

České vysoké učení technické v Praze
Fakulta elektrotechnická
Katedra kybernetiky



Bakalářská práce

Ověření škálovatelnosti Google App Engine aplikací

Jakub Škvára

Vedoucí práce: Ing. Marek Šmíd

Studijní program: Softwarové technologie a management, Bakalářský

Obor: Softwarové inženýrství

25. května 2011

Poděkování

Chtěl bych poděkovat svému vedoucímu panu Ing. Marku Šmídovi, za ochotu a pomoc při psaní této bakalářské práce.

Prohlášení

Prohlašuji, že jsem práci vypracoval samostatně a použil jsem pouze podklady uvedené v přiloženém seznamu.

Nemám závažný důvod proti užití tohoto školního díla ve smyslu §60 Zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon).

V Praze dne 27.5.2011

.....

Abstract

This bachelor thesis is describing advantages and disadvantages of cloud computing. Which applications are useful to place to cloud and also limitations which cloud brings. In part one, I am describing what is cloud generally and introduce various possibilities and compare them to readers. I have chosen App Engine from Google for my research, therefore I am describing this cloud in more detail.

In part two I am describing programming a several small applications for testing and comparing performance of storage. Then I made one larger application and described its structure and architecture. I tested scalability and performance of this application with large amount of requests. All results are summarized and compared in conclusion.

Abstrakt

Tato bakalářská práce popisuje výhody a nevýhody použití cloudu. Které aplikace se hodí pro umístění do cloudu a také omezení které cloud přináší. V první části popisuji co je to cloud obecně a seznamuji čtenáře s různými možnostmi a porovnávám je. Pro ověření jsem si vybral App Engine od firmy Google a popisuji tento cloud podrobněji.

V druhé části popisuji jak jsem naprogramoval několik menších aplikací pro otestování a porovnání rychlosti uložště. Dále jsem napsal jednu větší aplikaci a popsal její strukturu a architekturu. Na této aplikaci jsem pomocí velkého počtu požadavků otestoval škálovatelnost a rychlost App Engine. Všechny výsledky jsem přehledně shrnul a porovnal v závěru.

Obsah

1	Úvod: Výběr tématu	1
1.1	Co je to cloud	1
1.2	Motivace pro toto téma	1
1.3	Průběh testování	2
2	Teorie: Obecný popis cloudu	3
2.1	Novinka jménem cloud computing	3
2.2	Infrastructure as a Service, Platform as a Service, Software as a Service	3
2.3	Horizontální a vertikální škálování	4
2.4	Různé druhy pohledu na cloud computing	5
2.5	Porovnání Amazon Web Services, Windows Azure a Google App Engine	6
2.5.1	Amazon Web Services - Elastic Compute Cloud	6
2.5.2	Microsoft Azure	7
2.5.3	Google App Engine	7
2.6	API služby	9
2.6.1	Memcache	9
2.6.2	Mail	10
2.6.3	XMPP	10
2.6.4	Task Queues	10
2.6.5	URL Fetch	10
2.6.6	Blobstore	10
2.6.7	Images	11
2.6.8	Users	11
2.6.9	OAuth	11
2.6.10	Capabilities	12
2.6.11	Channel	12
2.6.12	Multitenancy	12
2.7	Omezení cloudu	12
2.8	Vývoj pro App Engine	13
2.9	Zajímavé aplikace	14
2.10	Budoucnost cloudů	15
3	Implementace: Vývoj v App Engine	17
3.1	Výběr cloudu	17
3.2	Aplikace	17

3.3	Výběr frameworku: Slim3	18
3.4	Práce s Datastore API s pomocí Slim3 frameworku	19
3.5	Model aplikace	22
3.5.1	Entity a metatřídy entit	22
3.5.2	DAO	23
3.5.3	Service	23
3.5.4	Converter	23
3.5.5	Validator	23
3.5.6	DTO	23
3.6	Propojení modelu - Google Guice	24
3.7	Slim3 Controller	25
3.8	View vrstva	26
3.9	Shrnutí implementace	26
4	Testování: Porovnání rychlosti práce s Datastore a testování zátěže	27
4.1	Hlediska a způsob testování	27
4.2	Testování Datastore	27
4.3	Test výběru	28
4.4	Test vkládání	28
4.5	Test úprava	29
4.6	Test mazání	29
4.7	Porovnání výsledků testů práce s Datastore	30
4.8	Testování zátěže	31
5	Závěr: Zhodnocení	35
5.1	Jednoduché škálování	35
5.2	Zhodnocení porovnání Datastore API, JPA, JDO	35
5.3	Zhodnocení porovnání testů zátěže	35
5.4	Osobní přínos	36
A	Seznam použitých zkratk	37
B	Obsah přiloženého CD	39

Seznam obrázků

2.1	IaaS, PaaS, SaaS (Zdroj: http://filiph.net/slides/idf-cloud/#slide11)	4
2.2	Horizontální a vertikální škálovatelnost (Zdroj: http://filiph.net/slides/idf-cloud/#slide9)	5
2.3	Amazon Web Services - logo	7
2.4	Windows Azure - logo	7
2.5	Google App Engine - logo	8
2.6	Zatížení App Engine serverů v době spuštění aplikace Google Moderator pro Bílý Dům [?]	14
3.1	Diagram tříd modelu	22
4.1	Graf porovnání výběru 1 000 entit	29
4.2	Graf porovnání vkládání 100 entit	30
4.3	Graf porovnání úpravy 100 entit	31
4.4	Graf porovnání mazání 100 entit	32
4.5	Graf porovnání mazání 100 entit	33
B.1	Seznam přiloženého CD — příklad	39

Seznam tabulek

2.1	Tabulka kvót	8
4.1	Tabulka porovnání výběru 1 000 entit (údaje jsou v uvedeny v milisekundách)	28
4.2	Tabulka porovnání vkládání 100 entit (údaje jsou v uvedeny v milisekundách)	29
4.3	Tabulka porovnání úpravy 100 entit (údaje jsou v uvedeny v milisekundách) .	30
4.4	Tabulka porovnání mazání 100 entit (údaje jsou v uvedeny v milisekundách) .	31
4.5	Tabulka porovnání zátěže	33

Kapitola 1

Úvod: Výběr tématu

1.1 Co je to cloud

Pro mou bakalářskou práci jsem si vybral poměrně nové a nezmapované téma a to popis cloudových služeb. Jedná se o nový způsob hostování internetových aplikací a ukládání dat na webu. U klasického způsobu ukládání dat se použije jeden nebo více nezávislých serverů a na něj se nahraje ve většině případů pouze jedna aplikace. Cloud nám přináší nový přístup, místo abychom měli jeden stroj, použijeme více strojů dohromady, které budou spolupracovat. Aplikace žádný rozdíl nepozná a může zde zároveň běžet mnoho programů.

Důležitým důvodem k využívání cloudů je větší efektivita využití hardware. Pokud je aplikace málo používaná, například v nočních hodinách, jsou zdroje využívány jinými aplikacemi, které mohou obsluhovat uživatele z jiné části světa, kde je třeba odpoledne. Nebo pokud je aplikace vytížena a nestíhá, může systém spustit další instanci té samé aplikace. O rozložení zátěže a správu počtu aplikací se stará cloud samotný.

Další výhodou je možnost dynamicky navyšovat hardwarové parametry infrastruktury přidáváním strojů bez nutnosti přerušit provoz. Cena hardwaru se postupem času snižuje, podle Mooreova zákona¹ z roku 1965 se složitost součástek každý rok zdvojnásobí při zachování stejné ceny. Tento zákon platí dodnes a předpokládá se, že bude platit minimálně do roku 2015 až 2020. S možností dynamicky přidávat hardware budeme připraveni rozšiřovat infrastrukturu podle potřeby. O toto se ale starají společnosti poskytující cloudy, nám tedy odpadá nutnost starat se o hardware.

Nevýhodou cloudu je, že nemáme hardware plně pod kontrolou a musíme se spolehnout na společnost poskytující hosting. Pokud bychom chtěli provozovat vlastní cloud, museli bychom mít obrovskou infrastrukturu a vyvinout vlastní řešení pro efektivní vytížení zdrojů všech aplikací. Takovouto možnost má jen pár společností na světě, které patří k těm největším.

1.2 Motivace pro toto téma

Hlavní motivací k výběru tohoto tématu bylo, že se v poslení době stává cloud čím dál tím více používanější. Jedná se o úplně nový přístup a tak o cloudech zatím nenajdeme mnoho

¹ Mooreův zákon - http://en.wikipedia.org/wiki/Moore's_law

zdrojů. Přišlo mi zajímavé vyzkoušet a prověřit možnosti jednoho z těchto cloudů. Jak se bude chovat při vysokém počtu požadavků, kde jsou limity takového cloudu a v neposlední řadě kolik hardwarových prostředků bude při zátěži spotřebováno a jaká bude cena.

1.3 Průběh testování

V mé bakalářské práci jsem ověřil a porovnal rychlost různých řešení práce s uložištěm dat na cloudu. Pro tento test jsem připravil jednoduchou aplikaci. Poté jsem napsal větší a složitější aplikaci, kterou jsem následně otestoval vysokým počtem požadavků. Porovnával jsem jak se bude cloud chovat a jak zařídí rozložení zátěže.

Kapitola 2

Teorie: Obecný popis cloudu

2.1 Novinka jménem cloud computing

V poslední době se čím dál tím více začíná mluvit o cloud computingu. Jedná se o nový typ hostingu a ukládání webového obsahu vůbec. Oproti klasickému způsobu, kde máme jeden konkrétní server, na určeném místě, se svojí danou pamětí, procesorem a pevným diskem, nám tento nový přístup umožňuje nezabývat se hardwarem, pro tento typ se používá název platforma.

Definice se značně různí, takže použiji verzi Národního institutu standardů a technologií (National Institute of Standards and Technology) [?], která volně přeložena zní: cloud computing je způsob poskytování sdílených škálovatelných zdrojů (výpočetní kapacity, uložení, služby, aplikace, ...), k nimž je přístupováno skrz síť a které jsou uživateli k dispozici ihned na vyžádání.

Mezi hlavní výhody je považováno snižování nákladů a zvyšování efektivity. Nemusíme vlastnit hardware, za jehož pořízení a správu je potřeba vynaložit nemalé finanční náklady, přičemž většina zdrojů není plně zatížena. Zvyšování efektivity se projevuje hlavně placením jen za využití zdroje. Pokud bychom měli vlastní infrastrukturu, tak v době nižší aktivity nevyužíváme možnosti serverů naplno a platíme vlastně za nevyužití zdroje. Naopak v cloudu jsou naše prostředky sdíleny s ostatními a v době neaktivity můžou být nabídnuty někomu jinému.

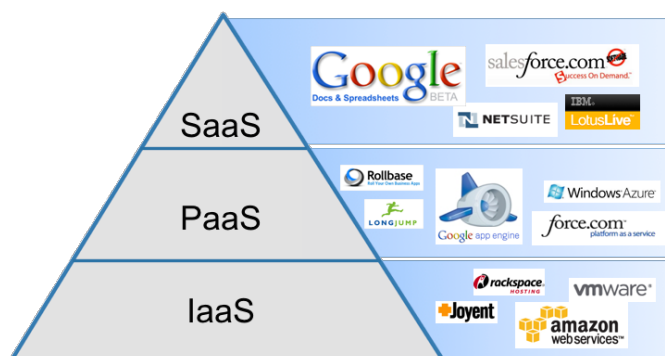
2.2 Infrastructure as a Service, Platform as a Service, Software as a Service

Existují různé nabídky cloudových řešení pro efektivní využívání zdrojů hardware pro více aplikací. Nejzákladnější je Iaas - Infrastructure as a Service (Infrastruktura jako Služba) - jedná se například o Amazon EC2 [?] cloud, kde platíme jen za spotřebované zdroje, které reálně využijeme a na hardware si můžeme sami instalovat co potřebujeme.

U PaaS - Platform as a Service (Platforma jako Služba) již nemáme přístup k hardwaru, to znamená že nelze instalovat žádný software, ale máme zde připravená API pro různé

služby které můžeme využívat a většinou i další nástroje pro vývoj na lokálním stroji a pro deployment.

Nejvíce jsme od fungování služby odstíněni u SaaS - Software as a Service (Software jako Služba) - jedná se například o online e-mailové služby jako `gmail.com` anebo `seznam.cz`, obrázkové galerie jako `flickr.com` anebo `rajce.cz`, tedy služby které používáme prostřednictvím internetu a nezajímá nás jak a kde jsou data uložena a nemáme ponětí, jak jsou naprogramované.



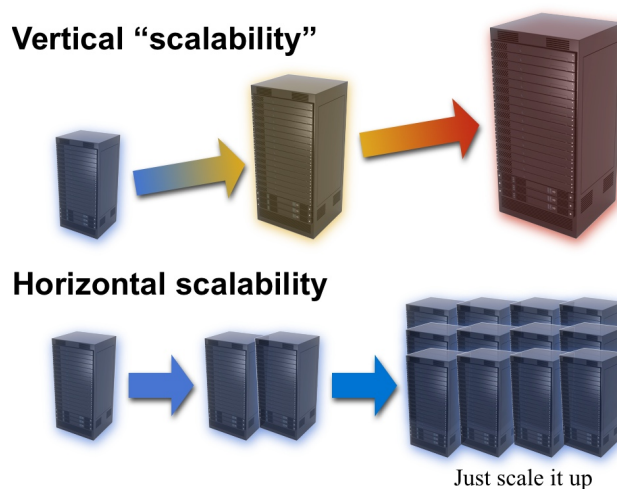
Obrázek 2.1: IaaS, PaaS, SaaS (Zdroj: <http://filiph.net/slides/idf-cloud/#slide11>)

2.3 Horizontální a vertikální škálování

Nové služby a především sociální sítě s rychlým nárůstem uživatelů a potřebou dynamicky měnit počet serverů donutily programátory a správce přemýšlet o novém způsobu ukládání a organizace dat. Pokud náš server nestíhá, tak máme dvě možnosti, jak tento problém řešit. Prvním z řešení je vertikální škálování, to znamená že koupíme silnější hardware, přidáme procesor, paměť a další komponenty podle potřeby. Nevýhodou tohoto řešení ale je, že takto nejde infrastruktura rozšiřovat do nekonečna, protože po čase narazíme na hardwarové limity.

Druhou z možností je nakoupení více serverů. Nemusí být ani velmi výkonné, ale řešení spočívá v propojení těchto počítačů dohromady, čímž můžeme zvětšovat naši infrastrukturu bez omezení. Můžeme toto řešení přirovnat k problému z běžného života, kdy potřebujeme převést určitý počet osob z jednoho místa na druhé. Můžeme objednat autobus, který jednoduše řeší tento problém. Pokud počet osob naroste, tak můžeme objednat autobus s větší kapacitou, jenže pokud se bude počet osob zvyšovat, tak časem již nenajdeme tak velký autobus pro přepravu všech osob. Takže jako řešení budeme muset objednat více vozidel, ale ty poté budeme moci efektivně zaplňovat podle počtu osob. Nevýhoda tohoto řešení spočívá v složitějším správě infrastruktury, potřebujeme software, který se stará o propojení, synchronizaci a spolupráci všech částí systému, nebo pokud nějaký stroj přestane fungovat, musíme zajistit, aby ho ihned zastoupil jiný se stejnou funkcí jako předchozí.

Pro cloud computing se obecně vžila značka mraku (v angličtině znamená cloud mrak) a vznikla proto, že na obrázcích a schématech se ve většině případů mrakem značí internet a vzdálená zařízení, které nejsou uloženy u nás. A to právě z toho důvodu, že jsou tyto služby většinou přístupné skrze internet a přistupujeme k nim vzdáleně.



Obrázek 2.2: Horizontální a vertikální škálovatelnost (Zdroj: <http://filiph.net/slides/idf-cloud/#slide9>)

2.4 Různé druhy pohledu na cloud computing

Cloud computing se jako každá jiná novinka potýkal s různými názory od těch pozitivních až po ty negativní. Někteří tvrdili, že se jedná jen o buzzword¹, který má nalákat nové zákazníky, jiní predikovali, že se takovýto princip nemůže nikdy uchytit. Zde je pár výroků z doby, kdy nebyl tolik rozšířený:

The interesting thing about Cloud Computing is that we've redefined Cloud Computing to include everything that we already do... I don't understand what we would do differently in the light of Cloud Computing other than change the wording of some of our ads.

Larry Ellison, Oracle Corporation CEO, Wall Street Journal, 26. září 2008

Larry Ellison říká, že cloud computing je jen pojmenování toho, co již dávno používáme a že jediná změna, která je potřeba je změna textů u reklam. Je pravda, že některé velké společnosti, jako třeba Oracle anebo Google používali tento přístup již mnohem dříve než vzniknul samotný název, ale pravdou je, že v posledních dvou letech se začal cloud computing používat masivněji a to hlavně díky možnosti pronájmu cloudů. Nyní si mohou programátoři vyzkoušet pracovat s cloudy a použít je i pro své menší aplikace, bez nutnosti spravovat a starat se o velké množství strojů.

A lot of people are jumping on the (cloud) bandwagon, but I have not heard two people say the same thing about it. There are multiple definitions out there of "the cloud."

Andy Isherwood, HP's vice president, ZDnet News, 11. prosinec 2008

¹módní slovo

Andy Isherwood naznačuje, že není přesně daná definice toho, co ještě cloud je a co již není. Je to způsobeno tím, že po vzniku tohoto názvu chtěl každý s více než jedním serverem označovat svoje služby jako cloudové. To vedlo spíše ke zmatení, ale v poslední době se toto slovní spojení ustálilo pro farmu serverů se snadnou škálovatelností a jednoduchou možností přidat novou instanci.

It's stupidity. It's worse than stupidity: it's a marketing hype campaign. Somebody is saying this is inevitable — and whenever you hear somebody saying that, it's very likely to be a set of businesses campaigning to make it true.

Richard Stallman, Founder of GNU Project and Free Software Foundation, The Guardian, 29. září 2008

Richard Stallman má na cloud poněkud negativní názor a pro The Guardian vyslovil, že se jedná o hloupost a jde jen o nafouknutou bublinu podpořenou businessovými kampaněmi. Kritizoval hlavně uložení dat mimo naši vlastní kontrolu a nutnost spolehnout se na společnost, které dáváme naše data a aplikace k dispozici. Nikdo nám nemůže na sto procent zaručit, že bude tato společnost fungovat i po několika letech. Navíc jsme většinou vázáni na API rozhraní, služby a možnosti platformy určené danou společností.

Pokud budeme chtít přenést službu k jinému poskytovateli cloudu, bude nám to s největší pravděpodobností činit nemalé potíže, v angličtině se používá termín *lock-in*, což znamená doslova zamknutí. V poslední době sice vznikají návrhy na jednotná rozhraní a sjednocení rozhraní těchto služeb, aby byl přechod co nejjednodušší, ale ty se bohužel zatím nesetkaly s větším rozšířením. Do budoucna by to mohla být jedna z klíčových vlastností při rozhodování, kterou službu zvolit.

2.5 Porovnání Amazon Web Services, Windows Azure a Google App Engine

Největší konkurenti Google App Engine jsou Amazon Web Services - EC2 (Elastic Compute Cloud) a Microsoft Azure. Pokud budeme mít zakázku, pro kterou je nejvýhodnější použít cloudovou infrastrukturu, budeme se pravděpodobně rozhodovat mezi těmito třemi, jedná se o velké a známé společnosti s rozsáhlým zázemím. Je tedy málo pravděpodobné, že by ze dne na den přestaly provozovat svoje služby, což může být velký problém u menších nebo méně známých společností.

2.5.1 Amazon Web Services - Elastic Compute Cloud

Amazon EC2 je spíše blíže modelu IaaS, takže dostaneme hardware s kterým si můžeme dělat co chceme, instalovat libovolný software a musíme si ho sami spravovat. Největší výhodou je rychlé přidávání nových serverů kdykoliv potřebujeme a platba jen za spotřebované zdroje. Navíc není tento cloud vázán žádnými API a omezeními, takže pokud budeme chtít přesunout naši aplikaci na náš server, tak nebudeme muset měnit aplikaci a to platí i naopak, tedy pro přesun aplikace na cloud. Jedná se čistě o pronájem hardwaru. Neýhoda Amazon Web Services je v tom, že platíme hned jakmile nahrejeme naši aplikaci, neexistují žádné volné



Obrázek 2.3: Amazon Web Services - logo

kvóty jako u App Engine. Amazon v rámci svých služeb nabízí i další možnosti, například speciální relační i nerelační uložště a další produkty, celý seznam je možné najít na stránce <http://aws.amazon.com/products/>.

2.5.2 Microsoft Azure



Obrázek 2.4: Windows Azure - logo

Microsoft Azure je již více podobný App Engine, jedná se o PaaS, máme zde již připravené prostředí pro několik jazyků: .NET (C# a VisualBasic), C++, PHP, Ruby, Python a Java. Výhodou je, že můžeme použít klasickou relační databázi nazvanou SQL Azure Database (SAD), což se vyplatí pokud migrujeme nějaký projekt postavený na relační databázi, ale tento typ je hůře škálovatelný. Vedle SAD můžeme použít i Azure Storage, která osahuje nerelační tabulky, tabulky pro velké objemy dat (blobs) a fronty (queues). Azure má speciální staging prostředí, kde můžeme přímo na cloudu vyvíjet aplikaci a nedostane se k ní nikdo, dokud není připravena na spuštění. V tomto prostředí také můžeme spouštět aplikaci v debug režimu, což nám umožňuje například nastavovat breakpoints, pokud potřebujeme aplikaci ladit přímo na cloudu. Azure také umožňuje propojení s Microsoft Live službami a s vývojovým prostředím Microsoft Visual Studio. Stejně jako u Amazonu platíme ihned jakmile nasadíme aplikaci do plného provozu, aby ji mohli vidět i ostatní. Azure je určitě výhodné použít, pokud vyvíjíme aplikace v technologiích od Microsoftu, naše infrastruktura je na těchto technologiích založena anebo pokud používáme jako vývojový nástroj Visual Studio.

2.5.3 Google App Engine

Oproti dvěma předchozím má App Engine hlavní výhodu v tom, že nemusíme platit ihned jak aplikaci nahrajeme na cloud. Jsou zde nastavené kvóty, které jsou velmi vysoké a je potřeba velká návštěvnost pro jejich přesáhnutí, což je pro začínající aplikaci výhodné obzvláště v České republice. Takže pokud začínáme se startupem, nemusíme se v počátcích obávat velkých investic a pokud bude náš projekt úspěšný a bude obsluhovat velký počet požadavků,



Obrázek 2.5: Google App Engine - logo

tak poté budeme muset platit jen za přesáhnuté limity, které se každých 24 hodin vynulují. Některé limity jsou nastaveny napevno a nejdou zvýšit ani za poplatek, je to kvůli tomu, aby se nemohlo stát, že jedna aplikace zaneprázdní celý cloud, což by mělo za následek zpomalení i ostatních aplikací. Tyto limity jsou naštěstí velmi vysoké, například datastore API má maximálně 141 241 791 volání za den a 784 676 volání za minutu. zde je přehledná tabulka kvót a omezení. Kvóty se průběžně navyšují, takže tato data jsou platná pro květen 2011 [2.1](#).

Tabulka 2.1: Tabulka kvót

Služba	Omezení
Procesorový čas	6.5 hodny/den
Odchozí data	1 GB/den
Přijatá data	1 GB/den
Uložená data v Datastore	1 GB
Počet indexů v Datastore	200
Volání Mail API	7 000/den
Příjemců mailu	2 000
Volání URL Fetch API	657 084
URL Fetch API odeslaná data	4 GB/den
URL Fetch API přijatá data	4 GB/den
Volání XMPP API	46 310 400/den
XMPP API odeslaná data	1 046 GB/den
Pdeslané XMPP zprávy	46 310 400/den
Poslaných XMPP pozvánek	100 000/den
Volání Channel API	46 310 400/den
Vytvořených Channel spojení	8 640/den
Channel API odeslaná data	1 046 GB/den
Volání Task Queue API	100 000/den
Uložených úkolů	1 000 000/den
Velikost uložených úkolů	1 GB/den
Nahrání aplikací	1 000/den

Standardně se aplikace nahraje na adresu jako doména třetího řádu ve tvaru `jmeno-aplikace.appspot.com`

a pokud potřebujeme můžeme nastavit i doménu vlastní. Další výhodou App Engine je možnost provozovat více verzí stejné aplikace najednou. Nahrají se pak jako poddomény, takže například `verze.jmeno-aplikace.appspot.cz`. Tyto verze mohou běžet na cloudu zároveň a v administraci se dá nastavit, která bude výchozí. Toto do jisté míry nahrazuje staging area z Azure, výhodou zde je, že můžeme mít libovolný počet verzí. App Engine bohužel podporuje jen jazyky Java, Python a Go². Díky různým projektům, které jsou schopny přeložit další jazyky do javovského bytekódu, zde tedy můžeme spouštět velké množství dalších jazyků i když je to vykoupeno nižší rychlostí. Můžeme zde tedy používat jazyky jako: Groovy, Scala, Ruby s pomocí JRuby, PHP díky projektu Quercus (viz dále), JavaScript za pomoci Rhino a další. Jedním z důvodů přidání Javy do App Engine byla právě možnost běhu dalších jazyků nad Java Virtual Machine. Co se Javy týká, tak nejsou povoleny všechny třídy, nemůžeme například vytvářet nová vlákna, nemůžeme vytvářet nové soubory a některá volání třídy `System` nedělají nic, například `System.exit()` a `System.gc()`. Seznam všech povolených tříd je možné najít na adrese <http://code.google.com/appengine/docs/java/jrewhitelist.html>.

Kvůli těmto omezením bohužel nemůžeme použít všechny knihovny, anebo musíme použít upravenou verzi pro App Engine. Seznam nejpoužívanějších frameworků a knihoven s popisem zda jsou kompatibilní a případné nastavení pro App Engine je dostupný na adrese <http://groups.google>. Na App Engine nemáme jistotu, že bude naše aplikace přímo připravena, protože kvůli tomu, aby se šetřily prostředky, jsou nahrány jen aktivní a využívané aplikace. Pokud na aplikaci nepřicházejí žádné požadavky od uživatelů, tak se po určité době neaktivity aplikace odstaví a nahradí ji jiná. Můžeme si za poplatek zařídit, že naše aplikace bude vždy k dispozici, protože každé nahrání stojí čas. Můžeme tedy využívat většinu frameworků, ale kvůli tomuto nahrávání se každý kód navíc negativně projeví na době prvního přístupu k aplikaci, což je velmi znatelné především u rozsáhlých frameworků.

2.6 API služby

Abychom mohli propojit naši aplikaci s ostatními máme na App Engine velké množství API sloužících k různým účelům. Pojdme si nyní projít jaké možnosti máme. Některé z nich asi vůbec nevyužijeme, ale některé jsou velmi užitečné a důležité. Google pro vývoj aplikací nabízí pro všechny podporované jazyky SDK³, aktuální je nyní verze 1.5.0.1 z 16. května 2011.

2.6.1 Memcache

Asi jednou z nejpoužívanějších služeb, pokud pomineme Datastore, je Memcache. Jedná se o možnost, jak zrychlit častý přístup do databáze. Jedná se o key-value cache, která je přibližně desetkrát až stokrát rychlejší, než přístup k Datastore. Nehodí se ale pro ukládání všech dat, protože položky jsou zde uloženy jen dočasně a po určité době zmizí. Místo klasického cachování na disk, tedy máme možnost ukládat data do paměti. Implementace je podle standardu JSR-107, takže bude kompatibilní s dalšími knihovnami.

²Přidání jazyka Go bylo oznámeno na konferenci Google I/O 10. května 2011

³Software Development Kit - sada vývojářských nástrojů určených pro speciální aplikaci nebo platformu

2.6.2 Mail

Mezi další užitečné služby patří Mail, posílání mailů funguje klasicky pomocí tříd `javax.mail`. Můžeme přidávat i přílohy. Některé soubory jsou z bezpečnostních důvodů zakázány, ale všechny běžně používané jsou povoleny. Přijímání e-mailů se ošetřuje pomocí servletu⁴. Ve `web.xml` se nastaví servlet pro url `/_ah/mail/jmeno-emailu` a e-mail má tvar: `jmeno-emailu@id-aplikace.appspotmail.com` a to bohužel i v případě, že máme nastavenou naši vlastní doménu. Příchozí e-mail se chová jako HTTP požadavek, takže v servletu se musíme zpracování postarat sami, podle toho co potřebujeme.

2.6.3 XMPP

Podobně jako Mail funguje i služba pro práci s XMPP protokolem⁵. Jedná se o otevřený standardizovaný protokol Jabberu postavený na XML. Princip na App Engine je podobný jako s e-mailem, identifikátor příjemce je JID, který se dá získat z e-mailu. Podporovány jsou i další funkce, jako posílání pozvánek, nastavování statusů a další. Přijímáme pomocí servletu nastaveného na adresu: `/_ah/xmpp/message/chat/`.

2.6.4 Task Queues

Kvůli absenci JMS⁶ máme na App Engine možnost použít Task Queues. Jedná se o frontu úloh, které by mohly zpomalit náš systém, takže je výhodnější je zpracovat později. Fronta funguje následovně: pomocí Task Queues API přidáme do fronty URL naší aplikace, můžeme jí předávat parametry stejně jako u klasické URL⁷. Provedení úlohy z fronty je záležitostí servletu, na který je URL nastavena pomocí `web.xml`. Vykonání úlohy je omezeno deseti minutami, toto by měl být dostatečný limit pro běžné úlohy. Pokud servlet vrátí HTTP status mimo rozmezí 200 - 299, což znamená chybu, tak se úloha zavolá znovu, aby proběhla v pořádku. Pokud potřebujeme informovat aplikaci o dokončení úlohy, musíme to řešit pomocí Datastore.

2.6.5 URL Fetch

Pokud potřebujeme naše stránky integrovat s nějakou webovou službou, anebo používat veřejná REST API, použijeme URL Fetch. Jedná se o klasické `java.net` API, můžeme použít HTTP i HTTPS, většinu běžných portů a samozřejmě i všechny HTTP metody: GET, POST, PUT, HEAD i DELETE pro správné fungování REST rozhraní. K požadavku můžeme nastavit i vlastní HTTP hlavičky.

2.6.6 Blobstore

Na některá data, jako například obrázky, nebo velké soubory se Datastore nehodí, maximální velikost entity je 1MB. Právě kvůli tomuto účelu můžeme na App Engine použít Blobstore,

⁴Servlet je komponenta napsaná v jazyce Java, určená pro spouštění na webovém serveru

⁵Extensible Messaging and Presence Protocol (<http://xmpp.org/about-xmpp/technology-overview/>)

⁶Java Message Service (http://en.wikipedia.org/wiki/Java_Message_Service)

⁷Uniform Resource Locator (http://en.wikipedia.org/wiki/Uniform_Resource_Locator)

jedná se o uložení pro velké soubory do velikosti až 2GB. Blobstore je plně oddělen od Datastore. Nahrávání je velice jednoduché, stačí použít formulář s prvkem `<input type="file" />` a atribut action formuláře nastavíme pomocí `<%= blobstoreService.createUploadUrl("/upload")%>`. Blobstore už se sám postará, aby se soubor nahrál na správné místo. Zobrazovat soubory můžeme pomocí `blobstoreService.serve(blob-key)`, potřebujeme k tomu klíč souboru. Tato služba umí vybrat všechny uložené soubory. Práce je velmi podobná jako s Datastore, jen s tím rozdílem, že zde pracujeme s velkými soubory. Jedná se o užitečnou službu, protože dříve než existovala tato služba, se ukládání řešilo rozdělením do mnoha samostatných entit o velikosti 1MB a při zobrazení jsme je museli zase nazpět složit. Toto všechno se dělo na aplikační úrovni, takže jsme museli ošetřit všechny chybové případy a bylo vše velmi zdlouhavé a nepohodlné. Takto se o ukládání souborů stará AppEngine sám a nám stačí jednoduché API.

2.6.7 Images

S předchozím Blobstore souvisí i další služba: Images. Jedná se o možnost úpravy obrázků přímo na serveru. Obrázky se načítají z Blobstore anebo můžeme službě předat přímo pole `byte[]`. Můžeme takto aplikovat jednoduché transformace jako je změna velikosti, otočení, oříznutí, skládní obrázků a také magická funkce "I'm feeling lucky", která změní nastavení tmavých a světlých barev a k tomu také zvýší kontrast obrázku, výsledkem jsou pak více barevnější obrázky. Upravené obrázky můžeme přímo posílat uživatelům anebo uložit do Blobstore, pokud se budou zobrazovat častěji. Služba obsahuje základní transformace, ale na vytvoření náhledů nebo menší úpravy jako zvětšení a zmenšení obrázku, které jsou pravěpodobně na webových stránkách nejpoužívanější, se Image API hodí výborně.

2.6.8 Users

Pokud potřebujeme u naší aplikace vytvořit sekci jen pro přihlášené uživatele, nabízí nám k tomu App Engine možnost použít Users API a interní přihlašovací mechanismus Googlu využívaný u všech aplikací této společnosti, například tedy `gmail.com`, `youtube.com` a dalších. Použití je jednoduché, pokud uživatel není přihlášený, tak ho přesměrujeme na vygenerovanou přihlašovací stránku. Ta je stejná pro všechny služby Googlu, zadáme e-mail a heslo. Poté můžeme nastavit, které všechny údaje o sobě chceme aplikaci, do které se právě přihlašujeme, poskytnout. Nakonec nás stránka přesměruje zpět na naši aplikaci. Nyní můžeme o uživateli zjistit základní informace: přihlašovací e-mail a jednoznačný identifikátor ID. Odhlásování funguje stejně jako přihlašování, přesměrujeme uživatele na odhlásovací stránku Google, která nás následně po odhlášení znovu přesměruje, tentokrát na naši aplikaci.

2.6.9 OAuth

Pokud chceme dát možnost přihlašování i pro uživatele, kteří nevlastní účet u Google, můžeme použít OAuth protokol. Ten není vázaný na konkrétní společnost, takže si můžeme vybrat poskytovatele. Jedná se o možnost přihlášení uživatelským jménem a heslem jiné aplikace a v naší aplikaci jen kontrolujeme token. Výhoda tohoto způsobu je, že uživateli stačí

jeden účet pro více aplikací, nemusí si tak pamatovat hesla pro každou stránku na které má účet. S touto službou se pracuje velmi podobně jako s předchozím API.

2.6.10 Capabilities

App Engine obsahuje Capabilities API pomocí něhož můžeme zjistit, zda daná služba běží anebo ne. Můžeme tak ošetřit případ, kdy zrovna probíhá údržba anebo výpadek a služby jsou nedostupné. Jsou zde dostupné informace o těchto službách: Blobstore, čtení z Datastore, zápis do Datastore, Images, Mail, MemCache, TaskQueue, Url Fetch a XMPP.

2.6.11 Channel

Pro lepší spolupráci s klientskou stranou máme k dispozici Channel API. To se stará o trvalé spojení JavaScriptu na stránce se serverem Googlu, aniž by se musel klient stále dotazovat serveru. Toto se hodí, pokud chceme uživatele informovat o nastalé akci, toto se hodí například na hry pro více hráčů a internetové chaty.

2.6.12 Multitenancy

Posledním rozšířením je Multitenancy API, to nám dává možnost používat jmenné prostory pro naše data, můžeme je aplikovat na: Datastore, Memcache, Task Queue a Blobstore. Můžeme tak provozovat více oddělených stránek z jedné aplikace a pro každou stránku budeme mít speciální jmenný prostor. Data se tak nebudou překrývat a budou správně oddělena.

2.7 Omezení cloudu

Pokud se rozhodneme naši službu provozovat na cloudu, tak musíme již od návrhu počítat s jistými omezeními a odlišnou strukturou aplikace, než na jakou jsme zvyklí z klasických aplikací. Většina z těchto omezení plyne z požadavku na škálovatelnost aplikací.

Pro většinu programátorů je největším problémem databáze, používá se totiž poměrně nový typ - NoSQL databáze (pro češtinu se nejlépe hodí překlad: nerelační databáze). Databáze používaná na App Engine se nazývá BigTable⁸, jedná se o vícerozměrnou distribuovanou mapu optimalizovanou pro rychlé čtení a pomalejší zápis, protože u běžných aplikací je čtení dat mnohem častější operace. Google navíc toto uložisko využívá i pro své ostatní služby. Naprostá většina dnešních aplikací využívá relační databázi, pravděpodobně jednu z nejpoužívanějších: Oracle, PostgreSQL, MySQL anebo MS-SQL, všechny tyto databáze mají tabulky a pomocí konstrukce JOIN je můžeme navzájem libovolně spojovat. Nevýhoda tohoto řešení ale spočívá v tom, že pro tuto operaci potřebujeme všechny tabulky, které v dotazu spojujeme. V praxi se tedy používá samostatný stroj jen pro databázi. U škálovatelných aplikací, se ale nemůžeme spolehnout na to, že jsou všechny tabulky na jednom místě, mohou totiž být v různých datacentrech na různých kontinentech. Řešením je tedy ukládání všech potřebných dat do jedné tabulky anebo přiřazování tabulek do skupin, které se budou

⁸<http://en.wikipedia.org/wiki/BigTable>

spojovat a databáze se sama postará o to, aby byla data uložena ve stejném datacentru. App Engine proto používá speciální dotazovací jazyk šitý na míru tomuto uložišti: GQL - Google Query Language⁹, což je podmnožina SQL jazyka pro App Engine Datastore. Nenajdeme v něm samozřejmě operátor JOIN a s ním spojené konstrukce.

Mezi další omezení patří žádná anebo omezená možnost vyhledávání nad sloupci databáze. Není totiž zaručeno jakou strukturu sloupců bude tabulka mít. Toto lze obejít vytvořením speciální tabulky obsahující slova a v kterých sloupcích se vyskytují, ale je to dosti složité a musíme se o vše starat sami. Pokud potřebujeme vyhledávat na internetové stránce, je jednodušší použít internetový vyhledávač například Google, Bing anebo český Seznam. Všechny jmenované mají nástroj pro vyhledávání podle domény, takže stačí jen přidat formulář na naše stránky. Pokud potřebujeme vyhledávat v našich interních datech budeme muset použít rozsáhlejší řešení.

2.8 Vývoj pro App Engine

Pokud se rozhodneme vytvářet naše aplikace pro App Engine, tak máme k dispozici poměrně vyspělou infrastrukturu. Google oficiálně podporuje Eclipse plugin pro App Engine¹⁰, ale dostupné jsou i plně funkční pluginy pro NetBeans IDE¹¹ a také pro vývojové prostředí IntelliJ IDEA¹². Pomocí těchto pluginů můžeme jednoduše vyvíjet aplikaci na našem domácím stroji bez nutnosti připojení k internetu. Součástí je totiž jednoduchý webový server simulující App Engine, jedná se o upravený Jetty server¹³. Máme zde úplně stejné API jako na produkčním serveru a pomocí URL `http://localhost/_ah/admin` máme k dispozici jednoduchou administraci obsahující správce Datastore, správce Task Queues a další. Data se lokálně ukládají do souboru `.bin` přímo ve složce `/build` projektu. Deploy na lokální prostředí je stejný jako u jakéhokoliv jiného serveru, tedy pomocí tlačítka v IDE, anebo můžeme použít některý z buildovacích nástrojů, jako například ANT anebo Maven. Pro upload přímo na produkční prostředí je možné také použít přímo plugin, stejně tak jako je integrováno tlačítko pro lokální upload, tak je zde možnost uploadu přímo na App Engine. Vše probíhá nahráním výsledného `war` souboru aplikace na speciální URL. Pokud bychom chtěli integrovat nahrávání do jiného nástroje, máme možnost provést upload pomocí skriptu pro příkazovou řádku. Ten provádí upload pomocí `jar` souboru, takže není problém celý deployment integrovat do naší infrastruktury. Dále můžeme mít libovolný počet verzí aplikace, všechny jsou na URL `verze.jmeno-aplikace.appspot.com`, kde verze je jakýkoliv řetězec definovaný ve `web.xml` a výchozí možnost se nastavuje v administraci přímo na App Enginu. Máme zde navíc oproti localhostu¹⁴ mnoho různých nastavení a statistik. Pro každou aplikaci, kterých můžeme k jednomu Google účtu mít až deset je zde podrobný přehled návštěv a spotřebovaných prostředků, počet aktivních instancí, logy, přehled a správa Datastore, nastavení aplikace a další.

⁹<http://code.google.com/appengine/docs/python/datastore/gqlreference.html>

¹⁰Google Plugin for Eclipse - <http://code.google.com/appengine/docs/java/tools/eclipse.html>

¹¹NetBeans support for Google App Engine - <http://kenai.com/projects/nbappengine/pages/Home>

¹²Google App Engine Integration for IntelliJ - <http://plugins.intellij.net/plugin/?id=4254>

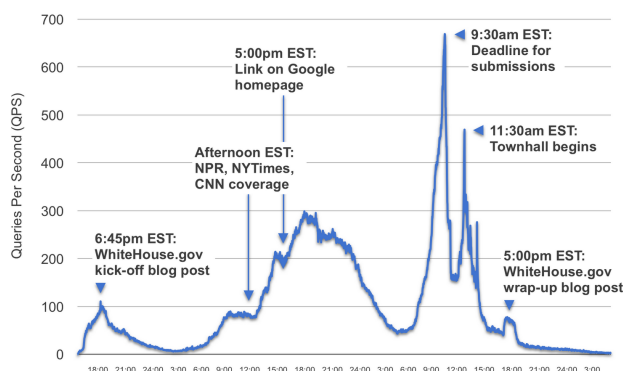
¹³Jetty je odlehčený open source HTTP server a servlet kontejner napsaný v Javě

¹⁴označení serveru, který běží na našem lokálním počítači

2.9 Zajímavé aplikace

Nyní představím zajímavé aplikace a stránky, které můžeme na App Engine cloudu najít. Nacházejí se zde zajímavé experimenty, jako například běh různých jazyků nad JVM (Java Virtual Machine) až po stránky s velkým zatížením. Nejzajímavější z nich je pravděpodobně stránka www.officialroyalwedding2011.org, založená k příležitosti svatby anglického prince Williama a Catherine Middleton. V pátek 29. dubna 2011, tedy v den oddání, bylo na hosting generováno 2 000 požadavků za vteřinu a dohromady bylo zobrazeno 15 milionů stránek od 5,6 milionu uživatelů. I přes tento nápor stránka běžela bez problémů a bez ovlivnění více jak 200 000 dalších aplikací běžících na stejném cloudu, které všechny dohromady za den vygenerovaly více jak 1,5 miliardy stránek. [?]

Další podobnou zkouškou pro App Engine byla aplikace Google Moderator. Tato aplikace běžela dva dny v březnu roku 2009 na stránce www.whitehouse.gov. Jednalo se o hlasovací systém určený pro obyvatele USA, kde může kdokoli vložit svůj dotaz a další uživatelé pak hlasují o tom, které dotazy jsou nejlepší. Vítězné otázky byly dne 29. března 2009 zodpovězeny prezidentem Barackem Obamou. Během 48 hodin zadalo 92 934 uživatelů 104 073 otázek a ohodnotilo je 3 605 984 hlasy. Při nejvyšší zátěži obsloužil App Engine 700 dotazů za vteřinu. [?]



Obrázek 2.6: Zatížení App Engine serverů v době spuštění aplikace Google Moderator pro Bílý Dům [?]

Nyní bych rád zmínil některé zajímavé a užitečné aplikace, které jsou ideální pro umístění v cloudu. Jednou z nich je i Socialwok (www.socialwok.com), jedná se o obdobu facebooku pro práci, můžeme zde sdílet naši práci v Google Apps (Docs, Calendar, Spreadsheet) se spolupracovníky a ti mohou do našich dokumentů zasahovat. Jedná se o zajímavou myšlenku a praktické využití sociálních sítí.

Podobnou službou je i Giftag (www.giftag.com), ta umí uložit část webové stránky a tu pak sdílet s dalšími. Existuje doplněk pro internetové prohlížeče, který zjednoduší uložení stránky. Všechny uložené části navíc můžeme organizovat a přidávat do seznamů. Tato služba nám může pomoci, pokud pracujeme na výzkumu anebo potřebujeme udělat prezentaci na které pracuje více lidí.

Největší uplatnění získaly cloudy díky sociálním sítím, další služba je určena právě pro ně. Jedná se o BuddyPoke! (www.buddypoke.com), umožňuje nám dát si na náš profil trojrozměrný obrázek postavičky s popisem jak se cítíme. Tuto aplikaci můžeme najít na všech používaných sociálních sítích, které vkládání obrázků dovolují, tedy: Facebook, Orkut, MySpace, hi5, Netlog, Ning a dalších. Tato aplikace nemá žádný přínos, ale díky velkému množství podporovaných sociálních sítí a jejich velké oblibě v poslední době je tato služba velice úspěšná.

Poslední zmíněná aplikace je naopak velmi přínosná. Mnoho vývojářů by rádo vidělo na App Engine podporu pro skriptovací jazyk PHP, ten se bohužel vývojáři v nejbližším čase přidat neplánují. Projekt Quercus (quercus.caucho.com) umožňuje právě běh PHP na JVM a vývojáři připravili speciální verzi pro App Engine, kde můžeme spouštět PHP s podporou některých Java frameworků.

Další zajímavé a úspěšné projekty jsou na stránce: <http://code.google.com/appengine/casestudies>

2.10 Budoucnost cloudů

Cloud je zcela nová možnost hostování webových aplikací. Jedná se o novinku, a tak bude nějaký čas trvat, než se společnosti a vývojáři přizpůsobí. Vývoj cloudových aplikací je dost odlišný a proto je potřeba přizpůsobit vývojový cyklus aplikací již od samého začátku. Je důležité poznamenat, že rozhodně ne všechny typy aplikací jsou pro cloud vhodné. Je tedy třeba rozhodnout, kdy se cloud vyplatí a kdy naopak ne. Důležité bude také sledovat, které velké společnosti budou cloud podporovat a jaké budou možnosti v nabídce cloudových hostingů. Je dobré vědět, že například Google s podporou cloudů do budoucna počítá a hodlá je dál rozšiřovat i do dalších sfér. Na každoroční konferenci *Google I/O*, která se letos konala 10. a 11. května 2011 v San Franciscu¹⁵, představil Google funkční prototyp takzvaného Chromebooku¹⁶. Jedná se o notebook s operačním systémem Chromium OS¹⁷, je optimalizován pro používání webu a rychlou práci s ním. Všechny aplikace jsou jednoduše webové stránky, odpadají tak problémy se synchronizací, novými verzemi a podobně. Nevýhoda tohoto přístupu je, že pokud se ocitneme bez internetu, tak nám nepůjde žádná aplikace. Myslím, že toto je dobrá záruka pro využití cloudových aplikací do budoucna a je třeba se přizpůsobit tomuto trendu.

¹⁵Google I/O 2011 - <http://www.google.com/events/io/2011/index-live.html>

¹⁶Chromebook - <http://www.google.com/chromebook/>

¹⁷Chromium OS - <http://www.chromium.org/chromium-os>

Kapitola 3

Implementace: Vývoj v App Engine

3.1 Výběr cloudu

Pro praktickou ukázkou a prověření implementace jsem vybral cloud App Engine od společnosti Google. Nejdůležitějším důvodem pro mne byla možnost nahrát plnohodnotnou aplikaci bez nutnosti vynaložení jakýchkoliv nákladů na hosting. Nastavené kvóty od kterých je nutné platit jsou vysoké (odpovídají několika stovkám tisíců požadavků za den) a stále se postupně zvyšují. Je tedy možné s tímto cloudem libovolně experimentovat a nahrávat různé testovací aplikace. Mnoho Java vývojářů hledá pro svoje malé aplikace hosting, kde by nemuseli platit, anebo mohli nahrát více aplikací. To je ale problém webového Java světa. Server počítá pouze s jednou aplikací a je často nutné si pamatovat prostředky a další zdroje v rámci aplikace mezi požadavky. Javovský server tak počítá s tím, že si alokuje většinu dostupné paměti systému. Výhodou je, že nemusíme při každém požadavku vytvářet nové spojení do databáze a k ostatním prostředkům. Naproti tomu u skriptovacích jazyků (jako například PHP nebo Python), se při každém požadavku provede celý kód znovu. Na serveru tak může běžet několik aplikací a zároveň se neovlivňují. V praxi se toho běžně využívá a proto je možné provozovat velké množství takovýchto aplikací na jediném hostingu. App Engine nám umožňuje stejný princip pro Javu, běh více aplikací na stejném hardware. Cena za tuto možnost je nutnost uvolnění zdrojů z nepoužívaných aplikací, aby nezabíraly místo ostatním. To App Engine sám hlídá a po určité době neaktivity aplikaci odstaví a nahraje aktivní.

3.2 Aplikace

Cílem této bakalářské práce je ověřit možnosti a omezení škálovatelných aplikací. Nejdůležitějším hlediskem je celková rychlost a odezva aplikace v závislosti na počtu souběžných požadavků. Tedy jak bude aplikace a celý cloud reagovat na zvýšený nápor požadavků, jak se s tím cloud vyrovná a zda bude služba stále použitelná. Také jsem se zaměřil na rychlost čtení a ukládání do Datastore. Pokud je klasická aplikace vystavena vysoké zátěži, je ve většině případů právě databáze úzkým hrdlem (takzvaný *bottleneck*) a přestává stíhat zpracovávat požadavky jako první.

Kromě ověření rychlosti a škálovatelnosti bylo také cílem vytvořit rozsáhlejší aplikaci a vyzkoušet všechny úskalí, které nám platforma App Engine staví do cesty. To znamená im-

plementace běžných požadavků na aplikace, například M:N¹ relace mezi entitami, zamykání dat proti přepsání a další požadavky, které jsou na aplikace běžně kladeny. Po vytvoření tohoto základu a překonání všech zádrhelů již bude jednoduché takovou aplikaci upravit pro jiné požadavky, anebo rozšířit o nové funkce.

Jako nejvhodnější řešení pro vyzkoušení implementace jsem vybral jednoduchý systém pro správu obsahu (CMS - Content Management System). Můžeme si tak vytvořit webovou stránku s plnou administrací, tedy s možností přidávání, úpravy a mazání stránek. Ke každé stránce můžeme přidat šablonu. Mimo samotných stránek lze přidávat, upravovat a mazat i novinky. Jedná se o kratší zprávy, pro které je zbytečné vytvářet samostatnou stránku. Ke každé stránce lze také přiřadit štítek (tag), tato možnost nahrazuje kategorie. Výhodou oproti kategoriím, kde může být jeden článek pouze v jedné kategorii je, že článek může být označen více štítky a zároveň jeden štítek může být přiřazen k více článkům. Z těchto údajů pak můžeme generovat takzvané *tag cloudy* (oblaky štítků)², kde velikost štítku určuje jeho význam, Čím více článků štítek označuje, tím je důležitější a výraznější. Z hlediska implementace se jedná o vazbu M:N, která se v NoSQL databázích musí implementovat složitěji než v relačních a bylo potřeba vymyslet vyhovující řešení.

3.3 Výběr frameworku: Slim3

Před započatím práce jsem zjišťoval, které knihovny a frameworky³ bych mohl použít na ulehčení práce. Kvůli omezením nejsou podporovány všechny knihovny a některé potřebují speciální úpravy anebo nastavení. Další nevýhodou je, že čím je náš kód rozsáhlejší, tím déle bude trvat načítání pokaždé, když bude App Engine nahrávat aplikaci do aktivního stavu. Tímto jsem eliminoval velké frameworky jako jsou Spring⁴ a Seam⁵. Poté jsem hledal mezi menšími, ale žádný nevyhovoval úplně potřebám App Engine. Bohužel je tento cloud relativně mladý a tak pro něj neexistují frameworky, anebo jsou málo známé. Uvažoval jsem tedy že použiji přímočaré řešení pomocí servletů a naprogramuji aplikaci od základů. Naštěstí jsem ale náhodou narazil na framework Slim3⁶ určený přesně pro potřeby App Engine.

Slim3 je MVC⁷ framework optimalizovaný pro App Engine, přináší rozšíření i jednodušší práci s Datastore API a další vylepšení. Jeho hlavní koncepty jsou: *simple* (jednoduchý) a *less is more* (méně je více), tedy že jednoduchost a srozumitelnost vedou ke správnému designu. Framework se drží Paretova pravidla, tedy že 80% aplikace, pramení z 20% práce, takže se snaží co nejvíce zjednodušit oněch 20%, a ve zbytku nechává programátorovi volnost.

Vývoj tohoto frameworku začal již v dubnu roku 2009, takže se nejedná o úplně nový projekt. Na internetu je k dispozici přehledná dokumentace a dvě diskuzní skupiny, jedna pro anglicky mluvící vývojáře⁸ a druhá pro japonské programátory⁹. To proto, že vývojáři Slim3

¹Many-to-Many - druh relace, kde více entit může být spojeno s více entitami (M:N), v relačních databázích je pro propojení použita spojovací tabulka

²http://en.wikipedia.org/wiki/Tag_cloud

³Framework je ucelený soubor knihoven a kódu pomáhající vytvořit aplikaci

⁴Spring Framework - <http://www.springframework.org/>

⁵Seam Framework - <http://seamframework.org/>

⁶Slim3 - <http://sites.google.com/site/slim3appengine/>

⁷Model View Controller - <http://en.wikipedia.org/wiki/Model-view-controller>

⁸<http://groups.google.com/group/slim3-user>

⁹<https://groups.google.com/group/slim3-user-japan?hl=ja>

pocházejí z Japonska, a dokonce o tomto frameworku vydali i knihu¹⁰, bohužel je celá také v Japonštině. Autoři i komunita jsou aktivní a reagují na všechny změny App Engine SDK¹¹ i připomínky a chyby na diskusních fórech. Díky této zpětné vazbě dokázali programátoři frameworku překonat některé počáteční problémy a zádrhele a nyní již upravují kód pro jednodušší práci s App Enginem. Mimo frameworku samotného je vyvíjen i Slim3 Eclipse plugin¹², který umí generovat některé části kódu a usnadňuje vývoj. Všechny zdrojové kódy jsou open source a jsou tedy zdarma dostupné v SVN repozitáři projektu¹³.

3.4 Práce s Datastore API s pomocí Slim3 framworku

Nejvíce nám framework pomáhá při práci s Datastore. Pokud s uložištěm chceme pracovat, máme na App Engine možnost použít klasické JPA¹⁴ s těmito omezeními: nemůžeme použít vztah `many-to-many`, v dotazu nemůžeme použít `JOIN` a agregační funkce (`GROUP BY`, `HAVING`, `SUM`, `AVG`, `MAX` a `MIN`) a polymorfické dotazy (slouží k získání podtřídy). Druhou možností je použít JDO¹⁵, jedná se o obecnější obdobu JPA, kde můžeme náš doménový model ukládat do různých uložišť - relačních, nerelačních, objektových databází, XML i do obyčejných souborů. Programátorská práce je stejná jako s JPA, definujeme model pomocí anotací¹⁶ a dotazujeme se pomocí objektu `Query`. Poslední možností je použít přímo Datastore API, které je optimalizované pro práci s BigTable¹⁷. Dovoluje nám měnit strukturu entit za běhu. Znamená to, že se nemůžeme spolehnout na to, že je daný sloupec v entitě obsažen, ani na jeho typ. Sloupce u entity je vlastně kolekce objektů, v Javě by BigTable vypadala následovně: `Map<Key, Set<Object>>`. Toto je důležité si uvědomit a může nám to přinést potíže v začátcích, dobrou radou je ukládat si přímo do objektu i verzi schématu. Při změně v aplikaci se totiž schémata uložených entit nezmění.

Další změnou jsou transakce, pro správné fungování potřebujeme, aby byl všechny typy entity používané v transakci na jednom stroji. To může být problémem, jelikož Google má několik datacenter a není tak jisté, že budou všechny entity pohromadě. Kvůli tomuto zavádí App Engine takzvané *entity groups* (skupiny entit). Entity přiřadíme do stejné skupiny tak, že nové entitě nastavíme tzv. *parent entity* (rodičovskou entitu), její klíč pak bude složen z klíče samotné entity a klíče rodičovské entity. Pokud se pokusíme provést operace v transakci na entitách z rozdílných skupin, vyhodí App Engine výjimku.

Uložiště dat na App Engine nepodporuje relace, na které jsme zvyklí z relačních databází pomocí cizích klíčů (tzv. *foreign keys*), můžeme ale tyto relace použít pomocí spojení přes klíče entit `Key`. Největším zádrhelem je, že se musíme sami postarat o referenční integritu a spojení těchto relací. To znamená, že pokud například smažeme jednu entitnu, musíme zrušit i všechny vazby, ve kterých se daná entita vyskytuje. Existují tři základní typy vztahů: *one-to-*

¹⁰<http://www.amazon.co.jp/exec/obidos/ASIN/4798026999/hatena-hamazou-22/>

¹¹Software Development Kit - sada vývojářských nástrojů určených pro speciální aplikaci nebo platformu

¹²<http://sites.google.com/site/slim3appengine/documents/eclipse-plugin>

¹³Zdrojové kódy frameworku Slim3 - <http://code.google.com/p/slim3/source/browse/>

¹⁴JPA - Java Persistence API - http://en.wikipedia.org/wiki/Java_Persistence_API

¹⁵JDO - Java Data Objects - http://en.wikipedia.org/wiki/Java_Data_Objects

¹⁶Anotace v Jazyce java představují speciální konstrukce začínající `@` a přidávající ke kódu dále zpracovatelné metainformace

¹⁷<http://en.wikipedia.org/wiki/BigTable>

one (1:1), *one-to-many* (1:M) / *many-to-one* (M:1) a *many-to-many* (M:N), dále rozlišujeme *unidirectional* (jednosměrné) a *bidirectional* (obousměrné) vazby.

Slim3 nám práci s vazbami velmi usnadňuje a stará se o závislosti. Pomocí speciálních anotací můžeme označit entity a z těchto údajů pak plugin vygeneruje javovské *meta třídy*. Výhoda tohoto přístupu je v rychlosti, údaje by se mohly získávat při každém použití pomocí reflexe¹⁸, ale ta je obecně prokázána jako o dosti pomalejší. Takto stačí vygenerovat meta třídy pouze při každé změně a o toto přegenerování se stará Slim3 Eclipse plugin. Výsledky porovnání čtení 100 000 entit z Datastore při použití Datastore API, Slim3 a JDO jsou přímo na stránkách frameworku i s funkční ukázkou (<http://slim3demo.appspot.com/performance/>), Datastore API vychází nejrychleji, asi desítky milisekund, druhý v pořadí je Slim3 přibližně 4 000 milisekund a nejhůře vyšel v testu podle očekávání JDO přístup, který trval třibližně 12 000 milisekund.

Slim3 nám dále umožňuje do entitních tříd přidat i proměnné odkazující na další entity. Používá k tomu třídu `ModelRef<Entity>` pro jednosměrný vztah a `InverseModelRef<Entity1, Entity2>` pro obousměrné vztahy na inverzní straně relace. Získat entity ze vztahu *one-to-many* pak můžeme vypadat následovně: 3.1

Listing 3.1: Získání entity

```

1  @Model
2  public class Employee {
3      @Attribute(primaryKey = true)
4      private Key key;
5
6      private ModelRef<Department> departmentRef
7          = new ModelRef<Department>(Department.class);
8
9      // ... other properties + getters and setters
10 }
11
12 Employee employee = Datastore.get(Employee.class, employeeKey);
13 Department department = employee.getDepartmentRef().getModel();

```

A u obousměrného propojení *one-to-many* relace získáme entity tímto způsobem: ??

Listing 3.2: Získání entit ze vztahu *many-to-many*

```

1  @Model
2  public class Employee {
3      @Attribute(primaryKey = true)
4      private Key key;
5
6      private ModelRef<Department> departmentRef
7          = new ModelRef<Department>(Department.class);
8
9      // ... other properties + getters and setters
10 }
11
12 @Model
13 public class Department {

```

¹⁸Způsob zjišťování metadat (například typ objektu a podobně) a modifikace programu za běhu

```

14     @Attribute(primaryKey = true)
15     private Key key;
16
17     @Attribute(persistent = false)
18     private InverseModelListRef<Employee, Department> employeeListRef =
19         new InverseModelListRef<Employee, Department>(Employee.class,
20             "departmentRef", this);
21
22     // ... other properties + getters and setters
23 }
24
25 Department department = Datastore.get(Department.class, departmentKey);
26 List<Employee> employeeList = department.getEmployeeListRef().getModelList()
    ;

```

Vazba M:N se realizuje pomocí dvou spojení *one-to-many* a spojovací tabulky (viz zdrojový kód: 3.3), kde máme dvě položky, první je klíč do jedné entity a druhou je klíč ke druhé entitě.

Listing 3.3: Ukázka many-to-many vztahu s použitím spojovací entitní třídy

```

1  @Model
2  public class EmployeeProject {
3      @Attribute(primaryKey = true)
4      private Key key;
5
6      private ModelRef<Employee> employeeRef =
7          new ModelRef<Employee>(Employee.class);
8
9      private ModelRef<Project> projectRef =
10         new ModelRef<Project>(Project.class);
11
12     // ... other properties + getters and setters
13 }
14
15 @Model
16 public class Employee {
17     @Attribute(primaryKey = true)
18     private Key key;
19
20     @Attribute(persistent = false)
21     private InverseModelListRef<EmployeeProject, Employee>
22         employeeProjectListRef =
23         new InverseModelListRef<EmployeeProject, Employee>(EmployeeProject.
24             class, "employeeRef", this);
25
26     // ... other properties + getters and setters
27 }
28
29 @Model
30 public class Project {
31     @Attribute(primaryKey = true)
32     private Key key;
33
34     @Attribute(persistent = false)

```

```

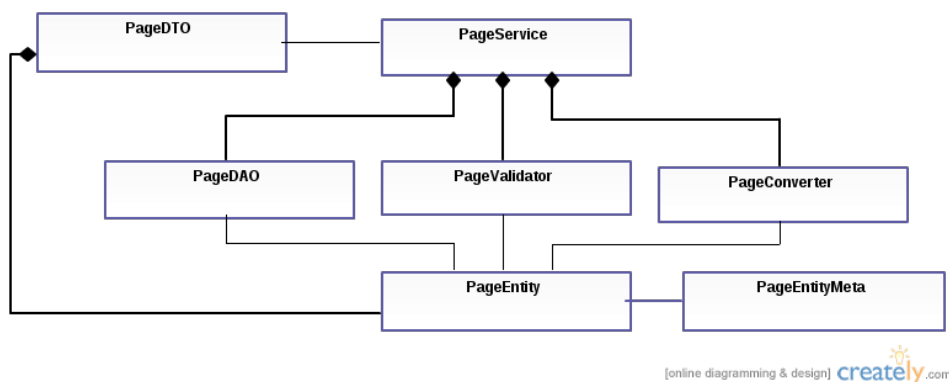
33     private InverseModelListRef<EmployeeProject, Project>
34         employeeProjectListRef =
35             new InverseModelListRef<EmployeeProject, Project>(EmployeeProject.
36                 class, "projectRef", this);
37
38     // ... other properties + getters and setters
39 }
40
41 Employee employee = Datastore.get(Employee.class, employeeKey);
42 for (EmployeeProject ep : employee.getEmployeeProjectRef().getModelList()) {
43     Project project = ep.getProjectRef().getModel();
44 }
45
46 Project project = Datastore.get(Project.class, projectKey);
47 for (EmployeeProject ep : project.getEmployeeProjectRef().getModelList()) {
48     Employee employee = ep.getEmployeeRef().getModel();
49 }

```

Takto lze realizovat všechny druhy spojení včetně obousměrných variant. O většinu rutinní práce se postará vygenerovaný metamodel entit a umožňuje programátorovi soustředit se jen na správné vzájemné nastavení spojení.

3.5 Model aplikace

Framework nám kromě obalení Datastore API z App Engine nedává moc dalších rozšíření pro část modelu návrhového vzoru MVC. Chtěl jsem mít model rozdělený do více samostatných a vyměnitelných vrstev. Model mé aplikace tedy vypadá jako na obrázku 3.1.



Obrázek 3.1: Diagram tříd modelu

3.5.1 Entity a metatřídy entit

Entity z Datastore jsou v Javě reprezentovány doménovým modelem. Každé entitě odpovídá jedna třída s klíčem `Key` a nastavenými třídními proměnnými jako parametry entity v uložišti. Tyto třídy jsou označeny anotacemi (například `@Model`, `@Attribute(primaryKey = true)`), jak bylo vidět na výpisech výše, což slouží ke generování metatříd.

3.5.2 DAO

Ke každé entitní třídě je vygenerován jeden metaobjekt. Pro CRUD (Create, Read, Update, Delete) operace s entitou je vyhrazen speciální DAO objekt¹⁹, každé entitní třídě odpovídá jeden DAO objekt. Máme zde veškerou práci s Datastore, pokud tedy budeme chtít změnit některé chování, máme tento kód na jednom místě a nemusíme procházet mnoho tříd.

3.5.3 Service

S DAO třídami pracují takzvané servisní (Service) třídy. Znovu platí pravidlo, že servisní třídy odpovídají těm entitním. Jedná se vlastně o naše rozhraní do aplikace, protože MVC návrhový vzor počítá s nahraditelností všech částí, to znamená, že pokud bychom chtěli udělat z této aplikace místo webové desktopovou, stačilo by změnit Controller pro řízení vstupu a View pro zobrazování dat. Model by nepotřeboval žádnou změnu. Chybové stavy jsou do View předávány přes `ServiceException`, která má navíc dvě specializované podtřídy `ConverterException` a `ValidatorException`, aby šlo jednoduše odlišit odkud chyba pochází.

3.5.4 Converter

Servisní třída tedy používá třídy `Converter` a `Validator` odpovídající entitám. Converter jak již název napovídá slouží ke konverzi vstupních parametrů od uživatele. Vše totiž do Controlleru aplikace přichází jako řetězec (`String`) a tak je potřeba všechny položky převést na odpovídající typy a některé i podle speciálního nastavení, například datum podle určeného formátu na Javovský typ `Date`. Converter se tedy postará o správné vytvoření entitní třídy podle vstupních pametetrů anebo vyhodí výjimku s popisem chyby.

3.5.5 Validator

Validační třída se pak postará o kontrolu entity, například zda textové pole nepřesáhlo maximální délku, anebo zda jsou vyplněna všechna pole, která nesmí být prázdná. S tímto nám také Slim3 pomáhá, máme zde již připravené některé běžné validátory (pro maximální délku řetězce, rozsah číselných hodnot anebo například kontrolu pomocí regulárních výrazů²⁰). Validátory navíc používají vygenerované metatřídy, abychom mohli specifikovat položku, kterou chceme zkontrolovat. Ve validátoru se rozlišují dva stavy, zda je vytvářen nový objekt, nebo zda jde jen o úpravu existující entity. To se používá například pokud chceme mít unikátní název URL pro stránku, aby systém věděl, kterou stránku má správně zobrazit. Při vytváření zkontrolujeme, zda již stejná URL neexistuje a při úpravě kontrolujeme, zda byla URL změněna, a poté zda nekoliduje s některou z existujících URL.

3.5.6 DTO

Poslední důležitou vrstvou jsou DTO²¹, ty slouží pro přenos entitních tříd mezi vrstvami Model a View. Jedná se o opak Converterů, pokud chceme zobrazit entitu uživateli, tak ji

¹⁹Data Access Objekt - objekt zapouzdřující práci entitních tříd s databází

²⁰Regulární výrazy (Regular Expression) - jedná se o speciální řetězec popisující množinu řetězců sloužící k vyhledávání nebo manipulaci s textem

²¹Data Transfer Object - objekty sloužící pro transport entitních a dalších tříd

obalíme DTO objektem a ten se postará o správné převedení typů. Například datový typ `Date` se podle nastavení převede do správného formátu data pro uživatele.

3.6 Propojení modelu - Google Guice

Ke spojení všech těchto vrstev jsem se rozhodnul pro použití Google Guice²². Jedná se o knihovnu pro Dependency Injection (injekce závislostí), kde můžeme pomocí anotace `@Inject` nastavit přiřazení konkrétní implementace do třídních proměnných pomocí typu proměnné.

Listing 3.4: Ukázka použití Google Guice za pomoci anotace `@Inject`

```
1 public class PageService {
2     @Inject
3     private PageDAO pageDAO;
4
5     // ...
6 }
```

Výhodou nastavení rozhraní oproti přesnému určení typu třídy, je možnost měnit injektované třídy podle potřeby, například v jednom případě pro produkční prostředí a jinou implementaci pro testování, kde nastavíme například jen mock objekty²³. Pokud existuje v projektu jediná implemetace rozhraní, Guice ji najde a pomocí reflexe ji do proměnné nastaví. Pokud aplikace obsahuje více implementací daného rozhraní, musíme použít konfiguraci, kde nastavíme která implementace se má do proměnné nastavit. Konfigurace vypadá následovně:

Listing 3.5: Příklad konfigurace Google Guice

```
1 public class GuiceModule extends AbstractModule {
2     public void configure() {
3         bind(PageDAO.class).to(PageDAOImpl.class);
4     }
5 }
```

Vše se nastavuje v jazyce Java, jen vybereme s jakou třídou má být rozhraní spojeno a takto vzniklou třídu předáme jako parametr při vytváření metodě `createInjector`. Vytváření instancí objektu s nastavenými závislostmi pomocí Guice se provádí následujícím způsobem

Listing 3.6: Získávání sestavených tříd pomocí Google Guice

```
1 Injector injector = Guice.createInjector(new BillingModule());
2 PageService pageService = injector.getInstance(PageService.class);
```

Výsledkem je instance třídy `PageService` se správně nastavenými všemi závislostmi podle konfigurace. Pokud některý z nastavovaných objektů také obsahuje závislosti označené pomocí `@Inject` nastaví se také.

²² <http://code.google.com/p/google-guice/>

²³ Mock objekty jsou simulované objekty, které napodobují reálné objekty používané hlavně při testování

Výhoda Google Guice je, že jeho kód je poměrně malý v porovnání s ostatními knihovnami, které obsahují stejnou funkčnost. Guice se stará čistě jen o nastavení závislostí, kdežto například Spring framework a podobné se starají i o řešení dalších záležitostí. Proto je Guice vhodné použít na App Engine, nebude tak zdržovat načítání naší aplikace a závislost komponent v naší aplikaci bude nastavena na jednom místě.

3.7 Slim3 Controller

Slim3 nám kromě ulehčení práce s Datastore pomáhá i v Controller vrstvě návrhového vzoru MVC. Tato část se stará o řízení požadavků od uživatele, to znamená, že pokud navštíví určitou URL adresu, tak framework přesměruje požadavek do správného controlleru. Překlad funguje následujícím způsobem: pokud navštívíme základní URL /, použije se třída **IndexController** ze základního javovského balíčku (package) aplikace. Pokud navštívíme konkrétní stránku, například /page, použije se třída **PageController** ze základního balíčku. Dále pokud navštívíme podstránku /page/subpage použije se třída **SubpageController** (tedy první písmeno je velké, jak je u tříd v Javě zvykem, a přidá se **Controller**) z balíku **page** a tak dále, můžeme do sebe vnořovat libovolný počet balíků a vždy když bude URL končit / použije se **IndexController** aktuálního balíku.

Dalším vylepšením je možnost konvertovat parametry URL na GET²⁴ parametry zpracovatelné pomocí `request.getParameter`. Vše se konfiguruje ve třídě **AppRouter** následujícím způsobem:

Listing 3.7: Konfigurace nastaveníURL

```
1 public class AppRouter extends RouterImpl {
2     public AppRouter() {
3         addRouting("/edit/{key}/{version}",
4                 "/edit?key={key}&version={version}");
5     }
6 }
```

URL /edit/xxx/1 se převede na /edit?key=xxx&version=1, použije se tedy **EditController** a v parametrech objektu **Request** bude: **key=xxx** a **version=1**. **EditController** je pak obyčejná třída dědicí od třídy **Controller**, kde přepíšeme metodu **runBare()** s naší požadovanou funkčností. Máme zde dostupné všechny objekty jako máme k dispozici v servletu²⁵, jsou zde tedy dostupné objekty: **Request**, **Response** i **ServletContext**.

U této vrstvy bylo důležité zajistit, aby webové uživatelské rozhraní správně zpracovávalo vstup od uživatele a také, aby aplikace správně informovala uživatele o nastalých akcích, jako jsou vložení stránky nebo chyba při validaci vstupu. Důležitým požadavkem bylo, aby byl uživatel informován i při přesměrování na jinou stránku (například po úspěšné akci) a aby byla zpráva zobrazena pouze jednou. Toto jsem vyřešil předáváním zpráv pomocí **Session**²⁶, pokud se má zobrazit po přesměrování. Aby se zpráva zobrazila pouze jednou, používá se

²⁴GET je parametr HTTP protokolu, kde se proměnné přenášejí v URL, například: `url?klic=hodnota&klic2=hodnota2`

²⁵Servlet je komponenta napsaná v jazyce Java, určená pro spouštění na webovém serveru

²⁶Session umožňují identifikovat a rozeznat jednotlivé klienty při přechodu a přesměrování webových stránek

trik, kdy se všechny zprávy předají do parametru objektu **Request**, kam se ukládají informační zprávy, pokud nepotřebujeme uživatele přesměrovat. Vše je implementováno pomocí **FlashScopeFiltru**²⁷, který je zařazen před **FrontController**, kde se přesun zpráv do objektu **Request** provede.

Pomocí připravených controllerů je vytvořena administrační část, kde jsou všechny URL adresy pevně dané a nebudou se měnit. Pro veřejnou část stránek, které jsou dynamicky generované uživateli aplikace, se používá třída **IndexController**. Ta podle URL adresy vybere stránku z Datastore a získá všechny související entity, jako šablonu, štítky a další. Ze všech těchto dat nakonec naplní šablonu, sestaví stránku a tu odešle klientovi. Pokud se nepodaří stránku najít, tak přesměruje řízení na **NotFoundController**, ten zobrazí zprávu o tom, že stránka nebyla nalezena a odešle chybový HTTP kód 404²⁸.

3.8 View vrstva

Poslední vrstva je View (neboli prezentační), která slouží k zobrazování webového obsahu uživatelům. Zde Slim3 používá klasické JSP²⁹, kde nám přidává několik značek (*tagů*) pro usnadnění vykreslování formulářů a údajů z Datastore. JSP bylo vybráno pravděpodobně z toho důvodu, že je velmi rozšířené a je standardní součástí App Engine. JSP je vcelku jednoduché. Do HTML stránky přidáváme speciální tagy, které se zpracují. Můžeme také vytvářet vlastní knihovny tagů, které pak v JSP můžeme použít.

3.9 Shrnutí implementace

Struktura celé aplikace využívá návrhový vzor Model-View-Controller, kde jsou všechny části odděleny a mohou být jednoduše nahrazeny jinou implementací. Největší oporou při implementaci celé aplikace byl framework Slim3, který je vyvíjený přímo pro nasazení na cloudu App Engine. Tento framework nejvíce usnadňuje práci s modelem, umožňuje pracovat s relacemi mezi entitami. Framework nám usnadňuje i práci s controllery. Stará se o přesměrování požadavku na správný controller a řídí předávání informací od klienta. V prezentační vrstvě je využito technologie JSP a používají se speciální tagy pro zjednodušení zobrazení formulářů. K propojení vrstev v modelu je použita knihovna Google Guice sloužící k injektování závislostí pomocí speciálních anotací. Pro celkový návrh aplikace byla klíčová oddělenost, znovupoužitelnost a nahraditelnost všech částí. Aplikace je se pak stává libovolně rozšiřitelnou, lehce upravovatelnou a také jednoduše testovatelnou.

²⁷Filtry jsou standardní součástí webové Javy, kdy můžeme transformovat jednotlivé příchozí požadavky a odchozí odpovědi

²⁸HTTP stavový kód 404 podle dohody oznamuje klientovi, že stránka kterou požaduje, nebyla na serveru nalezena, to může být užitečné například pro vyhledávače, aby stránku nezařazovaly do svých výsledků

²⁹Java Server Pages - technologie umožňující generování webových stránek
http://en.wikipedia.org/wiki/JavaServer_Pages

Kapitola 4

Testování: Porovnání rychlosti práce s Datastore a testování zátěže

4.1 Hlediska a způsob testování

Pro otestování cloudu jsem vybral dvě důležitá hlediska. Prvním byla rychlost Datastore API a provádění operací: čtení, vkládání, úprava a mazání záznamů v závislosti na použitém přístupu. Testovány byly tyto možnosti: JPA, JDO a Datastore API. Druhým typem testů bylo zatížení celé aplikace a reakce cloudu. Vytvořil jsem velké množství stránek a poté jsem pomocí nástroje Apache Bench¹ posílal na App Engine požadavky. Toto testování by mělo reflektovat situaci, kdy se o dané stránce dozví v krátkém čase velký počet návštěvníků, například vyjde článek nebo reportáž, spuštění velké kampaně, anebo například použití při propagaci jednorázové události jako jsou volby.

Všechny testy probíhaly přímo na App Engine. Sice existuje server pro lokální vývoj, ale není zcela stejný jako cloud, jedná se jen o webový server Jetty obohacený o API služeb cloudu. Důležitým důvodem také je, že aplikace běží spolu s dalšími stránkami a navzájem se tedy ovlivňují. Cílem testu bylo simulovat co nejvíce reálné prostředí, abychom mohli odhadnout chování aplikace.

Pokud je aplikace určitý čas neaktivní, tak se odnahráje (angl. undeploy) aby systém šetřil prostředky pro jiné aplikace. Toto nahrávání zabírá nezanedbatelný čas a mohlo by ovlivnit výsledky testů. Před každým testem byla tedy aplikace navštívena, aby bylo zajištěno, že bude připravena a nebude potřeba ji znovu nahrávat.

4.2 Testování Datastore

Pro testování samotného uložště jsem napsal jednoduchou aplikaci s názvem *ToDoList*. Tato aplikace slouží k jednoduché evidenci úkolů, ty můžeme přidávat, upravovat a mazat. Model aplikace obsahuje jednu entitu `ToDoEntity` a Datastore bylo naplněné několika tisíci záznamy. Pro každou z testovacích možností, tedy JPA, JDO a Datastore API, jsem vytvořil entitu

¹Apache Bench je nástroj pro příkazovou řádku určený k měření výkonu serverů - <http://httpd.apache.org/docs/2.2/programs/ab.html>

s odpovídajícími anotacemi a DAO třídu `TodoDAO` pro práci s Datastore. Zbytek aplikace jsem ponechal vždy stejný a při testech jsem měnil jen tuto vrstvu. Při testování byl měřen jen čistý čas operace, aby nebyly výsledky zkresleny síťovým zpožděním a dalšími faktory. Zdrojové kódy všech aplikací i s ukázkami jsou volně k dispozici²

Datastore API je optimalizováno přesně pro uložiště BigTable, používané na App Engine. Jedná se o nejpřímochařejší způsob a tak bylo očekáváno, že i tento způsob práce vyjde z testů nelépe. JPA a JDO používají anotace pro zjednodušení celé práce, ale vše je vykoupeno nutností tato data zpracovávat a převádět. To samozřejmě celou práci zpomaluje a dá se očekávat, že tyto přístupy vyjdou z testu rychlosti práce podobně, ale pomaleji než Datastore API.

První kolo testování probíhalo v odpoledních hodinách a kvůli vlivu ostatních aplikací byly některé výsledky výrazně odlišné, což celé porovnání zkreslovalo. Proto byly testy provedeny znovu kolem půlnoci našeho času, kde k takto výrazným odchylkám nedocházelo. Google má několik datacenter a vybírá, na které a na kolik z nich bude aplikace nahrána. Před dvěma lety měl App Engine datacentra dvě, jedno na východním a druhé na západním pobřeží USA, nyní je velice pravděpodobné, že má datacenter více, ale přesný počet a polohu Google nezveřejnil. Vliv ostatních aplikací byl u některých testů znatelný i v noci. App Engine poskytuje statisky využití svých služeb s grafy a dostupností přehledně zobrazené po dnech, vše je na adrese: <http://code.google.com/status/appengine/>.

4.3 Test výběru

Prvním testem bylo vybrání 1000 entit z Datastore pomocí všech způsobů. Výsledky jsou vidět v tabulce 4.1 a v grafu 4.1 Pro vybírání záznamů z uložiště je jasným favortiem Datastore API, toto rozhraní je pro výběr optimalizováno, jelikož výběr bývá u většiny aplikací několikanásobně častější než ostatní operace. Na druhém místě se umístil přístup pomocí JPA a o něco pomalejší je JDO.

Tabulka 4.1: Tabulka porovnání výběru 1 000 entit (údaje jsou uvedeny v milisekundách)

číslo testu	1	2	3	4	5	6	7	8	9	10	průměr
Datastore API	1591	1649	1736	1708	2292	1752	1768	1645	1738	1645	1752.4
JPA	2677	2438	2518	2345	2321	2337	2253	2304	2203	2346	2374.2
JDO	2447	2345	2341	2619	2799	2912	2581	2632	2411	2192	2527.9

4.4 Test vkládání

Pro vkládání záznamů již nejsou výsledky tolik rozdílné. Vítězem je znovu Datastore API, ale hned v těsném závěsu se umístil přístup pomocí JDO a hned za ním JPA.

²<http://code.google.com/p/ae-cms/wiki/Aplikace>



Obrázek 4.1: Graf porovnání výběru 1 000 entit

Tabulka 4.2: Tabulka porovnání vkládání 100 entit (údaje jsou uvedeny v milisekundách)

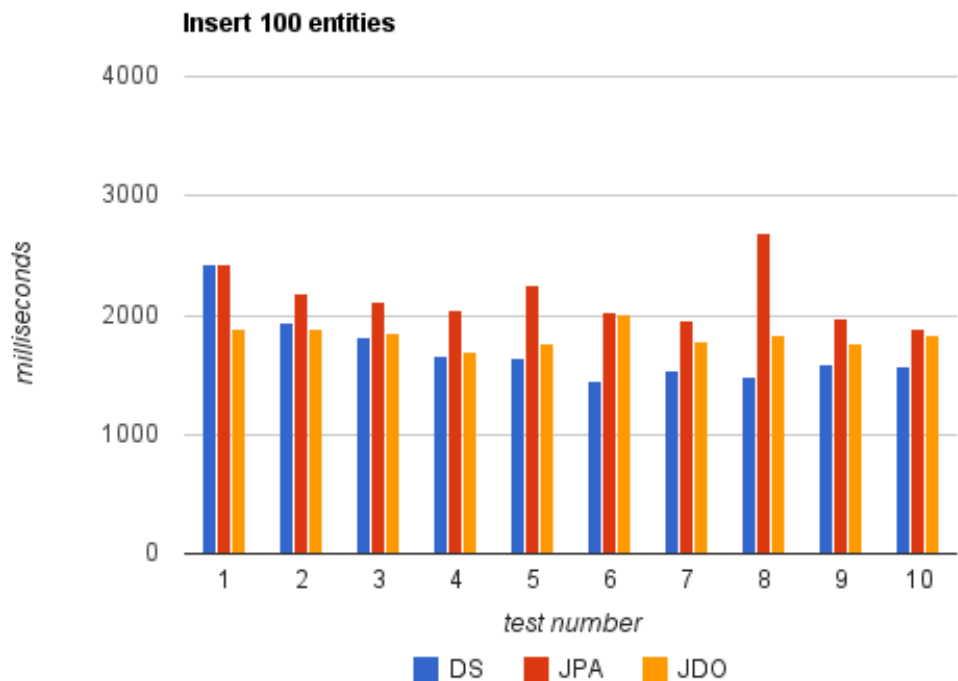
číslo testu	1	2	3	4	5	6	7	8	9	10	průměr
Datastore API	2421	1931	1810	1663	1637	1445	1543	1488	1585	1578	1710.1
JPA	2429	2178	2120	2044	2255	2023	1961	2698	1973	1878	2155.9
JDO	1891	1881	1851	1696	1764	2012	1781	1834	1765	1826	1830.1

4.5 Test úprava

Při úpravě entity překvapivě JPA i JDO pracovaly rychleji než Datastore API. Nejrychlejší byl nyní JDO přístup. Pomalost Datastore API byla pravděpodobně způsobena implementací DAO vrstvy. Nejdříve byla entita vybrána z databáze, poté byly změněny hodnoty a následně celá entita znovu uložena.

4.6 Test mazání

Pro mazání bylo Datastore API výrazně rychlejší. JPA a JDO na tom byly s rychlostí podobně, ale JPA vyšlo o něco lépe.



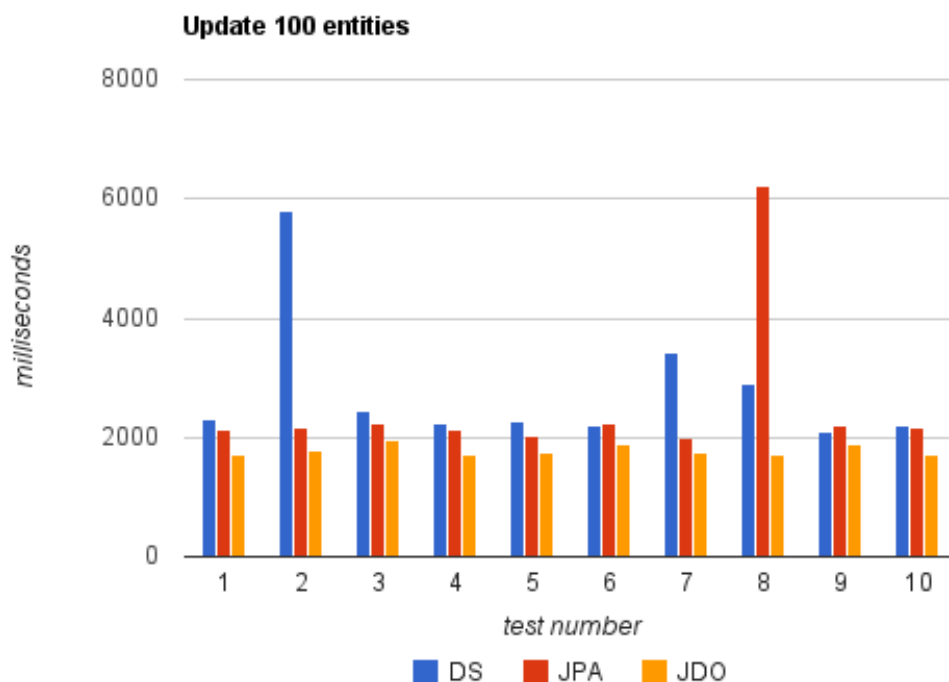
Obrázek 4.2: Graf porovnání vkládání 100 entit

Tabulka 4.3: Tabulka porovnání úpravy 100 entit (údaje jsou uvedeny v milisekundách)

číslo testu	1	2	3	4	5	6	7	8	9	10	průměr
Datastore API	2309	5809	2442	2245	2283	2199	3407	2908	2085	2217	2501.2
JPA	2139	2171	2219	2135	2026	2234	1997	6232	2191	2181	2162
JDO	1704	1775	1950	1703	1735	1870	1752	1711	1889	1719	1769.4

4.7 Porovnání výsledků testů práce s Datastore

Celkově tedy v testech dopadlo dle očekávání nejrychleji Datastore API. Jedná se o přímočarý přístup k uložení a nestará se o další transformace. Bohužel tato rychlost je vykoupena složitější implementací a odlišným způsobem práce. Při úpravě dat bylo podle výsledků testů Datastore API nejpomalejší, to bylo způsobeno odlišným způsobem práce, kdy jsme objekt vybírali z databáze a poté upravovali. Je to dáno rozdílností práce s Datastore API oproti JPA a JDO, pravděpodobně by se nám pomocí optimalizací podařilo dostat výsledky na stejnou úroveň jako ostatní dvě řešení. Každopádně i přesto je Datastore API jasným vítězem. JPA a JDO jsou na tom podle provedených testů výkonnostně podobně. JPA vyniká v rychlosti při čtení a mazání, kdežto JDO je rychlejší při vkládání a úpravě. Práce s těmito dvěma přístupy je podobná a záleží tedy hlavně na zvyku a zkušenosti programátora, kterou si vybere. Při velkém počtu čtení z Datastore oproti ostatním operacím je z těchto dvou



Obrázek 4.3: Graf porovnání úpravy 100 entit

Tabulka 4.4: Tabulka porovnání mazání 100 entit (údaje jsou v uvedeny v milisekundách)

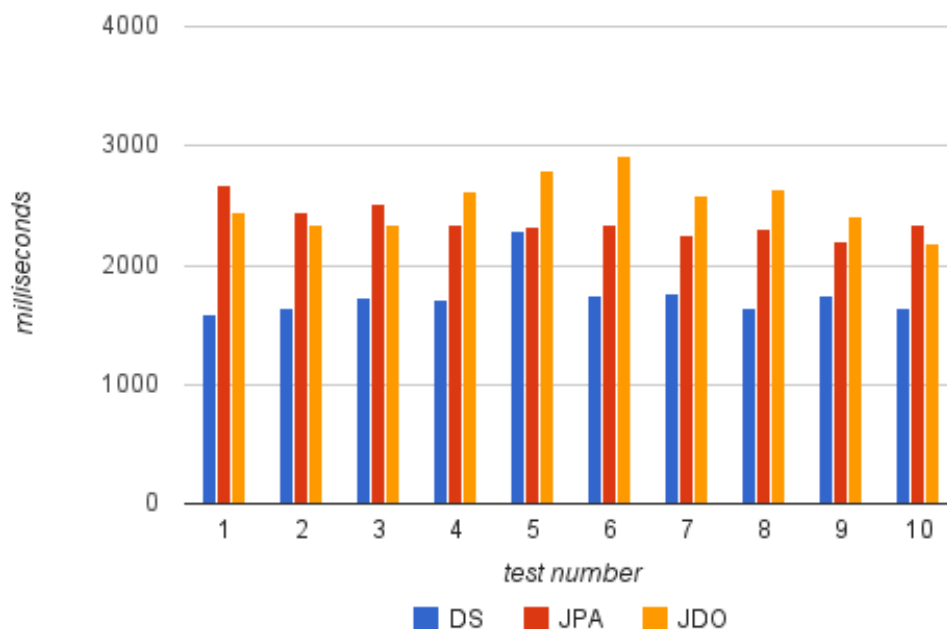
číslo testu	1	2	3	4	5	6	7	8	9	10	průměr
Datastore API	1591	1649	1736	1708	2292	1752	1768	1645	1738	1645	1752.4
JPA	2677	2438	2518	2345	2321	2337	2253	2304	2203	2346	2374.2
JDO	2447	2345	2341	2619	2799	2912	2581	2632	2411	2192	2527.9

přístupů z hlediska rychlosti vhodnější využít právě JPA.

4.8 Testování zátěže

Pro testování zátěže bylo na stránku posláno velké množství požadavků a cílem bylo zjistit, jak bude celá aplikace na takovýto nápor reagovat. Aby nešlo jen o jednoduché vybírání jedné stránky, tak bylo do aplikace vloženo celkem 10 000 stránek. Každá ze stránek má nastavenou svoji šablonu, takže ta se při sestavování odpovědi musela z Datastore získat také. Celá tato aplikace je veřejně přístupná na webu na adrese: <http://slim3cms.appspot.com>.

Samotné testování probíhalo pomocí programu Apache Bench (ab). Jedná se o jednoduchý program pro příkazovou řádku, kde specifikujeme adresu serveru, počet požadavků a počet souběžných požadavků, které se mají na server odeslat. Program poté vypíše výsledky testů. Vždy před testem byla aplikace navštívena, aby byla alespoň jedna instance aktivní.



Obrázek 4.4: Graf porovnání mazání 100 entit

Listing 4.1: Testování 1000 požadavků se 100 souběžnými spojeními

```
ab -n 1000 -c 100 http://slim3cms.appspot.com
```

Při prvním testu bylo posláno dohromady 1 000 požadavků se 100 současnými připojeními. Celý test trval 16,656 sekund a po skončení testu bylo na App Engine nahráno 13 aktivních instancí.

Při druhém testu byly parametry stejné. Tento test následoval hned po prvním, kdy byla většina instancí stále aktivních a trval test pouhých 8,619 sekund. Což je skoro polovina původního času, to díky již připraveným instancím, které stihly rychle zareagovat na vzniklý nápor. Na konci testu bylo aktivních 20 instancí.

Listing 4.2: Testování 4000 požadavků s 1000 souběžnými spojeními

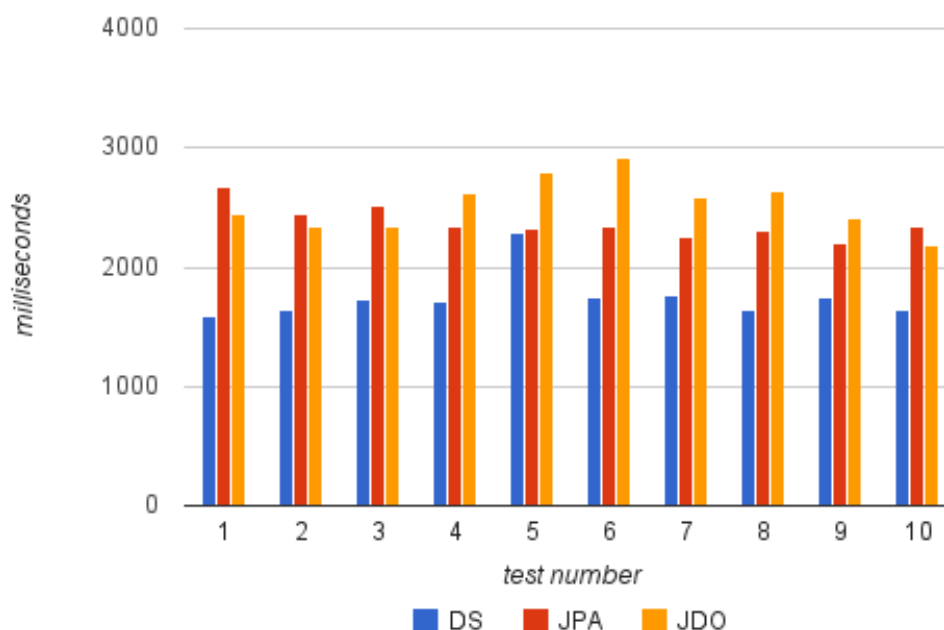
```
ab -n 4000 -c 1000 http://slim3cms.appspot.com
```

Při posledním testu bylo posláno 4 000 požadavků s 1 000 současnými připojeními a byl spuštěn krátký čas po druhém testování. Celý test trval 15,748 sekund a po skončení bylo aktivních celkem 30 instancí aplikace.

Bohužel více požadavků se mi odeslat nepodařilo, jelikož při překročení 4 000 požadavků program Apache Bench nechtěl test dokončit. Problém nebude u App Engine, protože se stejný problém projevoval i u jiných serverů. Testování jsem prováděl na 10 Mb domácí

Tabulka 4.5: Tabulka porovnání zátěže

požadavků	spojení	instancí	doba testu v sekundách
1000	100	13	16,656
1000	100	20	8,619
4000	1000	30	15,748



Obrázek 4.5: Graf porovnání mazání 100 entit

line. Kvóty byly tímto testem poznamenány minimálně až na procesorový čas. Toho bylo spotřebováno 1,24 hodiny, tedy 19% z celkových 6,5 hodiny za den. Jednalo se ale i o zdroje spotřebované při vkládání 10.000 entit do aplikace, takže na samotný test připadá zhruba 1 hodina procesorového času.

Test by mohl být zajímavější, pokud by App Engine dostával požadavky z více současně spuštěných programu Apache Bench z rozdílných sítí. Dalo by se předpokládat, že by byly nahrány další instance podle potřeby a požadavky by byly rovnoměrně roz distribuovány. Ke znatelnému zatížení App Engine by byla potřeba mnohem větší infrastruktura a zdroje, příkladem může být právě aplikace www.officialroyalwedding2011.org (viz kapitola 2.9), kde bylo požadavků mnohonásobně více a cloud si s nimi dokázal bez problému poradit.

Velkého zatížení naší aplikace se tedy na App Engine nemusíme bát, cloud se s nimi dokáže správně vypořádat a rozdělit zátěž. Jediné co se může stát je, že přesáhneme některou z kvót, ale ty lze jednoduše dokoupit. Velkou výhodou je samostatná správa počtu instancí

podle potřeby, App Engine takto dokáže efektivně využít hardwarové zdroje.

Kapitola 5

Závěr: Zhodnocení

5.1 Jednoduché škálování

App Engine nám umožňuje psát jednoduše škálovatelné aplikace a poskytuje nám k tomuto účelu zajímavou platformu pro jednoduchý vývoj a nasazení aplikací. Možnost jednoduché škálovatelnosti ale přináší do vývoje některá omezení a rozdílné vývojářské postupy. App Engine motivuje pro využití svých služeb velmi zajímavým business modelem, kde malé a málo využívané aplikace nemusí platit nic a platba za hosting je nutná až po překročení vysokých kvót. Toto je velmi zajímavá možnost pro začínající aplikace, takzvané *start-upy*, kde můžeme spustit projekt ihned a výdaje spojené s hostingem přijdou, až aplikace začne prosperovat.

5.2 Zhodnocení porovnání Datastore API, JPA, JDO

Jako výsledek této bakalářské práce vzniklo několik aplikací. Většina z nich byly jen testovací prototypy na ověření některé z funkcí, anebo pro vyzkoušení práce se službami, které App Engine nabízí. Nejzajímavějšími z nich byly tři aplikace TodoList, pro tři různé využití možnosti práce s uložištěm: Datastore API, JPA a JDO. Tyto aplikace sloužily k porovnání rychlosti každého z těchto přístupů. Nejrychlejším z těchto přístupů byl dle očekávání Datastore API, který je připraven právě pro práci s Datastore, nevýhodou je složitější práce než s JPA a JDO. Dalším důležitým poznatkem je, že uložiště na App Engine je optimalizované pro čtení, takže je výběr dat několikanásobně rychlejší než vkládání, úprava a mazání dat.

Do budoucna by se dal otestovat výběr více závislých entit z uložiště s různými typy relací (*one-to-one*, *one-to-many* a *many-to-many*). Každý ze způsobů práce s uložištěm může využívat jiný typ propojení - muselo by se zařídit, aby byly testy spravedlivé. Všechny typy spojení by musely být v Datastore uloženy stejně.

5.3 Zhodnocení porovnání testů zátěže

Největší a nejprínosnější aplikací je Content Management System využívající framework Slim3, který je optimalizovaný přesně pro použití na App Engine. Tato aplikace je nejrozsáhlejší a bez problému může být nasazena pro jednoduché stránky. Hlavním přínosem

této aplikace bylo vyzkoušet reálnou aplikaci s vazbami mezi entitami. Pro tuto aplikaci bylo velmi výhodné použití frameworku, který urychlil vývoj a pomohl usnadnit některé části programu, například optimistické zamykání. Navíc nám tento framework dává dost volnosti, takže z něj můžeme například použít jen část pro práci s uložištěm.

Tato aplikace byla použita na testování vysoké zátěže na rozsáhlejší a plnohodnotnou aplikaci. Cílem testu bylo zjistit, jak se bude App Engine chovat a jak kvalitní bude možnost rozložení zátěže. Na aplikaci bylo posláno velké množství požadavků a byl měřen celkový čas zpracování požadavků a rychlost odpovědi aplikace. App Engine si sám podle zatížení aplikace rozhodl, kolik instancí aplikace má být nahráno, aby byly všechny požadavky zpracovány a zátěž byla rovnoměrně rozložena. O toto všechno se stará App Engine sám, navíc rozhodne na které z datacenter aplikaci nahraje. Kvóty touto zátěží byly ovlivněny jen minimálně. Jedině u procesorového času byla vidět větší spotřeba, ale stále zde byly rezervy.

5.4 Osobní přínos

Největším přínosem pro mne osobně, bylo napsání větší aplikace pro App Engine. Dříve jsem zkoušel jen některé jednoduché aplikace pro vlastní potřebu. Většinu z nich sloužila jen pro demonstraci určitého API. Nyní mám praktické zkušenosti s větší aplikací a myslím, že jsem dokázal proniknout do fungování App Engine a dobře znám jeho výhody a nevýhody. Troufnu si nyní i na rozsáhlejší aplikaci s reálnou zátěží a plánuji do budoucna takovou aplikaci napsat. Zajímavá se mi jeví možnost naprogramovat aplikaci pro sociální sítě, kde se zajímavé projekty rozšíří velmi rychle a App Engine je tedy vhodným kandidátem, v opačném případě pokud aplikace nebude pro uživatele zajímavá, budou náklady na hosting nulové.

Kromě práce s App Engine jsem se při vytváření této bakalářské práce naučil efektivně využívat verzovací systém Git¹ a všechny zdrojové kódy jsou tak k dispozici na gitovském hostingu `github.com` na adrese: `verb|http://github.com/jskvara|`. Dále jsem pro publikování výsledků práce využíval hosting projektů na Google code², kde je možnost publikovat své výsledky, zdrojové kódy, aplikace ke stažení, dokumentaci projektu ve formátu wiki³ a další. Při samotném vytváření textu bakalářské práce, jsem se seznámil se nástrojem LaTeX, který slouží k profesionální sázbě a tisku dokumentů, což pro mne bylo velmi zajímavé a určitě tento formát v budoucnu použiji znovu.

¹Git - <http://git-scm.com/>

²Google code - <http://code.google.com/>

³Wiki formát je určen pro publikaci převážně HTML (HyperText Markup Language) obsahu generovaného běžnými uživateli, kde je syntaxe pro uživatele přívětivější

Příloha A

Seznam použitých zkratk

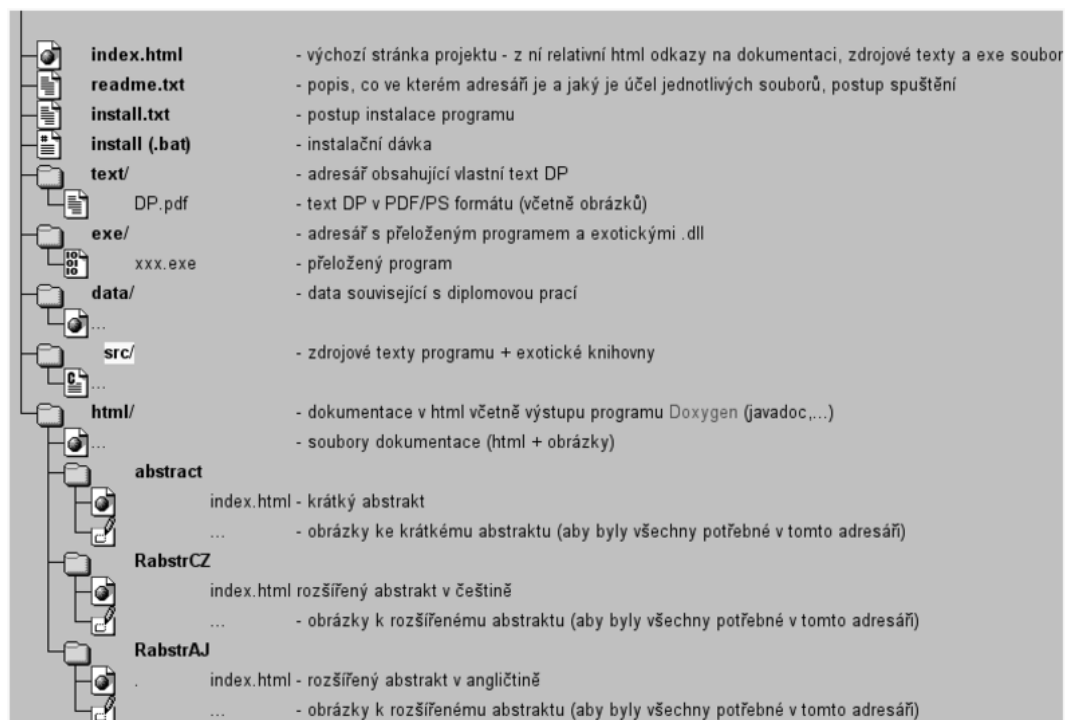
API Application Programming Interface - rozhraní pro programování aplikací

CEO Chief executive officer - ředitel

URL Uniform Resource Locator - jednoznačný identifikátor u webových stránek, například:
`http://www.example.com`

Příloha B

Obsah příloženého CD



index.html	- výchozí stránka projektu - z ní relativní html odkazy na dokumentaci, zdrojové texty a exe soubor
readme.txt	- popis, co ve kterém adresáři je a jaký je účel jednotlivých souborů, postup spuštění
install.txt	- postup instalace programu
install (.bat)	- instalační dávka
text/	- adresář obsahující vlastní text DP
DP.pdf	- text DP v PDF/PS formátu (včetně obrázků)
exe/	- adresář s přeloženým programem a exotickými .dll
xxx.exe	- přeložený program
data/	- data související s diplomovou prací
...	
src/	- zdrojové texty programu + exotické knihovny
...	
html/	- dokumentace v html včetně výstupu programu Doxygen (javadoc,...)
...	- soubory dokumentace (html + obrázky)
abstract	
index.html	- krátký abstrakt
...	- obrázky ke krátkému abstraktu (aby byly všechny potřebné v tomto adresáři)
RabstrCZ	
index.html	- rozšířený abstrakt v češtině
...	- obrázky k rozšířenému abstraktu (aby byly všechny potřebné v tomto adresáři)
RabstrAJ	
index.html	- rozšířený abstrakt v angličtině
...	- obrázky k rozšířenému abstraktu (aby byly všechny potřebné v tomto adresáři)

Obrázek B.1: Seznam příloženého CD

Na GNU/Linuxu si strukturu příloženého CD můžete snadno vyrobit příkazem:

```
$ tree . >tree.txt
```

Ve vzniklém souboru pak stačí pouze doplnit komentáře.

Z **README.TXT** (případně index.html apod.) musí být rovněž zřejmé, jak programy instalovat, spouštět a jaké požadavky mají tyto programy na hardware.

Adresář **text** musí obsahovat soubor s vlastním textem práce v PDF nebo PS formátu, který bude později použit pro prezentaci diplomové práce na WWW.