Vysoká škola ekonomická v Praze

Fakulta informatiky a statistiky

Návrh a tvorba open-source WebSocket server frameworku v PHP

BAKALÁŘSKÁ PRÁCE

ve studijním programu Aplikovaná informatika

Autor: Marek Frydrych

Vedoucí bakalářské práce: Ing. Rudolf Pecinovský, CSc.

Praha, květen 2018

Prohlášení:

Prohlašuji, že jsem bakalářskou práci Návrh a tvorba open-source WebSocket server frameworku v PHP vypracoval samostatně za použití v práci uvedených pramenů a literatury.

V Praze dne 30. dubna 2018 . . . . . .

Marek Frydrych

Poděkování

Rád bych poděkoval Ing. Rudolfu Pecinovskému, CSc. za vedení práce a cenné praktické rady.

Abstrakt

Tato práce se věnuje návrhu a vytvoření open-source frameworku pro tvorbu škálovatelných WebSocket serverů v programovacím jazyce PHP. Pro vysvětlení motivace pro jeho tvorbu bude nejprve provedena analýza současného stavu webových programovacích jazyků a také konkrétních řešení pro vývoj WebSocket serverů v jazyce PHP. Následně bude vysvětleno, proč by se měla využívat technologie WebSocket místo jiných obvyklých způsobů dosažení komunikace klientů se serverem v reálném čase, jako například AJAX polling či Long polling. Poté bude vysvětleno, jaké omezení vyplývají z použití jazyka PHP pro vývoj WebSocket serverů. Na základě získaných informací bude následně navržena struktura samotného frameworku, která bude poskytovat vývojářům jednoduchý a efektivní způsob tvorby WebSocket serverů. Po vytvoření struktury budou vybrány open-source komponenty, které poslouží k rychlejšímu vývoji samotného frameworku. Na závěr bude v práci zahrnuta příručka pro programátory, která na příkladu jednoduché chatovací aplikace vysvětlí strukturu frameworku, programování aplikační logiky a spuštění více instancí naprogramovaného WebSocket serveru.

Klíčová slova

WebSocket, PHP, framework, webové aplikace pracující v reálném čase, otevřený software

Abstract

This thesis is dedicated to designing and programming of open-source framework aimed for creating scalable WebSocket servers in PHP programming language. Analysis of current state of website programming languages as well as analysis of existing solutions for programming WebSocket servers in PHP will be made to support author's motivation for creating this thesis. After that, advantages of WebSockets over other alternative solutions for real-time communication between clients and server like AJAX polling or Long polling will be described. Summary of restrictions of PHP in usage for programming WebSocket servers will follow. Structure of the framework will be designed using gathered informations. Design should be easy to use and effective for creating WebSocket servers. Some available open-source components will be picked for programming of the framework. Handbook for programmers at the end will describe structure of the framework as well as programming application logic in it and running multiple instances of the server. Simple chat application will be used as example in this handbook.

Keywords

WebSocket, PHP, framework, real-time web applications, open-source software

Obsah

Obsah 6

Seznam tabulek 9

Seznam výpisů programů 10

Úvod 11

Cíle práce 11
Cílová skupina 11
Použité metody 12
Struktura práce 12

1 Komentovaná rešerše informačních zdrojů 14

1.1 Publikace a závěrečné práce 14
1.2 Internetové zdroje 15

2 Současný stav řešené problematiky 16

2.1 Podíly serverových programovacích jazyků na webových stránkách 16
2.2 Vhodnost PHP pro WebSocket servery 17
2.3 Řešení pro komunikaci PHP serveru s klientem v reálném čase 17
2.3.1 ReactPHP 18
2.3.2 Amp 18
2.3.3 Workerman 19
2.3.4 Kraken 19
2.4 Řešení zaměřené na WebSocket servery 19
2.4.1 Ratchet 19
2.4.2 Aerys 19
2.4.3 Kraken Network 20

3 Přínosy a problémy při nasazování PHP WebSocket serveru 21

3.1 Alternativy WebSocket serverů 21
3.1.1 AJAX polling 21
3.1.2 Long polling 21
3.1.3 HTML5 server sent events 22
3.2 Přínosy WebSocketů 22
3.3 Problémy při nasazení PHP WebSocket serveru 22
3.3.1 Asynchronní PHP 23
3.3.2 Omezení otevřených spojení 23
3.4 Škálování WebSocket serverů 24

4 Návrh struktury frameworku a komponent 26
4.1 Životní cyklus 26
4.1.1 Jedna třída s dvěma zpracovatelskými metodami 26
4.1.2 Dvě třídy po jedné zpracovatelské metodě 27
4.1.3 Volba vhodné možnosti a její zdůvodnění 27
4.1.4 Jmenné konvence tříd 27
4.2 Spouštěcí skript 28
4.2.1 Konfigurační soubory 29
4.2.2 Hlavní třída frameworku 30
4.3 Routování 30
4.4 Message broker 31
4.5 Dependency Injection 31
4.5.1 Způsoby využití vzoru dependency injection 31
4.5.2 Dependency injection kontejner 32

5 Výběr komponent pro framework 33
5.1 Verze PHP 33
5.2 Zpracování WebSocket zpráv 33
5.3 Implementace message brokeru 34
5.4 Zpracování konfiguračních souborů 34

6 Uživatelská příručka 35
6.1 Instalace 35
6.2 Struktura 36
6.2.1 ConnectionManager 36
6.2.2 Receiver 36
6.2.3 Handler 37
6.2.4 Message broker 37
6.3 Spouštěcí skript 37
6.4 Konfigurační soubory 38
6.4.1 Ukázkový konfigurační soubor 38
6.5 Dependency Injection 39
6.5.1 Factories 39
6.6 Ukázková aplikační logika 40
6.6.1 ChatConnectionManager 40
6.6.2 ChatReceiver 41
6.6.3 ChatHandler 43
6.6.4 Implementace message brokeru 43
6.6.5 Autoloading 45
6.7 Spuštění a otestování funkčnosti serveru 46
6.7.1 Spuštění dvou instancí serveru a nastavení škálování 46
6.7.2 Testovací JavaScript klient 47

Závěr **48**
Dosažení vytyčených dílů **48**
Přínos práce **48**
Terminologický slovník **49**
Použité informační zdroje **50**
Rejstřík **52**

Seznam tabulek

Tabulka 1: Statistika využití serverových programovacích jazyků na webových stránkách. Údaje jsou vždy platné k 1. 1. daného roku. Jazyky jsou seřazeny od nejvyššího podílu k nejmenšímu. Jedna webová stránka může využívat více programovacích jazyků. (Zdroj: W3techs 2018a) 17

Tabulka 2: Statistiky jednotlivých knihoven napsaných v jazyce PHP pro využití v aplikacích fungujících v reálném čase (Zdroj: Packagist 2018abcd) 18

Seznam výpisů programů

Výpis 1: Porovnání klasického synchronního blokujícího kódu a asynchronního kódu za využití react/promises.....23
Výpis 2: Ukázka minimální konfigurace frameworku obsahující všechny povinné parametry ve formátu Neon.....30
Výpis 3: Ukázkový spouštěcí skript.....37
Výpis 4: Ukázkový konfigurační soubor se všemi možnými parametry ve formátu Neon.....38
Výpis 5: Ukázková implementace továrny pro dependency injection.....40
Výpis 6: Ukázková implementace rozhraní IConnectionManager.....41
Výpis 7: Ukázková implementace rozhraní IReceiver.....42
Výpis 8: Ukázková implementace rozhraní IHandler.....43
Výpis 9: Ukázková implementace message brokeru.....44
Výpis 10: JavaScriptový kód pro otestování funkčnosti dvou běžících instancí WebSocket serveru.....47

Úvod

Mou hlavní motivací k vytvoření této práce je současný nedostatek jednoduchých řešení pro tvorbu WebSocket serverů v jazyce PHP. Kromě toho, že je těchto řešení velmi málo, mají také velmi omezené funkcionality, což dle mého vede ke zbytečně složitému vývoji takovýchto serverů. Důsledkem toho je, že se stále velmi často setkávám s využíváním starších řešení komunikace webových aplikací se serverem, především pak s AJAX pollingem.

Jelikož je PHP využíváno na většině webových stránek, tak si myslím, že jednoduché otevřené řešení může nabídnout programátorům rychlý a efektivní způsob, jak vyvíjet WebSocket servery v PHP.

K nápadu na vytvoření frameworku pro tvorbu WebSocket serverů mě vedla nedávná vlastní zkušenost. Při vývoji jedné aplikace jsem zjistil, že pro určitou funkcionalitu potřebuji mít možnost komunikovat mezi klienty a serverem v reálném čase a pro tento účel byly jiné alternativy naprosto nevhodné. Rozhodl jsem se tedy pro tvorbu mého prvního WebSocket serveru. Několik dní jsem ovšem strávil jen tím, že jsem metodou pokus-omyl přicházel na řešení různých problémů, především pak škálování. Po zhruba týdnu jsem se konečně dostal do fáze, kdy vše potřebné fungovalo tak, jak mělo, a já se mohl pustit do samotné aplikační logiky, která mi následně zabrala pouhý zlomek času. Tato zkušenost mě utvrdila v tom, že v open-source komunitě naprosto chybí nějaký framework, ve kterém by vývojáři mohli napsat a zprovoznit jednoduchý WebSocket server během několika málo hodin a šetřil by tak jejich čas.

Cíle práce

Hlavním cílem práce je vytvořit open-source framework, který bude sloužit pro tvorbu škálovatelných WebSocket serverů v PHP. K dosažení mi pomohou následující dílčí cíle:

- Zdůvodnit vhodnost PHP pro tvorbu WebSocket serveru
- Analyzovat současný stav otevřených řešení pro technologii WebSocket v jazyce PHP
- Vysvětlit přínosy a problémy nasazení PHP WebSocket řešení na webové stránky
- Navrhnout a vytvořit vlastní škálovatelné řešení WebSocket server frameworku v PHP
- Zvolit vhodné komponenty pro tvorbu vlastního škálovatelného řešení WebSocket server frameworku
- Napsat uživatelskou příručku, která by programátorům vysvětlila použití frameworku

Cílová skupina

Cílovou skupinou této práce jsou programátoři, kteří primárně využívají jazyk PHP, a to za použití plně objektového přístupu. Ti se v práci dozvědí, jaké jsou problémy nasazení WebSocket řešení na webové stránky a také jaké výhody jim toto řešení nabídne. Dále se dozví, jaké komponenty mohou

využít při tvorbě vlastního řešení, případně jak používat mnou vytvořené řešení, které bude v práci popsáno.

Dále z práce mohou čerpat zaměstnanci firem na především manažerských pozicích, kteří určují, jaké technologie bude jejich firma či skupina zaměstnanců využívat. Pro ně je důležitá zejména kapitola o přínosech WebSocket řešení.

Použité metody

Abych mohl splnit hlavní cíl této práce, musím splnit několik dílčích úkolů. V první části práce především analyzuji současný stav webových programovacích jazyků a řešení pro tvorbu WebSocket serverů v PHP. Díky této analýze vím, jaké nedostatky mají současná řešení. Na základě této analýzy pak úvahovým stylem navrhuji strukturu a vytvářím vlastní framework pro tvorbu škálovatelných WebSocket serverů v PHP, který zjištěné nedostatky eliminuje. Po vytvoření frameworku jsem napsal příručku pro jeho používání.

Struktura práce

V jednotlivých kapitolách se postupně propracuji od důvodů, proč framework vytvářím, přes teoretický popis konceptů WebSocketů, které jsou podstatné pro další části práce, až po samotnou tvorbu frameworku.

V úvodu popisuji, co mě vedlo ke zpracování tohoto tématu. Stanovuji si zde také cíle, kterých se v následujících kapitolách snažím dosáhnout.

První kapitolou je komentovaná rešerše zdrojů, které by mohly být pro čtenáře této práce užitečné pro doplnění informací z této práce, jelikož se v ní zabývám především konkrétním řešením WebSocket server frameworku a jen zjednodušeně vysvětluji všechny podstatné náležitosti komunikace serveru s webovou aplikací v reálném čase.

Ve druhé kapitole se věnuji analýze využití programovacích jazyků na webových stránkách a na základě této analýzy pak vysvětluji, proč je PHP ideální jazyk pro tvorbu WebSocket serverů. Dále v kapitole popisuji již existující řešení, pomocí kterých lze v PHP naprogramovat WebSocket server a jejich nevýhody, které mě vedly k vytvoření vlastního frameworku.

Třetí kapitola je věnována porovnání různých způsobů dosažení komunikace mezi serverem a webovou aplikací v reálném čase. Jde především o shrnutí informací z jiných prací a publikací zmíněných v druhé kapitole, které se tomuto srovnání věnují více do hloubky. Dále v kapitole také popisuji možné problémy při tvorbě WebSocket serveru v PHP, které je pro toto využití poměrně specifické. Nakonec popisuji problém škálovatelnosti, která je na rozdíl od klasických webových stránek složitější.

Čtvrtou kapitolu věnuji popisu navrhované struktury frameworku. Ta musí být taková, aby používání frameworku bylo co nejjednodušší pro programátory začínající s technologií WebSocket, ale zároveň neomezovala expertní programátory, kteří potřebují upravit i ty nejmenší detaily.

V páté kapitole popisuji výběr komponent pro můj framework. Jelikož jde o otevřené řešení, tak považuji za vhodné použít již existující řešení, které mi umožní urychlit vývoj frameworku a zároveň mi tím, že jsou již používány dalšími uživateli, dávají jakousi garanci, že jsou kvalitní a s minimem chyb.

V šesté kapitole je pak obsažena jednoduchá příručka pro programátory. Ta na praktických příkladech ukazuje možnosti využití frameworku, který navrhuji ve čtvrté kapitole. V příručce také popisuji, jak již vytvořený server spustit a také je nastíněno, jaké jsou možnosti dosažení jednoduchého škálování.

Poslední kapitolou je závěr, kde shrnuji, zda se mi podařilo dosáhnout vytyčených cílů z úvodu a jaký je celkový přínos této práce.

1 Komentovaná rešerše informačních zdrojů

V práci čerpám z různorodých zdrojů, které pojednávají především o technologii WebSocket, případně obecněji o webové komunikaci v reálném čase. Především o spojení WebSocketů a PHP nejsou téměř žádné publikace, což pro mě ale není překvapivé a důvody, proč tomu tak je, zmíním v příští kapitole. V této kapitole shrnu několik nejdůležitějších zdrojů, ze kterých v práci čerpám či které můžou být pro čtenáře zajímavé.

1.1 Publikace a závěrečné práce

Bakalářská práce *Real-time webové aplikace* (Toman 2012) je ideálním začátkem pro ty, kdo se ještě s technologií WebSocket a případně s komunikací webových aplikací se serverem v reálném čase nikdy nesetkali. V práci je popsán princip WebSocketů, jejich výhody oproti ostatním řešením, výčet různých možností jejich využití a také seznam technologií, na kterých lze stavět WebSocket server. Tento seznam je však vzhledem k relativnímu stáří práce neaktuální. Ve světě programování je 6 let opravdu dlouhá doba, během které se objevila spousta nových a lepších možností, a to především v oblasti webů.

Anglicky psaná kniha *Realtime Web Apps* (Lengstorf; Leggetter; Newman 2013) vysvětluje historii a současnost webových aplikací v reálném čase. Dále se zabývá procesem tvorby nové webové aplikace. Autoři nejprve vysvětlují, proč jsou webové aplikace pro většinu použití lepší než aplikace nativní. Následně podrobně popisují celý průběh tvorby webové aplikace, a to za použití technologií jako HTML, CSS, JavaScript, PHP a služby Pusher. Je vhodná pro ty, kteří chtějí projít celým procesem tvorby webové aplikace komunikující se serverem v reálném čase.

The Definitve Guide to HTML5 WebSocket (Wang; Salim; Moskovits 2013) je již velmi podrobná anglicky psaná publikace zaměřující se na WebSockety. Popisuje do detailu, jak komunikace skrz ně probíhá a také obsahuje informace o zabezpečení takové komunikace. Osobně bych ji doporučil těm, kteří chtěji WebSockety znát více do hloubky, ale pro jejich běžné využití není potřeba.

Další zajímavou závěrečnou prací z půdy Vysoké školy ekonomické je *Porovnanie efektivity technológií WebSocket a AJAX pri implementácii kolaboratívneho editoru* (Augustín 2013). Podobně jako první zmíněná práce se autor zabýval popisem technologií pro komunikaci webové aplikace se serverem v reálném čase. Hlavní přínos práce je praktické porovnání výkonnosti WebSocket serverů a AJAX pollingu.

Poslední prací, která se zabývá WebSockety je *WebSockets - technologie pro tvorbu real time webových aplikací* (Bulvas 2012). Autor v práci popsal mimo jiné praktické vytvoření jednoduchého serveru v PHP a s ním komunikujícího klienta v JavaScriptu. Také zde jsou popsány specifikace jednotlivých verzí WebSocket protokolu.

Dále v práci čerpám z knihy *Návrhové vzory v PHP* (Böhmer 2012), ve které jsou na praktických příkladech v PHP vysvětleny různé programovací návrhové vzory. Některé z nich využívám při návrhu frameworku.

Poslední podstatnou knihou, ze které čerpám, je *The SOLID Principles* (Arora 2015). V té jsou popsány tzv. SOLID principy, které je vhodné při programování používat, jelikož vedou k lépe strukturovanému kódu, který je jednodušeji čitelný.

1.2 Internetové zdroje

Za důležité považuji i zmínění podstatných internetových zdrojů. Tím nejdůležitějším je *specifikace WebSocket protokolu RFC 6455* (Internet Engineering Task Force 2011). Přečtení doporučuji těm, kterým nestačí výše zmíněné publikace, které tuto specifikaci shrnují do čitelnější podoby.

Dalším zdrojem, který doporučuji, je *specifikace WebSocket API* (W3C 2012). Ta popisuje způsob využití JavaScript WebSocket API ze strany klienta. Ačkoliv v této specifikaci jde o klientskou část, tak je dobré při programování serveru vědět, co a jak probíhá na klientské straně.

2 Současný stav řešené problematiky

V této kapitole zjistím podíly jednotlivých serverových programovacích jazyků používaných při tvorbě webových stránek. Na základě zjištěných dat pak vysvětluji, proč si myslím, že právě PHP je vhodně pro tvorbu WebSocket serveru, a to nejen z programátorského hlediska, ale také z pohledu byznysu.

Dále v této kapitole vysvětluji způsob analýzy současné problematiky WebSocket řešení v programovacím jazyce PHP a jejím následným provedením. Nejprve popisuji řešení pro tvorbu obecných řešení pro komunikaci PHP serveru s klientem v reálném čase a následně se věnuji knihovnám zaměřeným přímo na problematiku WebSocketů.

U každého řešení rozebírám jak jejich možnosti využití, tak jakousi uživatelskou přívětivost v podobě hodnotnosti dokumentace a samotné pohodlnosti použití. Během analýzy nezkoumám kód řešení příliš do hloubky a spíše se zaměřím na dostupné informace, které jsou nabízeny samotnými tvůrci, jako například dokumentace. Také zjistím statistiky použití jednotlivých knihoven z webu Packagist, který slouží jako zdroj balíčků pro prakticky jediný využívaný správce závislostí v PHP komunitě, Composer.

V kontextu celé práce bych rád zdůraznil, že reálný čas v případě komunikace serveru s klientem znamená ideálně minimální čas potřebný pro výměnu dat mezi serverem a klientem a zpracování dat serverem. Z povahy fungování počítačových sítí je nemožné, aby jakákoliv operace mezi dvěma různými zařízeními probíhala v reálném čase, jelikož vždy bude přítomna prodleva, během které budou data putovat mezi jedním a druhým zařízením.

2.1 Podíly serverových programovacích jazyků na webových stránkách

Data využívání serverových programovacích jazyků čerpám ze stránky W3techs (2018a). Ta sbírá údaje ze seznamu nejlepších desíti milionů webových stránek dle společnosti Alexa. Dle dat je zdaleka nejpoužívanějším serverovým programovacím jazykem pro tvorbu webů PHP, které je použito na zhruba 83 % webů, přičemž jeho podíl v posledních letech stále roste. Nechává tak za sebou jiné populární jazyky jako Java či ASP.NET. Kompletní data jsou k nalezení v tabulce 1.

Velmi mě překvapuje, jak silnou pozici PHP má. Očekával jsem, že především Java, Python a JavaScript budou mít větší podíl. Dominanci PHP přisuzuji tomu, že spousta vlastníků stránek je provozuje na redakčních systémech jako například Wordpress, Drupal nebo Joomla a eshopy na řešeních jako jsou PrestaShop či WooCommerce. Všechny tyto systémy jsou napsány v PHP a pro laiky jsou velmi jednoduché na instalaci a nastavení a především jsou zadarmo, jejich uživatelé tedy musí platit pouze hosting a doménu a případné drobné úpravy programátory. Pokud by si pořídili web na zakázku, stál by je násobně více peněz.

Tabulka 1: *Statistika využití serverových programovacích jazyků na webových stránkách. Údaje jsou vždy platné k 1. 1. daného roku. Jazyky jsou seřazeny od nejvyššího podílu k nejmenšímu. Jedna webová stránka může využívat více programovacích jazyků. (Zdroj: W3techs 2018a)*

Programovací jazyk	2011	2012	2013	2014	2015	2016	2017	2018
PHP	75,3 %	77,3 %	78,7 %	81,6 %	82,0 %	81,7 %	82,4 %	83,1 %
ASP.NET	23,4 %	21,7 %	20,2 %	18,2 %	17,1 %	16,0 %	15,3 %	14,1 %
Java	3,8 %	4,0 %	4,1 %	2,7 %	2,8 %	3,0 %	2,7 %	2,5 %
ColdFusion	1,3 %	1,2 %	1,1 %	0,8 %	0,7 %	0,7 %	0,6 %	0,6 %
Ruby	0,5 %	0,6 %	0,5 %	0,4 %	0,6 %	0,6 %	0,6 %	0,6 %
JavaScript	-	<0,1 %	<0,1 %	0,1 %	0,1 %	0,2 %	0,3 %	0,4 %
Perl	1,1 %	1,0 %	0,8 %	0,6 %	0,5 %	0,5 %	0,4 %	0,3 %
Python	0,3 %	0,2 %	0,2 %	0,2 %	0,2 %	0,2 %	0,2 %	0,2 %

2.2 Vhodnost PHP pro WebSocket servery

Z minulé podkapitoly vidíme, že PHP je majoritním serverovým programovacím jazykem pro webové stránky. Ačkoli jsou pro nás data zkreslená využíváním systémů pro správu obsahu laiky, tak se stále dá předpokládat, že firem stavějících své weby na PHP je spousta. Obvykle to znamená, že tyto firmy či především jejich dodavatelé mají programátory právě pro PHP.

Dle mé úvahy by dávalo ekonomicky největší smysl využít právě PHP programátory pro tvorbu WebSocket serveru. Pokud by měli k dispozici nástroj, který by jim umožnil rychle a efektivně vyvinout vlastní WebSocket server, tak by jim vývoj zabral velmi málo času a firmy by tak nemusely najímat programátory, kteří by WebSocket server psali v jiném jazyce, například JavaScriptu (Node.js), který je v open-source komunitě na tento účel používán zdaleka nejčastěji (GitHub 2018a). Ačkoli firmy věnující se dodávání webových stránek klientům budou nejspíše mít i programátory, kteří JavaScript umí, tak nejspíše půjde o front-end vývojáře, kteří nemusí mít zkušenosti či znalosti právě se serverovým JavaScriptem.

2.3 Řešení pro komunikaci PHP serveru s klientem v reálném čase

Pro napsání WebSocket serveru v PHP můžeme využít již vestavěných funkcí jazyka. Navrhovat však aplikaci za pomocí těchto základních funkcí, jako například funkcí stream, je velmi složité a zdlouhavé. Existuje proto několik řešení, které programátorům zjednodušují práci tím, že tyto funkce obalí do vlastních rozhraní, které obvykle využívají plně objektový přístup.

Jednotlivé řešení budou seřazeny podle počtu stažení jejich hlavního balíčku skrz správce závislostí Composer. Seznam těchto řešení spolu s daty o počtu stažení a oblíbenosti uživatelů ze stránky GitHub je v tabulce 2.

Tabulka 2: *Statistiky jednotlivých knihoven napsaných v jazyce PHP pro využití v aplikacích fungujících v reálném čase (Zdroj: Packagist 2018abcd)*

Název knihovny (balíček)	Počet stažení	Oblíbenost (GitHub hvězdy)
Amp (amphp/amp)	164 895	1297
Kraken (kraken-php/kraken)	772	70
ReactPHP (react/react)	1 197 952	5943
Workerman (workerman/workerman)	89 651	5927

2.3.1 ReactPHP

Nejrozšířenější a nejoblíbenější sadou knihoven je ReactPHP. Tato sada obsahuje několik knihoven, které zpříjemňují psaní aplikací fungující asynchronně a v reálném čase. Ačkoliv je možno použít balíček obsahující všechny komponenty patřící pod ReactPHP, většina projektů a jiných knihoven si vybírá pouze ty, které potřebují.

Naprosto nejpoužívanější komponentou je *react/promise*, která má přes 20 milionů stažení a je obsažena v 217 dalších knihovnách (Packagist 2018e). Dokonce ji využívají i ostatní knihovny, které se soustředí na asynchronní programování v PHP, jako například Amp. Tato knihovna implementuje takzvané promises známé především z JavaScriptu a bude popsána v následující kapitole.

Další hojně užívanou komponentou je *react/event-loop*, která má přes 5 milionů stažení a je na ní závislých 243 knihoven (Packagist 2018f). Tato komponenta se stará o celý běh asynchronního skriptu v PHP. Zjednodušeně se dá říct, že si programátor vytvoří instanci třídy implementující LoopInterface, následně této instanci přiřadí akce, které se mají vykonat po určitých událostech a nakonec smyčku spustí. Události mohou být například časovače nebo příchozí HTTP požadavky. Pomocí této komponenty se dá napsat například i jednoduchý HTTP server, který může nahradit zavedené servery jako Apache či IIS.

Samotný ReactPHP pak nabízí také komponenty pro práci se streamy či sockety, které dobře doplňují předchozí dvě komponenty. Dále také existuje spousta skvělých knihoven, které jsou psány asynchronně a jsou kompatibilní s ReactPHP. Ty jsou určeny pro přípojování k databázím, jako například MySQL, Redis nebo ElasticSearch, nebo třeba k využití knihoven, které jinak nejsou schopny pracovat asynchronně, jako například ZeroMQ či logovací knihovna Monolog.

2.3.2 Amp

Amp se v mnohém podobá ReactPHP. Jeho hlavní výhodou jsou tzv. Coroutines, které přináší trochu jiný styl psaní neblokujícího kódu. Na rozdíl od ReactPHP, který ve svých Promises používá callbacky, jsou zde použity generátory funkcí využívajících operátor `yield`.

Jednotlivé součásti, které Amp nabízí, jsou dost podobné jako ty, které má ReactPHP. Mezi jednotlivými balíčky lze najít HTTP server, WebSocket server, databázové klienty či práci se streamy. Nejvíce mě zaujal balíček *windows-registry*, který je určen ke čtení registrů operačního systému Windows. Ten je dle dokumentace určen především pro komponentu *dns*, která vyžaduje zjištění DNS serveru ze systému, která je uložena právě v registrech Windows.

2.3.3 Workerman

Workerman je čínský framework, který má opět velmi podobný obsah, jako předchozí dvě knihovny. Obsahuje součásti pro tvorbu jednoduchých HTTP, WebSocket či TCP serverů. Umožňuje také vytvoření vlastního komunikačního protokolu. V příkladech pak používá spoustu komponent z ReactPHP.

2.3.4 Kraken

Kraken je další knihovnou zaměřenou na asynchronní programování v PHP. Jeho jednotlivé součásti jsou velmi podobné jako ty, které se nachází v předchozích řešeních, ale snaží se o více objektový přístup a je spíše frameworkem, než pouhým seskupením různých komponent, jako je tomu u předchozích řešení. Jeho nevýhodou je však to, že nepodporuje operační systémy Windows.

2.4 Řešení zaměřené na WebSocket servery

Řešení, které by se zaměřovaly na WebSockety je logicky méně, než obecných řešení. Přitom platí, že všechny jsou jakousi nadstavbou právě nad knihovnami zmíněnými dříve v této kapitole.

2.4.1 Ratchet

Ratchet (2018a) je jednou z prvních PHP knihoven zaměřených na tvorbu WebSocket serverů. Její vývoj začal již v roce 2011 a je stavěna na komponentách ReactPHP. Knihovna řeší především správu spojení s klienty a na programátorovi tak je pouze aplikační logika, která je spuštěna po událostech vyvolanými klienty (připojení, odpojení, zpráva).

Jako hlavní nevýhodu této jinak skvělé knihovny vidím v absenci vestavěného řešení pro škálování. Ačkoliv v dokumentaci je zmínka, jak škálovatelnosti dosáhnout, tak ji stále musí řešit programátor, který by se jinak mohl věnovat aplikační logice. Navíc uvedené ukázkové řešení postrádá pružnost ve volbě systému, který by se staral o rozesílání zpráv mezi více serverů.

Při použití Ratchetu jsem také měl problémy s tím, že je potřeba psát spoustu neobjektového kódu. Prakticky veškeré součásti, které jsem potřeboval využít, jsem musel ručně inicializovat a předávat jednotlivým třídám s aplikační logikou.

2.4.2 Aerys

Aerys je jednou ze součástí Ampu. Staví tak na jeho ostatních komponentách a plně využívá Coroutines. Umožňuje tvorbu HTTP nebo WebSocket serverů. Dostupnou funkčností se moc neliší od Ratchetu. Jeho dokumentace však není tak dobře strukturovaná, ale je obsáhlá a všechny důleži-

té informace v ní jsou obsaženy. Hlavní rozdíl mezi Aerysem a Ratchetem tak vidím v rozdílném způsobu psaní kódu a je především na programátorovi, jaká z těchto klihoven mu bude více vyhovovat.

2.4.3 Kraken Network

Kraken Network je součást frameworku Kraken a plně těží z jeho součástí. Obsahuje tak například skvělé řešení závislostí tříd, které předchozí dvě řešení úplně postrádají. Hlavní problém Kraken Network však spočívá v tom, že je to framework určený pro psaní asynchronních aplikací a pro tvorbu WebSocket serverů tak obsahuje zbytečně moc nepotřebných součástí, které mohou programátorům klást překážky pod nohy. Podobně jako předchozí dvě řešení vůbec neřeší škálovatelnost.

3 Přínosy a problémy při nasazování PHP WebSocket serveru

V této kapitole se nejprve věnuji alternativám, které jsou či mohou být využívány pro dosažení stejné funkčnosti WebSocket serverů. Díky tomu vysvětlím, jaké přínosy WebSocket servery obecně mají a proč si myslím, že by se měly používat co nejčastěji, čemuž chci touto prací pomoci. Dále rozebírám praktické problémy při nasazení PHP WebSocket serveru. Nakonec popisuji, v čem je škálování WebSocket serverů jiné, než škálování klasických webových serverů.

3.1 Alternativy WebSocket serverů

Existuje několik alternativních možností, jak dosáhnout komunikace klienta s webovým serverem v reálném čase. V následujících podkapitolách je popsán jejich princip a nevýhody oproti využití WebSocketů. Celá tato podkapitola je jakýmsi zjednodušeným výtahem z prací, které jsem doporučil ve druhé kapitole.

3.1.1 AJAX polling

Nejčastěji používanou možností, ke které se uchýlí drtivá většina tvůrců webových stránek je tzv. Asynchronous Javascript and XML (dále pouze AJAX), který pomocí JavaScriptu ze strany klienta vyšle požadavek na server, který jim následně odešle data, která si klient zpracuje. Tyto požadavky se následně opakují ve zvoleném intervalu.

Pro některé systémy je toto řešení dostačující, ale na webových stránkách, kde potřebujeme docílit opravdové komunikace v reálném čase je naprosto nevyhovující. V takových případech by interval mezi jednotlivými požadavky musel být tak krátký, že by v případě více připojených klientů brzy přetížil webový server. Navíc se v každém požadavku posílá naprosto stejná hlavička, což se dá říct i o každé odpovědi ze serveru. Toto pak představuje veliký problém zejména pro uživatele mobilních zařízení, na kterých mají určený datový limit, či pro uživatele na velmi pomalém připojení.

3.1.2 Long polling

Long polling se částečně podobá AJAX pollingu. Rozdílem je, že dokáže fungovat v reálném čase. Podstatou long pollingu je to, že po přijmutí AJAX požadavku serverem nedojde k okamžitému odeslání odpovědi, ale server čeká na nějakou událost, po které by mohl odpověď zaslat. Klient po celou tuto dobu čeká na odpověď.

Jeden problém z klasického AJAX pollingu však stále přetrvává. Po každém úspěšném přijmutí odpovědi klientem se musí odeslat nový požadavek na server tak, aby klient byl opět připraven, až mu server zašle nějaké data. Opět tak dochází ke zbytečnému odesílání stejných hlaviček a stejně tak se při rychlé frekvenci událostí může server přetížit.

3.1.3 HTML5 server sent events

HTML5 přineslo řadu novinek, mezi které patří i tzv. Server sent events (dále jen SSE), někdy také nazývány EventSource. Tato technologie již umožňuje klientům, aby se připojili k serveru pouze jednou při úvodním požadavku na server. Od té chvíle je klient připraven přijmou kdykoliv jakékoliv data ze serveru, aniž by se na ně musel dotazovat opakovanými požadavky. Podstatné je zmínit, že toto připojení je realizování pomocí HTTP, na což v následujících řádcích navážu.

Velké omezení tohoto řešení je to, že neumožňuje stejným připojením zasílat data z klienta na server, jelikož je připojení pouze jednosměrné. Toto lze obejít tím, že v případě potřeby z klienta odešleme na server AJAX požadavek s daty, ale zde opět narážíme na problémy předchozích dvou řešení. Dalším problémem tohoto řešení je absence podpory v některých prohlížečích, především v Internet Explorer a Edge. Technologii SSE podporují prohlížeče necelých 90 % uživatelů (Can I Use 2018a). Problematické může být i to, že SSE umí pracovat pouze s UTF-8 daty.

Dalším potenciálním problémem je to, že některé prohlížeče (Firefox, Chrome) mají limitován počet možných HTTP připojení k serverům ve stejném čase na počet 6 (Bugzilla 2013, Chromium Bugs 2013). Samotní vývojáři těchto prohlížečů se vyjádřili v tom smyslu, že tento limit z výkonnostních důvodů nehodlají zvyšovat a ani nehodlají SSE připojení z tohoto limitu vyjmout. Pokud by tedy uživatel otevřel více oken prohlížeče, na nichž by 6 stránek využívalo SSE, nepovedlo by se mu načíst další stránku, jelikož by již předchozí otevřená okna vyčerpaly limit.

3.2 Přínosy WebSocketů

Z minulé části této kapitoly vyplynulo, jaké problémy mají jiné možnosti řešení komunikace mezi klientem a serverem v reálném čase. WebSockety tyto problémy prakticky eliminují.

Fungují podobně jako SSE s tím rozdílem, že umožňují i komunikaci od klienta k serveru. WebSockety také nevyužívají HTTP protokol, nýbrž definují svůj vlastní.

WebSockety mají také lepší podporu v prohlížečích, nativně technologii podporují prohlížeče zhruba 95 % uživatelů (Can I Use 2018b), což je o 5 % více, než u SSE. Navíc podporu pro WebSocket obsahují i Internet Explorer od verze 10 a Edge.

WebSockety netrpí také problémem limitu maximálního počtu připojení k serverům pomocí HTTP protokolu, jelikož používají vlastní protokol, který prohlížeče nijak nelimitují. Dále také mají podporu jak pro UTF-8, tak pro binární data.

3.3 Problémy při nasazení PHP WebSocket serveru

Obecné přínosy technologie WebSocketů jsou zřejmé. Problémy však nastávají při jejich implementaci na serveru, kdy jak samotné PHP, tak operační systémy dokáží programátorům klást překážky.

3.3.1 Asynchronní PHP

Specifikem PHP je, že běžné skripty v něm napsané jsou blokující. A to je v případě, kdy má fungovat asynchronně, docela problém. Při spuštění skriptu se totiž vykonávají jednotlivé příkazy jeden po druhém a vždy se čeká na dokončení jednoho příkazu, než je spuštěn následující. Naštěstí existují různé knihovny, které dokáží z PHP udělat plně asynchronní programovací jazyk. Nejlepším příkladem je knihovna *ReactPHP*, kterou jsem rozebíral v minulé kapitole. Jak samotná knihovna, tak její různé rozšíření a doplňky, nejčastěji využívají takzvané Promises. Při jejich použití můžeme psát plně asynchronní kód, ale má to svá specifika. Pokud potřebuji spustit nějaký kód až po dokončení určité operace, tak jej nemůžu jednoduše napsat za kód této operace. Promises pro tento účel využívají tzv. callbacků. Nejlépe to vysvětlím na příkladu ve výpisu 1.

Výpis 1: *Porovnání klasického synchronního blokujícího kódu a asynchronního kódu za využití react/promises.*

```
1  $query = 'INSERT INTO table (col1, col2) VALUES (?, ?)';
2  $values = ['val1', 'val2'];
3
4  // klasický synchronní kód
5  $mysql->query($query, $values);
6  echo 'Příkaz proveden';
7
8  // asynchronní kód za využití react/promise a react/mysql
9  $mysql->query($query, function () { echo 'Příkaz proveden'; }, $values);
10 echo 'V době vypsání tohoto textu ještě příkaz není proveden';
```

Jak lze vyčíst z příkladu, tak text na řádku 10 se vypíše ihned po započetí vkládání dat do databáze, přičemž samotné vkládání ještě není dokončeno. Obecně všechny operace, které neprobíhají pouze v rámci skriptu, budou mít nějakou prodlevu. Příkladem je čekání na provedení příkazu v databázi nebo na vložení dat do souboru. Na toto je potřeba při psaní asynchronního kódu myslet. Potenciál asynchronního kódu lze dobře použít v případě, kdy je potřeba spustit několik na sobě nezávislých operací najednou. Jejich provedení je pak mnohem rychlejší, než kdyby se spouštěly postupně a před spuštěním se vždy čekalo na dokončení předešlé operace.

Také kvůli výše zmíněnému se při psaní asynchronního kódu často využívají knihovny, které pracují na tomto principu. Často jsou psány přímo pod hlavičkou samotného *ReactPHP*, jako v příkladu použitý *react/mysql*. Záleží pak na autorovi, zda použije callbacky v samotných funkcích tak, jako je to patrné v ukázce, nebo se bude držet druhého způsobů využití Promises, kdy se využívá řetězení příkazů `then`. Příklad tohoto přístupu je v následujícím úryvku kódu, který využívá knihovnu *clue/redis-react*.

```
$redis->get('cache.sessions.25')
    ->then(function ($result) { echo 'Session data: ' . $result; });
```

3.3.2 Omezení otevřených spojení

Jak samotné PHP bez specifických rozšíření, tak i Linuxové operační systémy, limitují počet aktivních připojení, které může mít jeden proces otevřené. Dle diskuze na GitHubu (2015) knihovny *Ratchet* je obvykle nemožné připojit klienta k serveru, blíží-li se počet aktivních připojení k číslu 1024. Tento limit na straně linuxového operačního systému lze upravit v souboru

/etc/security/limits.conf. Podle uživatelů na portálu ServerFault (2011) je u nejnovějších operačních systémů Windows tento limit značně vyšší, a to 16 384 otevřených připojení na jednoho uživatele.

To ale stále neřeší problém na straně PHP, jehož oficiální buildy v sobě mají také nastaven limit otevřených připojení pro funkci `stream_select` (Ratchet 2018b). To se dá vyřešit instalací nějakého rozšíření. Nejlepší volba dle výše zmíněné diskuze je rozšíření *libevent*. Problém s tímto rozšířením ale je, že nepodporuje PHP verze 7 (PECL 2018a), a tudíž by bylo potřeba jej kompilovat.

Dalším doporučovaným řešením dle knihovny React je použít rozšíření *ev*, které již PHP 7 podporuje (PECL 2018b) a využívá knihovny libev napsané v jazyce C. Vzhledem k tomu, že toto rozšíření podporuje i nejnovější verze PHP a pro jeho funkčnost není potřeba instalovat nic dalšího, tak jej plně doporučuji.

3.4 Škálování WebSocket serverů

Klasické webové servery se škálují poměrně jednoduše. Stačí spustit několik samostatných instancí takových webových serverů a nasměrovat na ně nějakou reverzní proxy, respektive load balancer. Tím se zatížení rovnoměrně rozloží na jednotlivé webové servery. Obvykle tyto servery využívají také nějaké externí služby, jako databáze, které obvykle nějaké řešení škálovatelnosti mají již zabudované. Například relační databáze obvykle používají jeden hlavní server pro zápisy i čtení (master) a několik dalších serverů pouze pro čtení (slave). Díky tomuto vestavěnému škálování je zaručena konzistence externích dat na webovém serveru.

Na WebSocket server je ale potřeba pohlížet trochu jinak. Při stejném postupu jako u webových serverů by vyvstalo několik problémů. Při použití více serverů by se připojující klienti na jednotlivé servery rozprostřeli. Problém však nastává, potřebuji-li do nějaké operace na serveru zapojit dva různé klienty. Můžou pak nastat dvě situace.

První možností je, že oba klienti jsou připojeni ke stejnému WebSocket serveru. V takovém případě není pro server problém přijmout data od jednoho klienta, zpracovat je a odeslat druhému klientovi, jelikož je daný server aktivně spojen s oběma klienty.

Druhá možnost, která může nastat, je, že daní dva klienti jsou připojeni na různé WebSocket servery. V takovém případě dochází k tomu, že jeden server sice přijme data od prvního klienta, může je nějakým způsobem zpracovat, ale již nemá otevřené spojení ke druhému klientovi a tudíž mu nemá zpracovaná data jak poslat. Toto je tedy potřeba nějak vyřešit.

Ideálním řešením je využití nějakého message brokeru. Ten je určen k tomu, že přijme data z nějakého zdroje a dále je nějak zpracovává. Příklady takového zpracování může být převedení dat do jiného formátu, sdružení několika vzájemně souvisejících zprávy, odesílání zpráv na základě typu do jiné služby atd. Pro potřeby škálování WebSocket serverů je především důležitá podpora vzoru Publisher-subscriber (zkráceně PubSub). (Techopedia 2018)

Tento vzor funguje tak, že se u message brokeru mohou různé služby přihlásit k odběru zpráv (subscribe). Následně mohou služby, a to i ty, které k odběru přihlášené nejsou, posílat message brokeru zprávy (publish), které se při přijetí okamžitě rozešlou všem službám přihlášeným

k odběru. Obvykle message brokery podporují více kanálů, respektive témat, se kterými lze komunikovat.

Technologií podporujících PubSub je spousta. Mezi ty, s kterými se v odborných diskuzích a článcích setkávám nejčastěji, jsou RabbitMQ, Kafka nebo Redis. Často se stává, že jedna z těchto technologií slouží v konkrétním projektu i k jiným účelům. Já například často využívám Redis, který je primárně key-value databáze, ke kešování.

Volba message brokeru pro WebSocket server by ale samozřejmě měla záviset i na jiných požadavcích, jelikož každá technologie podporující PubSub může být zaměřena na něco jiného a také požadavky na funkčnost WebSocket serveru se mohou lišit. Například Redis je, jak jsem již zmínil, key-value databáze, která podporuje u hodnot do ní uložených nastavit expiraci. Lze pak nakonfigurovat, že v případě, kdy dojde k expiraci nějaké hodnoty, se pošle notifikace právě na PubSub kanál se speciálním názvem. Této funkcionality jsem využil při programování detekce, zda je nějaký uživatel systému přihlášen a připojen k chatovací části aplikace.

Prakticky je možné naprogramovat PubSub broker i pomocí WebSocket serveru. V případě, kdy by nebyla potřeba žádná jiná funkcionalita, by to dle mého bylo nejspíše i ideální řešení, jelikož takový WebSocket server by se spouštěl stejně jako ty, pro které by sloužil jako message broker. Využíval by tak prostředí, které by již bylo připravené pro ostatní WebSocket servery a nebylo by potřeba instalovat a konfigurovat žádné další technologie.

4 Návrh struktury frameworku a komponent

V této kapitole se věnuji především zamyšlení nad různými strukturami jednotlivých částí frameworku a volbou těch nejideálnějších. Spousta těchto voleb bude naprosto zásadní v následném využívání frameworku, které by mělo být co nejjednodušší.

Abych byl schopen zvolit ideální strukturu frameworku, musím nejprve prozkoumat, jaké jsou datové vstupy a výstupy serveru, respektive jakým způsobem server komunikuje s klientem a také s message brokerem.

4.1 Životní cyklus

Životní cyklus zpracování zprávy na straně serveru začíná přijmutím zprávy od klienta, kterým nemusí být nutně pouze webový prohlížeč koncového uživatele, ale i různé jiné servery. Ta by měla být ideálně nějakým způsobem zkontrolována, než se s ní bude cokoliv dělat, aby bylo zajištěno, že zpráva má náležitosti, které potřebujeme k dalšímu zpracování. Případně se zde můžou odehrávat i nějaká prvotní úprava dat, jako například doplnění časové známky. Máme-li centrální databázi a chceme do ní ukládat data ze zprávy, tak je ideální je do databáze vkládat v tomto momentu. Následně by se měla zpráva rozeslat na všechny ostatní běžící WebSocket servery. Toho dosáhneme pomocí message brokeru, který podporuje vzor Publisher-Subscriber.

Následně by message broker měl přijatou zprávu rozeslat mezi všechny připojené servery, včetně toho, ze kterého zpráva přišla. Je zde i možnost, že by se zpráva z message brokeru neodeslala na server, ze kterého přišla a rovnou se předala již v rámci serveru. Toto řešení ovšem závisí na podpoře jednotlivých message broker. Z vlastní zkušenosti vím, že například *Redis* nepodporuje selektivní vyjmutí konkrétních služeb přihlášených k odběru zpráv, a tak u něj tuto funkcionalitu využít nelze.

Po přijmutí zprávy z message brokeru by si každý ze serverů měl zprávu zpracovat. Toto zpracování by mělo ideálně být naprosto stejné na všech serverech. Framework bohužel nedokáže ošetřit, zda je na všech serverech nejaktuálnější verze kódu a tak je již pouze na programátorech, jak se s nasazením popasují. Pod zpracováním zprávy si lze představit uložení do databáze (nedošlo-li k němu již před odesláním na message broker nebo má-li každý server vlastní databázi) či rozeslání mezi relevantní připojené klienty.

Na základě popsaného životního cyklu mě napadají dvě konkrétní řešení struktury frameworku, které v následujících podkapitolách popíši, zmíním výhody a nevýhody a nakonec vyberu to, které je dle mě vhodnější především z pohledu čistoty kódu.

4.1.1 Jedna třída s dvěma zpracovatelskými metodami

První možností je užití jedné třídy, která využívá dvě různé metody. Jedna metoda slouží pro prvotní zpracování před odesláním na message broker a druhá následně zpracovává zprávy, které přijdou z message brokeru. Takováto třída by měla také obsahovat metody, která se bude starat o

správu klientů připojených k serveru po jejich připojení a odpojení, případně by měla být napojena na instanci jiné třídy, která by toto řešila.

Výhodou tohoto řešení je, že máme veškerý kód na jednom místě, což by při použití více takových tříd v rámci jednoho serveru urychlilo orientaci. Naopak nevýhodu vidím v tom, že takováto třída musí mít uloženy veškeré připojené klienty, k nimž by následně bylo možno přistupovat také z metody sloužící pro prvotní zpracování, což by bylo špatně. Cílem celého frameworku totiž je škálovatelnost a byla by možnost (byť neúmyslnou chybou) odesílat data jen klientům připojeným ke konkrétnímu serveru, což jde právě proti řešení škálování, při kterém chceme mít zpracované data rozeslány mezi všechny relevantní klienty napříč všemi servery.

4.1.2 Dvě třídy po jedné zpracovatelské metodě

Druhou možností je využití dvou různých tříd, kde každá bude zpracovávat pro ni relevantní událost. Jedna řeší pouze prvotní zpracování zprávy od klienta, zatímco druhá zpracovává zprávu z message brokeru a také se stará o správu klientů, případně využívá služby jiné třídy k tomuto určené.

Při této možnosti se odstraní nevýhoda předchozího řešení, jelikož se třída přijímající data od klientů nebude starat o správu připojených klientů, tudíž k nim nebude možno přistupovat při prvotním zpracování. Naopak dojde k roztržení kódu do dvou tříd, tudíž bude orientace v kódu mírně složitější. To se však dá minimalizovat správnou strukturou tříd, ve které bude na první pohled jasné, které dvě třídy spolu souvisí.

4.1.3 Volba vhodné možnosti a její zdůvodnění

V programování se obvykle dodržují nějaká nepsaná pravidla neboli principy. Jedním takovým je i tzv. SOLID princip. Pro vybrání ideálního řešení struktury frameworku potřebuji jeho první část, kterou je princip jedné odpovědnosti (Single Responsibility Principle), který je v názvu zastoupen úvodním S. Ten říká, že jedna třída by měla mít odpovědnost za vykonávání právě jedné činnosti, která by měla být jasně patrná z jejího názvu. V obou představených řešeních jsou tři takovéto činnosti. Jednou je úvodní přijetí dat od klienta, jejich zpracování a odeslání message brokeru. Druhou je přijetí zprávy od message brokeru a její následné zpracování. Třetí pak je správa připojených klientů. Ideální tedy je pro každou z popsaných funkčností vytvořit vlastní třídu.

4.1.4 Jmenné konvence tříd

Pro pokračování si vypůjčím citát, který trefně popisuje nejtěžší věci na programování:

There are only two hard things in Computer Science: cache invalidation and naming things. (Phil Karlton)

Pro mě je momentálně důležitá druhá jmenovaná věc, jelikož dané tři třídy nyní musím nějak pojmenovat. Rozhodl jsem se, že celý framework bude napsán anglicky, jelikož je to v open-source komunitě víceméně standart, podobně jako když lékaři užívají latinu. V případě třídy, která se bude starat o připojené klienty, je název relativně jednoduchý, a to `ConnectionManager`. Tato třída by měla mít metody, které reagují na událost připojení klienta (`onOpen`), odpojení klienta (`onClose`),

chybu při spojení (`onError`) a další metody, které budou zpřístupňovat klienty třídám, které tuto třídu budou využívat, dle parametrů, které závisí na konkrétní aplikační logice.

Zbylé dvě třídy však dělají v celku podobnou činnost, a to zpracování dat. Pouze mají jiné vstupy a výstupy. Pro třídu, která přijme data od klienta, zpracuje je a následně odešle message brokeru, mě napadají názvy `Forwarder` a `Receiver`. Název `Forwarder` by odpovídal tomu, že třída přeposílá data od klienta k message brokeru. To však není úplně pravda, jelikož se zde může nacházet aplikační logika a také nemusí vždy dojít k odeslání dat na message broker. `Receiver` mi připadá vhodnější, i přes to, že obě třídy přijímají data. U této třídy však jde o úplně první kontakt s daty od klienta, zatímco v druhém případě jde už o data, které prošly daným WebSocket serverem. Programátor by měl konec konců být frameworkem odstíněn od skutečnosti, že data mezi první a druhou zpracovatelskou třídou „navštívily“ message broker.

Pro druhou třídu mě prakticky okamžitě napadá název `Handler`. Především i kvůli tomu, že jsem tento název ve stejném kontextu použil již při mé minulé zkušenosti s programováním WebSocket serveru a už tehdy mi přišel nejvhodnější.

Ve frameworku počítám s tím, že si programátor může vytvořit několik různých aplikačních logik, a tak tyto třídy použije vícekrát s tím, že do názvu před ně doplní název, který danou aplikační logiku nejlépe vystihuje, například `ChatReceiver`. Za naprosto ideální pak považuji vytvoření rozhraní, která bude muset programátor implementovat. Také je žádoucí vytvořit nějakou základní implementaci pro správce připojení, kterou může programátor použít. Tato implementace bude obsahovat kromě povinných metod jednu vlastní pro poskytnutí všech připojených klientů.

4.2 Spouštěcí skript

Aby server mohl přijímat zprávy, je potřeba jej nejprve spustit. Toho dosáhnu kombinací spouštěcího skriptu, konfiguračního souboru a využití vzoru dependency injection. Teoreticky by se konfigurační soubor dal vynechat, ale pro přehlednost a jednoduchost jsem se rozhodl, že je lepší jej využít. Všechny nastavení v něm jsou přehlednější a dostupnější, než kdyby byly obsaženy přímo v kódu spouštěcího skriptu.

Samotný spouštěcí skript je velmi jednoduchý. Nejprve načte všechny třídy pomocí tzv. autoloadingu. Ten by ideálně měl probíhat pomocí skriptu správce závislostí Composer, ale je jen na programátorovi, jak třídy bude načítat. Následně se vytvoří `EventLoop` z ReactPHP, která se bude starat o celý běh serveru. Programátor zde může opět zvolit, jakou konkrétní implementaci využije, případně může použít předpřipravenou továrnu, která zvolí nejlepší dostupnou implementaci tak, aby programátor nemusel měnit žádné nastavení PHP. Poté je vytvořena instance samotného serveru, která vyžaduje jak `Loader`, tak `EventLoop`. Nyní je na programátorovi, jak bude postupovat dál. Hlavní třída frameworku má několik veřejných metod. Ty nejdůležitější jsou metody `addConfig` a `run`. První jmenovaná přidá do frameworku konfigurační soubor a lze ji použít několikrát. Druhá metoda pak spouští `EventLoop` a měla by být volána až na samotném konci skriptu. Její obsah podrobně popisuju v jedné z následujících kapitol.

4.2.1 Konfigurační soubory

Jak jsem zmínil v minulém odstavci, metodu `addConfig` lze volat opakovaně. To umožní programátorovi použít několik různých konfiguračních souborů, které se následně spojí do jednoho PHP objektu, který je dále využíván frameworkem. Tyto konfigurační soubory mohou mít praktický jakýkoliv formát, jelikož jsem se rozhodl, že je zbytečné limitovat programátory na jeden konkrétní formát, když můžou být zvyklí na jiné. Toho je dosaženo využitím implementací rozhraní, které definuje dvě statické funkce. Jedna vrací seznam přípon konfiguračních souborů. Má-li konfigurační soubor příponu konkrétního typu, bude spuštěna druhá metoda, která tento soubor zpracuje. Pro začátek jsem implementoval tyto relativně časté formáty:

- YAML
- Neon
- JSON
- XML

Programátor samozřejmě může přidat vlastní implementaci pro jeho oblíbený formát příslušnou metodou hlavní třídy frameworku.

Konfigurace samozřejmě musí obsahovat některé povinné položky potřebné pro fungování serveru. Naprosto zásadní je definice typu požadovaného message brokeru. Je pravděpodobné, že takovýto message broker bude potřebovat nějakou další konfiguraci, obvykle jeho adresu a přístupové údaje. Ty by měly mít v konfiguraci svou vlastní sekci. Při vytváření instance message brokeru se do něj pouze předá objekt s konfigurací a až samotná implementace si z tohoto objektu získá tuto sekci a nahlásí případné chyby.

Dále by mohly být povinné údaje o samotném serveru, přesněji IP adresa či adresy, ze které bude přístupný, port, na kterém poběží a také URL adresa, přes kterou na něj půjde přistupovat.

Nejčastější použití serveru je takové, že k němu jako klienti přistupují uživatelé webové aplikace z jejich prohlížeče, tudíž není známo, z jaké IP adresy se na server budou připojovat. Nastavení povolených IP adres tedy nechám volitelné s tím, že bez jeho uvedení budou povoleny příchozí spojení ze všech IP adres.

Dalším parametrem serveru je port. Ten by také mohl být pouze volitelný s nějakou definovanou základní hodnotou (nejčastěji se používá port 8080). Pak by mohlo ale dojít k situaci, kdy by programátor zapomněl port v konfiguraci nadefinovat a kvůli kolizi s jinou aplikací využívající tentýž port by se server nespustil. Programátor by pak mohl dlouho zjišťovat, co je špatně a nemuselo by jej napadnout, že musí nastavit port, který je volný. Když bude port povinný, tak by si při jeho nastavování nejspíše ověřil, zda daný port je opravdu volný a ke kolizi s jinou aplikací by tudíž nedošlo.

Posledním parametrem je URL adresa, přes kterou se klienti budou k serveru připojovat. Obvykle začíná na `ws://` či `wss://` a klient ji používá při vytváření instance WebSocket API. Zde opravdu nevidím žádnou smysluplnou možnost pro hodnotu v případě vynechání parametru, tudíž musí být povinný.

Poslední povinnou položkou v konfiguraci by měly být cesty, o kterých se zmiňuji v následující podkapitole s názvem Routování. Příklad minimální povinné konfigurace je ve výpisu 2. Všechny ostatní volitelné parametry i s jejich popisem jsou vypsány v sedmé kapitole.

Výpis 2: *Ukázka minimální konfigurace frameworku obsahující všechny povinné parametry ve formátu Neon.*

```
server:
    host: example.com
    port: 8080

broker:
    type: Example\Namespace\Brokers\RedisBroker

routes:
    /example: Example\Namespace\App\ExampleReceiver
```

4.2.2 Hlavní třída frameworku

Hlavní třída frameworku se stará o jeho kompletní inicializaci, nastavení a spuštění. Při jejím vytvoření jsou vytvořeny instance konfigurátoru, který se stará o zpracování konfiguračních souborů, a dependency injection kontejneru, který je podrobně popsán dále v této kapitole.

Po spuštění metody run se z konfigurátoru získá kompletní zpracovaná konfigurace, která je dále využita pro vytvoření WebSocket serveru. Také se zde nastavuje routování a message broker. Jejich podstata je vysvětlena v následujících podkapitolách. Nakonec je WebSocket server spuštěn.

4.3 Routování

Jak jsem již zmínil v části o jmenných konvencích hlavních zpracovatelských tříd, tak je možnost takových tříd mít několik s různými názvy. Aby byl programátor schopen více takových tříd využít, musí mít způsob, jak k nim klienty nasměrovat. Toho se dosáhne pomocí cesty v URI, na kterou se klient připojí a která bude pomocí konfigurace vázána na konkrétní zpracovatelské třídy. Názvy těchto cest musí v konfiguraci začínat vždy lomítkem následovaným názvem cesty. Tento název může obsahovat veškeré znaky, které se mohou vyskytovat v URI, včetně dalších lomítek. Cesta by měla směřovat na třídu implementující rozhraní první zpracovatelské třídy (`Receiver`). Jednoduchý příklad konfigurace ve formátu Neon je v následujícím výpisu.

```
routes:
    /chat: Example\App\ChatReceiver
    /dashboard: Example\App\DashboardReceiver
```

Předpokladem je, že druhá zpracovatelská třída (`Handler`) je ve stejném jmenném prostoru, jako `Receiver` a má také stejnou předponu. Ke třídě `Example\App\ChatReceiver` se tak bude vázat třída `Example\App\ChatHandler`. Toto řešení mi přijde nejjednodušší a nejintuitivnější. Druhou možností by bylo pro každou cestu specifikovat jak Receiver, tak Handler. To by jednak způsobilo větší složitost konfigurace, ale také by to mohlo vést ke špatnému umístění tříd do jmenných prostorů, kdy by spolu související třídy mohly být v úplně jiných jmenných prostorech a pak by se v kódu špatně orientovalo.

4.4 Message broker

Další velmi důležitou součástí frameworku je jeho připojení k message brokeru. Jelikož message brokerů, které by podporovaly PubSub, je nepřeberné množství, tak se nabízí řešení pomocí rozhraní, jehož implementaci si může programátor sám zvolit či vytvořit.

Implementace rozhraní message brokeru musí obsahovat tři metody. Jednu pro odesílání dat (`send`) a druhou pro přihlášení odběru zpráv z message brokeru a jejich předání správnému Handleru (`registerHandlers`). Nasměrování zpráv do kanálů či témat message brokeru se bude řešit až v konkrétní implementaci, jelikož různé brokery mohou mít různé pravidla pro pojmenování takovýchto kanálů. Pro tento účel slouží metoda `getChannelForHandler`, která vrací název kanálu pro konkrétní třídu typu Handler. Považuji také za vhodné přidat možnost nadefinování vlastních Handlerů pro specifické názvy kanálů. Toho lze využít například v již dříve zmíněném příkladu notifikačních kanálů Redisu. Toho se dosáhne pomocí předání dané sekce konfigurace metodě `registerHandlers`.

4.5 Dependency Injection

V rámci frameworku je třeba také vyřešit automatické vytváření instancí tříd tak, aby programátor nemusel všechny potřebné třídy inicializovat ručně. Pro dosažení tohoto je vhodné použití vzoru dependency injection. Obvykle použití tohoto vzoru značně komplikuje kód, proto jsem se rozhodl inspirovat se u nejznámějšího českého PHP frameworku Nette, ve kterém je obsažen takzvaný kontejner. Ten se stará o naprosto všechny závislosti frameworku a všech jeho tříd. Pro použití v mém frameworku je však toto řešení až zbytečně komplikované. Rozhodl jsem se napsat vlastní, jednodušší řešení, které sice nebude mít tak širokou škálu možností, avšak pro účel frameworku, kterým je umožnění rychlé a jednoduché tvorby WebSocket serverů, je dostatečné. Nejprve ale vysvětlím všechny možné přístupy k použití dependency injection.

4.5.1 Způsoby využití vzoru dependency injection

Nejjednodušším způsobem využitím dependency injection je předání závislostí parametrem metody, která danou závislost vyžaduje. Tento způsob ale začne být otravný v případě, že nějakou závislost vyžadují dvě metody v rámci jedné třídy. Pak by bylo potřeba danou závislost uvádět u každé metody zvlášť, a to by značně znepřehlednilo kód obzvlášť v případě, kdy daná metoda vyžaduje více závislostí a další parametry.

Druhou možností je využití takzvaných inject metod. Ty by měly obdržet závislost v parametru a následně ji uložit do atributu třídy. Z něj pak můžou závislost jednoduše získat všechny metody, které ji potřebují. Kód je díky tomu jednodušší a přehlednější. Nastává však problém, že při volání metody, která závislost vyžaduje pro svůj chod, ještě nebyla inject metoda zavolána a tudíž není daná závislost dostupná.

Nejlepším řešením pak je předávání závislosti konstruktorem. To funguje na stejném principu jako inject metody, ale s tím rozdílem, že konstruktor je volán vždy při vytvoření instance objektu. Díky tomu je jisté, že závislost bude vždy dostupná v atributech třídy.

4.5.2 Dependency injection kontejner

Samotné vyžadování závislostí konstruktorem ale neřeší problém automatického vytváření instancí tříd. Proto je potřeba vytvořit nějaký mechanizmus, který je schopný sám vytvořit, spravovat a poskytovat instance jednotlivých tříd. Ze zkušenosti s Nette frameworkem se mi osvědčilo používání takzvaného DI kontejneru.

Proto jsem se rozhodl, že si vytvořím vlastní kontejner, který musí mít následující funkcionality:

- Automatické vytváření instancí tříd ve chvíli, kdy jsou vyžadovány
- Poskytování již vytvořených instancí tříd
- Přidávání ručně vytvořených instancí tříd do kontejneru
- Předávání závislostí vytvářeným třídám konstruktorem, a to včetně jednoduchých datových typů
- Možnost vyžádat si konkrétní implementaci zvoleného rozhraní

Z těchto požadavků je jasné, že třída kontejneru musí mít atribut, do kterého se budou vkládat již vytvořené instance tříd. Tento atribut bude mít datový typ pole, jehož klíčem bude plný název třídy a hodnotou instance třídy. Dále je potřeba také atribut pro ukládání implementací rozhraní, který bude mít stejnou strukturu. Považuji za důležité zmínit, že kontejner může obsahovat pouze jednu instanci konkrétní třídy. Kdyby jich mohl obsahovat více, tak by nemohl vědět, kterou z nich zrovna uživatel v nějaké ze tříd vyžaduje.

Pro vyžádání instance nějaké třídy z kontejneru slouží metoda `get`, která jako parametr přijímá plný název vyžadované třídy či rozhraní. Tato metoda buď vrátí instanci z jednoho z polí, kam si kontejner ukládá všechny instance, nebo, v případě, že taková instance ještě neexistuje, vytvoří novou.

Vytváření nové instance začíná tak, že si kontejner za využití reflexe zjistí, zda a jaké instance tříd vytvářená třída vyžaduje v konstruktoru. Pokud vyžaduje nějakou závislost, tak kontejner využije právě svou vlastní metodu `get` a pomocí ní závislost získá či vytvoří. Problém nastává, pokud daná třída vyžaduje v konstruktoru nějaký jednoduchý datový typ.

Řešením tohoto problému jsou tzv. továrny, které si programátor může nadefinovat v konfiguračním souboru a které vytváří instance takových tříd místo kontejneru. Jedná se o jednoduché třídy implementující rozhraní s jedinou metodou `create`, která musí vracet instanci dané třídy. Této metodě je předána kompletní konfigurace, takže pokud vytvářená třída potřebuje například údaje pro připojení k databázi, může je továrna jednoduše z konfigurace získat.

Poslední veřejnou metodou kontejneru je metoda `add`, která dává možnost přidat do kontejneru ručně vytvořené instance, přičemž kontroluje, zda se již v kontejneru nenachází nějaká jiná instance stejného typu.

5 Výběr komponent pro framework

V této kapitole využiji znalosti získané v minulé kapitole a pokusím se zvolit vhodné komponenty, ze kterých následně vytvořím framework pro tvorbu škálovatelných WebSocket serverů v PHP.

5.1 Verze PHP

Zdaleka nejdůležitější je volba podporovaných verzí PHP. Ačkoliv verze 7 zaznamenala okamžitý úspěch, tak stále se z pochopitelných důvodů spousta programátorů drží starších verzí, především pak verze 5.6 (W3techs 2018b). Je ale potřeba brát v potaz, že spousta webů není buď aktivně udržována, nebo by přechod na verzi 7 vyžadoval investici času i peněz, kterou si spousta provozovatelů webů nemůže dovolit.

Toto je však problém u již běžících webů. Já sám se u všech webů, které tvořím, snažím využívat nejnovější verzi PHP. Sedmá verze totiž přináší výrazné vylepšení výkonnosti a především objektového modelu. Přináší totiž možnost striktního typování, které vede programátory k psaní čistého a pochopitelného kódu s minimem chyb.

Důležitým faktorem v tomto případě je i to, že framework je nově vyvíjený, nemá žádnou historii a tudíž ani žádný závazek k již existujícím uživatelům, který by mě nutil využívat starší verze. Dále je také potřeba uvažovat nad infrastrukturou, kterou budou mít uživatelé frameworku k dispozici. Hlavně v prvních měsících po vydání sedmé verze PHP nemohla spousta provozovatelů webů na tuto verzi přejít, jelikož jejich hosting novou verzi ještě nepodporoval a nemohli si dovolit přejít k jinému poskytovateli. Je ale nutno zmínit, že framework není určený k spouštění na klasickém hostingu, jelikož na nich jsou obvykle webové stránky poskytovány klientům skrz webové servery, jako jsou Apache, IIS nebo nginx. Framework musí být spuštěn přímo skrz PHP, tudíž provozovatel WebSocket serveru jej musí spouštět přímo přes fyzicky nainstalované PHP, a to ideálně na dedikovaném nebo cloudovém serveru.

Po zvážení všeho, co jsem zmínil v předchozích odstavcích, jsem se rozhodl pro omezení frameworku pouze na nejnovější verzi 7.2. Není důvod podporovat starší verze, jelikož jak infrastruktura, tak samotný kód bude teprve vznikat a udržování či rozšiřování frameworku bude v budoucnu jednodušší.

5.2 Zpracování WebSocket zpráv

První důležitou komponentou, kterou jsem si vybral, je v druhé kapitole analyzovaný *Ratchet*. Ten sám o sobě využívá několik dalších komponent a stará se o veškerou potřebnou komunikaci mezi serverem a klientem, což zahrnuje kompletní implementaci protokolu WebSocket.

Jedinou částí Ratchetu, která by mohla způsobit při programování frameworku komplikace, je jeho rozhraní pro aplikační logiku. To totiž obsahuje v jednom rozhraní metody jak pro správu spojení s klientem (`onOpen`, `onClose`, `onError`), tak pro samotnou komunikaci s ním (`onMessage`). Tím nejen

že porušuje SOLID princip rozdělení rozhraní (Arora 2015), ale především nepasuje na strukturu frameworku, kterou jsem navrhl. Tento problém obejdu tak, že vytvořím fasádní třídu, která bude implementovat toto rozhraní Ratchetu a volání konkrétních metod bude přesměrovávat na třídy aplikační logiky implementující rozhraní frameworku, konkrétně na ConnectionManager a Receiver.

5.3 Implementace message brokeru

Dále je potřeba vytvořit také nějakou implementaci message brokeru, která by byla ve frameworku již připravená k použití a sloužila by jako předloha pro vytváření dalších implementací. Prakticky jediný message broker, který jsem někdy využíval, byl Redis, tudíž jsem se rozhodl pro implementaci rozhraní pro něj. Již dříve jsem řešil využití Redisu pro vzor PubSub a narazil jsem na dvě knihovny. Více používaná z nich, *predis/predis-async*, nepodporuje připojení k Redisu přes SSL, a tudíž jsem se rozhodl použít *clue/redis-react*.

5.4 Zpracování konfiguračních souborů

V minulé kapitole jsem zmínil, že v základu framework má podporovat nejčastější typy konfiguračních souborů. Pro Json a XML již v PHP existuje vestavěná podpora, tudíž implementace tříd zpracovávajících konfigurační soubory těchto typů nebude vyžadovat žádné další komponenty. To ale není případ formátů Neon a YAML. Pro jejich zpracování je potřeba vybrat nějaké komponenty. Tvůrce formátu Neon vytvořil také parser souborů využívajících tento formát (*neon/neon*), nevidím tedy důvod jej nevyužít. Pro formát YAML jsem se rozhodl pro balíček *symfony/yaml*, který pochází z jednoho z nejznámejších PHP frameworků současnosti Symfony a dle Packagistu (2018) je nejpoužívanějším YAML parserem.

6 Uživatelská příručka

Tato kapitola obsahuje příručku pro uživatele frameworku, tedy programátory, kteří za pomocí něj budou vyvíjet vlastní WebSocket server v PHP. Provede uživatele od instalace frameworku, přes příklady použití jednotlivých součástí, až po finální spuštění serveru. Pro otestování funkčnosti bude uveden i jednoduchý klient napsaný v JavaScriptu.

Samotný příklad se bude zaměřovat na jednoduchou chatovací aplikaci, kdy klient může posílat zprávy jiným připojeným klientům na všech běžících serverech. Veškerá data budou odesílány ve formátu Json.

Jelikož se jedná o uživatelskou příručku a je určena k samostatnému použití, tak počínaje její první podkapitolou je psána tak, jako kdyby její čtenář zbytek této práce nečetl. Tím pádem se některé informace, které jsem uvedl již dříve, především v kapitole o struktuře, budou opakovat.

6.1 Instalace

Pro instalaci frameworku je nutno mít nainstalován správce závislostí Composer. Poté stačí v příkazovém řádku či terminálu přejít do složky, kam bude framework nainstalován a spustit následující příkaz:

```
composer create-project bearsys/wsf-sandbox .
```

Composer po jeho zadání stáhne veškeré soubory frameworku i jeho závislostí do dané složky. Nyní je v této složce několik složek a souborů. Jejich význam je vysvětlen v následujícím výčtu.

app\	Do této složky patří veškerý kód napsaný programátorem.
app\Routes\	V této složce se budou nacházet všechny implementace aplikační logiky. Dle cest budou rozděleny do podsložek.
cfg\	Složka určená pro vkládání konfiguračních souborů. Obsahuje ukázkový konfigurační soubor ve formátu Neon.
vendor\	Tato složka je plně spravována Composerem. Nachází se v ní veškeré závislosti, a to včetně frameworku samotného.
.gitignore	Soubor určený pro verzovací systém Git. Definuje, které složky a soubory se nemají verzovat.
composer.json	Soubor obsahující informace o projektu a seznam závislostí pro Composer.
composer.lock	Interní soubor Composeru. Jsou v něm obsaženy současné verze závislostí ve složce vendor.
index.php	Ukázkový skript určený pro spuštění WebSocket serveru.

Daná adresářová struktura není povinná, ale celý framework je na ní připraven. Chce-li programátor mít jednotlivé aplikační logiky rozděleny jinak, nic mu v tom nebrání. Může pak ale být potřeba upravit nastavení autoloadingu v souboru composer.json či jiné nastavení frameworku.

6.2 Struktura

Základem frameworku jsou tzv. cesty (routes). Cestou se rozumí stejnojmenná část URI, přes kterou se na server připojuje klient. Na každou cestu lze navázat konkrétní aplikační logiku. Tu tvoří tři typy tříd – `ConnectionManager`, `Receiver` a `Handler`. Jejich princip spolu s ukázkovým příkladem implementací je v následujících podkapitolách.

Další podstatnou součástí frameworku je message broker, který zajišťuje vzájemnou komunikaci mezi jednotlivými instancemi WebSocket serveru. Funguje jako jakási spojka mezi třídami typu Receiver konkrétní instance, ze které přijímá zprávu, a třídami typu Handler všech na něj připojených instancí.

6.2.1 ConnectionManager

Tato třída se stará o správu všech připojených klientů na danou instanci WebSocket serveru. Musí implementovat rozhraní `BearSys\WSF\ConnectionManagers\IConnectionManager`. Má celkem tři metody. První z nich je metoda `onOpen`, která je volána při připojení klienta na server. Další z nich je `onClose`. Ta se naopak volá při odpojení klienta. Poslední metodou je `onError`, která se volá poté, co nastane ve spojení s klientem nějaká chyba.

6.2.2 Receiver

Třída typu Receiver zpracovává zprávu od klienta. Musí implementovat rozhraní `BearSys\WSF\Receivers\IReceiver`, tudíž musí obsahovat metodu `onMessage`, která vrací implementaci rozhraní `BearSys\WSF\Receivers\IReceiverReturn`. Je nutné si uvědomit, že tělo této metody je voláno pouze na té instanci serveru, na kterou je připojen klient posílající zprávu. V této metodě je tedy vhodné spouštět akce, které potřebujeme vykonat pouze jednou.

Ideálním příkladem je uložení dat do databáze. Pokud bychom toto udělali až v třídě Handler, tak by data byla v databázi duplikována, a to tolikrát, kolik instancí serveru je spuštěných. Dalším příkladem může být validace dat. U ní je ale důvod, proč ji spouštět již v třídě Receiver, jiný. Zašle-li nám klient chybná data a my je zkontrolujeme již zde, tak nedojde k jejich odeslání na všechny běžící instance WebSocket serveru.

Dále musí Receiver implementovat metody `getHandler` a `getConnectionManager`, které vracejí instance Handleru a ConnectionManageru. Při tvorbě třídy Receiver doporučuji dědit z abstraktní třídy `BearSys\WSF\Receivers\BaseReceiver`, která obsahuje implementace těchto metod, kdy jsou Handler a ConnectionManager hledány ve stejném namespace a se stejným aplikačním prefixem. Pro Receiver `Example\Chat\ChatReceiver` je tedy použit `Example\Chat\ChatHandler` a `Example\Chat\ChatConnectionManager`. V případě, že programátor chce jinou strukturu aplikace, může tyto metody přetížit, případně vůbec nedědit a naprogramovat si je tak, jak potřebuje.

6.2.3 Handler

Handler je poslední třídou aplikační logiky. Když Receiver odešle zprávu na message broker, tak ji následně obdrží všechny třídy typu Handler napříč běžícími instancemi serveru. To znamená, že kód v třídě Handler se provede přesně tolikrát, kolik je spuštěných instancí serveru. V ideálním případě by takový kód měl pracovat pouze s klienty připojenými na konkrétní server. Implementace třídy Handler musí stavět na rozhraní `BearSys\WSF\Handlers\IHandler`. To požaduje jednu metodu, a to `onMessage`.

6.2.4 Message broker

Pro implementaci message brokeru je určeno rozhraní `BearSys\WSF\Brokers\IBroker`. To požaduje tři povinné metody. První z nich je `send`, ta se stará o odeslání zprávy na konkrétní message broker. Obsahuje tři parametry. Jelikož většina message brokerů, kteří podporují vzor PubSub, využívá kanály (channel) či témata (topic), tak je potřeba metodě poskytnout název kanálu či tématu, kterému se daná zpráva má odeslat. Druhým parametrem je samotná zpráva, která se má odeslat. Její formát může být jakýkoliv a je jen na programátorovi, zda jej bude nějak limitovat. Posledním parametrem, který je ovšem volitelný, je tzv. callback, který bude spuštěn po úspěšném odeslání zprávy.

Druhou povinnou metodou je `registerHandlers`. Ta přijímá dva parametry. Prvním z nich je seznam cest, které jsou jako klíče jednotlivých záznamů. Ve druhém seznamu jsou klíči speciální názvy témat či kanálů. V obou případech je hodnotou v seznamu konkrétní přiřazený Handler. Poslední povinnou metodou je `getChannelForHandler`, která získává jméno kanálu, na kterém je konkrétní Handler přihlášen k odběru zpráv.

6.3 Spouštěcí skript

V souboru `index.php` a také výpisu 3 lze najít ukázkový spouštěcí skript. Cílem frameworku je zachovat ho co nejjednodušší a nejčitelnější.

Výpis 3: *Ukázkový spouštěcí skript*

```
1 <?php
2 $loader = require_once(__DIR__ . '/vendor/autoload.php');
3 $loop = React\EventLoop\Factory::create();
4
5 $server = new BearSys\WSF\WebSocketServer($loop, $loader);
6 $server->addConfig(__DIR__ . '/config/example.neon');
7 $server->run();
```

Druhý řádek do skriptu vkládá autoloadovací skript vytvořený Composerem. Ten se uloží do proměnné `$loader` pro pozdější využití serverem. Následující řádek vytvoří instanci implementace `React\EventLoop\LoopInterface`. Lze zde vytvořit i konkrétní implementaci dle uvážení programátora, ale doporučuji využít tuto tzv. továrnu, která automaticky vybere nejlepší možnou implementaci na daném systému.

Následně se vytváří instance samotného serveru, která pro svůj běh vyžaduje dříve vytvořené instance. Této instanci pak lze přidávat konfigurační soubory pomocí metody `addConfig`. Nakonec

je server spuštěn metodou `run`. Veškerý kód po zavolání této metody je již ignorován, jelikož je spuštěna nekonečná smyčka.

6.4 Konfigurační soubory

Konfigurační soubory mohou mít různé formáty, v základu jsou podporovány Neon, XML, JSON a Yaml. Programátor si ale může vytvořit konfigurační soubory v jakémkoliv formátu, vytvoří-li k němu implementací rozhraní `BearSys\WSF\Configuration\Types\IType`. Tu pak před zavoláním metody `addConfig` zaregistruje pomocí metody `addConfigType`, do které jako parametr uvede název třídy s implementací. Příklad implementace bude uveden dále v této kapitole.

6.4.1 Ukázkový konfigurační soubor

Ukázkový konfigurační soubor se všemi možnými parametry je ve výpisu 4. Následuje podrobný popis jednotlivých sekcí a jejich parametrů.

Výpis 4: *Ukázkový konfigurační soubor se všemi možnými parametry ve formátu Neon*

```
server:
    host: localhost
    port: 8080
    address: 0.0.0.0

broker:
    type: BearSys\WSF\Brokers\Redis\Redis
    customHandlers:
            _keyevent@1_:expired: Example\TestHandler

routes:
    /test: Example\TestReceiver
    /restricted:
            receiver: Example\RestrictedReceiver
            allowedOrigins:
                    - 127.0.0.1
                    - 192.168.0.10

factories:
    - Example\MysqlFactory
```

Sekce `server` obsahuje informace o samotném WebSocket serveru. Parametr `host` obsahuje adresu, na kterou se budou klienti připojovat. Pro připojení na adresu `wss://example.com/test` je potřeba hodnotu parametru host nastavit na example.com. Tento parametr je povinný, stejně jako parametr `port`. Ten určuje, na jakém portu bude WebSocket server přijímat připojení. Je potřeba se ujistit, že je tento port volný. Posledním parametrem, který je ovšem nepovinný, je `address`. Ten určuje, na jaké IP adrese server poběží. Tento parametr je podstatný pro běh na serverech, které mají více síťových připojení. Při uvedení adresy `127.0.0.1` se na server půjde připojit pouze z počítače, na kterém běží. Pokud by daný počítač měl dvě externí IP adresy, tak lze nastavit pouze jednu z nich, skrz druhé připojení by se pak na server nedalo připojit. Doporučuji však nechat nastavení na `0.0.0.0`, serveru pak bude úplně jedno, skrz jakou IP adresu se klient připojuje.

Druhou sekcí je broker. Ta obsahuje informace specifické pro message broker, díky kterému bude moci server komunikovat s dalšími instancemi konkrétního WebSocket serveru. Parametr type určuje konkrétní implementaci rozhraní message brokeru (BearSys\WSF\Brokers\IBroker). Druhý parametr (customHandlers) umožňuje nastavit konkrétní implementace třídy Handler pro vlastní názvy kanálů (channel) či témat (topic). V příkladu je nastavena zpracovatelská třída pro notifikační kanál expirace klíče Redisu.

Další sekce s názvem routes obsahuje mapování cest z URI na třídy typu Receiver. V uvedeném příkladu to znamená, že když se klient připojí na URI wss://example.com/test, tak bude komunikovat s třídou Example\TestReceiver. K zápisu cesty je možno užít dva formáty. V obou je klíčem celá cesta, a to včetně úvodního lomítka. Ta musí splňovat požadavky URI na cestu, takže nesmí obsahovat například znaky ampersandu či otazníku. První možností je uvést jako hodnotu pouze plné jméno třídy Receiver. Druhou možností je využít asociativní pole s dvěma položkami. V klíči receiver pak musí být uvedeno plné jméno třídy Receiver. Klíč allowedOrigin pak obsahuje pole s povolenými adresami, ze kterých je možno se na danou cestu připojit. To se hodí, potřebujeme-li omezit připojení na konkrétní cestu pouze určeným klientům.

Poslední sekce pojmenovaná factories obsahuje seznam továren určených pro Dependency Injection Container, který bude popsán v následující podkapitole.

6.5 Dependency Injection

Využití vzoru dependency injection nám zaručí, že každá třída, která se v rámci frameworku inicializuje, dostane přesně ty závislosti, které potřebuje. O to se stará takzvaný kontejner, neboli třída BearSys\WSF\DI\Container. Ten se stará o veškerou správu závislostí, programátorovi stačí u jeho tříd pouze uvést požadované závislosti v konstruktoru. Pokud potřebuje získat z kontejneru třídu dynamicky, stačí si vyžádat samotný kontejner a následně na něm kdekoliv v kódu zavolat metodu get, která jako parametr přijímá název požadované třídy.

Framework si sám inicializuje několik druhů tříd. Jsou jimi všechny třídy aplikační logiky (Receiver, Handler a ConnectionManager), které jsou přiřazeny k nějaké cestě. Dále je to také instance zvoleného message brokeru. Ještě předtím však je do kontejneru frameworkem vloženo několik pro jeho běh nezbytně nutných tříd. Jsou jimi BearSys\WSF\Configuration\Configurator, který se stará o načítání a parsování konfiguračních souborů. Dále je to implementace LoopInterface a nakonec také instance Composer autoloaderu. Posledním typem automaticky načítaných tříd jsou takzvané továrny, neboli factories.

6.5.1 Factories

Továrny jsou určeny k vytvoření takových typů tříd, které pomocí klasického kontejneru vytvořit nelze. Jedná se především o třídy, které v konstruktoru vyžadují nějaký ze základních datových typů, pole nebo nějaký jiný seznam. Takováto továrna musí implementovat rozhraní BearSys\WSF\DI\IFactory, které obsahuje jedinou metodu create. Ta jako parametr od dependency injection kontejneru dostává kompletní konfiguraci. Následně pak musí vrátit nějaký vytvořený objekt, který si následně kontejner uloží pro použití v dalších třídách. Příklad použití je ve výpisu 5.

Skvělým příkladem použití továren je připojení k MySql databázi, které vyžaduje údaje potřebné k připojení ve formě pole. Ukázka implementace takovéto továrny je ve výpisu 5. Jak si lze všimnout, tak tato továrna si bere údaje k databázi ze sekce `mysql` v konfiguračním souboru. Nakonec vrací instanci `React\MySLQ\Connection`. Ta, kromě údajů k databázi, vyžaduje i `LoopInterface`. Tu vyžaduji v konstruktoru a dependency injection kontejner mi implementaci tohoto rozhraní předá.

Výpis 5: *Ukázková implementace továrny pro dependency injection*

```
 1 namespace Example\Factories;
 2
 3 use BearSys\WSF\DI\IFactory;
 4 use React\EventLoop\LoopInterface;
 5 use React\MySQL\Connection;
 6
 7 class MysqlFactory implements IFactory
 8 {
 9     protected $loop;
10
11     public function __construct(LoopInterface $loop)
12     {
13             $this->loop = $loop;
14     }
15
16     public function create(\stdClass $configuration)
17     {
18             $config = [
19                     'host'   => $configuration->mysql->host ?: 'localhost',
20                     'port'   => $configuration->mysql->port ?: 3306,
21                     'user'   => $configuration->mysql->user ?: 'root',
22                     'passwd' => $configuration->mysql->pass ?: '',
23                     'dbname' => $configuration->mysql->db   ?: '',
24             ];
25             return new Connection($this->loop, $config);
26     }
27 }
```

6.6 Ukázková aplikační logika

V této podkapitole bude ukázána a vysvětlena aplikační logika na příkladu chatovací aplikace. Nejprve je vhodné vytvořit ve složce `app` podsložku, ve které bude veškerá logika obsažena. Doporučuju ji pojmenovat takovým způsobem, aby její název odpovídal prefixům jednotlivým třídám aplikační logiky. V tomto příkladu se jmenuje `Chat`. V poslední podkapitole pak ukážu vzorovou implementaci message brokeru, a to konkrétně Redisu.

6.6.1 ChatConnectionManager

Jelikož uživatelé v ukázkové aplikaci budou moci posílat zprávy jiným uživatelům, a to dle jejich uživatelského jména, tak je potřeba vytvořit objekt, který bude mít uloženy připojení právě dle uživatelského jména (to klient při připojení zadá do query parametru v URI). K tomu je ideální klasické pole (v příkladu `$users`), kde klíčem bude uživatelské jméno. Hodnoty pak budou obsahovat instance `SplObjectStorage`, které v sobě budou mít jednotlivé připojení. Jednotlivé metody implementující rozhraní `IConnectionManager` se pak starají o ukládání či mazání připojení do a z

pole. Poslední veřejná metoda getConnectionsByUsername vrací všechny připojení dle uživatelského jména a je vhodná především pro použití v třídě Receiver. Kód implementace je ve výpisu 6.

Výpis 6: *Ukázková implementace rozhraní IConnectionManager*

```
use BearSys\WSF\ConnectionManagers\IConnectionManager;
use GuzzleHttp\Psr7\Request;
use Ratchet\ConnectionInterface;

class ChatConnectionManager implements IConnectionManager
{
    /** @var \SplObjectStorage[] */
    protected $users = [];

    public function onOpen(ConnectionInterface $conn)
    {
            $username = $this->getUsername($conn);
            if (!array_key_exists($username, $this->users))
                    $this->users[$username] = new \SplObjectStorage;
            $this->users[$username]->attach($conn);
    }

    public function onClose(ConnectionInterface $conn)
    {
            $username = $this->getUsername($conn);
            $this->users[$username]->detach($conn);
    }

    public function onError(ConnectionInterface $conn, \Exception $e)
    {
            $username = $this->getUsername($conn);
            $this->users[$username]->detach($conn);
    }

    public function getConnectionsByUsername(string $username): ?\SplObjectStorage
    {
            if (array_key_exists($username, $this->users))
                    return $this->users[$username];
            else
                    return NULL;
    }

    public function getUsername(ConnectionInterface $conn)
    {
            /** @var Request $request */
            $request = $conn->httpRequest;
            $query = $request->getUri()->getQuery();
            $matches = [];
            if (preg_match('/username=([a-zA-Z0-9]+)(&|$)/', $query, $matches))
                    return $matches[1];
            else
                    return NULL;
    }
}
```

6.6.2 ChatReceiver

Při přijmutí zprávy od klienta potřebuji provést několik věcí. Nejprve musím zvalidovat, zda je daná zpráva správně strukturovaná. Především potřebuji mít dvě informace, a to uživatelské jméno

příjemce a text zprávy. Pokud je mám, tak mohu zprávu odeslat na message broker, což provedu vrácením třídy `Forward`, která je implementací rozhraní `IReceiverReturn`. Jako informaci pro příjemce také zahrnu uživatelské jméno odesílatele. Ukázková implementace se nachází ve výpisu 7.

Výpis 7: *Ukázková implementace rozhraní IReceiver*

```
use BearSys\WSF\Brokers\IBroker;
use BearSys\WSF\ConnectionManagers\IConnectionManager;
use BearSys\WSF\DI\Container;
use BearSys\WSF\Handlers\IHandler;
use BearSys\WSF\Receivers\IReceiver;
use BearSys\WSF\Receivers\IReceiverReturn;
use BearSys\WSF\Receivers\Forward;
use BearSys\WSF\Receivers\Finish;
use Ratchet\ConnectionInterface;

class ChatReceiver implements IReceiver
{
    /** @var Container */
    protected $dic;

    public function __construct(Container $dic)
    {
            $this->dic = $dic;
    }

    public function onMessage(ConnectionInterface $connection, $src): IReceiverReturn
    {
            $srcObject = json_decode($src);
            if (!$srcObject->to || !$srcObject->message)
                    return new Finish;
            $srcObject->from = $this->getConnectionManager()->getUsername($connection);
            return new Forward(json_encode($srcObject));
    }

    public function getHandler(): IHandler
    {
            $handlerFQN = substr(get_class($this), 0, -8) . 'Handler';
            $handler = $this->dic->get($handlerFQN);
            if (!$handler instanceof IHandler)
                    throw new \Exception;
            return $handler;
    }

    public function getConnectionManager(): IConnectionManager
    {
            $conMgrFQN = substr(get_class($this), 0, -8) . 'ConnectionManager';
            $conMgr = $this->dic->get($conMgrFQN);
            if (!$conMgr instanceof IConnectionManager)
                    throw new \Exception;
            return $conMgr;
    }
}
```

V příkladu lze také vidět ukázkovou implementaci metod `getHandler` a `getConnectionManager`. Jejich implementace je shodná s dříve zmíněnou abstraktní třídou `BaseReceiver`, od které doporučuji všechny třídy typu Receiver dědit.

6.6.3 ChatHandler

Poslední třídou aplikační logiky je ChatHandler. Ta se stará o poslání zprávy příjemci. Zde si z třídy typu Connection Manager získám všechny aktivní připojení adresáta a na každé z nich odešlu zprávu spolu s uživatelským jménem odesílatele. Je nutno podotknout, že na dané instanci serveru může mít adresát několik různých připojení, stejně jako nemusí mít žádné. Ukázková implementace je ve výpisu 8.

Výpis 8: *Ukázková implementace rozhraní IHandler*

```
use Ratchet\ConnectionInterface;

class ChatHandler implements \BearSys\WSF\Handlers\IHandler
{
    /** @var ChatConnectionManager */
    protected $conMgr;

    public function __construct(ChatConnectionManager $conMgr)
    {
            $this->conMgr = $conMgr;
    }

    public function onMessage(string $message): void
    {
            $message = json_decode($message);

            $connections = $this->conMgr->getConnectionsByUsername($message->to);

            if ($connections)
                    /** @var ConnectionInterface $connection */
                    foreach ($connections as $connection)
                            $connection->send(json_encode([
                                    'from' => $message->from,
                                    'message' => $message->message,
                            ]));
    }
}
```

6.6.4 Implementace message brokeru

Jako ukázku implementace message brokeru jsem si zvolil Redis, jelikož s ním mám největší zkušenosti a ve svých aplikacích jej již používám pro kešování. Jeho omezením při využití vzoru PubSub je to, že není možné z jednoho klienta zároveň odebírat zprávy a přitom je také posílat. Dokonce nejde skrz klienta odebírajícího zprávy provádět ani žádné jiné příkazy. Proto je potřeba vytvořit dvě instance Redis klienta. Ukázková implementace se nachází ve výpisu 9.

Výpis 9: *Ukázková implementace message brokeru*

```
use Clue\React\Redis\Client;
use React\Promise\PromiseInterface;

class Redis implements IBroker
{
    const DEFAULT_CONFIG = [
            'protocol' => 'redis',
            'host' => 'localhost',
            'port' => 6379,
            'dbIndex' => 0,
    ];

    /** @var RedisClient */
    protected $publisher;

    /** @var RedisClient */
    protected $subscriber;

    /** @var Container */
    protected $dic;

    public function __construct(Container $dic, LoopInterface $loop, Configurator
$configurator)
    {
            $configuration = $configurator->getConfiguration();
            if (!$configuration->redis)
                    $redisConfig = (object) self::DEFAULT_CONFIG;
            else
                    $redisConfig = $configuration->redis;

            $this->publisher = $this->createClient($loop, $redisConfig,
RedisClient::ROLE_PUBLISHER);
            $this->subscriber = $this->createClient($loop, $redisConfig,
RedisClient::ROLE_SUBSCRIBER);
            $this->dic = $dic;

            $dic->add($this->publisher);
    }

    public function send(string $channel, string $message, callable $callback = NULL):
void
    {
            $this->publisher->publish($channel, $message)->then($callback);
    }

    public function registerHandlers(\stdClass $routes, \stdClass $customHandlers):
void
    {
            /** @var $channels array <string, IHandler> */
            $channels = [];

            foreach ($routes as $path => $handler) {
                    $channels[$path] = $handler;
            }

            foreach ($customHandlers as $channel => $handler) {
                    if (array_key_exists($channel, $channels))
                            throw new \Exception(); // TODO
```

```
54                      $handler = $this->dic->get($handler);
55                      $channels[$channel] = $handler;
56              }
57
58              foreach ($channels as $channel => $handler) {
59                      if (!$handler instanceof IHandler)
60                              throw new \Exception(); // TODO
61
62                      $this->subscriber->subscribe($channel);
63                      $this->handlers[get_class($handler)] = $channel;
64              }
65
66              $this->subscriber->on('message', function ($channel, $message) use
    ($channels) {
67                      $channels[$channel]->onMessage($message);
68              });
69      }
70
71      public function getChannelForHandler(IHandler $handler): string
72      {
73              return $this->handlers[get_class($handler)];
74      }
75
76      protected function createClient(LoopInterface $loop, \stdClass $redisConfig, string
    $role): RedisClient
77      {
78              $factory = new Factory($loop);
79              $target = ($redisConfig->protocol ?: 'redis') . '://' . ($redisConfig->host
    ?: 'localhost') . ':' . ($redisConfig->port ?: 6379) . '/' . ($redisConfig->dbIndex ?:
    0);
80              $promise = $factory->createClient($target);
81              return new RedisClient($promise, $redisConfig, $role);
82      }
83  }
```

V tomto příkladu používám knihovnu *clue/redis-react*, nad kterou jsem si napsal vlastní *dekorátor*, který slouží k jednoduššímu volání jednotlivých příkazů. Ten je v příkladu pojmenován `RedisClient`.

Na řádku 34 přidávám do dependency injection kontejneru pouze instanci klienta, který je určen pro zasílání zpráv na Redis. Kdybych do kontejneru přidal klienta určeného pro odebírání zpráv z Redisu, tak by jej nešlo použít pro většinu Redis příkazů, například nejpoužívanější příkazy `get` a `set`.

6.6.5 Autoloading

Nakonec je potřeba správně nastavit autoloading v souboru `composer.json`. V sekci `autoload` se v objektu `psr-4` musí správně nasměrovat jmenný prostor, ve kterém jsou třídy aplikační logiky, na složku, která obsahuje soubory se zdrojovými kódy těchto tříd. V mém případě jsou třídy umístěny ve jmenném prostoru `Example\App\Routes\Chat` a jsou obsaženy ve složce `app\Routes\Chat`. Správný zápis pak vypadá takto:

```
"Example\\App\\": "app"
```

Následně je potřeba spustit příkaz, který toto nasměrování zapíše do souborů autoloadingu Composeru:

```
composer dump-autoload
```

6.7 Spuštění a otestování funkčnosti serveru

Samotné spuštění serveru je velmi jednoduché. Stačí pouze spustit PHP a jako parametr mu poskytnout cestu ke spouštěcímu souboru:

```
php example.php
```

Pro otestování funkčnosti serveru je toto sice dostačující, ale pro použití serveru v produkčním prostředí je téměř nutností server spouštět jako službu, kterou operační systém sám spustí a při případném problému, kdy dojde k vypnutí serveru, jej opět zapne. Toto je ovšem téma spíše pro systémové administrátory a vysvětlování všech možností, jak toto provést, je mimo možnosti této příručky.

6.7.1 Spuštění dvou instancí serveru a nastavení škálování

Před spuštěním dvou instancí serveru musíme nastavit údaje k message brokeru. Nejprve je potřeba mít spuštěný Redis. Následně je potřeba nastavit, že má server použít právě Redis, a to v sekci konfiguračního souboru s názvem `broker`. Do parametru `type` je potřeba vyplnit celou cestu k implementaci rozhraní, a tou je `BearSys\WSF\Brokers\Redis\Redis`. Údaje k připojení k Redis serveru následně stačí přidat na konec konfiguračního souboru:

```
redis:
    host: localhost
    port: 6379
```

Pro spuštění dvou instancí na jednom počítači je potřeba vytvořit dva samostatné konfigurační soubory (`example1.neon` a `example2.neon`). Ty se budou lišit pouze zadaným portem serveru, který bude v jednom případě 8080 a v druhém 8081. Pak je potřeba ještě upravit spouštěcí soubor tak, aby pro každou instanci zvolil jiný konfigurační soubor, který bude předán jako parametr:

```
$app->addConfig(__DIR__ . '/cfg/' . $argv[1] . '.neon');
```

Následně by neměl být problém spustit dvě samostatné instance serveru, které se okamžitě připojí na message broker a jsou připraveny přijímat připojení od klientů. Následující příkazy je nutno zadat do samostatných terminálů:

```
php example.php example1
php example.php example2
```

6.7.2 Testovací JavaScript klient

Pro potřeby otestování serveru jsem připravil jednoduchý klient využívající JavaScriptové WebSocket API. Ten vytvoří dvě WebSocket připojení, přičemž jedno bude připojeno na server na portu 8080 a druhé na server běžícím na portu 8081. Následně první klient vytvoří a odešle zprávu pro druhého klienta. Ten po jejím obdržení zprávu vypíše do konzole prohlížeče. Tím, že jsou klienti připojení na rozdílných serverech, se ověří, že přeposílání zpráv z třídy typu Receiver jednoho serveru na Handler druhého serveru skrz message broker funguje. Kompletní kód klienta je ve výpisu 10.

Výpis 10: *JavaScriptový kód pro otestování funkčnosti dvou běžících instancí WebSocket serveru*

```
1  var ws1 = new WebSocket('ws://localhost:8080/chat?username=username1');
2  var ws2 = new WebSocket('ws://localhost:8081/chat?username=username2');
3
4  ws1.onopen = function (event) {
5      var data = {
6             'to': 'username2',
7             'message': 'test message'
8      };
9      ws1.send(JSON.stringify(data));
10 };
11
12 var onMessage = function (n) {
13     return function (event) {
14            var data = JSON.parse(event.data);
15            console.log('Data to #' + n + ': Message from ' + data.from + ': ' +
   data.message);
16     }
17 };
18
19 ws1.onmessage = onMessage(1);
20 ws2.onmessage = onMessage(2);
```

Po spuštění skriptu lze přes konzoli prohlížeče posílat libovolné zprávy mezi klienty:

```
ws2.send(JSON.stringify({'to': 'username1', 'message': 'Zpráva pro prvního klienta'}))
```

Závěr

Hlavním cílem práce bylo navrhnout a vytvořit framework pro tvorbu škálovatelných WebSocket serverů v PHP, který by umožnil programátorům vytvořit WebSocket server jednodušeji a efektivněji, než se současnými nástroji. Ty totiž postrádají důležité funkcionality a jejich použití je mnohdy krkolomné. Především žádný z nich nenabízí jednoduchý způsob dosažení škálování. Dalším problémem současných řešení je také nutnost psaní spousty neobjektového kódu, jehož množství by mělo být pokud možno naprosto minimální.

Dosažení vytyčených dílů

Hlavní cíl byl naplněn za pomocí dílčích cílů:

- Zdůvodnil jsem vhodnost použití PHP pro WebSocket servery. Při analýze jsem zjistil, že PHP má jasně nejvyšší podíl využití na webových stránkách ze všech serverových jazyků.
- Analyzoval jsem současná řešení pro tvorbu PHP aplikací v reálném čase. Z analýzy vyplynulo, že ačkoli jsou tyto řešení kvalitní a řeší širokou škálu problémů, tak jsou mnohdy nevhodné pro tvorbu WebSocket serverů, či jsou pro ni až příliš složité.
- Vysvětlil jsem přínosy využití WebSocket serverů pro webové stránky, které potřebují funkcionalitu vyžadující komunikaci v reálném čase. Dále jsem popsal i případné problémy, které využití PHP pro tvorbu WebSocket serveru přináší.
- Navrhl jsem strukturu frameworku pro jednoduchou a efektivní tvorbu škálovatelných WebSocket serverů v PHP.
- Vybral jsem komponenty, které jsem následně pro tvorbu frameworku použil.
- Napsal jsem uživatelskou příručku, podle které může libovolný PHP programátor vytvořit vlastní WebSocket server.

Přínos práce

Z analýzy současného stavu řešení pro tvorbu WebSocket serverů v PHP vyšlo najevo, že současný stav těchto řešení není dostatečný. Cílem každé vývojové firmy či programátora je vytvořit určitou funkcionalitu co nejjednodušeji a nejrychleji, a to současné řešení neumožňují. Vytvořením otevřeného frameworku pro tvorbu WebSocket serverů v PHP jsem umožnil jakémukoliv vývojáři vytvořit takový server během pouhých několika hodin. Při tom se nebude muset zabývat řešením škálování či struktury a může se plně soustředit na aplikační logiku. Framework je uveřejněn na portále GitHub (repozitář *BearSysCZ/WebSocketFramework*) a je dostupný skrz správce závislostí Composer (balíček *bearsys/wsf*).

Terminologický slovník

Termín	Význam [zdroj]
Klient	Webový prohlížeč koncového uživatele, služba nebo jakákoliv jiná aplikace, která se připojuje k serveru.
Reálný čas	Minimální možný čas nutný k výměně informací mezi klientem a serverem ve webové komunikaci.
Neblokující kód	Takový kód, na jehož dokončení se nečeká a rovnou se spustí kód, který po něm následuje.
Asynchronní programování	Takové programování, kde se využívá více vláken, které tak neblokují kód prováděný v ostatních vláknech (viz https://visualstudiomagazine.com/articles/2011/03/24/wccsp_asynchronous-programming.aspx).
Aplikační logika	Ta část kódu, která se stará o zpracování příchozí WebSocket zprávy.
Publisher-Subscriber (též PubSub)	Vzor pro komunikaci klientů se serverem, kdy se klienti přihlásí k odebírání zpráv ze serveru. Ten pak případnou příchozí zprávu rozešle mezi všechny přihlášené klienty.
ConnectionManager	Typ třídy spadající pod aplikační logiku frameworku. Stará se o správu připojených klientů k WebServeru. Viz kapitola 5.1.
Receiver	Typ třídy spadající pod aplikační logiku frameworku. Přijímá zprávy od klientů, zpracovává je a případně je pak posílá skrz message broker na ostatní instance WebSocket serveru. Viz kapitola 5.1.
Handler	Typ třídy spadající pod aplikační logiku frameworku. Dostává zprávy od všech instancí třídy typu Receiver napříč běžícími instancemi WebSocket serveru a dále je zpracovává. Viz kapitola 5.1.
Škálovatelnost	Možnost spustit více instancí konkrétního serveru tak, aby bylo možno obsloužit všechny připojující se klienty bez výpadků či prodlev.
Composer	Správce závislostí určený pro použití v PHP.
Packagist	Zdroj balíčků pro Composer.
GitHub	Největší světový git repozitář, který je primárně určen pro sdílení otevřeného softwaru.

Použité informační zdroje

ARORA, Guarav Kumar. 2015. The SOLID Principles. [online] Dostupné z: https://www.c-sharpcorner.com/ebooks/the-solid-principles.

AUGUSTÍN, Jakub. 2012. *Porovnanie efektivity technológií WebSocket a AJAX při implementácii kolaboratívneho editoru.* VŠE FIS KIT.

BÖHMER, Marian. 2012. *Návrhové vzory v PHP*. Computer Press. ISBN 978-80-251-3338-5.

BUGZILLA. 2013. Increase number of permitted EventSource connections. [online] Dostupné z: https://bugzilla.mozilla.org/show_bug.cgi?id=906896 [cit 2018-03-22].

BULVAS, Vít. 2012. *WebSockets - technologie pro tvorbu real time webových aplikací.* VŠE FIS KIT.

CAN I USE. 2018a. *Server-sent events support table.* [online] Dostupné z: https://caniuse.com/#feat=eventsource [cit 2018-03-22].

CAN I USE. 2018b. *Web Sockets support tables.* [online] Dostupné z: https://caniuse.com/#feat=websockets [cit 2018-03-22].

GITHUB. 2015. *Ratchet – Limited to 1024 concurrent connections.* [online] Dostupné z: https://github.com/ratchetphp/Ratchet/issues/300 [cit 2018-03-24].

GITHUB. 2018a. *Repository search result for term WebSockets.* [online] Dostupné z: https://github.com/search?utf8=%E2%9C%93&q=websocket&type=Repositories&ref=advsearch&l=&l [cit 2018-03-17].

CHROMIUM BUGS. 2013. Limit of 6 concurrent EventSource connections is too low. [online] Dostupné z: https://bugs.chromium.org/p/chromium/issues/detail?id=275955 [cit 2018-03-22].

INTERNET ENGINEERING TASK FORCE. 2011. *RFC 6455 – The WebSocket Protocol.* [online] Dostupné z: https://tools.ietf.org/html/rfc6455. [cit 2018-03-13].

LENGSTORF, Jason; LEGGETTER, Phil, NEWMAN, Alex. 2013. *Realtime Web Apps*. Apress. ISBN: 978-1-4302-4621-3.

PACKAGIST. 2018a. Package amphp/amp. [online] Dostupné z: https://packagist.org/packages/amphp/amp [cit 2018-03-18].

PACKAGIST. 2018b. Package kraken-php/kraken. [online] Dostupné z: https://packagist.org/packages/kraken-php/kraken [cit 2018-03-18].

PACKAGIST. 2018c. *Package react/react.* [online] Dostupné z: https://packagist.org/packages/react/react [cit 2018-03-18].

PACKAGIST. 2018d. *Package workerman/workerman.* [online] Dostupné z: https://packagist.org/packages/workerman/workerman [cit 2018-03-18].

PACKAGIST. 2018e. *Package react/promise.* [online] Dostupné z: https://packagist.org/packages/react/promise [cit 2018-03-18].

PACKAGIST. 2018f. *Package react/event-loop.* [online] Dostupné z: https://packagist.org/packages/react/event-loop [cit 2018-03-18].

PECL. 2018a. *Package libevent.* [online] Dostupné z: https://pecl.php.net/package/libevent [cit 2018-03-28].

PECL. 2018b. *Package ev.* [online] Dostupné z: https://pecl.php.net/package/ev [cit 2018-03-28].

RATCHET. 2018a. *Ratchet – PHP WebSockets.* [online] Dostupné z: http://socketo.me/ [cit 2018-03-19].

RATCHET. 2018b. *Deployment – Open Connection Limit.* [online] Dostupné z: http://socketo.me/docs/deploy [cit 2018-03-24].

SERVERFAULT. 2011. *Windows Server 2008 R2 max open files limit*. [online] Dostupné z: https://serverfault.com/questions/249477/windows-server-2008-r2-max-open-files-limit [cit 2018-03-24].

TECHNOPEDIA. 2018. *What is a Message Broker*. [online] Dostupné z: https://www.techopedia.com/definition/16959/message-broker [cit 2018-04-01].

TOMAN, František. 2012. *Real-time webové aplikace*. VŠE FIS KIT.

WANG, Vanessa; MOSKOVITS, Peter; SALIM, Frank. 2013. *The Definitive Guide to HTML5 WebSocket.* Apress. ISBN: 978-1-4302-4741-8.

W3C. 2012. *The WebSocket API.* [online] Dostupné z: https://www.w3.org/TR/websockets/ [cit 2018-03-13].

W3TECHS. 2018a. *Historical trends in the usage of server-side programming languages for websites.* [online] Dostupné z: https://w3techs.com/technologies/history_overview/programming_language [cit 2018-03-17].

W3TECHS. 2018b. Usage statistics and market share of PHP for websites. [online] Dostupné z: https://w3techs.com/technologies/details/pl-php/all/all [cit 2018-04-01].

Rejstřík

— A —

Aerys, 19
alternativy websocketů
  AJAX, 11, 14, 21, 22, 51
  HTML5 server sent events, 22
  long polling, 21
Amp, 18, 19
Asynchronní PHP, 23

— C —

Composer, 16, 18, 28, 36, 38, 40

— D —

databáze
  ElasticSearch, 18
  MySQL, 18, 41
  Redis, 18, 25, 26, 32, 35, 39, 44, 45, 46, 47

— F —

frameworky
  Nette, 32, 33
  Symfony, 35

— G —

GitHub, 17, 18, 50

— H —

HTTP, 18, 19, 22, 54

— J —

JavaScript, 54
JSON, 29, 39, 48

— K —

Kraken, 18, 19, 20

— N —

Neon, 10, 29, 31, 35, 36, 39

— P —

Packagist, 16, 18, 50
PHP, 4, 5, 11, 12, 14, 16, 17, 18, 19, 21, 22, 23, 24, 28, 29, 32, 34, 35, 36, 47, 49, 50, 52, 54
programovací jazyky
  ASP.NET, 16, 17
  ColdFusion, 17
  Java, 16, 17
  JavaScript, 15, 16, 17, 18, 21, 36, 48
  Perl, 17
  Python, 17
  Ruby, 17
PubSub, 24, 25, 32, 35, 38, 44

— R —

Ratchet, 19, 23, 24, 34, 51, 52
ReactPHP, 18, 19, 23, 28, 54

— S —

systémy pro správu obsahu
  Drupal, 16
  Joomla, 16
  PrestaShop, 16
  WooCommerce, 16
  Wordpress, 16

— Š —

Škálování, 24

— U —

URI, 31, 37, 40, 41

— W —

Workerman, 19

— X —

XML, 21, 29, 35, 39

— Y —

YAML, 29, 35