VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA PODNIKATELSKÁ
ÚSTAV INFORMATIKY

FACULTY OF BUSINESS AND MANAGEMENT
INSTITUTE OF INFORMATICS

NÁVRH VÝVOJE WEBOVÝCH APLIKACÍ S AUTOMATICKÝM VYTVÁŘENÍM DATABÁZOVÉHO SCHÉMATU

WEB APPLICATION DEVELOPMENT SCHEME WITH AUTOMATIC DATABASE SCHEMA CREATION

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — Bc. KRISTÍNA PROCHOCKÁ
AUTHOR

VEDOUCÍ PRÁCE — Ing. JIŘÍ KŘÍŽ, Ph.D., Ph.D.
SUPERVISOR

BRNO 2015

Vysoké učení technické v Brně
Fakulta podnikatelská

Akademický rok: 2014/2015
Ústav informatiky

ZADÁNÍ DIPLOMOVÉ PRÁCE

Prochocká Kristína, Bc.

Informační management (6209T015)

Ředitel ústavu Vám v souladu se zákonem č.111/1998 o vysokých školách, Studijním a zkušebním řádem VUT v Brně a Směrnicí děkana pro realizaci bakalářských a magisterských studijních programů zadává diplomovou práci s názvem:

Návrh vývoje webových aplikací s automatickým vytvářením databázového schématu

v anglickém jazyce:

Web Application Development Scheme with Automatic Database Schema Creation

Pokyny pro vypracování:

Úvod
Cíle práce, metody a postupy zpracování
Teoretická východiska práce
Analýza současného stavu
Vlastní návrhy řešení
Závěr
Seznam použité literatury
Přílohy

Seznam odborné literatury:

FOWLER M. Patterns of Enterprise Application Architecture. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA. 2002, ISBN 978-0321127426.
REDMOND ERIC, W. J. R. Seven Databases in Seven Weeks: A Guide to Modern Databases and the NoSQL Movement. Pragmatic Bookshelf Book, 2012, ISBN 978-1-93435-692-0.
SADALAGE P. J. and M. FOWLER. NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence. Addison-Wesley Professional, 2012, ISBN 978-0321826626.
SILBERSCHATZ A., H. KORTH and S. SUDARSHAN. Database System Concepts. McGraw-Hill, 2005, ISBN 978-0-07-295886-7.

Vedoucí diplomové práce: Ing. Jiří Kříž, Ph.D., Ph.D.

Termín odevzdání diplomové práce je stanoven časovým plánem akademického roku 2014/2015.

L.S.

doc. RNDr. Bedřich Půža, CSc. Ředitel ústavu	doc. Ing. et Ing. Stanislav Škapa, Ph.D. Děkan fakulty

V Brně, dne 28.2.2015

Abstrakt

Práce navrhuje a implementuje mezivrstvu pro backend i frontend část webové aplikace, spolu s uživatelským rozhraním pro výstupy analýzy a správu. Tato vrstva byla navržena na základě analýzy několik existujících řešení, v různých jazycích a prostředích, a implementována s použitím databází MongoDB a PostgreSQL, v jazyce PHP na straně serveru a JavaScriptu na straně klienta. Poskytuje okamžitě použitelnou flexibilní persistenci přímo ve frontendu, s analýzou, zpětnou vazbou a možnosti převodu do PostgreSQL databáze při zachování stejného API umožňující plynulý přechod.

Klíčová slova

MongoDB, PostgreSQL, PHP, JavaScript, AngularJS, REST API, analýza struktury dat

Abstract

This thesis designs and implements an intermediary layer on both the backend and the frontend part of a web application, together with a user interface used to show analysis outputs and to manage the data. This layer was designed based on analysis of several existing solutions, in various languages and environments, and implemented using MongoDB and PostgreSQL databases, written in PHP on the server side and in JavaScript on the client side. It offers an immediately usable flexible persistence layer directly in the frontend, with analysis, feedback and an option to convert collections to PostgreSQL while keeping the same API, enabling a fluent transition.

Keywords

MongoDB, PostgreSQL, PHP, JavaScript, AngularJS, REST API, data structure analysis

PROCHOCKÁ, K. *Návrh vývoje webových aplikací s automatickým vytvářením databázového schématu*: diplomová práce. Brno: Vysoké učení technické v Brně, Fakulta podnikatelská, Ústav informatiky, 2015. 96 s. Vedoucí diplomové práce Ing. Jiří Kříž, Ph.D., Ph.D.

PROHLÁŠENÍ

Prohlašuji, že svou diplomovou práci na téma „Návrh vývoje webových aplikací s automatickým vytvářením databázového schématu“ jsem vypracovala samostatně pod vedením vedoucího diplomové práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou všechny citovány v práci a uvedeny v seznamu literatury na konci práce.

Jako autor uvedené diplomové práce dále prohlašuji, že v souvislosti s vytvořením této diplomové práce jsem neporušila autorská práva třetích osob, zejména jsem nezasáhla nedovoleným způsobem do cizích autorských práv osobnostních a/nebo majetkových a jsem si plně vědoma následků porušení ustanovení § 11 a následujících autorského zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon), ve znění pozdějších předpisů, včetně možných trestněprávních důsledků vyplývajících z ustanovení části druhé, hlavy VI. díl 4 Trestního zákoníku č. 40/2009 Sb.

Brno ....

(podpis autora)

Poděkování

Ráda bych poděkovala panu Ing. Jiřímu Křížovi, Ph.D., Ph.D. za vedení mé diplomové práce, své rodině a přátelům za podporu při studiu.

Obsah

Úvod

V dnešní moderní době každá webová aplikace obsahuje tyto základní části: backend, frontend a databázi. Ve finální verzi se očekává, aby tato databáze byla klasickou relační databází s omezeními, právy, unikátními indexy a atd. Dalším požadavkem bývá jednoduchost, optimalizace a také snadnost vývoje. Diplomová práce bude pojednávat o návrhu postupu vývoje aplikace (softwarového produktu) spolupracující s chytrou databází (mezivrstva pro přepínání). Tato mezivrstva bude průběžně vytvářet schémata a omezení databáze. Tímto se zamezí stálému náhodnému manuálnímu měnění databáze.

Práce je rozdělena do čtyř kapitol. V první kapitole se budu zaobírat cílem práce a také jak tohoto cíle dosáhnout. To obnáší určení si metod postupu práce a zpracování uvedené problematiky.

Ve druhé kapitole se budeme zaobírat teoretickými východisky práce. Rozebereme si systém řízení báze dat (SŘBD), frontend, backend, přiblížím knihovnu AngularJS, a poté se budu zaobírat různými druhy databází a metodologií okolo nich.

V další kapitole budeme analyzovat současný stav. Nejprve začneme všeobecným úvodem do problematiky a poté se blíže podíváme na jednotlivá již existující řešení. Na konci této kapitoly provedeme shrnutí a vyvodíme závěry, se kterými budeme nadále pracovat.

V poslední kapitole budeme navrhovat konkrétní řešení uvedené problematiky. Tato kapitola začíná výběrem správné databáze, která musí splňovat dané podmínky. Poté už bude podrobněji rozepsán návrh řešení s podrobným popisem funkce mezivrstvy. Ke konci této kapitoly se zaměřím na instalaci a implementaci mezivrstvy.

V závěru zrekapitulujeme dosažené výsledky a zhodnotíme přínos této práce. Součástí závěru budou také možná rozšíření diplomové práce.

1 Cíl práce, metody a postupy zpracování

Cílem diplomové práce je vytvoření mezivrstvy starající se o automatické vytvoření databázového schématu na základě použitých dat, nikoli definovaných modelů. Tato vrstva musí být použitelná jak z frontendu (pro účely vytvoření uživatelského rozhraní), tak z backendu (pro účely přidání business logiky). Aby bylo tohoto cíle dosaženo, je nutné zanalyzovat dostupná řešení a v případě, že žádné nebude vyhovovat, buď použít elementy těchto řešení k vytvoření vlastního nebo vytvořit celé nové řešení. Dalším cílem je zachovat konceptuální jednoduchost i na úkor kompletnosti.

Na začátku vývoje by mezivrstva dovolila uložit jakékoliv informace do libovolných kolekcí bez typové kontroly, bez kontroly existence, apod. V průběhu vývoje by detekovala změny a upozorňovala na pravděpodobné nesrovnalosti (překlepy). Na konci vývoje by bylo možno tuto mezivrstvu nahradit relační databázi pro kterou vygeneruje schéma.

Životní cyklus výsledné mezivrstvy je více rozepsán v kapitole 4.

1.1 Metody a postupy zpracování

Diplomová práce je rozčleněna do pěti základních, na sebe navazujících částí:

- **Teoretická část** - obsahuje úvod do konceptu, skládá se převážně z citací a vysvětlení dané problematiky.
- **Analýza** - je z větší části originální obsah, analyzuje na základě subjektivního pohledu schopnosti a omezení jednotlivých řešení. Všechny informace jsou převzaty z praxe.
- **Návrh vlastního řešení** - aplikuje závěry dosažené z analýzy již existujících řešení, adresující nalezené nedostatky a poskytující schopnosti shledané potřebnými.
- **Implementace** - popisuje ilustrační implementaci vytvořeného návrhu
- **Závěr** - zhodnocuje celé řešení práce a navrhuje další rozšíření

Celkově práce vznikala v tomto pořadí: analýza dostupných řešení byla na prvním místě. Z ní se potom vedl celý návrh vlastního řešení. Teoretická část vznikala v průběhu zpracovávání této práce, když se objevila potřeba definovat nový pojem. Kapitola implementace vznikala též průběžně v rámci implementace, aby popis implementace skutečně seděl na vlastní implementaci. Až na úplný konec se sepisoval úvod a závěr práce.

2 Teoretická východiska práce

2.1 Systém řízení báze dat

Systém řízení báze dat, dále jen SŘBD či DBMS – podle anglického database management system, je kolekce vzájemně souvisejících dat a sad programů pro přístup k těmto datům. Kolekce těchto dat, které jsou často označovány jako databáze, obsahují údaje týkající se podniku. Hlavním cílem SŘBD je poskytnout způsob jak ukládat a načítat informace o databázi, což je zároveň pohodlné a efektivní.

Databázové systémy jsou navrženy tak, aby zvládly velké množství informací. Správa těchto dat zahrnuje jak vymezení struktury pro ukládání informací, tak poskytuje mechanismy pro manipulaci s informacemi. Kromě těchto věcí musí databázový systém zajistit bezpečnost uložených informací, a to i navzdory pádu systému nebo při pokusu o neoprávněný přístup. Mají-li se informace rozdělit mezi více uživateli, musí být systém schopný předejít možným chybným výsledkům. Vzhledem k tomu, že informace jsou tak důležité v mnoha organizacích, počítačoví odborníci vyvinuli velkou skupinu konceptů a technik pro správu dat [26].

2.2 Frontend

Frontend je součástí webových stránek, které můžeme vidět nebo s nimi komunikovat pomocí API. Toto rozhraní se obvykle skládá ze dvou částí: webový design a uživatelsky viditelné části aplikací.

Nástroje, které spolupracují s tímto rozhraním, jsou nejčastěji Photoshop a Fireworks, ve kterých se vytváří design aplikací. Aby bylo možné vytvořenou grafiku nasadit na webové stránky, je potřeba pomocí HTML, CSS, JavaScriptu nebo jQuery vytvořit kód, který je pak vykonán internetovým prohlížečem. Ten obstarává věci jako písma, drop-down menu, tlačítka, přechody, posuvníky, kontaktní formuláře a další [28].

2.3 Backend

Backend se obvykle skládá ze tří částí: server, aplikace a databáze. Pokud si uživatel rezervuje letenku nebo kupuje lístky na koncert, obvykle otevře webové stránky a už komunikuje s frontendovou částí. Jakmile ale jednou zadá informace, které se mají uložit do aplikace, tak se tyto informace vytvoří a uloží na serveru. Uživatel si může myslet, že se jedná o gigantickou databázi v Excelovských tabulkách na jeho počítači, ale ve skutečnosti je tato databáze uložena na serveru někde daleko v Arizoně [28].

Backend je tedy část aplikace, která zároveň obsluhuje všechny uživatele, kteří ji zrovna používají. Dále mezi nimi zprostředkovává synchronizaci, udržuje společná data a zajišťuje celkovou konzistenci.

2.4 AngularJS

Oficiální úvod do AngularJS jej popisuje jako client-side technologii, napsanou celou v JavaScriptu. Používá dlouho zavedené webové technologie (HTML, CSS a JavaScript), aby zjednodušil a zrychlil vývoj webových aplikací.

Jedná se o framework, který je primárně užíván k tvorbě single-page webových aplikací. AngularJS zjednodušuje tvorbu moderních a interaktivních webových aplikací zvýšením úrovně abstrakce mezi vývojářem a obvyklými úkony při tvorbě webů.

AngularJS tým jej popisuje jako ”strukturální framework pro dynamické webové aplikace”. Velmi zjednodušuje vytváření webových aplikací a také komplexních aplikací a obstarává pokročilé vlastnosti, na které si uživatelé v moderních aplikacích již zvykli. Například: Oddělení aplikační logiky, datových modelů a šablon, AJAXové služby, dependency injection, historie v prohlížeči (aby záložky i tlačítka zpět/vpřed fungovaly jako v tradičních webových aplikacích), testování a více [19].

2.4.1 Dependency injection

Obecně existují pouze tři možnosti, jak objekt může získat svoje závislosti:

1. Můžeme závislost vytvořit vnitřně tam, kde ji potřebujeme.
2. Můžeme vyhledávat nebo se na ni odkazovat do globální proměnné.
3. Můžeme závislost vytvořit zvnějšku a předávat tam, kde je vyžadována.

DI se zabývá touto třetí možností. Jde o návrhový vzor, který umožňuje zbavení se závislostí pevně v kódu, což nám umožňuje jejich odstranění, či změnu za běhu. Tato možnost změny závislosti za běhu nám umožňuje vytvářet izolovaná prostředí, což má využití například při testování.

Specificky vzor získává nebo dostává informace o požadovaných závislostech od svých uživatelů (tříd), získává či vytváří tyto závislosti a předává je konstruktoru při vytvoření příslušného objektu.

Během psaní komponent závisejících na ostatních objektech, či knihovnách, popíšeme jejich závislosti. Poté za běhu injector[1] vytvoří instance příslušných závislostí a předá je komponentům, které na nich závisí.

Zatímco nějaký náš modul nabíhá (za běhu aplikace), injector je zodpovědný za vlastní instanciaci instance objektu a předání těchto závislosti tomuto objektu [19].

2.5 Objektově relační mapování

Většina moderních vyvíjejících se business aplikací používá objektové technologie (jako je Java nebo C#) k vytvoření aplikačního softwaru a relační databáze pro ukládání dat [17]. Objektově relační mapování, dále jen ORM, je pokusem spojit oblasti objektového a relačního světa,aby spolu mohli lehce komunikovat v obou směrech. Pokud se podíváme blíže do nějaké aplikace, můžeme vidět, že aplikace je více či méně postavena kolem datového modelu. Ostatní databázové technologie v průběhu času postupně přišly a odešly, ale relační databáze jsou v databázových technologiích jasným vítězem [4].

Hlavním cílem ORM je pomoci vaší aplikaci dosáhnout persistenci, což znamená, že chceme, aby data (objekty) naší aplikace přežila ukončení aplikačního procesu. Specificky, ORM reší persistenci pomocí relačnich databází. Objektové a relační modely spolu snadno

[1] Injector – komponenta aplikace, která se stará o samotné přidání závislostí

nespolupracují (tzv. object-relational impendance mismatch), protože relační databáze reprezentují data jako tabulky, zatimco objektově orientované jazyky je reprezentují jako propojené grafy objektů. Různá ORM tento problém řeší různými způsoby [23].

2.6 Databázová migrace

Při vývoji aplikací závislých na databázi, vývojáři typicky zároveň píšou zdrojový kód a zároveň vyvíjí databázové schéma. Kód má typicky pevná očekávání jaké sloupce, tabulky a omezení jsou přítomná v databázovém schématu, kdykoli potřebuje komunikovat s databází, takže jen verze softwaru a databázové schéma, se kterým byl tento software vyvinut, jsou plně kompatibilní.

V oboru testování, zatímco vývojáři můžou mockovat[2] přítomnost kompatibilní databáze za účelem unit testu, jakákoli vyšší úroveň testování (např. integrační či systémové) očekává, že vývojáři budou testovat svoji aplikaci proti místní či vzdálené testovací databázi, která je schématem kompatibilní s právě testovanou verzí zdrojového kódu. V složitějších aplikacích může být migrace sama testována (tzv. migrační test).

S technologií migrace schémat již nemusí datové modely být plně navrženy dopředu a je snazší je přizpůsobit měnícím se požadavkům na projekt během celého životního cyklu vývoje softwaru [29].

2.7 Databázové schéma

Logická vrstva nebo logický model zahrnuje jednu ze dvou abstraktních vrstev v databázi: fyzická vrstva má konkrétní místo ve složkách operačního systému, kde logická vrstva existuje pouze jako abstraktní datová struktura složená z fyzické vrstvy. SŘBD transformuje tato data do datových souborů jednotné struktury. Tato vrstva se často nazývá **databázové schéma**, název se používá pro kolekci všech datových položek uložených ve vlastní databázi nebo patřící databázovému uživateli [21].

[2]Mock objekt – simulace objektu, která kopíruje chování reálného objektu kontrolovaným způsobem (např. nepřistupuje do reálné databáze)

Databázové schéma je tedy způsob jak logicky seskupit objekty jako tabulky, pohledy, uložené procedury, apod. Můžeme o něm přemýšlet jako o kolekci objektů. Konkrétnímu uživateli je možno přiřadit práva k jednotlivým schématům tak, že uživatel může přistupovat jen k objektům, ke kterým jsou autorizováni. Schémata mohou být vytvářena a měněna v databázi a uživatelé mohou mít přidělen přístup k schématu. Schéma může být vlastněno libovolným uživatelem a každé takové vlastnictví je přenosné [22].

2.8 NoSQL databáze

V této sekci se budu zabývat různými NoSQL databázemi, které spojuje jedna věc a sice to, že tyto databáze nepoužívají tradiční tabulková schémata. NoSQL pojmenování je velice obecné, a tedy je možné rozdělení do několika podskupin: Key-value databáze, dokumentové databáze, sloupcové databáze a grafové databáze. Níže se budu zaobírat Key-value databázemi a dokumentovými databázemi, které jsou známější.

2.8.1 Key-value databáze

Key-value (KV) úložiště je nejjednodušším modelem, kterým se můžeme zabývat. Jak už napovídá název, ukládat KV dvojice klíčů k hodnotám je v podstatě ten samý způsob, jako mapování v kterémkoli populárním programovacím jazyce. Některé KV implementace umožňují komplexní typy hodnot, jako jsou hashe nebo seznamy, ale není to pravidlem. Dále mohou poskytnou způsoby iterace klíče, ale toto opět může být přidaný bonus.

Souborový systém by mohl být ukládán jako klíč-hodnota za předpokladu, že cesta k souboru je klíčem a obsah souboru je hodnota. Jelikož jsou požadavky na KV tak malé, databáze tohoto typu může být neuvěřitelně výkonná v řadě situací. Avšak pokud bychom se složitě dotazovali do databáze a požadovali agregace, tak by toto neplatilo. Stejně jako u relačních databází jsou i zástupci těchto databází volně ke stažení [24].

Redis

Jedná se o open source databázi s key-value úložištěm. Redis je často používané jako lepidlo, to znamená, že se používá víceméně jako rychlá spojovací vrstva. Podporuje složité datové struktury (hash, seznamy, množiny a setříděné množiny, bitmapy, atd.).

V nastavení této databáze si můžeme zvolit čas pro vypršení platnosti dat, který je vhodný pro cachovani. Dalším mechanismem této databáze je publish-subscribe [3] [5].

Riak

Tato databáze používá distribuované key-value úložiště s tím, že všechny uzly jsou si rovnocenné a můžou být kdykoli bez problému odstraněny. V případě konfliktu ukládaní či aktualizovaní informaci, je řešení konfliktu necháno na čtenáři. Riak podporuje indexování uživatelských metadat a je vhodně škálovatelný. Sám o sobě poskytuje REST rozhraní, to znamená, že je možná komunikace přes HTTP dotazy [3].

2.8.2 Dokumentové databáze

Dokumenty jsou hlavní myšlenkou v dokumentových databázích. V databázi jsou ukládány a vyhledávány dokumenty, které mohou být napsány v XML, JSON, BSON či jiném formátu. Tyto dokumenty jsou samopopisný, hierarchický strom datové struktury, které mohou sestávat z map, kolekcí a skalárních hodnot. Dokumentová databáze ukládá dokumenty do části hodnot klíč-hodnota, kde hodnota je přezkoumatelná.

```
{
"firstname": "Martin",
"likes": [ "Biking",
           "Photography" ] ,
"lastcity": "Boston",
"lastVisited": null
}
```

Obr. 2.1: Ukázka kódu dokumentu (Zdroj: Převzato z [25])

[3] Publish-subscribe – toto spojení se nepřekládá do češtiny, jeho význam je skupinová notifikace přihlášených odběratelů na libovolném množství kanálů

```
{
"firstname": "Pramod",
"citiesvisited": [ "Chicago", "London", "Pune", "Bangalore" ] ,
"addresses":
    [{
        "state": "AK",
        "city": "DILLINGHAM",
        "type": "R"
    } ,
    {
        "state": "MH",
        "city": "PUNE",
        "type": "R"
    }] ,
"lastcity": "Chicago"
}
```

Obr. 2.2: Ukázka kódu dokumentu (Zdroj: Převzato z [25])

Při pohledu na dokumenty můžeme vidět, že jsou si podobné, ale mají rozdílné názvy atributů. Schéma dat se může lišit mezi dokumenty, ale tyto dokumenty mohou patřit do stejné kolekce, na rozdíl od relačních databází, kde každý řádek v tabulce musí být stejný jako v schématu. Některé atributy si mohou být podobné a ve stejný čas mohou tyto atributy být v jednom dokumentu a v jiném zase nemusí existovat.

Tato odlišná reprezentace dat není stejná jako v relačních databázích, kde každý sloupec musí být definovaný, a pokud neobsahuje žádná data, je označen jako prázdný nebo nastaven na hodnotu null. V dokumentech nejsou k dispozici žádné prázdné atributy, pokud daný atribut není nalezen, předpokládáme, že není nastaven nebo nenáleží dokumentu. Dokumenty také umožňují vytvoření nových atributů aniž by bylo potřeba je definovat či již existující dokumenty měnit [25].

CouchDB

Jedná se o dokumentovou databázi ukládající JSON dokumenty s REST rozhraním, která je vhodná pro malé nebo velké projekty. Podporuje autentizaci uživatelů či práva ke kolekcím, ale nevyžaduje ji. Výhodou této databáze je i uživatelsky přívětivé rozhraní. Konflikty jsou řešeny tak, že každý dokument má nejen svůj identifikátor, ale obsahuje i identifikátor jednotlivých revizí a při aktualizaci je nutné předat jak identifikátor záznamu, tak identifikátor revize. Řešení konfliktu je tedy na zapisovateli [12].

MongoDB

MongoDB je dokumentová databáze, která ukládá informace do JSON objektu uložených v binární formě. Je to výborná volba pro stále se vyvíjející skupinu webových projektů, jejichž nároky na úložiště mohou dosahovat velkých rozměrů. Podporuje i hledání podle libovolných políček a vyhledávaní podle zadaného výrazu. Škálování závisí na rozdělení dat podle libovolné hodnoty indexu (sharding) [20].

2.9 Objektové a relační databáze

2.9.1 Objektové databáze

Vývoj objektově-orientovaných databází byl pokus integrovat schopnosti moderních programovacích jazyků v oblasti modelování komplexních dat a principů softwarového inženýrství, s charakteristikami databázových technologií jako je persistence, koordinace a ochrana. Samozřejmě cílem je dosáhnout výhod obou.

Pro aplikace vyžadující podporu databáze jsou objekty přirozenou jednotkou pro zamykání, autorizaci, verzování a pod. Objektově-orientovaný model také nabízí další příležitosti pro vylepšení podpory aplikací na straně databázového systému [16].

Mezi zástupce objektově-orientovaných databází patří: Caché, GemStone, ITASCA, ObjectStore, Objectivity/DB.

2.9.2 Relační databáze

Relační databáze je kolekce datových položek organizovaných jako soubor formálně popsaných tabulek, ze kterých mohou být data přístupná nebo poskládaná různými způsoby, aniž by bylo nutné reorganizovat databázové tabulky. Tvůrcem relační databáze byl E.F. Codd v roce 1970 [26].

Vhodné:

Vzhledem ke své struktuře dávají relační databáze smysl, když forma dat je předem známa, ale není nezbytně známo, jak budou tato data používána. Neboli je třeba dopředu vyřešit organizační komplexitu za účelem následné flexibility v dotazování. Mnoho business problémů je snadné modelovat tímto způsobem od objednávek po dodávky a od inventáře k nákupním košíkům. Nemusíte dopředu vědět, jak se na tato data budete chtít později dotazovat – *Kolik objednávek jsme zpracovali během února?* – ale tato data jsou ve své podstatě pravidelná a vyžadování této pravidelnosti je užitečné.

Méně vhodné:

Pokud jsou vaše data velmi variabilní nebo hluboce hierarchická, relační databáze nejsou nejlepší variantou. Protože je nutné specifikovat schéma předem, data která projevují vysokou variaci od záznamu k záznamu budou problematická. Uvažujte návrh databáze popisující všechny tvory v přírodě. Vytvořit úplný seznam všech vlastností, o kterých je třeba mít přehled (má chlupy, počet nohou, klade vejce, atd.), by bylo neprůchodné. V takovém případě je vhodnější použít databázi, která klade předem méně omezení na data, která je do ní možná vložit [24].

MySQL

Klasická relační databáze obsahující podporu ACID transakcí, odstranění redundancí dat, automatické konverze typů a dalších známých vlastností [8]. Myslím, že není potřeba blíže popisovat.

SQLite

Přenosná relační databáze se základními vlastnostmi. Není to klasická serverová databáze, jedná se o aplikační databázi ve formě knihovny. Nepodporuje replikaci ani současný přístup ze dvou procesů ve stejný čas, ale je velice rozšířena, neboť všechny mobilní aplikace ji používají [6].

2.9.3 Objektově-relační databáze

Objektově-relační databáze jsou systémy pro správu dat podobný relační databázi, ale s objektově-orientovaným databázovým modelem: objekty, třídy a dědičnost jsou podporovány v databázovém schématu a v dotazovacím jazyku.

Objektově-relační databáze může být definovaná jako něco, co poskytuje kompromis mezi relačními databázemi a objektově orientovanými databázemi. Navíc jsou, stejně jako v čistě relačních systémech, podporovaná rozšíření datového modelu uživatelskými datovými typy a metodami.

V objektově-relačních databázích je přístup v zásadě stejný jako v relačních databázích: data přebývají v databázi a jsou manipulovány dohromady pomocí dotazu v nějakém dotazovacím jazyce. Na druhém konci jsou objektově-orientované databázové systémy, ve kterých je databáze de facto úložiště persistentních objektů pro software napsaný v nějakém objektově orientovaném programovacím jazyce, s API pro ukládání a načítání objektů a slabou či žádnou podporu pro hledání a složitější dotazy [26].

PostgreSQL

Tato databáze je databází relační, která podporuje mnohá moderní rozšíření. Kromě obecně očekávaných vlastností relační databáze, PostgreSQL umožňuje ukládání JSON objektu do políček (položek) a následné vyhledávaní v těchto poličkách. Dále umí indexovat vypočítávané hodnoty, což znamená, že není potřeba duplikovat vypočítávanou hodnotu v tabulce jen kvůli indexaci. Výhodou této databáze je vyhledávání podle specifických požadavků uživatele, například: vyhledávání podle kořene slova, vyhledávání podle nepřesného zadání hledaného slova či textu apod [24].

PostgreSQL tabulky mohou dědit z jiných tabulek. Jejich metody jsou děděné ale implicitní přetypování nejsou zřetězena, a jejich indexy děděny nejsou. To umožňuje vytvářet hierarchie objektové dědičnosti v PostgreSQL. Vícenásobná dědičnost je možná, oproti Oracle, DB2 i Informix, které všechny podporují pouze jednonásobnou dědičnost.

Tabulková dědičnost je pokročilý koncept a obecně je asi vhodné ji použít spíše v oblastech, kde je typičtěji používaná, jako například table partitioning[4]. Objektově-relační část PostgreSQL je postavena z jednotlivých pokročilých objektově-relačních vlastností, což umožňuje vytvářet složitější objektově-relační systémy nad touto databází, a příliš se nepodobá dnešním objektově orientovaným programovacím prostředím [27].

[4]Table partitioning – rozdělování fyzické tabulky do několika, podle určité vlastnosti dat v ní uložených, při zachování virtuálního pohledu na celek

3 Analýza současného stavu

Abychom mohli navrhnout funkční systém, musíme se nejdříve zaměřit na to, jaká jsou již dostupná jiná řešení. Je třeba zanalyzovat tato řešení, poté buď vybrat nejlepší variantu, anebo se poučit z nedostatků v těchto řešeních a vytvořit vlastní řešení.

V architektuře každé aplikace používající centrální databázi je funkcionalita systému rozdělena mezi dva typy modulů. Klientský modul je typicky navržen tak, aby běžel na uživatelově pracovní stanici nebo osobním počítači. Typicky aplikační programy a uživatelská rozhraní, která přistupují k databázi, pracují v klientském modulu. Druhým modulem je server modul sloužící k ukládání, přístupu a hledání dat [9].

Uživatel se ve většině případů setkává pouze s frontendovou částí aplikace, kde není důležitá vnitřní funkcionalita a ani způsob uložení dat. Struktura dat se odvíjí z potřeb frontendu a tedy backendová část průběžně mění svoji logiku i strukturu dat. Změny struktury dat probíhají ve většině případů změnou schématu databáze, migrací dat na nové schéma a úpravou již existujícího kódu. Vzhledem k časté frekvenci takovýchto změn jde o nezanedbatelné množství času a práce a tím i větší pravděpodobnost způsobení nějaké chyby při této změně. Bylo by proto vhodné nalézt anebo vyvinout nástroj, který by tyto problémy řešil.

Tento problém má svou obdobu i v oblasti návrhu uživatelského rozhraní, kde se nejčastěji řeší pomocí mock-up[5] modelů. Nejprve se vytvoří nefunkční, ale vizuálně věrné prostředí, které může uživatel zhodnotit, a poté se podle tohoto prostředí implementuje finální řešení.

V této kapitole se budu zabývat již existujícími řešeními tohoto problému. Objasním jejich funkčnost a bezpečnost. Na konci této kapitoly vše zhodnotím a dojdu k závěru, ze kterého budu vycházet v následující kapitole.

[5]Mock up – vytvoření ilustrační verze uživatelského rozhraní s omezenou funkcionalitou

3.1 Relační databáze bez pomocných nástrojů

Nejjednodušším řešením databázové vrstvy jsou relační databáze bez pomocných nástrojů. Na začátku práce je potřeba vytvořit databázové schéma, vložit jej do databáze a začít vyvíjet aplikaci. V průběhu vývoje je možné se dotazovat do databáze, to znamená ukládat a načítat data, ale při jakékoli nekonzistenci databázového schématu s představou uživatele je nutné toto schéma přeměnit či upravit, což může negativně ovlivnit chod celé aplikace. Kromě změny schématu je tedy ještě nutné udělat změnu kódu komunikujícího s databází. Zároveň, pokud je produkt provozován na vícero serverech, je nutno tuto úpravu provést všude.

K databázi je typicky přistupováno několika způsoby:

Webové rozhraní - webová stránka s přístupovým formulářem.

Příkazový řádek - přístupové údaje se předávají jako parametry, na vstupu, nebo načtením z konfiguračního souboru.

Knihovny (pro daný programovací jazyk) - přístupové údaje se načítají z konfiguračního souboru aplikace a předávají se knihovně při vytvoření spojení.

Z hlediska zabezpečení databáze každý z těchto druhů přístupů používá připojení k pojmenované rouře či TCP portu. Spojení je typicky šifrované a autentizované pomocí kombinace jména a hesla. Podporované způsoby autentizace, šifrování přenosu a možnosti granularity práv závisí na databázi, ale relační databáze měly mnoho času na vývoj a implementaci všech těchto vlastností.

3.1.1 Výhody

Největší výhodou tohoto řešení jsou minimální požadavky na znalosti. Není potřeba se učit složité frameworky či nástroje na správu databáze, postačí základní znalosti SQL dotazů. Také je to nejflexibilnější řešení nezávislé na konkrétní variantě SQL databáze, programovacím jazyku či platformě.

3.1.2 Nevýhody

Jak již bylo zmíněno výše, při každé změně schématu databáze je nutné všude tuto změnu provést manuálně a přizpůsobit tomu kód, který komunikuje s databází. Pokud v průběhu

změn není vytvořen dump[6] databáze, není možné se vrátit k předchozí verzi databáze a tedy nelze spolehlivě obnovit předchozí verzi aplikace, která je typicky vázaná na verzi databázového schématu.

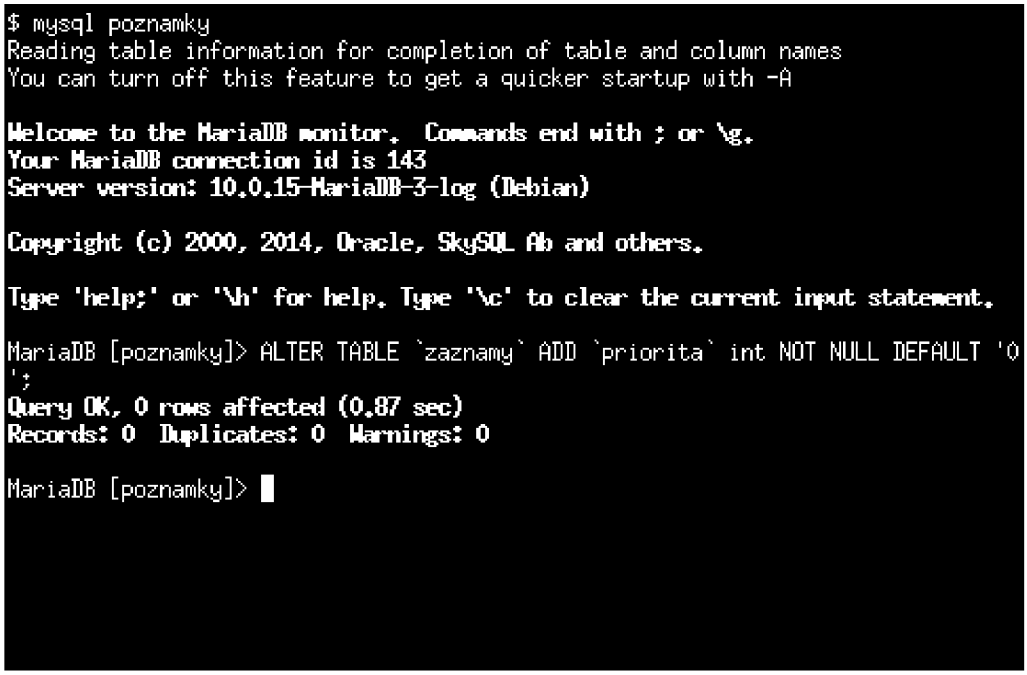
3.1.3 Příklad použití

Pro účely této analýzy navrhnu jednoduchou aplikaci připomínající úkolníček sloužící jako ukázkový příklad, na kterém budu ukazovat vlastnosti jednotlivých nástrojů. Nejdříve vytvoříme tabulku. Průběh její tvorby je vidět na obrázku níže 3.1.

Obr. 3.1: Program adminer pro práci s databází (Zdroj: Vlastní tvorba)

Pomocí SQL příkazů je možné měnit data v tabulkách nebo i celou strukturu tabulek. Existuje mnoho dalších nástrojů, které tyto příkazy pro specifické případy (vytvoření tabulky, změny dat, pohledy a další) generují samy. Tyto příkazy můžeme uložit do souboru a poté je manuálně nahrát do databáze nebo je ručně zadávat do příkazové řádky. Na obrázku 3.2 je ukázka úpravy záznamu v tabulce pomocí příkazové řádky.

[6]Dump – záloha databáze



Obr. 3.2: Úprava tabulky pomocí příkazového řádku (Zdroj: Vlastní tvorba)

3.2 Nástroje nevyžadující schéma databáze

Nástroje zmíněné v této sekci se vyhýbají problémům s migracemi využitím databází (objektových a dokumentových). Nevyžadují připravené schéma, a tedy není nutné mít speciální nástroje na jejich vytvoření a případné změny. Níže budou zmíněny dva různorodé příklady, které můžeme zařadit do této sekce. Nástroj Jalapeño můžeme považovat za reprezentanta všech objektových databází, zatímco Amazon SimpleDB reprezentuje cloudové a key-value databáze. Ač jsou oba tyto nástroje rozdílné, přicházejí různými způsoby řešení k velmi podobnému výsledku.

3.2.1 Jalapeño

Jalapeño je perzistentní Java knihovnou, kterou lze lehce zapojit bez nastavování mapování. Mnoho aplikací používá standardní relační databáze, které vyžadují formu objektově-relačního mapování k uložení Java objektů. Vzhledem k omezením objektově-relačního mapování se Jalapeño používá s databází Caché, což je objektová databáze, která byla původně navržena pro zdravotnictví a telekomunikace. Využití objektové databáze umožňuje ukládat objekty přímo, bez složitých transformací.

Caché podporuje autentizaci pomocí systému Kerberos, nativního přihlašování operačního systému, LDAPu, Caché loginu i vlastní implementace. Práva se nastavují pomocí rolí a je možné je nastavit pouze na úrovni databáze (a jiných zdrojů, nepodporuje však práva na úrovni kolekcí či objektů). Při spojení se vzdálenou databází podporuje SSL šifrování.

Jalapeño nabízí tyto výhody [7]:

Snadná perzistence Java objektů – Klasické Java objekty mohou být změněny v perzistentní[7] objekty. Jalapeño načítá definice tříd, analyzuje je pomocí introspekce[8] a automaticky vytvoří příslušné schéma v objektové databázi. Toto schéma může využívat objektově-orientovaných vlastností jako je dědičnost, kolekce, vztahy a komplexní datové typy.

Zachované vztahy mezi objekty – Uvnitř databáze jsou perzistentní objekty ukládány jako pravé objekty zachovávající atributy, vztahy a jiné objektově-orientované vlastnosti. Uvnitř aplikace se přistupuje přímo k třídám přes Jalapeño API, které poskytuje metody pro operace jako navázání spojení s databází, získávání objektů z databáze, ukládání objektů a dotazování se do databáze.

Standardní SQL přístup – Jalapeño API je implementováno jako nadmnožina JDBC[9] API. Na všechny objekty uvnitř databáze se dá automaticky nahlížet jako na relační tabulky pomocí standardního JDBC. Pro komunikaci s databází se není nutné učit nové API (nejsou ale potom k dispozici výše zmíněné objektově-orientované vlastnosti).

Nezávislost na databázi – Ačkoliv Jalapeño nejlépe pracuje, pokud je používáno s objektovou databází Caché, objektově databázové schéma může být exportováno do odpovídajícího relačního schématu.

Navzdory výše zmíněným výhodám, Jalapeño má i své nevýhody. Největší nevýhodou je závislost na jedné proprietární[10] databázi bez možnosti nahradit ji alternativním řešením. Další nevýhodou je špatná integrace s jinými programovacími jazyky, než je Java.

[7] Perzistentní objekty – objekty, které díky databázi zůstanou zachovány i po ukončení programu.

[8] Introspekce – možnost podívat se za běhu programu na seznam metod a atributů dané třídy.

[9] Java Database Connectivity – databázově nezávislá vrstva pro komunikaci mezi Javou a různými databázemi.

[10] Proprietární software je takový software, kdy je uživatel omezen platností licence.

Příklad použití

Pro Jalapeño nevytváříme ani tabulku, ani schéma, ale vytvoříme přímo třídu a přidáme anotaci[11], viz obr. 3.3. Při změně třídy *Zaznam* je nutné spustit schema builder, který automaticky přepracuje databázi tak, aby odpovídala změnám v souboru.

```
1  import com.intersys.pojo.annotations.CacheClass;
2
3  @CacheClass(name="Zaznam",primaryKey="id",sqlTableName="zaznamy")
4  public class Zaznam {
5      public int id;
6      public String nazev;
7      public String obsah;
8      public int priorita;
9  }
```

Obr. 3.3: Třída definující Cache schéma (Zdroj: Vlastní tvorba)

3.2.2 Amazon SimpleDB

Amazon SimpleDB je sice cloudovou databází, ale nyní se zaměřím na to, že se jedná také o key-value databázi, a tedy nepotřebuje žádnou podporu schématu a ani žádný ORM systém. Je velmi dostupným a flexibilním nerelačním úložištěm dat, který přebírá práci databázové administrace. Jednotlivé položky dat se lehce ukládají a získávají pomocí dotazovací webové služby, Amazon SimpleDB udělá zbytek.

Amazon AWS[12] autentizace je jediná podporovaná. Pro jednoho uživatele poskytuje jen jeden klíč, ale díky integraci s AWS Identity and Access Management je možné dát uživatelům v rámci AWS účtu přístupy k různým SimpleDB doménám v rámci daného účtu. SimpleDB poskytuje dotazování do databáze přes HTTP, navíc je možné použít i HTTPS pro šifrování.

[11] Anotace – metadata, která je možné přidat k třídě, metodě či atributu

[12] Amazon Web Services – kolekce Amazon API pro všechny služby

Amazon SimpleDB na pozadí vytváří a spravuje několik geograficky distribuovaných replik dat za účelem vysoké dostupnosti a trvalosti dat. Je možné změnit datový model za běhu a data jsou automaticky indexována. S Amazon SimpleDB je možné soustředit se na vývoj samotné aplikace bez nutnosti řešit infrastrukturu, uvedení do chodu, dostupnost, správu, schémata a indexy. Níže popíšu výhody této databáze [1]:

Bezúdržbovost – Amazon SimpleDB automaticky spravuje vytvoření infrastruktury, software i hardware a řeší replikaci, indexaci a optimalizaci databáze.

Vysoká dostupnost – Data jsou vysoce dostupná použitím nejblíže umístěného serveru. V případě selhání jedné kopie jsou vždy k dispozici další.

Flexibilita – Je možné zvolit si mezi plně konzistentním a eventual consistency[13] čtením, což umožní vybrat si mezi rychlostí a konzistencí.

Snadná použitelnost – Rychlé přidávání dat a snadné čtení a úpravy pomocí jednoduchého API.

Zabezpečení – Komunikace probíhá přes HTTPS tak, aby došlo k zajištění bezpečné a šifrované komunikace mezi aplikací a databází, navíc je s použitím jiných Amazon služeb možné nastavit uživatele nebo skupiny s různými úrovněmi přístupu k datům a operacím.

Hlavní nevýhodou tohoto nástroje je závislost na službách Amazonu, a proto ji nelze používat v offline módu. Pokud dojde k výpadku služeb, není možné dostat se k těmto datům. Vzhledem k velmi volným požadavkům na strukturu dat je náročnější s nimi pracovat v staticky typovaných jazycích (Java).

Příklad použití

V nástroji Amazon SimpleDB nastavujeme veškeré hodnoty pomocí webového nebo REST API rozhraní. Nejprve je nutné vytvořit Domain (databázi) a v ní Item (položky). Funkce *PutAttributes* změní nebo přidá pouze Attribute (atribut) a nepřepíše při každé změně celý objekt. Zajímavostí této funkce je, že funguje i na neexistujících položkách, což znamená, že není potřeba nejprve vytvořit položku. Při zavolání této funkce na nějakou položku s novým atributem se tento atribut vytvoří. Další funkcí je *DeleteAttributes*, která dokáže podle jména nebo podle hodnoty odstranit atributy. Níže na obrázcích si ukážeme vzorové

[13] Eventual consistency – je možné přeložit jako časem konzistentní

requesty, kde vytváříme objekt s barvou, velikostí a cenou. Barva a velikost se přidávají, pokud již existují, ale cena se přepíše.

```
https://sdb.amazonaws.com/
?Action=PutAttributes
&Attribute.1.Name=Color
&Attribute.1.Value=Blue
&Attribute.2.Name=Size
&Attribute.2.Value=Med
&Attribute.3.Name=Price
&Attribute.3.Value=0014.99
&Attribute.3.Replace=true
&AWSAccessKeyId=[valid access key id]
&DomainName=MyDomain
&ItemName=Item123
&SignatureVersion=2
&SignatureMethod=HmacSHA256
&Timestamp=2010-01-25T15%3A03%3A05-07%3A00
&Version=2009-04-15
&Signature=[valid signature]
```

Obr. 3.4: Ukázka API requetu, který přidává atribut (Zdroj: Převzato z [2])

```
<PutAttributesResponse>
  <ResponseMetadata>
    <RequestId>490206ce-8292-456c-a00f-61b335eb202b</RequestId>
    <BoxUsage>0.0000219907</BoxUsage>
  </ResponseMetadata>
</PutAttributesResponse>
```

Obr. 3.5: Ukázka odpovědi na předchozí request (Zdroj: Převzato z [2])

3.3 Cloudová řešení

V této sekci se budu zabývat full-stack[14] cloudovými nástroji, kde infrastrukturu spravuje poskytovatel.

Mezi nejznámější zástupce patří Cloud9, OpenShift a Meteor. Nejvíce se budu ale zabývat nástrojem Meteor, protože ostatní dva zástupci nabízejí pouze PaaS[15] a jejich způsob přístupu k databázi se nijak neliší od manuální správy, případně se tento přístup může řešit automatizovaně použitím nějakého frameworku.

3.3.1 Meteor

Klíčová myšlenka Meteoru je, že vše by se mělo chovat identicky jak v prohlížeči, tak na serveru. Odpadá tedy rozlišení mezi kódem pro server a kódem pro klienta, což se velmi dotýká databázové vrstvy. Meteor se skládá ze dvou částí: knihovny modulů a nástroje příkazové řádky.

Aplikace Meteoru jsou kombinací klientského JavaScriptu, který běží v prohlížeči nebo v mobilní aplikaci, serverového JavaScriptu, který běží na Meteor serveru pod Node.js, a podpůrných HTML šablon, CSS pravidel a statických zdrojů. Meteor automatizuje vytváření balíčků a přenos těchto komponent.

[14]Full-stack – platforma se dokáže postarat jak o databázi, servery tak i o uživatelské rozhraní.

[15]Platform as a Service – platforma jako cloudová služba

Meteor umožňuje vytvářet distribuovaný klientský kód stejně snadno jako komunikovat s lokální databází. Je to čistý, jednoduchý a bezpečný přístup, který odstraňuje potřebu implementovat vzdálená volání, manuálně cachovat data na klientovi a opatrně dirigovat invalidační zprávy každému klientovi při změně dat. Klient a server sdílí stejné databázové API a stejný aplikační kód často běží na obou místech (např. validace dat). Zatímco kód běžící na serveru má přímý přístup do databáze, kód na klientu sice používá stejné API, ale data jsou dále zpracovávána serverem. Na rozdíl od tradičních přístupů k backendu, kdy se používá REST API, Meteor používá protokol DDP[16], který je založený na websocketech, což umožňuje, aby server aktivně informoval klienta o změnách, aniž by se klient sám dotazoval.

Každý klient si v paměti uchovává kopii části databáze, která je zrovna využívaná. Kvůli správě klientské cache server publikuje množiny JSON dokumentů a klient odebírá tyto množiny. Většina Meteor aplikací používá MongoDB, jehož API je na klientské straně emulováno knihovnou minimongo [13].

V Meteoru není databáze oddělena od aplikace (resp. nejde nijak spravovat), bezpečnost se tu řeší přidáním modulu správy uživatelů, odstraněním modulu zveřejňujícího vše a nastavením práv pro aplikační uživatele. Je podporováno mnoho způsobů autentizace, od kolekce uživatelů v databázi až po Facebook. SSL je podporováno jak pro HTTPS data, tak pro DDP protokol.

Příklad použití

Předpokládáme-li stejnou strukturu dat jako v předchozích příkladech, přidání sloupce priorita celé kolekci by mohlo vypadat následovně:

[16]Distributed Data Protocol – jednoduchý protokol pro získávání strukturovaných dat od serveru a přijímání průběžných aktualizací při změně těchto dat na serveru.

```
var Zaznamy = new Mongo.Collection('zaznamy');
Zaznamy.update({
    /* vsechny ktere nemaji zadnou prioritu */
    priorita: { $exists: false }
}, {
   /* nastavit prioritu na 0 */
   $set: { priorita: 0 },
}, {
    /* zmeni vsechny zaznamy, ne jen ten prvni */
    multi: true,
});
```

Obr. 3.6: Ukázka přidání sloupce (Zdroj: Vlastní tvorba)

Tento krok není potřeba, slouží pouze k ilustraci složitějších úprav, které by byly potřeba.

3.4 Nástroje podporující předpřipravené migrace

V této sekci se blíže podíváme na nástroje, které v sobě nemají obsažené žádné databáze a pouze s těmito databázemi spolupracují. Typicky využívají SQL databáze, je možné použít i další dostupné druhy databází, u kterých však hrozí riziko, že pro ně bude minimální podpora. Tyto nástroje umí verzování databáze pomocí migrací, ale tyto migrace musí programátor předpřipravit. K provedení migrací je potřeba manuálního spuštění, které provede pouze ty, co jsou neprovedené.

Všechny přístupy v této a následující sekci jsou nezávislé na databázi. Není tedy nutné se zabývat databázovou bezpečností. Zabezpečení si musí aplikace řešit samy, s různou mírou pomoci od příslušného nástroje a prostředí.

3.4.1 Node.js

Node.js je JavaScriptové prostředí, které umožňuje sdílení kódu mezi serverem a klientem a je ho možné rozšířit o různé frameworky či knihovny. Dříve byla velmi populární sada nástrojů LAMP[17], který slouží a sloužil k vytváření dynamických webových stránek. Nyní tuto variantu nahrazuje spojení Node.js s Express frameworkem, což tvoří spodní část MEAN[18] stacku.

Jedním z rozšíření je node-db-migrate, framework určený pro migrace databází aplikací pracujících na serveru Node.js. Tento nástroj neřeší generování schématu a ani rozpoznávání změn. Migrace si vytváří programátor sám, db-migrate pouze provede ještě neprovedené změny a umožňuje návrat k předchozí verzi. Existují frameworky podobné tomuto, jako například Patio a migrate, které se liší použitím ORM a syntaxí zápisu. Toto nejsou zásadní rozdíly, které by měly být zmíněny v této práci.

Příklad použití

Na obrázku 3.7 je ilustrováno vytvoření JavaScriptového souboru, ve kterém jsou obsaženy příkazy s dotazy do databáze.

```
1 exports.up = function(db, callback) {
2     db.connection.query("ALTER TABLE ‘zaznamy‘ ADD ‘priorita‘
3     int NOT NULL DEFAULT ’0’", callback);
4 };
5 exports.down = function(db, callback) {
6     db.connection.query("ALTER TABLE ‘zaznamy‘ DROP ‘priorita‘",
7     callback);
8 };
```

Obr. 3.7: Ukázka přidání sloupce priorita (Zdroj: Vlastní tvorba)

[17]LAMP - spojení Linuxu, Apache, MySQL a PHP

[18]MEAN – spojení MongoDB, Express.js, Angular.js, Node.js

Na obrázku 3.8 je již vidět migrace v praxi. Pomocí dvou příkazů vytvoříme a spustíme migraci. Pokud bychom zaměnili příkaz *up* za *down*, proběhne migrace opačným způsobem.

```
$ db-migrate create pridat-prioritu
[INFO] Created migration at ./migrations/20150108115242-pridat-
prioritu.js
...
$ db-migrate up
[INFO] Processed migration 20150108115242-pridat-prioritu
[INFO] Done
```

Obr. 3.8: Ukázka shellových příkazů pro vytvoření a spuštění migrace (Zdroj: Vlastní tvorba)

3.4.2 Ruby on Rails

Ruby on Rails je rozvíjející se webový framework napsaný v programovacím jazyce Ruby. Jedná se o volně šiřitelný nástroj, dostupný pod MIT licencí. Rails byl jeden z prvních frameworků, který začal používat REST (Representational State Transfer) architekturu a CRUD (Create, read, update a delete). Rails umí mnoho běžných webových programovacích procedur jako je generování HTML, také datové modely a směrování podle URL. Pomocí Rails lze velmi snadno zajistit, aby byl výsledný kód aplikace stručný a čitelný [14].

Příklad použití

Ruby on Rails poskytuje generátory pro většinu často prováděných akcí. Pro naše účely se podíváme na vytvoření nového modelu (a odpovídající tabulky, automaticky pojmenované podle modelu) a na provedení změny v něm. Na obrázku 3.10 můžeme vidět definice migrace, které tento nástroj vygeneroval.

```
# vytvoreni modelu a~migrace pro vytvoreni tabulky
$ bin/rails generate model Zaznam nazev:string obsah:text
      invoke  active_record
      create    db/migrate/20150108135325_create_zaznams.rb
      create    app/models/zaznam.rb
      invoke    test_unit
      create      test/models/zaznam_test.rb
      create      test/fixtures/zaznams.yml
# provedeni zmen v~databazi

$ bin/rake db:migrate
== 20150108135325 CreateZaznams: migrating
-- create_table(:zaznams)
   -> 0.0019s
== 20150108135325 CreateZaznams: migrated (0.0020s)
# vygenerovani migrace pridavajici sloupec priorita
$ bin/rails generate migration AddPrioritaToZaznams priorita:int
      invoke  active_record
      create db/migrate/20150108135908_add_priorita_to_zaznams.rb
# provede jen jeste neprovedene zmeny
$ bin/rake db:migrate
== 20150108135908 AddPrioritaToZaznams: migrating
-- add_column(:zaznams, :priorita, :int)
   -> 0.0038s
== 20150108135908 AddPrioritaToZaznams: migrated (0.0039s)
```

Obr. 3.9: Ukázka vytvoření modelu a migrace (Zdroj: Vlastní tvorba)

```ruby
# db/migrate/20150108135325_create_zaznams.rb
class CreateZaznams < ActiveRecord::Migration
  def change
    create_table :zaznams do |t|
      t.string :nazev
      t.text :obsah
      t.timestamps
    end
  end
end

# db/migrate/20150108135908_add_priorita_to_zaznams.rb
class AddPrioritaToZaznams < ActiveRecord::Migration
  def change
    add_column :zaznams, :priorita, :int
  end
end
```

Obr. 3.10: Vytvořené migrace (Zdroj: Vlastní tvorba)

3.5 Nástroje podporující automatické vytvoření migrace

Tato třída nástrojů podporuje nejen standardní práci s migracemi, ale navíc dokáže automaticky vytvářet migrace na základě změn v definicích tříd entit modelů.

Níže rozeberu dva zástupce této sekce: Django (plnohotnotný pythonový webový framework) a Hibernate (databázová ORM vrstva Java aplikace).

Co se týče bezpečnosti u těchto nástrojů i zde platí to samé, co v předchozí sekci 3.4. Tj. zmíněné nástroje mohou sice poskytovat nějaké služby pro implementaci zabezpečení, nebudou však nijak svýzané s databázovou vrstvou.

3.5.1 Django

Django je rozvíjející se webový framework, který dokáže ušetřit čas při vývoji webových aplikací. Pomocí Django je možné vytvářet a spravovat webové aplikace ve vysoké kvalitě s minimální námahou. Je možné ho kombinovat s různými databázemi, anebo ho použít bez jakékoli databáze.

Django obsahuje dvě klíčové vlastnosti. První je Djangův „sweet spot“ – jedná se o administrátorské rozhraní. Nabízí hned několik funkcí, například pluginy, grafy návštěvnosti, správu entit a další [10]. Druhou vlastností je, že tento produkt je volně šiřitelný a je zaměřen na řešení problémů při vývoji webových aplikací. To znamená, že Django produkuje aplikace, které nejsou náročné na údržbu a fungují dobře při zatížení [15].

Příklad použití

V nástroji Django je nejprve nutné vytvořit si ručně soubor s příslušnými modely. Poté už jen stačí pomocí příkazů spustit vytvoření migrace, což můžeme vidět na obrázku 3.12. Django dokáže rozpoznat přidané řádky, změny typu, při rozeznání přejmenování sloupce se zeptá, zdali nastala tato změna [11].

Vygenerované migrace byly příliš dlouhé a principiálně se nelišily od vygenerovaných migrací v Ruby on Rails, proto nebylo nutné je sem přiložit.

```
1 # poznamky_app/models.py
2 from django.db import models
3
4 class Zaznam(models.Model):
5     nazev = models.CharField(max_length=64)
6     obsah = models.CharField(max_length=200)
```

Obr. 3.11: Soubor s manuálně vytvořenými modely (Zdroj: Vlastní tvorba)

```
# automaticky vytvori migraci podle modelu
$ python manage.py makemigrations poznamky_app
Migrations for 'poznamky_app':
  0001_initial.py:
    - Create model Zaznam
# provede migraci
$ python manage.py migrate
Operations to perform:
  Apply all migrations: admin, contenttypes, poznamky_app, auth,
  sessions
Running migrations:
  Applying poznamky_app.0001_initial... OK
# pridame pole priorita do modelu
$ echo 'priorita = models.IntegerField(default = 0)' >>
 poznamky_app/models.py
$ python manage.py makemigrations poznamky_app
Migrations for 'poznamky_app':
  0002_zaznam_priorita.py:
    - Add field priorita to zaznam

$ python manage.py migrate #$
Operations to perform:
  Apply all migrations: admin, contenttypes, poznamky_app, auth,
  sessions
Running migrations:
  Applying poznamky_app.0002_zaznam_priorita... OK
```

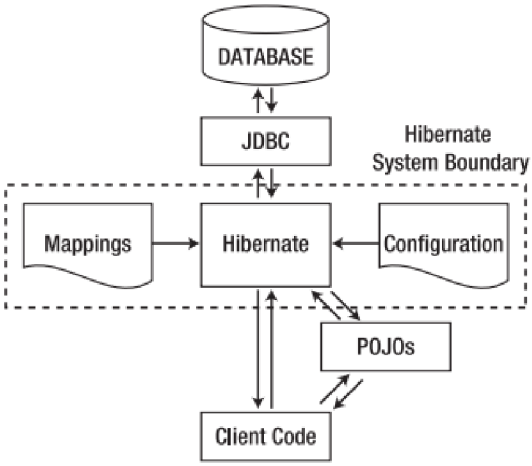
Obr. 3.12: Vytvořené migrace (Zdroj: Vlastní tvorba)

3.5.2 Hibernate

Java je objektově orientovaný jazyk se silným statickým typováním. K dispozici je mnoho knihoven a frameworků, které tento jazyk rozšiřují, jak již bylo zmíněno výše v kapitole 3.2.1. Nejpoužívanějším frameworkem při výrobě webových aplikací je Spring, který nemá svoji vlastní databázovou vrstvu, a proto se jim dále nebudu v této práci zabývat. Jako databázová vrstva se používají různé implementace Java Persistence API, například EclipseLink nebo Hibernate.

EclipseLink navíc podporuje ukládání do XML struktur a síťovou komunikaci. Jeho objektově relační mapování je srovnatelné s Hibernate. Dále bude blíže rozepsán nástroj Hibernate.

V našem ideálním světě by bylo triviální vzít objekt a umístit ho do databáze bez speciálního kódu a zpomalení chodu aplikace. Hibernate se k tomuto přibližuje, ale i přesto je potřeba ho nakonfigurovat a zvážit vliv na výkon celé aplikace, ovšem Hibernate dosahuje svého cíle - uložit objekty do databáze. Obrázek 3.13 ukazuje, jak Hibernate zapadá do celého systému [18].



Obr. 3.13: Úloha Hibernate v Java aplikaci (Zdroj: Převzato z [18])

Hibernate umí vytvořit databázové schéma z definic tříd objektů. Existuje mnoho způsobů, jak můžeme nástroji Hibernate předat všechny informace pro připojení k databázi a definovat objektově relační mapování. Nejčastěji se používají XML soubory, properties soubory a anotace v kódu [18].

Příklad použití

Hibernate obsahuje nástroj HBM2DDL, který řeší synchronizaci schématu a definic tříd. Je možné ho buď spustit manuálně nebo nastavit vlastnost hbm2ddl.auto, která při každém spuštění nastaví jednu z následujících možností:

validate – zvaliduje schéma, nezmění nijak databázi

update – zaktualizuje schéma

create – vytvoří schéma, předchozí data zničí

create-drop – smaže schéma při ukončení

Pro pokročilejší použití (analýzu rozdílů mezi databázemi, více databází, generování SQL změn, zachování dat a další.) je k dispozici externí nástroj Liquibase[19], kterým se dále nebudu v této práci zabývat.

```
1  @Entity
2  public class Zaznam {
3     @Id private Long id;
4     public Long getId() { return id; }
5     public void setId(Long id) { this.id = id; }
6
7     private String nazev;
8     public String getNazev() { return nazev; }
9     public void setNazev(String nazev) { this.nazev = nazev; }
10
11    @Column(length = 200)
12    private String obsah;
13    public String getObsah() { return obsah; }
14    public void setObsah(String obsah) { this.obsah = obsah; }
15 }
```

Obr. 3.14: Hibernate – anotovaná definice třídy (Zdroj: Vlastní tvorba)

[19] http://www.liquibase.org/

3.6 Shrnutí existujících řešení

Ve výše zmíněných řešeních byly rozebrány různé způsoby, jak nahlížet na správu změn v databázi a také zmíněna základní použitelnost těchto řešení. Pro přehlednost si shrneme všechna tato řešení.

Prvním nástrojem, který byl v této práci zmíněn, byly relační databáze bez použití pomocných nástrojů, u kterých po prostudování převažují spíše nevýhody nad výhodami. Zejména práce se schématem, nutnost ručních změn na vícero místech či nemožnost verzování schématu.

Zmíněné nevýhody relační databáze bez použití pomocných nástrojů řeší nástroje podporující předpřipravené či automatické migrace. Nástroje jsou poskytovány větší částí frameworků a dostupné pro všechny relační databáze a nejenom pro ně. Hlavní nevýhodou je nutnost správy migrací, ať již nutnost jejich manuálního vytváření, nebo potřeba spouštění jejich vygenerování. Další nevýhodou jsou problémy s verzováním migrací, kdy může vzniknout velké množství souborů, ze kterých si programátor musí vybrat ten správný.

Alternativním přístupem jsou NoSQL databáze, od řešení v podobě cloudového formátu (Meteor) až po objektové databáze (Jalapeño). Bez potřeby schématu není nutné schéma spravovat a tudíž nedochází k výše popsaným problémům. NoSQL databáze mají také své nedostatky, například nepodporují transakce nebo ACID zákony.

Jak můžeme vidět, žádný z výše zmíněných nástrojů není zcela vhodně optimalizován jak pro vývoj produktu, tak i pro produkci. Z těchto dostupných informací budu vycházet v následující kapitole.

4 Vlastní návrh řešení

Z analýzy již existujících řešení vyplynulo, že databáze vhodné pro nasazení nejsou často vhodné pro vývoj, neboť mají pevně danou strukturu nebo jsou nedostatečně flexibilní. A naopak databáze vhodné pro vývoj mají mnohá omezení, která je činí nevhodnými pro produkční nasazení. Proto se hodí během vývojové fáze použít jinou databázi a ještě před nasazením do rutinního provozu ji vyměnit. Nicméně toto řešení přináší mnohá úskalí, protože komunikace s různými databázemi je často zcela odlišná.

Řešením tohoto problému je zvolit dotazovací jazyk, který by napomohl k minimalizaci náročnosti této změny. Idea této práce tedy spočívá v navrhnutí mezivrstvy, která imituje reálnou relační databázi a zjednodušuje její pozdější vytvoření a postupný přechod na produkční verzi.

Všechna výše zmíněná řešení mají problémy s postupnými změnami při vytváření relačního schématu a také s postupnými a zpětnými migracemi. V rámci návrhu se budu zabývat postupnými změnami v datech a také změnami při vytváření relačního schématu.

Níže bude detailně rozepsán návrh této mezivrstvy i s popisem její funkcionality a v závěrečné kapitole budou popsaná možná rozšíření jako například zpětné migrace, či parsování více dotazovacích jazyků.

4.1 Výběr databází

V kapitole 2 byly popsány charakteristiky různých databází, které by mohly být použity v této práci. Pro potřeby této práce bude nutné vybrat dva zástupce z databází, které budou splňovat tyto podmínky:

Vývojová databáze

- Flexibilita
- Jednoduchost zprovoznění
- Snadný dotazovací jazyk

Produkční databáze

- Spolehlivost
- Škálovatelnost

- Stabilita
- Kvalita podpory produktu

Po důkladném zvážení všech aspektů jsem došla k závěru, že tito dva zástupci budou nejlepší volbou:

MongoDB je flexibilní dokumentovou databázi. Vzhledem k jednoduchosti zpracování jeho silného dotazovacího jazyka byla tato databáze zvolena s myšlenkou použití tohoto jazyka v celé mezivrstvě. Většina dokumentových nebo NoSQL databází umožňuje dotázování se na konkrétní záznam podle *id* nebo předem definovaného indexu, či na celou kolekci záznamů. Jazyk mongo dotazů umožňuje také ad-hoc filtrování podle libovolných políček.

PostgreSQL je vyspělý zástupce SQL databází. Nejedná se tedy jen o relační databázi, jak by tomu mohl název napovídat, ale podporuje vlastnosti objektových databází. Další předností této databáze je možnost ukládání dat do JSON polí, ve kterých je možno lehce vyhledávat pomocí SQL dotazů.

4.2 Návrh

Standardně se vyvíjí webové aplikace podle určité struktury, která je případně použita ve své základní podobě nebo rozšířena o testování či jiné specifické činnosti. Níže přiblížím tuto základní strukturu a popíši jednotlivé akce. Nejčastěji je použito při tomto vývoji nějaká určitá relační databáze, nějaký skriptovací jazyk pro logiku aplikace na serveru a v neposlední řadě JavaScript pro oživení stránek.

Standarní vývoj webové aplikace:

- **Instalace a konfigurace** – instalace potřebného softwaru (webový server, databáze);
- **Vytvoření databáze** – návrh schématu aplikace, vytvoření jednotlivých tabulek v databázi, přidání práv a omezení, vytvoření testovacích dat;
- **Serverová logika aplikace** – vytvoření API pro komunikaci s databází a frontendem, volba skriptovacího jazyka případně i frameworku, navržení routy[20], Business logika;

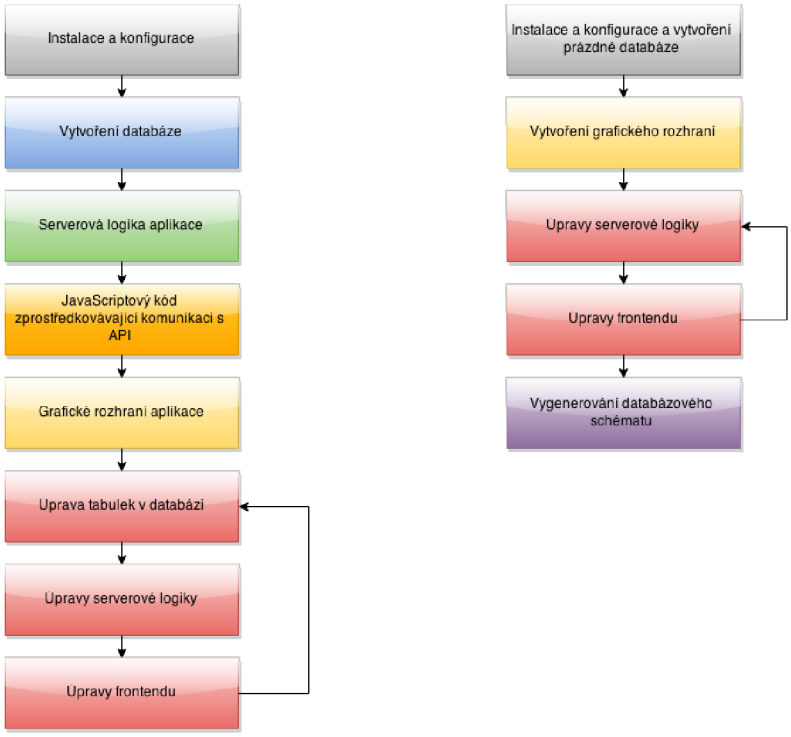
[20]Routa – slouží k zpřístupnění konkrétních funkcí na určitých URL adresách.

- **JavaScriptový kód zprostředkovávající komunikaci s API** – Mnohdy je potřeba převést data, která poskytuje API, do potřebnějšího tvaru. Proto je v každém případě vhodné obalit tuto komunikaci se serverem do oddělené části kódu zabývající se komunikací.
- **Grafické rozhraní aplikace** – vytvoření klientské části podle zadaného návrhu grafika, případně vytvoření vlastní kostry pro další vývoj a práci;
- **Při každé změně v požadavcích klienta se musí změnit tyto věci:**
  - Úprava tabulek v databázi;
  - Úpravy serverové logiky;
  - Úpravy frontendu;

Ve výše uvedeném postupu vývoje aplikace je nutno provádět při každé změně mnoho úkonů, které mění skladbu celé aplikace. Tomuto bych se chtěla ve svém návrhu vyvarovat a tím zjednodušit vývoj aplikace. To znamená, že některé kroky programátora udělá za něj mnou navrhovaná mezivrstva. Níže přiblížím její strukturu a poté ji porovnám se standardním vývojem.

Mnou navrhovaný vývoj webové aplikace:

- **Instalace, konfigurace a vytvoření prázdné databáze** - oproti předchozí struktuře se zde pojí mezivrstva s datatbázemi;
- **Vytvoření grafického rozhraní** - vytvoření základní struktury stránek buď podle návrhů grafika nebo vytvoření základní kostry programátorem; Ve standardním vývoji se tento krok provádí až v pozdějším stádiu, protože závisí na funkcionalitě, kterou je třeba vytvořit, ale v tomto návrhu ji poskytuje mezivrstva.
- **Při každé změně v požadavcích klienta se musí změnit tyto věci:**
  - Úpravy serverové logiky;
  - Úpravy frontendu;
- **Vygenerování databázového schématu** - Po ukončení vývoje je možné si vygenerovat databázové schéma, které je potřeba nahrát do relační databáze a poté je možné připojit aplikaci k této databázi a spustit ji v produkční verzi. Tento krok je možný i v průběhu vývoje pro jednotlivé funkční celky.



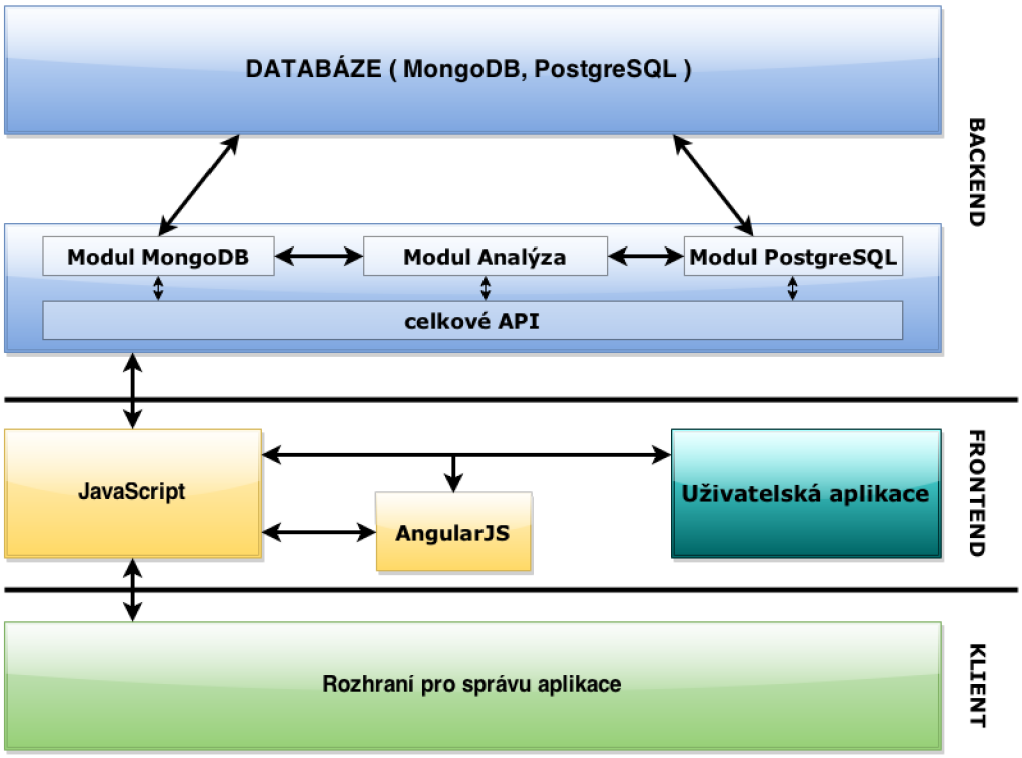
Obr. 4.1: Diagramy vývoje aplikace – vlevo tradiční, vpravo navrhovaný (Zdroj: Vlastní tvorba)

Na obrázku 4.1 si můžeme povšimnout, že na začátku vývoje v mém návrhu není nutné řešit žádné API či Business logiku, což obstarává prozatím mezivrstva. Tuto část samozřejmě programátor zajistit musí, může tak učinit mnohem později, kdy budou jasně stanoveny požadavky od zákazníka. Dalším velkým rozdílem je, že odpadá potřeba průběžné manuální úpravy tabulek.

Zatímco v původním návrhu je potřeba při každé změně v datech změnit strukturu tabulky (či vytvořit migraci), upravit kód backendu, který se stará o posílání dat a také o ukládání dat a upravit logiku na frontendu, aby došlo k projevení této změny Při použití mezivrstvy všechny tyto úpravy odpadávají, jak je naznačeno v obrázku 4.1. Pokud jsou tyto změny provedeny dostatečně brzy ve vývoji s použitím navrhované mezivrstvy zcela odpadá krok úpravy databáze. Jestliže nevznikla žádná backendová (business) logika, které by se tyto změny dotkly, není potřeba žádná změna na backendu. Přidání nového sloupce je samozřejmostí. Pomocí vrstvy mezi databází a HTTP požadavky je přidáno logování použití (kvůli provedení různých analýz). Na druhou stranu toto řešení není převratně rychlé ani neposkytuje záruky na konzistenci dat (ACID complicance), a podobně. Po

ustálení struktury dat je však možné provést převod kolekce do PostgreSQL tabulky a získat striktní schéma, ale také typy, indexy a možnost SQL přístupu (při vytvoření vlastní custom route). Mimo zmíněné typové kontroly také probíhá kontrola konzistence sloupců. Ty jsou sice při převodu označeny jako *NULL*, takže chyba nastane jen při použití neznámého sloupce. Pokud ale uživatel upraví sloupec v databázi a označí sloupec jako *NOT NULL* (a neposkytne jinou výchozí hodnotu), stane se chybou i vynechání tohoto sloupce. Mezivrstva je stále zachována a analýza tedy může dále probíhat (mimo přímého SQL přístupu). Postupným přepisem používaných dotazů na custom routes je možné získat ACID compliance a rychlost na úkor jednoduchosti.

Níže na obrázku 4.2 je vidět, jak do sebe části v navrhnované mezivrstvě do sebe zapadají. Každá z nich bude dále rozepsaná ve vlastní sekci.



Obr. 4.2: Schéma architektury navrhované mezivrstvy (Zdroj: Vlastní tvorba)

4.2.1 Backend – Server

Součástí této části mezivrstvy jsou dvě vybrané databáze (MongoDB, PostgreSQL). Tyto databáze bude koordinovat vrstva poskytující jednotné API, která bude vnitřně rozdělena na 4 části.

První dvě části se budou zaobírat řešením komunikace s databázemi. Každá databáze bude mít své vlastní moduly z důvodu odlišných dotazovacích jazyků. Vzhledem k tomu, že celé API je založeno na dotazovacím jazyku MongoDB, tak tento modul bude sloužit pouze k předávání dotazů do MongoDB databáze. Naopak modul PostgreSQL bude komplikovanější o převod těchto dotazů do SQL jazyka, kontrolu struktury dat a manipulaci s tabulkami.

Dalším modulem bude Analýza, kde bude docházet k analýze schématu a přepínání mezi moduly MongoDB a PostgreSQL. Hlavní funkcionalitou tohoto modulu je však průběžná analýza dat za účelem vytvoření předběžného schématu databáze v průběhu vývoje. V sekci 4.3.2 více rozepíši funkcionalitu tohoto modulu.

Posledním modulem bude celkové API, které se bude starat o komunikaci s frontendovou částí aplikace. Součástí bude REST API a pro každou tabulku routy s bežnými operacemi. Uživatel navíc může ke kolekcím přiřazovat vlastní funkcionalitu ve formě uživatelských rout, které mohou i nadále využívat dotazů přes mezivrstvu a být také analyzovány. Dále může uživatel přiřadit hook[21] k předdefinovaným REST operacím.

4.2.2 Frontend

Pro účely této práce bude použit framework AngularJS na straně frontendu, nicméně toto nebude v implementaci vyžadováno. Ve frontendu tedy bude zvlášť část, která bude sloužit pro komunikaci s backendem, a zvlášť adaptér pro AngularJS.

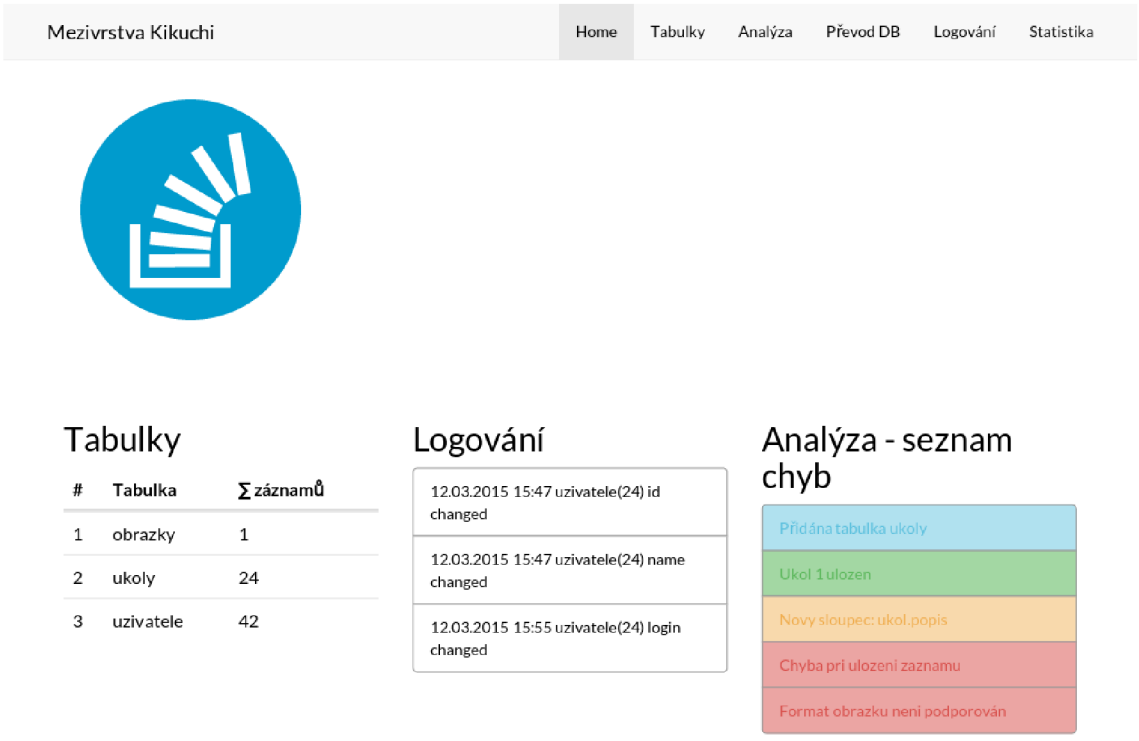
Obecná JavaScriptová část se bude starat o bezpečnou komunikaci s celkovým API v backendové vrstvě, interpretovat chybové hlášky serveru a příslušně na ně reagovat a poskytovat pomocné funkce pro Javascript, který chce pracovat s daty uloženými na serveru.

[21] Hook – funkce, která je spuštěna na vhodném místě z implementace předdefinované funkcionality

AngularJS část bude rozšířením funkcionality JavaScriptu poskytující integraci s AngularJS frameworkem. Zejména bude poskytovat AngularJS service[22], který bude pokrývat nekonzistenci čistého JavaScriptu s angularem a bude moci vytvářet podle potřeby service-like objekty obsluhující jednotlivé kolekce.

4.2.3 Klient

Klient bude obstarávat grafické rozhraní pro správu mezivrstvy, kde bude možno zobrazení předběžného schématu databáze, možnosti výběru převodu z MongoDB do PostgreSQL a zobrazení analýz. Níže na obrázku 4.3 můžeme vidět návrh hlavní strany tohoto rozhraní, kde budou zobrazeny novinky ohledně přidaných tabulek, logování všech akcí provedených v této mezivrstvě a v neposlední řadě seznam chyb. Tyto chyby jsou ty samé chyby co vrací API.



Obr. 4.3: Mock up klientského rozhraní (Zdroj: Vlastní tvorba)

[22]Service – je to singleton objekt použitelný s dependency injection funkcionalitou angularu.

4.3 Popis funkce mezivrstvy

Celá mezivrstva by měla co nejvíce zjednodušit obvyklé činnosti při vývoji aplikace a pokud možno přesunout nutnost vytvoření funkcionality celé aplikace na co nejpozdější dobu, což umožní snadné prototypování. Toto usnadní získání včasné zpětné vazby od zákazníka, takže je možné začít vyvíjet business logiku až ve chvíli, kdy jsou mnohem jasnější požadavky zákazníka.

Na rozdíl od standardního vývoje aplikace, kdy je použita jedna databáze, která se v průběhu mění, použije se zde flexibilní (dokumentová) databáze, která sbírá všechna data, co jsou v průběhu vývoje používána, a poté budou buď manuálně převedena v průběhu vývoje do pevné (relační) databáze nebo to bude provedeno automaticky po skončení vývoje. Mezivrstva dokáže rozeznat, které kolekce jsou používány a vytvořit pro ně příslušné routy.

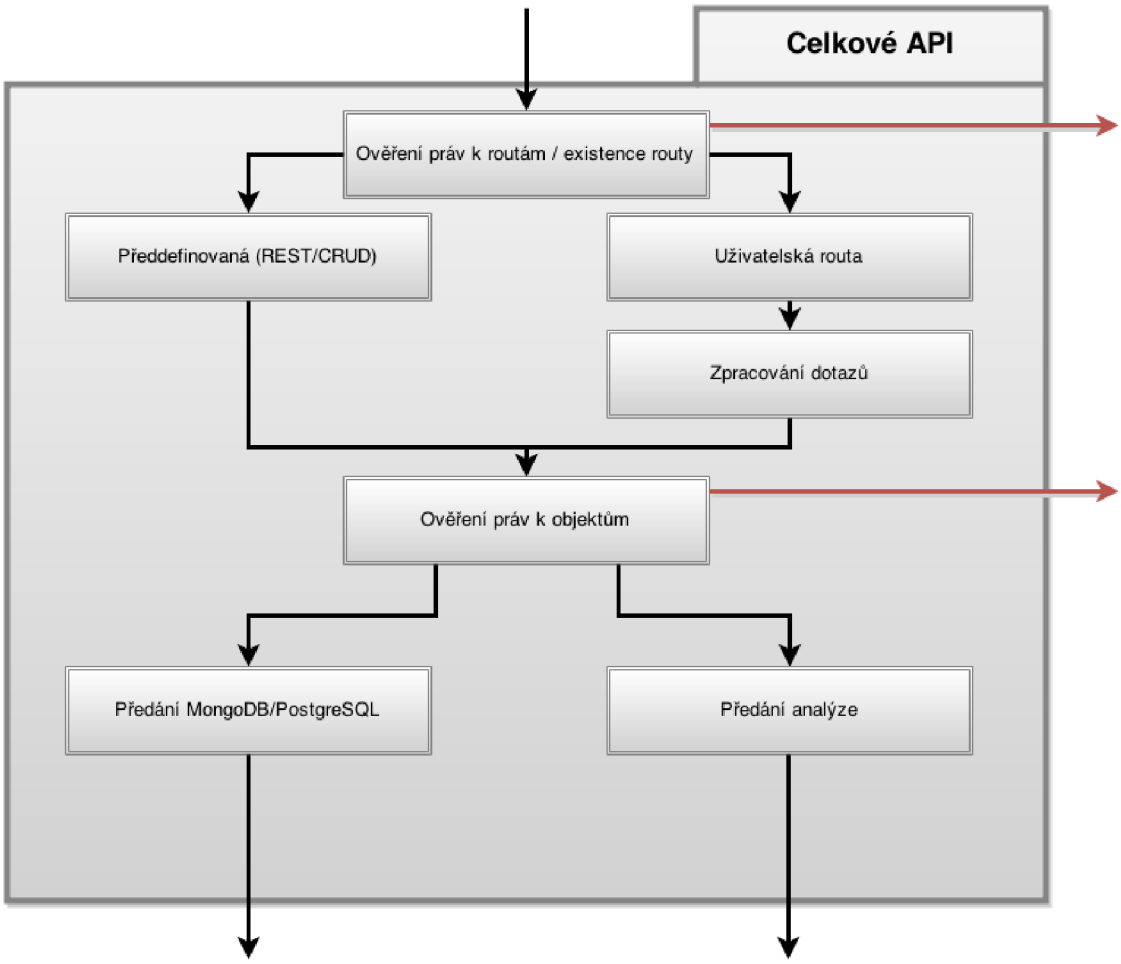
V rámci této práce je podporovaný pouze dotazovací jazyk Mongo, z důvodu jednoduchosti analýzy těchto dotazů. V případě dotazu do PostgreSQL databáze se tedy bude používat automatický převod poskytovaný modulem PostgreSQL. Alternativně je možné dotazovat se do této databáze přímo SQL jazykem, bez použití mezivrstvy.

4.3.1 Celkové API

Celkové API obaluje moduly MongoDB, Analýza a PostgreSQL. Zprostředkovává komunikaci mezi JavaScriptovou částí a jednotlivými moduly. Jeho hlavní úlohou je tedy sbírání odpovědí z modulů, následně je pospojovat a odeslat jako objekt do JavaScriptu. Tímto API prochází veškerá data, takže je vhodným místem pro implementaci kontroly práv objektu, určení práv k jednotlivým routám, autentizaci, validaci dotazů a řešení chybových hlášení.

Obrázek 4.4 zobrazuje strukturu celkového API. Z JavaScriptové části obdrží tato vrstva data, která nejprve ověří, zdali mají práva k routě. V případě že ne, API vrátí chybovou hlášku. Dále je nutné zjistit, jestli se jedná o předdefinované routy či uživatelské. Předdefinované routy jsou vždy pro standardní akce nad danou kolekcí – hledání, čtení, úprava, smazání a vytvoření. Zde je nutné provést validaci dotazu, aby bylo možno pokračovat dále v procesu a úspěšnému provedení všech akcí.

Zvláštním případem je routa uživatelská, jejíž logika je zcela definovaná vývojářem. Je tedy nutné zpracovat dotazy volané touto logiku, aby bylo možné analyzovat či dále pracovat s jejími objekty. V každém případě dalším krokem je zpracování dotazů na data, ověření jejich práv a následně předání dotazů k příslušným modulům. Všechny dotazy se předávají také do modulu Analýza, jehož výsledkem je výpis ladících či chybových hlášení, který se poté přidává do odpovědi API. Tyto dotazy se předávají příslušné databázi na základě rozhodnutí API, které ví, kde jsou jednotlivá data uložena.



Obr. 4.4: Struktura Celkového API (Zdroj: Vlastní tvorba)

Detaily API pro registraci uživatelských rout bude více rozepsáno v sekci 4.4, stejně jako kontrola práv.

4.3.2 Modul Analýza

Modul Analýza průběžně shromažďuje veškerá data, která jsou uložena v databázi, ať již v MongoDB nebo PostgreSQL a provádí nad nimi různé analýzy. Na základě shromážděných dat udržuje statistiky používanosti jednotlivých políček, kterým přidá příznak používanosti. Toto je vhodné při posouzení, zda je políčko vyžadováno, či se nejedná o chybu v názvu nebo s použitím hledání podobnosti v ostatních políčkách. Políčka, která bývají použita v podmínkové části dotazu, by měla být indexována, a políčka, která nabývají omezeného množství různých hodnot nám tímto dávají informaci o tom, jakého mohou být typu (enum). K zjištění typu se dále použijí regulární výrazy pro zjištění čísla, data apod.

Pro každý dotaz je provedena série těchto analyzačních kroků:

- Levenshteinovy vzdálenosti[23] jmen tabulek a sloupců, kvůli detekci překlepů
- Analýza typů – statistická analýza porovnáním s předdefinovanými typy
- Zjištění vyžadovaných políček – všechna vybraná políčka na základě způsobu používání
- Identifikace použití políčka v podmínkové části dotazu, kvůli vytvoření indexů pro často dotazovaná políčka
- Homogennost dat v tabulkách pro identifikaci stavu vývoje

Výsledkem těchto kroků jsou zanalyzovaná data v databázi a série textových výstupů vztahujících se k aktuálnímu dotazu (varování o duplicitě sloupce apod.). Tato data jsou dále použitelná i při bezpečnostních auditech, kdy je potřeba zjistit například kdo naposledy změnil dané políčko v konkrétním záznamu. Jednotlivé kroky budou podrobněji rozepsány v sekci 4.4.

4.3.3 Modul MongoDB

Vzhledem k tomu, že mezivrstva používá MongoDB dotazovací jazyk, tento modul se bude zaobírat pouze samotnou komunikací s databází. Přijímá dotazy od celkového API

[23]Levenshteinova vzdálenost – řetězcová metrika pro měření rozdílu mezi dvěma sekvencemi. Neformálně Levenshteinova vzdálenost mezi dvěma slovy je minimální počet jednotlivých změn znaku.

mezivrstvy, bez modifikace je předá databázi a následně při odpovědi z databáze tato data přepošle zpět celkovému API, které pak zprostředkovává další zpracování těchto dat.

4.3.4 Modul PostgreSQL

Tento modul zprostředkovává nejen komunikaci s databází, ale navíc řeší převod Mongo dotazů do SQL jazyka. V případě převodu modul PostgreSQL obdrží dotaz v Mongo formátu, což znamená několik JavaScriptových objektů popisujících vlastnosti dotazu, ze kterých se vytvoří SQL dotaz pomocí algoritmu blíže popsaného v sekci 4.4.

Odpovědi z PostgreSQL databáze bude nutno taktéž převádět na stejný formát, který podporuje MongoDB. Toto je potřeba zejména kvůli typu datum, který je v různých databázích uložen a vracen v odlišných formátech.

4.3.5 JavaScriptová část

JavaScriptová část je izolovaná od serverové části a veškerá komunikace probíhá přes REST API a poskytuje tedy zjednodušené nástroje pro používání výchozích rout.

Při vývoji aplikace jsou často v kódu používané pomocné proměnné, které vývojář nechce či nepotřebuje ukládat do databáze. Pro tento účel bude tato část identifikovat a poté při komunikaci s celkovou API odstraňovat tyto pomocné proměnné, jako příklad zde může být uvedena proměnná `$$hashKey`, kterou používá AngularJS pro interní účely.

Hlavní náplní této části je zpřístupnit serverovou databázi klientské části při zachování kompatibility s API MongoDB, aby mohla jednodušeji komunikovat s touto databází bez jakéhokoliv omezení. Také je zodpovědná za oboustranné převody dat, například převádí `Date` objekt do reprezentace data, kterou očekává Mongo dotazovací jazyk a naopak.

4.3.6 AngularJS část

Slouží pro obalení JavaScriptové části za účelem použití z AngularJS aplikace. Je to z důvodu zjednodušení komunikace mezi frameworkem angular a čistého JavaScriptu.

Zejména je při každé změně potřeba vyvolat `$digest` cyklus[24] angularu. Dále poskytuje angular `services` pro každou kolekci, čímž řeší modely pro všechny typy entit.

[24] $digest cyklus – angular zkontroluje všechny `watch` funkce a změny, které objevil, aplikuje.

4.3.7 Uživatelská aplikace

Jedná se o kód vývojáře, který používá tuto mezivrstvu ve svém projektu. Stačí pouze připojit celkové API mezivrstvy a JavaScriptovou část (případně i AngularJS část, pokud by aplikace byla napsaná v angularu) k uživatelské aplikaci, zavolat funkci pro inicializaci JavaScriptové části a vývojář může směle vyvíjet svoji aplikaci.

Vývojáři je dostupné stejné API dotazovacího jazyku jako při použití Mongo knihoven. Na pozadí je však použita mezivrstva s MongoDB nebo PostgreSQL databází.

4.3.8 Rozhraní pro správu

Pro jednodušší práci s mezivrstvou je vytvořeno webové rozhraní, které bude obsahovat následující položky:

Tabulky – seznam všech tabulek v databázích s identifikací, ke které databázi náleží. Jednotlivé záznamy bude možno rozkliknout pro detailnější zobrazení. Po rozkliknutí se zobrazí tabulka s příslušnými názvy jednotlivých políček a daty. Je možné přepínat mezi tímto zobrazením a zobrazením struktury tabulky a také je možné mazání dat.

Analýza – bude obsahovat informace, které byly zmíněny v sekci 4.3.2. Tato záložka bude obsahovat tři sekce:

- **Možné překlepy** – pohled na tabulky či sloupce podobného názvu,
- **Analýza sloupců** – výpis všech sloupců řazených podle tabulek s informacemi o jejich zjištěném typu, četnosti v datech a dotazech,
- **Seznam chyb** – archiv všech hlášení v průběhu vývoje aplikace.

Převod DB – seznam tabulek s identifikací, ke které databázi náleží, a dále s možností manuálního převodu z MongoDB do PostgreSQL čí vygenerování databázového schématu po ukončení vývoje s mezivrstvou.

Logování – přehled všech uživatelem vyvolaných akcí s filtrováním podle data, jména uživatele a IP adresy.

Statistika – grafy přístupů a změn k jednotlivým tabulkám

4.4 Implementace

Instalace mezivrstvy do aplikace je podobná jako zapracování JavaScriptové knihovny. Tedy je lehce použitelná v jakémkoli projektu. Postačí ji pouze nakopírovat do příslušné složky v aplikaci, nastavit databázi, kterou má používat, a poté už jen nastavit webserver, který bude používat tento adresář s mezivrtsvou. Tzn. je potřeba vytvořit *kikuchi/conf.local.php* s řádkem *conf('mongo_db', 'JMENO PROJEKTU')*, případně i *mongo_uri*, pokud databáze běží jinde, než na lokálním serveru, a *url_under* pokud je na webserveru pod jiným adresářem než */kikuchi*. Pro nastavení spojení s PostgreSQL databází je nutné tamtéž vyplnit parametry *pg_host*, *pg_username*, *pg_password* a *pg_database*. Pro načtení PHP knihoven je potřeba spustit composer[25] v adresáři kikuchi.

Pro spuštění druhé části mezivrstvy je potřeba mít nainstalovanou komponentu bower[26] a poté už jen postačí `bower install` v té složce. Tato komponenta je potřeba k nainstalování příslušných závislostí (fonty, AngularJS, JQuery, atd.). Poslední potřebnou věcí je jakýkoli webový server, který dokáže spustit PHP skript a přepsat url.

V rámci této práce nebyla implementována kontrola práv, protože by vyžadovala správu uživatelů, která má typicky jasné oddělení a není během vývoje natolik potřebná. Kontrola by používala kolekci uživatelů, ať již v databázi či pomocí externí služby (např. Google OAuth, či Facebook Login), a nějaké úložiště pro informace o sessions či autentizační tokeny, např v memcached[27]. Práva by měla být nastavitelná deklarativně v kódu, s možností omezení přístupu ke kolekcím, sloupcům či záznamům na základě rolí, uložených stejně jako uživatelé.

Jak již bylo zmíněno výše v návrhu, uživatel bude mít k dispozici kromě předdefinovaných rout i uživatelské routy. Zde je použit flexibilní routovací framework. Všechny routy jsou deklarovány pomocí statických metod třídy *CustomRoute* odpovídajících jednotlivým HTTP metodám. Tyto routy interně používají knihovnu Klein[28] a také její syntax. Funkce mají k dispozici celkové API, a jejich návratová hodnota je použita ve vytvářené JSON odpovědi. V adresáři kikuchi je předpřipraven soubor *customroutes.php*, kde je možné

[25] Instalace z této adresy https://getcomposer.org/, dále *composer.phar install*

[26] Potřeba nainstalování https://nodejs.org , dále `npm install -g bower`

[27] memcached – čistě pamětové key-value úložiště

[28] https://github.com/chriso/klein.php

psát své uživatelské routy pomocí *CustomRoute* metod.

4.4.1 Server

Backendová část je implementovaná v jazyku PHP s použitím knihovny Klein pro routování, knihovny dibi pro přistup k PostgreSQL databázi a knihovny mongodb pro přistup k MongoDB databázi. Průchod příchozího dotazu je vidět z obrázku 4.4. Dotaz přijde v REST a JSON formátu, dispatcher zavolá příslušnou předdefinovanou nebo uživatelskou routu, která provede svoji logiku.

Dotazy na data vyvolané v těchto funkcích jsou předávány příslušné databázi, stejně jako analytickému modulu. Předání databázi se děje na základě návratové hodnoty funkce `kde_su($table)`, která si při prvním spuštění načte, jaká kolekce je uložena ve které databázi. Databázový modul vykoná dotaz a vrátí odpověď od databáze, zatímco analytický modul zpracuje data, výsledky uloží do své databáze a vrátí informativní hlášky zpět. Všechny tyto odpovědi jsou pak spojeny do jednotného formátu odpovědi, která je odeslaná zpět klientu.

Chyby jsou taktéž řešeny jednotným způsobem. Pokud databázový modul vyhodí výjimku, je tato výjimka předána zpět ve formě chybové zprávy. Výjimka v analýze by neměla nastat, možnost jejího výskytu je však také zohledněna a je zpracována stejným způsobem.

Odpověď na každý dotaz má nastaven HTTP status kód[29] standardním způsobem a obsahuje JSON objekt s následujícími políčky:

[29]HTTP status kód – třímístný číselný kód odpovědi, popisující výsledek požadavku. První číslice je pevně definovaná a určuje typ výsledku – informace, úspěch, přesměrování, chyba klienta, chyba serveru

```
{
 status: 200, // HTTP status code
 messages: [ // zpravy od backendu
   { type: 'error', text: "Duplikatni sloupec [karel]" },
   { type: 'info', text: "Prvni pouziti ukol.termin" },
 ],
 error: null, // null nebo vyjimka
 data: { ... } // samotna odpoved na dotaz, nebo null
}
```

Obr. 4.5: JSON objekt (Zdroj: Vlastní tvorba)

Modul Mongo

Modul Mongo využívá knihovnu *php5-mongo* a slouží pouze jako způsob konfigurace a získání příslušné *MongoCollection* handle (nebo *MongoDB*). Na začátku se načte nastavení pomocí *mongo_url* a *mongo_db*. Podle načtených informací si vytvoří *MongoClient* instanci a z db si vytáhne *MongoDB* instanci pro zvolenou databázi. Dále poskytuje funkci mongo, která vrací instanci MongoDB, nebo pokud dostane parametr jméno kolekce, tak vrátí MongoCollection. Poslední poskytovanou funkcí je *mongo_tables*, která vrací seznam tabulek v MongoDB s informací o počtu záznamů.

Modul Analýza

Modul Analýza je manuálně volán na příslušných místech z celkového API. Popíši zde všechny jeho funkce, jak ty čistě interní (odlišené netučně) tak ty které jsou volány zvenku (odlišené tučně).

Funkce **Analyza::table** je volána na počátku každého dotazu se jménem tabulky, ke které se snažíme přistupovat. Od databáze si zjistí seznam tabulek a ověří si, zda se zmiňovaná tabulka v seznamu vyskytuje. Pokud ne, pro dotazy které přidávají položky do databáze, přidá zprávu o vytvoření tabulky a tak či tak, provede Levenshtein analýzu

(viz Analyza::leven_tb) jména a nabídne podobně pojmenované tabulky (do vzdálenosti 5), pokud takové tabulky existují, v pořadí počtu jejich záznamů.

Analyza::leven_tb je interní funkce, která porovná jméno předané tabulky s tabulkami, které již existují v databázi. Vrací pole objektu obsahujících pro každou tabulku jméno (name), vzdálenost (levenshtein) a počet záznamů v tabulce. K samotnému výpočtu vzdálenosti používá PHP funkci levenshtein obsaženou ve standardní knihovně. Váha změny, vložení i odstranění je nastavena na 1.

S každým dotazem, který obsahuje dotazovací část s jiným parametrem než jen *id*, je volána **Analyza::query**. Tato funkce zpracuje příslušnou dotazovací část, vybere z ní jména všech zmíněných políček, odfiltrovávaje mongo operátory (toto řeší interní funkce*_policka*), a pro odpovídající záznamy v kolekci *_kikuchi_policka* zvýší počítadlo *query* každého zmíněného políčka.

Analyza::_policka jedná se o rekurzivně volanou interní funkci, sloužící pro extrakci seznamu políček z mongo dotazu. Z dotazu odstraňuje operátory, a spojuje jména políček z vnořených objektů tečkou.

```
{
  "priorita": { "$gt": 2 },
  "obsah": {
    "text": "Nakoupit!",
    "datum": { "$exists": true },
  },
}
```

```
[ 'priorita', 'obsah.text', 'obsah.datum' ]
```

Obr. 4.6: Příklad vlastního dotazu a extrahovaný seznam polí (Zdroj: Vlastní tvorba)

Funkce **Analyza::insert**, **Analyza::update** a **Analyza::remove** jsou téměř identické funkce sloužící k zpracování vložení, úpravy či odstranění záznamu. Jsou volány z příslušných míst v celkovém API. Vložení dostává jako parametr nový záznam, úprava předchozí

i nový stav a odstranění poslední verzi záznamu. Prvním krokem je vytvoření šablony záznamu o změně, která je pak předána rekurzivní funkci *_rec*, kvůli zpracování všech políček záznamu.

Rekurzivní funkce *Analyza::_rec* slouží k průchodu všemi vnořenými políčky jednoho nebo dvou (stará a nová verze) objektů. Pro každý klíč (jméno políčka) v objektech uloží informace o změně (pomocí funkce *_save*) a pokud hodnota tohoto políčka je pole či objekt, vnořeně zavolá sebe sama na tomto podobjektu. Při průchodu do hloubky si udržuje kompletní jména políček, včetně cesty, a ukládá je do příslušného záznamu.

Analyza::_save zprostředkovává samotné uložení záznamu o změně do databáze. Ukládá do tabulky *_kikuchi_log*, s aktuálním datem. Ostatní položky v záznamu pochází buď ze šablony od funkcí insert/update/remove nebo jsou to hodnoty specifické pro jednotlivá políčka (např. název, stará a nová hodnota), vyplněné ve funkci *_rec*. Funkce *_save* dále také vyvolává funkce, které je třeba volat pro každé políčko. Po převedení jména políčka (viz funkce *nazvy*), provede levenshtein analýzu pro detekci chyby ve jméně, zvýší u políčka počítadlo vložení, úprav či odstranění (ve funkci *_policko*), a provede analýzu typu na nové hodnotě, ukládaje výsledek do záznamu políčka.

Pro některé statistiky je vhodné udržovat informace jako dvojici jméno tabulky a cesta k políčku, u jiných je vhodnější pohlížet na vnořené objekty jako součást jména kolekce a udržovat jako dvojici celou cestu včetně tabulky a jména políčka. Tato první forma již byla zmíněna v popisu funkce *_rec*, která ji generuje, funkce *Analyza::nazvy* slouží tedy k převodu do druhé z těchto forem. Jak lze vidět na obrazku 4.7, tento převod odstraňuje informace o indexu prvku v poli, protože pro účely analýzy je vhodné předpokládat homogennost polí. Tedy první i druhý prvek pole budou typicky obsahovat stejná políčka se stejnými typy.

```
1 Analyza::nazvy("ukol", "priorita")==["ukol", "priorita"]
2 Analyza::nazvy("ukol", "obsah.datum")==["ukol.obsah", "datum"]
3 Analyza::nazvy("ukol", "obsah.tagy[0].nazev")==["ukol.obsah.tagy"
4 , "nazev"]
```

Obr. 4.7: Ukázka převodu (Zdroj: Vlastní tvorba)

Funkce *Analyza::leven_sl* je interní a porovnává jméno předávaného sloupce se všemi sloupci, které jsou v databázi. Tedy princip této funkce je podobný jako u *Analyza::leven_tb*, ale navíc používá druhou formu převodu z *Analyza::nazvy*, takže provádí i analýzu v rámci vnořených objektů.

Analyza::_policko na základě typu změny (insert, update, remove) zvyšuje (vždy o 1) příslušné počítadlo pro dané políčko v tabulce *_kikuchi_policka*. Pokud záznam pro dané políčko ještě neexistuje, automaticky je vytvořen s vynulovanými počítadly. Tento záznam pak také slouží pro počítadla použitá v dotazu (viz funkce *query*) a získané informace o typech (viz *typy*). Záznamy jsou identifikovány podle cesty a názvu políčka, jak je popsáno ve funkci *nazvy*.

Funkce *Analyza::type* slouží pouze ke zvýšení počítadel výskytu jednotlivých typů ve změnách hodnot daného políčka. K samotnému rozpoznání typu používá funkci *Analyza::_type*, která dostane hodnotu a vrátí pole všech typů, kterým hodnota odpovídá. Implementováno je rozpoznávání následujících typů:

- **int** - celočíselná hodnota nebo řetězec obsahující pouze číslice
- **float** – číslo s desetinnou čárkou nebo řetězec obsahující číslice s desetinnou čárkou
- **string** – hodnota je typu řetězec
- **bool** – true nebo false nebo řetězec true či false
- **date** – objekt typu datum nebo řetězec s datem a připadným časem v ISO[30] formátu
- **null** – hodnota je null. Tento typ slouzi k odlišení sloupců, které mohou mít hodnotu null od těch, které mají vždy hodnotu definovanou.

Analyza::messages jednoduše vrací seznam všech zpráv, které byly vygenerovány v průběhu zpracování dotazu. Tato funkce je volána z celkového API vždy při vytváření odpovědi (v poli messages). Samotné zprávy jsou vytvářeny interní funkcí *Analyza::msg*, která dostává jeden z předdefinovaných typů (info, warning, error), a text zprávy. Modul Analýza generuje nasledující zprávy:

- Informace o vytvoření nové tabulky
- Informace o tabulkách s podobným jménem při použití neexistující tabulky
- Informace o přidání nového sloupce
- Informace o sloupcích s podobným jménem při použití neexistujícího sloupce

[30]ISO 8601 je mezinárodní standard popisující reprezentaci data a času

- Informace o nastávajících výjimkách
- Informaci o použití Mongo vlastností, které nejsou podporovány v PostgreSQL

Celkové API

Mezivrstva nabízí tradiční REST API, které umožňuje snadnou interakci s frontendovou částí pomocí CRUD rozšířenou o několik dalších API endpointů pro obecné akce nad všemi záznamy dané kolekce – skupinové měnění či mazání – a metodu pro vylistování.

V CRUD metodách je navíc možnost přidání parametru *fields*, který zapříčiní, že nejsou vráceny všechny atributy, ale pouze explicitně zmíněné v url parametru *fields* (použito v routách *GET /<table>* a *GET /<table>/id*), jak můžeme vidět na obrázku níže 4.8. Dalším parametrem je options (použito v routách *POST /<table>/update* a *POST /<table>/remove*), kde v každé routě má své specifické hodnoty. Obdobně je tomu i tak u parametru query (navíc použitá i v routě *GET /<table>*). Aktualizace existujícího objektu se provádí pomocí PUT routy s id daného objektu, mazáni pomocí obdobného DELETE.

Skupinové měnění se provádí pomocí routy *POST /<table>/update*, která jednoduše obaluje mongo funkci update. Tato funkce akceptuje tři parametry:

- **query** – tzv. selektor – popisuje na základě jakých parametrů vybrat objekty, které chceme změnit. Je předáván jako url parametr *query*, ve formátu json.
- **update** – dokument, kterým bude nahrazen obsah nebo soubor akcí, které budou nad obsahem provedeny. Je předáván v těle POST dotazu, opět ve formátu json.
- **options** – nepovinný parametr, předávaný jako url parametr *options* (json), umožňuje měnit chování funkce update, zejména zda bude vytvořen záznam, pokud žádný odpovídající nebyl nalezen (upsert) a zda bude změna provedena na jednom či více záznamech (multi).

Routa *POST /<table>/remove* je velice podobná výše popsané routě. Akceptuje též parametry query a options (justOne – smazání pouze jednoho záznamu), jediné co chybí, je parametr update.

Mezivrstva umožňuje také mazání celé tabulky pomocí routy *POST/<table>/drop*, které není potřeba předávat kromě jména mazané tabulky žádné další parametry.

```
GET: http://.../kikuchi/table/?fields[]=title&fields[]=priority
{
    "action": "list",
    "table": "todo",
    "data": [
        {
            "_id": {
                "$id": "553fbc3ca4a4207a38d072cc"
            },
            "title": "Uklidit",
            "priority": 3
        },
        {
            "_id": {
                "$id": "553fbcbba4a4206c7cd072cb"
            },
            "title": "Uspinit",
            "priority": 2
        }
    ],
    "status": 200,
    "messages": [],
    "error": null
}
```

Obr. 4.8: Ukázka výpisu při použití parametru fields (Zdroj: Vlastní tvorba)

Dalšími důležitými routami jsou *GET /meta/tables* (jsou vráceny všechny tabulky – z Mongo databáze i z PostgreSQL), *GET /meta/leven/tables* (vrátí pro každou tabulku v databázi nejbližší kandidáty s minimálním počtem jednopísmenných změn do leven-

steinovy vzdálenosti 5), *GET /meta/leven/sloupce* (pracuje na stejném principu jako routa GET /meta/leven/tables, jen vrátí kandidáty pro všechny sloupce každé tabulky), *GET /meta/prevod/db* (provede převod všech vybraných tabulek z Mongodb do PostgreSQL) a *GET /meta/prevod/sql* (vytvoří sql dump pro všechny vybrané tabulky).

Celkové API také poskytuje custom routes, které již byly popsány v úvodu této kapitoly. Těmto routám dává k dispozici API s metodami *find*, *findId*, *findOne*, *insert*, *updateId*, *removeId*, *update* a *remove*.

Modul PostgreSQL

Tento modul se stará o převod Mongo dotazů na SQL dotazy. Není možné, aby převod byl 100%, jelikož Mongo podporuje dotazy, které nejsou možné v SQL jazyce vytvořit. Zejména se jedná o rozdíl mezi SQL a NoSQL přístupem, kde Mongo nevyžaduje stejnou strukturu pro všechny záznamy. Poskytuje tedy funkce, které v PostgreSQL struktuře nedávají smysl, například *$exist*, *$unset*, update option *upsert*, *$all* (kontroluje, zda pole obsahuje všechny zmíněné prvky zmíněné v těle operátoru), *$elemMatch* (operátor, který slouží pro zjištění, zda všechny prvky pole splňují určitou podmínku) a další.

Při implementaci byla zvolena množina operátorů, které mezivrstva bude podporovat. Tato množina byla zvolena na základě subjektivního zhodnocení používanosti operátorů a jejich kompatibility s PostgreSQL modelem. Jedná se o jednoduché porovnávací operátory (*$gt*, *$gte* ...), o příslušnost do pole (*$in*, *$nin*), logickou negaci (*$not*) a kombinaci několika podmínek (*$and*, *$or*) a pro aktualizaci – operátory *$set*, *$mul* a *$inc*.

Pro převod tabulky z Mongo do PostgreSQL bylo třeba navrhnout strukturu. Kvůli jednoduchosti implementace bylo navrženo schéma převodu, kdy všechna políčka na první úrovni ukládaného objektu, jsou uložena jako sloupce v PostgreSQL tabulce, zatímco vnořené objekty jsou jednoduše reprezentovány jako JSON sloupec. Primární klíč je kvůli kompatibilitě pojmenován *_id*, a pro převedené záznamy má původní hodnotu – tj. 24 znakový ObjectId. Pro nové záznamy je použito náhodně vygenerované UUID (v4)[31]. Vzhledem k úskalím při převodu reálných dat mají navíc všechny PostgreSQL sloupce povolenou hodnotu *null*. Indexy jsou vytvořeny pro všechny sloupce, které někdy byly

[31]UUID – universally unique identifier – je 128bitová hodnota používaná k identifikaci záznamu a kvůli předejití kolizím. V našem případě jde o v4, která je generovaná náhodně.

použity v query části dotazu nad danou tabulkou. Typy ne-json sloupců jsou detekovány na základě dat shromážděných modulem Analýza, v případě nejistoty je použit ten nejčastější (v uživatelském rozhraní je možnost výběru).

Implementovat parciální změny ve vnořených objektech by za použití této reprezentace znamenalo načíst původní záznam, provést změnu a uložit celý. Z tohoto důvodu tato funkcionalita není implementována a systém při pokusu o tuto změnu vypíše chybovou hlášku. Explicitní přepis celého objektu je samozřejmě nadále podporován.

Modul PostgreSQL poskytuje funkce pro generování dotazů *find* (SELECT dotazy a převádí část query a fields), *insert* (INSERT sql pro daný záznam a zpracování dat), *update* (UPDATE dotaz pro 1 záznam podle primárního klíče), *remove* (DELETE dotaz pro 1 záznam podle primárního klíče), *update_multi* (UPDATE dotazu pro úpravu více záznamů stejným způsobem), *remove_multi*.

Převod jednotlivých dotazů probíhá rekurzívním průchodem, který je víceméně stejný jak pro query, tak pro update dokumenty. Je možné ho rozdělit na dvě fáze. První fází je parsování, kdy funkce *parseQuery* obdrží objekt query a projde každé políčko objektu vytvářejíc strom pro další zpracování. Pokud se jedná o jeden z operátorů $and, $or nebo $nor, rekurzivně se zavolá tato funkce pro všechny jeho operandy a přidá do stromu příslušnou větev. Pokud jde o $text či $comment, jsou pouze přidány do stromu. Pro ostatní políčka zkontroluje, zda v sobě nemají operátory (například `policko: { $gt: 4 }`), které zpracuje, jinak pouze přidá hodnotu do stromu.

Druhou fází je skládání všech částí do konečné podoby dotazu. Tato fáze je různá pro query a pro update: query část (zpracovávaná funkcí *pg_where*) rekurzivně prochází vygenerovaný strom a pro každý operátor generuje příslušnou část SQL dotazu v mezikód dibi (umožňuje mít text dotazu rozdělen do série vnořených polí, navíc s podporou *%and* a *%or* operátorů nad polem). Výsledek je poté použit ve WHERE části složeného dotazu.

Druhá fáze pro update je odlišná vzhledem k omezením na změnu ve vnořených polí v PostgreSQL. Není třeba strukturu procházet rekurzivně, postačí pouze projít dvojice klíč na první úrovni vnoření a testovat přítomnost operátorů ve druhé úrovni. Zde jsou očekávány operátory *$inc*, *$mul*, *$set* a *$currentDate*, které inkrementují, násobí hodnotu, nastavují novou hodnotu a aktuální datum. První tři tyto operátory jsou převedeny na SQL, *$currentDate* je přepsán na hodnotu aktuálního data a času. Operator *$set* navíc podporuje

vnořené *$currentDate*. Při této konverzi jsou také převáděny vnořené objekty do JSON reprezentace. Stejně jako v Mongu se kontroluje, zda není kombinace dat (tj. mód, kdy přepíšeme všechny sloupce) a *$set* (mód, kdy přepisujeme jen zvolená pole) a případně je vyhozená výjimka. Výsledek tohoto procesu je použit pro SET část výsledného UPDATE dotazu. Pro INSERT je *$inc* přeloženo na nastavení přičítané hodnoty, *$mul* na 0, *$set* se chová stejně jako by nastavované políčko bylo v datech a *$currentDate* se překládá stejně jako v update.

Další výraznou odlišnosti Mongo jsou typy políček. MongoDB ukládá celý dokument jako celek a zachovává (ale dále neřeší) typy hodnot v něm uložených (zdali je to text, JSON pole či třeba číslo). Naopak PostgreSQL vyžaduje konzistentní typ pro celý sloupec. Z tohoto důvodu bylo nutné vytvořit převod typů pomocí dat nasbíraných modulem analýza. Seznam typů:

- **int** - integer
- **float** - float8
- **datum** - timestamp
- **string** - text
- **bool** - bool
- **pole a objekt** - json

4.4.2 Frontend a Klient

Frontend

Část frontend byla napsána v čistém JavaScriptu, z toho důvodu nebude tato práce fungovat pod verzemi nižšími než IE10. Používá relativně nové Promises API[32]. K samotnému AJAX dotazu používá nové Fetch API, který tento Promise vrací.

Po načtení se JavaScriptová část dotáže backendu na seznam tabulek uložených v databázi a následně vytvoří properties pro snadný přístup k tabulkám. Dále zpřístupňuje podobné API jako databáze MongoDB, s tou výjimkou, že použití callback parametru pro předání výsledku je nahrazeno použitím promise, které tyto funkce vracejí. Tato změna

[32]Promise API – reprezentuje výsledek asynchronní operace. Promise je v jednom ze tří odlišných stavů (čekající, naplněn a selhal)

slouží k usnadnění použití vzhledem k tomu, že všechny databázové dotazy jsou zde asynchronní.

Kromě funkcí, které volají příslušnou PHP route se správně převedenými parametry, frontend obsahuje i tři funkce :

- **_pomocne_promenne** – volá se na vstupních datech před insertem a updatem, odstraňuje z dat typické pomocné proměnné, které zřídkakdy chceme skutečně uložit – odstraňuje jakýkoli sloupec, jehož jméno začíná na znak dolar nebo podtržítko
- **_common** – volá se na všechny výstupní data, zpracuje a zobrazí zprávy od modulu analýza a dále volá funkci _convert_types; vrací odpověď data
- **_convert_types** – rekurzivně převádí speciální hodnoty (datum a timestamp) na js Date, id na řetězec a JSON řetězce na rozbalené objekty/pole

AngularJS

Framework angular je také podporován pomocí angular factory Kikuchi, kterou je možné zavolat jménem tabulky a poskytuje stejné funkce obalené pro AngularJS. Vzhledem k tomu, že všechny tyto funkce vrací promise, zapojení do digest cyklu angularu spočívalo jednoduše v převodu ES6[33] Promise na Angular $q promise.

Klient

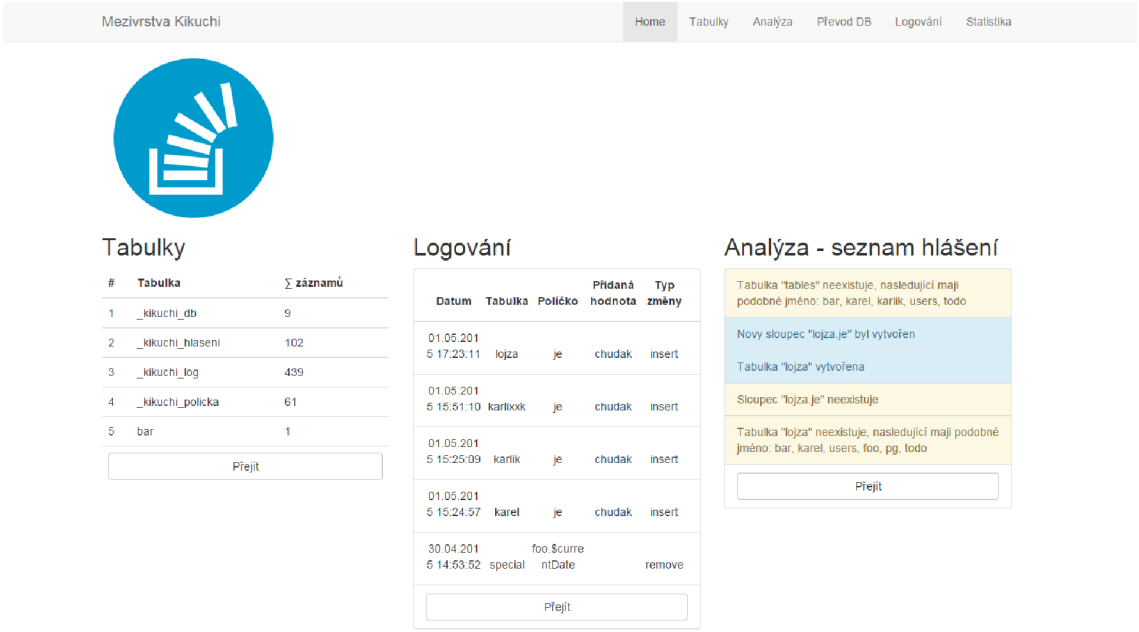
Implementace klienta byla provedena za pomocí technologií HTML, CSS, Bootstrap a několika JavaScriptových knihoven (angular, font-awesome, toastr, lodash, file-saver.js, Chart.js).

Níže rozepíši rozhraní pro správu, která poskytuje různé informace o tabulkách, statistiky o datech, grafy, logy přístupu a podobně. Dále umožňuje některé změny, jako například mazání sloupců a řádků, převod tabulek z Mongo do PostgreSQL .

Hlavní stránku rozhraní můžeme rozdělit do tří sekcí: tabulky, logování a seznam hlášení. V sekci tabulky je výpis 5 nejnovějších tabulek s počtem záznamů, kde pro výpis dalších tabulek stačí pouze stisknout tlačítko přejít a tabulky se zobrazí. V druhé sekci můžeme vidět 5 nejnovějších logů, jako u předchozí sekce je i zde možné přejít k celému výpisu logů a vidět všechny úpravy provedené v rámci vývojářského projektu. V poslední

[33] ECMAScript 6 – nadcházející standard pro jazyk JavaScript

sekci je výpis hlášení, které nastaly v průběhu vývoje aplikace. Celý výpis těchto hlášení můžeme najít v menu Analýza.



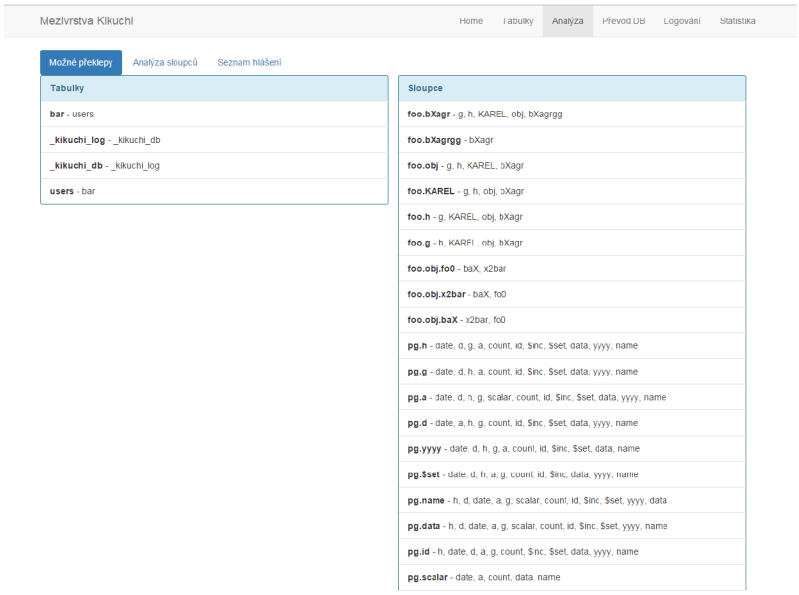
Obr. 4.9: Úvodní strana rozhraní (Zdroj: Vlastní tvorba)

Pokud v menu je vybrána položka Tabulky, zobrazí se stránka s výpisem všech tabulek dostupných jak v MongoDB, tak v PostgreSQL. Po kliku na vybranou tabulku se zobrazí detail této tabulky (struktura) a data obsažená v této tabulce. Zde lze odstraňovat nechtěná data, či odstraňovat sloupce, což je umožněno jen pro Mongo databázi.

Obr. 4.10: Stránka s výpisem všech tabulek (Zdroj: Vlastní tvorba)

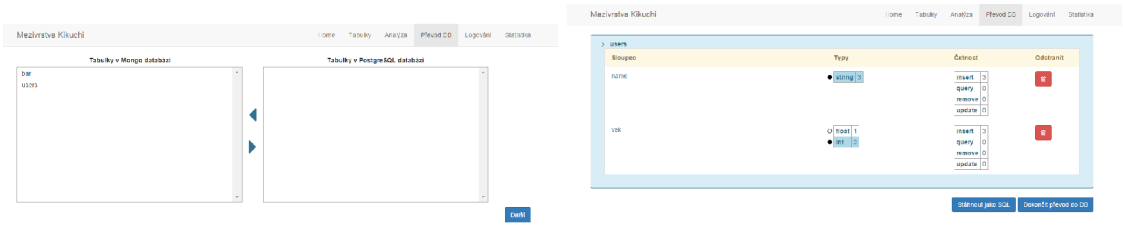
Na stránce Analýza se nachází tři taby: Možné překlepy, Analýza sloupců, Seznam hlášení. Tab Možné překlepy zobrazuje všechny tabulky i sloupce a k nim příslušné

nejpodobnější kandidáty. Analýza sloupců zobrazuje výpis všech tabulek s názvy políček, typy které mohou tato políčka mít, četnost přístupů k těmto políčkům a také vnořené sloupce, které jsou opatřeny tečkou pro reprezentaci vnořených polí. Tab seznam hlášení obsahuje obarvený seznam hlášení setříděný od nejnovějších.



Obr. 4.11: Stránka s analýzou (Zdroj: Vlastní tvorba)

Převod tabulek z MongoDB do PostgreSQL je možné při výběru položky v menu Převod DB. Zobrazí se stránka, kde je si možné vybrat tabulky vhodné pro převod. Při stisku tlačítka Další přejdeme na obrazovku, kde si uživatel může vybrat, jaký typ bude políčko v tabulce mít a vybrat všechna požadovaná políčka. Poté je možné stáhnout si dump SQL soubor určený pro nahrání tabulek i s daty do databáze nebo automaticky spustit převod těchto tabulek i s daty do databáze, bez mezikroku nahrání SQL souboru. Po převodu se tyto tabulky označí jako pg, aby bylo možné rozlišit, v které databázi se nacházejí. Není tedy třeba žádná změna v kódu komunikujícími s databází.



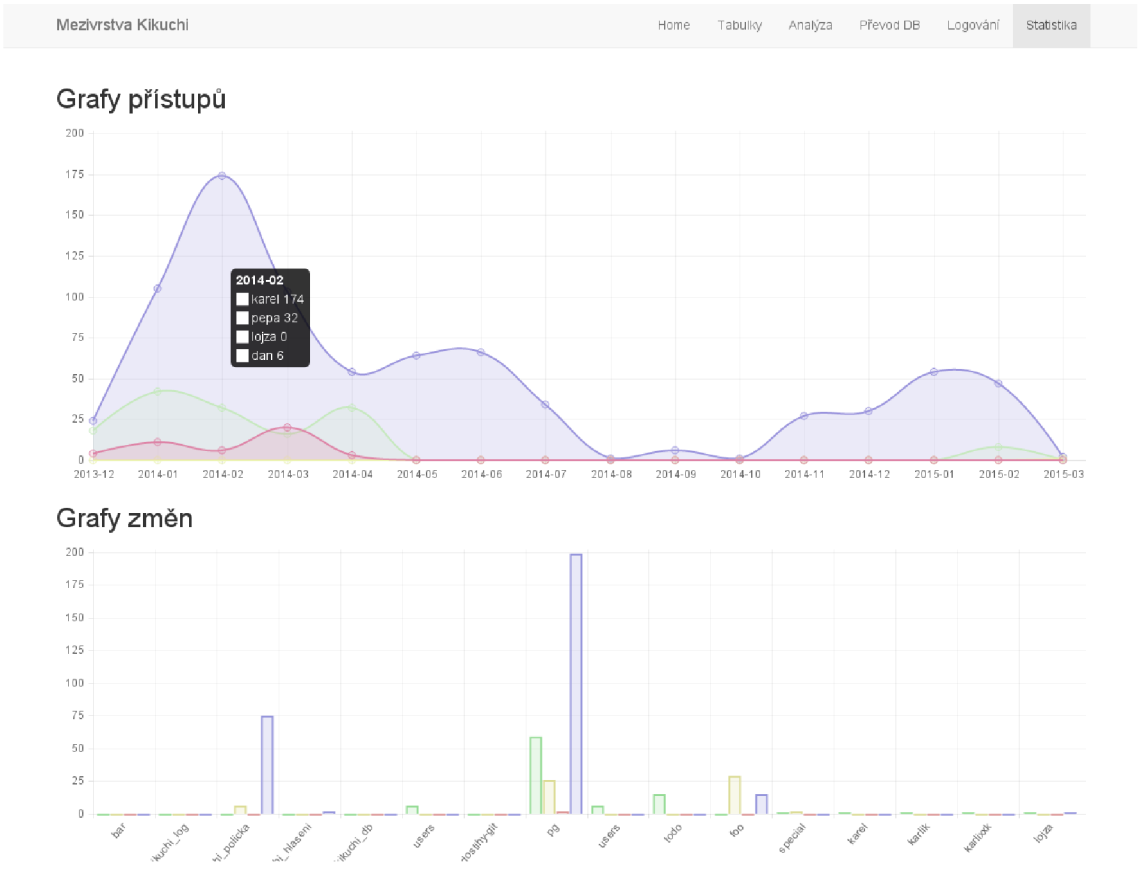
Obr. 4.12: Nalevo stránka s převodem, napravo výběr políček (Zdroj: Vlastní tvorba)

Na stránce logování je umístěn seznam všech logů, tzn. všech akcí provedených při vývoji aplikace. Tabulka obsahuje typ provedené změny, id, název tabulky a změněného políčka, předešlou a novou hodnotu a také datum této změny. Typ změny může být insert pro přidání nové hodnoty, update pro změnu existující a remove pro odstranění. Jméno políčka zde obsahuje celou cestu, může tedy například mít hodnotu *pole[2].jmeno*.

Datum a čas	Tabulka	Políčko	Id	Předešlá hodnota	Nová hodnota	Typ změny
01.05.2015 17:23:11	lojza	je	55439a5fa4a420cf5413369e		chudak	insert
01.05.2015 15:51:10	karlxxk	je	554504cea4a420c332133069e		chudak	insert
01.05.2015 15:25:09	karlik	je	55437cb5a4a4209a3113389e		chudak	insert
01.05.2015 15:24:57	karel	je	55437ea9a4a4209c3113389e		chudak	insert
30.04.2015 14:53:52	special	foo.$currentDate	55422225a4a420cc458892f8	true		remove
30.04.2015 14:53:52	special	foo.sec	55422225a4a420ce458892f8		1430398432	insert
30.04.2015 14:53:52	special	foo.usec	55422225a4a420ce458892f8		635000	insert
30.04.2015 14:53:52	special	foo	55422225a4a420ce458892f8	{"$currentDate":true}	"2015-04-30T12:53:52.635Z"	update
30.04.2015 14:38:39	special	foo	55422225a4a420ce458892f8	{"$currentDate":true}	{"$currentDate":true}	update
30.04.2015 14:38:39	special	foo.$currentDate	55422225a4a420ce458892f8	true	true	update
30.04.2015 14:37:57	special	foo.$currentDate	55422225a4a420ce458892f8		true	insert
30.04.2015 14:37:57	special	foo	55422225a4a420ce458892f8		{"$currentDate":true}	insert
30.04.2015 11:31:39	todo	extra	5541f67ba4a420a0670092f6		{"budik":"10:45"}	insert
30.04.2015 11:31:39	todo	extra.budik	5541f67ba4a420a0670092f6		10:45	insert
30.04.2015 11:31:39	todo	title	5541f67ba4a420a0678802f6		Vzuuu	insert
29.04.2015 16:33:58	_kikuchi_policka	update	553fbc3cc59ea02d03199fe5	0	0	update
29.04.2015 16:33:58	_kikuchi_policka	typ.int	553fbc3cc59ea02d03199fe5	1	1	update
29.04.2015 16:33:58	_kikuchi_policka	typ.null	553fbc3cc59ea02d03199fe5		1	insert
29.04.2015 16:33:58	_kikuchi_policka	tabulka	553fbc3cc59ea02d03199fe5	todo	todo	update
29.04.2015 16:33:58	_kikuchi_policka	typ	553fbc3cc59ea02d03199fe5	{"int":1}	{"int":1,"null":1}	update
29.04.2015 16:33:58	_kikuchi_policka	policko	553fbc3cc59ea02d03199fe5	priority	priority	update
29.04.2015 16:33:58	_kikuchi_policka	remove	553fbc3cc59ea02d03199fe5	0	0	update
29.04.2015 16:33:58	_kikuchi_policka	insert	553fbc3cc59ea02d03199fe5	1	1	update
29.04.2015 16:10:53	pg	date	16	2015-04-29T13:04:56+0000	2015-04-29T14:10:53+0000	update
29.04.2015 16:10:53	pg	count	16	2	2	update
29.04.2015 16:10:53	pg	data	16	{}	{}	update
29.04.2015 16:10:53	pg	name	16	Karel	Karel	update
29.04.2015 15:41:07	users	name	5540df73a4a4204a9c4ebf32		Kar32l	insert
29.04.2015 15:41:07	users	vek	5540df73a4a4204a9c4ebf32		5.5	insert
29.04.2015 15:41:00	users	vek	5540df6c4a4a4201e364ebf36		6	insert
29.04.2015 15:41:00	users	name	5540df6c4a4a4201e364ebf36		Kare2l	insert

Obr. 4.13: Stránka s výpisem logů (Zdroj: Vlastní tvorba)

Stránka statistika obsahuje dva grafy: graf přístupů (čárový graf obsahující pro každý měsíc od začátku vývoje aplikace počet log záznamů pro každou tabulku) a graf provedených změn (sloupcový graf obsahující pro každou tabulku hodnoty počítadel insert, update, remove a query). Tyto grafy je možno si stáhnout v podobě png souborů.



Obr. 4.14: Stránka s grafy (Zdroj: Vlastní tvorba)

4.5 Zhodnocení návrhu

V rámci této práce bylo dosaženo stanoveného cíle, tedy vytvoření návrhu mezivrstvy a následné implementace tohoto návrhu. Při návrhu bylo vycházeno z analýzy současného stavu, při kterém bylo zjištěno, že již existující řešení mají své nedostatky. Proto byl vytvořen návrh, který tyto nedostatky řeší zavedením mezivrstvy, která ukládá data do MongoDB a zároveň provádí analýzu těchto dat. Následně je možno na základě této analýzy se rozhodnout, jak budou vypadat tabulky a sloupce v produkční verzi. V průběhu vývoje bude umožněn vývojářům převod již hotových struktur tabulek do PostgreSQL databáze, kdy si mohou vybrat mezi automatickým nahráním těchto tabulek do databáze nebo stažením SQL souboru a následným nahráním do produkční databáze. Neznamená to však, že by tyto tabulky byly odstraněny z analýzy této vrstvy. Analýza kontroluje jak dotazy do MongoDB, tak i do PostgreSQL.

Aby bylo možné zjistit, zdali návrh řešení byl správně vytvořen, bylo jej nutné im-

plementovat a vyzkoušet. V průběhu implementace bylo postupováno relativně plynule, s výjimkou 2 větších úskalí. První problém nastal při vytváření modulu Analýza. Původně jsem zamýšlela použít Hammingovu vzdálenost při výpočtu vzdálenosti slov v detekci podobnosti, bohužel se toto ukázalo jako nesprávná volba a k tomuto výpočtu byl nakonec použit Levensteinův algoritmus. Druhý problém nastal při převodu Mongo dotazů na PostgreSQL dotazy. Jak je popsáno výše v kapitole 4.4.1, bylo nutné si zvolit množinu možných operátorů, které je možno převést. Obecný parser Mongo dotazů, který by nesloužil pouze k testování, se mi nepodařilo najít (zjevně neexistuje), a tedy bylo nutné implementovat tento parser a vymyslet, které operátory můžeme podporovat pro převod dotazů. Tato implementace tedy slouží jen pro zjištění, zdali byl návrh správně vytvořen, tedy obsahuje ještě mnoho nedostatků, které je nutné pro budoucí použití opravit. Například pro vnořená data by se mohly vytvořit tabulky, detekovat vazby a automaticky provádět JOIN při SELECT dotazech. Také by bylo možné implementovat větší množství operátorů, vyžadovalo by to ovšem přepis parseru a překladu updatu na jednorozměrné pole postupných změn. Nebylo by možné dotaz generovat, nýbrž jen rovnou spouštět, protože některé operace by vyžadovaly sérii několika na sobě závislých dotazů, obalených v transakci.

Celkově toto řešení poskytuje dobrý základ a dokonce zjednodušuje vývoj a prototypování aplikací. Pro budoucí užití by bylo vhodné přidat další výstupy a komplikovanější způsoby analýzy, poskytuje však dobrý základ pro další rozvoj této mezivrstvy.

Závěr

Cílem práce bylo vytvořit návrh vývoje webových aplikací s automatickým vytvářením databázového schématu. Při vytváření tohoto návrhu jsem vycházela z již existujících řešení, kde jsem se inspirovala zajímavými vlastnostmi vybraných implementací. Vytvořené řešení je možno použít k rychlé implementaci prototypu nebo i v dalších fázích jeho vývoje. Výsledné databázové schéma neobsahuje žádné explicitní vazby a obsahuje další potenciální nedostatky, je však již snadno upravitelné do finálního stavu.

Dopad použití implementace na výkon aplikace by byl velmi významný při produkční zátěži, ale při vývoji s několika uživateli je zanedbatelný. V okamžiku kdy aplikace dosáhne tisíce uživatelů, neměla by již tuto vrstvu používat, analýza samotná znamená přibližně pětinásobné množství databázových dotazů.

Jedním z možných rozšíření je kontrola práv uživatelů mezivrstvy. V rámci této práce je zmíněno, jak by tato kontrola zapadala do tohoto systému, nicméně za účelem prezentace práce předpokládám jen jednoho uživatele. Jak již bylo zmíněno výše, dalším rozšířením by byla podpora zpětných migrací. Toto by sloužilo k usnadnění návratu k předchozí verzi aplikace včetně formátu dat. Nicméně volba dokumentové databáze snižuje dopad chybějících migrací volnou strukturou databáze a zbývající problémy se vyskytnou jen při větší změně struktury dat.

Vzhledem k tomu, že mezivrstva dokáže generovat databázové schéma pro PostgreSQL databázi, není problém stejná data použít k vygenerování ostatních databázových schémat, generování definic tříd modelů pro jazyky používající ORM.

Pro zjednodušení vývoje programátorům zvyklých na jazyk SQL by bylo vhodné tuto vrstvu rozšířit o parser, který by umožnil přepsání SQL dotazů do Mongo dotazů. Tímto by se umožnila analýza i nad SQL dotazy a přibyla by možnost převodu tabulek z PostgreSQL databáze do MongoDB.

V neposlední řadě by bylo dobré rozšíření na detekci podobných tabulek podle jejich políček. Tímto by se dalo předejít duplikaci schématu databáze pro příbuzná data. Další fázi vývoje této práce by bylo zvětšení flexibility při práci s PostgreSQL databází a umožnění nastavení míry analýzy pro účely zefektivnění a použití v produkčních systémech s nižším počtem uživatelů. Případný další výzkum by šel směřovat k pokrytí potřeb v pozdějších

fázích životního cyklu aplikace.

Implementace mezivrstvy byla částečně validována jejím použitím při vývoji uživatelského rozhraní pro její správu. Velkým přínosem této práce je možnost kompletně se vyhnout změnám v backendu po velkou část vývoje aplikace.

Literatura

[1] AMAZON WEB SERVICES, I. *Amazon SimpleDB* [online]. 2015 [cit. 5.5.2015]. Dostupné na: <http://aws.amazon.com/simpledb/>.

[2] AMAZON WEB SERVICES, I. *Amazon SimpleDB* [online]. 2015 [cit. 5.5.2015]. Dostupné na: <http://docs.aws.amazon.com/AmazonSimpleDB/latest/DeveloperGuide/SDB_API_PutAttributes.html>.

[3] BASHO TECHNOLOGIES, I. *Riakdocs* [online]. 2015 [cit. 5.5.2015]. Dostupné na: <http://docs.basho.com/riak/latest/>.

[4] BHATT, L. *Object Relationship Mapping (ORM)* [online]. 2015 [cit. 5.5.2015]. Dostupné na: <http://tech.lalitbhatt.net/2014/07/object-relationship-mapping-orm.html>.

[5] CITRUSBYTE. *Redis* [online]. 2015 [cit. 5.5.2015]. Dostupné na: <http://redis.io/documentation>.

[6] CONSORTIUM, S. *SqlLite* [online]. 2015 [cit. 5.5.2015]. Dostupné na: <http://www.sqlite.org/docs.html>.

[7] CORP., I. *The Jalapeno Persistence Library for Java* [online]. 2015 [cit. 5.5.2015]. Dostupné na: <http://docs.intersystems.com/ens20101/csp/docBook/DocBook.UI.Page.cls?KEY=GBJJ_intro>.

[8] CORPORATION, O. *MySQL* [online]. 2015 [cit. 5.5.2015]. Dostupné na: <http://dev.mysql.com/doc/>.

[9] ELMASRI, R. a NAVATHE, S. *Fundamentals of Database Systems*. 6. vyd. : Addison-Wesley Publishing Company, 2010. ISBN 978-0-136-08620-8.

[10] FOUNDATION, D. S. *Django* [online]. 2015 [cit. 5.5.2015]. Dostupné na: <https://docs.djangoproject.com/en/1.7/ref/django-admin/#django-admin-makemigrations>.

[11] FOUNDATION, D. S. *Django* [online]. 2015 [cit. 5.5.2015]. Dostupné na: <https://docs.djangoproject.com/en/1.7/topics/migrations/>.

[12] FOUNDATION, T. A. S. *Apache CouchDB 1.6 Documentation* [online]. 2015 [cit. 5.5.2015]. Dostupné na: <http://docs.couchdb.org/en/1.6.1/>.

[13] GROUP, M. D. *METEOR 1.0 Documentation* [online]. 2015 [cit. 5.5.2015]. Dostupné na: <http://docs.meteor.com/#/full/concepts>.

[14] HARTL, M. *Ruby on Rails Tutorial: Learn Web Development with Rails*. 2. vyd. : Addison-Wesley Professional, 2012. ISBN 978-0321832054.

[15] HOLOVATY, A. a KAPLAN MOSS, J. *The Definitive Guide to Django: Web Development Done Right*. 1. vyd. Berkely, CA, USA: Apress, 2009. ISBN 978-1590597255.

[16] HOROWITZ, M. L. *An Introduction to Object-Oriented Databases and Database Systems*. 1. vyd. : Carnegie Mellon University, Information Technology Center, 1991. ISBN B0006RQ5WY.

[17] INC., A. *Mapping Objects to Relational Databases: O/R Mapping In Detail* [online]. 2015 [cit. 5.5.2015]. Dostupné na: <http://www.agiledata.org/essays/mappingObjects.html>.

[18] JEFF LINWOOD, D. M. *Beginning Hibernate*. 2. vyd. : Apress, 2010. ISBN 978-1-4302-2850-9.

[19] LERNER, A. *Ng-Book - The Complete Book on AngularJS*. 1. vyd. : Fullstack io, 2013. ISBN 978-0991344604.

[20] MONGODB, I. *MongoDB* [online]. 2015 [cit. 5.5.2015]. Dostupné na: <http://docs.mongodb.org/manual/>.

[21] OPPEL, A. *Databases: A Beginner's Guide*. 1. vyd. : McGraw-Hill, 2009. ISBN 978-0-07160-847-3.

[22] QUACKIT.COM. *SQL Server - Database Schemas* [online]. 2015 [cit. 5.5.2015]. Dostupné na: <http://www.quackit.com/sql_server/sql_server_2008/tutorial/sql_server_database_schemas.cfm>.

[23] REDHAT. *What is Object/Relational Mapping?* [online]. 2015 [cit. 5.5.2015]. Dostupné na: <http://hibernate.org/orm/what-is-an-orm/>.

[24] REDMOND ERIC, W. J. R. *Seven Databases in Seven Weeks: A Guide to Modern Databases and the NoSQL Movement*. 1. vyd. : Pragmatic Bookshelf Book, 2012. ISBN 978-1-93435-692-0.

[25] SADALAGE, P. a FOWLER, M. *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence*. 1. vyd. : Addison-Wesley, 2012. Always learning. ISBN 9780321826626.

[26] SILBERSCHATZ A., K. H. a S., S. *Database System Concepts*. 5. vyd. : McGraw-Hill, 2005. ISBN 978-0-07-295886-7.

[27] TRAVERS, C. *Perspectives on LedgerSMB* [online]. 2015 [cit. 5.5.2015]. Dostupné na: <http://ledgersmbdev.blogspot.cz/2012/08/intro-to-postgresql-as-object.html>.

[28] TREEHOUSE ISLAND, I. *I Don't Speak Your Language: Frontend vs. Backend* [online]. 2015 [cit. 5.5.2015]. Dostupné na: <http://blog.teamtreehouse.com/i-dont-speak-your-language-frontend-vs-backend>.

[29] WIKIMEDIA FOUNDATION, I. *Schema migration* [online]. 2015 [cit. 5.5.2015]. Dostupné na: <https://en.wikipedia.org/wiki/Schema_migration#Schema_migration_in_agile_software_development>.

Seznam obrázků

Seznam příloh

Obsah přiloženého CD

- **diplomka.pdf** – diplomová práce v elektronické podobě
- **kod/*** – zdrojové kódy implementace, rozhraní aplikace, detailněji níže
  - **app.js** – controllery a routování celé aplikace
  - **bower.json** – popis externích závislostí
  - **directives.js** – definice obecných komponent pro použití v angular šablonách
  - **index.html** – GUI Kikuchi mezivrstvy
  - **util.js** – pomocné funkce pro třídění, filtrování, atd.
  - **css/*** – zdrojové kódy kaskádových stylů
  - **images/*** – složka s obrázky - logo aplikace
  - **view/*** – zdrojové kódy šablon jednotlivých HTML stránek
    * **analyza.html** – HTML šablona tabu Analýza
    * **confirm_modal.html** – HTML šablona obecného potvrzovacího formuláře
    * **home.html** – HTML šablona tabu Home
    * **logovani.html** – HTML šablona tabu Logování
    * **menu.html** – HTML šablona menu
    * **prevod.html** – HTML šablona tabu Převod DB
    * **statistika.html** – HTML šablona tabu Statistika
    * **table-detail.html** – HTML šablona detailu jednotlivé tabulky
    * **table-view.html** – HTML šablona tabu Tabulky
  - **kikuchi/*** – zdrojové kódy mezivrstvy
    * **composer.json** – popis externích závislostí
    * **analyza.php** – zdrojový kód logiky celé analýzy
    * **conf.php** – defaultní nastavení pro připojení k databázi
    * **dispatch.php** – funkce pro práci s routami, HTTP stavy a přesměrováním
    * **mongo_meta.php** – funkce a routy pro práci s MongoDB - seznam tabulek, políček, Levenshtein analýza
    * **util.php** – funkce pro práci s HTTP parametry, PHP poli, apod.
    * **angular.js** – JS modul pro použití mezivrstvy v Angular.js aplikaci -

Kikuchi factory

* **customroutelib.php** – funkce pro použití v Custom routách - obaly nad Mongo funkcemi provádějící analýzu, třída CustomRoute
* **js.js** – JS modul pro použití mimo Angular aplikace (angular.js na něm ale závisí)
* **mongo_rest.php** – routy pro dotazování do MongoDB a akce nad kolekcemi
* **customroutes.php** – předpřipravený soubor pro vytváření uživatelských rout s příkladem jedné routy
* **migration.php** – funkce a routy pro převod z MongoDB do PostgreSQL
* **postgres.php** – funkce pro parsování a převod jednotlivých mongo dotazů do SQL
* **dibi.php** – inicializace externí knihovny pro práci s SQL databází
* **mongo.php** – inicializace externí knihovny pro práci s Mongo databází
* **rest.php** – jádro API, které includuje všechny PHP soubory

Postup instalace

Ke spuštění této práce je potřeba následující software: PostgreSQL 9.3+, MongoDB 3.0+, PHP 5.4+ a Apache od verze 1.2.

Postup je následující:

1. Naimportování databáze ze souboru *diplomka.json* do MongoDB databáze
2. Nastavení adresáře *kód* jako DocumentRoot
3. Upravit soubor *dibi.php* – přístupy k databázi
4. Nastavení webserveru, vytvoření souboru *kikuchi/conf.local.php* s řádkem *conf('mongo_db', 'JMENO PROJEKTU')*, případně i *mongo_uri*, pokud databáze běží jinde, než na lokálním serveru, a *url_under* pokud je na webserveru pod jiným adresářem než */kikuchi*
5. Pro nastavení spojení s PostgreSQL databází je nutné vyplnit parametry *pg_host*, *pg_username*, *pg_password* a *pg_database* v souboru *kikuchi/conf.local.php*.
6. Pro načtení PHP knihoven je potřeba spustit composer v adresáři kikuchi.
7. Pro instalaci příslušných závislostí je potřeba spustit bower install v adresáři kód

Overview

Classes summary	
Analyza	Statická třída pro analýzu dotazů
Arr	Implementace pole/hashe s utility metodami
CustomRoute	statická třída umožňujcí deklarovat custom routy; předaný callback by měl vracet objekt s daty
EmptyObj	Třída která pro každý atribut vrací NULL
Kikuchi	statická třída obalující akce nad databázemi, zajišťujíc volání modulu Analýza
Part	Interní třída reprezntujicí část MongoDB query objektu
Route	statická třída pro definici routes

Functions summary	
_aget	Pomocná funkce pro get a post
_file_writer	Vrátí funkci která zapisuje své parametry do zvoleného souboru
_fixurl	pomocná - řeší normalizaci URL routy - koncové lomítko
_pfx	private; hledá společný url prefix
_wrapcall	pomocná - obaluje route callbacky kvuli předání parametrů a odchycení vyjímek
array_first	vrací první položku pole
array_in	Pomocná funkce pro zjištení, zdali je proměnná v poli
array_kv_filter	stejná jako nativní array_filter, ale callback dostává klíč i hodnotu
array_kv_map	stejná jako nativní array_map, ale callback dostává klíč i hodnotu
coalesce	vrací první ne-falsy položku z pole (nebo null když žádná)
codegen	Generuje zapamatovatelné heslo nastavitelné délky
conf	nastavuje nebo vrací hodnotu konfiguračního parametru
dibi_sql	pro lazení - vrací SQL vygenerované z daného dibi query objektu

dispatch	Zavolá vybranou routu podle aktuálního URL, nebo defaultní
fieldOperand	Vrací true pokud je objekt Mongo operand
fields	Převede pole jmen políček do formátu pro Mongo.find
findone	Najde záznam daného id v MongoDB
flatten	spojí vnořená pole
get	Funkce pro vytažení GET parametrů; bez parametrů vrací true pokud *nešlo* o POST/PUT dotaz
groupby	GROUP BY - kategorizuje pole podle hodnoty zvoleného atributu
id	Funkce identita - vrací svůj první parametr beze změny
is_assoc	Tastuje zda se jedná a hash - tj. pole s nečíselnými indexy
isXmlHttp	isXmlHttp vrací true pokud se jedná o AJAX request
json	vrátí odpověď ve formátu JSON, včetně HTTP hlaviček
json_input	načte vstupní POST data jako JSON, vrací objekt či pole
k_is_date	Zjišťuje zda je parametr řetězec s datumem v ISO formátu
kde_su	Pomocná funkce pro zjištení, která tabulka patří do jaké databáze
matchData	Zpracování pole Part - pomocná pro pg_data
matches	Pomocná funkce pro spojení množiny operací
meta_tables	Pomocná funkce pro spojení výpisu tabulek z obou databází
method	vrací funkci která na jí předaném objektu zavolá metodu zvoleného jména
migrace_data	Funkce pro vytvorení SQL dotazu ukládající data do tabulek
migrace_db	Funkce, která provádí vnitřní logiku převodu
migrace_done	Pomocná funkce pro uložení informace o převedené tabulce a její odstranění z Mongodb
migrace_soubor	Funkce pro vytvoření SQL souboru s vybranými tabulkami a políčky
migrace_tabulky	Funkce pro vytvoření SQL dotazu vytvářejícího

	jednotlivé tabulky
model	pomocná funkce pro Model - vrací nacachovaný model objekt pro danou kolekci
mongo	vrací zvolenou mongo kolekci
mongo_tables	Pomocná funkce pro získání všech tabulek z MongoDB
obj2ary	Pomocná funkce pro převod objektu do pole
omit	vrátí kopii objektu *bez* zvolených atributů
parseElemMatch	Řeší elemMatch operátor
parseFieldOperator	Vrací Part objekt pro Mongo operaci
parseNot	Řeší not operátor
parseQuery	Pomocná funkce pro rozparsování query dotazu
partData	Zpracování jedné Part - pomocná pro pg_data
partMatches	Zpracování Mongo operatorů v dotazu - obecný dispatch
path	Vrací cestu (aktuální nebo specifikovanou) s odstraněným prefixem
pg_columns	Seznam sloupců v PostgreSQL tabulce
pg_data	Funkce pro zpracování Mongo operátoru v datech a pro zpracování JSON políček - generuje SQL pro UPDATE nebo INSERT
pg_findone	Najde jeden záznam v PostgreSQL tabulce, podle id, ekvivalent mongo_findone
pg_tables	Seznam tabulek v PostgreSQL
pg_where	Převádí query na WHERE část dotazu pro SELECT, UPDATE či REMOVE; vrací sql pole pro dibi, bez WHERE
pick	vybere zvolené atributy z objektu
pluck	Funkce pluck - z pole objektů a jména atributu udělá pole hodnot daného atributu
policka_tabulky	Vrátí hash ve kterém je pro každou tabulku pole jejich polí
post	Funkce pro vytažení POST parametrů; volána bez parametru vrací true pokud šlo o POST/PUT dotaz
preber_dotaz	Funkce pro převod Mongo dotazů do SQL - význam dalších parametrů závisí od hodnoty jmeno_funkce, která víceméně odpovída jménu funkce v MongoDB
prefix_field	Pomocná funkce sloužící pro vytvoření JSON

	selektoru pro SQL
qh	Rozdělí string do hashe
query	Převede query z json do formátu očekávaného MongoDb a provede analýzu query
qw	Rozdělí string do pole, podle mezer
redirect	Přesměruje na specifikované URL
require_dir	zavolá require na všechny php soubory v daném adresáří, v setříděném pořadí
rescue	zavolá předanou funkci, odchytávaje vyjímky
route	Definuje novou route
route_default	Definuje defaultní route
s2t	Formátuje počet sekund od začátku dne do H:M:S formátu
set	vrací hash kde všechny klíče jsou hodnoty ze vstupního pole - perl množina
status	nastaví návratový HTTP status a zprávu
typ2sql	Pomocná funkce pro převod jména typu na SQL typ
uniq	uniq - vrací pole bez duplikátních položek
uri	Vrací aktuální URI (bez parametrů) nebo URI pro zvolenou route

Class Kikuchi

statická třída obalující akce nad databázemi, zajišťujíc volání modulu Analýza

Located at customroutelib.php

Methods summary		
public static	`find( $table, $query = [], $fields = [] )` vrátí pomnožinu záznámů dané kolekce	#
public static	`findId( $table, $id, $fields = [] )` vrátí záznam s daným id	#
public static	`findOne( $table, $query = [], $fields = [] )` vrátí jeden/první záznam splňující danou podmínku	#
public static	`insert( $table, $data )` vloží data do db	#
public static	`updateId( $table, $id, $data )` aktualizuje záznam daného id	#
public static	`removeId( $table, $id )` odstraní záznam daného id	#
public static	`update( $table, $query, $data, $options = [] )` aktualizuje množinu záznamů	#
public static	`remove( $table, $query, $options = [] )` odstraní množinu záznamů	#

Class Analyza

Statická třída pro analýzu dotazů

Located at analyza.php

Methods summary		
public static	table($tabulka, $insupd = false) analyzuje jméno tabulky - vola levenshtein analýzu a vytváří zprávu pokud neexistuje	#
public static	leven_tb($tabulka) levenshtein analýza jména tabulky - vrací seznam kandidátu s vzdáleností	#
public static	leven_sl($tabulka, $policko) levenshtein analýza jména slouce; jméno tabulky může být složené	#
public static	nazvy($tabulka, $policko) převod ("tabulka", "pol.ick[0].o") na ("tabulka.pol.ick", "o")	#
public static	_policko($tabulka, $policko, $typ_zmeny) zvětšuje počítadla insert, update a remove pro dané políčko	#
public static	type($tabulka, $policko, $value) zvětšuje počítadla detekovaného typu pro dané políčko	#
public static	_type($value) detekuje možné typy daného políčka	#
public static	policko_existuje($tabulka, $policko) vrací informace o políčku, pokud existuje; očekává formu tabulka=a.b.c policko=d	#
public static	_save($data) ukládá log o změně, volá levenshtein analýzu jména sloupce a analýzu typu, generuje zprávy	#
public static	_rec($zaznam, $objekt, $prefix = "", $objekt2 = null) rekurzivní průchod a detekce změn	#
public static	insert($table, $new) analýza dat pro insert	#
public static	update($table, $old, $new) analýza dat pro update	#

public static	remove($table, $old) # analýza dat pro remove	
public static	_policka($query, $prefix = "", $dol = false #) rekurzivní průchod selektoru - vrací seznam políček v něm zmíněných	
public static	query($table, $query) # analýza dat pro query	
public static	msg($type, $text) # interní funkce pro vytvoření zprávy	
public static	messages( ) # vrací seznam zpráv, pro API	

Properties summary

public static array	$messages	[] #
public static array	$zaznam	[# 'policko' => '', 'tabulka' => '', 'new_hod' => null, 'old_hod' => null, 'id' => null, 'datum' => null, 'typ_zmeny' => '', 'typ_db' => 'mongo',]

Class CustomRoute

statická třída umožňující deklarovat custom routy; předaný callback by měl vracet objekt s daty

Located at customroutelib.php

Methods summary		
public static	get($url, $callback) GET	#
public static	post($url, $callback) POST	#
public static	delete($url, $callback) DELETE	#
public static	put($url, $callback) PUT	#

Global

Methods

_common(res) - skupina akcí prováděná pro každou přijatou odpověd - odchycení a vypsání chyb a převod typů()

Source: js.js, line 174

_pomocne_promenne(obj): Object - rekurzivně odstraňuje pomocné proměnné ()

Source: js.js, line 158

drop() - odstraní kolekci z databáze()

Source: js.js, line 119

Kikuchi.init() vytvoří Kikuchi.cokoli pro každou (již použitou) kolekci()

Source: js.js, line 11

Class: Kikuchi

Kikuchi

Kikuchi Třída Kikuchi je kolekcí instancí třídy KTable, pojmenovaných podle existujících kolekcí v databázích.

Constructor

`new Kikuchi()`

Source: js.js, line 4

Members

`(static) _promise`

Source: js.js, line 9

`(static) ready`

Properties:

Name	Type	Description
ready	bool | null	pokud true, inicializace se zdařila

Source: js.js, line 7

Class: KTable

KTable

KTable odpovídá jedné kolekci v DB

Constructor

`new KTable()`

Source: js.js, line 38

Members

`table`

Properties:

Name	Type	Description
`table`	string	jméno kolekce

Source: js.js, line 40

`table_info`

Properties:

Name	Type	Description
`table_info`	object	informace o kolekci

Source: js.js, line 42

Class: KikuchiAngular

KikuchiAngular

KikuchiAngular - obsahuje pouze statickou metodu init

Constructor

```
new KikuchiAngular()
```

Source: angular.js, line 6