České vysoké učení technické v Praze
Fakulta elektrotechnická

katedra počítačové grafiky a interakce

ZADÁNÍ DIPLOMOVÉ PRÁCE

Student: **Bc. Jakub Motyčka**

Studijní program: Otevřená informatika (magisterský)
Obor: Softwarové inženýrství

Název tématu: **SaaS Autoservis**

Pokyny pro vypracování:

Analyzujte aplikaci autoservis a navrhněte možnosti jejího rozšíření na multitenantní architekturu.
Analyzujte a navrhněte možnosti jejího nasazení u poskytovatelů cloudových služeb.
Vytvořte z dané existující aplikace autoservis jednotnou instanci, ve které může běžet několik klientských autoservisů současně. Budou mít společnou databázi, server a nastavení.
Data uživatelů je třeba oddělit, aby měli přístup k fakturám a zakázkám jen v rámci svého autoservisu a byl zachován princip multitenance SaaS aplikace.
Vytvořte administrační rozhraní pro správu jednotlivých autoservisů, statistiku počtu autoservisů, vytvořených zakázek, apod. Bude možno upravovat globální údaje jako sazby DPH a typy aut. Vytvořte stránku pro registraci nového autoservisu.
Zhodnoťte škálovatelnost na současné infrastruktuře a v cloudu, proveďte zátěžový test.
Zhodnoťte míru oddělení uživatelů, možná bezpečnostní rizika a přínos vašeho řešení oproti původnímu stavu.

Seznam odborné literatury:

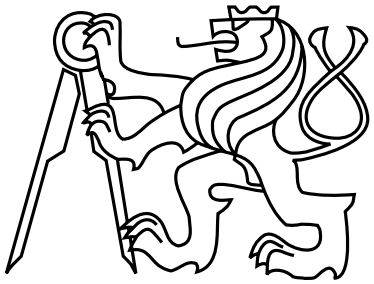
Hartl, Michael. Ruby on Rails Tutorial: Learn Web Development with Rails. Addison-Wesley, 2012.
Bigg, Ryan. Multitenancy with Rails. Leanpub.com, 2013.
Bezemer, Cor-Paul, and Andy Zaidman. "Multi-tenant SaaS applications: maintenance dream or nightmare?." Proceedings of the Joint ERCIM Workshop on Software Evolution (EVOL) and International Workshop on Principles of Software Evolution (IWPSE). ACM, 2010.

Vedoucí: Ing. Karel Pařízek

Platnost zadání: do konce letního semestru 2014/2015

V Praze dne 19. 2. 2014

České vysoké učení technické v Praze
Fakulta elektrotechnická
Katedra počítačů



Diplomová práce

SaaS Autoservis

Bc. Jakub Motyčka

Vedoucí práce: Ing. Karel Pařízek

Studijní program: Otevřená informatika, Magisterský

Obor: Softwarové inženýrství

6. ledna 2015

Poděkování

Chtěl bych poděkovat všem, kteří mi poskytli rady při realizaci mé diplomové práce, hlavně lidem ze společnosti Blueberryapps.

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 zákonu (autorský zákon).

V Praze dne 12. 5. 2014 .....

Abstract

The subject of the thesis is Aplikace Autoservis and it's converting to SaaS application. The application uses Ruby On Rails web framework and MongoDB database. The content of the thesis describes the application and possible converting options to SaaS and it's deploy to a cloud according to used technologies. The conversion of application is realised by a shared database approach, data are separated by identificators, this approach of data storage is most effective according to usage of storage.

The final application is able to provide it's functionality as a service without necessary manual effort. Consequences of that fact are reduction of effort needed to create a new instance, maintenance of only one instance of application and further reduction of costs of operating the application because of reduction of storage requirements.

Abstrakt

Předmětem práce je Aplikace Autoservis a její převod na SaaS aplikaci, tedy aplikaci, která je poskytována formou služby. Aplikace Autoservis je vytvořena pomocí webového frameworku Ruby On Rails a využívá databázi MongoDB. Práce popisuje zmiňovanou aplikaci a dále možnosti jejího převodu na SaaS a nasazení v cloudu vzhledem k použitým technologiím. Pro převod aplikace na SaaS byl pro oddělení dat zvolen přístup společné databáze s daty oddělenými pomocí identifikátorů, tento způsob uložení je neefektivnější vzhledem k využití úložiště.

Výsledná aplikace umožňuje nabízet její funkčnost jako službu bez nutnosti manuálního nasazení. To má za důsledky snížení úsilí při vytváření nové instance, údržbu pouze jedné instance aplikace a dále snížení nákladů spojených s provozem díky snížení nároků na velikost úložiště.

Obsah

Seznam obrázků

Seznam tabulek

Kapitola 1

Úvod

Jak již název práce napovídá, předmětem práce bude SaaS aplikace. Tento pojem je v pozadí všech velkých současných aplikací. Každý z dnešních uživatelů webových aplikací přichází denně do styku se SaaS aplikacemi. Významem této zkratky je slovní spojení Software As a Service, což v překladu znamená Software Jako Služba. Jedním z nejjednodušších příkladů služby, kterou lze označit slovem SaaS, je webmail, který jistě každý dobře zná. V případě webmailu je službou ona aplikace pro obsluhu e-mailu, neuvažujeme-li samotné poskytování emailové schránky.

Nejedná se tedy o běžný model aplikace, tak jak je známý z klasického desktopového prostředí, ve kterém si uživatel nainstaluje aplikaci na svůj počítač, kde ji dále využívá. SaaS označuje takové aplikace, které jsou uživateli sdíleny, všichni uživatelé používají jednu a tutéž aplikaci, pouze s jinými daty. Tento model má spoustu výhod ale i některé nevýhody.

Mezi hlavní výhody z pohledu uživatele patří ta skutečnost, že uživatel se nemusí starat o instalaci, nevznikají problémy s různými prostředími, uživatel může používat libovolný operační systém. Další nespornou výhodou takového systému je vždy aktuální verze taková, kterou poskytovatel služby nabízí.

Využívání takových aplikací není jen snazší ale mnohdy také finančně výhodnější. Uživatel není nucen například k pořízení operačního systému. Uvažuje-li se nasazení takové aplikace například ve firmě, jediným požadavkem pro použití SaaS aplikací je zpravidla webový prohlížeč, který je stejný v linuxových distribucích jako v operačním systému Windows, uživatelské rozhraní aplikace je tedy v obou případech také stejné a nejsou třeba hlubší znalosti daného systému.

Velmi důležité jsou i výhody z pohledu poskytovatele SaaS aplikace, poskytovatel se stará pouze o jednu instanci a díky tomu odpadají problémy s nasazováním nových verzí, provozem aplikace v různých prostředích, vytvářením záloh apod. A dále se díky použití jediné instance sníží náklady na provoz, protože jedna instance slouží pro velké množství uživatelů.

Tento způsob nabízení aplikaci má ale i své nevýhody, uživatel je přímo závislý na poskytovateli a pokud bude mít poskytovatel výpadek, pak nemá uživatel možnost s aplikací pracovat. Dále je uživatel závislý na tom, že jeho data se nacházejí v prostředí, o kterém nic neví. Nemůže vědět, kdo všechno může k takovým datům přistupovat nebo zda jsou dostatečně zálohována.

Popularita SaaS aplikací stále roste, tento fakt je patrný například z toho, že nejpopulárnější balík kancelářských aplikací Microsoft Office, který je v myslích mnoha uživatelů považován za nedílnou součást instalace operačního systému Windows, je nyní dostupný jako webová služba, tedy SaaS aplikace. Obdobné funkce jako Microsoft Office nabízí i Google formou webové aplikace Google Drive.

Tyto webové aplikace mají oproti běžným desktopovým apikacím již zmíněné výhody, a dále nabízejí specifické výhody jako například sdílení dokumentů, které například umožňuje několika lidem pracovat v jednu chvíli na stejném dokumentu apod.

Práce bude popisovat vytvoření takové SaaS aplikace, která umožní libovolnému počtu uživatelů sdílet jednu aplikaci tak, aby každý měl svá data a nastavení. Konkrétně se bude jednat o vytvoření SaaS Aplikace Autoservis, která umožní uživatelům snadno a rychle založit vlastní systém pro vedení autoservisu. Tato aplikace umožní spravovat veškeré akce běžného autoservisu od vytváření faktur po správu pokladen.

Dále bude práce popisovat vytvoření systému pro správu těchto jednotlivých instancí Aplikace Autoservis. V tomto systému bude možno sledovat data jednotlivých autoservisů, případně vymazávat nevhodné autoservisy.

Po popisu samotné aplikace budou diskutovány možnosti nasazení takové aplikace v cloudovém prostředí. Bude uvedeno několik poskytovatelů těchto služeb, jejich služby budou popsány a bude uvedena vhodnost nasazení a finanční stránka. Dále bude následovat popis postupu, který byl aplikován k vytvoření zmíněné SaaS Aplikace Autoservis.

Na závěr bude vytvořená SaaS aplikace otestována zátěžovým testem, který prověří kolik uživatelů bude moci současně aplikaci používat.

Kapitola 2

Popis problému, specifikace cíle

2.1 Základní pojmy

Před samotnou definicí problému je třeba definovat některé pojmy, které jsou s danou problematikou úzce spjaté.

2.1.1 Tenant

Pojmem tenant se rozumí organizační entita, která si pronajímá multitenantí SaaS řešení. Typicky tenant seskupuje uživatele dané organizace.

2.1.2 Multiuser

Multiuser aplikace umožňuje více uživatelům používat stejnou aplikaci. Takovou aplikací je i stávající verze Aplikace Autoservis, která umožňuje pro daný autoservis definovat libovolný počet uživatelů, kteří ovšem sdílejí stejná data.

2.1.3 Multitenantní aplikace

Multitenantní aplikace umožňuje zákazníkům(tenantům) sdílet stejné hardwarové zdroje prostřednictvím poskytování sdílené aplikace a databáze, ale i přes sdílení umožňuje nastavení aplikačního prostředí tak, aby vyhovovalo jejich potřebám. [10]

Klíčové vlastnosti multitenantní aplikace tedy jsou:

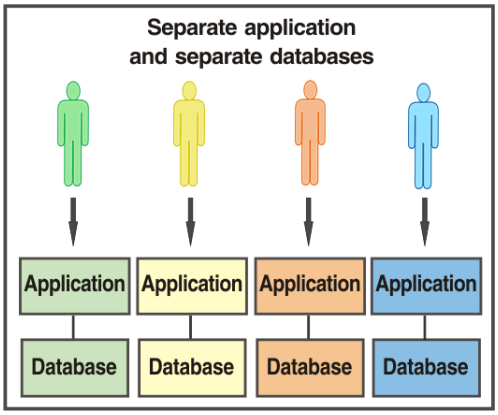
1. Schopnost aplikace sdílet hardwarové zdroje.
2. Možnost vysoké úrovně konfigurovatelnosti aplikace.
3. Architektonický přístup, kterým jednotliví tenanté používají jednu instanci aplikace a databáze.

2.2 Popis problému

Hlavním problémem, kterým se práce zabývá je vytvoření SaaS aplikace. Popis pojmu SaaS je již nastíněn v úvodu (viz 1). Jedná se o model nabízení aplikace formou služby. Taková aplikace se používá pomocí tenkého klienta, kterým je zpravidla webový prohlížeč. Uživatel SaaS aplikace tedy nemusí řešit instalaci, provoz ani údržbu dané aplikace.

Pojem SaaS úzce souvisí s pojmem multitenance, který označuje aplikace sloužící více různým zákazníkům v rámci jedné dané aplikace. Dá se tedy říci, že SaaS aplikace je aplikací multitenantní (viz 2.1.3), není to však pravidlem, SaaS může být pouze typu multiuser.

Pro představu je uveden aktuální model používaný Aplikací Autoservis [12]:



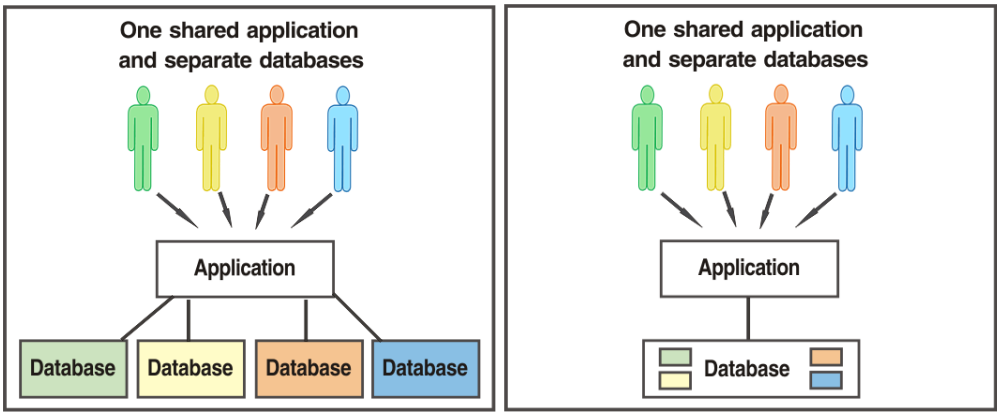
Obrázek 2.1: Aktuální použití Aplikace Autoservis

Uživatelé i přes to, že využívají stejnou aplikaci, tak mají každý svou databázi a aplikaci, která se liší pouze konfigurací. Proč je to problém? Tento model samozřejmě funguje, ale je značně nevýhodný a to z několika důvodů. Tím hlavním důvodem jsou finance, pro každou instanci aplikace musí být vlastní server, to dnes lze samozřejmě řešit pomocí serveru jednoho, ale i tak se jedná o oddělené instance, které je třeba udržovat a při nasazování nové verze musí být verze nasazena pro všechny, v případě různého nastavení daných aplikací je pak třeba upravovat konfigurační soubory, případně používat specifická řešení. Dalším problémem z finančního hlediska je oddělená databáze, tento přístup vyžaduje mnohem více diskového prostoru než je tomu v případě sdílené databáze a s diskovým prostorem rostou finanční náklady.

Problematické je pak i vytvoření nové instance takové aplikace, je třeba upravovat nastavení serveru, vytvořit novou databázi, vytvořit specifickou konfiguraci apod. V případě multitenantní aplikace je toto zpravidla řešeno pouhým vyplněním formuláře.

Náplní této práce tedy bude převod Aplikace Autoservis používající výše uvedený model na aplikaci multitenantí, která bude používat jeden z modelů uvedených na obrázku 2.2. [12]

Stávající Aplikace Autoservis je vytvořená pomocí webového frameworku Ruby On Rails a využívá databázi MongoDB, tyto dvě technologie je třeba zachovat. Během popisu řešení



Obrázek 2.2: Multitenantní přístupy

problému budou diskutovány různé možnosti řešení a jejich vhodnost vzhledem k použitým technologiím.

Aplikace Autoservis slouží pro vedení evidence zakázek, sladových položek, služeb, zákazníků a pokladen v běžném autoservisu. Z definice multitenance, která je uvedena výše, vyplývá, že je třeba vytvořit aplikaci, která umožní vytváření instancí autoservisů, nebude tedy sloužit pro správu jednoho autoservisu ale libovolného počtu autoservisů. Veškerá funkčnost musí být v rámci jedné instance aplikace.

Výsledná aplikace tedy umožní přidávání nových autoservisů do systému pomocí jednoduchého webového registračního formuláře. Uživatel pak po zadání přihlašovacích údajů, které byly zadány při registraci, bude moci pracovat s vlastní instancí Aplikace Autoservis.

Zásadním problémem vytvoření takové aplikace je oddělení dat jednotlivých instancí aplikace. Oddělení dat je možné provést několika způsoby, které budou popsány a bude diskutována vhodnost jejich použití pro aplikaci Autoservis. Dále bude zapotřebí rozhodnout, která data budou společná pro všechny instance v rámci celé aplikace a která budou dostupná pouze uživatelům dané instance Aplikace Autoservis.

2.3 Specifikace cíle

Cílem práce je:

1. Analyzovat Aplikaci Autoservis a navrhnout řešení rozšíření aplikace na multitenantní SaaS aplikaci.
2. Analyzovat a navrhnout možnosti uložení dat jednotlivých instancí vzhledem k použitým technologiím.
3. Realizovat a otestovat navržené rozšíření aplikace.
4. Vytvořit administrační rozhraní pro správu jednotlivých instancí autoservisů, které dále umožní nastavovat globální údaje pro všechny instance a sledovat statistiku počtu vytvořených instancí apod.

5. Návrh vhodného nasazení v cloudu pro výslednou aplikaci.

Kapitola 3

Analýza

Tato kapitola se bude zabývat analýzou Aplikace Autoservis. Budou definovány stávající funkční a nefunkční požadavky. Dále budou následovat diagramy případů užití a doménový model. Na závěr budou uvedeny požadavky, které rozšíří stávající požadavky tak, aby zahrnovaly multitenanci aplikace.

3.1 Aplikace Autoservis

Aplikace Autoservis slouží k evidenci činností prováděných v běžných autoservisech. Umožňuje vedení evidence zákazníků, historie jejich oprav, tisk faktur a formulářů. Dále umožňuje správu skladových položek a skladů, kromě skladových položek je možné definovat i vlastní služby, které nejsou na rozdíl od položek limitovány počtem a nemusí se přiřazovat ke skladu.

Zakázka je pak z těchto definovaných položek a služeb poskládána pomocí vyhledávání v jejich definovaných názvech. Po poskládání zakázky je možné generovat formuláře a faktury ve formátu PDF.

V zadaných informacích(zákaznících, službách, položkách) je možné vyhledávat a dále je upravovat a vymazávat. Zakázky je možné filtrovat, podle stavu objednávky, zda byla nebo nebyla zaplacena.

Dále aplikace umožňuje spravování pokladen a nastavení libovolných informací dodavatele od IČ až po logo a razítko firmy.

Aplikace rozlišuje tři role a to roli uživatel, administrátor a roli údržba. Role administrátor sdílí veškeré možnosti s rolí uživatel a navíc má možnost spravovat pokladny. Role údržba slouží pro správu uživatelů.

3.2 Požadavky

Softwarové požadavky přímo definují funkcionalitu aplikace, zpravidla jsou stěžejní částí SRS, což je smlouva mezi zákazníkem a dodavatelem softwaru, kde jsou jasně definovány veškeré požadavky na software, tyto požadavky jsou dále neměnné, proto musí být definovány jasně a jednoznačně. Požadavky se dále dělí na funkcionální a nefunkcionální. Funkcionální

požadavky definují požadovanou funkčnost aplikace, zatímco nefunkcionální definují veškeré požadavky, které se funkčnosti netýkají, tedy například definují prostředí pro běh aplikace, licence a apod.

3.2.1 Funkcionální požadavky stávající Aplikace Autoservis

- Aplikace bude pracovat s rolemi uživatel, administrátor, údržba.
- Aplikace bude umožňovat správu uživatelů systému uživateli s rolí údržba.
- Aplikace bude umožňovat spravování zakázek, položky zakázky budou tvořit skladové položky a služby, před přidáním bude možné položku nebo službu vyhledat zadáním části jejího názvu.
- Aplikace umožní při vytvoření zakázky vyhledání stávajícího nebo vytvoření nového zákazníka.
- Aplikace bude umožňovat vygenerování záručního listu ze zakázky.
- Aplikace umožní vyhledávání a filtrování(vše, zaplacené, nezaplacené) zobrazených zakázek.
- Aplikace bude umožňovat odeslání zakázky emailem.
- Aplikace bude umožňovat vygenerování formuláře o příjmu a výdeji zakázky.
- Aplikace bude umožňovat zaplacení zakázky.
- Aplikace bude umožňovat zobrazení a spravování zákazníků.
- Aplikace bude umožňovat vyhledávání v zobrazených uživatelích.
- Aplikace bude umožňovat zobrazení a správu pokladen uživateli v roli administrátor dále správu výběrů pro danou pokladnu.
- Aplikace bude umožňovat zobrazení pokladen a zobrazení výběrů z dané pokladny.
- Aplikace bude umožňovat zobrazení a správu služeb.
- Aplikace bude umožňovat zobrazení a správu skladů a dále u každého skladu zobrazit a spravovat jeho položky.
- Aplikace bude umožňovat změnu údajů dodavatele, údajů banky a změnu razítka a loga.

3.2.2 Nefunkcionální požadavky stávající Aplikace Autoservis

- Aplikace bude využívat databázi Mongo DB.
- Aplikace bude napsaná za pomoci webového frameworku Ruby On Rails verze 3.0.19.

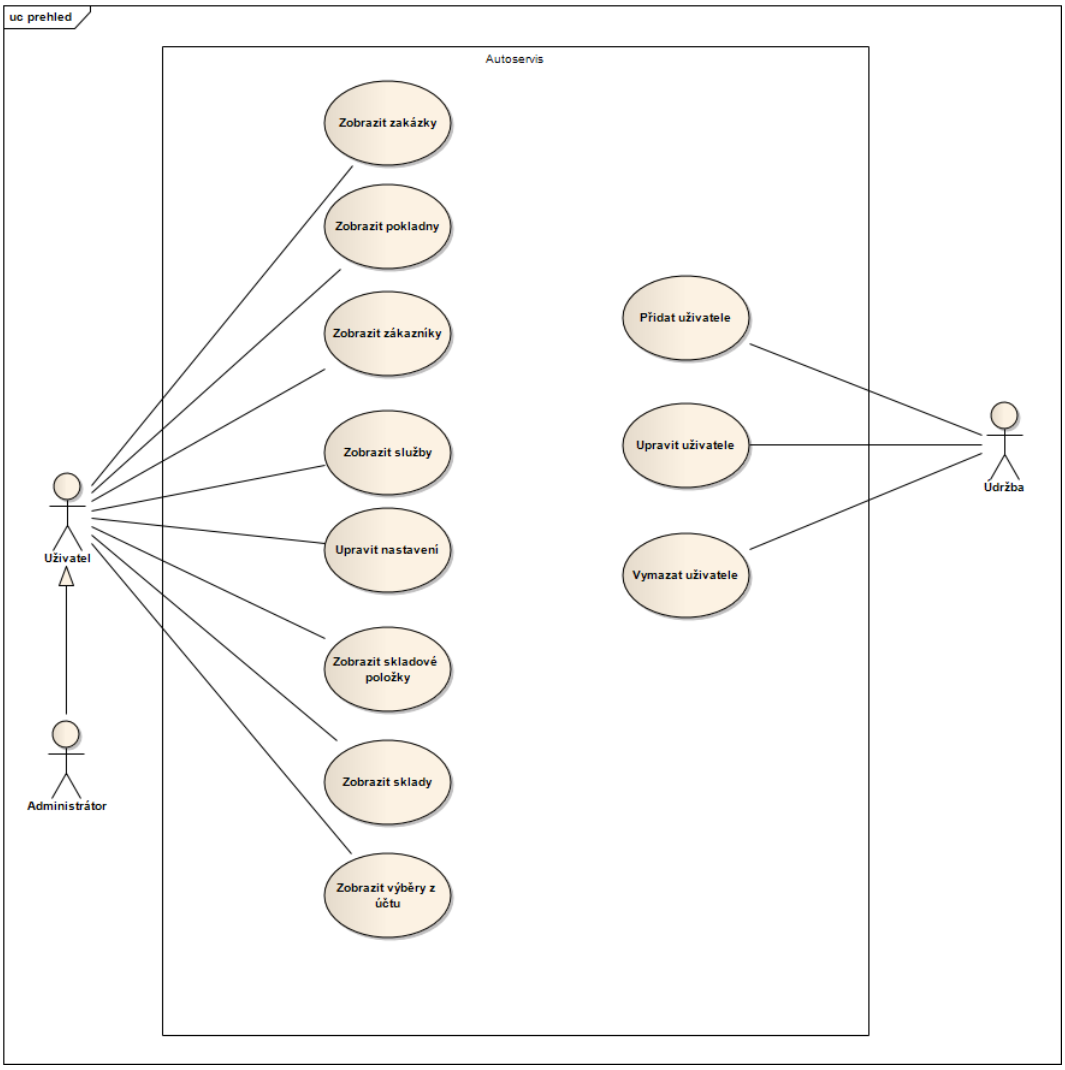
3.3 Případy užití

Diagramy případů užití jasně zobrazují jakým způsobem bude aplikace používána a jaké možnosti bude nabízet.

3.3.1 Případy užití - přehled

Uživatel Uživatel se přihlásí do systému. Aplikace zobrazí výchozí obrazovku s objednávkami a dále odkazy na další případy užití.

Údržba Uživatel v roli údržba se přihlásí do aplikace, aplikace zobrazí seznam uživatelů, možnosti úpravy, vymazání a přidání nového uživatele.



Obrázek 3.1: Případy užití - přehled

3.3.2 Případy užití - zakázky

Uživatel zvolil možnost zobrazit zakázky. Aplikace zobrazí seznam všech zakázek.

Vytvořit zakázku
Uživatel zvolí možnost vytvoření zakázky. Aplikace zobrazí formulář vytvoření zakázky. Uživatel zadá údaje o zakázce, vyplní případně vyhledá zákazníka a přidá do zakázky požadované položky. Zákazník potvrdí zakázku. Aplikace zobrazí informaci o vytvoření zakázky a zobrazí uživateli seznam zakázek.

Upravit zakázku
Uživatel vybere možnost zobrazit přehled zakázek, vybere ze seznamu zakázku a zvolí možnost upravit zakázku. Aplikace zobrazí formulář pro úpravu zakázky. Aplikace zobrazí možnosti vygenerování záručního listu, faktury a dále formulář s údaji zakázky. Uživatel upraví údaje a potvrdí formulář. Aplikace zobrazí informaci o upravení zakázky a zobrazí uživateli seznam zakázek.

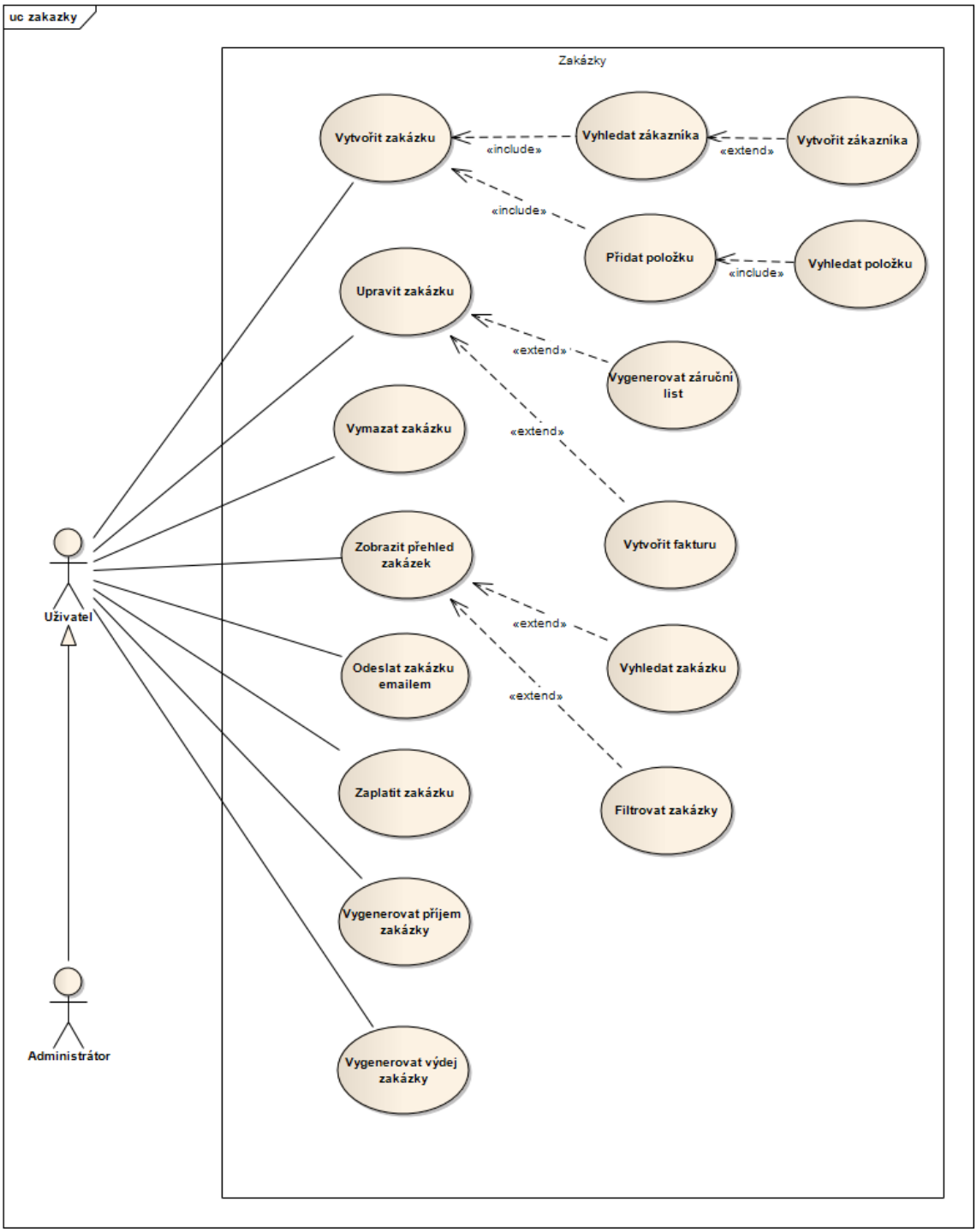
Vymazat zakázku
Uživatel vybere možnost zobrazit přehled zakázek, vybere ze seznamu zakázku a zvolí možnost smazat zakázku. Aplikace zobrazí okno s dotazem, zda uživatel chce skutečně vymazat zakázku. Uživatel potvrdí. Aplikace zobrazí informaci o smazání zakázky a zobrazí seznam zakázek.

Zobrazit přehled
Uživatel vybere možnost zobrazit přehled zakázek. Aplikace zobrazí možnost zaplacené, nezaplacené zakázky pro filtrování a dále možnost zadat kritéria pro vyhledání zakázky. Uživatel zadá kritéria, potvrdí. Aplikace zobrazí nalezené zakázky.

Odeslat emailem
Uživatel vybere možnost zobrazit přehled zakázek, vybere ze seznamu zakázku a zvolí možnost odeslat zakázku emailem. Aplikace zobrazí formulář pro odeslání zakázky emailem. Uživatel zadá email a potvrdí. Aplikace zobrazí informace o odeslání zakázky.

Vygenerovat příjem
Uživatel vybere možnost zobrazit přehled zakázek, vybere ze seznamu zakázku a zvolí možnost vygenerovat příjem. Aplikace vygeneruje formulář příjmu a nabídne uživateli uložení formuláře. Uživatel uloží formulář do počítače.

Vygenerovat výdej
Uživatel vybere možnost zobrazit přehled zakázek, vybere ze seznamu zakázku a zvolí možnost vygenerovat výdej. Aplikace vygeneruje formulář výdeje a nabídne uživateli uložení formuláře. Uživatel uloží formulář do počítače.



Obrázek 3.2: Případy užití - zakázky

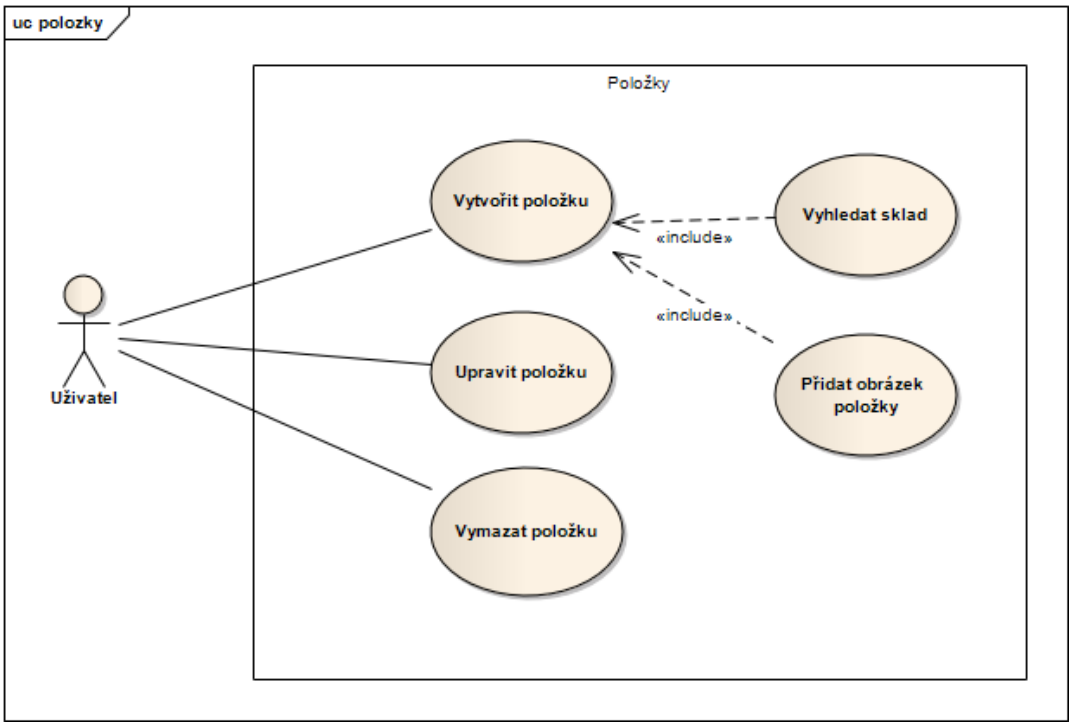
3.3.3 Případy užití - skladové položky

Uživatel vybere možnost skladové položky.

Vytvořit položku
Uživatel zvolí možnost vytvořit položku. Aplikace zobrazí formulář pro vytvoření položky. Uživatel zadá údaje, případně obrázek položky, vybere sklad a potvrdí. Aplikace zobrazí informace o přidání položky a zobrazí seznam položek.

Upravit položku
Uživatel vybere položku ze seznamu a zvolí možnost upravit položku. Aplikace zobrazí formulář pro upravení položky. Uživatel zadá údaje, případně obrázek položky, vybere sklad a potvrdí. Aplikace zobrazí informace o upravení položky a zobrazí seznam položek.

Vymazat položku
Uživatel vybere ze seznamu položku a zvolí možnost vymazat položku. Aplikace zobrazí okno s dotazem, zda uživatel chce skutečně vymazat položku. Uživatel potvrdí. Aplikace zobrazí informace o vymazání položky a zobrazí seznam položek.



Obrázek 3.3: Případy užití - skladové položky

3.3.4 Případy užití - sklady

Vytvořit sklad

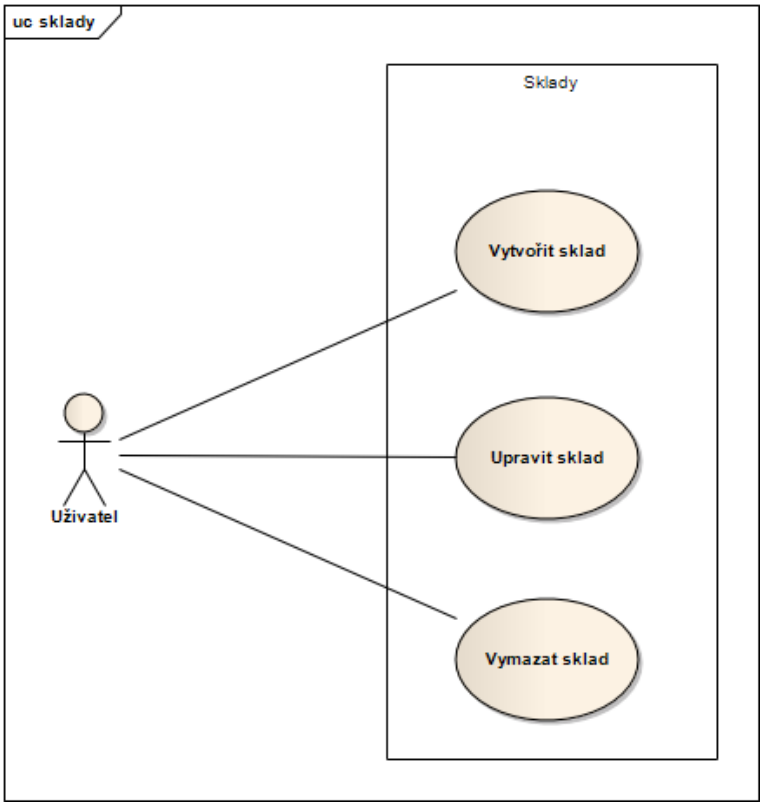
Uživatel zvolí možnost vytvořit sklad. Aplikace zobrazí formulář pro vytvoření skladu. Uživatel zadá název skladu a potvrdí. Aplikace zobrazí informace o přidání skladu a zobrazí seznam skladů.

Upravit sklad

Uživatel vybere ze seznamu sklad a zvolí možnost upravit sklad. Aplikace zobrazí formulář pro upravení skladu. Uživatel zadá název skladu a potvrdí. Aplikace zobrazí informace o upravení skladu a zobrazí seznam skladů.

Vymazat sklad

Uživatel vybere ze seznamu sklad a zvolí možnost vymazat sklad. Aplikace zobrazí okno s dotazem, zda uživatel chce skutečně vymazat sklad. Uživatel potvrdí. Aplikace zobrazí informace o vymazání skladu a zobrazí seznam skladů.



Obrázek 3.4: Případy užití - sklady

3.3.5 Případy užití - služby

Vytvořit službu

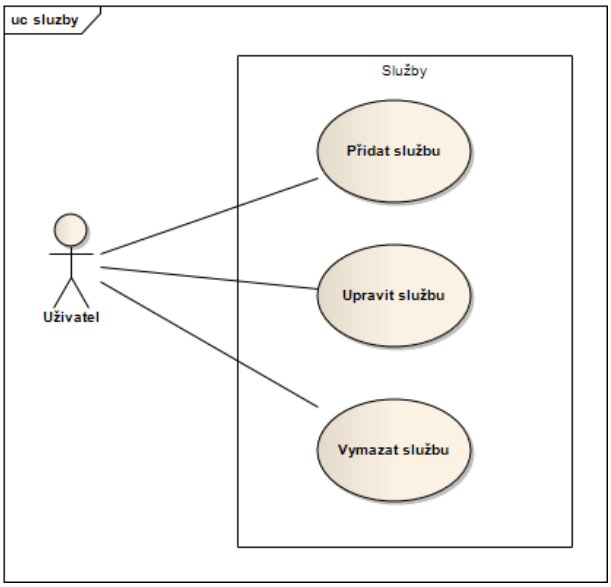
Uživatel zvolí možnost vytvořit službu. Aplikace zobrazí formulář pro vytvoření služby. Uživatel zadá název a cenu služby, potvrdí. Aplikace zobrazí informace o vytvoření služby a zobrazí seznam služeb.

Upravit službu

Uživatel vybere ze seznamu službu a zvolí možnost upravit službu. Aplikace zobrazí formulář pro upravení služby. Uživatel upraví název a cenu služby, potvrdí. Aplikace zobrazí informace o upravení služby a zobrazí seznam služeb.

Vymazat službu

Uživatel vybere ze seznamu službu a zvolí možnost vymazat službu. Aplikace zobrazí okno s dotazem, zda uživatel chce skutečně vymazat službu. Uživatel potvrdí. Aplikace zobrazí informace o vymazání služby a zobrazí seznam služeb.



Obrázek 3.5: Případy užití - služby

3.3.6 Případy užití - zákazníci

Vytvořit zákazníka

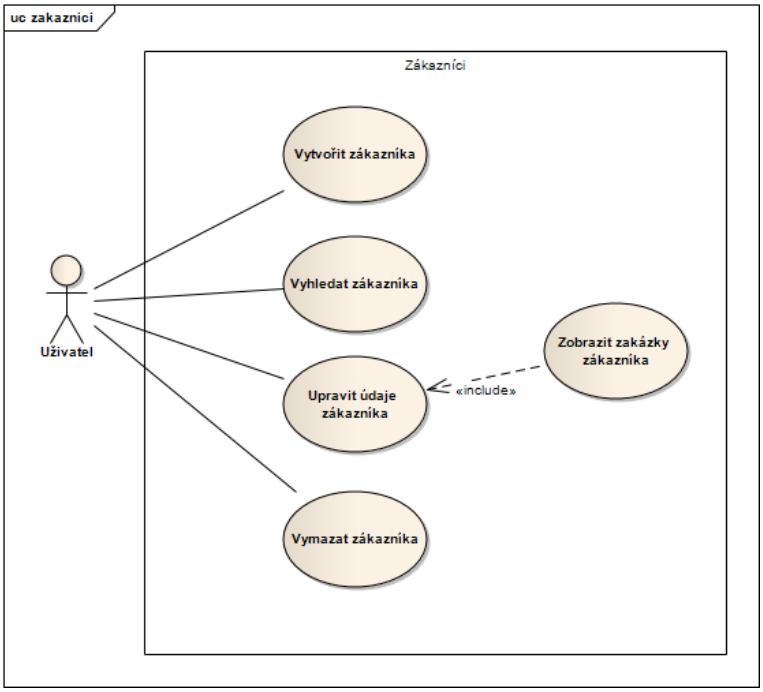
Uživatel zvolí možnost vytvořit zákazníka. Aplikace zobrazí formulář pro vytvoření zákazníka. Uživatel zadá údaje zákazníka a potvrdí. Aplikace zobrazí informace o vytvoření zákazníka a dále zobrazí seznam zákazníků.

Upravit zákazníka

Uživatel vybere ze seznamu zákazníka a zvolí možnost upravit zákazníka. Aplikace zobrazí formulář pro upravení zákazníka a dále zobrazí seznam zakázek zákazníka. Uživatel

upraví údaje zákazníka a potvrdí. Aplikace zobrazí informace o upravení zákazníka a dále zobrazí seznam zákazníků.

Vymazat zákazníka
Uživatel vybere ze seznamu zákazníka a zvolí možnost vymazat zákazníka. Aplikace zobrazí okno s dotazem, zda uživatel chce skutečně vymazat zákazníka. Uživatel potvrdí. Aplikace zobrazí informace o vymazání zákazníka a zobrazí seznam zákazníků.



Obrázek 3.6: Případy užití - zákazníci

3.3.7 Případy užití - pokladny

Vytvořit pokladnu
Administrátor zvolí možnost vytvořit pokladnu. Aplikace zobrazí formulář pro vytvoření pokladny. Administrátor zadá údaje a potvrdí. Aplikace zobrazí informace o vytvoření pokladny a dále zobrazí seznam pokladen.

Upravit pokladnu
Administrátor vybere ze seznamu pokladnu a zvolí možnost upravit pokladnu. Aplikace zobrazí formulář pro upravení pokladny. Administrátor upraví údaje pokladny a potvrdí. Aplikace zobrazí informace o upravení pokladny a dále zobrazí seznam pokladen.

Vymazat pokladnu
Administrátor vybere ze seznamu pokladnu a zvolí možnost vymazat pokladnu. Apli-

kace zobrazí okno s dotazem, zda uživatel chce skutečně vymazat pokladnu. Administrátor potvrdí. Aplikace zobrazí informace o vymazání pokladny a zobrazí seznam pokladen.

Vytvořit výběr

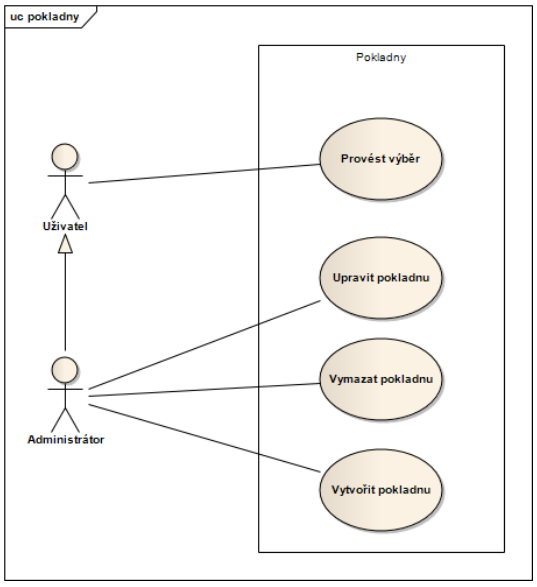
Administrátor ze seznamu zvolí pokladnu. Aplikace zobrazí výběry a možnost vytvořit výběr. Administrátor zvolí možnost vytvořit výběr. Aplikace zobrazí formulář pro provedení výběru. Administrátor zadá hodnotu výběru a potvrdí. Aplikace zobrazí informace o vytvořeném výběru a dále zobrazí seznam výběrů pro danou pokladnu.

Upravit výběr

Administrátor ze seznamu zvolí pokladnu. Aplikace zobrazí výběry a možnost vytvořit výběr. Administrátor vybere výběr a zvolí možnost upravit výběr. Aplikace zobrazí formulář pro upravení výběru. Administrátor upraví hodnotu výběru a potvrdí. Aplikace zobrazí informace o upraveném výběru a dále zobrazí seznam výběrů pro danou pokladnu.

Vymazat výběr

Administrátor ze seznamu zvolí pokladnu. Aplikace zobrazí výběry a možnost vytvořit výběr. Administrátor vybere výběr a zvolí možnost vymazat výběr. Aplikace zobrazí okno s dotazem, zda uživatel chce skutečně vymazat výběr. Administrátor potvrdí. Aplikace zobrazí informace o vymazání výběru a zobrazí seznam výběrů pro danou pokladnu.



Obrázek 3.7: Případy užití - pokladny

3.3.8 Případy užití - nastavení

Upravit údaje

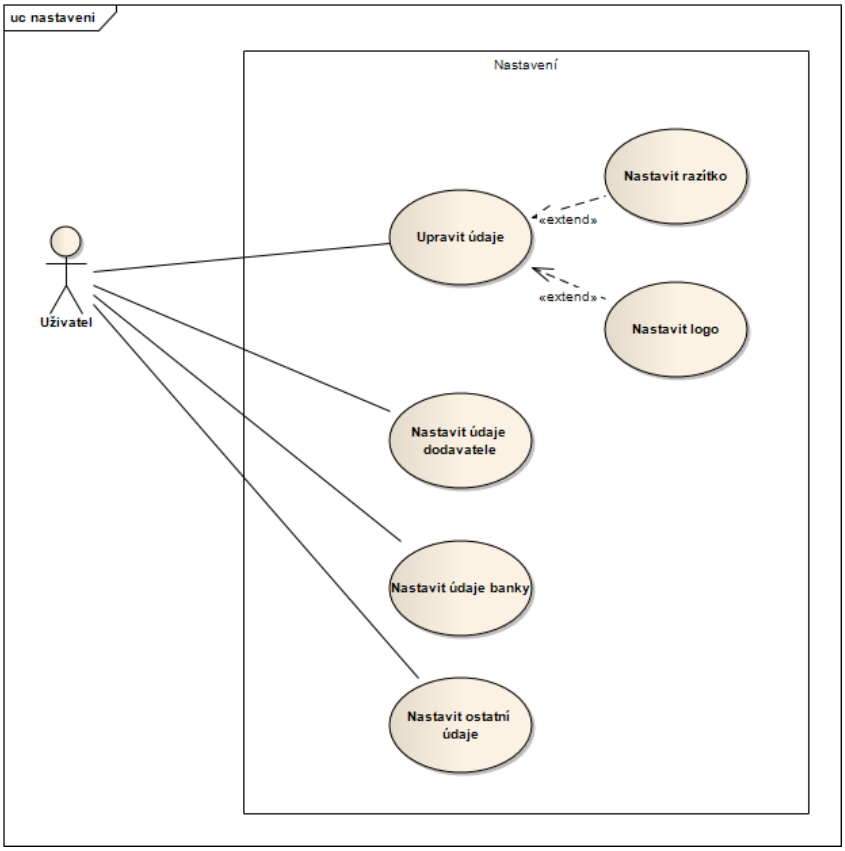
Uživatel zvolí možnost nastavení. Aplikace zobrazí formulář nastavení údajů. Uživatel nastaví údaje, razítko, případně logo a potvrdí. Aplikace zobrazí informace o provedení nastavení údajů a dále formulář nastavení.

Nastavit údaje dodavatele

Uživatel zvolí možnost nastavení. Aplikace zobrazí formulář nastavení údajů. Uživatel nastaví údaje dodavatele a potvrdí. Aplikace zobrazí informace o provedení nastavení údajů a dále formulář nastavení.

Nastavit údaje banky

Uživatel zvolí možnost nastavení. Aplikace zobrazí formulář nastavení údajů. Uživatel nastaví údaje banky a potvrdí. Aplikace zobrazí informace o provedení nastavení údajů a dále formulář nastavení.



Obrázek 3.8: Případy užití - nastavení

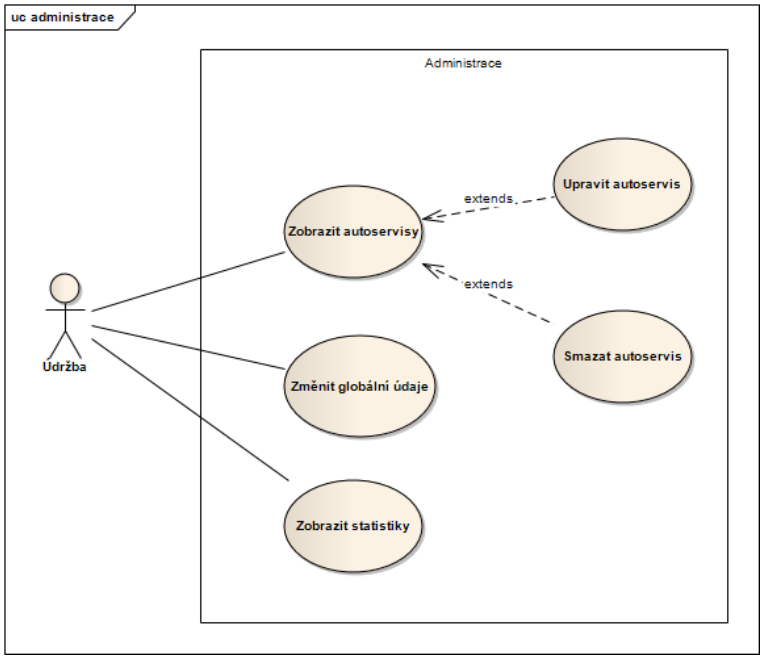
3.4 Rozšíření požadavků a případů užití

Stávající požadavky je nutné rozšířit o funkcionální požadavky kladené na administrační rozhraní. Dalším rozšířením bude přidání role návštěvník, která bude mít možnost registrace nového autoservisu.

3.4.1 Rozšiřující funkcionální požadavky

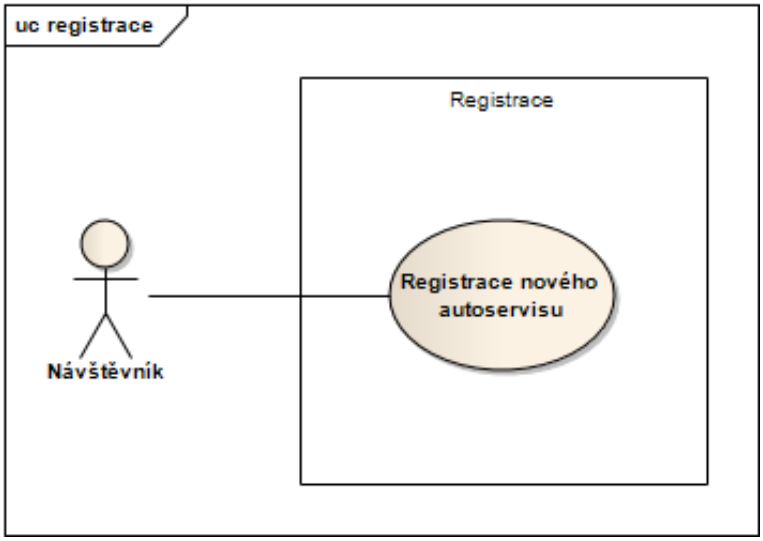
- Aplikace umožní návštěvníkům registraci nového autoservisu.
- Aplikace umožní uživateli v roli údržba zobrazení a správu instancí autoservisů.
- Aplikace umožní uživateli v roli údržba zobrazení statistik počtu vytvořených autoservisů.
- Aplikace umožní uživateli v roli údržba změnu globálních údajů.

3.4.2 Případy užití - administrace



Obrázek 3.9: Případy užití - administrace

3.4.3 Případy užití - registrace



Obrázek 3.10: Případy užití - registrace

3.5 Doménový model

Doménový model zobrazuje entity řešené problematiky a jejich vzájemné vztahy.

Popis entit doménového modelu

Uživatel uživatel aplikace.

Zakázka reprezentuje zakázku autoservisu, nese údaje o vzniku a reference na další součásti zakázky, kterými jsou zákazník, zakládající uživatel, vůz a jednotlivé položky zakázky.

PoložkaZakázky reprezentuje jednu položku zakázky, která se může být buď skladová položka nebo služba.

SkladováPoložka reprezentuje jednu skladovou položku, která má vazbu na sklad, pod který spadá a dále jednotku a hodnotu počtu této jednotky.

ObrázekPoložky obrázek skladové položky, nese informace o obrázku, který byl k dané skladové položce přidán.

Sklad reprezentuje sklad, sklad má vazby na položky, které jsou v něm naskladněny.

Služba reprezentuje službu poskytovanou autoservisem.

Vůz entita vůz obsahuje informace o opravovaném automobilu.

Výrobce reprezentuje výrobce automobilů.

Jednotka slouží k odlišení jednotek používaných v rámci autoservisu, používá se pro služby a položky, kde určuje jednotku dané položky, může nabývá hodnot jako jsou litr, hodina, kus.

Pokladna slouží ke sjednocení finančních výdajů.

Výběr reprezentuje jeden výběr v rámci dané pokladny.

Zákazník reprezentuje osobu, která si objednala opravu automobilu, obsahuje kontaktní informace.

Prodejce entita sloužící k uložení informací o autoservisu, který používá aplikace.

Faktura nese informace o vytvořené faktuře, faktura spadá pod danou zakázku a je vytvářena uživatelem, skládá se z položek faktury.

PoložkaFaktury reprezentuje jednu položku v rámci dané faktury.

Společnost reprezentuje společnost v rámci aplikace, společnost nese vazby na prodejní a nákupní faktury k daným zakázkám.

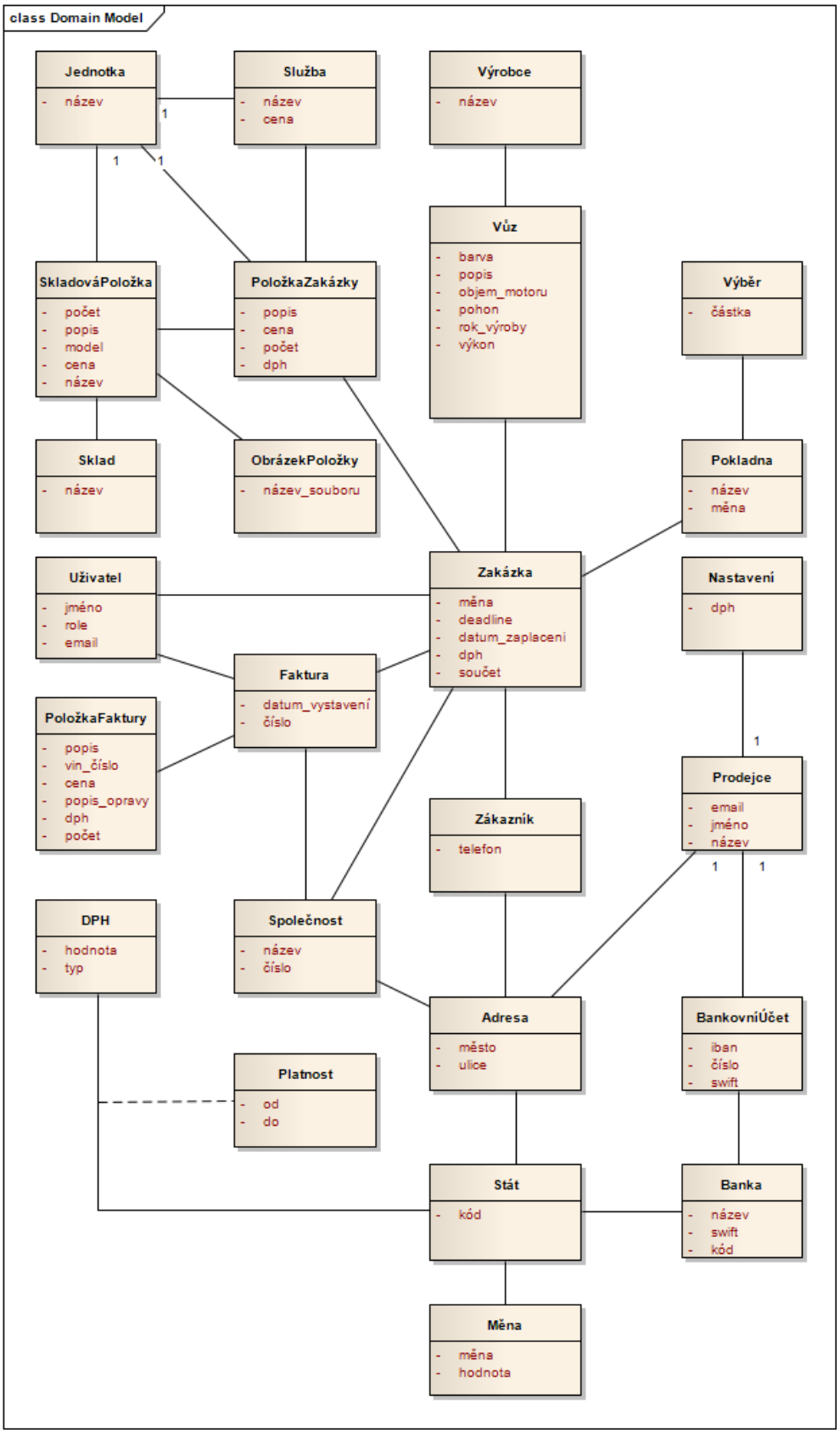
Adresa entita, jež nese adresní informace, město, ulice apod.

Stát entita pod níž spadají sazby DPH, banky a měny.

Banka reprezentuje banku, nese informace o dané bance jako je například kód banky apod.

BankovníÚčet entita sloužící pro uložení informací o bankovním účtu autoservisu.

Měna reprezentuje měnu daného státu.



Obrázek 3.11: Doménový model

Kapitola 4

Návrh

V této kapitole budou diskutovány možnosti úprav stávající aplikace tak, aby byla dosažena požadovaná funkcionalita.

Možnosti rozšíření aplikace jsou omezeny použitým frameworkem a dále použitou databází MongoDB, při výběru metody oddělení dat je nutné tuto skutečnost brát v úvahu. Převod aplikace na multitenantní lze popsat následujícími kroky:

1. určení společných dat pro všechny instance aplikace
2. oddělení dat, které jsou různá pro různé instance dané aplikace
3. realizace oddělení dat v aplikaci

Architektura uspořádání dat v multitenantních aplikacích velmi závisí na technických a obchodních cílech. Je nutné uvažovat počet tenantů a možnosti infrastruktury, která bude k běhu aplikace poskytnuta. Pokud se bude jednat o službu požadující vysokou úroveň izolace dat, pak je třeba použít způsob oddělení dat použitím vlastních databází pro každého tenanta, což má vyšší hardwarové nároky, než použití sdílené databáze pro všechny tenanty.

Pro oddělení dat se nabízí tři možnosti multitenant-data architektury [13]:

- Oddělené databáze
- Sdílená databáze, oddělení dat pomocí schemat
- Sdílená databáze, sdílené schema, oddělení dat pomocí id atributu

4.1 Datový model

Datový model zobrazuje strukturu dat, se kterými systém pracuje. Tento návrh musí být přepracován tak, aby bylo možné vytvářet více samostatných instancí Aplikace Autoservis. Pro dosažení této skutečnosti budou navrženy možnosti oddělení dat jednotlivých instancí Aplikace Autoservis. V každém případě bude nezbytné přidání entity, jež ponese informace o jednotlivých tenantech.

4.1.1 Společná data

Společná data v multitenantní aplikaci, jsou data, která se sdílí mezi jednotlivými instancemi aplikace. Tato data jsou stejná pro všechny instance a proto není nutné je oddělovat. Z datového modelu mezi společná data patří:

- Country
- Bank
- VatScheme
- VatRate
- CurrencyRate
- Unit

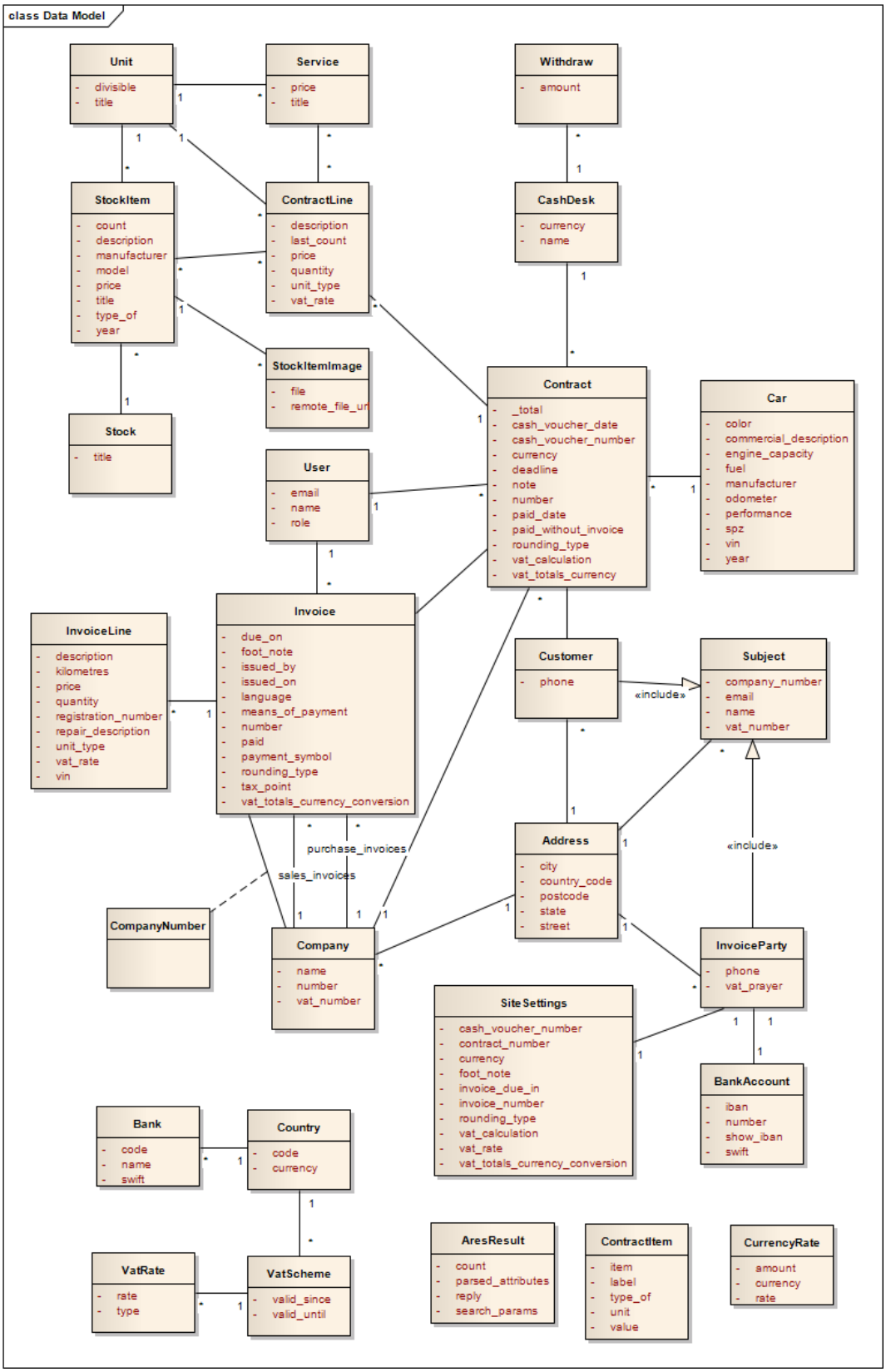
4.1.2 Nespolečná data

Nespolečná data označují data, která jsou specifická pro danou instanci, tedy pro daného tenanta. Pro Aplikaci Autoservis se jmenovitě jedná o entity:

- Address
- BankAccount
- Car
- CashDesk
- Company
- CompanyNumber
- Contract
- ContractItem
- ContractLine
- Customer
- Invoice
- InvoiceLine
- InvoiceParty
- Service
- SiteSettings

- Stock
- StockItem
- StockItemImage
- Subject
- User
- Withdraw

Data obsažena v těchto entitách musí být od sebe navzájem oddělena.



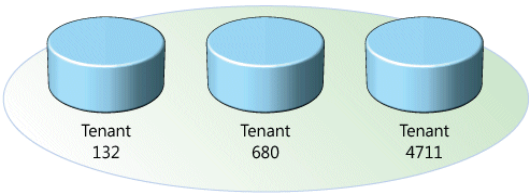
Obrázek 4.1: Datový model

4.2 Oddělení dat pomocí použití více databází

Při použití tohoto způsobu všichni tenanti sdílí výpočetní zdroje, ale mají vlastní databázi, čímž jsou jejich data oddělena od ostatních tenantů. Pro asociaci jednotlivých tenantů a jejich databází se používá databáze společná.

Vlastní databáze pro každého tenanta umožní snazší naplnění specifických potřeb daného zákazníka, například specifické plánování záloh, případně při vysokých nárocích na výkon přenos jeho dat na vlastní server apod. Tyto výhody jsou ale vykoupeny vyšší cenou na údržbu a vyššími nároky na použitý hardware.

Problémem tohoto způsobu je velká režie. V případě databáze MongoDB, je tato režie opravdu velmi značná, proto by tento způsob pro aplikaci Autoservis byl vhodný pouze v případě předpokládaného nízkého počtu tenantů. [1] [13]



Obrázek 4.2: Oddělení dat použitím více databází

Výhody

- vysoká úroveň oddělení dat
- snadná implementace

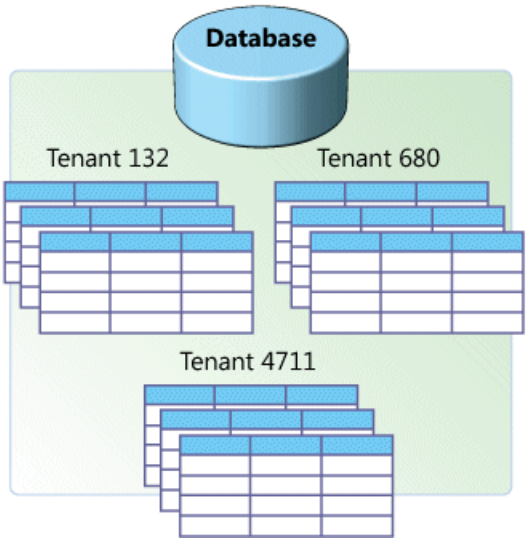
Nevýhody

- velké nároky na hardware
- velké nároky na údržbu

Tento způsob se hodí pro aplikace, které pracují s osobními daty, tedy například banky. A dále pro zákazníky, kteří vyžadují vlastní přizpůsobení aplikace.

4.3 Oddělení dat pomocí schémat

Způsob oddělení dat pomocí schémat je limitovaný možnostmi nabízenými danou databází. Schémat může být v rámci jedné databáze libovolný počet a každé schéma může mít libovolné tabulky. Schéma je tedy možné použít pro oddělení jednotlivých tabulek, tak že každý tenant bude mít svá data ve svém vlastním schématu. Společná data by se pak nacházela v jednom schématu společném pro všechny tenanty, zatímco data různá pro každého tenanta by byla ve vlastních schématech. [13]



Obrázek 4.3: Oddělení dat použitím schémat

Toto řešení je jednoduché, elegantní a nabízí vyšší izolaci dat, než pouhé oddělení dat pomocí id atributu.

V databázi MongoDB se nenachází přímo schémata, ale nabízí se alternativní přístup a to použití kolekcí.

Kolekce v MongoDB slouží k oddělení jednotlivých dokumentů, každý dokument je možné mít v různé kolekci. Bylo by tedy možné pro data každé instance použít různou kolekci. Každá kolekce by tedy nesla prefix daného tenanta a podle něj by byla vybrána konkrétní kolekce.

Tento způsob se nedoporučuje, jelikož počet kolekcí je limitován 24 000 na databázi. Dalším důvodem je, že databáze MongoDB není vytvořena tak, aby škálovala na úrovni kolekcí. [4]

Výhody

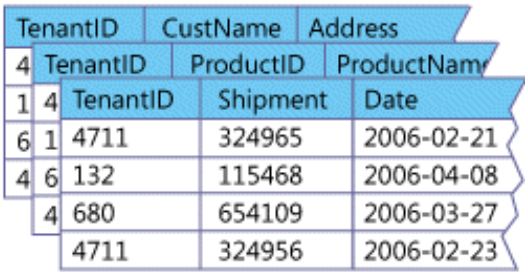
- nízké nároky na hardware
- snadná implementace

Nevýhody

- problematická práce s daty jedné instance, například přenos na vlastní server
- nehodí se pro databázi MongoDB

4.4 Sdílená databáze

Při použití sdílené databáze je třeba všechny modely rozšířit stejným id atributem, například client_id. Do této hodnoty se pak ukládá client_id dané instance Aplikace Autoservis, kdy každá instance má přiřazenu entitu Client. [13]



Obrázek 4.4: Oddělení dat použitím sdílené databáze

Vzhledem k použité databázi se jedná o nejvhodnější způsob oddělení dat.

Tento způsob je doporučovaný pro počáteční vývoj. Oproti použití oddělených databází má tento způsob menší režii, avšak s rostoucím počtem tenantů dochází ke snižování výkonu, jelikož se databázové dotazy provádějí nad velkým počtem záznamů. [4]

Realizace je u tohoto způsobu na první pohled nejobtížnější, protože je nutné upravit databázové dotazy v aplikaci tak, aby zahrnovaly podmínku pro tenant_id atribut. Toto ale není s použitím Ruby On Rails problém, protože je možné využít scopes.

Pojem scope ve frameworku Ruby On Rails označuje jakýsi pohled na databázi, je možno ho přirovnat přímo k pohledu v SQL databázích, kdy se definují podmínky, které data vyfiltrují. Je možné definovat výchozí scope pro danou entitu, nastavením výchozího scope na tenant_id lze tedy snadno docílit oddělení dat v rámci jedné entity. Tento výchozí scope ale musí být nastaven u všech entit, jejichž data jsou závislá na instanci aplikace a nejsou společná.

Pomocí scopes lze realizaci oddělení dat provést poměrně snadno. Pro realizaci tohoto způsobu lze použít již hotová řešení, kterým je například knihovna mongoid-multitenancy. Tato knihovna umožňuje všechny entity, jejichž data nejsou sdílená, rozšířit modulem mongoid-multitenancy a následně použít funkci tenant, která automaticky v daném modelu zavede tenant_id identifikátor a provede výchozí scopování.

Tento způsob je vzhledem k požadavkům nejvhodnějším pro realizaci multitenatní Aplikace Autoservis.

Výhody

- nízké nároky na hardware

Nevýhody

- nízká úroveň oddělení dat
- problematická práce s daty jedné instance, například přenos na vlastní server
- obtížnější implementace

Tento způsob je vhodné použít pro velké množství tenantů s malým počtem serverů.

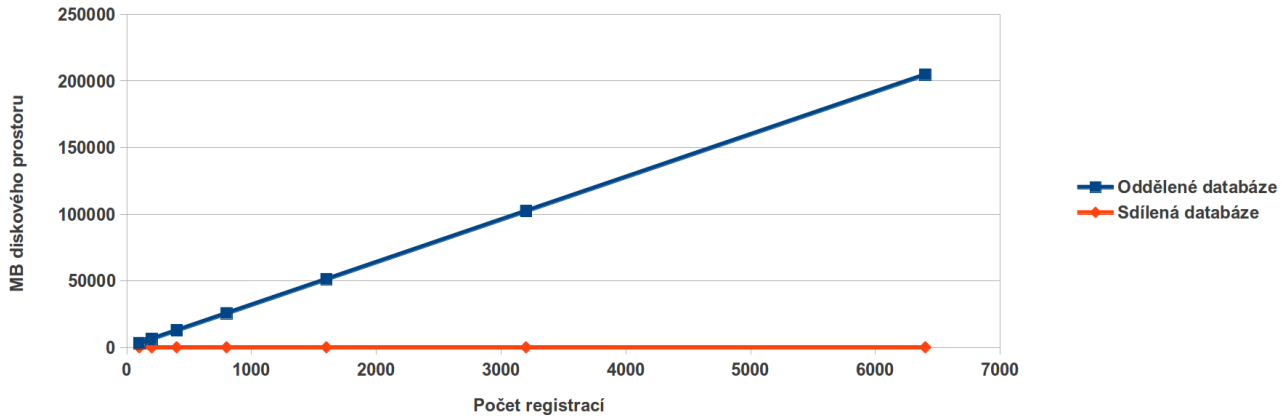
4.5 Porovnání režie

Výše uvedené způsoby je třeba uvažovat vzhledem k použité databázi. Databáze MongoDB předalokovává místo, pro každou novou databázi vytváří soubor <databasename>.0, který alokuje 64 megabajtů, další soubor má 128 megabajtů a tak dále až do velikosti 2 gigabajty. Dále je vytvářen namespace soubor <databasename>.ns, jehož výchozí velikost je 16 megabajtů. Z uvedeného vyplývá, že při použití oddělených databází a výchozího nastavení MongoDB by bylo třeba jen pro registraci jednoho autoservisu alokovat 80 megabajtů.

Výchozí alokování lze upravit pomocí parametru smallfiles, který omezí maximální velikost souboru na 512 megabajtů a dále zmenší velikost prvního souboru na 16 megabajtů. V tomto případě by počáteční velikost jedné registrace byla 32 megabajtů. Nevýhodou využití tohoto parametru je snížení výkonu u větších databází.

Dále je možné vypnout předalokování, ale tato možnost není určena pro produkční prostředí, oficiální dokumentace uvádí, že tato možnost je určena pouze pro testovací účely.

Využití diskového prostoru vzhledem k počtu registrací je demonstrováno na následujícím grafu. Hodnoty využití diskového prostoru jsou uvažovány s použitím parametru smallfiles. Pro registraci s použitím sdílené databáze se uvažují 2 dokumenty zhruba o velikosti 1kB. [1]



Obrázek 4.5: Využití diskového prostoru vzhledem k počtu registrací

[1] Velikost byla odvozena pomocí příkazu BSON::serialize(Contract.first.as_document).size

Kapitola 5

Možnosti nasazení

V této kapitole budou popsány cloudové služby a dále budou diskutovány možnosti nasazení multitenantní Aplikace Autoservis u vybraných poskytovatelů.

5.1 Cloud

Pojem cloud označuje aplikace a služby provozované na distribuovaných sítích pomocí virtualizovaných zdrojů, ke kterým je přistupováno pomocí běžných internetových protokolů a síťových standardů.

Cloud využívá dva zásadní koncepty abstrakci a virtualizaci. Abstrakce slouží k zakrytí detailů o implementaci systému od uživatelů a vývojářů. Virtualizace je využita ke sdílení zdrojů, jejichž využití je měřeno a na základě toho vyúčtováno. [16]

5.1.1 Proč použít cloud

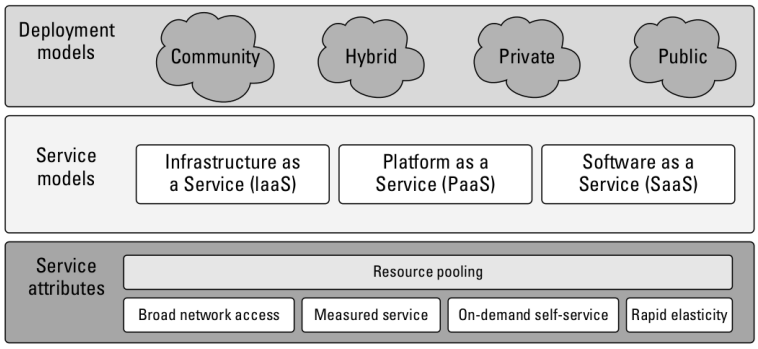
Cloudy nabízejí placení pay as you go a neomezené škálování. Díky cloud poskytovatelům není třeba investovat do pořizování a údržby hardware, případně i software. Platí se pouze za využité zdroje. Díky tomu je možné začít s malým nevýkonným serverem pro pár zákazníků a v případě, že zájem o aplikaci roste, tak navyšovat výkon jednoduchým naklikáním vyššího výkonu.

Pro SaaS aplikací je využití cloud řešení na místě, protože díky cloudu lze snadno škálovat. Škálovat lze nejen daný hardware, škálované jsou i služby nabízené poskytovateli cloud řešení. Toho lze využít například při použití databáze. V SaaS aplikaci všichni tenanté v případě společné databáze využívají jednu instanci databáze a je nezbytné, aby byla databáze dostatečně výkonná.

5.1.2 Služby

Cloud služby se rozdělují do dvou základních tříd deployment model, service model. Pro nasazení aplikace je podstatná třída service model.

Americký mezinárodní institut standardů a technologií(National Institute of Standards and Technologies, NIST). Definuje cloudové služby dle obrázku.



Obrázek 5.1: Definice cloudových služeb dle NIST

5.1.3 Vlastnosti cloudových služeb

Pro označení služby jako cloud je třeba, aby služba splňovala následující vlastnosti [16]:

On-demand self-service Klient si může opatřovat výpočetní zdroje bez pomoci zaměstnance poskytovatele služby.

Broad network access Přístup ke zdrojům je možný prostřednictvím sítě a standardních protokolů. Přístup ke zdrojům je platformně nezávislý.

Resource pooling Fyzické a virtuální zdroje jsou sdílené a dynamicky přiřazované jednotlivým klientům.

Rapid elasticity Může dojít k rapidní změně požadavků na zdroje. Z pohledu klienta jsou výpočetní zdroje neomezené, případné škálování zdrojů je prováděno manuálně nebo automaticky.

Measured service Veškeré použití služeb je měřeno a klient má přehled o využitých zdrojích. Klient platí za předem známé metriky jako například využití úložiště, počet transakcí, síťové přenosy apod.

5.1.3.1 Modely služeb

Hlavními službami poskytovanými cloudovými poskytovateli jsou:

1. Software as a service(SaaS)
2. Platform as a service(PaaS)
3. Infrastructure as a service(IaaS)

IaaS poskytuje virtuální stroje, virtuální úložiště, virtuální infrastrukturu a další hardwarové prostředky jako zdroje, které si mohou klienti v rámci poskytovatele pořizovat.

Poskytovatelé IaaS udržují pouze infrastrukturu, za veškeré další aspekty je zodpovědný klient. Mezi tyto aspekty spadá prostředí, kam se daná aplikace nasazuje, tedy operační systém, aplikace, služby apod.

PaaS poskytuje kompletní prostředí pro nasazení aplikace, tedy virtuální stroje, operační systém, aplikace, služby, vývojové frameworky, transakce a kontrolní nástroje.

Klient může nasazovat své aplikace do této infrastruktury nebo může používat již hotové nástroje, které jsou podporované danou platformou. Poskytovatel udržuje infrastrukturu, operační systém, a použitý software. Klient je zodpovědný za instalaci a údržbu nasazené aplikace.

SaaS označuje kompletní prostředí s aplikacemi. V SaaS je aplikace poskytována klientovi prostřednictvím rozhraní tenkého klienta(zpravidla se jedná o webový prohlížeč). Uživatel je zodpovědný za spravování svých dat skrze uživatelské rozhraní, vše ostatní má na starosti poskytovatel služby. [16]

Tyto modely jsou souhrnně označovány také jako SPI. Existují i další modely: Storage as a Service, Identity as a Service a dále. Nicméně modely SPI tyto možnosti zahrnují.

Z výše uvedených kategorií jsou pro nasazení relevantní pouze PaaS a IaaS, jelikož SaaS je přímo kategorie nabízející software.

Použití PaaS se hodí pro vývojáře, kteří nechtějí řešit instalaci prostředí. PaaS poskytovatelé zpravidla poskytují ideální prostředí pro danou platformu. Není třeba instalovat operační systém, konfigurovat webové servery, databáze, zabezpečení, instalovat frameworky apod. PaaS poskytovatelé řeší například i cachování. Výhodou je velmi snadné škálování.

IaaS se hodí, pokud aplikace potřebuje specifické prostředí pro běh aplikace. IaaS vyžaduje instalaci veškerého software, ale to je mnohdy ze strany poskytovatele usnadněno tím, že nabízí instalaci hotového prostředí pro danou platformu.

5.2 Vybraní poskytovatelé

K porovnání byli vybrání poskytovatelé obou kategorií PaaS i IaaS. Z každé kategorie byl vybrán jeden z hlavních poskytovatelů v dané oblasti a druhý méně známý. V případě vyvíjené aplikace je platformou Ruby On Rails, a proto budou z kategorie PaaS uvažováni pouze poskytovatelé nabízející tuto platformu. Všichni vybraní poskytovatelé jsou zahraniční, ale nabízejí výběr geografického umístění serveru v Evropě. Čeští poskytovatelé nabízejí pro platformu Ruby On Rails pouze IaaS řešení.

PaaS

- Heroku
- AppFog
- AWS Elastic Beanstalk

IaaS

- AWS EC2
- Digital ocean

5.2.1 Požadavky na cloud služby

Před analýzou možností jednotlivých poskytovatelů je třeba definovat požadavky pro běh vyvíjené aplikace.

5.2.1.1 Konfigurace

Konfigurace je odvozena od konfigurace, která je použita pro nasazení Aplikace Faktura Online[1], jejíž je Aplikace Autoservis odnož.

U počátečního provozu aplikace bude uvažována následující konfigurace:

Položka	**Hodnota**
paměť	2048MB
diskový prostor	20GB
cpu	2x

Tabulka 5.1: Požadovaná konfigurace

Hodnoty jsou uvažované pouze pro produkční prostředí. V případě nasazení vlastního systému se uvažuje použití open source software, proto jsou náklady na software nulové.

5.2.1.2 Datové přenosy

U některých poskytovatelů je pro výpočet ceny nabízených služeb třeba uvažovat i datové přenosy. Požadovaná hodnota datových přenosů je odvozena od datových přenosů používaných u stávající Aplikace Autoservis.

Hodnota přenosů byla zjištěna pomocí příkazu cat /proc/net/dev. Následně byl pomocí příkazu uptime zjištěn počet dnů a hodnoty byly přepočítány tak, aby odpovídaly jednomu měsíci.

Výsledný odhad na měsíc tedy činý **30GB**.

5.2.1.3 IOPS

Dále je třeba pro výpočet výsledné ceny u některých poskytovatelů zjistit počty operací na perzistentních úložištích.

Tyto hodnoty jsou vypočítány pomocí postupu popsaného v [2].

1. Nejprve je pomocí příkazu cat /proc/diskstats třeba zjistit počty operací. Z výstupu zmíněného příkazu je třeba vzít první a páty sloupec u daného zařízení. První sloupec vyjadřuje počet započatých operací pro čtení z disku, hodnota pátého sloupce označuje počet dokončených operací zápisu. Hodnoty se sečtou, tím máme hodnotu počtu operací.

[1] http://www.fakturaonline.cz/
[2] http://www.caseylabs.com/how-to-calculate-amazon-ebs-io-costs

2. Dále je třeba zjistit počet dnů, který se vztahuje ke zjištěným hodnotám. Počet dnů se zjistí pomocí příkazu uptime.

3. Hodnota počtu operací se pak vydělí zjištěným počtem dnů a výsledek se vynásobí hodnotou 30, která vyjadřuje požadovaný počet dnů. Výsledek po násobení je odhadovaná hodnota IOPS za měsíc.

```
cat /proc/diskstats
8       0 sda 220591 1081 7659106 566761 144806 94746 1916464 778475 0 743032 1344932
uptime
10:21:29 up 5 days,  9:02,  1 user,  load average: 0.06, 0.11, 0.14
```

Výpočet: (((220591 + 144806)/5)*30)
Výsledná hodnota počtu operací je **2192382**.

5.2.1.4 Velikost databáze

Někteří poskytovatelé nabízejí databázi jako službu, jedná se především o poskytovatele PaaS řešení. Pro tyto případy je tedy třeba určit velikost databáze. Vzhledem k rychlosti růstu velikosti databáze MongoDB bude uvažována velikost databáze alespoň 8 GB.

5.2.2 Databáze

V případě PaaS poskytovatelů je databázi MongoDB třeba získat od externích poskytovatelů. Samotná databáze je tedy poskytována jako služba. U všech vybraných poskytovatelů lze volit mezi MongoHQ a MongoLab. Cenově jsou na tom oba poskytovatelé obdobně, ceník uváděný u poskytovatele Heroku uvádí cenu $144/měsíc pro 8GB databázi u poskytovatele MongoHQ, u MongoLab pak $149/měsíc.

Cena databáze je závislá na její velikosti, pro počáteční provoz bude uvažována velikost 2GB a zvolen plán MongoLab pro produkční prostředí se dvěma instancemi. Tento plán začíná na ceně $89/měsíc, cena zahrnuje 2GB úložiště, každý další GB stojí $5/měsíc.

5.2.3 Heroku

Heroku je zřejmě neznámější PaaS poskytovatel pro platformu Ruby On Rails. Výkon serveru se určuje pomocí takzvaných dynos, což jsou jednotky výkonu. Dynos se dle výkonu rozdělují na dva druhy. Levnější verze dyno má 512MB RAM a 1 cpu za $0.05/hodinu, výkonnější verze pak 1024MB RAM a 2 cpu $0.10/hodinu. Velkou výhodou je, že první dyno levnější verze je zdarma. Databázi MongoDB lze přidat jako addon MongoLab, na výběr je několik verzí, které se liší velikostí a úrovní sdílení. Verze poskytující 8GB je za $49/měsíc.

Jednotky dyno jsou jedno vláknové, proto je pro produkční prostředí třeba použít alespoň dvě dyno jednotky.

Pro dosažení požadované konfigurace je třeba využít dvě dyno verze za $106.50/měsíc. Dále je třeba přikoupit rozšíření úložiště za $20/měsíc a SSL Endpoint pro možnost používání

SSL za $20/měsíc. Náklady na počáteční provoz aplikace při použití poskytovatele Heroku tedy jsou ve výši $235.50/měsíc.

Omezení pro datové přenosy má Heroku stanoveno na 2TB/měsíc, což je dostačující.

Položka	**Počet**	**Cena/měsíc**
dyno	2	$106.50
diskový prostor	20GB	$20
SSL Endpoint	1	$20
mongolab shared cluster	1	$89
Celkem	$235.50	

Tabulka 5.2: Vhodné služby poskytovatele Heroku

Položka	**Hodnota**
paměť	1024MB
diskový prostor	20GB
cpu	2x

Tabulka 5.3: Vhodná konfigurace u poskytovatele Heroku

5.2.4 AppFog

AppFog je PaaS poskytovatel pro mnoho platforem, mezi které patří i Ruby On Rails. AppFog má plány rozdělené do několika kategorií, kde je u každé uvedena měsíční cena. Už nejlevnější nabízená služba za $20/měsíc poskytuje 2 GB RAM ale pouze měsíční datový přenos 10 GB. Ideální konfigurací z nabídky AppFog je nabídka za $100/měsíc a to hlavně z důvodu 30 GB přenosu dat za měsíc. Tato cena umožňuje běh aplikací až do 4 GB RAM, dále cena zahrnuje i 2x SSL Endpoint ale pouze 1 GB úložiště pro databáze. Proto je třeba použít poskytovatele MongoLab, čímž cena naroste o $89/měsíc.

Položka	**Počet**	**Cena/měsíc**
Konfigurace Unlimited apps within 4GB RAM	1	$100.00
mongolab shared cluster	1	$89
Celkem	$229.00	

Tabulka 5.4: Vhodné služby poskytovatele AppFog

5.2.5 AWS Elastic Beanstalk, EC2

Amazon Elastic Compute Cloud(EC2) je platforma virtuálních serverů umožňující uživatelům vytvářet a spouštět virtuální stroje. EC2 umožňují spouštět instance AMI(Amazon

Machine Images) s různými operačními systémy. Virtuální servery se mohou přidávat nebo odebírat elasticky dle potřeby.

Amazon nabízí opravdu velké množství služeb, pro nasazení aplikace jsou zajímavé následující služby:

1. EC2 Elastic Compute 2
2. Amazon Elastic Block Store
3. Elastic Beanstalk

Služby AWS(Amazon web services) lze využít jak způsobem PaaS, tak způsobem IaaS. Velmi záleží na preferencích uživatele. Opět je nabízen výběr z několika geografických umístění.

5.2.5.1 PaaS

Díky službě Elastic Beanstalk lze zdroje Amazonu používat způsobem PaaS, přitom se platí pouze za využité zdroje. Elastic Beanstalk umožňuje load balancing, ale load balancer je zpoplatněn.

5.2.5.2 IaaS

V případě použití AWS jako IaaS lze instalaci systému provést pomocí připravených AMI, do tohoto prostředí je pak možné nainstalovat databázi MongoDB, nebo využít jiných možností uvedených níže.

5.2.5.3 Konfigurace

Amazon nabízí velký výběr výkonových konfigurací, nejbližší konfigurace k požadované je: **Amazon EC2 Linux m1.small**.

Ke každé konfiguraci je nabízeno několik možností platby:

1. **On-demand** Platí se za výpočetní čas.
2. **Reserved** Servery se předplácí dle předpokládaného provozu.

Jelikož se jedná o webovou službu, u které se předpokládá provoz 24/7, tak lze využít možnost platby předplacením. Ideální je zde využít zhruba půlroční předplatné tedy medium utilization za $101, $0.020/hodina, $14.64/měsíc a po půl roce případně zvýšit výkon, nebo koupit delší předplatné.

Dále je třeba připočítat cenu za používání úložiště EBS, kam bude nainstalován systém, a přenosy dat na tomto úložišti.

Hodnoty datových přenosů pro load balancer jsou odhadnuty obdobným způsobem jako v kapitole 5.2.1.2, ale uvažují se pouze příchozí přenosy.

Hodnoty datových přenosů provedených ke službě MongoLab jsou odvozeny obdobným způsobem jako v kapitole 5.2.1.2, ale uvažují se pouze přenosy na rozhraní lo, tedy localhost.

5.2.5.4 Databáze MongoDB

Databázi MongoDB je možné získat několika způsoby:

1. **vlastní instalace** Instalace přímo do instance společně s aplikací, v tomto případě se bude využívat AWS jako IaaS.

2. **další EC2 instance** Vlastní instalace do small instance, čímž naroste cena minimálně o $14.64/měsíc, databáze bude v kategorii IaaS, aplikace pak s použitím Beanstalk bude v kategorii PaaS.

3. **mongolab** Využití služeb poskytovatele mongolab, cena $89/měsíc

Ceny byly odhadnuty pomocí Amazon Web Services Simple Monthly Calculator[3].

Položka	**Počet**	**Cena/měsíc**
Elastic Load Balancer	1	$20.50
mongolab shared cluster	1	$89
Předplatné EC2	1	$101($14.64/měsíc)
EBS úložiště	20GB	$1.10
EBS IOPS	3	$0.44
Data zpracovaná LB	1GB	$0.01
Přenos dat ze serveru	30GB	$0.16
Přenos dat pro mongolab	1GB	$0.02
Celkem za měsíc	$128.75	

Tabulka 5.5: Vhodné služby poskytovatele AWS při použití mongolab

Položka	**Počet**	**Cena/měsíc**
Elastic Load Balancer	1	$20.50
mongolab shared cluster	1	$89
Předplatné	1	$202($29.28/měsíc)
EBS úložiště	40GB	$2.20
EBS IOPS	3	$0.44
Data zpracovaná LB	1GB	$0.01
Přenos dat ze serveru	30GB	$0.16
Přenos dat pro přenos z databáze	1GB	$0.02
Celkem za měsíc	$55.49	

Tabulka 5.6: Vhodné služby poskytovatele AWS při použití vlastní instance pro databázi

Pro porovnání se službou Heroku, jedna micro instance odpovídá zhruba jednomu dyno.

[3] http://calculator.s3.amazonaws.com/

⊖	Amazon EC2 Service (Europe)		$ 137.27
	Compute:	$ 14.64	
	EBS Volumes:	$ 1.10	
	Reserved Instances (One-time Fee):	$ 101.00	
	Elastic LBs:	$ 20.50	
	Data Processed by Elastic LBs:	$ 0.01	
	Inter-Region Data Transfer Out	$ 0.02	
⊕	AWS Data Transfer Out		$ 3.48
⊕	AWS Support (Basic)		$ 0.00
Free Tier Discount:		**$**	-23.28
Total One-Time Payment:		**$**	101.00
Total Monthly Payment:		**$**	16.47

Obrázek 5.2: Snímek Amazon Web Services Simple Monthly Calculator - databáze mongolab

⊖	Amazon EC2 Service (Europe)		$ 254.01
	Compute:	$ 29.28	
	EBS Volumes:	$ 2.20	
	Reserved Instances (One-time Fee):	$ 202.00	
	Elastic LBs:	$ 20.50	
	Data Processed by Elastic LBs:	$ 0.01	
	Inter-Region Data Transfer Out	$ 0.02	
⊕	AWS Data Transfer Out		$ 3.48
⊕	AWS Support (Basic)		$ 0.00
Free Tier Discount:		**$**	-23.83
Total One-Time Payment:		**$**	202.00
Total Monthly Payment:		**$**	31.66

Obrázek 5.3: Snímek Amazon Web Services Simple Monthly Calculator - vlastní databáze

Položka	**Hodnota**
paměť	1700MB
diskový prostor	20GB
cpu	1x

Tabulka 5.7: Vhodná konfigurace u poskytovatele AWS

5.2.6 DigitalOcean

DigitalOcean je zajímavý především díky velmi zajímavým cenám vzhledem ke službám, které nabízí. Nabízí několik konfigurací serverů, uživatel má možnost instalace operačního systému dle výběru. Navíc je možné si vybrat instalaci systému s prostředím nastaveným pro Ruby On Rails aplikace, při výběru této možnosti, pak pro nasazení vyvíjené aplikace stačí pouze doinstalovat MongoDB databázi.

Položka	Počet	Cena/měsíc
DigitalOcean $20/month	1	$20.00
Celkem		$20.00

Tabulka 5.8: Vhodné služby poskytovatele DigitalOcean

Položka	Hodnota
paměť	2048MB
diskový prostor	40GB
cpu	2x

Tabulka 5.9: Vhodná konfigurace u poskytovatele DigitalOcean

5.2.7 Shrnutí

Dle předpokladů jsou mnohem dražší služby poskytovatelů PaaS řešení než poskytovatelů IaaS řešení. Výběr z těchto možností záleží na preferencích uživatele, pokud jsou dostupné vhodné lidské zdroje, pak je v rámci finančních úspor na místě použití IaaS. Největší komfort a jednoduchost při nasazování Ruby On Rails aplikací nabízí poskytovatel Heroku, pokud se příliš nehledí na finanční stránku, tak může být aplikace nasazena velmi snadno a na velmi kvalitní infrastruktuře.

Při výběru poskytovatele cloudových služeb je třeba brát v úvahu i bezpečnostní rizika související s ukládanými daty, jedná-li se o citlivá data, pak je určitě vhodné vybírat ze společností, které mají s danou problematikou dlouhodobou zkušenost. Poskytovaný hardware může být využíván i cizí osobou, například pokud se vytvoří nový server a starý se smaže, úložiště kde byl starý server nainstalován může používat další osoba a v případě, že není toto úložiště vhodně vymazáno, může se daná osoba dostat k citlivým datům, která tam byla uložena.[4].

Z hlediska možností vede jednoznačně AWS, které nabízí mnoho možností přizpůsobení, dobré služby zdarma, nejlepší možnosti škálování a několik způsobů placení za služby. O kvalitách AWS svědčí i fakt, že sami vybraní poskytovatelé PaaS využívají služeb AWS. Množství možností má ale i své stinné stránky. Díky velkému množství možností není jasné, co všechno bude při provozu daného serveru zaúčtováno a přesnější informace o vyúčtování lze vyvozovat až po jednom měsíci provozu, kdy k vyúčtování dojde.

Pokud se uvažují pouze nízké finanční zdroje, pak je vhodný poskytovatel DigitalOcean, ale je třeba brát v úvahu, že se jedná pouze o IaaS poskytovatele a možností je mnohem méně než v případě AWS.

[4] http://venturebeat.com/2013/12/30/iaas-provider-digitalocean-finds-itself-back-in-security-trouble/

Kapitola 6

Realizace

Kapitola popisuje kroky provedené při realizaci řešeného problému pomocí oddělení dat v rámci jedné databáze. Nejprve je popsán framework Ruby On Rails, dále kroky provedené pro nastavení přihlášeného tenanta, oddělení dat a realizace rozhraní pro údržbu, které slouží pro správu globálních údajů, kterými jsou státy, sazby DPH, výrobci automobilů, modely automobilů a nakonec klienti a jejich statistiky.

6.1 Ruby On Rails

Ruby On Rails je webový MVC framework napsaný v jazyce Ruby. Je navržen tak, aby usnadnil vývoj aplikací používáním konvencí. Díky tomuto přístupu se redukuje množství kódu a díky tomu je kód mnohem přehlednější.

Rails staví na dvou základních principech:

DRY Don't repeat yourself, zásadou tohoto principu je vyhnutí se duplikaci kódu.

Convention Over Configuration Tento princip je založen na preferování dané konvence před vlastním nastavením, Rails nenabízí tolik možností konfigurace, vždy existuje jeden správný způsob, kterým se s použitím Rails daná problematika řeší, díky tomu je velmi snadné se v Rails projektech orientovat, vše je tam, kde to vývojář předpokládá. To se týká nejen umístění konfiguračních souborů, struktury projektu ale i způsobu pojmenování modelů, views, kontrolerů atd.

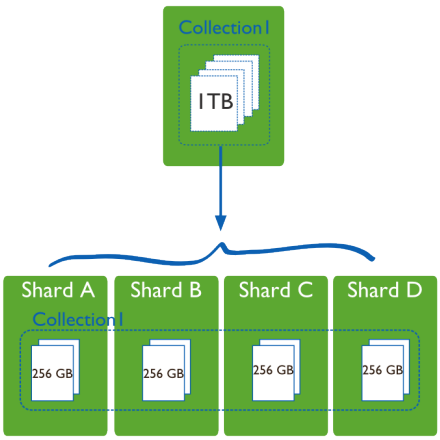
Rails nabízí spoustu nástrojů pro usnadnění a urychlení práce. Jedním z těchto nástrojů je aplikace rails, která umožňuje generování struktury projektu, spuštění webového serveru, generování jednotlivých částí MVC.

Závislosti jsou v Rails spravovány nástrojem Bundler. Závislosti jsou definovány v souboru Gemfile, kde jsou definovány jednotlivá rozšíření, tato rozšíření se nazývají jako gemy a lze je chápat jako knihovny, které do projektu přidávají funkcionalitu. V Gemfile se určuje název gemu, verze, případně umístění v repozitáři nebo adresáři. Bundler umožňuje snadnou instalaci těchto gemů pomocí příkazu bundle install. [6]

6.2 MongoDB

MongoDB je dokumentová databáze poskytující vysoký výkon a snadné škálování. Na rozdíl od běžných relačních databází používá k ukládání dat dokumenty. Výhodou MongoDB oproti relačním databázím z pohledu vývoje je, že se nemusí definovat schémata uložení dat, tak jak je v relačních databázích třeba definovat tabulky. Dokument se vytvoří a jeho položky mohou být libovolné. Tento přístup má, ale i nevýhodu v obtížnějším dotazování nad daty. MongoDB spadá do kategorie databází NoSQL. Dotazování nad daty je prováděno pomocí objektově orientovaného API. Je možné v rámci jednoho dokumentu ukládat další dokumenty, tím lze docílit vyšší rychlosti oproti relačním databázím, kde jsou jednotlivé tabulky propojeny pomocí klíčů a data je třeba skrze relace dohledávat. [2]

Zásadní výhodou MongoDB je možnost snadného škálování, to je pro SaaS aplikaci velmi vhodné. V případě rostoucího počtu tenantů se zvýší nároky na databázi a databázi je třeba škálovat. MongoDB podporuje tzv. sharding, díky kterému je možné škálovat databázi horizontálně, nezvyšuje se výkon daného serveru, ale přidá se další server. Data jsou rozdělena na tzv. shardy, každý shard je nezávislá databáze a dohromady tvoří jednu databázi. [3]



Obrázek 6.1: Diagram znázorňující distribuci dat na 4 shardy

MongoDB ukládá data v dokumentech, které jsou ve formátu JSON(Javascript Object Notation).

```
1{ "item": "pencil", "qty": 500, "type": "no.2" }
```

Typy atributů mohou být z množiny BSON Types, do které spadají například Double, String, Boolean, Date, Array, Object type a další.

Na disku pak MongoDB ukládá data ve formátu BSON, což je binární reprezentace formátu JSON. [1]

6.3 ActiveRecord a Mongoid

6.3.1 ActiveRecord

ActiveRecord je výchozí datovou vrstvou pro framework Ruby On Rails. ActiveRecord je implementace návrhového vzoru Active Record. Dle tohoto vzoru objekty obsahují persistentní data i logiku, která s daty pracuje. ActiveRecord používá objektově relační mapování, což je technika, která mapuje data z tabulek relačních databází do objektů používaných v aplikaci. ActiveRecord je tedy ORM frameworkem. Umožňuje:

- Reprezentování modelů a jejich dat
- Reprezentovat asociace mezi objekty
- Reprezentovat dědičnosti mezi objekty
- Validaci modelů
- Provádět databázové operace pomocí objektového rozhraní

Typický model ActiveRecord je potomkem třídy ActiveRecord::Base, což umožňuje nad ním provádět databázové operace.

```
1class Product < ActiveRecord::Base
2end
```

Nad instancí třídy Product je možné provádět CRUD operace:

Create

product.save
: provede uložení existujícího objektu do databáze

Product.create
: provede vytvoření objektu s danými parametry a následně uložení do databáze

Read

Product.all
: získá z databáze všechny objekty Product

Product.first
: získá první objekt třídy Product dle primárního klíče

Product.find
: nalezne objekty Product dle zadané hodnoty id

Product.where
: nalezne objekty Product dle zadaných parametrů

Update

product.update
provede update daného objektu product

Product.update
vstupními parametry jsou hodnota id objektu Product, který má být změněn a dále Hash s hodnotami, na které má být změněn

Delete

product.delete
provede odstranění objektu product z databáze

Product.delete
vstupním parametrem je id objektu product, který má být vymazán

V novější verzích se pro vyhodnocení používá líné vyhodnocování, takže například Product.all se vyhodnotí až v případě použití. [6]

6.3.2 Mongoid

Pro framework Ruby On Rails existují dva hlavní gemy, které zajišťují práci s databází MongoDB, jsou jimi Mongoid a MongoMapper. Stávající Aplikace Autoservis využívá první jmenovaný, tedy Mongoid.

Mongoid je objektově dokumentový mapper(ODM), což je obdoba klasického objektově relačního mapování(ORM) v relačních databázích. Úlohou gemu Mongoid je poskytnout obdobné API pro práci s daty v databázi MongoDB, které nabízí ActiveRecord pro relační databáze. Funkční rozhraní gemu Mongoid je velmi podobné jako rozhraní ActiveRecord popsané výše.

```
1class Artist
2  include Mongoid::Document
3  field :name, type: String
4  embeds_many :instruments
5end
```

Před použitím je třeba rozšířit třídu o funkcionalitu Mongoid::Document a dále je možné definovat atributy dané entity pomocí funkce field a dalších. Relace je možné definovat dvěma způsoby. [5]

embedded Entity jsou vedeny v rámci jednoho dokumentu, výhodou tohoto přístupu je, že je není třeba dále dohledávat. Nevýhodou je pak nemožnost získat entitu samostatně, vždy je třeba se k entitě dostat přes jejího rodiče.

referenced Každá entita je vedena ve svém dokumentu a relace je uskutečněna pomocí jejich id.

6.4 Multitenance

Celý proces realizace multitenance lze rozdělit na dvě části:

1. nastavení tenanta v kontroleru
2. oddělení dat v modelu

Nastavením tenanta se rozumí zjištění o jakého tenanta v rámci systému se jedná a dále nastavení této hodnoty pro vykonání daného požadavku. Nastavení tenanta se provádí pomocí subdomény, pokud uživatel přistoupí na subdoménu jemu přidělenou, aplikace podle této subdomény nalezne tenanta a nastaví ho. Oddělení dat v modelu pak zajistí, že z databáze budou získávána pouze data právě přihlášeného tenanta.

6.4.1 Výsledný datový model

Datový model uvedený v 4.1 bylo třeba upravit, aby bylo možné v systému evidovat jednotlivé instance. Do modelu byla přidána entita Client, která reprezentuje tenanta. Vzhledem k rozsahu není výsledný diagram uveden, jedná se o stejný diagram s tím rozdílem, že je rozšířen o entitu Client, která má vazbu na všechny nespolečné entity uvedené v 4.1.2.

6.5 Oddělení dat v modelu

Dle vybraného způsobu oddělení dat budou nesdílená data oddělena pomocí parametru id. K evidování jednotlivých id byl zaveden model Client, který reprezentuje jednoho tenanta. Veškerá nespolečná data spadají do jedné instance Client. Jednotlivá data budou tedy oddělena parametrem client_id. Tento parametr je třeba zavést do všech modelů, které mají nesdílená data. Dále je třeba rozšířit modely tak, aby používaly pouze data nastaveného tenanta.

Realizace těchto problémů byla docílena pomocí gemu mongoid-multitenancy. Mongoid-multitenancy řeší daný problém přímo využitím částí frameworku Ruby On Rails a dále gemu mongoid. Danou problematiku řeší pomocí scopes popsaných v sekci 4.4 a dále zavádí dodatečnou funkcionalitu jako například přidání indexů k daným id parametrům a překrytí validačních metod, tak aby byla data validována pouze v rámci daného tenanta.

Tento gem nebyl kompatibilní s verzemi použitými v aplikaci, proto musel být upraven.

6.6 Pry

Úprava gemu mongoid-multitenancy byla provedena za pomoci nástroje pry. [1] Pry je nástroj typu REPL(Read–Eval–Print loop), je alternativou k výchozímu irb(Interactive Ruby Shell), což je příkazová řádka pro jazyk Ruby.

Pry umožňuje pomocí umístění příkazu binding.pry do kódu zastavit proces provádění v daném místě a prozkoumat stav aplikace, ve kterém bylo provádění zastaveno. Díky tomu lze poměrně rychle zjistit, jakým způsobem daný kód pracuje a vyzkoušet případné změny přímo v daném místě.

[1] http://pryrepl.org/

6.7 Mongoid-multitenancy

Mongoid multitenancy je napsán pro mongoid verze 3 a výše, Aplikace Autoservis, ale používá mongoid verze 2, která používá jinou formu kritérií v příkazech where, tato místa byla přepsána tak, aby byla pracovala s mongoid verzí 2.

Dále bylo zapotřebí upravit samotné využití scopes, mongoid multitenancy spoléhá na využití scopes dle nastavení default scope, kde je využita lambda funkce, která filtruje data na základě aktuálně nastavené globální proměnné Mongoid::Multitenancy.current_tenant. S použitím mongoid verze 2 se ale ona lambda funkce vyhodnotí pouze jednou a dochází k zachování hodnoty proměnné v okamžiku volání, tento problém byl vyřešen pomocí překrytí funkce criteria, ve které se nyní nastavuje scope s aktuálním Multitenancy.current_tenant.id.

```
1 def criteria(embedded = false, scoped = true)
2   if scoped
3     crit = Multitenancy.current_tenant ? {
4       :where => { tenant_field.to_sym => Multitenancy.current_tenant.id }} : {}
5     default_scope(crit)
6   end
7   super(embedded, scoped)
8 end
```

Následné použití mongoid-multitenancy je velmi snadné, do každého modelu s nesdílenými daty stačí přidat následující 2 řádky:

```
1 include Mongoid::Multitenancy::Document
2 tenant(:client)
```

První řádek znamená použití modulu. Moduly v jazyce Ruby definují metody, které je možné dále použít ve třídách, po provedení příkazu include jsou metody definované v modulu součástí dané třídy, případně instancí dané třídy.

Druhý řádek zavolá funkci s tenant parametrem, který určuje název modelu realizujícího multitenanci.

Funkce tenant, jak již bylo zmíněno výše, tedy zavede do daného modelu parametr client_id, který bude odlišovat jednotlivá data. Dále pro daný model nastaví default_scope tak, aby byly použity pouze záznamy s hodnotou client_id rovné MongoidMultitenancy::current_tenant. Dále provede zavedení indexu na daný parametr client_id, což zajistí optimální výkon při použití scopes dle daného id.

6.8 Nastavení tenant id

V předchozí části je popsáno, jakým způsobem jsou jednotlivá nespolečná data oddělena. Přihlášenému uživateli je třeba zobrazit pouze data, která mají shodné client_id s tenantem,

pod kterého daný uživatel spadá. Tedy nastavit proměnnou nesoucí id aktuálního tenanta. Použitý gem Mongoid Multitenancy pro tento účel používá globální proměnnou Mongoid-Multitenancy::current_tenant. K nastavení této proměnné dochází před zobrazením každé stránky při vyhledání, instance třídy Client dle subdomény. Po nastavení této proměnné dochází k využití scopes dle daného id a je tedy přístup pouze k těmto datům.

Nastavení tenanta je provedeno v before filteru u ApplicationController, což je předek všech kontrolerů v rámci Aplikace Autoservis, proto použití scopes tímto způsobem ovlivňuje všechny kontrolery a není třeba jej dále v aplikaci řešit, pouze v administračním rozhraní, kde se použití scopes vypíná tak, aby bylo možné pracovat se všemi daty.

Dále bylo třeba vyřešit situaci, kdy se uživatel snaží přihlásit v rámci hlavní domény, v takovém případě nelze uživatele přihlásit, protože není znám rámec ve kterém se uživatel chce přihlásit. Proto je v tomto případě zobrazen formulář pro zadání subdomény, který uživatele přesměruje na přihlášení v rámci dané subdomény.

6.9 Použití subdomén

Subdoména je v tomto případě uvažována jako nižší úroveň domény o jeden stupeň, než je hlavní doména aplikace. Jedná se o část adresy, která je od hlavní domény oddělena tečkou. Díky tomu mohou mít všechny instance aplikace vlastní doménu a je možné pro každou instanci zobrazovat specifickou úvodní stranu.

Pro fungování subdomén je třeba správně nastavit DNS A záznamy tak, aby nejen hlavní doména odkazovala na ip serveru ale také každá subdoména. Toho se zpravidla docílí vytvořením DNS A záznamu začínajícím hvězdičkou a končícím názvem hlavní domény začínající tečkou. Nicméně toto nastavení slouží pro nasazení na serveru, pro lokální ověření funkčnosti je třeba použít jiný postup. Je několik možností, jak toho docílit, pro unixové systémy je například možné upravení souboru /etc/hosts, ale nejsnazší možností je použití některé z veřejných domén, které odkazují na localhost tedy na ip adresu 127.0.0.1. Díky tomu je možné požadavky na lokální server směřovat pomocí některé z těchto domén. Takovými doménami jsou například lvh.me, localtest.me.

Pokud máme tedy lokální server pro rails spuštěný na portu 3000, pak je možné se na něj odkázat pomocí adresy http://lvh.me:3000, v případě použití subdomény http://subdomena.lvh.me:3000.

Ruby On Rails umožňuje sdílení sessions v rámci jedné domény, ale při takovém nastavení docházelo k problémům, proto bylo zavedeno takové nastavení, že každá subdoména má svou vlastní session.

Doména se eviduje v přidané entitě Client. ApplicationController je rozšířen o before filter callback, který zjistí aktuální subdoménu a podle ní pak vyhledává instanci třídy Client.

```
1....
2
3  before_filter :clear_tenant, :check_subdomain
4
5  protected
6
7  def clear_tenant
8    Mongoid::Multitenancy.current_tenant = nil
9  end
10
11 def set_tenant_by_subdomain
12   Mongoid::Multitenancy.current_tenant = @subdomain_client
13 end
14
15 def check_subdomain
16   if main_subdomain?
17     if devise_controller?
18       redirect_to(set_subdomain_index_url, subdomain: MAIN_SUBDOMAIN)
19     end
20     return
21   end
22
23   Mongoid::Multitenancy.with_tenant(nil) {
24     @subdomain_client = Client.find_by_subdomain(request.subdomain)
25     if @subdomain_client.nil?
26       redirect_to(root_url(subdomain: MAIN_SUBDOMAIN))
27       return
28     end
29   }
30   if !@subdomain_client.nil?
31     set_tenant_by_subdomain
32   end
33 end
34
35 def main_subdomain?
36   subdomain = request.subdomain
37   subdomain.nil? || subdomain.empty? || subdomain == MAIN_SUBDOMAIN
38 end
39....
40end
```

Gem Mongoid Multitenancy nabízí funkci Mongoid::Multitenancy.with_tenant(tenant), která umožňuje nastavení tenanta na námi požadovanou hodnotu, to je vhodné pokud je třeba vybrat data v rámci všech tenantů, v takovém případě se jako tenant použije hodnota nil.

6.10 Přihlašování a multitenance

Přihlášení je ve stávající Aplikaci Autoservis realizováno pomocí gemu Devise. Ve frameworku Ruby On Rails je zvykem používat jako předka pro všechny kontrolery ApplicationController a právě použitý gem Devise přidává do tohoto kontroleru metody pro spravování případně získávání informací o přihlášení, např.: authenticate_user!, user_signed_in?, current_user apod.

Tyto metody nebylo nutné nijak upravovat, k zavedení scopování totiž dochází před provedením těchto metod a proto je jejich vykonání provedeno na datech právě nastaveného tenanta.

6.11 Přidání modelů automobilů

Dle požadavků bylo dále třeba rozšířit aplikaci o správu výrobců a modelů automobilů, tak aby si uživatel mohl vybrat z množiny definovaných.

Výběr výrobců je realizován stejně jako v původní aplikaci pomocí listu s možností výběru výrobce. Výběr modelu je libovolný, proto je řešen textovým polem, které v průběhu psaní uživateli nabízí již předem definované možnosti nalezené pomocí obsahu, který do pole zadal. Toto vyhledávací pole je realizováno pomocí funkce autocomplete javascriptové knihovny JQuery.

Pro zavedení této funkčnosti bylo třeba aplikaci rozšířit o model pro evidování výrobců a model pro evidování modelů automobilů.

6.11.1 JQuery autocomplete

```
1  var manufacturer_id = "#contract_car_attributes_manufacturer";
2  $(manufacturer_id).change(function(){
3    $("#contract_car_attributes_commercial_description").autocomplete({
4      source: "/car_models/?vendor_id=" + $(manufacturer_id).val(),
5      focus: function( event, ui ) {
6        $( this ).val( ui.item.label );
7        return false;
8      },
9      select: function( event, ui ) {
10       $( this ).val( ui.item.label );
11       return false;
12     }
13   });
14 });
```

Výše uvedený kód při změně výrobce nastaví funkci autocomplete s parametrem adresy, kam se posílají AJAX požadavky, které jsou následně zpracovány a v případě, že byly nalezeny položky, tak jsou předány daným funkcím focus a select.

6.11.2 Mongoid-fulltextable

Aplikace Autoservis využívá pro vyhledávání v modelech gem Mongoid-fulltextable, gem byl použit pro vyhledání typu vozidla po přijetí AJAX požadavku.

Tento gem umožňuje snadné rozšíření instance Mongoid::Document pomocí příkazu fulltextable o metodu fulltext, která umožňuje vyhledávání v definovaných atributech.

6.12 Rozšíření administračního rozhraní a registrační formulář

Administrační rozhraní bylo rozšířeno o rozhraní pro správu jednotlivých tenantů, je možné zobrazit všechny tenanty v rámci aplikace, upravovat jejich údaje, případně spravovat uživatele vedené pro daný autoservis.

Dále bylo přidáno rozhraní pro správu DPH, které umožňuje upravovat data jednotlivých států, kterými jsou měna a kód státu. Dále je možné upravovat definovaná schémata pro DPH, kde každé schéma obsahuje počáteční a koncové datum platnosti daného schématu. Ke každému schématu je možné definovat hodnotu DPH, tyto hodnoty se rozlišují dle typu dané daně.

6.12.1 HAML

Jednotlivé stránky jsou ve stávající Aplikaci Autoservis definovány pomocí jazyka HAML, tento jazyk byl použit i při definování přidaných rozhraní.

HAML je jazyk, který vyjadřuje jazyk HTML mnohem snazším zápisem. Eliminuje ukončovací tagy tím, že definuje zanoření pomocí odsazení. Díky tomu je kód mnohem přehlednější, rychleji se vytváří, případně edituje.

6.12.2 Simple forms

Formuláře jsou ve stávající Aplikaci Autoservis definovány pomocí gemu sipmle forms, který umožňuje snazší zápis formulářů než standardní funkce pro formuláře ve frameworku Ruby On Rails.

Další vlastností tohoto gemu je, že ve výchozím nastavení automaticky zjišťuje typ vstupu, například pro heslo tedy není třeba definovat typ vstupního pole. Díky tomu dochází k zpřehlednění kódu. [7]

Tento gem byl použit při tvorbě formulářů v přidaných rozhraní.

```
1= form.input :password
```

Ukázka použití HAML se Simple forms pro registrační formulář:

```
%h1= page_title

%br

= render :partial => "form",
         :locals => {:submit_title => t(".submit", :scope => "views")}
```

```
= simple_form_for [:front, @user],
    :html => {novalidate: :novalidate, class: "well"} do |form|
  %h2=t(".subtitle1", :scope => :views)
  = form.input :name
  = form.input :email
  = form.input :password
  = form.input :password_confirmation
  %h2=t(".subtitle2", :scope => :views)
  = form.simple_fields_for @user.client do |client_form|
    = client_form.input :name
    = client_form.input :subdomain
    .notes
      = client_form.input :description, :as => :text,
        input_html: {class: "span6"}
  .form-actions
    = form.submit submit_title, class: "btn btn-success"
```

6.12.3 Responders

Stávající Aplikace Autoservis využívá pro zjednodušení kódu kontrolerů gem Reponders. Vše je založeno na funkci respond_with, které se předá proměnná, jejíž hodnota určí, jaká bude odpověď na požadavek. Tímto způsobem se eliminují podmínky, které jsou často pro každou metodu kontroleru stejné. Existuje několik druhů responders, v aplikaci se používá flash reponder, který na základě vyhodnocení proměnné předané funkci respond_with, vybere z konfiguračního souboru odpovídající flash zprávu. Flash zpráva je zpráva zobrazující uživateli informaci a prováděné akci, například v případě vytváření nového záznamu zobrazuje informaci o úspěšném, případně neúspěšném provedení této akce.

Responders umožňují definování generických flash zpráv, kterých je využito v administraci, kde není nezbytné definovat specifické zprávy pro každou akci.

6.12.4 Will paginate

Pro stránkování využívá stávající Aplikace Autoservis gem Will paginate, který rozšiřuje instance třídy Active Record, které se používají pro definování entit, o možnost stránkování. Každému potomkovi třídy Active Record přidá funkci paginate, která má dva vstupní

parametry. První parametr určuje stranu, druhý parametr počet stran na jednu stránku. Stránkování se pak generuje ve view pomocí funkce will_paginate s parametrem instance s daty, která byla vrácena z volání funkce paginate.

Ukázka kontroleru a využití responders, will paginate Následující kód je ukázkou elegantnosti jazyka Ruby ve spojení s výbornými gemy, celý controller lze vytvořit několika řádky. Takto snadný zápis umožňuje snadnou udržovatelnost kód, nutno ovšem dodat že tento kód využívá generické flash message.

```
class Maintenance::CarVendorsController < Maintenance::MaintenanceController
  respond_to :html

  before_filter :find_car_vendor

  def index
    @car_vendors = CarVendor.paginate :page => params[:page],
                                      :per_page => per_page
    respond_with(@car_vendors)
  end

  def show
    respond_with(@car_vendor)
  end

  def new
    @car_vendor = CarVendor.new
    respond_with(@car_vendor)
  end

  def edit
    respond_with(@car_vendor)
  end

  def create
    @car_vendor = CarVendor.new(params[:car_vendor])

    @car_vendor.save
    respond_with(@car_vendor, location: maintenance_car_vendors_url)
  end

  def update
    @car_vendor.update_attributes(params[:car_vendor])
    respond_with(@car_vendor, location: maintenance_car_vendors_url)
  end

  def destroy
```

```
38    @car_vendor.destroy
39    respond_with(@car_vendor, location: maintenance_car_vendors_url)
40  end
41
42  private
43
44  def find_car_vendor
45    @car_vendor = CarVendor.find(params[:id]) unless params[:id].nil?
46  end
47
48  def per_page
49    25
50  end
51
52end
```

6.12.5 Nastavení rout

Routami se v tomto případě rozumí URL, která jsou přidělena různým kontrolerům a akcím. V Ruby On Rails se routy ukládají do souboru config/routes.rb, pro jejich definování se také používá DSL. Díky příkazu resources je možné definovat routy pro celý kontroler. URL je vytvořena z názvu kontroleru. V Ruby On Rails je konvencí definovat metody kontrolerů jako REST. V kontroleru jsou tedy metody index, show, new, create, edit, update, destroy. REST mapování je následující:

Create create, metoda new slouží pro zobrazení formuláře vytvoření

Read show

Update update, metoda edit slouží pro zobrazení formuláře editace

Delete destroy

Přidané routy pro administrační rozhraní:

```
1  namespace :maintenance do
2    resources :countries do
3      resources :vat_schemes do
4        resources :vat_rates
5      end
6    end
7    resources :clients do
8      resources :users
9    end
10   resources :car_vendors do
11     resources :car_models
```

```
12    end
13    resources :car_models
14    resources :users
15    resources :stats
16  end
```

Jsou využity tzv. nesting routes, zanořené routy. Ty umožňují zanořování rout, toho je využito u kontrolerů spravujících entity, které jsou rodiči jiných entit. Příklad URL pro přidání uživatele ke klientovi:

```
1maintenance/clients/5313492aa81aeb790700001c/users/new
```

Získávání id v kontroleru je pak opět dle konvencí, id dané entity, tedy poslední id v url je dostupné v Hashi params[:id] a všechny ostatní dle názvu kontroleru v jednotném čísle, tedy pro clients je id dostupné pod params[:client_id].

6.12.6 Lokalizace

Aplikace Autoservis má view lokalizovány, lokalizace je provedena pomocí standardních funkcí frameworku Ruby On Rails. Aplikace používá několik souborů s lokalizacemi, jsou rozděleny dle účelu, pro který jsou lokalizace použity.

views obsahuje lokalizace views

activemodel obsahuje lokalizace názvů entit a jejich parametrů, toho se využívá například při výpisu tabulek

flash obsahuje lokalizované zprávy o stavu

helpers obsahuje lokalizované texty používané v helperech

rails obsahuje standardní překlady frameworku Ruby On Rails, například lokalizaci měsíců a dnů

Soubory pro dané jazyky jsou odlišeny pomocí prefixu s kódem jazyka, výsledný název souboru pak vypadá například cs.views.yml. Lokalizace jsou uloženy ve formátu yml.

Ukázka použití lokalizace ve view:

```
1%td= link_to t('common.detail', :scope => :views)
```

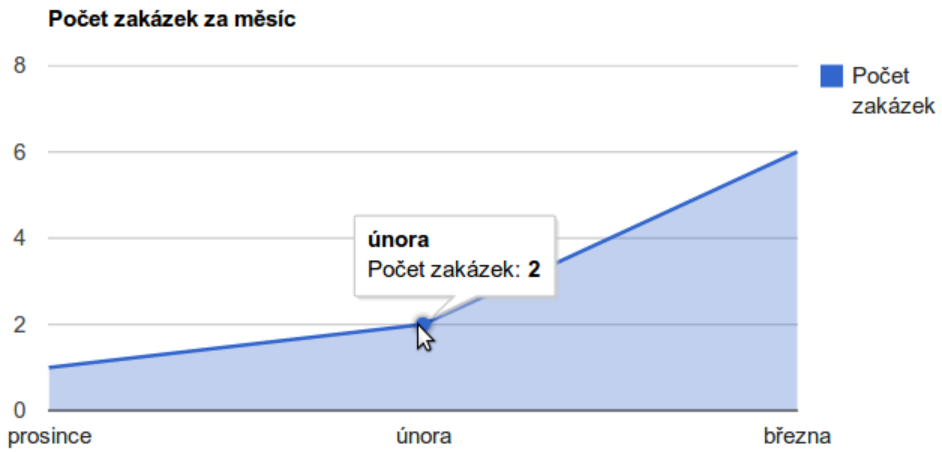
Lokalizovaný výraz je získán pomocí funkce t, která jako první parametr používá identifikátor, v druhém parametru se udává dodatečné nastavení, v uvedeném případě se jedná o soubor s lokalizací, který je použit.

6.13 Zobrazení statistik

Statistiky v administračním rozhraní jsou zobrazeny grafem a znázorňují počty registrací, případně počty zakázek, v daném kalendářním měsíci za poslední rok. Grafy jsou zobrazeny pomocí Google Visualization API. Pro použití tohoto API byl do aplikace přidán gem google_visualr.

Pro zobrazení byla vytvořena funkce month_stat_chart, která přijímá parametr s třídou, pro kterou mají být zobrazeny statistiky a dále popis dat. Zobrazení pak probíhá následovně:

```
1= month_stat_chart(Client,
2  t('.registrations_per_month', :scope => :views),
3  t('.month', :scope => :views),
4  t('.registrations_count', :scope => :views))
```



Obrázek 6.2: Zobrazení statistik zakázek

6.14 Capistrano

Pro nasazení využívá Aplikace Autoservis nástroj Capistrano, který byl použit při nasazování na testovací server. Tento nástroj umožňuje automatizaci nasazení. Pro každé prostředí je možné definovat specifické kroky prováděné při nasazování aplikace, tedy například předkompilaci assetů, úpravu konfiguračních souborů, restartování služeb apod. Prováděné akce je možné provádět v různých fázích průběhu nasazení. Konfigurační soubory, které se pro Capistrano používají se nazývají recipes a využívají specifický jazyk DSL. Kromě akcí prováděných při nasazování aplikace je dále možné definovat vlastní libovolné akce, například restartování serveru apod.

Nastavení pro Capistrano je umístěno v souboru config/deploy.rb a nastavení specifická pro daná prostředí jsou uložena v config/deploy/nazev_prostredi, kde nazev_prostredi je název daného prostředí, tedy například production.

Nasazení na produkční prostředí je pak možné provést příkazem v konzoli:

```
1cap production deploy
```

Kapitola 7

Testování

Jelikož je Ruby dynamicky typovaný jazyk, není možné provádět statickou kontrolu typování kódu, proto je kód náchylnější ke vzniku chyb a měl by být více testován, než staticky typovaný jazyk, který chyby typování odhalí již při kompilaci.

Aplikace Autoservis pro testování používá framework RSpec. RSpec používá DSL, který umožňuje snazší a rychlejší tvorbu testů. Jeho vznik byl podnícen potřebou pro behavioral driven development a test driven development.

RSpec definuje testy pomocí bloků describe a it, v bloku describe se popisuje testovací případ a v blocích it, které jsou vnořeny v blocích describe, se určuje jaké vlastnosti by měl daný případ splňovat.

RSpec umožňuje tvorbu jednotkových testů a také integračních testů. Framework dále obsahuje funkce pro tvorbu snadno čitelných testů, kterými jsou například should a should_not.

Původní verze verze Aplikace Autoservis obsahovala 145 testů, ke kterým bylo přidáno dalších 110 testů pro ověření přidaného kódu.

7.1 FactoryGirl

Při testování jsou potřebná testovací data, v Aplikaci Autoservis jsou tyto data vytvářena pomocí tzv. továren. Factory girl je gem sloužící k zjednodušenému vytváření těchto továren. Továrny slouží pro vytváření instancí entit, tedy potomků třídy Active Record, v případě Aplikace Autoservis se jedná o objekty Mongoid::Document. Továrny vytváří instance volitelnými způsoby, je například možné vytvořit pouze instanci bez uložení do databáze, nebo je možné vytvořený objekt ihned uložit.

Aby nebyla všechna data stejná, například pro uživatele, je možné definovat sekvence, které jednotlivé instance libovolně odliší.

Pro přidané testy bylo třeba doplnit několik těchto továren.

Ukázka továrny client s použitím sekvencí:

```
1 ##############################################################################
2 # Client
3
4 Factory.define :client do |client|
5   client.sequence(:name)       {|n| "Client #{n}" }
6   client.sequence(:subdomain)  {|n| "client#{n}" }
7 end
```

7.2 Travis CI

Stávající Aplikace Autoservis využívá nástroj pro kontinuální integraci Travis CI. Jedná se o postup, kdy se průběžně spojují vývojové kopie sloužící ke snížení počtu problémů při integraci.

Travis CI spouští testy po každém provedení příkazu push do git repozitáře a tím mají vývojáři přehled o tom, zda jejich úpravy nezpůsobily problémy.

Nastavení se provádí pomocí souboru .travis.yml, který je ve formátu yml a je umístěn v kořenovém adresáři repozitáře s aplikací. [8] V souboru se definuje:

- Programovací jazyk, který aplikace používá.
- Jaké příkazy mají být provedeny před každým sestavením.
- Jakým příkazem se spouští testy aplikace.
- Email, případně další způsoby pro informování o selháních.

Pro Aplikaci Autoservis je v tomto souboru definováno nastavení konfiguračního souboru pro nastavení testovací databáze a dále spuštění testů.

Aplikace Autoservis využívá magnum.travis-ci.com, což je služba poskytující Travis CI přímo na github.com, kde je veden repozitář aplikace.

7.3 Úprava stávajících testů

Stávající testy přestaly po úpravě na multitenantní verzi fungovat, protože počítaly pouze s jednou instancí. Testy bylo samozřejmě potřeba zachovat, ale takovým způsobem, aby byly prováděny v rámci jedné instance.

Před každým testem bylo tedy třeba provést nastavení tenanta, díky tomu se test provede pro tenanta, který byl nastaven.

Framework RSpec umožňuje definovat callback, který je proveden před každým testovacím případem:

```
config.before(:each) do
  DatabaseCleaner.clean
  client = Client.first || Factory.create(:client)
  Mongoid::Multitenancy.current_tenant = client
  begin
    host! "#{client.subdomain}.example.com"
  rescue
  end
end
```

Tento přidaný kód zajistí, že pokud neexistuje žádný tenant, tak dojde k jeho vytvoření a následně nastavení aktuálního tenanta na vytvořeného, případně na tenanta, který již byl v databázi přítomný.

V případě integračních testů je dále třeba nastavit subdoménu, v rámci které jsou testy prováděny. Toho je docíleno pomocí funkce host!, funkce je v bloku pro zachycení výjimky, protože tento kód se provádí před všemi testy, jednotkovými i integračními, a v jednotkových testech není metoda host! přítomna.

7.4 Jednotkové testy

Hlavním cílem testování přidaného kódu bylo ověření oddělení dat. Oddělení bylo otestováno pro všechny modely jednotkovými testy, dále byly otestovány všechny kontrolery integračními testy.

Přidaný kód byl otestován tak, aby bylo ověřeno oddělení dat pro všechny modely. Příklad testu pro model Contract:

```
describe "check contract multitenancy" do
  let(:client1)            { Factory.create :client }
  let(:contract1)   {
                      Mongoid::Multitenancy.current_tenant = client1
                      Factory.create :contract
  }
  let(:client2)     { Factory.create :client }
  let(:contract2)   {
                      Mongoid::Multitenancy.current_tenant = client2
                      Factory.create :contract
  }

  before do
    client1.save
    client2.save
    contract1.save
```

```
17      contract2.save
18    end
19
20    it "contract1 should belong to client1" do
21      contract1.client.should == client1
22    end
23
24    it "contract2 should belong to client2" do
25      contract2.client.should == client2
26    end
27
28    it "should show contract1 only to client1" do
29      Mongoid::Multitenancy.current_tenant = client1
30      Contract.all.count.should == 1
31      Contract.first.should == contract1
32    end
33
34    it "should show contract2 only to client2" do
35      Mongoid::Multitenancy.current_tenant = client2
36      Contract.all.count.should == 1
37      Contract.first.should == contract2
38    end
39  end
```

Uvedený test provede vytvoření dvou instancí třídy Client a dále dvou instancí třídy Contract tak, aby instance contract1 patřila tenantovi client1 a instance contract2 patřila tenantovi client2. Následně je proveden test, který ověřuje, zda má instance client1 přístup k instanci contract1 a dále, zda má přístup pouze k jedné instanci. Tento test se obdobným způsobem provádí i pro instanci client2 a contract2. Tím dochází k ověření, že pokud je nastaven aktuální tenant na hodnotu instance client1 a dojde k vytvoření instance třídy Contract, pak musí tato instance patřit k tenantovi client1 a stejně tak instance contract2 patřit ke client2.

Tímto způsobem jsou ověřeny všechny modely obsahující nesdílená data. Je tedy ověřeno oddělení dat popsané v 6.7.

7.5 Integrační testy

Integrační testy byly jako jednotkové testy zaměřeny na otestování oddělení dat a dále na otestování přidaných rozhraní v administrační sekci.

7.5.1 Testy oddělení dat

```
1  let!(:client1) do
2            Mongoid::Multitenancy.current_tenant = Factory.create(:client)
```

```
3           {
4             user:          Factory.create(:user),
5             contract:      Factory.create(:contract),
6           }
7   end
8
9   let!(:client2) do
10          Mongoid::Multitenancy.current_tenant = Factory.create(:client)
11          {
12            user:          Factory.create(:user),
13            contract:      Factory.create(:contract),
14          }
15  end
16  describe 'as clients' do
17    ["client1", "client2"].each do |var_name|
18      let(:client)       { self.send(var_name) }
19      let(:other_client) { (var_name == "client1") ? client2 : client1 }
20
21      describe 'as signed user from #{var_name}' do
22        before {
23          host! "#{client[:user].client.subdomain}.example.com"
24          post user_session_url, {
25                                   :user => { :email => client[:user].email,
26                                   :password => client[:user].password }
27                                 }
28        }
29
30        describe 'GET contracts/index' do
31          it 'responds with success and contains contract of #{var_name} only' do
32            get contracts_url
33            response.should be_success
34            response.body.should include(client[:contract].number)
35            response.body.should_not include(other_client[:contract].number)
36          end
37        end
38      end
39    end
```

Výše uvedený test provádí test oddělení dat pro kontroler Contracts, který slouží pro správu zakázek. Test začíná vytvořením dvou instancí client1 a client2 třídy Client a dále dvou instancí třídy Contract tak, aby ke každé instanci Client byla přiřazena jedna instance třídy Contract. Tím jsou zavedeni dva tenanti a každý má svůj jeden kontrakt. Dále je obdobným způsobem pro každého tenanta vytvořen uživatel. Před každým testem je provedeno nastavení subdomény daného tenanta a dále je přihlášen uživatel k němu přiřazený. Dále dochází pro každou instanci client1 a client2 k ověření, zda se ve výpisu kontraktů zobrazuje pouze

kontrakt, který byl k dané instanci přiřazen. To je provedeno navštívením stránky, která zobrazuje výpis všech kontraktů a kontrolou, zda se na této stránce nachází číslo kontraktu přiřazeného tenantovi a nenachází se číslo druhého tenanta. Tímto způsobem jsou otestovány všechny kontrolery, které se starají o spravování nespolečných dat.

7.5.2 Testy přidaných rozhraní

Přidaná rozhraní jsou testovány obdobným způsobem jako bylo provedeno testování rozhraní pro oddělení dat. Je otestováno přihlášení a dále všechny REST metody daného kontroleru.

7.6 Zátěžový test

Zátěžový test byl proveden pomocí aplikace JMeter. JMeter je projekt korporace Apache, je napsán v jazyce Java a jedná se o nástroj určený k provádění funkcionálních testů a testů výkonnosti. Jedná se o velice robustní nástroj, který umožňuje testování na mnoha protokolech, či serverech(http, https, soap, ftp, jms, mail ..). Díky tomu, že je napsán v jazyce Java, je multiplatformní. Testovací případy se definují pomocí formátu JMX, je definovaný pomocí XML. [9]

Testovací případy je možné definovat přímo pomocí grafického rozhraní aplikace JMeter, ale toto rozhraní není příliš přívětivé, proto byl pro generování testů použit gem Ruby-jmeter, který umožňuje snadné definování testovacích případů pomocí jazyka ruby a jejich následný export do formátu JMX.

K testovacím případům je možné přidávat tzv. listenery, které provádí požadovanou funkci s výslednými daty získanými z testu. Data je možné zobrazovat graficky v průběhu testu, ale dle dokumentace JMeteru není tento postup vzhledem k náročnosti vykreslení doporučen. Grafický výstup ze získaných dat tedy musí být generován další aplikací.

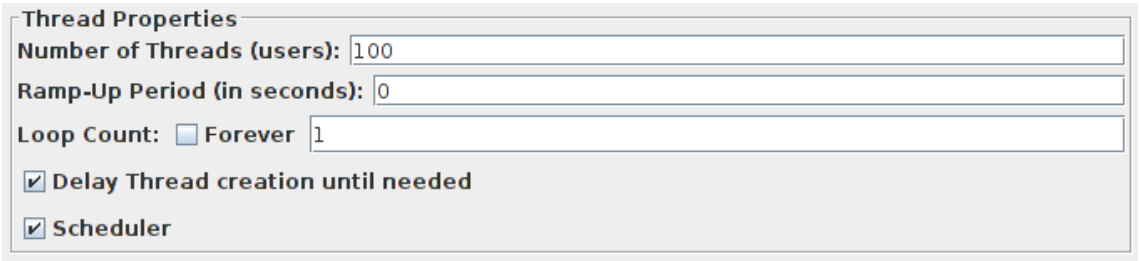
Ukázka testovacího případu definovaného pomocí ruby-jmeter:

```
require "rubygems"
require "ruby-jmeter"
user = {
  email: "user1@mail.cz",
  password: "******",
  subdomain: "******"
}
test do
  defaults domain: "#{user[:subdomain]}.autosw.cz",
           port:   80
  cookies clear_each_iteration: true
  cache  clear_each_iteration: true
  threads count: 10 do
    extract name: "authenticity_token",
```

```
15            regex: "input name=\"authenticity_token\" type=\"hidden\" value=\"(.+?)\""
16    transaction "list_contracts" do
17      submit "/users/sign_in", {
18                fill_in: {
19                  "utf8"                  => "%E2%9C%93",
20                  "authenticity_token"    => "${authenticity_token}",
21                  "user[email]"           => user[:email],
22                  "user[password]"        => user[:password],
23                  "commit"                => "Sign In",
24                }
25              }
26      visit name: "Contracts", url: "/contracts" do
27        assert contains: "Zakazky"
28      end
29    end
30  end
31end.jmx
```

Zátěžový test ověřuje odezvu aplikace při zátěži, proto je v testovacím případu třeba provést takové operace, které vyvolají zatížení aplikace. Pokud by testovací případ prováděl pouze zobrazení úvodní strany, pak by nebyl test dostatečně průkazný. Proto testovací případ simuluje přihlášení a následné zobrazení zakázek, což jsou z pohledu aplikace dostatečně složité operace, aby ve větším počtu aplikaci dostatečně zatížily.

Výše uvedený kód definuje chování i nastavení testu. Pro provádění testu bylo dále upraveno nastavení vláken, které simulují jednotlivé uživatele. Test byl proveden s nastavením uvedeným na obrázku 7.1. Počty vláken byly nastaveny dle hodnot uvedených v tabulce 7.1. Každé vlákno provedlo test jednou ve stejný okamžik.

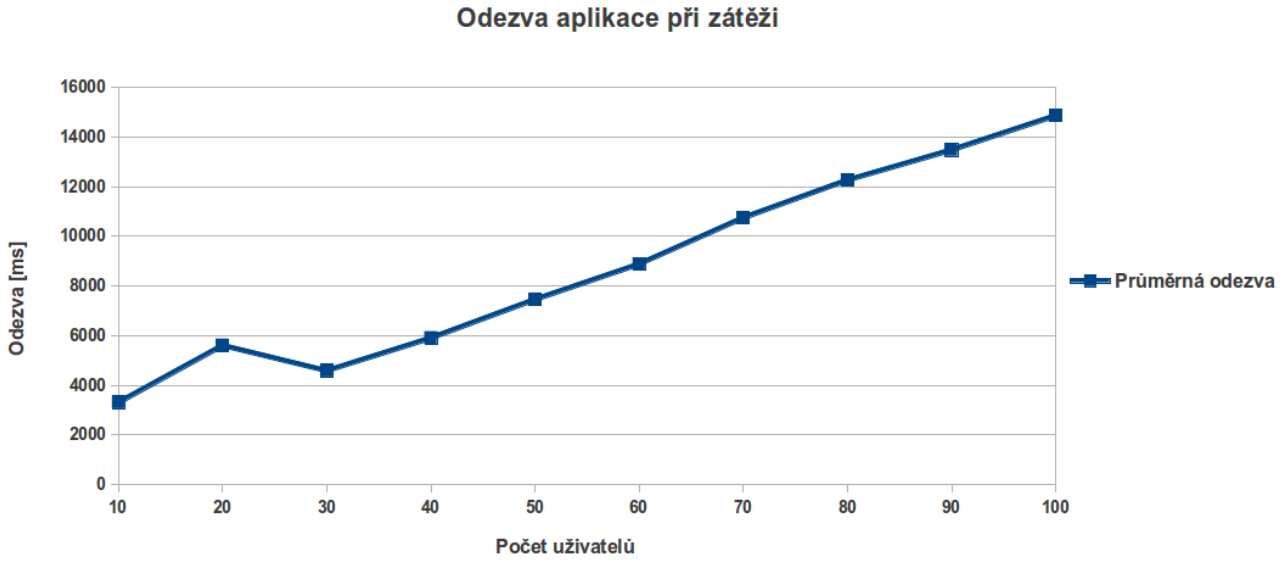


Obrázek 7.1: Nastavení programu JMeter

Daný testovací případ byl spuštěn pro určitý počet uživatelů, veškeré odezvy byly zaznamenány, zprůměrovány a zaneseny do grafu 7.2.

Počet uživatelů	Průměrná odezva [ms]
10	3311,1
20	5608,2
30	4587
40	5905,35
50	7459
60	8882,833
70	10750,885
80	12264,7
90	13479,588
100	14864,92

Tabulka 7.1: Průměrné hodnoty odezvy při dané zátěži



Obrázek 7.2: Graf odezvy při daném počtu uživatelů

Na grafu je vidět, že nárůst odezvy vzhledem k počtu uživatelů je téměř lineární. Při vyhodnocení grafu je třeba brát v úvahu, že testovací případ se skládal ze dvou HTTP požadavků. Doba, kterou uživatel musí vyčkat na odezvu systému, je tedy zhruba poloviční. Dále je třeba brát v úvahu, že požadavky byly vykonány ihned po sobě, běžný uživatel bude požadavky vykonávat s prodlevou, ve které bude zadávat případně číst data. Limit pro udržení pozornosti uživatele je 10s, 1s je pak limit pro udržení toku myšlenek [15], je-li doba delší než 1s pak uživatel ztrácí dojem, že operuje s daty přímo. Z toho lze vyvozovat, že pro zachování pohodlí při používání systému, je při současné konfiguraci, maximální počet současně operujících uživatelů zhruba 40. Při hodnotě 40 uživatelů je odezva zhruba 3s, což by mělo zaručit, že uživatel bude mít pocit, že operuje s daty přímo, protože je třeba uvažovat prodlevu mezi jednotlivými požadavky, která tuto hodnotu sníží.

Požadavky tenantů se mohou značně lišit, proto z naměřených hodnot nelze odvodit maximální počet tenantů, který daná konfigurace dokáže obsloužit.

Hodnotu maximálního počtu tenantů pro danou konfiguraci by bylo možné stanovit například pomocí zavedení kategorií s limity počtu uživatelů. Každý tenant by pak měl dán limit počtu uživatelů. Díky těmto limitům by bylo možné odvodit kolik uživatelů může v daný okamžik současně aplikaci používat a bylo by možné určit, zda je daná konfigurace dostatečná.

Tímto způsobem by ale mohla být daná konfigurace značně předimenzovaná. To lze řešit odvozením počtu tenantů, které dokáže daná konfigurace obsloužit, pomocí analýzy statistiky využití aplikace. Tuto statistiku lze získat například pomocí nástroje Google Analytics.

Kapitola 8

Závěr

Cílem práce bylo přepracovat danou Aplikaci Autoservis tak, aby jí bylo možné využívat jako aplikaci multitenantní a provozovat jako SaaS aplikaci.

Aplikace byla nejprve analyzována, z čehož byla vyvozena omezení a následně bylo navrženo několik způsobů oddělení dat jednotlivých instancí autoservisů.

Z navržených způsobů realizace oddělení dat byl z hlediska bezpečnosti nejlepší způsob uložení instancí do oddělených databází, ale vzhledem k vysoké režii na velikost úložiště v databázi MongoDB nebyl tento způsob vybrán. Bylo tedy třeba vybrat ze dvou způsobů uložení do sdílené databáze. Z možnosti použití kolekcí nebo použití identifikátorů k oddělení dat byl nakonec vybrán způsob oddělení pomocí identifikátorů, protože počet kolekcí je omezen a dále databáze MongoDB není vytvořena tak, aby škálovala na úrovni kolekcí.

Data jednotlivých instancí jsou tedy uložena ve stejné databázi, tento způsob má nejmenší úroveň izolace dat ze všech navržených způsobů. Výhodou ovšem jsou nízké nároky na úložiště a tím i nižší náklady na provoz aplikace. Oddělení dat bylo ověřeno jednotkovými a integračními testy. Jednotlivé instance autoservisů jsou odděleny pomocí subdomén a aktuální tenant je nastavován podle dané subdomény, i tento způsob zvyšuje úroveň zabezpečení, protože jsou instance dále odděleny pomocí HTTP sessions. Ke kritické situaci může dojít v případě chyby v aplikaci, ale tyto možnosti jsou pokryty zmiňovanými testy.

Dále bylo cílem vytvoření rozhraní pro správu jednotlivých instancí a globálních dat, které používají všechny instance. Tento cíl byl splněn rozšířením stávajícího rozhraní pro údržbu. Rozhraní umožňují správu jednotlivých instancí, jejich uživatelů, výrobců a modelů automobilů, sazeb DPH a dále je možné zobrazit statistiky o počtech vytvoření zakázek a počtů instancí autoservisů. Tato rozhraní jsou otestována pomocí integračních testů.

Dále bylo navrženo několik možností nasazení aplikace u poskytovatelů cloudových služeb. Nasazení v cloudu se u SaaS aplikace nabízí díky předpokládanému růstu nároků aplikace, které se budou zvyšovat s rostoucím počtem hostovaných instancí. Tyto nároky lze díky nasazení v cloudu naplnit snazším škálováním, než je to v případě nasazení na současné infrastruktuře ale s vyššími náklady na provoz.

Na současné infrastruktuře je aplikační server a databáze spuštěna v rámci jednoho serveru. Případné škálování je možné provést přidáním dalšího serveru, který by sloužil pouze pro databázi a původní server by sloužil pouze pro aplikační server, tato úprava by umožnila

na původním serveru, díky uspořené paměti po přesunu databázového serveru, spustit více instancí aplikačního serveru.

Další škálování by pak bylo možné přidáním serverů pro databázi, což databáze MongoDB umožňuje. Případně přidáním serverů pro další instance aplikačního serveru. V případě přidání sereru pro další instance aplikačního serveru by bylo nezbytné nastavit jednotné úložiště pro HTTP sessions.

Pokud by byla aplikace nasazena u cloud poskytovatele, škálování by bylo snazší v závislosti na poskytovateli, u kterého by byla aplikace nasazena.

V případě PaaS poskytovatelů by se jednalo o pouhé přidání instancí, které slouží k běhu aplikace. Vše je snadno řešitelné pomocí webového rozhraní poskytovatelů služeb.

Při nasazení u IaaS poskytovatele je možné daný server škálovat vertikálně, tedy navýšit jeho výkon a v rámci něj zvýšit počet instancí pro běh aplikace, případně pro běh databáze. Dále je možné použít horizontální škálování, jehož postup by byl obdobný jako škálování na současné infrastruktuře.

Přínosem celého řešení je snížení nákladů při provozování všech instancí autoservisů v rámci jednoho serveru, jedné aplikace a jedné databáze, z čehož pramení i další výhody, kterými jsou například údržba pouze jedné instance aplikace. Další nezanedbatelnou výhodou je možnost snadného přidání nové instance formou registrace.

Obdobný postup popsaný v této práci lze, i díky konvencím frameworku Ruby On Rails, použít s menšími úpravami i pro jiné aplikace vytvořené s použitím tohoto frameworku. Postup není vázán na databázi MongoDB, použitý gem Mongoid-multitenancy lze snadno nahradit gemem acts_as_tenant a postup aplikovat na aplikace využívající relační databáze.

Literatura

[1] MongoDB documentation - document, .
`http://docs.mongodb.org/manual/core/document/`, stav z 20. 1. 2014.

[2] MongoDB documentation - introduction, .
`http://www.mongodb.org/about/introduction/`, stav z 20. 1. 2014.

[3] MongoDB documentation, .
`http://docs.mongodb.org/manual/`, stav z 20. 1. 2014.

[4] MongoHQ docs, .
`http://docs.mongohq.com/use-cases/multi-tenant.html`, stav z 20. 1. 2014.

[5] Mongoid Documentation, .
`http://mongoid.org/en/mongoid/index.html`, stav z 20. 1. 2014.

[6] Ruby On Rails Guides - Active Record.
`http://guides.rubyonrails.org/`, stav z 20. 1. 2014.

[7] Simple Forms.
`https://github.com/plataformatec/simple\_form`, stav z 20. 1. 2014.

[8] Travis CI.
`http://docs.travis-ci.com/user/build-configuration/`, stav z 20. 1. 2014.

[9] APACHE.ORG. JmxTestPlan.
`https://wiki.apache.org/jmeter/JmxTestPlan`, stav z 9. 4. 2014.

[10] BEZEMER, A. Z. C.-P. Multi-tenant SaaS applications: maintenance dream or nightmare?
ACM, 2010.

[11] BIGG, R. Multitenancy with Rails.
Leanpub.com, 2013.

[12] CHATE, S. Convert your web application to a multi-tenant SaaS solution.
http://public.dhe.ibm.com/software/dw/cloud/library/cl-multitenantsaas-pdf.pdf.

[13] FREDERICK CHONG, R. W. G. C. Response Times: The 3 Important Limits.
`http://msdn.microsoft.com/en-us/library/aa479086.aspx`, stav z 20. 1. 2014.

[14] HARTL, M. Ruby on Rails Tutorial: Learn Web Development with Rails. Addison-Wesley, 2012.

[15] NIELSEN, J. Multi-Tenant Data Architecture. `http://www.nngroup.com/articles/response-times-3-important-limits/`, stav z 9. 4. 2014.

[16] SOSINSKY, B. *Cloud Computing Bible*. Wiley Publishing, 1st edition, 2011. ISBN 9780470903568.

[17] web:infodp. K336 Info — pokyny pro psaní diplomových prací. `https://info336.felk.cvut.cz/clanek.php?id=400`, stav ze 4. 5. 2009.

Příloha A

Instalační a uživatelská příručka

Pro instalaci a spuštění Aplikace Autoservis ve vývojovém prostředí je třeba vykonat následující kroky.

1. Instalace interpreteru jazyka Ruby, instalaci je možné provést pomocí balíčkovacího systému dané platformy, ale doporučuje se použití správců verzí, které umožňují volbu verzí. Jedná se o RVM nebo o rbenv.

2. Dále je třeba nainstalovat databázi MongoDB, na operačních systémech založených na linuxové distribuci Debian lze použít apt-get install mongodb.

3. Dále je třeba nainstalovat veškeré gemy a framework Ruby On Rails, toho lze snadno docílit přepnutím se do adresáře s aplikací pomocí příkazu cd a následně spuštěním příkazu bundle install. Tím se nainstalují veškeré závislosti.

4. Dále je třeba nainstalovat výchozí data, která obsahují informace o státech a jejich DPH, jména výrobců apod. To se provede pomocí příkazu rails c v kořenovém adresáři s aplikací a následně pomocí load 'db/seeds/rb'.

5. Po úspěšném provedení výše uvedených kroků je možné aplikaci spustit příkazem rails s, zadaném opět v kořenovém adresáři s aplikací.

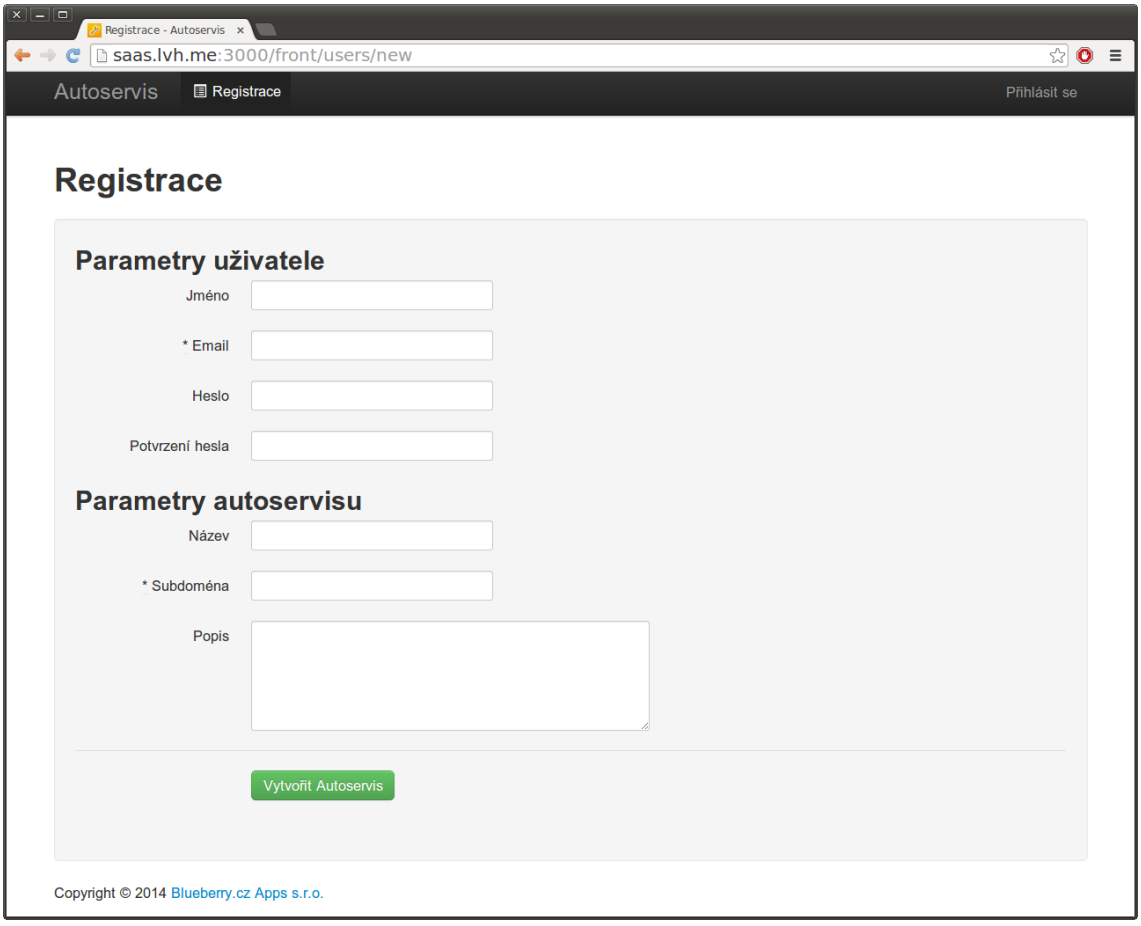
6. Následně je aplikace přístupná na adrese http://localhost:3000

Použití vytvořených instancí lze ověřit pomocí domény http://lvh.me, pokud se tedy registruje například doména autoservis1, pak bude adresa tohoto autoservisu, při výše uvedeném spuštění, http://autoservis1.lvh.me:3000.

A.1 Uživatelská příručka - role uživatel

A.1.1 Registrace

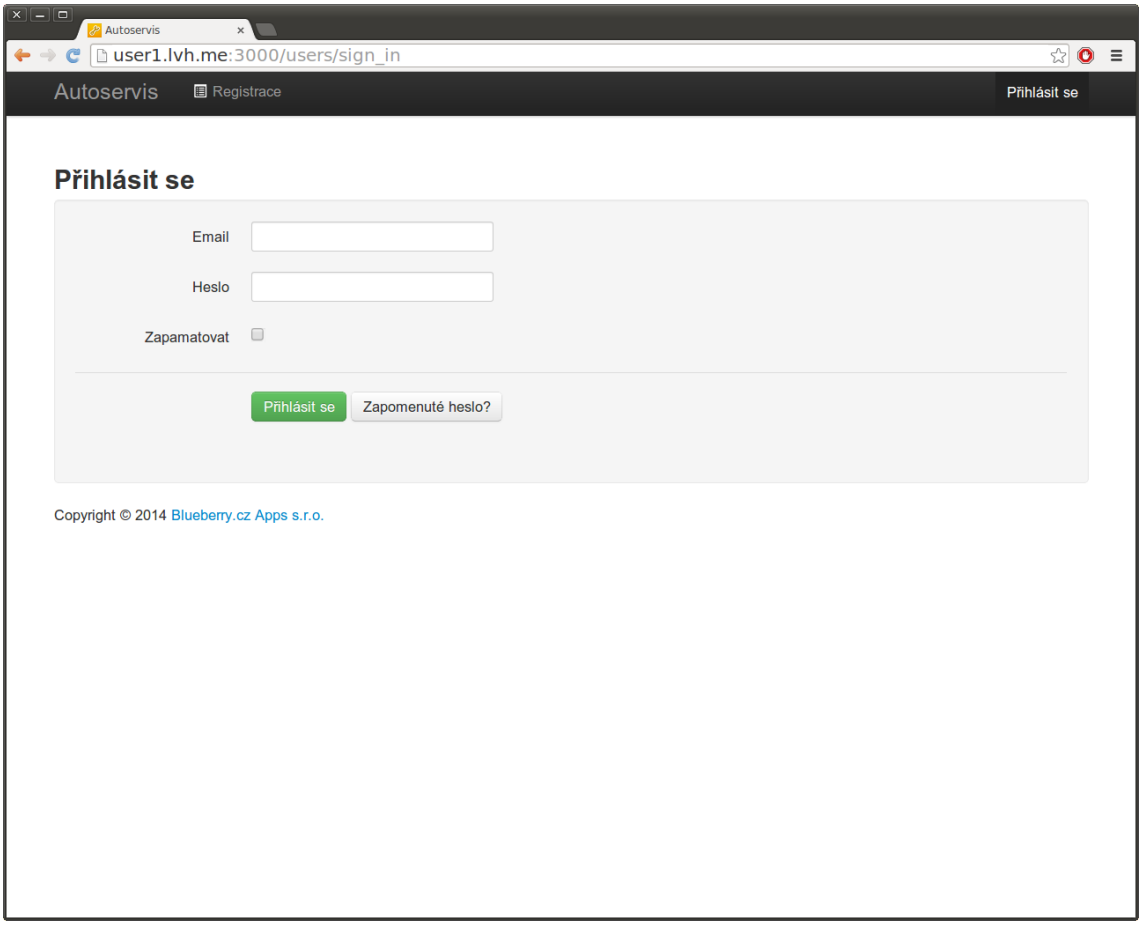
Registrace se skládá ze dvou formulářů, v prvním se vyplňují informace o uživateli a ve druhém informace o vytvářeném autoservisu. Udává se subdoména, která slouží pro identifikaci autoservisu a dále pro přístup k autoservisu a dále název a popis, tyto dva údaje se zobrazují na úvodní stránce daného autoservisu, tedy po přístupu na registrovanou subdoménu.



Obrázek A.1: Snímek - registrace

A.1.2 Přihlášení

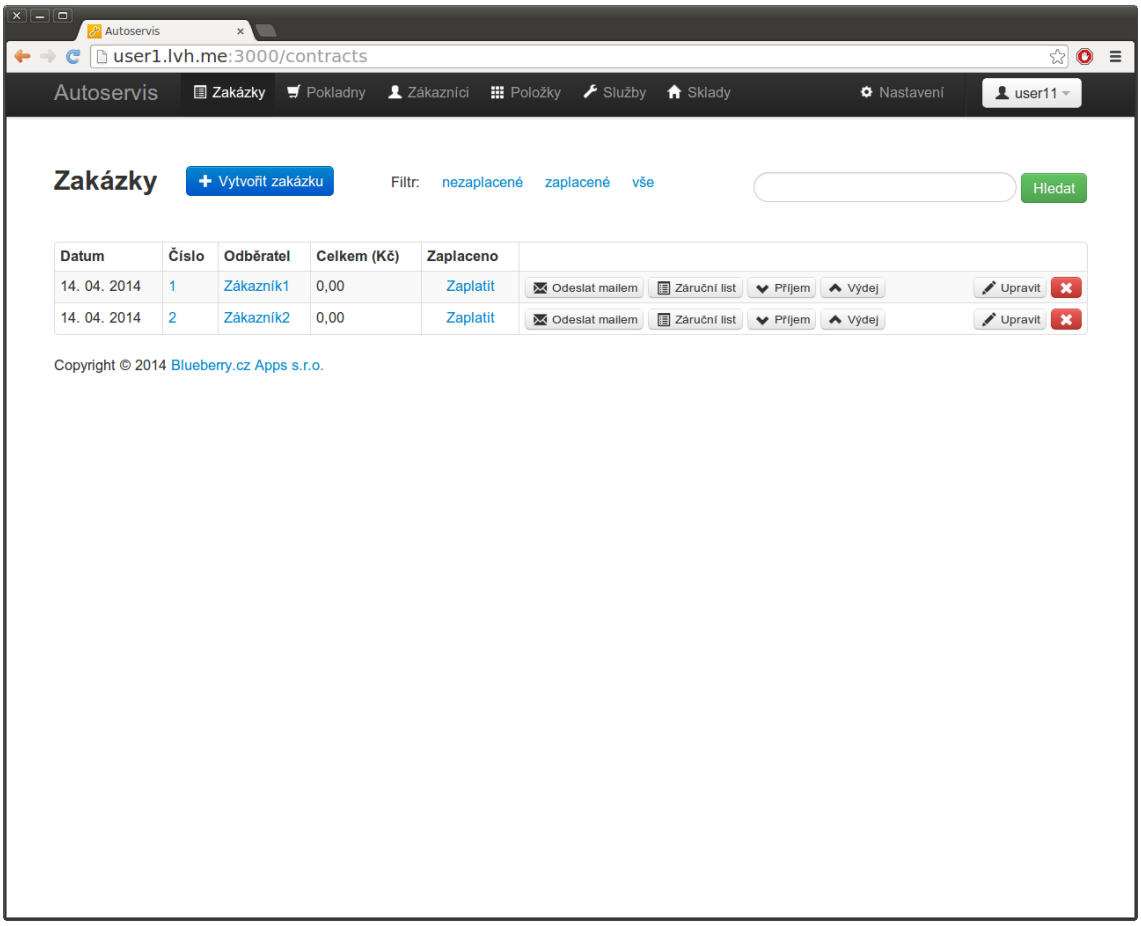
Přihlášení se provádí v rámci registrované subdomény. Po přihlášení je uživatel přesměrován do sekce zakázky. Odtud se může pomocí menu v horní části přesouvat do dalších sekcí.



Obrázek A.2: Snímek - přihlášení

A.1.3 Zakázky

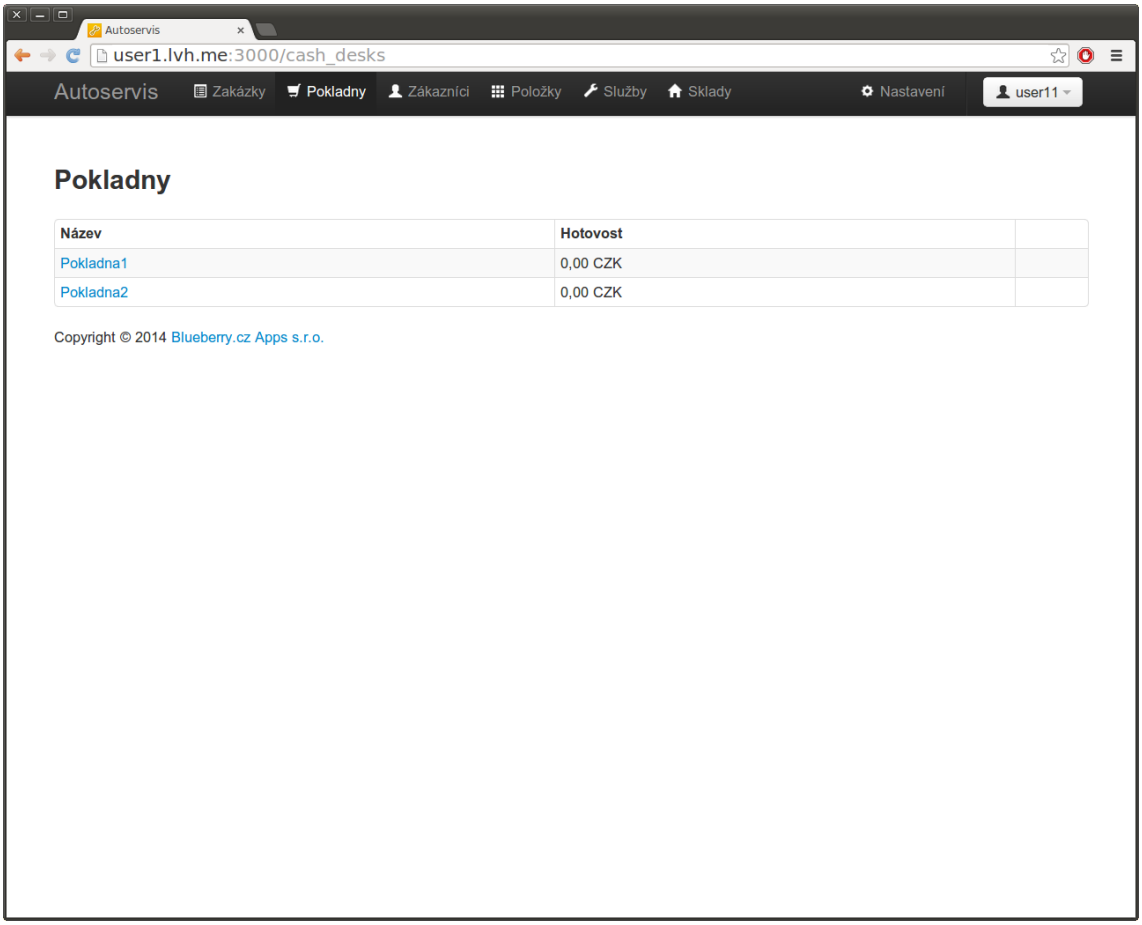
Zobrazení všech zakázek daného autoservisu. V horní části je možné nastavit filtrování na zaplacené/nezaplacené zakázky a dále je možné vyhledat požadovanou zakázku.



Obrázek A.3: Snímek - zakázky

A.1.4 Pokladny

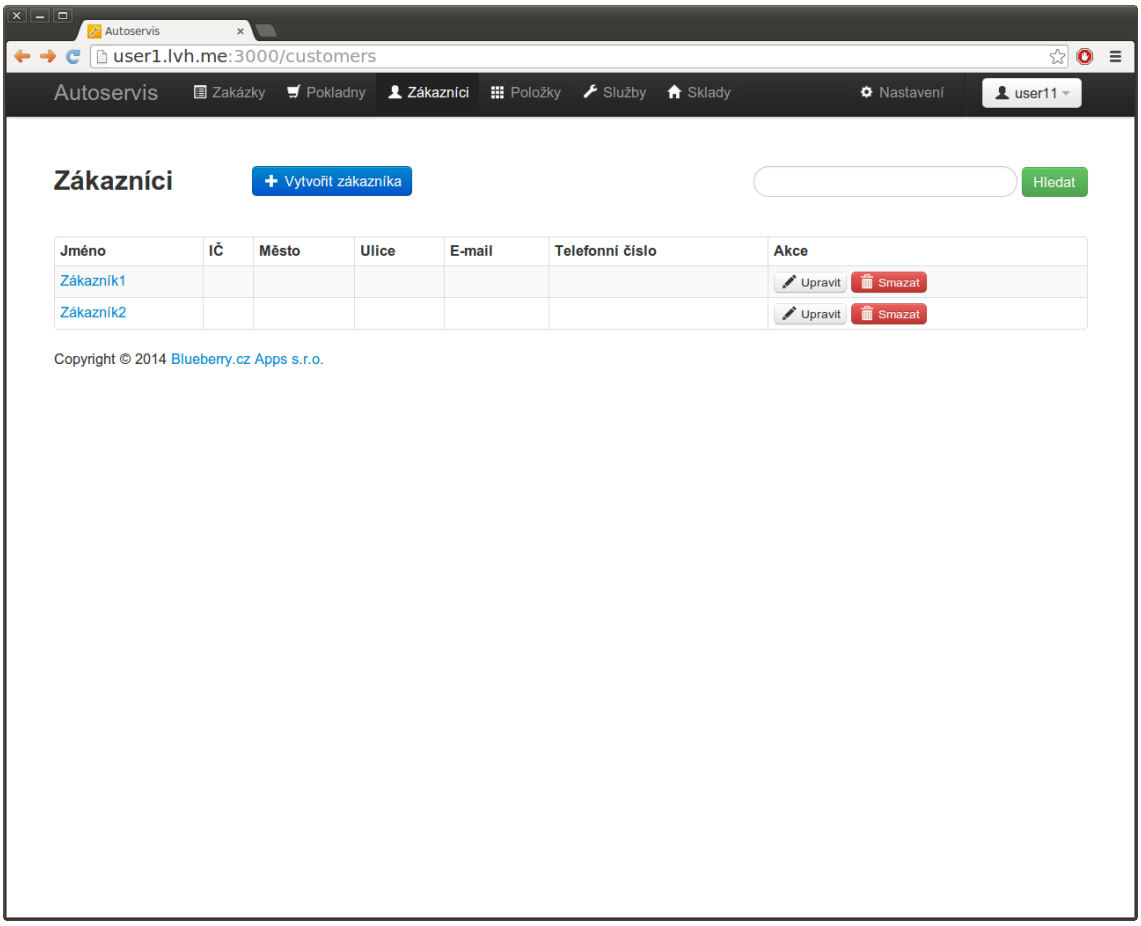
Zobrazení pokladen daného autoservisu. Spravování pokladen je možné pouze přes roli admin.



Obrázek A.4: Snímek - pokladny

A.1.5 Zákazníci

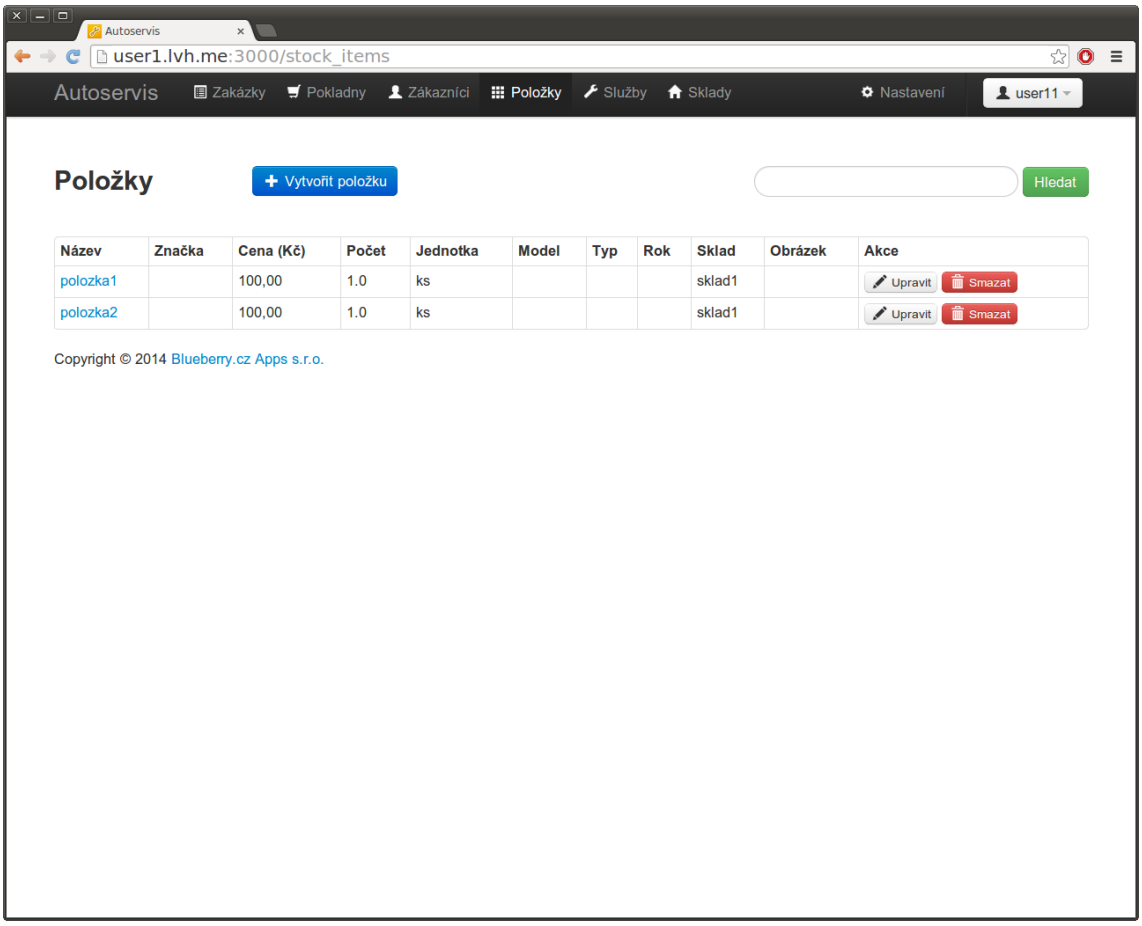
Zobrazení zákazníků evidovaných v daném autoservisu.



Obrázek A.5: Snímek - zákazníci

A.1.6 Položky

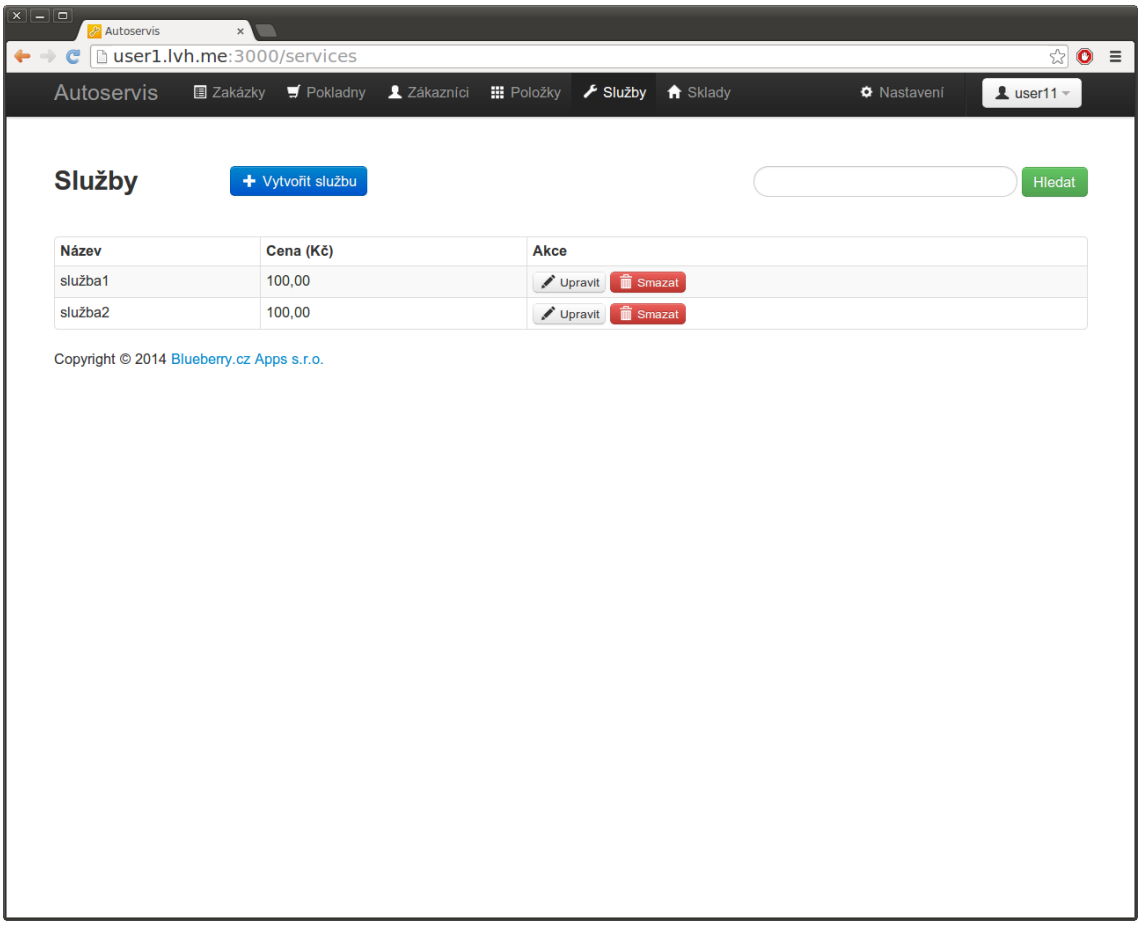
Zobrazení položek vedených pro daný autoservis.



Obrázek A.6: Snímek - položky

A.1.7 Služby

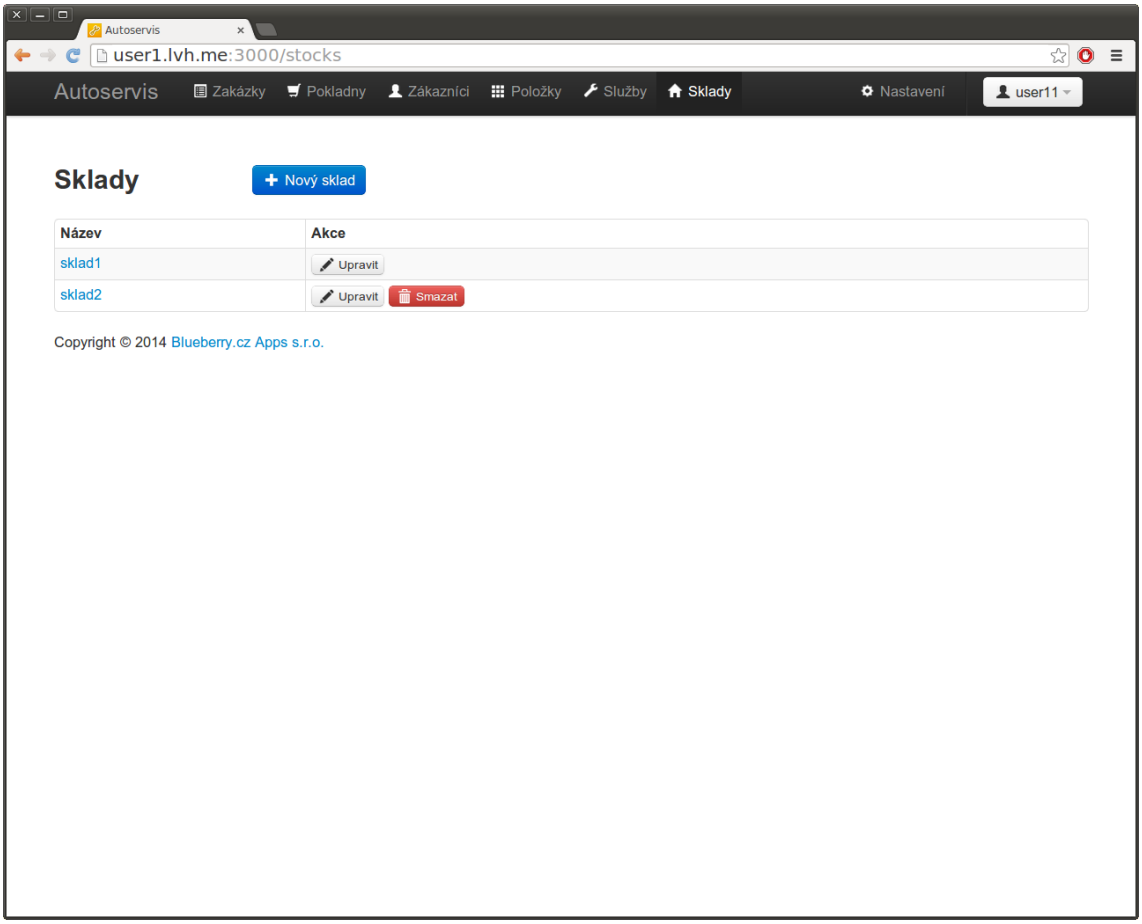
Zobrazení služeb pro daný autoservis.



Obrázek A.7: Snímek - služby

A.1.8 Sklady

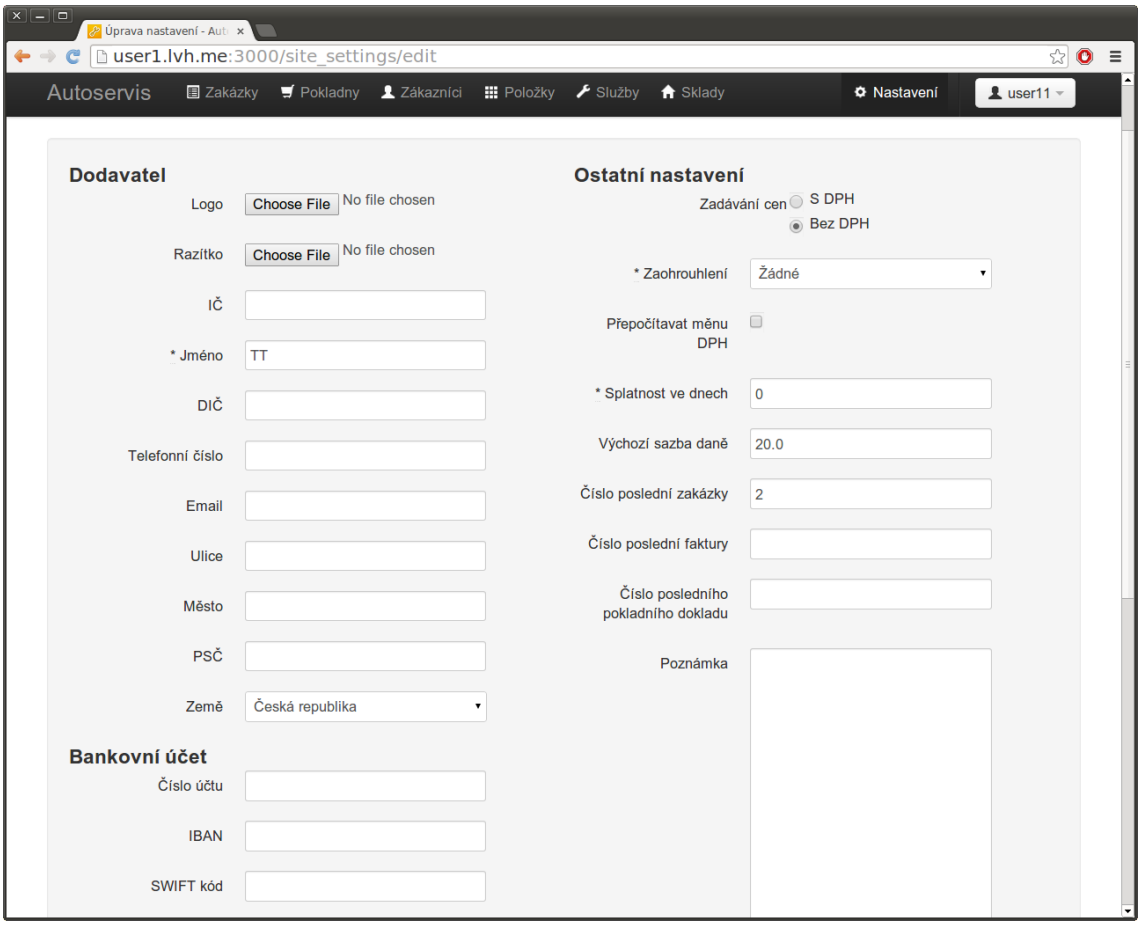
Zobrazení skladů daného autoservisu.



Obrázek A.8: Snímek - sklady

A.1.9 Nastavení

Zobrazení nastavení daného autoservisu, oproti původní Aplikaci Autoservis bylo rozšířeno o nastavení názvu a popisu autoservisu.

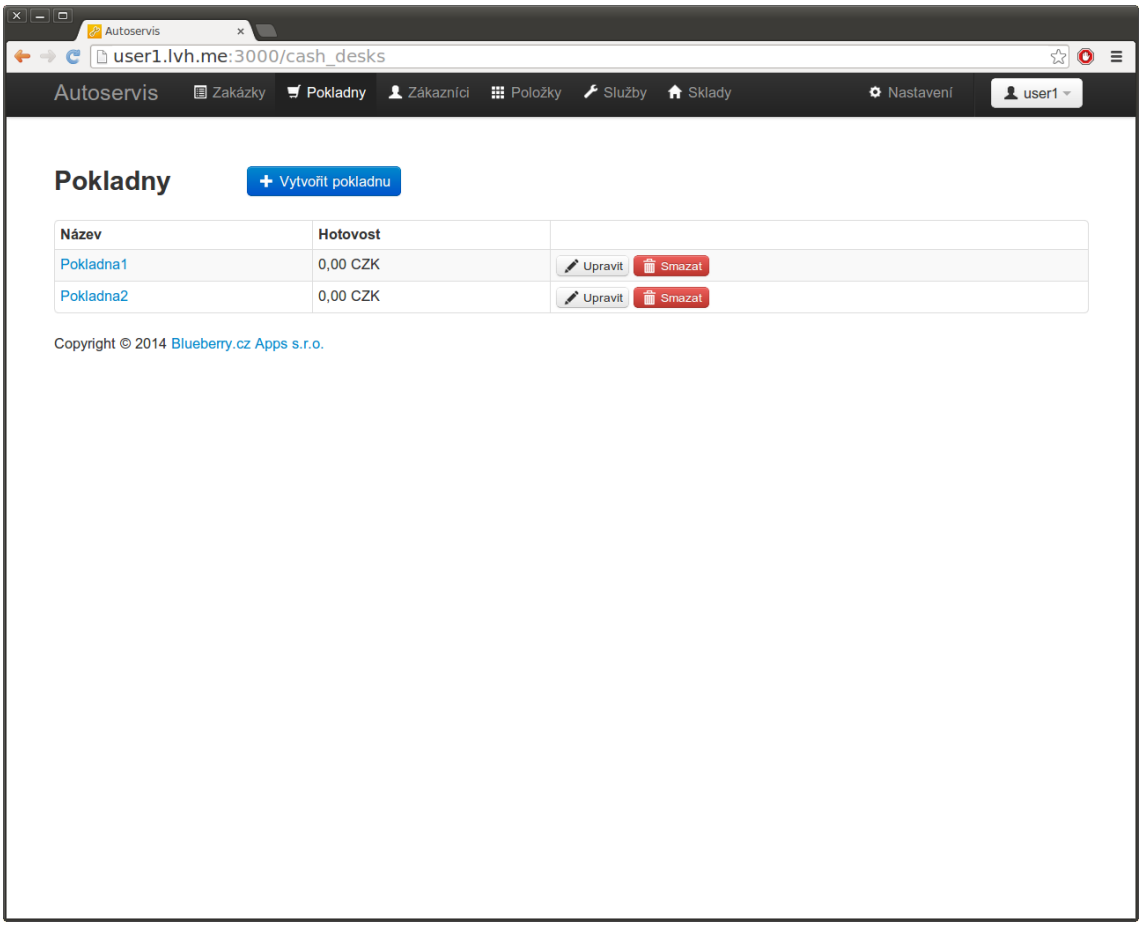


Obrázek A.9: Snímek - nastavení

A.2 Uživatelská příručka - role administrátor

Zobrazení pokladen v roli administrátor, oproti běžnému uživateli může administrátor pokladny spravovat.

A.2.1 Pokladny

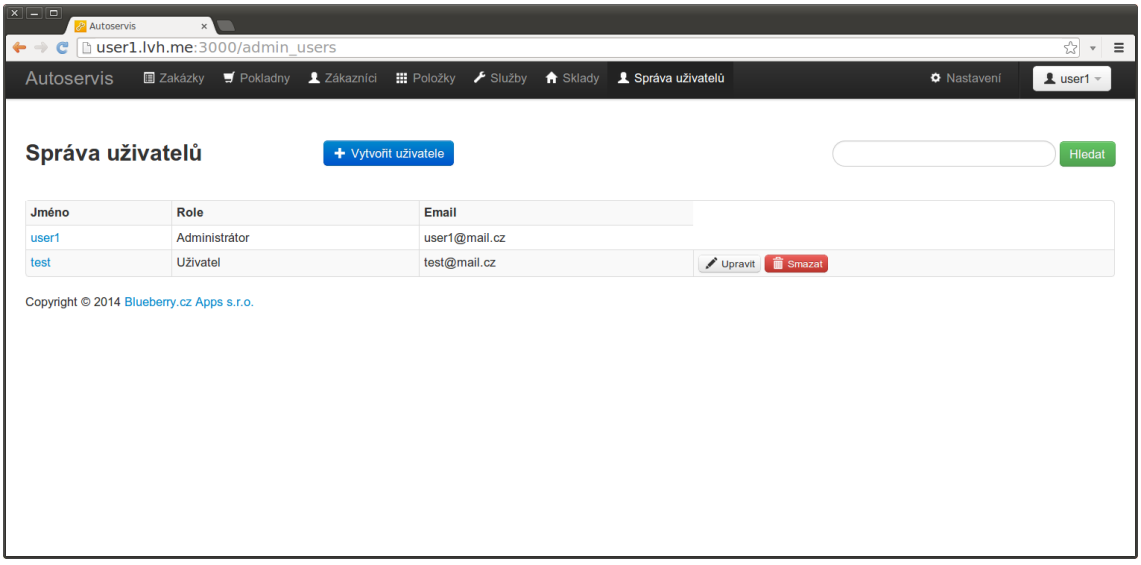


Obrázek A.10: Snímek - pokladny

Zobrazení správy uživatelů v roli administrátor, oproti běžnému uživateli může administrátor spravovat uživatele.

Toto rozhraní bylo do aplikace přidáno.

A.2.2 Správa uživatelů



Obrázek A.11: Snímek - správa uživatelů

A.3 Uživatelská příručka - role údržba

Zde uvedená rozhraní byla do aplikace přidána, jsou určena pro správu jednotlivých instancí.

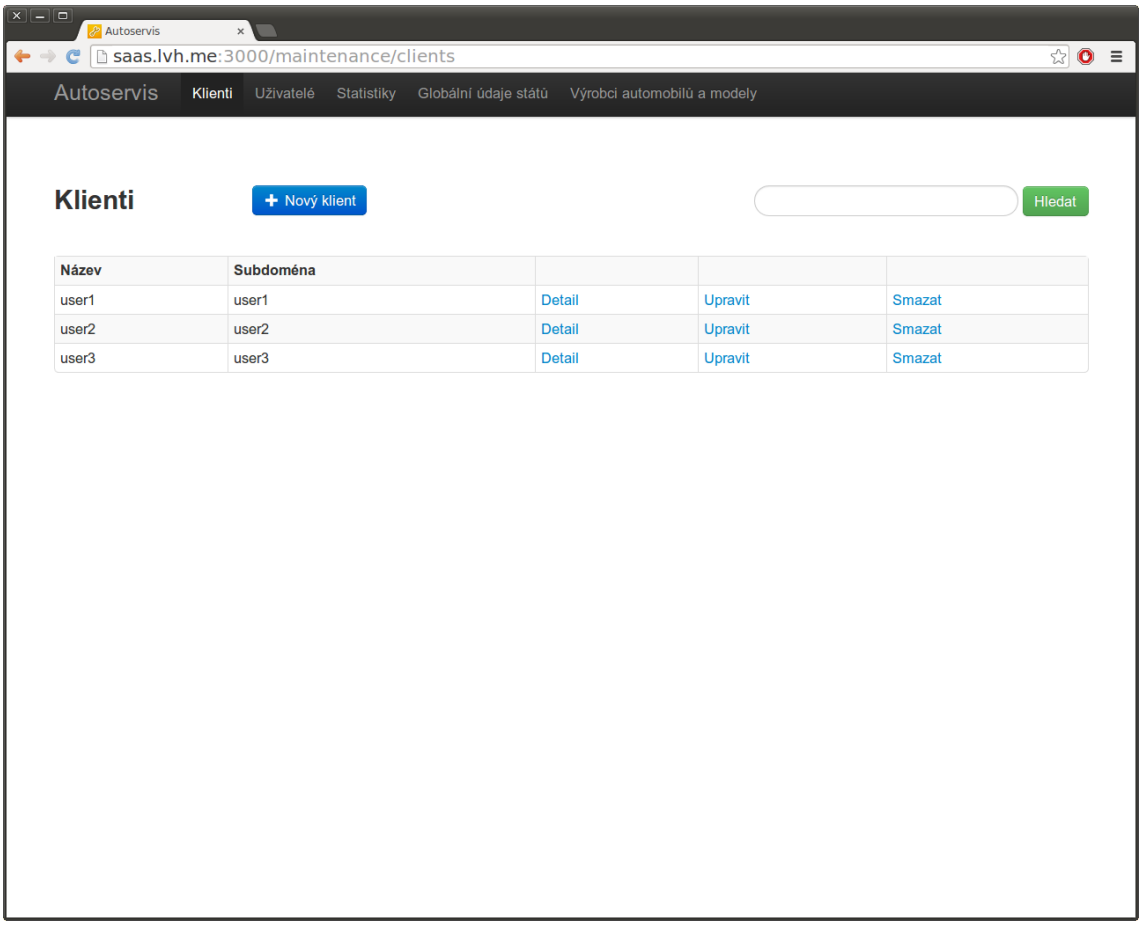
A.3.1 Přihlášení

Pokud je aplikace spuštěna ve vývojovém prostředí, pak se přihlášení provádí na adrese: http://saas.lvh.me:3000/maintenance/

A.3.2 Klienti

Zobrazení seznamu klientů umožňuje správu klientů, v detailu klienta je možné spravovat jeho uživatele.

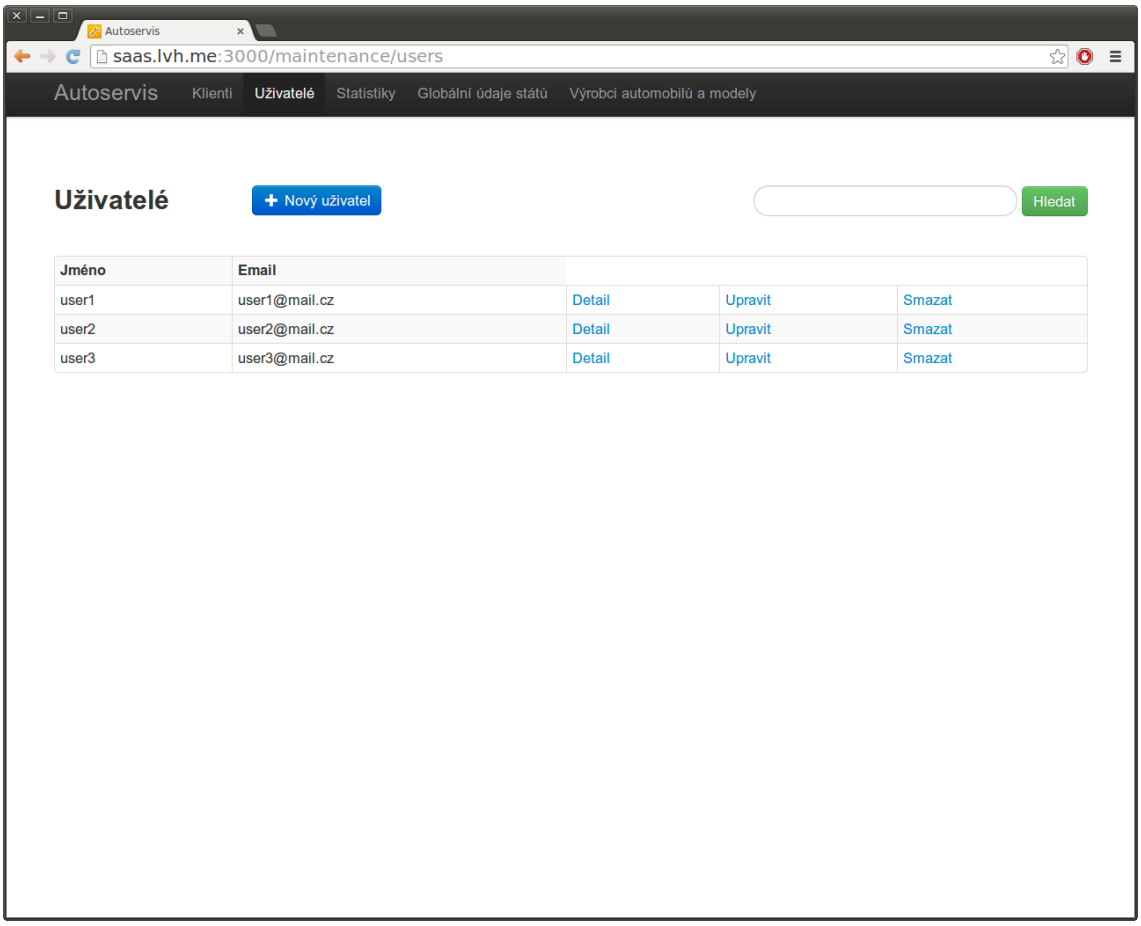
Klienty je dále možné vyhledávat pomocí vstupního pole v pravé horní části a tlačítka hledat.



Obrázek A.12: Snímek - klienti

A.3.3 Uživatelé

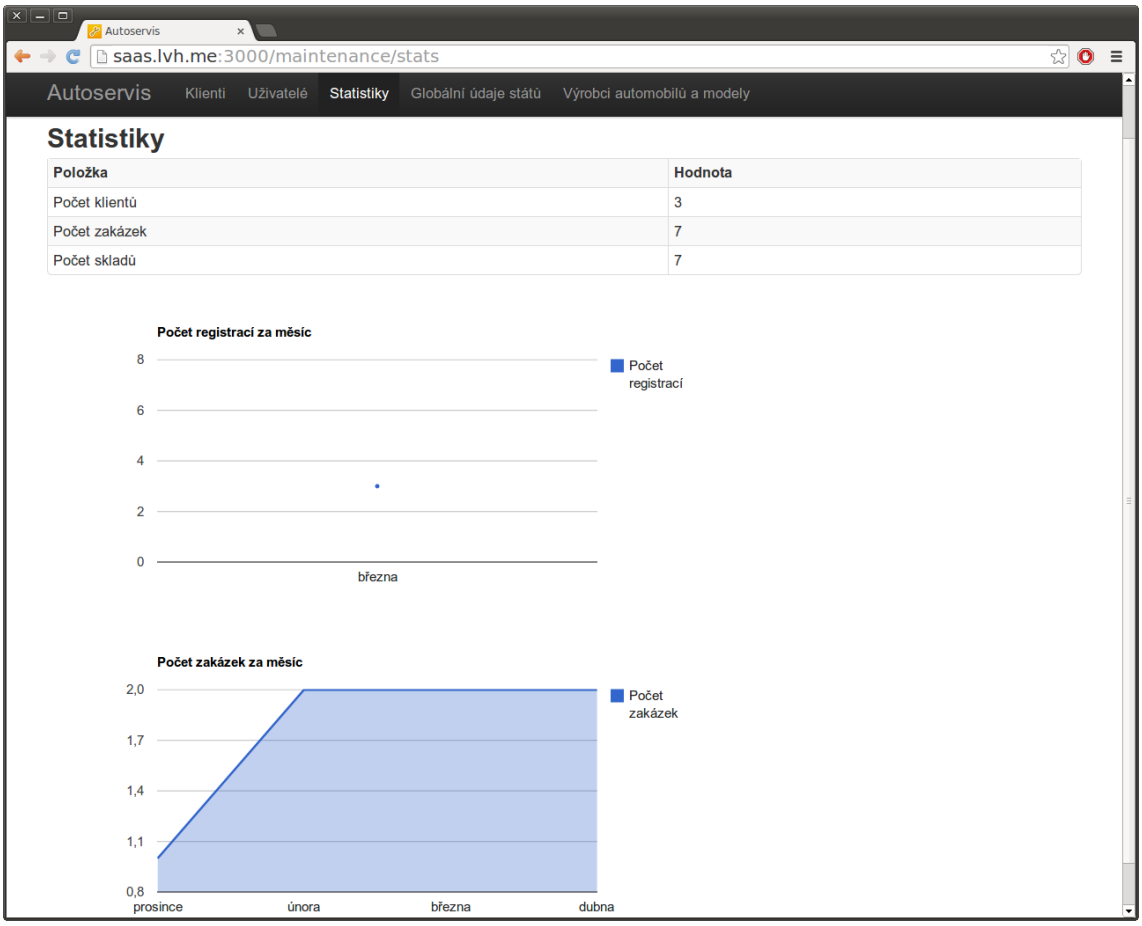
Zobrazení všech uživatelů vedených v systému, je možné provádět správu uživatelů. Uživatele je možné vyhledávat například pomocí emailu.



Obrázek A.13: Snímek - uživatelé

A.3.4 Statistiky

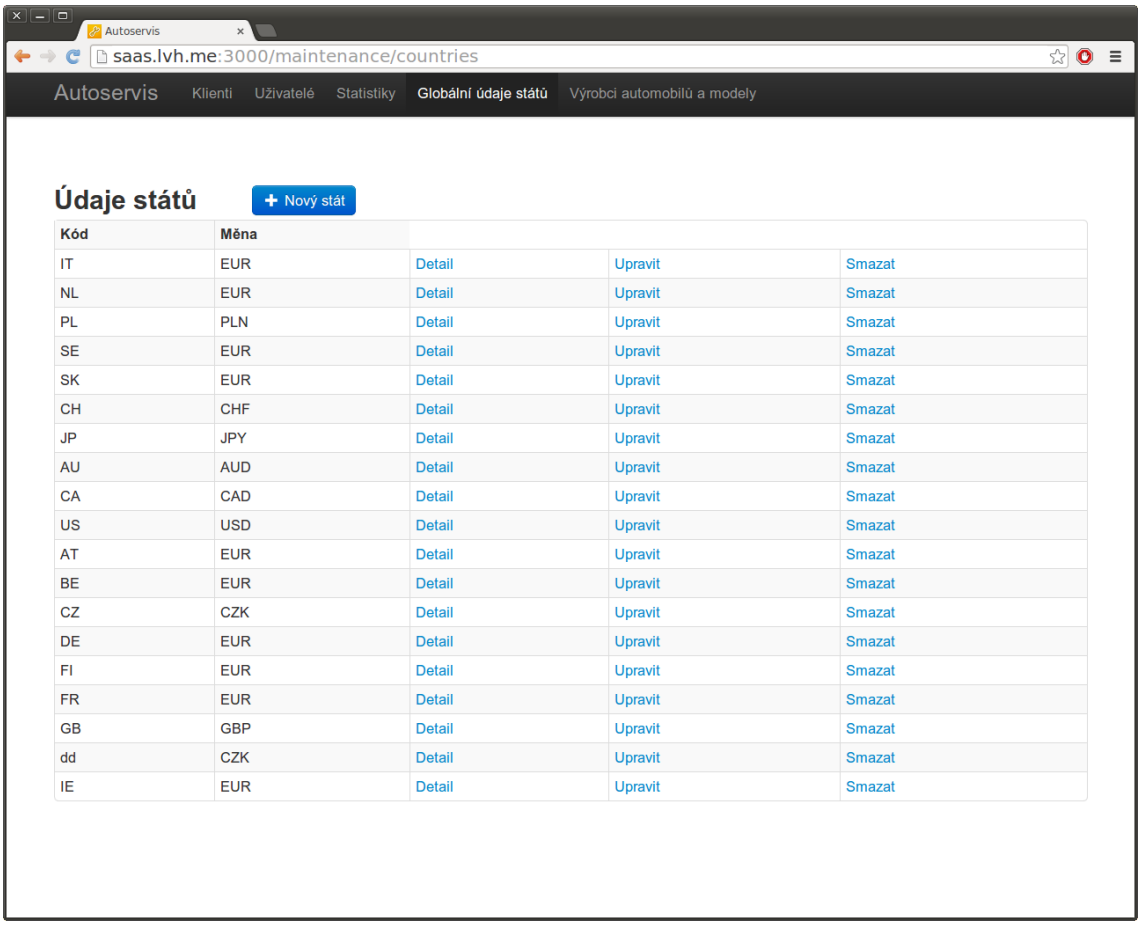
Rozhraní statistik zobrazuje aktuální počet klientů vedených v systému, dále je zobrazen počet uživatelů a počet skladů. Pro přehled jsou dále přítomny grafy zobrazující počty klientů a počty zakázek v měsících daného roku.



Obrázek A.14: Snímek - statistiky

A.3.5 Globální údaje států

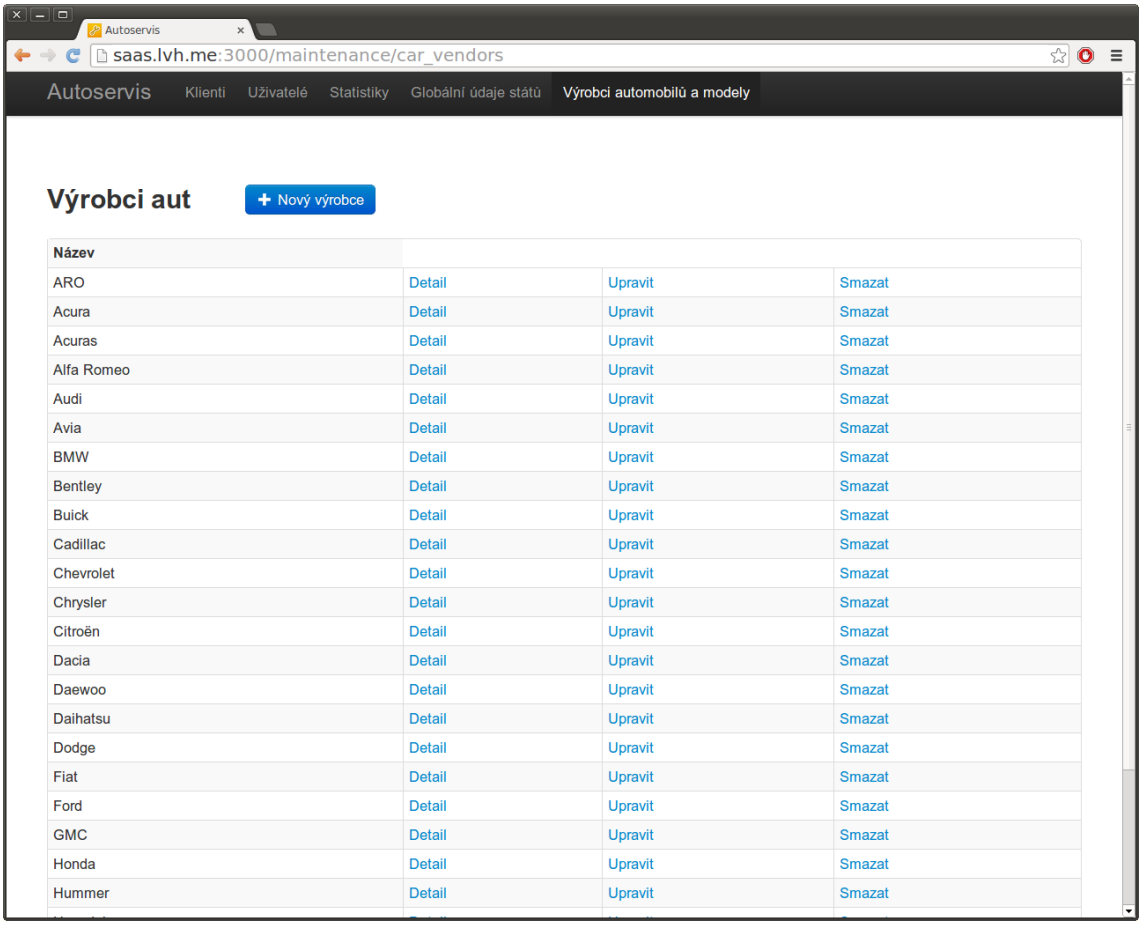
Rozhraní globální údaje států slouží ke správě států a jejich DPH sazeb. Úprava daňových sazeb se provádí v detailu daného státu.



Obrázek A.15: Snímek - globální údaje států

A.3.6 Výrobci automobilů a modely

Rozhraní výrobci automobilů a modely slouží k definování výrobců a modelů automobilů. Výrobci definovaní v tomto rozhraní jsou uživateli nabízeni při vytváření zakázky. U každého výrobce je dále možné definovat modely automobilů, tyto modely jsou pak uživateli nabízeny v průběhu zadávání názvu modelu automobilu. Modely automobilů se přidávají v detailu daného výrobce.



Obrázek A.16: Snímek - výrobci automobilů a modely

Příloha B

Obsah přiloženého CD

```
.
|-- code
|   |-- app
|   `-- mongoid-multitenancy .. upravená verze gemu Mongoid-multitenancy
|-- test
|   |-- jmeter .. testovací případ pro JMeter
|   |-- results .. výslednky testů
|   `-- ruby-jmeter .. testovací případ v ruby-jmeter
|-- text .. text práce
`-- readme.txt
```