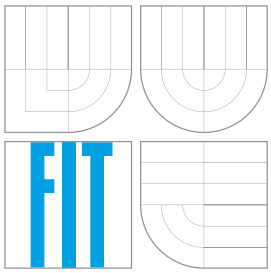
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

SLEDOVAČ AKTUÁLNÍHO DĚNÍ

ACTUAL EVENT TRACKER

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. MARTIN ODSTRČILÍK

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. JAN KOUŘIL

BRNO 2013

Abstrakt

Cílem diplomové práce bylo vytvoření aplikace pro sledování aktuálního dění v okolí jejího uživatele. Tato aplikace by měla umožňovat jejím uživatelům události nejen sledovat, ale také přidávat své vlastní či komentovat již existující. Diplomová práce se mimo tvorbu dané aplikace zaobírá analýzou předloženého problému. Analýza zahrnuje průzkum existujících řešení, dostupných technologií a aplikačních rámců využitelných k implementaci. Součástí práce je i popis teorie klasifikace dat, která je v rámci vyvíjené aplikace použita k analýze událostí a komentářů. V textu práce je dále zahrnut návrh řešení, jenž se zaměřuje na návrh uživatelského rozhraní, architektury aplikace, databáze, komunikačního protokolu a klasifikátorů. Jádrem práce je pak popis implementace aplikace. V závěru práce je pak shrnut její průběh a jsou navrhována vhodná rozšíření do budoucna.

Abstract

The goal of the master thesis project was to develop an application for tracking of actual events in the surrounding area of the users. This application should allow the users to view events, create new events and add comments to existing ones. Beyond the implementation of developed application, this project deals with an analysis of the presented problem. The analysis includes a comparison with existing solutions and search for available technologies and frameworks applicable for implementation. Another part inside this work is description of the theory in behind of data classification that is internally used for event and comment analysis. This work also includes a design of appliction including design of user interface, software architecture, database, communication protocol and data classifiers. The main part of this project, the implementation, is described aftewards. At the end of this work, there is a summary of the whole process and also there are given some ideas about enhancing the application in the future.

Klíčová slova

Android, Spring framework, Hibernate framework, RESTful služby, návrhové vzory, MVC, klasifikace textu, SVM, TF-IDF

Keywords

Android, Spring framework, Hibernate framework, RESTful services, design patterns, MVC, text classification, SVM, TF-IDF

Citace

Martin Odstrčilík: Sledovač aktuálního dění, diplomová práce, Brno, FIT VUT v Brně, 2013

Sledovač aktuálního dění

Prohlášení

Prohlašuji, že jsem diplomovou práci vypracoval samostatně pod vedením pana inženýra Jana Kouřila a že jsem uvedl všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Martin Odstrčilík
20. května 2013

Poděkování

Tímto bych rád poděkoval panu inženýru Janu Kouřilovi za svědomité a příkladné vedení při řešení diplomové práce.

Obsah

Kapitola 1

Úvod

Žijeme v době sociálních sítí. Pojem být sociální již zdaleka neznamená stýkat se s přáteli na tradičních místech, potkávat se na ulicích a prohodit několik společenských vět, nýbrž mít účet na Facebooku, Twitteru a jim podobných službách. Většina dění se odehrává právě zde, kde se lidé informují o tom, co se událo, přidávají své zážitky, pocity. Častým trendem je mít na sociální síti co nejvíce přátel a být členem co největšího počtu skupin s cílem ukázat světu, že jsem tou nejvíce společenskou osobou. Uživatelé vstřebávají informace ze sociálních sítí každý den, mnozí i několikrát denně a nelze tak pochybovat o tom, že sociální sítě jsou silným informačním médiem schopným předat požadovanou informaci lidem po celém světě.

S rostoucím počtem uživatelů sociálních sítí se na ně přesunuly i organizace a firmy. Svoji působnost zde v drtivé většině směřují k propagaci své agendy. V konečném důsledku se tak náš sociální život zaplní tím, co můžeme vidět na billboardech u dálnic a na každém rohu ve městech. Relevantnost informací získaných ze sociálních sítí se tak ztrácí v záplavě marketingových akcí, událostí a zpráv – v reklamě.

Malou revoluci na poli mobilních zařízení přinesl rok 2007 a společnost Apple se svým produktem iPhone a operačním systémem iOS [14]. Jeho inovátorský vzhled, přístup k ovládání a prezentaci údajů započal éru mobilních telefonů, jak je známe dnes. Ještě téhož roku přišla s odpovědí společnost Google, která dala za vznik operačnímu systému pro mobilní telefony – Android [17]. Po dohodě s předními výrobci mobilních zařízení a mobilními operátory té doby vzniklo uskupení Open Handset Alliance a začaly postupně vznikat přístroje s tímto operačním systémem. Oběma platformám se podařilo překonat tehdejší konkurenci – BlackBerry, Windows Mobile/Phone či Symbian OS. Dnes můžeme říci, že právě platformy Android a iOS, dle statistiky zastoupení na trhu [42], určují trend těchto zařízení.

Přání obyvatel celého světa získávat informace skrze sociální sítě je jistě možné využít k mnoha účelům. Jedním z nich se zabývá tato práce. Jejím cílem je vytvořit aplikaci na bázi sociální sítě pro sledování aktuálního dění. Pomocí této aplikace by se k uživatelům měly dostat informace o tom, co se děje v jejich okolí, ať už se jedná o dopravní nehodu, havárií potrubí, výpadky proudu, internetu, demonstrace, společenské události, přítomnost revizorů v hromadné dopravě či policejní hlídky. Přidávání jednotlivých události bude plně v režii uživatelů a bude záležet na každém, o kterých událostech, dle kategorie, chce být informován. Příslušnost události do jedné či více kategorií bude aplikace schopna rozeznat sama pomocí analýzy obsahu události a následné klasifikace. Aplikace by také měla poskytovat pouze relevantní obsah, čehož chce dosáhnout analýzou reakcí uživatelů, resp. komentářů k přítomným událostem. Aplikace tak bude mimo kategorizaci událostí posuzovat i jejich sentiment s cílem nabídnout uživatelům právě aktuální a žádané informace.

Povaha aplikace vyžaduje schopnost přispívat do sítě v okamžiku vzniku události. Pro uživatele musí být vložení události jednoduchý, rychlý a proveditelný úkon odkudkoliv a kdykoliv. Jako ideální řešení se nabízí využití mobilních zařízení – chytrých telefonů a tabletů. Logickým vyústěním popsaných skutečností je vytvoření aplikace pro sledování aktuálního dění s prvky sociální sítě na některé z předních mobilních platforem.

1.1 Struktura práce

Kapitola 2 se zaobírá průzkumem trhu a existujících řešení daného problému. Dále obsahuje, mimo zevrubný popis dostupných technologií a aplikačních rámců, také důvody, proč byly právě tyto vybrány. Další kapitola 3 pojednává o teorii na pozadí klasifikace textu a jsou zde rozebrány nejčastěji používané přístupy. V kapitole 4 jsou detailněji popsány použité technologie, návrhové vzory v klientské i serverové části a jsou nastíněny vybrané algoritmické postupy při klasifikaci textu. Návrh samotné aplikace je rozebrán v kapitole 5. Stěžejní částí celé práce – implementaci, je věnována kapitola 6. Poslední kapitola 7 pak shrnuje celkový průběh práce, dosažené výsledky a pojednává o budoucnosti aplikace zejména z pohledu zdokonalování funkcí a služeb pro uživatele.

1.2 Návaznost na semestrální projekt

Tato práce přejímá ze semestrálního projektu určitou část již hotové práce. Jedná se zejména o větší část textu úvodní kapitoly 1, analýzu problému uvedenou v kapitole 2 a část obsahu kapitol 4 a 5. Tyto převzaté kapitoly jsou v rámci diplomové práce zaktualizovány a doplněny o části zabývající se klasifikací textů. Hlavní přidanou hodnotou této práce k semestrálnímu projektu je samostatná kapitola 3 zabývající se teorií klasifikace dat a dále pak kapitola 6 popisující implementaci výsledné aplikace. Ke změnám oproti semestrálnímu projektu došlo i v rámci závěrečné kapitoly 7 hodnotící celou práci.

Kapitola 2

Analýza problému

V první části kapitoly je diskutována konkurence vyvíjené aplikace. Jsou zde rozebrány oblíbené (dle počtu stažení a hodnocení uživatelů) aplikace jak pro platformu Android, tak iOS. Následně jsou popsány možné přístupy k implementaci vyvíjené aplikace. Dále se kapitola soustředí na jednotlivé technologie, které vyplynuly na základě analýzy problému jako vhodné pro vývoj. Ke každé z nich je uveden důvod jejího výběru, základní charakteristika a krátký přehled jejich kladů a případných nedostatků.

2.1 Dostupná řešení

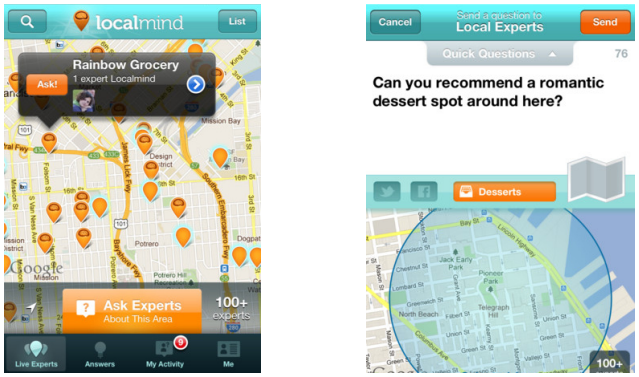
Průzkum trhu na podobná, nebo dokonce stejná řešení problému, byl zaměřen na aplikace pro platformy Android a iOS, neboť právě tyto společně dle statistiky [42] pokrývají 63,97 % trhu a získaný vzorek je možné považovat za dostatečně reprezentativní. Při analýze dostupných řešení se nepodařilo nalézt žádné takové, které by se svojí funkcí zcela blížilo k vytvářené aplikaci. Jednotlivá řešení se buď specializují na jeden typ událostí, například dopravní nehody, uzavírky cest, policejní hlídky, kontroly revizorů v hromadné městské dopravě anebo se snaží o komplexnější záběr, kde doporučují cestovatelům a návštěvníkům dané lokality restaurace, památky, turistické trasy a jim podobné oblasti zájmu.

2.1.1 Localmind

Jedná se o aplikaci, která se zaměřuje na poskytování informací vztahujících se k probíhajícím událostem v určité lokaci. Aplikace pracuje na principu „*check-in*“ známého například ze sociální sítě Foursquare [5]. Princip spočívá v ohlášení uživatele, že je návštěvníkem daného místa nebo události a ostatní uživatelé se pak, v případě zájmu, mohou ohlášené osoby, resp. experta (jak jej aplikace nazývá) dotázat na doplňující informace. Získávají se tak odpovědi na aktuální otázky, kterých nelze dosáhnout jinak, než samotnou přítomností v daném místě. Jedná se například o délku čekání na jídlo v restauraci, právě hranou píseň na koncertě nebo stav terénu turistické trasy. Ukázku aplikace je možné shlédnout na obrázku 2.1. [6]

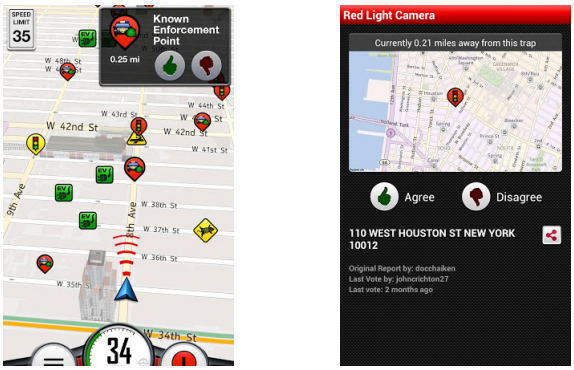
2.1.2 Trapster

Tato aplikace poskytuje uživatelům informace o policejních hlídkách měřících rychlost, kamerových systémech hlídající provoz na světelných křižovatkách, rychlostních omezeních a jim podobných dopravních hlášeních. Jednotlivá hlášení jsou přidávána samotnými uživateli



Obrázek 2.1: Uživatelské prostředí aplikace Localmind pro Apple iPhone. Přehled událostí v okolí uživatele aplikace je zobrazen na prvním obrázku. Možnost zaslání dotazu expertovi v dané oblasti pak ilustruje druhý obrázek [6].

aplikace nebo jejími provozovateli s využitím příbuzných databází. Uživatelům je dovoleno nahlášenou událost hodnotit a tím ověřovat její aktuálnost a relevantnost. Užitečnou funkcí aplikace je zvýraznění trasy, která byla v nedávné době prověřena jiným uživatelem a vzhledem k nenahlášení žádného problému, je tato považována za „bezpečnou“. Autoři aplikace deklarují více než 18,9 miliónů uživatelů po celém světě. Vzhled aplikace je přiblížen na obrázku 2.2. [7]



Obrázek 2.2: Uživatelské prostředí aplikace Trapster pro Android. První obrázek zachycuje výchozí obrazovku, která uživateli poskytuje základní informace o rychlosti a nástrahách v okolí ve směru jízdy. Na druhém obrázku je pak ilustrován způsob, jakým uživatelé hodnotí aktuálnost, resp. validitu nástrah [7].

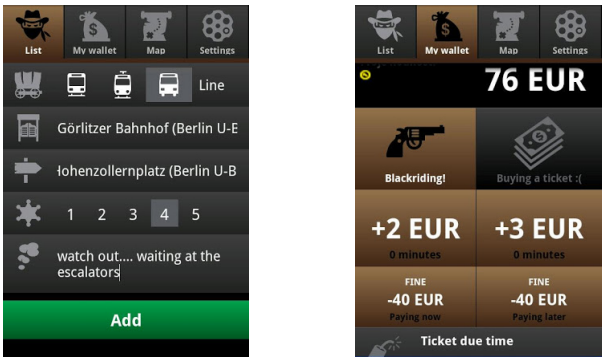
2.1.3 Dopravní informace tudyNE

Jedná se o službu poskytující aktuální informace z dopravy na území České republiky. Služba je dostupná jak pomocí webového prohlížeče, tak pomocí nativní aplikace pro mobilní zařízení s operačním systémem Android. Přínosem aplikace pro uživatele, zejména ty, kteří používají mobilní verzi, je možnost nastavení si sledované oblasti a upozornění při změnách situace v dané lokalitě. Ukázku prostředí aplikace je možné shlédnout na obrázku 2.3. [8]

Obrázek 2.3: Uživatelské prostředí aplikace tudyNE pro Android. Obsahem prvního obrázku je mapa s jednotlivými dopravními událostmi. Po kliknutí na požadovanou událost je zobrazen její detail, který je ilustrován druhým obrázkem. [8]

2.1.4 FareBandit

Hlavní funkcí aplikace FareBandit je upozorňovat cestující využívající hromadnou dopravu v podporovaných městech na přítomnost revizorů v jednotlivých linkách. Údaje o výskytu a pohybu revizorů se do aplikace dostávají pomocí hlášení uživatelů. Uživatelé jsou rovněž v případě výskytu nové hrozby aplikací na tuto skutečnost upozorněni. Zajímavou doplňkovou funkcí je přehled jízd, které uživatel učinil bez jízdenky a kolik tím ušetřil peněz. Vzhled aplikace je ilustrován obrázkem 2.4. [4]



Obrázek 2.4: Uživatelské prostředí aplikace FareBandit pro Android. První obrázek ilustruje formulář pro přidání revizora, kde uživatel volí typ hromadné dopravy, zastávku, počet revizorů a vlastní komentář. Druhý obrázek pak zachycuje peněženku, která obsahuje statistiky používání aplikace (počet jízd s nebo bez jízdenky a množství ušetřených financí) [4].

2.2 Možný přístup k řešení

V úvodní části této práce byly naznačeny hlavní cíle a požadavky vytvářené aplikace. V této části jsou jednotlivé myšlenky dále rozvíjeny až do podoby, kterou je možné označit za neformální specifikaci problému. Aplikace by měla sloužit ke sledování několika druhů událostí

v okolí uživatele. K tomuto účelu je nezbytné pracovat s prostorovými údaji jednotlivých událostí – s jejich polohou. Jelikož je zamýšleno, že aplikace bude simultánně používána několika uživateli, neboť se snaží použít koncept sociálních sítí, je žádoucí celou aplikaci implementovat jako architekturu klient – server.

Klientská část bude sloužit uživatelům jako prostředek k prohlížení a práci s jednotlivými událostmi. K rozlišení uživatelů bude potřeba vytvořit přihlašovací a registrační systém. Pro tuto funkci je na místě použít již existující služby, které uživatel potenciálně využívá, například Facebook, Twitter či Google. Nicméně by aplikace neměla podmiňovat vlastnictví účtu v některé ze jmenovaných služeb, ale měla by nabídnout i možnost registrovat se pouze pomocí emailu a hesla. Po registraci, resp. přihlášení bude uživateli umožněno zobrazit seznam nebo mapu s událostmi a také přidávat nové události. Uživatelé aplikace mohou rovněž události komentovat. S každou událostí se bude mimo její titulek, popis a kategorii ukládat i její GPS pozice, datum a čas vzniku. Pozice události bude ve výchozím stavu zaznamenána z GPS polohy zařízení, na kterém klientská aplikace běží. Polohu bude ale možné zadat i manuálně výběrem místa na mapě. V případě, že se bude jednat o událost, kterou přidal sám uživatel, bude ji moci upravit a smazat. Stejná podmínka vlastnictví pak platí i pro správu komentářů událostí.

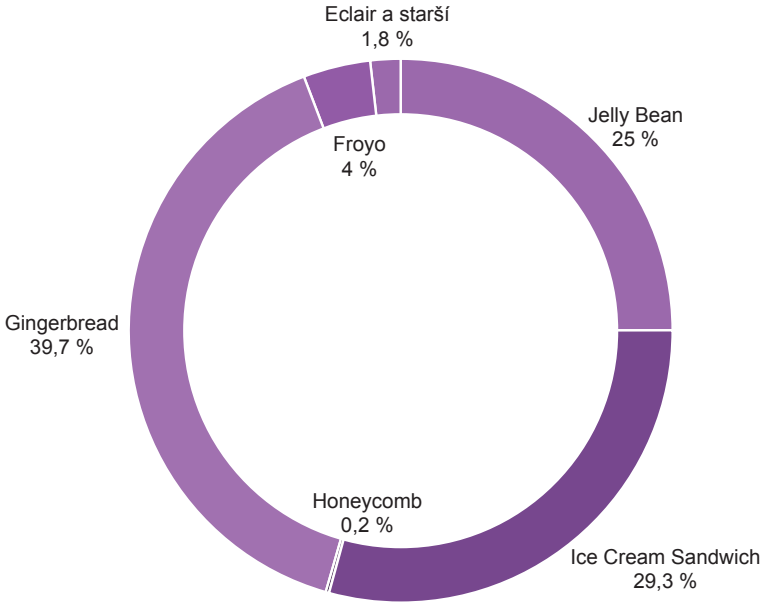
Uložení událostí a komentářů z klientských aplikací bude zajišťovat serverová část. Mimo persistenci dat bude server také provádět dodatečné výpočty nad událostmi. Bude se jednat například o výpočet vzdálenosti události od uživatele a analýzu obsahu. Pro komunikaci mezi klientskou a serverovou částí by mělo být využito dostupného internetové připojení. Z tohoto pohledu je nutné vytvořit komunikační protokol. Přístup k serverové části by měl být umožněn pouze autorizovaným klientským aplikacím.

K oběma částem je potřeba vybrat technologie, pomocí kterých bude provedena implementace. Jak již bylo předesláno v úvodu, pro klientskou část je vhodné, z důvodu nutnosti přístupu k událostem a jejich vytváření kdekoli a kdykoli, použít mobilní zařízení. V návaznosti na tuto volbu a s uvážením rozložení trhu mobilních platforem je zřejmou volbou platforma iOS nebo Android. Výběr technologie pro serverovou část tolik jednoznačný není, a proto bude na místě zvolit takovou, které se bude vhodně párovat s technologií klienta.

2.2.1 Android

K implementaci klientské části byla vybrána platforma Android jakožto zástupce jedné ze dvou dominantních na trhu s mobilními zařízeními. Dalším faktorem pro její volbu je osobní zkušenost s vývojem pro tuto platformu a rovněž znalost programovacího jazyka Java, který se pro implementaci aplikací pro tuto platformu používá. Ve prospěch Androidu hraje i fakt, že vývoj na této platformě je možný na kterémkoli běžném operačním systému – Windows či Linux a pro testování na fyzickém zařízení není z mé strany potřeba nakupovat dodatečný hardware.

Naopak úskalím Androidu je velké množství různých zařízení, na kterých běží. Zařízení se odlišují především verzí operačního systému (viz graf na obrázku 2.5) a rozlišením displejů (viz tabulka 2.1). To přináší z pohledu vývoje problémy se zpětnou kompatibilitou a také je potřeba se vypořádat s omezeními danými velikosti displejů. Popsané problémy jsou poměrně dobře odstíněny například na platformě iOS, neboť zařízení s tímto operačním systémem je v současné době pouze 18, existují jen 4 hlavní vývojové verze a v podstatě 3 rozlišení displejů [1].



Obrázek 2.5: Rozdělení verzí Androidu mezi distribuovanými zařízeními k 17. 4. 2013 [20].

	ldpi	**mdpi**	**hdpi**	**xhdpi**	**celkem**
small	9,5 %				9,5 %
normal	0,1 %	16,1 %	37,9 %	25,8 %	79,9 %
large	0,7 %	2,7 %	1,5 %	0,8 %	5,7 %
xlarge	0,1 %	4,6 %	0,1 %	0,1 %	4,9 %
celkem	10,4 %	23,4 %	39,5 %	26,7 %	

Tabulka 2.1: Rozdělení velikostí (*small* – 426dp × 320dp, *normal* – 426dp × 320dp, *large* – 640dp × 480dp a *xlarge* – 960dp × 720dp)[1] a hustoty (*ldpi* – 120dpi, *mdpi* – 160dpi, *hdpi* – 240dpi a *xhdpi* – 320dpi)[2] displejů mezi distribuovanými zařízeními s Androidem k 17. 4. 2013 [20].

Samotný Android je vyvíjen společností Google, nicméně celý ekosystém je zastřešen společenstvím Open Handset Alliance, které je tvořeno více než 300 softwarovými a hardwarovými firmami a mobilními operátory z celého světa. Android je rovněž označován za nejrychleji se rozvíjející platformu [18]. O oblíbenosti Androidu svědčí i fakt, že k 15. 5. 2013 bylo prodáno 900 miliónů zařízení s tímto operačním systémem [12].

Z technologického pohledu, jak již bylo nastíněno na začátku této části, jsou aplikace pro Android psány v jazyce Java. Od klasické Javy se ale Android odlišuje zejména vlastní implementací virtuálního stroje, který se nazývá Dalvik. Také Android nepoužívá Java bytecode, ale Dalvik bytecode. Více detailů k technologii platformy Android lze nalézt v knize [24] a práci [33].

[1] Hodnota (výška × šířka) značí minimální rozměr displeje pro spád do dané kategorie. Jednotka *dp* značí obrazový bod nezávislý na hustotě displeje. Z této hodnoty je při zobrazení systémem spočtena výsledná hodnota v tradičních obrazových bodech (pixelech).

[2] Jednotka *dpi* značí počet obrazových bodů na jeden palec.

2.2.2 Architektura REST

Komunikační protokol mezi klientskou a serverovou částí by měl být co nejjednodušší, aby přenos dat probíhal dostatečně svižně a příliš nezatěžoval spojení mobilního zařízení – vzhledem k rychlostem základního mobilního připojení a FUP (*Fair User Policy*) limitu. Protokol by rovněž měl být vytvořen tak, aby nebyl závislý na použité technologii klienta a serveru, a aby se s ním pohodlně pracovalo.

Vhodným kandidátem je architektura REST (*Representational State Transfer*). Pomocí ní je možné vytvořit uniformní a rovněž jednoduché rozhraní pro výměnu dat mezi distribuovanou architekturou, kterou klient – server je. V terminologii architektury REST jsou data a stavy aplikace označovány jako zdroje, kde každý z nich je označen svým unikátním identifikátorem v podobě URI (*Uniform Resource Identifier*). Pomocí dané URI je možné se zdroji manipulovat. Definovány jsou 4 typy operací známé pod zkratkou CRUD (*Create, Read, Update, Delete*). Samotnou reprezentaci dat přenášených protokolem je možné zvolit na základě způsobu dalšího zpracování. K přenosu se využívá HTTP protokol a příkazy GET a POST, eventuálně PUT nebo DELETE. Architektura REST je bezstavová, takže se musí v každé zprávě přenášet veškeré informace potřebné například k autorizaci pro přístup ke zdrojům. [16]

V porovnání s protokolem SOAP (*Simple Object Access Protocol*) je REST považována za odlehčenou architekturu, kde není potřeba dodatečný popis pomocí XML. Architektura REST je také shledávána jako flexibilnější pro možnost přenosu dat mimo formát XML také například v JSON, HTML či prosté textové podobě. V kontrastu je SOAP považován za bezpečnější, neboť nabízí díky zabalování do XML typovou kontrolu a navíc obsahuje podporu pro atomické transakce a některé dodatečné bezpečnostní funkce, například WS-Security. Protokol SOAP je rovněž, na rozdíl od architektury REST, standardizován. [37]

2.2.3 Spring framework

Výběr technologie pro implementaci serverové části aplikace vycházel z již vybraných technologií pro klientskou část a komunikační protokol – Android a REST. Obě technologie jsou ve Spring frameworku oficiálně podporovány, a proto se jeho použití přímo nabízí. Dalším impulzem byla i jistá zkušenost s vývojem aplikace postavené na tomto frameworku. Spring framework se vyznačuje několika klíčovými vlastnostmi, kterými jsou zejména [41, 27, 34]:

- snaha o odlehčení velmi komplexní Java EE (*Java Enterprise Edition*),
- staví na návrhových vzorech Inversion of Control (*IoC*) a Dependency Injection (*DI*), které odbourávají těsné vazby mezi objekty,
- snaží se programátorovi poskytnout možnost zaměřit se na architekturu aplikace namísto psaní věcí kolem,
- podporuje dostupné knihovny pro přístup k databázi a perzistentním objektům – JDBC a Hibernate,
- usnadňuje testování pomocí jednotkových testů a
- je modulární – programátor může použít pouze část, kterou pro daný projekt potřebuje.

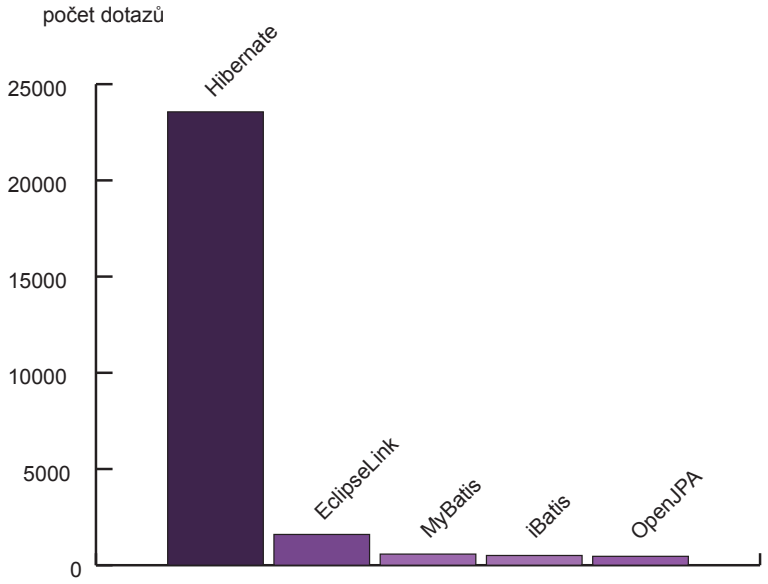
Podpora architektury REST je do Spring frameworku zanesena přímo jako součást modulu pro tvorbu webových aplikací – Spring MVC. Při zaslání dotazu na server provede framework

dle nastavení automaticky namapování požadavku na parametry obslužné funkce. Mapovat je možné jak hodnoty z URL, tak tělo přijaté zprávy. Programátor má pomocí Springu kontrolu i nad zasíláním odpovědí ze serveru. U odpovědi je možné kompletně nastavit hlavičku odesílané zprávy. Nastavitelné jsou třeba atributy jako HTTP kód, datum a čas expirace nebo modifikace a jiné. [39]

Rozšíření ke Spring frameworku – Spring for Android je knihovna pro zjednodušení práce při komunikaci se vzdálenou službou pomocí REST rozhraní. Knihovna obsahuje REST klienta a podporu pro autentizaci k službám zabezpečeným pomocí protokolu OAuth. V kombinaci se Spring MVC a jeho podporou REST architektury se jedná o velmi pohodlný způsob propojení klientské a serverové části. [40]

2.2.4 Hibernate framework

Další nutnou funkcí aplikace, resp. serverové části, je ukládání uživateli vytvořených událostí. K tomuto účelu byl vybrán framework Hibernate. Jedná se o knihovnu pro objektově-relační mapování (ORM) objektů do relační databáze, čímž je zajištěna jejich persistence mezi stavy aplikace. Oproti klasickému přístupu za použití JDBC (*Java Database Connectivity*) přináší ORM mimo jiné velké zjednodušení, co se množství napsaného kódu týče. To s sebou přináší méně programátorských chyb. Hibernate jako takový byl zvolen z důvodu, že se jedná známý a o značně používaný ORM framework pro Javu (viz graf 2.6), a je zároveň podporován v rámci Spring frameworku. Navíc je plně kompatibilní s JPA (*Java Persistence API*). Hibernate je v podstatě jeho implementací. Specifikace JPA vznikla jako součást Java EE standardu pro ukládání objektů do relační databáze. Pokud je v kódu tento standard dodržován, je velmi jednoduché zaměnit jeho implementaci. Například místo Hibernate použít ORM framework EclipseLink. Mapování objektů na databázové tabulky



Obrázek 2.6: Množství dotazů k jednotlivým ORM frameworkům pro programovací jazyk Java dle serveru *stackoverflow.com* ke dni 17. 4. 2013.

je možné realizovat buďto pomocí definic v XML souboru, anotací nebo kombinací obou přístupů. Hibernate zvládá jak relaci 1:1 a 1:N, tak i N:M. Pozitivní vlastností je používaný

dotazovací jazyk HQL (*Hibernate Query Language*) podobný SQL, který je funkčně přenositelný na jiné databázové technologie, resp. dialekty. Určitou nevýhodou Hibernate je schopnost perzistentně ukládat pouze POJO (*Plain Old Java Object*) objekty. Tyto objekty musí obsahovat neparametrizovaný konstruktor, což ale z pohledu vyvíjené aplikace nečiní žádné problémy. [2]

2.2.5 RapidMiner

K vytvoření datových struktur pro klasifikátory sentimentu a kategorií událostí se nabízí použití platformy pro dolování dat RapidMiner. Jedná se o platformu, která poskytuje uživatelům práci skrze velmi pohodlné a intuitivní uživatelské prostředí. Platforma je rovněž volně dostupná a multiplatformní. Při práci s RapidMinerem není potřeba mít žádné programovací znalosti a celý klasifikátor je možné si interaktivně vytvořit z grafického prostředí. Vytváření klasifikátoru je tak velmi rychlé. RapidMiner lze také použít jako knihovnu v rámci vlastní Java aplikace. Alternativou k RapidMineru může být například knihovna Weka [11] nebo systém R [9]. Obě alternativy jsou pomocí rozšíření integrovatelné do RapidMineru. [10]

Kapitola 3

Klasifikace dat

Tato kapitola se zaměřuje na popis teorie v pozadí klasifikace dat, resp. kategorizace textů, která je ve vyvíjené aplikaci použita k analýze sentimentu a kategorizaci událostí. V kapitole je uveden postup během klasifikace textů a následně jsou popsány často používané klasifikátory a princip jejich fungování. V závěru kapitoly jsou shrnuty metody pro vyhodnocování kvality, resp. přesnosti modelu klasifikátoru.

3.1 Klasifikace textů

Klasifikace nebo také kategorizace dat je používána v několika oblastech informačních technologií. Mezi známé aplikace patří například detekce nevyžádané pošty a je možné se s ní setkat také při analýze provozu v internetových sítích. Klasifikaci dat lze automatizovat použitím algoritmů strojového učení, kdy je pomocí předem označených dat natrénován klasifikátor, který je následně schopen analyzovat a kategorizovat neznámá data. Tento postup se nazývá učení s učitelem. Opačným přístupem je učení bez učitele, kdy je klasifikátor trénován na neoznačených datech (nejsou stanoveny třídy klasifikace) a vyhodnocování probíhá na základě shlukování sobě podobných dat. Při klasifikaci textů se ptáme, zda analyzovaný dokument spadá do jedné či více předem definovaných tříd. Na základě počtu tříd jsou pak použity různé typy klasifikace. Pokud testovaný dokument patří nebo naopak nepatří do právě jedné třídy, hovoříme o binární klasifikaci. Řešeným problémem může být také situace, kdy dokument může patřit zároveň do k disjunktních tříd. Tento problém je pak označován jako *multi-class*. Pokud by dané třídy nebyly disjunktní, hovoří se o *multi-label* problému. V praktické aplikaci se posledně jmenovaný problém převádí na řešení, kdy je použito k binárních klasifikátorů a každý vstupní dokument je pak testován k-krát. Trénovací data pro tento problém jsou koncipována tak, že za pozitivní jsou brána ta, která odpovídají právě klasifikované třídě a zbylá, patřící do ostatních tříd, jsou vůči ní považována za negativní. [26, 13, 43]

3.2 Vektor příznaků

Pro samotnou klasifikaci textů je potřeba nejprve vstupní data převést na reprezentaci, která je vhodná pro strojové učení a tedy samotnou úlohu klasifikace – tzv. vektor příznaků. Jedná se o n-dimenzionální vektor sestávající z jednotlivých slov (termů) nebo jejich skupin vstupního dokumentu. Proces vytváření vektoru příznaků zahrnuje nejprve nahrazení všech velkých písmen slov jejich malou variantou. Dále jsou věty rozčleněny na základě oddělovače

do posloupnosti tokenů, ze kterých jsou následně odfiltrovány stop slova (předložky, spojky a podobné prvky jazyka). Poté je na zbylé tokeny aplikována metoda *stemming* (viz článek [35]), která ze slov odstraňuje jejich koncovku[1]. Následně jsou vyfiltrovány jen ty tokeny, které mají alespoň určitou délku. Konečnou úpravou bývá ořezání celkového počtu příznaků dle množství jejich výskytů ve vstupních dokumentech. Tato metoda se nazývá anglicky *prunning* a je často užívána například v rozhodovacích stromech. [25, 23]

3.2.1 Ohodnocení vektoru příznaků

K ohodnocení vektoru příznaků je dle Thorstena Joachimse [25] vhodné použít váhovací schéma TF – IDF (*term frequency – inverse document frequency*), které přiřazuje jednotlivým termům dokumentu jejich váhu. První hodnota $TF(w_i, d)$ určuje počet výskytů termu w_i v dokumentu d. Hodnota $IDF(w_i)$ je počítána vzorcem 3.1, kde $DF(w_i)$ značí počet dokumentů, ve kterých se term w_i vyskytuje, a n značí celkový počet dokumentů. Výsledná hodnota TF – IDF se spočítá součinem hodnot $TF(w_i, d)$ a $IDF(w_i)$.

$$IDF(w_i) = \log\left(\frac{n}{DF(w_i)}\right) \tag{3.1}$$

Výsledný vektor příznaků daného dokumentu $\vec{d_i}$ je následně kvůli možnosti rozdílných délek všech dokumentů normalizován tak, aby velikost vektoru vůči ostatním byla v Euklidovské metrice 1. Normalizace se provede podělením vektoru jeho délkou, viz vzorec 3.2.

$$\vec{v}(d_1) = \frac{\vec{V}(d_1)}{|\vec{V}(d_1)|} \tag{3.2}$$

Vektory příznaků všech dokumentů tvoří model vektorového prostoru (angl. *vector space model*), ve kterém každý term zabírá jednu dimenzi v prostoru. Vektorový prostor je možné reprezentovat maticí termů dokumentů (angl. *term-document matrix*) o velikosti $M \times N$, kde M je počet řádků (dimenzí) daný počtem termů a N je počet sloupců matice na základě počtu dokumentů. [29, 25]

Podobnost dokumentů, resp. jejich vektorů příznaků se při následné klasifikaci provádí pomocí kosinové podobnosti, která počítá podobnost $sim(d_1, d_2)$ dokumentů d_1 a d_2 na základě jím odpovídajících vektorů příznaků $\vec{V}(d_1)$ a $\vec{V}(d_2)$ za použití vzorce 3.3, kde je počítán skalární součin již vektorů příznaků a ve jmenovateli je násobena jejich velikost v Euklidovské metrice.

$$sim(d_1, d_2) = \frac{\vec{V}(d_1) \cdot \vec{V}(d_2)}{|\vec{V}(d_1)||\vec{V}(d_2)|} \tag{3.3}$$

Po provedení normalizace vektorů příznaků $\vec{V}(d_1)$ a $\vec{V}(d_2)$ vzorcem 3.2 se výpočet kosinové podobnosti zjednoduší na vtah 3.4. [29]

$$sim(d_1, d_2) = \vec{v}(d_1) \cdot \vec{v}(d_2) \tag{3.4}$$

Model vektorového prostoru velmi často dosahuje vysokého počtu dimenzí a následně pak není možné klasifikátor dobře natrénovat. Buď je časová a paměťová náročnost učení neúnosná nebo dochází k jevu označovanému jako přeučení (angl. *overfitting*). Přeučení nastává ve chvíli, kdy je výsledný model klasifikátoru nepřesný z důvodu, že popisuje spíše

[1] Slova *computes*, *computer* a *computing* mají po aplikaci metody *stemming* podobu *comput* [25].

šum a chyby v trénovacích datech, něž z pohledu klasifikace důležité a významné termy. Model klasifikátoru, který trpí přeučením, sice vykazuje velmi dobrou přesnost na trénovacích datech, ale pro neznámá data tomu tak není. Nastává zde ztráta obecnosti (angl. *generalization*). Tento jev je možné ověřit například pomocí křížové validace, viz část 3.4. K redukci počtu dimenzí se využívá několika postupů a metod popsaných dále. [23, 25]

3.2.2 Výběr příznaků

Metody pro výběr příznaků pracují na principu výběru a ohodnocování podmnožin ze vstupního vektoru příznaků. Cílem je výběr takové podmnožiny, která obsahuje nejmenší počet redundantních a irelevantních příznaků. Za redundantní příznaky jsou považovány ty, které ve výsledném modelu mají stejnou informační hodnotu. Irelevantní příznaky jsou pak takové, které do modelu naopak nepřináší žádnou informační hodnotu. Metody výběru příznaků se dělí do 3 tříd:

- **Wrappers** je třída, která pracuje s klasifikátorem jako černou skříňkou, kdy algoritmy spadající do této třídy iterativně vybírají na základě algoritmu klasifikátoru podmnožinu vektoru příznaků. V každé iteraci je zvolená podmnožina doplňována nebo naopak zužována o příznaky s cílem zlepšit přesnost výsledného modelu. Jelikož algoritmus v každém kroku vytváří pro každou podmnožinu vektoru příznaků nový model, který je používán na daném klasifikátoru pro testování přesnosti, je průběh algoritmu výpočetně velmi náročný. Algoritmy z této skupiny poskytují z uváděných tříd nejlepší výsledky, ale jsou také nejvíce náchylné na přeučení.

- **Filters** je skupina algoritmů, které jsou na rozdíl od *wrappers* nezávislé na používaném algoritmu pro klasifikaci nebo predikci a k odhadu relevance používají statistické ohodnocení. Rovněž jsou tyto algoritmy oproti předchozí skupině výpočetně méně náročné.

- Algoritmy ze třídy **embedded** jsou kombinací předchozích tříd. Průběh metod vychází ze třídy *wrappers*, nicméně dosahují menší časové náročnosti a náchylnosti k přeučení jako je tomu u skupiny *filters*. Embedded algoritmy jsou typicky součástí procesu vytváření klasifikačního modelu, a jsou tudíž zabudovány přímo v samotném algoritmu klasifikátoru. Poskytují tak vhodnou podmnožinu pro zvolený klasifikátor. Příkladem je například algoritmus RFE (*Recursive Feature Elimination*) z klasifikátoru SVM (*Support Vector Machine*).

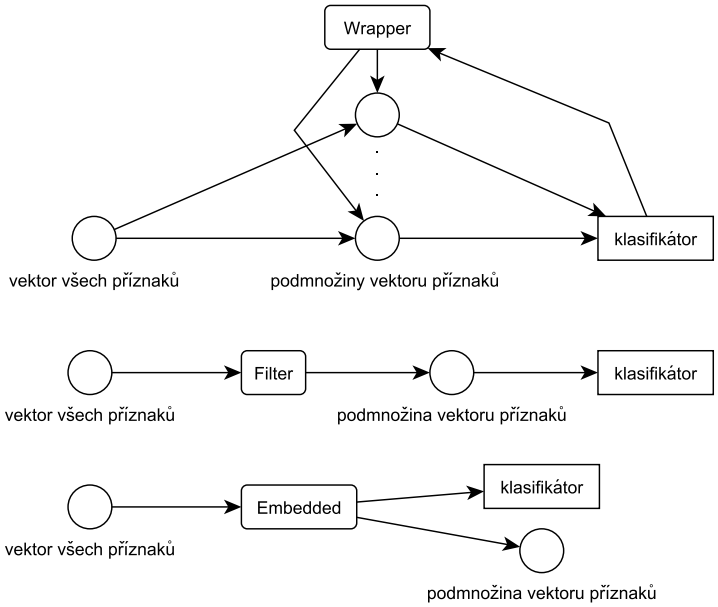
Průběh výběru příznaků v rámci algoritmů jednotlivých tříd ilustruje obrázek 3.1. Pro úlohy klasifikace textů se dle autorů příspěvků [44, 28, 38] typicky používají pro výběr příznaků níže popsané metody[2][3]. [23, 22, 15]

Frekvence výskytů v dokumentech

Tento algoritmus (angl. *document frequency*) počítá množství dokumentů, ve kterých se příznak vyskytuje. Redukce dimenze vektoru příznaků se provádí nastavením spodního prahu, který určuje minimální hodnotu výskytu, které musí příznak dosáhnout, aby byl považován

[2] V citovaných zdrojích autoři pracují v popisovaných metodách se slovy, resp. termy ve zdrojových dokumentech. V této práci jsou tyto pojmy nahrazeny termínem příznaky, neboť v úloze klasifikace textů jsou tyto ekvivalentní.

[3] Tato práce se pro rozsáhlost zabývá popisem pouze některých, autory uváděných, metod.



Obrázek 3.1: Obrázek inspirovaný výukovými materiály autorů Isabelle Guyon a André Elisseeff [22] ilustruje popořadě výběr příznaků při redukci dimenzí vstupního vektoru pro třídy algoritmů **wrappers**, **filters** a **embedded**.

z pohledu klasifikace za důležitý. Aplikací této metody se tak z vektoru odstraní ty příznaky, které nemají buďto žádnou informační hodnotu nebo neovlivňují přesnost výsledného modelu. Nastavením horního prahu je rovněž možné z příznaků odstranit typická stop slova jazyka. Hodnota pro jednotlivé příznaky je spočtena ze vzorce 3.5 uvedeného v příspěvku [28], kde m značí celkový počet dokumentů a A_i právě hodnocený příznak. Pro výpočet hodnoty není potřeba, aby byla vstupní data označena. [44, 38]

$$DF = \sum_{i=1}^{m} A_i \tag{3.5}$$

Vzájemná informace

Výpočet hodnoty vzájemné informace (angl. *mutual information*) obecně reprezentuje míru závislosti dvou veličin. V případě klasifikace se jedná o závislost mezi příznakem vektoru a zvolenou kategorií dokumentu. Hodnota závislosti příznaku f na kategorii c je vypočtena ze vzorce 3.6 uvedeném ve článku [38], kde A značí počet výskytů příznaku f v kategorii c, B značí počet výskytů f mimo c, C značí počet výskytů c bez f a M značí celkový počet dokumentů. Pro nesourodý příznak a kategorii tak metoda vrací hodnotu 0. Informační hodnota příznaku $MI(f)$ pro výsledný model je následně určena ze vztahu 3.7 uvedeném v příspěvku [44], kde m reprezentuje celkový počet kategorií. Yangová a Pedersen [44] uvádí slabinu této metody, kterou je neporovnatelnost příznaků se značně rozdílnou frekvencí výskytu z důvodu velké závislosti hodnoty $MI(f)$ na okrajové (angl. *marginal*) pravděpodobnosti. Díky této závislosti může nastat jev, kdy zřídka obsažené příznaky mají na základě výpočtu větší důležitost než ty časté. [44, 38]

$$MI(f,c) = \log\left(\frac{A \times M}{(A+B) \times (A+C)}\right) \tag{3.6}$$

$$MI(f) = \max_{i=1}^{m}\{MI(f,c_i)\} \tag{3.7}$$

Informační zisk

Tato metoda (angl. *information gain*) zjišťuje množství informace, resp. entropie $G(f)$ poskytované příznakem f při jeho výskytu a naopak absenci v dokumentu určené kategorie. Množství informace, jak uvádí autoři publikace [28], je definováno předpisem 3.8 s následující sémantikou [44, 28]:

- $P(c_i)$ je pravděpodobnost náležitosti dokumentu do kategorie c_i,
- $P(f)$ je pravděpodobnost výskytu příznaku f v dokumentu,
- $P(c_i|f)$ je pravděpodobnost jevu, že dokument patří do kategorie c_i, pokud obsahuje příznak f
- a podobně $P(c_i|\bar{f})$ pokud neobsahuje f.

$$G(f) = -\sum_{i=1}^{m} P(c_i)\log P(c_i) + P(f)\sum_{i=1}^{m} P(c_i|f)\log P(c_i|f) + P(\bar{f})\sum_{i=1}^{m} P(c_i|\bar{f})\log P(c_i|\bar{f}) \tag{3.8}$$

Chi-square

Chi-square vychází ze stejnojmenného pravděpodobnostního rozložení χ^2 a pracuje na principu výpočtu odchylky od jeho předpokládaného rozložení za předpokladu nezávislosti daného příznaku f dokumentu na kategorii c. Ohodnocení příznaku f se dle Yangové a Pedersena [44] provede výpočtem vzorce 3.9. Hodnota M uvádí celkový počet dokumentů, A je počet výskytů příznaku f v kategorii c, B je počet výskytů f mimo c, C je pak množství výskytů c bez f a proměnná D značí množství dokumentů, ve kterém se nevyskytuje ani příznak f, ani kategorie c.

$$\chi^2(f,c) = \frac{M \times (AD - CB)^2}{(A+C) \times (B+D) \times (A+B) \times (C+D)} \tag{3.9}$$

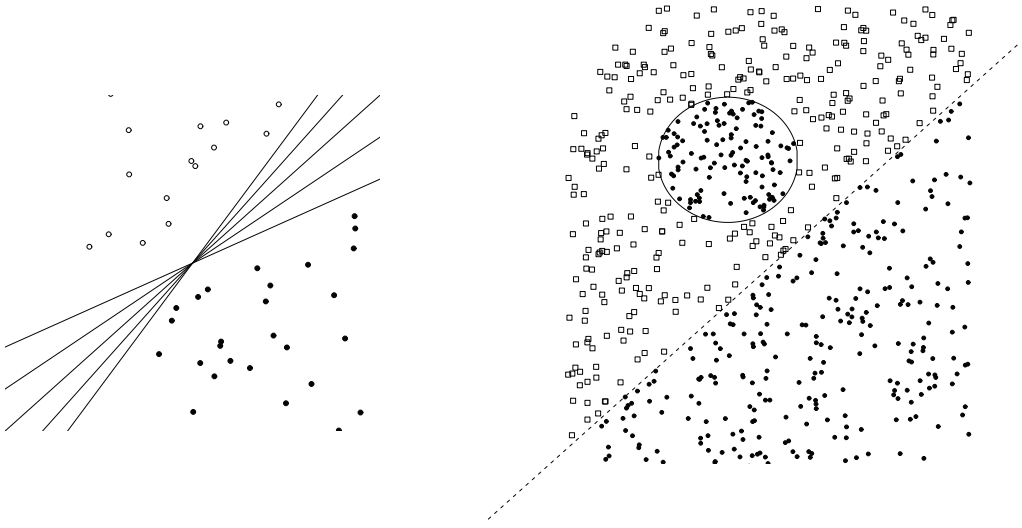
Výše uvedené metody výběru příznaků spadají do skupiny tříd *filters* a dosahují lineární časové složitosti. Výjimku tvoří metoda *Chi-square*, která spadá to kvadratické třídy složitosti. Ze závěrům uvedených v práci Yangové a Pedersena [44] dosahují nejlepších výsledků metody *Information Gain* a *Chi-square*. [44, 23].

3.3 Klasifikátory

Poté co je ze vstupních dokumentů vytvořena vhodná reprezentace, tzv. vektor příznaků, je možné použít ke kategorizaci dokumentů nějaký algoritmus z oblasti strojového učení. Takový algoritmus se obyčejně nazývá klasifikátor. Aby byl klasifikátor schopen provádět kategorizaci dat, je potřeba jej nejprve natrénovat. Při trénování hledá klasifikační algoritmus pravidlo h, takové, které z n-dimenzionálního prostoru příznaků přiřadí třídu, resp. třídy, do kterých analyzovaná data, resp. dokument náleží. Přesnost klasifikačního pravidla h určit třídu neznámých dat lze dle autorů Yangové a Joachimse [43] vyjádřit jako hodnotu rizika udávanou vztahem 3.10, kde funkce $L(h(x), y)$ vrací výši negativního dopadu při vyhodnocování příznaku x pravidlem $h(x)$ pro třídu y. Hodnota $P(x, y)$ je pak dané rozložení pravděpodobnosti pro příznak x a třídu y při klasifikaci dokumentu. Jednotlivé klasifikátory se pak liší právě ve funkci L odhadující riziko.

$$R(h) = \int L(h(x), y) dP(x, y) \tag{3.10}$$

Klasifikátory lze rozdělit na dvě skupiny – na lineární a nelineární. Rozdíl těchto skupin spočívá ve funkci, která tvoří jádro klasifikátoru. Na základě této funkce je rozdělován prostor příznaků nadrovinami. U lineárních klasifikátorů je to přímka, u nelineárních pak například křivka udaná polynomem. Rozklad prostoru příznaků nadrovinami je ilustrováno obrázkem 3.2. Problém klasifikace lze v tomto pojetí chápat jako hledání strany od nadroviny, do které klasifikovaná data náleží. [29, 43]



Obrázek 3.2: Obrázky převzaté z knihy Chistophera Manniga [29] ilustrují popořadě lineární a nelineární problém klasifikace. V prvním případě je prostor příznaků rozdělován nadrovinami určené lineární jádrovou funkcí klasifikátoru, tedy přímkami. V druhém případě se pak jedná o nelineární funkci. Zde je prostor separován kružnicí. Příznaky jsou v ilustraci reprezentovány tečkami a čtverečky, kdy intuitivně stejně značené příznaky dokumentu patří do jedné třídy.

Rovněž je možné klasifikátory rozdělit na generativní a diskriminativní. Skupina první, dle autorů práce [32], vytváří model pro příznak x a třídu y na základě pravděpodobnostního ohodnocení $p(x, y)$. Následný výběr y pro x pak závisí na hodnotě udané podmíněnou

pravděpodobností $p(x|y)$[4], kdy je vybrána třída y s největším pravděpodobnostním ohodnocením za použití Bayesova pravidla 3.11. Diskriminační klasifikátory naopak při formování modelu nehledí na pravděpodobnostní rozdělení $p(x, y)$ a následné ohodnocení $p(x|y)$ získávají přímo ze vstupních dat hledáním klasifikačního pravidla s nejnižší chybou, resp. rizikem. [43, 32]

$$p(y|x) = \frac{p(x|y)p(y)}{p(x)} \tag{3.11}$$

3.3.1 Rozhodovací stromy

Dle knihy [23] jsou rozhodovací stromy často vyhledávaným přístupem pro klasifikaci. Jedná se o hierarchický model budovaný nad prostorem příznaků. Trénovaní klasifikátoru probíhá tak, že nejprve je celá trénovací množina brána jako kořen stromu, a následně je tato množina rekurzivně dělena na disjunktní části reprezentující konkrétní třídu klasifikace. Dělení probíhá na základě logických pravidel určujících, kterým směrem se pak při průchodu stromem vydat. Výběr příznaků se odvíjí od právě budovaného podstromu, kde je vybírán ten příznak, který je pro danou část z hlediska informační hodnoty významný. Z globálního pohledu může být jeho důležitost zanedbatelná. Obecně se algoritmus rozhodovacího stromu skládá ze tří částí:

- Strategie budování stromu – nejčastěji se používá technika *hill climbing*,
- metody rozdělování uzlů – nejběžnějším kritériem je *purity gain*, resp. *impurity reduction*, předepsaným vztahem 3.12, kde pro rozdělení s je $I(s, N)$ ohodnocení *impurity*[5] uzlu N, N_i^s je i-tý potomek uzlu N, jenž vznikl po dělení s, a p_i je odhadovaná pravděpodobnost náležitosti k uzlu N_i^s v případě, že data spadají do třídy udané uzlem N
$$\Delta I(s, N) = I(N) - \sum_i p_i I(N_i^s) \tag{3.12}$$
- a způsobem, jakým dosáhnout obecnosti[6] modelu – například pomocí řezů/ořezávání (angl. *prunning*).

Mezi známé metody budování rozhodovacích stromů patří CART (*Classification and Regression Trees*), C4.5, SSV (*Separability of Split Value*) nebo FACT (*Fast algorithm for Classification Trees*). [23]

3.3.2 Neuronové sítě

Umělé neuronové sítě, používané v oblasti výpočetní techniky, jsou inspirovány funkcí lidského mozku. Inženýři se snaží o napodobení jeho schopností zejména v oblastech počítačového vidění, zpracování zvuku a schopnosti učení. Paralelně pracující výpočetní uzly v mozku se nazývají neurony propojené synapsemi. Umělé neuronové sítě tyto elementy nahrazují jednotkami zvanými perceptrony mající vstup a výstup. Výstupní hodnota perceptronu o je dle Mitchella [31] dána lineární kombinací vektoru vstupů x_i a příslušných vah w_i vztahem 3.13. Perceptron v tomto pojetí definuje nadrovinu určenou $\vec{w} \cdot \vec{v} = 0$, která

[4] Udává hodnotu podmíněné pravděpodobnosti jevu x za předpokladu, že nastal jev y

[5] Uzel je brán jako čistý (angl. *pure*), pokud jemu náležející vzorky dat patří do jedné a té samé třídy.

[6] Obecností je myšlena schopnost klasifikovat nová data s předpokládanou přesností udanou modelem.

dělí lineárně separabilní vstupní prostor (např. prostor příznaků) na dvě části. Jedná se v podstatě o binární klasifikátor, který na základě prahu w_0, poskytuje výsledek 1 nebo -1.

$$o = \begin{cases} 1 & \text{pro} \sum_{i=0}^{n} w_i x_i > 0 \\ -1 & \text{jindy} \end{cases} \tag{3.13}$$

Perceptrony jsou schopny řešit pouze lineární problémy. Pro nelineární problémy se používá vícevrstvá neuronová síť. Perceptron je zde nahrazen sigmoidální jednotkou, která stejně jako perceptron počítá lineární kombinaci vektoru vstupů a vah, ale výsledkem je spojitá sigmoidální funkce $o = \sigma(\vec{w} \cdot \vec{v})$, kde výpočet σ je definován vzorcem 3.14. Hodnoty vah pro vstupy jsou počítány algoritmem *Backpropagation* [31]. Neuronové sítě tedy mohou být zástupci jak lineárního tak nelineárního klasifikátoru. [31, 13]

$$\sigma(y) = \frac{1}{1 + e^{-}y} \tag{3.14}$$

3.3.3 Algoritmus k-nejbližších sousedů

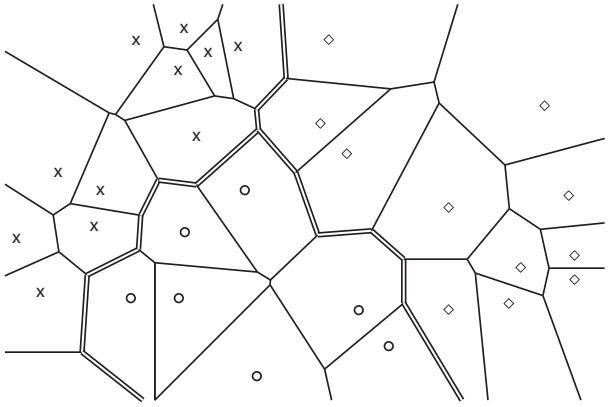
Zástupcem čistě nelineárních klasifikátorů je algoritmus k-nejbližších sousedů (kNN). Při klasifikaci jsou instance dat (například klasifikované dokumenty) reprezentovány body v n-dimenzionálním prostoru. Nechť je instance klasifikovaných dat x popsána vektorem obsahujícím n příznaků $< a_1(x), a_2(x), ..., a_n(x) >$, kde $a_r(x)$ reprezentuje hodnotu r-tého příznaku. Pak vzdálenost dvou instancí x_i a x_j v prostoru je dle Mitchella [31] počítána pomocí Euklidovské vzdálenosti $d(x_i, x_j)$ definované vzorcem 3.15. Instance x_i reprezentuje klasifikovaná data, x_j pak instanci dat, na kterých byl klasifikátor natrénován.

$$d(x_i, x_j) = \sqrt{\sum_{r=1}^{n} (a_i(x_i)) - (a_r(x_j))} \tag{3.15}$$

N-dimenzionální prostor bodů (příznaků klasifikovaných dokumentů) je rozdělován pomocí Voronoiovy teselace na buňky ohraničené konvexním polygonem. V každé takto vzniklé buňce jsou obsaženy body, které jsou k-nejbližší sousedé dané instance dat. Na obrázku 3.3 je takové rozdělení znázorněno pro $k = 1$. [31, 29]

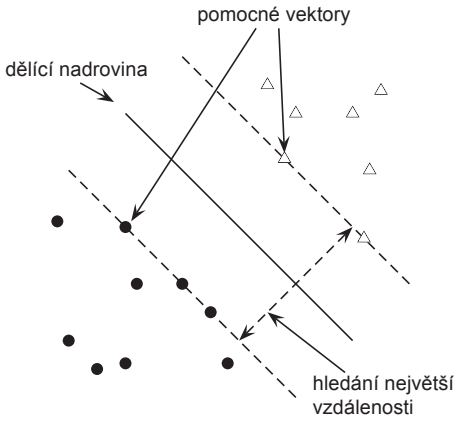
3.3.4 Support Vector Machine

Klasifikátor *Support Vector Machine*, zkráceně označovaný jako SVM, používá k rozdělování prostoru příznaků podpůrné vektory (angl. *support vectors*). Tyto vektory sestávají z vybrané části příznaků. Ostatní jsou zanedbány. Podpůrné vektory následně slouží k proložení nadroviny v prostoru příznaků tak, aby vzdálenost vektorů byla od dělící nadroviny co největší. Čím větší vzdálenosti je dosaženo, tím je klasifikace přesnější. V případě, že prostor není možné rozdělit nadrovinou s požadovanou vzdáleností od pomocných vektorů, je možné zanést chybu zavedením dodatečných proměnných (angl. *slack variables*), které umožní, že část příznaků z jedné třídy přesahuje přes nadrovinu do třídy druhé. Toto řešení se v angličtině označuje jako hledání *soft margin*. SVM je v základní podobě binární klasifikátor používající při hledání nadroviny lineární jádrovou funkci. Pro lineárně neřešitelné problémy lze nahradit jádrovou funkci klasifikátoru jinou, nelineární funkcí, například polynomiální nebo sigmoidální. Hledání nadroviny SVM v základní variantě je ilustrováno obrázkem 3.4. Dle Joachimse [25] jsou SVM velmi vhodné pro klasifikaci textů, neboť dokáží s úspěchem zvládat vysoký počet dimenzí vektoru příznaků s velkým počtem relevantní



Obrázek 3.3: Ilustrace převzatá z knihy Christophera Manninga [29] zobrazuje rozdělení prostoru Voronoiovu teselací pro 3 třídy reprezentované symboly X, kruhu a diamantu. Dvojití čára pak vyznačuje hranice rozhodování náležitosti do vymezených tříd pro algoritmus 1NN.

příznaků. Odpadá tak potřeba použití metod pro výběr příznaků a rovněž je SVM díky zabudované ochraně odolný vůči přeučení. [25, 29]



Obrázek 3.4: Obrázek převzatý z knihy Christophera Manninga [29] znázorňuje rozdělení prostoru příznaků dvou tříd reprezentovaných kolečky a trojúhelníky nadrovinou určenou klasifikátorem SVM s lineární jádrovou funkcí.

3.3.5 Naive Bayes

Dle knihy Toma Mitchella [31] je klasifikátor Naive Bayes pro některé problémy stejně výkonný jako neuronové sítě nebo rozhodovací stromy, což z něj dělá často používaný klasifikátor. Algoritmus používá statistický model, při jehož budování předpokládá, že vstupní data jsou nezávislá. Díky tomuto předpokladu bývá klasifikátor dle autorů příspěvku [30] nazýván „naivním“, neboť v reálné aplikaci tato skutečnost nenastává. Nicméně algoritmus i tak poskytuje velmi dobré výsledky. Dle stejného článku [30] jsou v klasifikátoru používány dva statistické modely. V prvním z nich je klasifikovaný dokument reprezentován

vektorem atributů udávajícím, zda se slovo vyskytuje v dokumentu či nikoliv (nazývaný též Bernoulliho s více proměnnými). Naopak druhý model zachycuje dokument jako množinu obsažených slov s počtem jejich výskytů. Tento model je označován jako multinomiální. Výsledek klasifikace je dle Mitchella [31] určen pomocí vzorce 3.16, který na základě Bayesova pravidla 3.11 a zmíněného předpokladu, že vstupní data jsou vzájemně nezávislá, přidělí klasifikovanému dokumentu třídu v_j z konečné množiny V všech možných tříd. Na základě tohoto předpokladu je možné substituovat $P(a_1, a_2, ..., a_n|v_j)$ za $\prod_i P(a_i|v_j)$, kde $< a_1, a_2, ..., a_n >$ je n-tice popisující právě klasifikovaný dokument.

$$v_{NB} = \arg\max_{v_j \in V} P(v_j) \prod_i P(a_i|v_j) \tag{3.16}$$

Klasifikátor Naive Bayes se na rozdíl od ostatních, výše uvedených diskriminativních algoritmů, řadí mezi generativní algoritmy. [31, 30]

3.4 Vyhodnocení přesnosti

Posledním krokem při vytváření klasifikátorů je ověření přesnosti modelu natrénovaného klasifikátoru. Hlavní otázkou testování je, s jakou úspěšností je klasifikátor schopen analyzovat nová, resp. neznámá data. Validace také odhaluje, zda model netrpí přeučením. Hodnocení klasifikátoru je možné provést dále uvedenými metodami. [23]

3.4.1 Křížová validace

V rámci metody křížové validace jsou trénovací data rozdělena do několika podmnožin, kde část je použita pro natrénování klasifikátoru a zbylá část k testování. Samotná metoda má několik variant, z nichž je dle G'erarda Dreyfuse a Isabelle Guyon [23] preferovaná varianta *D-fold*. Ta rozděluje množinu testovacích dat na D podmnožin. $D-1$ podmnožin je použito k natrénování modelu a na poslední, resp. zbylé podmnožině je provedena validace. Tento postup se následně opakuje D-krát s opětovných rozdělením původní testovací množiny tak, že každá podmnožina je pro validaci použita právě jednou. Výsledek testování je pak sumou jednotlivých iterací. Speciální případem *D-fold* je varianta *leave-one-out*, kdy D je rovno počtu testovacích vzorků. [23]

3.4.2 Bootstrap aggregating

Jádro tohoto přístupu, někdy též zvaného *bagging*, spočívá v náhodném rozdělení dat trénovací množiny na m podmnožin o stejné velikosti jako původní trénovací množina, které jsou následně použity k naučení m duplikátů klasifikátoru. Rozdělení dat na podmnožiny probíhá rekurzivně, a tudíž se některé sady dat mohou a naopak vůbec nemusí ve vzniklých podmnožinách objevit, resp. opakovat. Pro potřeby klasifikace je pro výsledek testování použit princip většinového hlasování, kdy je za směrodatný výsledek uvažován ten, který se objevuje u nadpolovičního počtu klasifikátorů. [23]

3.4.3 Reprezentace výsledků

Přesnost binárního klasifikátoru je možné reprezentovat kontingenční tabulkou se dvěma sloupci a řádky obsahující postupně počty pozitivně pozitivních (TP), negativně pozitivních (FP), pozitivně negativních (TN) a negativně negativních (FN) případů klasifikace pro dané

dvě kategorie. Pro N[7] dokumentů použitých při trénování jsou dle Yangové a Jochimse [43] pro hodnocení kvality binárních klasifikátorů využívány například tyto metriky [43]:

- Přesnost určení kategorie (angl. *precision*) počítána jako $\frac{TP}{TP+FP}$,
- senzitivita (angl. *recall*) definována jako $\frac{TP}{TP+FN}$, obdobně specificita $\frac{TN}{TN+FP}$
- celková přesnost modelu (angl. *accuracy*) vypočtena pomocí $\frac{TP+TN}{N}$,
- F_1 skóre hodnotící přesnost klasifikátoru ze vztahu $\frac{2 \cdot recall, \cdot precision}{recall+precision}$ a
- chybovost počítaná ze vzorce $\frac{FP+FN}{N}$.

[7] Hodnotu N je možné též reprezentovat jako $TP + FP + TN + FN$

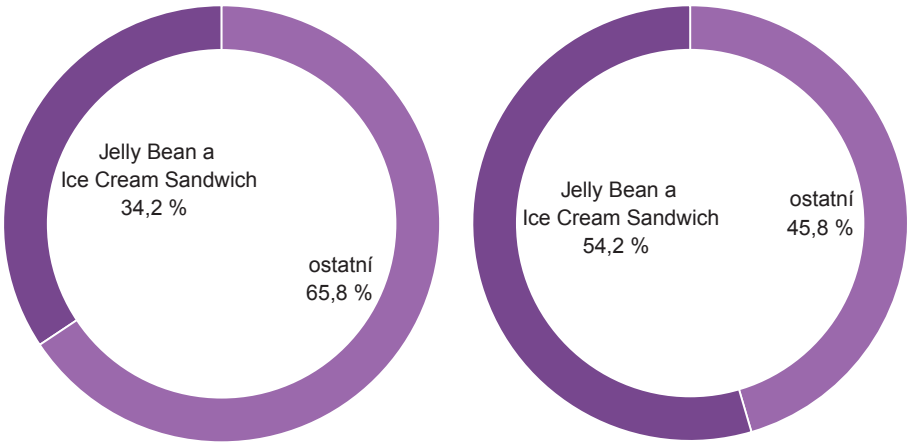
Kapitola 4

Použité technologie, návrhové vzory a algoritmy

Tato kapitola se zaměřuje na popis specifik zvolených technologií, aplikačních rámců a použitých knihoven třetích stran. Dále jsou rozebrány využívané návrhové vzory a algoritmické přístupy.

4.1 Klientská část

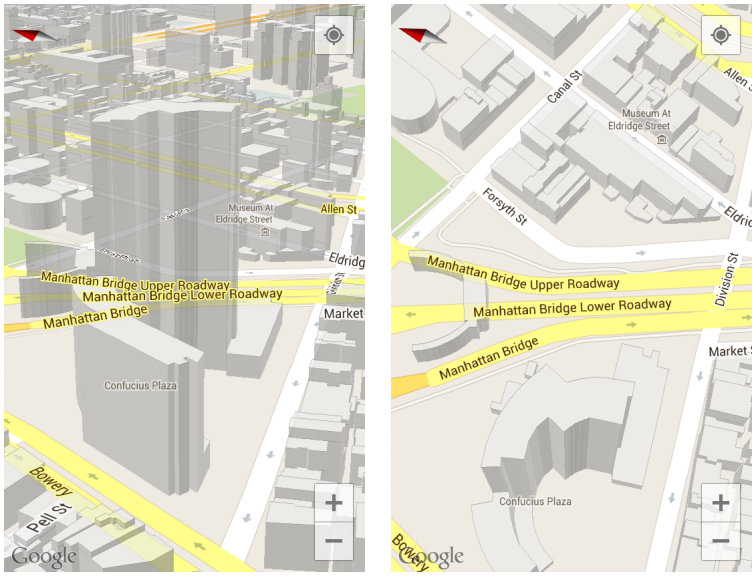
K implementaci klientské aplikace se využívá Android SDK ve verzi 4.2 (Jelly Bean). Z něj používané části jsou zpětně kompatibilní pouze do verze 4.0 (Ice Cream Sandwich). V návaznosti na analýzu problému je možné tyto dvě verze Androidu považovat za dominantní, neboť společně pokrývají 54,3 % distribuce. Pro výběr těchto verzí hovoří i trend rozložení verzí mezi distribuovanými zařízeními za první kvartál roku 2013 je ilustrován grafem na obrázku 4.1.



Obrázek 4.1: Trend distribuce verzí Androidu vycházející ze statistik [20]. Na vyobrazených grafech je možné shlédnout 20% nárůst distribuce verzí Jelly Bean a Ice Cream Sandwich v rozmezí měsíců prosinec 2012 až duben 2013.

K zobrazování mapových podkladů je zvoleno Google Map API ve druhé verzi, které je součástí knihovny `Google Play Services`. Použitá verze nabízí zabudované zobrazování

polohy uživatele. V předchozí verzi bylo nutné si pozici zjišťovat manuálně a dodatečně ji v mapě vykreslovat. Dalším zlepšením je automatické cachování stáhnutých mapových materiálů. Druhá verze API rovněž zjednodušila práci při vkládání vlastních značek a grafických objektů do mapy. K velkému posunu směrem k funkcionalitě webových map od společnosti Google došlo i v samotném vykreslování map, které je nyní možné plně hardwarově akcelerovat, a mapu tak například dle libosti rotovat či zobrazit její 3D pohled viz obrázek 4.2.



Obrázek 4.2: Ukázka nových funkcí (3D pohled a rotace mapy) ve druhé verzi Google Map API pro mobilní telefony.

Pomocným prostředkem pro komunikaci se serverem pomocí REST rozhraní je knihovna `spring-android`, konkrétně její třída `RestTemplate`, která zcela zapouzdřuje odesílání, resp. příjem zpráv ze serveru. K serializaci, resp. deserializaci objektů z těla zpráv se využívá knihovna `Jackson`, která atributy objektu a jejich hodnoty převede do, resp. z formátu JSON.

4.2 Serverová část

Serverová část je postavena na Spring frameworku, konkrétně jeho webovém modulu `Spring MVC`. Jak již jeho název napovídá, při vývoji pomocí něj se používá návrhový vzor MVC (*Model–View–Controller*). V adaptaci tohoto návrhového vzoru se ve skutečnosti plnohodnotně využívá pouze vrstev *Controller* a *Model*. Z pohledu vrstvy *View* je zpět odesílána odpověď v textové podobě, a tudíž nejsou používány žádné JSP (*JavaServer Pages*) ani dodatečné servlety. K nastavení mapování požadavků a jednotlivých parametrů v URL adrese na obslužné funkce a jejich lokální proměnné je využito pouze anotačního přístupu. Anotace se používají také k definování jednotlivých komponent, například beanů, aspektů, služeb a repositářů. Nastavení pomocí XML souboru je uplatněno v případě konfigurace Spring a Hibernate frameworku. Konkrétně je provedeno nastavení:

- hlavního obslužného servletu,

- směrování dotazů na tento servlet,
- kontextu aplikace,
- dialekt databáze,
- transakční zpracování,
- objektově-relačního mapování – definice anotovaných tříd a
- strategie pojmenování – automatizované použití prefixů tabulek v databázi.

Při implementaci serveru je používáno aspektové programování – AOP(*aspect–oriented programming*). Konkrétně je pomocí něj zcela přesunuta ladící a logovací logika mimo funkce vrstvy *Controller*. Definování jednotlivých aspektů je provedeno rovněž použitím anotací, které specifikují:

- ve které fázi provádění obslužné funkce se má aspekt vykonat – před, po nebo během,
- název funkce, na kterou je aspekt navázán a
- její parametry.

K aspektovému programování je použita knihovna `AspectJ`. Samotné zaznamenávání požadavků a odpovědí serveru se provádí pomocí knihovny `log4j` do lokálního souboru.

Pro ukládání objektů do databáze se používá ORM framework Hibernate. Jelikož Hibernate sám o sobě nepodporuje prostorová rozšíření databází, je současně s ním použita nástavba Hibernate Spatial. Jedná se o knihovnu, která rozšiřuje Hibernate o definici prostorových typů, kritérií a další pomocné funkce pro ukládání a získávání prostorových objektů.

4.3 Komunikační protokol

Komunikace mezi klientskou a serverovou částí je řešena pomocí HTTP protokolu. Jeho šifrovaná varianta HTTPS nebyla zvolena, neboť z pohledu posílaných dat se nejedná o důvěrné informace a tudíž není potřeba realizovat bezpečný kanál. Nicméně je nutné řídit přístup k serveru pomocí API (*Application Programming Interface*) tak, aby k němu mohli přistupovat pouze klientské aplikace. Nabízí se možnost autentifikace pomocí samotného protokolu HTTP. Ten nicméně neřeší možnost odposlechu komunikace a odhalení přihlašovacích údajů či použití *replay* útoků. Proto byl zvolen jiný přístup na principu podepisování jednotlivých zpráv. Podpis se vytváří pomocí hashovací funkce `HmacSHA256`, jejímž vstupem je hashovaná zpráva a privátní klíč. Ochranu proti *replay* útokům zajišťuje mechanizmus na bázi časových razítek.

4.4 Analýza sentimentu a kategorizace událostí

K analýze sentimentu a kategorizaci událostí byl zvolen klasifikátor SVM se sigmoidální jádrovou funkcí. Filtrování a výběr příznaků je prováděn následující posloupností operací:

- převod velkých písmen slov na jejich malou variantu,
- rozdělení slovních spojení na tokeny přes oddělovač netisknutelných znaků (mezery, tabulátory a konce řádků),

- odfiltrování stop slov jazyka (předložky, spojky a jiné),
- aplikace metody *stemming* pomocí Porterova algoritmu a
- finální odstranění tokenů, jejichž délka je menší jak 3 znaky.

K ohodnocení příznaků je použita metoda TF-IDF. Pro validaci je zvolena křížová validace ve variantě *D-fold* s parametrem $D = 10$. K vytvoření klasifikátoru pomocí popsaných metod a algoritmů byla použita aplikace Rapidminer. Postup vytvoření, resp. natrénování klasifikátoru je uveden v kapitole 5 věnované návrhu aplikace.

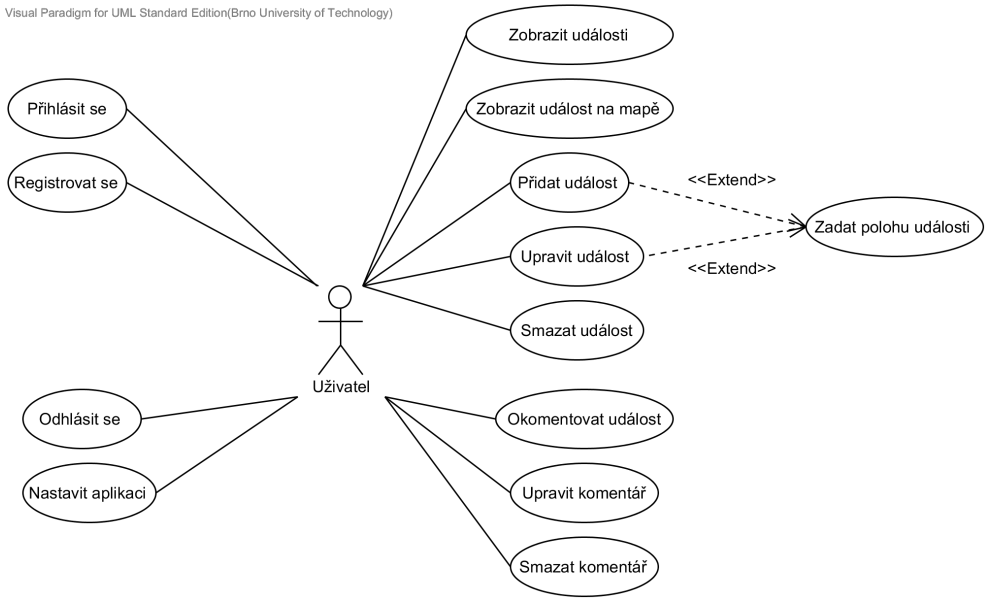
Kapitola 5

Návrh aplikace

V této části práce jsou nejprve rozebrány případy použití aplikace, které vycházejí z neformální specifikace popsané v kapitole 2. Dále je zde popsán návrh uživatelského rozhraní klienta a architektury celé aplikace. Společně s databází je komentován i návrh realizace komunikačního protokolu. V závěru kapitoly je popsán způsob natrénování klasifikátorů.

5.1 Funkcionalita aplikace

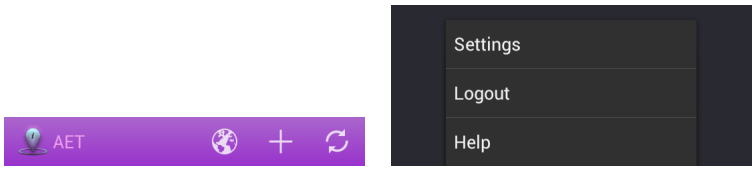
V kapitole 2, zabývající se analýzou problému, byla uvedena neformální specifikace problému (viz 2.2). Následující diagram 5.1 znázorňuje jednotlivé případy užití aplikace z pohledu jejího uživatele, které ze specifikace vyplývají. Případy užití *Odhlásit se*, *Nastavit aplikaci* a všechny na pravé straně diagramu může uživatel provést pouze po předchozím přihlášení, resp. registraci. Pro provedení případů užití *Upravit*, resp. *Smazat* událost a komentář musí navíc uživatel být autorem dané položky. Ostatní případy použití může provádět bez ohledu na vlastnictví.



Obrázek 5.1: Diagram případů užití aplikace.

5.2 Uživatelské rozhraní

Návrh uživatelského rozhraní probíhal v souladu s doporučenými postupy, jenž jsou uvedeny v manuálu [19] pro návrháře uživatelských rozhraní pro platformu Android. Nabídka akcí v aplikaci je řešena pomocí komponenty `ActionBar`. Ta nabízí mimo titulek aktuálně zobrazované obrazovky také hlavní ovládací prvky aplikace, jako jsou například tlačítka pro přidání nové události nebo obnovení obsahu. Jednotlivé ovládací prvky se zobrazují v pravé části nabídky dle volného místa na displeji. Pokud není na displeji dostatek místa, položky se přesunou do vyjíždějícího menu na spodní straně displeje, které je ve výchozím stavu skryté. Vzhled hlavní nabídky a rozložení obsažených ovládacích prvků je zachycen na obrázku 5.2.



Obrázek 5.2: Hlavní nabídka aplikace řešená pomocí komponenty Androidu `ActionBar`.

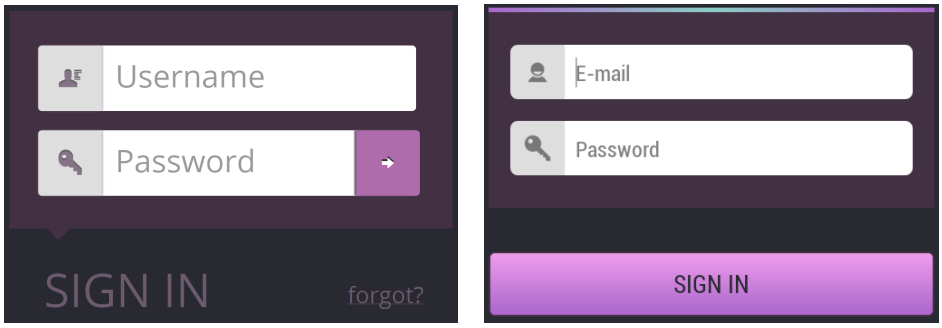
Zobrazení událostí bylo navrženo ve dvojí podobě. První z nich je zobrazení v seznamu. Každá položka seznamu obsahuje základní informace o události. Jedná se zejména o titulek události, její zevrubný popis, kategorii, datum a čas vzniku a vzdálenost od aktuální polohy uživatele. Události vytvořené přihlášeným uživatelem jsou patřičně označeny v pravé horní části položky seznamu. Druhou variantou je zobrazení událostí na mapě. V tomto případě jsou na mapě vyznačeny body, kde se události nacházejí. Po stisknutí daného bodu se zobrazí informační oblast s hlavními údaji události. Vlastnictví události je v této variantě zobrazení řešeno pomocí symbolu panáčka v levém horním rohu. V obou případech zobrazení lze přejít na přehled detailních informací o události. V tomto pohledu je zobrazen navíc úplný popis události a uživatelské komentáře. Přidávání nových komentářů je realizováno pomocí dialogového okna se vstupním polem.

Prostředí celé aplikace je barevně laděno do tmavých odstínů. Návrh se snaží o zachování kontrastních poměrů mezi barvou pozadí a textu kvůli zajištění snadné čitelnosti. Samotná volba barevné palety aplikace vychází z návrhu přihlašovacího, resp. registračního formuláře, jenž se odráží od implementace publikované na stránce [3]. Porovnání vzhledu předlohy přihlašovacího formuláře s vlastní realizací ilustruje obrázek 5.3.

V návrhu aplikace jsou ve snaze zachovat přirozený ráz, z pohledu trendů ovládání Android aplikací, používány výhradně výchozí prvky uživatelského rozhraní. K jejich modifikaci dochází pouze z pohledu barevného provedení. Samotná funkcionalita se nemění v žádném z případů.

5.3 Architektura aplikace

Návrh architektury aplikace je zachycen pomocí diagramů tříd A.5 a A.6, ve kterých pro přehlednost nejsou, až na drobné výjimky, uvedeny třídní proměnné a metody. Diagramy se spíše zaměřují na závislosti jednotlivých tříd, balíčků a vrstev tak, aby poskytly ucelený náhled na způsob, jakým zajišťují funkcionalitu popsanou diagramem případů užití 5.1. V klientské i serverové části je použit návrhový vzor MVC, který je zvláště v pohledové



Obrázek 5.3: Porovnání předlohy (vlevo) přihlašovacího formuláře s vlastním řešením (vpravo).

vrstvě *View* upraven dle specifik použitého aplikačního rámce a potřeb zvolených postupů, viz 2).

O interakci s uživatelem se v klientské části starají třídy v balíku `CaseControllers`. Výchozí z nich je třída `MainActivity`, která slouží jako rozcestí. Pokud není přihlášen žádný uživatel, obstará přesměrování na třídu `LoginActivity`. V opačném případě, pokud je uživatel přihlášen, dle volitelného nastavení zobrazí uživateli buď seznam, nebo mapu s událostmi. Zobrazení událostí je zajišťováno potomky třídy `EventActivity`. Detail zvolených událostí má pak na starost třída `EventDetailActivity`. Nastavení aplikace zajišťuje třída `SettingsActivity`. Třída `HelpActivity` poskytuje uživatelskou nápovědu. Zajištění komunikace se serverovou částí je v režii modelové vrstvy a její průběh je popsán v následující části 5.5.

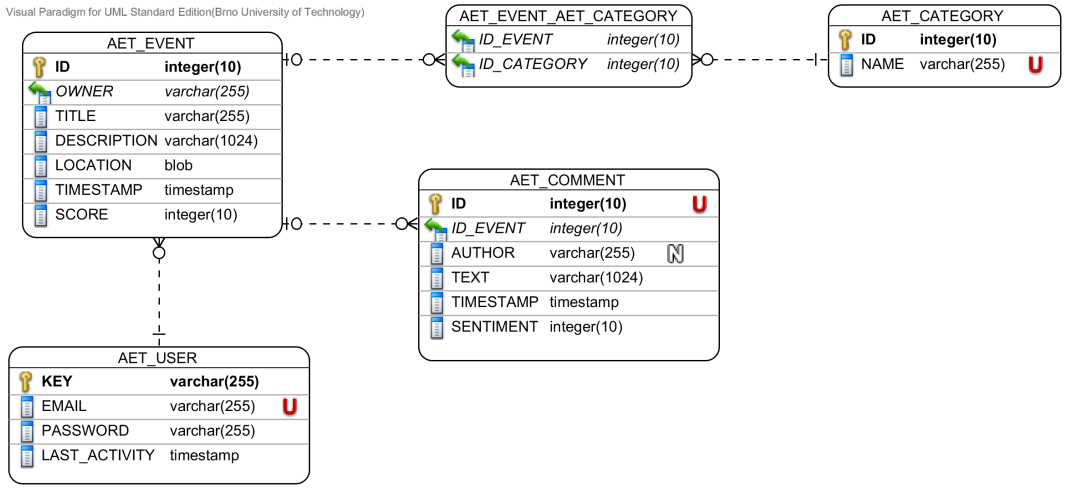
Příchozí požadavky na server a odpovědi na ně jsou zpracovávány třídami z balíčku `FrontControllers`. Aspekty obsažené v balíčku `Aspects` zajišťují zaznamenávání komunikace s klientkou částí. Persistence objektů v databázi je řešená v rámci modelové vrstvy. Balíček `Service` zapouzdřuje byznys logiku aplikace, jejíž součástí je i klasifikace událostí. Přímý přístup k databázové vrstvě je řešen v rámci balíčku `DAO`. Jeho jednotlivé třídy poskytují abstrakci nad mapováním objektů tříd z balíčku `Entity`. Samotné mapování pak zajišťuje framework Hibernate, který pro práci využívá datový zdroj ze třídy `SessionFactory`.

5.4 Databáze

Struktura databáze je reprezentována E-R diagramem na obrázku 5.4. Na jednotlivé tabulky jsou na straně serveru mapovány používané doménové entity.

5.5 Komunikace

Průběh komunikace mezi klientskou a serverovou částí z pohledu interakce jednotlivých tříd je zachycen v diagramech sekvence A.1 a A.2 popisujících aktualizaci, resp. stáhnutí událostí ze serveru. Diagramy A.3 a A.4 pak ilustrují případ, kdy uživatel vytváří novou událost. Komunikace při práci s komentáři k jednotlivým událostem a přihlašování či registrace uživatele probíhá v obdobné režii, kdy se mění pouze použité třídy, ale princip zůstává stejný. Zprávy zasílané z klienta na server jsou opatřeny autentizačními údaji,



Obrázek 5.4: E-R diagram realizované databáze

které do zprávy na straně klienta přidává třída `RequestBuilder` a na straně serveru ověřuje třída `RequestValidator`. Všechny dotazy na server je možné charakterizovat následujícím schématem:

```
http://<adresa serveru>:[port]/<cesta k aplikaci>/<skupina dotazů>/<akce>/
[volitelné parametry]/<klíč uživatele>/<časové razítko>/<autentizační hash>
```

Komunikace na straně klienta probíhá asynchronně, aby nebylo blokováno vlákno uživatelského rozhraní. Tuto funkcionalitu nabízejí třídy z balíčku `Loaders`, které využívají asynchronního mechanizmu pro načítaní dat z SDK Androidu. O zasílání požadavků na server se starají třídy implementující rozhraní z balíčku `Service`. Jednotlivé třídy v sobě zapouzdřují práci s třídou `RestTemplate`.

5.6 Trénování klasifikátorů

K natrénování klasifikátorů pro analýzu sentimentu a kategorizaci událostí bylo potřeba nejprve vytvořit množinu testovacích dat, což zahrnovalo především sběr vhodných textů pro tyto účely. Průběh sběru testovacích vzorků pro oba řešené problémy je popsán v následujících dvou podkapitolách. Získaná množina trénovacích dat, resp. dokumentů byla posléze použita jako vstup procesu vytvoření klasifikátoru v aplikaci Rapidminer. Pro zpracování trénovacích dat je použit blok `Process Documents From Files`. Jeho funkcí je filtrace dokumentů za použití metod uvedených v kapitole 4, například *stemming*. Výsledkem zpracování je vektor příznaků (v aplikaci Rapidminer nazýván jako *worldlist*). Z vektoru příznaků jsou blokem `Select attributes` vybrány ty příznaky, které mají přiřazenu nějakou hodnotu. Vybrané příznaky putují do bloku `Validation`, kde je vytvářen model klasifikátoru. Vytváření modelu podléhá algoritmu obsaženému v jádře SVM. Závěrem je nad vytvořeným modelem proveden, použitím bloků `Apply Model` a `Performance`, výpočet přesnosti pomocí křížové validace. Přesnost tímto způsobem vytvořených a natrénovaných klasifikátorů je reprezentována patřičnými tabulkami B.1, B.2, B.3, B.4, B.5, B.6 a B.7 v příloze B. Oba výsledné produkty (*wordlist* a model) jsou uloženy do lokálního repositáře, ze kterého jsou dále využívány při klasifikaci nových dat. Jejich použitím se zabývá část kapitoly 6.

5.6.1 Analýza sentimentu

Vyvíjená aplikace analyzuje komentáře jednotlivých událostí a odhaduje dle textu jejich pozitivní nebo negativní zabarvení. Tento odhad je následně použit k označení relativních a naopak irelevantních událostí. Za relevantní jsou považovány ty události, které uživatelé hodnotí pozitivně zabarvenými komentáři. Opačně pro irelevantní události. Na základě této úvahy byly jako testovací vzorky dat vybrány pozitivní a negativní komentáře, resp. hodnocení uživatelů ze služeb:

- Google Play – internetový obchod s aplikacemi pro mobilní zařízení s operačním systémem Android. Zde byla vybrána nejprodávanější aplikace a za pozitivní vzorky dat byla považována hodnocení uživatelů s 5 hvězdičkami, za negativní pak ta s 1 hvězdičkou. Počtem hvězdiček uživatelé vyjadřují spokojenost s aplikací a lze tedy předpokládat, že spokojený uživatel napsal pozitivní text hodnocení.
- Amazon – internetový obchod se zbožím různých kategorií, především knihami. V tomto obchodě byla jako referenční prvek zvolena nejlépe hodnocena kniha. Z těchto hodnocení bylo za pozitivní opět považováno to s 5 hvězdičkami a s pouze 1 hvězdičkou za negativní.
- eBay – internetový obchod, resp. aukce se zbožím z mnoha kategorií. Zdrojem dat zde byla hodnocení uživatelů k nejprodávanějšímu produktu v kategorii s elektronikou. Výběr pozitivních hodnocení se řídil stejnými pravidly jako v předchozích dvou případech.
- TripAdvisor – služba nabízející návštěvníkům možnost naplánování dovolených a výletů. Na stránkách této služby byly vybrány pozitivní a negativní hodnocení k nejoblíbenějším hotelům, atrakcím a restauracím v New Yorku. Za pozitivní bylo považováno hodnocení s označením *excellent*, za negativní pak to s příznakem *terrible*.

Pro každou z obou hodnot sentimentu bylo posbíráno 120 vzorků.

5.6.2 Kategorizace událostí

Mějme kategorii *fire* reprezentující výskyt požáru a s ním souvisejících událostí. Pomocí slovníku *Thesaurus* byla vyhledána nejčastější synonyma a příbuzná slova zkoumanému *fire*. Výskyt jednotlivých výsledků hledání byl posléze testován v titulcích zpráv a článků poskytovaných Google News API, které svou sémantikou spadají do právě zvolené kategorie *fire*. Aplikací tohoto postupu získáme první část trénovacích dat. Druhou část si zajistíme použitím toho postupu i pro ostatní kategorie. Nyní lze natrénovat jeden binární klasifikátor, kde první část dat je použita k definování pozitivního ohodnocení klasifikátorem a druhá k negativnímu. Další binární klasifikátory pro kategorizaci událostí vzniknou obměnou vybrané kategorie za další možnou s využitím stejného přístupu. Vytvářená aplikace je schopna klasifikace do následujících kategorií s uvedenou sémantikou:

- *crash* – značí dopravní nehodu,
- *concert* – reprezentuje kulturní události, převážně hudebního charakteru,
- *fire* – symbolizuje výskyt požáru a s ním související události (například výjezd hasičů),
- *inspector* – značí kontrolu revizorů v hromadné dopravě,

- *party* – zapouzdřuje různé formy oslav (například narozeniny nebo výročí) a
- *sport* – zabaluje sportovní události (například fotbalový zápas nebo tenisové utkání).

Při zjišťování kategorií je provedena klasifikace konkrétní události každým z vytvořených binárních klasifikátorů. Tento přístup kategorizace binárními klasifikátory byl zvolen z toho důvodu, že kategorie sami o sobě nejsou disjunktní (například může vypuknout požár na basketbalovém utkání). Jedná se tedy o tzv. *multi-label* problém, viz kapitola 3. Pro jednotlivé kategorie byl nasbírán, v závislosti na počtu synonymních a příbuzných slov, následující počet pozitivních testovacích vzorků: *crash* 310, *concert* 403, *fire* 131, *inspector* 39, *party* 161 a *sport* 226.

Kapitola 6

Implementace

V této kapitole je shrnuta implementace vyvíjené aplikace. První část kapitoly se soustředí na popis klientské části aplikace, která je vyvíjena na platformě Android. Mimo vysvětlení funkcionality jednotlivých tříd jsou zde uvedena i specifika související s implementační platformou. Druhá část kapitoly je věnována objasnění implementace serverové části. Ta je programována pod aplikačním rámcem Spring MVC. I zde jsou komentovány implementované třídy a jsou také uvedeny hlavní aspekty vývoje spojené s tímto frameworkem. V závěru kapitoly je popsána knihovna, kterou obě části užívají k zabezpečení komunikace.

6.1 Klientská aplikace

Popis implementace klientské části aplikace je nejprve zaměřen na vysvětlení funkcionality drobných celků, které ale formují základní stavební kameny celé aplikace. Poté je přistoupeno k popisu z pohledu uživatele. Při implementaci byly požity výhradně nové přístupy z Androidu 4.0 – 4.2, které do jisté míry znemožňují zpětnou kompatibilitu se staršími verzemi. Nicméně při implementaci byly použity postupy doporučované oficiální dokumentací Androidu [21].

6.1.1 Základní entity

V aplikaci je používáno několik tříd reprezentující základní používané entity:

- Přihlášený uživatel je realizován třídou `User`, které je implementována jako *singleton*. Její instance je tak dostupná v kterémkoli kontextu aplikace, což je využíváno například při kontrole vlastnictví události nebo komentáře. Pro zachování instance přihlášeného uživatele je třída `User` serializovaná do souboru v interním úložišti aplikace. Po spuštění aplikace je kontrolován výskyt tohoto souboru a případná instance uživatele je z něj obnovena.
- Třídy `Event`, `Category` a `Comment` dohromady reprezentují událost s patřičnými kategoriemi a případnými komentáři. U těchto tříd dochází k serializaci při komunikaci se serverem, kdy jsou atributy tříd převáděny na JSON reprezentaci, která například v případě ukládání události tvoří tělo zasílané zprávy.

6.1.2 Komunikace se serverem

Stěžejní částí klientské aplikace je komunikace se serverem, která je funkčně rozdělena do dvou vrstev. První z nich je tvořena skupinou tříd rozšiřující třídu `AsyncTaskLoader` z knihovny Androidu souhrnně nazývaných jako *Loaders*. Primárním úkolem těchto tříd je provádět požadované asynchronní operace na pozadí aplikace. Je to především z důvodu, že na platformě Android není možné provádět síťovou komunikaci přímo z vlákna, které obstarává vykreslování uživatelského rozhraní. Dalším problémem je, že z tradičních vláken implementovaných pomocí třídy `Thread` a rozhraní `Runnable` nelze modifikovat obsah uživatelského prostředí. *Loaders* oba problémy úspěšně řeší. Komunikace s nimi je řešena pomocí rozhraní `LoaderCallbacks`, které obsahuje triptych metod `onCreateLoader(()`, `onLoadFinished()` a `onLoaderReset()`.

První z nich slouží k předání parametrů do prováděné operace. Druhá metoda je volána po jejím dokončení a v parametrech předává výsledek. Právě zde se provádí manipulace s uživatelským rozhraním. Poslední metoda umožňuje programátorovi provést potřebné kroky ve chvíli, kdy už operace běží, ale na základě dalšího požadavku je potřeba provádění resetovat. V implementaci klienta jsou *Loaders* dále rozděleny na 2 skupiny. První z nich slouží k načítání dat ze serveru kdy výsledkem operace je kolekce událostí nebo komentářů. Druhou skupinu tvoří třídy `EventTask`, `CommentTask` a `UserTask` provádějící konkrétní úkol. Jedná se například o uložení události (`TASK.SAVE`), smazání komentáře (`TASK.DELETE`) nebo přihlášení uživatele (`TASK.SIGNIN`). Výsledkem je v tomto případě koncový stav operace reprezentovaný výčtem `Service.STATE`, který předala odpovídající volána služba. Data načtená ze serveru jsou v rámci jednotlivých metod cacheovány třídou `EventCache`. Díky tomu má uživatel například přístup ke staženým událostem i ve chvíli, kdy nemá dostupné připojení.

Druhou vrstvou jsou služby zajišťující již přímou síťovou komunikaci se serverem. Před zasláním dotazu na API serveru je nejprve pomocí třídy `RequestBuilder` vytvořena URL dotazu. Detaily budování URL jsou popsány v části 6.3. Následně je pomocí knihovní třídy `RestTemplate` poslán na vytvořenou URL finální dotaz. U dotazu je pomocí instancí potomků třídy `AbstractHttpMessageConverter` možné nastavit formát, v jakém jsou data posílána a naopak jaký očekávány formát odpovědi. V případě, že je v těle dotazu posílán serializovaný objekt nebo jejich kolekce, je používán formát JSON. Serializaci do tohoto formátu zajišťuje knihovna `Jackson`. Pokud je požadována jen stavová informace, používá se obyčejný řetězec. V případě problému při komunikaci je vyhazována výjimka `ServiceException`.

6.1.3 Získávání GPS polohy

Aktuální GPS polohu uživatele je potřeba znát na několika místech v aplikaci. Z tohoto důvodu je její získávání řešeno pomocí mechanizmu služeb nabízených platformou Android. Principem tohoto mechanizmu je možnost se k požadované službě pomocí metody `bindService()` kdykoli připojit. Pokud služba neběží, je systémem automaticky spuštěna. V případě, že již běží, je využita existující instance. V případě služby pro zjišťování polohy je její běh ukončen ve chvíli, kdy se odpojí poslední zájemce.

Po připojení služby je na straně připojovaného zaregistrováno odebírání proprietárního[1] všesměrového vysílání aktualizací, které tato služba rozesílá v okamžiku, kdy se jí podaří

[1] Vysílání je opatřeno interním identifikátorem, díky kterému mohou zprávy z vysílání přijímat pouze komponenty v rámci vyvíjené aplikace.

načíst pozici. Ta je pak obsahem šířené zprávy. Po ukončení spojení se službou je provedeno odregistrování příjmu tohoto vysílání. Princip odebírání zpráv všesměrového vysílání je v Androidu nazýván *Broadcast Receivers*.

Služba při zjišťování polohy kontroluje přesnost aktuálně získaného měření a na základě hodnoty kvality buď novou hodnotu použije, nebo zahodí. Ohodnocování tzv. „fixů“ GPS polohy provádí model ve třídě `LocationModel` inspirovaný oficiální dokumentací Androidu [21]. Samotná služba je implementována třídou `LocationService`.

6.1.4 Interakce s uživatelem

O interakci aplikace s uživatelem se starají třídy označované jako aktivity nebo případně fragmenty. Aktivity a fragmenty zároveň reprezentují vrstvu *Controller* z návrhového vzoru MVC. Jednotlivé třídy jsou potomky generické třídy `Activity`, resp. `FragmentActivity` z knihovny Androidu. Rozdíl uvedených zástupců spočívá v možnosti připojení fragmentů k aktivitě. Fragmenty jsou znovupoužitelné komponenty, které v sobě zapouzdřují určitou část celkové funkcionality použitelnou z více míst jednotným způsobem. Výhodou jejich použití je především možnost připojení bez nutnosti změny kontextu, která nastává při přechodu z jedné aktivity do druhé. Hlavním specifikem derivátů aktivit je vytvoření viditelného uživatelského rozhraní (vrstva *View* z MVC) pro uživatele. Rozložení a vzhled prvků uživatelského rozhraní je v aplikaci řešen pomocí definic v XML souborech. Tyto soubory se označují jako *layout resources*. Společné vlastnosti, především grafické, jednotlivých *layoutů* jsou vyjmuty do jiných typů zdrojů (angl. *resources*, kterými jsou:

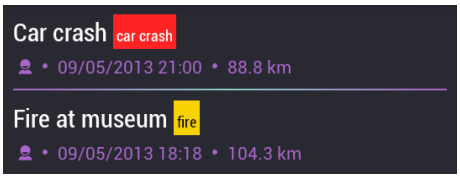
- styly (`styles.xml`), jež jsou obdobou kaskádových stylů z technologie HTML předepisující barvu pozadí a textů, zarovnání a uspořádání,
- dimenze (`dimens.xml`) sloužící především k centralizované specifikaci například velikostí písma, okrajů, mezer a
- barvy (`color.xml`) definující barevnou paletu používanou v aplikaci.

Dalšími používanými typy zdrojů jsou řetězce (`strings.xml`) a pole (`arrays.xml`), které slouží k definováni statických textů. Specifikace společných vlastností a statických hodnot v samostatných souborech je výhodná zejména z důvodu rychlého prototypování, podpory rozdílných zařízení a také z pohledu vícejazyčnosti. Aplikace je přeložena do anglického a českého jazyka.

Při spuštění aplikace je nejprve vytvořena instance třídy `MainActivity`, které ověřuje, zda se uživatel v minulosti již přihlásil. Pokud ne, je uživateli zobrazen registrační, resp. přihlašovací formulář. O obsluhu přihlášení nebo registrace se stará třída `LoginAcitivty`. Ta nejprve pomocí třídy `FormValidator` ověřuje, zda vůbec byly do formuláře zadány vstupní hodnoty a následně pomocí asynchronního úkolu skrze třídu `UserTask` kontroluje správnost zadaných údajů s hodnotami uloženými v databázi. Pokud jsou korektní, je uživatel přihlášen. V případě registrace je uživateli, za podmínky unikátního přihlašovacího jména v podobě e-mailové adresy, vytvořen účet. Při obou akcích je v případě neúspěchu uživateli zobrazena chybová hláška s patřičným obsahem.

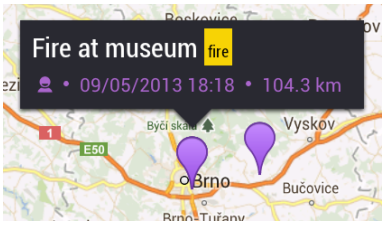
Pokud byl uživatel v aplikaci již přihlášen nebo úspěšně prošel fázi přihlášení, resp. registrace, je přesměrován na nastavený výchozí pohled pro přehled událostí. Aplikace je schopna uživateli prezentovat události buďto v podobě seznam nebo na mapě. První varianta je řešena v rámci třídy `EventListActivity`. Specifikem této třídy je ukládání jednotlivých událostí do adaptéru `EventAdapter`, což je třída mapující v metodě `getView()`

hodnoty každé události na odpovídající pohled definovaný souborem `event_list_item.xml`. Vlastní implementace adaptéru je použita z potřeby zobrazit položky seznamu způsobem, který Android v základu neumožňuje. Samotný seznam je klasická komponenta `ListView` dostupná v rámci Android SDK, která je nativně svázána s použitým adaptérem. Napojení adaptéru k seznamu dovoluje při výběru položky vrátit přímo instanci odpovídají události, což například při zobrazení jejího detailu znamená, že není potřeba událost opakovaně stahovat ze serveru. Podoba položky seznamu řešená vlastí implementací je zachycena na obrázku 6.1.



Obrázek 6.1: Reprezentace událostí v podobě seznamu pomocí vlastní implementace adaptéru `EventAdapter`.

Zobrazení událostí na mapě obstarává třída `EventMapActivity`, která využívá mapové podklady poskytované oficiální komponentou `MapView`. Hlavním úskalím při implementaci této třídy byla asociace událostí s grafickými prvky mapy. Asociování je řešeno pomocí kolekce objektů třídy `EventMarker`, která v sobě zapouzdřuje instanci události a k ní odpovídající grafický prvek na mapě reprezentovaný třídou `Marker` z knihovny `google-play-services`. Nad touto kolekcí je pak používána série pomocných metod, které v ní vyhledávají buďto požadovanou událost nebo jejího reprezentaci na mapě. Stejně jako v případě seznamu, je pro zobrazení informací o událostech použita vlastní implementace. Ta sestává jak z použití vlastních grafických podkladů (popup.9.png) s použitím technologie *9-patch*[2], tak i implementace patřičných metod vyvážených rozhraním `InfoWindowAdapter`. Hlavní je zde metoda `getInfoWindow()`, která podobně jako `getView()` u adaptéru pro seznam provádí mapování vlastností událostí na příslušné části pohledu specifikovaném souborem `event_info_window.xml`. Vlastní řešení zobrazení popisku události na mapě je ilustrováno obrázkem 6.2.



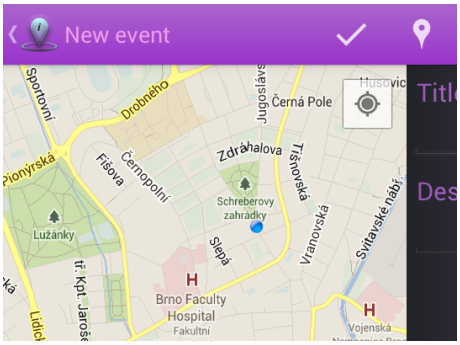
Obrázek 6.2: Zobrazení informací o události v mapovém pohledu formou bubliny. Nestandardního vzhledu je dosaženo vlastní implementací rozhraní `InfoWindowAdapter`.

Společné vlastnosti a funkcionalita obou tříd je řešena v rámci implementace přímého předka, kterým je třída `EventActivity`. Ta má na starost především obsluhu společných

[2]Pomocí technologie *9-patch* je možné vytvořit grafické elementy ve formátu PNG tak, aby systém sám tyto elementy upravoval dle parametrů displeje se zachováním požadované kvality detailů (např. stínů a zaoblených rohů)

akcí v menu. Jedná se zejména o načtení, resp. aktualizaci dat ze serveru a o možnost přidat novou událost. V prvním případě je nejprve připojena služba `LocationService`, která pomocí GPS modulu zjišťuje aktuální polohu. Získávání polohy je časově omezeno na dobu 15 vteřin. Je tak učiněno z důvodu, aby potencionálně dlouhé využívání GPS modulu nespotřebovalo příliš mnoho energie z baterie. Pokud vyprší časový limit, je uživatel informován o nedostupnosti polohy a je požádán o opakování akce později. V případě, že se podařilo polohu načíst, je využívání této služby ukončeno. Nyní je zahájena komunikace se serverem, resp. stažení aktuálních událostí, které zajišťuje třída `EventsLoader`. O výsledku operace je iniciátor informován pomocí mechanizmu zasílání událostí. Iniciátorem mohou být v tomto případě instance tříd `EventListActivity` a `EventMapActivity`. Ty implementují rozhraní `OnEventActionListener`, přes které jim jejich rodič, třída `EventActivity`, pošle zprávu `onEventSync()`. Obsahem zprávy je kolekce načtených události. Nyní je už na každém z příjemců, jak s výsledkem naloží. Zjednodušeně je buď překreslen seznam nebo mapa s událostmi.

Pokud uživatel požadoval přidání nové události, je ze stávající aktivity přesměrován na aktivitu `EventFormActivity`. Po jejím vytvoření je připojena služba `LocationService` a začíná získávání GPS polohy. Zjišťování polohy v tomto případě není po prvním zaměření ukončeno. Je to zejména proto, aby poloha události byla co nejpřesnější. V případě, že je uživatel v pohybu nebo se z důvodu slabého signálu nedaří polohu získat, je uživateli umožněno zadat pozici manuálně výběrem místa na mapě. Integrace map do aktivity je řešena pomocí fragmentu, viz obrázek 6.3. Mapa je v pohledu umístěna ve výchozím bodě na pozadí a ve chvíli, kdy je požadováno její zobrazení, je za pomocí animace zviditelněna. Primárním



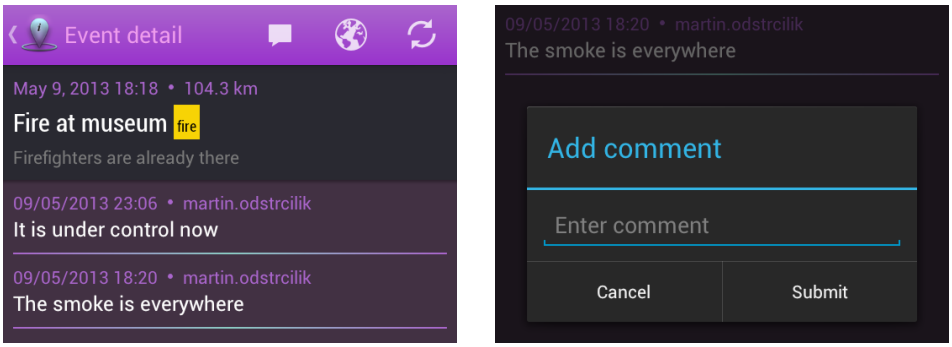
Obrázek 6.3: Demonstrace integrace mapy do formuláře pro vytvoření nové události pomocí fragmentu.

obsahem pohledu je formulář pro zadání titulku a popisu události. Vyplnění obou hodnot je povinné. Splnění podmínky je ověřováno validátorem formulářů `FormValidator`. Pro uložení je dále nutné mít k dispozici polohu události (buď automaticky, nebo manuálně získanou) a připojení. Pokud nejsou obě náležitosti splněny, je vypsána chybová hláška. V opačném případě je provedeno uložení události třídou `EventTask` s úkolem `TASK.SAVE`. Editace probíhá obdobně s tím rozdílem, že hodnoty formuláře jsou společně s polohou již předvyplněné a také není zjišťována aktuální GPS poloha.

K editaci události může uživatel přistoupit pomocí kontextového menu, které je v případě seznamu vyvoláno dlouhým stisknutím požadovaného záznamu. V mapovém pohledu se kontextová nabídka zobrazí po vybrání události stisknutím bodu na mapě. Pomocí kontextové nabídky je možné také událost smazat. Samotné odstranění události je provedeno

úkolem `TASK.DELETE` přes třídu `EventTask`. Oba úkony jsou uživateli zpřístupněny pouze v případě, že je vlastníkem dané události. V opačné situaci je provedení operace odmítnuto. Vlastnictví události je interně stanoveno na základě unikátního klíče uživatele. Graficky jsou uživatelovy události označeny symbolem panáčka, viz obrázek 6.1.

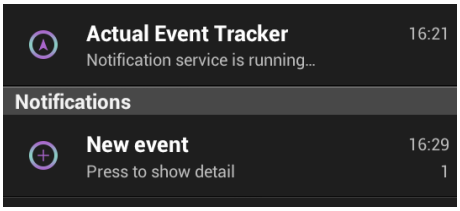
Po krátkém stisknutí položky v seznamu nebo informační bubliny na mapě je zobrazen detail asociované události. Hlavní funkcí detailu události je zobrazení komentářů. Jednotlivé komentáře jsou uživateli prezentovány formou standardního seznamu komponentou `ListView` s vlastní implementací adaptéru `CommentAdapter`. Přidávání komentářů je řešeno pomocí dialogového okna obsahujícího vstupní pole vyvolaného po stisknutí tlačítka v menu. Totožný dialog s předvyplněným textem je zobrazen i při editaci komentáře. Obsah komentáře je povinný. Validaci formuláře obstarává opět `FormValidator`. Editaci je možné vykonat skrze kontextovou nabídku. Její zobrazení se provede po dlouhém stisknutí požadovaného záznamu v seznamu. Kontextové menu nabízí rovněž možnost smazat vybraný komentář. Stejně jako v případě událostí, je uživateli dovoleno takto spravovat pouze jím přidané komentáře. Uživatelovy komentáře jsou opatřeny částí e-mailové adresy použité při registraci. Načítání komentářů je řešeno pomocí třídy `CommentsLoader`, jejich editaci a smazání realizuje třída `CommentTask`. Ukázka detailu události je zachycena na obrázku 6.4.



Obrázek 6.4: Ilustrace detailu události, který doplňuje k informacím popis události a zároveň slouží ke komentování událostí.

6.1.5 Výskyt nových událostí

Součástí aplikace je systém pro upozornění uživatele při výskytu nové události. Tento systém je implementován ve třídě `NotificationService` jako služba, která je spuštěna se startem aplikace, resp. po případném přihlášení. Služba od této chvíle běží na pozadí a přežívá i ukončení samotné aplikace. Interně služba provádí pravidelnou aktualizaci událostí. Uživatel je o aktuálním stavu informován v notifikační oblasti systému. Zde se zobrazují i hlášení informující o výskytu nové události. Po stisknutí záznamu je uživateli zobrazen detail nové události, nebo v případě většího počtu událostí jejich přehled. K registraci událostí, na které byl uživatel již upozorněn, je využívaná cache implementována třídou `NotificationCache`, která udržuje identifikátory zpracovaných událostí. Obrázek 6.5 ilustruje způsob, jakým je uživatel upozorněn na výskyt nové události.



Obrázek 6.5: Způsob prezentace výskytu nové události uživateli pomocí systému upozorňování.

6.1.6 Nastavení aplikace

Aplikace nabízí několik možností vlastního nastavení. Jedná se především o volbu parametrů událostí, které si uživatel přeje stahovat. Parametry se nastavují zvlášť jak pro samotnou aplikaci, tak pro systém upozorňování. Nabídka nastavení je dostupná přes hlavní menu aplikace. Nabízené možnosti jsou:

- **výchozí pohled** – poskytuje uživateli výběr mezi seznamem a mapou
- **vzdálenost** – umožňuje nastavit maximální vzdálenost stahovaných událostí od uživatele
- **stáří událostí** – definuje, jak staré události uživatele zajímají
- **relevantnost** – při povolení této volby budou stahovány pouze pozitivně hodnocené a neutrální události
- **filtr kategorií** – umožňuje vybrat podmnožinu kategorií, které jsou pro uživatele důležité
- **automatická synchronizace** – povolí automatickou aktualizaci po spuštění aplikace
- **velikost cache** – definuje maximální velikost cache pro události
- **server** – adresa serveru, se kterým má klient komunikovat
- **povolení notifikací** – zapne systém pro upozorňování

6.1.7 Uživatelská nápověda

V rámci aplikace je pro začínající uživatele vytvořena nápověda. Její snahou je uživateli vysvětlit jednotlivé prvky ovládání pomocí graficky znázorněných úkonů. Nápovědu uživatel zpřístupní opět skrze hlavní nabídku aplikace. Implementačně je nápověda řešena aktivitou, která ale vnitřně přepíná pomocí gest fragmenty. Každý fragment pak v sobě zaobaluje určitou část nápovědy.

6.1.8 Testování

Testování klientské aplikace probíhalo formou beta testování pomocí tří subjektů[3] s cílem odhalit chyby zejména ve funkcionalitě. Vzhledem k počtu testujících se nepodařilo dostatečně otestovat použitelnost klasifikace v reálném prostředí.

[3] Testující subjekty disponovaly fyzickými zařízeními střední třídy sestávajících z mobilních telefonů HTC Evo 3D, Sony Xperia S a tabletu Toshiba AT100.

6.2 Serverová část

Serverová část je implementována jako webová Java EE aplikace pomocí aplikačního rámce Spring MVC starajícího se zejména o propojení služeb, repositářů a beanů z jednotlivých vrstev aplikace. Další důležitou funkcí je mapování příchozích požadavků od klienta na výchozí servlet a obslužnou funkci. V obou případech jsou k provedení potřebných akcí použity anotace. Například injekce potřebné služby je docíleno označením třídní proměnné anotací `@Autowired`. Instance napojeného objektu je pak dostupná v celém těle třídy. Pro namapování požadavku na konkrétní obslužnou funkci je použita anotace `@RequestMapping`, v jejímž těle je uvedeno schéma požadavku. Spring se následně postará o rozparsování příchozího dotazu a na základě hodnot jednotlivých částí URL nastaví příslušné[4] vstupní parametry označené anotací `@PathVariable`. Definice třídy s použitím popsaných anotací je ilustrována následující části kódu:

```
public class EventController {
    private @Autowired IEventService eventService;

    @RequestMapping(value = "/events/delete/{id}")
    public boolean delete(@PathVariable Long id) {
        return eventService.delete(id);
    }
}
```

6.2.1 Zpracování požadavků

Vstupním bodem aplikace při přijetí požadavku je MVC vrstva *Controller*, která sestává ze tříd `EventController`, `CommentController` a `UserController`. Prvně uvedená třída zpracovává dotazy týkající se událostí. Podporované akce jsou získání všech aktuálních událostí, uložení nové a smazání existující události. Proces při zpracování požadavků je následující. Nejprve jsou na základě hodnot příchozího dotazu nastaveny parametry obslužné funkce. Pokud požadavek obsahuje i tělo[5], Spring jej dokáže namapovat do vstupní proměnné označené anotací `@RequestBody`. Interně tak činí pomocí knihovny `Jackson` pro parsování JSON formátu a následnou deserializaci do objektu přenášené třídy. V těle obslužné funkce je při vykonání provedena validace požadavku pomocí třídy `RequestValidator`. Postup validace je uveden v části 6.3. V případě, že je dotaz legitimní, je pomocí služby `UserService` provedena aktualizace záznamů o poslední aktivitě uživatele. Následně je přistoupeno k samotnému vykonání požadované akce. Její provedení se v rámci této třídy odehrává nad službou `EventService`. Pokud je výsledkem zpracování požadavku instance objektu, je uživateli poslána odpověď ve formátu JSON. Serializaci do tohoto formátu zajišťuje opět knihovna `Jackson`. V případě pouhé stavové odpovědi je poslán obyčejný řetězec. O formátu odpovědi rozhoduje Spring sám na základě návratového typu obslužné funkce. Pro zajištění této funkcionality je funkce označena anotací `@ResponseBody`.

Při zpracovávání požadavku může dojít k situaci, kdy je dotaz při validaci z bezpečnostních důvodů zamítnut (výjimka `SecurityException`) nebo když používaná služba není schopna požadavek zpracovat (`výjimka ServiceException`). Zpracování vzniklých výjimek obstarává třída `GlobalExceptionHandler`, která nastaví příslušný kód odpovědi v HTTP hlavičce. Tělo zasílané odpovědi pak obsahuje vysvětlující zprávu.

[4] Pořadí vstupních parametrů funkce musí odpovídat pořadí mapovaných části z URL požadavku.
[5] V tělech požadavků jsou přenášeny pouze doménové objekty.

Popsaný způsob zpracování požadavků se obdobně opakuje i v dalších uvedených třídách s tím rozdílem, že každá pro zpracování požadavku využívá jí odpovídající službu z modelové vrstvy MVC.

6.2.2 Zaznamenávání dotazů a odpovědí

V aplikaci jsou zaznamenávány příchozí požadavky a odpovídající odpovědi. Záznam je implementován pomocí aspektových tříd. Tyto třídy jsou označeny anotací `@Aspect`. Příslušná metoda aspektu je opatřena anotací `@Around`, která zajistí obalení zaznamenávané obslužné metody a provede mapování parametrů. Obalovaná metoda je zapouzdřena v instanci třídy `ProceedingJoinPoint`, nad kterou se posléze v těle metody aspektu volá její vykonání. Následující část kódu ilustruje princip zaznamenávání požadavků pomocí aspektu:

```
@Aspect
public class EventRequestLoggingAspect {
    @Around("execution(* EventController.delete(..)) " + "&& args(id))")
    public Object onDeleteRequest(ProceedingJoinPoint pjp, Long id) throws
    Throwable {
        LogMF.debug(LOGGER, "Request /events/delete/{0}/", id);
        //vykonání obalované metody delete(..) ze třídy EventController
        Boolean result = (Boolean) pjp.proceed();
        LogMF.debug(LOGGER, "Request result is {0}", result);

        return result;
    }
}
```

6.2.3 Doménové entity

Aplikace pracuje s několika třídami, které jsou jak předmětem požadavků na API, tak objekty ukládanými do databáze. Jedná se o třídy `Event`, `Category`, `Comment` a `User` souhrnně označované jako doménové entity. Právě tyto třídy jsou při komunikaci s klienty serializovány, resp. deserializovány z formátu JSON. Zároveň jsou atributy těchto tříd pomocí anotací standardního rozhraní JPA mapovány na tabulky v databázi. Pomocí JPA anotací jsou definovány i vztahy mezi entitami. Následující část kódu demonstruje použití JPA anotací při definici identifikátoru události a vztahu mezi událostmi a kategoriemi:

```
public class Event {
    @Id
    @Column(name="ID")
    @GeneratedValue(generator="generator")
    @GenericGenerator(name="generator", strategy="sequence-identity",
     parameters = @Parameter(name="sequence", value="AET_EVENT_ID_SEQ"))
    private Long id;

    @ManyToMany
    @JoinTable(name="AET_EVENT_CATEGORY",
     joinColumns={@JoinColumn(name="ID_EVENT")},
     inverseJoinColumns={@JoinColumn(name="ID_CATEGORY")})
```

```
    private Set<Category> categories;
}
```

6.2.4 Mezivrstva služeb

Vykonávání požadavků při jejich zpracování v kontrolérech se děje pomocí služeb reprezentujících byznys logiku aplikace. Služby jsou zastoupeny třídami s anotací `@Service`, díky které jsou použitelné přes anotaci `@Autowired` v kterémkoli místě aplikace. Interně se jedná o klasickou Java Bean komponentu s přidanou sémantikou. Hlavním úkolem služeb v této vrstvě je příprava dat před uložením do databáze a v opačném případě před zasláním odpovědi klientovi. Konkrétně se před uložením události provádí konverze její polohy. Poloha je v požadavku zasílána jako dvojice hodnot v plovoucí řádové čárce, ale pro uložení do databáze je potřeba mít pozici v objektu třídy `Geometry`, se kterou pracuje Hibernate, resp. jeho rozšíření pro prostorové operace. Opačným směrem zase probíhá konverze při požadavku na načtení událostí.

Další konkrétní demonstrací využití mezivrstvy je klasifikace, kdy při ukládání události probíhá její kategorizace. Provedení kategorizace má za následek vytvoření asociací události s odpovídajícími kategoriemi. Rovněž při vložení komentáře dochází ke klasifikaci sentimentu jeho obsahu a na základě výsledku je patřičně upravena příslušná událost.

Vrstva služeb také řídí transakční zpracování operací a práci z cache. První vlastnosti je docíleno označením třídy služby anotací `@Transactional`. Spring pak na základě této anotace sám při zahájení databázové operace započne (*begin*) transakci a v případě úspěšného provedení všech součástí transakce následně celou transakci automaticky potvrdí (*commit*). Pokud dojde při zpracovávání k chybě, provede návrat (*rollback*). Průběh transakce je možné řídit i manuálně. Pomocí anotací je řízeno i ukládání, resp. mazání výsledků operací do, resp. z cache. Uložení výsledku do cache probíhá až po první dotazu. Při druhém je již použita hodnota z cache. Ke smazání hodnoty z cache dochází při její změně na základě klíče, kterým je identifikátor záznamu. Interně je ke cacheování používána knihovna `EhCache`. Cacheování při práci s událostmi ilustruje následující část kódu:

```
@Service
public class EventService {
    private @Autowired IEventDao eventDao;

    @Cacheable(value="event", key="#id")
    public Event get(Long id) {
        return eventDao.get(id);
    }

    @CacheEvict(value="event", key="#id")
    public boolean delete(Long id) {
        return eventDao.delete(id);
    }
}
```

6.2.5 Služba klasifikace

Klasifikaci dat provádí služba reprezentována třídou `ClassificationService` metodami `getSentiment()` a `getCategories()`. Služba provádí klasifikaci pomocí aplikace Rapid-

Miner. Při spouštění serveru je provedena metodou `init()` inicializace služby. Inicializace zahrnuje připravení instance RapidMineru, připojení repositáře obsahujícího klasifikační modely a vektory příznaků a načtení definic klasifikátorů. Definice jsou uloženy v souborech `sentiment_model_application.xml` a `category_model_application.xml`. Na základě těchto definic je RapidMiner schopen vytvořit instance klasifikátorů. Při samotné klasifikaci uvedenými metodami služby dochází již pouze k připojení odpovídajícího modelu a vektoru příznaků do klasifikátoru a ke spuštění klasifikace nad vstupními daty. Výsledek klasifikace je uložen a vrácen v pomocných třídách `Sentiment` a `Category`. Samotná interpretace výsledků je již v režii volající třídy. V případě analýzy sentimentu je to služba `CommentService`, při kategorizaci události pak služba `EventService`.

6.2.6 Interakce s databází

Interakci s databází zajišťují DAO třídy opatřené anotací `@Repository`. Stejně jako v případě služeb se jedná o klasické komponenty s dodanou sémantikou. Operace nad databází jsou prováděny v rámci aktuálního sezení (angl. *session*), ke kterému má DAO třída přístup skrze třídu `SessionFactory`, jejíž instance je automaticky připojena. Nad sezením jsou v jednotlivých metodách tříd typicky volány CRUD operace. Restrikce a řazení výsledků dotazů jsou řešeny pomocí kritérií implementovaných ve třídě `Criteria` knihovny Hibernate. Práce s databází při získávání komentářů k události ilustruje následující kód:

```
@Repository
public class CommentDao {
    private @Autowired SessionFactory sessionFactory;

    public List<Comment> getAll(Long eventId) {
        Session session = sessionFactory.getCurrentSession();
        Criteria criteria = session.createCriteria(Comment.class);
        criteria.add(Restrictions.eq("eventId", eventId));
        criteria.addOrder(Order.desc("timestamp"));
        criteria.addOrder(Order.desc("id"));

        return criteria.list();
    }
}
```

6.2.7 Testování

V rámci testování serverové části aplikace bylo vytvořeno několik jednotkových testů. Testování je rozděleno do dvou částí. První z nich ověřuje modelovou vrstvu aplikace, kdy je testována vrstva služeb voláním jednotlivých metod s různými vstupními parametry. Touto sadou testů je zároveň vyhodnocena i správná funkcionalita databázové vrstvy. Druhá sada testů se zaměřuje na testování API. V rámci této sady testů jsou na API posílány požadavky a následně je provedeno vyhodnocení testu porovnáním HTTP kódu v hlavičce obdržené odpovědi s předpokládanou hodnotou. Všechny testy jsou tvořeny s pomocí knihovny pro podporu testování `JUnit`. Při testování API je navíc použita knihovna `mockito`, která umožňuje simulovat zaslání požadavku.

6.2.8 Nastavení a kontext aplikace

Základní nastavení aplikace je obsaženo v souboru `web.xml`, ve kterém je provedeno mapování všech dotazů na hlavní a jediný servlet `aet`. Rovněž je zde v souboru `aet-context.xml` nastaven kontext aplikace. Součástí definice kontextu je nastavení používaných knihoven včetně samotného Springu, a komponent. Konstantní hodnoty nastavení jako jsou hesla a přístupové cesty jsou vyjmuty do speciálního konfiguračního souboru `aet.properties`.

6.3 Zabezpečení komunikace

K zabezpečení komunikace je používána knihovna `aet-security` obsahující dvě protichůdné třídy `RequestBuilder` a `RequestValidator`. První z nich se stará o vytvoření URL dotazu, druhá kontroluje jeho platnost. Proces vytváření URL adresy probíhá následovně. Přes konstruktor třídy `RequestBuilder` jsou nastaveny povinné parametry dotazu, kterými jsou skupina dotazů, požadovaná akci a klíč uživatele. Případné volitelné parametry a tělo dotazu je možné nastavit pomocí *setterů*. Nad inicializovanou instancí je volána metoda `build()`, která provede sestavení URL a vrátí ji v podobě řetězce. V rámci sestavování jsou postupně jednotlivé nastavené atributy skládány za sebe. Jako oddělovač je použit znak lomítka. K takto vniklé části je na konec připojeno časové razítko a výsledný autentizační hash. Hash je počítán funkcí `hmacHash()` ze třídy `Encryption`. Jejím vstupem je již vytvořená část (včetně časového razítka) URL společně se zakódovaným tělem dotazu. Před samotnou aplikací hashovacího algoritmu je vstup dodatečně osolen. Kódování těla dotazu zajišťuje funkce `uriEncode`, která na základě proprietární anotace `@UriEncode` spojuje jí označené vlastnosti kódované třídy do posloupnosti řetězců ve tvaru `<název atributu>=<hodnota>` oddělených znakem `&`. Výsledný dotaz pro získání všech aktuálních událostí ze serveru `147.229.8.225` poslouchajícím na portu `8080` a s cestou k aplikaci `aet` by měl podobu:

```
http://147.229.8.225:8080/aet/events/getAll/49.5218379/17.9653369/10000/12/
false/67a88ef2-3275-46ea-b637-8e18cdbe8a58/1368197235708/cf96e66ec71ca98ea7
07f4062088d80542fa19970647e2b7c8fcc2d33c80fe7f
```

Hodnota `events` značí skupinu dotazu a `getAll` akci, hodnoty `49.5218379`, `17.9653369`, `10000`, `12`, `false` jsou parametry dotazu, řetězec začínající `67a88ef2-` je klíč uživatele, číslo `1368197235708` reprezentuje časové razítko a poslední část je vypočítaný autentizační hash.

Při validaci požadavku metodou `validate()` třídy `Requestvalidator` je nejprve kontrolováno časové razítko dotazu s posledním záznamem o aktivitě uživatele, který je dle klíče autorem požadavku. Pokud je datum a čas poslední aktivity časově před časovým razítkem dotazu, je nadále považován za platný[6]. Následně je naprosto stejným způsobem jako při vytváření dotazu vyroben autentizační hash. Ten je nakonec porovnán s hashem, který je součástí v požadavku. Pokud se vzájemně shodují, je požadavek shledán validním. V opačném případě je vyhozena výjimka `SecurityException`.

[6] Datum a čas poslední aktivity uživatele je uchováváno v databázi. Tímto mechanizmem se snaží aplikace chránit proti *replay* útokům.

Kapitola 7

Závěr

Závěrečná kapitola hodnotí průběh celé práce a shrnuje dosažené výsledky v porovnání s předpokládaným cílem. Kapitola se také zaobírá možnými rozšířeními aplikace do budoucnosti v návaznosti na aktuálně řešené diplomové práce v příbuzné oblasti.

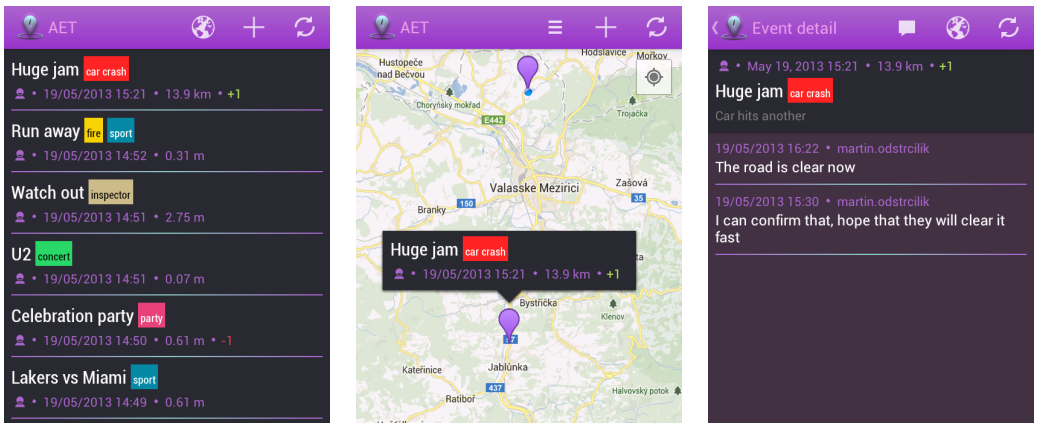
7.1 Průběh a výsledek práce

K zadání práce bylo přistupováno už od začátku velmi svědomitě, neboť zadání jako takové vzešlo z mých nápadů. Po prvotní konzultaci s vedoucím práce byly původní plány obohaceny o prvek klasifikace, který se rozrostl až do podoby, kdy je aplikace mimo původně zamýšlenou analýzu sentimentu událostí schopna i jejich kategorizace. Do tohoto projektu jsem vstupoval s cílem se zdokonalit v programovacích praktikách při vývoji aplikací pro mobilní zařízení s operačním systémem Android. S tímto záměrem jsem se v průběhu snažil používat pro mě neznámé a neprobádané zákoutí této platformy, i když jsem si tím poměrně znesnadňoval práci. Na začátku jsem měl rovněž drobné znalosti Java EE standardu a aplikačního rámce Spring MVC. Tyto znalosti jsem prací na tomto projektu značně rozšířil a již v průběhu roku zúročoval na jiných projektech. Největším přínosem pro mě samotného však zůstávají poznatky obdržené studiem problematiky klasifikace textů. Z pohledu dodržení předepsaných bodů zadání jsem zcela určitě naplnil bod první. Při práci na bodu druhém jsem nejprve provedl analýzu existujících řešení na trhu a shledal, že úplnou podobu mnou řešeného problému zatím nikdo neřešil[1]. V rámci tohoto bodu jsem dále hledal možnosti realizace ještě nerozhodnutých částí aplikace, kterou byla například komunikace mezi klientskou a serverovou částí aplikace nebo databázová a ORM technologie. Rovněž došlo i na výběr programu pro podporu klasifikace. Poznatky z této analýzy vyústily v obsah kapitoly 2. Stěžejní části bylo studium problematiky klasifikace dat. Velkou oporou v této věci mi byl pan inženýr Kouřil, který mi poskytl velmi cenné informace. Mé porozumění této problematice reflektuje kapitola 3. V kapitole 4 jsem provedl shrnutí v aplikaci použitých technologií a algoritmů. Posledním krokem ke splnění druhého bodu zadání bylo navrhnutí aplikace. Zde jsem se zaměřil především na návrh architektury klientské i serverové aplikace a klasifikátorů. Rovněž jsem se snažil o realizaci pohledného a použitelného uživatelského rozhraní. Pozornosti neušla ani databáze a komunikace. Výsledky návrhu jsou popsány v kapitole 5. Časově nejnáročnější část celého projektu tvořil bod třetí, kterým byla implementace shrnutá v kapitole 6. Při popisu implementace jsem se snažil o vystih-

[1] V průběhu roku se objevila poměrně podobná aplikace pro sledování pohybu revizorů FareBandit, viz kapitola 2.

nutí důležitých částí doplněných pro názornost grafickou ukázkou nebo alespoň malou částí kódu. Splnění posledních dvou bodů zadání je realizováno v rámci této kapitoly.

Hlavním výstupem této práce je aplikace složená ze dvou částí. První z nich slouží jako klient, pomocí kterého mohou jeho uživatelé sledovat aktuální dění v jejich okolí. Druhou aplikací je server, který zajišťuje ukládání a distribuci dat mezi klienty. Cílem celé aplikace je poskytovat uživatelům informace ať se nacházejí kdekoli. Proto je klientská část řešena formou aplikace pro mobilní zařízení. Rovněž se aplikace snaží o poskytování co nejvíce relevantních informací. Za tímto účelem dochází na straně serveru k analýze vkládaných dat. Z pohledu zadání tak aplikace naplňuje očekávaný výsledek. Část výsledku práce v podobě klientské aplikace je ilustrována obrázkem 7.1.



Obrázek 7.1: Ukázka uživatelského rozhraní aplikace klienta. První obrázek zleva ilustruje zobrazení událostí v seznamu, druhý pak na mapě. U jednotlivých událostí je mimo základní informace naznačena také kategorie (barevný popisek za titulkem) a relevantnost (zelená nebo červená +/- hodnota) události. Z obou pohledů je možné přidávat nové události, a upravit nebo odstranit existující. Rovněž je možné zobrazit detail události. Ten je zachycen na třetím obrázku. Obrazovka detailu událostí slouží také ke správě existujících a přidávání nových komentářů k události.

7.2 Vhodná rozšíření aplikace

Aplikace je schopna klasifikovat sentiment a kategorie událostí pouze pro anglický jazyk. Z tohoto pohledu by bylo vhodné rozšířit podporu o dalších jazyky, minimálně češtinu. V návaznosti na podporu dalších jazyků v rámci klasifikace by mohly být přidány i související lokalizace samotné aplikace. Dále by bylo přínosné doplnit do aplikace více kategorií. K dalšímu vylepšení aplikace by přispěla integrace jiných sociálních sítí. Na základě diplomové práce Jiřího Rašky [36] by z integrovaných sítí bylo možné získávat pro aplikaci užitečná data. V neposlední řadě by součástí dalších prací na aplikaci mělo být dopsání automatizovaných testů klientské části, zejména pak uživatelského rozhraní. Rovněž by bylo vhodné podrobit aplikaci širšímu beta testování se zaměřením na schopnosti klasifikování sentimentu a kategorií reálných událostí.

Literatura

[1] Apple [online]. http://www.apple.com, 2012 [cit.2012-12-10].

[2] Hibernate [online]. http://www.hibernate.org, 2012 [cit.2012-12-10].

[3] Minimal style login form [online]. http://cssdeck.com/labs/minimal-style-login-form, 2012 [cit.2013-01-02].

[4] FareBandit [online]. https://play.google.com/store/apps/details?id=revizorwatch.cz, 2012 [cit.2013-04-17].

[5] Foursquare [online]. https://foursquare.com, 2013 [cit.2013-04-17].

[6] Localmind [online]. https://itunes.apple.com/us/app/localmind/id422776889, 2013 [cit.2013-04-17].

[7] Trapster [online]. https://play.google.com/store/apps/details?id=com.trapster.android, 2013 [cit.2013-04-17].

[8] tudyNE [online]. https://play.google.com/store/apps/details?id=cz.tudyne.app, 2013 [cit.2013-04-17].

[9] The R Project for Statistical Computing [online]. http://www.r-project.org/, 2013 [cit.2013-05-10].

[10] RapidMiner [online]. http://rapid-i.com/, 2013 [cit.2013-05-10].

[11] Weka [online]. http://www.cs.waikato.ac.nz/ml/weka, 2013 [cit.2013-05-10].

[12] Google: There Are 900 Million Android Devices Activated [online]. http://www.businessinsider.com/900-million-android-devices-in-2013-2013-5, 2013 [cit.2013-05-18].

[13] ALPAYDIN, E.: *Introduction to machine learning*. Cambridge, Mass: MIT Press, 2010, ISBN 026201243X.

[14] Apple: Apple Reinvents the Phone with iPhone [online]. http://www.apple.com/pr/library/2007/01/09Apple-Reinvents-the-Phone-with-iPhone.html, 2007 [cit.2012-12-07].

[15] BONTEMPI, G.: Machine learning methods for bioinformatics: Feature selection [online]. http://www.ulb.ac.be/di/map/gbonte/bioinfo/course4.pdf, 2013 [cit.2013-04-27].

[16] FIELDING, R. T.: *Architectural Styles and the Design of Network-based Software Architectures*. Dizertační práce, University of California, Irvine, 2000.

[17] Google: Open Handset Alliance Releases Android SDK [online]. http://www.openhandsetalliance.com/press_111207.html, 2007 [cit.2012-12-07].

[18] Google: Android, the world's most popular mobile platform [online]. http://developer.android.com/about/index.html, 2012 [cit.2012-12-09].

[19] Google: Design [online]. http://developer.android.com/design/index.html, 2012 [cit.2013-01-02].

[20] Google: Dashboards [online]. http://developer.android.com/about/dashboards/index.html, 2013 [cit.2013-04-17].

[21] Google: Android for Developers [online]. http://developer.android.com/develop, 2013 [cit.2013-10-05].

[22] GUYON, I.; ELISSEEFF, A.: Lecture 9: Embedded Methods [online]. http://clopinet.com/isabelle/Projects/ETH/lecture9.pdf„ 2006 [cit.2013-04-27].

[23] GUYON, I.; GUNN, S.; NIKRAVESH, M.; aj.: *Feature Extraction: Foundations and Applications*. Springer, 2006, ISBN 3540354875.

[24] HASHIMI, S.; KOMATINENI, S.; MACLEA, D.: *Pro Android 2*. Apress, 2010, ISBN 978-1-4302-2659-8.

[25] JOACHIMS, T.: Text Categorization with Support Vector Machines: Learning with Many Relevant Features. *LS VIII-Report*, ročník 23, 1997, ISSN 0943-4135.

[26] JOACHIMS, T.: *Learning to Classify Text Using Support Vector Machines*. Kluwer Academic Publishers, 2002, ISBN 0-7923-7679-X.

[27] JOHNSON, R.: Introduction to the Spring Framework [online]. http://www.theserverside.com/news/1364527/Introduction-to-the-Spring-Framework, 2005 [cit.2012-12-10].

[28] LI; SHOUSHAN; XIA; aj.: A framework of feature selection methods for text categorization. *Proceedings of the Joint Conference of the 47th Annual Meeting of the ACL and the 4th International Joint Conference on Natural Language Processing of the AFNLP*, 2009: s. 692–700.

[29] MANNING, C.: *Introduction to information retrieval*. New York: Cambridge University Press, 2008, ISBN 0521865719.

[30] McCALLUM, A.; NIGAM, K.: A comparison of event models for naive bayes text classification. In *AAAI-98 workshop on learning for text categorization*, ročník 752, Citeseer, 1998, s. 41–48.

[31] MITCHELL, T.: *Machine Learning*. New York: McGraw-Hill, 1997, ISBN 0070428077.

[32] NG, A. Y.; JORDAN, M. I.: On Discriminative vs. Generative Classifiers: A comparison of logistic regression and naive Bayes. In *NIPS*, 2001, s. 841–848.

[33] ODSTRČILÍK, M.: *Osobní plánovač financí*. Diplomová práce, Fakulta infromačních technologií Vysokého učení technického v Brně, 2010.

[34] PICHLÍK, R.: Spring framework - J2EE s lidskou tváří [online]. http://www.slideshare.net/pichlik/spring-framework-j2ee-s-lidskou-tvari, 2009 [cit.2012-12-10].

[35] PORTER, M. F.: An algorithm for suffix stripping. *Program*, ročník 14, č. 3, 1980: s. 130–137.

[36] RAŠKA, J.: *Dolování dat v prostředí sociálních sítí*. Diplomová práce, Fakulta infromačních technologií Vysokého učení technického v Brně, 2013.

[37] RICHARDSON, L.; RUBY, S.: *RESTful Web Services*. O'Reilly, 2007, ISBN 978-0-596-52926-0.

[38] SIMEON, M.; HILDERMAN, R.: Categorical Proportional Difference: A Feature Selection Method for Text Categorization. *Proceedings of the 7th Australasian Data Mining Conference*, 2008: s. 201–208.

[39] SpringSource: Creating RESTful services [online]. http://static.springsource.org/spring/docs/3.0.0.M3/reference/html/ch18s02.html, 2012 [cit.2012-12-10].

[40] SpringSource: Spring for Android [online]. http://www.springsource.org/spring-android, 2012 [cit.2012-12-10].

[41] SpringSource: SpringSource Community [online]. http://www.springsource.org, 2012 [cit.2012-12-10].

[42] StatCounter: Top 8 Mobile Operating Systems from Jan to Apr 2012 [online]. http://gs.statcounter.com/#mobile_os-ww-monthly-201301-201304-bar, 2013 [cit.2013-04-17].

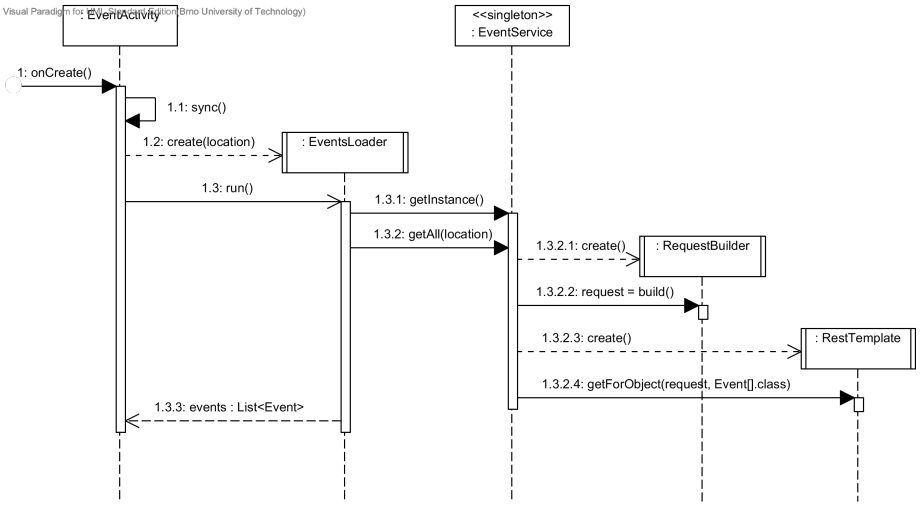
[43] YANG, Y.; JOACHIMS, T.: Text categorization. *Scholarpedia*, ročník 3, č. 5, 2008: str. 4242.

[44] YANG, Y.; PEDERSEN, J. O.: A Compatative Study on Feature Selection in Text Categorization. *Machine Learning-International*, 1997: s. 412–420.

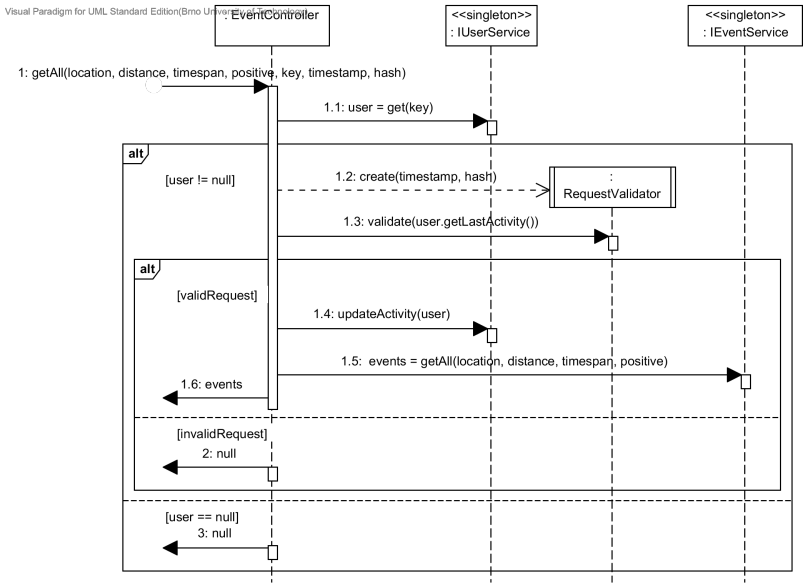
Seznam příloh

Příloha A

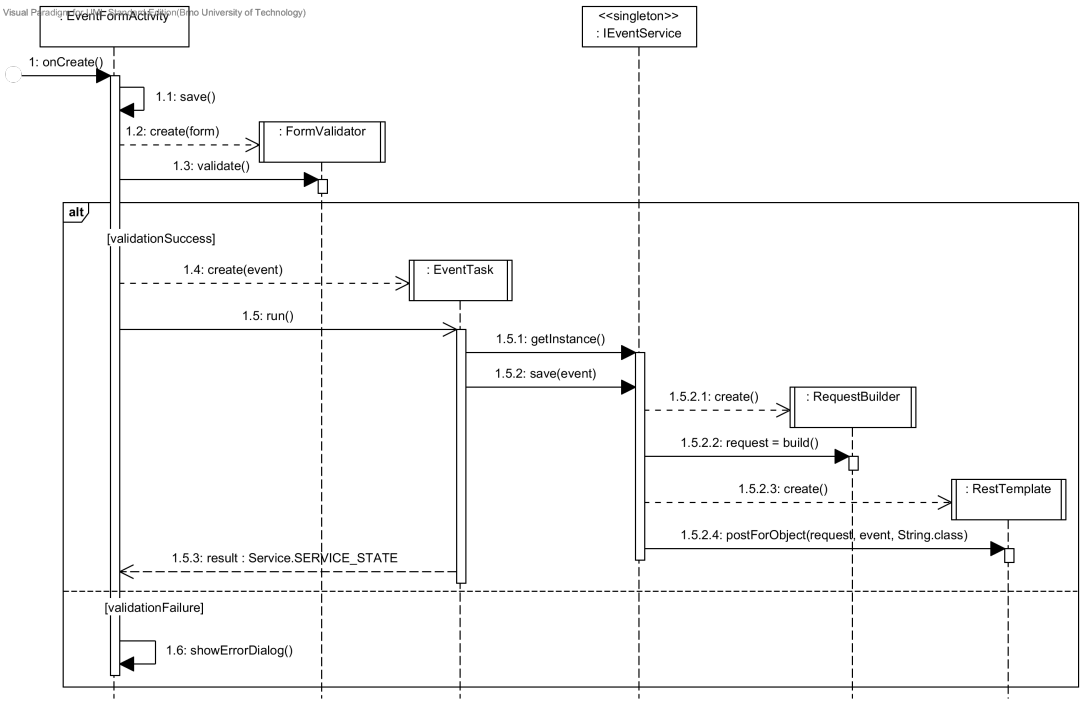
Návrhové diagramy



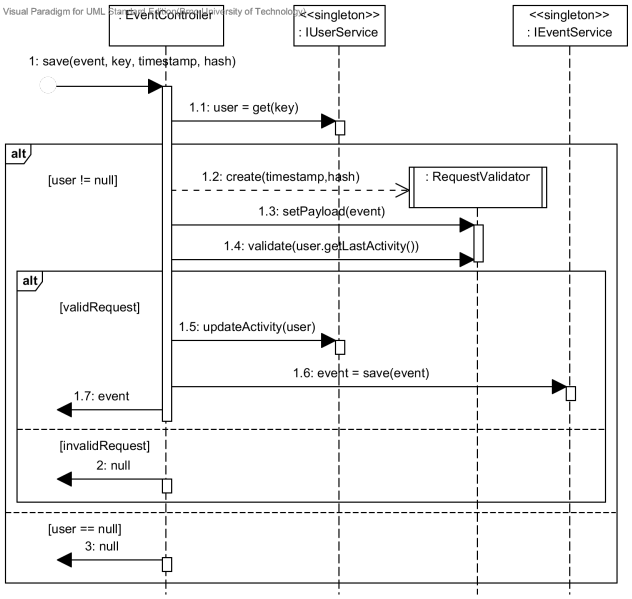
Obrázek A.1: Diagram sekvence pro případ užití *zobrazit události* z pohledu klientské části aplikace.



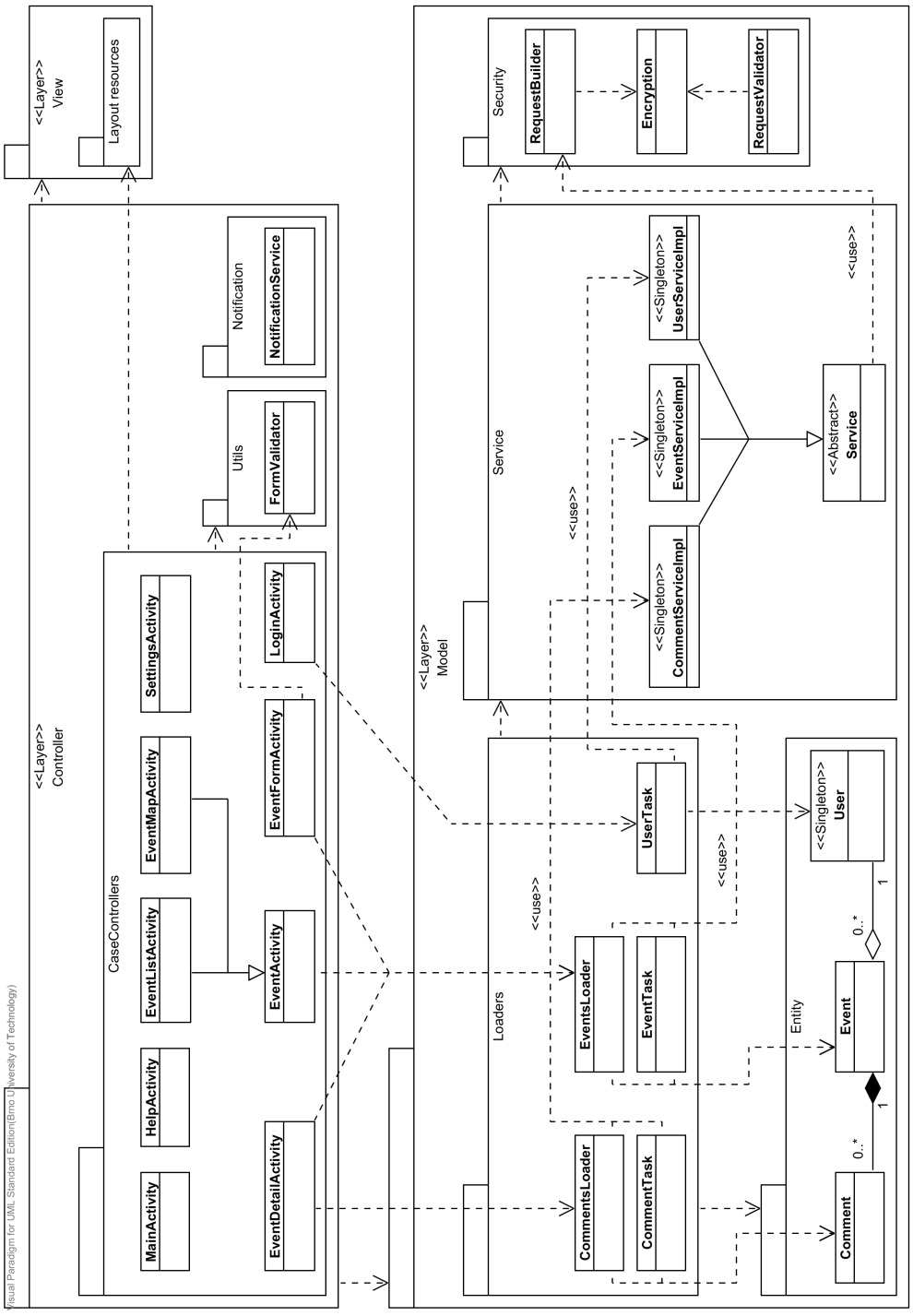
Obrázek A.2: Diagram sekvence pro případ užití *zobrazit události* z pohledu serverové části aplikace.



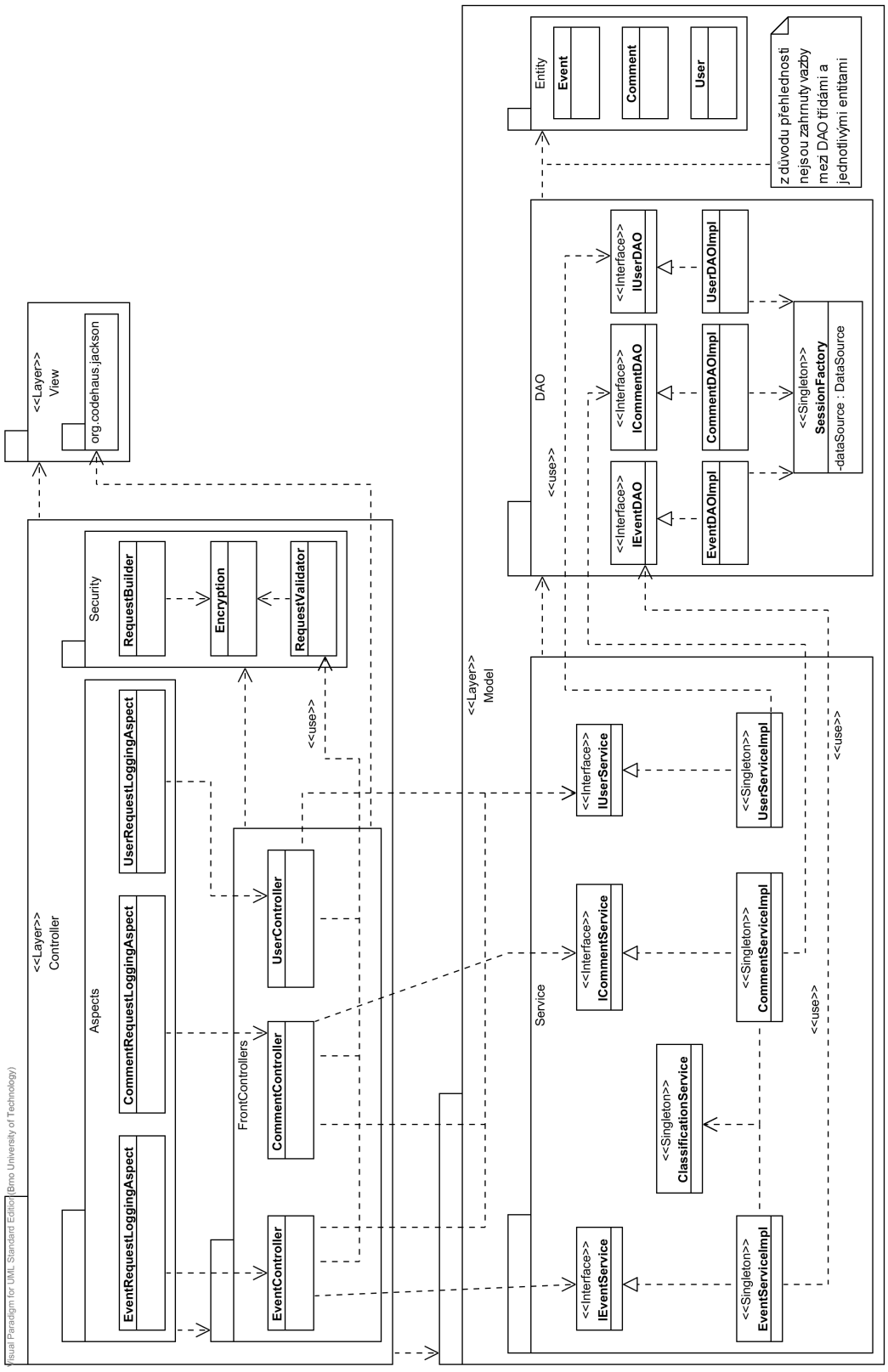
Obrázek A.3: Diagram sekvence pro případ užití *vytvořit událost* z pohledu klientské části aplikace.



Obrázek A.4: Diagram sekvence pro případ užití *vytvořit událost* z pohledu serverové části aplikace.



Obrázek A.5: Diagram tříd reprezentující architekturu klientské části aplikace.



Obrázek A.6: Diagram tříd reprezentující architekturu serverové části aplikace.

Příloha B

Měření přesnosti klasifikátorů

	pozitivně negativní	**pozitivně pozitivní**	**přesnost třídy**
negativní	109	10	91,60 %
pozitivní	11	110	90,91 %
specificita	90,83 %		
senzitivita		91,67 %	
přesnost modelu	91,25 %		
chybovost	8,75 %		
F_1 **skóre**	91,30 %		

Tabulka B.1: Přesnost modelu pro klasifikaci sentimentu událostí

	pozitivně negativní	**pozitivně pozitivní**	**přesnost třídy**
negativní	956	5	99,48 %
pozitivní	5	305	98,39 %
specificita	99,48 %		
senzitivita		98,39 %	
přesnost modelu	99,21 %		
chybovost	0,79 %		
F_1 **skóre**	98,38 %		

Tabulka B.2: Přesnost modelu pro klasifikaci kategorie *car crash*

	pozitivně negativní	**pozitivně pozitivní**	**přesnost třídy**
negativní	703	3	99,58 %
pozitivní	4	400	99,01 %
specificita	99,43 %		
senzitivita		99,26 %	
přesnost modelu	99,37 %		
chybovost	0,63 %		
F_1 **skóre**	99,13 %		

Tabulka B.3: Přesnost modelu pro klasifikaci kategorie *concert*

	pozitivně negativní	**pozitivně pozitivní**	**přesnost třídy**
negativní	1134	4	99,65 %
pozitivní	6	127	95,49 %
specificita	99,47 %		
senzitivita		96,95 %	
přesnost modelu	99,21 %		
chybovost	0,79 %		
F_1 **skóre**	96,26 %		

Tabulka B.4: Přesnost modelu pro klasifikaci kategorie *fire*

	pozitivně negativní	**pozitivně pozitivní**	**přesnost třídy**
negativní	1231	1	99,92 %
pozitivní	1	39	97,50 %
specificita	99,92 %		
senzitivita		97,50 %	
přesnost modelu	99,84 %		
chybovost	0,16 %		
F_1 **skóre**	96,89 %		

Tabulka B.5: Přesnost modelu pro klasifikaci kategorie *inspector*

	pozitivně negativní	**pozitivně pozitivní**	**přesnost třídy**
negativní	1107	4	99,64 %
pozitivní	3	157	98,12 %
specificita	99,73 %		
senzitivita		97,52 %	
přesnost modelu	99,45 %		
chybovost	0,55 %		
F_1 **skóre**	97,80 %		

Tabulka B.6: Přesnost modelu pro klasifikaci kategorie *party*

	pozitivně negativní	**pozitivně pozitivní**	**přesnost třídy**
negativní	1041	8	99,24 %
pozitivní	4	208	98,20 %
specificita	99,62 %		
senzitivita		96,46 %	
přesnost modelu	99,06 %		
chybovost	0,94 %		
F_1 **skóre**	97,28 %		

Tabulka B.7: Přesnost modelu pro klasifikaci kategorie *sport*