Česká zemědělská univerzita v Praze

Provozně ekonomická fakulta

Katedra informačního inženýrství

Bakalářská práce

Srovnání výkonu PHP a Javascript při vývoji RESTful API

Jan SOBĚSLAV

Čestné prohlášení

Prohlašuji, že svou bakalářskou práci "Srovnání výkonu PHP a Javascript při vývoji RESTful API" jsem vypracoval samostatně pod vedením vedoucího bakalářské práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou citovány v práci a uvedeny v seznamu použitých zdrojů na konci práce. Jako autor uvedené bakalářské práce dále prohlašuji, že jsem v souvislosti s jejím vytvořením neporušil autorská práva třetích osob.

V Praze dne 15.3.2017 _____

Poděkování

Děkuji Ing. Jiřímu Brožkovi, Ph.D. za vedení práce a poskytnuté konzultace a podněty.

Dále děkuji kolegovi Ondřeji Hoosovi za zapůjčení stroje a za oporu při prvním měření.

Zvláštní poděkování patří též Pánu Ivo Krátkému od Kostečků.

Abstrakt a klíčová slova

Srovnání výkonu PHP a Javascript při vývoji RESTful API

Abstrakt

Tato práce se zabývá problematikou výkonu dvou programovacích jazyků v prostředí webových aplikací, a sice PHP a JavaScript. Specificky je měřena rychlost vykonávání požadavků kladených na webové aplikačních rozhraní v těchto jazycích naprogramovaných.

V teoretické části práce také autor rozebírá pravidla návrhu tzv. RESTful API a specifika obou jazyků při realizaci těchto návrhů.

Klíčová slova

API, REST, RESTful API, PHP, JavaScript, měření výkonu

Comparing the performance of PHP and Javascript in developing RESTful API

Abstract

This work compares the performance of two programming languages used in web application developement, namely PHP and JavaScript. Specifically author measures execution time required for solving reqests sent to web APIs programmed in these languages.

The theoretical part of the thesis also analyzes the design rules of so-called RESTful API and the specifics of both languages in the implementation of these designs.

Keywords

API, REST, RESTful API, PHP, JavaScript, performance comparison

Obsah

Seznam tabulek

Seznam grafů a ilustrací

Seznam ukázek kódu

1 Úvod

Vývoj webu skáče mílovými skoky kupředu. Zárodek internetu vznik v roce 1969, HTTP protokol a první statický web v roce 1990, první dynamicky generované webové stránky vznikají kolem roku 1995 a v novém miléniu již interaktivní webové aplikace běžně nahrazují ty desktopové.

Je to zajímavý, ale logický krok: každý počítač s předinstalovaným operačním systémem se dostává zákazníkovi do rukou s připraveným prohlížečem webu a webové aplikace nejsou závislé na operačním systému a dokonce ani na prohlížeči. Web se tedy stal nejjednodušší variantou, jak poskytnout dodávanou aplikaci nejširšímu okruhu uživatelů a to navíc s nespornou ekonomickou výhodou, neboť není zapotřebí vyvíjet více mutací aplikace pro každý operační systém, nebo dokonce typ zařízení.

Navíc ani vývojáři sami nejsou omezeni výběrem programovacího jazyka: ten je použit jenom proto aby generoval komunikaci v přesně vymezeném protokolu, zkrze který server komunikuje s uživatelovým prohlížečem.

Web vytvořil dokonalou abstraktní vrstvu mezi vyvíjenou aplikací a uživatelským strojem, skrze kterou jednotlivé strany nemohou[1], ne nepodobně Turingovu testu, zjistit která technologie pohání stranu druhou. Prohlížečům nezáleží, jestli je webová aplikace napsána v .NET, nebo PHP a server stejně tak nezajímá, jestli se výsledný kód odesílá prohlížeči běžícím na OS Windows, Mac OS, nebo Android.

S růstem dostupnosti internetu ho navíc začínají používat i aplikace, které ho k vlastní funkci vůbec nepotřebují. Příkladem budiž Google Dokumenty: textové procesory jsou standardním vybavením operačních systémů, ale přidaná hodnota Google Dokumentů tkví v možnostech sdílení a zálohování v cloudu. Potřeba sdílení dat – relativně nově nejen mezi uživateli, ale také mezi různými zařízeními jednoho uživatele – je jeden z faktorů rostoucí popularity webových řešení na úkor nativních aplikací.

Stali jsme se tedy svědky zajímavého procesu, kdy se ze statických webů staly weby interaktivní a ty následně přerostly v plnohodnotné aplikace. Přesto se ale k jejich

1 ...ani nemusí, nepotřebují to a obvykle je to ani nezajímá

programování stále využívá stejných technologií. To vede část programátorů k hledání vhodnějšího jazyka a tím se má stát JavaScript, jehož agilní, event-driven filozofie má lépe odrážet požadavky nových projektů.

Rozvoj interakce a potřeba sdílení ale vnesly do vývoje webu ještě novou výzvu. Její druhou stranou se záhy stal i rozvoj mobilních zařízení.

Tou výzvou je poskytování obsahu nikoliv uživateli, nýbrž dalšímu skriptu či programu. Webové i mobilní aplikace potřebují společný bod, kde budou uchovávat a sdílet svá data a to posílilo zájem o kvalitní návrhy aplikačních rozhraní poskytujících (a v některých případech přijímajících) data a metody právě prostřednictvím webu. API představují další druh webu, který už není vůbec stvořen pro to aby byl člověkem přímo čten, ani ovládán.

Stejně jako v případě ostatních technologií, i ve světě API existuje touha po sjednocení, normalizaci jejich návrhů a jako nejoblíbenější z návrhových vzorů se aktuálně jeví architektura REST Roye Fieldinga, jednoho z autorů klíčového HTTP protokolu a projektu Apache server.

A právě návrh RESTful web API má v této práci prověřit síly na poli webu stále nejčastěji užívaného jazyka, PHP, a jeho vyzyvatele, JavaScriptu v populárním moderním prostředí Node.js.

2 Cíl práce a metodika

2.1 Cíl práce

Hlavním cílem práce je zhodnotit rozdíly ve výkonu dvou programovacích jazyků: stabilního a obecně zažitého PHP a relativně nového a slibného server-side JavaScriptu, při návrhu tzv. RESTful webových aplikačních rozhraní.

Nulová hypotéza dává za pravdu zastáncům JS a tvrdí, že event-driven JavaScript lépe využije potenciál serveru a tudíž dokáže zpracovat všechny požadavky rychleji než PHP.

Alternativní hypotéza říká, že mezi oběma platformami neexistuje statisticky významný rozdíl v čase potřebném k vykonání stejně náročných operací, a to při žádném reálně dosažitelném množství paralelních požadavků.

Vedlejším cílem je zmapovat teoretická východiska potřebná pro realizaci RESTful web API v obou jazycích.

2.2 Metodika

Srovnání bude probíhat v následujících bodech:

1) Naprogramování dvou virtuálně shodných API (jedno naprogramované v PHP a jedno v JavaScriptu, specificky v prostředí Node.js) a jejich nasazení na serveru v lokální síti, případně u stejného poskytovatele webhostingu, nebo VPS. Na tyto API jsou kladeny tyto požadavky:

- Obě API budou nasazeny ve stejných laboratorních podmínkách, aby rozdíly při transportu dat neovlivňovaly výsledky měření.
- Obě API budou přijímat shodné požadavky a vracet stejné návratové hodnoty.
- Obě API budou užívat stejnou SQL databázi, na kterou budou pokládat naprosto shodné dotazy aby databázová vrstva nemohla zkreslovat výsledky měření.

2) Měření času nutného k vykonání vzrůstajícího počtu požadavků na obě API za následujících podmínek:

- Každá série měření obsahuje pět metod manipulace dat, které jsou od API vyžadovány: create, read, update, (znovu) read a delete.
- Počty požadavků narůstají ve 20 sériích podle Fibonacciho posloupnosti (která byla zvolena jako kompromis mezi lineárním a exponenciálním růstem) tzn.: 10, 20, 30, 50, 80, 130, 210, 340, 550, 890, 1440, 2330, 3770, 6100, 9870, 15970, 25840, 41810, 67650 a 109460 paralelních požadavků.
- Všechna měření jsou pro potřeby eliminace statistické chyby pětkrát opakována.
- Samozřejmě se všechna měření opakují pro obě platformy.
- Dohromady bude k dispozici nakonec následující množství měření:
    - 5 tabulek měření pro obě platformy.
    - Každá z tabulek obsahuje 5 sloupců (jeden pro každou metodu)
    - Každá z tabulek obsahuje 20 řádků pro počty HTTP requestů v sériích.
    - Celkově tedy 1000 buňek záznamů ze zhruba 2,7 milionu odeslaných requestů.

3) Ověření existence rozdílu v době vykonávání požadavků mezi oběma platformami

3 Teoretická východiska

3.1 Web API

3.1.1 Obecné vymezení

Pro potřeby této práce se užívá termínu aplikační rozhraní, nebo API pouze ve smyslu webového API (Anglicky: *web application programming interface*), jehož smysl se od obecné definice liší. Například anglická Wikipedie obecný termín API definuje následovně:

Aplikační programové rozhraní je množina definic subrutin, protokolů a nástrojů pro vývoj software a aplikací. Dobrá API zjednodušuje vývoj programu poskytováním stavebních bloků, které programátor skládá dohromady. API může být webový systém, operační systém, databázový systém, počítačový hardware, nebo softwarová knihovna.

> (V originále: A*n application programming interface (API) is a set of subroutine definitions, protocols, and tools for building software and applications. A good API makes it easier to develop a program by providing all the building blocks, which are then put together by the programmer. An API may be for a web-based system, operating system, database system, computer hardware, or software library.*) [16]

Na druhou stranu konkrétní případ webové API popisuje následovně:

Webová API jsou definovaná rozhraní skrze které dochází k interakcím mezi systémem a aplikacemi, které využívají jeho obsah. (...) V kontextu vývoje webu, API je typicky definováno jako set HTTP požadavků spolu s definicí struktur odpovědí, které jsou obvykle vraceny ve formátech XML nebo JSON.

> (V originále: *Web APIs are the defined interfaces through which interactions happen between an enterprise and applications that use its assets. (...) When used in the context of web development, an API is typically defined as a set of Hypertext Transfer Protocol (HTTP) request messages, along with a definition of the structure of response messages, which is usually in an XML or JSON format.*) [16]

Zatímco ve vývoji software API především poskytuje metody, funkce a konstanty, ve smyslu webového API se jedná o výměnu dat prostřednictvím internetu skrze HTTP protokol.[16]

Podle D. Jacobsona, G. Braila a D. Woodse v jejich knize APIs: A Strategy Guide [1] je nejjednodušší definice termínu aplikační rozhraní následující:

API je způsob kterým spolu dvě počítačové aplikace mohou hovořit prostřednictvím sítě (nejčastěji internetu) za použití společného jazyka, kterému obě rozumí.

> (V originále: *An API is a way for two computer applications to talk to each other over a network (predominantly the internet) using common language that they both understand.*) [1]

Podle stejné knihy je esenciální součástí API také její dokumentace, která jasně stanovuje několik bodů, například:

- Jaké funkce poskytuje, jak je lze využít a jaké odpovědi očekávat
- Kdy budou které funkce změněny zpětně nekompatibilním způsobem
- Jaká technická a právní omezení se vztahují na použití

Volitelnými součástmi mohou být dále například zdrojové kódy ukázkových programů konzumujících API, nebo blíže vysvětlující knowhow a tutoriály pomáhající vývojářům pochopit a využít poskytovanou službu.

Kromě technických náležitostí, které jsou citovány dále v rešerši, se citovaná kniha zabývá především obchodním potenciálem a business modely API, které nejsou předmětem této práce. Ve zkratce o možnostech využití API píší autoři takto:

> API může představovat vstupní bod pro kolegy, partnery, nebo vývojáře třetích stran, zkrze který mohou přistupovat k datům a službám, se kterými jednoduše vytváří například mobilní applikace. Existují API otevřené všem vývojářům, API dostupné jenom partnerům a API, které jsou využívány interně pro lepší funkci firmy a usnadnění spolupráce mezi týmy.

(V originále: *An API can provide hook for colleagues, partners or a third-party developers to access data and services to build applications such as iPhone apps quickly. There are APIs that are open to any developer, APIs that are open to only partners, and APIs that are used internally to help run the business better and facilate collaboration between teams.*) [1]

Za důležité považují rozlišování veřejně přístupných a privátních rozhraní, ačkoliv se jejich implementace technicky prakticky neliší. Rozdíly se nacházejí především ve strategii jejich použití a dále například v legálních otázkách, což jsou ovšem rozdíly pro tuto práci opět zanedbatelné.

Pouze pro úplnost: Právě ve veřejných API je větší tlak na zpětnou kompatibilitu a stabilitu, neboť lze předpokládat, že privátní mají menší a přesně definovaný okruh odběratelů a vývoj jejich produktů lze s vývojem API poměrně dobře synchronizovat.

Autoři zde mimojiné označují veřejné API (například ty poskytované službami Twitter a Facebook) za „viditelnou špičku ledovce," zatímco naprostá většina API jsou neviditelné soukromé služby, které mají největší obchodní hodnotu pro jejich vlastníky a jejich partnery.

3.1.2 Požadavky

3.1.2.1 Konzistence a stabilita

V knize APIs: A Strategy Guide je zdůrazněno, že poskytovatel by měl chápat poskytování API jako smlouvu mezi ním a uživatelem, jejíž součástí je především zpětná kompatibilita a určitá konzistence dodávaných služeb.

Totiž: jako každý software a služba musí API procházet vývojem aby udrželo krok s požadavky uživatelů,[2] nebo systému jehož data zpřístupňuje. Změny ale musí probíhat nejčastěji v interním běhu služby a pouze ve vážných případech by se měl poskytovatel

2 Uživateli se v tomto případě už nemyslí uživatelé aplikace, která konzumuje API, nýbrž vývojáři kteří ji vyvíjejí

uchýlit ke změně protokolu (tzn. ke změně rozsahu, nebo struktury vstupních a výstupních dat). V tom případě musí na změny připravit uživatele (například nechat staré metody dosažitelné ještě po nějakou dobu, ale označit je jako zastaralé a určit, kdy budou ze systému kompletně odstraněny).

Alternativou nekompatibilního refactoringu je verzování služby: autoři navrhují jako možnost v momentě, kdy je nutné vytvořit zpětně nekompatibilní změny v protokolu, vytvoření druhé služby (nové verze) běžící po určitý čas paralelně s původní. Noví uživatelé potom mohou začít využívat nové funkce, zatímco stávajícím se jejich služby nezhroutí dříve, než je stihnou aktualizovat pro novou verzi zdroje dat.

Dalším aspektem, který je nutný z hlediska konzistence a spolehlivosti služby zohlednit, je fakt, že API musí obvykle být dostupné nepřetržitě, protože na něm závisí další služby, které by mohly přestat fungovat. Nesmí přestat odpovídat ani kvůli údržbě a aktualizaci, technické chybě, nadměrné zátěži, nebo DoS útoku, což může představovat značné technické překážky. [1]

3.1.2.2 Rychlost adaptace

Opět z knihy APIs: A Strategy Guide pochází vysvětlení tohoto bodu:

> Obecně jsou nejlepší API pečlivě navržené, jednoduché k pochopení, logicky strukturované a vnitřně konzistentní, takže vývojáři mohou úspěšně překonat překážky v pochopení jejich funkcí. API by mělo být trochu jako auto. Každé auto má volant, brzdy a plyn. Možná zjistíte, že výstražná světla, otevírání kufru, nebo rádio jsou trochu jiné, ale je vzácné, aby zkušený řidič neuměl řídit auto z půjčovny.
>
> (V originále: *In general, the top APIs are carefully designed, easy to learn, logically structured and internally consistent so that developers can guess how to bridge gaps in their understanding with relative success. An API should be a bit like a car. Every car has a steering wheel, brake pedals, and an accelerator. You might find that hazard lights, the trunk release, or radio are slightly different, but it's rare that an experienced driver can't figure out how to drive a rental car*) [1]

Maximalizování rychlosti, kterou se nový vývojář přizpůsobí API je důležitý cíl při konstrukci projektu; v případě veřejných služeb je obzvláště důležitý pro přitahování nových odběratelů.

Obecně je tento cíl důvodem pro standardizaci logických návrhů i používaných technologií:

- Na poli architektury existují dva důležité logické návrhy a sice SOAP (XML-RPC) a REST.
- De-facto standardy pro formáty přenášených dat jsou XML a JSON, zřídka také CSV.
- Mezi způsoby autentizace našel velkou popularitu systém OAuth, nicméně podstatně jednodušší možnost nabízí HTTP Basic Authentication zabezpečený TLS protokolem. Další možnosti zahrnují například JSON Web Tokens.

I když v každé oblasti existují různé možnosti volby (a z nich nelze vybrat jedno „*one-size-fits-all*“ řešení), zmíněné technologie jsou značně rozšířené, z čehož plyne při jejich použití výhoda pro odběratele: pokud v minulosti pracoval s API, které používalo tyto standardy, nebude mít problém pracovat s jiným rozhraním využívajícím stejné technologie, a to i pokud se úplně liší business logika a ve všech ohledech i data (například jejich typ, stukura, rozsah apod.). V každém případě je také dostupnější dobrá dokumentace, know-how a rady od komunity.

Použitím standardů - i kdyby proprietární řešení bylo objektivně lepší - se zlepšuje uživatelská přívětivost a noví i stávající odběratelé budou potřebovat stručnější dokumentaci a méně času k prvnímu i každému dalšímu úspěšnému využití rozhraní.[3]

Dokumentace je nicméně stále požadavkem a její kvalita je výrazným způsobem, jak zrychlit adaptaci nových uživatelů. Nejefektivnějším krokem je v tomto ohledu vytváření okomentovaných ukázkových kódu, tutoriálů a podobných detailních know-how.[4]

3 Autor ze své praxe může potvrdit, že první použití nové API, s níž nebyl vůbec seznámen, ale která využívá standardizované postupy, je velmi rychlé a intuitivní i bez čtení vyčerpávající dokumentace

4 To ale přirozeně vyžaduje čas, který nově vznikající projekt obvykle nemá v rozpočtu

Autoři Jacobson, Brail a Woods potvrzují nutnost rychlé adaptace z pohledu marketingu v případě veřejných rozhraní: potenciální odběratel rychle ztrácí zájem o službu a to zvlášť pokud ta má na trhu konkurenci. Je potřeba co nejrychleji ukázat zájemci skutečnou hodnotu poskytovaných dat; pokud API umožňuje jenom placený přístup bez možnosti testovacího provozu zdarma, potenciální klient jeho využití ani nebude zvažovat a podobně rychle ztrácí zájem, pokud se musí registrovat, nebo pokud se mu službu nedaří využít z technických důvodů.[1]

3.1.3 Architektura REST

REST, zkratka z anglického názvu *Representational state transfer*, je architektura či návrhový vzor pro vytváření aplikačních rozhraní; je to set zásad, omezení, doporučení a *best practises*, nikoliv přísně vymezený protokol.

Vznikl v roce 2000 v disertační práci Roye Fieldinga - Architectural Styles and the Design of Network-based Software Architectures [19] – a v současné době je pravděpodobně nejpopulárnějším vzorem API: jedním z faktorů, který dopomohl jeho úspěchu, je široká podpora HTTP protokolu, na nemž staví, ze strany klientů.

3.1.3.1 Filozofie a vývoj

Fielding popisuje filozofický vývoj RESTu jako architektury síťového software v několika vývojových krocích.

V prvním kroku rozhoduje, jestli vývoj architektury začíná na zelené louce a přidává nové komponenty včetně jejich podmínek a omezení postupně, nebo jestli vývojář bere v potaz potřeby zamýšleného celku a z představy celku poté identifikuje a aplikuje podmínky a omezení kladené na jeho součásti.

Fielding zvolil druhý přístup, který nazývá *the null style*:

> Z pohledu architektury, *the null style* popisuje systém ve kterém nejsou rozlišeny žádné hranice mezi komponentami.
>
> (V originále: *From an architectural perspective, the null style describes a system in which there are no distinguished boundaries between components.*) [19]

Na začátku tedy představuje kompletní systém, který je nutné dále dekomponovat. Poté pokračuje stanovením hranic.

V první řadě rozděluje zodpovědnosti mezi klientskou a serverovou část (client-server model).

To prakticky umožňuje (mimo spoustu dalších výhod) vývoj obou součástí systému nezávisle na sobě: je možné paralelně rozšiřovat podporu klientů napříč platformami a zároveň škálovat a aktualizovat serverovou stranu.

Třetím krokem určuje jako podmínku RESTu bezstavovost komunikace: z každého požadavku od klienta musí server získat všechny informace potřebné k jeho vykonání. Udržování stavu aplikace je tak zodpovědnost pouze klientů.

Fielding obhajuje tento přístup, neboť zjednodušuje správu systému (neúspěšné požadavky lze snadno replikovat klientem i správcem monitorujícím běh API) a rychleji uvolňuje zdroje dalším klientům.

Také je jednodušší bezstavové API škálovat horizontálně, tj. spustit více load-balancing serverů, neboť ty nemusí synchronizovat stavy všech připojených uživatelů (jejichž požadavky mohou být rozděleny mezi servery náhodně).

Na druhou stranu ale bezstavová komunikace přináší nevýhodu repetitivního přenášení dat (například přihlašovacích údajů), které nemohou být uloženy na straně serveru.

Ve čtvrtém kroku přibývá podpora cache. Odpověď serveru může obsahovat informace o tom, jak dlouho může klient místo opakování požadavků stejného typu získaná data využívat předtím, než pravděpodobně zastarají a bude potřeba je obnovit.

Cílem zavedení vyrovnávací paměti je snížení počtu požadavků na server a množství dat přenášených po síti. Dále také zrychlení reakce klientské aplikace (která nemusí čekat na odpověď serveru), které se pozitivně promítne do uživatelského zážitku.

Pátý krok vyžaduje sjednocené rozhraní mezi komponentami systému. Tento bod obsahuje čtyři dílčí vlastnosti, které musí komunikace mezi komponentami splňovat:

- Unikátní identifikace zdrojů (všechny zdroje systému jsou jednoznačně identifikovány)

- Manipulace zdrojů zkrze jejich reprezentace (se zdroji lze manipulovat i bez znalosti jejich skutečné implementace; reprezentací dat je myšleno například vyjádření obsahu databáze ve formě JSON, XML, či CSV)
- Samovysvětlující zprávy (každá zpráva obsahuje informace o tom, jak jí lze zpracovat; například hlavičku určující tzv. *media type*)
- Využití hypertextových odkazů na další dostupné zdroje (princip HATEOAS; viz následující kapitola)

Další bod umožňuje vrstvení systému zavedením dalších komponent, které převezmou část jeho zodpovědnosti, jako například historické funkce starších verzí API pro udržení zpětné kompatibility, nebo vyrovnávání zátěže (tzv. *load balancing*). Dalším využitím vrstveného modelu je kontrola bezpečnostní politiky, typicky nasazením firewallů zkrze které prochází ke klíčovým částem jenom validní požadavky.

Posledním volitelným krokem nazvaným *code-on-demand* umožňuje architektura serveru rozšířit funkcionalitu klienta poskytnutím vykonavatelného kódu (např. JavaScript knihovny nebo Java applets) jako zdroje. [19]

Datová orientace

Ideologický rozdíl od konkurenčního návrhu SOAP je představa, že API poskytují především zdroje, nikoliv služby. Tento rozdíl je do určité míry pouze myšlenkový spor: REST k datům přistupuje jako k víceméně samostatně přístupnému zdroji, zatímco SOAP jako ke službě která je poskytuje[5].

V základních příkladech se tento rozpor prakticky neprojeví: v REST architektuře klient požaduje seznam příspěvků, zatímco u SOAP volá funkci „vraťMiSeznamPříspěvků“; v obou případech ale získá stejná data.

Rozdíl může existovat tam, kde je potřeba silně agregovat data na straně serveru: rozsáhlé

5 Česká Wikipedie tento rozdíl popisuje slovy „REST je orientován datově, nikoli procedurálně„

výpočty a specifické účelové kombinace různých dat už spadají spíše do významu služby, než mezi jednoduché manipulace zdroje.

3.1.3.2 Praktická implementace

Fielding sám stál u zrodu protokolu HTTP; REST proto tohoto, dnes masově rozšířeného, protokolu maximálně využívá: počítá například se systémem URI a s HTTP metodami, parametry, hlavičkami, autentizací a stavovými kódy odpovědí.

Mimo orientace client-server, která je pro webové prostřední přirozená, je patrně nejvýraznějším praktickým omezením RESTu jeho bezstavovost: všechny požadavky přicházejí a jsou vykonávány naprosto individuálně a server reaguje pouze na parametry získané z konkrétního požadavku. [19]

Nejdůležitějším parametrem jsou přitom samotná URI na kterou byl požadavek kladen, a zvolená HTTP metoda.

Systém URI

URI definuje cestu ke zdroji od obecných kolekcí (kategorií) po konkrétní uzly; každý bod v cestě URI dodává detaily ke zdroji s nímž se snaží klient manipulovat.

Jednotlivé dostupné zdroje se nazývají *end-points*, čili koncové body. Například pro systém v němž existují autoři příspěvků, kteří vlastní své články k nimž se vztahují komentáře, může vypadat seznam koncových bodů následovně:

- `http://example.com/users/` (vrací seznam uživatelů)
- `http://example.com/users/1` (vrací detail uživatele s ID 1)
- `http://example.com/users/1/articles/` (vrací seznam článků od uživatele 1)
- `http://example.com/users/1/articles/2` (vrací článek s ID 2 od uživatele 1)
- `http://example.com/users/1/articles/2/comments/` (vrací seznam komentářů u článku 2 od uživatele 1)
- `http://example.com/users/1/articles/2/comments/3` (vrací komentář s ID 3 u článku

2 od uživatele 1)[6]

Jak detailně budou všechny end-pointy reagovat záleží na praktickém uvážení:

- Pokud by v tomto příkladu end-point `/users/1/articles/2/comments/` navracel pouze seznam komentářů (místo celých jejich obsahů), znamenalo by to, že na výpis celé diskuze by bylo zapotřebí vyslat na API tolik požadavků, kolik je dostupných komentářů a navíc jeden, který v první řadě získá seznam ID komentářů.
- Na druhou stranu pokud by endpoint `/users/1/articles/` navracel plný obsah všech příspěvků, u plodného autora by se při každém takovém požadakvku přenášely masivní objemy dat. [19,1, 2]

HTTP parametry

Samotné URI zdrojů neříkají nic o způsobu, jakým si přeje uživatel chování API ovlivňovat: k tomu je nutné využít parametry HTTP požadavku. Uvažme tři příklady:

- Pokud by end-point `/users/1/articles/` měl vracet pouze seznam ID článků, bylo by nutné při vypisování všech článků autora (například v archivu, kde je potřeba zobrazit titulek, část textu a thumbnail) posílat – stejně jako v příkladu s komentáři – požadavek pro každý jeden příspěvek.

  Pokud by ale měl požadavek vracet všechna data, opět vystává problém s velkými objemy dat. Pokud autor API neví, kterou z těchto variant bude uživatel spíše potřebovat (a dost možná budou dva různí uživatelé požadovat každý jednu variantu), lze použít HTTP GET parametr pomocí něhož se uživatel sám rozhodne, která varianta bude pro něj praktičtější: `/users/1/articles/?details=true`.
- V minulém příkladu přetrvává problém objemu dat detailů. V tomto případě se nabízí možnost stránkování výsledků: `/users/1/articles/?details=true?count=20&page=1`.
- Samozřejmý příklad akce, kdy je nutné využít parametrů, je vytváření nového, nebo

6 Za poznámku stojí, že URI v tomto tvaru skutečně zůstávají pohodlně čitelné a srozumitelné i bez dokumentace, jak bylo zmíněno v kapitole o rychlosti adaptace.

modifikace stávajícího zdroje. Zřídkakdy nebude v těchto případech API potřebovat od uživatele žádná data. [1,2]

HTTP metody

REST podporuje čtyři HTTP metody, které lze provádět nad danými URI:

- GET
- POST
- PUT
- DELETE

Ty jsou využívány ke čtyřem základním datovým manipulacím:

- Create (vytvoření zdroje)
- Read (získání)
- Update (aktualizace)
- Delete (smazání)

Tyto metody rozlišují různé akce, které se provádějí nad jedním zdrojem označeným URI. Například `GET http://example.com/users/1/articles/` by v předchozím příkladě vrátil seznam článků autora 1, zatímco `POST http://example.com/users/1/articles/` by autorovi vytvořil nový článek.

Zatímco spojení dvojic „GET = Read“ a „DELETE = Delete“ je pochopitelné a na místě, funkce PUT a POST tak přímá není, neboť obě metody lze využít k vytvoření nového zdroje i k jeho aktualizaci.

PUT lze využít vždy na konkrétně zadaný zdroj (např. `/users/1/articles/2`) nezávisle na jeho existenci: pokud existuje, jeho data jsou přepsána, pokud ne, je vytvořen na zadané URI.

Jedná se o tvrdé nastavení zdroje na poslaná data: server se může podle jeho existence rozhodovat jestli provede Create, nebo Update, ale stejný efekt by měla kombinace Delete a Create v každém případě.

POST by měl být využíván pouze na existující zdroje. Vytvářet touto metodou nová data přímo na URI /users/1/articles/123 by nemělo být akceptováno; je ale na druhou stranu být perfektně validní použít POST na /users/1/articles a nechat server, aby adresu nového zdroje vygeneroval.

V posledním příkladě by použití PUT metody mělo[7] místo vytvoření zdroje přepsat celý seznam uživatelových příspěvků. [2]

HATEOAS

Další princip, který Fielding převzal a znovu využil, bylo užívání hypertextů – odkazů na další zdroje. Každá odpověď serveru má, podle původní specifikace, obsahovat seznam dalších dostupných end-pointů aby tak mohl klient API plynule procházet – v podstatě naprosto stejně, jako by návštěvník procházel webovou stránku – bez potřeby dokumentace a především bez nutnosti psát formát URI napevno do kódu klientských aplikací.

Například GET požadavek na kořenový end-point http://example.com/ (ze stále stejného příkladu) by klientovi vrátil uvítací zprávu a zároveň hypertextové odkazy na všechny hlavní kategorie zdrojů, v tomto přípapdě pouze http://example.com/users/. Tento end-point by vracel nejen seznam uživatelů, ale také přímé URI detailů každého z nich, detail každého z nich by obsahoval seznam odkazů na jeho články a tak dále zkrze celou strukturu.

Tento koncept je nazýván *Hypermedia as the Engine of Application State*, zkráceně HATEOAS. [1]

HTTP hlavičky

HTTP hlaviček využívá čistý REST například pro autentizaci požadavku: při striktním dodržení bezstavovosti nelze použít ani mechanismus přihlašování zkrze přihlašovací metodu a následném ukládání tokenů aktivních přihlášení do PHP Sessions na serveru a HTTP Cookies na klientu, jak je běžný postup ve webových aplikacích, ani systémy typu OAuth, které navíc vyžadují komunikaci s třetí stranou.

---

7 Podmiňovací způsob je občas využíván protože REST je stále jenom souhrn doporučení

Místo toho je při použití metody HTTP Basic Authentication kombinace přihlašovacích údajů (jméno a heslo, nebo častěji spíše kombinace dvou náhodných řetězců znaků) obsažena v HTTP hlavičce, zašifrována do base64 řetězce a poté spolu s celou HTTP zprávou zašifrována pomocí validního a platného TLS certifikátu.

Mezi další časté užívání hlaviček patří například kontrola klientské cache (ze strany serveru nastavení expirace poskytovaného zdroje) a stanovení vyžadovaného textového kódování. [1]

Stavové kódy

Stavových kódů HTTP by mělo být využíváno v hlavičce odpovědi k rozlišení výsledků požadovaných akcí. Vrátit HTTP response 200 OK v jehož těle je v JSONu zakódována chybová hláška je pro uživatele matoucí chyba návrhu.

Některé důležité kódy jsou následující:

- 1xx – Informační zprávy
- 2xx – Úspěšně zpracovaný požadavek
  - 200 OK – Úspěšně vrácený zdroj, nebo vykonaná akce
  - 201 Created – Zdroj byl vytvořen
  - 204 No Content – Zdroj nalezen, ale neobsahuje data (prázdný seznam)
- 3xx – Přesměrování
  - 301 Moved Permanently – Zdroj byl trvale přesunut
  - 302 Found – Zdroj byl obecně (dočasně) přesunut
  - 304 Not Modified – Zdroj se od posledního požadavku nezměnil (cache)
- 4xx – Chyba klienta
  - 400 Bad Request – Špatně formulovaný požadavek (špatné parametry)
  - 401 Unauthorized – Neautorizovaný požadavek
  - 404 Not Found – Zdroj nenalezen (chybně zadaná URI)

    - 405 Method Not Allowed – Nepodporovaná HTTP metoda pro daný zdroj
- 5xx – Chyba serveru
    - 500 Internal Server Error – Obecná neočekávaná chyba
    - 503 Service Unavailable – Zdroj dočasně nedostupný (přetížený server) [1]

3.1.3.3 Pragmatický REST

Zatímco většinu doporučení REST architektury je snadné aplikovat, některé teoretické principy naráží při praktické realizaci na překážky.

HATEOAS

Prvním z problémů je princip HATEOAS.

Má sice potenciál lépe chránit aplikace před zpětnou nekompatibilitou (protože i nepatrné změny ve struktuře URI mohou klientské aplikace, v nichž jsou URI vloženy napevno, rozbít), ale iterativní procházení API ke konkrétnímu zdroji představuje zbytečnou režii na straně klienta i serveru (a navíc zatěžování sítě), kterou by šlo odbourat kvalitním návrhem, pečlivě zpracovanou dokumentací a určitou opatrností při aktualizaci API.

Kromě toho je pečlivé konstruování odkazů v odpovědi serveru zbytečná práce, pokud je uživatelé nebudou využívat, což je, vzhledem k tomu, že je procházení API postupně směrem k cílovému zdroji složitější proces a nakonec přinese stejný výsledek, jako definování cílového end-pointu jako konstanty[8], pravděpodobný scénář. [1]

Bezstavová komunikace

Další rozpory může vzbuzovat požadavek na bezstavovost komunikace v otázce bezpečnosti.

Pokud jsou na zabezpečení API kladeny vyšší požadavky, nemusí být HTTP autentizace – ačkoliv vždy šifrovaná pomocí TLS – dostačující řešení. Již v minulosti proběhly různé

8 V takovém případě se programátor klientské aplikace chová stejně jako by si návštěvník webu uložil konkrétní, často navštěvovanou, stránku do záložek

úspěšné útoky na TLS šifrování[9][20] a bez něj je HTTP autentizace bezbranná vůči útoku typu man-in-the-middle.[10]

Bezpečnější alternativy ale často vyžadují určitou spolupráci serveru na udržování stavu aplikace pro daného klienta. Některé konkrétní možnosti autentizace jsou rozebírány v následující kapitole.

Druhým problém v této oblasti je aplikace limitů a restrikcí.

Součástí veřejných API je obvykle vrstva monitorující dění v systému a automaticky ošetřující nebezpečné, podezřelé a jinak nežádoucí chování, případně aplikující omezení vyplývající z obchodního modelu poskytovatele.

Takové omezení typicky představuje limitovaný počet požadavků za časovou jednotku,[11] nebo odepření některých funkcí na základě klientem (ne)zakoupeného tarifu. K realizaci tohoto principu je ale nutné zaznamenávat chování přihlášených uživatelů, (nebo, pokud není registrace vyžadována, alespoň IP adres), respektive kontrolovat jejich status.

9 Útok může směřovat v nejhorším případě přímo na certifikační autoritu s cílem vydávat falešné certifikáty. Druhým běžným cílem jsou vlastníci certifikátu, jehož platnost závisí na udržení privátního klíče v tajnosti. V obou případech může tento útok vést represivní autoritativní vláda, která představuje v tomto ohledu nejhorší představitelné nebezpečí

10 Typ útoku při němž útočník odposlouchává komunikaci na síti, nebo nakaženém počítači a může odposlechnout přihlašovací údaje a využít je k vlastním požadavkům maskovaným za požadavky oběti

11 Taková omezení se navíc mohou měnit v reálném čase, aby reflektovaly vytížení serverové části systému

3.1.4 Další body ke zvážení při realizaci

V této kapitole se nachází několik dodatků, které je vhodné zmínit s ohledem na praktickou realizaci API.

3.1.4.1 Bezpečnost

Ve většině případů je potřeba identifikovat a autentizovat uživatele a autorizovat ho k vykonání žádané metody: v případě privátních API je důvodem ochrana před nežádoucím vnějším přístupem k datům, ve veřejných jde důležitý o prostředek ochrany před DoS útoky a případně pro nastavení obchodních tarifů.

Bylo několikrát zmíněno, že autentizace zkrze HTTP hlavičky je jednoduchá na implementaci a při použití TLS ochrany také relativně bezpečná. Zde zbývá zmínit, že jedna praxe, která zlepšuje tento princip, je registrace unikátního klíče pro každého klienta: při odposlechnutí jednoho páru přístupů potom není kompromitován celý systém.

Při vyšších nárocích na ochranu systému je ovšem na místě zvážit alternativy.

V nejjednodušším případě může API reagovat na přihlašovací požadavek tak, že si po úspěšném ověření údajů uloží do databáze záznam o aktivním přihlášení s náhodně vygenerovaným tokenem, který poté navrátí klientovi. Ten se ve všech dalších požadavcích už neprokazuje kombinací přihlašovacích údajů, ale dočasným, lehce invalidovatelným, náhodným řetezcem znaků.

Tento přístup dává serveru nejlepší možnosti kontroly a operativních zásahů, ale přidává mu náklady na režii. Kromě toho se nejedná o bezstavové řešení.

Substitucí databáze místo PHP Sessions jako úložiště aktivních přihlášení je zajištěna lepší synchronizace ve škálovaném systému, zatímco tím utrpí jeho efektivita (protože čtení databáze je podstatně náročnější, než čtení PHP sessions).

Elegantním řešením problémů nižší efektivity a vyšší režie je udržování všech potřebných informací (zejména id uživatele a expirace tokenu) v tokenu samotném. Pokud je token pečlivě zašifrován dvoucestným algoritmem s tajným klíčem, nemusí jeho kopii server skladovat a není možné jej podvrhnout.

Opět je nutné využít přihlašovací metodu, která token vygeneruje, zašifruje a vrátí uživateli. Každý přijatý token potom server dekóduje svým tajným klíčem a pokud se shoduje s očekávanou strukturou, musel být tento token zašifrován stejným klíčem a tím je jeho autenticita potvrzena.

Protože je možné u dvoucestných algoritmů uhádnout klíč brutální silou[12], dalším bezpečnostním prvkem se nabízí užití jednocestného hashovacího algoritmu s jiným klíčem k vytvoření kontrolního řetězce.

Tímto druhým algoritmem vygeneruje server při vytváření tokenu hash obsahu a připojí ho do řetězce před zašifrováním symetrickou (dvoucestnou) šifrou. Po dešifrování přijatého řetězce navíc vygeneruje nový hash ze zaslaného obsahu a jeho autenticitu potvrdí, pokud se vygenerovaný i klientem zaslaný hash shodují.

Tento způsob řešení rozvíjejí JSON Web Tokens (JWT), které vznikly jako standard RFC 7519.

Problémem tohoto přístupu je fakt, že server neudržuje přehled o vydaných tokenech, což může ztěžovat jejich invalidaci před stanoveným časem expirace a aplikaci limitů a restrikcí. Jinak lze JWT, nebo podobnou technologii, zvážit jako poměrně robustní možnost zabezpečení, která by zároveň dodržovala požadavek bezstavovosti komunikace.

Proti použití tokenů je třeba zvážit následující body:

- Stále je nutné minimálně v jednom případě poslat přihlašovací údaje po síti a proto je stále kritické využívání TLS. Použití tokenů sice redukuje možnost odposlechnutí citlivých údajů, ale neeliminuje jí úplně

12 Případ kdy útočník získá jeden token a zkouší ho za pomoci značné výpočetní síly dešifrovat tolika hesly a algoritmy, dokud nezačne jeho obsah dávat smysl

- Klient musí kontrolovat validitu svého tokenu a včas ho obnovit. Pokud není schopen sám dešifrovat čas jeho expirace, musí při každém požadavku na API kontrolovat jeho odpověď a v případě expirace tokenu zažádat o nový a předchozí požadavek zopakovat
- Generování a validace tokenů představuje další náklady na vývoj a později určitou režii na straně serveru
- Je nutné vystavit přihlašovací metodu, která není zdrojem (drobný konflikt s filozofií datové orientace REST architektury)

Ještě náročnější bezpečnostní požadavky a případy kdy je nutné komunikovat se třetí stranou uspokojuje OAuth.

OAuth je rodina komplexních zabezpečovacích protokolů založená na komunikaci s autorizační autoritou, která službám třetích stran dokazuje autentičnost klientova požadavku a to aniž by jim (třetí straně) klient musel poskytovat příhlašovací údaje pro autoritu.

Při komunikaci s jedinou službou může být toto řešení, které vyžaduje kromě služby samotné také autorizační server, zbytečně složité na implementaci, pro rozsáhlé služby se ovšem může jednat o smysluplnou variantu. Google například využívá vlastní autorizační server, který vydává tokeny s nimiž je možné provádět autentizované požadavky na jeho značný počet API. [1, 17, 24]

3.1.4.2 Rozsah služby

V závislosti na přepokládané budoucí a skutečné aktuální popularitě služby a rozsahu dat musí její poskytovatel zvažovat možnosti jejího škálování a optimalizace výkonu.

Na tomto bodu závisí v první řadě volba infrastruktury. Možnosti zahrnují především hostování služby (typické webhostingové služby), pronájem virtuálního serveru, správu vlastního fyzického serveru, pronájem cloudového řešení a v krajních případech i stavbu vlastního datacentra.

Vybraná infrastruktura může vyžadovat specifické přizpůsobení aplikace po technologické stránce (problémy může činit zejména horizontální škálování a synchronizace dat mezi více fyzickými stroji).

Výpočetní nároky na rozsah architektury (a současně čas odezvy) API je možné snížit dvěma principy, které Roy Fielding navrhoval: vyrovnávací pamětí a vrstvením systému.

Vhodné rozložení duplikovaných nebo specializovaných prvků v architektuře systému odlehčuje práci vytíženým kritickým součástem. Každý z těchto prvků navíc může dočasně obsahovat lokální kopie dat – svojí cache – díky čemuž může určitou část provozu odbavit bez interakce s důležitějšími prvky hlouběji v systému.

Geografické rozložení prvků vyvažujících zátěž dodává možnost dále snížit odezvu systému vytvořením efektivní distribuční sítě - *content delivery network,* CDN – která přepojuje klientské požadavky na nejbližší dostupný uzel, který si opět udržuje cache díky níž může určitý nápor obsloužit bleskurychle.

Ke tvorbě cache je ale důležité přistupovat opatrně: při příliš volně nastavených pravidlech mohou být klientům bleskurychle předávána zastaralá data, zatímco při příliš rychlé expiraci nebo časté invalidaci cache přináší její zavedení jenom režii potřebnou k její správě, což může paradoxně celou situaci zhoršit. [1]

3.1.4.3 Limity a restrikce

Otázka limitování počtu požadavků je záležitost na pomezí zajištění kvality služby a jejího obchodního modelu. Kvóty v první řadě ochraňují API před vyčerpáním jejích zdrojů, což by zapříčinilo její nedostupnost (ať už se jedná o přirozenou zátěž, nebo DoS útoky).

Z obchodního hlediska umožňuje jejich diferenciace například značně omezený testovací běh pro neregistrované potenciální nové uživatele, průměrné kvóty pro běžné, registrované uživatele a ještě menší omezení pro náročné prémiové klienty.

Nastavení výše kvót záleží na dostupné infrastruktuře a obchodním rozhodnutí, ale samotný princip je ve veřejných API téměř vždy potřebný. [1]

3.1.4.4 Verzování služby

Pokud striktně nesynchronizujeme vývoj klientů a serveru, je vhodný přístup předpokládat možnost, že během vývoje systému bude potřeba udělat nekompatibilní změny v protokolu.

Pro tento případ je vhodné mít API připravenou pro spuštění nové verze za zachování předchozí plně funkční: například mít číslo verze přítomné v základní URI (ať už jako root kolekci, nebo jako subdoménu) a v kódu připravený mechanismus routování požadavků. [1]

3.1.4.5 Reprezentace dat

Jedno z menších rozhodnutí při realizaci API musí padnout v otázce formátu reprezentace dat.

XML představuje robustní formát, který nachází své uplatnění v popisu skutečně komplexních datasetů. Již stabilní nástroje (DFD, XSLT) je umožňují efektivně, automatizovaně validovat, kombinovat a transformovat.

JSON je oproti tomu minimalistický a extrémně úsporný. Jeho parsování je rychlé a parsery se staly běžnou součástí frameworků i samotných programovacích jazyků.

Pro většinu implementací je JSON pro svojí úspornost vhodnější, nicméně mohou existovat výjimky, kde je XML volbou ke zvážení. [1]

3.2 PHP

PHP (rekurzivní zkratka *PHP: Hypertext Preprocessor*, dříve *Personal Home Page*) je programovací jazyk poprvé zveřejněný v červnu 1995 a určený primárně pro server-side tvorbu webových stránek (v podstatě pro procedurální dynamické generování HTML souborů), který nicméně díky svému vývoji (například s výrazným zlepšením podpory objektově orientovaného programování ve verzi 5) lze využít k realizaci jiných webových, command-line i desktopových grafických aplikací.

PHP parser může být buďto modulární součástí web server aplikace, nebo spustitelný soubor, který je web server aplikací volaný zkrze Common Gateway Interface – CGI – protokol. PHP parser je tedy jenom součástí běhového prostředí a závisí na implementaci web serveru – známá je jeho kombinace s Apache ve stacku LAMP a z něj odvozených.

Standardní, oficiální vývojářskou skupinou vyvíjený parser je Zend Engine vyvinutý pro verzi PHP 4 (jeho druhá a třetí verze vyšly pro verze PHP 5 a 7). Protože do roku 2014 neexistovala formální specifikace[13], je Zend Engine považován za referenční implementaci jazyka PHP.
Specifikaci začal spravovat Facebook při vývoji vlastního parseru, HipHop Virtual Machine.

PHP je interpretovaný jazyk, takže s každým spuštěním jeho skriptu je tento pokaždé parsován a kompilován na strojové instrukce, což je – vzhledem k tomu, že se jeho kód málokdy mění – neefektivní. Tento problém řeší tzv *„opcode cache“* - dočasná paměť ukládající výsledky kompilace, Její implementace je pod názvem Zend OpCache výchozím rozšířením Zend Engine od verze PHP 5.5.

13 Tvůrce PHP, Ramus Lerdorf, ostatně s prvními verzemi PHP ani neměl ambice vyvíjet nový programovací jazyk a když se do jeho rozvoje vložila komunita, zapříčinila absence dokumentace například dodnes patrnou inkonzistenci pojmenování nativních funkcí.

Ještě dále v této optimalizaci šel právě Facebook s HHVM. Jeho první inkarnace řešila potřebu většího výkonu překladem PHP skriptů do kompilovaného jazyka C++. Novější verze místo toho překládají skripty do bytecode, který je poté parsován a vykonáván just-in-time kompilátorem. Tento přístup způsobil masivní nárust výkonu oproti tehdejší verzi PHP 5.6.

Poslední major release, PHP 7, přinesla kromě obvyklého vývoje sémantiky především nový, sílně refaktorovaný a podstatně výkonější Zend Engine. Testy výkonu na instalaci CMS WordPress ukázaly až stoprocentní kladnou změnu výkonu. [26]

3.3 Server-side JavaScript

JavaScript byl vyvinut společností Netscape Communications, poprvé představen v květnu 1995 a na trh uveden v březnu 1996, kdy byl jeho interpreter implementován v Netscape Navigator 2.0; Jeho prvotním cílem bylo umožňovat dynamické chování dokumentu (stránky) na straně klienta.

Vzápětí vypuknuvší válka o implementaci client-side skriptování mezi Netscape Navigator a Microsoft Internet Explorer, v níž jediní poražení byli web developeři, kteří museli své weby opravovat pro oba prohlížeče, vyuústila ve vytvoření formální specifikace ECMAScript, jehož je JavaScript implementací.[14]

Jeho popularitu později zvyšovala dostupnost frameworků (např. uvedení dodnes populárního jQuery v roce 2005), a také specifikace Ajax.

Tato technika využívá objektu XMLHttpRequest (implementován poprvé v Internet Explorer v roce 1999) pro asynchronní volání HTTP serveru, což v praxi umožňuje dynamické načítání obsahu a modifikaci DOM bez nutnosti znovu načíst celou stránku.

Termín Ajax (asynchronous JavaScript and XML) vznikl v článku Ajax: A New Approach to Web Applications od Jesse James Garretta z roku 2005 [21] a technologie samotná se stala důležitým krokem ve vývoji webu.[15]

Interpreterů JavaScriptu (lépe nazývaných *JavaScript engine*) je z důvodu paralelního vývoje (umožněného existencí formální specifikace) pro více prohlížečů, samozřejmě více než PHP parserů. Mezi nejrozšířenější patří například SpiderMonkey (Netscape, Firefox), Chakra (Internet Explorer & Edge), JavaScriptCore (Safari) a především V8 (Chrome & Opera).

14 JavaScript soupeřil se svým neperfektním klonem Jscriptem.

15 Vzpomeňme na úvod a vývoj webu od statických stránek přes dynamicky generované po interaktivní webové aplikace. Právě druhá transformace by bez JavaScriptu a Ajaxu nebyla možná.

Přirozeným a proto nejrozšířenějším prostředím byl od počátku webový prohlížeč, nicméně už krátce po jeho uvedení pro prohlížeče, v prosinci 1995, vydala firma Netscape Communications první implementaci server-side JavaScriptu v Netscape Enterprise Server. Větší popularity se na tomto poli ale JavaScript dočkal s uvedením runtime prostředí Node.js v roce 2009.

Node.js využívá engine V8 pro interpretování skriptu a vlastního web server modulu (s prostým názvem http), který obsluhuje komunikaci s klientem. Node aplikace je tedy v podstatě svůj vlastní web server.

S vytvořením stabilního prostředí pro vykonávání na straně serveru lze tedy JavaScript využít i pro stejné účely jako ostatní server-side jazyky. [25,5,6]

4 Vlastní práce

Pro potřeby srovnání efektivity programovacích jazyků (resp. jejich parserů a prostředí v nichž jejich vykonávání probíhá) je potřeba minimalizovat vliv okolních faktorů. Z toho důvodu mají obě aplikace společnou většinu teoretického návrhu, shodný protokol a společnou databázovou vrstvu.

4.1 Společný protokol a datová struktura

Obě API mají minimalistické zadání navržené tak, aby testovalo především schopnost aplikace odbavovat velké množství paralelních požadavků. Protože jsou datové manipulace záležitost především databázového serveru, který je pro obě implementace společný, může být datová struktura jednoduchá.

4.1.1 Protokol

Pro čtyři základní datové manipulace byly navrženy čtyři koncové body na něž budou aplikace reagovat zdokumentovaným způsobem:

End-point	CRUD	Akce
POST /articles/	Create	Vytvoří nový zdroj a vrátí ID
GET /articles/<id_article>	Read	Získá obsah daného zdroje
PUT /articles/<id_article>	Update	Nastaví nový obsah zdroje
DELETE /articles/<id_article>	Delete	Smaže záznam zdroje

- Tabulka 1:Seznam koncových bodů dle společného protokolu

POST /articles/

Ekvivalent CRUD: Create

Vytvoří nový zdroj a vrátí vytvořené ID.

Povinné parametry:

- (string) content

Volitelné paramety: (žádné)

Vrací:

{

"response": 1,

"message": "success",

"id_article": <id_article>

}

GET /articles/<id_article>

Ekvivalent CRUD: Read

Získá obsah daného zdroje.

Povinné parametry: (žádné)

Volitelné parametry: (žádné)

Vrací:

{

"response": 1,

"message": "success",

"content": <content>

}

PUT /articles/<id_article>

Ekvivalent CRUD: Update

Nastaví nový obsah zdroje.

Povinné parametry:

- (string) content

Volitelné paramety: (žádné)

Vrací:

{

"response": 1,

"message": "success",

"id_article": <id_article>

}

DELETE /articles/<id_article>

Ekvivalent CRUD: Delete

Smaže záznam zdroje.

Povinné parametry: (žádné)

Volitelné parametry: (žádné)

Vrací:

{

"response": 1,

"message": "success",

}

4.1.2 Databáze

Obě aplikace sdílejí stejný databázový server se shodnou datovou strukturou na nějž pokládají shodné dotazy. Databázovým serverem je relační databázový engine MariaDB.

Celou funkčnost datové manipulace obstarává jedna triviální tabulka:

articles	
id_article *	INT (255)
content	TEXT

- Tabulka 2: Atributy relační tabulky articles

Na tuto tabulku jsou pokládány následující dotazy:

CRUD	**SQL dotaz**
Create	INSERT INTO articles (content) VALUES (?)
Read	SELECT content FROM articles WHERE id_article = ?
Update	UPDATE articles SET content = ? WHERE id_article = ?
Delete	DELETE FROM articles WHERE id_article = ?

- Tabulka 3: SQL dotazy pro CRUD operace

Pro kontrolu uživatele (všechny CRUD operace vyžadují autorizaci) je potřeba tabulky uživatelů:

users	
id_user *	INT (255)
username	VARCHAR (128)
password	VARCHAR (128)
salt	VARCHAR (64)
active	INT (1)

- Tabulka 4: Atributy relační tabulky users

Na tuto tabulku bude vždy pokládán jediný dotaz:

CRUD	SQL dotaz
Read	SELECT * FROM users WHERE id_user = ?

- Tabulka 5: SQL dotazy pro operace s tabulkou users

4.2 Společná technická charakteristika

Ačkoliv mají oba programovací jazyky svá specifika implementace, obě aplikace využívají stejný návrhový vzor. Odchylky od něj jsou popsány v samostatných kapitolách.

Tento obecný vzor vznikl s ohledem na tři konkrétní hlediska: efektivitu obsluhy, bezpečnost a možnosti jeho rozšíření. Má poskytovat určitý logický rámec, který slouží jako společné zadání obou implementací a současně jako souhrn doporučení pro jejich vývoj.

4.2.1 Efektivita obsluhy

Návrhový vzor je koncipován podle způsobu líného vykonávání[16] (tzv. *lazy evaluation*).

Jeho proces vyhodnocování požadavku je řazen tak, že všechny úkony náročné na paměť, nebo výpočetní výkon jsou odloženy na poslední možný termín. Program nejprve kontroluje všechny vstupy aby odhalil případy, kdy nemůže požadavek vykonat, dříve než bude pracovat s databází a soubory.

Priorita dílčích funkcí je proto následující:

1. Výběr výstupního formátu
    - Návrh počítá s abstrakcí vrstvy formující výstup, což by v důsledku mělo uživatelům umožňovat možnost volby formátu pro návratové hodnoty, mezi XML, JSON, nebo dalšími.
    - Takto brzo je tento bod zařazen proto, aby i všechna chybová hlášení program zobrazoval v preferovaném formátu.
    - V konečné realizaci je z důvodu zjednodušení tento krok vynechán a jako výstupní formát je pevně vybrán JSON.
2. Výběr vnitřní volané funkce podle metody HTTP metody a URI zdroje
    - Pokud je špatně zvolený end-point, nebo metoda, nemá důvod se požadavkem dále zabývat.

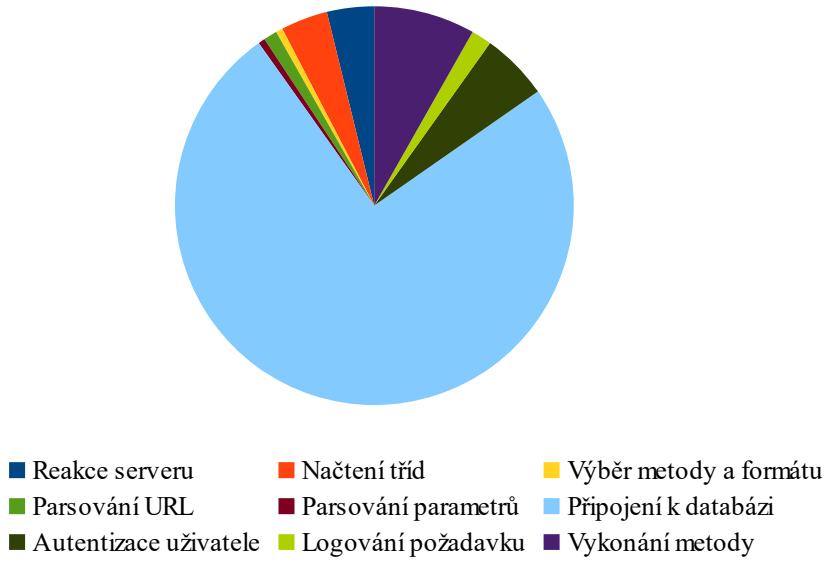
16 Autorův oblíbený způsob vykonávání čehokoliv

3. Kontrola povinných parametrů dané funkce
    - Pokud nejsou s požadavkem zaslány povinné parametry, nebo jsou vyplněny špatně, program končí.
4. Ošetření nepovinných parametrů
5. (volitelně) Autentizace uživatele
    - Pokud má být funkce zabezpečena, v tomto momentě se poprvé inicializuje spojení s databází.
    - I to ovšem teprve tehdy, pokud jsou přihlašovací údaje přítomny v požadavku.
6. (volitelně) Logování požadavku pro potřeby monitorování
7. Vykonávání vlastní funkce

Pozitiva lazy evaluation nastiňují následující data: jedná se o orientační měřené časy potřebné k dílčím úkolům v prostředí PHP a lokálního Apache serveru:

Reakce serveru	7ms
Načtení a inicializace potřebných tříd	7ms
Výběr HTTP metody a výstupního formátu	1ms
Parsování URL požadavku	2ms
Parsování HTTP parametrů	1ms
Připojení k databázi	137ms
Autentizace uživatele	10ms
Logování požadavku do textového souboru	3ms
Vykonání žádané metody a parsování JSON odpovědi	15ms

- Tabulka 6: Orientační časy zpracování dílčích úkolů obsloužení požadavku



- Graf 1: Vizualizace časů zpracování dílčích úkolů obsloužení API požadavku

Je evidentní, že kámen úrazu výkonu programu je inicializace připojení k databázi - samotné požadavky na ní (autentizace uživatele a vykonání žádané metody) už probíhají plynuleji (ačkoliv v tomto konkrétním případě byly dotazy na databázi velmi jednoduché). Navzdory očekávání trval zápis do textového souboru jen velmi krátkou dobu.

Přesto bylo ilustrováno, že včasné ošetření (chybných) vstupů může ušetřit podstatně delší aplikační čas strávený dlouhými vstupně výstupními operacemi.

Aby bylo možné realizovat tuto filozofii, je nutné mít deklarovaný seznam služeb ještě před vykonáváním vlastní funkce. Toto blíže řeší podkapitola „možnosti rozšíření."

4.2.2 Bezpečnost

Ať už má být API veřejná či privátní, je třeba podniknout několik bezpečnostních opatření.

První z nich je využití podpory TLS šifrování na produkčním hostingu služby. To se sice nijak netýká návrhu API, nicméně je to na tomto místě zdůrazněno, protože bez tohoto opatření zůstává návrh zranitelný vůči útoku typu man-in-the-middle.

Druhou ochranou je generování páru veřejného a tajného klíče, které nahrazují kombinaci přihlašovacího jména a hesla. Důvodem k tomuto opatření je ochrana před brute-force hádáním přihlašovacích údajů: pokud jsou oba klíče náhodné a dostatečně dlouhé, riziko jejich uhádnutí se limitně blíží nule.

Všechny požadavky na zabezpečené funkce API musí obsahovat přihlašovací údaje, vydané serverem, přiložené jako metadata požadavku. K tomu je využíván prostý mechanismus HTTP Basic Authentication a to z důvodů, které byly nastíněny v teoretické části a které také například popisuje Randal Deggess v článku Why I Love Basic Auth. [17][17]

Ochranou před dolováním přihlašovacích údajů z databáze je jejich šifrování: v databázi se nikdy nesmí nacházet heslo (v tomto případě tajný klíč) nezašifrované. Systém nepotřebuje heslo znát – při registraci a znovu při pokusu o přihlášení zadané heslo zašifruje stejným algoritmem a porovnává jeho výsledky. Pokud se shodují výsledky, shodují se i hesla.

Rozšířením ochrany hesel je používání tzv. kryptografické soli při generování hashe. Tato sůl je jednak z větší části generována náhodně pro každou registraci a jednak je její součástí identifikátor konkrétní instalace aplikace. Tento identifikátor je přítomný v konfiguraci a je přítoment proto, aby nebylo možné vytvořit novou registraci na lokálním prostředí a podstrčit jí do produkční databáze.

API klíče i náhodné soli jsou při registrování generovány pomocí kryptograficky bezpečného generátoru pseudonáhodných čísel (openssl_random_pseudo_bytes u PHP, nebo csprng u Node.js) a tajné klíče (hesla) jsou šifrovány bezpečným algoritmem bcrypt.

4.2.3 Možnosti rozšíření

Je rozumné předpokládat, že se bude API dále vyvíjet a vývojáři budou potřebovat lehce přidávat, upravovat a mazat funkce a také bude vhodné držet v aktuální verzi alespoň minimální dokumentaci.

Dokumentaci – strojově čitelnou – potřebujeme i pro princip líného vykonávání, jak bylo

17 Pro takto malý, navíc ukázkový, projekt je toto řešení dostačující. Pro další možnosti viz kapitolu Další body ke zvážení při realizaci

nastíněno v kapitole Efektivita obsluhy. Logickým řešením se jeví přímo v kódu udržovat datovou strukturu, která bude obsahovat všechna data o poskytovaných funkcích. Ve formátu JSON může vypadat následovně:

```
{
"PUT": {},
"GET": {
    "hello": {
        "secured": false,
        "mandatory_parameters": {},
        "optional_parameters": {
            "name": "string"
        }
        "docs": {
            "url": "<API_URL>/hello",
            "status": "active",
            "description": "Returns positive response if
           connection was successful.",
            "return": [
                "(int) response: response code",
                "(string) message: human readable response"
            ]
        }
    }
},
"POST": {},
"DELETE": {}
}
```

- Ukázka 1: Strojová dokumentace API ve formátu JSON

Tato struktura se stává jádrem operační logiky API.

V první řadě je jednoduché přidávat a odebírat nové vnitřní metody API a měnit end-points na které reagují: stačí do kódu přidat mechanismus, který zavolá funkci s variabilním názvem v závislosti na metodě a požadavku.

V PHP to lze například udělat těmito řádky:

```
$action = $method."_".$request);
if(method_exists($this,$action)){
   $this->$action();
}
```

- Ukázka 2: Volání variabilní metody v PHP

Tento krátký kód hledá v dané třídě metodu nazvanou například get_hello a pokud existuje, volá ji. JavaScript může v tomto místě využít své schopnosti pracovat s funkcemi jako s proměnnou a proto může být funkce samotná (resp. přímá reference, nikoliv jenom její název) součástí konfigurace.

Pokud jsou všechny ostatní mechanismy na místě (přihlašování, zpracování vstupních údajů apod), přidání nové funkce API bude spočívat pouze ve vytvoření nové metody třídy API (nebo dílčího Controlleru, pokud se skládá z více modulů) a vložení jejích metadat do strojové dokumentace.

Také se tímto způsobem automatizuje zmiňovaná kontrola vstupních parametrů (za předpokladu, že se návrh aplikace stále drží vzoru).

Proměnná secured potom programu říká, jestli má před voláním vnitřní funkce kontrolovat přihlášení uživatele.

Objekt docs nám udržuje aktuální minimální dokumentaci a je tak velmi jednoduché ji zpřístupnit uživatelům[18]. Jeho atribut status například má stanovovat, jestli mají uživatelé tuto funkci využívat, nebo jestli má v dohledné době zastarávat.

Následující metoda se nachází ve třídě/modulu API (kód pochází z PHP implementace) a zpřístupňuje standardním výstupem seznam a náležitosti dostupných požadavků tak, jak jsou definovány přímo ve strojové dokumentaci:

```
protected function get_help() {
   return new Message(1, "Available requests", array(
      "available_requests" => $this->available_requests
   ));
}
```

- Ukázka 3: Zpřístupnění strojové dokumentace uživateli

Tento přístup nemusí být pro rozměrné veřejné API vhodný, nicméně menším interním týmům umožňuje podstatně lépe pochopit dostupné požadavky a to s menší šancí na chybu v dokumentaci, která se během vývoje mění. Kupříkladu parametry požadavku, které tato nápověda uvádí, jsou naprosto ty samé, které samotné API parsuje a vyžaduje.

18 Přesněji řečeno to je primární účel přítomnosti tohoto pole přímo v datové struktuře

4.3 Specifika PHP API

PHP API je bližší návrhovému vzoru pro jeho tradičnější procedurální pojetí.

Jádrem PHP implementace je funkce `init` třídy Controller, která postupně volá metody zodpovídající za jednotlivé kroky nastíněné návrhovým vzorem a kontroluje jejich výstupy. Všechny vnitřní výsledky se ukládají jako třídní proměnné a tak k nim dílčí metody také přistupují: třída jim poskytuje rámec v němž mohou data sdílet.

Tento v podstatě procedurální postup je jednoduché sledovat, protože funkce `init` slouží jako rozcestník a po každém kroku lze sledovat změnu třídních proměnných.

PHP je objektové a třídy umí dědit a proto je snadné oddělit operativní a „obchodní“ logiku aplikace. Třída Controller (který je zároveň routerem) dědí ze základního BaseControlleru, který se stará o všechny operace na pozadí. Třída Controller tedy obsahuje jenom cesty a jejich dokumentace a metody specifické pro vlastní logiku aplikace. Stejně tak BaseModel obsahuje metody na pozadí (kontrolu přihlášení, kryptografické funkce), které má dostupné i vlastní aplikační Model.

Vzhledem k autorovým předchozím zkušenostem s PHP nebyly s implementací této aplikace výraznější problémy.

4.4 Specifika JS API

JavaScript API využívá funkcionální event-driven povahy jazyka: jeho funkce na sebe navzájem reagují bez účasti centrální metody. Objekt požadavku se neukládá ve společné paměti, ale předává se každé další metodě, přičemž se v každém kroku modifikuje a rozšiřuje.

Implementace http modulu přečte parametry požadavku a postoupí vlastní zjištěná data směřovací funkci. Ta se pokusí najít funkci odpovídající požadavku a dokumentaci této funkce spolu s daty od serveru posílá funkci, která má ošetřit vstupní parametry. Ta ošetří vstupy a pokud je nutná kontrola zabezpečení, volá funkci která má tuto zodpovědnost.

Zabezpečovací funkce navíc získá jako parametr celý požadavek z jehož dokumentace dokáže přečíst referenci na koncovou funkci: nemusí proto výsledek svého ověřování vracet předchozí funkci ale rovnou volá další krok.

Všechny dosud zmíněné funkce si také kromě požadavku předávají i funkci serveru, kterou mohou kdykoliv – zpravidla při chybě – ukončit zpracování[19].Výsledek proto nemusí cestovat celou dosud popisovanou cestou zpět (v procedurálním programování by to byla série příkazů return), protože i koncová funkce má k dispozici přímo výstupní funkci serverového modulu.

Všechny interakce s databází navíc využívají jako parametr anonymní funkce (tzv. closures), které se volají po úspěšném čtení z DB. V následujícím kódu je patrná deklarace funkce na místě parametrů jiné funkce (od řádku `function (message){`). Na stejném kódu je zároveň vidět jak funkce `model.executeQuery` tuto funkci spustí: stejně jako `model.getArticle` dostává parametr `callback`, který na konci kódu spouští voláním `callback(message)`:

19 Užitečná funkce JS je možnost předávat reference na funkce jako kterékoliv jiné proměnné – tomuto principu se říká callback.

```
model.getArticle = function (id_article, callback) {
    var sql = "SELECT content FROM articles" +
            "WHERE id_article = ?";
    model.executeQuery(
            sql,
            [id_article],
            "Article retrieved",
            "Failed to retrieve article",
            function (message) {
                if (message.rowCount() == 0) {
                    message = createMessage(
                            0, "Article not found", [], 404
                            )
                } else {
                    message.parameters = message.fetch();
                }
                callback(message);
            }
    );
};
```

- Ukázka 4: Callbacky a anonymní funkce předávané jako parametr funkce

Sledovat dataflow v tomto systému může být pro nezasvěceného člověka přinejmenším nestandardní zážitek.

JavaScript sice umí objektové programování na principu prototypů, nicméně toho v praktické implementaci nebylo využito. Separování zodpovědností bylo dosaženo využitím tzv. modulů dostupných v ECMAScriptu 6. Využitím modulů, které samy implementovaly obecnější rodičovské moduly, bylo možné replikovat funkci dědění.

Při implementaci se objevila řada menších překážek, z nichž stojí za pozornost tyto:

Node.js funguje odlišně než PHP parser: Node spustí jednu instanci aplikace, která naslouchá na zvoleném portu a zpracovává příchozí požadavky. To je zajímavé především protože tato jedna aplikace si udržuje otevřené spojení s databází a to není potřeba inicializovat při každém požadavku. Proto by bylo v pořádku navzdory návrhu inicializovat spojení s databází už při startu aplikace.

Autor ponechal inicializaci na svém místě ve vzoru, ale musel mezi startem JavaScript API a měřením manuálně odeslat jeden požadavek, aby se inicializovalo spojení s DB, které jinak trvalo pár stovek milisekund a bezpečně rozhodilo první bloky měření.

Dále bylo nutné nahradit balíček `nodejs-bcrypt` za `bcrypt`, protože první z nich byl zhruba pětkrát pomalejší, což také neúměrně zkreslovalo měření.

4.5 Testovací kód

Testovací kód byl napsán v Node.js za pomoci knihovny async. Jeho princip je následující:

V prvním kroku vytvoří seznam jednotlivých měření: každé měření obsahuje číslo opakování, detaily platformy na které má pokládat požadavky a seznam samotných požadavků (CRU(R)D operace opakované podle požadovaného počtu paralelních požadavků).

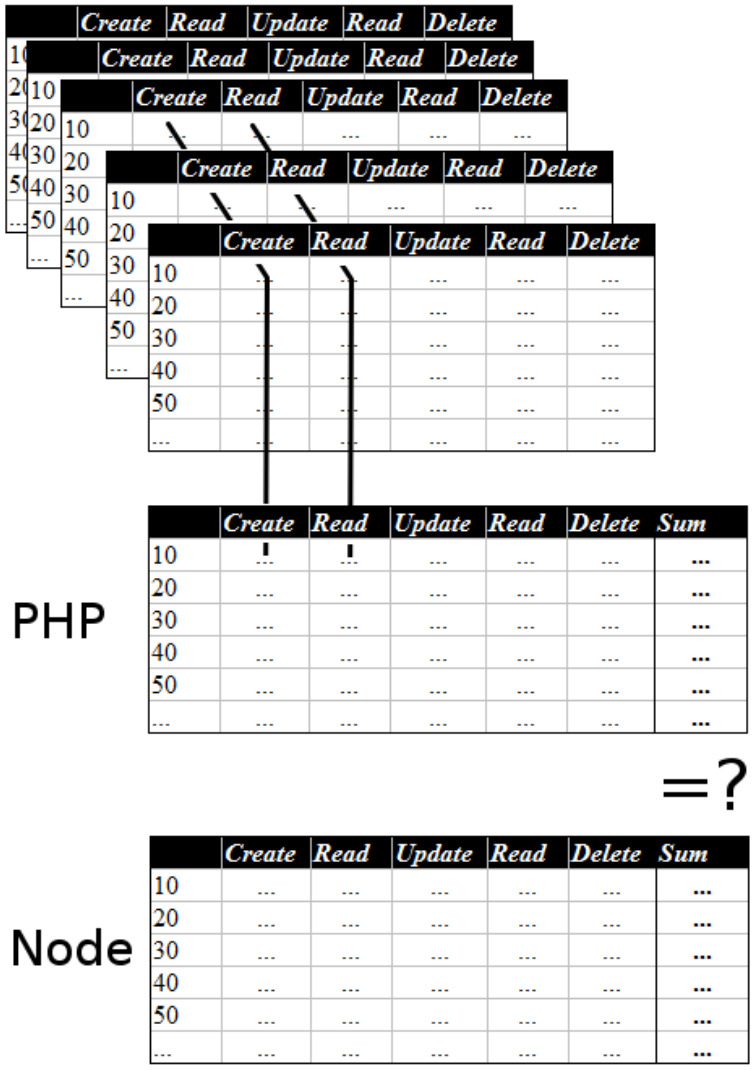
Ve druhém kroku jsou postupně tato měření sériově vykonávána: Nejprve iteruje podle počtu vyžadovaného počtu požadavků, poté podle platformy a nakonec podle počtu opakování měření:

- 1. opakování, PHP, 10 požadavků
- 1. opakování, PHP, 20 požadavků
- (…)
- 1. opakování, Node, 10 požadavků
- 1. opakování, Node, 20 požadavků
- (…)
- 2. opakování, PHP, 10 požadavků
- (…)

Každá dávka požadavků obsahuje rovnoměrně rozložené CRU(R)D operace; nejprve jsou vykonávány všechny Create akce, poté co získá odpověď poslední požadavek, pokračuje blok Read akcí a tak dále.

Každý čas je měřen od spuštění série požadavků pro danou operaci po obdržení výsledku všech požadavků (přirozeně včetně jejich opakování v případě chyby).

V momentě, kdy jsou všechna měření u konce, jsou data uložena, agregována a pročištěna. Jak je znázorněno na následující ilustraci, odpovídající buňky všech opakování jsou zprůměrovány a z těchto průměrů se poté konstruují sumy, které jsou předmětem statistické analýzy.



- Ilustrace 1: Agregování vícenásobných měření

Pokud se kterákoliv hodnota odlišuje od průměru o více než 20% (kterýmkoliv směrem), je považována za extrémní a je z průměru vyřazena. Počet vyřazených hodnot se zaznamenává a v případě, že by bylo vyřazeno příliš hodnot, lze toleranci zvětšit.

Už při prvních testech se ale hodnota 20% ukázala jako poměrně stabilní kompromis, do něhož se nevejde pouze podmnožina o velikosti v jednotkách procent celku, jejíž prvky toleranci navíc přesahují pouze nepatrně.

4.6 Předběžné měření a úprava zadání

4.6.1 Setup

Následující měření probíhalo na lokálním prostředí: testovací kód i obě API se nacházely na stejném zařízení: laptopu Acer Aspire E5-553G (procesor AMD A12-9700P 2.5GHz 4j. / 4 GB RAM).

PHP vykonával stack XAMPP 7.1.1, JavaScript prostředí Node.js 6.10.0.

4.6.2 Průběh

Již první testovací měření, které měly vytvořit bližší představu, jak bude skutečné měření probíhat, přinesly první předběžné, nicméně zajímavé výsledky.

Při prvních větších testech na XAMPP 5.6.21 s verzí PHP 5.6 stihlo PHP API odbavit pouze 3750 požadavků předtím, než Apache začal odmítat nové připojení, zatímco Node pohodlně pokračoval dál.

Druhý test s nejnovější instalací XAMPP stacku (7.1.1) obsahujícím PHP 7.1 prokázal stejné[20] výsledky, což poukazovalo na problém na straně web serveru.

Apache byl tedy nahrazen instalací web serveru Nginx ve stacku WTServer, nicméně i ten při určitém, podobném počtu požadavků, začal vracet chybové stavové kódy 504 a 502.

Podle všeho tedy parser PHP toto množství paralelních požadavků nezvládl zpracovat a přestal včas odpovídat, nebo narazil na interní bezpečnostní limity a začal požadavky odmítat úmyslně. Oba web servery proto nemohly ve stanovené době obsloužit další požadavky a testovacímu kódu vrátily error, který test přerušil. Stejný problém se několikrát objevil i na straně MariaDB, ačkoliv to byl vzácnější případ: obvykle dříve zkolabovalo PHP.

Bylo proto nutné manuálně regulovat počet paralelních požadavků. Toho bylo dosáhnuto tak, že byla experimentálně určena nejvyšší hladina při které bylo PHP API schopné reagovat: 300 paralelních požadavků na lokálním prostředí, 500 na ostrém setupu.

20 Dokonce horší, protože v době druhého testu byl výpočetní výkon stroje využíván i jinými aplikacemi

Když bylo této hodnoty při vykonávání dosáhnuto, funkce odesílající požadavek se na náhodný počet milisekund (0-200) odmlčela a poté se zkusila sama zavolat znovu – pokud i tehdy nebyl volný slot pro požadavek, opět se odmlčela.

Aby bylo ověřeno, nakolik toto odmlčování zkresluje finální výsledky, byl proveden orientační test na třech dávkách požadavků s variabilním maximálním počtem paralelních požadavků. Kvůli povaze testu (nejprve se vykoná jedna CRU(R)D operace, poté druhá apod) je maximálně dosahovaný počet paralelních požadavků ve skutečnosti pětina celkového počtu.

Pro přehled byly stejné testy spuštěny i pro Node.js a to jak se zpomalováním, tak bez něj.

Protože byla tato funkce implementována pro PHP, které bez ní nebylo schopné testy vůbec splnit, bylo před testy rozhodnuto, že Node.js bude nebo nebude využívat stejnou techniku v závislosti na výsledku testů: varianta, která dosáhne v těchto předběžných testech lepších výsledků bude používána v konečném srovnání.

Výsledky byly následující:

Max počet požadavků	**500 požadavků celkem / 100 paralelních**	**2000 celkem / 400 paralelních**	**6000 celkem / 1200 paralelních**
300	15399* / 14525 / 14020*	63139 / 56404 / 60968	201909 / 170900 / 193606
100	16103* / 14297 / 14430*	64904 / 58794 / 62950	224745 / 172200 / 236681
30	15951 / 15301 / 17307	63896 / 65674 / 66630	205283 / 190625 / 267370
15	16569 / 15509 / 16363	66647 / 59252 / 66739	232429 / 186267 / 300709
5	17852 / 15608 / 17916	74747 / 59550 / 73808	330444 / 188230 / 432485

Obsah buňky = PHP s limity / Node bez limitu / Node s limity

* tyto hodnoty ve skutečnosti vůbec nebyly zpomalovány protože nebylo dosáhnuto limitu

- Tabulka 7: Vliv snížení počtu paralelních požadavků na výkon

I bez použití statistických metod je pozorovatelný – a vcelku logický – trend nepřímé úměrnosti mezi snižujícím se limitem paralelních požadavků a vzrůstajícím časem potřebným k vykonání celého měření.

Vzhledem k tomu, že limitování paralelního počtu požadavků je, jak se zdá, relevantní faktor, který bude hýbat s výsledky, bylo rozhodnuto, že v ostrých testech budou data měřeny pro všechny tři platformy (Node včetně a bez zpomalení) a tato skutečnost bude součástí vyhodnocení výsledků a diskuze.

Tyto testy zároveň poukázaly na fakt, že původní rozsah měření byl příliš ambiciózní (poslední počet požadavků je téměř 110 tisíc). Blok o pouhých 6000 požadavcích trval zhruba tři a půl minuty. Pokud by potřebný čas stoupal lineárně, trvalo by jedno měření jedné platformy přes hodinu a tudíž lze předpokládat že i na výkonějším stroji by mohlo celé měření trvat pár dní.

Bylo proto nutné upravit seznam měření novým skutečnostem: čtyři nejnáročnější měření byly odstraněny a aby byl zachován dostatečný počet hodnot pro analýzu, byly mezi nové čtyři nejvyšší vloženy mezikroky. Kromě toho přibylo ještě první měření jehož součástí je jenom jeden požadavek pro každou CRU(R)D operaci.

Nový seznam tedy je: **5**, 10, 20, 30, 50, 80, 130, 210, 340, 550, 890, 1440, 2330, 3770, **4935**, 6100, **7985**, 9870, **12920**, 15970.

Dále byla nastavena určitá tolerance vůči chybám: na timeout error (mezi klientem a serverem) reaguje server znovuzařazením požadavku do fronty s náhodným zpožděním z rozmezí 0-6000ms. Pro chyby tohoto typu nebyl nastaven limit.

Stejně reaguje test na kód HTTP odpovědi jiný než 200 OK. Počet těchto případů je ale monitorován a v případě že překročí limit 100 chyb, test končí

V případě jiné nezmapované chyby, je implicitně považována za příliš závažnou a test je ukončen po uložení mezivýsledků.

4.7 Ostré měření

4.7.1 Setup

Následující měření probíhalo na lokální síti mezi dvěma počítači propojenými přímo UTP kabelem, bez prostředního prvku a bez spojení s dalšími sítěmi.

Testovací kód zodpovědný za organizaci požadavků byl umístěn na stejném stroji jako v případě předběžných testů, laptopu Acer Aspire E5-553G (procesor AMD A12-9700P 2.5GHz 4j. / 4 GB RAM).

Obě API se potom nacházely na stroji ASUS ROG GL502VM (procesor Intel Core i7 7700HQ 2.8GHz 4j. / 16 GB RAM).

PHP vykonával opět stack XAMPP 7.1.1, stejně jako JavaScript opět prostředí Node.js 6.10.0.

4.7.2 Průběh

Díky předběžným testům a zavedeným opatřením proběhlo měření na druhý pokus v pořádku.

Při prvním pokusu vyhodil po pár hodinách usilovné práce PHP server jednu odpověď 500 způsobenou odmítnutím požadavku od databáze, která celé měření zastavila a proto musela být nastavena určitá tolerance chybovosti, jak bylo popsáno v předchozí kapitole.

Test probíhal 4 hodiny, 30 minut a 18 vteřin a bylo zpracováno přinejmenším 1014525 požadavků (nejsou počítány opakované požadavky při timeout erroru).

Při čištění dat se jenom dvě hodnoty se odlišovaly od průměru v míře mimo toleranci.

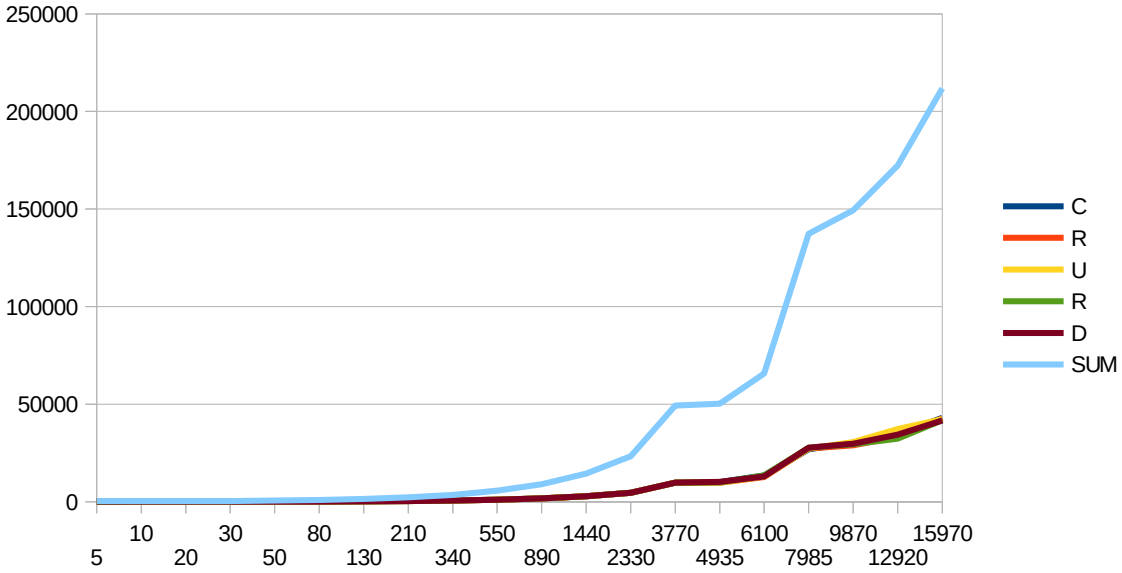
4.7.3 Výsledky

Všechny časy jsou měřeny a udávány v milisekundách.

4.7.3.1 PHP

	C	R	U	R	D	SUM
5	89	68	88	68	88	401
10	83	83	89	78	83	417
20	88	89	89	83	89	438
30	94	94	93	94	94	469
50	156	146	133	131	151	716
80	182	198	203	193	198	974
130	297	307	302	313	292	1511
210	464	460	458	448	464	2294
340	713	715	742	724	714	3607
550	1161	1130	1163	1136	1141	5731
890	1818	1818	1834	1813	1810	9092
1440	2907	2881	2912	2902	2907	14507
2330	4712	4633	4719	4673	4706	23443
3770	9949	9820	9803	9892	9914	49379
4935	10216	9928	9883	10059	10258	50344
6100	13502	12732	12989	13580	13006	65809
7985	26986	27339	27393	27818	27768	137304
9870	30412	29028	30657	29529	29748	149373
12920	34715	33391	37324	32477	34481	172388
15970	43087	42798	42365	41830	41744	211824

- Tabulka 8: Naměřené hodnoty pro PHP

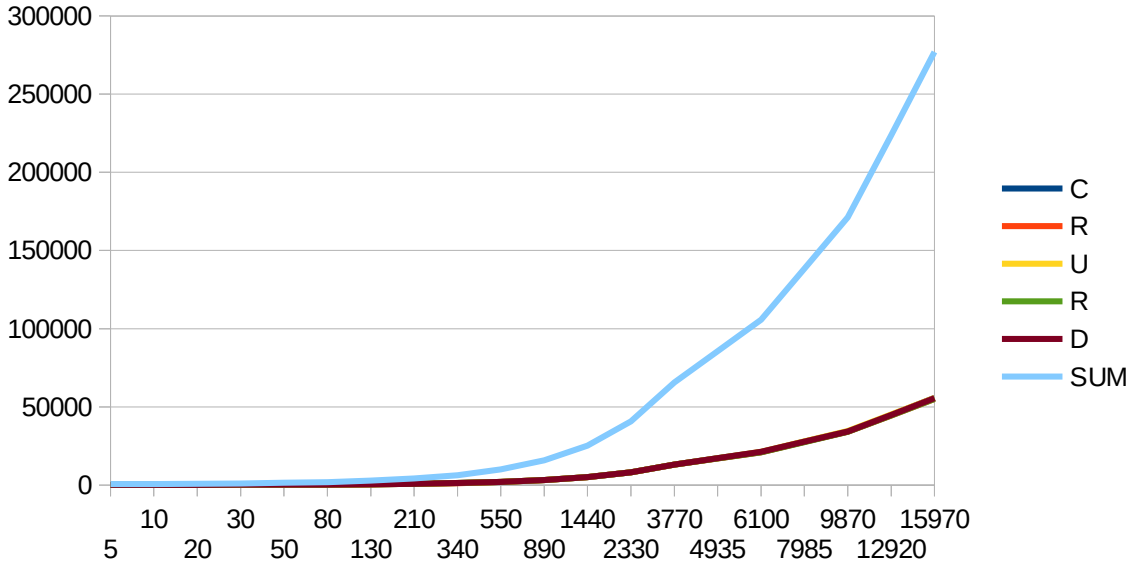


- Graf 2: Vizualizace naměřených hodnot pro PHP

4.7.3.2 Node bez limitu

	C	R	U	R	D	SUM
5	135	125	141	125	130	656
10	140	141	141	140	141	703
20	151	146	156	156	157	766
30	208	224	214	203	214	1063
50	302	265	287	271	284	1409
80	359	359	360	354	355	1788
130	563	553	558	552	568	2793
210	829	833	845	828	840	4176
340	1282	1246	1271	1267	1262	6328
550	2017	1995	2029	1985	2016	10041
890	3176	3173	3200	3163	3168	15881
1440	5049	5005	5112	5031	5065	25262
2330	8115	8085	8212	8120	8181	40713
3770	13116	13054	13176	13051	13153	65550
4935	17146	17039	17192	17023	17140	85540
6100	21178	21015	21251	21015	21226	105685
7985	27700	27520	27877	27485	27758	138340
9870	34276	34016	34607	33995	34234	171127
12920	44795	44483	45106	44509	44907	223800
15970	55648	54983	55785	54991	55496	276902

- Tabulka 9: Naměřené hodnoty pro Node.js bez limitu

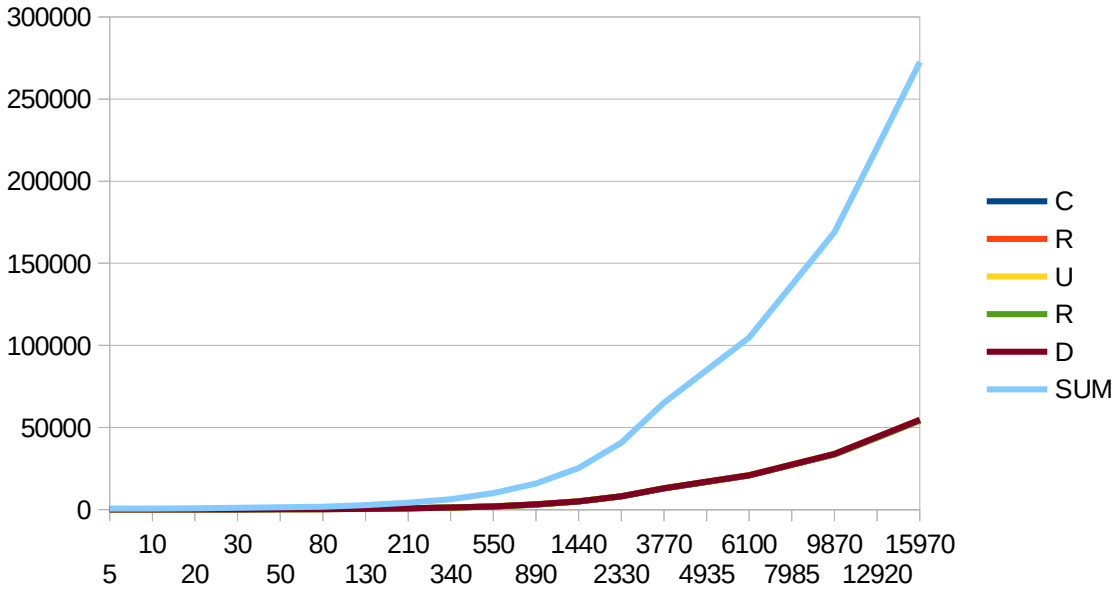


- Graf 3: Vizualizace naměřených hodnot pro Node.js bez limitu

4.7.3.3 Node s limity

	C	R	U	R	D	SUM
5	141	125	141	125	141	672
10	141	135	141	140	141	698
20	146	151	156	141	151	745
30	219	203	213	219	219	1073
50	281	276	286	271	281	1396
80	360	355	364	354	360	1793
130	552	552	562	558	570	2794
210	844	829	833	830	844	4179
340	1276	1245	1266	1253	1266	6305
550	2007	2011	2037	1996	2007	10058
890	3173	3183	3172	3148	3185	15862
1440	5047	5050	5117	5055	5071	25340
2330	8121	8107	8191	8117	8164	40700
3770	13052	12932	13155	13005	13041	65184
4935	16973	16936	17099	16906	17050	84964
6100	20931	20855	21105	20876	21022	104790
7985	27322	27225	27551	27255	27475	136828
9870	33740	33609	34031	33609	33871	168861
12920	44018	43894	44367	43918	44275	220472
15970	54441	54257	54805	54244	54734	272481

- Tabulka 10: Naměřené hodnoty pro Node.js s limity

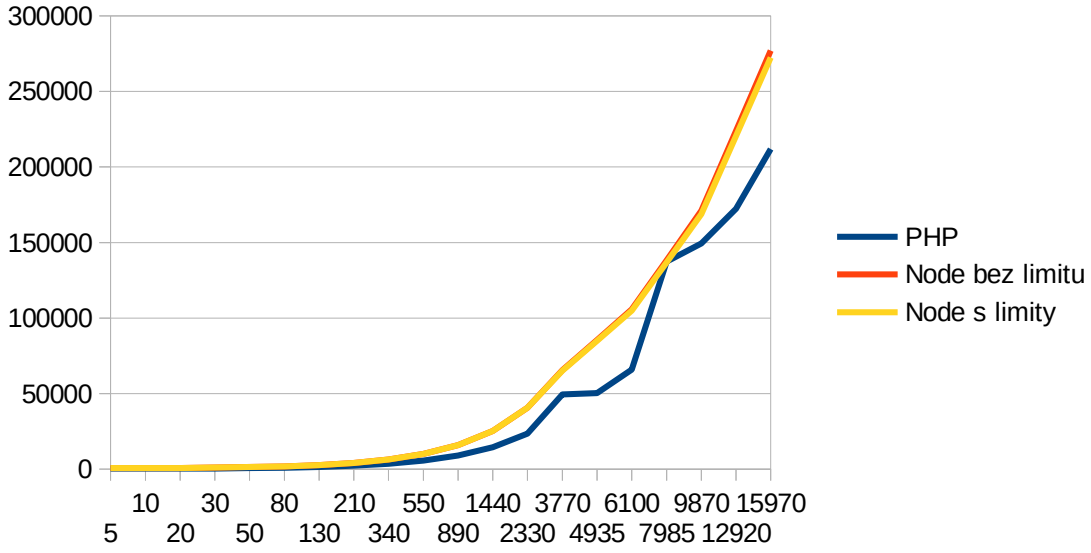


- Graf 4: Vizualizace naměřených hodnot pro Node.js s limity

4.7.3.4 Srovnání sum

	PHP	Node bez limtu	Node s limity
5	401	656	672
10	417	703	698
20	438	766	745
30	469	1063	1073
50	716	1409	1396
80	974	1788	1793
130	1511	2793	2794
210	2294	4176	4179
340	3607	6328	6305
550	5731	10041	10058
890	9092	15881	15862
1440	14507	25262	25340
2330	23443	40713	40700
3770	49379	65550	65184
4935	50344	85540	84964
6100	65809	105685	104790
7985	137304	138340	136828
9870	149373	171127	168861
12920	172388	223800	220472
15970	211824	276902	272481

- Tabulka 11: Sumy naměřených hodnot pro všechna měření



- Graf 5: Vizualizace sum naměřených hodnot pro všechna měření

4.8 Ověření hypotézy

Prvním krokem v ověřování rozdílů byl výběr varianty Node pro srovnání s PHP. Jak bylo předesláno, je to ta, která dosahuje lepších výsledků.

Protože se navzdory průvodnímu předpokladu vycházejícího z orientačních měření obě varianty liší pouze na úrovni statistické chyby, nemá tato volba žádný praktický význam.

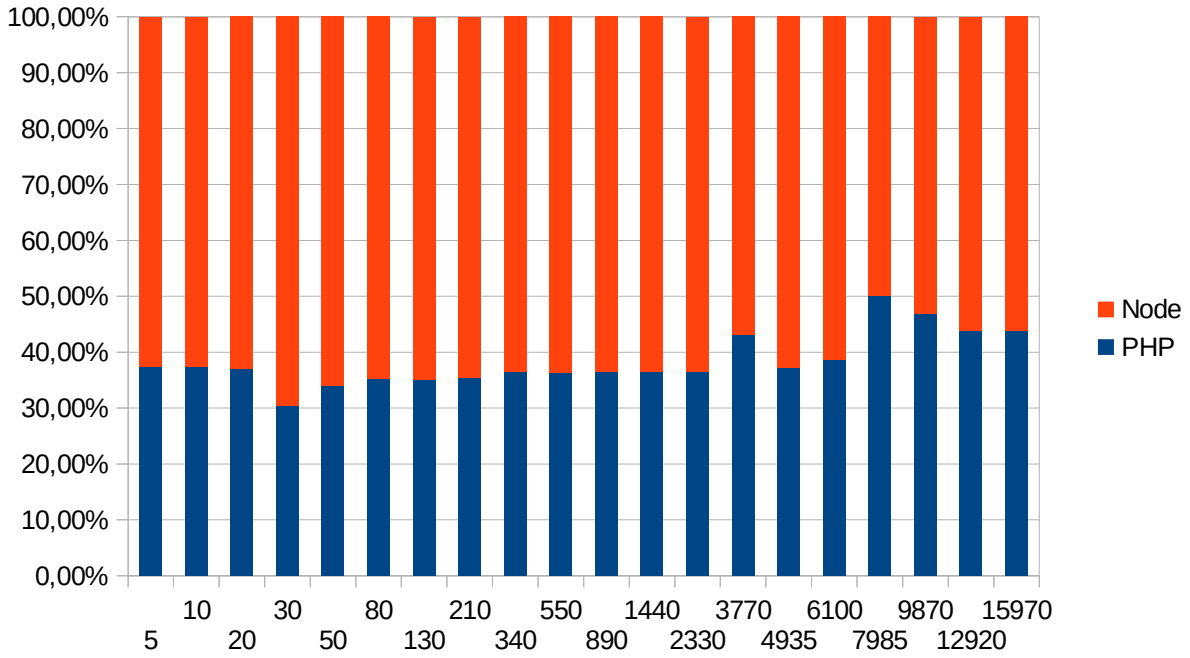
Přesto byla vybrána varianta s limity, která dosahuje v půměru o 0,42% nižších hodnot.

Ve všech kromě jednoho měření dosahuje PHP výrazně lepších výsledků: v průměru bylo ke zpracování všech požadavků potřeba o 36,73% méně času.

Tato úspora se se zvyšujícím počtem požadavků zdánlivě snižuje, nicméně ověření tohoto trendu regresní analýzou by vyžadovalo větší vzorek dat.

	PHP	**Node**	**Node (l)**	**Node / Node (l)**	**PHP / Node (l)**
5	401	656	672	97,67	59,72
10	417	703	698	100,72	59,74
20	438	766	745	102,82	58,75
30	469	1063	1073	99,04	43,71
50	716	1409	1396	100,93	51,30
80	974	1788	1793	99,72	54,31
130	1511	2793	2794	99,98	54,09
210	2294	4176	4179	99,92	54,90
340	3607	6328	6305	100,35	57,21
550	5731	10041	10058	99,83	56,97
890	9092	15881	15862	100,12	57,32
1440	14507	25262	25340	99,69	57,25
2330	23443	40713	40700	100,03	57,60
3770	49379	65550	65184	100,56	75,75
4935	50344	85540	84964	100,68	59,25
6100	65809	105685	104790	100,85	62,80
7985	137304	138340	136828	101,11	100,35
9870	149373	171127	168861	101,34	88,46
12920	172388	223800	220472	101,51	78,19
15970	211824	276902	272481	101,62	77,74
Průměr				**100,42**	**63,27**

- Tabulka 12: Sumy naměřených hodnot rozšířené o poměrné sloupce



- Graf 6: Vizualizace vývoje poměru rychlostí (hodnota 50% určuje shodný výkon)

5 Zhodnocení výsledků

V první řadě je na místě pokusit se zdůvodnit absenci rozdílu mezi verzemi měření JavaScript API. Autor si tento fakt vysvětluje tím, že Node nevyužívá procesových vláken[6] a proto musí všechny požadavky odbavovat postupně. To znamená, že ho neomezuje počet paralelních požadavků, neboť je – nezávisle na frekvenci se kterou přicházejí – řadí do fronty a odbavuje sériově. V tomto případě nezáleží na délce fronty: buďto se v ní nachází tisíc požadavků a nebo pětset požadavků, které se doplňují v momentě kdy je každý požadavek vyhodnocen. [21]

Toto vysvětlení také odpovídá na otázku, proč Node neměl problémy s paralelními požadavky na kterých při každém z větších testů před manuálním limitováním vyhořelo PHP API.

Problém s prostředím Node.js totiž tkví právě v jeho omezení na jedno vlákno procesu: Aplikace běžící v prostředí Node nejsou asynchronní jenom kvůli používání tzv. callback funkcí. Ve skutečnosti jsou skutečně asynchronní pouze nativní funkce využívající vstupně výstupní kanály, jako například pevný disk, síťovou komunikaci, nebo ostatní služby spuštěné v operačním systému (například databázový server).[6]

Provedené testy ovšem tolik nevyužívaly vstupně výstupních kanálů (pouze databázový server a požadavky na něj kladené byly triviální), ale zato obsahovaly parsování URL a parametrů a ověřování přihlašovacích údajů bcrypt hashováním, což jsou funkce náročné na využití CPU, které Node, jak bylo zmíněno, neumí paralelizovat.

Pokud by byly uměle zpomalené nebo komplikované požadavky na DB (aby více odpovídaly reálným implementacím, které budou sotva operovat nad jednou tabulkou), nebo pokud by logika API například vyžadovala síťovou komunikaci s API třetích stran, mohly by se misky vah posunout výrazně ve prospěch Node. Tato hypotéza by měla být předmětem dalšího šetření.

21 Data předběžného měření byla pravděpodobně zkreslena faktem, že v době testu nebyl výkon stroje plně koncentrován na tento úkol a že se test i API nacházely na stejném zařízení

Dalším zajímavým poznatkem, který je vhodné zmínit je fakt, že kvůli PHP musel být upraven test. PHP už od nejnižších iterací sice fungovalo rychleji, ale za cenu toho, že začalo v momentě přetížení úplně odmítat odbavovat více požadavků.

Samozřejmě dává smysl, že API, na kterém se sejde v rozmezí dvaceti milisekund pár tisíc požadavků, nelze vytýkat že tento nápor nezvládne: Node a PHP mají v tomto případě principiálně rozdílné, ale stejně validní přístupy: Node bude na nové požadavky reagovat pomaleji, ale zato jistěji. Na druhou stranu PHP rychle odmítne požadavek který nemůže splnit a záleží na klientu, jestli se rozhodne požadavek opakovat.

V obou případech je zodpovědnost poskytovatele API podniknout kroky proto, aby jeho aplikace nebalancovala na hranici technických možností.

Autor ještě cítí potřebu zmínit ještě několik doplňujících myšlenek.

Programovací jazyk, respektive jeho interpreter, je pouhou součástí soukolí umožňující běh webových aplikací: lepší efektivitu odbavování požadavků může poskytnout nový web server (například Nginx), nebo databázový systém (NoSQL databáze), nebo dokonce i alternativní interpreter jazyka, jak ukázal případ HHVM versus Zend Engine 2.

Pokud je možné vyměnit jednu součást prostředí bez nutnosti přepisovat celou aplikaci do nového jazyka, měla by se tato možnost využít.

Dále API představuje pouze jedno užití webu. Nezávisle na výsledku je potřeba si uvědomit, že tato ani jiná práce nemůže definitivně rozsoudit spor o rychlosti a vhodnosti využití obou jazyků, neboť ilustruje pouze omezené použití.

Kromě toho kvalitu jazyka určuje více metrik, než výkon parseru. Pro budoucnost JavaScriptu může být zajímavý aktuální trend pokusů vytvořit z něj programovací lingua franca: kromě prohlížeče a web serveru totiž již poměrně úspěšně probíhají pokusy typu React Native – mobilní aplikace psané také v JavaScriptu

6 Závěr

Primárním cílem práce bylo objektivně změřit a následně zhodnotit výkon dvou programovacích jazyků: PHP a JavaScript při realizaci webového aplikačního rozhraní: kromě obou API samotných byl pro potřeby práce vytvořen testovací skript, který na tyto API odesílá vzrůstající počet požadavků a měří čas potřebný k jejich vykonání.

Obě API sdílely stejný teoretický návrh data-flow a databázové vrstvy a byly nasazeny ve stejných laboratorních podmínkách, aby byly výsledky co nejlépe srovnatelné.

Během předběžných měření muselo být zadání upraveno, neboť se ukázalo, že původní plán byl příliš ambiciózní a měření by trvalo neúnosnou dobu, rozsah byl proto snížen.

Další, výraznější, problém vyvstal z faktu, že PHP reaguje na vysoký počet paralelních požadavků rychlým odmítáním dalších požadavků a proto musel být tento počet spojení na straně testu manuálně regulován a test byl dále upraven takovým způsobem aby byl schopen se s timeout chybami vypořádat znovuzařazením neúspěšného požadavku do fronty. Protože se zdálo, že tento mechanismus bude data zkreslovat, bylo JavaScript API měřeno ve dvou variantách: za použití stejného limitu a bez něj.

Test dopadl výrazně ve prospěch PHP oproti oběma variantám měření Node.js.

Autor si tento závěr vysvětluje povahou testu, která obsahovala příliš málo vstupně výstupních operací (přesněji řečeno ty byly příliš jednoduché a krátké). Na déle trvajících operacích tohoto typu by mohl Node.js prokázat sílu asynchronicity a tuto hypotézu by bylo vhodné prozkoumat dalším měřením s upraveným zadáním.

Místo toho mohlo PHP využít své podpory multithreadingu a to mu poskytlo značnou výhodu v procesorově náročných operacích.

V rámci vedlejšího cíle, rešerše teoretických znalostí potřebných k realizaci obou API byly zmapovány pravidla architektury REST, tyto byly konfrontovány s problémy implementací a také byly rozebírány další praktická hlediska a doporučení.

7 Zdroje

1. JACOBSON, D. -- BRAIL, G. -- WOODS, D. APIs: A Strategy Guide.
   O'Reilly Media, 2011. ISBN: 978-1-4493-0891-9.
2. MASSE, M. REST API Design Rulebook.
   O'Reilly Media, 2011. ISBN: 978-1-4493-1049-3.
3. LOPEZ, A. Learning PHP 7.
   Packt Publishing, 2016. ISBN: 978-1-78588-341-5.
4. ABEYSINGHE, S. RESTful PHP Web Services.
   Packt Publishing, 2008. ISBN: 978-1-84719-553-1.
5. KIESSLING, D. The Node Beginner Book.
   Leanpub, 2015. ISBN: 978-1-4716-2844-3.
6. KIESSLING, D. The Node Craftsman Book.
   Leanpub, 2015.
7. BOJINOV, V. RESTful Web API Design with Node.js.
   Packt Publishing, 2015. ISBN: 978-1-78398-587-6.
8. SVATOŠOVÁ, L. -- KÁBA, B. Statistické metody I.
   ČZU v Praze, 2013- ISBN: 978-80-213-1672-0.
9. ALLAMARAJU, S. RESTful Web Services Cookbook.
   O'Reilly Media/Yahoo Press, 2010. ISBN: 978-1-4493-8248-3.
10. WEBBER, J. -- PARASTATIDIS, S. -- ROBINSON, I. REST in practise.
    O'Reilly Media, 2010. ISBN: 978-1-4493-9494-3.
11. RICHARDSON, L. -- RUBY, S. RESTful Web Services.
    O'Reilly Media, 2007. ISBN: 978-0-596-52926-0.
12. LECKY-THOMPSON, E. -- NOWICKI S.D. PHP 6.
    Praha: CPress, 2010. ISBN: 978-80-251-3127-5.
13. POWERS, S. Learning Node.
    O'Reilly Media 2016. ISBN: 978-1-4919-4306-9.

14. COOK, F. Node.js Essentials.

Packt Publishing, 2015. ISBN: 978-1-78528-594-3.

15. HOWARD, D. Node.js for PHP Developers.

O'Reilly Media, 2012. ISBN: 978-1-4493-3379-9.

16. WIKIPEDIA. Application programming interface.

[online], 2016. Dostupné z: https://en.wikipedia.org/wiki/Application_programming_interface

17. DEGGES, R. Why I Love Basic Auth

[online], cit. 2017. Dostupné z: https://www.rdegges.com/2015/why-i-love-basic-auth/

18. JAVOREK, J. Jak psát API.

[online], 2014. Dostupné z: https://github.com/honzajavorek/jakpsatapi

19. FIELDING, R. Architectural Styles and the Design of Network-based Software Architectures. University of California, 2000.

20. ECKERSLEY, P. How secure is HTTPS today? How often is it attacked?

[online], 2011. Dostupné z: https://www.eff.org/deeplinks/2011/10/how-secure-https-today

21. GARRETT, J. J. Ajax: A New Approach to Web Applications

[online], 2005. Dostupné z: http://adaptivepath.org/ideas/ajax-new-approach-web-applications/

22. CRACKSTATION. Salted Password Hashing - Doing it Right

[online], cit. 2017. Dostupné z: https://crackstation.net/hashing-security.htm

23. WIKIPEDIA. PHP

[online], cit. 2017. https://en.wikipedia.org/wiki/PHP

24. AUTH0. JWT.

[online], cit 2017. Dostupné z: https://jwt.io/

25. WIKIPEDIA, JavaScript

[online], cit. 2017. Dostupné z: https://en.wikipedia.org/wiki/JavaScript

26. ZEND TECHNOLOGIES LTD. Turbocharging the Web with PHP7

[online], cit 2017. Dostupné z: https://www.zend.com/en/resources/php7_infographic