VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

MOBILNÍ SYSTÉM PRO ROZPOZNÁNÍ TEXTU NA IOS

MOBILE SYSTEM FOR TEXT RECOGNITION ON IOS

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — **Bc. PETR BOBÁK**
AUTHOR

VEDOUCÍ PRÁCE — **prof. Dr. Ing. PAVEL ZEMČÍK**
SUPERVISOR

BRNO 2017

Zadání diplomové práce/19682/2016/xbobak01

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav počítačové grafiky a multimédií Akademický rok 2016/2017

Zadání diplomové práce

Řešitel: **Bobák Petr, Bc.**
Obor: Počítačová grafika a multimédia
Téma: **Mobilní systém pro rozpoznání textu na iOS**
Mobile System for Text Recognition on iOS
Kategorie: Uživatelská rozhraní

Pokyny:

1. Prostudujte literaturu na téma rozpoznávání textů a tvorbu mobilních aplikací.
2. Navrhněte postup pro implementaci systému pro rozpoznání textu na mobilní platformě iOS.
3. Zhodnoťte možnosti a vlastnosti navrženého postupu a dosažitelné výsledky.
4. Implementujte systém pro rozpoznávání textu a demonstrujte funkčnost na vhodném příkladu.
5. Diskutujte dosažené výsledky a možnosti dalšího pokračování práce.

Literatura:

- Dle pokynů vedoucího.

Při obhajobě semestrální části projektu je požadováno:

- Body 1 až 3 zadání

Podrobné závazné pokyny pro vypracování diplomové práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva diplomové práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap, které byly vyřešeny v rámci dřívějších projektů (30 až 40% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Zemčík Pavel, prof. Dr. Ing.**, UPGM FIT VUT
Datum zadání: 1. listopadu 2016
Datum odevzdání: 24. května 2017

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav počítačové grafiky a multimédií
612 66 Brno, Božetěchova 2

L.S.

doc. Dr. Ing. Jan Černocký
vedoucí ústavu

Abstrakt

Tato práce dokumentuje postup vývoje moderní klient-server aplikace pro rozpoznání textu na platformě iOS. Čtenář je v úvodu seznámen s obecným principem klient-server modelu, včetně jeho známých architektur, a také s členěním logických vrstev mezi obě strany. Následuje popis současných trendů a používaných technologií vhodných pro tvorbu aplikačního rozhraní webového serveru. Dále jsou diskutovány principy a možnosti rozpoznání textu na straně serveru. V rámci klientské části práce poskytuje základní poznatky o platformě iOS a zmiňuje také některé podstatné koncepty charakteristické pro vývoj iOS aplikací. Vlastní implementace pak klade důraz na možnost obecného použití serverové části tak, aby ji bylo možné integrovat přímo s koncovým klientem, případně i s jiným aplikačním serverem třetí strany. Součástí výstupu práce je také framework pro přímou komunikaci iOS klienta se serverem. Jako příklad použití je implementována demonstrační aplikace pro vyhodnocení aditivních látek z etiket potravin.

Abstract

This thesis describes a development of a modern client-server application for text recognition on iOS platform. The reader is acquainted with common principles of a client-server model, including its known architecture styles, and with a distribution of logical layers between both sides of the model. After that the thesis depicts current trends and examples of suitable technologies for creating an application programming interface of a web server. Possible ways of text recognition on the server side are discussed as well. In context of a client side, the thesis provides an insight into iOS platform and a few important concepts in iOS application development. Following implementation of the server side is stressed to be reusable as much as possible for different kinds of use cases. Last but not least, the thesis provides a simple iOS framework for a direct communication with the recognition server. Finally, an application for evaluation of food ingredients from a packaging material is implemented as an example of usage.

Klíčová slova

Django, Django REST Framework, Google Cloud Vision, iOS, klient-server model, Microsoft Computer Vision, optické rozpoznání textu, Python, REST, Swift, Tesseract, webová aplikace, webový framework, webová služba, webový server

Keywords

Django, Django REST Framework, Google Cloud Vision, iOS, client-server model, Microsoft Computer Vision, optical character recognition, Python, REST, Swift, Tesseract, web application, web framework, web service, web server

Citace

BOBÁK, Petr. *Mobilní systém pro rozpoznání textu na iOS*. Brno, 2017. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Zemčík Pavel.

Mobilní systém pro rozpoznání textu na iOS

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana prof. Dr. Ing. Pavla Zemčíka. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Petr Bobák
17. května 2017

Poděkování

Velmi děkuji mému vedoucímu za jeho vedení, hodnotné připomínky a poznatky v průběhu řešení této práce. Dále chci poděkovat doktorandům z pracovny Q201 za jejich ochotu a pomoc při procesu zprovoznění rozpoznávacího serveru. Děkuji též mému kolegovi Janu Tomeškovi, který se v rámci své práce věnoval předzpracování obrazu pro ukázkovou aplikaci. Dále děkuji mé přítelkyni Lucii za projevenou podporu a trpělivost v průběhu vzniku této práce. Na závěr bych chtěl poděkovat mým rodičům, kteří mi s obětavostí poskytli ideální podmínky pro studium.

Obsah

Kapitola 1

Úvod

Stal se neodmyslitelnou součástí našeho každodenního života. Vstává s námi, pomáhá nám v průběhu dne a večer poslušně uléhá po našem boku. Řeč je o novodobém nejlepším příteli člověka – chytrém telefonu. To, co mobilní telefon dělá chytrým, jsou však především prozíraví vývojáři, kteří skrze mobilní aplikace řeší všední problémy průměrně smýšlejícího jedince. Ruku v ruce s rozšiřováním chytrých telefonů se ovšem ukázalo, že ne vše lze vyřešit pouze v rámci samotného zařízení.

Práce popisuje proces tvorby klient-server aplikace pro rozpoznání textu pomocí OCR. Provádí čtenáře počínaje historickými začátky spojenými s klient-server modelem přes výběr vhodných prostředků pro implementaci webové služby umožňující obecné rozpoznání textu pomocí několika různých technologií až po principy vývoje mobilních aplikací pro platformu iOS. Výsledkem je univerzální webová služba pro rozpoznání textu s REST aplikačním rozhraním a možností správy i analýzy chování serveru skrze webový prohlížeč. Výsledná služba tvoří základní stavební kámen další diplomové práce, která se podrobně zabývá předzpracováním obrazu pro OCR systémy na platformě Android[1]. Jako příklad možného užití je implementována demonstrační aplikace pro vyhodnocení složení potravin z etikety. Cílem této aplikace je podat uživateli informaci o celkové „zdravosti“ a dnes tolik diskutovaných přidatných potravinářských látkách v daném produktu.

Potenciální výhodou vzniklé webové služby, kterou jsem pracovně pojmenoval jako *Texie Cloud API*, je především jednotné rozhraní zapouzdřující několik technologií pro rozpoznání textu – *Tesseract*, *Microsoft Computer Vision* a *Google Cloud Vision*. Kromě jiného služba disponuje také moderním autentizačním systémem založeným nad standardem *OAuth 2.0*, analytickým systémem, datovým skladem pro rozpoznané snímky, interaktivním prohlížečem aplikačního rozhraní a frameworkem pro operační systém iOS.

Počáteční motivací pro volbu vlastního zadání mi byla právě představa existence výše popsané aplikace. V předchozí bakalářské práci jsem se podrobně věnoval vývoji mobilní aplikace, zabývající se hodnocením kvality pokrmů v restauracích, intenzivně komunikující se vzdáleným serverem. Tehdy jsem se zabýval čistě vývojem aplikace, ale ve skrytu duše jsem se podvědomě zajímal i o serverovou část, kterou měl na starosti jiný student. Zvědavost a vývoj v oblasti mobilních aplikací směrem ke stále se zvyšujícímu propojení s webovými a cloudovými službami, mě pak přesvědčil věnovat se serverové části, o které jsem měl do této chvíle jen částečné povědomí, i v rámci diplomové práce.

[1] Tomešek, Jan. *Mobilní systém pro rozpoznání textu na Androidu.* Brno, 2017. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Zemčík Pavel.

Kapitola 2 popisuje klient-server model, jeho dvouvrstvou a třívrstvou architekturu a prostředky pro komunikaci mezi oběma stranami. Kapitola 3 se podrobněji věnuje serverové části a softwarovým nástrojům pro implementaci webových služeb a aplikací. Kapitola 4 se zabývá postupem rozpoznání textu a existujícími nástroji pro rozpoznání. Kapitola 5 popisuje nejdůležitější principy související s vývojem iOS aplikací a představuje vybrané aplikace pro rozpoznání textu. Kapitola 6 prezentuje osobní názor na aktuální stav a dále také technickou specifikaci webové služby i demonstrační iOS aplikace. Kapitola 7 uvádí návrh a dekompozici služby na jednotlivé subsystémy a prezentuje návrh uživatelského rozhraní doprovodné webové aplikace a mobilní iOS aplikace. Kapitola 8 popisuje postup realizace, použité nástroje, knihovny a postupy. Předposlední kapitola 9 uvádí návrh testování a diskutuje výsledky provedených testů.

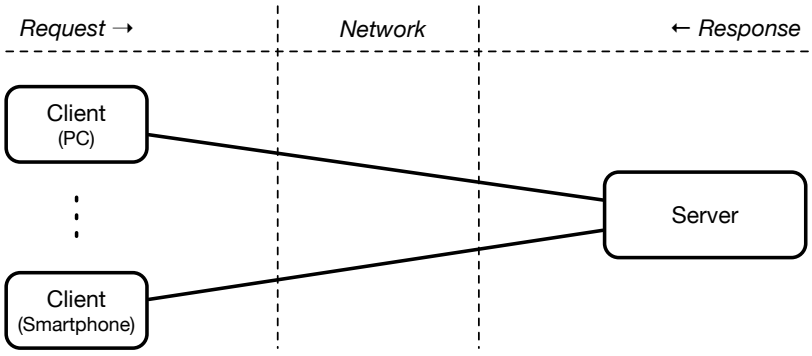
Kapitola 2

Klient-server model

Tato kapitola se zabývá významem klient-server modelu při vývoji distribuovaných aplikačních systémů. V úvodu je čtenář seznámen s obecnou definicí tohoto modelu a charakteristickými vlastnostmi, které vytváří potřebnou znalost pro pozdější realizaci klient-server systému pro rozpoznání textu. Následně jsou blíže popsány existující architektury, případy jejich užití a způsob dělení logických vrstev mezi klientskou a serverovou část. V závěru je pak zmíněna souvislost modelu s *webovou službou* a prostředky pro vzájemnou komunikaci mezi oběma stranami.

2.1 Model a jeho části

Klient-server model (někdy také *klient-server architektura*[1]) je definován jako architektonický vzor tvořený *klientem* a *serverem*, kde *klient* zasílá požadavky, zatímco *server* na zaslané požadavky reaguje [1]. Jiná definice zase popisuje klient-server model jako vztah mezi počítači, které mezi sebou vzájemně komunikují [2]. Komunikace typicky probíhá skrze síť ať už lokální, nebo globální (internet), existují však i aplikace tohoto modelu pouze v rámci jednoho počítače (tento způsob lze připodobnit k modulárnímu programování).



Obrázek 2.1: Ilustrace obecného klient-server modelu. Klient inicializuje komunikaci zasláním dotazu na server, který dotaz aplikačně specifickým způsobem zpracuje a výsledek odešle zpět klientovi.

[1] V této práci bude použito označení *klient-server model* pro odlišení od jeho možných architektur popsaných dále.

Klient je aktivní prvek modelu, neboť zahajuje komunikaci zasláním požadavku na server. To znamená, že musí znát dostupné servery a jejich poskytované služby. Klient může komunikovat pouze se serverem (případně více servery), komunikace mezi klienty navzájem není možná. Z hardwarového pohledu se jedná o osobní počítač nebo například mobilní telefon, jehož výpočetní výkon není nijak specificky definován. Ze softwarového pohledu jde typicky o program s grafickým rozhraním, jehož příkladem může být webový prohlížeč nebo emailový klient.

Server je pasivní prvek modelu, čeká na příchozí požadavek od klienta a poskytuje odpověď. Adresa serveru by měla zůstat neměnná, případně musí být jiným mechanismem zajištěna jeho dostupnost. Server může komunikovat nejen s klienty, ale i s jinými servery navzájem. Taková architektura se pak označuje jako *vícevrstvá* [1] – viz další sekce 2.2. Z hardwarového pohledu jde o speciální sálový počítač s velmi vysokým nárokem na výkon. Na rozdíl od klienta se také klade vysoký důraz na nepřetržitou dostupnost a kvalitu připojení k síti[2]. Ze softwarového pohledu jde o speciální program, který je schopný obsloužit více klientů současně – viz sekce 3.1.2 v následující kapitole.

Následuje stručný popis charakteristických vlastnosti klient-server modelu dle článku [3]:

- **Encapsulation of services** – Činnost serveru může být bez zásadních potíží aktualizována, dokud se nemění jeho komunikační rozhraní s klientem.
- **Integrity** – Data i kód je udržován centrálně, což umožňuje snadnější údržbu a schopnost dosáhnout vyšší integrity dat.
- **Message-based exchanges** – Klient a server na sebe nejsou pevně vázáni. Jejich vzájemná komunikace probíhá skrze zprávy podle protokolu, který je znám oběma stranám.
- **Scalability** – Model umožňuje *horizontální* a *vertikální* škálovatelnost služby. *Horizontální* škálovatelností se rozumí schopnost odebírat a přidávat klienty s pouze minimálním dopadem na výkonnost celé služby. *Vertikální* škálovatelnost znamená schopnost přidat nebo odebrat server, případně migrovat celou službu na výkonnější servery.
- **Modular, extensible design** – Modulární návrh klient-server modelu umožňuje vytvořit službu, která je tolerantní vůči chybovým stavům. V takovém systému může dojít k poruše, aniž by tato skutečnost ovlivnila funkčnost všech poskytovaných služeb. Další výhodou modulárního návrhu je dynamická schopnost rozšířit model o další server podle míry zatížení konkrétní služby.
- **Platform independence** – Klient-server model je v ideálním případě nezávislý na hardwaru i operačním systému. Na obou stranách se tak mohou operační systémy i použitý hardware lišit. Tento fakt umožňuje každé straně optimální využití dostupných zdrojů.

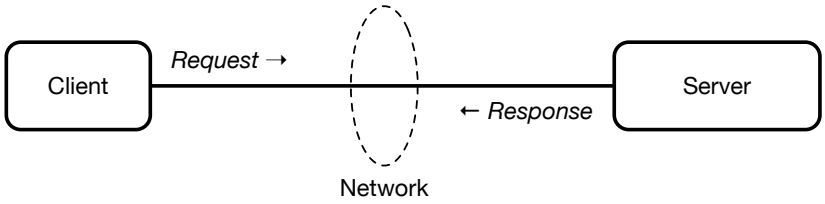
[2] Takto vysoké nároky není snadné splnit, a proto vznikla tzv. *datová centra*, což jsou speciální budovy, ve kterých je umístěno velké množství výpočetních jednotek. Provozovatel datového centra pak poskytuje pronájem výpočetního výkonu s garancí jisté dostupnosti, konektivity atd.

2.2 Architektura a logické vrstvy

Mluvíme-li o klient-server modelu, většinou máme na mysli dvouvrstvou architekturu. Existuje však i třívrstvá a obecně až n-vrstvá architektura. Z pohledu architektury se pak můžeme zabývat také kompozicí aplikace. V takovém případě nás zajímá rozdělení tří základních logických vrstev – *prezentační*, *aplikační* a *databázové* – mezi klienta a server.

2.2.1 Dvouvrstvá architektura

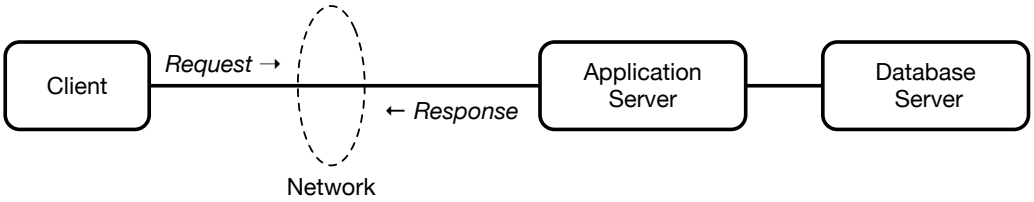
Tento druh architektury (někdy také označováno jako *plochá architektura*) zahrnuje pouze klienta a server. Klient vytváří požadavek, načež server odpovídá odpovědí, kterou získal **pouze z vlastních zdrojů** bez nutnosti další komunikace. Výhodou je jednoduchý způsob komunikace a snadná údržba. Při větším počtu dotazujících se klientů však může dojít k přetížení serveru a následně až k omezení dostupnosti hostované služby. Z tohoto důvodu je dvouvrstvá architektura vhodná především pro aplikace s menším počtem současně komunikujících klientů.



Obrázek 2.2: Příklad dvouvrstvé klient-server architektury. Klient skrze síť zasílá požadavek aplikačnímu serveru, který požadavek zpracuje a odpovídá zpět požadovaným způsobem.

2.2.2 Třívrstvá architektura

Třívrstvá architektura přidává mezi klienta a server další vrstvu (angl. middleware). Ta se většinou označuje jako *aplikační server* a následuje ji *databázový server*. Aplikační server komunikuje s klientem a na základě jeho požadavku získává zdroje z databázového serveru. N-vrstvá architektura je rozšířením třívrstvé architektury o další aplikační servery. Výhodou je menší zatížení klienta, který zastupuje pouze *prezentační* logiku aplikace. Dalším přínosem je bezpečnost a flexibilita, neboť jednotlivé části – *prezentační*, *aplikační*, *databázová* – jsou na sobě z velké části nezávislé. Třívrstvá (a obecně n-vrstvá) architektura je vhodná pro aplikace s vysokým počtem současně komunikujících klientů, jelikož umožňuje tzv. *load balancing* – rozdělení zátěže mezi více serverů.



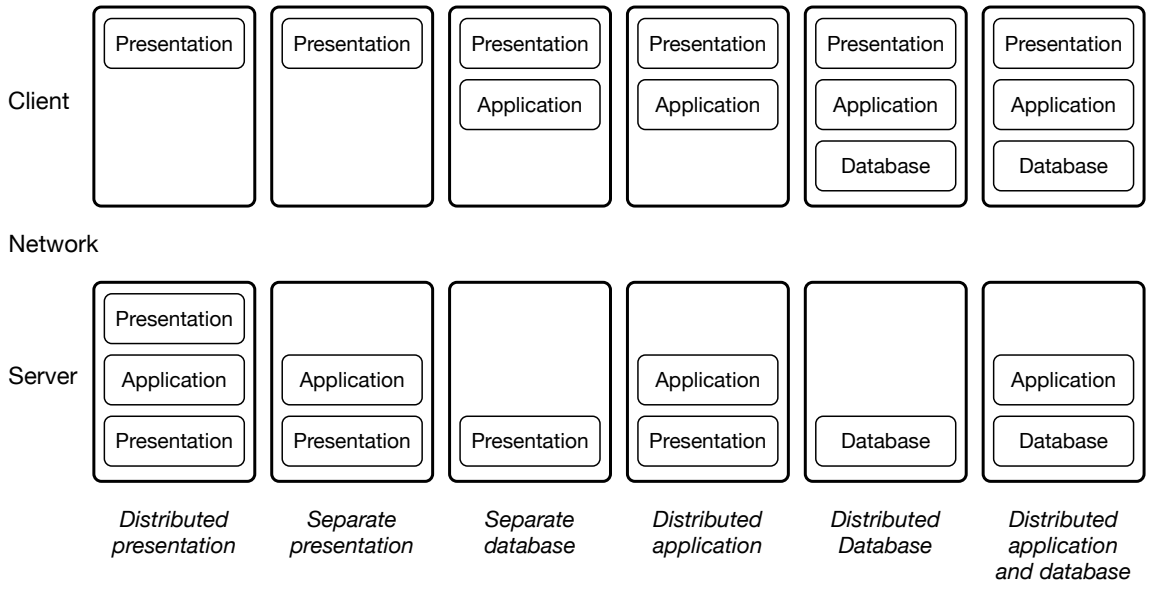
Obrázek 2.3: Ilustrace třívrstvé klient-server architektury. Klient skrze síť zasílá požadavek aplikačnímu serveru, který pro získání odpovědi dále komunikuje s databázovým serverem.

2.2.3 Rozdělení logických vrstev

U dvouvrstvé architektury se musíme zabývat také rozdělením jednotlivých logických vrstev – *prezentační*, *aplikační* a *databázové* logiky – mezi klientskou a serverovou část. U třívrstvé architektury je rozdělení dáno implicitně. V praxi však hranice jednotlivých logických vrstev nemusí být dobře rozpoznatelné, například aplikační a databázová logika může téměř splývat [4].

- **Prezentační logika** zobrazuje data a slouží jako prostředek pro komunikaci se serverem – zjednodušeně se jedná o uživatelské rozhraní.
- **Aplikační logika** se stará o řízení, potřebné výpočty a chod celé aplikace. Příkladem může být výpočet mezd ve výplatním systému nebo například interpretace dat v profesionálním tiskařském systému [5]. Tvoří prostřední vrstvu mezi prezentační a databázovou logikou aplikace, tzn. je zodpovědná za přenos a zpracování dat mezi klientem a databázovým serverem.
- **Databázová logika** zodpovídá za řízení databázového systému. Stará se o ukládání a získávání dat podle požadavků aplikační logiky.

Obrázek 2.4 zobrazuje možné způsoby dělení logických vrstev mezi klienta a server dle knihy [5]. Vynechány jsou dva extrémní příklady, kdy jsou všechny části umístěny na stejné straně dvouvrstvé architektury klient-server modelu.



Obrázek 2.4: Příklad možného dělení jednotlivých vrstev mezi klienta a server dle knihy [5]. Klienta pro tři případy vlevo nazýváme jako *tenkého klienta* – na starosti má především prezentaci dat ze serveru. Tenkého klienta můžeme, v souvislosti se sálovými počítači, připodobnit k tzv. terminálu. Klient v druhé polovině případů se nazývá *tlustý (silný) klient* – jeho činnost již nespočívá pouze v prezentaci dat, ale angažuje se také v oblasti aplikační a databázové logiky.

2.3 Komunikace mezi klientem a serverem

Rozvoj klient-server modelu přišel s rozvojem webových stránek a následně také s příchodem *webových služeb* (angl. *Web Services*).

Dle definice [6] lze webovou službu chápat jako klient-server model, který dovoluje komunikaci mezi různými systémy (jeden vystupuje v roli klienta, druhý v roli serveru) bez nutnosti použití proprietárních standardů. Jiná definice [6] označuje webovou službu jako skupinu otevřených protokolů a standardů používaných pro výměnu dat mezi aplikacemi nebo systémy. Softwarové aplikace implementované různými jazyky běžící na různých platformách mohou pro výměnu dat skrze síť použít právě webové služby. Zajímavý pohled na problematiku nabízí také článek [7], který webové služby označuje jako webové stránky pro stroje[3].

V počátcích webových služeb se pro komunikaci mezi klientem a serverem používala technologie *vzdáleného volání procedur* (RPC – Remote Procedure Call) pomocí protokolu XML-RPC. Nástupcem tohoto protokolu se stal protokol SOAP, jehož alternativou se později stala metoda vzdálené manipulace se zdroji tzv. REST.

2.3.1 SOAP

Simple Object Access Protocol je platformě nezávislý protokol pro výměnu strukturovaných dat mezi dvěma systémy, z nichž jeden vystupuje v roli klienta a druhý v roli serveru. Zprávy tohoto protokolu jsou postaveny nad značkovacím jazykem XML a jejich přenos je možné zajistit téměř jakýmkoliv transportním protokolem – nejčastěji HTTP, lze však použít také SMTP (Simple Mail Transfer Protocol) nebo XMPP (Extensible Messaging and Presence Protocol). [8]

Protokol poskytuje dva druhy zpráv – dokumentové nebo RPC. Druhý zmíněný typ dovoluje volat vzdálené procedury prostřednictvím jazyka pro popis webových služeb WSDL (Web Services Description Language), který je opět založený na jazyku XML. Struktura SOAP zprávy je tvořena následujícími XML elementy – obálkou, hlavičkou, tělem zprávy a nepovinným chybovým elementem.

Následuje stručné shrnutí hlavních charakteristik protokolu SOAP vycházejících z diplomové práce [7] a knihy [8]:

- **Independence** – SOAP je ve všech ohledech značně nezávislý protokol, který může být použit z různých programovacích jazyků na různých platformách s různými operačními systémy. Jeho nezávislost, však přináší i nevýhody z nichž nejpodstatnější je různá velikost a vzájemná nekompatibilita datových typů na odlišných platformách.
- **XML-Based** – Protokol je postaven nad jazykem XML, což přináší všechny jeho výhody jako snadné strojové zpracování, ale i nevýhody – typicky jde o přílišnou vláčnost a rozsáhlost výsledné XML struktury. Tyto problémy se ve výsledku nepříznivě projevují na velikosti přenášených zpráv, čehož dalším důsledkem je pomalejší odezva webové služby.
- **Complexity** – Přestože protokol nese ve svém názvu slovo „simple“ a bývá prezentován jako jednoduchý protokol pro komunikaci, je v mnoha ohledech přespříliš komplikovaný a zbytečně složitý.

[3] Stroj nemůže správně zpracovat webovou stránku, protože HTML značky neposkytují dostatek sémantických informací o svých hodnotách.

2.3.2 REST

Representational State Transfer není protokolem ale architektonickým stylem, který poskytuje principy pro návrh distribuovaných volně vázaných služeb. Není pevně spjat s žádnou konkrétní technologií ani platformou. Principy RESTu definoval Roy Fielding ve své dizertační práci *Architectural Styles and the Design of Network-based Software Architectures* [9]. Autor se podílel i na vývoji protokolu HTTP a současně také projektu Apache.

REST je založen nad klient-server modelem, kde jsou požadavky a odpovědi přenášeny jako reprezentace zdrojů. *Zdroj* je základním stavebním prvkem Webu identifikovaný pomocí adresy URI[4], např. `http://api.example.com/vehicles`. *Reprezentací* je myšlen abstraktní dokument, který zachycuje (reprezentuje) aktuální stav zdroje. Příkladem zdroje může být například kolekce struktur se složkami `značka` a `model` popisující vozidlo. Reprezentace takového zdroje pak může být soubor obsahující tutéž kolekci struktur se stejnými složkami `značka` a `model`, avšak popsanou například pomocí specifikace JSON. Stejně tak se může jednat o reprezentaci pomocí XML nebo HTML souboru.

Komunikace mezi klientem a serverem je v rámci webových služeb s architekturou REST nejčastěji zajištěna pomocí protokolu HTTP ve verzi 1.1. *RESTful* webovou službou je označována taková služba, která využívá HTTP dotazů jak pro *zasílání* dat (create/update), tak i pro *čtení* (read) a *mazání* dat (delete) na serveru. Tyto základní operace jsou označovány angl. zkratkou CRUD. Komunikační protokol HTTP definuje několik metod pro komunikaci se serverem, z nichž jen některé jsou použity pro implementaci CRUD operací v rámci RESTu. Mapování jednotlivých metod na operace včetně jejich významu je popsáno v následující tabulce 2.1.

Operace	**Metoda**	**Význam**
Create	POST	Vytvoří zdroj na serveru.
Retrieve	GET	Získá zdroj ze serveru.
Update	PUT	Změní stav zdroje nebo aktualizuje jeho hodnotu.
Delete	DELETE	Odstraní zdroj ze serveru.

Tabulka 2.1: Mapování metod HTTP protokolu na CRUD operace.

Následuje bližší popis hlavních principů REST architektury dle Fieldingovi dizertační práce [9] a informačního dokumentu CA Technologies [11]:

- **Client-Server** – Webová služba poskytuje minimálně jednu nebo více operací, přičemž server čeká na požadavek od klient specifikující žádanou operaci.
- **Stateless** – Stav komunikace mezi klientem a serverem nesmí být uchováván na straně serveru. Veškeré potřebné stavové informace uchovává klient a jsou-li potřebné pro vzájemnou komunikaci, přenáší se jako součást zprávy mezi klientem a serverem.
- **Cacheable** – Odpovědi ze strany serveru musí být jasně označeny, zdali je možné využít jejich uložení do vyrovnávací paměti či nikoliv. Správně nastavená *cache politika* umožňuje v některých případech zcela eliminovat komunikaci mezi klientem a serverem, čímž je možné dosáhnout menší přenosové náročnosti a menšího zatížení serveru.

[4]Uniform Resource Identifier – dle definice RFC 3986 [10] se jedná o textový řetězec pro identifikaci abstraktních nebo fyzických zdrojů.

- **Uniform interface** – Jedná se o jeden z hlavních principů RESTu, který přináší zjednodušení celé architektury a umožňuje nezávislý vývoj klientské a serverové části. Zároveň tento princip nabádá všechny poskytovatele RESTful webových služeb, aby dodržovali obvyklé zvyklosti a principy popsané v celé této části.
- **Layered system** – Klient nesmí být schopen rozpoznat, jestli komunikuje přímo s koncovým serverem nebo serverem na bližší vrstvě n-vrstvé klient-sever architektury.
- **Code on demand** – Umožňuje rozšířit nebo upravit logiku klienta (například webový prohlížeč) ze strany serveru přenosem spustitelného kódu ve formě appletu nebo skriptu. To přináší zjednodušení klientské části, která může následně implementovat pouze nezbytné vlastnosti. Nevýhodou je ztráta přehlednosti (kde je jaká část implementována), a proto je tento princip RESTu nepovinný.

V současné době je REST považován za lepší alternativu k protokolu SOAP [12]. Dle dalšího článku [13] REST dnes již zcela nahradil příliš komplikovaný a zbytečně složitý protokol SOAP. Hlavním argumentem architektonického stylu je především jednoduchost a menší přenosová náročnost. Například velikost zasílaných zpráv je oproti webovým službám založených na protokolu SOAP 9× až 10× menší [12]. Podobně je tomu i s potřebnou dobou pro přenos a zpracování zprávy – vůči protokolu SOAP potřebuje REST 5× až 6× méně času [12].

Kapitola 3

Webový server, aplikace a služby

Tato kapitola se podrobněji věnuje straně serveru – předně vysvětluje pojem *webový server*, následně se pokouší objasnit významový rozdíl mezi *webovou aplikací* a *webovou službou*. Dále je uveden význam *webového frameworku* a bližší popis vybraných zástupců – Django a Ruby on Rails. V závěru kapitoly jsou rozebrány nově vznikající frameworky vybudovány nad jazykem Swift – Kitura, Vapor a Perfect.

3.1 Webový server

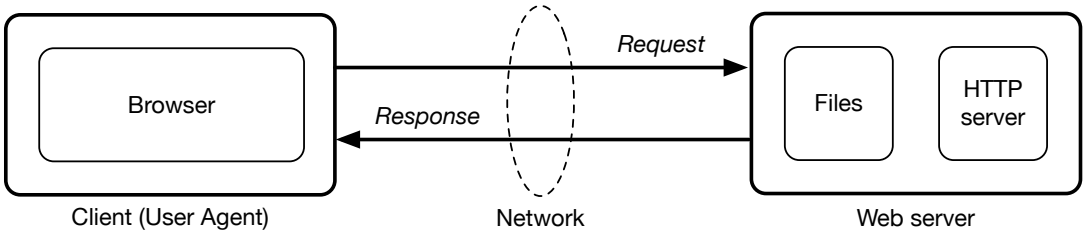
Pojem *webový server* (nebo jen web server) lze chápat dvěma různými způsoby. Může se jednat buď o hardware (např. server), na kterém je webová služba hostována, nebo o specifický software, který je na serveru spuštěn. Někdy je možné setkat se také s výkladem, který pod tímto pojmem souhrnně označuje obě jmenované části a zdůrazňuje jejich vzájemnou spolupráci. [14, 15]

První webový server (hardware i software) byl vytvořen roku 1990 sirem T. J. Berners-Lee, zaměstnancem Evropské organizace pro jaderný výzkum (CERN), jako součást nově vznikajícího projektu WorldWideWeb. Hardware web serveru tvořil počítač NeXT Cube od společnosti NeXT Computer. Software, později označovaný jako CERN httpd, běžel na operačním systému NeXTSTEP od stejnojmenné společnosti.

3.1.1 Hardware

Z hardwarového pohledu se jedná o specifické zařízení, které uchovává potřebné soubory (např. HTML dokumenty, CSS stylopisy, obrázky, serverové skripty) patřící ke konkrétní webové stránce, aplikaci či službě. Laicky řečeno jde většinou o velmi výkonné zařízení, se kterým komunikuje webový prohlížeč (klient) při prohlížení webových stránek. Pro webový server jsou v tomto smyslu charakteristické následující body [14, 15]:

- neustálá a velmi stabilní síťová konektivita,
- statická IP adresa ve většině případů dostupná skrze doménové jméno,
- přítomnost výkonného procesoru a dostatečného množství rychlé operační paměti.



Obrázek 3.1: Obecný příklad komunikace klienta s web serverem [15]. Ve své podstatě se jedná o rozšířený pohled na klient-server model. Pro snazší představu je klienta možné chápat jako webový prohlížeč a web server jako výkonný počítač s odpovídajícím softwarovým vybavením.

3.1.2 Software

Web server ve významu softwaru slouží k přenosu souborů a dokumentů popsaných v předchozí podsekci 3.1.1 na základě požadavků od klienta. Přenos informací či souborů je nejčastěji zajištěn protokolem HTTP. V takovém případě můžeme dle článku [15] server označit jako *HTTP server*. V mnoha případech však takto specifikovaný software poskytuje podporu i pro další protokoly. Typicky se jedná o protokol pro zabezpečený přenos HTTPS nebo protokol pro přenos souborů FTP a další. Charakteristické vlastnosti web serveru ve smyslu softwaru jsou následující [12, 16]:

- možnost hostování více webových služeb v rámci jednoho serveru,
- autentizace uživatelů, šifrování dat a přístup k databázi,
- možnost použití skriptovacích jazyků, podpora CGI a FastCGI,
- logování informací o provozu a stavu serveru,
- možnost nastavit maximální počet souběžných přístupů.

Nejznámějším a nejpoužívanějším softwarem pro web server je *Apache*. Jedná se o multiplatformní open-source řešení dostupné pod vlastní Apache licencí[1]. Zajímavou vlastností je možnost rozšíření základního jádra o další funkcionalitu pomocí modulů (označovány jsou prefixem `mod_` a názvem požadované vlastnosti, např. `mod_rewrite`). Jako další zástupce v pořadí s největším podílem na trhu můžeme jmenovat *nginx*, *IIS* od Microsoftu a *GWS* od Google. [17]

Při vývoji webových aplikací je možné setkat se také se zkratkou *LAMP* – Linux (operační systém), Apache (web server), MySQL (databáze), PHP (skriptovací jazyk) [18]. Tato zkratka popisuje zažité a rozšířené vrstvení jednotlivých technologií potřebných pro vývoj webových aplikací.

3.2 Webové aplikace a služby

Hlavním rozdílem mezi *webovou aplikací* a dříve zmíněnou *webovou službou* je především cílový konzument dat. V případě *webových služeb* je příjemce dat program, a tudíž je tomuto

[1] Apache licence na rozdíl od GPL (GNU Public License), nevyžaduje zveřejňovat změny provedené v chráněném díle.

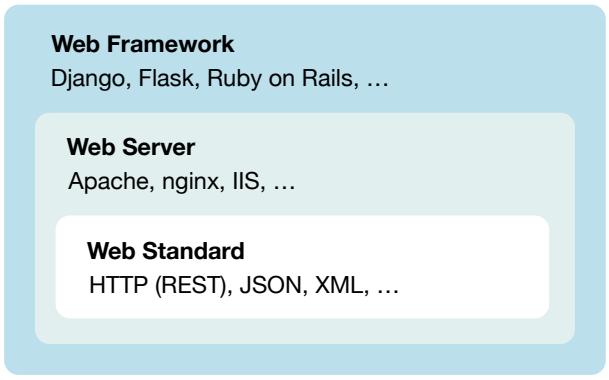
podřízen také formát těchto dat. Ve většině případů jde o strojově snadno čitelné formáty, např. JSON nebo XML. Webové služby tedy slouží primárně pro komunikaci a interakci mezi různými zařízeními skrze síť. Naopak *webová aplikace* je určena pro konzumaci dat lidmi. Typickým příkladem je například formát HTML, který prohlížeč interpretuje do snadno čitelné podoby. Webovou službu je však možné chápat také jako webovou aplikaci, ale pouze v případě, že existuje program, který poskytovaná data interpretuje do člověkem čitelné podoby.

V druhé polovině 90. let se pro vývoj webových aplikací a služeb začal hojně používat standard CGI (*Common Gateway Interface*). Ten umožňoval definovat ke konkrétní URL adrese tzv. obslužný *CGI skript*, který dovoloval generovat dynamický HTML kód pomocí různých jazyků (Perl, Python, C, C++, Java). Tehdejší programátor typické webové aplikace postavené nad CGI měl však vše zcela pod vlastní kontrolou ať už se jednalo o komunikaci s databázovou vrstvou, bezpečnost nebo například autentizaci uživatelů [18]. Z toho důvodu se mnoho základních vlastností webové aplikace implementovalo stále dokola, a proto se postupně začaly objevovat znovupoužitelné knihovny. Postupný trend příliš neovlivnil ani příchod jazyka PHP, který zmíněné problémy nedokázal (a pravděpodobně ani nemohl) vyřešit [18]. Knihovny postupně pokrývaly stále více oblastí, až se začaly označovat jako *webové frameworky*.

3.2.1 Web framework

Webový framework označuje skupinu nástrojů, knihoven, programovacích prostředí a návrhových architektur[2], které usnadňují běžné a často se opakující činnosti programátora při vývoji webových služeb či aplikací. Umožňuje zaměřit se na logiku vyvíjené aplikace/služby, aniž by bylo nutné zabývat se nízkoúrovňovými detaily jako je řízení procesů, interpretací dotazů či vytváření odpovědí. Dovoluje snadněji definovat a interpretovat URL adresu a výrazně ulehčuje práci s pokročilejšími koncepty jako *sessions* či *cookies*. V neposlední řadě může framework poskytovat mechanismy pro autentizaci uživatele webové aplikace/služby, případně pro zabezpečení proti vnějším útokům. [19]

Dále následuje nástin dvou základních charakteristických vlastností z obrázku 3.3, které vedou k celkově rychlejšímu vývoji a snadnější údržbě webových aplikací a služeb.



Obrázek 3.2: Ilustrace vztahů mezi web frameworkem, web serverem a standardy [19]. Webový framework musí poskytnout postupy, nástroje a infrastrukturu pro snadnou správu a nasazení na daný webový server (Apache, nginx, IIS, ...).

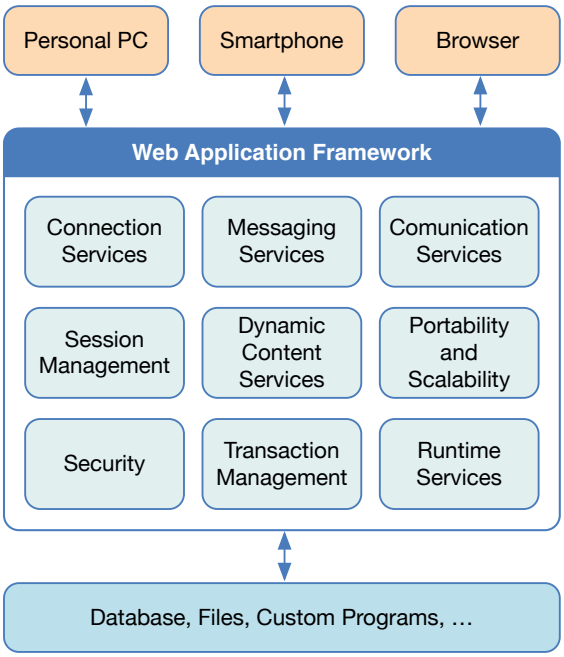
[2]Většina dnes používaných frameworků využívá architekturu *Model-View-Controller* (MVC).

Šablony

Součástí mnoha webových frameworků je dnes i podpora pro dynamické generování odpovědí na požadavek pomocí *šablon* (angl. templates). Šablonou je myšlen HTML dokument (nebo dokument vytvořený jiným značkovacím jazykem) doplněný o meta značky umožňující vložení dynamických dat. Značky mohou vkládat obsah, sloužit jako řídicí struktury (např. podmíněný příkaz, cyklus) nebo mohou mít také význam obecných proměnných [20]. Interpretaci šablony zajišťuje *renderer*, který rozvine a vyhodnotí všechny značky, čímž dojde k dynamickému vytvoření finálního dokumentu. Hlavním cílem použití šablon je tedy schopnost oddělit prezentační vrstvu aplikace od prezentovaných dat.

Objektově relační mapování

Velmi důležitou vlastností webových frameworků je také možnost perzistentního ukládání dat, s čímž souvisí konverze dat mezi většinou objektově orientovaným programovacím jazykem a relační databází (*Object Relational Mapper* zkráceně ORM). Jedná se o programovací techniku, která umožňuje atributy entitní množiny reprezentovat pomocí instančních proměnných a dotazy specifikovat jako třídní metody. Ve výsledku tato technika vývojáře zcela odstiňuje od použitého systému řízení báze dat (SŘBD). Nasazení ORM se dle knihy [20] také pozitivně projevuje na menším počtu SQL útoků, způsobených například metodou *SQL injection*[3], ve webových aplikacích.



Obrázek 3.3: Obecná struktura webového frameworku včetně komunikace s dalšími vrstvami [19]. Hlavním cílem je dosažení jednotné a komplexní infrastruktury pro snadný vývoj webových aplikací.

[3] Útočník do nezabezpečeného webového formuláře zapíše škodlivý obsah obsahující SQL kód, který může být po odeslání interpretován systémem řízení báze dat.

3.3 Vybrané webové frameworky

Postupný rozvoj a stále větší dostupnost internetu vedla kolem roku 2005 ke vzniku několika v dnešní době dobře známých frameworků, jako je *Django, Ruby on Rails, CakePHP, Symphony, Zend* a dalších. Je až s podivem, že všechny jmenované příklady jsou v době vzniku této práce (přelom roku 2016/2017) stále aktivně vyvíjeny a co víc, přibylo k nim minimálně 100 dalších [21]. Aktuálně si lze vybrat z široké palety různě komplexních frameworků určených pro nejrůznější jazyky od C++ přes Python až po Swift. Poslední zmíněný jazyk, používaný společností Apple, si po uvedení do open-source stavu získal velkou oblibu a začala kolem něj vznikat rozsáhlá komunita v čele s IBM. Postupem času byly představeny frameworky *Kitura*, *Vapor*, *Perfect*, které vynikají výborným výkonem a malou spotřebou operační paměti.

Další část této sekce se zaměřuje na dva hlavní open-source frameworky *Django*, *Ruby on Rails* a na rostoucí skupinu moderních frameworků založených nad jazykem *Swift*. Nejprve ale následuje popis sledovaných kritérií, podle kterých jsou frameworky zkoumány. Podrobné srovnání je možné nalézt ve článcích [22, 23].

- programovací jazyk
- charakteristické principy
- návrhová architektura
- škálovatelnost aplikace
- dostupnost ORM
- použití šablon
- velikost komunity[4]
- příklady použití

3.3.1 Django

Django je podle oficiálních stránek projektu [24] vysokoúrovňový *full-stack*[5] webový framework pro Python, který umožňuje rychlý vývoj a čistý pragmatický design. Django staví na těchto základních principech [20]:

- **Pythonic** – Django se snaží k jazyku Python přistupovat tak, jak byl původně navržen nebo tak, jak je obecným zvykem jej používat. Jde především o vhodné použití jak syntaktických konvencí, tak o sémantické zvyklosti a datové struktury.
- **Don't Repeat Yourself (DRY)** – Pokud se tatáž informace nachází na dvou místech a je třeba ji změnit, znamená to pro programátora dvakrát více práce a ztratí dvakrát více času.
- **Loose Coupling and Flexibility** – Přestože je Django *full-stack* framework, je možné z něj použít jen to, co je opravdu zapotřebí. Zároveň lze některé základní součásti (systém šablon či ORM) vyměnit za vlastní nebo za součásti poskytnuté vývojáři třetích stran.
- **Share Nothing Architecture** – Aplikaci vyvinutou pomocí Django frameworku je možné škálovat na všech úrovních – na úrovní databázových, vyrovnávacích i aplikačních serverů.

[4] Přibližnou představu o velikosti komunity může poskytnout portál `stackoverflow.com` podle počtu příspěvků, které jsou označeny tagem daného frameworku.

[5] *Full-stack web framework* označuje takový framework, který se stará o všechny aspekty vývoje webové aplikace – od uživatelského rozhraní až po způsob uložení dat.

Stejně jako většina webových frameworků, tak i Django používá architekturu *Model-View-Controller*. V tomto případě se však názvosloví jednotlivých částí architektury označuje jinými výrazy. *View* není chápáno jako prostředek pro prezentaci dat, ale spíše jako selektor dat, které mají být prezentovány. *View* tedy v obecném smyslu architektury MVC můžeme chápat jako *Controller*. O to, jak mají být data zobrazena, se stará *Template* (šablona), která tedy v obecném významu nahrazuje *View*. Význam *Modelu* zůstal zachován. Na základě předcházejícího vysvětlení se Django může označovat jako *Model-Template-View* (MTV) framework. [24]

Django proti svým konkurentům poskytuje také speciální *Django REST framework* (DRF), který slouží pro vytváření REST aplikačních rozhraní. Zajímavou a ojedinělou vlastností DRF pro vývojáře je možnost vyzkoušet aplikační rozhraní přímo ve webovém prohlížeči – není tedy nutné používat konzolové nástroje jako `curl` nebo `httpie`. Dále DRF podporuje autentizační politiky, serializaci dat a mnoho dalších funkcionalit typických pro vývoj aplikačního rozhraní webových služeb. [25]

Počet příspěvků na portálu StackOverflow v době vzniku této práce činil cca 133 tisíc a v současnosti jej používají webové aplikace či služby jako Instagram, Pinterest, Mozzila nebo National Geographic.

3.3.2 Ruby on Rails

Ruby on Rails (zkráceně Rails nebo RoR) je dle autorů [26] *full-stack* webový framework pro Ruby, navržený přesně podle potřeb webových vývojářů tak, aby mohli začít okamžitě vyvíjet. Tak jako Django staví nad principem DRY a dalších [26, 27]:

- **Convention over Configuration** – Jedná se o přístup, který se snaží snížit množství možných nastavení (např. skrze konfigurační soubor) pomocí vhodných výchozích hodnot. Vlastní možnost nastavení je zachována, není ale vyžadována.
- **Rails is Opinionated** – Často se opakující postupy při řešení webových aplikací mají v Rails své jasné konvence. Pokud se tyto konvence při vývoji respektují, je možné dosáhnout rychlejšího vývoje, hladší spolupráce mezi vývojáři a snadnější údržby. Opakem je *non-opinionated* přístup, ve kterém neexistují žádné konvence a každý postup vedoucí k cíli je možné označit za správný.

Vývoj webových aplikací pro Rails – stejně jako u většiny frameworku – probíhá podle MVC architektury. Objektově relační mapování (ORM) není na rozdíl od Djanga implementováno frameworkem samotným, ale je možné využít knihoven třetích stran. Výchozí implementací je ActiveRecord, kterou je případně možné zaměnit za DataMapper. O škálovatelnosti webových aplikací postavených nad Rails se vedou vášnivé diskuze – viz dokument [28]. Obecně je možné konstatovat, že dosažení škálovatelnosti není tak jednoduché jako u jiných konkurentů, lze jí ale dosáhnout vhodnou architekturou aplikace. Šablony jsou podporovány přímo jazykem Ruby (*ERB – Embedded RuBy*). Existují ovšem i další implementace, z nichž nejpopulárnější je *Erubis*, dále pak *Tilt*, *Haml*, případně *Slim* [29].

Počet příspěvků na portálu StackOverflow v době vzniku této práce činil cca 263 tisíc a v současnosti jej používají webové aplikace či služby jako Basecamp, GitHub, Shopify, Airbnb, Twitch, SoundCloud nebo Hulu.

3.3.3 Swift na straně serveru

Kitura, Vapor, Perfect jsou hlavní zástupci nově vznikající skupiny webových frameworků postavených nad jazykem Swift. *Kuturu* (od společnosti IBM) autoři označují jako nový modulární web framework, který umožňuje začít vyvíjet za méně než 3 minuty včetně instalace [30]. *Vapor* je jeho autory označován jako budoucnost tvorby webových aplikací, který je již nyní vybaven všemi podstatnými komponentami pro úspěšný vývoj [31]. *Perfect* je autory popisován jako web server a skupina nástrojů pro vývojáře, kteří chtějí použít stejný jazyk na straně klienta i serveru [32]. Obecné principy těchto frameworků ještě nejsou kompletně ustáleny, a proto následuje alespoň soupis klíčových vlastností [30, 31, 32].

- **One Development Language** – Jeden jazyk pro vývoj klienta, resp. serveru je velkou výhodou pro korporátní společnosti (což dokládá i rozsáhlá podpora IBM) i pro malé začínající vývojáře. Swift je totiž vhodný jako první jazyk pro výuku principů programování – student se tak může v rámci jediného jazyka snadno seznámit s principy programování jak pro klienta, tak i pro server.
- **No More Double Vision** – Tento bod úzce souvisí s předcházejícím. Díky použití jednoho jazyka, je možné sdílet stejné implementační prostředky mezi vývojáři klienta i serveru, což ve výsledku šetří čas a peníze.
- **Full Debugging Support** – Na platformě macOS je možné použít stejné nástroje, na které je programátor zvyklý z vývoje klientských aplikací. Ladění pomocí nástroje *Xcode* umožňuje pokročilé vlastnosti (krokování, stav instančních proměnných v daném čase) nevídané pro vývoj webových aplikací.
- **Swift Package Manager** – Všechny jmenované frameworky jsou koncipovány jako vysoce modulární. Pro správu a distribuci modulů (balíčků) je použit nový systém *Swift Package Manager*, který je součástí aktuální verze Swift 3.1.
- **Ready for the Cloud** – Webové aplikace je možné distribuovat přes stále rostoucí množství cloudových služeb jako je *Heroku, Digital Ocean a Amazon Web Services*. Pro *Kituru* je primárně určena služba *Bluemix* od společnosti IBM, která pro distribuci webové aplikace nabízí dokonce doplňující macOS aplikaci (IBM Cloud Tools for Swift).

Šablony je možné použít u všech jmenovaných frameworků, u většiny je dokonce možný výběr z více variant[6]. Objektově relační mapování (ORM) je dostupné zatím pouze pro *Vapor* (Fluent) a *Perfect* (StORM). *Kitura* prozatím poskytuje pouze moduly s aplikačním rozhraním pro přístup k většině známých systémů pro řízení báze dat. Návrhová architektura v případě *Kitury* není jasně specifikována, v ostatních případech se však jedná o MVC. Škálovatelnost webových aplikací těchto frameworků je zajištěna systémem *Grand Central Dispatch* (zkráceně GCD), který je znám z platforem macOS a iOS.

Kitura, Vapor, Perfect vynikají svou rychlostí (až 100× vyšší oproti konkurenci) a nízkou paměťovou náročností – viz srovnání [33]. Ve většině případů (kromě *Kitury*) se jedná stále o komunitní projekty, které mají zanedbatelný počet příspěvků na portálu StackOverflow. Jejich úspěch je možné zaznamenat a změřit pomocí služby GitHub, kde je označilo od 10 tisíc (*Perfect*) do 5 tisíc (*Kitura*) uživatelů značkou „to se mi líbí“.

[6] Společným šablonovým systémem dostupným napříč všemi je *Mustache*.

Kapitola 4

Rozpoznávání textu

V brzkých dobách prvních počítačů byl vstup strojem čitelných dat zajištěn pomocí děrných štítků, později pomocí periferních zařízení, např. klávesnice. V bankovním sektoru se v této době hojně uplatňovaly šeky, jejichž manuální zpracování bylo velmi pomalé a příliš náchylné k chybám. S řešením přišli v polovině 50. let vědci ze Stanford Research Institute, kteří vymysleli metodu MICR (Magnetic Ink Character Recognition). Font znaků byl navržen tak, aby při strojovém čtení každý znak vytištěný pomocí speciálního magnetického inkoustu vyvolal specifické vlnění. [34]

Optical Character Recognition (zkráceně OCR) je další metodou pro strojové zpracování dokumentů. Český překlad – optické rozpoznání textu – napovídá, že se jedná o rozpoznávání především tištěného textu optickou analýzou záznamu daného dokumentu. Záznam je obvykle vytvořen pomocí skeneru nebo fotoaparátu. Z pohledu skupiny metod pro zpracování obrazu se jedná o rozpoznání vzorů. Oproti předchozí metodě MICR přináší OCR podstatnou výhodu – nevyžaduje kontrolu nad způsobem vzniku dokumentu. [34]

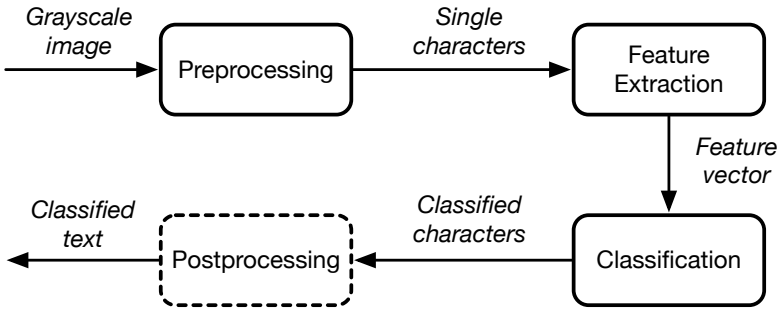
Vývoj metody OCR lze rozdělit do několika generací. V první generaci se jednalo o samostatná hardwarová zařízení, která dokázala zpracovávat dokumenty pouze s jedním přísně specifikovaným fontem, jenž byl pro člověka obtížně čitelný. Druhá generace přinesla standardizaci fontů pro rozpoznávání, které již byly bezproblémově čitelné člověkem i strojem. Další generace přinesly především zlepšení úspěšnosti rozpoznání a přeměnu těžkopádných specializovaných zařízení na efektivní softwarové nástroje. [34]

Dnešní OCR software si již dokáže poradit s širším množstvím běžně používaných fontů, přičemž úspěšnost rozpoznání se pohybuje mezi 71 % až 98 % [35]. Metoda pro dosažení 100% úspěšnosti ovšem stále nebyla nalezena, i přestože se vědci rozpoznáváním textu zabývají již téměř jedno století.

Následuje popis jednotlivých částí obecného systému pro rozpoznání textu, který současně ilustruje i obrázek 4.2. Poslední fáze *postprocessing* není nutně potřebná, a tudíž není v dalším výkladu uvedena. V závěru kapitoly jsou pak zmíněny vybrané prostředky pro rozpoznání textu.

4.1 Předzpracování

Snímek získaný akvizicí pomocí fotoaparátu je pro rozpoznání nevhodný, a je tudíž vhodné aplikovat na něj metody předzpracování. V první fázi je obraz převeden do *binární reprezentace* prostřednictvím adaptivního prahování, které se lépe vypořádá s nekonzistentním osvětlením, jež je typické u snímků zachycených fotoaparátem.



Obrázek 4.1: Fáze obecného systému pro rozpoznání textu [36].

V následující fázi je provedena *segmentace textu*, přičemž jsou izolovány jednotlivé symboly. V závislosti na rozlišení snímače[1] se mohou projevit některé defekty (např. deformace znaků, kaňky inkoustu, rozdíly v nasvětlení), které mohou výrazně snížit úspěšnost rozpoznání. Defektní snímky lze opravit například aplikací morfologických operací, případně normalizací rotace pomocí Houghovy transformace. [34, 36]

Poslední fáze, *převod znaku do jiné reprezentace*, je nepovinná a uplatní se pouze v případech, kdy to vyžaduje zvolená metoda extrakce příznaků. Příkladem alternativní reprezentace je kostra či kontura symbolu. [36]

4.2 Extrakce příznaků

Dle článku [36] je výběr metody extrakce příznaků jedním z nejdůležitějších kroků, který podstatně ovlivňuje celkovou úspěšnost rozpoznání. Jedná se o zachycení skupiny charakteristických vlastností, které je možné použít pro klasifikaci jednotlivých symbolů dané abecedy a které jsou co nejvíce invariantní vzhledem k použitému fontu. Ideální příznaky by tak měly mít co nejnižší *vnitro-třídní variabilitu* (rozdíly mezi stejným symbolem reprezentovaným různými fonty) a vysokou *mezi-třídní variabilitu* (rozdíly mezi odlišnými symboly). Následuje popis několika metod extrakce příznaků dle článků [34, 36].

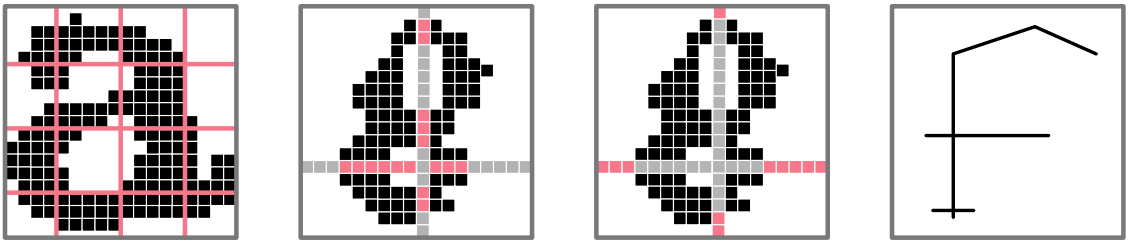
Template matching Jedná se o korelační metodu, která porovnává bod po bodu rastr vstupního symbolu s rastrem šablon všech rozpoznávaných symbolů. Na rozdíl od dále zmíněných metod v tomto případě fakticky neprobíhá žádná extrakce příznaků. Nevýhodou této metody je špatná adaptace vzhledem k rotaci jednotlivých symbolů a vysoká citlivost na grafické rozdíly použitých fontů.

Zoning V tomto případě je extrakce příznaků založena na statistické distribuci bodů (pixelů) v různých zónách obalového tělesa symbolu. Jednotlivé zóny mohou být navzájem překrývající i nepřekrývající se oblasti, přičemž hustota černých bodů v jednotlivých zónách určuje vektor příznaků.

Crossings and distances Jde o metodu nalezení počtu průsečíků přímek v různě definovaných směrech s rastrovou nebo i jinou reprezentací rozpoznávaného symbolu. V případě modifikace *crossing* se počítá počet bodů při přechodu z pozadí do popředí a u modifikace *distance* se počítá vzdálenost rastru symbolu od okrajů obalového tělesa. Obecně je tato metoda charakteristická vysokou rychlostí a nízkou složitostí algoritmu.

[1] Rozlišení by se mělo pohybovat nad dolní hranicí 200 dpi (pixelů na palec).

Structural analysis Jedná se o postup patřící mezi nejsložitější metody pro extrakci příznaků, které v tomto případě nejsou reprezentovány numerickými hodnotami, nýbrž pomocí průsečíků, tahů, koncových bodů aj. Spočívá v extrakci geometrických a topologických struktur rozpoznávaného symbolu. Tato metoda poskytuje velké množství charakteristik, které jsou odolné vůči šumu a různým fontům téhož symbolu. Složitost algoritmu, pouze malá tolerance vůči rotaci či translaci symbolů a stále probíhající vývoj patří k hlavním nevýhodám této metody.



Obrázek 4.2: Vlevo metoda *Zoning*, dále pak *Crossing*, *Distance* a *Structural analysis* [34]

4.3 Klasifikace

Klasifikace je proces určení příslušnosti symbolů ke správné třídě na základě získaných příznaků. V současné době existuje široká škála klasifikátorů, které je pro rozpoznání symbolů možné použít.

Jedním z nejjednodušších přístupů je klasifikace založená na *minimální vzdálenosti*, např. pomocí Euklidovy vzdálenosti – každá třída je charakterizována jedním referenčním vektorem, případně K Nearest Neighbor (kNN) – každá třída je charakterizována množinou referenčních vektorů. Další skupina klasifikátorů je založena na *pravděpodobnostních výpočtech*. Příkladem může být metoda Naive-Bayes. V neposlední řadě lze použít metody založené na *neuronových sítích* a dále jmenovitě *AdaBoost* nebo *Support Vector Machine*.

Výše popsané metody jsou vhodné pro příznaky, které je možné reprezentovat numerickou hodnotou, případně vektorem hodnot. Pro klasifikaci příznaků vzniklých pomocí metody *Structural analysis* se využívá vztahů mezi strukturálními komponentami, které bývají často formulované pomocí speciálních gramatik [34].

4.4 Vybrané nástroje pro rozpoznání textu

Tato sekce uvádí možné prostředky pro rozpoznání textu, které jsou vhodné nejen pro webovou službu. Jednotlivé nástroje či služby jsou popsány s ohledem na následující vlastnosti:

- druh nebo typ licence,
- rok uvedení na trh,
- přesnost rozpoznání,
- počet podporovaných jazyků,
- podporovaný vstupní/výstupní formát,
- podpora programovacích jazyků.

4.4.1 Tesseract

Jedná se o open-source OCR software uvolněný pod licencí Apache, který začala v roce 1985 vyvíjet společnost Hewlett Packard. Jako open-source byl projekt zveřejněn v roce 2005 a od následujícího roku je vyvíjen pod záštitou společnosti Google. Aktuálně dostupná stabilní verze nese označení V3.04.01 a je kompatibilní se všemi hlavními operačními systémy. [37]

Z pohledu přesnosti se dle článků [38, 39] dlouhodobě jedná o jedno z nejlepších řešení pro rozpoznání textu. Vlastnosti Tesseractu jsou přístupné skrze program `tesseract` pro příkazovou řádku nebo prostřednictvím vývojářské knihovny `libtesseract`, jejíž API je dostupné pro jazyk C a C++. Aktuální verze podporuje rozpoznání textu ve více jak 100 jazycích a další je možné natrénovat. Pokud je Tesseract ve verzi 3.00 a výše sestaven společně s knihovnou Leptonica[2], je možné získat také informace o formátování a rozložení textu v dokumentu. Výsledek rozpoznání je dostupný ve formě plain-text, hOCR[3] a PDF.

Na druhou polovinu roku 2017 je naplánováno uvolnění nové verze 4.00, která slibuje podporu OpenMP[4] a díky použití neuronových sítí by mělo dojít také k dalšímu zpřesnění výsledků rozpoznání [40].

4.4.2 ABBYY FineReader

Jde o komerční řešení pro rozpoznání textu, které je vyvíjené od roku 1989 ruskou společností ABBYY. Aktuálně je FineReader dostupný ve verzi 14. Základní cena začíná na €199 a šplhá se až k €499.

Dle zprávy [41], která se zabývá vzájemným srovnáním s dříve zmíněným Tesseractem, dosahuje FineReader lehce vyšší přesnosti. Je ovšem nutné poznamenat, že rozpoznání probíhalo na textech s historickým fontem, na které byly obě metody natrénovány, a tudíž se reálné výsledky na soudobých fontech mohou lišit. Samotná společnost ABBYY slibuje až 99,8% přesnost rozpoznání [42].

FineReader je zaměřen především na koncového uživatele s operačním systémem Windows. Je možné využít také tzv. terminal mode, který je vhodný pro nasazení na Microsoft Windows Server. Aktuální verze dokáže rozpoznat text ve více než 190 jazycích. Stejně jako Tesseract umožňuje FineReader rozpoznat rozložení (včetně poznámek pod čarou a číslování stránek) a formátovaní textu dokumentu. Podporuje širokou škálu vstupních formátů (např. TIFF, PNG) a také výstupních formátů (plain-text, CSV, DOCX, EPUB). Standard hOCR však není podporován. [42]

Kromě nástroje FineReader poskytuje společnost ABBYY také další produkty jako například *Recognition Server* – pro rozpoznání a archivaci dokumentů na serveru, *FineReader Engine* – SDK pro rozpoznání textu dostupné pro všechny hlavní OS nebo *Cloud OCR* – SDK pro zpracování a rozpoznání textu v cloudu.

4.4.3 Google Cloud Vision

Jedná se o poměrně nové API (představené v únoru roku 2016) pro zpracování obrazu v cloudu, jenž je vyvíjené společností Google nad vlastní platformou Google Cloud. Kromě rozpoznání textu poskytuje služba také detekci obličeje, objektů, významných míst nebo například loga obchodních značek. Jednotlivé funkce jsou dostupné skrze REST i RPC nebo prostřednictvím poskytnutých knihoven pro jazyk C#, Go, Java, PHP, Python a Ruby. [43]

[2] Open-source knihovna pro zpracování a analýzu obrazu.
[3] Standard pro reprezentaci formátovaného textu získaného pomocí OCR ve formě XML.
[4] API pro podporu multiplatformního paralelního programování na zařízeních se sdílenou pamětí.

Podle prvních testů přesnosti dává Google Cloud Vision velmi dobré výsledky ve scénách s přirozeným výskytem textu (tzn. snímky obsahující kromě jiného značky, loga a jiné nápisy). V takovém případě si vede lépe než Tesseract, naopak ve scénách obsahující pouze text zaostává v přesnosti i rychlosti zpracování. [44]

Google Cloud Vision podporuje rozpoznání textu ve více než 55 jazycích (vč. češtiny) a poskytuje také automatickou identifikace jazyka dle předloženého snímku. Informace o formátování textu nejsou k dispozici, rozložení je dostupné pouze ve formě obalového tělesa. Parametrem dotazu je přímo obrázek nebo odkaz na něj ve službě Google Cloud Storage. Odpověď, která obsahuje rozpoznaný jazyk, text a souřadnice obalového tělesa, obdrží klient ve formátu JSON. [43]

Služba je zpoplatněna dle počtu dotazů za měsíc, přičemž prvních 1000 je zdarma. Za každý další tisíc dotazů si Google účtuje 0,6 až 1,5 dolarů v závislosti na souhrnném počtu dotazů za měsíc [43].

4.4.4 Microsoft Computer Vision

Jedná se také o poměrně nové API (představené v březnu roku 2016) pro cloudové zpracování obrazu, jenž je vyvíjené společností Microsoft nad vlastní platformou Azure. Kromě rozpoznání textu podporuje také komplexní analýzu obrazu (detekce objektů, prostředí, obličejů včetně odhadu věku a identifikace pohlaví, určení dominantních a akcentních barev aj.) i videa (téměř v reálném čase), rozpoznání celebrit a generování náhledů. Služba Computer Vision je součástí širší skupiny zvané Microsoft Cognitive Services, která seskupuje služby pro přirozené strojové vnímání vjemů z obrazu. [45]

Microsoft Computer Vision podporuje rozpoznání textu v 21 světových jazycích (vč. češtiny) a poskytuje také automatickou identifikace jazyka dle předloženého snímku. Informace o formátování textu nejsou k dispozici, rozložení je dostupné pouze ve formě obalového tělesa. Na rozdíl od Google Cloud Vision poskytuje Microsoft Computer Vision také detekci orientace a automatickou korekci před samotným procesem rozpoznání textu. Obrázek je dotazu předán přímo nebo pomocí libovolné URL adresy. Odpověď je specifikována ve formátu JSON a obsahuje rozpoznaný jazyk a text, souvislé regiony textu, jednotlivé řádky i slova a souřadnice obalového tělesa. [46]

Služba je zpoplatněna podobně jako Google Cloud Vision s tím rozdílem, že prvních 5000 dotazů je zdarma. Za každý další tisíc dotazů je cena stanovena na 1,5 dolarů [45].

Kapitola 5

Mobilní platforma iOS

Historie platformy iOS se začala psát na začátku ledna roku 2007, kdy byl uveden první mobilní telefon společnosti Apple – iPhone. Jeho uvedení způsobilo vlnu velkého nadšení, neboť se dle mnohých jednalo o přelomové zařízení, na které však tehdejší společnost ještě nebyla připravena. iPhone měl 3,5 palců velký dotykový displej a žádnou hardwarovou klávesnici. Poháněl jej vlastní operační systém, v té době označovaný ještě jako iPhone OS, který je nyní známý pod označením iOS. Rok po uvedení prvního iPhonu bylo oznámeno zveřejnění balíčku vývojových nástrojů iPhone SDK, který dodnes, jen pod jiným názvem (iOS SDK), umožňuje vývojářům vytvářet vlastní aplikace a dále je poskytovat prostřednictvím distribuční sítě AppStore. Současně pak Apple také oznámil integraci iPhone SDK do vlastního integrovaného vývojového prostředí Xcode. Hlavním nativním jazykem pro vývoj iOS aplikací byl až do roku 2014 jazyk Objective-C.

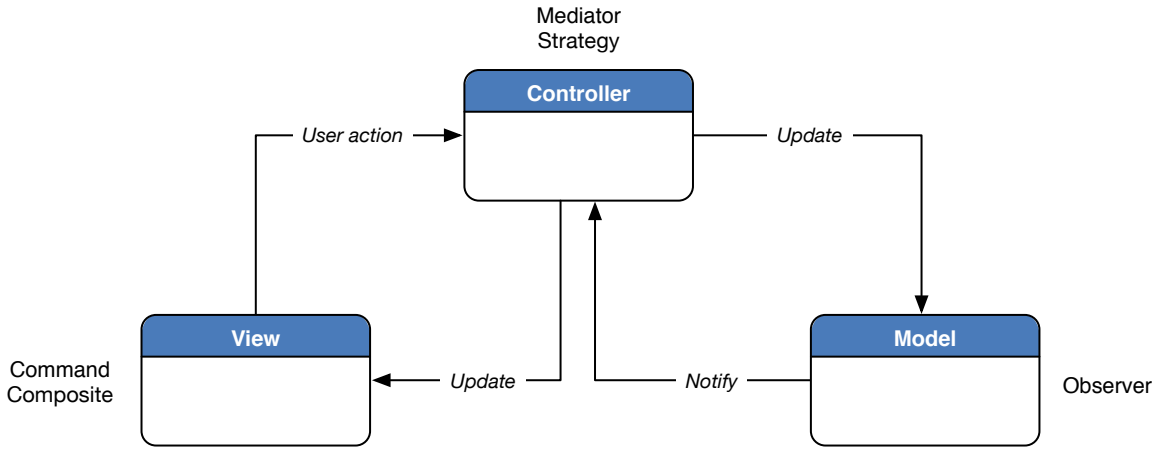
V současné době však již jeho roli převzal nový moderní jazyk Swift, který je aktuálně dostupný ve verzi 3.1. Rodina Apple zařízení se rozšířila o další členy – tablet iPad (iOS), multimediální centrum AppleTV (tvOS) a nejnovějším přírůstkem jsou chytré hodinky AppleWatch (watchOS). Apple každoročně na přelomu května a června pořádá vývojářskou konferenci WWDC, na které pravidelně představuje novou verzi výše uvedených operačních systémů, čímž podněcuje nejen zájem vývojářů, ale také široké veřejnosti.

5.1 Koncepty v Cocoa Touch

Cocoa Touch obecně označuje hlavní knihovny *Foundation Framework*, *CoreData* a *UIKit*, které jsou esenciální součástí vývoje téměř každé mobilní aplikace pro iOS. Prostředky, které tyto knihovny poskytují, by však nebylo možné efektivně použít bez znalosti následujících tří základních konceptů. Další text vychází především z bakalářské práce [47] a vývojářské příručky [48].

5.1.1 Model-View-Controller

Jedná se o objektový návrhový vzor (zkráceně označovaný jako MVC), který rozděluje objekty aplikace do tří základních rolí a zároveň definuje, jak mezi sebou jednotlivé objekty vzájemně komunikují. Správné použití tohoto návrhového vzoru přináší chtěné charakteristiky znovupoužitelnosti, snadné údržby a jednoduché rozšiřitelnosti. V tomto okamžiku je důležité poznamenat, že návrhový vzor MVC používaný v Cocoa Touch se lehce odlišuje od jeho tradičního pojetí – viz obrázek 5.1.



Obrázek 5.1: Cocoa Touch varianta návrhového vzoru Model-View-Controller [48].

Hlavním rozdílem je zákaz přímé komunikace mezi objekty *Model* a *View* v Cocoa Touch MVC, jež je ovšem možná v tradičním pojetí. Příčina je nasnadě – oba objekty by měly zůstat co nejvíce nezávislé, tak aby byla zajištěna jejich snadná znovupoužitelnost. Příkladem může být jednoduchá tabulka. Pokud by byl způsob prezentace přímo svázán s daty tabulky, nebylo by možné považovat tabulku jako znovupoužitelnou komponentu, proto jsou samotná data vždy oddělena od jejich reprezentace. Prostředník, který zajišťuje komunikaci mezi oběma objekty, se nazývá *Controler*.

Model Zapouzdřuje data a poskytuje operace pro snadnou manipulaci s nimi. Ne neprávem je proto označován jádrem a podstatou celé aplikace. Data, která *Model* uchovává mají většinou perzistentní charakter. Mohou být buď serializována a uložena v klasickém souboru ve formě lokální databáze, nebo mohou být uložena ve vzdáleném úložišti například prostřednictvím komunikačního rozhraní REST. Speciálním případem vzdáleného úložiště je služba iCloud a s ním související framework CloudKit. *Model* může komunikovat s *Controllerem* pouze pomocí mechanismu *Key-Value Observing* (zkráceně KVO). Komunikace v opačném směru (tedy od *Controlleru* směrem k *Modelu*) je možná přímo pomocí definovaných metod *Modelu*.

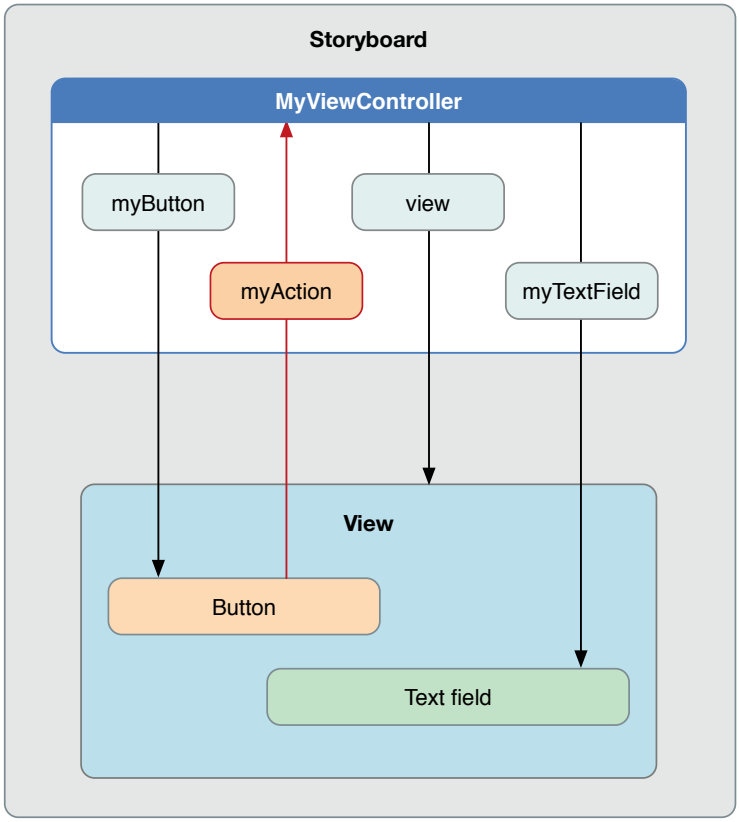
View Definuje, jak jsou data vizualizována na obrazovce a jak s obrazovkou může uživatel interagovat. Již dříve bylo zmíněno, že objekt *View* je koncipován jako znovupoužitelná komponenta pro tvoru grafického rozhraní. Proto v knihovně *UIKit* existuje mnoho předdefinovaných tříd, podle kterých je možné vytvořit objekty s konzistentním vzhledem a chováním. Jednotlivé *View* objekty je možné do sebe hierarchicky zanořovat a jejich rozložení lze definovat graficky pomocí nástroje *Interface Builder*, který je součástí IDE Xcode.

Controller Chová se jako prostředník mezi *Modelem* a *View*. Na základě konceptu Target-Action popsaného níže obsluhuje události, které vznikly interakcí uživatele s uživatelským rozhraním aplikace (*View*). S *Modelem* komunikuje *Controller* přímo prostřednictvím *Modelem* definovaných metod.

5.1.2 Outlet a Target-Action

Outlet je ve své podstatě instanční proměnná, která drží odkaz na další objekt typu *View* (např. tlačítko, tabulka, obrázek) umístěný v top-level *View* objektu. Na obrázku 5.2 jsou outlety vyznačeny černou orientovanou čarou a odpovídající proměnné světle modrými obdélníky umístěnými v *Controller* objektu. *Outlet Collection* je opět instanční proměnná v tomto případě však typu pole, jež dovoluje držet odkaz na více než jeden objekt.

Target-Action je mechanismus, který zasílá události vzniklé interakcí uživatele ve *View* do *Controlleru*. V něm je se vzniklou událostí svázána předem definovaná metoda, která je přijetím události vykonána. Reprezentace události může nést odkaz na odesílatele (tzv. `sender`), případně informaci o typu události (dotek, pohyb).



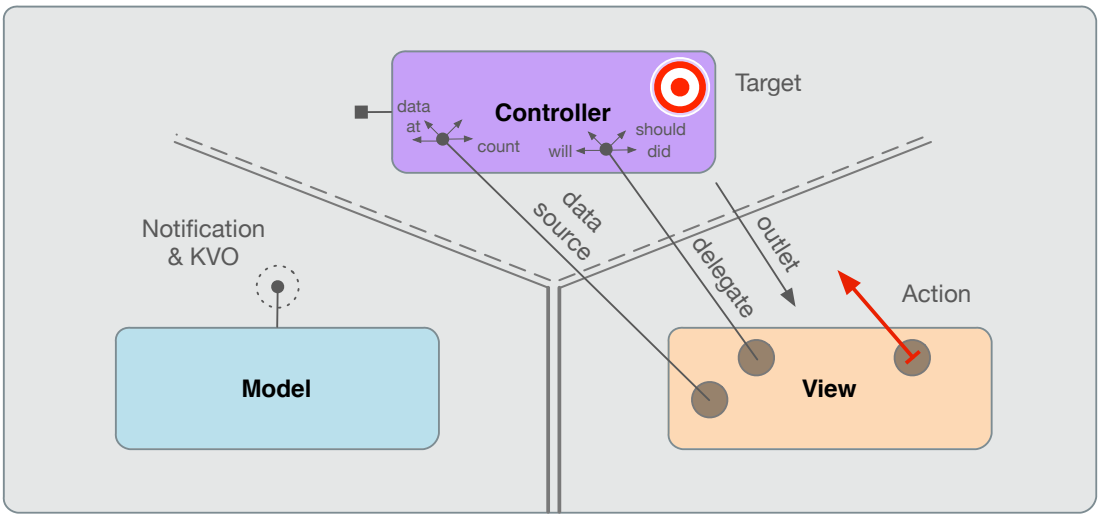
Obrázek 5.2: Vztah mezi objektem *View* a *Controller* [48]. Černé orientované přímky reprezentují *Outlet* a červené mechanismus *Target-Action*. Pokud je tlačítko označené jako *Button* stlačeno, příjme *Controller* akci, která spustí invokaci metody *myAction*.

5.1.3 Delegace a Data Source

Tento koncept podporuje charakteristiky znovupoužitelnosti tím, že z definice objektu vyčleňuje jisté specifické oblasti a deleguje jejich správu jinému objektu. Nejlépe je možné tento koncept vysvětlit na příkladu objektu třídy *UITableView* – tedy tabulky. Aby bylo možné zachovat chtěný generický ráz tohoto prvku, je nutné vyčlenit data, která mají být zobrazena a také reakce na uživatelem vyvolané události v uživatelském rozhraní, např. stisk jisté buňky.

Delegát je objekt, který jinému objektu říká, jak se má v jistých situacích a za různých okolností chovat. Prakticky tedy umožňuje objektu A (např. *Controlleru*) měnit vlastnosti a chování standardního generického objektu B (např. instanci třídy *UITableView*) bez nutnosti použití konceptu dědičnosti. Konkrétně se jedná o instanční proměnnou objektu B (tabulky), nazvanou příznačně `delegate`, která ukazuje na objekt A (*Controller*), zodpovědný za interpretaci událostí. Zodpovědný objekt A obvykle podléhá definovanému protokolu, který obsahuje deklarace metod potřebných pro správnou činnost objektu B.

Data Source je objekt jako *delegát*, avšak místo reakce na události poskytuje objektu data, která mají být prezentována uživateli. Stejně jako delegát podléhá definovanému protokolu, jehož metody poskytují potřebná data. V praxi obvykle *delegát* i *Data Source* ukazují na totožný objekt, zpravidla na *Controller*.



Obrázek 5.3: MVC navrhový vzor s výše popsanými koncepty. Oddělení jednotlivých objektů odpovídá významu vodorovného značení na pozemních komunikacích. Přerušovaná čára dovoluje přímou komunikaci, plná pouze nepřímou pomocí uvedených mechanismů. [49]

5.2 Vybrané aplikace pro rozpoznání textu

Mobilních iOS aplikací pro rozpoznání textu existuje velké množství, avšak žádná se nezabývá rozpoznáním specifických dat. Většina je obecně zaměřená na pouhé rozpoznání vyfoceného textu, případně pro získání a uložení kontaktů z vizitky. Podstatná část je nabízena formou placených aplikací, případně je aplikace omezena na určitý počet rozpoznání textu, který je možné odemknout pomocí in-app nákupů.

5.2.1 ABBYY TextGrabber

Jde o placenou aplikaci ($4,99) od ruských vývojářů ze společnosti ABBYY, která se zabývá rozpoznáním textu již od svého vzniku – viz sekce 4.4.2. Jejich mobilní aplikace dovoluje rozpoznat text ve více než 60 jazycích včetně češtiny a dále umožňuje překlad rozpoznaného textu do více než 100 jazyků. Samozřejmostí je možnost sdílení textu na sociální sítě

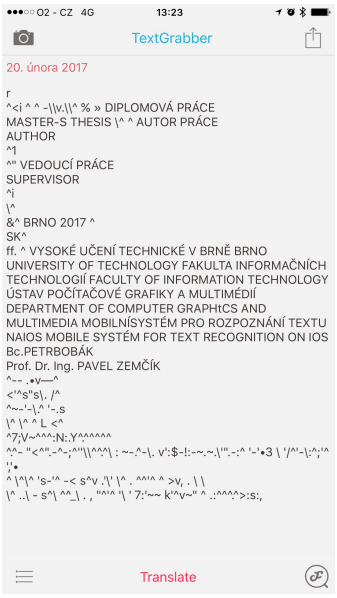
a historie předchozích rozpoznání. Jako bonus poskytuje aplikace čtení QR kódů a možnost předčítání textu pomocí VoiceOver[1]. Aplikaci v americkém AppStore ohodnotilo 1120 uživatelů s průměrným hodnocením 4 z 5 hvězdiček. Zajímavostí je, že totožná aplikace pro operační systém Android stojí dvojnásobně více.

5.2.2 Scanner Pro

Jedná se opět o placenou aplikaci ($3,99) od vývojářského studia Readdle, která je s omezením dostupná i zdarma pod názvem Scanner App. Tato aplikace je především zaměřena na skenování pomocí mobilního telefonu a podle autorů poskytuje pokročilé metody zpracování obrazu (detekce okrajů, odstranění stínu, oprava perspektivního zkreslení), aby výsledný snímek co nejvíce odpovídal kvalitě optického skeneru. Jako doplňkovou službu poskytuje rozpoznání textu ve 21 jazycích včetně češtiny. Aplikaci v americkém AppStore ohodnotilo 30168 uživatelů s průměrným hodnocením 4,5 z 5 hvězdiček.

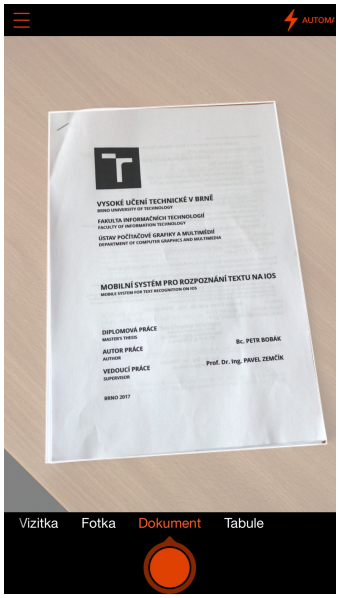
5.2.3 Office Lens

Office Lens je volně dostupná aplikace od vývojářů společnosti Microsoft, která se zaměřuje na skenování dokumentů s možností rozpoznání textu a identifikaci údajů z vizitek. Doplňková služba rozpoznání textu tištěných dokumentů podporuje 21 jazyků včetně češtiny, přičemž rozpoznání psaného textu je dostupné pouze v angličtině a rozpoznání vizitek je možné pouze v angličtině, čínštině, němčině a španělštině. Aplikace umožňuje inteligentní detekci okrajů skenovaného dokumentu včetně úpravy snímané perspektivy. Sdílení je možné do všech mobilních kancelářských aplikací společnosti Microsoft a dále také do formátu PDF. Aplikaci v americkém AppStore ohodnotilo 4605 uživatelů s průměrným hodnocením 5 z 5 hvězdiček.



Obrázek 5.4: TextGrabber

Obrázek 5.5: ScannerPro



Obrázek 5.6: MicrosoftLens

Z výše uvedeného je patrné, že rozpoznání textu většinou tvoří doplňkovou službu aplikací pro skenování dokumentů či vizitek. Nápadnější využití OCR jsem v AppStore nenalezl.

[1] Jedná se o technologii, kterou poskytuje přímo systém iOS v rámci zpřístupnění pro zrakově postižené.

Kapitola 6

Postoj k řešené problematice a technická specifikace

V této kapitole prezentuji vlastní názor na současný stav některých oblastí popsaných v dřívějších kapitolách a osobní motivaci pro volbu zadání této práce. V závěru kapitoly pak blíže specifikuji nástroje a použité technologie pro vývoj systému pro rozpoznání textu i demonstrační aplikace pro iOS.

6.1 Webové aplikace

Webové aplikace si získávají v posledních letech stále více pozornosti a začínají značně konkurovat klasickým desktopovým aplikacím. Příkladem může být například aplikace Office 365 od společnosti Microsoft. Ta své klasické kancelářské nástroje přetavila na webové aplikace, které jsou podlé mého názoru momentálně velmi populární jak v osobní, tak i firemní oblasti. Konec konců i Vysoké učení technické v Brně poskytuje tuto aplikaci svým studentům i zaměstnancům zdarma. Obdobnou aplikaci pojmenovanou jako G Suite poskytuje také společnost Google, z niž nejoblíbenější je zřejmě aplikace Google Docs. Ani společnost Apple tuto oblast nezanedbává a nabízí tak službu iCloud, která zastřešuje celý ekosystém jablečných zařízení. Aplikace dostupná skrze internetový prohlížeč nabízí webový kancelářský balík, prohlížeč fotek uložených v cloudu a další.

Kromě kancelářských balíků můžeme dále jmenovat služby jako Dropbox, Facebook, Twitter, Youtube a mnoho dalších, které již není možné chápat jako pouhé webové stránky, ale jako webové aplikace. Extrémním případem je pak operační systém Chrome OS od společnosti Google, který téměř zcela spoléhá na webové aplikace. Zajímavým příkladem je také služba Geforce Now od společnosti Nvidia, která dovoluje hraní počítačových her ve vysokém rozlišení pouze s minimálními nároky na hardware (a s rozumnými nároky na internetovou konektivitu).

Hlavní výhodou webových aplikací je pak výpočetní nenáročnost z pohledu koncových zařízení, jelikož na něm v konečném důsledku probíhá pouze vykreslování uživatelské rozhraní. Všechny podstatné operace se provádějí na výkonných výpočetních serverech. Paradoxně se však tímto směrováním vracíme zpět k počátkům výpočetní techniky. Tehdy se výpočetní výkon přesouval od sálových počítačů k uživateli, nyní se přesouvá zpět do sálových počítačů (výpočetních serverů). Díky stále kvalitnější konektivitě v domácnostech je však tento krok logickým rozuzlením současného směřování informačních technologií.

Hodně diskutovaným pojmem souvisejícím s webovými aplikacemi je také koncept *single-page application* (SPA), jehož cílem je poskytnout Webu uživatelskou zkušenost známou z desktopových aplikací. V principu jsou v průběhu prvního požadavku klienta z web serveru získány všechny potřebné zdroje (např. skripty) pro logiku aplikace a prezentaci dat. Následně jsou prezentovaná data typicky dynamicky dočtena prostřednictvím komunikace s RESTful webovou službou. Aplikaci je takto možné snadněji vyvinout univerzálně pro různé platformy a zařízení. Logika aplikace zůstává většinou zachována a jediné co se mění je pouze kontext, ve kterém je aplikace spuštěna. Takový vývoj umožňuje například technologie React Native od společnosti Facebook. Pro podporu vývoje single-page aplikací existuje velmi široké množství knihoven a frameworků, které mezi sebou bojují o přízeň vývojářů – jmenovitě se jedná například o již zmíněný Rect.js a dále pak Angular.js, Ember.js, Backbone.js a mnoho dalších.

Webové aplikace spolu s konceptem *single-page application* jsou dle mého názoru budoucností Webu a v brzké době by jistě mohly nahradit klasické statické HTML stránky. Nejprve je však nutné dořešit problémy s kompatibilitou a podporou prohlížečů. Naopak univerzální aplikace nepovažuji za hrozbu pro nativní aplikace, neboť dle mých zkušeností stále nedosahují a nemyslím si, že dosáhnou kvalit aplikací vyvinutých pomocí nativních technologií a nástrojů.

6.2 Webové služby

Webové služby jsou neodmyslitelnou součástí webových aplikací. Webové služby chápu jako motor každé webové aplikace, tedy něco, na čem je aplikace postavena. Jednoduchým příkladem může být například sociální síť Twitter, tedy služba pro sdílení vlastních názorů s jinými lidmi. Vnitřně služba implementuje práci s databází, autentizaci, load balancing a další operace spojené s logikou služby jako celku – publikování příspěvků, jejich komentování, informace o vlastním profilu aj. Tyto operace a informace následně služba zveřejňuje prostřednictvím aplikačního rozhraní, např. formou REST. Nad tímto rozhraním je pak postavena webová aplikace, která webovou službu obléká do lidsky vstřebatelného a pochopitelného hávu.

Webová služba však nemusí tvořit pouze webovou aplikaci. Může být dostupná například jako mobilní aplikace, aplikace pro HbbTV[1] nebo dokonce může být součástí inteligentního autonomního vozidla jako v případě automobilky Tesla, Opel aj. V případě automobilů společnosti Tesla jde například o systém autonomního systému vozidla, který komunikuje se servery a neustále vytváří a aktualizuje virtuální mapu jízdních pruhů, překážek, dopravních značení atd. V případě automobilů Opel pak jde o službu OnStar, která poskytuje průběžnou diagnostiku vozidla, nouzovou asistenci při dopravní nehodě aj.

Jak jsou data přenášena a jak spolu jednotlivé strany klient-server modelu komunikují, specifikuje architektonický styl REST a protokol SOAP. Prvně zmíněný, dle mého názoru, již zcela jistě porazil a překonal protokol SOAP, který byl příliš těžkopádný a složitý. To potvrzuje i skutečnost, že většina dnešních webových služeb poskytuje už pouze REST aplikační rozhraní.

Rozmach webových služeb, který začal na přelomu milénia, jistě nekončí. Nyní se proprietární webové služby pomalu, ale jistě otevírají, a dopomáhají tak ke vzniku chytrých domácností, budov či měst.

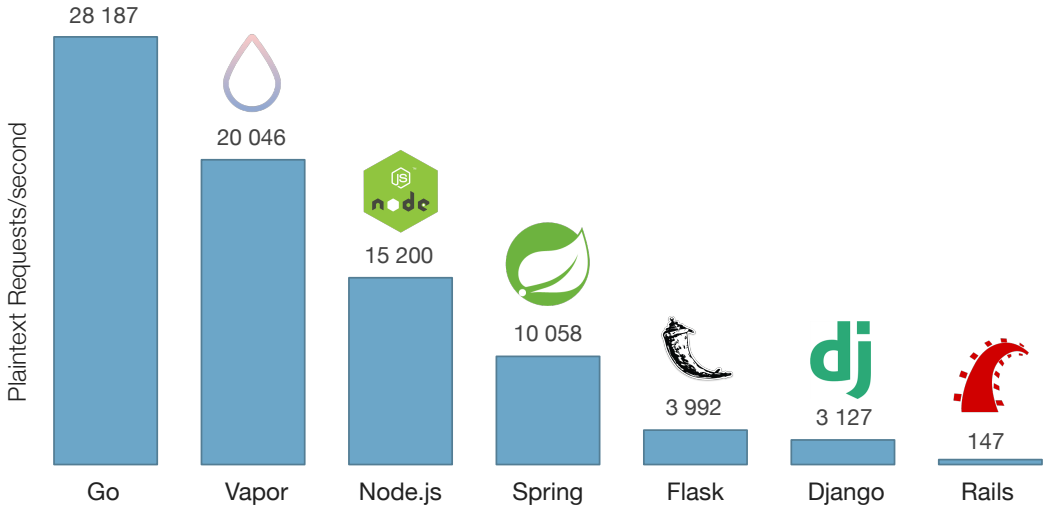
[1] Hybridní televizní vysílání je platforma, která rozšiřuje klasické pozemní vysílání o možnost vazby s širokopásmovým připojením.

6.3 Webové frameworky

Webových frameworků existuje nepřeberné množství. Liší se dle programovacího jazyka, použitých návrhových architektur, případně dle použitých knihoven. Mezi momentálně nejrozšířenější považuji Django, Ruby on Rails a ASP.NET.

Django je, dle mého názoru, jednou z nejlepších možných variant pro tvorbu RESTful aplikačního rozhraní webových služeb. Původně však Django vzniklo jako redakční systém pro zprávu obsahu (CMS – Content Management System). Postupem času se ovšem začalo přeměňovat do stávající podoby full-stack frameworku, se silnými základy databázového systému, administrací a výbornou podporou šablon. Dnes je Django velmi aktivně vyvíjeno a existuje kolem něj rozsáhlá komunita, která publikuje všemožná rozšíření ve formě Python modulů. Jedním z takových je také Django REST framework – excelentní rozšíření pro tvorbu REST aplikačních rozhraní. Vyniká především díky vlastnosti nazvané *Browsable API*, což je možnost zpřístupnit vytvořené RESTful rozhraní ve formě interaktivní webové stránky. Tyto vlastnosti společně s výtečně zvládnutou dokumentací a podrobně vysvětlenými příklady, zcela jistě patří mezi hlavní důvody, proč jako primární webový framework momentálně zvolit právě Django.

Budoucnost však vidím také ve skupině frameworků založených na programovacím jazyku Swift, které jsou souhrnně označovány jako *Server-side Swift*. Schopnost vyvíjet webové služby ve stejném jazyce jako klientské aplikace je nesmírně lákavá, ba co víc, tento pocit je navíc podpořen také bezkonkurenčním výkonem s ohledem na výpočetní výkon serveru – viz obrázek 6.1. Swift, coby primární programovací jazyk společnosti Apple, je open-source jazykem, a proto nic nebrání rozvoji této skupiny frameworků.



Obrázek 6.1: Porovnání propustnosti webového serveru využívající různé frameworky [50].

6.4 Motivace ke vzniku této práce

Na počátku bylo první myšlenkou vytvořit mobilní aplikaci pro rozpoznání aditivních látek v potravinách. V současné se zabývá složením potravin dostatečné množství lidí, které by výsledná aplikace mohla oslovit, a toto tvrzení podporuje také výsledek ankety Mladé fronty DNES [51], ze které vyplývá, že více než 90 % dotázaných složení potravin skutečně sleduje.

Složení potraviny je strojově možné získat pomocí dvou základnách přístupů – buď pomocí informace z EAN kódu, nebo pomocí rozpoznání složení pomocí OCR. V případě EAN kódu je však nutné mít k dispozici i databázi výrobků, kterou není lehké získat a její aktuálnost může být diskutabilní. Nutno podotknout, že aplikace založené na EAN kódu nyní existují alespoň dvě – *NutriAtlas* a *Tasty* – nezdají se však být příliš rozšířené.

Z výše popsaných důvodů jsem se rozhodl jít cestou přístupu OCR. Na úvodní schůzce s vedoucím práce jsme se společně shodli, že by bylo přínosné pro rozpoznání textu vyčlenit samostatný server. Proto se moje pozornost začala postupně otáčet k webovým službám, o kterých jsem měl do té doby jen částečné povědomí.

Osobním přínosem jsou pro mě tedy nové znalosti z široké oblasti webových služeb a technologií pro rozpoznání textu. Ze získaných poznatků si kladu za cíl vytvořit univerzální webovou službou pro rozpoznání textu, která bude splňovat všechny parametry pro reálné použití. Zaměřím se tedy nejen na použití existujících technologií OCR, ale také na registraci a autentizaci uživatelů pomocí stávajících nároků a standardů. Dále se pokusím realizovat webovou aplikaci pro správu služby, která navíc nabídne analytické a realizační zázemí pro vývojáře. Jako doplněk ke službě poskytnu také iOS framework pro jednoduchou komunikaci se službou a ukázkovou mobilní aplikaci pro iOS implementující rozpoznání aditivních látek v potravinách. Výslednou webovou službu pak použije v rámci své diplomové práce [52] také kolega Jan Tomešek, který se v ní podrobně zabývá předzpracováním obrazu pro OCR a implementuje velmi podobnou mobilní aplikaci pro operační systém Android.

6.5 Technická specifikace

Následující tabulka popisuje výchozí požadavky na klientskou i serverovou část řešené služby. Vychází z analýzy dostupných technologií a osobních preferencí pro jednotlivé nástroje.

Server		
Parametr	Použité řešení	Verze
Operační systém	Ubuntu Server	16.04 LTS
Web server	Apache 2 + `mod_wsgi`	2.4.18
Programovací jazyk	Python 3	3.5.2
Komunikační řešení	REST	–
Webový framework	Django	1.11 LTS
REST framework	Django REST Framework	3.6.2
Databáze	SQLite 3	3.8.11
Autentizace klientů	Django OAuth Toolkit	0.12.0
Rozpoznání textu	Tesseract	3.04.01
	Microsoft Cognitive Services – Computer Vision API	1.0 (Beta)
	Google Cloud Platform – Cloud Vision API	1.1 (Beta)
Klient		
Parametr	Použité řešení	Verze
Operační systém	iOS	10.3
Programovací jazyk	Swift	3.1
Databáze	SQLite 3	–

Tabulka 6.1: Výchozí požadavky pro technickou specifikaci řešení.

Kapitola 7

Návrh a dekompozice

Tato kapitola se věnuje návrhu a dekompozici služby pro rozpoznání textu, kterou jsem pracovně pojmenoval jako *Texie Cloud API*. V jednotlivých sekcích nejprve popisuji celkovou architekturu služby a následně blíže seznamuji s jejími částmi. Součástí je také popis navrženého uživatelského rozhraní klientské demonstrační aplikace *Ingredients*, které je ilustrováno pomocí drátěných modelů. Podobně je v sekci věnující se serverové části popsáno rozhraní webové správy.

7.1 Celkový pohled na navržený systém

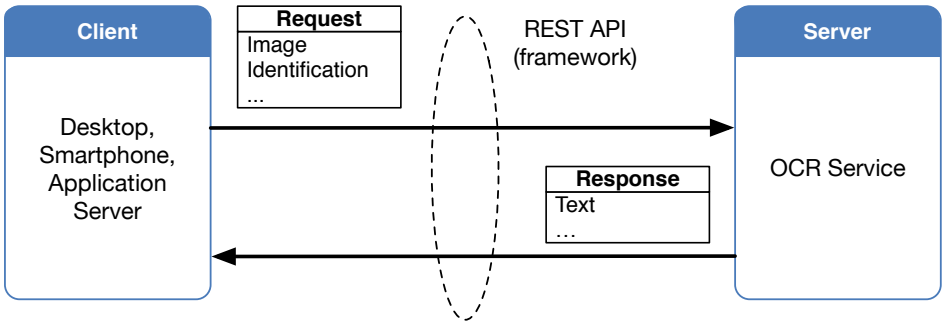
Systém pro rozpoznání textu jsem navrhl v souladu s klient-server modelem, neboť bylo mým cílem vytvořit univerzální řešení vhodné nejen pro platformu iOS. Navržený systém vychází z dvouvrstvé klient-server architektury, jejíž bližší ilustrace je uvedena na obrázku 7.1.

Serverová část zprostředkovává nástroj pro rozpoznání textu pomocí několika různých technologií, mezi kterými lze pomocí parametrizace daného koncového bodu[1] (angl. endpoint) libovolně přepínat. Server musí kromě jiného řešit také identifikaci jednotlivých dotazů tak, aby je bylo možné spojit s konkrétním uživatelem služby. S tímto je spojena také potřeba autentizačního mechanismu. V neposlední řadě by měl server poskytovat webovou aplikaci, ve které je možné sledovat statistiky dotazů, spravovat účty aj.

Klientská část se serverem komunikuje pomocí dodaného frameworku, který usnadňuje zaslání požadavku na server a zpracování jeho odpovědí. Získaná data jsou pak v rámci klienta zpracována aplikačně specifickým způsobem – v případě ukázkové aplikace *Ingredients*, je výsledek rozpoznání porovnán s interní databází aditivních a škodlivých látek vyskytujících se v potravinách.

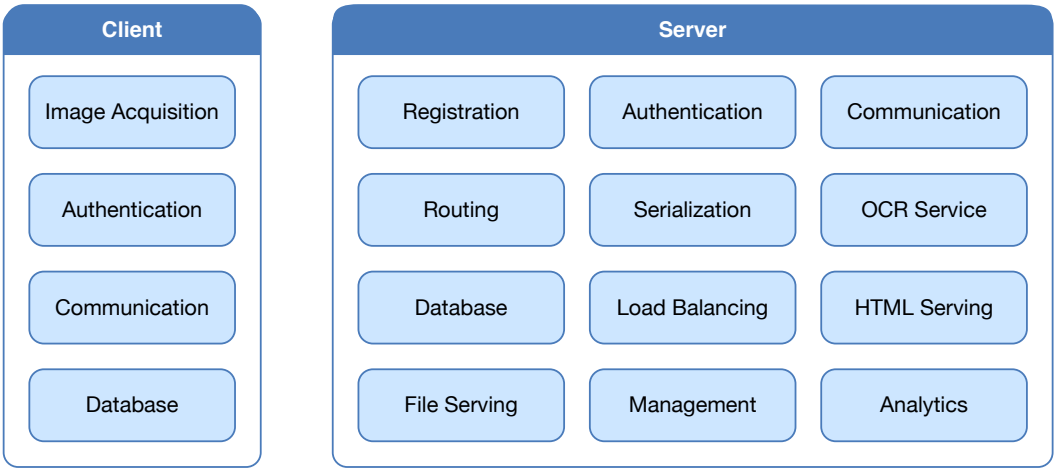
Kromě možnosti komunikace klienta se serverem prostřednictvím frameworku v jazyce Swift, se však nabízí také REST aplikační rozhraní, které ve své podstatě framework pouze objektově zapouzdřuje. Přítomnost tohoto rozhraní pak umožňuje vznik dalších knihoven či frameworků pro širokou škálu dalších programovacích jazyků. Výhodou pro případné vývojáře by měla být také dostupnost interaktivní dokumentace, díky které bude možné vyzkoušet činnost a chování poskytovaného aplikačního rozhraní přímo ve webovém prohlížeči.

[1] Koncový bod je „vzdálená funkce“, která přeměňuje jisté vstupní parametry na výstupní data.



Obrázek 7.1: Příklad komunikace mezi klientem a serverem. Klient zasílá požadavek, jehož hlavním parametrem je reprezentace snímku. Server požadavek příjme, provede rozpoznání textu a ten jako odpověď odešle zpět klientovi, který jej zpracuje aplikačně specifickým způsobem.

Rozdělení odpovědností mezi obě strany klient-server modelu, zřetelné z obrázku 7.2, odpovídá definici tenkého klienta. Ten poskytuje vstupní data ve formě snímku, která server zpracuje a transformuje na výstupní data obsahující rozpoznaný text. V závěru klient získaná data interpretuje do člověkem srozumitelné reprezentace. Tímto způsobem dekompozice je tak možné například snížit celkovou energetickou náročnost klientské aplikace, což je podstatné především u aplikací pro mobilní zařízení.



Obrázek 7.2: Podrobný diagram ilustrující subsystémy jednotlivých částí služby. Klientská část odpovídá příkladové aplikaci *Ingredients* pro rozpoznání složení potravin.

Další dvě sekce se podrobně věnují jednotlivým částem navrženého systému. Výše zmíněné informace jsou postupně upřesňovány a konkretizovány tak, aby bylo možné získat lepší přehled o všech přítomných subsystémech.

7.2 Klientská část

Demonstrační aplikace *Ingredints* pro rozpoznání aditivních látek z obalového materiálu potravin reprezentuje klientskou část klient-server modelu. Aplikace je tvořena akvizičním systémem pro zachycení snímaného obrazu a databázovým subsystémem uchovávajícím informace o aditivních látkách – viz obrázek 7.2. Autentizaci aplikace a komunikaci se serverem zajistí obecný iOS framework pro komunikaci s webovou službou *Texie Cloud API*.

7.2.1 Uživatelské rozhraní

Návrh uživatelského rozhraní nejprve předcházela definice vlastností, které bude umožňovat tzv. MVP – *Minimal Viable Product*[2]. Jádro celé aplikace tvoří detekce aditivních látek v potravinách a vhodná prezentace získaných informací uživateli. Další podstatnou vlastností MVP je historie již provedených rozpoznání a poslední doplňující funkcionalitu pak tvoří seznam existujících aditiv včetně podrobnějšího popisu. Na základě tohoto výčtu vlastností jsem rozdělil uživatelské rozhraní aplikace do tří funkčních částí (nebo také top-level obrazovek).

Obrazovka *Capture* je hlavní obrazovka vždy po spuštění aplikace. Její součástí je především výrazné tlačítko, které spustí akviziční subsystém pro zachycení obrazu. V horní části se pak již nachází pouze přepínač, který dovoluje aktivovat *privátní režim*, ve kterém server neukládá příchozí snímek do interní databáze a stejně tak v aplikaci nedovoluje výsledek rozpoznání uložit do historie. Po akvizici snímku je uživateli zobrazena obrazovka *Result*, která prezentuje rozpoznaná aditiva, na jejichž základě je vyhodnocena celková „zdravost“ dané potraviny.

Obrazovka *Recents* slouží jako seznam dříve provedených rozpoznání. Jednotlivé buňky jsou tvořeny fotografií pořízenou pro účel rozpoznání, časovým razítkem a hodnocením „zdravosti“ potraviny. Dotykem buňky se zobrazí výše popsaná obrazovka *Result*, která uživateli poskytne bližší informace.

Obrazovka *Additives* poskytuje seznam nebezpečných aditivních látek, které nepodporují zdravý životní styl uživatele. Každá buňka zobrazuje kódové i slovní označení látky a její účel v potravinářském průmyslu (např. barvivo, konzervant). Dotykem buňky se zobrazí obrazovka *Detail*, která zobrazuje bližší informace a nežádoucí účinky dané látky.

Zamýšlený případ užití je pak následující – uživatel například při nákupu potravin spustí aplikaci, načež se vzápětí objeví obrazovka *Capture* s výrazným tlačítkem doplněným o pulzující animaci. Jakmile se uživatel tohoto tlačítka dotkne, spustí se akviziční subsystém, který uživateli zobrazí klasické rozhraní fotoaparátu (náhled a spoušť). Po zachycení etikety potraviny se obraz odešle na server *Texie Cloud API*, kde proběhne rozpoznání textu. Následně se již v rámci aplikace rozpoznaný text porovná s klíčovými slovy aditivních látek (e-kód, název látky), které jsou dostupné z interní databáze aplikace. V poslední fázi se uživateli zobrazí celkové hodnocení potraviny, založené na kategorii nejškodlivější rozpoznané látky, vizualizované pomocí jednoduchého indikátoru a seznam všech rozpoznaných aditiv. Návrh výše popsaných obrazovek je možné zhlédnout na obrázku 7.3, zbývající obrazovky je pak možné nalézt v příloze B.

[2]Produkt, který obsahuje pouze nejpodstatnější funkcionalitu pro uspokojení cílové skupiny uživatelů.

(a) Obrazovka *Capture* (b) Obrazovka *Result* (c) Obrazovka *Detail*

Obrázek 7.3: Ilustrace jednotlivých obrazovek uživatelského rozhraní vycházejících z top-level obrazovky *Capture*. Obrazovka (a) poskytuje hlavní tlačítko pro spuštění akvizičního subsystému a vedlejší tlačítko pro aktivaci privátního režimu. Hlavní tlačítko je zvýrazněno pulzující animací. Obrazovka (b) poskytuje výsledek rozpoznání etikety v uživatelsky přívětivé formě. Detail nalezených aditiv, který ukazuje obrazovka (c), je možné zobrazit dotykem dané buňky.

7.2.2 Akviziční subsystém

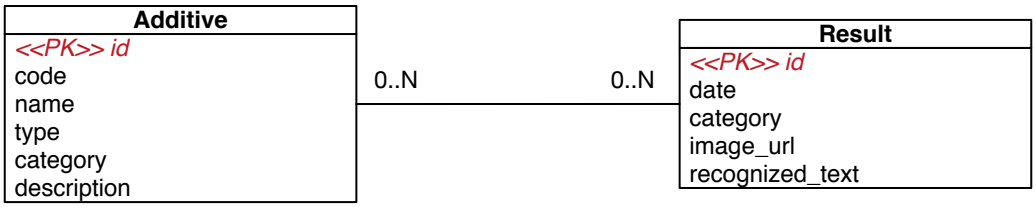
Akviziční subsystém zajišťuje obsluhu fotoaparátu mobilního telefonu. Podstatným kritériem pro aplikaci *Ingredients* je schopnost zachytit co možná nejostřejší obraz. Problém může nastat především v případě špatných světelných podmínek, kdy by mělo být možné automaticky aktivovat blesk. Po dokončení akvizice snímku je následně vhodné provést konverzi surových dat ze snímače do komprimovaného formátu, který sníží datovou náročnost aplikace při odesílání snímku na server. Kompresní poměr je však nutné vhodně zvolit tak, aby nedošlo k přílišnému poškození vysokofrekvenční části získaného obrazu.

7.2.3 Framework pro komunikaci se serverem

Komunikaci aplikace s REST rozhraním serveru *Texie Cloud API* usnadní framework pro jazyk Swift, který bude obecně použitelný nejen pro aplikaci *Ingredients*. Cílem je vytvořit jednoduché aplikační rozhraní, které umožní nikoli jen získaný snímek odeslat na server, ale také snadno zpracovat příchozí odpověď. Součástí webové služby *Texie Cloud API* je však i autentizační mechanismus založený nad standardem OAuth 2.0 (viz následující sekce 7.3.2), jehož implementace bude muset být také nutnou součástí výsledného frameworku.

7.2.4 Databázový subsystém

V rámci klientské aplikace *Ingredients* slouží databázový systém především jako zdroj informací o aditivních látkách. Současně je však potřebný také pro historii již rozpoznaných potravin, kterou poskytuje výše popsaná top-level obrazovka *Recents*. Následující obrázek 7.4 ilustruje jednotlivé entitní množiny a vzájemné vztahy mezi nimi.



Obrázek 7.4: ER diagram ilustrující entitní množiny databáze na straně klienta a vzájemné vazby mezi nimi.

Entitní množina *Additive*, reprezentuje jednotlivé aditivní látky, přičemž atribut `code` zastupuje kódové označení dané látky, `type` zachycuje účel látky v potravinářském průmyslu, `category` udává míru nebezpečnosti a atribut `description` uvádí bližší informace k nežádoucím účinkům dané látky.

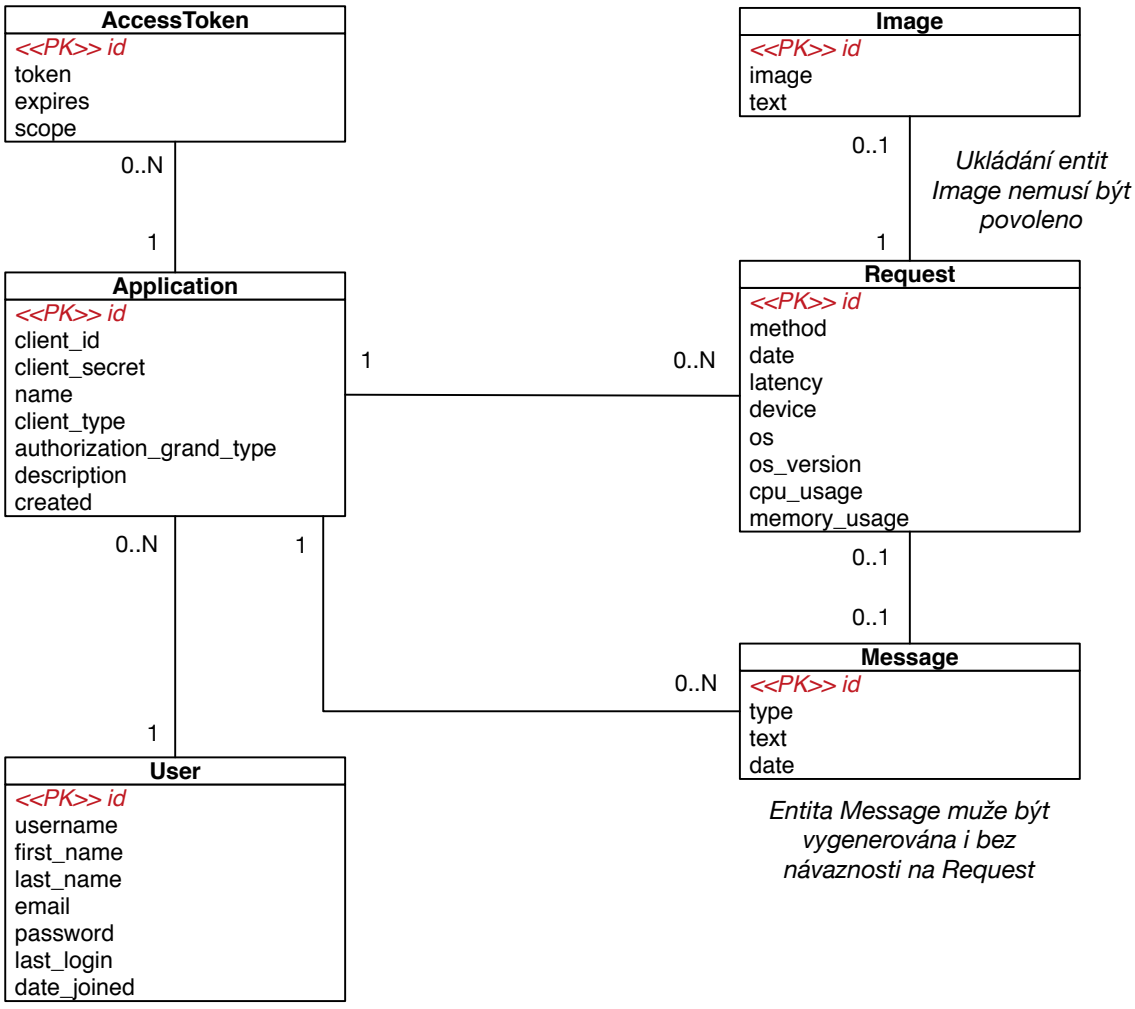
Entitní množina *Result*, reprezentuje výsledek již provedeného rozpoznání, ale jen v případě, kdy nebyl aktivován privátní režim. Atribut `category` označuje míru nebezpečnosti celé potraviny a `image_url` poskytuje odkaz na snímek spojený s konkrétním rozpoznáním. Zde je využita vlastnost serveru *Texie Cloud API*, který umožňuje získaný výsledek rozpoznání včetně souvisejícího snímku uložit na serveru, což ve výsledku šetří paměťový prostor klientského zařízení.

7.3 Serverová část

Navržená serverová část je tvořena několika subsystémy – viz obrázek 7.2. Jádro serveru tvoří subsystém pro rozpoznání textu a komunikační subsystém. Na tyto základní prvky je pak navázán databázový systém společně se systémem pro serializaci dat. Víceméně samostatnou částí je webová aplikace pro správu služby, která spolupracuje především se subsystémem pro poskytování souborů a dále analytickým ale i databázovým subsystémem. V dalších sekcích jsou podrobněji popsány vybrané části.

7.3.1 Databázový subsystém

Databázový systém je pro řešenou webovou službu nezbytnou součástí. Nejenže je potřebný pro autentizační subsystém, ale i pro analytický subsystém a částečně také pro subsystém pro rozpoznání textu. Následující obrázek 7.5 ilustruje jednotlivé entitní množiny a vzájemné vztahy mezi nimi.



Obrázek 7.5: ER diagram ilustrující entitní množiny a vazby mezi nimi. Příčiny vzniku vazeb 1:1 mezi entitní množinou *Request* – *Annotations* a *Request* – *Message* jsou uvedeny přímo v diagramu.

7.3.2 Autentizační subsystém

Autentizační subsystém umožňuje propojit konkrétní požadavek s konkrétní aplikací, resp. uživatelem webové služby. Pro tento účel jsem se rozhodl využít mechanismus založený nad standardem OAuth 2.0. Jeho cílem je nahradit přenášení identity klienta v rámci každého dotazu *přístupovým tokenem*, z něhož není možné skutečné identifikační údaje klienta odvodit. Tento token má omezenou dobu platnosti, po kterou je jej možné použít. Jakmile jeho platnost expiruje, je nutné zažádat o nový token.

Identita klienta se v případě autentizace vůči serveru vztahuje ke konkrétní aplikaci (ne uživateli). Aplikaci je možné vytvořit v rozhraní webové aplikace pro správu serveru. Poté co tak vývojář klientské aplikace učiní, získá identifikační údaje ve formě dvou klíčů – `client_id` a `client_secret`.

Jakmile jsou klientovi známy identifikační údaje, může server požádat o přidělení přístupového tokenu (`access_token`). Pouze v této situaci je třeba oba identifikační klíče `client_id` a `client_secret` v definované formě odeslat serveru, který vzápětí odpoví odpovědí obsahující přístupový token. V další komunikaci se serverem už klient uvádí pouze tento token spolu s dalšími parametry, které jsou dané požadovaným koncovým bodem.

Popsaný způsob autentizace využívá přístupový typ *Client Credentials*, který je definován v RFC 6749 – *The OAuth 2.0 Authorization Framework* [53], ve kterém je také možné nalézt další informace k této problematice.

7.3.3 Aplikační rozhraní služby

Aplikační rozhraní služby je navrženo pomocí dříve uvedeného architektonického stylu REST. Jednotlivé koncové body jsou specifikovány podle typických konvencí, jež jsou podrobně popsány ve skvěle zpracované příručce CA Technologies [11].

Koncový bod	Metoda	Parametry dotazu	Význam
`annotations/`	`POST`	`method`	Rozpozná text a dle parametru `store` případně uloží do databáze.
		`store`	
	`GET`	`page`	Vrátí *seznam* uložených rozpoznání.
`annotations/{id}/`	`GET`	–	Vrátí konkrétní *detail* rozpoznání specifikovaný pomocí *id*.
`token/`	`POST`	–	Vrátí přístupový token s informací o expiraci.
`revoke_token/`	`POST`	–	Zneplatní daný přístupový token.
`endpoints/`	`GET`	–	Vrací seznam poskytovaných koncových bodů.

Tabulka 7.1: Návrh koncových bodů serveru pro komunikaci s klientem. Bližší specifikaci je možné nálezt v příloze A.

Koncový bod `annotations` je určen pro práci se subsystémem pro rozpoznání textu. Pomocí metody `POST` je realizováno rozpoznání textu ve snímku, který je součástí těla dotazu pod klíčem položky formuláře `image`. Tělo dotazu (`Content-type`) je specifikováno typem MIME `multipart/form-data`. Hlavička dotazu musí kromě jiného obsahovat položku

`Authorization`, jejíž hodnota odpovídá přístupovému tokenu vztahujícímu se k dané aplikaci. Na takto formulovaný dotaz server odpoví buď chybovým kódem s doplňující hlášením, nebo zprávou, která obsahuje rozpoznaný text, označení použité metody, odkaz na archivovaný snímek a identifikátor vytvořeného záznamu z databáze. Metodou `GET` je možné získat seznam všech provedených rozpoznání v rámci dané aplikace, případně detail konkrétního požadavku.

Koncový bod `token` slouží pro získání přístupového tokenu, který je potřebný pro použití výše zmíněného bodu `annotations`. Tento koncový bod je tedy možné použít pouze ve spojení s HTML metodou `POST`. Tělo dotazu (`Content-type`) je specifikováno typem MIME `application/x-www-form-urlencoded`. Jeho jediná položka `grand_type`, určuje variantu přístupu dle standardu OAuth 2.0 – viz předchozí sekce 7.3.2. V případě použití *Texie Cloud API* bude tato položka vždy obsahovat hodnotu `client_credentials`. Hlavička dotazu je tvořena položku `Authorization`, která odpovídá HTTP metodě autentizace *Basic* operující s klíči `client_id` a `client_secret`. Odpovědí na správně formulovaný dotaz je přístupový token doplněný o informaci určující dobu expirace.

Koncový bod `revoke_token` slouží v případě *Client Credentials* pouze pro zneplatnění dříve vystaveného přístupového tokenu. Tělo dotazu (`Content-type`) je specifikováno typem MIME `application/x-www-form-urlencoded`. Musí jej tvořit identifikační údaje dané aplikace (`client_id`, případně `client_secret`), typ tokenu (pouze `access_token`) a nakonec také konkrétní token, nad kterým má být požadovaná operace provedena.

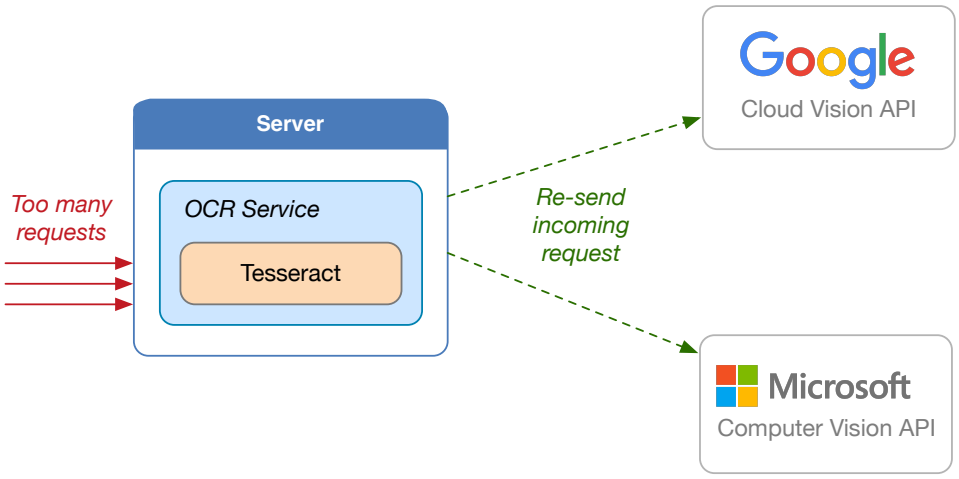
Poslední koncový bod `endpoints` vrací klientovi seznam aktuálně implementovaných koncových bodů pro komunikaci s rozpoznávacím serverem *Texie Cloud API*.

7.3.4 Subsystém pro rozpoznání textu

Tento subsystém tvoří stěžejní část serverové části. Jeho vstupem je reprezentace snímku nejlépe ve formátu JPEG. Samotné rozpoznání textu poté proběhne pomocí předem specifikovaných metod uvedených v kapitole 6. Metoda rozpoznání pomocí nástroje Tesseract je primární variantou, neboť běží přímo na serveru. Jelikož je však server umístěn na klasickém stolním počítači s běžnou konfigurací, rozhodl jsem se vyzkoušet i dvě poměrně nové cloudové služby od společnosti Microsoft a Google. Tyto služby jsou sice zpoplatněny, avšak pro účel této práce zatím postačí maximální limit 5000, resp. 1000 volných dotazů bez poplatku. Hlavním přínosem zapojení těchto technologií je rozdělení zátěže na výkonné servery třetí strany – viz obrázek 7.6. Předpokládám, že rozpoznání textu pomocí nástroje Teseract může server v případě několika souběžných požadavků téměř zahltit. Proto se pokusím v závislosti na zatížení serveru využít vždy vhodnou technologii tak, aby k takové situaci docházelo pouze ve výjimečných případech. Dalším bonusem je sjednocení aplikačních rozhraní těchto služeb, což může být výhodné pro vývojáře, kteří chtějí mít jednoduché řešení pojistky pro případ výpadku jedné ze služeb.

7.3.5 Uživatelské rozhraní webové správy

Uživatelské rozhraní tvoří pro vývojáře vstupní bránu ke všem poskytovaným vlastnostem této webové služby. Klíčem k zaujetí uživatele je tak především úvodní stránka (angl. landing page), která by měla stručně vystihnout smysl, případně účel služby a v krátkosti shrnout její nejpodstatnější vlastnosti. Jakmile se uživatel odhodlá k přihlášení do služby, je důležité nabídnout důvěryhodný a bezpečný postup registrace. Proto jsem se rozhodl použít mechanismus verifikace uživatele pomocí emailu, který je v dnešní době již zcela standardní.



Obrázek 7.6: Pokud je server zahlcen příchozími dotazy, rozdělí lokální server zatížení pomocí využití technologie *Computer Vision API*, případně *Cloud Vision API* na server společnosti Microsoft, resp. Google.

Po úspěšném procesu registrace dostane uživatel přístup k webové správě (angl. dashboard). Rozhraní této části je tvořeno třemi hlavními oblastmi – *primárním navigačním panelem*, *sekundárním navigačním panelem* a *obsahovou částí*.

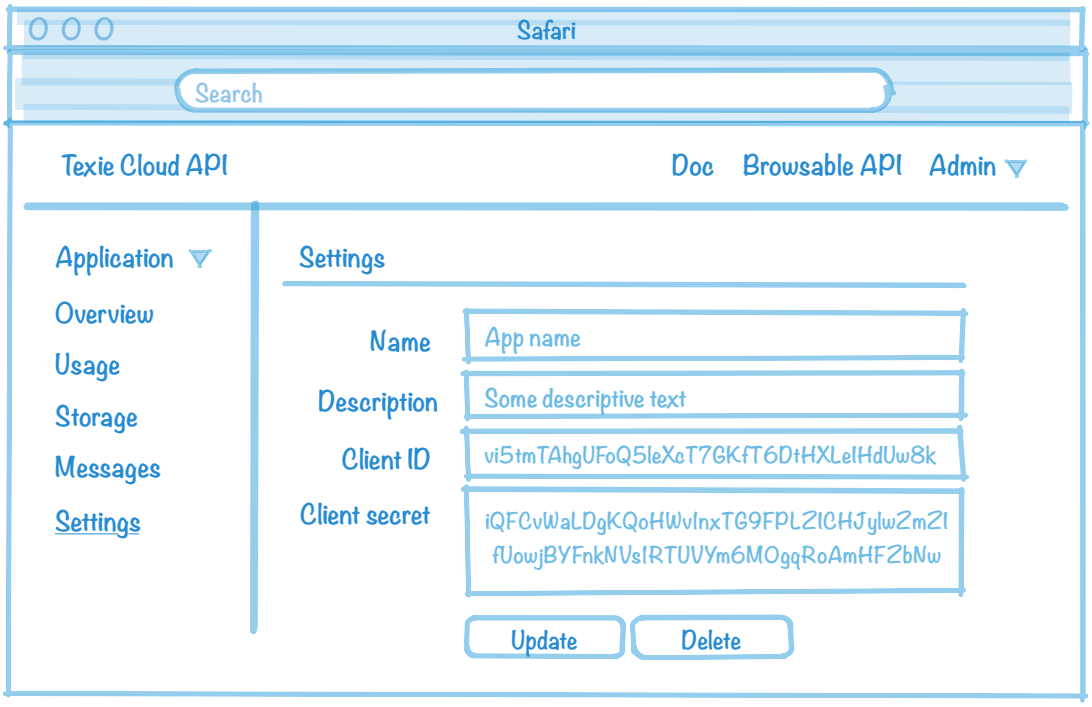
Primární navigační panel slouží k navigaci mezi top-level obrazovkami – hlavní stránkou (landing page), webovou správou (dashboard), dokumentací a interaktivním režimem aplikačního rozhraní v rámci webového prohlížeče. Dále pak panel nabízí prvek informující o aktuálně přihlášeném uživateli, který poskytuje kontextovou nabídku pro operace s uživatelským účtem (změna hesla a odhlášení).

Sekundární navigační panel slouží pouze pro účely navigace v rámci stránky *dashboard* poskytující informace o konkrétní aplikaci. V horní části se nachází kontextová nabídka pro výběr aktivní aplikace, pro kterou mají být informace zobrazeny. Součástí nabídky je také prvek, který umožňuje vytvořit novou aplikaci. Pod touto nabídkou se nachází prvky (*Overview, Usage, Storage, Messages, Settings*), které umožňují navigaci mezi jednotlivými záložkami určujícími obsah obsahové části. Stránka nazvaná *Overview* (obrázek 7.7) a *Ussage* poskytuje analytické a statistické informace. *Storage* slouží jako prohlížeč rozpoznaných snímků a *Messages* zobrazuje uživateli informační, chybové nebo varovné zprávy. Poslední stránka *Settings* (obrázek 7.8) umožňuje uživateli zobrazit informace o aktuálně aktivní aplikaci, případně ji dovoluje editovat nebo trvale odstranit.

Obsahová část vykresluje informace podle vybraného prvku v rámci sekundárního navigačního panelu. Součástí stránky *Overview, Ussage, Storage* a *Message* je pak další navigační panel, který slouží pro navigaci v rámci aktuálně zobrazeného obsahu, především pro případ potřeby stránkování obsahově rozsáhlých částí.

Ilustrace popsané kompozice a členění navigačních prvků je uvedena na obrázku 7.7, který zachycuje stav webové správy po přihlášení – aktivní je stránka *Overview*. V podobném duchu jsou navrženy také další části webové aplikace, které je možné shlédnout v příloze D.

Obrázek 7.7: Drátěný model zobrazující koncepční návrh rozhraní. Primární navigační panel se nachází v horní části, sekundární je umístěn vlevo. Obsahová část zachycuje analytické informace stránky *Overview*.



Obrázek 7.8: Drátěný model zobrazující koncepční návrh rozhraní obrazovky *Settings*. Obsahová část umožňuje uživateli zobrazit informace o aktuálně aktivní aplikaci, případně ji dovoluje editovat nebo trvale odstranit.

7.3.6 Analytický subsystém

Analytický subsystém poskytuje vývojáři dané aplikace souhrnné informace o způsobu používání koncového bodu `annotations`. Vývojář tak získá analytický nástroj pro snadné vyhodnocení počtu požadavků a latence rozpoznání, dále pak statistiky rozložení využívaných metod (Tesseract, Microsoft, Google), případně informace o zařízení, ze kterého byl požadavek odeslán. Zmíněný výčet pak doplňují procentuální informace využití procesoru a operační paměti serveru v době, kdy byl požadavek uskutečněn. Všechny zmíněné a další odvozené hodnoty je možné přehledně získat z aplikace webové správy (obrázek 7.7), přičemž některé informace budou dostupné také formou interaktivních grafů.

Pokud dojde k neočekávané události – chybě, přetížení nebo změně výchozí metody rozpoznání – vygeneruje server varovnou zprávu, která je uložena do databáze. Později může vývojář zprávu zhlédnout v uživatelském rozhraní webové správy na stránce *Messages*. Zároveň je možné mechanismus zpráv použít pro komunikaci vlastníka *Texie Cloud API* s uživatelem této služby za účelem obecnějšího, případně komerčního sdělení.

Kapitola 8

Realizace

V této kapitole popisuji realizaci klient-server služby *Texie Cloud API* pro rozpoznání textu, jejíž možnou dekompozici jsem uvedl v předcházející kapitole. Zaměřím se především na konkrétní nástroje, postupy a knihovny, které jsem v rámci implementace služby použil. Kapitola je členěna stejným způsobem jako předcházející – první část se zabývá realizací klientské aplikace *Ingredients* a souvisejícím frameworkem, další sekce pak popisuje realizaci vybraných subsystémů serverové části.

8.1 Subsystémy klientské části

Nativní mobilní aplikace pro platformu iOS je možné vyvíjet pouze na počítačích společnosti Apple s operačním systémem macOS. Nástroj, který sdružuje všechny potřené programy pro vývoj, se nazývá *Xcode*[1] – v aktuální chvíli je dostupný ve verzi 8.3.2. Xcode je integrované vývojové prostředí, které obsahuje kromě klasických nástrojů také tak zvaný *Interface Builder*, který slouží pro vývoj uživatelského rozhraní aplikací.

Ještě před několika málo roky byl hlavním programovacím jazykem pro vývoj iOS aplikací jazyk *Objective-C 2.0*, který byl však až nečekaně rychle nahrazen novým modernějším jazykem *Swift* uvedeným na konferenci WWDC roku 2014. Znalost jazyka Objective-C 2.0 se však stále hodí, neboť po internetu koluje ještě mnoho knihoven, které doposud nebyly přepsány do nového jazyka. Swift totiž umožňuje bez problému invokovat metody implementované v Objective-C 2.0 a vice versa. Nový programovací jazyk Swift za tuto krátkou dobu prošel již třemi hlavními verzemi a momentálně je dostupný ve verzi 3.1.

Knihovny rozšiřující vlastnosti a možnosti základních knihoven a frameworků (UIKit, Foundation atd.) jsou dostupné pomocí velmi rozšířeného programu pro správu balíčku *CocoaPods*[2]. Tento program umožňuje snadnou instalaci a správu verzí jednotlivých knihoven, které jsou specifikovány v souboru `Podfile`. Dalším méně rozšířeným správcem balíčků je *Carthage* a nově *Swift Package Manager*, který je však momentálně dostupný pouze pro platformu macOS. Knihovny použité v rámci aplikace *Ingredients* jsem se rozhodl spravovat pomocí programu *CocoaPods*, neboť jej podporuje téměř každá knihovna dostupná pomocí vyhledávače `cocoacontrols.com`, která sdružuje velké množství knihoven vývojářů třetích stran. V průběhu řešení se však vyskytl problém s infrastrukturou, kterou pro *CocoaPods* zajišťuje `github.com`, a i proto bych pro příští projekt zvolil raději alternativní řešení v podobě programu *Carthage*.

[1] Dostupné na: `https://itunes.apple.com/us/app/xcode/id497799835`
[2] Dostupné na: `https://cocoapods.org`

8.1.1 Uživatelské rozhraní

Pro grafickou[3] tvorbu uživatelského rozhraní včetně navigačních prvků a vztahů mezi obrazovkami slouží tzv. *Interface Builder*. Kreslící plátno, ve kterém vývojář sestavuje a dává do vztahu jednotlivé obrazovky, se nazývá *storyboard*. Obrazovka je reprezentována instancí třídy odvozené od `UIViewController`, případně `UITableViewController` a jiných. Instance třídy `UIViewControler` v rámci návrhového vzoru MVC odpovídá části *Controller*, a to co pozorujeme ve storyboardu odpovídá části *View*. Ve storyboardu je však obrazovka označena jako `ViewController`, což může být ze začátku lehce matoucí. Je však důležité si uvědomit, že *View* definuje, jak mají být data prezentována a *Controller* říká, jak a kde jsou data získána plus jakým způsobem má *View* reagovat na podněty od uživatele.

Uživatelské prostředí aplikace *Ingredients* je, stejně jako většina jiných aplikací, tvořeno pomocí několika základních ovládacích prvků, které poskytuje framework *UIKit* a prostředí *Cococa Touch*.

Instance `UILabel` je jeden z nejzákladnějších prvků uživatelského rozhraní. Účelem je zobrazit textový popisek, který je možné editovat pouze z kódu aplikace, kde je instance `UILabel` dostupná prostřednictvím speciální instanční proměnné označované jako `IBOutlet`. Při vývoji je třeba pamatovat na možnosti zpřístupnění pro slabozraké a nevidomé, které platforma iOS hluboce integruje. Proto je vhodné pro instance třídy `UILabel` vhodně nastavit druh textu (nadpis, popisek) nebo správně nadefinovat *Size Classes*, tak aby bylo zpřístupnění vždy dostupné.

Instance `UIButton` je základním prvkem pro interakci uživatele s aplikací. Invokaci metody v závislosti na výskytu konkrétní události (nejčastěji *TouchUpInside* – dotyk vnitřní části ovládacího prvku) zajišťuje dříve popsaný mechanismus *Target-Action*. Metoda *akce* je označována klíčovým slovem `IBAction`.

Instance `UIImageView` je potomkem třídy `UIView` a slouží pro vykreslení rastrových nebo i vektorových obrázků. Obrázek je reprezentován instancí třídy `UIImage` a instanci `UIImageView` je jej možné předat prostřednictvím instanční proměnné `image`. Často je potřeba vytvořit také masku, která definuje tvar výsledného obrázku. Tohoto efektu je možné dosáhnout úpravou související instanční proměnné `layer` třídy `CALayer`, která definuje výsledný vzhled každé instance `UIView`, případně umožňuje vytvořit animační vrstvu. Nejjednodušší úpravou je zaoblení rohů prostřednictvím nastavení poloměru zakřivení skrze instanční proměnnou `layer.cornerRadius`. Složitější masku lze vytvořit pomocí vrstvy `CAShapeLayer` a `CGPath`.

Samotnou kapitolu tvoří prvek třídy `UITableView`[4], který slouží pro tvorbu tabulek a v praxi vytváří layout pro podstatnou část obrazovek každé iOS aplikace. Třída `UITableView` je potomkem třídy `UIScrollView`, což v důsledku znamená, že se lze po obrazovce pohybovat ve vertikálním směru pomocí gesta *scroll*. Jelikož se skutečně jedná o jeden z nejrozšířenější prvků, poskytuje UIKit také speciální třídu `UITableViewController`, která již implicitně obsahuje objekt třídy `UITableView` jako instanční proměnnou. Současně třída zapouzdřuje také princip *delegace* a *data source*, a tudíž `UITableViewController` implicitně adoptuje protokolu `UITableViewDelegateDataSource` a `UITableViewDataSource`. Z posledního zmíněného protokolu musí objekt označený jako `dataSource` implementovat alespoň následující dvě metody.

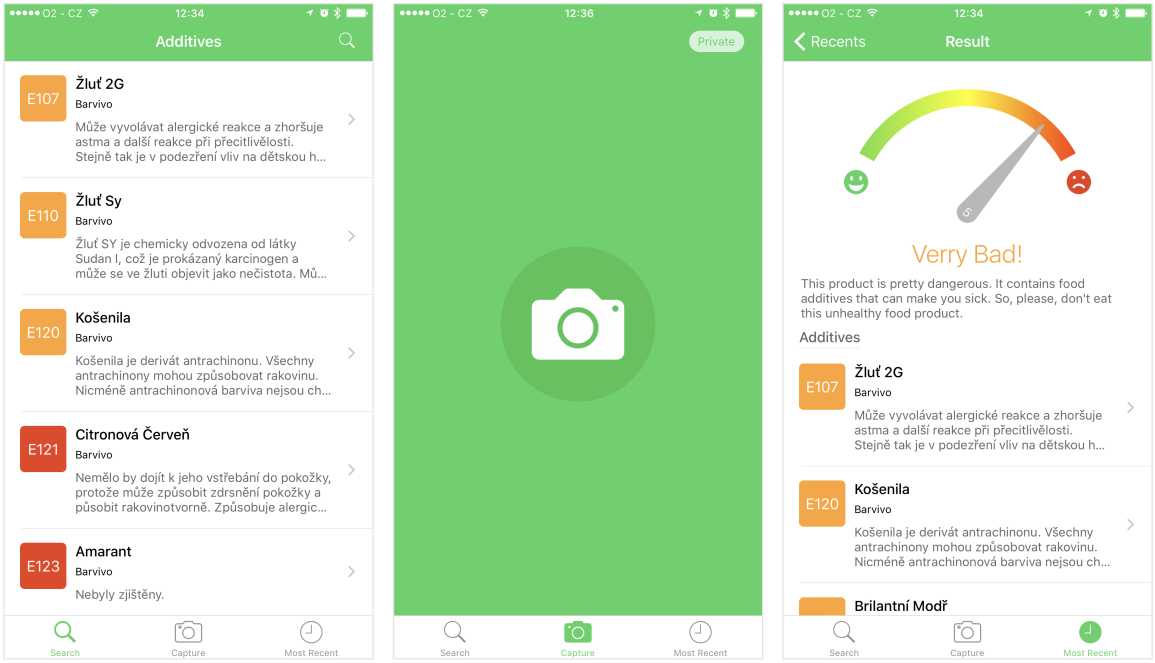
[3] Vývoj uživatelského rozhraní v grafickém prostředí je narozdíl od jiných platforem zcela běžnou praxí.
[4] Alternativou k `UITableView` je instance třídy `UIContainerView`, případně `UIStackView`.

Metoda `tableView(_:numberOfRowsInSection:)` se dotazuje objektu `dataSource`, který adoptuje protokol `UITableViewDataSource`, na počet buněk v každé sekci dané instance třídy `UITableView`.

Metoda `tableView(_:cellForRowAt:)` zjišťuje od objektu `dataSource`, jaká buňka má být na daný index vložena a jaká data bude zobrazovat. Každá buňka je instancí třídy `UITableViewCell`, případně potomkem této třídy. V základu poskytuje UIKit čtyři předdefinované typy – *Basic*, *Left-Detail*, *Right-Detail* a *Subtitle*. Často je však třeba vytvořit vlastní buňku, která obsahuje jiné specifické prvky, čehož je možné dosáhnout odvozením konkrétní třídy od rodičovské třídy `UITableViewCell`.

Vztahy a navigace mezi obrazovkami jsou v iOS definovány prostřednictvím dvou základních konceptů, kterým odpovídají třídy `UINavigationController` a `UITabBarController`. V aplikaci *Ingredients* jsem využil oba přístupy – `UITabBarController` pro navigaci mezi top-level obrazovkami *Additives*, *Capture*, *Recents* a `UINavigationController` pro zanořené obrazovky *Detail* a *Result*.

Výsledná implementace uživatelského rozhraní je zobrazena na obrázku 8.1, ostatní obrazovky je možné zhlédnout v příloze C. V rámci obrazovky 8.1c jsem použil knihovnu `DTGaugeView`[5], která poskytuje široce přizpůsobitelný prvek ve tvaru tachometru, který jsem využil pro zobrazení celkové „zdravosti“ dané potraviny.



(a) Obrazovka *Addtives* (b) Obrazovka *Capture* (c) Obrazovka *Result*

Obrázek 8.1: Finální design a implementace uživatelského rozhraní vybraných obrazovek. Pro implementaci obrazovky (a) i (c) jsem použil instanci `UITableView` a vlastní třídu pro buňky tabulky odvozené od třídy `UITableViewCell`. Obrazovka (b) poskytuje tlačítko pro spuštění akvizičního subsystému, které jsem dokreslil pulzující animací vytvořenou pomocí `CABasicAnimation` a `CALayer`.

[5]Dostupné na: https://github.com/devTechi/DTGaugeView.git

8.1.2 Akviziční subsystém

Aplikační rozhraní sytému iOS poskytuje pro akvizici snímku a videa obecně dvě základní technologie.

UIImagePickerController je třída z frameworku UIKit, která poskytuje základní možnosti pro snímání obrazu pomocí vybraného objektivu. Parametry snímání není možné pomocí tohoto rozhraní ovládat, jejich hodnota je nastavena automaticky dle dané scény. Kromě zmíněného nabízí také standardní uživatelské rozhraní pro ovládání fotoaparátu.

AVFoundation framework je framework pro audio-vizuální media dostupný pro iOS a macOS, který kromě jiného poskytuje flexibilní řešení pro plně nastavitelné a přizpůsobitelné snímání statických, ale i dynamických snímků. Parametry snímání (fokus, délka expozice a ISO) je možné nastavit manuálně, ale také automaticky dle dané scény. Na rozdíl od předchozí varianty poskytuje méně abstraktní aplikační rozhraní a neposkytuje žádné standardní uživatelské rozhraní.

Pro účel aplikace *Ingredients* se zcela jistě nabízí první technologie zajištěná pomocí třídy `UIImagePickerController`. Prezentace tohoto specifického `ViewControleru` je možná instanciací dané třídy `UIImagePickerController` a následnou invokací instanční metody `present(_:animated:completion:)`, která modálně zobrazí obrazovku pro akvizici obrazu. Podobně jako u tabulky je i u fotoaparátu zaveden princip *delegace*, který podporuje tvorbu generických prvků. Delegát instance třídy `UIImagePickerController` adoptující protokol `UIImagePickerControllerDelegate` by měl implementovat následující dvě metody.

Metoda `imagePickerController(_:didFinishPickingMediaWithInfo:)` oznamuje delegátovi, že uživatel zmáčkl tlačítko spouště a proces akvizice snímku byl dokončen. Implementující metoda delegáta by v této chvíli měla získaný snímek předat k dalšímu zpracovaní a poté zobrazující akviziční obrazovku pomocí invokace metody `dismiss` odstranit. Získaný snímek reprezentuje instance třídy `UIImage`, která je v těle této metody dostupná jako součást parametru `info` třídy `Dictionary` pod klíčem zastoupeným konstantou `UIImagePickerControllerOriginalImage`.

Metoda `imagePickerControllerDidCancel(_:)` oznamuje delegátovi, že bylo zmáčknuto tlačítko *cancel*. Ve většině případu by měla implementovaná metoda delegáta odstranit právě se zobrazující akviziční obrazovku pomocí invokace metody `dismiss`.

8.1.3 Framework pro komunikaci se serverem

Síťová komunikace se serverem *Texie Cloud API* je další klíčovou částí demonstrační aplikace *Ingredients*. Předně potřebuje aplikace komunikovat s endpointy `token` a `revoke_token`, které představují autentizační mechanismus serveru. Jakmile je zajištěna autentizace klientské aplikace, je možné použít koncový bod `annotations`, pro rozpoznání textu v konkrétním snímku.

Aplikační rozhraní platformy iOS vývojářům momentálně poskytuje dvě technologie pro síťovou komunikaci. První zastřešuje třída `NSURLConnection`, která však již není v posledních letech udržována a je označena jako *deprecated*. Náhradním řešením se stala technologie `NSURLSession`, která již přináší moderní koncepty, jako například možnost vytvořit *session*

a každou její instanci konfigurovat zvlášť. Ještě před uvedením `NSURLSession` se však z důvodu nepříliš velké obliby `NSURLConnection` začalo objevovat množství knihoven a frameworků, z nichž momentálně asi nejznámější je knihovna *Alamofire*[6]. Zmíněná knihovna poskytuje široké možnosti síťové komunikace od jednoduchých asynchronních dotazů, přes autentizační techniky a automatickou validaci, až po složitější mechanismy *Sessions* nebo *Routing*.

První verze frameworku, nebo zatím spíše knihovny pro komunikaci se serverem *Texie Cloud API*, je implementována třídou `TexieCloudService`, která je navržena podle návrhového vzoru *Singleton*. Pro správnou funkčnost je nejprve nutné nastavit identifikační údaje, které je možné získat z webové správy po registraci konkrétní aplikace.

Metoda `setCredentials(clientID: clientSecret:)` inicializuje odpovídající instanční proměnné třídy `TexieCloudService`, které jsou použity v dalších metodách. Pokud nejsou poskytnuty, upozorní vývojáře příslušná výstražná hláška.

Metoda `token(completion:)` zapouzdřuje stejnojmenný koncový bod služby *Texie Cloud API*. Vnitřně tato asynchronní metoda využívá metodu `request` z knihovny *Alamofire*. Parametr `completion`[7] typu `((token: error:) -> Void)` reprezentuje *callback* funkci, která je volána v okamžiku, kdy server odpověděl na požadavek pro získání přístupového tokenu. Pokud došlo k chybě, je možné získat bližší informace o jejím výskytu pomocí atributu `error`. V opačném případě je přístupový token dostupný skrze atribut `token`.

Metoda `revoke(completion:)` opět zapouzdřuje stejnojmenný koncový bod služby *Texie Cloud API*. Vnitřně tato asynchronní metoda využívá metodu `request` z knihovny *Alamofire*. Parametr `completion` typu `(error: Error?) -> Void)` reprezentuje *callback* funkci, která je volána v okamžiku, kdy server odpověděl na požadavek pro revokaci přístupového tokenu. Pokud došlo k chybě, je možné získat bližší informace o jejím výskytu pomocí atributu `error`.

Metoda `authenticate()` zapouzdřuje předchozí dvě metody. Existuje-li přístupový token je revokován a následně je invokována metoda `token(completion:)`. Metoda `authenticate` je určena pro jednoduché zajištění platného přístupového tokenu a je vhodné ji invokovat z metody `applicationDidBecomeActive(application:)` třídy `AppDelegate`.

Metoda `annnotate(image: store: completion:)` zapouzdřuje stejnojmenný endpoint služby *Texie Cloud API*. Vnitřně tato asynchronní metoda využívá metodu `upload` z knihovny *Alamofire* ve variantě vhodné pro MIME typ `multipart/form-data`. První parametr `image` třídy `UIImage` specifikuje snímek určený k rozpoznání textu. Další parametr `store` typu `Bool` určuje, zdali má být výsledek rozpoznání uložen na serveru, či nikoliv. Poslední parametr `completion` typu `((text: url: error:) -> Void)` reprezentuje *callback* funkci, která je volána v okamžiku, kdy server odpověděl na požadavek pro rozpoznání textu. Atribut `text` reprezentuje rozpoznaný text a atribut `url` poskytuje odkaz na obrázek pro případ kdy `store=True`. Pokud došlo k chybě, je možné získat bližší informace o jejím výskytu pomocí atributu `error`.

[6] Dostupné na: https://github.com/Alamofire/Alamofire

[7] V jazyce Swift implementováno pomocí *closure*, v jiných jazycích známé jako *lambda* funkce.

8.1.4 Databázový subsystém

Perzistentní úložiště databázového subsystému je implementováno pomocí frameworku *CoreData*. Ten vyžaduje vytvořit tzv. *Core Data Stack*, který zahrnuje potřebné instance tříd `NSManagedObjectModel`, `NSPersistentStoreCoordinator` a `NSManagedObjectContext`. Pro potřeby aplikace *Ingredients* jsem použil typ koordinátoru `NSSQLiteStoreType`, který označuje systém řízení báze dat *SQLite 3*. Význam dalších tříd a jejich bližší popis je možné nalézt v dokumentaci [54]. Jakmile je ve třídě `AppDelegate` *Core Data Stack* inicializován, je možné vytvořit entitní množiny a vazby mezi nimi pomocí nástroje *Core Data model editor*, který Xcode také poskytuje.

Přímá inicializace záznamů v databázi není možná, ale lze ji vyřešit pomocí exportu dat z CSV souboru při prvním spuštění aplikace. Tento mechanismus je implementován v třídě `AppDelegate` metodami `loadCSV` a `loadData`. Informace, zdali byla aplikace spuštěna poprvé či nikoliv, je uložena v části `UserDefaults` pod klíčem `is_loaded`.

Data, která aplikační databáze momentálně poskytuje jsou získaná z komerční stránky `zdravapotravina.cz`, neboť evropská databáze aditiv[8], na kterou odkazuje také web Státní zemědělské a potravinářské inspekce, není dlouhodobě dostupná. Z tohoto důvodu aplikační databáze poskytuje pouze omezené informace o nejrizikovějších potravinářských aditivech[9], se kterými je možné setkat se na území Evropské unie.

8.2 Subsystémy serverové části

Serverová část je dle technické specifikace realizována pomocí full-stack frameworku *Django*, který nabízí nástroje pro realizaci širokého spektra požadovaných vlastností různorodých webových služeb. Části, které Django přímo neposkytuje, jsou často dostupné jako komunitní rozšíření ve formě volně dostupných knihoven či frameworků. Příkladem je *Django REST framework*, který se zaměřuje na vývoj RESTful aplikačního rozhraní webové služby.

Framework Django je vytvořen v programovacím jazyce Python. V momentálně aktuální verzi 1.11 LTS je framework dostupný pro Python ve verzi 2 (2.7) i pro verzi 3 (3.4 až 3.6). S ohledem na budoucí kompatibilitu s plánovanou verzí Django 2.0 (pouze pro Python 3.5+), jsem pro realizaci služby zvolil Python ve verzi 3.5.2. Jako vývojové prostředí pro Python se mi již dříve osvědčilo vynikající IDE PyCharm[10] od společnosti JetBrains. Díky dostupnosti školní licence je možné toto prostředí získat i ve verzi Professional, která shodou okolností poskytuje speciální nadstavbu přímo pro vývoj Django aplikací. Často se opakující činnosti, jako například spouštění obslužných skriptů nebo nasazení (angl. deployment) na produkční server, je tak možné obsloužit přímo z rozhraní vývojového prostředí, což podstatně usnadňuje a především urychluje vývoj aplikace.

Django dále vývoj zpříjemňuje také díky vlastnímu vývojovému web serveru, který není nutné z počátku nijak složitě konfigurovat, což lze ocenit zejména při seznamování s novou technologií v počátcích vývoje. Vývojový web server také poskytuje dobře fungující mechanismus automatického sledování zdrojových kódů, který zajistí samočinný restart serveru v případě změny kódu aplikace.

[8] Dostupné na: `https://webgate.ec.europa.eu/sanco_foods/main/?event=display`
[9] Na stránce `zdravapotravina.cz` jsou označny kategorií škodlivosti 5 a 6.
[10] Dostupné na: `https://www.jetbrains.com/pycharm/`

8.2.1 Databázový subsystém

Databázový subsystém slouží pro ukládání perzistentních dat – v hierarchii návrhového vzoru MVC se tedy jedná o část *Model*. Subsystém je v Djangu velmi dobře implementován pomocí techniky objektově relačního mapování – ORM. Již v základu je tak podporován systém pro řízení báze dat od společnosti Oracle nebo dobře známé SQLite3, MySQL a PostgreSQL. Výchozím SŘBD v Djangu je SQLite3, které jsem se také rozhodl použít v rámci implementace *Texie Cloud API*.

Jednotlivé entitní množiny z dříve uvedeného ER diagramu – viz obrázek 7.5, jsou díky ORM definovány jako třídy, které dědí chování ze třídy `django.db.models.Model`. Atributy entitní množiny jsou reprezentovány pomocí instančních proměnných třídy `Field`. Textové pole je například reprezentováno instancí třídy `CharField`, datum je možné reprezentovat pomocí instance třídy `DateTimeField`. Vazby mezi entitními množinami jsou definované jako instance třídy `ForeignKey` nebo například `OneToOneField`. Entitní množinu, resp. třídu *User*, Django poskytuje již implicitně v rámci autentizačního subsystému `django.contrib.auth`.

Jako součást ORM poskytuje Django také databázově nezávislé aplikační rozhraní, které implementuje základní CRUD operace, ale i řešení pro složitější databázové dotazy. Entitu je možné vytvořit jednoduše pomocí konstruktoru dané třídy, pro uložení do databáze je pak navíc nutné nad vytvořeným objektem invokovat metodu `save`. Jednotlivé entity dané entitní množiny je možné získat jako instanci třídy `QuerySet` prostřednictvím třídní proměnné `objects`, nad kterou je invokována metoda `all`. Pro složitější dotazy je určena metoda `filter`, `aggregate`, `annotate` a další.

8.2.2 Serializační subsystém

Data z databázového systému je v případě REST aplikačního rozhraní třeba serializovat, tuto oblast však již samotné Django neřeší. Naštěstí ale existuje již několikrát zmíněný *Django REST framework*[11], který byl stvořen za účelem snazšího a flexibilního vývoje RESTful webových služeb.

Jako *serializer* je v Django REST frameworku (zkráceně DRF) chápán objekt, který přesně popisuje, jak mají být jednotlivé atributy entitní množiny serializovány. *Serializer* je vždy konkrétní implementací generické třídy `Serializer`, případně její variace `ModelSerializer` a dalších z modulu `rest_framework.serializers`. Atributy entitní množiny jsou reprezentovány, podobně jako v případě *modelu*, pomocí instančních proměnných třídy `Field`, avšak Django `Field` není totéž co DRF `Field`. DRF specifikuje vlastní druhy, jako `EmailField` nebo `URLField`, které jsou více konkrétní a poskytují sémantickou informaci o daném atributu. O mapování instančních proměnných *modelu* na instanční proměnné *Serializeru* se v případě použití generické třídy `ModelSerializer` není třeba starat. Stačí jen poskytnout meta třídu, která definuje ke kterému *modelu* se *serializer* vztahuje, a které atributy mají být serializovány.

Serializer umožňuje nejen serializaci dat z databáze, ale i deserializaci dat například z těla dotazu od klienta. V případě deserializace *serializer*, poskytuje také mechanismus validace dat a to nejen na úrovni celkové struktury, ale i v rámci formátu jednotlivých atributů.

Souborový formát, do kterého mají být data serializována, definuje tzv. *renderer*. Ve výchozím nastavení je v DRF možné použít pouze formát JSON, ale v případě potřeby

[11] Dostupné na: http://django-rest-framework.org

je možné pomocí knihoven renderer rozšířit i o XML, YAMP, JSONP aj. Druh rendereru klient určí obvyklým způsobem – buď pomocí položky `Accept` v hlavičce dotazu, nebo prostřednictvím přípony za daným koncovým bodem v URL (`.json`, `.xml`). V rámci služby *Texie Cloud API* je momentálně plně dostačující JSON renderer, který je však v případě potřeby možné rychle doplnit o další.

8.2.3 Autentizační subsystém

Django, a tím pádem i DRF, v základu poskytuje autentizační mechanismy *Basic HTTP Authentication* a *Session Authentication*, které jsou vhodné především pro webové aplikace s uživatelským rozhraním. Pro RESTful webovou službu se však tyto metody příliš nehodí, a proto DRF navíc nabízí *Token Authentication*. Dnes se již však stává téměř standardem autentizační mechanismus vybudovaný nad *OAuth 2.0*, který jsem podrobněji popisoval v předcházející kapitole 7. DRF tento mechanismus sice přímo neposkytuje, avšak oficiálně doporučuje knihovnu *Django OAuth Toolkit*[12], se kterou je plně kompatibilní.

Django OAuth Toolkit (DOAT) je knihovna pro Django poskytující téměř „plug and play“ autentizační mechanismus OAuth 2.0 využívající knihovnu *OAuthLib* implementovanou v plném souladu se zněním RFC 6749 [53]. DOAT poskytuje předdefinované všechny klíčové prvky – potřebné koncové body pro získání přístupového tokenu, autentizační logiku i související databázový model. Entitní množina *Application* dle navrženého ER diagramu 7.5 se však v atributech neshoduje s výchozími v modelu DOAT. Proto bylo třeba vytvořit vlastní třídu `Application` vycházející z abstraktní třídy `AbstractApplication`, která doplňuje požadované atributy `description` a `created`.

Autentizace prostřednictvím mechanismu OAuth 2.0 je v rámci služby *Texie Cloud API*, použita výhradně pro přístup klienta k REST API serveru. Související webové aplikace pro správu služby a interaktivní aplikační rozhraní využívají mechanismus *Session Authentication*, který je pro tento účel vhodnější.

8.2.4 Aplikační rozhraní služby

Aplikační rozhraní v Djangu, respektive v DRF je definováno v rámci návrhového vzoru MVC v části *View*. Tímto se Django částečně vymyká obecně chápanému významu *View*, které v obecném znění určuje, jak a pomocí kterých komponent mají být data uživateli prezentována. *View* je tedy v případě Djanga nutné chápat spíše jako část *Controller*.

View je v DRF možné definovat pomocí *function-based view, class-based view* nebo prostřednictvím nejabstraktnějšího přístupu *view set*. V nejjednodušším případě *function-based view* jde o klasickou Python funkci doplněnou dekorátorem `@api_view(list)`, jehož parametr `list` určuje, na které HTTP metody (např. `GET`, `POST`) má takto označené *View* odpovídat. Tento druh jsem v případě rozhraní *Texie Cloud API* použil u koncového bodu `endpoints`, který vrací seznam všech implementovaných koncových bodů.

Další koncový bod `annotations` je definován pomocí nejabstraktnějšího přístupu *view set*. Abstraktní třída `ViewSet` definuje základní metody pro CRUD operace. V konkrétní třídě je pak možné základní chování změnit pomocí redefinice dané metody. V případě koncového bodu `annotations` jsem tento postup použil u metody `create`, která implementuje rozpoznání textu. V metodě se nejprve deserializuje příchozí požadavek a následně se provede validace získaných dat. Poté je zjištěno aktuální zatížení serveru pomocí

[12] Dostupné na: https://github.com/evonove/django-oauth-toolkit

knihovny *psutil*[13], na jehož základě se zvolí vhodná metoda rozpoznání textu – Tesseract, Microsoft Computer Vision nebo Google Cloud Vision. Pokud je pomocí parametru `annotations/?method=tesseract` požadováno rozpoznání pomocí metody Tesseract a zatížení CPU nebo operační paměti přesahuje 90 %, je požadavek odmítnut. Pouze pokud je rozpoznání ponecháno s výchozím parametrem `auto`, je při přetížení požadavek zpracován pomocí služby Microsoft Computer Vision. V případě parametru `microsoft` nebo `google` je požadavek vždy obsloužen požadovaným způsobem.

Odpověď na dotaz je tvořena pomocí instance třídy `Response`, která umožňuje tzv. *content-type negotiation* pro vyjednání požadovaného formátu odpovědi, na jehož základě se použije odpovídající renderer (`JSONRenderer`, `BrowsableAPIRenderer`). RESTful aplikační rozhraní je možné implementovat i pomocí samostatného Djanga pomocí standardního *function-base view* vracející instanci `JsonResponse`, v tomto případě je ale nutné serializaci, deserializaci a další potřebné kroky provézt v rámci vlastní režie. Koncové body `token` a `revoke_token` poskytnuté pomocí *Django OAuth Toolkit* využívají právě tohoto přístupu.

URL adresy jednotlivých koncových bodů jsou v Djangu, resp. DRF specifikovány pomocí modulu `URLconf`. Ten definuje jedinou proměnnou `urlpatterns`, která obsahuje seznam instancí `django.conf.urls.url`. Tyto instance popisují mapování URL vzorů pomocí regulárních výrazů na Python funkce (*Views*) – viz doplňující zdrojový kód 8.1. Pro mapování koncových bodů popsaných pomocí přístupu *view set*, je v DRF možné použít instanci třídy `SimpleRouter`, která automaticky vytvoří `URLconf` modul se všemi CRUD operacemi. Do seznamu `urlpatterns` jej lze následně začlenit pomocí funkce `include` z modulu `django.conf.urls`.

Kód 8.1: Ilustrace mapování URL adresy na *View* Python funkci v modulu `URLconf`.

```
from django.conf.urls import url
from api import views

urlpatterns = [
    url(r'^endpoints/$', views.api_root, name="api-root"),
    url(r'^dashboard/apps/(?P<app_id>[0-9]+)/delete/', views.dashboard_delete)
    ...
]
```

Autentizační subsystém spojuje konkrétní požadavek s identitou jisté aplikace, ale přístupová práva ke koncovým bodům nijak neupravuje. V případě bodu `annotations` je však žádoucí, aby byl přístup povolen pouze registrovaným uživatelům (potažmo aplikacím) a pro ostatní zamítnut. Přístupová práva jsou v DRF definována pomocí přístupových tříd. Základní třídy, jako `AllowAny`, `IsAuthenticated` nebo `IsAuthenticatedOrReadOnly`, jsou v základu definovány v modulu `rest_framework.permissions`. Pokud je ale použit jiný autentizační systém než ten, který DRF nabízí, je nutné nadefinovat vlastní přístupové třídy. V *Django OAuth Toolkit* jsou však tyto třídy již předdefinovány. Přístupová práva se v rámci *class-based view* nastavují pomocí instanční proměnné `permission_classes`, přičemž je možné definovat i více než jen jednu přístupovou třídu. Pro koncový bod `annotations` je použita třída `IsAuthenticatedOrTokenHasScope`, která umožňuje nejen autentizaci prostřednictvím *OAuth 2.0*, ale i pomocí výchozí *Session Authentication*. U *function-based view* jsou přístupová práva definována pomocí dekorátoru `@permission_classes()`.

[13]Dostupné na: https://github.com/giampaolo/psutil

8.2.5 Subsystém pro rozpoznání textu

Subsystém pro rozpoznání textu je vývojářům dostupný skrze koncový bod `annotations`. Snímek z těla dotazu je zpracován pomocí Python knihovny *Pillow*[14], která podporuje všechny standardní obrazové formáty. Reprezentace snímku prostřednictvím instance třídy `Image` je následně předána konkrétní funkci pro rozpoznání textu, která je specifikována pomocí parametru `method` koncového bodu.

Metoda `auto` zajišťuje automatický výběr metody rozpoznání textu dle aktuálního zatížení serveru. Výchozí metodou je `tesseract`. Pokud je však server příliš zatížen, jsou přijaté požadavky implicitně zpracovány alternativní metodou `microsoft`.

Metoda `tesseract` je hlavním nástrojem pro rozpoznání textu. Tesseract[15] je aktuálně dostupný ve verzi 3.04 a v linuxové distribuci Ubuntu jej lze nainstalovat pomocí konzolového programu `apt-get` příkazem `install tesseract-ocr`. Aplikační rozhraní Tesseractu je momentálně dostupné pouze v jazyce C/C++. Naštěstí existují volně dostupné knihovny, které aplikační rozhraní zpřístupňují i z jiných programovacích jazyků. Pro Python jsem zvolil knihovnu *tesserocr*[16], která podporuje Python 3.5 a zdá se být ze všech dostupných nejlépe udržovaná. Jakmile jsou všechny potřebné nástroje nainstalované a dostupné, je implementace rozpoznání textu víceméně přímočará.

Metoda `microsoft` je alternativní metodou pro rozpoznání textu. Využívá služby *Computer Vision* ze skupiny Microsoft Cognitive Services, která je momentálně ve verzi 1.0 Beta. Aplikační rozhraní služby je dostupné skrze RESTful koncový bod `ocr`, jehož parametrem je atribut `language` pro specifikaci jazyka a `detectOrientation` pro detekci a korekci rotace textu ve snímku. Tělo dotazu je tvořeno binární reprezentací snímku. Pro sestavení a zaslání dotazu koncovému bodu jsem použil Python knihovnu *Requests: HTTP for Humans*[17]. Odpověď je dostupná ve formátu JSON.

Metoda `google` je další alternativní metodou pro rozpoznání textu. Využívá služby *Cloud Vision API* ze skupiny Google Cloud Platform, která je momentálně ve verzi 1.1 Beta. Aplikační rozhraní služby je dostupné nejen jako REST API, ale i formou klientských knihoven, mezi nimiž je i knihovna *google-cloud-vision* pro Python. Rozpoznání textu implementuje metoda `detect_full_text`, která má jediný parametr `language_hints` pro specifikaci jazyka. Odpověď je dostupná ve formě kolekce instancí třídy `EntityAnnotation`.

8.2.6 Uživatelské rozhraní webové správy

Webová správa a analýza využití aplikačního rozhraní služby je implementována jako standardní Django aplikace z čehož vyplývá, že základní principy pro zpracování požadavků od klienta (v tomto případě webového prohlížeče) jsou velmi podobné DRF aplikaci. *View* je možné definovat jako *function-based view* nebo *class-based view*. Mapování konkrétních *View* k jednotlivým URL adresám se provátí zcela totožným způsobem jako v případě *View* pro koncové body ve výše uvedené sekci 8.2.4.

[14] Dostupné na: https://github.com/python-pillow/Pillow
[15] Dostupné na: https://github.com/tesseract-ocr/tesseract
[16] Dostupné na: https://github.com/sirfz/tesserocr
[17] Dostupné na: https://github.com/kennethreitz/requests

Odlišný je již ale způsob formulace odpovědi. Od webové aplikace se totiž očekává bohatá, vizuálně přitažlivá a nejlépe dynamicky generovaná odpověď ve formátu HTML. Django tento požadavek uspokojuje pomocí *šablonového systému*, který v obecné hierarchii MVC nejlépe odpovídá části *View*.

Šablona je HTML kód reprezentující statické části doplněný o speciální konstrukce jazyka *Django Template Language* (DTL) – viz doplňující kód 8.2, který popisuje, jak má být dynamický obsah začleněn do statického. DTL definuje syntaxi pro vkládání obsahu pomocí proměnných, řídící konstrukce, filtry (pro formátování hodnoty proměnných) a další užitečné značky (tagy). Syntaxi jazyka je možné rozšířit také o vlastní značky, čehož jsem využil k definici vlastního filtru pro převod znakového řetězce na hodnotu typu `int`. Další velmi užitečnou vlastností jazyka DTL je princip *šablonové dědičnosti*. Tato vlastnost umožňuje vybudovat výchozí skeleton, který obsahuje všechny základní elementy stránky (např. navigační panel), a definovat *bloky*, které mohou potomci této *generické šablony* redefinovat. Šablonovou dědičnost jsem v rámci webové správy použil pro definici základní struktury, kterou jsem rozdělil na *primární navigační panel*, *sekundární navigační panel* a *obsahovou část* – viz sekce 7.3.5.

Kód 8.2: Příklad jednoduché generické šablony. Hodnota parametru `href` je definována pomocí DTL značky `url`, která dynamicky vkládá relativní cestu k danému *View*. Dále je v šabloně použita značka podmíněného příkazu `if` a definice bloku pomocí značky `block`.

```
<!DOCTYPE html>
<html lang="en">
...
<body>
   <nav id="main-nav">
      <a class="page-scroll" href="{% url 'api-docs:docs-index' %}">Doc</a>
      {% if user.is_authenticated %}
         <a class="page-scroll" href="{% url 'api-root' %}">Browsable API</a>
      {% endif %}
   </nav>
   <div id="content">
      {% block content %}{% endblock %}
   </div>
</body>
</html>
```

Odpověď webové aplikace na dotaz je tvořena pomocí instance třídy `HttpResponse` z modulu `django.http`. Jelikož se ovšem jedná o velmi častou operaci, je pro tvorbu této instance definována pomocná funkce `renderer` z modulu `django.shortcuts`. Parametrem funkce je objektová reprezentace požadavku, cesta k souboru dané šablony a kontext. *Kontext* je proměnná typu `dict`, která slouží pro předání hodnot požadovaných proměnných z Python kódu do šablony. Řídící konstrukce a značky, které jsou definovány jazykem DTL, jsou při požadavku o vykreslení interpretovány pomocí *šablonového enginu*. V této fázi jsou všechny proměnné nahrazeny jejich hodnotou známou z kontextu. Dále jsou vyhodnoceny všechny řídící konstrukce a na závěr je provedena kompozice výsledného HTML dokumentu v souvislosti se šablonovou dědičností.

Často se vyskytujícím prvkem webových stránek je formulář pro vstup dat od uživatele. Příkladem použití formuláře v rámci webové správy *Texie Cloud API* je například stránka pro vytvoření nové aplikace, případně pro její editaci a téměř všechny stránky spojené s registrací a autentizací uživatele. Django, jako full-stack framework, poskytuje řadu

nástrojů a knihoven nejen pro snadnou tvorbu formulářů, ale také pro verifikaci a zpracování zaslaných formulářových dat. Základní nástroj pro tvorbu formuláře je třída `Form` z modulu `django.forms`, která je vhodná pro tvorbu obecných formulářů bez návaznosti na *model*. Pokud je však formulář spojen s konkrétní třídou *modelu*, je vhodnější použít třídu `ModelForm` ze stejného modulu. Princip práce s formuláři je velmi podobný *serializéru*, který jsem popisoval v sekci 8.2.2. Instanční proměnné konkrétní třídy odvozené od `Form` nebo `ModelForm` reprezentují jednotlivé položky daného formuláře. Položka HTML formuláře `input` typu `text` například odpovídá instanční proměnné třídy `CharField` atd. V případě formuláře spojeného s konkrétní třídou modelu vytvořeného pomocí třídy `ModelForm` je mapování typů položek *modelu* na formulář provedeno automaticky. Stačí jen pomocí meta třídy definovat, ke kterému *modelu* formulář patří a případně které položky mají být použity.

Pro stylování stránek je možné uplatnit známé techniky, knihovny a frameworky jako pro vývoj statických HTML stránek. V případě webové správy *Texie Cloud API* jsem použil kombinaci frameworku Bootstrap, šablony pro úvodní stránku New Age[18] a vlastních CSS stylopisů. Formuláře je možné snáze stylovat pomocí knihovny *Crispy*[19], která nabízí široké možnosti přizpůsobení nejen vzhledu, ale i rozložení jednotlivých vstupních polí a dalších formulářových prvků.

8.2.7 Analytický subsystém

Analytický subsystém poskytuje informace o způsobu používání aplikačního rozhraní *Texie Cloud API*. Při každém příchozím `POST` požadavku ke koncovému bodu `annotations` je vytvořena entita entitní množiny *Request*, která uchovává atributy popsané v tabulce 8.1. Z těchto atributů jsou pak pomocí databázových dotazů skrze Django ORM systém získány statisticky a analyticky významné hodnoty, které jsou prezentovány uživateli v uživatelském rozhraní webové správy. Získané statistické informace jsou v rozhraní rozděleny do záložek *Requests*, *Latency*, *Methods* a *Operating Systems*.

Atribut	Význam
`latency`	Určuje dobu zpracování příchozího požadavku včetně rozpoznání textu v příchozím snímku. Měření je provedeno pomocí knihovny *time*.
`device`†	Popisuje druh zařízení, ze kterého byl požadavek odeslán (např. iPhone, iPad, Generic Smartphone).
`os`†	Udává operační systém rozpoznaného zařízení (např. iOS, Android).
`os_version`†	Specifikuje verzi rozpoznaného operačního systému (např. 10.3, 5.0.1).
`date`	Zachycuje datum a čas příchozího požadavku.
`cpu_usage`	Udává aktuální využití procesoru serveru v době zpracování daného požadavku. Měření je provedeno pomocí knihovny *psutil*.
`memory_usage`	Udává aktuální využití operační paměti serveru v době zpracování daného požadavku. Měření je provedeno pomocí knihovny *psutil*.

Tabulka 8.1: Popis jednotlivých atributů entitní množiny *Request*. Atributy označené křížem jsou získány prostřednictvím knihovny *django-user-agents* dostupné přes program `pip`.

[18] Dostupné na: https://github.com/BlackrockDigital/startbootstrap-new-age
[19] Dostupné na: https://github.com/django-crispy-forms

Requests poskytuje informace o celkovém počtu dotazů, udává průměrný, minimální a maximální počet dotazů za posledních 30 dní. Formou sloupcového grafu je prezentována informace o celkovém počtu dotazů v posledních třech měsících a pomocí čárového grafu je možné sledovat počet dotazů v jednotlivých dnech za posledních 30 dní.

Latency nabízí informace o průměrné době rozpoznání v celkovém období existence aplikace, případně v posledních 30 dnech. Dále je uvedena maximální a minimální latence. Formou sloupcového, resp. čárového grafu je možné sledovat průměrnou latenci v posledních třech měsících i v průběhu jednotlivých dní.

Methods poskytuje informace o nejčastěji použité metodě rozpoznání a procentuální zastoupení jednotlivých metod za období existence aplikace. Formou sloupcového grafu jsou prezentovány počty použití jednotlivých metod v posledních třech měsících a pomocí koláčového grafu je zobrazen procentuální podíl jednotlivých metod za posledních 30 dní.

Operating Systems nabízí informace o operačním systému klientů využívající koncový bod `annotations`. Zobrazeny jsou informace o nejvíce a nejméně častému systému společně s koláčovým grafem reprezentujícím procentuální zastoupení operačních systému za posledních 30 dní.

Grafy jsou vykresleny pomocí výborné open-source knihovny *Charts.js*[20], pro kterou vytvořil Matthisk Heimensen wrapper *django-jchart*[21] pro Python, resp. Django. Wrapper poskytuje abstraktní třídu `Chart`, jež deklaruje dvě metody `get_datasets` a `get_labels`, které definují data a označení jednotlivých veličin v grafu. Pomocí této abstraktní třídy je možné vytvořit konkrétní třídu, která může reprezentovat čárový, sloupcový, ale i další typ grafu – viz dokumentace ke knihovně *Charts.js*.

[20] Dostupné na: http://chartjs.org
[21] Dostupné na: https://github.com/matthisk/django-jchart

Kapitola 9

Testování

V této závěrečné kapitole se zabývám testováním zejména serverové části. V první části kapitoly popisuji jednotlivé oblasti, které je v rámci služby *Texie Cloud API* vhodné sledovat. Následně prezentuji výsledky provedených testů a popisuji podmínky, za kterých testování probíhalo. Na závěr shrnuji získané poznatky a seznamuji s kroky, na které se bude nutné v průběhu dalšího vývoje aplikace zaměřit.

9.1 Návrh testování a vhodné metody

Testování klient-server aplikace je velmi rozsáhlou a časově náročnou činností. Jen na straně serveru je nutné zhodnotit mnoho aspektů, které se prolínají napříč mnoha obory informačních technologií od *hardwaru* přes *síťové služby* a *informační systémy* až po *uživatelská rozhraní*. Podobně lze na různých úrovních přistoupit i k jednotlivým subsystémům serveru a také ke klientské části.

9.1.1 Serverová část

V rámci serverové části je možné sledovat tři základní oblasti – testování výkonu, bezpečnosti a uživatelského rozhraní.

Testování výkonu (angl. performace testing) spadá pod kategorii nefunkcionálního testování[1]. Jeho cílem je zjistit výkonnost a projevy chování serveru při různé zátěži. V rámci testování se pak zkoumá responzivita, škálovatelnost, spolehlivost nebo například míra využití systémových i infrastrukturních zdrojů. Výkonové testování je dále možné rozdělit na následující oblasti. [55]

Load Testing poskytuje informace o chování serveru v případě velkého množství současně se dotazujících klientů. V průběhu testování se postupně zvyšuje počet současně se dotazujících klientů, přičemž se nejčastěji sleduje latence a propustnost serveru. Tyto metriky je pak možné doplnit také o využití systémových zdrojů – CPU, operační paměti a diskových I/O operací. Jakmile je dosažen práh schopnosti serveru, hodnoty metrik se začnou zhoršovat a současně začne server generovat chybové stavy 429 (příliš mnoho požadavků) a 5xx (chyba serveru). Tímto testováním je možné odhalit úzká místa softwaru a limity hardwaru i infrastruktury.

[1] Zkoumá aspekty, které nesouvisí s aplikačně specifickým chováním serveru.

Stress Testing na rozdíl od předchozího testování, zatíží server extrémním množstvím současně se dotazujících klientu, jejichž počet je za hranicí hardwarových a softwarových možností. V takovém případě se zkoumá, jak rychle systém kolabuje a jaká je jeho schopnost návratu do provozního stavu.

Spike Testing zkoumá chování serveru v případě rychlých a extrémních změn v počtu současně se dotazujících klientů. Zkoumá se především odolnost systému vůči těmto rychlým změnám a schopnost škálovatelnosti.

Soak Testing je opakem předchozího testování. Po dobu dlouhodobého očekávaného zatížení serveru, se sleduje využití operační paměti a případné úniky paměti, degradace výkonu diskových polí nebo například průběh teploty procesoru.

Testování uživatelského rozhraní souvisí s webovou aplikací serveru. Ve většině případů je nutné oslovit skupinu relevantních uživatelů, na které bude průběh testování vyzkoušen. Typicky je skupina rozdělena do dvou částí, přičemž první je na úvod s aplikací seznámena a druhá ne. Následně jsou uživatelé podle sestaveného plánu vyzváni ke splnění jednotlivých úkonů. V průběhu plnění je sledován čas, sekvence operací, množství provedených chyb atd. Testování webové aplikace je pak vhodné doplnit také validací zdrojových souborů uživatelského rozhraní (HTML, CSS) a testem kompatibility s různými prohlížeči.

Bezpečnost webové služby a souvisejícího rozhraní, případně webové aplikace, je vhodné řešit již v počátcích vývoje – pro kritické oblasti je vhodné použít standardní bezpečnostní protokoly a postupy. Testování bezpečností je založené na důkladné znalosti daného systému a analýze možných bezpečnostních rizik, na jejichž základě je možné sestavit testovací plán. Webové aplikace typicky ohrožují například následující útoky. [56]

SQL Injection je metoda napadení databázové vrstvy skrze nezabezpečenou aplikační vrstvu. Útočník typicky zneužije formulářová data, které jsou v aplikační vrstvě sloučena do podoby výsledného SQL dotazu bez tzv. *escapování* znaků.

Cross Site Request Forgery je technika, při které útočník poskytne uživateli odkaz na falešnou stránku, přičemž se předpokládá, že je uživatel již přihlášen do systému skutečné stránky. V takovém případě může útočník způsobit pomocí škodlivého dotazu (typicky při odesílání formulářových dat) nenávratné škody, za které fiktivně nese zodpovědnost oklamaný uživatel.

9.1.2 Klientská část

Klientskou část je primárně vhodné ověřit pomocí testů uživatelského rozhraní, které se téměř nikterak neliší od výše zmíněného popisu. V rámci platformy iOS lze pro testování uživatelského rozhraní využít metodu *UI Testing*, která je součástí frameworku *XCTest*. Ten dovoluje vytvořit automatizované testování pomocí záznamu činností uživatele na skutečném zařízení. Získaný záznam je po ukončení snímání převeden na instrukce programovacího jazyka, které je následně možné doplnit o požadované *assertions*. S uživatelským rozhraním částečně souvisí i *testování uživatelské zkušenosti*, které se snaží zjistit pocitové rozpoložení uživatele při i po použití dané aplikace. Měření uživatelské zkušenosti je možné například pomocí rozhovoru, dotazníku, systému sledující pohled uživatele nebo třeba sledování mozkové aktivity pomocí přístroje EEG. Další oblastí klientské aplikace vhodné pro otestování je framework *TexieCloudService* pro komunikaci se serverem. Test je možné provést například pomocí metody *Unit Testing*, která je v iOS podporována frameworkem *XCTest*.

9.2 Load&Stress Testing

Z výše popsaných oblastí jsem pro testování výkonu webové služby zvolil *Load Testing* a *Stress Testing*. Testování probíhalo při nasazení na standardní web server *Apache* s rozšířením o modul `mod_wsgi`, který dovoluje běh Python aplikací. Jako nástroj pro testování a analýzu získaných výsledků jsem vybral program *Apache JMeter*[2] ve verzi 3.2, který umožňuje simulovat zatížení serveru různým počtem současně se dotazujících klientů. V programu je nejprve nutné vytvořit tzv. *Test Plan* a následně *Thread Group*, což je definovaná skupina vláken simulující jednotlivé klienty. Dále je nutné specifikovat *Sampler* (popisuje typ požadavku – např. HTTP, FTP), a *Listener* (definuje formát výsledku měření – graf, tabulka, strom). Následně je z obslužného skriptu takto vytvořený *Test Plan* spuštěn pro různý počet vláken vždy po dobu 15 sekund. Vizualizace dále uvedených výsledků je vytvořena pomocí modifikovaného skriptu[3], který volně poskytl Dylan Tack ze společnosti Metal Toad. Kompletní *Test Plan* a obslužný skript je možné nalézt na přiloženém CD.

Výsledky získané simulací je však nutné vztáhnout vzhledem ke konkrétnímu zařízení, které bylo pro simulaci zátěže použito. V mém případě se jednalo o zařízení Macbook Pro Mid 2009 s procesorem Intel Core 2 Duo 2,53 GHz a pamětí 4 GB 1067 MHz DDR3. Zařízení bylo připojeno bezdrátově k síti o rychlosti 85/4 Mbit/s.

9.2.1 Koncový bod `annotations`

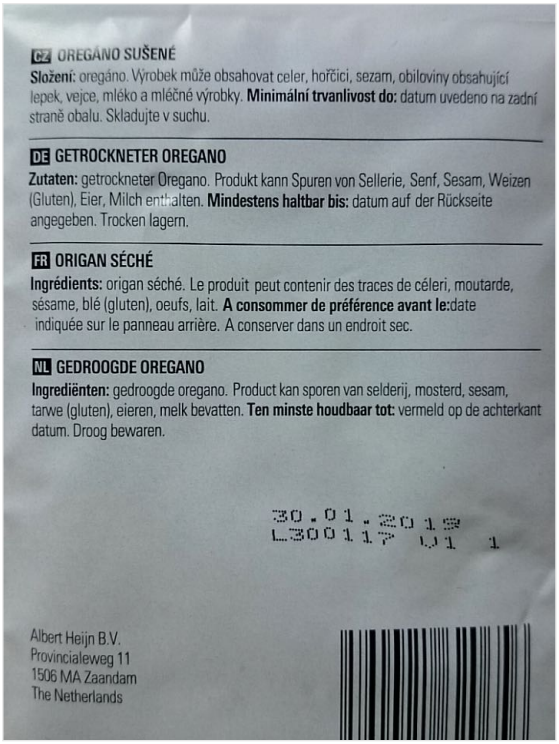
Testování tohoto koncového bodu je velkou měrou závislé na velikosti použitého snímku a kvalitě dostupné konektivity. *Doba odezvy* (angl. response time) je vyjádřena jako rozdíl času mezi přijetím odpovědi od serveru a zasláním požadavku na server. Z toho vyplývá, že menší snímek bude rychleji odeslán a výsledná doba odezvy tak bude úměrně velikosti kratší.

Snímek	Rozlišení	Velikost	Metoda	Doba rozpoznání [ms]	Doba odezvy [ms]
A	724×540	44kB	Tesseract	390	606
			Microsoft	901	1191
			Google	594	916
B	1024×800	103kB	Tesseract	1132	1487
			Microsoft	1478	1899
			Google	1801	2593
C	720×960	100kB	Tesseract	8320	8644
			Microsoft	1590	1911
			Google	1994	2309
D	720×960	76kB	Tesseract	8473	8838
			Microsoft	1314	1615
			Google	2221	2540

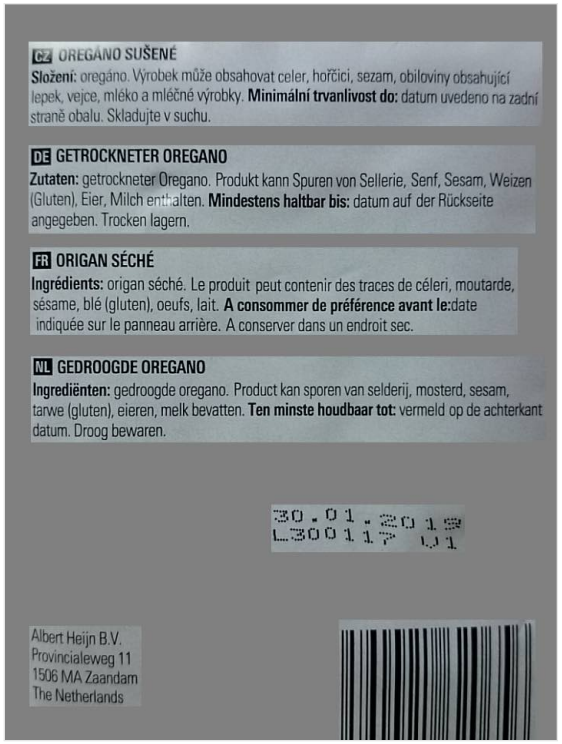
Tabulka 9.1: Ilustrace doby rozpoznání a odezvy pomocí všech implementovaných metod. Uvedené hodnoty odpovídají vždy průměrné hodnotě získané z 10 měření.

[2] Dostupné na: http://jmeter.apache.org
[3] Dostupné na: https://www.metaltoad.com/blog/plotting-your-load-test-jmeter

Další velmi významnou složku doby odezvy tvoří samotné rozpoznání textu. Tesseract dosahuje v závislosti na rozlišení snímku, množství textu v obraze a dalších parametrech širokého rozptylu *doby rozpoznání*, která také podstatně ovlivňuje výslednou dobu odezvy. Bližší podrobnosti o době rozpoznání je možné nalézt ve výše uvedené tabulce 9.1.



(a) Snímek C (`original.jpg`)



(b) Snímek D (`preprocessed.jpg`)

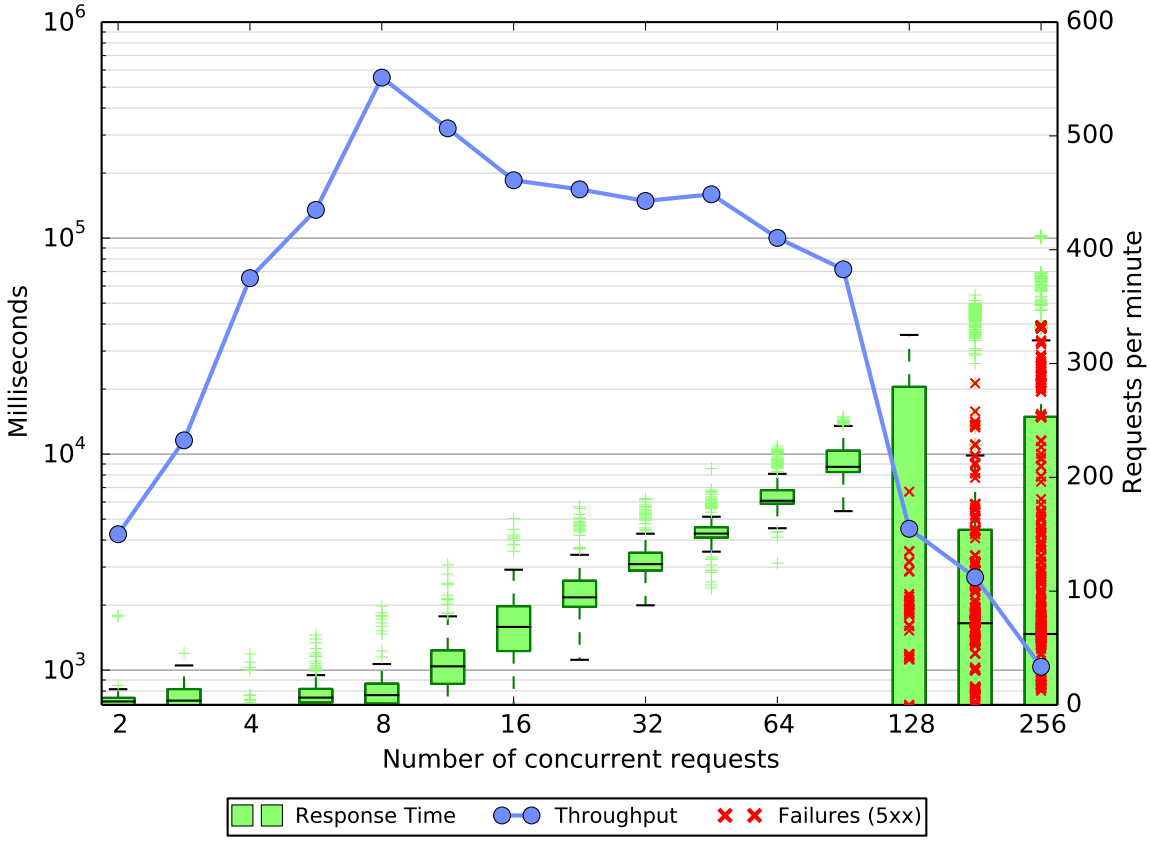
Obrázek 9.1: Ilustrace dvou snímků použitých pro testování. Jejich autorem je J. Tomešek, který se ve své diplomové práci [52] zabývá předzpracováním textu pro OCR systémy. Souhrn všech použitých snímků je možné nalézt v příloze F.

Následující tzv. *krabicové grafy* (angl. box plots) prezentují dobu odezvy pomocí kvartilů[4]. Spodní část boxu představuje kvartil Q_1 a horní Q_3, mezi nimi se nachází linie určující medián. Linie vycházející ze spodní a horní části se označují jako *vousy* (angl. whiskers) a v tomto konkrétním případě označují $1,5$ násobek *interkvartilního rozsahu* ($IQR = Q_3 - Q_1$) pod kvartilem Q_1 a nad kvartilem Q_3 – viz rovnice 9.1, resp. 9.2. Zelené body označené křížem reprezentují data, která jsou mimo zmíněný rozsah. Červené body reprezentují chyby serveru s kódem 5xx (Internal Server Error, Service Unavailable, Gateway Timeout). Osa y pro hodnoty doby odezvy (vlevo) je zobrazena v logaritmickém měřítku, počet požadavků (vpravo) je zobrazen lineárně. Testování je provedeno pouze prostřednictvím metody Tesseract a to především z důvodu přečerpání volného limitu na počet požadavků u metody Google Cloud Vision a Microsoft Computer Vision.

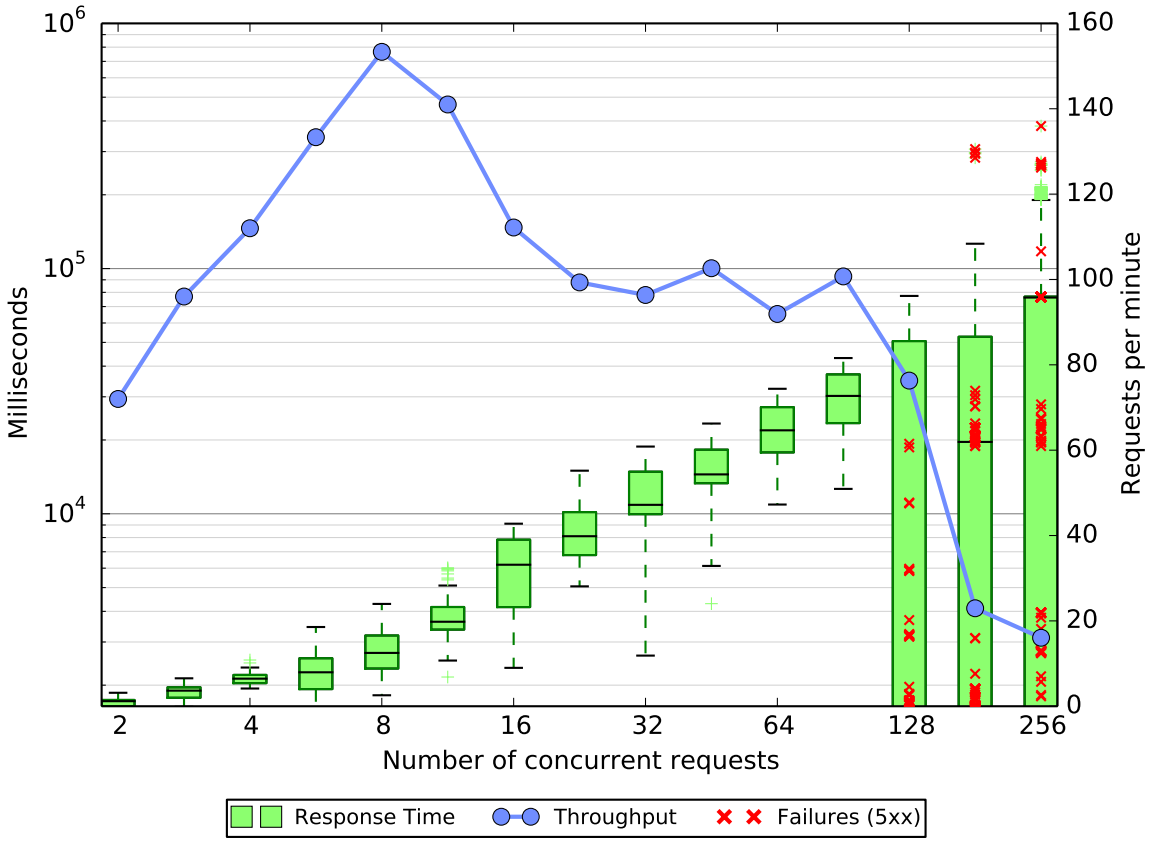
$$Lowerfence = Q_1 - 1,5 \times IQR \tag{9.1}$$

$$Upperfence = Q_3 + 1,5 \times IQR \tag{9.2}$$

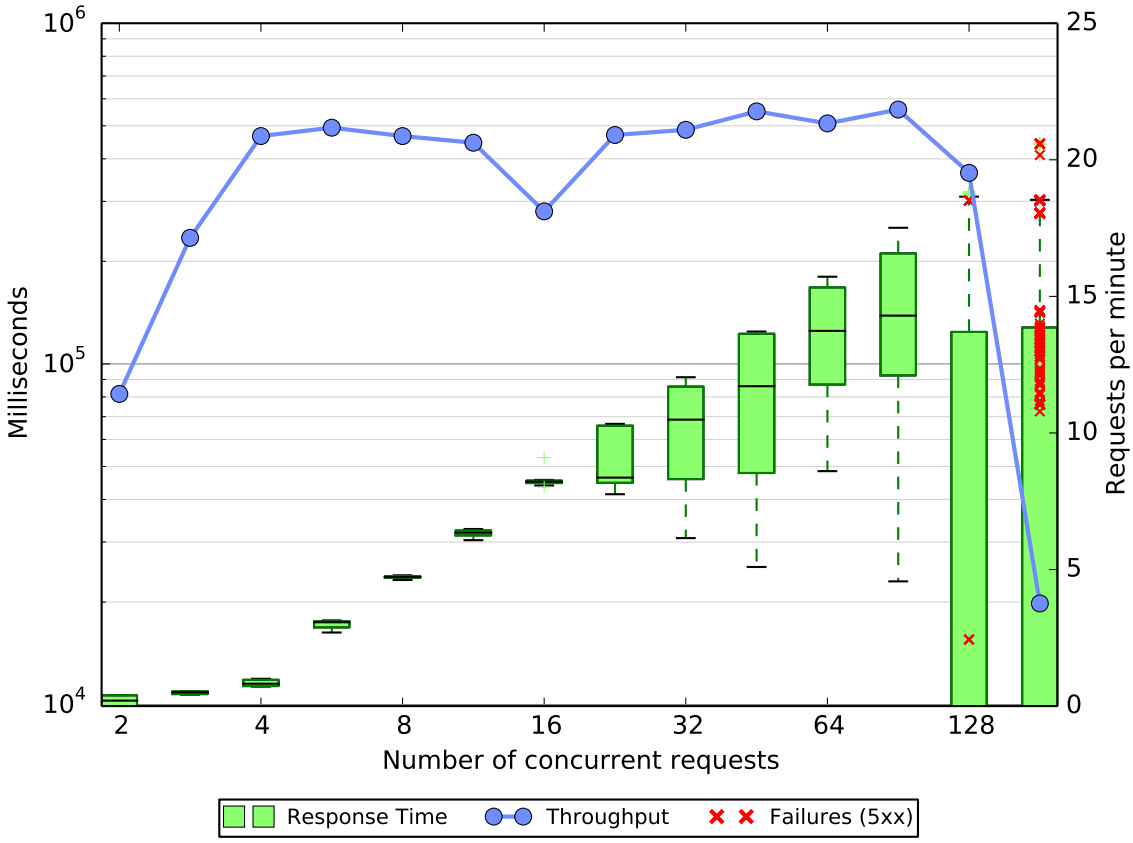
[4]Tři kvartily rozdělují statistický soubor seřazených dat na čtyři rovnocenné skupiny.



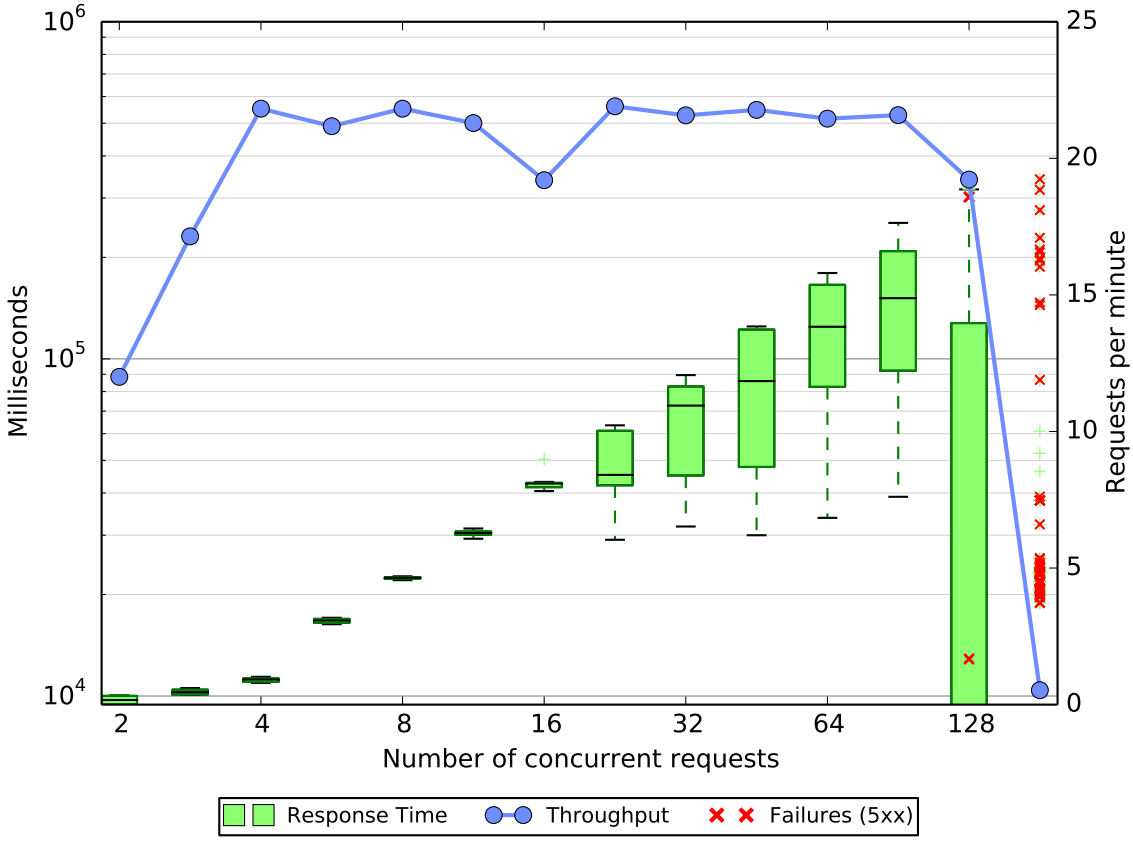
Obrázek 9.2: V případě velmi naivního snímku A je možné dosáhnout vysoké propustnosti, která v maximu činí zhruba 550 požadavků za minutu. Doba odezvy se při současném požadavku od 8 klientů pohybuje kolem 0.8 s, zatímco procesor serveru je v tomto okamžiku zatížen asi na 90 % a operační paměť je využita z 11 %. Následně se začně propustnost pomalu snižovat, přičemž až do 96 současně příchozích požadavků se její hodnota pohybuje kolem 400 požadavků za minutu. Při 128 požadavcích se začne vyskytovat chyba 503 (Service Unavailable), která se dále projevuje stále častěji, až při 256 požadavcích dosáhne propustnost pouze 25 úspěšných požadavků za minutu. V tomto okamžiku je server téměř nedostupný. Serverová příčina této chyby, kterou je možné dohledat v log souboru Apache2 (`error.log`), zní následovně – *Resource temporarily unavailable: Unable to connect to WSGI daemon process*. Význam tohoto hlášení se mi nepodařilo dohledat, předpokládám však, že se jedná o projev přetížení serveru velkým množstvím dotazů, případně o limitní hranici současných přístupů do databáze *SQLite3*. Jakmile však nápor klientů opadne, je server schopný vrátit se zpět do výchozího stavu a bez potíží přijímat nové požadavky.



Obrázek 9.3: V případě rozsáhlejšího avšak vhodně předzpracovaného a binarizovaného textu snímku B je možné dosáhnout stále dobré propustnosti, která v maximu činí 150 požadavků za minutu. Doba odezvy se při současném požadavku od 6 klientů pohybuje kolem 2 s, přičemž procesor serveru je v tomto okamžiku zatížen na 100 % a operační paměť je využita z 12 %. Následně se začně propustnost rychle snižovat, přičemž až do 96 současně příchozích požadavků se její hodnota pohybuje kolem 100 požadavků za minutu. Při 128 požadavcích se začne opět vyskytovat chyba 503 (Service Unavailable), která se dále projevuje stále častěji až při 256 požadavcích dosáhne propustnost téměř k nule. V tomto okamžiku je server téměř nedostupný, avšak po opadnutí náporu je server schopný vrátit se zpět do výchozího stavu a bez potíží přijímat nové požadavky.



Obrázek 9.4: Pro nepředzpracovaný snímek C se propustnost pohybuje celkem stabilně kolem 21 požadavků za minutu. Drobný pokles je možné pozorovat při 16 současně se dotazující klientech, kdy propustnost klesne pod 20 požadavků za minutu. Příčina toho jevu není známa. Doba odezvy se poměrně rychle zvyšuje od 10 s (při 3 současných požadavcích) až po 150 s (při 96 požadavcích). Při 128 požadavcích se začne vyskytovat chyba 503 (Service Unavailable) a 504 (Gateway Timeout), která se dále projevuje stále častěji až při 181 požadavcích dosáhne propustnost téměř k nule. V tomto okamžiku je server téměř nedostupný. Chyba 504 (Gateway Timeout) se při 128 požadavcích projeví 3× a při 181 požadavcích 28×. V konečném důsledku výskyt této chyby signalizuje, že server nebyl schopen odpovědět do 300 s, což je výchozí hodnota nastavení web serveru Apache2. Jakmile však nápor klientů opadne, je server schopný vrátit se zpět do výchozího stavu a bez potíží přijímat nové požadavky.



Obrázek 9.5: V případě předzpracovaného snímku D se propustnost pohybuje stejně jako u nepředzpracovaného snímku celkem stabilně kolem 21 požadavků za minutu. Při 16 současně se dotazující klientech je možné pozorovat opětovné zakolísaní, kdy propustnost klesne pod 20 požadavků za minutu. Příčina toho jevu není známa. Následující průběh je již srovnatelný s předchozím případem.

9.3 Diskuze získaných výsledků

Ze získaných výsledků je možné pozorovat, že i běžný[5] stolní počítač zvládne dobře zpracovat požadavky od 64 souběžně se dotazujících klientů. Následně se výkon serveru rychle propadá až je zhruba při 128 souběžných požadavcích téměř zcela přetížen. V průběhu testování se také ukázalo, že i přes velký počet požadavků (256) se server po opadnutí náporu zvládne vrátit zpět do provozuschopného stavu bez potřebného zásahu správce serveru.

Velkým překvapením je pak velmi vysoká doba rozpoznání pomocí metody Tesseract u nepředzpracovaných snímků – viz tabulka 9.1. Rozpoznání textu ve snímku C trvá téměř 8,4 s. Abych vyloučil vlastní chybu při implementaci serveru, případně nevhodný výběr Python modulu *tesserocr*, zkusil jsem rozpoznání textu totožného snímku pomocí standardního konzolového programu `tesseract` při stejné hardwarové i softwarové konfiguraci. Měření jsem v tomto případě provedl pomocí programu `time`. Výsledná naměřená hodnota se v průměru pohybovala kolem 8,1 s, což odpovídá dříve zmíněné době rozpoznání skrze aplikační rozhraní serveru.

Z výsledků také vyplývá, že předzpracování snímku pouhou detekcí textu není **pro urychlení rozpoznání** dostačující[6]. U metody Tesseract a Google se dokonce v případě snímku C a jeho předzpracované varianty D projevilo drobné zhoršení výsledné doby odezvy. Pouze v případě metody Microsoft se doba odezvy snížila zhruba o 300 ms.

[5] Procesor Intel Core i5-2500 3,30GHz, 8GB RAM.

[6] Přednosti předzpracování pomocí detekce textu blíže popisuje diplomová práce J. Tomeška [52].

Kapitola 10

Závěr

Cílem této práce byla realizace univerzálně použitelného klient-server systému pro rozpoznání textu a implementace demonstrační aplikace pro mobilní platformu iOS. Osobním cílem bylo poznání, mnou donedávna neobjevené, oblasti webových aplikací a webových služeb, které se dnes již zcela jistě stávají součástí každé moderní mobilní aplikace. Oba stanovené cíle se podařilo pokořit, ba co víc, první cíl se podařilo přesáhnout díky realizaci webové správy poskytující realizační a analytické zázemí pro vývojáře.

Rozpoznání textu a tvorbu mobilních aplikací jsem prostudoval a zkráceně popsal v kapitolách 4 a 5. Studijní oblast jsem dle domluvy s vedoucím práce rozšířil o webové služby a aplikace, s čímž esenciálně souvisí také podstata klient-server modelu. Získané informace a znalosti z těchto oblastí jsou popsány v kapitolách 2 a 3. Podrobný návrh systému pro rozpoznání včetně webové aplikace pro správu služby je popsán v kapitole 7. Implementace systému a demonstrační iOS aplikace je rozebrána v realizační kapitole 8. Způsob testování a dosažené výsledky jsou diskutovány v kapitole 9.

Výsledné serverové řešení je implementováno prostřednictvím současných, ale i zcela nových technologií. Rozpoznání textu je realizováno pomocí tří metod – knihovny Tesseract a čerstvě uvedených webových služeb Google Cloud Vision a Microsoft Computer Vision. Autentizace klientů je řešena prostřednictvím mechanismu založeného nad OAuth 2.0, který v současnosti používají všechny známé služby, příkladem je Facebook nebo Twitter. Společně s webovou aplikací a prezentační stránkou, vytváří celý systém komplexní balíček pro vývojáře poohlížející se po možnosti rozpoznání textu pomocí nástroje třetí strany. Demonstrační aplikace *Ingrediens* slouží pro rozpoznání aditivních látek v potravinách. Informace je netradičně získána přímo z textu etikety, což oproti přístupu za použití čárového kódu poskytuje vždy správné a aktuální složení potraviny.

Při realizaci jsem se dostal k neuvěřitelně širokému spektru činností – ať už se jednalo o zprovoznění vlastního serveru, výběr vhodných nástrojů pro rozpoznání textu, implementaci RESTful webové služby, návrh designu webové aplikace, studium použitých knihoven, metody testování a nespočet dalších činností souvisejících s klientskou i serverovou částí. V průběhu realizace mě nejvíce překvapila velmi dlouhá doba rozpoznání textu pomocí knihovny Tesseract, která je však i dle zkušeností jiných vývojářů v normě. Přesto jsem však hledáním řešení strávil velké množství času.

Další vývoj by měl tedy směřovat k akceleraci rozpoznání textu, ať už cestou vhodnějšího předzpracování snímku přímo na serveru nebo akceptací budoucí majoritní verze knihovny Tesseract 4.0. Pro reálné nasazení systému je dále nutné navrhnout a dopracovat monetizační subsystém, stanovit ekonomický rámec a zajistit odpovídající hardwarové vybavení, případně pronajmout výpočetní výkon v datovém centru.

Literatura

[1] Oluwatosin, H. Client-Server Model. In *Journal of Computer Engineering*. Malaysia: International Organization Of Scientific Research, 2014. S. 67–71. ISSN 2278-0661.

[2] A Bit History of Internet: Client-Server. *Wikibooks: Open books for an open world* [online]. Wikibooks, 2003 [cit. 2016-12-23]. Dostupné na: https://en.wikibooks.org/wiki/A_Bit_History_of_Internet/Chapter_5_:_Client-Server.

[3] Anatomy of the Client/Server Model. *BEA: Documentation for the e-generation* [online]. Redwood City: Oracle, c2001 [cit. 2016-12-24]. Dostupné na: https://docs.oracle.com/cd/E13203_01/tuxedo/tux80/atmi/intbas3.htm.

[4] Moltchanov, D. Client/server and peer-to-peer models: Basic concepts. *Tampere University of Technology* [online]. Tampere, 2013 [cit. 2016-12-24]. Dostupné na: http://www.cs.tut.fi/kurssit/ELT-53206/lecture01.pdf.

[5] Evans, C., Lacey, D., Harvey, D. et al. *Client/Server: A Handbook of Modern Computer Design*. 1st ed. London: Prentice Hall, 1995. ISBN 0133772012.

[6] Albreshne, A., Fuhrer, P. a Pasquier, J. *Web Services Technologies – State of the Art: Definitions, Standards, Case Study* [online]. Fribourg: Université de Fribourg, Department of Informatics, 2009 [cit. 2016-12-25]. Dostupné na: https://diuf.unifr.ch/drupal/softeng/sites/diuf.unifr.ch.drupal.softeng/files/file/publications/internal/WP09-04.pdf.

[7] Suda, B. *SOAP Web Services*. Edinburgh: University of Edinburgh, School of Informatics, 2003. Master of Science. Dostupné na: http://suda.co.uk/publications/MSc/brian.suda.thesis.pdf.

[8] Adamczyk, P., Smith, P. H., Johnson, R. E. et al. *REST and Web Services: In Theory and in Practice* [online]. New York, NY: Springer, 2011 [cit. 2016-12-25]. S. 35–57. ISBN 9781441983022. Dostupné na: http://link.springer.com/10.1007/978-1-4419-8303-9_2.

[9] Fielding, R. *Architectural Styles and the Design of Network-based Software Architectures*. Irvine: University of California, 2000. Disertační práce. Dostupné na: http://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf.

[10] Uniform Resource Identifier (URI): Generic Syntax. *IETF Tools: IETF-related tools, standalone or hosted on tools.ietf.org* [online]. Fremont: IETF [cit. 2017-01-06]. Dostupné na: https://tools.ietf.org/html/rfc3986.

[11] *A Guide to REST and API Design* [online]. CS200-110010. New York: CA Technologies, c2015 [cit. 2016-12-29]. Dostupné na: http://transform.ca.com/rs/catech/images/A%20Guide%20to%20REST%20and%20API%20Design.pdf.

[12] Aulds, C. *Linux Apache web server administration*. 1st ed. San Francisco: Sybex, c2001. ISBN 0782127347.

[13] Rodriguez, A. RESTful Web services: The basics. *IBM developerWorks* [online]. New York: IBM Corporation, 2008 [cit. 2016-12-29]. Dostupné na: https://www.ibm.com/developerworks/webservices/library/ws-restful/.

[14] What is web server – a computer OR a program? *Web Developers Notes* [online]. c2016 [cit. 2016-12-24]. Dostupné na: http://www.webdevelopersnotes.com/what-is-web-server.

[15] What is a web server? *Mozzila Developer Network* [online]. Mountain View: Mozzila Corporation, c2005-2016 [cit. 2016-12-20]. Dostupné na: https://developer.mozilla.org/en-US/docs/Learn/Common_questions/What_is_a_web_server.

[16] Kabir, M. J. *Apache Server 2 bible*. 1st ed. New York: Hungry Minds, c2002. ISBN 0764548212.

[17] November 2016 Web Server Survey. *Netcraft* [online]. Bath: Netcraft, 2016 [cit. 2016-12-21]. Dostupné na: https://news.netcraft.com/archives/2016/11/22/november-2016-web-server-survey.html.

[18] Holovaty, A., Kaplan Moss, J. a Gilmore, J. *The definitive guide to Django: Web development done right*. 1st ed. New York: Springer, c2008. ISBN 9781430203315.

[19] Samtani, G. a Sadhwani, D. *Web Services Business Strategies and Architectures: Web Services and Application Frameworks (.NET and J2EE)*. 1st ed. Berkeley: Apress, 2002. S. 273–289. ISBN 9781430253563.

[20] Forcier, J., Bissex, P. a Chun, W. *Python Web development with Django*. 1st ed. Upper Saddle River: Addison-Wesley, c2009. ISBN 9780132356138.

[21] Find your new favorite web framework: Measuring web framework popularity so you can find interesting frameworks to check out. *HotFrameworks* [online]. [cit. 2016-12-21]. Dostupné na: http://hotframeworks.com.

[22] Salas Zárate, M., Alor Hernández, G., Valencia García, R. et al. Analyzing best practices on Web development frameworks: The lift approach. *Science of Computer Programming* [online]. 2015, vol. 102 [cit. 2016-12-21]. S. 1–19. ISSN 01676423. Dostupné na: http://linkinghub.elsevier.com/retrieve/pii/S0167642314005735.

[23] Plekhanova, J. *Evaluating web development frameworks: Django, Ruby on Rails and CakePHP* [online]. Philadelphia: Temple University, September 2009 [cit. 2016-12-21]. Dostupné na: http://ibit.temple.edu/wp-content/uploads/2011/03/ibitwebframeworks.pdf.

[24] Meet Django. *Django: The web framework for perfectionists with deadlines.* [online]. Lawrence: Django Software Foundation, c2005-2016 [cit. 2016-12-21]. Dostupné na: https://www.djangoproject.com.

[25] Christie, T. Django REST framework. [online]. c2011-2016 [cit. 2016-12-21]. Dostupné na: http://www.django-rest-framework.org.

[26] Getting Started with Rails. *Rails Guides* [online]. [cit. 2016-12-21]. Dostupné na: http://guides.rubyonrails.org/getting_started.html.

[27] Kehoe, D. What is Ruby on Rails? *RailsApps* [online]. Last updated 11 October 2013 [cit. 2016-12-21]. Dostupné na: http://railsapps.github.io/what-is-ruby-rails.html.

[28] Gibbs, N. Why Do They Say Rails Doesn't Scale? *Codefolio* [online]. Jan 4th, 2015 [cit. 2016-12-21]. Dostupné na: http://codefol.io/posts/why-do-they-say-rails-doesnt-scale.

[29] Template Engines. *The Ruby Toolbox* [online]. Hamburg, 2009 [cit. 2016-12-21]. Dostupné na: https://www.ruby-toolbox.com/categories/template_engines.

[30] Kitura: A high performance and simple to use web framework for building modern Swift applications. [online]. New York: IBM Corporation [cit. 2016-12-22]. Dostupné na: http://www.kitura.io.

[31] Nelson, T. a Wright, L. Vapor Documentation. *Vapor (Web Framework for Swift)* [online]. Qutheory, 2016 [cit. 2016-12-22]. Dostupné na: https://vapor.github.io/documentation/#vapor-documentation.

[32] About Perfect. *Server-side Swift – Perfect* [online]. Newmarket: PerfectlySoft, c2015-2016 [cit. 2016-12-22]. Dostupné na: https://www.perfect.org/about.html.

[33] Collins, R. Benchmarks for the Top Server-Side Swift Frameworks vs. Node.js. [online]. [cit. 2016-12-22]. Dostupné na: https://medium.com/@rymcol/benchmarks-for-the-top-server-side-swift-frameworks-vs-node-js-24460cfe0beb#.vxagroo4a.

[34] Eikvil, L. *Optical Character Recognition* [online]. 1993 [cit. 2017-02-10]. Dostupné na: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.217.4980&rep=rep1&type=pdf.

[35] Patel, C., Patel, A. a Patel, D. Optical character recognition by open source OCR tool tesseract: A case study. *International Journal of Computer Applications* [online]. October 2012, roč. 55, č. 10 [cit. 2017-02-10]. S. 50–56. ISSN 0975-8887. Dostupné na: https://www.researchgate.net/profile/Chirag_Patel27/publication/235956427_Optical_Character_Recognition_by_Open_source_OCR_Tool_Tesseract_A_Case_Study/links/00463516fa43a64739000000.pdf.

[36] Due Trier Øivind, Jain, A. K. a Taxt, T. Feature extraction methods for character recognition-A survey. *Pattern Recognition* [online]. 1996, roč. 29, č. 4 [cit. 2017-02-10]. S. 641–662. ISSN 0031-3203. Dostupné na: http://www.sciencedirect.com/science/article/pii/0031320395001182.

[37] Ravina, M., Supriya, I. a Nilam, D. Optical character recognition. *International Journal of Recent Technology and Engineering* [online]. 2013, roč. 2, č. 1 [cit. 2017-02-10]. S. 72–75. Dostupné na: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.673.8061&rep=rep1&type=pdf.

[38] Willis, N. Google's Tesseract OCR engine is a quantum leap forward. *Linux.com: News for the Open Source Professional* [online]. San Francisco: Linux Foundation, 2006 [cit. 2017-02-12]. Dostupné na: https://www.linux.com/news/googles-tesseract-ocr-engine-quantum-leap-forward.

[39] Dhiman, S. a Singh, A. Tesseract Vs Gocr A Comparative Study. *International Journal of Recent Technology and Engineering* [online]. 2013, roč. 2, č. 4 [cit. 2017-02-12]. S. 80–83. ISSN 2277-3878. Dostupné na: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.676.9685&rep=rep1&type=pdf.

[40] Release Notes. *Tesseract Open Source OCR Engine* [online]. Mountain View: Google, 2017 [cit. 2017-02-12]. Dostupné na: https://github.com/tesseract-ocr/tesseract/wiki/ReleaseNotes.

[41] Heliński, M., Kmieciak, M. a Parkoła, T. *Report on the comparison of Tesseract and ABBYY FineReader OCR engines* [online]. [b.m.]: Poznań Supercomputing and Networking Center, 2012 [cit. 2017-02-12]. Dostupné na: http://lib.psnc.pl/Content/358/PSNC_Tesseract-FineReader-report.pdf.

[42] ABBYY FineReader 14: Your documents in action. [online]. Moskva: ABBYY, c2017 [cit. 2017-02-12]. Dostupné na: https://www.abbyy.com/media/13509/8143_fr14_brochure_black_2.pdf.

[43] Cloud Vision API: Derive insight from images with our powerful Cloud Vision API. [online]. Mountain View: Google, 2016 [cit. 2017-02-12]. Dostupné na: https://cloud.google.com/vision/.

[44] Nguyen, D. Using Python 3 + Google Cloud Vision API's OCR to extract text from photos and scanned documents. *GitHubGist* [online]. 2016 [cit. 2017-02-12]. Dostupné na: https://gist.github.com/dannguyen/a0b69c84ebc00c54c94d.

[45] Computer Vision API: Extract rich information from images to categorize and process visual data. [online]. Redmond: Microsoft, 2016 [cit. 2017-02-12]. Dostupné na: https://www.microsoft.com/cognitive-services/en-us/computer-vision-api.

[46] Computer Vision API Reference. *Microsoft* [online]. Redmond: Microsoft, 2016 [cit. 2017-02-12]. Dostupné na: https://westus.dev.cognitive.microsoft.com/docs/services/56f91f2d778daf23d8ec6739/operations/56f91f2e778daf14a499e1fc.

[47] Bobák, P. *Aplikace pro hodnocení kvality pokrmů na iOS*. Brno: Vysoké učení technické v Brně, Fakulta informačních technologií, 2015. Bakalářská práce.

[48] Concepts in Objective-C Programming. *Apple Developer* [online]. Cupertino: Apple, 2012 [cit. 2017-02-19]. Dostupné na: https://developer.apple.com/library/ios/documentation/General/Conceptual/CocoaEncyclopedia/Model-View-Controller/Model-View-Controller.html.

[49] Hegarty, P. Developing iOS 7 Apps for iPhone and iPad. [online]. Stanford: Stanford University, 2013 [cit. 2017-02-19]. Dostupné na: https://itunes.apple.com/us/course/lecture-1-slides/id733644550.

[50] Harrison, S., Wilson, P., Abrams, W. et al. Server Side Swift vs. The Other Guys 2: Speed. *Qutheory* [online]. [cit. 2016-12-22]. Dostupné na: https://medium.com/@qutheory/server-side-swift-vs-the-other-guys-2-speed-ca65b2f79505.

[51] Anketa: Složení potravin. *DNES.cz: Zprávy, kterým můžete věřit* [online]. Praha: MAFRA, 2010 [cit. 2017-04-01]. Dostupné na: http://ekonomika.idnes.cz/ankety.aspx?idanketa=A20100215_spi_983.

[52] Tomešek, J. *Mobilní systém pro rozpoznání textu na Androidu*. Brno: Vysoké učení technické v Brně, Fakulta informačních technologií, 2017. Diplomová práce.

[53] D., H. *RFC 6749: The OAuth 2.0 Authorization Framework* [online]. [b.m.]: Internet Engineering Task Force (IETF), October 2012 [cit. 2017-04-06]. Dostupné na: https://tools.ietf.org/html/rfc6749.

[54] Guides and Sample Code: Core Data Programming Guide. *Apple Developer* [online]. Cupertino: Apple, 2017 [cit. 2017-04-20]. Dostupné na: https://developer.apple.com/library/content/documentation/Cocoa/Conceptual/CoreData/index.html.

[55] What is Performance Testing? *Software Testing Class* [online]. c2017 [cit. 2017-04-21]. Dostupné na: http://www.softwaretestingclass.com/what-is-performance-testing/.

[56] Application Security Testing of Thick Client Applications. [online]. Chicago: Infosec Institute, 2013 [cit. 2017-04-21]. Dostupné na: http://resources.infosecinstitute.com/application-security-testing-of-thick-client-applications/.

Přílohy

Příloha A

Specifikace koncových bodů

/token/			
POST	Headers	Authorization	Specifikuje identitu klienta prostřednictvím klíčů client_id a client_secret. Údaje jsou zakódovány algoritmem Base64. Hodnota parametru je pak tvořena typem (Basic) a získaným řetězcem.
	Body	grant_type	Určuje variantu přístupu dle frameworku OAuth 2.0. V případě služby *TexieCloud API* bude tato hodnota vždy nastavena na client_credentials.
	Response	access_token	Nově vygenerovaný přístupový token.
		expires_in	Určuje dobu expirace vygenerovaného tokenu.
		token_type	Typ přístupového tokenu. V případě služby *TexieCloud API* bude tato hodnota vždy Bearer.
		scope	Rozsah platnosti získaného tokenu. V případě služby *TexieCloud API* není tato hodnota zatím potřebná.

Tabulka A.1: Podrobný popis koncového bodu token.

/revoke_token/			
POST	Body	token	Specifikuje token, který má být revokován.
		client_id	Specifikuje aplikaci pomocí client_id, v rámci které má být uvedený token revokován.
		token_type_hint	Specifikuje typ tokenu. V případě služby *TexieCloud API* bude tato hodnota vždy nastavena na access_token.

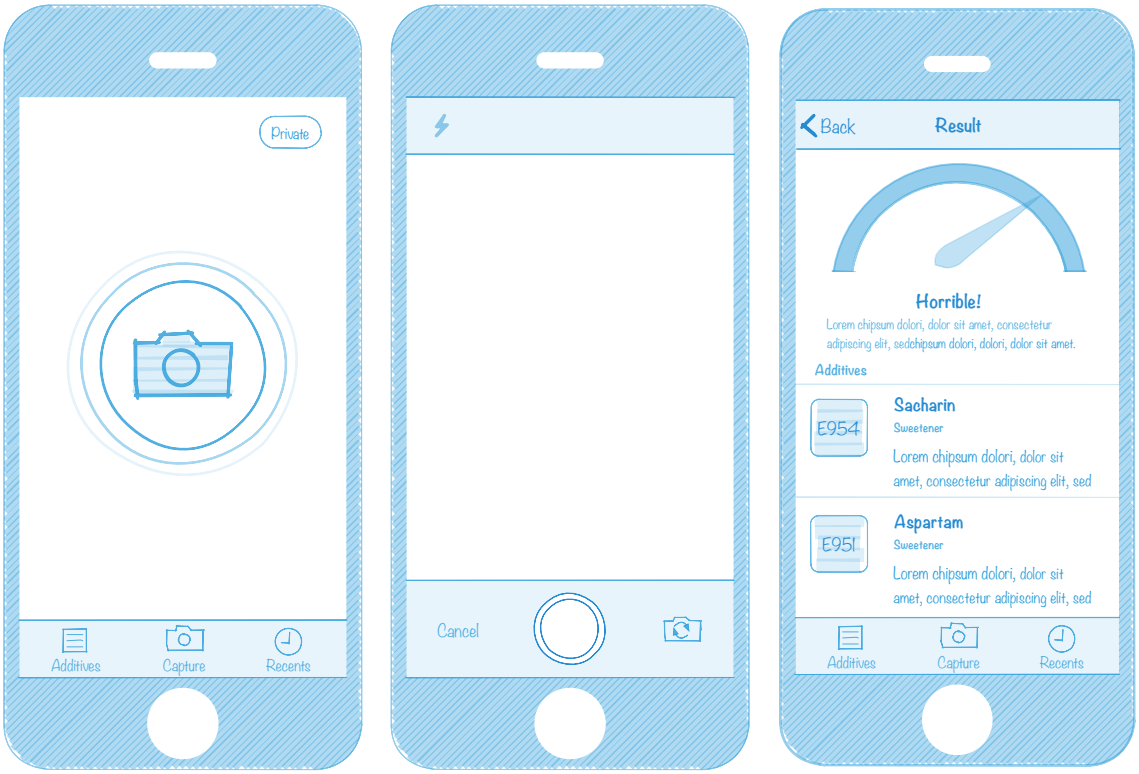
Tabulka A.2: Podrobný popis koncového bodu evoke_token.

/annotations/?method=<*method*>&store=<*bool*>			
POST	Parameters	method (optional)	Specifikuje metodu, která má být použita pro rozpoznání textu ve snímku. Přípustná je jedna z následujících možností: • `auto` – Implicitní hodnota parametru. Zátěž serveru je rozdělena mezi Tesseract a Microsoft. • `tesseract` – Rozpoznání pouze pomocí metody Tesseract. V případě přetížení je požadavek zamítnut. • `microsoft` – Text snímku je rozpoznán pomocí služby Microsoft Computer Vision. • `google` – Text snímku je rozpoznán pomocí služby Google Cloud Vision.
		store (optional)	Hodnota typu `bool`. Implicitní hodnota parametru je `true`. Pokud je specifikována hodnota `false`, není snímek ani rozpoznaný text uložen do interní databáze služby.
	Headers	`Authorization`	Specifikuje typ (`Bearer`) a přístupový token získaný pomocí koncového bodu `token`.
	Body	`image`	Binární reprezentace daného snímku.
	Response	`id`	Unikátní identifikátor anotace. Uvedeno pouze v případě `store=true`.
		`text`	Rozpoznaný text snímku.
		`image`	URL adresa zaslaného snímku. Uvedeno pouze v případě `store=true`.
		`request`	Hodnota typu `Dictionary` obsahující následující klíče: • `date` – Časové razítko získané těsně po rozpoznání textu v příchozím snímku. • `method` – V případě metody `auto` se jedná o skutečně použitou metodu rozpoznání. V ostatních případech, odpovídá hodnotě z požadavku. Uvedeno pouze v případě `store=true`.

Tabulka A.3: Podrobný popis koncového bodu `annotations`.

Příloha B

Návrh uživatelského rozhraní mobilní aplikace



(a) Obrazovka *Capture* (b) Obrazovka *Camera* (c) Obrazovka *Result*

Obrázek B.1: Ilustrace jednotlivých obrazovek uživatelského rozhraní vycházejících z top-level obrazovky *Capture*. Obrazovka (a) poskytuje hlavní tlačítko pro spuštění akvizičního subsystému a vedlejší tlačítko pro aktivaci privátního režimu. Hlavní tlačítko je zvýrazněno pulzující animací. Obrazovka (b) zobrazuje uživatelské rozhraní akvizičního subsystému. Poslední obrazovka (c) poskytuje výsledek rozpoznání etikety v uživatelsky přívětivé formě.

(a) Obrazovka *Additives* (b) Obrazovka *Detail*

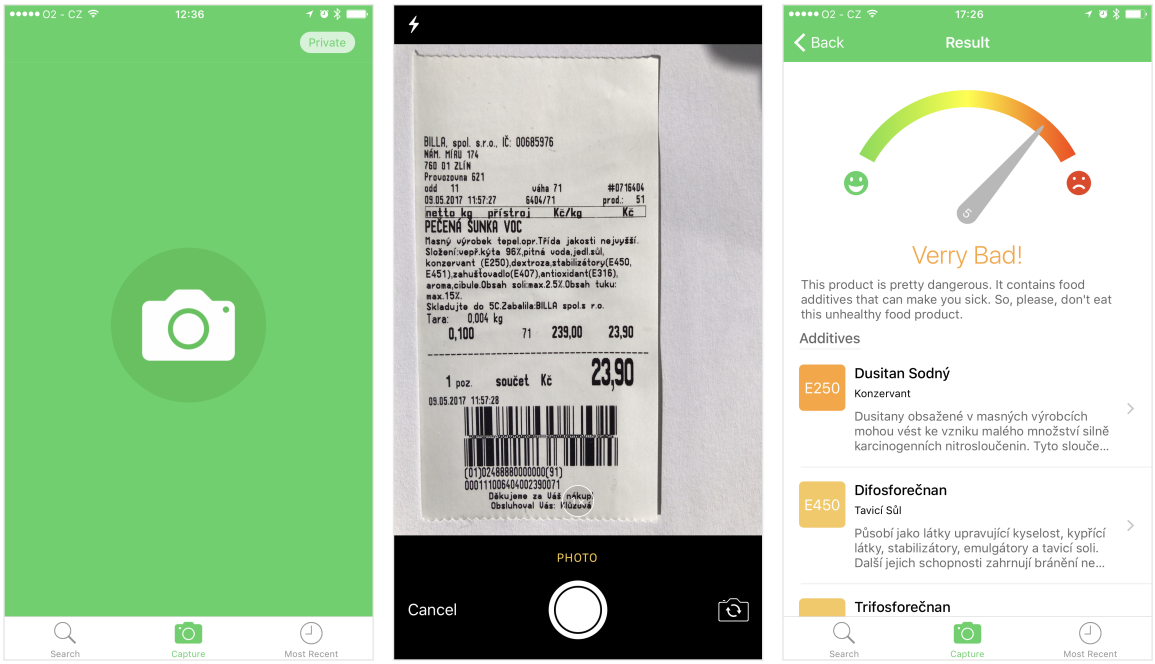
Obrázek B.2: Ilustrace jednotlivých obrazovek uživatelského rozhraní vycházejících z top-level obrazovky *Additives*. Obrazovka (a) poskytuje seznam nebezpečných aditivních látek, které nepodporují zdravý životní styl uživatele. Každá buňka zobrazuje kódové i slovní označení látky a její účel v potravinářském průmyslu (např. barvivo, konzervant). Dotykem buňky se zobrazí obrazovka (b), která zobrazuje bližší informace a nežádoucí účinky dané látky.

(a) Obrazovka *Recents* (b) Obrazovka *Result* (c) Obrazovka *Detail*

Obrázek B.3: Ilustrace jednotlivých obrazovek uživatelského rozhraní vycházejících z top-level obrazovky *Recents*. Obrazovka (a) slouží jako seznam dříve provedených rozpoznání. Buňky jsou tvořeny fotografií pořízenou pro účel rozpoznání, časovým razítkem a hodnocením „zdravosti“ potraviny. Dotykem buňky se zobrazí již popsaná obrazovka (b), která uživateli poskytne bližší informace. V rámci této obrazovky lze zobrazit obrazovku (c), která poskytuje bližší informace a nežádoucí účinky dané rozpoznané látky.

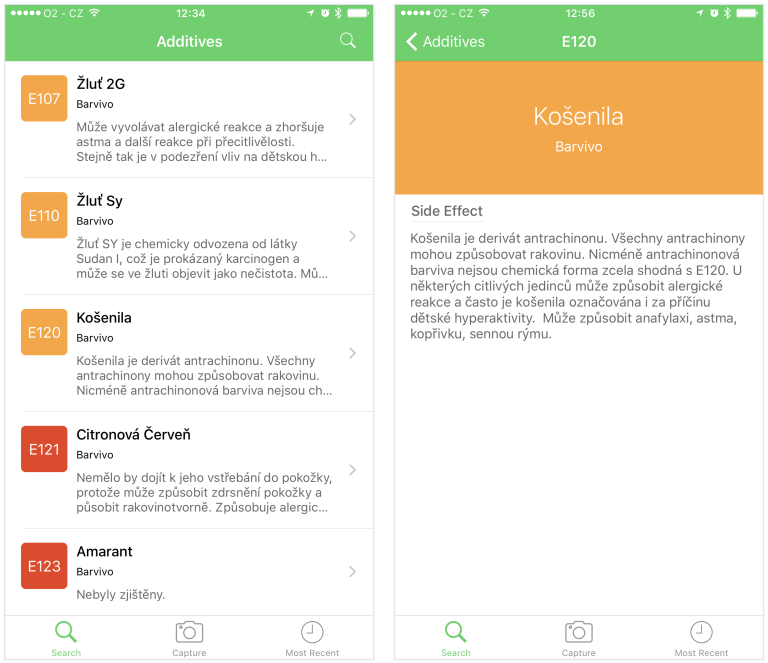
Příloha C

Uživatelské rozhraní dokončené mobilní aplikace



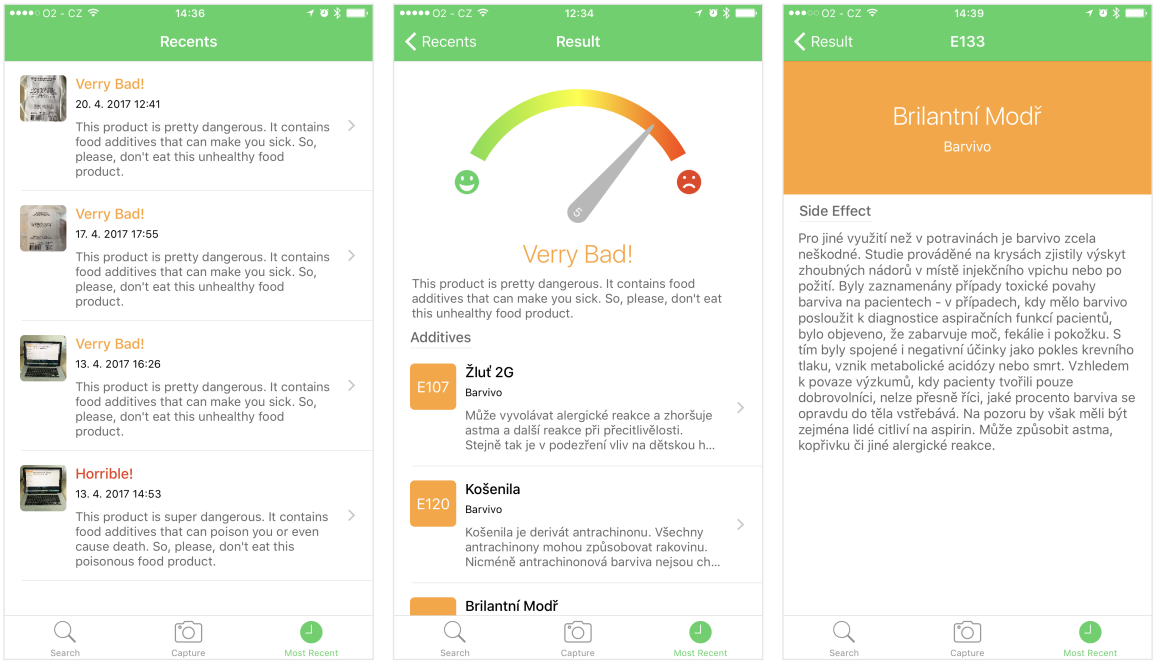
(a) Obrazovka *Capture* (b) Obrazovka *Camera* (c) Obrazovka *Result*

Obrázek C.1: Finální vzhled obrazovek uživatelského rozhraní vycházejících z top-level obrazovky *Capture*. Obrazovka (a) poskytuje hlavní tlačítko pro spuštění akvizičního subsystému a vedlejší tlačítko pro aktivaci privátního režimu. Hlavní tlačítko je zvýrazněno pulzující animací. Obrazovka (b) zobrazuje uživatelské rozhraní akvizičního subsystému. Poslední obrazovka (c) poskytuje výsledek rozpoznání etikety v uživatelsky přívětivé formě.



(a) Obrazovka *Additives* (b) Obrazovka *Detail*

Obrázek C.2: Finální vzhled obrazovek uživatelského rozhraní vycházejících z top-level obrazovky *Additives*. Obrazovka (a) poskytuje seznam nebezpečných aditivních látek, které nepodporují zdravý životní styl uživatele. Každá buňka zobrazuje kódové i slovní označení látky a její účel v potravinářském průmyslu (např. barvivo, konzervant). Dotykem buňky se zobrazí obrazovka (b), která zobrazuje bližší informace a nežádoucí účinky dané látky.



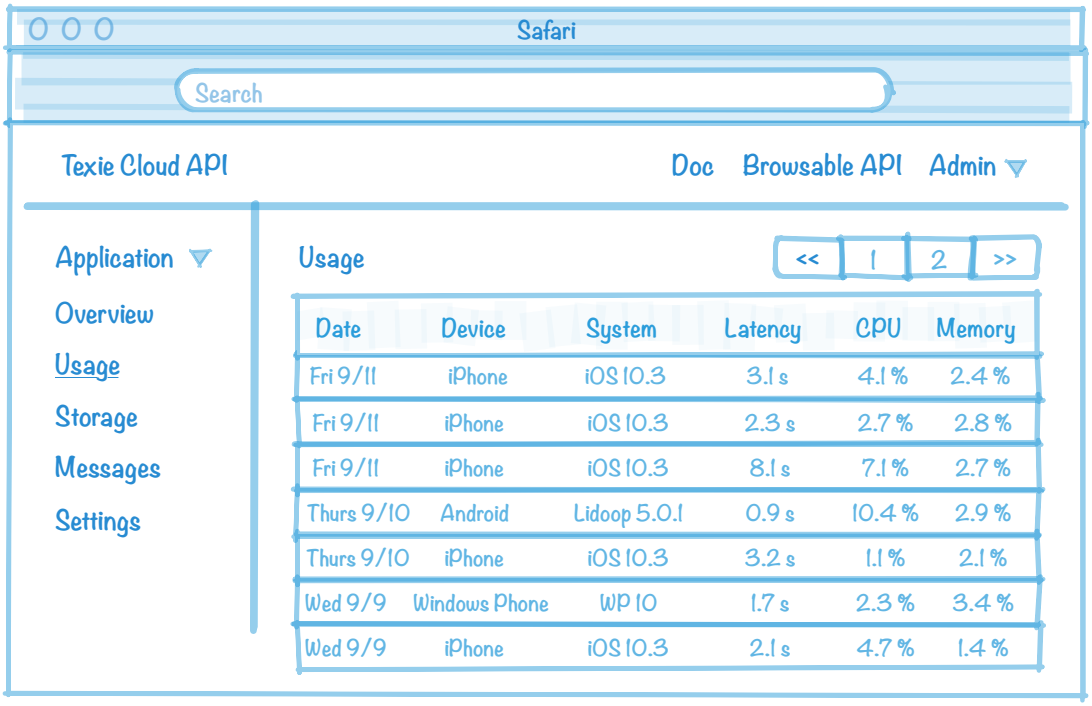
(a) Obrazovka *Recents* (b) Obrazovka *Result* (c) Obrazovka *Detail*

Obrázek C.3: Finální vzhled obrazovek uživatelského rozhraní vycházejících z top-level obrazovky *Recents*. Obrazovka (a) slouží jako seznam dříve provedených rozpoznání. Buňky jsou tvořeny fotografií pořízenou pro účel rozpoznání, časovým razítkem a hodnocením „zdravosti“ potraviny. Dotykem buňky se zobrazí již popsaná obrazovka (b), která uživateli poskytne bližší informace. V rámci této obrazovky lze zobrazit obrazovku (c), která poskytuje bližší informace a nežádoucí účinky dané rozpoznané látky.

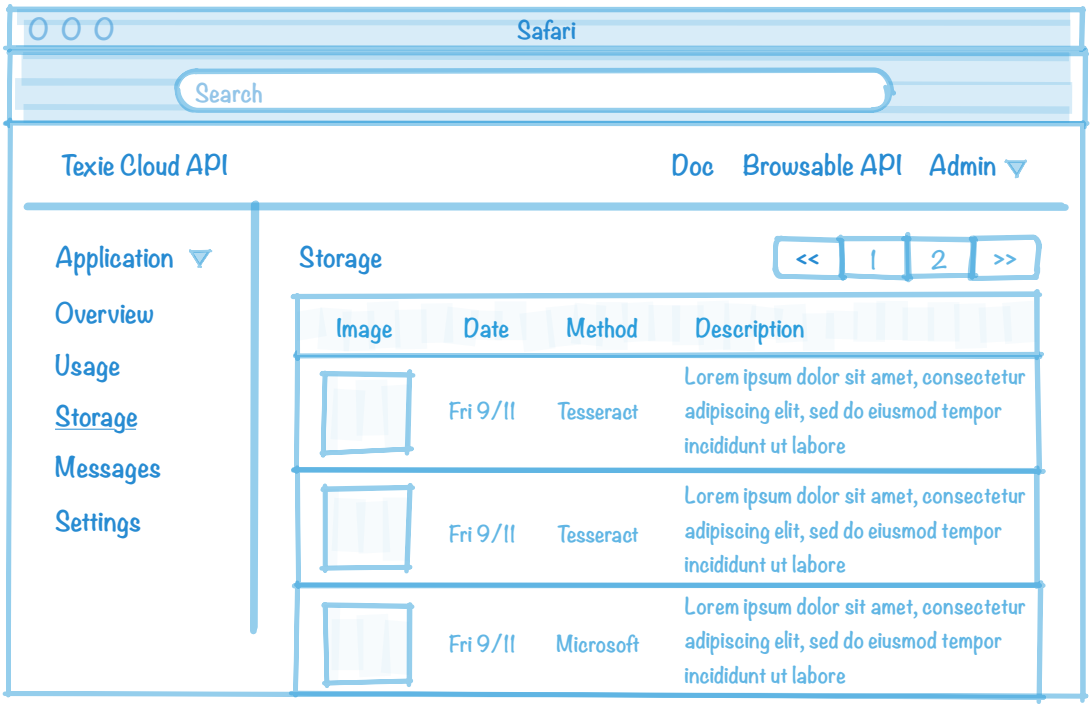
Příloha D

Návrh uživatelského rozhraní webové správy

Obrázek D.1: Drátěný model zobrazující koncepční návrh rozhraní obrazovky *Overview*. Obsahová část zachycuje analytické informace pomocí číselných hodnot a interaktivních grafů.

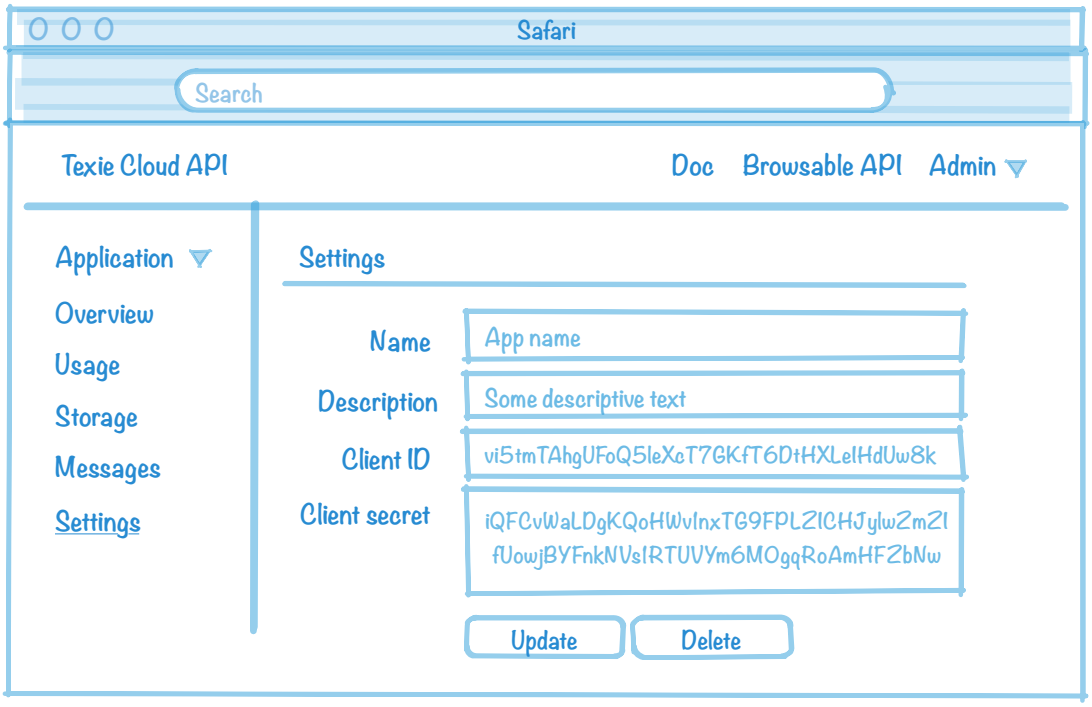


Obrázek D.2: Drátěný model zobrazující koncepční návrh rozhraní obrazovky *Usage*. Obsahová část zachycuje analytické informace pomocí tabulkového zobrazení.



Obrázek D.3: Drátěný model zobrazující koncepční návrh rozhraní obrazovky *Storage*. Obsahová část zachycuje úspěšně rozpoznané snímky včetně získaných a doplňujících informací.

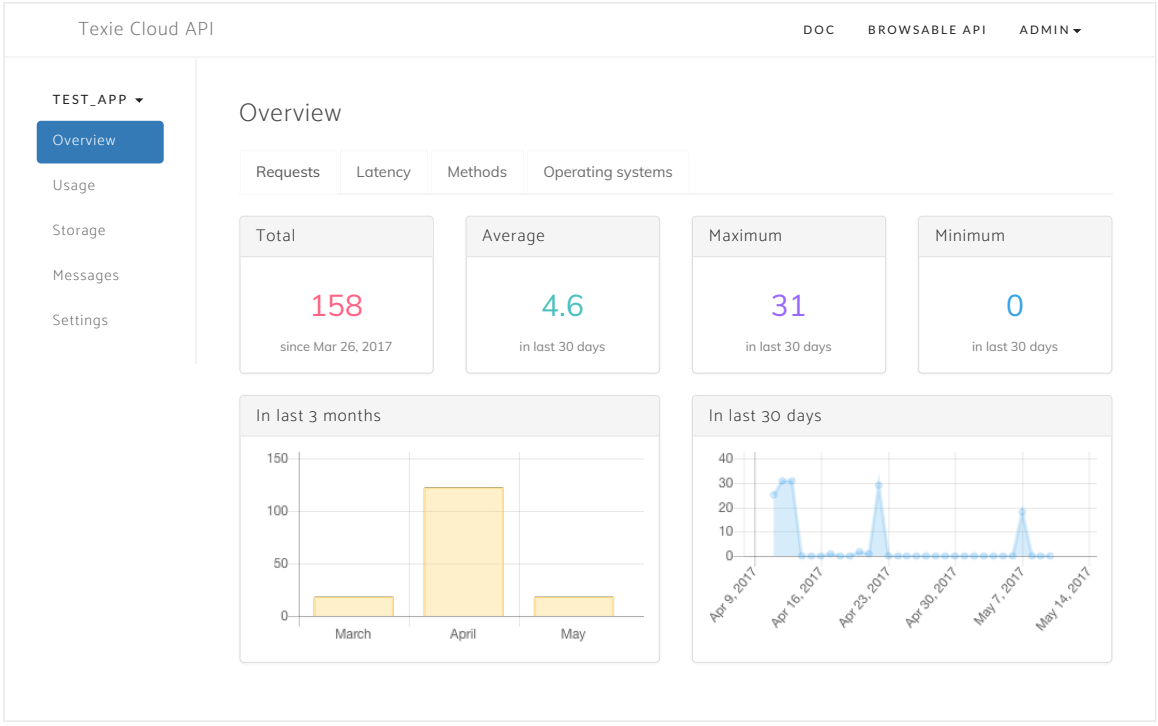
Obrázek D.4: Drátěný model zobrazující koncepční návrh rozhraní obrazovky *Messages*. Obsahová část zachycuje informační, chybové nebo varovné zprávy určené vývojáři dané aplikace.



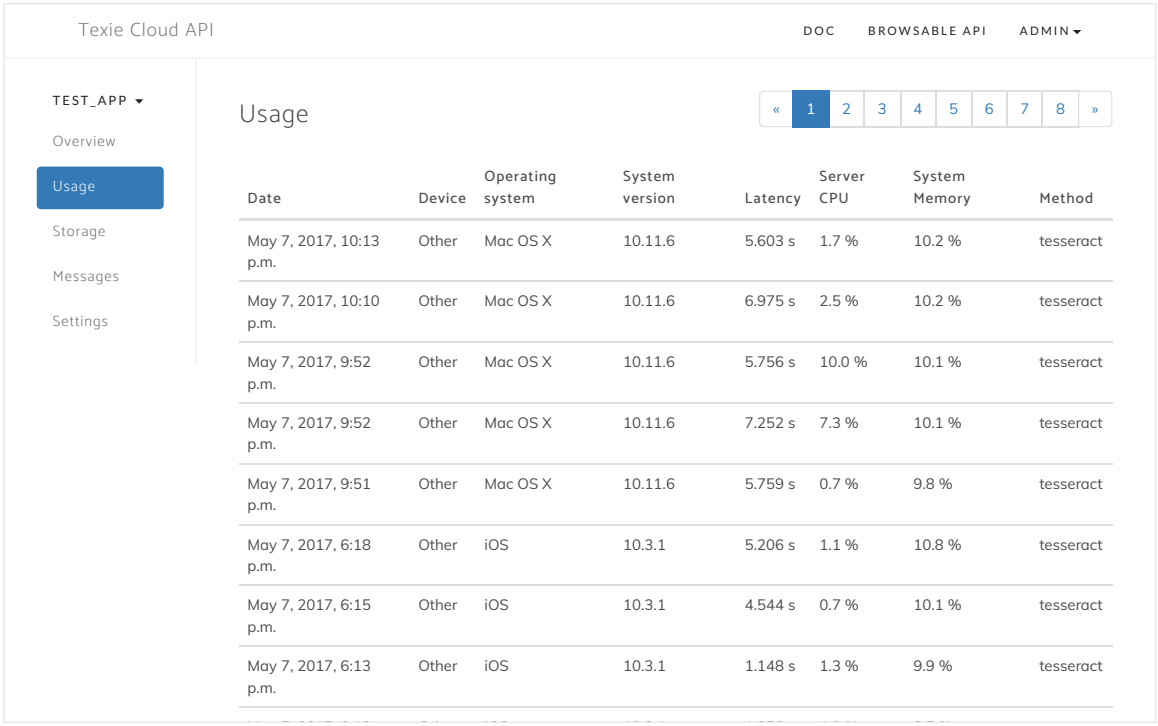
Obrázek D.5: Drátěný model zobrazující koncepční návrh rozhraní obrazovky *Settings*. Obsahová část umožňuje uživateli zobrazit informace o aktuálně aktivní aplikaci, případně ji dovoluje editovat nebo trvale odstranit.

Příloha E

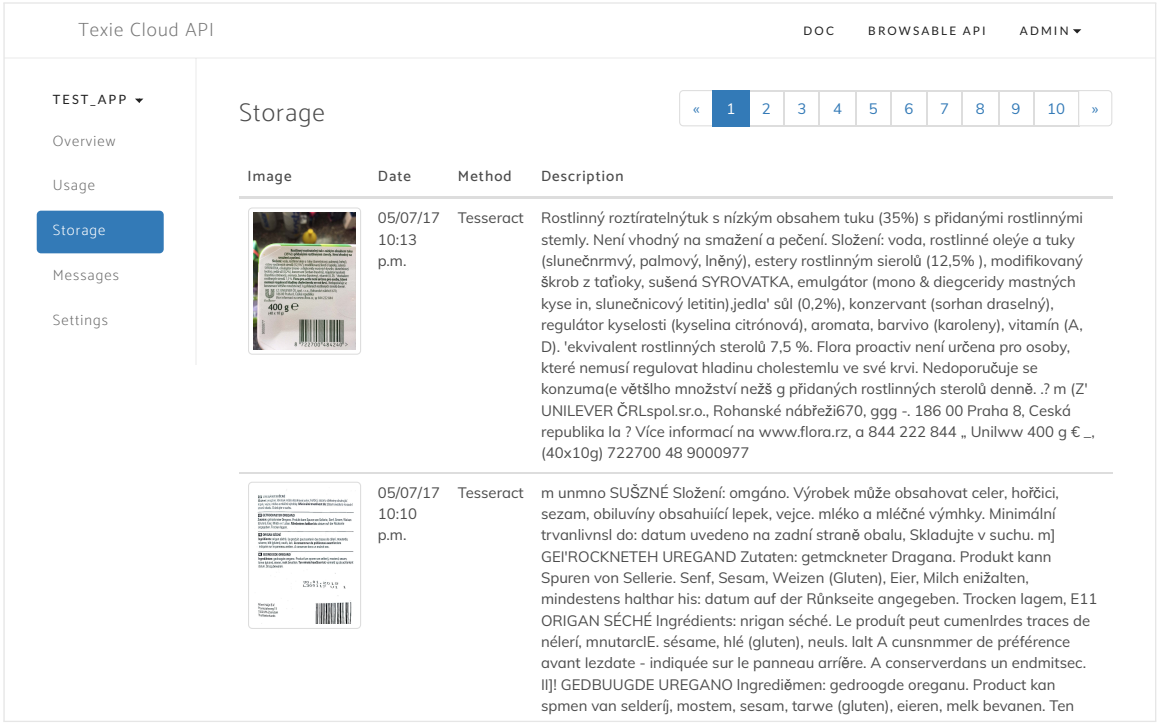
Uživatelské rozhraní dokončené webové správy



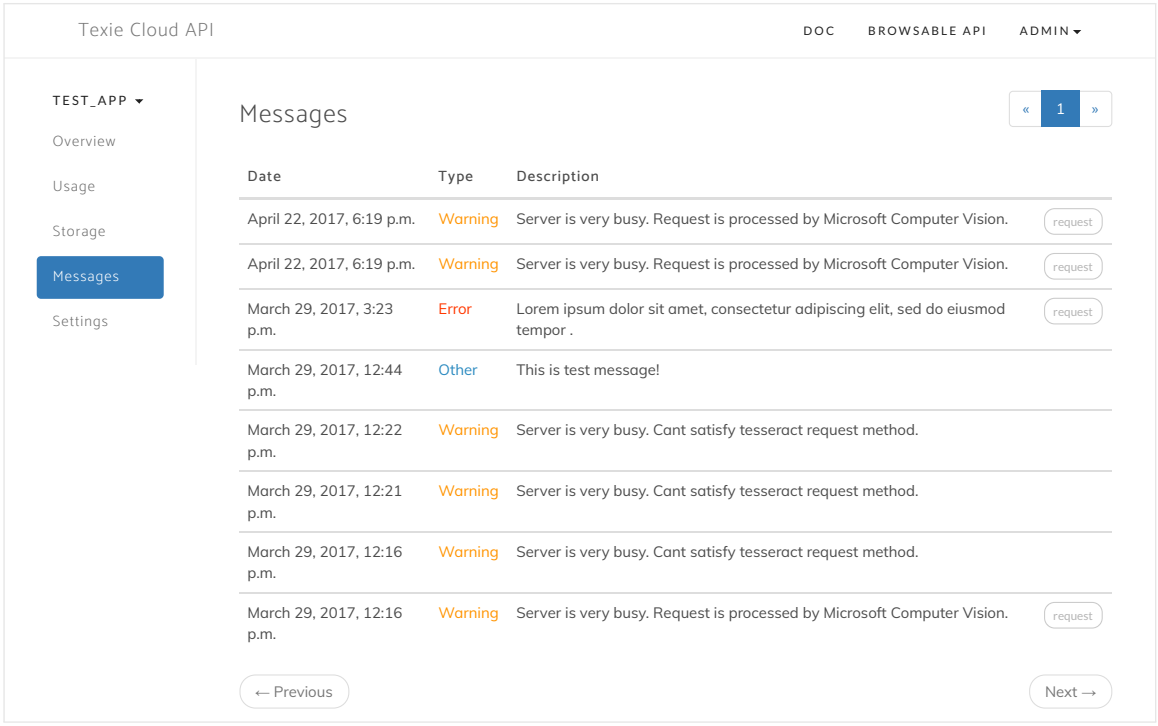
Obrázek E.1: Finální vzhled rozhraní obrazovky *Overview*. Obsahová část zachycuje analytické informace pomocí číselných hodnot a interaktivních grafů.



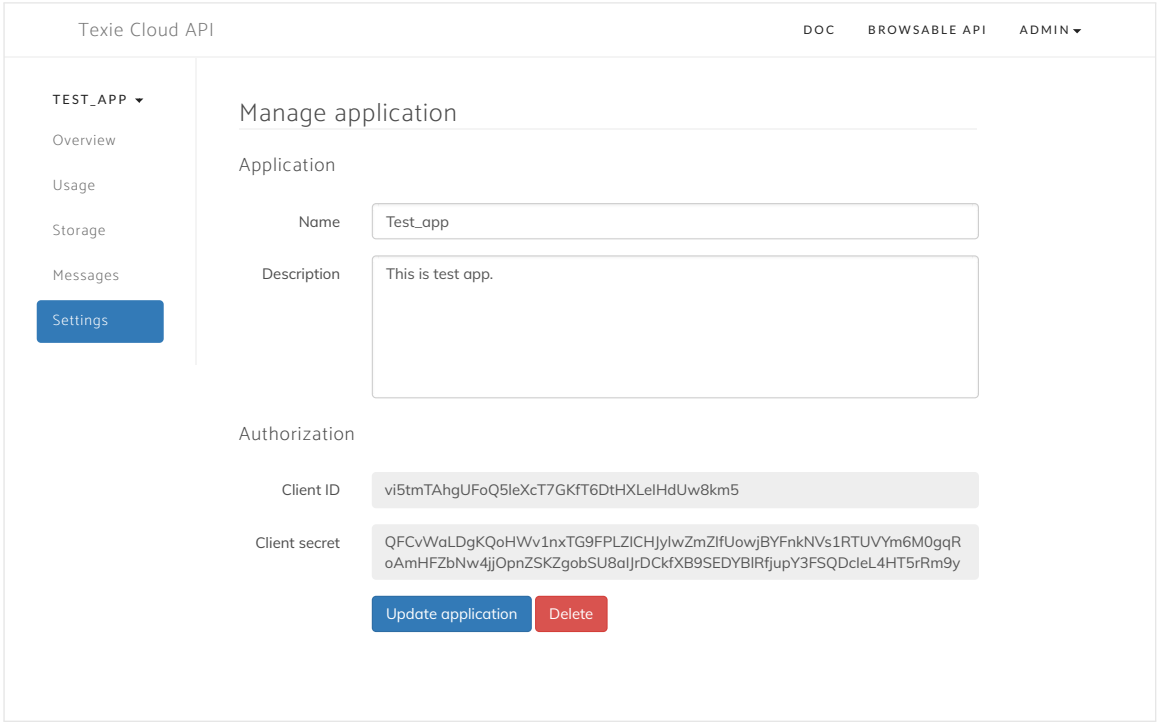
Obrázek E.2: Finální vzhled rozhraní obrazovky *Usage*. Obsahová část zachycuje analytické informace pomocí tabulkového zobrazení.



Obrázek E.3: Finální vzhled rozhraní obrazovky *Storage*. Obsahová část zachycuje úspěšně rozpoznané snímky včetně získaných a doplňujících informací.



Obrázek E.4: Finální vzhled rozhraní obrazovky *Messages*. Obsahová část zachycuje informační, chybové nebo varovné zprávy určené vývojáři dané aplikace.

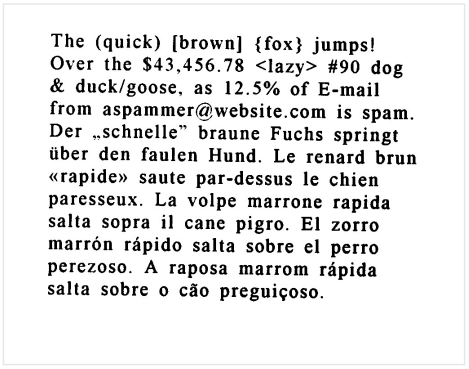


Obrázek E.5: Finální vzhled rozhraní obrazovky *Settings*. Obsahová část umožňuje uživateli zobrazit informace o aktuálně aktivní aplikaci, případně ji dovoluje editovat nebo trvale odstranit.

Příloha F

Snímky použité pro testování

(a) Snímek `eurotext_small.tiff`



(b) Snímek `eurotext.tiff`

(c) Snímek `original.jpg`

(d) Snímek `preprocessed.jpg`

Obrázek F.1: Snímky pro testování uvedené v kapitole 9. Poslední dva snímky jsou poskytnuty J. Tomeškem, který se ve své diplomové práci [52] zabývá předzpracováním textu. Všechny zobrazené snímky je možné nalézt na přiloženém CD.

Příloha G

Plakát