na specifikaci Reactive streams a ne přímo na koncové knihovny. Díky plnému pochopení principů reaktivních rozhraní Publisher, Subscriber, Subscription, Processor a jejich vzájemné interakce lze i plně pochopit fungování všech moderních reaktivních frameworků a to i napříč programovacími jazyky a technologiemi a to včetně těch, které v době psaní této práce nemusely ještě existovat.

Specifikace Reactive streams a její pravidla jsou tak přesně definovaná, že jednotlivé knihovny jsou si v základu funkčně velmi podobné, drobné rozdíly lze pak snadno dohledat v dokumentaci a většinou závisí na jemných nuancích daných technologií. Jako příklad lze uvést třídu z knihovny RxJava *io.reactivex.Flowable* (42) a třídu z knihovny Reactor *reactor.core.publisher.Flux* (43) – obě třídy představují emitor v daných technologiích, obě implementují rozhraní *Publisher* a emitují data skrze asynchronní neblokující stream s podporou backpressure. V obou knihovnách lze snadno dohledat konečnou implementaci všech principů popsaných v Reaktivních streamech. Např. v knihovně RxJava lze vytvořit emitor s anotací *@BackpressureSupport(BackpressureKind.PASS_THROUGH)* (42), z čehož jde i bez znalosti dané knihovny díky specifikaci Reactive streams odvodit, že se jedná o emitor s přímým kaskádovým backpressure, který pouze propaguje žádost o regulaci toku nadřazeným komponentám (viz kapitola 4.3.6). Jako opak lze uvést příklad z knihovny Reactor, která obsahuje třídu *reactor.core.publisher.FluxBuffer* (43), což logicky značí emitor schopný ukládat elementy do zásobníku, pokud konzument nestíhá, bez potřeby snižovat rychlost datového toku v jiných částech řetězce zpracování.

Díky mnoha podobnostem a prakticky shodným možnostem v oblasti reaktivního programování se nabízí otázka, kterou z knihoven zvolit a podle jakých kritérií. Samozřejmě díky odlišné implementaci vnitřní logiky lze spekulovat o rozdílu rychlosti jednotlivých knihoven a dále třeba o optimalizaci fúzi operátorů jednotlivých implementací. K tomu se vyjádřil i Dávid Karnarok, vedoucí člen projektu RxJava a nezávislý kontributor projektu Reactor, na svém Twitteru (44), kde navrhuje použít Reactor na projektech s Javou 8+ a RxJava na starších projektech s Java verzí 6+ (kterou Reactor již nepodporuje). Rychlost obou těchto hlavních reaktivních knihoven označil obecně za téměř shodnou.

Volba dané knihovny tedy nemusí být otázkou přímo z oblasti reaktivního programování, kde je jejich funkčnost a výkonnost prakticky totožná, ale spíše je vhodné zamyslet se nad případnými dalšími funkcionalitami, které určitá knihovna nabízí a podle toho určit její vhodnost použití na projektu.

RxJava (ReactiveX)

RxJava je reaktivní knihovna pro konstrukci asynchronních aplikací na bázi šíření událostí za použití pozorovatelných sekvencí (25). Jako jedna z mála i na nejnovější verzi stále podporuje Java verzi 6+ a Android 2.3+.

Tato knihovna slouží čistě jako implementace reaktivního programování a neobsahuje předpřipravenou podporu např. pro zakomponování v rámci reaktivního systému, komunikaci s reaktivní databází či jednoduché vystavění reaktivního serveru (všechno z toho lze samozřejmě s knihovnou naprogramovat svépomocí). Jedná se tedy o celkem lightweight jednoúčelovou knihovnu vhodnou pro tvorbu koncových klientských aplikací – tato vlastnost může být velký plus pro vývoj reaktivních aplikací pro operační systém Android (45). Knihovna navíc obsahuje i emitory bez jakékoliv schopnosti backpressure, které mohou být užitečné na snížení režijních nákladů v situacích, kdy backpressure nelze vynutit – např. příjem uživatelových instrukcí skrz dotykový displej.

Akka

Akka je knihovna zaměřující se na tvorbu reaktivních distribuovaných systémů, které jsou odolné, pružné, decentralizované a fungují na bázi zasílání zpráv (46). Knihovna implementuje *Aktorový model*, ve kterém jednotlivý *Actors* představují nezávislé a uzavřené jednotky systému zasílající se mezi sebou zprávy (47). Tento přístup přímo staví na principech reaktivních systémů a mikroservisní architektury.

Podpora reaktivního programování na principu reaktivních streamů a backpressure pak přichází jako „bonus“ a klade důraz na snadnou implementaci reaktivního HTTP serveru/klienta pro komunikaci mezi jednotlivými *Actors*.

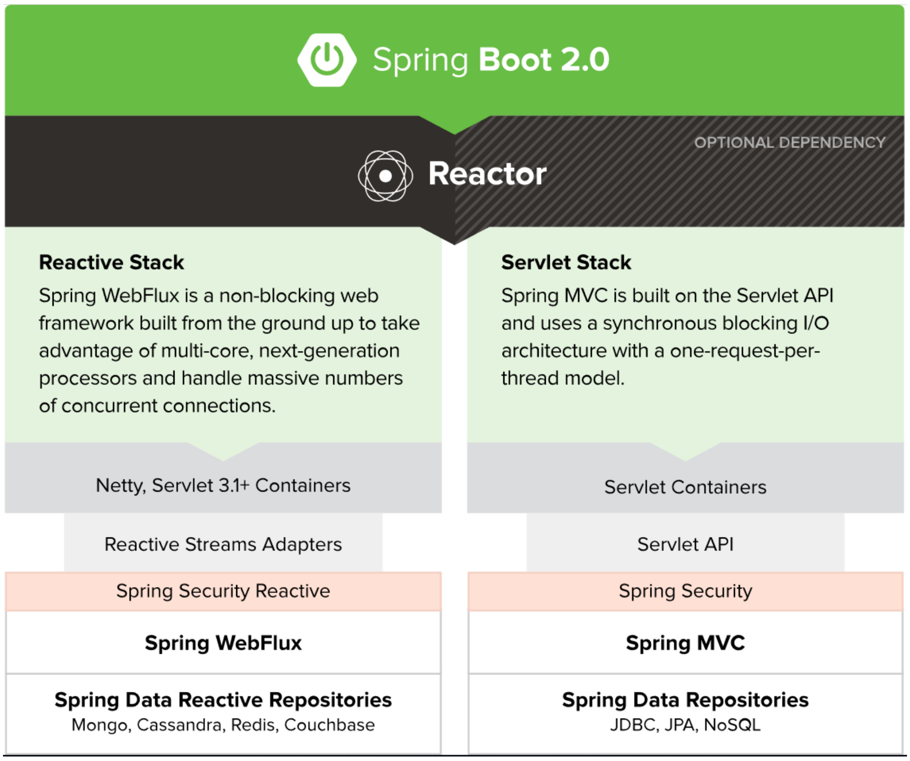
Project Reactor

Reactor je další z knihoven zaměřující se na čistě na reaktivní programování implementující specifikaci Reacive streams (32). Stejně jako RxJava ji lze použít v aplikaci samostatně, mnohem zajímavější je ale její využití ve spojení s dalšími frameworky. Za vývojem Reactoru stojí firma Pivotal, jejíž tým má na starost mimo jiné vývoj Springu – a právě ve Springu 5 slouží Reactor jako základ všech jeho reaktivních modulů. Dalším příkladem může být R2DBC (48), což je vyvíjený projekt týmem Pivotal využívající Reactor se snahou vytvořit reaktivní ovladače pro relační databáze a nahradit tak JDBC, které svou podstatou nabízí pouze blokující přístup.

Spring 5

Novinkou ve Springu 5 je mimo jiné integrace podpory pro reaktivní programování se specializací na webové aplikace zvané WebFlux (49). Spring 5 staví na Reactoru jako základní knihovně a nabízí podporu reaktivních servletů, databázových ovladačů, zabezpečení a reaktivní napojení pro interakci s dalšími frameworky ve Springu často využívanými (např. Thymeleaf).

WebFlux jako reaktivním modul koexistuje s klasickým servlet modulem Spring MVC a je koncipován tak, aby reaktivní abstrakce, kterou nabízí, byla tomuto modulu velmi podobná. Detailněji bude WebFlux představen v kapitole 5.



Obr. 3 Webstack schéma Spring 5 (49)

Ostatní

Mezi další implementace se řadí např. Vert.X (50) nebo RatPack (51).

5 Aplikace reaktivních principů

V této kapitole bude popsán proces návrhu a implementace ukázkové webové aplikace za použití reaktivního programování.

Hlavním účelem aplikace bude ověření principů a výhod reaktivního přístupu, které byly teoreticky popsány v předchozích kapitolách. Toto ověření proběhne praktickým měřením v 6. kapitole a to v přímém porovnání s přístupem imperativním – pro účely porovnání tedy k ukázkové reaktivní aplikaci musí vzniknout i klasický imperativní ekvivalent. Výslednou aplikaci tedy bude možné označit jako „benchmark“ pro porovnání chování reaktivní a klasické webové služby v určitých zátěžových situacích.

Díky koncipování ukázkové aplikace jako velmi obecného benchmarku mezi reaktivním a imperativním přístupem ji bude možné kromě prostého ověření reaktivních principů použít i jako simulaci chování libovolné webové služby pod zátěží. Druhým účelem aplikace tedy bude pomoc s odpovědí na otázku, zda-li by se vyplatilo určitou webovou aplikaci napsat reaktivně, nebo jestli by daná aplikace příliš nečerpala z výhod, které reaktivní přístup může přinášet. Více v kapitole 6.4.

5.1 Implementační požadavky

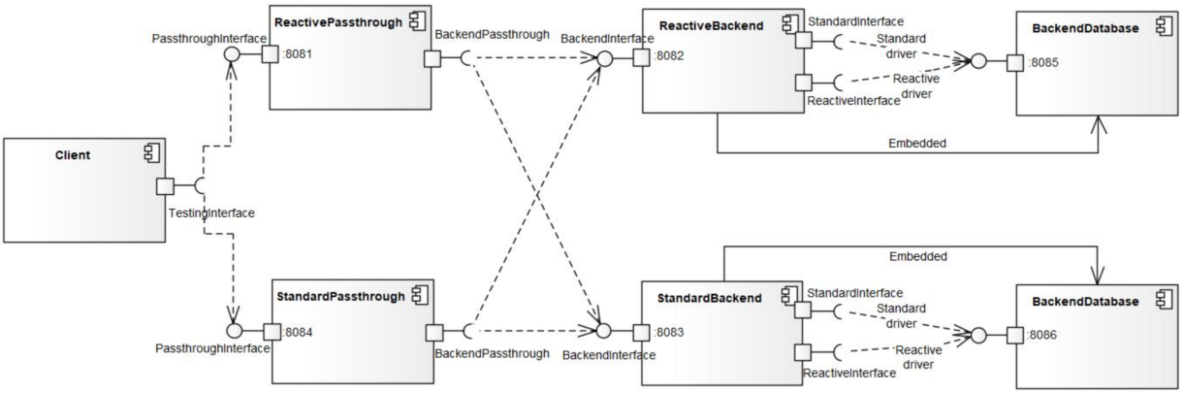
Díky teoretickým znalostem z předchozích kapitol o chování reaktivních webových aplikací lze nyní specifikovat implementační požadavky na benchmark aplikaci:

- Aplikace musí testovat komunikaci mezi reaktivním klientem, reaktivní webovou službou a reaktivní databází – každý tento prvek bude představovat samostatnou komponentu v aplikaci
- Pro každou testovanou reaktivní komponentu musí vzniknout její klasický imperativní ekvivalent
- Pro testovací účely musí být umožněno napojení mezi reaktivní a klasickou komponentou. Reaktivní komponenta a její klasický protějšek proto budou mít vždy shodná vystavovaná i konzumovaná rozhraní.
- Benchmark musí být schopen sledovat tok testovací zprávy v čase skrz aplikaci. Díky tomu bude možné identifikovat komponentu představující případné úzké hrdlo v datovém toku.

- Benchmark musí být schopen simulovat prodlení odpovědi z webové služby v důsledku interní business logiky reálných aplikací.
- Webová služba musí být schopna číst a zapisovat do databáze.
- Benchmark bude obsahovat grafické uživatelské rozhraní (GUI)
- Pomocí GUI bude umožněno vytvářet simulované uživatele zasílající dotazy na testovací aplikaci. Bude možné specifikovat počet virtuálních uživatelů, počet jejich dotazů a prodlení mezi jejich dotazy.
- Pomocí GUI bude možné jednoduše kontrolovat tok testovacích zpráv skrz aplikaci. Tedy bude obsahovat volby mezi reaktivními a klasickými komponentami, interakci s databází, délku prodlení odpovědi apod.
- GUI bude umožňovat grafické zobrazení výsledku jednotlivých testů pomocí grafů. Bude umožněno porovnání výsledky více testů v jednom grafu.
- Grafy musí umět zobrazovat jak celková data, tak standardní statistické údaje (průměr, max, min, směrodatná odchylka, průtok…).
- Grafy musí umět zobrazovat údaje o toku zprávy skrz aplikaci jako celku, ale také pouze mezi jednotlivými komponentami

5.2 Návrh ukázkové aplikace

Ze stanovených požadavků lze identifikovat 4 druhy komponent – testovací klient obsahující GUI pro interaktivní testování a zobrazení výsledků (*Client*), dále mezičlánková komponenta v roli konzumenta jiné služby (*Passthrough)*, konečná webová služba se simulací vnitřní logiky (*Backend*) a nakonec databáze (*Database*). Poslední tři zmíněné komponenty musí existovat v reaktivní a klasické variantě. Z toho lze vyvodit schéma aplikace zachycené v UML diagramu komponent (Obr. 4):



Obr. 4 UML diagram komponent ukázkové aplikace. Vlastní zpracování

Scénář použití

Další krok zahrnuje vytvoření scénáře použití aplikace a popis průchodu zprávy datovým tokem skrz komponenty.

Uživatel spustí testovacího klienta aplikace. Klient pomocí GUI zobrazí uživateli možnost nadefinovat test-case. Uživatel vybere typ implementace pro komponenty *Passthrough, Backend* a *Database* z výběrových hodnot *Standard*/*Reactive.* Dále zadá délku prodlení odpovědi z komponenty *Backend* pro simulaci interní logiky a jestli má dojít ke čtení a zápisu z databáze. V poslední řadě vybere počet simulovaných uživatelů aplikace, počet dotazů zaslaných každým uživatelem a prodlevu mezi jejich dotazy. Po vyplnění všech hodnot systém umožní uživateli zahájit test kliknutím na tlačítko „Start test".

Systém vytvoří požadovaný počet simulovaných uživatelů a ti začnou simultánně zasílat zprávy dále po řetězci zpracování. Před odesláním zprávy každý simulovaný uživatel nejprve vyhodnotí adresu komponenty *Passthrough* dle jejího požadovaného typu a označí do zprávy čas zaslání dotazu klientem.

Zpráva je následně přijata komponentou *Passthrough* a čas tohoto přijetí je do zprávy zaznamenán. Systém vyhodnotí adresu komponenty *Backend* dle jejího požadovaného typu a zprávu ji postoupí.

Po přijetí zprávy komponentou *Backend* je znovu tento čas do zprávy zaznamenán. Pokud si uživatel přál provést čtení z databáze, jsou z databáze z důvodu zátěžového testování přečteny veškeré přítomné záznamy a do zprávy je uložen první nalezený záznam. Pokud si uživatel přál provést zápis do databáze, je vytvořen nový záznam s náhodnými daty a do databáze uložen. Následně služba počká uživatelem požadovaný čas než zašle odpověď zpět komponentě *Passthrough*.

Komponenta *Passthrough* po přijetí odpovědi z *Backend* komponenty tento čas standardně do zprávy zaznamená a zprávu vrátí jako odpověď zpět testovacímu klientu, přesněji příslušnému simulovanému uživateli.

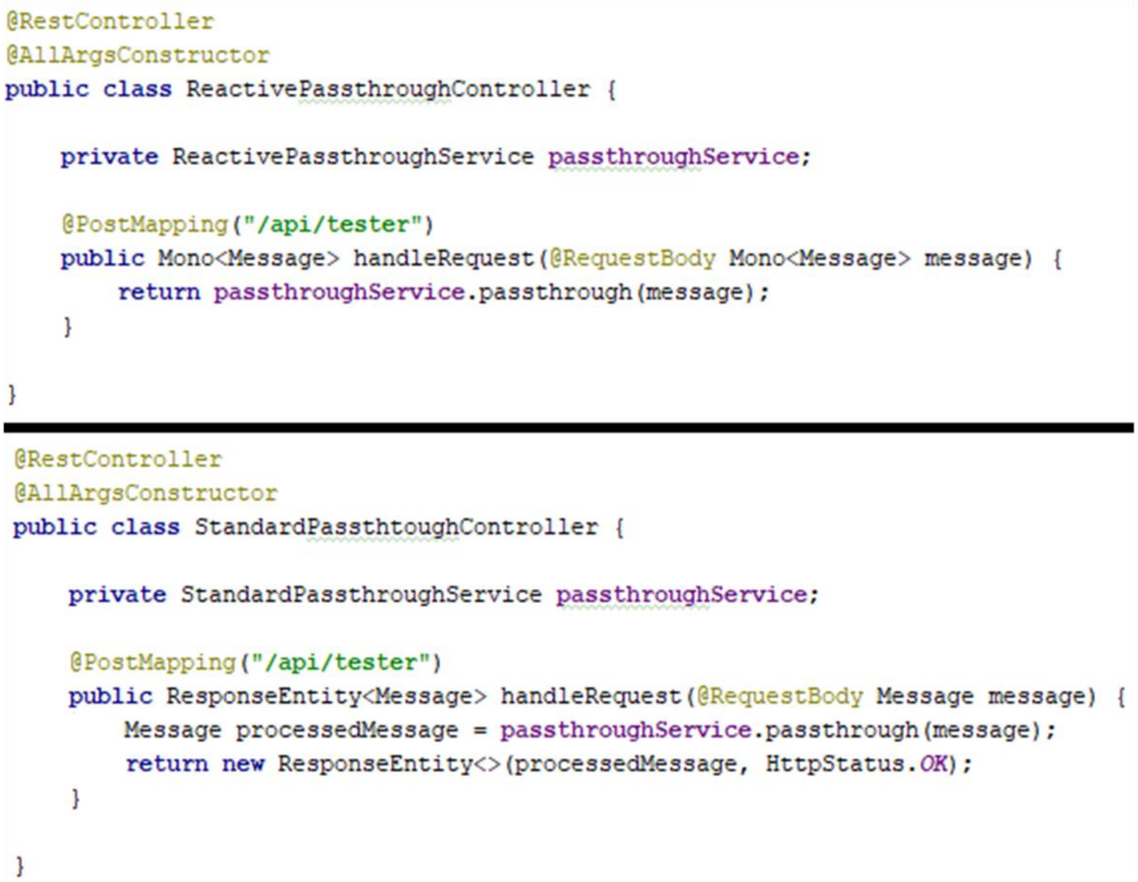
Simulovaný uživatel zaznamená čas vrácení odpovědi po průchodu zprávy celým datovým tokem. Pokud došlo v určité části průchodu zprávy tokem k chybě, tuto skutečnost do zprávy zaznamená též.

Po skončení zasílání zpráv všemi simulovanými uživateli systém shromáždí a statisticky vyhodnotí časy průchodů zpráv aplikací. Tato data systém zobrazí uživateli ve formě grafů. Při spuštění více testů bude uživateli nabídnuta možnost porovnat v jednom grafu libovolné množství výsledků těchto testů.

5.3 Implementace ukázkové aplikace

Aplikace bude implementována v jazyce Java pomocí frameworku Spring 5 s reaktivním základem v knihovně Reactor.

Reaktivní modul knihovny Spring 5 se nazývá Spring WebFlux a obsahuje stejné programové abstrakce v oblasti vývoje webových aplikací jako klasický modul Spring MVC. Jako příklad lze uvést způsob vystavení rozhraní REST pro komunikaci mezi komponentami aplikace. K tomuto účelu jsou v reaktivním i klasickém modulu k dispozici shodné anotace a shodný způsob jejich použití. Jako ukázka můžou sloužit implementovaná rozhraní *PassthroughInterface* komponent *Passthrough* na obrázku č. 5:



```
@RestController
@AllArgsConstructor
public class ReactivePassthroughController {

    private ReactivePassthroughService passthroughService;

    @PostMapping("/api/tester")
    public Mono<Message> handleRequest(@RequestBody Mono<Message> message) {
        return passthroughService.passthrough(message);
    }

}
```

```
@RestController
@AllArgsConstructor
public class StandardPassthtoughController {

    private StandardPassthroughService passthroughService;

    @PostMapping("/api/tester")
    public ResponseEntity<Message> handleRequest(@RequestBody Message message) {
        Message processedMessage = passthroughService.passthrough(message);
        return new ResponseEntity<>(processedMessage, HttpStatus.OK);
    }

}
```

Obr. 5 *PassthroughInterface Controllers*. Reaktivní komponenta nahoře, klasická dole. Vlastní zpracování

Na obrázku lze vidět absolutní abstrakci REST rozhraní od interní reaktivní vs. klasické implementace. Spring dokáže sám vyhodnotit konečnou implementaci podle importovaných modulů a to včetně serveru, na kterém budou rozhraní vystaveny při spuštění aplikace – v případě klasických komponent je využit Tomcat, reaktivní komponenty pak využívají Reactor Netty.

Na obrázku lze také vidět reaktivní obal nad objektem *Message*. V Reactoru rozeznáváme 2 typy reaktivních obalů – *Flux* a *Mono*. V obou případech se jedná o emitor implementující rozhraní *Publisher* specifikace *Reactive Streams*. *Flux* potom představuje emitor emitující libovolný počet elementů, *Mono* je speciálním případem emitoru, který je schopen emitovat maximálně jeden prvek.

Díky jednoduchosti komponenty *Passthrough* na ni lze jednoduše ukázat rozdíl mezi imperativním stylem programování pomocí jednotkových příkazů a deklarativním stylem reaktivního programování pomocí operátorů (obrázek č. 6):

```
public Mono<Message> passthrough(Mono<Message> message) {
    return message
            .map(Message::markFirstPassthrough)
            .flatMap(this::makeBackendRequest)
            .map(Message::markSecondPassthrough);
}

private Mono<Message> makeBackendRequest(Message message) {
    return webClient
            .post()
            .uri(message.isBackendReactive() ? reactiveBackendUrl : standardBackendUrl)
            .body(BodyInserters.fromObject(message))
            .retrieve()
            .bodyToMono(Message.class);
}
```

```
public Message passthrough(Message message) {
    message.markFirstPassthrough();
    Message receivedMessage = makeBackendRequest(message);
    receivedMessage.markSecondPassthrough();
    return receivedMessage;
}

private Message makeBackendRequest(Message message) {
    String url = message.isBackendReactive() ? reactiveBackendUrl : standardBackendUrl;
    return restTemplate.postForObject(url, message, Message.class);
}
```

Obr. 6 *Passthrough Service*. Reaktivní komponenta nahoře, klasická dole. Vlastní zpracování

Na obrázku lze také vidět použití reaktivního Spring REST klienta *WebClient* a klasického klienta *RestTemplate* (52).

Backend a Databáze

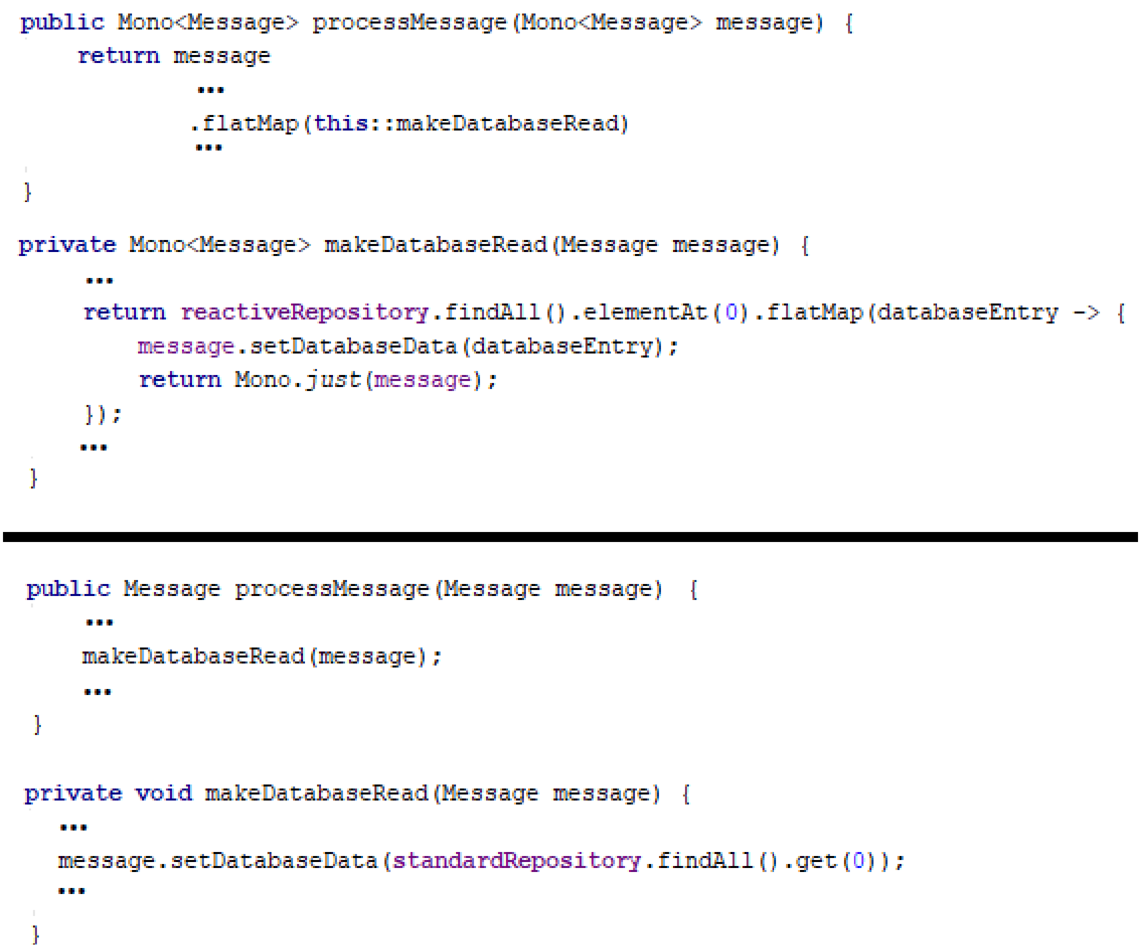
Účelem komponenty *Backend* je převážně komunikace s databází a simulace interní logiky aplikace zpožděním vrácení odpovědi.

Jako databáze byla zvolena MongoDB, která má integrovanou přímou podporu ve Springu 5 v modulu Spring Data MongoDB. Podporovány jsou jak reaktivní, tak klasické repozitáře a samozřejmostí je sada integrovaných ovladačů pro komunikaci s databází, též v reaktivní a klasické verzi. Databáze jako taková tedy neexistuje v reaktivní a klasické instanci, jako druhé dvě testovací komponenty, komponenta *Backend* pouze obsahuje dvě napojení na jednu instanci databáze – jednou pomocí rozhraní využívající reaktivní ovladače, a druhé napojení pomocí ovladačů klasických. Obecně nic jako „reaktivní databáze“ neexistuje, vždy hovoříme pouze o reaktivním přístupu do databáze, pomocí reaktivních ovladačů.

Pro snadné nasazení a testování databáze byla MongoDB integrována (*embedded*) do *Backend* komponenty (53). Databáze stále existuje jako samostatná komponenta, pouze životní cyklus její instance byl pro jednoduchost svázán s životním cyklem instance komponenty *Backend*, která ho dostala pod správu.

I *Backend* komponenta nabízí zajímavé porovnání klasického a reaktivního přístupu, v tomto případě u způsobu zažádání o všechny záznamy v databázi a vložení prvního nalezeného záznamu do odpovědi zprávy (zkráceno - obrázek č. 7).

Zatímco klasický přístup pracuje vždy přímo s objekty, reaktivní přístup pracuje vždy s reaktivními obaly (zde emitory Mono/Flux). V klasickém přístupu tedy lze přistoupit přímo k objektům a proto není problém vložit nalezená data z databáze pomocí *setteru* do zprávy. Oproti tomu v reaktivním přístupu, jediný způsob jak přistoupit k objektům v reaktivním obalu je skrz operátory (např. zde *flatMap*), které představují bod překročení asynchronní hranice. Tedy např. operátor *flatMap* se chová jako *Subscriber*, registruje se k danému emitoru, po přijetí elementu provede transformaci nad objektem a znovu vrátí *Publisher* tvořící reaktivní obal nad daným elementem.



```
public Mono<Message> processMessage(Mono<Message> message) {
    return message
            ...
            .flatMap(this::makeDatabaseRead)
            ...
}

private Mono<Message> makeDatabaseRead(Message message) {
    ...
    return reactiveRepository.findAll().elementAt(0).flatMap(databaseEntry -> {
        message.setDatabaseData(databaseEntry);
        return Mono.just(message);
    });
    ...
}
```

```
public Message processMessage(Message message) {
    ...
    makeDatabaseRead(message);
    ...
}

private void makeDatabaseRead(Message message) {
   ...
   message.setDatabaseData(standardRepository.findAll().get(0));
   ...
}
```

Obr. 7 *Backend Service*. Reaktivní komponenta nahoře, klasická dole. Zkráceno.
Vlastní zpracování

Na ukázkovém obrázku pak vzniká situace, kdy služba přijímá *Mono<Message>* z *Controlleru* a *Mono<DatabaseEntry>* z databáze, tedy dva nezávislé reaktivní obaly a je potřeba přistoupit do jejich vnitřních objektů a vložit jeden do druhého. Situace na obrázku je řešena použitím vnitřního operátoru *flatMap* uvnitř jiného operátoru *flatMap* pro přístup k oběma objektům. Tato situace tak poukazuje na rozdílnost obou přístupů, kdy klasický přístup může být považován oproti reaktivnímu za více přímočarý.

Další skutečností vhodnou k povšimnutí na obrázku 7 je volba operátoru *flatMap* pro volání metody *makeDatabaseRead*. Mezi metodou *map* a *flatMap* existují dva hlavní rozdíly (43). První rozdíl je ten, že *map* vždy vrací reaktivní obal nad stejnou instancí třídy (zde *Message*), oproti tomu *flatMap* může vracet obal nad jinou instancí třídy – např. vnitřní operátor *flatMap* v metodě *makeDatabaseRead* má na vstupu Mono<*DatabaseEntry*>, ale vrací *Mono<Message>*. Druhým rozdílem je, že *flatMap* je provedeno asynchronně, zatímco *map* je provedeno synchronně. Proto kdyby byl omylem použit operátor *map* pro volání metody *makeDatabaseRead*, čtení z databáze by proběhlo synchronně a mohlo by značně omezit výkonnost celé aplikace. Stejná situace je zobrazena na obrázku č. 6, kde je *map* použito pro jednotkové operace uložení aktuálního času do zprávy, ale pro volání vzdálené *Backend* služby je použit operátor *flatMap* pro zajištění asynchronicity volání.

Tyto dvě skutečnosti poukazují na vyšší obtížnost reaktivního programování oproti klasickému, kdy i pro nejjednodušší operace mohou vznikat relativně nepřímočará řešení a vývojář si musí stále dávat pozor, aby v některém nevhodném úseku nepoužil synchronní či dokonce blokující operaci, která by mohla ucpat datový tok, dramaticky tak snížit výkonnost celé aplikace a potlačit výhody vycházející z obecně asynchronního a neblokujícího přístupu reaktivního programování.

Klient

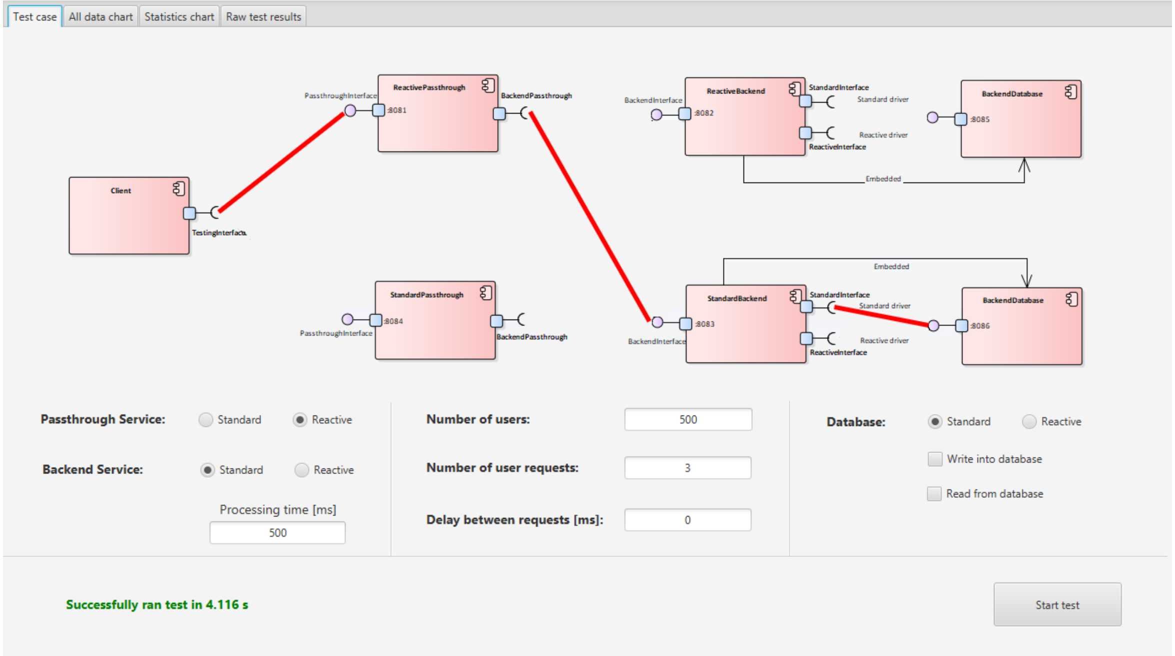
Testovací klient se skládá ze dvou částí – interní logiky pro testování aplikace a GUI.

Jakmile uživatel nastaví *a spustí test case,* režii převezme interní logika klienta, přesněji objekt *TestCaseRunner.* Tato komponenta vytvoří uživatelem požadovaný počet testovacích vláken, která začnou simultánně zasílat požadavky komponentě *Passthrough*. Každé vlákno zasílá požadavky synchronně pomocí *RestTemplate*, aby bylo docíleno simulace reálného uživatele čekajícího na výsledek dotazu, než zašle dotaz další.

Po spuštění všech vláken čeká *TestCaseRunner* na jejich ukončení pomocí metody *Thread.join()*. Po ukončení všech vláken předává objekt s výsledky do GUI, který jej zobrazí uživateli.

Uživatelské rozhraní klienta je vytvořeno pomocí knihovny JavaFX. Jedná se o standardní knihovnu pro tvorbu GUI v rámci Java SE (54). Rozhraní obsahuje 4 zobrazovací panely – *Test case, All data chart, Statistical chart* a *Raw test results*.

Panel *Test case* obsahuje nastavení testovacího scénáře se vstupními poli definovanými v implementačních požadavcích. Mimo to zobrazuje i schéma průtoku zpráv skrz komponenty pro daný scénář. Panel je k vidění na obrázku 8.



Passthrough Service: ○ Standard ● Reactive

Backend Service: ● Standard ○ Reactive

Processing time [ms]: 500

Number of users: 500

Number of user requests: 3

Delay between requests [ms]: 0

Database: ● Standard ○ Reactive

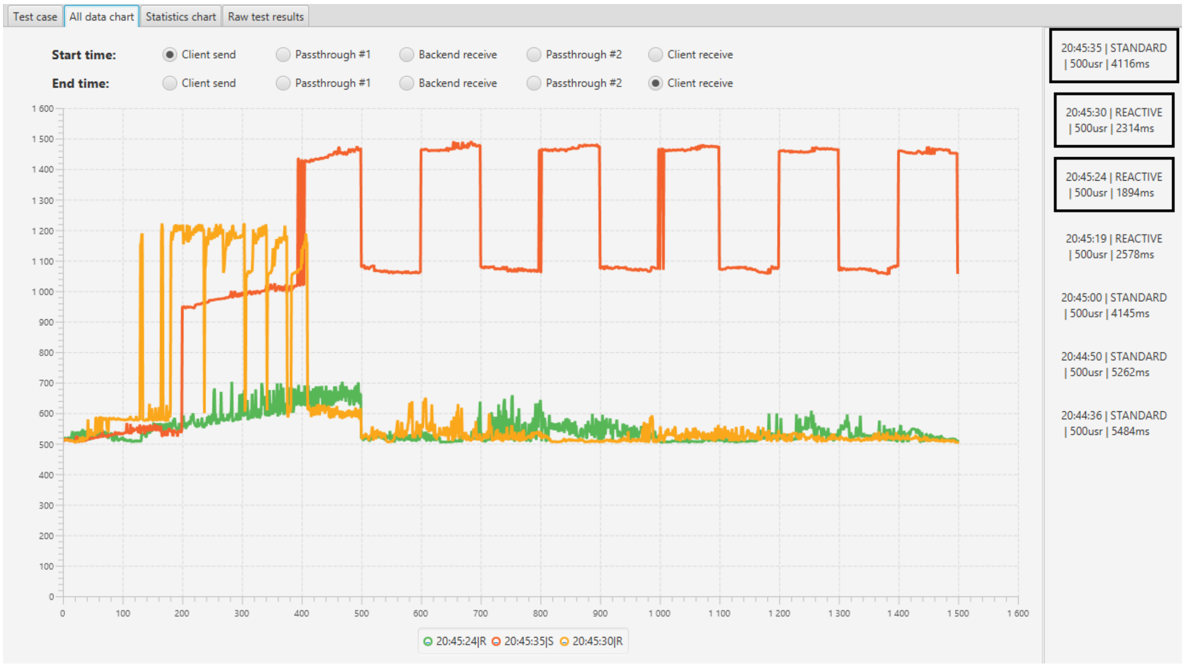
☐ Write into database

☐ Read from database

Successfully ran test in 4.116 s

Obr. 8 *Test case panel* v testovacím klientovi. Vlastní zpracování

Panel *All data chart* vizualizuje trvání průchodu mezi dvěma zvolenými komponentami všech zaslaných zpráv dle pořadí jejich odeslání. V pravém bočním panelu pak lze vybrat libovolné množství proběhlých testů, jejichž data se do grafu vizualizují. Samotný graf se nachází uprostřed panelu, přičemž osa X reprezentuje pořadí zaslané zprávy, osa Y představuje čas v milisekundách, který zprávě trval,



Start time: ● Client send ○ Passthrough #1 ○ Backend receive ○ Passthrough #2 ○ Client receive

End time: ○ Client send ○ Passthrough #1 ○ Backend receive ○ Passthrough #2 ● Client receive

20:45:35 | STANDARD | 500usr | 4116ms

20:45:30 | REACTIVE | 500usr | 2314ms

20:45:24 | REACTIVE | 500usr | 1894ms

20:45:19 | REACTIVE | 500usr | 2578ms

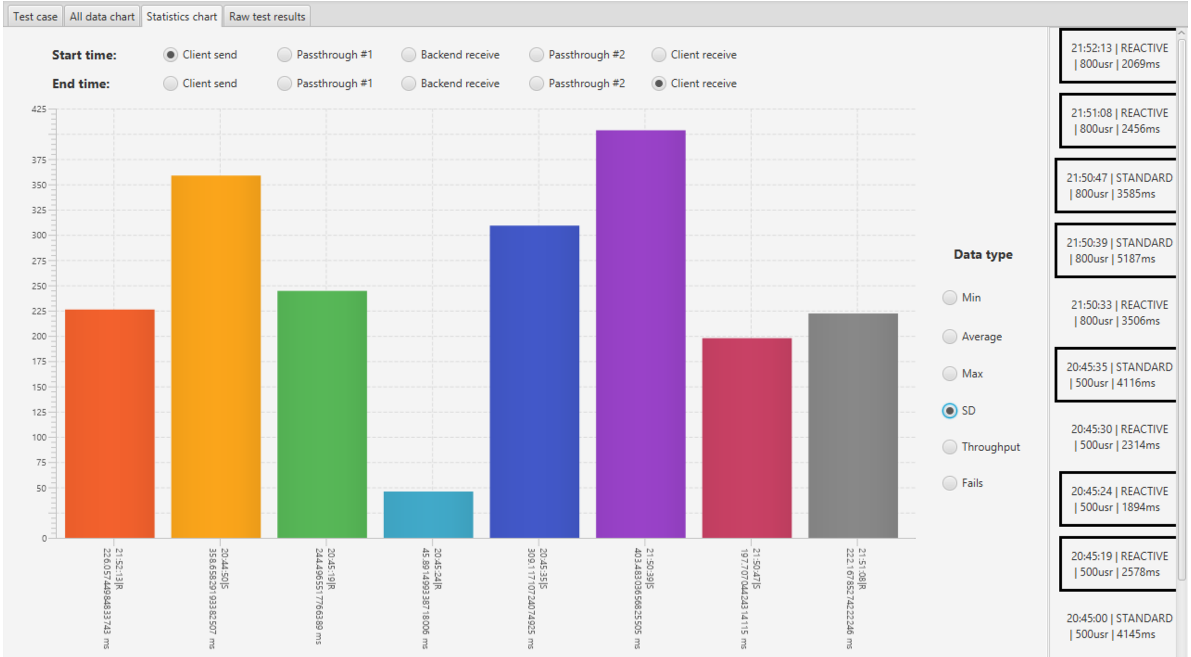
20:45:00 | STANDARD | 500usr | 4145ms

20:44:50 | STANDARD | 500usr | 5262ms

20:44:36 | STANDARD | 500usr | 5484ms

Obr. 9 *All data chart* panel v testovacím klientovi. Vlastní zpracování než doputovala datovým tokem mezi dvěma komponentami zvolenými v horním panelu. Panel je k vidění na obrázku 9.

Statistical chart panel slouží k zobrazení a porovnání statistických a jiných shrnujících hodnot za celý *test case*. Horní a boční panel je totožný jako v případě *All data chart* panelu, přibývá ovšem možnost výběru zobrazené shrnující hodnoty, mezi které patří např. průměr, směrodatná odchylka, průtok apod. Panel je k vidění na obrázku 10.



Start time: ● Client send ○ Passthrough #1 ○ Backend receive ○ Passthrough #2 ○ Client receive

End time: ○ Client send ○ Passthrough #1 ○ Backend receive ○ Passthrough #2 ● Client receive

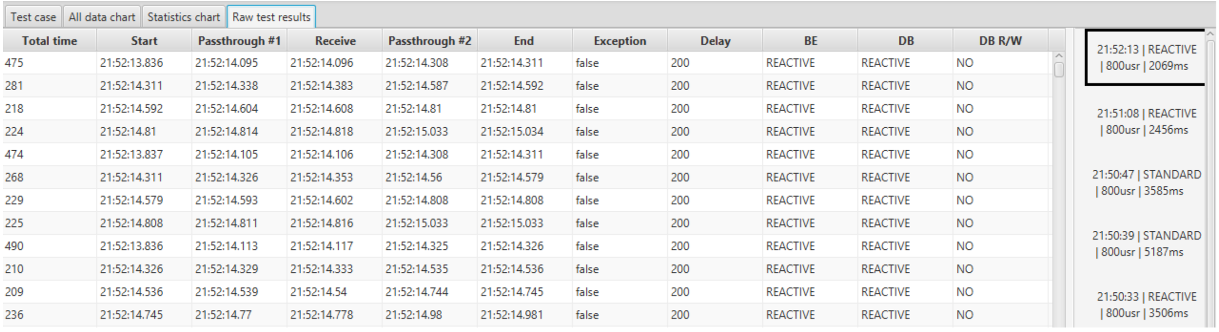
Data type

○ Min
○ Average
○ Max
● SD
○ Throughput
○ Fails

21:52:13 | REACTIVE | 800usr | 2069ms
21:51:08 | REACTIVE | 800usr | 2456ms
21:50:47 | STANDARD | 800usr | 3585ms
21:50:39 | STANDARD | 800usr | 5187ms
21:50:33 | REACTIVE | 800usr | 3506ms
20:45:35 | STANDARD | 500usr | 4116ms
20:45:30 | REACTIVE | 500usr | 2314ms
20:45:24 | REACTIVE | 500usr | 1894ms
20:45:19 | REACTIVE | 500usr | 2578ms
20:45:00 | STANDARD | 500usr | 4145ms

Obr. 10 *Statistical chart* panel v testovacím klientovi. Vlastní zpracování

Poslední, *Raw test results* panel, slouží k zobrazení jednotlivých nezpracovaných hodnot ze zasílaných zpráv v tabulce. Slouží primárně ke kontrole výsledných hodnot, případně k dohledání určité informace skrz řazení těchto hodnot v jednotlivých sloupcích. Ukázka části panelu je vidět na obrázku 11.



Total time	Start	Passthrough #1	Receive	Passthrough #2	End	Exception	Delay	BE	DB	DB R/W
475	21:52:13.836	21:52:14.095	21:52:14.096	21:52:14.308	21:52:14.311	false	200	REACTIVE	REACTIVE	NO
281	21:52:14.311	21:52:14.338	21:52:14.383	21:52:14.587	21:52:14.592	false	200	REACTIVE	REACTIVE	NO
218	21:52:14.592	21:52:14.604	21:52:14.608	21:52:14.81	21:52:14.81	false	200	REACTIVE	REACTIVE	NO
224	21:52:14.81	21:52:14.814	21:52:14.818	21:52:15.033	21:52:15.034	false	200	REACTIVE	REACTIVE	NO
474	21:52:13.837	21:52:14.105	21:52:14.106	21:52:14.308	21:52:14.311	false	200	REACTIVE	REACTIVE	NO
268	21:52:14.311	21:52:14.326	21:52:14.353	21:52:14.56	21:52:14.579	false	200	REACTIVE	REACTIVE	NO
229	21:52:14.579	21:52:14.593	21:52:14.602	21:52:14.808	21:52:14.808	false	200	REACTIVE	REACTIVE	NO
225	21:52:14.808	21:52:14.811	21:52:14.816	21:52:15.033	21:52:15.033	false	200	REACTIVE	REACTIVE	NO
490	21:52:13.836	21:52:14.113	21:52:14.117	21:52:14.325	21:52:14.326	false	200	REACTIVE	REACTIVE	NO
210	21:52:14.326	21:52:14.329	21:52:14.333	21:52:14.535	21:52:14.536	false	200	REACTIVE	REACTIVE	NO
209	21:52:14.536	21:52:14.539	21:52:14.54	21:52:14.744	21:52:14.745	false	200	REACTIVE	REACTIVE	NO
236	21:52:14.745	21:52:14.77	21:52:14.778	21:52:14.98	21:52:14.981	false	200	REACTIVE	REACTIVE	NO

21:52:13 | REACTIVE | 800usr | 2069ms
21:51:08 | REACTIVE | 800usr | 2456ms
21:50:47 | STANDARD | 800usr | 3585ms
21:50:39 | STANDARD | 800usr | 5187ms
21:50:33 | REACTIVE | 800usr | 3506ms

Obr. 11 *Raw test results* panel v testovacím klientovi. Výstřižek. Vlastní zpracování

6 Porovnání reaktivního a imperativního přístupu

V této kapitole bude provedeno ukázkové porovnání reaktivního a klasického imperativního přístupu při tvorbě webových aplikací pomocí benchmarku vytvořeného v kapitole 5. Na konci kapitoly bude navíc prezentován způsob využití benchmarkové aplikace při rozhodování se o vhodnosti využití technologie reaktivního programování na určitých projektech.

6.1 Testovací prostředí

Pro testovací účely budou komponenty *Client, Passthrough* a *Backend + Database* spouštěny na odlišných zařízeních vzájemně komunikujících skrz počítačovou síť.

K testování jsou autorovi k dispozici tato testovací zařízení (TZ):

TZ1

- **Model:** Asus Zenbook UX331UA-EG029T
- **CPU:** Intel Core i7-8550U, 1,8-4,0 GHz, 4C/8T
- **RAM:** 8 GB LPDDR3, 1600 MHz, 1x8 stick
- **Storage:** SSD MICRON 1100, 256GB; R530MB/s, W500MB/s
- **OS:** Windows 10 Home, Ubuntu 19.04
- **Network:** Local network, 100 Mbit/s

TZ2

- **Model:** HP ZBook Studio G5
- **CPU:** Intel Core i7-8750H, 2,2-4,1 GHz; 6C/12T
- **RAM:** 16 GB DDR4; 2666 MHz; 2x8 stick
- **Storage:** SSD Samsung PM981, 512 GB; R3000MB/s, W1800MB/s
- **OS:** Windows 10 Enterprise
- **Network:** Local network, 100 Mbit/s

TZ3

- **Model:** HP ProLiant DL320e Gen8 v2
- **CPU:** Intel Xeon E3-1231 v3, 3,4 GHz, 4C/8T
- **RAM:** 32 GB DDR3, 1600 MHz, 4x8 stick
- **Storage:** HDD HP MB1000GCWCV, 2x1TB, 7200 RPM
- **OS:** Windows Server 2012
- **Network:** Remote network, Prague, 1 Gbit/s

Při případném testování vysokého počtu simultánně připojených uživatelů je na některých operačních systémech potřeba zvýšit maximální počet dynamických portů přidělených pro komunikaci skrz TCP/IP. Pro zvýšení tohoto limitu na platformě Windows bylo postupováno dle pokynů na Microsoft Technet Blogu (55). V případě Linuxové platformy autor nenarazil na problémy v přidělování TCP portů testovacímu klientovi.

6.2 Postupy a zásady testování

Před zahájením samotného testování je třeba stanovit pravidla a postupy při měření výsledků. Benchmarking a obecně testování výkonu je velmi komplexní téma. Je třeba si uvědomit, že žádným měřením nelze získat absolutní číselné vyjádření výkonnosti obou technologií. Každé měření představuje pouze relativní porovnání dvou výsledků pro daný *use-case* s vazbou na dané testovací prostředí. Ale zatímco výsledná čísla se mohou lišit pro různá prostředí a různé testovací scénáře, vždy by měla být zachována paralela mezi výsledky – tedy i když se pro různá měření mohou lišit výsledky v absolutním vyjádření, v relativním porovnání by měly vůči sobě zůstat srovnatelné.

Pro dosažení vypovídajících výsledků, zvláště kvůli práci s relativní metrikou, budou dodržována následující pravidla měření:

- Každá testovací komponenta bude nasazena samostatně na vlastní testovací zařízení (TZ) představené v kapitole 6.1. Výchozí nasazení je *Client* na TZ1, *Passthrough* na TZ2 a *Backend + Database* na TZ3
- Každá sada měření bude provedena několikrát tak, aby se prostřídalo nasazení komponent na testovacích zařízení a byly pokryty všechny kombinace. Na TZ1 bude vždy otestováno na platformě Windows i Linux. Slouží pro ověření nezkreslení výsledků v rámci určitého prostředí.
- Každé měření bude provedeno alespoň 5x před vyřčením závěru. Předpokládá se mírná oscilace kolem určitého finálně prezentovaného výsledku. Slouží k odstranění náhodných ojedinělých odlehlých hodnot.
- Před zahájením měření bude vždy test několikrát spuštěn nanečisto. Slouží pro inicializaci servletů, přiřazení dostatečných prostředků JVM apod.
- Pokud to bude možné, nalézt a uvést zdroj podobného měření provedeného jinou osobou a porovnat výsledky

6.3 Testování

Prvotním cílem veškerého prováděného testování je ověření reaktivních principů a jeho výhod. Mezi hlavními výhodami popsanými v kapitole 3.1.1 byla uvedena vyšší rychlost zpracování více dotazů zároveň a dále efektivnější využití vícejádrových procesorů díky neblokujícímu přístupu.

Pro účely testování jsou k dispozici hlavní 3 nastavitelné proměnné – počet simulovaných uživatelů, počet zpráv zaslaných každým uživatelem a délka simulovaného prodlení *Backend* komponentou. Nastavení každého testu tedy bude uváděno v matici ve stejném pořadí, tedy T(uživatelé, počet dotazů, zpoždění). Hodnota pro délku pauzy mezi dotazy simulovaných uživatelů je pro obecný typ testování nepodstatná a je nastavena na hodnotu 0. Čtení a zápis do databáze není použit, není-li řečeno jinak.

Testování vyšší rychlosti zpracování dotazů

Účel: Účelem prvního testu je dokázat vyšší rychlost reaktivního přístupu při zpracování velkého množství dotazů zároveň.

Postup: Bude provedena sada měření se stále se zvyšujícím počtem simultánně připojených uživatelů zasílajících dotazy.

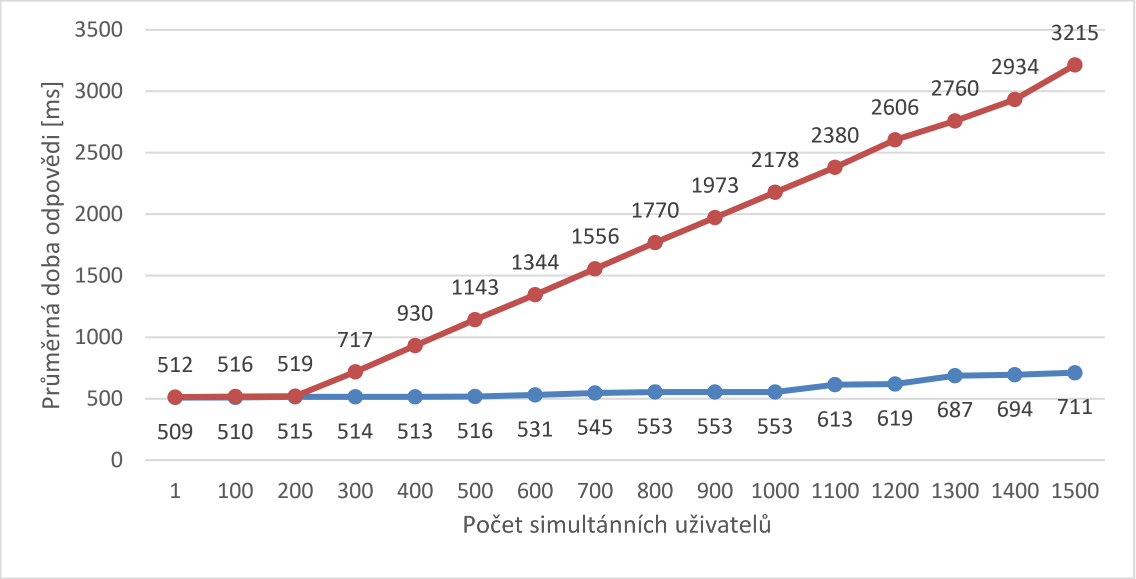
Očekávaný výsledek: Pro nízký počet simulovaných uživatelů bude průměrná délka odpovědi srovnatelná, a až se zvyšujícím se počtem simultánně připojených uživatelů by se měla nějakým způsobem projevit převaha reaktivního přístupu.

Průběh testu:

Jako první musí být zvoleny statické hodnoty pro počet dotazů jednoho uživatele a simulované zpoždění odpovědi. Počet dotazů simulovaného uživatele zvyšuje počet opakování a tím i přesnost testu. Experimentálně byla zjištěna ideální hodnota 3, kdy nižší číslo by způsobilo nepřesný a relativně proměnlivý výsledek a číslo vyšší by zbytečně navýšilo testovací čas, avšak už bez přínosu větší přesnosti. Simulované zpoždění bylo pro tuto sadu měření nastaveno na 500ms.

Počet simulovaných uživatelů začne na čísle 1, v dalším měření se zvýší na 100, dále na 200, 300, atd. až do čísla 1500. Každé jednotlivé měření bude provedeno dle definovaných pravidel a střední hodnota výsledků zanesena do grafu.

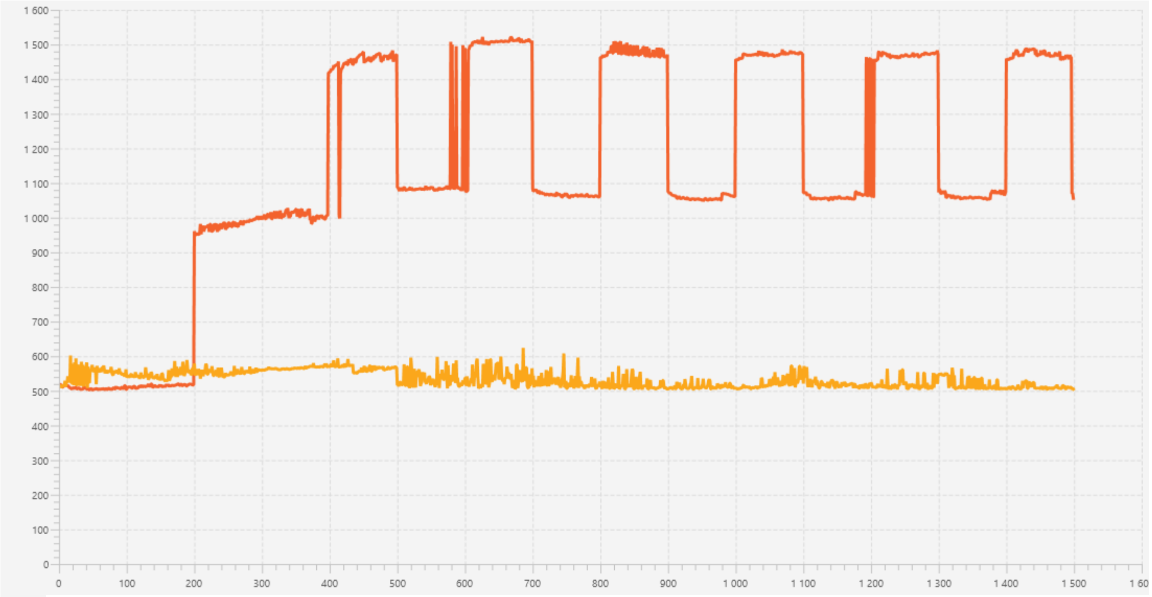
Výsledek sady měření pro T(1-1500, 3, 500) je na obrázku č. 12:



Obr. 12 Graf měření T(1-1500, 3, 500), průměrné hodnoty. Vlastní zpracování

Na grafu lze vidět shodný průběh u hodnot průměrné doby odpovědi až do úrovně přibližně 200 simultánně připojených uživatelů. Od úrovně 200 uživatelů se silně projeví synchronní a blokující povaha klasického přístupu a doba odezvy prudce stoupá, zatímco odezva reaktivních komponent zůstává prakticky neměnná, pouze s nepatrným navýšením ve nejvyšších měřených hodnotách.

K podobným výsledkům měření a shodným závěrům dospěli i autoři benchmarkového měření v (56) a (57).



Obr. 13 Graf měření T(500, 3, 500), všechny hodnoty. Oranžové klasické, žluté reaktivní. Vlastní zpracování

Na obrázku 13 lze vidět zajímavý „schodkovitý“ vývoj zaznamenaný v celé sadě měření, zde konkrétně T(500, 3, 500), na kterém tento jev lze nejsnáze pozorovat.

Vysvětlením úkazu je právě blokující chování klasického přístupu, kdy pro každý příchozí požadavek server vytvoří nové vlákno a na něm požadavek obslouží. A právě těchto vláken, které má server možnost přidělit, je omezené množství. Měřené komponenty využívaly při výchozí hodnotě 200 přidělených vláken, tedy každá klasická komponenta byla schopna simultánně obsloužit maximálně 200 požadavků. Vrchol schodku na grafu pak představuje dobu, kdy zprávy čekaly na obsloužení, tedy na uvolnění vláken, které **byly blokovány** aktuálně obsluhovanými požadavky.

Oproti tomu reaktivní server funguje na principu *Event Loop Group*, který sestává ze statického počtu pracovních vláken (*Worker Threads*). Tato vlákna jsou schopna spolupracovat, sdílet zdroje, předávat si odpovědnost dle potřeby a pracují neustále (ve smyčce - Loop) (58). Jedná se o implementaci neblokujícího přístupu. Počet těchto vláken je stále stejný a výchozí hodnota je shodná, jako počet logických procesorů (tedy 8 pro TZ1 a TZ3, 12 pro TZ2).

Toto také vysvětluje, proč při testování došlo ke zlomu při hodnotě 200 simultánních požadavků, což byl zlomový bod, kdy zprávy musely začít čekat na vyřízení zpráv předchozích. Proto se nabízí otázka, jak by vypadalo porovnání, pokud by klasické komponenty nebyly brzděny počtem přidělených vláken.

Testování brzdění výkonu přidělenými vlákny

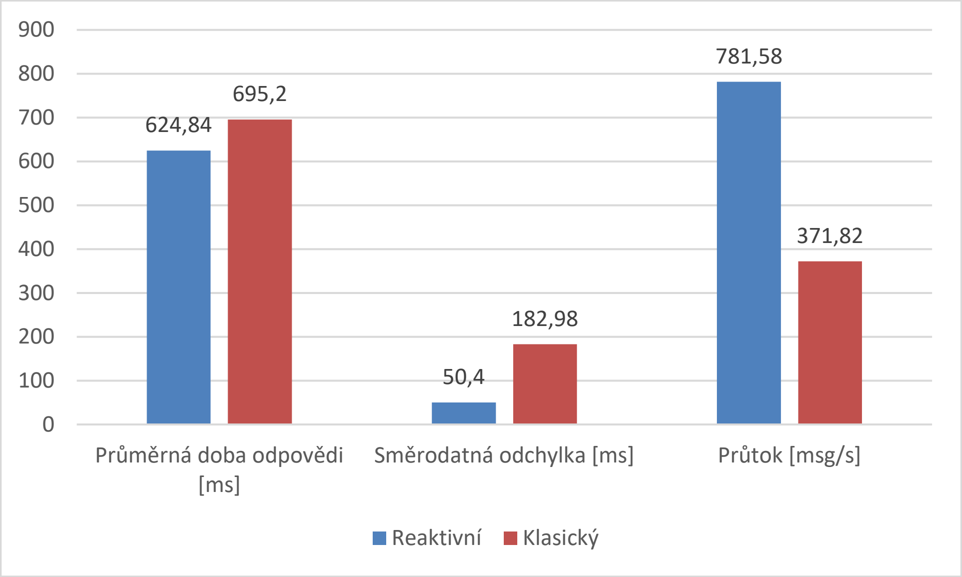
Účel: Účelem druhého testu je potvrdit vyšší rychlost zpracování požadavků reaktivním přístupem, kdy klasický přístup nebude omezen přidělenými vlákny

Postup: Klasickým komponentám bude zvýšen počet přidělených vláken na 10 000, bude proveden test T(600, 3, 500) a výsledky důkladně prozkoumány

Očekávaný výsledek: Klasické komponenty nebudou brzděny přidělenými vlákny a budou vykazovat rychlejší odezvu.

Průběh testu:

Výsledek testu je k dispozici v grafu na obrázku č. 14. Dle očekávání se hodnoty průměrné doby odpovědi značně přiblížily. Při předchozím měření se reaktivní komponenty zdály při tomto testu více než 2x rychlejší a nyní jsou hodnoty velmi srovnatelné.



Obr. 14 Graf měření T(600, 3, 500), statistický výtah. Vlastní zpracování

Samotný průměr trvání odpovědi ale nemusí být vždy známkou vyšší rychlosti. Důkazem je naměřená hodnota průtoku zpráv, kdy reaktivní komponenty obsloužily více než dvojnásobný počet požadavků za stejnou časovou jednotku. Doplňující hodnotou může zároveň být směrodatná odchylka, která ukazuje na značné výkyvy doby odpovědi u klasických komponent, což jde ruku v ruce s nižším průtokem při srovnatelné průměrné době odpovědi.

Je nutné dodat, že v praxi je situace s přidělením 10 000 vláken velmi nepravděpodobná. Většina dnešních webových aplikací využívá databázi a v případě tak velkého množství dotazů by byl vyčerpán *Connection Pool* této databáze, což by znovu vedlo k blokování příchozích zpráv.
Podobným scénářem lze otestovat reaktivní přístup do databáze.

Testování reaktivní databáze

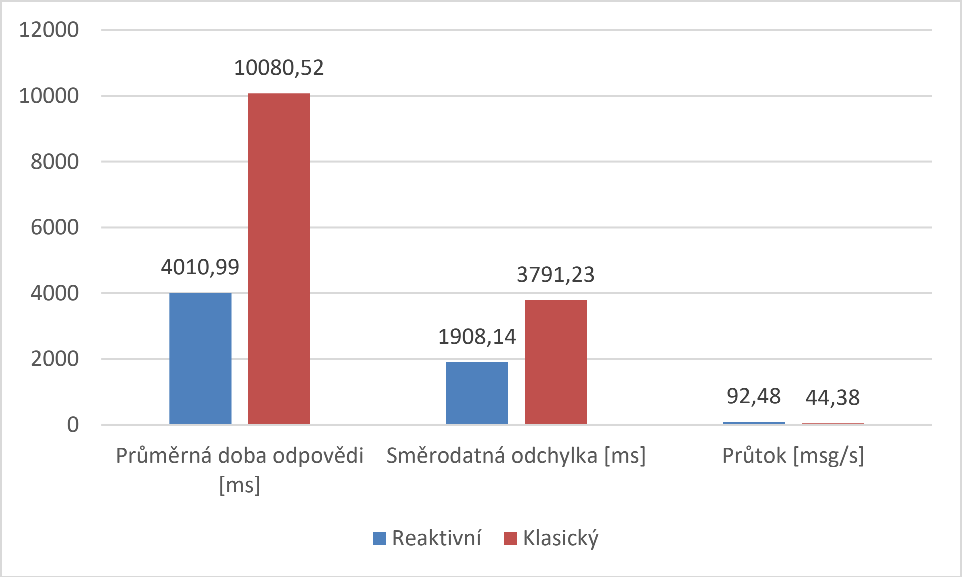
Účel: Účelem databázového testu je porovnat výkonnost obou přístupů při intenzivní komunikaci s databází

Postup: Klasickým komponentám bude zvýšen počet přidělených vláken na 10 000, aby nebyly zprávy blokovány na straně servletu. Bude zaslán dotaz se žádostí o přečtení dat z databáze.

Očekávaný výsledek: Reaktivní databázové ovladače ukážou výhodu asynchronního a neblokujícího přístupu do databáze.

Průběh testu:

Před začátkem testu bylo do databáze vloženo 5000 testovacích záznamů. Poté byl znovu spuštěn test T(500, 3, 500), nově s požadavkem o přečtení dat z databáze. Každý jednotlivý simultánní dotaz si vyžádal přečtení všech 5000 záznamů. Výsledek testu lze vidět na obrázku č. 15.



Obr. 15 Graf měření T(500, 3, 500) + DB W, statistický výtah. Vlastní zpracování

Dle předpokladu došlo u klasické databáze k nahromadění zpráv po vyčerpání *Connection Poolu* – ten měl nastavenou výchozí velikost 100 připojení jak pro klasický, tak pro reaktivní ovladač.

Stejný efekt zaznamenal i autor benchmarkového měření v (59).

Provedenými testy byla úspěšně ověřena vyšší výkonnost reaktivního přístupu při větším množství simultánních dotazů. Stejně tak bylo dokázáno šetření systémovými zdroji a lepší využití možností vícejádrových procesorů, kdy použitých 8 vláken (12 pro TZ2) v *Event Loop Group* módu výkonnostně bezesporu předčilo 600 spuštěných vláken v klasickém módu 1-dotaz-1-vlákno.

6.4 Simulace reálných aplikací

V předchozí kapitole byla dokázána mimo jiné vyšší výkonnost reaktivního přístupu při zpracování více simultánních požadavků. Výraz „více simultánních požadavků“ ovšem představuje problémový fuzzy pojem a vyvstává otázka, v kterých případech se už při vývoji webové aplikace vyplatí použít reaktivní přístup a v kterých by daná aplikace ještě nebyla schopna příliš těžit z výhod, které tato technologie přináší. Bohužel tato informace není obecně vyčíslitelná, záleží na mnoha faktorech, mimo jiné např. na délce zpracování jednotkového dotazu jednotlivými komponentami, na počtu komponent a jejich topologii, počtu přiřazených výpočetních vláken serveru, použité databázi a jejím nastavení apod. Tuto informaci lze však zjistit experimentálně provedením simulace dané aplikace v reaktivní a klasické instanci a porovnáním jejich výkonu. A přesně to je druhým účelem ukázkové aplikace navržené v kapitole 5.

Jednodušší případ nastává, pokud je třeba simulovat již existující aplikaci pro zjištění, jestli by se ji vyplatilo přepsat do reaktivní podoby. V takovém případě není problém statisticky zjistit relevantní maximum počtu simultánně připojených uživatelů a intervaly jejich dotazů z analýzy produkčních logů aplikace. Stejně tak lze jednoduchým testem zjistit trvání zpracování průměrného jednotkového dotazu a tím získat veškeré informace k definování testu v ukázkové aplikaci. Testovací aplikace byla ovšem navržena co nejobecněji a proto je vhodné upravit ji na míru dle aplikace testované. To znamená převážně vystavit přesnou topologii jejích komponent, jedná-li se o distribuovaný systém. Toho lze docílit jednoduchým rozšířením testovací aplikace, která je již v základu připravena na možnost řetězení více *Passthrough* a *Backend* komponent (tyto komponenty sdílí stejnou definici vystaveného i konzumovaného rozhraní). Stejně tak je zde možnost jednoduché výměny typu databáze díky využití *Spring Data Repositories*. Po co nejpřesnějším nastavení simulovaných komponent (počet vláken serveru, connection pool databáze apod.) lze očekávat relativně přesnou simulaci chování klasické a reaktivní verze testované aplikace při zátěži a věrné výsledky srovnání jejich výkonu.

Složitější situace nastává, pokud je třeba simulovat ještě neexistující aplikaci pro účely analýzy. V takovém případě je třeba učinit odhad na počet simultánně připojených uživatelů a trvání zpracování jednotkového dotazu. Ostatní specifikace by již měly být k dispozici z provedené analýzy a návrhu aplikace. Výsledky těchto porovnání budou pouze hrubé odhady, ale i tak mohou podávat důležité informace pro rozhodování. Snadno se totiž i u větších webových aplikací může stát, že ani při nejoptimističtějších odhadech ohledně aktivní uživatelské báze by se příliš neprojevilo zrychlení v případě použití reaktivního přístupu.

6.5 Shrnutí a diskuze výsledků

V úvodních kapitolách bylo uvedeno, že reaktivní programování vzniklo jako odpověď na zvyšující se nároky na webové aplikace z hlediska počtu uživatelů a potřeby ustát nápor jejich dotazů bez výkyvu stability či rychlosti odezvy. Z tohoto důvodu bylo veškeré testování koncipováno jako testování zátěžové a aplikace byla vystavena opravdu extrémním podmínkám. Testování dokázalo, že reaktivní programování svůj požadovaný účel plní opravdu dobře a oproti klasickému přístupu vykázalo v těchto extrémních podmínkách absolutní převahu.

Tyto výsledky ale v žádném případě nenaznačují, že by se reaktivní programování dalo prohlásit za obecně lepší technologii. Naopak první test poukázal na shodné výsledky výkonnosti při nižších hodnotách simultánně dotazujících se uživatelů.

Je třeba si uvědomit, že za reaktivním programováním stojí převážně velké společnosti jako Microsoft a Netflix (viz kapitola 4.1). Netflix, který zastřešil celý vývoj RxJava, tak udělal převážně pro svou stejnojmennou streamovací platformu, která přenáší obrovská množství dat mnoha uživatelům zároveň. Tato kvanta dat se diametrálně liší od počtu dat přenášených nejen průměrnou webovou aplikací, ale i mnohými informačními systémy velkých mezinárodních firem. V těchto aplikacích se pak podmíněný vyšší výkon reaktivního přístupu nemusí vůbec projevit.

Takto by se mohlo zdát, že reaktivní programování je přeci jen lepší technologií a klasický přístup pouze „někdy postačuje“. Proto je třeba znovu zdůraznit negativa spojená s adopcí reaktivního programování, na které bylo poukázáno již v kapitole 5 při implementaci ukázkové aplikace. Jako první lze zmínit častou nepřímočarost až vyšší složitost reaktivního přístupu, který tak má obecně mnohem delší proces učení se dané technologie. Díky nepřímočarosti řešení je navíc vývojový cyklus reaktivních aplikací často delší a testování takové aplikace je mnohem složitější. Zároveň ale význam testů roste, protože u reaktivního programování stále hrozí možnost použití synchronní či blokující operace na nesprávném místě a tím ucpání celého datového toku. Přibývají tak kromě klasických jednotkových testů např. testy správné propagace backpressure v určitých částech aplikace, kontrola neblokujícího průběhu apod. Stejně jako testování je složitější i debugging kvůli zcela asynchronním propagacím objektů, které jsou navíc neustále zapouzdřené v reaktivních obalech.

Dalším omezením reaktivního programování je nutnost mít reaktivní všechny komponenty v rámci datového toku. Pokud byť jen jedna komponenta nebude reaktivní, může dojít k vytváření blokující fronty, jako bylo ukázáno v prvním testu. Tak nastává problém pro projekty, které pracují již se sadou hotových, vzájemně komunikujících aplikací – je třeba přepsat všechny do reaktivní podoby, nemůže zůstat ani jedna klasická, jinak se její blokace datového toku promítne do celého řetězce zpracování.

Nelze tedy ani jednu technologii označit jako horší nebo lepší. Reaktivní i imperativní programování jsou nástroje a jako každý nástroj mají svůj účel, výhody a nevýhody. Tato práce porovnala obě technologie a poskytla informace a metody ke zjištění vhodnosti jejich použití v určitých situacích.

7 Závěr

Reaktivní programování představuje zcela nový přístup k návrhu moderních webových aplikací. Čtenář byl nejprve seznámen s teoretickými principy reaktivního programování a reaktivní architektury. Následně byla popsána implementace těchto principů se zaměřením na programovací jazyk Java. Za pomoci představených technologií byla v další části navržena a implementována ukázková aplikace s využitím reaktivního modulu knihovny Spring 5. V poslední části pak byla tato aplikace použita pro ověření reaktivních principů teoreticky popsaných v části první a bylo provedeno přímé porovnání s přístupem klasickým.

Výstupem práce je důkladné zasvěcení čtenáře do problematiky reaktivního programování. Čtenář by měl být schopen detailně popsat principy reaktivního přístupu, včetně vhodných případů užití, výhod a nevýhod. Dále by měl čtenář získat dostatečné teoretické znalosti specifikace reaktivních standardů pro snadné osvojení si libovolného reaktivního frameworku. V neposlední řadě by měl čtenář být schopen posoudit vhodnost použití reaktivního programování v rámci určitého projektu a svoje stanovisko experimentálně ověřit.

Na informacích obsažených v této práce lze dále stavět při návrhu a konstrukci reaktivních systémů za použití reaktivního programování. Této problematice se práce věnuje pouze zběžně v první, teoretické části. Implementačně se jedná o mnohem komplexnější problematiku, kde reaktivní programování je pouze jednou z mnoha složek v rámci reaktivní architektury.

8 Seznam použitých zdrojů

1. GARTNER SUMMITS. *Gartner Application Architecture, Development & Integration Summit 2014*. [online] 2014. [cit. 1.10.2018] Dostupné z: http://www.gartner.com/imagesrv/summits/docs/apac/application-development/AADI-APAC-2014-Brochure.pdf
2. ORACLE. *JEP 266*: *More Concurrency Updates*. [online] 2015. [cit. 1.10.2018]. Dostupné z: http://openjdk.java.net/jeps/266
3. BONÉR, Jonas; KLANG, Viktor. *Reactive Programming versus Reactive Systems* [online] 2016. [cit. 1.10.2018]. Dostupné z: https://www.lightbend.com/reactive-programming-versus-reactive-systems.
4. OXFORD. *English Oxford Dictionary - asynchronous*. [online] [cit. 15.2.2019]. Dostupné z: https://en.oxforddictionaries.com/definition/asynchronous
5. CARKCI, Matt. *Dataflow and Reactive Programming Systems: A Practical Guide.* CreateSpace Independent Publishing Platform, 2014. ISBN 978-1497422445.
6. GUTHER, Neil. *How to Quantify Scalability: The Universal Scalability Law*. [online] Performance Dynamics, 2007. [cit. 1.10.2018]. Dostupné z: http://www.perfdynamics.com/Manifesto/USLscalability.html
7. ELIOTT, Conal. *Functional Reactive Animation*. [online] 1997. [cit. 1.10.2018]. Dostupné z: http://conal.net/papers/icfp97/icfp97.pdf
8. ELIOTT, Conal. *The Essence of FRP*. [online videozáznam]. Videozáznam z konference BayHac 2015. Kalifornie, USA, 2015 [cit. 1.10.2018]. Dostupné z: https://begriffs.com/posts/2015-07-22-essence-of-frp.html
9. ERICSSON. *Ericsson Mobility Report*. [online] 2018. [cit. 1.10.2018]. Dostupné z: https://www.ericsson.com/assets/local/mobility-report/documents/2018/Ericsson-mobility-report-june-2018.pdf
10. BARTLETT, Joel; GRAY, Jim; BOB, Horst. *Fault Tolerance in Tandem Computer Systems*. [online] 1986. [cit. 1.10.2018]. Dostupné z: http://www.hpl.hp.com/techreports/tandem/TR-86.2.pdf
11. ARMSTRONG, Joe. *Making Reliable Distributed Systems in Presence of Software Errors*. [online] 2003. [Cit. 1.10.2018.]. Dostupné z: http://erlang.org/download/armstrong_thesis_2003.pdf

12. BONÉR, Jonas; FARLEY, Dave; KUHN, Roland; THOMPSON, Martin*. Reactive Manifesto.* [online] 2014. [Cit. 1.10.2018]. Dostupné z: https://www.reactivemanifesto.org/
13. BARNE, Blaise. *Message Passing Interface (MPI)*. [online] 2018. [cit. 1.10.2018]. Dostupné z: https://computing.llnl.gov/tutorials/mpi/
14. FARCIC, Viktor. *The Devops 2.0 Toolkit: Automating the Continuous Deployment Pipeline with Containerized Microservices*. Birmingham: Packt Publishing Limited, 2016. 405 s. ISBN 978-1785289194
15. BROOKS, Frederick. *No Silver Bullet - Essence and Accident in Software Engineering.* [online] 1986. [cit. 1.10.2018]. Dostupné z: http://worrydream.com/refs/Brooks-NoSilverBullet.pdf
16. MCCALISTER, Brian. *Bulkheads*. [online] 2009. [cit. 1.10.2018]. Dostupné z: https://skife.org/architecture/fault-tolerance/2009/12/31/bulkheads.html
17. MALAWSKI, Konrad. *Why Reactive?* Kalifornie: O'Reilly Media, 2016. 38 s. ISBN 978-1-491-96157-5
18. REACTIVEX. *RX Languages*. [online] [cit. 30.1.2019]. Dostupné z: http://reactivex.io/languages.html
19. TIOBE. *Tiobe Index*. [online] [cit. 30.1.2019]. Dostupné z: https://www.tiobe.com/tiobe-index/
20. PIVOTAL. *Spring Framework Reactive Stack Reference Guide*. [online] [cit. 30.1.2019]. Dostupné z: https://docs.spring.io/spring/docs/current/spring-framework-reference/web-reactive.html
21. COLDEWEY, Jens. *Decoupling of Object-Oriented Systems*. [online] 2000 [Cit. 30.1.2019]. Dostupné z: http://www.coldewey.com/publikationen/Decoupling.1.1.PDF
22. SOMASEGAR, S. *Reactive Extensions for .NET (Rx)*. [online] 18.11. 2009. [cit. 30.1.2019]. Dostupné z: https://devblogs.microsoft.com/somasegar/reactive-extensions-for-net-rx/
23. LIGHTBEND. *Akka Github Repository*. [online] [cit. 30.1.2019]. Dostupné z: https://github.com/akka/akka
24. CALDATO, Claudio. *MS Open Tech Open Sources Rx (Reactive Extensions) – a Cure for Asynchronous Data Streams in Cloud Programming*. [Online]

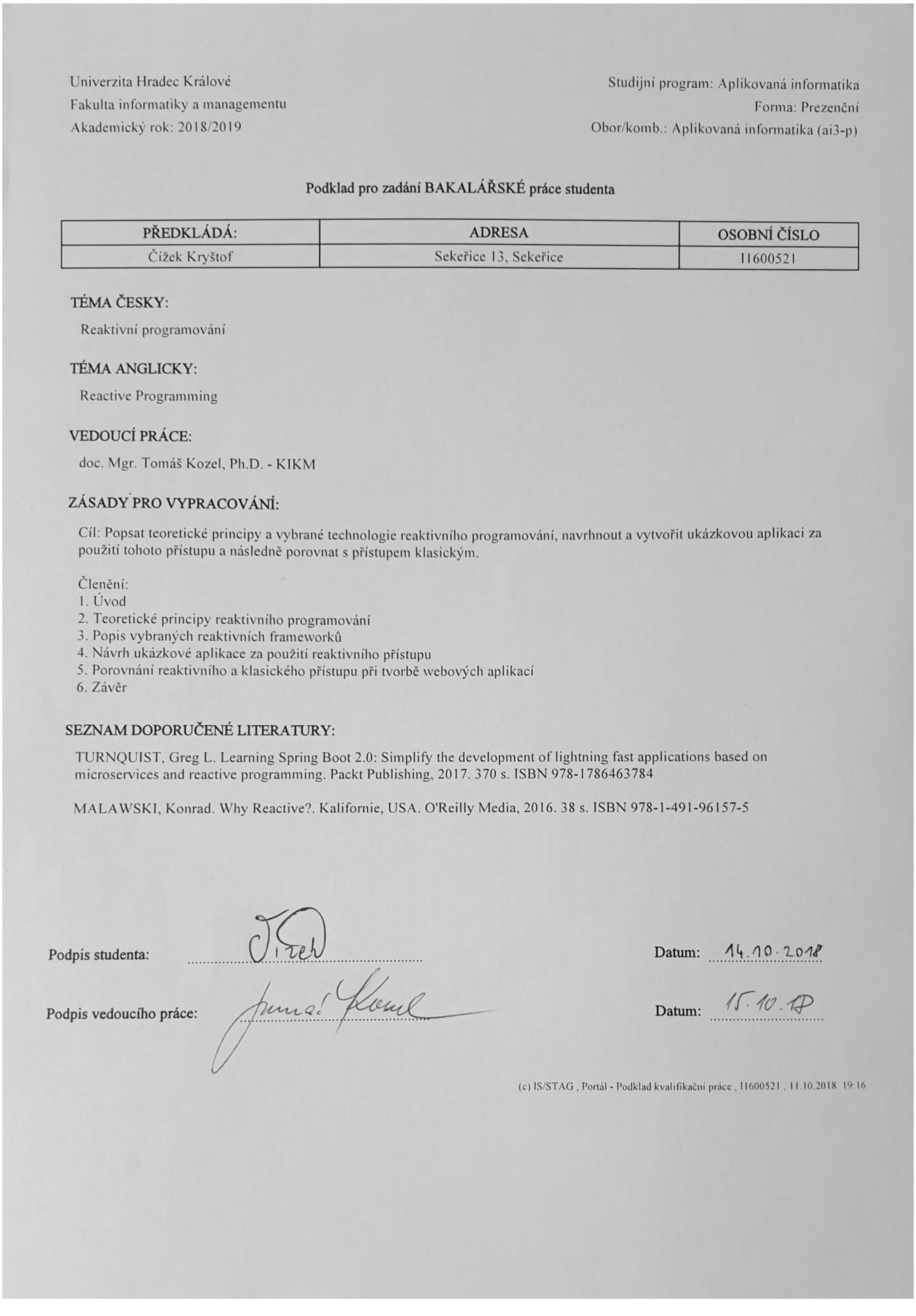
microsoft.com, 6.12.2012. [cit. 30.1.2019.]. Dostupné z: https://blogs.msdn.microsoft.com/interoperability/2012/11/06/ms-open-tech-open-sources-rx-reactive-extensions-a-cure-for-asynchronous-data-streams-in-cloud-programming/

25. NETFLIX. *RxJava Github Repository*. [Online] [cit. 30.1.2019]. Dostupné z: https://github.com/ReactiveX/RxJava
26. MEIJER, Erik. *Foreword for Reactive Programming with RxJava by Ben Christensen, Tomasz Nurkiewicz*. [online] [cit. 30.1.2019]. Dostupné z: https://www.oreilly.com/library/view/reactive-programming-with/9781491931646/foreword01.html
27. BRISBIN, Jon. *Reactor - a foundation for asynchronous applications on the JVM*. [online] 13.5.2013. [cit. 30.1.2019]. Dostupné z: https://spring.io/blog/2013/05/13/reactor-a-foundation-for-asynchronous-applications-on-the-jvm
28. PIVOTAL. *Pivotal Open Source*. [Online] [cit. 30.1.2019]. Dostupné z: https://pivotal.io/open-source
29. LIGHTBEND; PIVOTAL, NETFLIX, a další. *Reactive Streams*. [online] [cit. 30. 1.2019]. Dostupné z: http://www.reactive-streams.org/
30. HOELLER, Juergen*. Spring Framework 5.0 goes GA*. [Online] 28.9.2017. [Cit. 30. 1 2019.]. Dostupné z: https://spring.io/blog/2017/09/28/spring-framework-5-0-goes-ga
31. KARNOK, David. *Operator-fusion (Part 1).* [online] 11. 3 2016. [cit. 30.1.2019]. Dostupné z: http://akarnokd.blogspot.com/2016/03/operator-fusion-part-1.html
32. PIVOTAL. *Project Reactor*. [online] [cit. 30.1.2019.]. Dostupné z: https://projectreactor.io/
33. KARNOK, David. *Writing operators for 2.0.* [Online] 4.5.2018 [cit. 30. 1 2019]. Dostupné z: https://github.com/ReactiveX/RxJava/wiki/Writing-operators-for-2.0
34. REACTIVEX. *Implementing Your Own Operators*. [Online] [cit. 30.1.2019]. Dostupné z: http://reactivex.io/documentation/implement-operator.html

35. LIGHTBEND. *Basics and working with Flows - Operator Fusion*. [Online] [cit. 30.1.2019]. Dostupné z: https://doc.akka.io/docs/akka/2.5.3/java/stream/stream-flows-and-basics.html#operator-fusion
36. KLANG, Viktor. *Reactive Streams 1.0.0 interview*. [Online] 1.6.2015. [cit. 30.1.2019]. Dostupné z: https://medium.com/@viktorklang/reactive-streams-1-0-0-interview-faaca2c00bec
37. LIGHTBEND; NETFLIX; PIVOTAL; a další. *Reactive Streams JVM Github Repository*. [online] [cit. 15.2.2019]. Dostupné z: https://github.com/reactive-streams/reactive-streams-jvm
38. LIGHTBEND; NETFLIX; PIVOTAL; a další. *Reactive Streams 1.0.0 is here!* [online] [cit. 15.2.2019]. Dostupné z: http://www.reactive-streams.org/announce-1.0.0
39. LIGHTBEND; NETFLIX; PIVOTAL; a další. *Reactive Streams .NET Github Repository*. [online] [cit. 15.2.2019]. Dostupné z: https://github.com/reactive-streams/reactive-streams-dotnet
40. LIGHTBEND; NETFLIX; PIVOTAL; a další. *Reactive Streams JS Github Repository*. [online] [cit. 15.2.2019]. Dostupné z: https://github.com/reactive-streams/reactive-streams-js
41. HUGHES, Andrew. *Build a Reactive App with Spring Boot and MongoDB .* [Online] 21.2.2019. [cit. 25. 3 2019.]. Dostupné z: https://developer.okta.com/blog/2019/02/21/reactive-with-spring-boot-mongodb
42. NETFLIX. *RxJava Javadoc*. [Online] [cit. 25.3.2019.]. Dostupné z: http://reactivex.io/RxJava/2.x/javadoc/
43. PIVOTAL. *Project Reactor Javadoc*. [online] [cit. 25.3.2019]. Dostupné z: https://projectreactor.io/docs/core/release/api/
44. KARNOK, David. *Reactor vs RxJava Tweet.* [Online] 9.10.2016. [cit. 25.3.2019]. Dostupné z: https://twitter.com/akarnokd/status/774590596740685824
45. KŘENECKÝ, Robin. *Mobile applicationdevelopment using theReactiveX framework*. [Online] 2018. [cit. 25.3.2019]. Dostupné z: https://is.muni.cz/th/o4q2v/bachelors_krenecky.pdf
46. LIGHTBEND. *Akka*. [online] akka.io. [cit. 25.3.2019] https://akka.io/.

47. LIGHTBEND. *Akka Documentation: Why modern systems need a new programming model*. [online] [cit. 25. 3 2019]. Dostupné z: https://doc.akka.io/docs/akka/current/guide/actors-motivation.html
48. PIVOTAL. *R2DBC*. [online] [cit. 25.3.2019]. Dostupné z: https://r2dbc.io/
49. PIVOTAL. *Spring*. [online] [cit. 25.3.2019]. Dostupné z: http://www.spring.io
50. FOX, Tim. *VertX.* [online] [cit. 25.3.2019]. Dostupné z: http://vertx.io/
51. RATPACK. *Ratpack Framework*. [online] [cit. 25. 3 2019.]. Dostupné z: https://ratpack.io/
52. PIVOTAL. *Spring Framework Javadoc*. [online] [cit. 25. 3 2019.]. Dostupné z: https://docs.spring.io/spring-framework/docs/current/javadoc-api/
53. MOSMANN, Michael. *Embedded MongoDB Github Repository*. [Online] [cit. 10.4.2019]. Dostupné z: https://github.com/flapdoodle-oss/de.flapdoodle.embed.mongo
54. ORACLE. *JavaFX*. [online] [cit. 10.4.2019.]. Dostupné z: https://openjfx.io.
55. TRISTAN, K. *MaxUserPort – what it is, what it does, when it's important.* [Online] 11.3.2008. [cit. 10. 4 2019]. Dostupné z: https://blogs.technet.microsoft.com/tristank/2008/03/11/maxuserport-what-it-is-what-it-does-when-its-important/
56. CUNHA, Goncalo Trincao. *Reactive vs. Synchronous Performance Test with Spring Boot 2.0*. [online] 21. 2 2018. [cit. 10.4.2019.]. Dostupné z: https://dzone.com/articles/spring-boot-20-webflux-reactive-performance-test
57. SAXENA, Raj. *SpringBoot 2 performance—servlet stack vs WebFlux reactive stack*. [Online] 5.3.2018. [cit. 10. 4 2019.]. Dostupné z: https://medium.com/@the.raj.saxena/springboot-2-performance-servlet-stack-vs-webflux-reactive-stack-528ad5e9dadc
58. MALDINI, Stephane A Georgieva, Violeta. *Reactor Netty Reference Guide*. [online] [cit. 10.4.2019]. Dostupné z: https://projectreactor.io/docs/netty/snapshot/reference/index.html.
59. SHACKMANN, Rene. *Reactive Java Performance Comparison.* [online] 24.5.2017. [cit. 10.4.2019]. Dostupné z: https://tech.willhaben.at/reactive-java-performance-comparison-c4d248c8d21f

9 Seznam obrázků



Univerzita Hradec Králové
Fakulta informatiky a managementu
Akademický rok: 2018/2019

Studijní program: Aplikovaná informatika
Forma: Prezenční
Obor/komb.: Aplikovaná informatika (ai3-p)

Podklad pro zadání BAKALÁŘSKÉ práce studenta

PŘEDKLÁDÁ:	ADRESA	OSOBNÍ ČÍSLO
Čížek Kryštof	Sekeřice 13, Sekeřice	I1600521

TÉMA ČESKY:

Reaktivní programování

TÉMA ANGLICKY:

Reactive Programming

VEDOUCÍ PRÁCE:

doc. Mgr. Tomáš Kozel, Ph.D. - KIKM

ZÁSADY PRO VYPRACOVÁNÍ:

Cíl: Popsat teoretické principy a vybrané technologie reaktivního programování, navrhnout a vytvořit ukázkovou aplikaci za použití tohoto přístupu a následně porovnat s přístupem klasickým.

Členění:
1. Úvod
2. Teoretické principy reaktivního programování
3. Popis vybraných reaktivních frameworků
4. Návrh ukázkové aplikace za použití reaktivního přístupu
5. Porovnání reaktivního a klasického přístupu při tvorbě webových aplikací
6. Závěr

SEZNAM DOPORUČENÉ LITERATURY:

TURNQUIST, Greg L. Learning Spring Boot 2.0: Simplify the development of lightning fast applications based on microservices and reactive programming. Packt Publishing, 2017. 370 s. ISBN 978-1786463784

MALAWSKI, Konrad. Why Reactive?. Kalifornie, USA. O'Reilly Media, 2016. 38 s. ISBN 978-1-491-96157-5

Podpis studenta: .... **Datum:** 14.10.2018

Podpis vedoucího práce: .... **Datum:** 15.10.18

(c) IS/STAG , Portál - Podklad kvalifikační práce , I1600521 , 11.10.2018 19:16