FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

ZADÁNÍ DIPLOMOVÉ PRÁCE

Název:	Systém pro vyhledávání obrázků v rozšířené realitě
Student:	Bc. Petr Chmelař
Vedoucí:	Ing. Jaroslav Kuchař, Ph.D.
Studijní program:	Informatika
Studijní obor:	Webové a softwarové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	Do konce letního semestru 2018/19

Pokyny pro vypracování

Cílem práce je vyvinout systém umožňující rozšířit mobilní aplikace o vyhledávání předem definovaných obrázků v rozšířené realitě. Práce se skládá ze tří komponent - serverové části, knihovny pro iOS a webové aplikace pro správu obrázků.
1. Proveďte rešerši aktuálního stavu zahrnující zhodnocení existujících řešení.
2. Analyzujte a popište dostupné metody pro určení podobnosti obrázků na základě deskriptorů.
3. Definujte požadavky na architekturu serverové části. Hlavní funkcionalitou bude ukládání obrázků do databáze a vyhledání podobného obrázku na základě deskriptorů.
4. Navrhněte knihovnu pro iOS umožňující vyhledávání podobností kontinuálním odesíláním deskriptorů na server.
5. Zpracujte návrh uživatelského rozhraní webové aplikace pro správu obrázků a jím příslušných metadat.
6. Implementujte prototypy jednotlivých částí. Pro samotné vyhledávání podobností je možné využít existující knihovny.
7. Otestujte funkcionalitu systému.

Seznam odborné literatury

Dodá vedoucí práce.

Ing. Michal Valenta, Ph.D.
vedoucí katedry

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
děkan

V Praze dne 31. ledna 2018

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Diplomová práce

Systém pro vyhledávání obrázků v rozšířené realitě

Bc. Petr Chmelař

Katedra softwarového inženýrství
Vedoucí práce: Ing. Jaroslav Kuchař, Ph.D.

6. května 2018

Poděkování

Děkuji své rodině a přátelům za pomoc a podporu po celou dobu studia. Dále děkuji Ing. Jaroslavu Kuchařovi, PhD. za vedení a konzultaci práce.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval(a) samostatně a že jsem uvedl(a) veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 46 odst. 6 tohoto zákona tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou, a veškeré jejich dokumentace (dále souhrnně jen „Dílo"), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla, a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu), licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným způsobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 6. května 2018 .....

Odkaz na tuto práci

Chmelař, Petr. *Systém pro vyhledávání obrázků v rozšířené realitě*. Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2018.

Abstrakt

Práce se věnuje problematice počítačového vidění, konkrétně analýze obrazových dat, vyhledávání obrázků na základě jejich obsahu a možnostmi využití v aplikacích pracujících s konceptem rozšířené reality.

Hlavním cílem je vytvoření prototypu systému, který umožní vývojářům mobilních aplikací definování obrázků prostřednictvím webového rozhraní a jejich následné vyhledávání v prostředí rozšířené reality na platformě iOS. Jednotlivé kapitoly práce se postupně věnují analýze požadavků, návrhům řešení hlavních částí systému, samotné implementaci a následnému testování.

Hotový prototyp systému byl nasazen v testovacím módu a nic nebrání následnému oficiálnímu spuštění.

Klíčová slova Počítačové vidění, OpenCV, Rozšířená realita, iOS, JavaScript

Abstract

The thesis deals with the problematics of computer vision, specifically analysis of image data, content-based image retrieval and possibilities for use in the augmented reality applications.

The main goal of this thesis is to implement a prototype of the system that will enable developers of mobile applications to define images through the web interface and search for them in the augmented reality on iOS platform. Individual chapters of the thesis deal with the analysis of requirements, the design of main parts of the system, the implementation itself and the subsequent testing.

The finished prototype of the system was deployed in the test mode and nothing prevents it's subsequent official release.

Keywords Computer vision, OpenCV, Augmented reality, iOS, JavaScript

Obsah

Seznam příkladů

Seznam tabulek

Úvod

Úvod do problematiky

Aplikace pro mobilní telefony se staly každodenní součástí našich životů. Usnadňují nám vzájemnou komunikaci, organizaci času, orientaci v prostoru a v neposlední řadě i zábavu. Jejich pole působnosti je ale zatím ve většině případů striktně limitované displejem daného zařízení.

Jedním ze současných trendů ve vývoji mobilních aplikací je právě rozšiřování možností aplikace za hranice samotného přístroje – do prostředí virtuální, potažmo rozšířené reality. Jedná se o unikátní propojení reálného světa se světem informačních technologií.

Rozšířenou realitou rozumíme kontinuální zachycování okolního prostředí fotoaparátem daného zařízení, porozumění jeho kontextu a následné zobrazení rozšířené o definovanou přidanou hodnotu. Pro zmíněné porozumění kontextu často dochází k zapojení netriviálních disciplín, jako je počítačové vidění, strojové učení nebo umělá inteligence.

V mé práci se zabývám vývojem prototypu jednotného systému, který dává vývojářům mobilních aplikací možnost obohacení aplikace na základě vyhledávání obrázků v rozšířené realitě. Obrázky určené k vyhledávání je možné definovat a spravovat prostřednictvím webového rozhraní.

Motivace

Hlavní motivací pro mě bylo seznámení se s problematikou počítačového vidění. Konkrétně s vyhledáváním obrázků na základě jejich obsahu a možnostmi využití v prostředí rozšířené reality. Další motivací bylo zdokonalení mých znalostí vývoje komplexních systémů, zahrnující implementaci serverové a webové aplikace v jazyce JavaScript[1].

[1]Programovací jazyk určený primárně pro vývoj dynamických webových stránek.

Kapitola 1

Analýza

1.1 Cíle práce a možnosti realizace

1.1.1 Cíle práce

Cílem práce je vytvořit prototyp systému, který vývojářům mobilních aplikací poskytne jednoduchý způsob rozšíření jejich aplikace o možnost zobrazení definovaného obsahu v prostředí rozšířené reality na základě vyhledání stanoveného obrázku. Mezi typické případy užití se řadí například přehrání filmové upoutávky po rozpoznání filmového plakátu, přehled recenzí k dané knize identifikované jejím přebalem, otevření webové stránky na základě zachycení venkovního billboardu a nespočet dalších akcí vyvolaných v rozšířené realitě vyhledáním předem definovaného obrázku.

Proces vyhledávání obrázků v rozšířené realitě tedy sestává celkem ze tří částí. První častí je nepřetržité zachycování okolního prostředí fotoaparátem mobilního zařízení a extrakce jednotlivých snímků. Dále je nutné porovnat každý extrahovaný snímek oproti předem definované databázi obrázků, které chceme rozpoznávat a vyhodnotit případné shody. Poslední částí je vyvolání příslušné akce v prostředí rozšířené reality. Akce příslušející jednotlivým obrázkům je možné definovat jako metadata.

Pro budoucí bezproblémovou funkčnost systému je důležité nastavit určitá kritéria, která systém musí splňovat. Obrázků určených k rozpoznávání může být vysoký počet a systém by měl poskytovat stejnou úroveň služeb nezávisle na počtu obrázků v databázi. Systém musí být dále také připraven na potřebu časté a okamžité modifikace databáze obrázků jako je například přidání nového obrázku pro vyhledávání.

Vzhledem k zacílení systému na vývojáře mobilních aplikací, kteří jej budou dále využívat pro své aplikace, je vhodné distribuovat výsledný kód pro mobilní zařízení jako knihovnu, která zajistí snadnou a rychlou implementaci do výsledné aplikace.

1.1.2 Možnosti realizace

1.1.2.1 Zdroj dat v mobilní aplikaci

Nejjednodušším způsobem realizace požadované funkcionality je přibalení obrázků, které mají být vyhledávány, přímo do mobilní aplikace. Samotné porovnávání obrázků a vyhodnocení shod potom probíhá pouze v mobilním zařízení bez nutnosti jakékoli vnější komunikace.

Výhodou řešení je jednoduchost implementace a funkčnost i v offline režimu. Nevýhodou je špatná škálovatelnost, kdy při velkém počtu obrázků dochází k významnému zvýšení velikosti aplikace. Dalším problémem je nutnost distribuce nové verze aplikace s každou modifikací databáze obrázků.

1.1.2.2 Server jako zdroj dat

Pro zajištění nezávislosti mobilní aplikace na modifikaci databáze obrázků je nutné hlavní zdroj dat umístit mimo samotnou aplikaci. Standardním řešením problému je implementace serveru, se kterým následně aplikace komunikuje prostřednictvím aplikačního rozhraní přes mobilní nebo Wi-Fi[2] připojení.

Databáze obrázků je při prvním spuštění aplikace stažena do mobilní aplikace ze serveru a následně synchronizována při splnění stanovených podmínek. Například v předem definovaných intervalech vždy, když je mobilní zařízení připojeno k Wi-Fi síti nebo po manuální interakci v aplikaci – stisknutím tlačítka pro obnovu dat.

Samotné vyhledávání obrázků probíhá pouze na mobilním zařízení, server zde slouží pouze jako zdroj dat. Výhodou je snadná možnost modifikace databáze obrázků a po prvotní synchronizaci i funkčnost v offline režimu. Nevýhodou je opět vzrůstající velikost aplikace a uživatelská nepřívětivost, kdy při velkém počtu obrázků bude prvotní stahování časově náročné.

1.1.2.3 Kontinuální komunikace se serverem

Implementací kontinuální komunikace se serverem lze dosáhnout odstranění závislosti mezi počtem obrázků a velikostí aplikace. Mobilní zařízení automaticky a pravidelně odesílá zachycené snímky okolního prostředí na server, kde dochází k porovnání vůči databázi obrázků a následnému informování zařízení o výsledcích.

Databáze obrázků je uložena pouze na serveru, výhodou je tedy nenáročnost na velikost aplikace a snadná modifikace databáze. Nevýhodou je složitější implementace a nefunkčnost v offline režimu, kterou ale lze částečně řešit ukládáním výsledků určitého počtu předchozích hledání v aplikaci.

[2]Standard popisující bezdrátovou komunikaci v počítačových sítích.

1.1.3 Výběr řešení

S ohledem na cíle práce byla vyřazena možnost implementace systému pomocí zdroje dat v mobilní aplikaci, která nerespektuje požadavek na potřebu časté modifikace databáze obrázků. V následném rozhodování byla zvolena implementace kontinuální komunikace se serverem, která zaručuje stejnou úroveň služeb nezávisle na velikosti databáze obrázků a zároveň nesnižuje uživatelskou přívětivost aplikace prvotním stahováním velkého objemu dat.

Cílem práce je dále distribuce knihovny pro vývojáře mobilních aplikací. V současné době mobilním zařízením prakticky dominují platformy iOS a Android. [1] Obě mají vlastní řešení pro přístup k rozšířené realitě. V případě iOS se jedná o ARKit, který je dostupný pro telefony iPhone 6s a novější v systému verze 11.0 a vyšší. Řešením pro platformu Android je ARCore a v době implementace systému byl podporován pouze malou skupinou nejvýkonnějších telefonů. Pro možnost otestování prototypu systému na co největším množství zařízení byla zvolena implementace knihovny pro platformu iOS. Po úspěšném otestování a odladění systému bude následovat implementace pro platformu Android.

Pro účely snadné modifikace databáze obrázků bylo rozhodnuto o vývoji jednoduchého webového rozhraní, které bude primárně sloužit k přidávání nových obrázků a editaci metadat u obrázků stávajících. Webové rozhraní bude dále vstupním bodem celého systému, kde si vývojáři mobilních aplikací mohou vytvořit vlastní účet a stáhnout knihovny pro implementaci nabízené funkcionality do svých aplikací.

Celkový prototyp systému budou tedy tvořit tři hlavní části. Serverová aplikace pro uložení databáze obrázků a vyhodnocení shod na základě přijatých snímků. Knihovna pro mobilní platformu iOS s hlavní funkcionalitou kontinuálního odesílání snímků okolního prostředí na server. A dále webová aplikace pro správu databáze obrázků a jim příslušných metadat.

1.2 Rešerše existujících řešení

Existuje velké množství řešení, která nabízejí vyhledávání obrázků v rozšířené realitě. Jejich zaměření a kvalita zpracování se ovšem liší. Rešerše se věnuje pouze řešením, které při vyhledávání shod kontinuálně komunikují se serverem a poskytují možnost jednoduché správy databáze obrázků prostřednictvím webového rozhraní.

Konkrétně se jedná o dvě služby reprezentující danou skupinu – Wikitude a Vuforia. Obě jsou komerčním produktem a pro plnohodnotné využití je nutné zakoupit licenci. Aktuálně neexistuje žádné bezplatné řešení poskytující plnohodnotný systém, který by vyhledávání shod realizoval kontinuální komunikací mobilní aplikace se serverem.

1.2.1 Wikitude

Wikitude je jedním z nejvýznamnějších hráčů na poli poskytování služeb pro VR/AR aplikace[3] se širokým polem působnosti. Většinu služeb je možné vyzkoušet vytvořením bezplatného vývojářského účtu. Použitá knihovna následně umisťuje do aplikace logo Wikitude.

Pro vyhledávání obrázků v rozšířené realitě je k dispozici služba Wikitude Cloud Recognition. Služba obsahuje základní webové rozhraní pro správu databáze obrázků, které jsou následně vyhledávané prostřednictvím mobilního zařízení kontinuálním odesíláním snímků okolního prostředí.

Jednoduchá implementace do mobilních aplikací je zajištěna distribucí knihovny pro platformy iOS a Android. Knihovna umožňuje vyhledávání obrázků v rozšířené realitě nativně nebo prostřednictvím proprietárního renderovacího enginu v jazyce JavaScript. Nativní přístup k rozšířené realitě aktuálně podporuje pouze metodu OpenGL[4], chybí zde podpora moderních řešení ARKit/ARCore.

Využití proprietárního enginu má výhodu snadné znovupoužitelnosti. Pro více mobilních platforem není nutné implementovat stejnou funkcionalitu vícekrát. Nicméně engine je v mobilním zařízení spouštěn jako WebView[5] a nedosahuje tedy výkonu nativního přístupu.

U komerční verze je nastaven limit databáze na 50 000 obrázků a limit pro vyhledávání 1 000 000 nalezených shod měsíčně. Knihovna neukládá předchozí nalezené shody, proto se do limitu započítává i nalezení opakované shody.

1.2.2 Vuforia

Vuforia je poskytovatel komplexních služeb pro VR/AR aplikace. Všechny služby je možné vyzkoušet v omezeném módu prostřednictvím bezplatného plánu, kdy dochází k umisťování loga Vuforia do aplikace.

Služba Vuforia Cloud Recognition umožňuje vyhledávání obrázků v rozšířené realitě na bázi kontinuální komunikace s mobilním zařízením a správu databáze obrázků ve webovém rozhraní.

Knihovna pro mobilní aplikace poskytuje nativní přístup k rozšířené realitě prostřednictvím OpenGL a pro platformu iOS existuje open-source rozšíření podporující ARKit. Nevýhodou je nedostatečná dokumentace poskytnuté knihovny a neexistující dokumentace zmíněného rozšíření.

Komerční verze služby má nastaven limit databáze na 100 000 obrázků a limit pro vyhledávání je 10 000 nalezených shod měsíčně s možností připlacení za každou další shodu nad limit. Knihovna ukládá předchozí nalezené shody do mezipaměti, opakované nalezení shody se tedy do limitu nezapočítává.

[3] Aplikace pracující s konceptem virtuální nebo rozšířené reality.
[4] Open Graphics Library – multiplatformní rozhraní pro tvorbu grafických aplikací.
[5] Způsob zobrazení webové stránky v mobilní aplikaci.

1.3 Analýza požadavků

Na počátku vývoje je nutné analyzovat jednotlivé požadavky, které musí výsledný systém splňovat. Požadavky musí být jednoznačné a nemohou vzájemně kolidovat. Obvykle se rozdělují na funkční a nefunkční požadavky.

Funkční požadavky definují chování systému. Obvykle se jedná o akce, které je nutné vykonat, aby byla naplněna požadovaná funkcionalita. Nefunkční požadavky potom stanovují dodatečné výkonnostní vlastnosti jednotlivých funkcí systému. Funkčním požadavkem může být například popis způsobu přenášení dat mezi mobilní aplikací a serverem, nefunkčním požadavkem je potom popis zabezpečení daného přenosu.

1.3.1 Definice pojmů

- **API – Application Programming Interface**

  Aplikační rozhraní sloužící k propojení aplikací. Jde o sbírku procedur, funkcí, tříd či protokolů dané aplikace nebo knihovny, které může programátor využívat.

- **REST – Representational State Transfer**

  Architektura rozhraní sestávající z koordinované sady architektonických omezení aplikovaných na komponenty, konektory a datové prvky v rámci distribuovaného hypermediálního systému. Nejčastěji se využívá pro komunikaci mezi aplikacemi pomocí HTTP[6] volání. [2]

1.3.2 Požadavky na serverou aplikaci

Serverová aplikace je hlavní komponentou celého systému. Slouží především jako úložiště databáze obrázků a k vyhodnocení shod na základě snímků přijatých z mobilní aplikace. Dále pak poskytuje API pro komunikaci s webovou aplikací a s knihovnou pro mobilní zařízení.

1.3.2.1 Funkční požadavky

- Uchování databáze obrázků
  - Serverová aplikace zahrnuje databázový server, který slouží jako úložiště pro jednotlivé databáze obrázků.
  - Do každé databáze je možné přidat nové obrázky nebo modifikovat, případně smazat obrázky stávající.
  - Ke každému obrázku mohou dále být přiřazena metadata pro definování příslušné akce.

[6] HyperText Transfer Protocol - protokol pro komunikaci v počítačové síti.

- Vyhodnocení shod
  - Server akceptuje validní snímky odeslané z mobilní aplikace.
  - Pro každý akceptovaný snímek provádí porovnání s danou databází obrázků a o výsledku informuje mobilní aplikaci.
- Aplikační rozhraní
  - Na serveru je spuštěno REST API pro komunikaci s ostatními prvky systému.
  - Server odpovídá na korektní HTTP požadavky typu GET, POST, PUT a DELETE.

1.3.2.2 Nefunkční požadavky

- Zabezpečení serverové aplikace
  - Ke každé databázi obrázků má přístup pouze oprávněný uživatel systému na základě předchozí autorizace.
  - Veškerá komunikace se serverem prostřednictvím REST API probíhá pouze na zabezpečeném protokolu HTTPS[7].
- Formát přenášených dat
  - Data, která server odešle nebo přijme prostřednictvím REST API jsou ve formátu JSON[8].

1.3.3 Požadavky na knihovnu pro platformu iOS

Knihovna pro mobilní platformu iOS kontinuálně zachycuje okolní prostředí. Získané snímky odesílá na API serveru k porovnání a o výsledcích informuje mobilní aplikaci.

1.3.3.1 Funkční požadavky

- Kontinuální odesílání snímků
  - Opakovaně dochází k zachycení snímku okolního prostředí a jeho odeslání na server k vyhodnocení shod.
- Vyhodnocení výsledků
  - V případě kladné odpovědi serveru knihovna pouze informuje mobilní aplikaci o nalezené shodě a případných metadatech.
  - Spuštění příslušné akce je již záležitostí samotné aplikace.

[7] HyperText Transfer Protocol Secure - zabezpečená verze protokolu HTTP.
[8] JavaScript Object Notation – způsob zápisu dat.

1.3.3.2 Nefunkční požadavky

- Přístup k rozšířené realitě
  - Kontinuální zachycování snímků a následné vyhodnocení výsledků v prostředí rozšířené reality je implementované nativním ARKitem.
- Způsob kontinuální komunikace
  - Zachycené snímky jsou odesílány prostřednictvím REST API.
- Komprese přenášených dat
  - Snímky odesílané na server jsou neprve převedeny do vhodného formátu pro úsporu objemu přenášených dat.
  - Před samotným odesláním je celý požadavek ještě dodatečně zkomprimován do formátu gzip.

1.3.4 Požadavky na webovou aplikaci

Webová aplikace je vstupním bodem systému. Poskytuje možnost vytvoření uživatelského účtu a následně dává k dispozici jednoduché rozhraní pro správu databáze obrázků. O každé změně informuje server prostřednictvím API.

1.3.4.1 Funkční požadavky

- Uživatelská relace
  - Návštěvník webové aplikace se může stát uživatelem systému registrováním uživatelského účtu.
  - Uživatel systému se přihlašuje uživatelským jménem a heslem.
- Správa databáze obrázků
  - Přihlášený uživatel může vytvářet vlastní databáze obrázků a provádět jejich modifikaci.
  - Pro editaci metadat je k dispozici jednoduchý editor.
- Dokumentace systému
  - Webová aplikace poskytuje veřejnou stránku s dokumentací všech funkcionalit systému.
  - Dokumentace dále obsahuje odkazy ke stažení knihovny pro platformu iOS a stručný postup implementace do mobilní aplikace.

1.3.4.2 Nefunkční požadavky

- Způsob komunikace se serverem
  - Registrace nebo přihlášení uživatele a veškeré změny databáze obrázků jsou odesílány na server prostřednictvím REST API.
- Zachování uživatelské relace
  - Aktivní uživatelská relace je ukládána do webového prohlížeče pomocí HTTP Local storage[9].

1.4 Srovnání algoritmů pro analýzu obrázků

Pro vyhodnocení shody dvojice obrázků je nejprve nutné provést jejich analýzu. Jedná se o komplexní problematiku, kterou lze řešit různými přístupy. Při volbě vhodné metody je nutné brát v potaz především požadavky na konfidenci a rychlost vyhodnocování shod.

1.4.1 Metody pro analýzu obrázků

1.4.1.1 Porovnání histogramů

Jedna z nejjednodušších a nejrychlejších metod. Hlavní myšlenka je stará již několik desetiletí a vychází z faktu, že například fotografie lesa bude mít velký podíl zelené barvy, kdežto fotografie pláže velký podíl žluté barvy. Porovnáním dvou obrázků lesa tedy dostaneme míru podobnosti mezi jejich histogramy, která bude vyšší než při porovnání obrázku lesa s obrázkem pláže. Hlavní výhoda metody – její jednoduchost, je zároveň její největší nevýhodou. Metoda nerozumí kontextu obrázků a například při porovnání obrázku banánu a obrázku pláže bude udávat relativně velkou míru podobnosti, protože oba obrázky obsahují vysoký podíl žluté barvy.

1.4.1.2 Vyhledávání vzorů

Vyhledávání vzorů je technika sloužící pro nalezení oblastí v obrázku, které se shodují, nebo jsou podobné tzv. vzorovému obrázku. Nejčastěji se využívá pro určení, zda daný obrázek uvnitř sebe obsahuje vzorový obrázek. Porovnání je realizováno postupným posouváním vzorového obrázku po druhém obrázku. Posun probíhá vždy o jeden pixel ve směru zleva doprava a shora dolu. Pro každou pozici je vypočítána míra podobnosti určující o jak kvalitní shodu se jedná. Pozice s nejvyšší hodnotou míry podobnosti je potom kandidátem na shodu. Nevýhodou metody je velmi nízká odolnost vůči změně světelných podmínek, měřítka nebo rotaci. [3]

[9] Metoda pro uložení dat do webového prohlížeče.

1.4.1.3 Extrahování lokálních vlastností

Nejvíce komplexní, ale zároveň také nejvíce efektivní metoda pro analýzu obrázků. Na obrázku je identifikován určitý počet malých oblastí, které splňují definovanou charakteristiku. Ta se liší dle použitého algoritmu, ale zpravidla se jedná o hrany, rohy, bloby nebo vysoce kontrastní místa. Podstatné je, že algoritmus u takto identifikovaných vlastností garantuje vysokou míru odolnosti vůči změně úhlu pohledu, měřítka, rotaci a dalším modifikacím. Následným porovnáním dvou setů lokálních vlastností lze určit, zda oba obrázky popisují stejnou scénu nebo objekt. [4]

Existuje velké množství algoritmů pro extrakci lokálních vlastností. Chování každého z nich je dále možné ovlivnit vstupními parametry. Nesprávné použití algoritmu může negativně ovlivnit celkový výkon systému. V závislosti na potřebách aplikace nebo systému je tedy nutné zvolit a správně nastavit vhodný algoritmus.

1.4.2 Algoritmy pro extrakci lokálních vlastností

1.4.2.1 Definice pojmů

- **Lokální vlastnost (Image feature)**

  Malá oblast obrázku, která je něčím unikátní. Zpravidla je složena z klíčového bodu a deskriptoru.

- **Klíčový bod (Feature keypoint)**

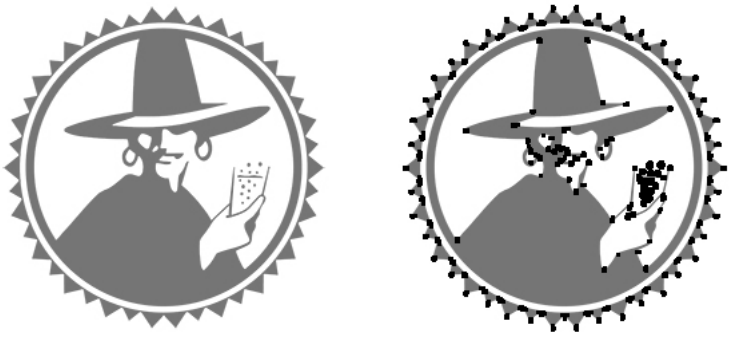
  Obvykle obsahuje informace o 2D pozici vlastnosti a případné doplňující informace jako je velikost nebo orientace.

- **Deskriptor (Feature descriptor)**

  Deskriptor obsahuje vizuální popis lokální vlastnosti. Porovnáním deskriptorů dvou obrázků lze zjistit jejich vzájemnou shodu.

1.4.2.2 Harris Corner Detector

Za jeden z prvních algoritmů extrahující lokální vlastnosti obrázků je možné považovat Harris Corner Detector. Navrhli jej v roce 1988 Chris Harris a Mike Stephens jako součást systému, který analyzoval snímky z pohybující se kamery. Zaměřili se na sledování rohů a hran, které jsou charakteristické velkými změnami intenzity ve všech směrech. Matematickým vyjádřením dosáhli rychlého a spolehlivého algoritmu použitelného pro analýzu obrázků. [5]



Příklad 1.1: Detekce rohů algoritmem Harris Corner Detector

1.4.2.3 SIFT – Scale-Invariant Feature Transform

Detekce rohů je účinným způsobem analýzy obrázků a pro určité případy využití je plně dostačující. Nicméně při zvětšení měřítka dochází ke značnému zkreslení – z rohů se stávají hrany a naopak při zmenšení měřítka. To je problém zejména při potřebě analyzovat a porovnat obrázky různých měřítek. S řešením přišel v roce 2004 David G. Lowe návrhem algoritmu SIFT, který je odolný vůči změně měřítka.

Algoritmus je založen na principu transformace obrázku do měřítkově nezávislých souřadnic. Následně extrahuje jednotlivé klíčové body a pro každý spočítá příslušný deskriptor. Průběh algoritmu se skládá ze čtyř hlavních kroků. Nejprve je zkonstruována měřítkově nezávislá reprezentace obrázku a pomocí aproximace Laplacianu rozdílem Gaussových funkcí jsou nalezeny lokální extrémy, které reprezentují potenciální klíčové body. Druhým krokem je zpřesnění lokalizace klíčových bodů na základě Taylorova rozvoje. Body které nesplní určité podmínky, jsou vyloučeny a zůstavají pouze silné klíčové body. Třetím krokem je určení orientace klíčových bodů pro zajištění invariance vůči rotaci. V poslední fázi algoritmu je pak pro každý klíčový bod vypočítán příslušný deskriptor – vektor o 128 binech popisující vizuální okolí bodu. [6, 7]

1.4.2.4 SURF – Speeded Up Robust Features

SIFT je robustním algoritmem a porovnáním získaných deskriptorů je možné identifikovat shodné obrázky s vysokou mírou konfidence. Nicméně právě jeho komplexní přístup k určení klíčových bodů je náročný na výpočetní výkon. Zejména u větších obrázků může docházet k prodlevě, která již není akceptovatelná u systémů citlivých na rychlost porovnání obrázků.

V roce 2006 Herbert Bay, Tinne Tuytelaars a Luc Van Gool popsali algoritmus SURF. Jeho princip volně vychází z algoritmu SIFT, ale v kritických místech je implementován odlišně s důrazem na rychlost. Měřítkově nezávislá reprezentace obrázku je generována pomocí determinantu Hessovy matice a produkuje integrální obrázky. Laplacian zde není aproximován rozdílem Gaussových funkcí, ale využitím obdélníkových funkcí. Následně je možné využít výhod integrálních obrázků, počítat konvoluci paralelně pro různá měřítka a tím dosáhnout značného zrychlení. Odlišný je i výpočet deskriptoru, který produkuje vektor poloviční délky (64 binů) a tím zajišťuje další zvýšení výkonu při následném porovnávání deskriptorů. [6, 8]

1.4.2.5 ORB – Oriented FAST and Rotated BRIEF

I značné zrychlení, které přináší algoritmus SURF, nemusí být dostatečné pro rychlé porovnání obrázku oproti velké databázi nebo pro analýzu obrázků v zařízeních s nízkým výpočetním výkonem. Problém se snaží řešit algoritmus ORB, který vznikl vylepšením a spojením algoritmů FAST (Features from Accelerated Segment Test) a BRIEF (Binary Robust Independent Elementary Features).

Nejprve je použit FAST pro nalezení klíčových bodů. Principem je identifikace rohů podobně jako v případě Harris Corner Detector. Vyřazením nekvalitních kandidátů vysokorychlostním testem je dosaženo výrazného zrychlení. Algoritmem BRIEF jsou následně vypočteny deskriptory, reprezentované jako binární řetězec. Defaultní délka binárního řetězce je 256 bitů, tedy 32 bajtů. Pro srovnání, algoritmus SIFT reprezentuje deskriptor vektorem o 128 binech. Jedná se o čísla s plovoucí desetinnou čárkou, takže jeden deskriptor potřebuje 512 bajtů. V případě algoritmu SURF obsahuje vektor 64 binů, tedy 256 bajtů. Jedná se o výrazné snížení paměťové složitosti, které následně vede k rychlejšímu porovnávání deskriptorů. Samotné deskriptory ale nejsou tak robustní, dochází tedy k určité ztrátě invariance vůči změnám. [9]

Příklad 1.2: Klíčové body detekované algoritmem ORB

1.4.2.6 AKAZE – Accelerated-KAZE

Všechny doposud představené algoritmy operují v lineárním (Gaussově) prostoru. Algoritmus KAZE pracuje v nelineárním prostoru. Prostřednictvím nelineární difuze dokáže detekovat klíčové body a následně vypočítat příslušné deskriptory. Původní algoritmus je výpočetně přibližně stejně náročný jako SIFT, nicméně existuje vylepšená varianta Accelerated-KAZE. Využívá matematického postupu FED – Fast Explicit Diffusion, který dramaticky zrychluje proces výpočtu v nelineárním prostoru. Dle autorů je algoritmus několikanásobně rychlejší oproti algoritmům SIFT a SURF při zachování vysoké míry robustnosti deskriptorů. [10, 11]

1.4.3 Výběr algoritmu

Jednou z potřeb systému je rychlá analýza přijatého snímku a porovnání se všemi ostatními obrázky v databázi. Je důležité brát v potaz také potřebu co nejpřesnějšího vyhledání shod. Z toho důvodu byl vyřazen algoritmus Harris Corner Detector, který je sice rychlý, ale zároveň náchylný na změnu měřítka obrázku. Algoritmy SIFT a SURF poskytují nejvyšší míru konfidence u nalezených shod, nicméně jsou výpočetně náročné. Od určitého počtu obrázků v databázi by úzkým hrdlem systému bylo právě porovnávání shod.

Algoritmus ORB využívá pro určení shod binárních deskriptorů, které jsou paměťově méně náročné a dosahuje díky tomu násobně rychlejšího porovnávání. Dostupné studie a vlastní provedené experimenty potvrzují domněnky o poklesu invariance vůči rotaci, změně měřítka a dalším modifikacím. Ztráta ale není markantní a algoritmus je stále dobře použitelný pro účely vyhodnocování shod vůči databázi obrázků. [12]

Součástí experimentů byl také algoritmus AKAZE, který pracuje v nelineárním prostoru. Dosahuje vysoké rychlosti porovnávání při zachování robustních deskriptorů. [13] Oproti algoritmu ORB se však ukázalo jako obtížné limitovat počet extrahovaných deskriptorů. Vzhledem k povaze nelineárního prostoru algoritmus extrahuje řádově rozdílné počty deskriptorů pro obrázky stejných rozměrů. Při realizaci systému je cílem dosáhnout co nejvyšší úspory přenášených dat z mobilní aplikace a proto je důležité mít kontrolu nad počtem extrahovaných deskriptorů. Pro analýzu obrázků byl tedy vybrán algoritmus ORB. Architektura systému je ale modulární, nezávislá na vybraném algoritmu, a v případě potřeby není problém algoritmus vyměnit.

1.5 Metody porovnání deskriptorů

Výsledkem analýzy obrázku jsou deskriptory popisující vizuální okolí klíčových bodů. Následným porovnáním deskriptorů dvou obrázků lze zjistit jejich vzájemnou shodu. Samotné porovnání je nejčastěji založené na vyhledání nejbližších deskriptorů a následném zpřesnění výsledku.

1.5.1 Vyhledání nejbližších deskriptorů

Deskriptory jsou porovnány na základě jejich vzájemné vzdálenosti, vypočtené stanovenou metrikou. Pro každý deskriptor z prvního obrázku je nalezen příslušný deskriptor z druhého obrázku s nejmenší vzdáleností. Výsledkem jsou dvojice nejbližších deskriptorů společně s hodnotou jejich vzdálenosti, pomocí které je možné dále zpřesnit výsledek.

1.5.1.1 Brute-Force Matcher

Nejjednodušším způsobem nalezení nejbližších deskriptorů je řešení hrubou silou. Pro každý deskriptor z prvního obrázku je vypočtena vzdálenost ke všem deskriptorům druhého obrázku a výsledkem jsou dvojice s nejmenší vzdáleností.

Pro vypočtení vzdálenosti je nutné stanovit jaká metrika bude základem výpočtu. Volba je závislá na typu porovnávaných deskriptorů. Mezi základní metriky patří Euklidovská nebo Manhattanská vzdálenost, které dobře fungují s deskriptory extrahovanými algoritmem SIFT nebo SURF. Pro deskriptory reprezentované binárním řetězcem (případ algoritmu ORB nebo AKAZE) je možné použít Hammingovu vzdálenost. [14] Výhodou je provádění výpočtu pouhým počítáním bitů a aplikováním operace XOR, která je v moderních procesorech velmi rychlá. Dochází tedy k celkovému urychlení vyhledávání.

Brute-Force Matcher defaultně vyhledává pouze dvojice deskriptorů s nejmenší vzdáleností. V některých případech je ale užitečné k danému deskriptoru z prvního obrázku vyhledat více blízkých deskriptorů z druhého obrázku. Využitím algoritmu k nejbližších sousedů lze vyhledat k nejbližších dvojic pro každý deskriptor.

Příklad 1.3: Vyhledání dvojic nejbližších deskriptorů

1.5.1.2 FLANN Based Matcher

FLANN je zkratkou pro Fast Library for Approximate Nearest Neighbors. Jedná se o knihovnu, která obsahuje kolekci algoritmů optimalizovaných pro rychlé vyhledávání nejbližších sousedů. Oproti Brute-Force Matcher je násobně rychlejší zejména při analýze velkých datasetů obsahujících deskriptory reprezentované čísly s plovoucí desetinnou čárkou (algoritmy SIFT a SURF). [14]

1.5.2 Vyhodnocení shody

Na základě dvojic nejbližších deskriptorů je již možné vyhodnotit, v jaké míře se obrázky shodují. Pro každou dvojici deskriptorů je známa jejich vzdálenost. Jednoduše lze definovat akceptační kritérium jako maximální hodnotu vzdálenosti deskriptorů. Dvojice, jejichž vzdálenost je nižší než stanovená hodnota jsou akceptovány jako validní, ostatní jsou vyřazeny. V případě dostatečného počtu validních dvojic jsou obrázky vyhodnoceny jako shodné.

1.5.2.1 Poměrový test

Kritérium zohledňující pouze vzdálenost dvojic deskriptorů není často dostatečné. Některé deskriptory jsou více diskriminační než ostatní a dochází k akceptaci nevalidních dvojic. Efektivnější metodu pro nalezení validních dvojic navrhl David G. Lowe jako součást své práce o algoritmu SIFT, je ale použitelná nezávisle na zvoleném algoritmu pro extrakci deskriptorů.

Příklad 1.4: Eliminace falešných dvojic poměrovým testem

Metoda je založena na porovnání vzdálenosti nejbližší dvojice deskriptorů s druhou nejbližší. Pokud je poměr jejich vzdáleností vyšší než určitá mez, jedná se pravděpodobně o nevalidní dvojici. Motivací pro uvedený postup je očekávání, že v případě validních dvojic bude nejbližší dvojice mít výrazně nižší vzdálenost než druhá nejbližší. Autor doporučuje nastavit mez pro poměr vzdáleností mezi 0,7 – 0,8 a uvádí, že je možné dosáhnout eliminace až 90 % falešných dvojic při ztrátě pouhých 5 % dvojic validních. [7]

1.5.2.2 Homografie

Homografie je matice o rozměrech 3x3 popisující transformaci shodných bodů mezi dvěma obrázky. Pro její výpočet je nutné znát alespoň 4 dvojice shodujících se bodů. Výpočtem homografie pro nalezené dvojice nejbližších deskriptorů a zobrazením perspektivní transformace lze získat relativně přesnou pozici prvního obrázku uvnitř druhého obrázku.

Dvojice, které se nenacházejí uvnitř transformovaného prvního obrázku, negativně ovlivňují výpočet homografie. Jejich odfiltrování je zajištěno iterativním algoritmem RANSAC – Random sample consensus, který klasifikuje dvojice jako tzv. inliers nebo outliers. Ověřením determinantu navíc lze zjistit zachování orientace, tedy zda se jedná o kvalitní homografii. [15]

Příklad 1.5: Lokalizace pozice obrázku využitím homografie

1.5.3 Klasifikace obrázků

V případě velké databáze obrázků může být výhodné jednotlivé obrázky určitým způsobem klasifikovat. Databáze se tak rozdělí do několika skupin obrázků, které sdílí podobné vlastnosti. Přijatý snímek je pak nejprve klasifikován a vyhledávání dále probíhá pouze v příslušné skupině.

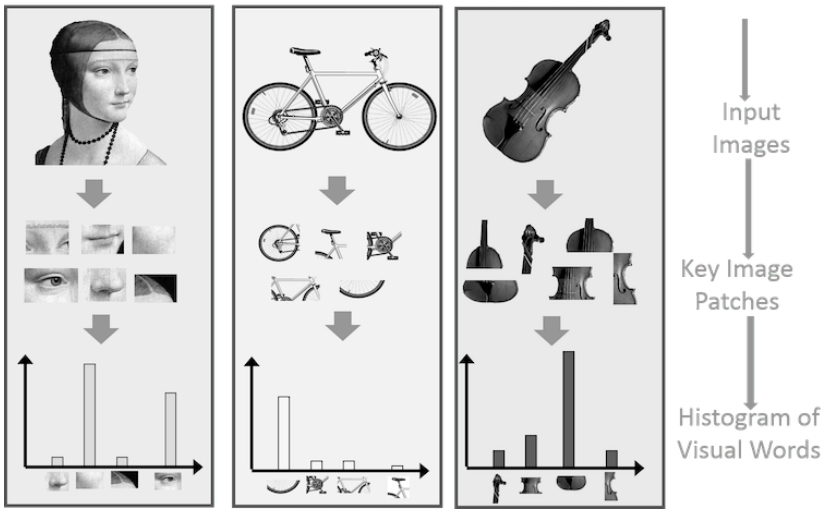
1.5.3.1 Bag of Words

Pro potřeby počítačového vidění se nejčastěji používá klasifikace Bag of Visual Words, vycházející z Bag of Words. Jedná se o reprezentační model využívaný pro zpracování a klasifikaci textových dokumentů. Text je převeden na „balík“ slov, obvykle vyjádřený vektorem dvojic, které reprezentují jednotlivá unikátní slova a počet jejich výskytu v daném textu. Například text „Club Mate je limonáda. Petr má rád Club Mate.“ je reprezentován následujícím vektorem:

```
{Club:2,Mate:2,je:1,limonada:1,Petr:1,ma:1,rad:1}
```

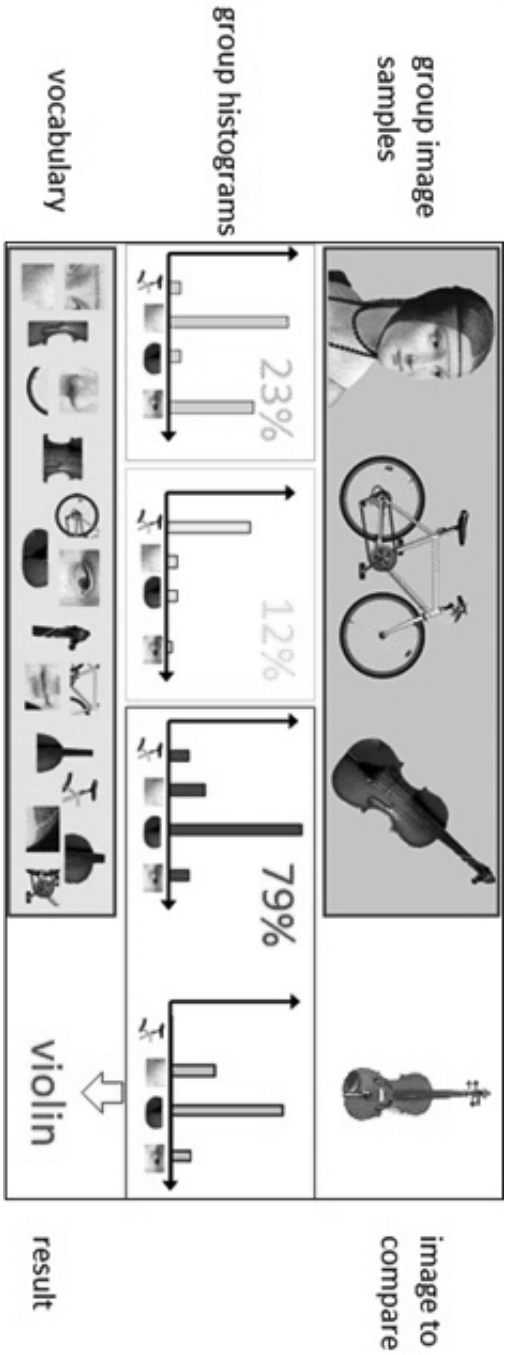
1.5.3.2 Bag of Visual Words

Každý obrázek je možné reprezentovat „balíkem“ vizuálních slov. Obdobně jako v případě textu se jedná o vektor dvojic. Každá dvojice obsahuje vizuální slovo a počet jeho výskytu v daném obrázku. Vizuální slovo tvoří klíčový bod a příslušný deskriptor. Vektor je často převáděn na histogram zobrazující procentuální výskyt jednotlivých vizuálních slov.



Příklad 1.6: Reprezentace obrázků jako Bag of Visual Words

Pro klasifikaci obrázků je neprve nutné vytvořit tzv. slovník vizuálních slov z trénovacího datasetu. Dataset by ideálně měl reprezentovat skupiny odpovídající obrázkům určeným k následné klasifikaci. V opačném případě může docházet k chybné klasifikaci. Následně je pro každou skupinu vypočítán histogram představující výskyt jednotlivých vizuálních slov ze slovníku. Klasifikace nového obrázku pak probíhá výpočtem histogramu a jeho následným porovnáním s histogramy jednotlivých skupin. [16, 17]



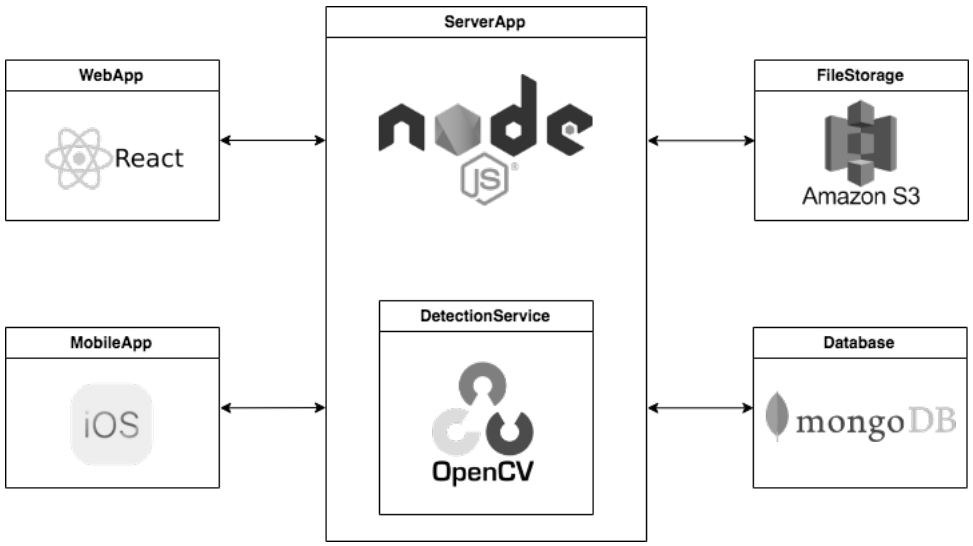
Příklad 1.7: Klasifikace obrázků na základě Bag of Visual Words

Kapitola 2

Návrh

2.1 Architektura systému

Systém se skládá z několika komponent. Hlavní komponentou je serverová aplikace, která pro naplnění požadované funkcionality komunikuje s dalšími aplikacemi a službami. Navenek však působí jako celek s jedním aplikačním rozhraním, pomocí kterého dochází ke komunikaci s knihovnou pro platformu iOS a webovou aplikací.



Příklad 2.1: Diagram architektury systému

2.1.1 Komponenty systému

2.1.1.1 Serverová aplikace

Hlavní komponenta systému poskytující aplikační rozhraní, prostřednictvím kterého knihovna pro mobilní aplikace a webová aplikace získavají a odesílají potřebná data. Zajišťuje zabezpečený přístup – požadavek může vždy vykonat pouze oprávněný uživatel. Slouží jako zprostředkovatel dalších úzce specifikovaných služeb a aplikací.

2.1.1.2 Služba pro detekci shod

Samotná analýza a detekce shodných obrázků je komplexní problém, který se svou podstatou zásadně liší od zbytku serverové aplikace. Pro lepší údržbu systému a případné škálování je vhodné danou funkcionalitu implementovat jako samostatnou službu. Přístup k ní je řízen a zprostředkováván serverovou aplikací.

2.1.1.3 Databáze

Serverová aplikace komunikuje s databázovým serverem, který zprostředkovává přístup k databázi. V případě přijetí oprávněného požadavku k persistentnímu zaznamenání dat kontaktuje databázový server a ten zapíše data do databáze. Obdobně v případě přijetí požadavku k získání dat. Dojde k přečtení dat z databáze a předání nazpět serverové aplikaci, která je odešle jako odpověď na původní požadavek.

2.1.1.4 Úložiště souborů

Systém pracuje i s daty netextové povahy, konkrétně se jedná o obrázky určené k vyhledávání, které není vhodné ukládat do databáze z důvodu zbytečné zátěže. [18] Samotné soubory s obrázky jsou uloženy mimo databázi ve specializovaném úložišti souborů. V databázi je potom uložena pouze cesta k danému souboru. Pokud tedy serverová aplikace obdrží požadavek pro čtení nebo zápis souboru, předá jej dále službě obsluhující dané úložiště.

2.1.1.5 Knihovna pro platformu iOS

Mobilní aplikace využívá funkce knihovny pro kontinuální odesílání snímků okolního prostředí na server a následnému vyhodnocení výsledků. Knihovna komunikuje se serverovou aplikací výhradně prostřednictvím navrženého aplikačního rozhraní. Jednotlivé snímky jsou před odesláním vhodně zkomprimovány pro úsporu objemu přenášených dat.

2.1.1.6 Webová aplikace

Vstupním bodem celého systému je webová aplikace poskytující možnost vytvoření uživatelského účtu a následnou správu obrázků určených k vyhledávání. Prostřednictvím navrženého aplikačního rozhraní webová aplikace nejprve registruje nového nebo přihlásí stávajícího uživatele. Všechny následující požadavky v rámci otevřené uživatelské relace jsou podepisovány identitou přihlášeného uživatele.

2.1.2 Možnosti implementace serverové aplikace

Serverová aplikace je hlavní komponentou celého systému. Pro implementaci je tedy nutné vybrat kombinaci programovacího jazyka a příslušných frameworků[10], které dokáží obsloužit všechny ostatní komponenty systému. Mezi nejpoužívanější programovací jazyky pro implementaci serverových aplikací se řadí Java, PHP, Python a v posledních letech i JavaScript.

2.1.2.1 Java

Objektově orientovaný programovací jazyk představený společností Sun Microsystems v roce 1995. Od roku 2007 je uvolněn jako open-source[11]. Jedná se o jeden z nejpoužívanějších a nejpopulárnějších programovacích jazyků. [19] Okolo Javy se v průběhu let vytvořil komplexní ekosystém obsahující řešení téměř pro všechny výpočetní platformy od čipových karet až po distribuované systémy. Pro vývoj serverových aplikací se jedná o platformu JavaEE.

JavaEE, neboli Java Enterprise Edition, je ustálenou komplexní platformou určenou především pro vývoj rozsáhlých informačních systémů a podnikových aplikací. Obsahuje velké množství specifikací, například pro implementaci aplikačního rozhraní, přístup k databázím, integraci aplikací, vývoj webových aplikací a mnoho dalších. Aplikace a služby jsou následně spouštěny prostřednictvím tzv. aplikačního serveru. Mezi nejznámnější aplikační servery patří GlassFish, Apache Tomcat nebo WildFly. [20]

2.1.2.2 PHP

PHP je rekurzivní zkratkou názvu PHP: Hypertext Preprocessor. Jedná se o skriptovací programovací jazyk původně vytvořený Rasmusem Lerdorfem v roce 1994 pro vývoj dynamických webových aplikací. Aktuálně jeho další rozvoj zajišťuje skupina The PHP Group. [21] Podporuje mnoho knihoven pro přístup k databázím, implementaci aplikačního rozhraní a další funkce nezbytné pro vývoj serverové aplikace. Často je využíván v kombinaci s některým z dostupných frameworků, mezi nejpoužívanější patří Symfony nebo Nette.

[10] Aplikační rámec usnadňující vývoj aplikací.

[11] Software s otevřeným zdrojovým kódem a obvykle i bezplatnou dostupností.

V kombinaci s relační databází MySQL tvoří základ mnoha populárních publikačních systémů jako je Drupal, Joomla nebo WordPress. Zejména poslední jmenovaný se těší velké oblibě u správců i vývojářů webových stránek. Přes 30 % ze všech webových stránek využívá právě WordPress. [22]

2.1.2.3 Python

Programovací jazyk Python navrhl v roce 1991 Guido van Rossum jako zábavný programátorský projekt. V průběhu času dospěl v plnohodnotný jazyk podporující většinu hlavních programovacích paradigmat. Aktuálně je vyvíjen komunitou jako open-source a jsou udržovány dvě vzájemně nekompatibilní verze Python 2.x a Python 3.x. Ukončení podpory verze 2.x se očekává v roce 2020. Těší se velké oblibě zejména v odvětvích datové analýzy, strojového učení a umělé inteligence. Mezi nejznámnější frameworky pro vývoj webových a serverových aplikací patří Django nebo Flask.

2.1.2.4 JavaScript

Objektově orientovaný skriptovací jazyk představený v roce 1995 společnostmi Netscape a Sun Microsystems jako doplněk k jazykům HTML[12] a Java. S jazykem Java nemá kromě názvu téměř nic společného. Původní motivací pro vývoj byla potřeba „oživit“ statické webové stránky. JavaScript byl tedy vydán jako součást webového prohlížeče Netscape Navigator a umožňoval přidání dynamické interakce do webových stránek bez nutnosti komunikace se serverem. [23] Jazyk byl později standardizován asociací ECMA[13] a postupně se stal běžnou součástí všech moderních prohlížečů webových stránek.

V roce 2009 navrhl Ryan Dahl běhové prostředí Node.js, určené pro spouštění JavaScript kódu na straně serveru. Kombinací JavaScript enginu V8 od společnosti Google a několika nízkoúrovňových knihoven pro I/O operace[14] vytvořil vysoce efektivní rozhraní pro implementaci serverových aplikací. Aplikace využívají smyčku událostí a neblokující I/O operace, díky tomu jsou snadno škálovatelné a dokáží obsluhovat velké množství souběžných klientských spojení. [24]

2.1.2.5 Výběr programovacího jazyka

Navrhnutý systém je možné implementovat v každém z popsaných jazyků, volba tedy závisí především na preferencích vývojáře. Pro implementaci serverové aplikace byl zvolen jazyk JavaScript, který se stal trendem ve vývoji webových aplikací a dle průzkumů se jedná o nejpopulárnější jazyk i mezi vývojáři serverových aplikací. [25] Velkou výhodou je potom možnost používání stejného základního jazyka pro webovou i serverou aplikaci.

[12] HyperText Markup Language – značkovací jazyk pro tvorbu webových stránek.
[13] Mezinárodní nezisková asociace pro normalizaci informačních a komunikačních systémů.
[14] Input/Output – vstupně výstupní operace.

2.2 Databázový model

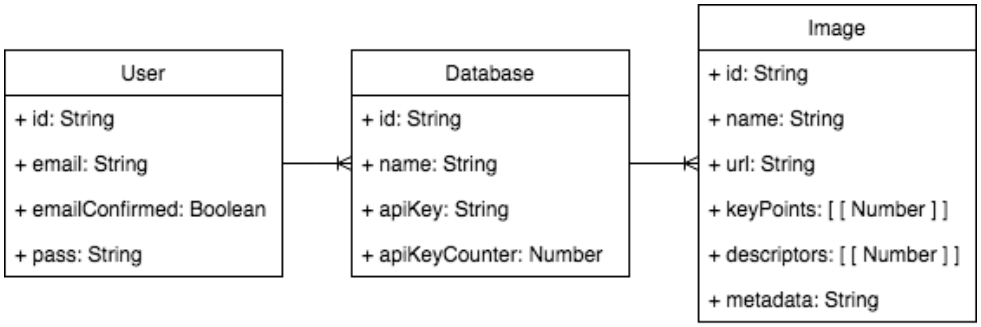
2.2.1 Reprezentace databáze obrázků

Serverová aplikace slouží mimo jiné pro vyhodnocení shod přijatého snímku z mobilní aplikace vůči uložené databázi obrázků. Každý přijatý snímek je nutné porovnat se všemi obrázky v dané databázi. Pro každé porovnání musí aplikace nejprve provést analýzu dané dvojice obrázků. To je výpočetně náročný úkon. S rostoucím počtem obrázků v databázi není únosné opakovaně provádět jejich analýzu. Bylo nutné navrhnout lepší řešení.

Pro obrázky uložené v databázi je analýza nezávislá na přijatém snímku, vždy vrací stejné klíčové body a deskriptory. Toho lze využít pro výraznou úsporu výpočetního výkonu. Pro každý nově přidaný obrázek je provedena analýza. Získané klíčové body a deskriptory jsou uloženy společně s obrázkem. Při následném vyhodnocení shod již není nutné obrázek znovu analyzovat, použijí se uložená data.

2.2.2 Návrh databázového modelu

Databázový model serverové aplikace obsahuje celkem tři typy entit. Entita typu User reprezentuje jednotlivé uživatele systému, kteří se zaregistrují prostřednictvím webové aplikace. Každý registrovaný uživatel si může vytvořit více databází, reprezentovaných entitou typu Database. Každá databáze potom obsahuje obrázky určené k vyhledávání – entita typu Image. Mezi entitami User a Database je tedy vazba 1:N, stejně jako mezi entitami Database a Image. Vazby reflektují výše uvedené požadavky – jeden uživatel může vytvořit N databází a každá databáze dále obsahuje N obrázků.



Příklad 2.2: Databázový model serverové aplikace

2.2.2.1 Entita typu User

- **id** – Unikátní identifikátor uživatele.
- **email** – Emailová adresa uživatele zadaná při registraci a využívaná pro následná přihlášení uživatele.
- **emailConfirmed** – Příznak reprezentující, zda nově registrovaný uživatel již potvrdil svou emailovou adresu.
- **pass** – Otisk uživatelského hesla, využívaný pro přihlášení uživatele. Z otisku hesla není možné zjistit zadané heslo.

2.2.2.2 Entita typu Database

- **id** – Unikátní identifikátor databáze obrázků.
- **name** – Název databáze obrázků.
- **apiKey** – Unikátní klíč zajišťující oprávněný přístup k databázi prostřednictvím aplikačního rozhraní serverové aplikace.
- **apiKeyCounter** – Pomocný čítač sloužící pro bezpečné vygenerování nového unikátního klíče.

2.2.2.3 Entita typu Image

- **id** – Unikátní identifikátor obrázku.
- **name** – Název obrázku.
- **url** – URL[15] adresa určující lokaci obrázku v rámci příslušného úložiště souborů.
- **keyPoints** – Matice čísel reprezentující klíčové body daného obrázku.
- **descriptors** – Matice čísel reprezentující deskriptory daného obrázku.
- **metadata** – Metadata definující příslušné akce, které se mají vykonat po detekci obrázku v prostředí rozšířené reality.

[15] Uniform Resource Locator – identifikátor zdrojů na internetu.

2.2.3 Databázové technologie

Serverová aplikace je přímo propojena s databázovým serverem. Ten zprostředkovává přístup k databázi, která slouží jako hlavní zdroj dat. Téměř s každým požadavkem přijatým prostřednictvím aplikačního rozhraní je nutné kontaktovat databázový server, který se postará o čtení nebo zápis příslušných dat do databáze.

Při výběru řešení databázového serveru je nutné brát ohledy na povahu aplikace a její budoucí vývoj. Volba nesprávné technologie může zásadně ovlivnit výkon celého systému. Databázových technologií existuje velké množství. Z pohledu přístupu k datům je možné databáze rozdělit do dvou základních skupin – SQL a NoSQL.

2.2.3.1 SQL databáze

Skupina databází, které získaly svůj název dle jejich dotazovacího jazyka SQL – Structured Query Language. Jedná se o tzv. relační databáze založené na relačním modelu. Základem každé databáze jsou tabulky obsahující řádky, které reprezentují jednotlivé záznamy. Každá z tabulek může dále obsahovat sloupce reprezentující relace mezi jednotlivými tabulkami (tzv. cizí klíče). Pro operace s daty se používá právě dotazovací jazyk SQL.

Počátky relačních databází sahají až do 70. let 20. století. Postupně došlo ke standardizaci jazyka SQL a dnes jsou k dispozici desítky databází, které podporují základní konstrukce jazyka. Často se však liší v pokročilých funkcích a některých detailech. Většina relačních databází zaručuje, že všechny provedené transakce dodržují soubor vlastností tzv. ACID. [26]

- **Atomicity** – Transakce se úspěšně provede celá, nebo se neprovede vůbec. Předchází modifikaci pouze některých dat v případě, že transakce selže v průběhu vykonávání.

- **Consistency** – Data budou zapsána pouze pokud splňují všechny definované podmínky a omezení. Je tedy zajištěno, že transakce převede databázi z validního stavu do jiného validního stavu.

- **Isolation** – Operace které probíhají uvnitř transakce neovlivní ostatní transakce. V závislosti na módu izolace často nejsou vidět ani výsledky nedokončených transakcí.

- **Durability** – Změny provedené uspěšně vykonanými transakcemi jsou bezpečně uloženy. Je zajištěna odolnost vůči jakýmkoli chybám nebo selháním databáze.

Výhodou relačních databází je především zajištění datové integrity a deklarativní syntaxe ustáleného jazyka SQL. Nicméně právě kvůli dodržování vlastností ACID je jejich horizontální škálování[16] často problematické. Mezi nejznámnější relační databáze patří Oracle Database, MySQL, Microsoft SQL Server nebo PostgreSQL.

2.2.3.2 NoSQL databáze

Pro databáze jejichž základem není relační model se vžilo označení NoSQL, původně znamenající „Not SQL“. Nicméně některé databáze i přesto podporují dotazování prostřednictvím jazyka SQL, proto se často lze setkat i s výkladem „Not Only SQL“. Jedná se o relativně volnou definici, zahrnující množství databází různých typů. Nejčastěji se jedná o databáze dokumentové, grafové nebo tzv. key-value.

- **Key-value** – Databáze založené na slovníku unikátních klíčů a hodnot, kdy jeden klíč odpovídá jedné hodnotě nebo datové struktuře. Klíče mohou být navíc řazeny abecedně pro vysokou efektivitu čtení z databáze. Mezi nejznámnější zástupce patří Redis nebo Riak.

- **Dokumentové** – Rozšiřují key-value databáze. Unikátní klíče neodkazují pouze na hodnotu, ale na dokument v definovaném formátu (JSON, BSON[17], XML[18]). Často disponují vlastním dotazovacím jazykem pro operace nad daty na základě obsahu dokumentů. Patří sem MongoDB, Couchbase nebo Apache CouchDB.

- **Grafové** – Jejich základem je většinou dokumentová databáze, která navíc obsahuje další vrstvu reprezentující vztahy mezi jednotlivými dokumenty. Vzniklý graf pak umožňuje rychlé přechody mezi jednotlivými uzly při čtení nebo zápisu dat. Grafovou databází je například Neo4J nebo Apache Giraph

Společným rysem téměř všech NoSQL databází je rychlé provádění operací za cenu nedodržování některých ACID vlastností. Díky tomu je také umožněno snadné horizontální škálování. Při škálování databáze na více serverů je nutné brát v potaz vlastnost, kterou v roce 1999 popsal Eric Brewer, známou jako CAP teorém. [27]

[16] Škálování systému přidáním více prvků, zpravidla serverů.
[17] Binary JSON – Binární reprezentace JSON formátu.
[18] Extensible Markup Language – Značkovací jazyk a také způsob zápisu dat.

- **Consistency** – Všechny servery systému poskytují stejná data. Uživatel má jistotu, že vždy dostane stejnou odpověď nezávisle na tom, jaký server jeho požadavek obslouží.

- **Availability** – Systém vždy poskytne odpověď na přijatý požadavek. Odpověď musí poskytnout i v případě, že nemá k dispozici aktuální nebo konzistentní data.

- **Partition Tolerance** – Systém dále funguje jako celek i v případě, že některý ze serverů není z jakéhokoli důvodu dostupný.

Dle teorému není možné zaručit všechny tři vlastnosti. Pokud má tedy systém dále fungovat i v případě výpadku některého ze serverů, je nutné zvolit mezi konzistencí dat nebo dostupností systému. Dle chování databáze v takové situaci je možné databáze rozdělit na skupiny AP a CP. Relační databáze, pro dodržení ACID vlastností, většinou upřednostňují konzistenci nad dostupností – skupina CP. Existují techniky pro snížení nedostupnosti na nezbytné minimum, takové databáze se pak označují jako „vysoce dostupné“. NoSQL databáze často obětují konzistenci dat pro zajištění dostupnosti systému – skupina AP. Je možné narazit na označení takových databází jako „eventuálně konzistentní“, kdy všechny servery systému eventuálně (typicky v rámci milisekund) poskytují stejná data. Výše uvedené rozdělení neplatí vždy. Existují relační databáze spadající do skupiny AP i NoSQL databáze ze skupiny CP. V některých případech je dokonce možné vybrat, zda se má v případě výpadku upřednostit konzistence nebo dostupnost.

2.2.3.3 Volba technologie

Vzhledem k povaze systému a navrženému databázovému modelu je nutné mít k dispozici možnost snadného škálování celého systému včetně databáze. Vertikální škálování[19] obecně není od určité meze efektivní a horizontální škálování je u klasických relačních databází problematické. NoSQL databáze jsou navržené právě pro snadné škálování systému. Z běžně dostupných NoSQL databází byla zvolena dokumentová databáze MongoDB.

MongoDB je open-source dokumentově orientovaná databáze vyvíjená od roku 2009 společností MongoDB Inc. Základem jsou dokumenty v BSON formátu. Jedná se o binární zápis JSON formátu, který je optimalizovaný pro efektivní ukládání a prohledávání. Databáze je silně konzistentní a poskytuje vysokou dostupnost prostřednictvím automatického failover[20]. Patří mezi nejpoužívanější NoSQL databáze. [28]

[19] Škálování systému optimalizací stávajících prvků, zpravidla zvyšením výkonu serveru.
[20] Přesměrování požadavků na redudantní prvek systému v případě výpadku.

2.2.4 Ukládání souborů

Systém kromě databáze potřebuje také úložiště souborů pro uchování dat, která není vhodné zapisovat do databáze. Jedná se zejmená o grafické soubory obsahující samotné obrázky. Serverová aplikace obdrží požadavek pro přidání nového obrázku do dané databáze obrázků. Požadavek obsahuje soubor s obrázkem a případné další údaje. Nejprve dojde k nahrání souboru do zvoleného úložiště souborů. Následně je vytvořen nový záznam v databázi obsahující URL adresu, na které je nahraný soubor dostupný.

Jako jednoduché úložiště souborů je možné využít diskový prostor na serveru, kde je spuštěna serverová aplikace. Takové řešení ale znamená značné zvýšení nároků na daný server. Populární alternativou je využití služeb některého z poskytovatelů cloudových řešení. Mezi největší poskytovatele patří společnosti Amazon, Google a Microsoft, každý z nich nabízí vlastní řešení úložiště souborů.

- **Amazon S3** – Součást cloudového řešení AWS – Amazon Web Services od roku 2006. Poskytuje úložiště souborů dostupné přes rozhraní typu REST, SOAP[21] a BitTorrent[22]. Amazon S3 využívají světově známé webové služby jako například Netflix, Dropbox nebo Pinterest.
- **Google Cloud Storage** – Služba poskytující cloudové úložiště souborů, součást platformy Google Cloud Platform. Komunikace je možná prostřednictvím REST API.
- **Microsoft Azure Storage** – Součást balíku služeb Microsoft Azure Cloud Services. Poskytuje standardní REST API pro přístup k úložišti.

	Amazon S3	**Google Cloud**	**Microsoft Azure**
0-50 TB/měsíc	$ 0,0230	$ 0,0260	$ 0,0184
50-500 TB/měsíc	$ 0,0220	$ 0,0260	$ 0,0177
500+ TB/měsíc	$ 0,0210	$ 0,0260	$ 0,0170

Tabulka 2.1: Srovnání cloudových úložišť souborů – cena za 1 GB

Ze srovnání je vidět, že všechna tři porovnávaná řešení mají řádově stejné ceny. Zároveň i poskytované služby jsou srovnatelné. Při výběru cloudového úložiště souborů tedy často záleží na preferencích vývojářského týmu a případném záměru využívání více služeb od jednoho poskytovatele. Jako datové úložiště bylo vybráno řešení Amazon S3.

[21] Simple Object Access Protocol – Procedurálně orientovaný komunikační protokol.
[22] Protokol pro decentralizované sdílení souborů mezi klienty.

2.3 Aplikační rozhraní serverové aplikace

2.3.1 REST API

Serverová aplikace poskytuje aplikační rozhraní typu REST pro komunikaci s ostatními aplikacemi prostřednictvím jednoduchých HTTP volání. Veškerá komunikace ze strany webové aplikace a knihovny pro mobilní aplikace probíhá právě přes REST API.

2.3.1.1 Požadavek

Serverová aplikace obdrží HTTP požadavek, který je jednoznačně definovaný URL adresou a typem. URL adresa identifikuje zdroj na serveru, kterého se požadavek týká. Typ požadavku potom specifikuje požadovanou operaci. Standardními typy jsou GET, POST, PUT a DELETE. V některých případech (POST a PUT) obsahuje požadavek i tzv. tělo, které obsahuje data určená pro zápis do databáze.

- **GET** – Požadavek na získání dat reprezentujících specifikovaný zdroj. Obvykle se jedná o detail objektu nebo kolekci objektů určitého typu.
- **POST** – Obvykle využívaný pro vytvoření nového objektu a jeho přiřazení do specifikované kolekce. Objekt je vytvořen z těla požadavku.
- **PUT** – Slouží pro náhradu reprezentace specifikovaného zdroje. Nová reprezentace zdroje je obsažena v těle požadavku.
- **DELETE** – Požadavek pro smazání specifikovaného zdroje, obvykle objektu nebo kolekce objektů.

Následně dochází k vyhodnocení požadavku. Často je nutné kontaktovat další komponenty systému (například databázi v případě požadavku o čtení/zápis dat do databáze). Na základě výsledku je nazpět odeslána příslušná odpověď sestávající ze stavového HTTP kódu a těla odpovědi.

2.3.1.2 Odpověď

Obsah těla odpovědi se liší dle výsledku vyhodnocení požadavku. Pokud došlo k chybě, tělo obsahuje příslušnou chybovou hlášku. V případě úspěšného vykonání najdeme v těle odpovědi data dle typu požadavku. Pro požadavky typu GET je vrácen požadovaný objekt. Pro typ POST nebo PUT se jedná o nově vytvořený nebo modifikovaný objekt. U požadavků typu DELETE obvykle tělo odpovědi neobsahuje žádná data.

Data, která jsou vracena v těle odpovědi, dodržují některý ze standardizovaných formátů pro reprezentaci dat. Aplikační rozhraní typu REST obvykle podporuje formáty XML a JSON. Data ve formátu XML je následně nutné parsovat[23] prostřednictvím speciálního XML parseru. Oproti tomu data ve formátu JSON je obvykle možné rychle namapovat na příslušný objekt. Obecně se jedná o preferovaný způsob reprezentace dat v moderních webových a mobilních aplikacích. [29] Pro reprezentaci dat v těle odpovědi byl tedy zvolen formát JSON.

2.3.1.3 Stavový kód

Vzhledem k tomu, že REST API pro jednotlivé požadavky využívá HTTP požadavků, tak i odpovědi využívají stavových kódů protokolu HTTP. Stavový kód je vždy jednoznačně definován unikátním číslem a slovním popisem. Jejich hlavním účelem je poskytnutí strojově čitelné odpovědi, na kterou může aplikace zareagovat například zpracováním přijatých dat nebo zobrazením chybové hlášky. Navrhnuté REST API pracuje s následujícími stavovými kódy:

- 200 OK – Požadavek byl úspěšně vyhodnocen.
- 201 Created – Požadovaný zdroj byl úspěšně vytvořen.
- 204 No Content – Požadavek byl úspěšný, ale nevrací žádná data.
- 304 Not Modified – Daný zdroj se od předešlého požadavku nezměnil.
- 400 Bad Request – Nesprávná syntaxe požadavku.
- 401 Unauthorized – Uživatel není správně autentizován.
- 403 Forbidden – Uživatel nemá přístup k požadovanému zdroji.
- 404 Not Found – Požadovaný zdroj neexistuje.
- 409 Conflict – Zdroj nelze vytvořit, protože již existuje.
- 500 Internal Server Error – Při zpracování požadavku došlo k chybě.
- 503 Service Unavailable – API je nedostupné.

[23] Proces analýzy posloupnosti formálních prvků využívaný pro konzumaci dat.

2.3.2 Zdroj typu User

```
POST /user
```

Registrace nového uživatele, emailová adresa a heslo je odesláno v těle požadavku. Na zadanou emailovou adresu je zaslán email s verifikačním odkazem. Otevřením odkazu dojde k aktivaci uživatele a umožnění přihlášení.

```
POST /user/verification
```

Aktivace registrovaného uživatele. Požadavek je volán při otevření odkazu z verifikačního emailu. Token pro verifikaci je předáván v těle požadavku.

```
POST /login
```

Vytvoření nové uživatelské relace. Tělo požadavku obsahuje emailovou adresu a heslo uživatele. Při úspěšném vyhodnocení je v těle odpovědi odeslán přístupový token. Ten se dále používá u všech následujících požadavků pro ověření uživatele.

```
GET /user
```

Detail přihlášeného uživatele. V aktuální verzi systému obsahuje pouze emailovou adresu. Heslo se z bezpečnostních důvodů nezobrazuje.

```
PUT /user
```

Editace přihlášeného uživatele. Lze využít pro změnu hesla. Emailovou adresu není možné v aktuální verzi systému změnit.

2.3.3 Zdroj typu Database

```
GET /database
```

Kolekce všech databází přihlášeného uživatele. Každá databáze obsahuje identifikátor a název, API klíč není zobrazován.

```
GET /database/:id
```

Detail zvolené databáze, `:id` v URL požadavku je nutné zaměnit za validní identifikátor databáze. Obsahuje identifikátor, název a API klíč.

```
PUT /database/:id
```

Editace zvolené databáze. Lze využít pro změnu názvu databáze.

```
PUT /database/:id/apikey
```

Vygenerování nového API klíče pro zvolenou databázi. Starý klíč je zneplatněn a dále není akceptován.

```
POST /database
```

Vytvoření nové databáze, název databáze je předán v těle požadavku.

```
DELETE /database/:id
```

Smazání zvolené databáze. Dojde rovněž ke smazání všech obrázků, které daná databáze obsahuje.

2.3.4 Zdroj typu Image

```
GET /database/:id/image?page=0&limit=50
```

Vybraná kolekce obrázků ze zvolené databáze. Každý obrázek obsahuje identifikátor, název a URL adresu, metadata nejsou zobrazována. Databáze může obsahovat stovky až tisíce obrázků, v takovém případě by vyhodnocení požadavku trvalo neúnosně dlouho. Zdroj proto podporuje stránkování pomocí URL parametrů. Parametr `page` určuje požadovanou stránku a parametr `limit` potom počet obrázků na jednu stránku (maximum je 100). Uvedený příklad tedy zobrazí prvních 50 obrázků dané databáze.

```
GET /database/:id/image/:id
```

Detail zvoleného obrázku z dané databáze. Obsahuje identifikátor, název, URL adresu a metadata.

```
PUT /database/:id/image/:id
```

Editace zvoleného obrázku z dané databáze. Lze využít pro změnu názvu nebo metadat.

```
POST /database/:id/image
```

Vytvoření nového obrázku v dané databázi. Tělo požadavku obsahuje data ve formátu `multipart/form-data` reprezentující soubor s obrázkem, jeho název a metadata. Po přijetí požadavku je soubor s obrázkem nejprve analyzován a následně nahrán do úložiště souborů. Získané klíčové body, deskriptory a URL adresa jsou, společně s přijatými parametry, uloženy do dané databáze jako nový obrázek.

```
DELETE /database/:id/image/:id
```

Smazání zvoleného obrázku z dané databáze.

```
GET /files/:file
```

Stažení zvoleného souboru z úložiště souborů. Řetězec `:file` v URL požadavku je nutné zaměnit za název požadovaného souboru.

2.3.5 Vyhledání shod

```
POST /database/:id/detection
```

Vytvoření nové detekce shod nad danou databází. Požadavek je kontinuálně volán mobilní aplikací pro odeslání snímku okolního prostředí. S ohledem na časté volaní požadavku a snahu o minimalizaci objemu dat, přenášených mezi mobilní aplikací a serverem, bylo nutné zvolit vhodnou datovou reprezentaci odesílaných snímků.

Byla zvolena možnost analýzy snímku v mobilním zařízení před samotným odesláním na server. Tělo požadavku pak obsahuje dvě matice čísel, reprezentující klíčové body a deskriptory daného snímku. Dohromady představují zlomek velikosti původního souboru. Přijaté matice jsou následně porovnány s klíčovými body a deskriptory všech obrázků dané databáze. V případě nalezení shody je jako odpověď odeslán detail shodného obrázku. Pokud nedojde k nalezení žádné shody, je vrácena chyba reprezentovaná stavovým kódem `404 Not Found`.

2.4 Analýza obrázků na serveru

Algoritmy pro analýzu obrázků popsané v sekcích 1.4 a 1.5 je teoreticky možné implementovat vlastním řešením v jakémkoli programovacím jazyce. Základem jednotlivých algoritmů jsou však netriviální operace s maticemi a často složité matematické postupy. Proto je výhodné využít některou z volně dostupných implementací, která byla řádně otestována.

2.4.1 OpenCV

Multiplatformní open-source knihovna pro počítačové vidění a zpracování obrazu v reálném čase. Původně představená společností Intel v roce 2000 s účelem poskytnout otevřený a optimalizovaný základ pro vývoj aplikací pracujících s počítačovým viděním. Postupně došlo k rozšíření o další moduly. Vývoj aktuálně zajišťuje nezisková organizace OpenCV.org. [30]

Knihovna je dnes de facto standardem v dané oblasti. Nachází uplatnění například pro rozpoznávání obličejů, sledování pohybujících se objektů, rozpoznávání objektů na základě neuronových sítí nebo právě vyhledávání shodných obrázků. Samotná nízkoúrovňová implementace v programovacím jazyce C/C++ garantuje snadnou spustitelnost z většiny běžně používaných programovacích jazyků. Knihovna je dobře dokumentovaná a má jednu z největších komunit vývojářů v dané oblasti.

Jedním z modulů knihovny je `features2d`. Obsahuje algoritmy pro extrakci lokálních vlastností z obrázků jako je Harris Corner Detector, ORB, KAZE, AKAZE a další. Pro porovnání deskriptorů je k dispozici Brute-Force Matcher i FLANN Based Matcher. Modul dále obsahuje metody potřebné pro klasifikaci obrázků na základě Bag of Visual Words. Algoritmy SIFT a SURF jsou patentované a nachází se proto v dedikovaném modulu `xfeatures2d`. Jejich využití pro edukativní účely je bezplatné, pro komerční účely je nutné kontaktovat vlastníky patentů. Dalším užitečným modulem je `calib3d`, který obsahuje algoritmy a metody pro výpočet homografie.

2.4.2 Využití knihovny OpenCV v prostředí Node.js

Navrhnutá serverová aplikace bude implementovaná v programovacím jazyce JavaScript za využití běhového prostředí Node.js. Pro využití knihovny OpenCV je tedy nutné zajistit spuštění C/C++ kódu. Node.js poskytuje možnost implementace tzv. nativních addonů. Jedná se o dynamicky linkované sdílené objekty napsané v jazyce C++. Po jejich nahrání do prostředí Node.js mohou být používány stejně jako standardní moduly. [31]

Knihovna OpenCV je rozsáhlý projekt. Navržená aplikace využívá pouhý zlomek funkcí, ale i přesto se jedná o desítky tříd a metod. Pro jejich volání z prostředí Node.js by bylo nutné implementovat velké množství addonů. Naštěstí existují volně dostupné knihovny, které poskytují právě potřebnou mezivrstvu mezi Node.js a knihovnou OpenCV.

2.4.2.1 node-opencv

Knihovnu pro Node.js od roku 2012 vytváří Peter Branden ve spolupráci s vývojářskou komunitou. Umožňuje volání hlavních funkcí knihovny OpenCV verze 2.x, podpora pro verzi 3.x zatím není dokončena. Implementace jednotlivých modulů je různá. Například pro modul `features2d` je dostupná pouze obecná funkce pro porovnání dvou obrázků, jednotlivé funkce modulu nejsou přístupné.

2.4.2.2 opencv4nodejs

Knihovna, kterou v roce 2017 vytvořil a dále aktivně rozvíjí Vincent Mühler. Podporuje volání hlavních funkcí knihovny OpenCV verze 3.x a dále zpřístupňuje podstatné části většiny modulů. Z modulu `features2d` je k dispozici většina funkcí potřebných pro analýzu a porovnání obrázků.

2.4.2.3 Výběr řešení

Pro využití knihovny OpenCV v prostředí Node.js bude použita knihovna opencv4nodejs, která podporuje podstatnou část potřebného modulu `features2d`. K dosažení požadované funkcionality bude pravděpodobně nutné rozšířit knihovnu pouze o podporu několika málo chybějících funkcí.

2.5 Funkcionalita iOS knihovny

Systém je využíván vývojáři mobilních aplikací pro rozšíření jejich aplikace o vyhledávání definovaných obrázků v prostředí rozšířené reality. Za tím účelem poskytuje serverová aplikace možnost vyhledání shod ve snímku přijatém prostřednictvím aplikačního rozhraní (viz 2.3.5). Kontinuálním zachycováním snímků okolního prostředí a následným odesíláním na server je pak dosaženo efektu vyhledávání obrázků v prostředí rozšířené reality. Pro usnadnění implementace je vývojářům poskytnuta knihovna, která výše uvedený proces automatizuje a zrychluje tak vývoj mobilní aplikace.

2.5.1 Zachycení snímku

Hlavní funkcí knihovny je poskytnutí UI[24] komponenty, která zobrazuje data z fotoaparátu daného zařízení. Zachycování snímků a jejich odesílání na server probíhá automaticky na pozadí bez jakékoli interakce. Uživatel výsledné mobilní aplikace tedy vidí pouze data z fotoaparátu a případné akce vyvolané rozpoznáním některého z definovaných obrázků. Knihovna však pouze informuje aplikaci o nalezené shodě. Implementace samotných akcí je již záležitostí vývojáře mobilní aplikace. Pro získání dat z fotoaparátu existuje na platformě iOS více možností:

- **UIImagePicker** – Jednoduchá třída poskytující rozhraní pro pořizování fotografií a videí. Má omezené možnosti a předdefinované chování.
- **AVFoundation** – Framework pro zaznamenávání a zpracování audiovizuálních dat na platformě iOS. Poskytuje plnohodnotný a neomezený přístup k fotoaparátům daného zařízení, ale klade vyšší nároky na implementaci.
- **ARKit** – Relativně nový framework určený pro implementaci funkcí v rozšířené realitě. Mimo jiné obsahuje i třídu ARSCNView, která poskytuje přístup k fotoaparátu a možnost zachycení aktuálního snímku. Nevýhodou je pouze absence podpory pro starší verze systému, framework je dostupný od iOS verze 11.0 a vyšší. Pro získání dat z fotoaparátu byl vybrán právě framework ARKit.

2.5.2 Využití knihovny OpenCV na platformě iOS

Aplikační rozhraní serverové aplikace očekává, že přijatý snímek bude reprezentovaný maticí klíčových bodů a maticí deskriptorů. Proto před samotným odesláním snímku je nutné provést jeho analýzu. Opět lze využít knihovnu OpenCV, která je multiplatformní. Oficiální verze knihovny pro platformu iOS obsahuje všechny funkce a moduly. Navíc poskytuje několik metod usnadňující použití na platformě iOS. Užitečný je například konvertor obrázků reprezentovaných třídou UIImage[25] na číselné matice využívané pro výpočty v knihovně OpenCV.

Původním programovacím jazykem platformy iOS je Objective-C, který podporuje přímé spuštění kódu napsaného v jazyce C/C++. Nicméně od roku 2014 je pro vývoj iOS aplikací využíván nový jazyk Swift. Prostřednictvím tzv. `bridging header` souboru je možné volat funkce a metody implementované v Objective-C. Pro využití knihovny OpenCV bude nutné implementovat pomocnou třídu v jazyce Objective-C, která poslouží jako mezivrstva mezi jazyky Swift a C++. [32]

[24] User Interface – uživatelské rozhraní dané aplikace nebo systému.
[25] Standardní třída pro reprezentaci obrázků na platformě iOS.

2.5.3 Komunikace se serverem

Navržená komponenta tedy zobrazuje data z fotoaparátu získaná prostřednictvím frameworku ARKit. Opakovaně zachycuje aktuální snímek a provádí jeho analýzu prostřednictvím knihovny OpenCV. Získané matice klíčových bodů a deskriptorů jsou následně odeslány na server, kde dojde k vyhodnocení. V případě nalezení shody informuje komponenta mobilní aplikaci, která má možnost zareagovat vyvoláním příslušné akce. Samotnou výměnu dat mezi knihovnou a serverem je možné realizovat více způsoby:

- **BSD sockety** – Berkeley Software Distribution socket je nízkoúrovňové rozhraní pro přístup k síťové komunikaci v operačních systémech s unixovým jádrem, mezi které patří i systém iOS. Ostatní řešení využívají na pozadí právě BSD sockety. Výhodou je velká volnost a flexibilita při implementaci. Je však nutné naprogramovat kompletně celou logiku komunikace.

- **URLSession** – Nativní třída poskytující rozhraní pro implementaci síťové komunikace v iOS aplikacích. Podporuje všechny potřebné funkce a metody pro běžnou komunikaci prostřednictvím HTTP protokolu. Mezi vývojáři je terčem časté kritiky pro svou „upovídanost“, kdy často dochází k nutnosti psaní velkého množství opakujícího se kódu.

- **Alamofire** – Populární open-source knihovna pro platformu iOS, která rozšiřuje výše zmíněnou třídu URLSession o přehlednější a ucelenější rozhraní.

Knihovna Alamofire je považována za standard pro implementaci síťové komunikace v iOS aplikacích. Použitím by však došlo ke zbytečnému zanesení závislosti do distribuované knihovny. Pro využití knihovny by následně bylo nutné nainstalovat zároveň i knihovnu Alamofire. Z toho důvodu bude veškerá síťová komunikace implementovaná prostřednictvím URLSession.

2.6 Uživatelské rozhraní webové aplikace

Webová aplikace je vstupním bodem celého systému. Poskytuje grafické rozhraní pro registraci nového uživatelského účtu a následné přihlášení. V uživatelské sekci je k dispozici plnohodnotný editor určený ke správě databází obrázků. Uživatel může vytvářet nové databáze nebo upravit, či smazat stávající. V detailu databáze má potom možnost přidávat jednotlivé obrázky určené k vyhledávání a spravovat jim příslušná metadata.

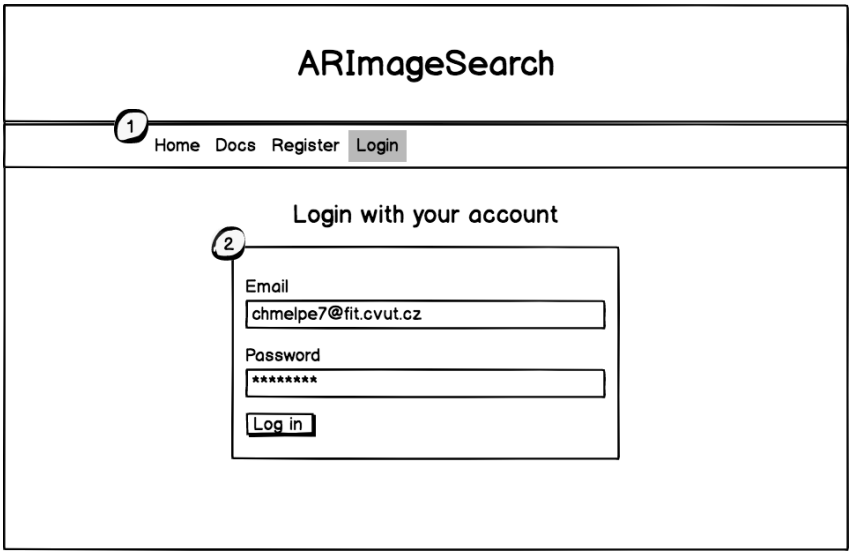
2.6.1 Navrhování formulářů

Téměř žádná aplikace s uživatelským rozhraním se neobejde bez formulářů pro zadávání dat. Často se jedná o klíčové komponenty, pomocí kterých uživatel komunikuje s danou aplikací. Formulář se obvykle skládá z jednotlivých komponent, jako jsou popisky, boxy pro zadávání textu, tlačítka a další. S počtem použitých komponent roste i celková komplexita formuláře. Často je možné narazit na nesprávně navržené formuláře, které jsou pro koncového uživatele nepřehledné nebo špatně použitelné. Při návrhu formulářů je proto vhodné dodržovat konvence a osvědčené postupy, které zajistí maximalizaci uživatelské přívětivosti aplikace. [33]

- Formuláře by měly být jednosloupcové. Více sloupců narušuje vertikální průchod formulářem, který je pro většinu uživatelů přirozený.
- Popisky je vhodné umístit nad příslušné komponenty. Studie dokazují, že takové formuláře uživatelé vyplňují rychleji než v případě umístění popisků vlevo od příslušných komponent.
- Popisky komponent by vždy měly být napsané přirozeným písmem, tedy s velkým počátečním písmenem. Popisky psané pouze malými nebo pouze velkými písmeny jsou hůře čitelné.
- Často se lze setkat s nahrazením popisků pomocí zástupného textu, který je umístěn uvnitř boxu pro zadávání textu. Jedná se o nevhodnou praktiku, která zhoršuje celkovou použitelnost formuláře. [34]
- Popisky tlačítek by měly být co nejvíce deskriptivní. Nevhodným popiskem je například „Odeslat“ pro tlačítko sloužící k přihlášení uživatele. Lepší volbou je popisek „Přihlásit“.
- Tlačítka pro vykonání primární akce formuláře (například odeslání) je vhodné barevně odlišit od ostatních tlačítek.
- V případě chyby formuláře by mělo dojít k jasnému označení komponenty, která chybu způsobuje (zpravidla červenou barvou).

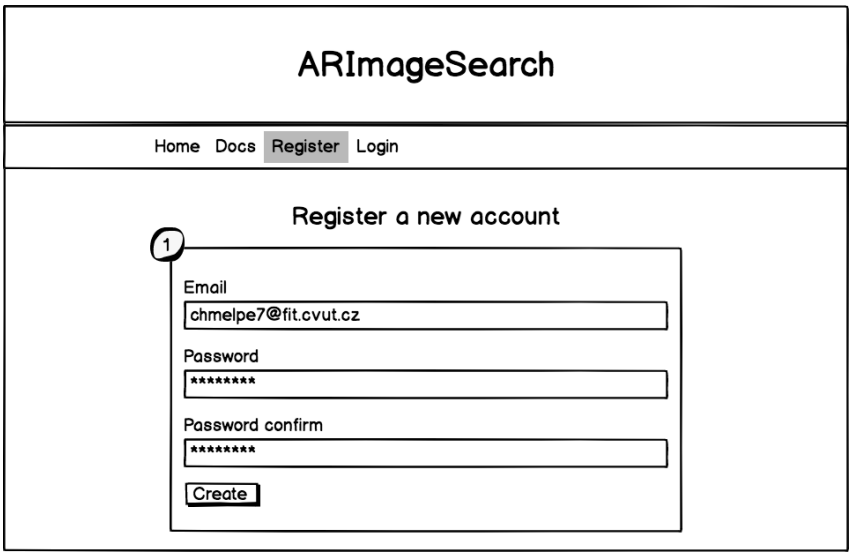
2.6.2 Wireframy webové aplikace

Před samotnou implementací aplikace s uživatelským rozhraním je vhodné nejprve dané rozhraní navrhnout prostřednictvím tzv. wireframů. Jedná se o „drátěný model“ zobrazující kostru aplikace a rozmístění jednotlivých komponent. Wireframe je většinou černobílý a neobsahuje žádné obrázky ani grafické prvky. Při návrhu wireframů často dojde k odhalení podstatných chyb v návrhu aplikace, které by jinak značně zvýšily finanční i časové náklady na implementaci aplikace.



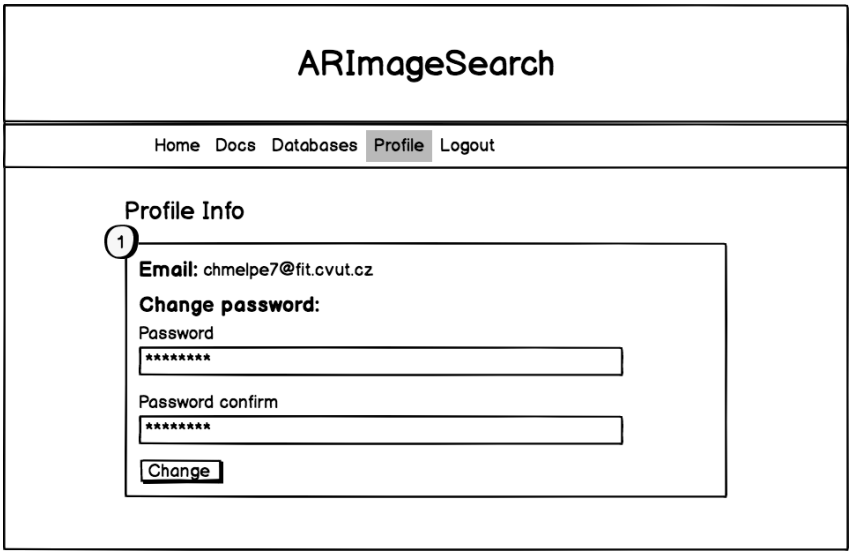
Příklad 2.3: Wireframe – Přihlášení uživatele

1. Hlavní menu aplikace, po přihlášení automatická změna možností.
2. Formulář pro přihlášení stávajícího uživatele.



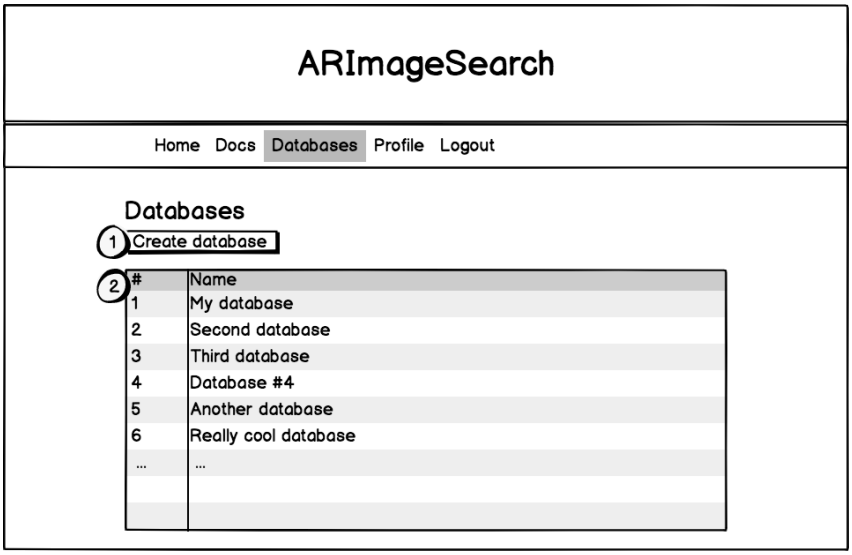
Příklad 2.4: Wireframe – Registrace uživatele

1. Formulář pro registraci nového uživatele



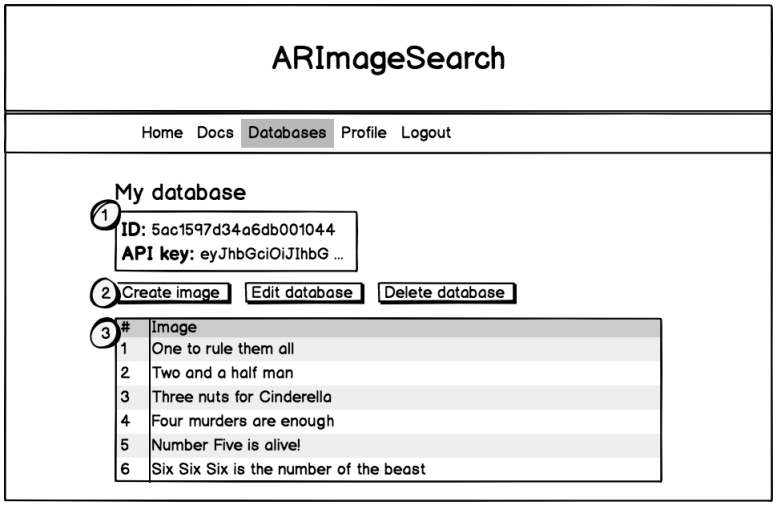
Příklad 2.5: Wireframe – Profil uživatele

1. Detail přihlášeného uživatele a formulář pro změnu hesla.



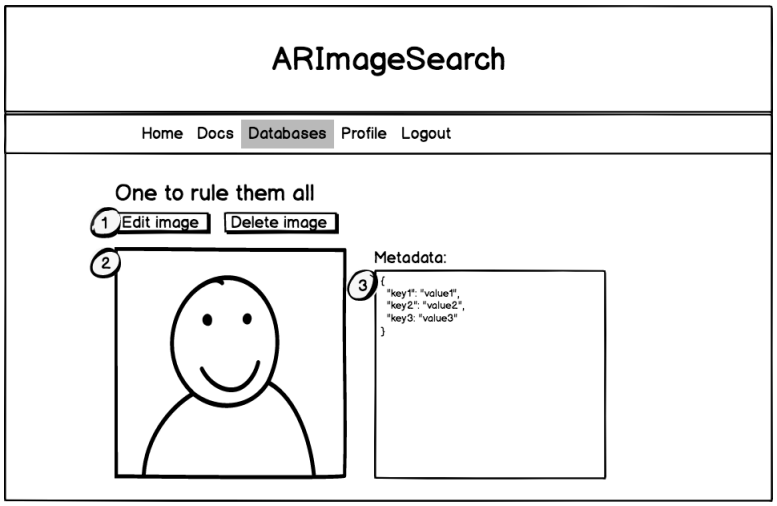
Příklad 2.6: Wireframe – Seznam databází

1. Tlačítko vyvolá modální formulář pro vytvoření nové databáze.
2. Seznam všech vytvořených databází, po kliknutí přechod na detail.



Příklad 2.7: Wireframe – Detail databáze

1. Identifikátor a API klíč databáze (pro použití v iOS knihovně).
2. Vyvolání formulářů pro přidání obrázku a editaci/smazání databáze.
3. Seznam všech přidaných obrázků, po kliknutí přechod na detail.



Příklad 2.8: Wireframe – Detail obrázku

1. Vyvolání formulářů pro editaci/smazání obrázku.
2. Zobrazení daného obrázku.
3. Metadata daného obrázku.

2.7 Možnosti implementace webové aplikace

Každou webovou aplikaci tvoří jednotlivé stránky identifikované unikátními URL adresami. Po zadání adresy do adresního řádku webového prohlížeče je kontaktován příslušný server. Dojde k vyhodnocení a webový prohlížeč obdrží požadovanou stránku. Samotné stránky jsou tvořeny kombinací jazyků HTML, CSS a JavaScript. Značkovací jazyk HTML, společně s kaskádovými styly CSS, udává webovému prohlížeči jak danou stránku vykreslit. JavaScript potom umožňuje obohatit stránku o dynamickou interakci (viz 2.1.2.4).

Zobrazená stránka zpravidla obsahuje odkazy na další stránky aplikace. Po zvolení odkazu dochází opět ke kontaktování serveru a načtení nové stránky. Jedná se o náročný proces, který způsobuje prodlevu ve vykreslování uživatelského rozhraní. Aktuálním trendem je vývoj webových aplikací jako tzv. SPA – Single Page Application, které předcházejí právě zbytečnému kontaktování serveru při přechodech mezi jednotlivými stránkami.

2.7.1 Single Page Application

Jako SPA označujeme webové aplikace, které pro přechody mezi jednotlivými stránkami dynamicky přepisují aktuální stránku. Pro načtení takové aplikace zpravidla stačí počáteční a také jediný požadavek, kdy dojde ke stažení veškerého statického obsahu. Všechny požadavky na zobrazení dalších stránek jsou následně vyhodnoceny využitím jazyka JavaScript na straně webového prohlížeče. Server potom slouží pouze jako zdroj dynamických dat, ke kterému webová aplikace přistupuje prostřednictvím aplikačního rozhraní (obvykle typu REST). [35]

Výhodou je minimalizování interupce při průchodu aplikací. Není nutné čekat na odpověď serveru a jednotlivé stránky jsou zobrazovány téměř instantně. Dochází tak k výraznému zlepšení uživatelské přívětivosti aplikace. Dalším pozitivem je úspora objemu přenášených dat, která může být důležitá při využívání aplikace na mobilních zařízeních.

Jsou zde ale i určitá omezení a negativa. Prvotní načítání může trvat déle, protože dochází ke stažení celé aplikace. Obsah SPA je obtížně zpracovatelný pro roboty vyhledávačů. Zdánlivým omezením může být i nefunkčnost aplikace bez JavaScriptu, který klade určité nároky na použité zařízení. Nicméně i nejlevnější mobilní telefony jsou dnes již dostatečně výkonné a se spuštěním SPA nemají problém. Obavy z nefunkčnosti aplikace bez JavaScriptu nejsou tedy v dnešní době na místě. Tvrzení potvrzuje i skutečnost, že poskytovatelé široce používaných webových služeb v Indii pro své aplikace vyžadují zapnutý JavaScript. [36]

2.7.2 Definice pojmů

- **MVC – Model-View-Controller**

  Webové aplikace často využívají návrhový vzor Model-View-Controller. Aplikace je rozdělena na tři samostatné části – datový model (Model), uživatelské rozhraní (View) a řídící logiku (Controller). Každá část je od ostatních dvou oddělena rozhraním přes které komunikuje.

- **MVVM – Model-View-ViewModel**

  Populární návrhový vzor který vychází z MVC. ViewModel zde slouží jako mezivrstva mezi View a Modelem. Funkce kontroleru je potom nahrazena tzv. databindingem[26] mezi ViewModelem a View.

2.7.3 SPA frameworky

Pro samotnou implementaci SPA je tedy nutné využít jazyk JavaScript. Existuje velké množství frameworků, které vývoj značně usnadňují. Zpravidla se liší ve své filosofii, syntaxi a samotném způsobu vykreslování stránek.

2.7.3.1 Angular

Jedním z prvních dostupných řešení pro tvorbu SPA byl framework Angular vydaný v roce 2009 společností Google, která jej od té doby dále aktivně rozvíjí pod open-source licencí. Podporuje návrhové vzory MVC a MVVM. Hlavním principem frameworku jsou tzv. direktivy usnadňující propojení HTML s JavaScriptem. Další známou funkcionalitou je dvoucestný databinding. Jedná se o obousměrnou synchronizaci dat mezi View a Modelem, která ale při nevhodném použití může způsobit zacyklení. V roce 2014 byla vydaná zpětně nekompatibilní verze Angular 2, která již defaultně podporuje pouze jednocestný databinding (dvoucestný je ponechán jako volitelná možnost).[37]

Angular je plnohodnotný robustní framework. Obsahuje mnoho tříd a služeb usnadňujících implementaci webových aplikací. Například službu `$http` pro volání HTTP požadavků, prostřednictvím které lze získavat data z aplikačního rozhraní serveru. Angular je jednou ze 4 komponent vývojařského balíčku MEAN (MongoDB, Express, Angular, Node), který slouží jako osvědčený základ pro vývoj systémů skládajících se ze serverové a webové aplikace implementované v jazyce JavaScript.

[26] Technika pro automatické předávání dat mezi objekty odlišného typu.

2.7.3.2 React

Původně interní projekt společnosti Facebook, který byl v roce 2013 uvolněn jako open-source knihovna. Nejedná se o plnohodnotný framework pro vývoj SPA, ale pouze o knihovnu, která se stará o efektivní vykreslování jednotlivých stránek dané webové aplikace. Z pohledu návrhového vzoru MVC tedy poskytuje řešení jenom pro část View, řešení pro Model a Controller je nutné naprogramovat nebo pro ně využít jiných knihoven. Kolem knihovny React se rychle vytvořila aktivní vývojářská komunita, která dala vzniknout mnoha rozšířením. Mezi nejznámnější patří knihovny Flux a Redux zajišťující jednosměrný datový tok. Společně s nimi tvoří React vyspělý a moderní ekosystém pro vývoj SPA.

React je založen na principu skládání jednotlivých stránek z relativně malých a samostatných komponent. Každá z komponent může mít svůj stav a při jeho změně dochází k překreslení komponenty včetně jejich potomků. Hovoříme tedy o manipulaci s DOM[27] dané HTML stránky. DOM komplexní webové stránky se může skládat z tisíců uzlů a jakékoli změny jsou pak výpočetně náročné. Proto React představuje koncept tzv. virtuálního DOMu. Jedná se o virtuální verzi skutečného DOMu, která je uložena v paměti. Požadovaná změna DOMu je nejprve vyhodnocena nad virtuální verzí. Dojde k efektivnímu nalezení minimálního počtu nutných operací, které jsou následně vykonány nad skutečnou verzí. [38]

2.7.3.3 Ember

Open-source framework pro implementaci SPA vyvíjený komunitou od roku 2011. Jeho základem je návrhový vzor MVVM a šablonovací systém postavený na knihovně Handlebars, který umožňuje snadné propojení HTML s JavaScriptem. Framework je vysoce modulární a existují tisíce rozšíření spravovaných komunitou. Rozšíření usnadňují implementaci většiny podstatných částí webových aplikací, jako je například volání HTTP požadavků, správa uživatelské relace, ukládání dat a další. Aktuální verze Ember 2 byla vydaná v roce 2015 a převzala některé osvědčené principy z knihovny React. Došlo k zajištění jednosměrného datového toku a představení vlastního vykreslovacího řešení Glimmer, který využívá princip virtuálního DOMu. [39]

2.7.4 Výběr řešení

Jakýkoli z představených frameworků je možné použít pro implementaci navržené webové aplikace. Zvolena byla knihovna React. Dle dostupných průzkumů je mezi vývojáři nejoblíbenějším řešením právě knihovna React, která tvoří základ moderního ekosystému pro vývoj webových aplikací. [40]

[27] Document Object Model – Reprezentace HTML dokumentu objektovým modelem.

2.8 Ověření uživatele a bezpečnost systému

Je důležité zajistit, aby přístup k modifikaci dané databáze obrázků měl vždy pouze oprávněný uživatel, zpravidla ten který databázi vytvořil. Přístupovým bodem ke všem zdrojům je aplikační rozhraní serverové aplikace. Před samotným obsloužením přijatého HTTP požadavku je tedy nutné ověřit identitu uživatele a jeho přístupová práva k danému zdroji.

2.8.1 Definice pojmů

- **Autentizace**

  Proces ověření identity uživatele. Z pohledu serverových aplikací se zpravidla jedná o zkontrolování obdržených přístupových údajů. Ty mohou být reprezentované kombinací jména a hesla nebo přístupovým tokenem. Obvykle jsou odesílané jako součást HTTP požadavku v hlavičce `Authorization`. V případě chyby serverová aplikace odmítne uživatele autorizovat a aplikační rozhraní odešle stavový kód `401 Unauthorized`.

- **Autorizace**

  Ověřuje zda daný uživatel má patřičná práva pro vykonání požadované akce. V serverové aplikaci obvykle následuje po procesu autentizace a v případě kladného vyhodnocení dochází k provedení akce. Pokud uživatel nemá potřebná práva, serverová aplikace odmítne požadavek obsloužit a aplikační rozhraní vrací stavový kód `403 Forbidden`. [41]

2.8.2 Registrace a přihlášení uživatele

2.8.2.1 Proces registrace

Aplikační rozhraní serverové aplikace obvykle poskytuje veřejně přístupné body pro registraci a přihlášení uživatelů. Základem registrace nového uživatelského účtu je emailová adresa a heslo. Požadavek je odeslán prostřednictvím zabezpečeného protokolu HTTPS. Ten rozšiřuje protokol HTTP o protokol TLS[28], který šifruje přenášená data a zamezuje tak odposlechnutí komunikace třetí stranou.

2.8.2.2 Bezpečné uložení hesla

Serverová aplikace následně zapíše nový záznam do databáze. Před samotným zapsáním je ale nutné patřičně zabezpečit přijaté heslo. Pokud by hesla byla uložena pouze v textovém formátu, jsou uživatelé systému vystaveni riziku zneužití jejich účtů při případném úniku dat z databáze.

[28] Transport Layer Security – Protokol využívaný pro šifrování dat.

Do databáze je proto ukládán pouze tzv. otisk hesla, ze kterého není možné získat původní heslo využitím běžně dostupných prostředků. Pro vytvoření otisku hesla se využívají tzv. hashovací funkce.

- **MD5** – Funkci, která vytváří otisk o velikosti 128 bitů, navrhl v roce 1991 Ronald Rivest. V průběhu let bylo odhaleno několik zásádních bezpečnostních chyb a použití algoritmu se dnes zásadně nedoporučuje.

- **SHA** – Skupina hashovacích funkcí původně inspirovaná algoritmem MD5. První verze SHA-0 a SHA-1 byly již prolomeny a nejsou tedy použitelné pro bezpečné ukládání hesel. Verze SHA-2 navržená v roce 2001 zatím nebyla prolomena a je široce využívaná. Vychází ale ze stejného základu jako předchozí verze a existují tedy obavy ohledně jejího prolomení. [42]

- **bcrypt** – Hashovací funkce založená na šifře Blowfish byla poprvé představena v roce 1999 Nielsem Provosem a Davidem Mazièresem. Umožňuje záměrné zvýšení náročnosti výpočtu otisku. Tím dochází ke zpomalení průběhu funkce a tedy i zajištění ochrany proti útokům hrubou silou. [43] Pro ukládání hesel byl zvolen právě bcrypt, jelikož se jedná o nejbezpečnější z běžně dostupných a dlouhodobě prověřených algoritmů.

2.8.2.3 Ověření emailové adresy

Je vhodné ověřit zda emailová adresa zadaná při registraci skutečně existuje a uživatel k ní má přístup. Po úspěšné registraci uživatele je tedy běžnou praxí odeslání emailu s verifikačním odkazem na zadanou adresu. Uživatel se následně přihlásí do své emailové schránky, kde nalezne daný email. Otevřením odkazu dojde k odeslání HTTP požadavku na aplikační rozhraní serverové aplikace, která do databáze zaznamená informaci o tom, že uživatel potvrdil svou emailovou adresu. Tento krok obvykle aktivuje možnost přihlášení, která je do té doby nedostupná.

2.8.2.4 Proces přihlášení

Uživatel, který potvrdil svou emailovou adresu, se následně může přihlásit. Pro přihlášení využívá kombinaci emailové adresy a hesla, kterou zadal při registraci svého účtu. Požadavek je odeslán serverové aplikaci, která z přijatého hesla vytvoří otisk a porovná jej s otiskem hesla uloženým v databázi. Pokud se shodují, uživatel zadal správné přihlašovací údaje a dochází k zahájení uživatelské relace. Server v odpovědi poskytne přístupové údaje určené pro autentizaci všech následujících požadavků dle stanoveného způsobu.

2.8.3 Způsoby autentizace uživatele

Přihlášený uživatel může prostřednictvím aplikačního rozhraní odesílat podepsané požadavky na jednotlivé zabezpečené zdroje. K podpisu požadavků obvykle využívá hlavičku `Authorization` v HTTP požadavcích. Serverová aplikace uživatele vždy nejprve autentizuje a autorizuje. K vykonání požadované akce dojde pouze pokud uživatel má skutečně přístup k danému zdroji.

2.8.3.1 Basic access authentication

Nejjednodušší způsob autentizace. V hlavičce `Authorization` je odesílána kombinace emailové adresy a hesla, která je oddělena dvojtečkou a zakódována do formátu `Base64`[29]. Na serveru jsou potom údaje ověřeny oproti databázi, stejně jako v případě přihlášení. Nevýhodou je nutnost uchování přihlašovacích údajů na straně klientské aplikace, kde je problematické zajistit bezpečné uložení hesla. Dalším negativem je fakt, že serverová aplikace nemá žádnou kontrolu nad uživatelskou relací.

2.8.3.2 Cookie based authentication

Cookie je malý objem dat, který server odešle klientské aplikaci a ta jej následně odesílá nazpět. Po úspěšném přihlášení server vytvoří uživatelskou relaci a odešle nazpět cookie, která obsahuje identifikátor dané relace. Cookie je následně odesílána jako součást všech následujících HTTP požadavků v hlavičce `Cookie`. Serverová aplikace udržuje v databázi seznam aktivních relací a na základě přijaté cookie vyhodnotí, zda má uživatel oprávnění k vykonání požadované akce. Nevýhodou je nutnost komunikace s databází a obtížnější škálování serverové aplikace, kdy je nutné zajistit distribuci relací mezi jednotlivými servery. [44]

2.8.3.3 JWT – JSON Web Token

Otevřený standard pro vytváření přístupových tokenů založený na formátu JSON. Uživatel při úspěšném přihlášení obdrží unikátní přístupový token, ve kterém je zakódována identita uživatele. Klientská aplikace si token uloží (obvykle do Local Storage webového prohlížeče) a následně jej odesílá v hlavičce `Authorization` všech následujících HTTP požadavků. Serverová aplikace na základě přijatého tokenu autentizuje a autorizuje uživatele. Jedná se o bezestavový způsob autentizace. Server neudržuje žádné relace, veškerá data potřebná k autentizaci jsou zakódovaná v přístupovém tokenu. To přináší zrychlení procesu i značné výhody při škálování serverové aplikace a proto byla zvolena právě autentizace prostřednictvím JWT. [44]

[29] Formát pro zakódování binárních dat do posloupnosti znaků.

Kapitola 3

Implementace

3.1 Serverová aplikace

Hlavní komponentou celého systému je serverová aplikace, která je implementovaná v jazyce JavaScript a spuštěna v běhovém prostředí Node.js. Základ navržené aplikace se skládá z několika standardních celků. Jedná se například o poskytování aplikačního rozhraní, komunikaci s databází, ověření uživatelů a další. Uvedené funkce jsou typické pro většinu serverových aplikací. Pro jejich řešení obvykle existují volně dostupné frameworky nebo knihovny, které jsou spravované komunitou vývojářů. Využívání knihoven se stalo standardem zejména kvůli zkrácení času potřebného pro implementaci aplikace.

3.1.1 Balíčkovací systém npm

Pro většinu běžně používaných programovacích jazyků existuje balíčkovací systém. Jedná se o program pro správu knihoven daného jazyka, který vývojářům usnadňuje jejich instalaci a aktualizaci. Často pracuje s konceptem tzv. konfiguračního souboru, ve kterém vývojář definuje knihovny použité v dané aplikaci.

Vývojáři prostředí Node.js představili v roce 2010 systém npm určený původně pro správu knihoven serverových aplikací. V průběhu let se npm stal hlavním balíčkovacím systémem pro jazyk JavaScript a dnes je široce využíván pro všechny typy aplikací. Konfigurace pro danou aplikaci je uložena v souboru `package.json`, jednotlivé knihovny se potom nacházejí v adresáři `node_modules`. Samotný systém npm je implementován rovněž v jazyce JavaScript a je instalován společně s Node.js. Počet balíčků dostupných v systému npm již několikanásobně přesáhl počet balíčků v systémech jiných programovacích jazyků. Jazyk JavaScript je tedy evidentně aktuálním trendem ve vývoji serverových a webových aplikací. [45]

```
{
  "name": "ar-image-search-server",
  "version": "1.0.0",
  ...
  "dependencies": {
    "aws-sdk": "^2.210.0",
    "bcrypt": "^1.0.3",
    "body-parser": "^1.18.2",
    "dotenv": "^5.0.1",
    "express": "^4.16.2",
    "jsonwebtoken": "^8.1.1",
    "mongoose": "^5.0.2",
    "multer": "^1.3.0",
    "nodemailer": "^4.6.3",
    "opencv4nodejs": "^4.2.1"
  }
}
```

Příklad 3.1: Serverová aplikace – Konfigurační soubor `package.json`

- **aws-sdk** – Oficiální knihovna pro komunikaci s AWS S3.
- **bcrypt** – Implementace stejnojmenné hashovací funkce.
- **body-parser** – Rozšíření pro framework Express určené pro zpracování těla HTTP požadavků.
- **dotenv** – Modul pro načtení konfiguračních proměnných aplikace.
- **express** – Populární minimalistický framework pro implementaci serverových aplikací a aplikačních rozhraní.
- **jsonwebtoken** – Implementace standardu JSON Web Token.
- **mongoose** – Knihovna pro komunikaci s databází MongoDB.
- **multer** – Rozšíření pro framework Express určené pro zpracování formátu `multipart/form-data`.
- **nodemailer** – Knihovna umožňující odesílání emailů.
- **opencv4nodejs** – Mezivrstva mezi Node.js a C++ knihovnou OpenCV (viz 2.4.2.2).

Veškeré příklady v kapitolách Implementace a Testování jsou pouze ilustrační. Reflektují hlavní podstatu problému, ale skutečná implementace je často rozsáhlejší. Kompletní zdrojové soubory jsou k dispozici na přiloženém DVD.

3.1.2 Komunikace s databází

Vyhodnocení většiny přijatých požadavků vyžaduje komunikaci se zvolenou dokumentovou databází MongoDB. Pro implementaci byla využita knihovna mongoose, která automatizuje propojení s databází a poskytuje jednoduché rozhraní pro samotnou komunikaci.

3.1.2.1 Definice modelů

Knihovna pracuje s konceptem tzv. modelů, které představují jednotlivé typy dokumentů v databázi. Příklad níže ukazuje definici modelu pro dokumenty typu User. Nejprve je vytvořeno schéma popisující atributy daného dokumentu (atribut `id` je přidáván automaticky). Ze schématu je následně vytvořen model, který je exportován pro další využití v aplikaci.

```
let mongoose = require('mongoose');
let Schema = mongoose.Schema;

let userSchema = new Schema({
  email: String,
  emailConfirmed: Boolean,
  pass: String,
});

let User = mongoose.model('user', userSchema, 'user');
module.exports = User;
```

Příklad 3.2: Serverová aplikace – Definice databázového modelu

3.1.2.2 Operace nad modely

Kdekoliv v aplikaci je následně možné importovat definovaný model a využít jej pro komunikaci s databází. Knihovna podporuje všechny běžné typy operací jako je čtení dokumentů podle jejich typu nebo `id`, zápis nového dokumentu, mazání dokumentů a další. Veškeré operace jsou asynchronní a neblokují tedy hlavní smyčku událostí v prostředí Node.js. Na uvedeném příkladu je ukázka komunikace. Nejprve je importován definovaný model a následuje operace čtení dokumentu s `id` odpovídající proměnné `userId`.

```
let User = require('../models/user');
User.findById(userId, function(err, user) {
    ...
});
```

Příklad 3.3: Serverová aplikace – Komunikace s databází

3.1.3 Aplikační rozhraní

Struktura aplikace do značné míry kopíruje strukturu aplikačního rozhraní. Použitý framework Express zjednodušuje a zpřehledňuje samotnou implementaci. Neexistuje žádný oficiální standard pro organizaci aplikace, je ale vhodné dodržovat obecné principy. Aplikace je tak členěna do rozumně velkých souborů, které vždy představují určitý celek. [46]

```
app.js ..... Hlavní soubor aplikace
routes ..... Jednotlivé části aplikačního rozhraní
controllers .... Kontrolery obsahující funkce pro zpracování požadavků
models ..... Definice databázových modelů
utilities ..... Podpůrné služby
package.json ..... Konfigurační soubor pro npm
node_modules ..... Knihovny a frameworky spravované npm
.env ..... Lokální konfigurační proměnné aplikace
```

Příklad 3.4: Serverová aplikace – Adresářová struktura

3.1.3.1 Hlavní soubor

Každá Node.js aplikace má svůj hlavní soubor (obvykle pojmenovaný `app.js` nebo `main.js`), ve kterém se definuje počáteční konfigurace pro spuštění aplikace. Při použití frameworku Express zde dochází k jeho inicializaci a nastavení. V hlavním souboru také často dochází k členění aplikačního rozhraní dle první části URL adresy přijatého požadavku na tzv. routery, kde dochází k dalšímu zpracování. Aplikační rozhraní typu REST je následně spuštěno na definovaném síťovém portu.

```
let express = require('express');
let app = express();

let bodyParser = require('body-parser');
app.use(bodyParser.json({ limit: '1mb' }));
app.use(bodyParser.urlencoded({ extended: true }));

app.use('/login', require('./routes/loginRouter'));
app.use('/user', require('./routes/userRouter'));
app.use('/database', require('./routes/databaseRouter'));

app.listen(process.env.PORT);
```

Příklad 3.5: Serverová aplikace – Konfigurace frameworku Express

3.1.3.2 Router

Implementace aplikačního rozhraní je rozdělena do více samostatných routerů. Jedná se o směrovače, které zajišťují obsloužení požadavků na základě jejich typu a URL adresy. Pro každý požadavek je definována sekvence funkcí. Příklad zobrazuje router `userRouter` obsluhující požadavky s URL adresou začínající řetězcem `/user`. Například pro požadavek GET odeslaný na URL adresu `/user/` se postupně vykonají funkce `auth.checkToken`, `user.get` a `user.update`. V terminologii frameworku Express jsou jednotlivé funkce označované jako tzv. middleware.

```
module.exports = (function() {
  let router = require('express').Router();
  let auth = require('../utilities/auth');
  let user = require('../controllers/userController');

  router.post('/', user.create);
  router.post('/verification', auth.checkToken, user.verify);
  router.get('/', auth.checkToken, user.get, user.view);
  router.put('/', auth.checkToken, user.get, user.update);
  return router;
})();
```

Příklad 3.6: Serverová aplikace – Implementace routeru

3.1.3.3 Middleware

Jako middleware je ve frameworku Express označována funkce, která má přístup k přijatému požadavku – `req`, příslušné budoucí odpovědi – `res` a následujícímu middlewaru – `next`. [47] Jednotlivé middlewary mohou být řetězeny za sebe. Vytvářejí tak sekvenci operací, která se vykoná pro daný požadavek. Na příkladu níže je implementace middlewaru `user.get` pro získání uživatele z databáze.

```
exports.get = function(req, res, next) {
  User.findById(req.token.userId, function(err, user) {
    if (user === null) {
      res.statusCode = 404;
      res.send('Error 404: User not found');
    } else {
      res.locals.user = user;
      next();
    }
  });
};
```

Příklad 3.7: Serverová aplikace – Implementace middlewaru

3.1.4 Ověření uživatele

3.1.4.1 Registrace nového uživatele

Po přijetí požadavku na registraci nového uživatelského účtu je spuštěn middleware `user.create`, který zajišťuje zápis do databáze a odeslání verifikačního emailu. Knihovnou bcrypt je nejprve vytvořen otisk přijatého uživatelského hesla. Poté je inicializován nový dokument typu User, který obsahuje přijatou emailovou adresu, příznak jejího (ne)ověření a vytvořený otisk. Dokument je zapsán do databáze a v případě korektního zápisu je odeslána odpověď `201 Created`.

Dále je vytvořena emailová zpráva obsahující odkaz s verifikačním tokenem, který slouží pro ověření emailové adresy. Token je generován knihovnou jsonwebtoken a má platnost 48 hodin. V tokenu je zakódována emailová adresa, která slouží pro pozdější ověření. Emailová zpráva je odeslána na danou adresu prostřednictvím knihovny nodemailer.

```
exports.create = function(req, res) {
  bcrypt.hash(req.body.pass, 10, function(err, hash) {

    let newUser = User({
      email: req.body.email, emailConfirmed: false, pass: hash
    });
    newUser.save(function(err) {
      res.statusCode = 201;
      res.location('/user');
      res.json(newUser);
    });

    let token = jwt.sign({ email: newUser.email },
      (process.env.JWTKEY), { expiresIn: '48h' });
    let emailData = { ... };
    transporter.sendMail(emailData, function(err, info) { ... });
  });
};
```

Příklad 3.8: Serverová aplikace – Registrace uživatele

3.1.4.2 Verifikace emailové adresy

Otevřením odkazu v přijatém emailu dojde k odeslání požadavku, který obsahuje verifikační token. Zpracování zajišťuje middleware `user.verify`. Na základě přijatého tokenu identifikuje o jakého uživatele se jedná a zaznamená do databáze změnu příznaku `emailConfirmed` na hodnotu `true`. Uživatel je následně považován za verifikovaného a může se přihlásit do systému.

3.1.4.3 Přihlášení uživatele

Požadavek pro přihlášení uživatele obsluhuje samostatný router `loginRouter`. Uživatelský účet je nejprve nalezen v databázi na základě zadané emailové adresy. Pokud se jedná o účet s dosud neověřenou emailovou adresou, je odeslána odpověď `401 Unauthorized` společně s příslušnou chybovou hláškou. V opačném případě je vypočítán otisk přijatého hesla a následně je porovnán s otiskem uloženým v databázi. Pokud se shodují, je vytvořen nový přístupový token, který je odeslán jako součást odpovědi `200 OK`.

```
User.findOne({ email: req.body.email }, function(err, user) {
  if (!user.emailConfirmed) {
    res.statusCode = 401;
    res.send('Error 401: Email not yet verified');
  } else {
    bcrypt.compare(req.body.pass, user.pass, function(err, bRes){
      if (bRes == true) {
        let token = jwt.sign({ userId: user.id },
          (process.env.JWTKEY), { expiresIn: '7d' });
        res.json({ 'id': user.id,
          'email': user.email, 'token': token });
      } else {
        res.statusCode = 401;
        res.send('Error 401: Wrong email or password');
      }
    });
  }
});
```

Příklad 3.9: Serverová aplikace – Přihlášení uživatele

3.1.4.4 Ověření přístupového tokenu

Přístupový token, získaný jako odpověď na přihlášení, je následně využíván pro ověření všech následujících požadavků. Middleware `checkToken` nejprve získá token z hlavičky `Authorization`. Proběhne jeho dekódování a získáné informace jsou předané dalšímu middlewaru v pořadí.

```
exports.checkToken = function(req, res, next) {
  let token = req.headers.authorization.split(' ')[1];
  jwt.verify(token, (process.env.JWTKEY), function(err, data) {
    req.token = data;
    next();
  });
}
```

Příklad 3.10: Serverová aplikace – Ověření tokenu

3.1.5 Testování a dokumentace aplikačního rozhraní

V průběhu implementace každé aplikace je nutné pravidelně testovat požadovanou funkcionalitu. Vývojář tak získává zpětnou vazbu a může rychle opravovat případné chyby. Pro účely testování aplikačních rozhraní existuje několik specializovaných programů.

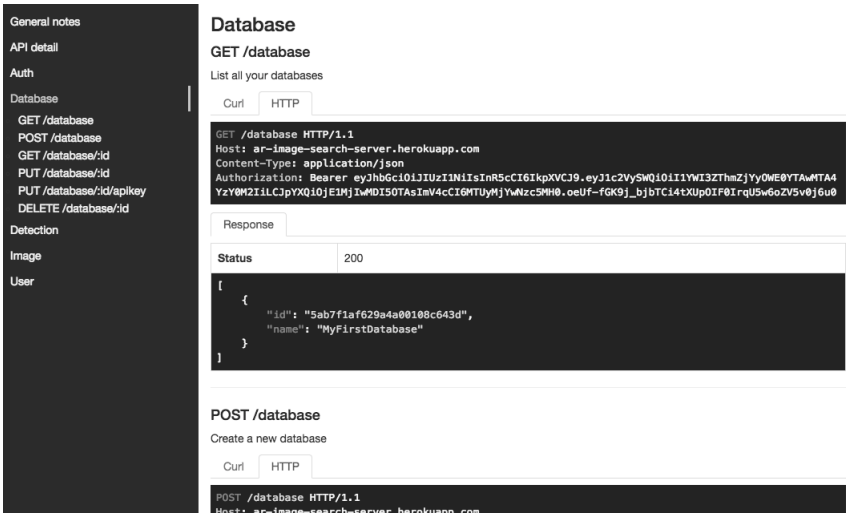
3.1.5.1 Postman

Postman je multiplatformní komplexní nástroj pro prototypování, testování a ladění REST API. Základní verze programu je k dispozici zdarma, pokročilé funkce jsou zpoplatněné. Při implementaci aplikačního rozhraní byl pro účely průběžného testování využíván právě Postman. Umožňuje definování vlastních kolekcí, do kterých je možné ukládat jednotlivé požadavky pro rychlé znovuopakování testů. Kolekce je následně možné exportovat do formátu JSON a sdílet s ostatními vývojáři.

3.1.5.2 Postmanerator

Součástí implementace aplikačního rozhraní by měla být i dokumentace popisující jednotlivé požadavky. Manuální vytvoření je však časově náročné a proto vzniklo několik způsobů, které daný proces automatizují.

Při využití nástroje Postman je možné vygenerovat dokumentaci aplikačního rozhraní na základě vytvořených kolekcí. K tomu slouží open-source skript Postmanerator, který vytvořil Aurélien Baumann. Vstupem skriptu je exportovaný JSON soubor reprezentující kolekci. Na jeho základě je vygenerován zdrojový kód webové stránky ve formátu HTML, která obsahuje jednoduchou a přehlednou dokumentaci všech požadavků. [48]



Příklad 3.11: Dokumentace aplikačního rozhraní

3.2 Analýza obrázků a vyhodnocení shod

Každý uživatel systému má možnost vytvoření databází obrázků. Do nich může nahrávat jednotlivé obrázky určené k pozdějšímu vyhledávání využitím knihovny pro platformu iOS. Pro zpracování některých požadavků je nutná komunikace serverové aplikace s knihovnou OpenCV.

3.2.1 Nový obrázek

Při nahrání nového obrázku do databáze obrázků je spuštěna sekvence middlewarů. Prvním je `checkToken`, který ověří a dekóduje přístupový token přijatého požadavku. Následuje `database.get` pro nalezení příslušné databáze obrázků. Samotný obrázek je odeslán ve formátu `multipart/form-data`, proto další middleware `upload.saveLocal` zajistí zpracování přijatých dat prostřednictvím knihovny multer. Ověří zda data obsahují obrázek ve formátu `.jpg` a uloží jej do dočasného úložiště pro další zpracování.

3.2.1.1 Nahrání obrázku do úložiště souborů

Obrázek je na serveru uložen pouze pro účely analýzy a následně je vymazán. K dlouhodobému uchování obrázků slouží cloudové úložiště souborů AWS S3. Middleware `upload.saveS3` provede nahrání daného obrázku do úložiště prostřednictvím oficiální knihovny aws-sdk. Do databáze je poté uložen pouze odkaz, pod kterým je obrázek uložen. Budoucí požadavky na zobrazení obrázku pak obsluhuje middleware `upload.loadS3` po ověření přístupového tokenu.

3.2.1.2 Analýza nového obrázku

Každý obrázek je reprezentován dokumentem obsahujícím matice klíčových bodů a deskriptorů (viz 2.2.1). Middleware `upload.createImage` tedy před přidáním obrázku do databáze nejprve provede jeho analýzu. Algoritmem ORB jsou nalezeny klíčové body a vypočteny deskriptory. Počet extrahovaných vlastností byl po otestování systému (viz 4.3.1) zpětně snížen na 150 (defaultní počet je 500).

```
let cv = require('opencv4nodejs');
exports.createImage = function(req, res) {
    ...
    let detector = new cv.ORBDetector({ nfeatures: 150 });
    let img = cv.imread('./' + req.file.path);
    let keyPoints = detector.detect(img);
    let descriptors = detector.compute(img, keyPoints);
    ...
};
```

Příklad 3.12: Analýza nového obrázku

3.2.2 Vyhodnocení shod

Hlavní funkcí systému je vyhodnocení shody přijatého snímku oproti databázi obrázků. Pro úsporu dat je analýza snímku prováděna na mobilním zařízení (viz 2.3.5). Úlohou serveru je potom tedy pouze porovnání přijatých klíčových bodů a deskriptorů s klíčovými body a deskriptory, které reprezentují jednotlivé obrázky dané databáze. Zpracování požadavku zajišťuje middleware `detection.create`.

3.2.2.1 Konstrukce matic

Pro efektivní proces porovnávání je nejprve vhodné zkonstruovat dvě velké matice. Jedna obsahuje klíčové body všech obrázků dané databáze, druhá potom jejich deskriptory. Zvolený algoritmus následně může porovnat přijatý snímek oproti celé databázi najednou. To přináší značné zrychlení ve srovnání s klasickým porovnáváním po jednom obrázku.

Konstrukce je provedena jednoduchým algoritmem, který k vytvořeným maticím postupně připojuje klíčové body a deskriptory jednotlivých obrázků. Pro pozdější vyhodnocení shod zapisuje ke každému obrázku jeho počáteční a koncovou pozici ve vytvořených maticích.

```
let allKp = [];
let allDesc = [];
let matrixIndex = 0;
for (var i = 0; i < images.length; i++) {
  images[i].startIndex = matrixIndex;
  matrixIndex += images[i].descriptors.length
  images[i].endIndex = matrixIndex;
  allKp.push.apply(allKp, images[i].keyPoints);
  allDesc.push.apply(allDesc, images[i].descriptors);
}
```

Příklad 3.13: Matice reprezentující databázi obrázků

3.2.2.2 Vyhledání nejbližších deskriptorů

Pro porovnání deskriptorů databáze s deskriptory přijatého snímku je používán Brute-Force Matcher s Hammingovou vzdáleností, který vrátí 2 nejbližší dvojice pro každý deskriptor z databáze obrázků. Je spuštěn asynchronně a neblokuje tedy zbytek serverové aplikace. Vzhledem k binární povaze ORB deskriptorů je Brute-Force Matcher poměrně rychlý a použití FLANN Based Matcheru nepřináší téměř žádné zrychlení. Získané dvojice nejbližších deskriptorů jsou následně dále zpracovány pro vyhodnocení shod.

Funkce knihovny OpenCV jsou z prostředí Node.js volané prostřednictvím knihovny opencv4nodejs, která v době implementace podporovala vyhledání pouze jedné nejbližší dvojice. Pro další zpřesnění výsledku je však nutné vyhledávat 2 nejbližší dvojice využitím algoritmu k nejbližších sousedů. Chybějící podporu pro volání příslušných funkcí knihovny OpenCV bylo nutné doimplementovat. Dále byl dán podnět k zahrnutí do knihovny opencv4nodejs prostřednictvím tzv. pull-requestu, který byl akceptován. [49]

3.2.2.3 Poměrový test a homografie

Nalezené dvojice nejbližších deskriptorů jsou nejprve podrobeny poměrovému testu, který vyřadí falešné dvojice. Pokud zůstane více jak 10 validních dvojic, jedná se o kandidáta na shodu. Konstanta 10 je doporučena přímo v dokumentaci knihovny OpenCV. [15]

Pro takové kandidáty je následně vypočtena homografie a ověřena její kvalita funkcí `niceHomography`. [50] Kandidát s kvalitní homografií je prohlášen za validní shodu. Z případných více validních shod je vybrána ta s nejvyšším počtem inliers a detail odpovídajícího obrázku je odeslán jako odpověď na požadavek. Opět bylo nutné upravit knihovnu opencv4nodejs pro získání masky homografie. Změny byly zahrnuty prostřednictvím pull-requestu. [51]

```
cv.matchKnnBruteForceHammingAsync(mat,des,2,function(e,matches) {
  let best = 0; let bestIndex = -1;
  let index = 0; let matched1 = []; let matched2 = [];

  for (var i = 0; i < matches.length; i++) {
    if (matches[i][0].distance < 0.75 * matches[i][1].distance) {
      matched1.push(mat.kp[matches[i][0].queryIdx]);
      matched2.push(kp[matches[i][0].trainIdx]);
    }
    if (mat.images[index].endIndex == i) {
      if (matched1.length > 10) {
        let {hom, mask} = cv.findHomography(matched1, matched2);
        if (niceHomography(hom) && countInliers(mask) >= best) {
          best = countInliers(mask); bestIndex = index;
        }
      }
      index += 1; matched1 = []; matched2 = [];
    }
  }
  if (bestIndex > -1) {
    res.statusCode = 201;
    res.json({ mat.images[bestIndex].toJSON() });
  }
});
```

Příklad 3.14: Vyhodnocení shod s databází obrázků

3.3 Knihovna pro platformu iOS

Vývojáři využívají knihovnu pro usnadnění implementace vyhledávání obrázků v rozšířené realitě do svých aplikací. Důraz je proto kladen na snadnou instalaci a integraci. Knihovna je implementovaná v jazyce Swift a pro analýzu fotek jsou volány funkce z knihovny OpenCV.

3.3.1 Implementace

Funkcionalita knihovny je implementována ve třech vzájemně propojených třídách. Nejnižší vrstvou je `OpenCVWrapper` pro získání klíčových bodů a vypočtení jejich deskriptorů. Následuje třída `ARImageSearch` zajišťující komunikaci s aplikačním rozhraním serverové aplikace. Na vrcholku knihovny je potom třída `ARImageSearchView`, která zobrazuje data z fotoaparátu a informuje aplikaci o případném nalezení shody.

3.3.1.1 ARImageSearchView

Zobrazení dat z fotoaparátu daného zařízení je zajištěno rozšířením třídy `ARSCNView`, která je součástí ARKitu. Konstruktoru je nutné předat rámec, do kterého budou data vykreslována a kombinaci identifikačních údajů dané databáze obrázků. Dojde k vytvoření instance třídy `ARImageSearch` a spuštění relace třídy `ARSCNView`, která započne s vykreslováním dat.

Třída dále obsahuje veřejné funkce `startCapturing` a `stopCapturing`, kterými mobilní aplikace zapne nebo vypne kontinuální zachycování snímků. Po zapnutí je zavolána privátní rekurzivní funkce `capturePhoto`. V každém volání je nejprve zachycen aktuální snímek, který je nasledně předán funkci `getMatch` třídy `ARImageSearch`. Při nalezení shody dochází k informování mobilní aplikace prostřednictvím delegáta `imageSearchDelegate`.

```
public class ARImageSearchView: ARSCNView {
  public init(frame: CGRect, databaseId: String, apiKey: String){
    ar = ARImageSearch(databaseId: databaseId, apiKey: apiKey)
    session.run(ARWorldTrackingConfiguration())
  }
  private func capturePhoto() {
    let image = snapshot()
    ar.getMatch(image: image, success: { [weak self] (n, m) in
      self?.imageSearchDelegate?.matchFound(name: n, metadata: m)
      self?.capturePhoto()
    }, failure: { [weak self] (error) in
      self?.capturePhoto()
    })
  }
}
```

Příklad 3.15: Knihovna pro iOS – Zachycování snímků

3.3.1.2 ARImageSearch

Před odesláním snímku na server je nejprve provedena jeho analýza zavoláním funkce `getKeyPointsAndDescriptors` třídy `OpenCVWrapper`. Získané klíčové body a deskriptory jsou následně odeslány společně s přístupovým tokenem, jako HTTP požadavek na URL adresu dané databáze obrázků prostřednictvím funkcí nativní třídy URLSession. Celý požadavek je před odesláním ještě zkomprimován do formátu gzip pro úsporu objemu přenášených dat.

```
public func getMatch(image: UIImage, success(), failure()) {
  let kp, des = openCVWrapper.getKeyPointsAndDescriptors(image)
  let url = API_URL + "/database/\(databaseId)/detection"
  let request = NSMutableURLRequest(url: URL(string: url)!)
  request.httpMethod = "POST"
  request.addValue("Authorization", "Bearer \(apiKey)")
  let body = json(["keyPoints": kp, "descriptors": des])
  request.httpBody = try! body.gzipped()
  session.dataTask(with: request) {
    if (data) success()
    else failure()
  }
}
```

Příklad 3.16: Knihovna pro iOS – Komunikace s aplikačním rozhraním

3.3.1.3 OpenCVWrapper

Pro korektní vyhodnocení shod je nutné snímek analyzovat stejným způsobem jako obrázky uložené v databázi. Je tedy aplikován stejný postup jako při analýze na serveru. Algoritmem ORB je nalezeno 150 klíčových bodů a následně vypočteny jejich deskriptory. Kvůli volání C++ funkcí knihovny OpenCV bylo nutné třídu implementovat v jazyce Objective-C. Volání z jazyka Swift je zajištěno přidáním třídy do `bridging headeru`.

```
- (void)getKeyPointsAndDescriptors:(UIImage *)image {
  cv::Mat matrix = UIImageToMat(image);
  cv::Ptr<cv::FeatureDetector> orb = cv::ORB::create(150);
  std::vector<cv::KeyPoint> cvKeyPoints;
  orb->detect(matrix, cvKeyPoints);
  cv::Mat cvDescriptors = cv::Mat();
  orb->compute(matrix, cvKeyPoints, cvDescriptors);
  NSMutableArray *kp = kpToArray(cvKeyPoints);
  NSMutableArray *des = desToArray(cvDescriptors);
  return(kp, des)
}
```

Příklad 3.17: Knihovna pro iOS – Analýza snímků

3.3.2 Distribuce knihovny

Defaultním vývojovým prostředím pro platformu iOS je volně dostupný program XCode, který mimo jiné umožňuje i archivaci aktuálního projektu jako knihovnu. Výsledný archiv je možné distribuovat nahráním do libovolného úložiště souborů a následným sdílením URL adresy. Pro usnadnění instalace a verzování je však vhodné knihovnu distribuovat využitím balíčkovacího systému, kterým je na platformě iOS systém CocoaPods.

Každá knihovna v systému CocoaPods obsahuje vlastní konfigurační soubor `podspec`. Ten specifikuje jméno, verzi a licenci knihovny, jméno autora, minimální platformu, URL adresu a případné další informace. Zavoláním příkazu `pod trunk push ARImageSearch.podspec` je potom možné publikovat novou knihovnu nebo aktualizovat stávající. [52]

```
Pod::Spec.new do |s|
  s.name = 'ARImageSearch'
  s.version = '2.2.0'
  s.summary = 'iOS framework for AR Image Search project.'
  s.license = { :type => 'Apache-2.0', :file => 'LICENSE' }
  s.author = { 'Petr Chmelar' => 'info@pchmelar.cz' }
  s.platform = :ios, '11.0'
  s.source = { :http => URL + '/ios-sdk-2.2.0.zip' }
  s.ios.vendored_frameworks = 'ARImageSearch.framework'
  s.pod_target_xcconfig = { 'SWIFT_VERSION' => '4' }
end
```

Příklad 3.18: Knihovna pro iOS – Konfigurační soubor `podspec`

3.3.3 Integrace do aplikace

Vývojář mobilní aplikace má možnost si stáhnout aktuální verzi knihovny prostřednictvím odkazu ve webové aplikaci nebo využitím systému CocoaPods. Pokud se rozhodne pro instalaci přes CocoaPods, přidá do svého konfiguračního souboru `Podfile` řádek `pod 'ARImageSearch'`. Poté zavolá příkaz `pod install`, který knihovnu nainstaluje a integruje do projektu.

Na příslušném místě v aplikaci vývojář nejprve importuje knihovnu. Následně má k dispozici třídu `ARImageSearchView`, delegáta informujícího o nalezené shodě a metody `startCapturing` a `stopCapturing`. Součástí knihovny je její kompletní dokumentace s příklady použití, která je k dispozici také prostřednictvím webové aplikace.

Typickým příkladem použití je zobrazení dat z fotoaparátu přes celý displej zařízení. Snímky jsou kontinuálně odesílány, dokud není nalezena shoda, při které dojde k pozastavení odesílání a vyvolání příslušné akce. Po uživatelské reakci na danou akci je odesílání opět obnoveno.

3.4 Webová aplikace

Správa jednotlivých databází obrázků je možná prostřednictvím webové aplikace. Ta je implementovaná jako SPA – Single Page Application v jazyce JavaScript. Pro usnadnění vývoje je využita knihovna React.

3.4.1 Vývoj moderních SPA

3.4.1.1 Babel a Webpack

Jazyk JavaScript je v posledních letech aktivně vyvíjený poměrně rychlým tempem. Motivací je zejména poskytnutí moderních programovacích technik a zjednodušení implementace komplexních SPA, pro které nebyl jazyk původně určen (viz 2.1.2.4). Vývojáři prohlížečů webových stránek ovšem nestíhají reagovat na takto rychlý vývoj jazyka, v případě některých starších prohlížečů je podpora dokonce nemožná. Při vývoji v nových verzích jazyka (ES2015 a vyšší) je tedy nutné zajistit překlad do starší verze (ES5), která je podporována většinou prohlížečů. [53]

Knihovna React dále rozšiřuje syntaxi jazyka JavaScript o možnost využívání XML tagů pro jednoduchou implementaci uživatelského rozhraní. Rozšíření se označuje jako JSX a není standardní součástí jazyka (dle specifikace to ani není záměrem). [54] Podobně jako při použití nových verzí jazyka je tedy nutné zajistit překlad do kódu, který bude kompatibilní s většinou prohlížečů webových stránek. Vývojáři knihovny React doporučují využití překladače Babel, který je obecně jedním z nejpoužívanějších.

Při vývoji větších webových aplikací je vhodné členit zdrojový kód do jednotlivých modulů, které je možné vzájemně importovat. Prohlížeč webových stránek ovšem neumí takový kód zpracovat. Pro účely nasazení je proto nutné efektivně sestavit celou aplikaci speciálním nástrojem jako je například Webpack. [55]

3.4.1.2 Nástroj create-react-app

Konfigurace překladače a nástroje pro sestavení aplikace může být často komplexní a časově náročná. Vývojářská komunita proto aktivně vyvíjí nástroje pro usnadnění celého procesu. Jedním z nich je i create-react-app, který vytvoří nový projekt pro vývoj webové aplikace, nainstaluje knihovnu React a nastaví potřebnou konfiguraci pro Babel a Webpack. Vývojář potom může rovnou přejít k implementaci samotné aplikace.

Většina SPA využívá pro správu knihoven balíčkovací systém npm a jinak tomu není ani při použití create-react-app. Automaticky je vytvořeno několik skriptů pro vývoj a nasazení aplikace. Příkaz `npm run start` spustí lokální webový server, který na adrese `http://localhost:3000` zobrazuje a automaticky obnovuje vyvíjenou aplikaci. Sestavení aplikace k nasazení je možné příkazem `npm run build`.

3.4.2 Struktura aplikace

Základní projekt vytvořený nástrojem create-react-app byl dále rozšířen o několik adresářů pro snadnější organizaci zdrojových kódu aplikace. Běžnou konvencí při využití knihovny React je členění aplikace do modulů, které představují jednotlivé stránky a komponenty. [56]

```
public
    index.html ..... Vstupní stránka aplikace
src
    index.js ..... Hlavní soubor aplikace
    pages ..... Jednotlivé stránky složené z komponent
    components ..... Komponenty uživatelského rozhraní
    services ..... Podpůrné služby
    assets ..... Statické soubory (například obrázky)
package.json ..... Konfigurační soubor pro npm
node_modules ..... Knihovny a frameworky spravované npm
.env.local ..... Lokální konfigurační proměnné aplikace
```

Příklad 3.19: Webová aplikace – Adresářová struktura

3.4.2.1 Hlavní soubor

Jedinou HTML stránkou celé aplikace je `index.html`, do které je dynamicky vkládán obsah dle aktuálního stavu aplikace. Soubor `index.js` zajišťuje spuštění aplikace a vykreslení příslušné stránky na základě aktuální URL adresy. K vyhodnocení slouží komponenty `Switch` a `Route` z knihovny react-router. Některé komponenty, jako například hlavička a menu, jsou společné pro všechny stránky a proto jsou nadřazeny samotnému procesu routování.

```
<div>
  <Header />
  <NavBar />
  <Switch>
    <Route exact path='/' component={HomePage} />
    <Route path='/register' component={RegisterPage} />
    <Route path='/login' component={LoginPage} />
    <Route path='/profile' component={ProfilePage} />
    <Route path='/databases' component={DatabasesMainPage} />
    <Route path='/verify' component={VerifyPage} />
    <Route path='/docs' component={DocsPage} />
  </Switch>
  <Footer />
</div>
```

Příklad 3.20: Webová aplikace – Vykreslení aktuální stránky

3.4.2.2 Stránky a komponenty

Každá stránka webové aplikace je v podstatě pouze větší komponentou, která je složena z několika menších komponent. Knihovna React poskytuje rozhraní pro jednoduché vytváření komponent a ovládání jejich stavu.

Komponenta má svůj stav, který představuje všechny její dynamické hodnoty a je uložen v objektu `this.state`. Počátační stav je nastaven v konstruktoru komponenty. Následující změny je nutné provádět zavoláním funkce `this.setState`, která zajistí příslušné překreslení komponenty včetně případných potomků. [57]

Na příkladu níže je ukázka implementace stránky zobrazující profil přihlášeného uživatele (viz příklad 2.5). V konstruktoru je nastaven počáteční stav, který obsahuje prázdný objekt `user`. Funkce `render` definuje samotné vykreslení stránky, která je složena z dalších komponent. Po prvotním vykreslení je zavolána funkce `componentDidMount`. Dochází k odeslání požadavku na server a následné změně stavu prostřednictvím funkce `this.setState`. Ta způsobí znovuzavolání funkce `render` a tím i patřičné překreslení stránky.

```
class ProfilePage extends Component {
  constructor(props) {
    super(props);
    this.state = {
      user: {
        id: '',
        email: ''
      }
    }
  }
  componentDidMount() {
    NetworkService.makeRequest(this.props, 'get', '/user')
    .then((res) => {
      this.setState({
        user: res.data
      });
    })
  }
  render() {
    return (
      <div>
        <PageHeader>Profile Info</PageHeader>
        <UserInfo user={this.state.user} />
        <ChangePasswordForm />
      </div>
    );
  }
}
```

Příklad 3.21: Webová aplikace – Implementace komponent

3.4.3 Podpůrné služby

Knihovna React není plnohodnotným frameworkem pro implementaci SPA. Zajišťuje pouze vykreslování stránek. Pro ostatní části webové aplikace je nutné naprogramovat vlastní řešení nebo využít jiných knihoven.

3.4.3.1 Komunikace se serverem

Odesílání HTTP požadavků je v jazyce JavaScript možné realizovat nativní třídou XMLHttpRequest. Její rozhraní je však zbytečně „upovídané“. Existuje mnoho open-source knihoven, které obalují právě třídu XMLHttpRequest a poskytují přehlednější rozhraní. Pro implementaci byla zvolena knihovna axios, která tvoří základ služby `NetworkService`. Kdekoli v aplikaci je následně možné zavolat funkci `makeRequest` s příslušnými parametry, která se postará o odeslání požadavku.

```
static makeRequest(method, endpoint, data = {}) {
  return axios({
    method: method,
    url: this.apiUrl + endpoint,
    headers: {'Authorization': AuthService.getUser().token},
    data: data
  }).then((res) => { return res; });
}
```

Příklad 3.22: Webová aplikace – Služba pro síťovou komunikaci

3.4.3.2 Zachování uživatelské relace

Po úspěšném přihlášení obdrží aplikace od serveru údaje o identitě uživatele včetně přístupového tokenu. Získaná data je nutné uložit pro zachování uživatelské relace. Služba `AuthService` poskytuje jednoduché rozhraní pro zápis a čtení uživatelské identity do Local Storage webového prohlížeče. Z jakéhokoli místa v aplikaci je poté možné získat údaje o aktuální relaci.

```
static authenticateUser(data) {
  localStorage.setItem('currentUser', JSON.stringify(data));
}
static deauthenticateUser() {
  localStorage.removeItem('currentUser');
}
static getUser() {
  return JSON.parse(localStorage.getItem('currentUser'));
}
```

Příklad 3.23: Webová aplikace – Správa uživatelské relace

3.5 Nasazení systému

V průběhu implementace je obvykle postačující lokální provoz aplikace na zařízení vývojáře. Pro otestování systému a následné spuštění je však nutné aplikaci nasadit a umožnit tak její dostupnost prostřednictvím internetu. Existuje velké množství poskytovatelů řešení pro hostování aplikací. Z důvodů jednoduché konfigurace a předchozí kladné zkušenosti byl pro nasazení serverové i webové aplikace zvolen poskytovatel Heroku. Databáze MongoDB je hostována u poskytovatele mLab. Pro úložiště souborů bylo již dříve vybráno řešení AWS S3 (viz 2.2.4). Všichni vybraní poskytovatelé nabízejí základní verzi služeb zdarma.

3.5.1 Webová aplikace

Pro nasazení webové aplikace je nutné nastavit webový server tak, aby obsluhoval požadavky na zobrazení jednotlivých stránek. Při nasazování aplikací na Heroku je možné využít tzv. buildpacků, které připraví prostředí pro běh dané aplikace. Existuje velké množství buildpacků pro různé typy aplikací. Jeden takový existuje i pro aplikace vytvořené nástrojem create-react-app. Automaticky nakonfiguruje webový server pro obsluhu aplikace. Následně je možné aplikaci jednoduše nasadit využitím verzovacího nástroje git. Zavoláním příkazu `git push heroku master` dojde k sestavení nové verze aplikace a jejímu spuštění na Heroku. [58]

3.5.2 Serverová aplikace

Serverová aplikace pro svůj běh vyžaduje prostředí Node.js a knihovnu OpenCV. Pro takovou kombinaci již nestačí využití jednoduchých buildpacků. Poskytovatel Heroku ale umožňuje i nasazení aplikací využitím platformy Docker. Jedná se o open-source projekt pro izolaci aplikací do kontejnerů. Jednoduše je tak možné vytvořit kontejner obsahující požadované prostředí a v něm spustit danou aplikaci. Konfigurace kontejneru pro aplikaci je definována v souboru `Dockerfile`. Pro nasazení nové verze aplikace pak stačí zavolat následující příkazy, které zajistí sestavení kontejneru s novou verzí aplikace a jeho nahrání na Heroku. [59]

```
docker build -t pchmelar/ar-image-search-server .
docker push registry.heroku.com/ar-image-search-server/web
```

Příklad 3.24: Nasazení serverové aplikace prostřednictvím Dockeru

Kapitola 4

Testování

4.1 Způsob testování

Před samotným spuštěním je nutné systém řádně otestovat. Existuje mnoho způsobů testování, mezi nejpoužívanější se řadí testování absence chyb v implementaci, uživatelské testování a výkonnostní testování. Pro budoucí využití systému je klíčová jeho přesnost a škálovatelnost, proto bylo provedeno několik výkonnostních testů.

4.1.1 Testovací databáze obrázků

Typickým případem užití systému je rozpoznávání filmových plakátů, přebalů knih, billboardů a dalších obrázků. Lze tedy předpokládat, že databáze obrázků bude obsahovat převážně 2D grafiku vytvořenou počítačem. Pro účely testování bylo využito 500 log z volně dostupného souboru dat LLD – Large Logo Dataset. [60]

Na přesnost a škálovatelnost systému má velký vliv počet extrahovaných vlastností jednotlivých obrázků. Proto bylo vytvořeno 10 databází. První databáze extrahuje z každého obrázku 50 vlastností, druhá 100 vlastností, a tak dále až k poslední databázi, která extrahuje 500 vlastností.

Pro automatizaci nahrání 500 obrázků do každé z vytvořených databází byl vytvořen skript `upload.sh`. Prostřednictvím příkazu `cURL` postupně nahraje všechny obrázky z daného adresáře do určené databáze.

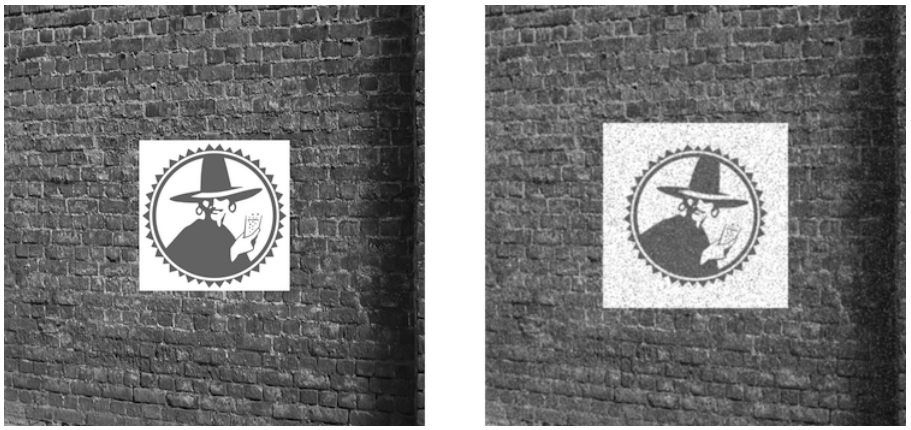
```
for f in "./LLD-logo_sample/*"
do
  curl -s --output /dev/null -H "Authorization: Bearer $token"
    -F image=@$f -F name=${f##*/} -F metadata=meta
    ".../database/$id/image"
done
```

Příklad 4.1: Skript pro nahrání obrázků do databáze

4.1.2 Testovací snímky

K otestování bylo dále nutné nasimulovat odesílání snímků z mobilního zařízení. Systém by měl vyhodnotit shodu u snímků, které obsahují některý z obrázků dané databáze. Skript `scene-producer.sh` vygeneruje pro každé logo z datasetu LLD obrázek složený z jednoduchého pozadí a daného loga. Kompozice je vytvořena volně dostupným nástrojem ImageMagick pomocí přepínače `composite`.

Vygenerované obrázky představují ideální případ, při realném použití často dochází k určité míře deformace. Může se jednat například o rotaci, horší světelné podmínky nebo špatnou ostrost. Pro každé logo byla proto vygenerována i druhá varianta složeného obrázku, určena pro otestování chování systému za zhoršených podmínek. Na dané logo je nejprve aplikována rotace o 15 ° využitím přepínače `rotate`. Přepínač `noise` aplikuje na obrázek šum typu `Impulse`. Výsledek je nakonec ještě rozostřen pomocí přepínače `blur`.



Příklad 4.2: Ukázka testovacích snímků

```
for f in "./LLD-logo_sample/*"
do
  # Method 1 - composition
  convert ./bg.jpg $f -gravity Center
    -composite ./LLD-scene/${f##*/}
  # Method 2 - composition + corruption
  convert ./bg.jpg \( $f -rotate 15 \) -gravity Center
    -composite ./LLD-scene2/${f##*/}
  convert ./LLD-scene2/${f##*/} +noise Impulse -blur 0x1
    ./LLD-scene2/${f##*/}
done
```

Příklad 4.3: Skript pro vytvoření testovacích snímků

4.1.3 Reprezentace testovacích snímků

Aplikační rozhraní serverové aplikace akceptuje snímky reprezentované maticí klíčových bodů a maticí deskriptorů (viz 2.3.5). Před samotným odesláním testovacího snímku je tedy nejprve nutné provést jeho analýzu. Získané matice je následně možné odeslat, a tím vyvolat stejnou akci jako při odeslání z mobilního zařízení.

Skript `matrix-producer.py` provede analýzu všech testovacích snímků a získané matice následně zapíše do určených souborů ve formátu definovaném aplikačním rozhraním serverové aplikace. Pro jednoduché využití knihovny OpenCV byl skript implementován v jazyce Python.

```
orb = cv2.ORB_create(nfeatures=500)
scenes = os.listdir('./LLD-scene/')
for scene in scenes:
  img = cv2.imread(scene,0)
  kp, des = orb.detectAndCompute(img,None)
  f = open('./LLD-matrix/' + scene.name + '.txt', 'w')
  f.write("{\n\"keyPoints\": " + kp.json + ",\n")
  f.write("\"descriptors\": " + des.json + "\n}")
```

Příklad 4.4: Skript pro reprezentaci testovacích snímků

4.1.4 Skript pro automatické testování

Samotné otestování přesnosti a škálovatelnosti systému je zajištěno skriptem `test.sh`, který postupně žádá server o vyhodnocení shody pro všech 500 testovacích snímků oproti zadané databázi. V průběhu testování dochází k měření počtu nalezených a nenalezených shod. Nalezené shody jsou dále děleny na správná nebo chybná rozpoznání. K chybnému rozpoznání dochází, pokud server vyhodnotí testovací snímek jako shodující se s jiným obrázkem, než který je ve skutečnosti na snímku.

```
for f in "./LLD-matrix/*"
do
  res=$(curl ... --data "@$f" ".../database/$id/detection")
  totalTime=$(($totalTime + $resTime))
  if [ "$resCode" == "201" ]
  then
    if [ "$resName" == "${f##*/}" ]
    then ((correct++))
    else ((incorrect++))
  else ((notfound++))
done
```

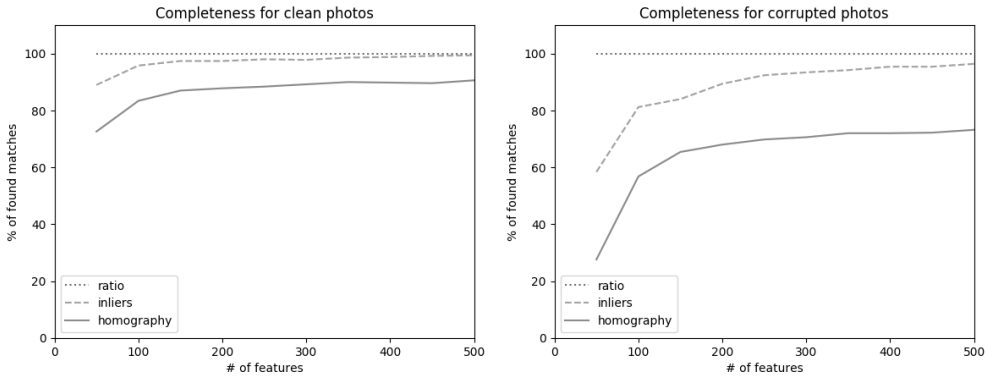
Příklad 4.5: Skript pro otestování systému

4.2 Přesnost systému

Postupně byly otestovány tři různé metody vyhodnocení shody přijatého snímku s databází obrázků. Metoda `ratio` pouze provede poměrový test a následně prohlásí za shodu obrázek s nejvyšším počtem shodujících se bodů. Výsledek je v metodě `inliers` dále zpřesněn výpočtem homografie. Za shodu je pak prohlášen obrázek s nejvyšším počtem inliers. Metoda `homography` navíc přidává odfiltrování nekvalitních homografií na základě ověření determinantů (viz 3.2.2.3).

4.2.1 Úplnost vyhledávání

Níže uvedená tabulka a graf vyjadřují procentuální úplnost vyhledávání, tedy pro kolik procent snímků (z celkového počtu 500) byla nalezena shoda. Metoda `ratio` má konstantní úplnost 100 % nezávisle na počtu extrahovaných vlastností. Pro každý z testovacích snímků byla nalezena shoda. Úplnost metod `inliers` a `homography` se ustálí přibližně při počtu 150 extrahovaných vlastností a dále se pouze mírně zvyšuje. Konkrétně metoda `homography` dosahuje úplnosti 87 % pro kvalitní testovací snímky a 65 % pro snímky deformované.



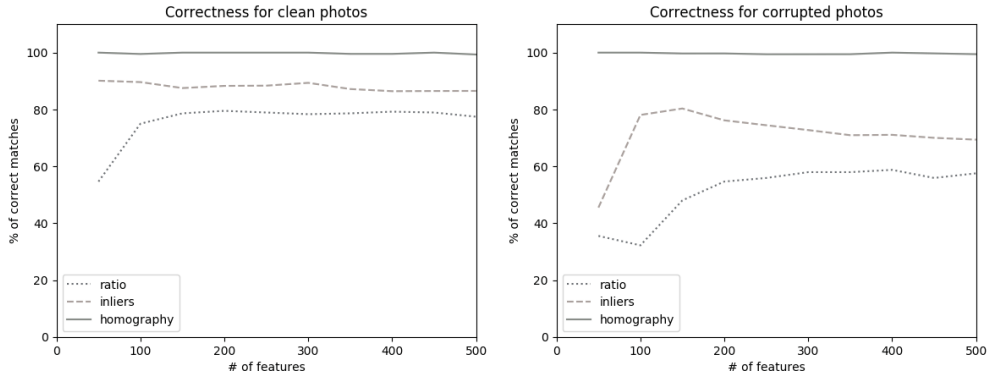
Příklad 4.6: Procentuální úplnost vyhledávání

	50	**100**	**150**	**200**	**250**	**300**	**350**	**400**	**450**	**500**
ratio	100	100	100	100	100	100	100	100	100	100
inliers	89	96	97	97	98	98	99	99	99	99
homogr.	73	83	87	88	88	89	90	90	90	91
ratio	100	100	100	100	100	100	100	100	100	100
inliers	58	81	84	89	92	93	94	95	95	96
homogr.	28	57	65	68	70	71	72	72	72	73

Tabulka 4.1: Procentuální úplnost vyhledávání

4.2.2 Správnost vyhledávání

Vyjádřením poměru chybných a správných rozpoznání je zjištěna správnost vyhledávání. Metoda `ratio` dosahuje úplnosti 100 %, ale její správnost je vcelku nízká – zhruba 80 % pro kvalitní testovací snímky a pouze 50 % pro snímky deformované. Opakem je metoda `homography`, která má nejnižší úplnost, ale dosahuje velmi vysoké správnosti 99 – 100 %.



Příklad 4.7: Procentuální správnost vyhledávání

	50	**100**	**150**	**200**	**250**	**300**	**350**	**400**	**450**	**500**
ratio	55	75	79	80	79	78	79	79	79	77
inliers	90	90	88	88	88	89	87	86	86	87
homogr.	100	100	100	100	100	100	100	100	100	99
ratio	36	32	48	55	56	58	58	59	56	58
inliers	46	78	80	76	74	72	71	71	70	69
homogr.	100	100	100	100	99	99	99	100	100	99

Tabulka 4.2: Procentuální správnost vyhledávání

4.2.3 Vyhodnocení

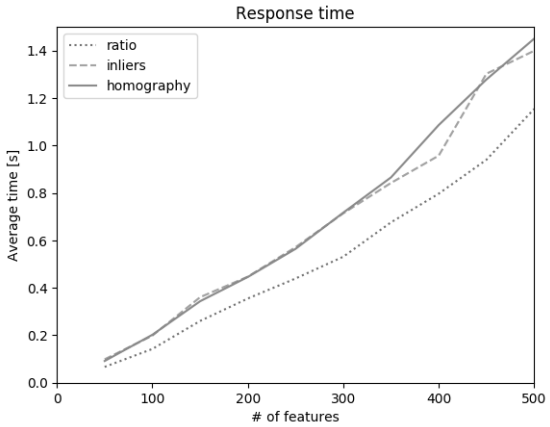
Výsledky testování potvrzují zjištěná fakta z analýzy dostupných metod pro porovnání deskriptorů (viz 1.5.2.2). Odfiltrováním nekvalitních homografií na základě ověření jejich determinantů je možné výrazně zvýšit správnost vyhledávání. Na množině testovacích snímků bylo pro 150 extrahovaných vlastností dosaženo správnosti 100 %. Úplnost dosahuje 65 – 87 % v závislosti na kvalitě snímku. Nenalezení shody je ve většině případů způsobené nekvalitním obrázkem v databázi. Testovací dataset LLD obsahuje náhodná loga z internetu a některá z nich nejsou vhodná pro účely vyhledávání – jsou v nízkém rozlišení nebo obsahují málo klíčových bodů.

4.3 Škálovatelnost systému

Vzhledem k požadavku na kontinuální komunikaci mezi mobilním zařízením a serverovou aplikací je nutné zajistit rychlé vyhledávání shod. Akceptovatelná doba vyhledávání jedné shody, kterou vykazuje i služba Vuforia [61], je zhruba 1,5 sekundy. Rychlost je ovlivněna počtem extrahovaných vlastností a počtem obrázků v databázi.

4.3.1 Počet extrahovaných vlastností

Dle očekávání dochází s rostoucím počtem extrahovaných deskriptorů k téměř lineárnímu růstu průměrné doby vyhledávání. Metoda `ratio` neprovádí výpočet homografie a je tedy zhruba o 25 % rychlejší. Ověření determinantů v metodě `homography` nemá na dobu vyhledávání výrazný vliv. Předchozí testy ukázaly, že přesnost systému se od 150 extrahovaných vlastností již příliš nezvyšuje. Jako optimální počet extrahovaných vlastností byla tedy stanovena hodnota 150.



Příklad 4.8: Průměrná doba vyhledávání dle počtu extrahovaných vlastností

	50	**100**	**150**	**200**	**250**	**300**	**350**	**400**	**450**	**500**
ratio	0,07	0,14	0,26	0,35	0,44	0,53	0,68	0,80	0,94	1,15
inliers	0,10	0,20	0,36	0,45	0,57	0,71	0,84	0,96	1,30	1,40
homogr.	0,09	0,20	0,34	0,46	0,56	0,72	0,87	1,09	1,28	1,45

Tabulka 4.3: Průměrná doba vyhledávání dle počtu extrahovaných vlastností

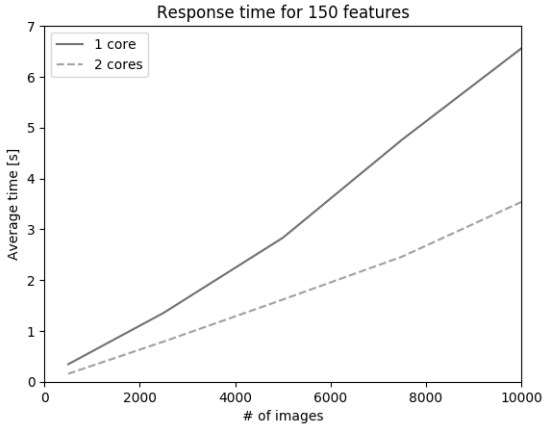
Průměrná doba vyhledávání byla měřena lokálně na osobním počítači MacBook Pro Mid 2017 s procesorem Intel Core i5-7360U a 16 GB RAM. Nasazená verze systému má horší parametry vzhledem k využití bezplatného hostingu od poskytovatele Heroku.

4.3.2 Počet obrázků v databázi

Veškeré dosavadní testy byly prováděny nad databází obsahující 500 obrázků. Systém ale musí být schopen obsloužit řádově větší databáze. Služba Wikitude má nastaven limit na 50 000 obrázků, služba Vuforia dokonce 100 000. Pro zachování rychlého vyhledávání shod i při tak vysokém počtu obrázků je nutné paralelizovat danou část serverové aplikace.

Dodatečně byla implementována experimentální paralelní verze middlewaru `detection.create`, který má na starosti vyhodnocení shody přijatého snímku (viz 3.2.2). Principem je rovnoměrné rozdělení databáze obrázků mezi více procesů využitím modulu `child_process` v prostředí Node.js. Každý proces pak vyhledává shody pouze v přidělené podmnožině. Vytvořené procesy je na vícejádrovém procesoru možné spustit paralelně, a tím dosáhnout požadovaného zrychlení.

Výsledky testování ukazují téměř dvojnásobné zrychlení při spuštění na dvoujádrovém procesoru. Pro 150 extrahovaných deskriptorů je potom limitem pro splnění 1,5 sekundové doby vyhledávání zhruba 5 000 obrázků. Běžně dostupné serverové procesory disponují až 24 jádry. Po optimalizaci a spuštění na výkonném serveru je tedy teoreticky možné dosáhnout stejných limitů jako poskytují služby Wikitude a Vuforia.



Příklad 4.9: Průměrná doba vyhledávání dle počtu obrázků v databázi

	500	**2 500**	**5 000**	**7 500**	**10 000**
1 core	0,34	1,35	2,83	4,76	6,56
2 cores	0,16	0,79	1,61	2,46	3,54

Tabulka 4.4: Průměrná doba vyhledávání dle počtu obrázků v databázi

Závěr

Zhodnocení práce

Hlavním cílem práce byl návrh a implementace prototypu systému, který by umožnil obohatit mobilní aplikace o vyhledávání předem definovaných obrázků v prostředí rozšířené reality.

Nejprve byly prověřeny možnosti realizace. Pro maximalizaci využitelnosti systému byla zvolena varianta kontinuální komunikace se serverem. Systém je tak složen ze tří hlavních komponent – serverové aplikace s aplikačním rozhraním, knihovny pro platformu iOS a webové aplikace pro správu obrázků. Následně byly stanoveny požadavky na jednotlivé části systému a byla provedena důkladná analýza dostupných metod pro určení podobnosti obrázků.

Na základě stanovených požadavků byla navržena architektura systému a způsob implementace jednotlivých částí. Pro serverovou aplikaci byl definován databázový model a struktura aplikačního rozhraní. Bylo rozhodnuto o analýze obrázků propojením prostředí Node.js s knihovnou OpenCV, která je rovněž využita v knihovně pro mobilní platformu iOS. Dále bylo navrženo uživatelské rozhraní webové aplikace prostřednictvím wireframů.

Dle návrhu byly následně implementovány prototypy jednotlivých částí systému. Pro vyhodnocení shodných obrázků na serveru bylo nutné rozšířit open-source knihovnu opencv4nodejs o několik chybějících funkcí. Změny byly akceptovány autorem knihovny a jsou tak k dispozici ostatním vývojářům. Hotový prototyp systému byl nasazen v testovacím módu využitím bezplatných služeb pro hostování aplikací.

Na závěr bylo provedeno několik výkonnostních testů. Vyhledávání obrázků dosahuje přesnosti 99 – 100 %. Úplnost vyhledávání je závislá na použitých obrázcích, pro testovací dataset se pohybuje mezi 65 – 87 %. Implementovaná verze systému dokáže obsloužit databáze obsahující nižší tisíce obrázků. Testování ukázalo, že při paralelizaci vyhledávání je možné dosáhnout řádově vyšších limitů, které jsou srovnatelné s komerčními službami.

Přínos práce

Vyzkoušel jsem si vývoj komplexního systému od analýzy požadavků, přes návrh architektury, až po samotnou implementaci a finální testování. Prohloubil jsem své znalosti v odvětví počítačového vidění a důkladně jsem se seznámil s algoritmy pro analýzu a porovnání obrázků. Nezanedbatelným přínosem je také podíl na vývoji open-source knihovny opencv4nodejs.

Budoucí rozšíření funkcionality systému

Implementovaný prototyp systému je plně funkční a splňuje všechny body, které byly vytyčené v zadání práce. I přesto je zde prostor pro další rozšíření stávající funkcionality.

Akce v knihovně pro platformu iOS

Knihovna pro platformu iOS v aktuální verzi pouze informuje mobilní aplikaci o nalezené shodě. Implementace příslušné akce je již záležitostí samotné aplikace. Přidání několika základních akcí (otevření webové stránky, přehrání videa, apod.) by ještě více usnadnilo využití v mobilních aplikacích.

Měřítko kvality obrázků

Testování ukázalo, že některé obrázky nejsou příliš vhodné pro účely vyhledávání. Uživatel neznalý problematiky počítačového vidění může mít problém s vybráním vhodných obrázků. Webová aplikace by u jednotlivých obrázků mohla zobrazovat určité měřítko kvality pro usnadnění výběru.

Optimalizace vyhledávání obrázků

Vyhledávání shodných obrázků je výpočetně náročné a průměrná doba odezvy roste v závislosti na počtu obrázků. Navrhnutá paralelizace kritické části přináší výrazné urychlení. Ještě lepších výsledků by bylo možné docílit využitím optimalizované verze knihovny OpenCV určené pro běh na grafických procesorech. Dle dostupných testů je možné dosáhnout až padesátinásobného zrychlení. [62]

Použité zdroje

[1] Smartphone OS Market Share [online]. 2018, [cit. 2018-05-04]. Dostupné z: https://www.idc.com/promo/smartphone-market-share/os

[2] Richardson, L.; Ruby, S.: *Restful Web Services*. O'Reilly, první vydání, 2007, ISBN 9780596529260, 13–18 s.

[3] Template Matching [online]. 2014, [cit. 2018-08-04]. Dostupné z: https://docs.opencv.org/3.0-beta/doc/py_tutorials/py_imgproc/py_template_matching/py_template_matching.html

[4] Understanding Features [online]. 2014, [cit. 2018-09-04]. Dostupné z: https://docs.opencv.org/3.0-beta/doc/py_tutorials/py_feature2d/py_features_meaning/py_features_meaning.html

[5] Harris, C.; Stephens, M.: A Combined Corner and Edge Detector. In *Proceedings of the 4th Alvey Vision Conference*, 1988, s. 147–151.

[6] Bílek, P.: *Významné body v obraze: detekce, korespondence a lokalizace ve 3D*. Bakalářská práce, 2007.

[7] Lowe, D. G.: Distinctive Image Features from Scale-Invariant Keypoints. *International Journal of Computer Vision*, ročník 60, č. 2, 2004: s. 91–110.

[8] Bay, H.; Tuytelaars, T.; Van Gool, L.: SURF: Speeded Up Robust Features. In *Computer Vision – ECCV 2006*, Springer Berlin Heidelberg, 2006, ISBN 978-3-540-33833-8, s. 404–417.

[9] Rublee, E.; Rabaud, V.; Konolige, K.; aj.: ORB: An Efficient Alternative to SIFT or SURF. In *International Conference on Computer Vision*, Barcelona, 2011.

[10] Alcantarilla, P. F.; Bartoli, A.; Davison, A. J.: KAZE Features. In *Computer Vision – ECCV 2012*, Springer Berlin Heidelberg, 2012, ISBN 978-3-642-33783-3, s. 214–227.

[11] Alcantarilla, P. F.; Nuevo, J.; Bartoli, A.: Fast Explicit Diffusion for Accelerated Features in Nonlinear Scale Spaces. In *BMVC*, 2013.

[12] Karami, E.; Prasad, S.; Shehata, M. S.: Image Matching Using SIFT, SURF, BRIEF and ORB: Performance Comparison for Distorted Images. *CoRR*, 2017.

[13] Roosab, D. R.; Shiguemori, E. H.; Lorena, A. C.: Comparing ORB and AKAZE for visual odometry of unmanned aerial vehicles. 2015.

[14] Feature Matching [online]. 2014, [cit. 2018-10-04]. Dostupné z: `https://docs.opencv.org/3.0-beta/doc/py_tutorials/py_feature2d/py_matcher/py_matcher.html`

[15] Feature Matching and Homography to find Objects [online]. 2014, [cit. 2018-11-04]. Dostupné z: `https://docs.opencv.org/3.0-beta/doc/py_tutorials/py_feature2d/py_feature_homography/py_feature_homography.html`

[16] Fei-Fei, L.; Fergus, R.; Torralba, A.: Recognizing and Learning Object Categories. 2005. Dostupné z: `http://people.csail.mit.edu/torralba/shortCourseRLOC/`

[17] Kandráč, J.: Bag of visual words in OpenCV. 2015, [cit. 2018-16-04]. Dostupné z: `http://vgg.fiit.stuba.sk/2015-02/bag-of-visual-words-in-opencv/`

[18] Thahir, A.: Which is Better? Saving Files in Database or in File System [online]. 2017, [cit. 2018-13-04]. Dostupné z: `https://habiletechnologies.com/blog/better-saving-files-database-file-system/`

[19] TIOBE Index. 2018, [cit. 2018-13-04]. Dostupné z: `https://www.tiobe.com/tiobe-index/`

[20] Java EE at a Glance. 2018, [cit. 2018-13-04]. Dostupné z: `http://www.oracle.com/technetwork/java/javaee/overview/index.html`

[21] History of PHP. 2018, [cit. 2018-13-04]. Dostupné z: `http://php.net/manual/en/history.php.php`

[22] Usage of content management systems for websites [online]. 2018, [cit. 2018-13-04]. Dostupné z: `http://w3techs.com/technologies/overview/content_management/all`

[23] Rauschmayer, A.: *Speaking JavaScript*. O'Reilly Media, Inc., první vydání, 2014, ISBN 1449365035, 9781449365035, 43–44 s.

[24] Teixeira, P.: *Professional Node.Js: Building Javascript Based Scalable Software*. Wrox Press Ltd., první vydání, 2012, ISBN 1118185463, 9781118185469, 15–19 s.

[25] Stack Overflow Annual Developer Survey [online]. 2017, [cit. 2018-14-04]. Dostupné z: https://insights.stackoverflow.com/survey/2017#technologies-and-occupations

[26] Haerder, T.; Reuter, A.: Principles of Transaction-oriented Database Recovery. *ACM Computing Surveys*, ročník 15, č. 4, 1983: s. 287–317.

[27] Browne, J.: Brewer's CAP Theorem [online]. 2009, [cit. 2018-15-04]. Dostupné z: http://www.julianbrowne.com/article/brewers-cap-theorem

[28] DB-Engines Ranking [online]. 2018, [cit. 2018-15-04]. Dostupné z: https://db-engines.com/en/ranking

[29] Sahni, V.: Best Practices for Designing a Pragmatic RESTful API [online]. 2016, [cit. 2018-16-04]. Dostupné z: https://www.vinaysahni.com/best-practices-for-a-pragmatic-restful-api

[30] Bradski, D. G. R.; Kaehler, A.: *Learning OpenCV*. O'Reilly Media, Inc., první vydání, 2008, ISBN 9780596516130, 6–8 s.

[31] Tarkus, K.: How to call C/C++ code from Node.js [online]. 2017, [cit. 2018-17-04]. Dostupné z: https://medium.com/@tarkus/how-to-call-c-c-code-from-node-js-86a773033892

[32] Ohayon, B.: OpenCV and Swift [online]. 2016, [cit. 2018-17-04]. Dostupné z: https://medium.com/@borisohayon/ios-opencv-and-swift-1ee3e3a5735b

[33] Coyle, A.: Design Better Forms [online]. 2016, [cit. 2018-18-04]. Dostupné z: https://uxdesign.cc/design-better-forms-96fadca0f49c

[34] Sherwin, K.: Placeholders in Form Fields Are Harmful [online]. 2014, [cit. 2018-18-04]. Dostupné z: https://www.nngroup.com/articles/form-design-placeholders/

[35] Jahoda, B.: Single page application [online]. 2015, [cit. 2018-19-04]. Dostupné z: http://jecas.cz/spa

[36] Lawson, N.: Progressive enhancement isn't dead, but it smells funny [online]. 2016, [cit. 2018-19-04]. Dostupné z: https://nolanlawson.com/2016/10/13/progressive-enhancement-isnt-dead-but-it-smells-funny/

[37] Ruebbelke, L.: Awesome AngularJS Features [online]. 2012, [cit. 2018-19-04]. Dostupné z: https://code.tutsplus.com/tutorials/5-awesome-angularjs-features--net-25651

[38] Dawson, C.: JavaScript's History and How it Led To ReactJS [online]. 2014, [cit. 2018-19-04]. Dostupné z: https://thenewstack.io/javascripts-history-and-how-it-led-to-reactjs/

[39] Ember's Core Concepts [online]. 2018, [cit. 2018-19-04]. Dostupné z: https://guides.emberjs.com/v3.1.0/getting-started/core-concepts/

[40] Benitte, R.: The State Of JavaScript [online]. 2017, [cit. 2018-19-04]. Dostupné z: https://stateofjs.com/2017/front-end/results

[41] Irvine, D.: Understanding 403 Forbidden [online]. 2011, [cit. 2018-19-04]. Dostupné z: http://www.dirv.me/blog/2011/07/18/understanding-403-forbidden/index.html

[42] Schneier, B.: Cryptanalysis of SHA-1 [online]. 2015, [cit. 2018-20-04]. Dostupné z: https://www.schneier.com/blog/archives/2005/02/cryptanalysis_o.html

[43] Provos, N.; Mazières, D.: A Future-adaptive Password Scheme. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, USENIX Association, 1999, s. 81–92.

[44] Kukic, A.: Cookies vs Tokens: The Definitive Guide [online]. 2016, [cit. 2018-20-04]. Dostupné z: https://auth0.com/blog/cookies-vs-tokens-definitive-guide/

[45] DeBill, E.: Module Counts [online]. 2018, [cit. 2018-24-04]. Dostupné z: http://www.modulecounts.com/

[46] McGary, S.: How to structure a Node.js Express project [online]. 2016, [cit. 2018-24-04]. Dostupné z: https://seanmcgary.com/posts/how-to-structure-a-nodejs-express-project/

[47] Using Express middleware [online]. 2017, [cit. 2018-25-04]. Dostupné z: https://expressjs.com/en/guide/using-middleware.html

[48] Baumann, A.: Postmanerator [online]. 2017, [cit. 2018-25-04]. Dostupné z: https://github.com/aubm/postmanerator

[49] Chmelař, P.: opencv4nodejs – Pull Request #154 [online]. 2018, [cit. 2018-25-04]. Dostupné z: https://github.com/justadudewhohacks/opencv4nodejs/pull/154

[50] Check if homography is good [online]. 2012, [cit. 2018-30-04]. Dostupné z: http://answers.opencv.org/question/2588/check-if-homography-is-good/

[51] Chmelař, P.: opencv4nodejs – Pull Request #239 [online]. 2018, [cit. 2018-25-04]. Dostupné z: https://github.com/justadudewhohacks/opencv4nodejs/pull/239

[52] Nava, E.: Publish a Universal Binary iOS Framework in Swift using CocoaPods [online]. 2016, [cit. 2018-27-04]. Dostupné z: https://eladnava.com/publish-a-universal-binary-ios-framework-in-swift-using-cocoapods/

[53] Zaytsev, J.: ECMAScript compatibility table [online]. 2018, [cit. 2018-27-04]. Dostupné z: http://kangax.github.io/compat-table/es5/

[54] JSX Specification [online]. 2014, [cit. 2018-27-04]. Dostupné z: https://facebook.github.io/jsx/

[55] Janča, M.: Webpack – moderní Web Development [online]. 2017, [cit. 2018-27-04]. Dostupné z: https://www.ackee.cz/blog/moderni-web-development-webpack/

[56] Vepsäläinen, J.: How to Structure a React Project [online]. 2015, [cit. 2018-27-04]. Dostupné z: http://reactjsnews.com/structuring-react-projects

[57] Ceddia, D.: A Visual Guide to State in React [online]. 2016, [cit. 2018-27-04]. Dostupné z: https://daveceddia.com/visual-guide-to-state-in-react/

[58] Hall, M.: Deploying React with Zero Configuration [online]. 2016, [cit. 2018-28-04]. Dostupné z: https://blog.heroku.com/deploying-react-with-zero-configuration

[59] Dockerizing a Node.js web app [online]. 2017, [cit. 2018-28-04]. Dostupné z: https://nodejs.org/en/docs/guides/nodejs-docker-webapp/

[60] Sage, A.; Agustsson, E.; Timofte, R.; aj.: LLD – Large Logo Dataset. 2017, [cit. 2018-02-05]. Dostupné z: https://data.vision.ee.ethz.ch/sagea/lld/

[61] Vuforia – Number of targets and response time. 2017, [cit. 2018-03-05]. Dostupné z: https://developer.vuforia.com/forum/cloud-recognition/does-number-targets-affect-response-time?sort=2

[62] Bowley, J.: OpenCV 3.4 GPU CUDA Performance Comparison. 2018, [cit. 2018-04-05]. Dostupné z: https://jamesbowley.co.uk/opencv-3-4-gpu-cuda-performance-comparison-nvidia-vs-intel/

Příloha A

Seznam použitých zkratek

AKAZE Accelerated-KAZE

API Application Programming Interface

AR Augmented Reality

AWS Amazon Web Services

BRIEF Binary Robust Independent Elementary Features

BSD Berkeley Software Distribution

BSON Binary JSON

CMS Content Management System

DOM Document Object Model

ECMA European Computer Manufacturers Association

FAST Features from Accelerated Segment Test

HTML HyperText Markup Language

HTTP HyperText Transfer Protocol

HTTPS HyperText Transfer Protocol Secure

I/O Input/Output

JSON JavaScript Object Notation

JWT JSON Web Token

LLD Large Logo Dataset

MVC Model-View-Controller

MVVM Model-View-ViewModel

OpenGL Open Graphics Library

ORB Oriented FAST and Rotated BRIEF

PHP PHP: Hypertext Preprocessor

RANSAC Random sample consensus

REST Representational State Transfer

SIFT Scale-Invariant Feature Transform

SOAP Simple Object Access Protocol

SPA Single Page Application

SQL Structured Query Language

SURF Speeded Up Robust Features

TLS Transport Layer Security

UI User Interface

URL Uniform Resource Locator

VR Virtual Reality

XML Extensible Markup Language

Příloha **B**

Obsah přiloženého DVD

- `src`
  - `impl`
    - `apidoc` ..... Dokumentace aplikačního rozhraní
    - `ios` ..... Zdrojové kódy iOS knihovny
    - `python` ..... Pomocné skripty pro ověření funkcí OpenCV
    - `server` ..... Zdrojové kódy serverové aplikace
    - `tests` ..... Skripty pro otestování systému
    - `webapp` ..... Zdrojové kódy webové aplikace
  - `thesis` ..... Zdrojová forma práce ve formátu LaTeX
- `text`
  - `thesis.pdf` ..... Text práce ve formátu PDF
- `readme.txt` ..... Stručný popis obsahu DVD